IBM Informix 4GL

Reference Manual

Version 7.31
January 2002
Part No. 000-8776

Note:
Before using this information and the product it supports, read the information in the appendix entitled "Notices."

Table of Contents

Introduction

Chapter 3 Data Types and Expressions

Introduction

In This Introduction

This Introduction provides an overview of the information in this manual and describes the conventions it uses.

About This Manual

This manual is designed to be a day-to-day, keyboard-side companion for 4GL programmers. It describes the features and syntax of the 4GL language, including 4GL statements, forms, reports, and the built-in functions and operators.

Organization of This Manual

This manual is divided into two volumes and includes the following chapters and appendixes:

- Chapter 1, "Compiling INFORMIX-4GL Source Files," describes the *C Compiler* and *Rapid Development System* implementations of INFORMIX-4GL. It also explains how to create executable versions of 4GL source files, both from the Programmer's Environment and from the command line.
- Chapter 2, "The INFORMIX-4GL Language," provides an overview of 4GL language features and graphical features of the applications that you can create with INFORMIX-4GL.
- Chapter 3, "Data Types and Expressions," describes 4GL data types, expressions, and other syntax topics that affect several statements.

- Chapter 4, "INFORMIX-4GL Statements," describes the statements of 4GL in alphabetical order.
- Chapter 5, "Built-In Functions and Operators," includes an overview of the predefined functions and operators of 4GL and describes their individual syntax, with examples of usage.
- Chapter 6, "Screen Forms," provides an overview of 4GL screen forms and form drivers and describes the syntax of 4GL form specification files. It also describes how to create forms with the **form4gl** form compiler and describes how the **upscol** utility can set default attributes.
- Chapter 7, "INFORMIX-4GL Reports," offers an overview of 4GL reports and report drivers, and describes the syntax of 4GL report definitions. It also describes the syntax of statements and operators that can appear only in 4GL reports.
- Appendix A, "The ASCII Character Set," lists the ASCII characters and their numeric codes.
- Appendix B, "INFORMIX-4GL Utility Programs," describes the **mkmessage** and **upscol** utility programs.
- Appendix C, "Using C with INFORMIX-4GL," describes how to call C functions from 4GL programs, and vice versa, and describes a function library for conversion between the DECIMAL data type of 4GL and the C data types.
- Appendix D, "Environment Variables," describes the environment variables that are used by 4GL.
- Appendix E, "Developing Applications with Global Language Support," describes the internationalization and localization features that are provided with 4GL, and shows how to develop 4GL applications that are world-ready and easy to localize.
- Appendix F, "Modifying termcap and terminfo," describes the modifications you can make to your **termcap** and **terminfo** files to extend function key definitions, to specify characters for window borders, and to enable 4GL programs to interact with terminals that support color displays.
- Appendix G, "Reserved Words," lists words that you should not declare as identifiers in 4GL programs. It also lists the ANSI reserved words of SQL.

- Appendix H, "The Demonstration Application," lists the code of the **demo4.4ge** demonstration application.
- Appendix I, "SQL Statements That Can Be Embedded in 4GL Code," lists SQL syntax that is directly supported in 4GL.
- A Notices appendix describes IBM products, features, and services.
- The Glossary defines terms used in the 4GL documentation set.

Types of Readers

This manual is written for all 4GL developers. You do not need database management experience nor familiarity with relational database concepts to use this manual. A knowledge of SQL (*structured query language*), however, and experience using a high-level programming language would be useful.

Software Dependencies

This manual is written with the assumption that you are using an Informix database server, Version 7.x or later.

Informix offers two implementations of the 4GL application development language:

- The INFORMIX-4GL C Compiler uses a preprocessor to generate Informix ESQL/C source code. This code is preprocessed in turn to produce *C source code*, which is then compiled and linked as object code in an executable command file.
- The INFORMIX-4GL Rapid Development System (RDS) uses a compiler to produce *pseudo-machine code* (called p-code) in a single step. You then invoke a *runner* to execute the p-code version of your application.

Both versions of 4GL use the same 4GL statements. Chapter 1, "Compiling INFORMIX-4GL Source Files," describes the differences between the two versions of 4GL and explains how to use both versions.

You can easily use applications developed with an earlier version of 4GL, such as Version 4.x or 6.x or 7.2, with this version of 4GL. For RDS programs that use p-code, however, you must first recompile your 4GL source code.

Assumptions About Your Locale

Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a GLS (Global Language Support) locale.

Examples in this manual are written with the assumption that you are using the default locale, **en_us.8859-1**. This supports U.S. English format conventions for dates, times, and currency, and the ISO 8859-1 code set. Some versions of this locale support various non-ASCII 8-bit characters such as é, è, and ñ. Like all locales that Informix provides with its GLS libraries, however, the default locale supports the ASCII characters (as listed in Appendix A).

If you plan to use non-ASCII characters in your data or your SQL identifiers, or if you want nondefault collation of character data, you need to specify the appropriate nondefault locale. For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS, see the *Informix Guide to GLS Functionality*. See also Appendix E, "Developing Applications with Global Language Support."

Demonstration Database and Examples

4GL includes several 4GL demonstration applications, along with a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. You can create the **stores7** database in the current directory by entering the following command:

```
dbaccessdemo7
```

Many (but not all) of the examples in the 4GL documentation set are based on the **stores7** database. This database is described in detail in Appendix A of *INFORMIX-4GL by Example*. For more information on creating the demonstration database and using the examples, see the introduction to *INFORMIX-4GL by Example*.

Accessing Databases from Within 4GL

The 4GL language approximates a superset of the Informix implementation of the industry-standard SQL language. Because the 4GL compiler does not recognize some SQL statements, however, three methods are supported for including SQL statements within the 4GL program:

- For most SQL syntax that was supported by Informix 4.1 database servers, you can directly embed SQL statements in 4GL source code.
- For SQL statements that can be prepared, you can use the PREPARE feature of SQL to include SQL statements as text in prepared objects.
- For all SQL statements that can be prepared, you can also use the SQL ... END SQL delimiters to enclose the SQL statement. Unlike PREPARE, these can include host variables for input and output.

You must use one of the last two methods for SQL statements that include syntax that was introduced later than Informix 4.1 database servers. Such embedded, prepared, or delimited SQL statements are passed on to the database server for execution, as in the following 4GL program fragment:

```
DEFINE bugfile, setdeb CHAR(255)
DATABASE stores7               --Directly embedded SQL
LET setdeb = "set debug file to /u/tanya/bugfile"
PREPARE bugfile FROM setdeb    --Prepared SQL (post-4.1)
EXECUTE IMMEDIATE bugfile      --Directly embedded SQL
SQL SET PDQPRIORITY HIGH       --Delimited SQL (post-4.1)
END SQL
```

Appendix I, "SQL Statements That Can Be Embedded in 4GL Code," lists the supported SQL syntax.

If you are uncertain which method is needed, use SQL...END SQL delimiters, which generally offer wider functionality and greater ease of coding than PREPARE for SQL statements that cannot be directly embedded.

For additional information on SQL statements, see *Informix Guide to SQL: Syntax*.

Enhancements to Version 7.31

This product includes the following enhancements:

- The NCHAR and NVARCHAR data types are now recognized and supported. For information about the NCHAR and NVARCHAR data types, see Chapter 3, "Data Types and Expressions," of this book.
- For database servers that support long SQL identifiers, your 4GL applications can reference SQL identifiers that are up to 128 bytes in length. For more information about IFX_LONGID, see Chapter 2, "The INFORMIX-4GL Language," of this book.
- IBM INFORMIX-4GL is compatible with IBM Informix Client SDK 2.70.xC3 and above. Client SDK components allow developers to write applications in the language they are familiar with, whether it be Java™, C++, C, or ESQL. For more information about Client SDK, see the documentation set at www.informix.com / answers.

Documentation Conventions

This section describes certain conventions that this manual uses.

These conventions make it easier to gather information from this and other volumes in the documentation set. The following conventions are discussed:

- Typographical conventions
- Icon conventions
- Example-code conventions
- Syntax conventions

Typographical Conventions

This manual uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics ***italics*** `italics`	Within text, new terms and emphasized words appear in italics. Within syntax diagrams and code examples, identifiers or values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` *`monospace`*	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys to press appear in uppercase letters in a sans serif font.

Tip: *When you are instructed to "enter" characters or to "execute" a command, immediately press* RETURN *after the entry. When you are instructed to "type" the text or to "press" other keys, no* RETURN *is required.*

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Feature, Product, and Platform Icons

Feature, product, and platform icons identify paragraphs that contain feature-specific, product-specific, or platform-specific information.

Icon	Description
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature.
IDS	Identifies information or syntax that is specific to Informix Dynamic Server.
SE	Identifies information or syntax that is specific to INFORMIX-SE.
SQL	Identifies SQL statements that you must put in an SQL block or prepare statement.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the feature-specific, product-specific, or platform-specific information.

Compliance Icons

Compliance icons indicate paragraphs that provide guidelines for complying with a standard.

Icon	Description
ANSI	Identifies information that is specific to an ANSI-compliant database

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the compliance information.

Example-Code Conventions

Examples of 4GL source code occur in several sections of this manual. For readability, 4GL or SQL keywords generally appear in uppercase characters in code examples, and identifiers usually appear in lowercase or mixed case.

For instance, you might see code as in the following example:

```
MENU "CUSTOMER"
   COMMAND "Query" "Search for a customer"
      CALL query_data( )
      NEXT OPTION "Modify"
   ...
   COMMAND "Modify" "Modify a customer"
   ...
END MENU
```

Ellipsis (…) symbols in a code example indicate that more code would be added in a complete application, but for clarity and simplicity, the example omits code that is extraneous to the current topic. (In most contexts, a compile-time or runtime error occurs if literal ellipsis symbols appear in code, or if you omit code that is necessary to your program logic.) Ellipsis symbols do not begin or end code examples, however, even if additional code would be required for a complete program, or for a complete statement.

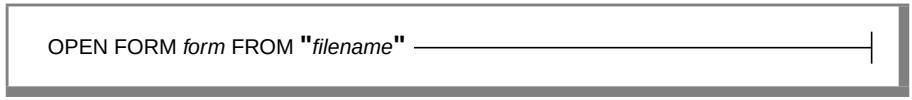
Most 4GL code examples are fragments of programs, rather than complete programs that can be compiled and executed. Because most examples are provided to illustrate some specific topic, the examples sometimes do not strictly conform to good programming practice. For example, they might not check to verify that an operation successfully executed without error, or they might not include the comments that normally should be included to improve readability and simplify maintenance of your code.

Syntax Conventions

SQL statement syntax is described in the *Informix Guide to SQL: Syntax*. The syntax for 4GL statements is described in Chapter 4 of this manual. Most chapters of this book describe the syntax of some aspect of the 4GL language, such as expressions, form specifications, and reports.

This section describes conventions for syntax diagrams. Each diagram displays the sequences of required and optional keywords, terms, and symbols that are valid in a given statement or segment. Figure 1 shows the syntax diagram of the OPEN FORM statement.

Figure 1
Example of a Simple Syntax Diagram



Each syntax diagram begins at the upper-left corner and ends at the upper-right corner with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement. (For a few diagrams, however, notes in the text identify path segments that are mutually exclusive.)

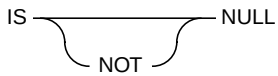
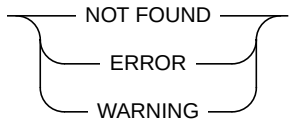
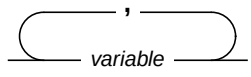
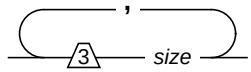
Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. The path always approaches elements from the left and continues to the right, except in the case of separators in loops. For separators in loops, the path approaches counterclockwise. Unless otherwise noted, at least one blank character separates syntax elements.

Elements That Can Appear on the Path

You might encounter one or more of the following elements on a path.

Element	Description
KEYWORD	A word in UPPERCASE letters is a keyword. You must spell the word exactly as shown; you can, however, use either uppercase or lowercase letters.
(. , ; @ + * - /)	Punctuation and other nonalphanumeric characters are literal symbols that you must enter exactly as shown.
" " ' '	Double quotation marks must be entered as shown. If you prefer, a pair of double quotation marks can replace a pair of single quotation marks, but you cannot mix double and single quotation marks.
variable	A word in italics represents a value that you must supply. A table that follows the diagram explains the value.
ATTRIBUTE Clause p. 3-288 ATTRIBUTE Clause	A reference in a box represents a subdiagram. Imagine that the subdiagram is spliced into the main diagram at this point. When a page number is not specified, the subdiagram appears on the same page. The aspect ratios of boxes are not significant.
SELECT Statement *see* SQL:S	A reference to SQL:S in a syntax diagram represents an SQL statement or segment that is described in the *Informix Guide to SQL: Syntax*. Imagine that the segment were spliced into the diagram at this point.
SE	An icon is a warning that this path is valid only for some products, or conditions. The following icons appear in some syntax diagrams: SE — This path is valid only for INFORMIX-SE database servers. IDS — This path is valid only for Informix Dynamic Server.

(1 of 2)

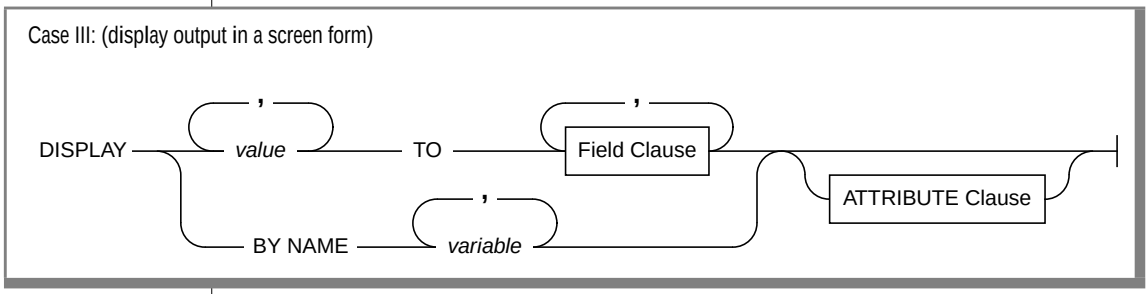
Element	Description
	 This path produces a warning (of a syntax extension to the ANSI/ISO standard for SQL) if DBANSIWARN is set, or if the program is compiled with the -ansi flag.
	A shaded option is the default, if you provide no other specification.
	A syntax segment within a pair of arrows is a subdiagram.
	The vertical line terminates the syntax diagram.
	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
	A set of multiple branches indicates that a choice among more than two different paths is available.
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items. If no symbol appears, a blank space is the separator, or (as here) the Linefeed that separates successive statements in a source module.
	A gate (3) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. You can specify *size* no more than three times within this statement segment.

(2 of 2)

How to Read a Syntax Diagram

Figure 2 shows a syntax diagram that uses some of the path elements that the previous table lists.

Figure 2
Example of a Syntax Diagram



The Case III label above the diagram implies that this statement can have at least two other syntax patterns. To use this diagram to construct a statement, start at the top left with the keyword DISPLAY. Then follow the diagram to the right, proceeding through the options that you want.

Figure 2 illustrates the following steps:

1. Type the keyword `DISPLAY`.
2. You can display the values of a list of variables to an explicit list of fields within the current screen form:
   - Type the name of a *value*. If you want to display several *values*, separate successive *values* by a comma.
   - Type the keyword `TO` after the name of the last *value*.
   - Type the name of a *field* in the current form in which to display the first *value*. To find the syntax for specifying field names, go to the Field Clause segment on the specified page.
3. If you are using a form whose fields have the same names as the *values* that you want to display, you can follow the lower path:
   - Type the keywords `BY NAME` after `DISPLAY`.
   - Type the name of a *variable*. If you want to display the values of several variables, separate successive variables by comma.

4. You can optionally set a screen attribute for the displayed values:
   - Use the syntax of the ATTRIBUTE Clause segment on the specified page to specify the screen attribute that you desire.
5. Follow the diagram to the terminator.

   Your DISPLAY TO or DISPLAY BY NAME statement is now complete.

   A restriction on step 2 (that there must be as many fields as variables) appears in notes that follow the diagram, rather than in the diagram itself. If 4GL issues an error when you compile a statement that seems to follow the syntax diagram, it might be a good idea to also read the usage notes for that statement.

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- Documentation included with 4GL
- On-line manuals
- On-line help
- On-line error messages
- Related reading

Documentation Included with 4GL

The INFORMIX-4GL documentation set includes the following additional manuals:

- *INFORMIX-4GL Installation Guide* is a pamphlet that describes how to install the various 4GL products.
- *INFORMIX-4GL Concepts and Use* introduces 4GL and provides the context needed to understand the other manuals in the documentation set. It covers 4GL goals (what kinds of programming the language is meant to facilitate), concepts and nomenclature (parts of a program, ideas of database access, screen form, and report generation), and methods (how groups of language features are used together to achieve particular effects).
- *INFORMIX-4GL by Example* is a collection of 30 annotated 4GL programs. Each is introduced with an overview; then the program source code is shown with line-by-line notes. The program source files are distributed as text files with the product; scripts that create the demonstration database and copy the applications are also included.
- Documentation notes, which contain additions and corrections to the manuals, and release notes are located in the directory where the product is installed. Please examine these files because they contain vital information about application and performance issues.

If you have also purchased the INFORMIX-4GL Interactive Debugger product (which requires the INFORMIX-4GL Rapid Development System), your 4GL documentation also includes the following manual:

- *Guide to the INFORMIX-4GL Interactive Debugger* is both an introduction to the Debugger and a comprehensive reference of Debugger commands and features. The Debugger allows you view the source code and to interact with your 4GL programs while they are running. It helps you to analyze the logic of your 4GL program and to determine the source of runtime errors within your programs.

On-Line Manuals

The Informix Answers OnLine CD allows you to print chapters or entire books and perform full-text searches for information in specific books or throughout the documentation set. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see the installation insert that accompanies Answers OnLine. You can also access Answers OnLine on the Web at the following URL:

```
http://www.informix.com/answers
```

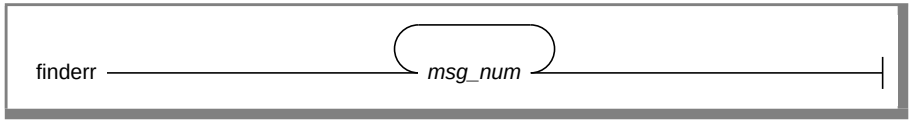
On-Line Help

The Programmer's Environment of 4GL provides on-line help; you can invoke help by pressing CONTROL-W.

On-Line Error Messages

Use the **finderr** script to display a particular error message or messages on your screen. The script is located in the **$INFORMIXDIR/bin** directory.

The **finderr** script has the following syntax.



msg_num	Indicates the number of the error message to display. Error message numbers range from -1 to -32000. Specifying the - sign is optional.

For example, to display the -359 error message, you can enter either of the following:

```
finderr -359
```

or, equivalently:

```
finderr 359
```

The following example demonstrates how to specify a list of error messages. The example also pipes the output to the UNIX **more** command to control the display. You can also direct the output to another file so that you can save or print the error messages:

```
finderr 233 107 113 134 143 144 154 | more
```

A few messages have positive numbers. These messages are used solely within the application tools. In the unlikely event that you want to display them, you must precede the message number with the + sign.

The messages numbered -1 to -100 can be platform-dependent. If the message text for a message in this range does not apply to your platform, check the operating system documentation for the precise meaning of the message number.

Related Reading

The following Informix database server publications provide additional information about the topics that this manual discusses:

- Informix database servers and the SQL language are described in separate manuals, including *Informix Guide to SQL: Tutorial*, *Informix Guide to SQL: Syntax*, and *Informix Guide to SQL: Reference*.
- Information about setting up Informix database servers is provided in the *Administrator's Guide* for your Informix database server.

Informix Press, in partnership with Prentice Hall, publishes books about Informix products. Authors include experts from Informix user groups, employees, consultants, and customers. Recent titles about INFORMIX-4GL include:

- *Advanced INFORMIX-4GL Programming*, by Art Taylor, 1995.
- *Programming Informix SQL/4GL: A Step-by-Step Approach*, by Cathy Kipp, 1998.
- *Informix Basics*, by Glenn Miller, 1998.

You can access Informix Press on the Web at the following URL:

```
http://www.informix.com/ipress
```

Informix Developer Network

Informix maintains the Informix Developer Network (IDN) as a resource by which developers of Informix-based applications can exchange information with their peers and with Informix experts. You can access the Informix Developer Network on the Web at the following URL:

```
http://www.informix.com/idn
```

Informix Welcomes Your Comments

Let us know what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send electronic mail, our address is:

`doc@informix.com`

We appreciate your suggestions.

Important: *The* ***doc*** *alias is monitored only by the Informix departments that create and maintain manuals and on-line documentation files. It is not an appropriate channel for technical support issues, sales inquiries, or questions about the availability of Informix products.*

Chapter

1

Compiling INFORMIX-4GL Source Files

In This Chapter

This chapter describes how to create INFORMIX-4GL source-code modules, and how to produce executable 4GL programs from these modules, both at the operating system prompt and in the Programmer's Environment. Procedures to do this are shown for the INFORMIX-4GL C Compiler, as well as for the INFORMIX-4GL Rapid Development System. These two implementations of 4GL differ in how they process 4GL source-code modules. This chapter begins by identifying differences between the two implementations of 4GL. It then goes on to describe each implementation of 4GL.

Except as otherwise noted, the other chapters and appendixes of this manual describe features that are identical in both the C Compiler and the Rapid Development System implementations of 4GL.

Two Implementations of INFORMIX-4GL

To write a 4GL program, you must first create an ASCII file of 4GL statements that perform logical tasks to support your application. This chapter explains the procedures by which you can transform one or more source-code files of 4GL statements into an executable 4GL program. Informix offers two implementations of the 4GL application development language:

- The INFORMIX-4GL C Compiler, whose preprocessor generates extended ESQL/C source code. This code is further processed in several steps to produce C source code, which is compiled and linked as object code in an executable command file.
- The INFORMIX-4GL Rapid Development System (RDS), which uses a compiler to produce *pseudo-machine code* (called p-code) in a single step. You then invoke a *runner* to execute the p-code version of your application. For more details, see "Compiling and Linking Multiple Source Files" on page 1-35.

Important: *This version of the runner or Debugger cannot interpret programs compiled to p-code by releases of 4GL earlier than Version 7.30. You must first recompile your source files and form specifications. Similarly, releases of the 4GL runner or Debugger earlier than Version 7.30 cannot interpret p-code that this release produces.*

Runtime and Compile-Time Requirements

You must set the UNIX environment variable that specifies the pathname to the function libraries that 4GL requires to compile and execute the 4GL program. If you are using the Bourne or Korn shell, specify the following:

```
LD_LIBRARY_PATH=$INFORMIXDIR/lib:$INFORMIXDIR/lib/esql:$INFORMIXDIR/lib/
   tools
export LD_LIBRARY_PATH
```

If you are using the C shell, use these equivalent commands:

```
setenv LD_LIBRARY_PATH
$INFORMIXDIR/lib:$INFORMIXDIR/lib/esql:$INFORMIXDIR/lib/tools
```

Important: *On some platforms,* ***LD_LIBRARY_PATH*** *has a different name. Please see the machine-specific notes that are provided with 4GL for the name of this environment variable on your platform. Earlier releases of 4GL before Version 7.31 required a different setting.*

Differences Between the C Compiler and RDS Versions

Both implementations of 4GL use the same 4GL statements and nearly identical Programmer's Environments. Because they use different methods to compile 4GL source files into executable programs, however, there are a few differences in the user interfaces, as described in sections that follow.

Differences in the Programmer's Environment

The Programmer's Environment is a system of menus that supports the various steps in the process of developing 4GL application programs. The **Drop** option on the **PROGRAM** design menu of the C Compiler is called **Undefine** in the Rapid Development System implementation.

The **New** and **Modify** options of the **PROGRAM** design menu display a different screen form in the two implementations. Both of these screen forms are illustrated later in this chapter.

The Rapid Development System includes a **Debug** option on its **MODULE** design menu and **PROGRAM** design menu. This option does not appear in the C Compiler. (The Debugger is based on p-code, so it can execute only 4GL programs and modules that have been compiled by the Rapid Development System.)

The INFORMIX-4GL Interactive Debugger is available as a separate product.

Differences in Commands

The commands you use to enter the Programmer's Environments, compile and execute 4GL programs, and build or restore the **stores7** demonstration database vary between implementations of 4GL.

C Compiler	RDS	Effect of Command
i4gl	**r4gl**	Enter Programmer's Environment
c4gl *sfile*.**4gl**	**fglpc** *sfile*	Compile 4GL source file *sfile*.**4gl**
xfile.**4ge**	**fglgo** *xfile*	Execute compiled 4GL program *xfile*
i4gldemo	**r4gldemo**	Create the demonstration database

The C Compiler requires no equivalent command to the **fglgo** command, because its compiled object files are executable without a runner. The Rapid Development System also contains a command-file script to compile and execute 4GL programs that call C functions or INFORMIX-ESQL/C functions, as described in "RDS Programs That Call C Functions" on page 1-83.

Differences in Filename Extensions

The differences in filename extensions are as follows.

C Compiler	RDS	Significance of Extension
.o	**.4go**	Compiled 4GL source-code module
.4ge	**.4gi**	Executable (runable) 4GL program file

The backup file extensions **.4bo** and **.4be** for compiled modules and programs have the same names in both implementations. These files are not interchangeable between the two 4GL implementations, however, because object code produced by a C compiler is different from p-code.

Other filename extensions that are the same in both the C Compiler and the Rapid Development System designate interchangeable files, if you use both implementations of 4GL to process the same 4GL source-code module.

The C Compiler Version

This section describes the following aspects of the C compiler version of 4GL:

- The five steps of the compilation process
- All the menu options and screen form fields of the Programmer's Environment
- The steps for compiling and executing 4GL programs from the Programmer's Environment
- The equivalent command-line syntax for compiling and executing 4GL programs
- The filename extensions of 4GL source-code, object, error, and backup files

The Five-Phase 4GL Compilation Process

Versions of 4GL earlier than 6.0 were built on ESQL/C. To make 4GL more independent of ESQL/C, the compilation sequence requires one or more extra processes. Figure 1-1 on page 1-8 shows the five compilation phases in INFORMIX-4GL 6.0 and subsequent release versions. The five phases are as follows:

1. The i4glc1 preprocessor converts a 4GL source file with **.4gl** extension into a file with **.4ec** extension. It parses the 4GL language and generates C code to handle function and report definitions, computations, and function calls. It generates extended ESQL/C statements to handle forms, menus, input statements, and display statements, and pure ESQL/C to handle SQL statements and declarations of variables. **i4glc1** is similar to Version 4.12 and earlier **fglc**, except that **i4glc1** generates a **.4ec** file instead of a **.ec** file.
2. The **i4glc2** preprocessor translates the extended form, menu, input, and display statements to pure C code but leaves variable declarations and SQL statements unchanged. **i4glc2** accepts a **.4ec** file generated by **i4glc1** as input and produces a **.ec** file containing pure ESQL/C code.

3. The **i4glc3** preprocessor is a copy of the ESQL/C compiler. The **i4glc3** preprocessor accepts a **.ec** file (produced by **i4glc2** or written as pure ESQL/C code), and produces a **.c** file. The declarations and the SQL statements are mapped to pure C language code.

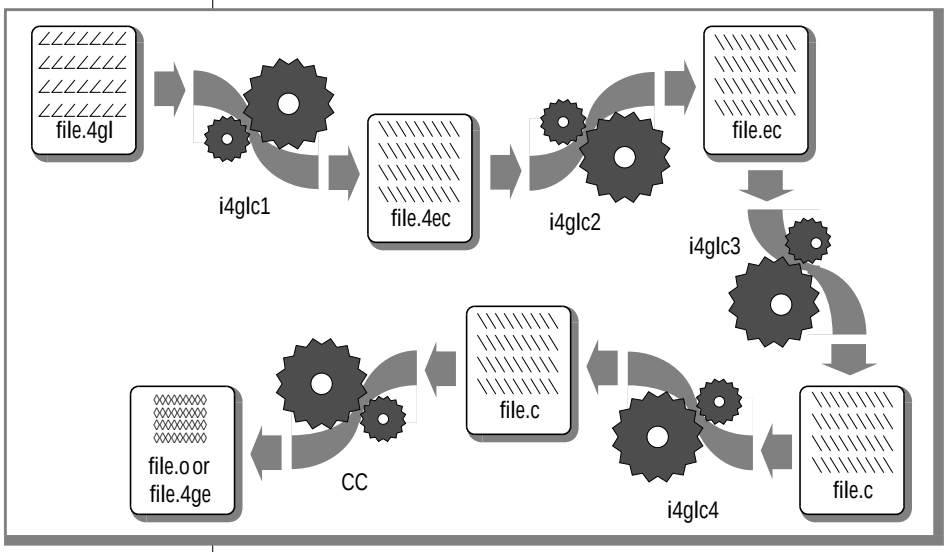


Figure 1-1
Five-Phase Compilation Process

4. The **i4glc4** preprocessor converts C code, which may contain non-ASCII characters in variable names, into de-internationalized names. This step ensures that defining a record like **table.*** or a field like **table.column** does not produce C code that contains non-ASCII character in identifiers (because very few C compilers accept non-ASCII characters in the names of variables).

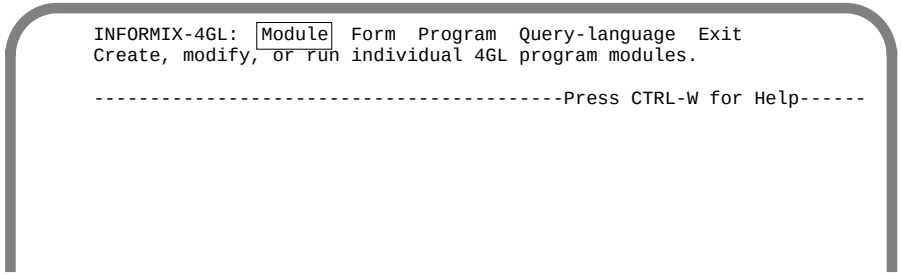
5. Informix uses the system C compiler to convert the C code generated by **i4glc3** or **i4glc4** into object files (a file with **.o** extension) and executable programs (a file with **.4ge** extension).

The Programmer's Environment

The C Compiler provides a series of nested menus, called the *Programmer's Environment*. These menus support the steps of 4GL program development and keep track of the components of your application. You can invoke the Programmer's Environment by entering `i4gl` at the system prompt.

The INFORMIX-4GL Menu

The **i4gl** command briefly displays the INFORMIX-4GL banner. Then a menu appears, called the **INFORMIX-4GL** menu.



This is the highest menu, from which you can reach any other menu of the Programmer's Environment. You have five options:

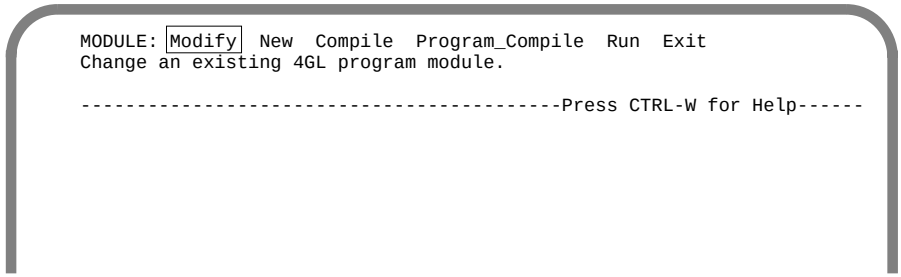
- **Module.** Work on a 4GL program module.
- **Form.** Work on a screen form.
- **Program.** Specify components of a multi-module program.
- **Query-language.** Use an SQL interactive interface, if you have either INFORMIX-SQL or DB-Access installed on your system. (See the documentation of these Informix products for details of their use.)
- **Exit.** Terminate **i4gl** and return to the operating system.

The first three options display new menus that are described in the pages that follow. (You can also press CONTROL-W at any menu to display an on-line help message that describes your options.) As at any 4GL menu, you can choose an option in either of two ways:

- By typing the first letter of the option
- By using the SPACEBAR or arrow keys to move the highlight to the option that you choose, and then pressing RETURN

The MODULE Design Menu

You can press RETURN or type m or M to choose the **Module** option of the **INFORMIX-4GL** menu. This displays a new menu, called the **MODULE** design menu. Use this menu to work on an individual 4GL source-code module.



Use this menu to create and compile source-code modules of a 4GL application. (For information on creating and compiling 4GL screen forms, see "The FORM Design Menu" on page 1-15. For details of how to create and compile help messages, see the **mkmessage** utility in Appendix B, "INFORMIX-4GL Utility Programs.")

The **MODULE** design menu supports the following options:

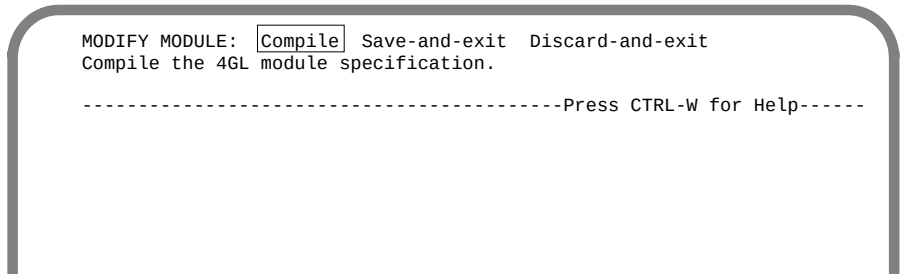
- **Modify.** Change an existing 4GL source-code module.
- **New.** Create a new source-code module.
- **Compile.** Compile a source-code module.
- **Program_Compile.** Compile a 4GL application program.
- **Run.** Execute a compiled 4GL program module or a multi-module application program.
- **Exit.** Return to the **INFORMIX-4GL** menu.

Within the Programmer's Environment, the **Exit** option returns control to the higher menu from which you accessed the current menu (or, when you choose **Exit** from the **INFORMIX-4GL** menu, terminates the 4GL session and returns to the system prompt).

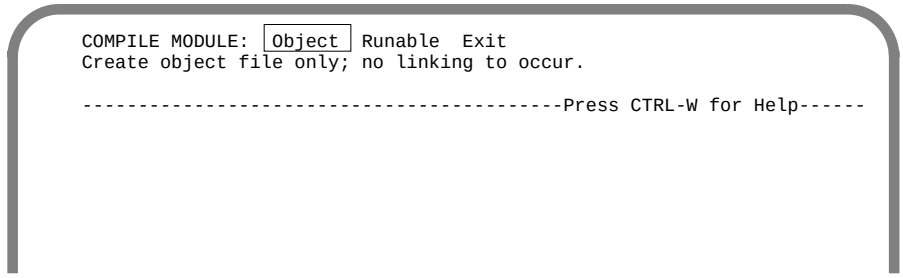
The Modify Option

Choose this option to edit an existing 4GL source-code module. If you choose this option, 4GL requests the name of the 4GL source-code file to be modified and then prompts you to specify a text editor. If you have designated a default editor with the **DBEDIT** environment variable (which is described in Appendix D) or if you specified an editor at the Programmer's Environment previously in this session, 4GL invokes that editor. The **.4gl** source file whose filename you specified is the current file.

When you leave the editor, 4GL displays the **MODIFY MODULE** menu, with the **Compile** option highlighted.



If you press RETURN or type c or C to choose the **Compile** option, 4GL displays the **COMPILE MODULE** menu:



The **Object** option creates a compiled file with the **.o** extension but makes no attempt to link the file with other files.

The **Runable** option creates a compiled file with the **.4ge** extension. 4GL assumes that the current module is a complete 4GL program, and that no other module needs to be linked to it. Choose the **Runable** option if the current program module is a stand-alone 4GL program. If this is not the case (that is, if the file is one of several 4GL source-code modules within a multi-module program), then you should use the **Object** option instead, and you must use the **PROGRAM** design menu to specify all the component modules.

After you choose **Object** or **Runable**, a message near the bottom of the screen will advise you if 4GL issues a compile-time warning or error. If there are warnings (but no errors), an object file is produced. Choose the **Exit** option of the next menu, and then **Save-and-exit** at the **MODIFY MODULE** menu, if you wish to save the executable file without reading the warnings.

Alternatively, you can examine the warning messages by choosing **Correct** at the next menu. When you finish editing the **.err** file that contains the warnings, you must choose **Compile** again from the **MODIFY MODULE** menu, because the **Correct** option deletes the executable file.

If there are compilation errors, the following menu appears.

```
COMPILE MODULE: Correct  Exit
Correct errors in the 4GL module.

-----------------------------------------Press CTRL-W for Help------
```

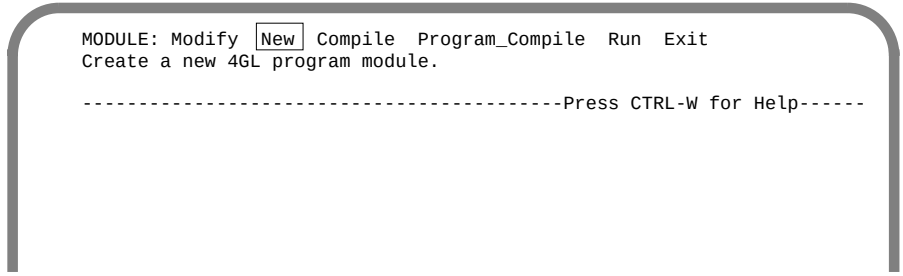
If you choose to correct the errors, an editing session begins on a copy of your source module with embedded error messages. You do not need to delete the error messages because 4GL does this for you. Correct your source file, save your changes, and exit from the editor. The **MODIFY MODULE** menu reappears, prompting you to recompile, save, or discard your changes without compiling.

If you choose not to correct the errors, you are prompted to **Save** or **Discard** the file.

If there are no compilation errors, the **MODIFY MODULE** menu appears with the **Save-and-Exit** option highlighted. Choose this option to save the current source-code module as a file with extension **.4gl**, and create an object file with the same filename, but with the extension **.o**. If you specified **Runable** when you compiled, the executable version is saved with the extension **.4ge**. The **Discard-and-Exit** option discards any changes that were made to your file after you chose the **Modify** option.

The New Option

Choose this option to create a new 4GL source-code module.

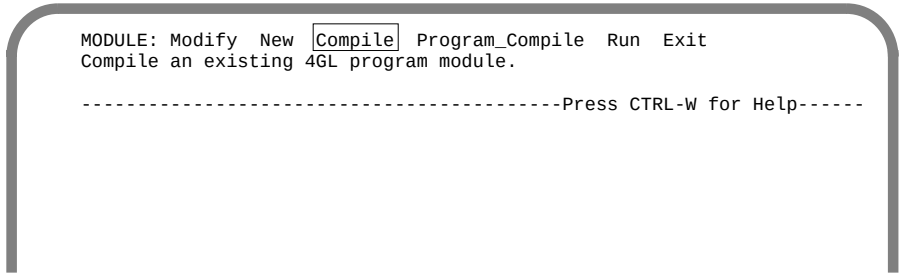


This option resembles the **Modify** option, but **NEW MODULE** is the menu title, and you must enter a new module name, rather than choose it from a list.

The filename of the module must be unique among source-code modules of the same 4GL program, and can include up to 10 characters, not including the **.4gl** file extension. If you have not designated an editor previously in this session or with **DBEDIT**, you are prompted for an editor. Then an editing session begins.

The Compile Option

The **Compile** option enables you to compile an individual 4GL source-code module.



After you specify the name of a 4GL source-code module to compile, the screen displays the **COMPILE MODULE** menu. Its **Object**, **Runable**, and **Exit** options were described earlier in the discussion of the **Modify** option.

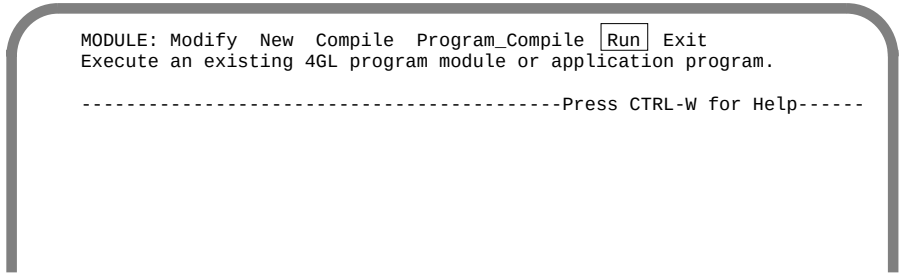
The Program_Compile Option

The **Program_Compile** option of the **MODULE** design menu is the same as the **Compile** option of the **PROGRAM** design menu. This option can compile and link modules, as described in the program specification database, taking into account the time when the modules were last updated.

This option is useful after you modify a single module of a complex program, and need to test it by compiling and linking it with the other modules.

The Run Option

Choose the **Run** option to begin execution of a compiled 4GL program.

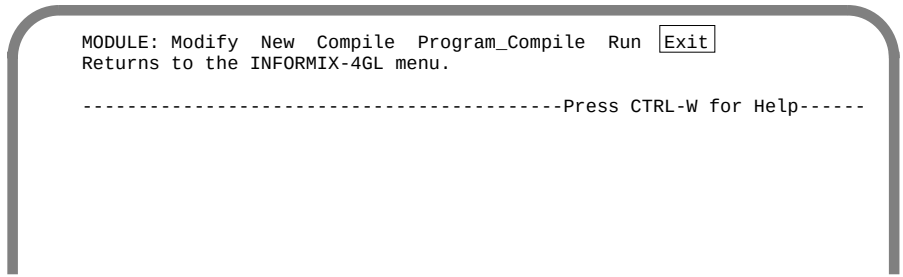


The RUN PROGRAM screen lists compiled modules and programs, with the highlight on the module corresponding to the current file, if any has been specified. Only compiled programs with extension **.4ge** are listed. If you compile a program outside the Programmer's Environment and you want it to appear in this list, give it the extension **.4ge**.

If no compiled programs exist, 4GL displays an error message and returns to the **MODULE** design menu. You can exit to the maine **INFORMIX-4GL** menu, and select the **Program** option to create the database.

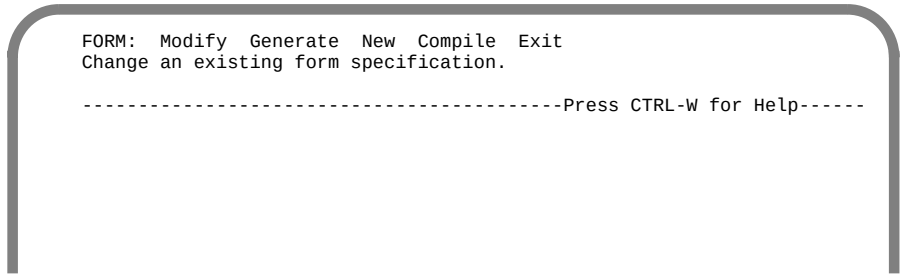
The Exit Option

Choose this option to exit from the **MODULE** design menu and display the **INFORMIX-4GL** menu.



The FORM Design Menu

You can type `f` or `F` at the **INFORMIX-4GL** menu to choose the **Form** option. This option displays a menu, called the **FORM** design menu.



You can use this menu to create, modify, and compile *screen form* specifications. These define visual displays that 4GL applications can use to query and modify the information in a database. 4GL form specification files are ASCII files that are described in Chapter 6, "Screen Forms."

The **FORM** design menu supports the following options:

- **Modify.** Change an existing 4GL screen form specification.
- **Generate.** Create a default 4GL screen form specification.
- **New.** Create a new 4GL screen form specification.
- **Compile.** Compile an existing 4GL screen form specification.
- **Exit.** Return to the **INFORMIX-4GL** menu.

Readers familiar with INFORMIX-SQL may notice that this resembles the menu displayed by the **Form** option of the INFORMIX-SQL main menu.

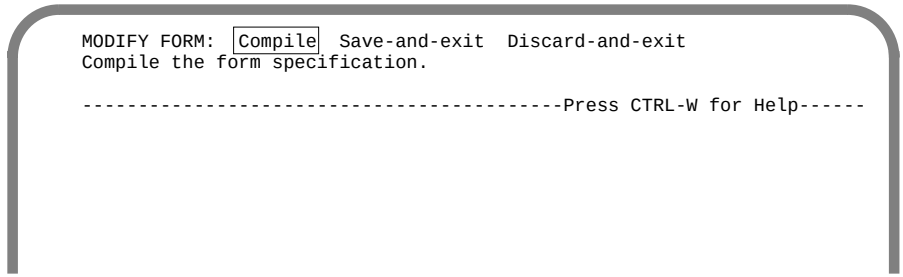
The Modify Option

The **Modify** option of the **FORM** design menu enables you to edit an existing form specification file. It resembles the **Modify** option in the **MODULE** design menu, because both options are used to edit program modules.

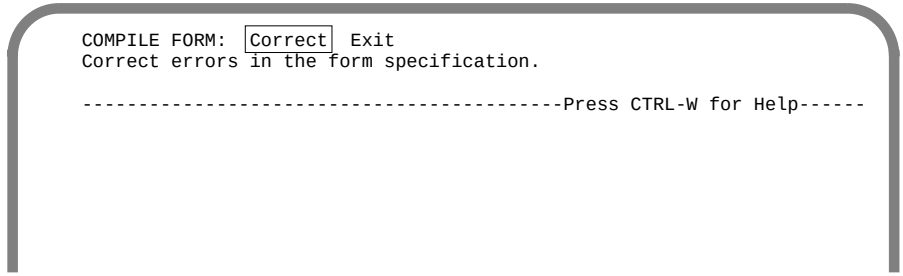
```
FORM:  Modify  Generate  New  Compile  Exit
Change an existing form specification.

-----------------------------------------Press CTRL-W for Help------
```

If you choose this option, you are prompted to choose the name of a form specification file to modify. Source files created at the **FORM** design menu have the file extension **.per**. (If you use a text editor outside of the Programmer's Environment to create form specification files, you must give them the extension **.per** before you can compile them with the FORM4GL screen form facility.)

If you have not already designated a text editor in this 4GL session or with **DBEDIT**, you are prompted for the name of an editor. Then an editing session begins, with the form specification source-code file that you specified as the current file. When you leave the editor, 4GL displays the **MODIFY FORM** menu with the **Compile** option highlighted. Now you can press RETURN to compile the revised form specification file.



If there are compilation errors, 4GL displays the **COMPILE FORM** menu.

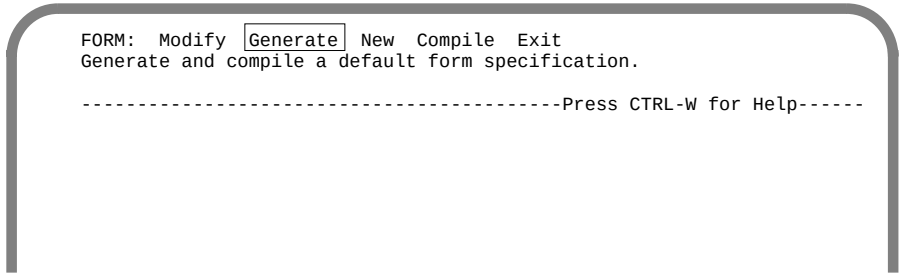


Press RETURN to choose **Correct** as your option. An editing session begins on a copy of the current form, with diagnostic error messages embedded where the compiler detected syntax errors. 4GL automatically deletes these messages when you save the file and exit from the editor. After you have corrected the errors, the **MODIFY FORM** menu appears again, with the **Compile** option highlighted. Press RETURN to recompile. Repeat these steps until the compiler reports no errors. (If you choose **Exit** instead of **Correct**, you are prompted to **Save** or **Discard** the file.)

If there are no compilation errors, you are prompted to save the modified form specification file and the compiled form, or to discard the changes. (Discarding the changes restores the version of your form specifications from before you chose the **Modify** option.)

The Generate Option

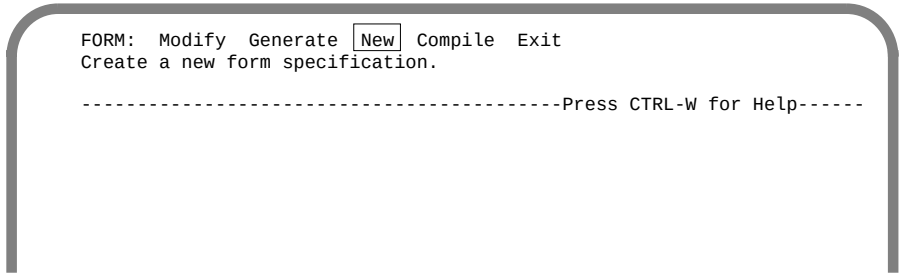
You can type g or G to choose the **Generate** option. This option creates a simple *default* screen form that you can use directly in your program, or that you can later edit by choosing the **Modify** option.



When you choose this option, 4GL prompts you to select a database, to choose a filename for the form specification, and to identify the tables that the form will access. After you provide this information, 4GL creates and compiles a form specification file. (This is equivalent to running the **-d** (default) option of the **form4gl** command, as described in "Compiling a Form at the Command Line" on page 6-87.)

The New Option

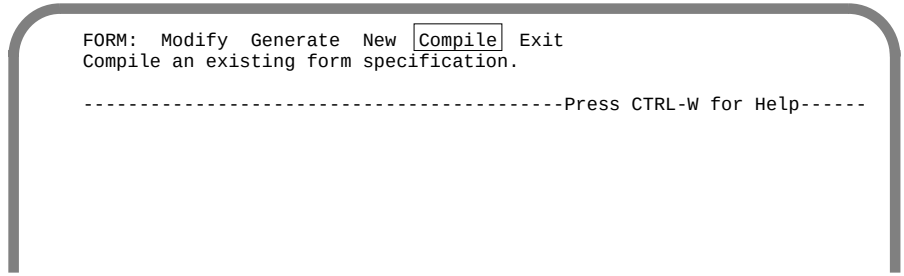
The **New** option of the **FORM** design menu enables you to create a new screen form specification.



After prompting you for the name of your form specification file, 4GL places you in the editor where you can create a form specification file. When you leave the editor, 4GL transfers you to the **NEW FORM** menu that is like the **MODIFY FORM** menu. You can compile your form and correct it in the same way.

The Compile Option

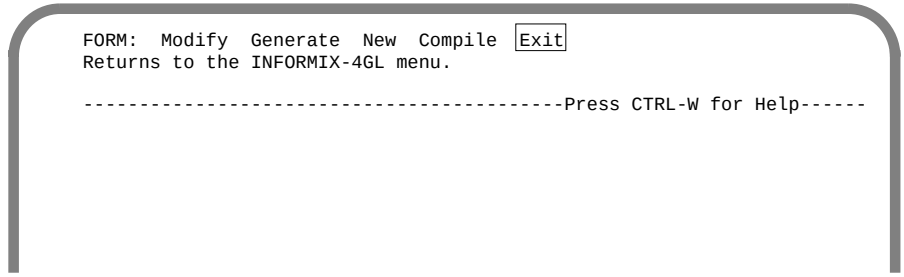
The **Compile** option enables you to compile an existing form specification file without going through the **Modify** option.



4GL compiles the form specification file whose name you specify. If the compilation fails, 4GL displays the **COMPILE FORM** menu with the **Correct** option highlighted.

The Exit Option

The **Exit** option restores the **INFORMIX-4GL** menu.



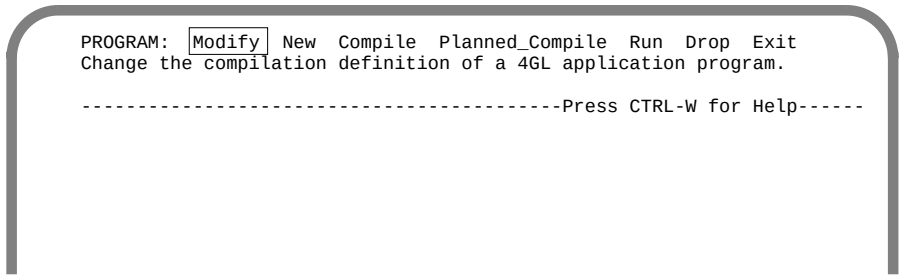
The PROGRAM Design Menu

A 4GL program can be a single source-code module that you create and compile at the **MODULE** design menu. For applications of greater complexity, however, it is often easier to create separate 4GL modules. The **INFORMIX-4GL** menu includes the **Program** option to create multi-module programs. If you choose this option, 4GL searches your **DBPATH** directories for the *program design database*, which stores the names of the objects that are used to create programs and their build dependencies. (For more information on the **DBPATH** environment variable, see Appendix D.)

This program design database describes the component modules and function libraries of your 4GL program. By default, its name is **syspgm4gl**, but you can use the **PROGRAM_DESIGN_DBS** environment variable to specify some other name. (For more information on the **PROGRAM_DESIGN_DBS** environment variable, see Appendix D.)

If 4GL cannot find this database, you are asked if you want one created. If you enter y in response, 4GL creates the **syspgm4gl** database, grants CONNECT privileges to PUBLIC, and displays the **PROGRAM** design menu. As database administrator of **syspgm4gl**, you can later restrict the access of other users.

If **syspgm4gl** already exists, the **PROGRAM** design menu appears.



You can use this menu to create or modify a multi-module 4GL program specification, to compile and link a program, or to execute a program.

The **PROGRAM** design menu supports the following options:

- **Modify.** Change an existing program specification.
- **New.** Create a new program specification.
- **Compile.** Compile an existing program.
- **Planned_Compile.** List the steps necessary to compile and link an existing program.
- **Run.** Execute an existing program.
- **Drop.** Delete an existing program specification.
- **Exit.** Return to the **INFORMIX-4GL** menu.

You must first use the **MODULE** design menu and the **FORM** design menu to enter and edit the 4GL statements within the component source-code modules of a 4GL program. Then you can use the **PROGRAM** design menu to identify which modules are part of the same application program, and to link all the modules as an executable command file.

The Modify Option

The **Modify** option enables you to modify the specification of an existing 4GL program. (This option is not valid unless at least one program has already been specified. If none has, you can create a program specification by choosing the **New** option from the same menu.)

4GL prompts you for the name of the program specification to be modified. It then displays a menu and form that you can use to update the information in the program specification database, as shown in Figure 1-2.

Figure 1-2
Example of a Program Specification Entry

```
MODIFY PROGRAM:  4GL  Other  Libraries  Compile_Options  Rename  Exit
Edit the 4GL sources list.

-----------------------------------------Press CTRL-W for Help------
 Program
 [myprog  ]

 4gl Source    4gl Source Path
 [main      ]  [/u/john/appl/4GL                          ]
 [funct     ]  [/u/john/appl/4GL                          ]
 [rept      ]  [/u/john/appl/4GL                          ]
 [          ]  [                                          ]
 [          ]  [                                          ]

 Other Source  Ext    Other Source Path
 [cfunc     ]  [c  ]  [/u/john/appl/C                             ]
 [          ]  [   ]  [                                           ]
 [          ]  [   ]  [                                           ]
 [          ]  [   ]  [                                           ]

 Libraries [m       ]           Compile Options [          ]
           [        ]                           [          ]
```

The name of the program appears in the **Program** field. In Figure 1-2 the name is **myprog**. You can change this name by choosing the **Rename** option. 4GL assigns the program name, with the extension **.4ge**, to the executable program produced by compiling and linking all the source files and libraries. (Compiling and linking occurs when you choose the **Compile_Options** option, as described later in this section.) In this example, the resulting executable program would have the name **myprog.4ge**.

Use the **4GL** option to update the entries for the **4gl Source** fields and the **4gl Source Path** fields on the form. The five rows of fields under these labels form a screen array. When you choose the **4GL** option, 4GL executes an INPUT ARRAY statement so you can move and scroll through the array. See the INPUT ARRAY statement in Chapter 4, "INFORMIX-4GL Statements," for information about how to use your function keys to scroll, delete rows, and insert new rows. (You cannot redefine the function keys, however, as you can with a 4GL program.)

The 4GL source program that appears in Figure 1-2 on page 1-22 contains three modules:

- One module contains the main program (**main.4gl**).
- One module contains functions (**funct.4gl**).
- One module contains REPORT statements (**rept.4gl**).

Each module is located in the directory **/u/john/appl/4GL**.

If your program includes a module containing only global variables (for example, **global.4gl**), you must also list that module in this section.

Use the **Other** option to include non-4GL source modules or object-code modules in your program. Enter this information into the three-column screen array with the headings **Other Source**, **Ext**, and **Other Source Path**. Enter the filename and location of each non-4GL source-code or object-code module in these fields. Enter the name of the module in the **Other Source** field, the filename extension of the module (for example, ec for an INFORMIX-ESQL/C module, or c for a C module) in the **Ext** field, and the full directory path of the module in the **Other Source Path** field. The example in Figure 1-2 includes a file containing C function source-code (**cfunc.c**) located in **/u/john/appl/C**. You can list up to 100 files in this array.

The **Libraries** option enables you to indicate the names of up to ten special libraries to link with your program. 4GL calls the C compiler to do the linking and adds the appropriate **-l** prefix, so you should enter only what follows the prefix. The example displayed in Figure 1-2 calls only the standard C math library.

Use the **Compile_Options** option to indicate up to ten C compiler options. Enter this information in the **Compile Options** field. Do not, however, specify the **-e** or **-a** options of **c4gl** in this field, because they will cause the compilation to fail. (See "Creating Programs at the Command Line" on page 1-33 for more information about the options of the **c4gl** command.)

The **Exit** option exits from the **MODIFY PROGRAM** menu and displays the **PROGRAM** design menu.

The New Option

Use the **New** option on the **PROGRAM** design menu to create a new specification of the program modules and libraries that make up an application program. You can also specify any necessary compiler or loader options.

```
PROGRAM:  Modify [New] Compile  Planned_Compile  Run  Drop  Exit
Add the compilation definition of a 4GL application program.

-----------------------------------------Press CTRL-W for Help------
```

The submenu screen forms displayed by the **New** and the **Modify** options of the **PROGRAM** design menu are identical, except that you must first supply a name for your program when you choose the **New** option. (4GL displays a blank form in the **NEW PROGRAM** menu.) The **NEW PROGRAM** menu has the same options as the **MODIFY PROGRAM** menu, as illustrated earlier.

The Compile Option

The **Compile** option performs the compilation and linking described in the program specification database, taking into account the time when each file was last updated. It compiles only those files that have not been compiled since they were last modified.

```
PROGRAM:  Modify  New [Compile] Planned_Compile  Run  Drop  Exit
Compile a 4GL application program.

-----------------------------------------Press CTRL-W for Help------
```

4GL lists each step of the preprocessing and compilation as it occurs. An example of these messages appears in the illustration of the **Planned_Compile** option, next.

The Planned_Compile Option

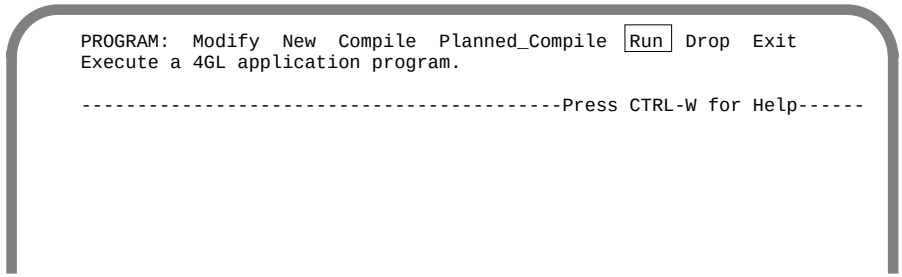
Taking into account the time when the various files in the dependency relationships last changed, the **Planned_Compile** option prompts for a program name and displays a summary of the steps that will be executed if you choose the **Compile** option. No compilation actually takes place.

```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Drop  Exit
Show the planned compile actions of a 4GL application program.

-----------------------------------------Press CTRL-W for Help------
Compiling INFORMIX-4GL sources:
                /u/john/appl/4GL/main.4gl
                /u/john/appl/4GL/funct.4gl
                /u/john/appl/4GL/rept.4gl
Compiling Embedded SQL sources:
Compiling with options:
Linking with libraries:
                m
Compiling/Linking other sources:
                /u/john/appl/C/cfunc.c
```

In this instance, changes were made to all the components of the 4GL program that were listed in Figure 1-2 on page 1-22. This display indicates that no source-code module has been compiled after the program was changed.

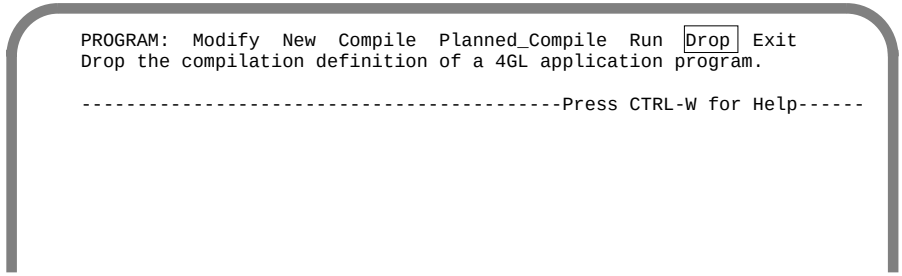
The Run Option

The **Run** option of the **PROGRAM** design menu is the same as the **Run** option of the **MODULE** design menu. It displays a list of any compiled programs (files with the extension **.4ge**) and highlights the current program, if a program has been specified. 4GL then executes the program that you choose.



The Drop Option

The **Drop** option of the **PROGRAM** design menu prompts you for a program name and removes the compilation and linking definition of that program from the **syspgm4gl** database. This action removes the definition only. Your program and 4GL modules are *not* removed.



The Exit Option

The **Exit** option clears the **PROGRAM** design menu and restores the **INFORMIX-4GL** menu.

The QUERY LANGUAGE Menu

The SQL interactive interface is identical to the interactive SQL interface of INFORMIX-SQL. If you do not have INFORMIX-SQL, 4GL uses the DB-Access utility. The **Query-language** option is placed at the top-level menu so you can test SQL statements without leaving the 4GL Programmer's Environment. You can also use this option to create, execute, and save SQL scripts.

Creating Programs in the Programmer's Environment

To invoke the C Compiler version of the Programmer's Environment, enter the following command at the system prompt:

```
i4gl
```

After a sign-on message is displayed, the **INFORMIX-4GL** menu appears.

To create a 4GL application with the C Compiler version of 4GL

1. Create a new source module or revising an existing source module
2. Compile the source module
3. Link the program modules
4. Execute the compiled program

This process is described in the sections that follow.

Creating a New Source Module

This section outlines the procedure for creating a new source module. If your source module already exists, see "Revising an Existing Module," next.

To create a source module

1. Choose the **Module** option of the **INFORMIX-4GL** menu.

   The **MODULE** design menu is displayed.

2. If you are creating a new **.4gl** source module, choose the **New** option of the **MODULE** design menu.

3. Enter a name for the new module.

   The name must begin with a letter and can include letters, numbers, and underscores. No more than 10 characters are allowed in this name, which must be unique among the files in the same directory, and among any other modules of the same program. 4GL attaches the extension **.4gl** to this filename of your new module.

4. Press RETURN.

Revising an Existing Module

If you are revising an existing 4GL source file, use the following procedure.

To modify a source file

1. Choose the **Modify** option of the **MODULE** design menu.

   The screen lists the names of all the **.4gl** source modules in the current directory and prompts you to choose a source file to edit.

2. Use the arrow keys to highlight the name of a source module and press RETURN, or enter a filename (with no extension).

   If you specified the name of an editor with the **DBEDIT** environment variable, an editing session with that editor begins automatically. Otherwise, the screen prompts you to specify a text editor.

   Specify a text editor, or press RETURN for **vi**, the default editor. Now you can begin an editing session by entering 4GL statements.

3. When you have finished entering or editing your 4GL code, use an appropriate editor command to save your source file and end the text editing session.

Compiling a Source Module

The **.4gl** source file module that you create or modify is an ASCII file that must be compiled before it can be executed.

To compile a module

1. Choose the **Compile** option from the **MODULE** design menu.
2. Choose the type of module that you are compiling, either **Runable** or **Object**.

   If the module is a complete 4GL program that requires no other modules, choose **Runable**.

   If the module is one module of a multi-module 4GL program, choose **Object**. This option creates a compiled object file module, with the same filename, but with extension **.o**. See also the next section, "Linking Program Modules."
3. If the compiler detects errors, no compiled file is created, and you are prompted to fix the problem.

   Choose **Correct** to resume the previous text editing session, with the same 4GL source code, but with error messages in the file. Edit the file to correct the error, and choose **Compile** again. If an error message appears, repeat this process until the module compiles without error.
4. After the module compiles successfully, choose **Save-and-exit** from the menu to save the compiled program.

   The **MODULE** design menu appears again on your screen.
5. If your program requires *screen forms*, choose **Form** from the **INFORMIX-4GL** menu.

   The **FORM** design menu appears. For information about designing and creating screen forms, see Chapter 6.
6. If your program displays help messages, you must create and compile a help file.

   Use the **mkmessage** utility to compile the help file. For more information on this utility, see Appendix B.

Linking Program Modules

If your new or modified module is part of a multi-module 4GL program, you must link all of the modules into a single program file before you can run the program. If the module that you compiled is the only module in your program, you are now ready to run your program. (See "Executing a Compiled Program" on page 1-33.)

To link modules

1. Choose the **Program** option from the **INFORMIX-4GL** menu.

   The **PROGRAM** design menu appears.

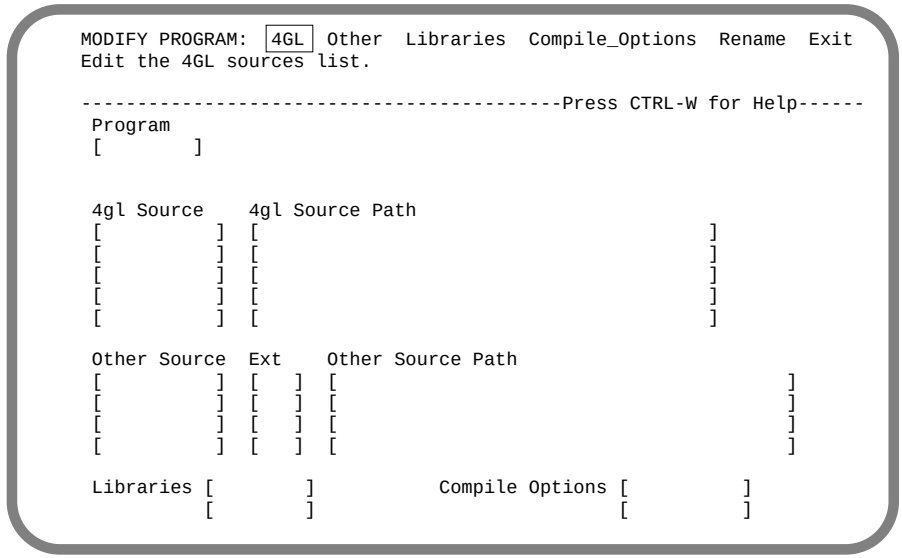
2. If you are creating a new multi-module 4GL program, choose the **New** option; if you are modifying an existing one, choose **Modify**.

   In either case, the screen prompts you for the name of a program.

3. Enter the name (without a file extension) of the program that you are modifying, or the name to be assigned to a new program.

Names must begin with a letter, and can include letters, underscores (_), and numbers. After you enter a valid name, the PROGRAM screen appears, with your program name in the first field.

If you chose **Modify**, the names and pathnames of the source-code modules are also displayed. In that case, the PROGRAM screen appears below the **MODIFY PROGRAM** menu, rather than below the **NEW PROGRAM** menu. (Both menus list the same options.)



4. Identify the files that make up your program:

   - To specify new 4GL modules or edit the list of 4GL modules, choose the **4GL** option.

     You can enter or edit the name of a module, without the **.4gl** file extension. Repeat this step for every module. If the module is not in the current directory or in a directory specified by the **DBPATH** environment variable, enter the pathname to the directory where the module resides.

   - To include any modules in your program that are not 4GL source files, choose the **Other** option.

     This option enables you to specify each filename in the **Other Source** field, the filename extension in the **Ext** field, and the pathname in the **Other Source Path** field.

     These fields are part of an array that can specify up to 100 *other* modules, such as C language source files or object files. If you have INFORMIX-ESQL/C installed on your system, you can also specify ESQL/C source modules (with extension **.ec**) here.

   - To specify any function libraries that should be linked to your program (besides the 4GL library that is described in Chapter 5, "Built-In Functions and Operators"), choose the **Libraries** option. This option enables you to enter or edit the list of library names in the **Libraries** fields.

   - To specify compiler flags, choose the **Compile_Options** option. These flags can be entered or edited in the **Compile Options** fields.

5. After you have correctly listed all of the modules of your program, choose the **Exit** option to return to the **PROGRAM** design menu.

6. Choose the **Compile** option of the **PROGRAM** design menu.

   This option produces an executable file that contains all your 4GL program modules. Its filename is the program name that you specified, with extension **.4ge**. The screen lists the names of your **.4gl** source modules, and displays the **PROGRAM** design menu with the **Run** option highlighted.

Executing a Compiled Program

After compiling and linking your program modules, you can execute your program. To do so, choose the **Run** option from the **MODULE** or **PROGRAM** design menu. This option begins execution of the compiled 4GL program.

Your program can display menus, screen forms, windows, or other screen output, according to your program logic and any keyboard interaction of the user with the program.

Creating Programs at the Command Line

You can also create **.4gl** source files and compiled **.o** and **.4ge** files in makefiles or at the operating system prompt. Figure 1-3 shows the process of creating, compiling, linking, and running a 4GL program from the command line.

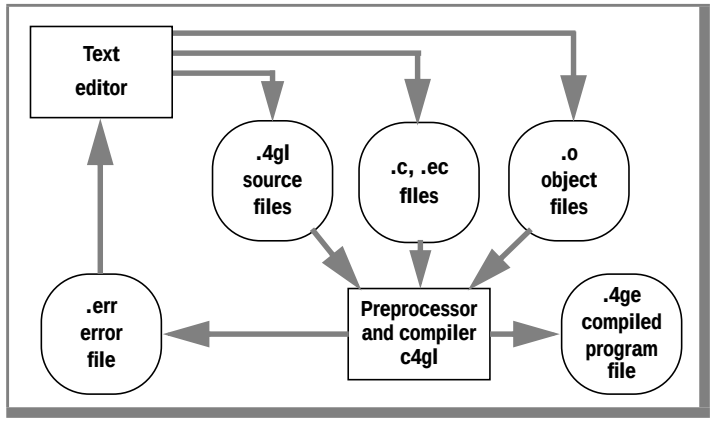


Figure 1-3
Creating and Running a 4GL Program

In Figure 1-3 the rectangles represent processes controlled by specific commands, and the circles represent files. Arrows indicate whether a file can serve as input or output (or both) for a process. This diagram is simplified and ignores the similar processes by which forms, help messages, and other components of 4GL applications are compiled, linked, and executed. The diagram outlines the following process:

- The cycle begins in the upper-left corner with a text editor, such as **vi**, to produce a 4GL source module.
- A multi-module program can include additional 4GL source files (**.4gl**), ESQL/C source files (**.ec**), C language source files (**.c**), and object files (**.o**).
- The program module can then be compiled by invoking the **c4gl** preprocessor and compiler command. (If error messages result, find them in the **.err** file and edit the source file to correct the errors. Then recompile the corrected source module.)

  The resulting compiled **.4ge** program file is an executable command file that you can run by entering its name at the system prompt:

  ```
  filename.4ge
  ```

  Here *filename*.**4ge** specifies your compiled 4GL file.

The following table shows the correspondence between commands and menu options.

Menu Option	Invokes	Command
Module New/Modify	UNIX System Editor	vi
Compile	4GL Preprocessor/C Compiler	**c4gl**
Run	4GL Application	*filename*.**4ge**

For information on the use of makefiles to create 4GL applications, visit the Informix Developer Network (IDN) on the Web at the following URL:

```
http://www.informix.com/idn
```

Creating or Modifying a 4GL Source File

Use your system editor or another text editing program to create a **.4gl** source file or to modify an existing file. For information on the statements that you can include in a 4GL program, see Chapter 4, "INFORMIX-4GL Statements."

Compiling a 4GL Module

You can compile a 4GL source file by entering a command of the form:

```
c4gl source.4gl -o filename.4ge
```

The **c4gl** command compiles your 4GL source-code module (here called *source***.4gl**) and produces an executable program called *filename***.4ge**. The complete syntax of the **c4gl** command appears in the next section.

Compiling and Linking Multiple Source Files

A 4GL program can include several source-code modules. You cannot execute a 4GL program until you have preprocessed and compiled all the source modules and linked them with any function libraries that they reference.

You can do all this in a single step at the system prompt by using the **c4gl** command, which performs the following processing steps:

1. Invokes the **i4glc1** preprocessor, which reads your 4GL source-code files (extension **.4gl**) and preprocesses them to produce extended ESQL/C code (extension **.4ec**).
2. Invokes the **i4glc2** preprocessor, which reads the extended ESQL/C code and preprocesses it to produce ESQL/C code (extension **.ec**).
3. Invokes the **i4glc3** preprocessor, which reads the ESQL/C code and preprocesses it to produce C code (extension **.c**).
4. Invokes the **i4glc4** preprocessor, which reads the C code and compiles it to produce an object file (extension **.o**).
5. Links the object file to the ESQL/C libraries and to any additional libraries that you specify in the command line.

You must assign the filename extension **.4gl** to 4GL source-code modules that you compile. The resulting **.4ge** file is an executable version of your program.

Tip: *The ESQL/C source files (with extension* ***.ec****), C source files (with extension* ***.c****), and C object files (with extension* ***.o****) are intermediate steps in producing an executable 4GL program. Besides* ***.4gl*** *source files, you can also include files of any or all of these types when you specify a* ***c4gl*** *command to compile and link the component modules of a 4GL program.*

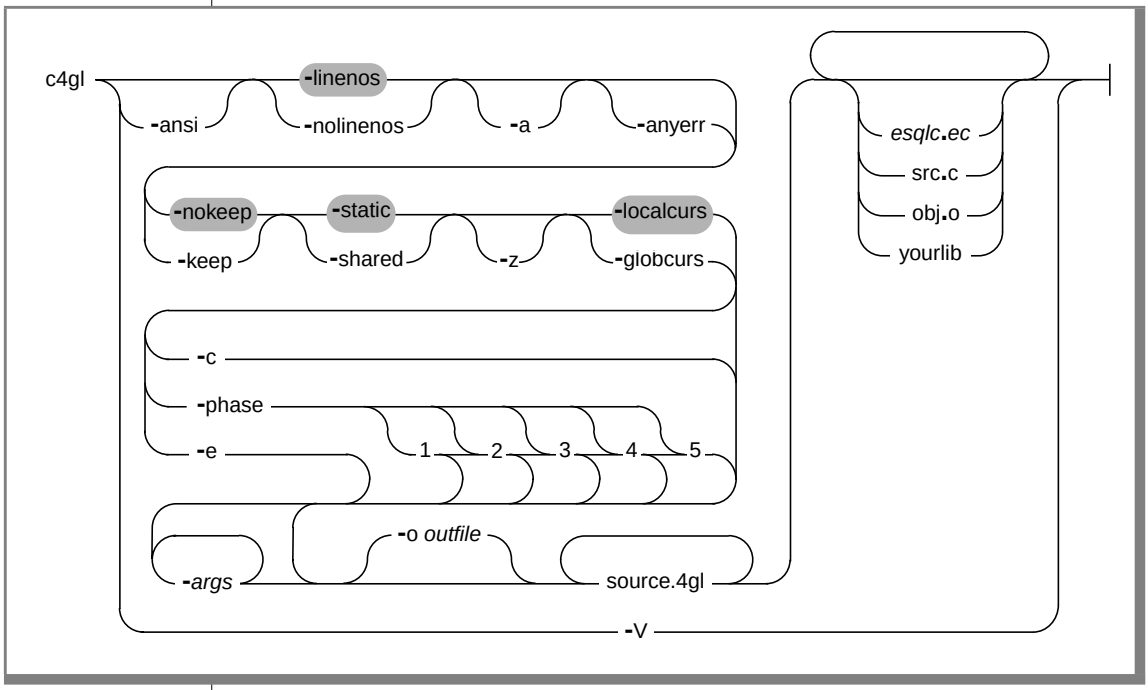
Using the c4gl Script for Compilation

For compatibility with Informix database servers, releases of INFORMIX-4GL must take into account that the database, table, and column names might contain non-ASCII characters. Most C compilers do not accept these in variable names, but the **i4glc4** process now maps the locale-dependent characters so that C compilers will accept them.

The Programmer's Environment invokes **c4gl** to do the compilation, using the **-phase** option described in the next section. The Programmer's Environment now runs **c4gl -phase 12** (**i4glc1** and **i4glc2**) as its own phase 1, followed by **c4gl -phase 34** (**i4glc3** and **i4glc4**) as its own phase 2, and finally it runs **c4gl -phase 5** (the C compiler) as its own phase 3. Because the **c4gl** script is now used for the compilations, however, you can also use the **C4GLFLAGS** environment variable (see Appendix D) with the Programmer's Environment. (This is equivalent to the **-a** option at the command line.)

c4gl Command

The **c4gl** command is a shell script that supports the following syntax.



Element	Description
-args	are other arguments for your C compiler.
***esql*.ec**	is an ESQL/C source file to compile and link.
***obj*.o**	is an object file to link with your 4GL program.
outfile	is a name that you assign to the compiled 4GL program.
***source*.4gl**	is the name of a 4GL source module and its **.4gl** extension.
***src*.c**	is a C language source file to compile and link.
yourlib	is a function library other than the 4GL or ESQL/C libraries.

The **c4gl** command passes all C compiler arguments (*args*) and other C source and object files (***src*.c**, ***obj*.o**) directly to the C compiler (typically **cc**).

If you omit the **-o** *outfile* option, the default filename is **a.out**.

You can compile 4GL modules separately from your MAIN program block. If there is no MAIN program block in ***source*.4gl**, your code is compiled to *source*.**o** but is not linked with other modules or libraries. You can use **c4gl** to link your code with a module that includes the MAIN program block at another time. (For more information, see "MAIN" on page 4-245.) If you typically compile with the same options, you can set the **C4GLFLAGS** environment variable to supply those options implicitly. See the section "C4GLFLAGS" on page D-10 for details of this feature.

To display the release version number of your SQL software, use the **-V** option. If you specify the **-V** option, all other arguments are ignored, and no output files are produced.

The -phase Option

The **c4gl** compilation command recognizes the five phases of compilation and can specify which phases to perform. To perform only the preprocessor steps, with no compilation or linking, include the **-e** option.

More generally, the **-phase** option takes an argument, separated from **-phase** by one or more blank spaces. This argument can contain any contiguous sequence of the positive integers in the range from 1 to 5 (that is, 1, 2, 3, 4, 5, 12, 23, 34, 45, 123, 234, 345, 1234, 2345, 12345). These digits specify which phases of compilation to perform. The **-c** option implies **-phase 12345**, and the **-e** option implies **-phase 1234**,.

ANSI Compliance

To instruct the compiler to check all SQL statements for compliance with the ANSI/ISO standard for SQL syntax, include the **-ansi** option. If you specify the **-ansi** option, it must appear first in your list of **c4gl** command arguments. The **-ansi** option asks for compile-time and runtime warning messages if your source code includes Informix extensions to the ANSI/ISO standard for SQL. Compiler warnings and error messages are saved in a file called ***source*.err**.

ANSI C compilers generate a warning if line numbers generated from the compilation are greater than 32767. Line numbers greater than 32767 can occur in compiled 4GL when the underlying ESQL/C compiler works on a large program. You can suppress these warnings with the **-nolinenos** option of **c4gl**. You can also explicitly set the default ANSI warnings with the **-linenos** option.

Array Bounds

To have your compiled program check array bounds at runtime, include the **-a** option, which must appear on the command line before the ***source*.4gl** filename. The **-a** option requires additional runtime processing, so you might prefer to use this option only during development to debug your program.

Error Scope

If you specify the **-anyerr** option, 4GL sets the **status** variable after evaluating expressions. The **-anyerr** option overrides any WHENEVER ERROR statements in your program.

Intermediate Files

When the compilation completes successfully, **c4gl** automatically removes the intermediate files with extensions **.c**, **.ec**, and **.4ec**, which are generated during the first four phases of compilation. (Some earlier versions of 4GL did not delete these files.) If the compilation fails or is interrupted, however, all the intermediate files are left intact.

The **-keep** option explicitly specifies that the intermediate files be retained. The default is the **-nokeep** option, which specifies that the intermediate files be removed. The **.o** file is retained if you specify the **-c** flag, but if an executable is produced, whether the **.o** file is kept or removed depends on the C compiler in use. Some compilers keep the **.o** file, and others remove it depending on what else you specify on the command line. If you direct **c4gl** to do **-phase 1234**, the **.c** file is no longer an intermediate file and it is retained. Similarly, if you request **-phase 1**, the **.4ec** files are no longer intermediate files, and so they are kept.

Informal Functions

The **-z** option enables **c4gl** to compile a program that invokes a single function with a variable number of arguments without **i4glc1** giving an error at compile time.

Although **fglc** supports the **-z** option, some earlier releases of **c4gl** ignore the option, so it is not possible to use the standard script to compile programs that include such functions. (Most developers should not use this option, because it suppresses error messages for all functions with variable numbers of arguments.)

The -globcurs and -localcurs Options

In ESQL/C releases prior to Version 5.00, the scope of reference of names of cursors and prepared statements is local to a single source code module; the same name can be reused without conflict in different modules. All 4.1x versions of 4GL used a 4.1x version of ESQL/C. In Version 5.00 and later of ESQL/C, all cursor and prepared statement names are global by default. Thus the cursor **c_query** in **filea.ec** is the same as the cursor **c_query** in **fileb.ec**.

To preserve the legacy behavior, the compiler mangles all cursor and prepared statement names using the same algorithm in both compilers. See "CURSOR_NAME( )" on page 5-53 for the mangling algorithm. (Contact Informix Technical Support if any pairs of cursor or prepared statement names are mangled to the same value. The workaround for mangled-name conflicts is to change one of the affected cursor or statement names.)

The **-globcurs** option makes the names of cursors and of prepared objects global to the entire program. The compilers still require you to declare the cursor before using it for any other purpose in the module, so this option is seldom useful. This option might help in debugging, however, because the cursor names are not modified.

The **-localcurs** option can override the **-globcurs** option if that was set in the **C4GLFLAGS** environment variable. The **-localcurs** option makes the names of cursors and prepared objects local to the module in which they were declared.

Using Source Code Debuggers with 4GL Programs

The primary conversion of **i4glc4** ensures that the generated C code will compile when you use 4GL with an NLS or GLS database. In such databases, the table and column names can contain non-ASCII characters ranging from 128 to 255. You can define variables using the RECORD LIKE *table*.* to refer to these names, but C compilers do not normally allow variable names to contain such characters. To avoid compilation problems with the C compiler, **i4glc4** adjusts the code.

To make the variable names safe, Informix replaces any non-ASCII characters that occur outside quoted strings (which only happens in variable or function names) with a mapped value. For values in the range `0xA0` to `0xFF`, Informix uses the hexadecimal value, printed in uppercase, for the character. The system maps characters in the range `0x80` to `0x8F`, to `G0` to `GF`, and values in the range `0x90` to `0x9F`, to `H0` to `HF`. The system converts all ordinary 4GL identifier names to lowercase to avoid a naming conflict. By the time this translation occurs, the names of tables and columns in SQL statements are not altered; quotation marks protect the names passed to the database server. The **i4glc4** compiler does one other translation, and that only inside strings. It converts *y-umlaut* (hex `0xFF`) into the escape sequence `\377`, because some C compilers are not fully internationalized and read this character as *end-of-file*.

Using the ESQL/C compiler for phase 3 introduces yet another complication to the compilation process. 4GL uses a different view of the **SQLCA** record from ESQL/C. The warning flags are a series of single characters in ESQL/C, but 4GL code treats them as a string. The ESQL/C compiler automatically includes the **sqlca.h** header ahead of any user-defined code such as the 4GL declaration of the **SQLCA** record. This process would lead to two discrepant definitions of the **SQLCA** record, and the compilations would fail, unless the **C4GL** script handled this. To overcome this problem, **i4glc1** emits a line that starts `#define I4GL_SQLCA` just before the declaration of the SQLCA record. The **i4glc2** and **i4glc3** compilers pass this through. If the .c file to be processed by **i4glc2** contains this definition but does not contain the line `#define SQLCA_INCL`, **C4GL** passes an extra flag to **i4glc4** and adds the line `#define SQLCA_INCL` in front of the C file it translates. The C preprocessor handles this so that the contents of the **sqlca.h** header are ignored, leaving just the 4GL version of the **SQLCA** record visible to the C compiler.

You can use **i4glc4** on its own. It takes the following arguments:

-V prints version information (does not process any files).

-D emits `#define SQLCA_INCL` as the first line of output.

-s *ext* creates backup file with the extension *.ext*.

-o overwrites input files.

By default, **i4glc4** writes the converted file or files to standard output, but you can overwrite the original file using the **-o** option; you can back up the original file with any extension you choose; there is no default file extension. The **i4glc4** compiler automatically inserts a period (.) between the name and the extension. The **-o** and **-s** options are mutually exclusive and require a filename argument. Otherwise, **i4glc4** processes any files specified, or processes standard input if no filenames are provided.

Shared Libraries

Effective with INFORMIX-4GL C Compiler Version 6.0, Informix provides a shared-library implementation of the 4GL program libraries on many platforms. The shared library provides reduced memory consumption, faster program start-up, and substantially reduced program file sizes (thereby saving file system space).

Shared-library support exists for compiled 4GL only. RDS runners (**fglgo** or customized runners) are inherently shared because all active users run the same executable file. This feature is most useful for those installations that have a variety of compiled 4GL applications. The 4GL library code exists in only one place in memory and does not have to be added to each 4GL executable file. On a system with a large number of 4GL programs, the disk space and memory savings can be substantial.

Informix does not provide a shared-library implementation on all platforms. On some platforms, shared libraries are not available and on others the operating system implementation of shared libraries is not compatible with the Informix code stream.

To determine if your platform has a shared-library implementation of 4GL, look at the **C4GL** help messages. You can display these messages by running **c4gl** with no arguments. A help line for the shared option contains one of these messages:

```
-shared       Use dynamic linking and shared libraries
```

or:

```
-shared       (Not available on this platform)
```

If the former message is the one given for your platform, a 4GL shared-library implementation is provided, and the **-shared** option is available for your use.

You can demonstrate the memory and file-size savings for your platform by compiling the 4GL demonstration program (**demo4**) with and without the -shared flag, and comparing the outputs of **ls** and **size** for each of the following programs:

- **i4gldemo**
- **c4gl -shared d4_*.4gl -o demo4.shared**
- **c4gl d4_*.c -o demo4**
- **ls -l demo4***
- **size demo4***

Some platforms provide commands that show the dependencies of a compiled program on the shared libraries. For instance, on current Sun platforms, the command is **ldd**.

For more technical information about shared-library concepts, refer to your operating system documentation. If your system has man pages (on-line manuals), the man page for **ld** might direct you to the appropriate area of your system documentation.

Using the Shared-Library Facility

To compile a 4GL program for shared-library execution, add the **-shared** parameter to your **C4GL** command line:

```
c4gl -shared d4_*.4gl -o demo4.shared
```

You must set the **-shared** parameter explicitly, because the default is **-static**, specifying not to use shared libraries. If you attempt to use the -shared option on a platform for which no shared-library support exists, a warning message is displayed to standard error, and compilation continues with the normal static libraries.

Many platforms require that dynamically linked (shared-library) programs be compiled with position-independent code production from the C compiler. The **c4gl** script automatically takes care of this for you.

Mixing normal and position-independent code can produce errors. When you compile with the **-shared** flag, be sure to recompile all modules from the .4gl source if you had previously compiled any without the **-shared** flag.

Consider the following example:

```
c4gl myprog.4gl myutil1.4gl myutil2.4gl -o myprog
c4gl -shared myprog.o myutil1.o myutil2.o -o myprog.shared
```

Executing this code can produce errors because the objects have not been compiled with the position-independent option. Alternatively, the following code is perfectly acceptable, as the **myutil** objects have been compiled with position-independent code (if applicable to your platform):

```
c4gl -shared myprog.4gl myutil1.4gl myutil2.4gl -o myprog.shared
 <change myprog.4gl>
c4gl -shared myprog.4gl myutil1.o myutil2.o -o myprog.shared
```

Important: *For some platforms, the system linker (**ld**) enforces much stricter name-collision constraints when you use shared libraries. If you have multiple functions in your program with the same name, you might get errors when compiling with shared libraries even if the program links successfully with the static libraries. In such a case, to eliminate the name collision you need to rename one of the functions.*

Technical Details

The name and location of the 4GL shared library varies depending on the version of 4GL you are using, the naming convention for shared libraries on your platform, and the ability of the linker on your platform to locate shared libraries in nonstandard directories. The name of the shared library begins with **lib4gsh** and continues with a three-digit version indicator (for example, 604 for the 6.04 release). The suffix is platform dependent; common values are **.so** and **.a**.

In most cases, the 4GL shared library resides with the other 4GL libraries in the **$INFORMIXDIR/lib/tools** directory. If your platform does not allow shared libraries in nonstandard directories, your system administrator might have to copy the library to a standard system directory such as **/lib** or **/usr/lib**. Look in the machine-specific notes for your platform to see if this is necessary. Most, if not all, platforms require that any programs that change their user ID dynamically while running (often referred to as *setuid programs*) and use shared libraries can only access those shared libraries in standard system directories. Therefore, if you have a **setuid** 4GL program that uses the 4GL shared library, your system administrator must copy or link the 4GL shared library to a standard directory.

Runtime Requirements

Unlike static-linked 4GL programs, 4GL programs that use the shared library must have access to that library at runtime. Most platforms provide an environment variable that instructs the linking program loader of the operating system to add one or more nonstandard directories to its shared-library search list. Common examples of names for this variable are **LD_LIBRARY_PATH**, **LPATH**, or **SHLIB_PATH**. The machine-specific notes provided with 4GL contain the appropriate variable name for your platform. To run your shared-library 4GL applications, you must have this variable set properly in its shell environments. See the following examples.

Shell	Command

Bourne or Korn	`LD_LIBRARY_PATH=$INFORMIXDIR/lib:`
	`  $INFORMIXDIR/lib/esql:$INFORMIXDIR/lib/tools`
	`export LD_LIBRARY_PATH`

C variants	`setenv LD_LIBRARY_PATH ${INFORMIXDIR}/lib/tools`

Releases of 4GL earlier than Version 7.31 required a different setting. Be sure that all potential users set their environments accordingly or update global environment scripts as applicable for their site.

If you develop 4GL applications that are sent out to other systems, the shared library must be available to those systems also. All platforms that have 4GL shared-library support also have the 4GL shared library included in corresponding runtime versions of 4GL. Be sure to notify your remote users and runtime customers of these environment variable needs.

Compiling with c4gl

The simplest case is to compile a single-module 4GL program. The following command produces an executable program called **single.4ge**:

```
c4gl single.4gl -o single.4ge
```

In the next example, the object files **mod1.o**, **mod2.o**, and **mod3.o** are previously compiled 4GL modules, and **mod4.4gl** is a source-code module. Suppose that you wish to compile and link **mod4.4gl** with the three object modules to create an executable program called **myappl.4ge**. To do so, enter the following command line:

```
c4gl mod1.o mod2.o mod3.o mod4.4gl -o myappl.4ge
```

Invoking a Compiled 4GL Program at the Command Line

As noted in the previous section, a valid **c4gl** command line produces a **.4ge** file (or whatever you specify after the **-o** argument) that is an executable command file. To execute your compiled 4GL application program, enter the executable filename at the system prompt.

For example, to run **myappl.4ge** (the program in the previous example), enter the following command:

```
myappl.4ge
```

Some 4GL programs might require additional command-line arguments, such as arguments or filenames, depending on the logic of your application. See the descriptions of the built-in functions "ARG_VAL( )" on page 5-18 and "NUM_ARGS( )" on page 5-99, which can return individual command-line arguments (and the number of command-line arguments) to a calling context within the 4GL application.

No special procedures are needed to create, compile, or execute programs that call C or ESQL/C functions when you use the C Compiler implementation of 4GL. For more information, see Appendix C, "Using C with INFORMIX-4GL."

Program Filename Extensions

Source, executable, error, and backup files generated by 4GL are stored in the current directory and are labeled with a filename extension. The following list shows the file extensions for the source, runable, and error files. These files are produced during the normal course of using the C Compiler.

File	Description
file.**4gl**	4GL source file.
file.**o**	4GL object file.
file.**4ge**	4GL executable (runable) file.
file.**per**	FORM4GL source file.
file.**frm**	FORM4GL object file.
file.**err**	FORM4GL source error file.

The last three files do not exist unless you create or modify a screen form specification file, as described in Chapter 6, "Screen Forms."

Under normal conditions, 4GL also creates certain backup files and intermediate files as necessary and deletes them when a compilation is successful. If something interrupts a compilation, however, you might find one or more of these backup or intermediate files in your current directory. For more information, see "Intermediate Files" on page 1-39.

Warning: *INFORMIX-4GL is not designed to support two or more programmers working concurrently in the same directory. If several developers are working on the same 4GL application, make sure that they do their work in different directories.*

The following table identifies some backup and intermediate files that can be produced when you compile 4GL code to C code from the Programmer's Environment.

File	Description
file.**4bl**	4GL source backup file, created during modification and compilation of **.4gl** program modules
file.**4bo**	Object backup file, created during compilation of **.o** program modules
file.**4be**	Object backup file, created during compilation of **.4ge** program modules
file.**err**	4GL source error file, created when an attempt to compile a module fails. The file contains 4GL source code, as well as any compiler syntax error or warning messages.
file.**ec**	Intermediate source file, created during the normal course of compiling a 4GL module.
file.**c**	Intermediate C file, created during the normal course of compiling a 4GL module.
file.**erc**	4GL object error file, created when an attempt to compile or to link a non-4GL source-code or object module fails. The file contains 4GL source code and annotated compiler errors.
file.**4ec**	Intermediate output of the **i4glc1** preprocessor, containing extended ESQL/C statements as input for the **i4glc2** preprocessor.
file.**pbr**	FORM4GL source backup file
file.**fbm**	FORM4GL object backup file

During the compilation process, 4GL stores a backup copy of the *file*.**4gl** source file in *file*.**4bl**. The time stamp is modified on the (original) *file*.**4gl** source file, but not on the backup *file*.**4bl** file. In the event of a system crash, you might need to replace the modified *file*.**4gl** file with the backup copy contained in the *file*.**4bl** file.

The Programmer's Environment does not allow you to begin modifying a **.4gl** or **.per** source file if the corresponding backup file already exists in the same directory. After an editing session terminates abnormally, for example, you must delete or rename any backup file before you can resume editing your 4GL module or form from the Programmer's Environment.

The Rapid Development System

This section describes the following aspects of the Rapid Development System version of 4GL:

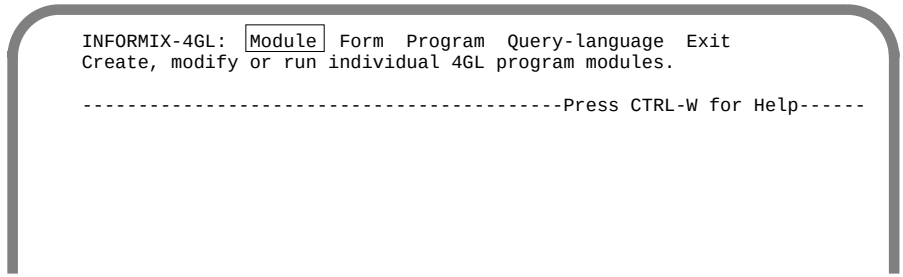
- All the menu options and screen form fields of the RDS Programmer's Environment
- The steps for compiling and executing 4GL programs from the menus of the Programmer's Environment
- The equivalent command-line syntax
- The filename extensions of 4GL source-code, object, error, and backup files

The Programmer's Environment

The Rapid Development System provides a series of menus called the *Programmer's Environment*. These menus support the steps of 4GL program development and keep track of the components of your application. You can invoke the Programmer's Environment by entering `r4gl` at the system prompt.

The INFORMIX-4GL Menu

The **r4gl** command briefly displays the INFORMIX-4GL banner and sign-on message. The **INFORMIX-4GL** menu appears.



This is the highest menu, from which you can reach any other menu of the Programmer's Environment. You have five options:

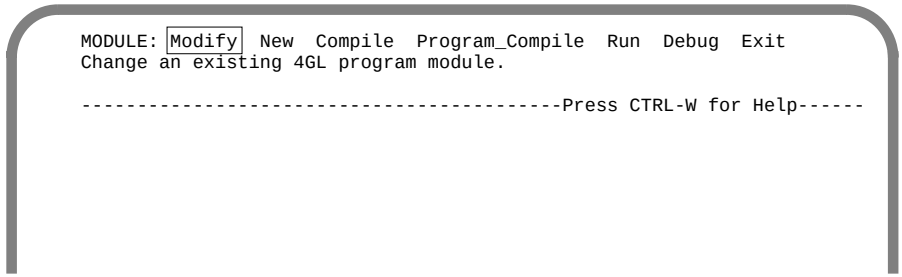
- **Module.** Work on an INFORMIX-4GL program module.
- **Form.** Work on a screen form.
- **Program.** Specify components of a multi-module program.
- **Query-language.** Use an SQL interactive interface, if you have either INFORMIX-SQL or DB-Access installed on your system. (See the documentation of these Informix products for details of their use.)
- **Exit.** Return to the operating system.

The first three options display new menus that are described in the pages that follow. (You can also press CONTROL-W at any menu to display an on-line help message that describes your options.) As at any 4GL menu, you can choose an option in either of two ways:

- By typing the first letter of the option.
- By using the SPACEBAR or arrow keys to move the highlight to the option that you choose, and then pressing RETURN.

The MODULE Design Menu

You can press RETURN or type `m` or `M` to choose the **Module** option of the **INFORMIX-4GL** menu. This option displays a new menu, called the **MODULE** design menu. Use this menu to work on an individual 4GL source-code file.



The **MODULE** design menu supports the following options:

- **Modify.** Change an existing 4GL source-code module.
- **New.** Create a new 4GL source-code module.
- **Compile.** Compile an existing 4GL source-code module.
- **Program_Compile.** Compile a 4GL application program.
- **Run.** Execute a compiled 4GL module or multi-module application program.
- **Debug.** Invoke the INFORMIX-4GL Interactive Debugger to examine an existing 4GL program module or application program (if you have the Debugger product installed on your system).
- **Exit.** Return to the **INFORMIX-4GL** menu.

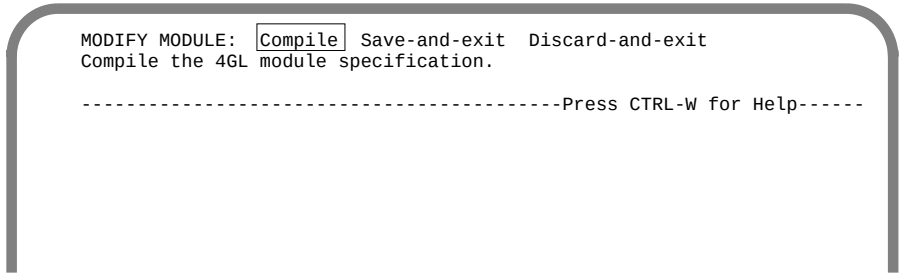
As in all of the menus of the Programmer's Environment except for the **INFORMIX-4GL** menu, the **Exit** option returns control to the higher menu from which you accessed the current menu (or, when you choose **Exit** from the **INFORMIX-4GL** menu, terminates the 4GL session and returns to the system prompt).

You can use these options to create and compile source-code modules of a 4GL application. (For information on creating 4GL screen forms, see "The FORM Design Menu" on page 1-56. For information on creating and compiling programmer-defined help messages for a 4GL application, see the description of the **mkmessage** utility in Appendix B.)

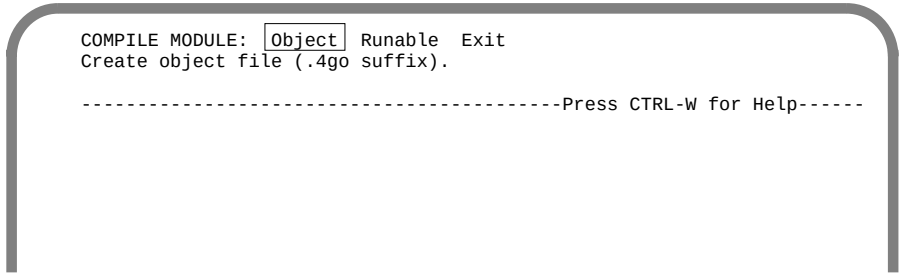
The Modify Option

Choose this option to edit an existing 4GL source-code module. You are prompted for the name of the 4GL source-code file to modify and the text editor to use. If you have designated a default editor with the **DBEDIT** environment variable (see Appendix D, "Environment Variables") or named an editor previously in this session at the Programmer's Environment, 4GL invokes that editor. The **.4gl** source file whose filename you specified is the current file.

When you leave the editor, 4GL displays the **MODIFY MODULE** menu, with the **Compile** option highlighted.



If you press RETURN or type c or C to choose the **Compile** option, 4GL displays the **COMPILE MODULE** menu.



The **Object** option creates a file with a **.4go** extension. The **Runable** option creates a file with a **.4gi** extension. Choose the **Runable** option if the current program module is a stand-alone 4GL program. If this is not the case (that is, if the file is one of several 4GL source-code modules within a multi-module program), you should use the **Object** option instead, and you must use the **PROGRAM** design menu to specify all the component modules.

After you choose **Object** or **Runable**, a message near the bottom of the screen advises you if 4GL issues a compile-time warning or error. If there are warnings (but no errors), a p-code file is produced. Choose the **Exit** option of the next menu, and then **Save-and-exit** at the **MODIFY MODULE** menu, if you prefer to save the p-code file without reading the warnings.

Alternatively, you can examine the warning messages by choosing **Correct** at the next menu. When you finish editing the **.err** file that contains the warnings, you must choose **Compile** again from the **MODIFY MODULE** menu, because the **Correct** option deletes the p-code file.

If there are compilation errors, the following menu appears.

```
COMPILE MODULE:  Correct  Exit
Correct errors in the 4GL module.

-----------------------------------------Press CTRL-W for Help------
```

If you choose to correct the errors, an editing session begins on a copy of your source module with embedded error messages. (You do not need to delete error messages, because 4GL does this for you.) Correct your source file, save your changes, and exit from the editor. The **MODIFY MODULE** menu reappears, prompting you to recompile, save, or discard your changes without compiling.

If there are no compilation errors, the **MODIFY MODULE** menu appears with the **Save-and-Exit** option highlighted. If you choose this option, 4GL saves the current source-code module as a disk file with the filename extension **.4gl**, and saves the compiled version as a file with the same filename, but with the extension **.4go** or **.4gi**. If you choose the **Discard-and-Exit** option, 4GL discards any changes that were made to your file after you chose the **Modify** option.

The New Option

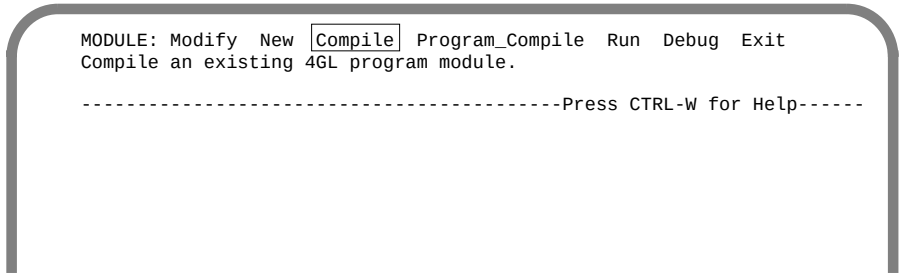
Choose this option to create a new 4GL source-code module.

```
MODULE: Modify  New  Compile  Program_Compile  Run  Debug  Exit
Create a new 4GL program module.

-----------------------------------------Press CTRL-W for Help------
```

The **New** option resembles the **Modify** option, but **NEW MODULE** is the menu title, and you must enter a new module name, rather than choose it from a list. If you have not designated an editor previously in this session or with **DBEDIT**, you are prompted for the name of an editor. Then an editing session begins.

The Compile Option

The **Compile** option enables you to compile an individual 4GL source-code module without first choosing the **Modify** option.



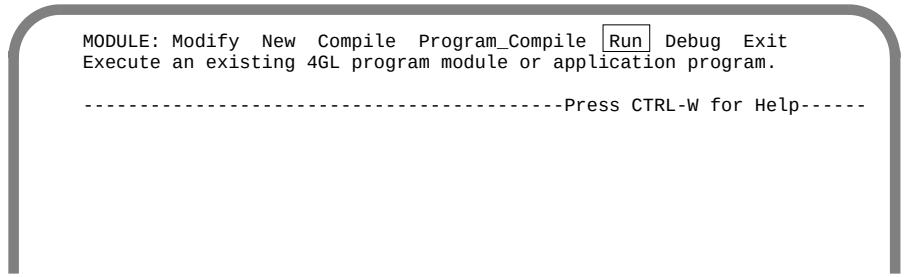
After you specify the name of a 4GL source-code module to compile, the screen displays the **COMPILE MODULE** menu. For information on the **COMPILE MODULE** menu options, see "The Modify Option" on page 1-51.

The Program_Compile Option

The **Program_Compile** option of the **MODULE** design menu is the same as the **Compile** option of the **PROGRAM** design menu (see "The Compile Option" on page 1-66). The **Program_Compile** option enables you to compile and combine modules as described in the program specification database, taking into account the time when the modules were last updated. This option is useful when you have just modified a single module of a complex program and wish to test it by compiling it with the other modules.

The Run Option

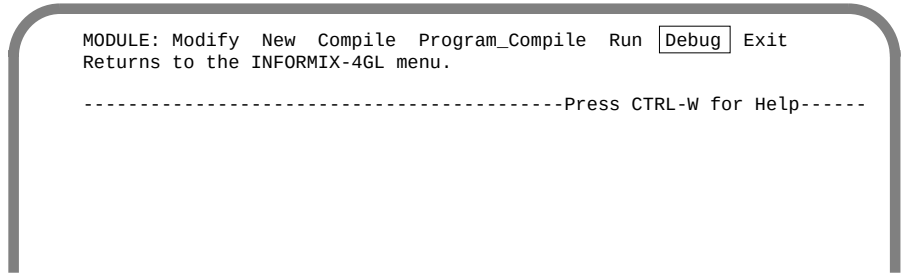
Choose this option to begin execution of a compiled program.



The RUN PROGRAM screen presents a list of compiled modules and programs, with the highlight on the module corresponding to the current file, if any has been specified. Compiled programs must have the extension **.4gi** to be included in the list. If you compile a module with the extension **.4go**, you can run it by typing the filename and extension at the prompt. If no compiled programs exist, 4GL displays an error message and restores the **MODULE** design menu.

The Debug Option

Choose this option to use the Debugger to analyze a program. This option is implemented only if you have separately purchased and installed the Debugger on your system.



If you have the Debugger product, refer to the *Guide to the INFORMIX-4GL Interactive Debugger* for more information about this option.

The Exit Option

Choose this option to exit from the **MODULE** design menu and display the **INFORMIX-4GL** menu.

```
MODULE: Modify  New  Compile  Program_Compile  Run  Debug [Exit]
Returns to the INFORMIX-4GL menu.

------------------------------------------Press CTRL-W for Help------
```

The FORM Design Menu

You can type `f` or `F` at the **INFORMIX-4GL** menu to choose the **Form** option. This option replaces the **INFORMIX-4GL** menu with a new menu, called the **FORM** design menu.

```
FORM:  [Modify]  Generate  New  Compile  Exit
Change an existing form specification.

------------------------------------------Press CTRL-W for Help------
```

You can use this menu to create, modify, and compile *screen form* specifications. These specifications define visual displays that 4GL applications can use to query and modify the information in a database. 4GL screen form specifications are ASCII files that are described in Chapter 6.

The **FORM** design menu supports the following options:

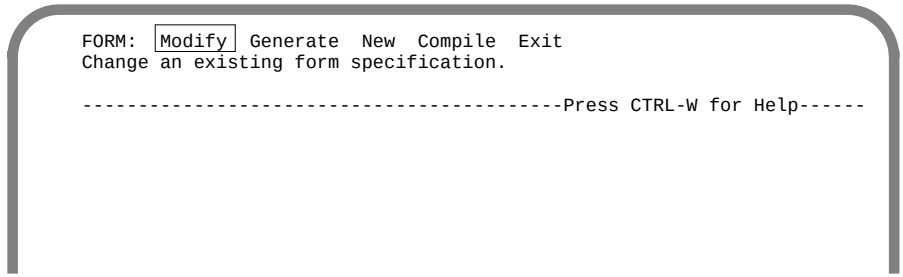
- **Modify.** Change an existing 4GL screen form specification.
- **Generate.** Create a default 4GL screen form specification.
- **New.** Create a new 4GL screen form specification.
- **Compile.** Compile an existing 4GL screen form specification.
- **Exit.** Return to the **INFORMIX-4GL** menu.

If you are familiar with the menu system of INFORMIX-SQL, you might notice that this menu resembles the menu displayed by the **Form** option of the INFORMIX-SQL main menu.

For descriptions of the usage and statement syntax of 4GL screen form specifications, see Chapter 6.

The Modify Option

The **Modify** option of the **FORM** design menu enables you to edit an existing form specification file. It resembles the **Modify** option in the **MODULE** design menu because both options are used to edit program modules.



If you choose this option, you are prompted to choose the name of a form specification file to modify. Source files created at the **FORM** design menu (or at the command line by the **form4gl** screen form facility) have the file extension **.per**.

If you have not already designated a text editor in this 4GL session or with **DBEDIT**, you are prompted for the name of an editor. Then an editing session begins, with the form specification source-code file that you specified as the current file. When you leave the editor, 4GL displays the **MODIFY FORM** menu with the **Compile** option highlighted.

```
MODIFY FORM:  Compile  Save-and-exit  Discard-and-exit
Compile the form specification.

-----------------------------------------Press CTRL-W for Help------
```

Now you can press RETURN to compile the revised form specification file. If the compiler finds errors, the **COMPILE FORM** menu appears.

```
COMPILE FORM:  Correct  Exit
Correct errors in the form specification.

-----------------------------------------Press CTRL-W for Help------
```

Press RETURN to choose **Correct** as your option. An editing session begins on a copy of the current form, with diagnostic error messages embedded where the compiler detected errors. 4GL deletes these messages when you save the edited file and exit from the editor. After you correct the errors, the **MODIFY FORM** menu appears again, with the **Compile** option highlighted. Press RETURN to recompile.

If there are no compilation errors, you are prompted to either save the modified form specification file and the compiled form, or discard the changes. (Discarding the changes restores the version of your form specifications from immediately before you chose the **Modify** option.)

The Generate Option

You can type g or G to choose the **Generate** option. This option creates a simple *default* screen form for use directly in your 4GL program, or for you to edit later by choosing the **Modify** option.

```
FORM:  Modify [Generate] New  Compile  Exit
Generate and compile a default form specification.

-----------------------------------------Press CTRL-W for Help------
```

When you choose this option, 4GL prompts you to choose a database, to choose a filename for the form specification, and to identify the tables that the form will access. After you provide this information, 4GL creates and compiles a form specification file. This process is equivalent to running the **-d** (default) option of the **form4gl** command, as described in "Compiling a Form at the Command Line" on page 6-87.

The New Option

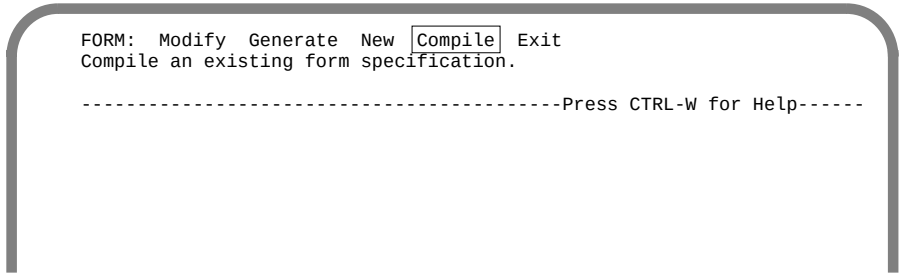
The **New** option of the **FORM** design menu enables you to create a new screen form specification.

```
FORM:  Modify  Generate [New]  Compile  Exit
Create a new form specification.

-----------------------------------------Press CTRL-W for Help------
```

After prompting you for the name of your form specification file, 4GL places you in the editor where you can create a form specification file. When you leave the editor, 4GL transfers you to the **NEW FORM** menu that is like the **MODIFY FORM** menu. You can compile your form and correct it in the same way.

The Compile Option

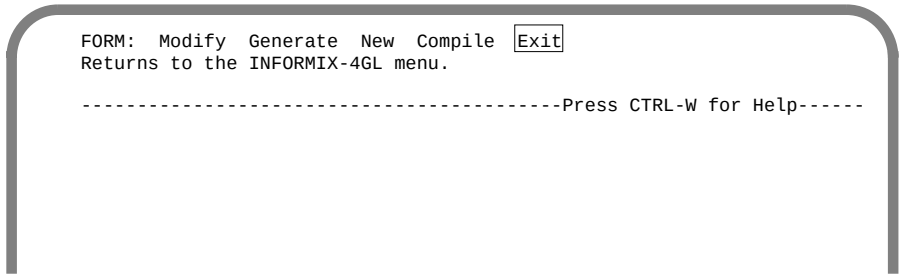
The **Compile** option enables you to compile an existing form specification file without going through the **Modify** option.



4GL prompts you for the name of the form specification file and then performs the compilation. If the compilation is not successful, 4GL displays the **COMPILE FORM** menu with the **Correct** option highlighted.

The Exit Option

The **Exit** option clears the **FORM** design menu from the screen.



Choosing this option restores the **INFORMIX-4GL** menu.

```
INFORMIX-4GL:  Module  Form  Program  Query-language  Exit
Create, modify or run individual 4GL program modules.

-----------------------------------------Press CTRL-W for Help------
```

The PROGRAM Design Menu

A 4GL program can be a single source-code module that you create and compile at the **MODULE** design menu. For applications of greater complexity, however, it is often easier to develop and maintain a 4GL program that includes several modules. The **INFORMIX-4GL** menu includes the **Program** option so that you can create multiple-module programs. When you choose this option, 4GL searches your **DBPATH** directories (as described in Appendix D) for the program design database, called **syspgm4gl**. This database stores the names of objects that are used to create 4GL programs, and their build dependencies.

If 4GL cannot find this database, you are asked if you want one created. If you enter y in response, 4GL creates the **syspgm4gl** database, grants CONNECT privileges to PUBLIC, and displays the **PROGRAM** design menu. As Database Administrator of **syspgm4gl**, you can later restrict the access of other users.

If **syspgm4gl** already exists, the **PROGRAM** design menu appears.

```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Debug  Undefine
 Exit
Change the compilation definition of a 4GL application program.

-----------------------------------------Press CTRL-W for Help------
```

You can use this menu to create or modify a multi-module 4GL program specification, or to compile, execute, or analyze a program.

The **PROGRAM** design menu supports the following eight options:

- **Modify.** Change an existing program specification.
- **New.** Create a new program specification.
- **Compile.** Compile an existing program.
- **Planned_Compile.** Display the steps to compile an existing program.
- **Run.** Execute an existing program.
- **Debug.** Invoke the Debugger.
- **Undefine.** Delete an existing program specification.
- **Exit.** Return to the **INFORMIX-4GL** menu.

You must first use the **MODULE** design menu and **FORM** design menu to enter and edit the 4GL statements within the component source-code modules of a 4GL program. Then you can use the **PROGRAM** design menu to identify which modules are part of the same application program, and to combine all the 4GL modules in an executable program.

The Modify Option

The **Modify** option enables you to modify the specification of an existing 4GL program. (This option is not valid unless at least one program has already been specified. If none has, you can create a program specification by choosing the **New** option from the same menu.) 4GL prompts you for the name of the program specification that you wish to modify. It then displays a screen and menu that you can use to update the information in the program specification database, as shown in Figure 1-4.

```
MODIFY PROGRAM: 4GL  Globals  Other  Program_Runner  Rename  Exit
Edit the 4GL sources list.

-----------------------------------------Press CTRL-W for Help------
 Program  [myprog  ]
 Runner   [fglgo   ]   Runner Path   [           ]
 Debugger [fgldb   ]   Debugger Path [           ]

 4gl Source     4gl Source Path
 [main      ]   [/u/john/appl/4GL                      ]
 [funct     ]   [/u/john/appl/4GL                      ]
 [rept      ]   [/u/john/appl/4GL                      ]
 [          ]   [                                      ]
 [          ]   [                                      ]

 Global Source  Global Source Path
 [          ]   [                                    ]
 [          ]   [                                    ]

 Other .4go     Other .4go Path
 [obj       ]   [                                    ]
 [          ]   [                                    ]
```

Figure 1-4
Example of a Program Specification Entry

The name of the program appears in the **Program** field. In Figure 1-4 this name is **myprog**. You can change the name by choosing the **Rename** option. The program name, with extension **.4gi**, is assigned to the program produced by compiling and combining all the source files. (Compiling and combining occurs when you choose the **Compile** option, as described in "The Compile Option" on page 1-66, or the **Program_Compile** option of the **MODULE** design menu.) In this case, the runable program would have the name **myprog.4gi**.

The **4GL** option enables you to update the entries for the **4gl Source** and **4gl Source Path** fields. The five rows of fields under these labels form a screen array. If you choose the **4GL** option, 4GL executes an INPUT ARRAY statement so that you can move through the array and scroll for up to a maximum of 100 entries.

The INPUT ARRAY statement description in Chapter 4 explains how to use function keys to scroll, delete rows, and insert new rows. (You cannot redefine function keys, however, as you can with a 4GL program.)

In the example shown in Figure 1-4, the 4GL source program has been broken into three modules:

- One module contains the main program (**main.4gl**).
- One module contains functions (**funct.4gl**).
- One module contains REPORT statements (**rept.4gl**).

These modules are all located in the directory **/u/john/appl/4GL**. If a module contains only global variables, you can list it here or in the Global Source array.

The **Globals** option enables you to update the Global Source array. If you use the Global Source array to store a globals module, any modification of the globals module file causes all 4GL modules to be recompiled when you choose the **Compile** option.

The **Other** option enables you to update the entries for the **Other .4go** and **Other .4go Path** fields. This is where you specify the name and location of other 4GL object files (**.4go** files) to include in your program. Do not specify the filename extensions. You can list up to 100 files in this array.

The **Program_Runner** option enables you to specify the name and location of the p-code runner to execute your program. You can run 4GL programs with **fglgo** (the default) or with a *customized p-code runner*. A customized p-code runner is an executable program that you create to run 4GL programs that call C functions. (See "RDS Programs That Call C Functions" on page 1-83.) If you do not modify the **Runner** field, your program is executed by **fglgo** when you choose the **Run** option from the **PROGRAM** design menu.

The MODIFY PROGRAM screen form contains two additional fields labeled **Debugger** and **Debugger Path**. If you have the Debugger, you can also use the **Program_Runner** option to enter the name of a customized debugger. See "RDS Programs That Call C Functions" on page 1-83 for information about the use of a customized debugger. For the procedures to create a customized debugger, refer to Appendix C of the *Guide to the INFORMIX-4GL Interactive Debugger*, which includes an example.

The **Exit** option of the **MODIFY PROGRAM** menu returns you to the **PROGRAM** design menu.

The New Option

The **New** option of the **PROGRAM** design menu enables you to create a new specification of the program modules and libraries that make up the desired application program.

```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Debug  Undefine  Exit
Add the compilation definition of a 4GL application program.

------------------------------------------------Press CTRL-W for Help------
```

The filename of the module must be unique among source-code modules of the same 4GL program, and can include up to ten characters, not including the **.4gl** file extension. The **New** option is identical to the **Modify** option, except that you must first supply a name for your program. 4GL then displays a blank form with a **NEW PROGRAM** menu that has the same options as the **MODIFY PROGRAM** menu.

The Compile Option

The **Compile** option compiles and combines the modules listed in the program specification database, taking into account the time when files were last updated. 4GL compiles only those files that have been modified after they were last compiled, except in the case where you have modified a module listed in the Global Source array. In that case, all files are recompiled.

```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Debug  Undefine  Exit
Compile a 4GL application program.

-----------------------------------------Press CTRL-W for Help------
```

The **Compile** option produces a runable p-code file with a **.4gi** extension. 4GL lists each step of the compilation as it occurs.

The Planned_Compile Option

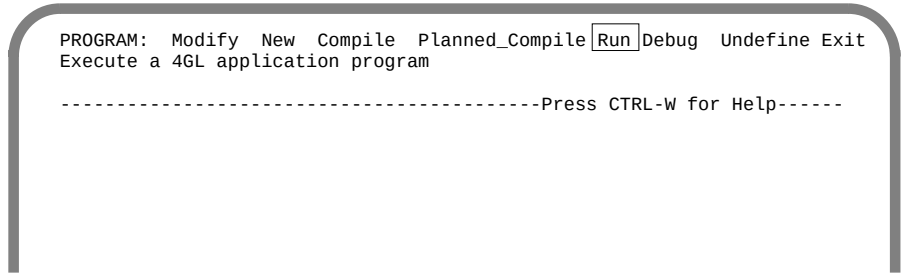
Taking into account the time of last change for the various files in the dependency relationships, the **Planned_Compile** option prompts for a program name and displays a summary of the steps that will be executed if you choose **Compile**. No compilation actually takes place.

```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Debug  Undefine  Exit
Show the planned compile actions of a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
Compiling INFORMIX-4GL sources:
                /u/john/appl/4GL/main.4gl
                /u/john/appl/4GL/funct.4gl
                /u/john/appl/4GL/rept.4gl
Linking other objects:
                /u/john/appl/Com/obj.4go
```

If you have made changes in all the components of the program listed in Figure 1-4 on page 1-63 since the last time that they were compiled, 4GL displays the previous screen.

The Run Option

Choose the **Run** option to execute a compiled program.

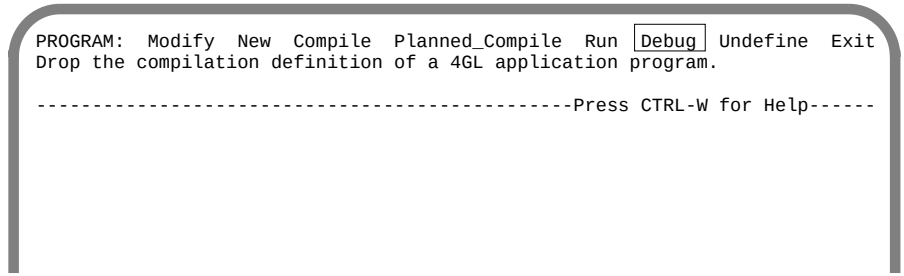


The screen lists any compiled programs (files with the extension **.4gi**) and highlights the current program, if one has been specified. This option resembles the **Run** option of the **MODULE** design menu.

Although **.4go** files are not displayed, you can also enter the name and extension of a **.4go** file. Whatever compiled program you choose is executed by **fglgo** or by the runner that you specified in the **Runner** field of the Program Specification screen. This screen was illustrated earlier, in the description of the **MODIFY PROGRAM** menu.

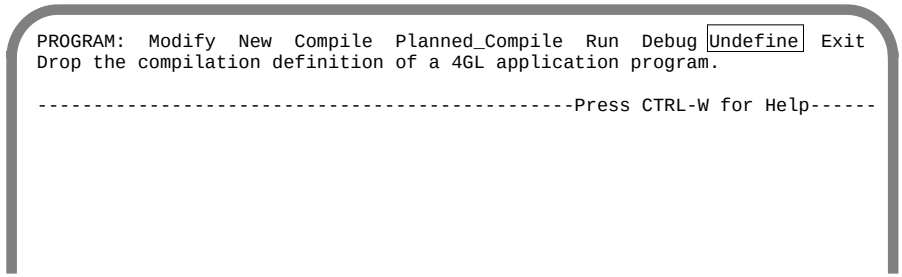
The Debug Option

The **Debug** option works like the **Run** option but enables you to examine a 4GL program with the Debugger. This option is not implemented unless you have purchased the Debugger.



The Undefine Option

The **Undefine** option of the **PROGRAM** design menu prompts you for a program name and removes the compilation definition of that program from the **syspgm4gl** database. This action removes the definition only. Your program and 4GL modules are *not* removed.



The Exit Option

The **Exit** option clears the **PROGRAM** design menu from the screen and restores the **INFORMIX-4GL** menu.

The QUERY LANGUAGE Menu

The SQL interactive interface is identical to the interactive SQL interface of INFORMIX-SQL, if you have separately purchased and installed the INFORMIX-SQL product on your system. If you have not, this option invokes the DB-Access utility, which is provided with some Informix databases, if 4GL can locate the executable DB-Access file.

The **Query-language** option is placed at the top-level menu so that you can test SQL statements without leaving the Programmer's Environment. You can also use this option to create, execute, and save SQL scripts.

Creating Programs in the Programmer's Environment

Enter the following command at the system prompt to invoke the Programmer's Environment:

```
r4gl
```

After a sign-on message, the **INFORMIX-4GL** menu appears.

Creating a 4GL application with the Rapid Development System requires the following steps:

1. Creating a new source module or revising an existing source module
2. Compiling the source module
3. Linking the program modules
4. Executing the compiled program

This process is described in the sections that follow.

Creating a New Source Module

This section outlines the procedure for creating a new module. If your source module already exists but needs to be modified, skip ahead to the next section, "Revising an Existing Module."

To create a source module

1. Choose the **Module** option of the **INFORMIX-4GL** menu by pressing M or by pressing RETURN.

   The **MODULE** design menu is displayed.
2. If you are creating a new **.4gl** source module, press N to choose the **New** option of the **MODULE** design menu.
3. Enter a name for the new module.

   The name must begin with a letter, and can include letters, numbers, and underscores. The name must be unique among the files in the same directory, and among the other program modules, if it will be part of a multi-module program. 4GL attaches the extension **.4gl** to this identifier, as the filename of your new source module.
4. Press RETURN.

Revising an Existing Module

If you are revising an existing 4GL source file, use the following procedure.

To modify a source file

1. Choose the **Modify** option of the **MODULE** design menu.

   The screen lists the names of all the **.4gl** source modules in the current directory and prompts you to choose a source file to edit.

2. Use the arrow keys to highlight the name of a source module and press RETURN, or enter a filename (with no extension).

   If you specified a default editor with the **DBEDIT** environment variable, an editing session begins automatically. Otherwise, the screen prompts you to specify a text editor.

   Specify the name of a text editor, or press RETURN for **vi**, the default editor. Now you can begin an editing session by entering 4GL statements. (Chapters that follow describe 4GL statements and expressions, as well as built-in functions and operators.)

3. When you have finished entering or editing your 4GL code, use an appropriate editor command to save your source file and end the text editing session.

Compiling a Source Module

The **.4gl** source file module that you create or modify is an ASCII file that must be compiled before it can be executed.

To compile a module

1. Choose the **Compile** option from the **MODULE** design menu.
2. Select the type of module that you are compiling, either **Object** or **Runable**.

   If the module is a complete 4GL program that requires no other modules, choose **Runable**. This option creates a compiled p-code version of your program module, with the same filename, but with the extension **.4gi**.

   If the module is one module of a multi-module 4GL program, choose **Object**. This creates a compiled p-code version of your program module, with the same filename, but with the extension **.4go**. For more information, see "Combining Program Modules" on page 1-72.
3. If the compiler detects errors, no compiled file is created, and you are prompted to fix the problem.

   Choose **Correct** to resume the previous text editing session, with the same 4GL source code, but with error messages in the file. Edit the file to correct the error, and choose **Compile** again. If an error message appears, repeat this process until the module compiles without error.
4. After the module compiles successfully, choose **Save-and-exit** from the menu to save the compiled program.

   The **MODULE** design menu appears again on your screen.
5. If your program requires screen forms, choose **Form** from the **INFORMIX-4GL** menu to display the **FORM** design menu. For information about designing and creating screen forms, see Chapter 6.
6. If your program displays help messages, you must create and compile a help file.

   Use the **mkmessage** utility to compile the file. For more information about this utility, see Appendix B.

Important: *This version of the runner or Debugger cannot interpret programs compiled to p-code by releases of 4GL earlier than Version 7.30. You must first recompile your source files and form specifications. Similarly, releases of the 4GL runner or Debugger earlier than Version 7.30 cannot interpret p-code that this release produces.*

Combining Program Modules

If your new or modified module is part of a multi-module 4GL program, you must combine all of the modules into a single program file before you can run the program. If the module that you compiled is the only module in your program, you are now ready to run your program. (For more information, see "Executing a Compiled RDS Program" on page 1-74.)

To combine modules

1. Choose the **Program** option from the **INFORMIX-4GL** menu.

   The **PROGRAM** design menu appears.

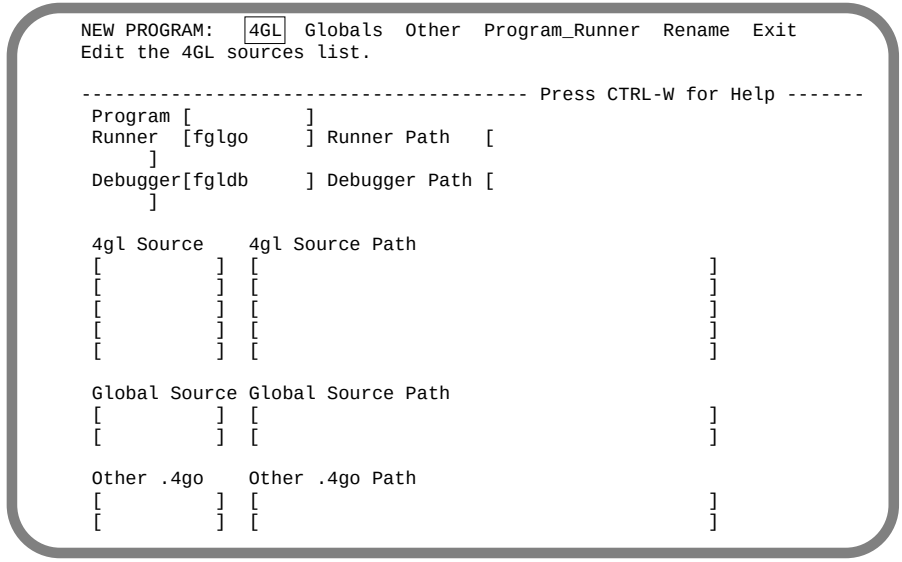
2. If you are creating a new multi-module 4GL program, choose the **New** option; if you are modifying an existing one, choose **Modify**.

   In either case, the screen prompts you for the name of a program.

3. Enter the name (without a file extension) of the program that you are modifying, or the name to be assigned to a new program.

   Names must begin with a letter, and can include letters, underscores (_) symbols, and numbers. After you enter a valid name, the PROGRAM screen appears, with your program name in the first field.

   If you chose **Modify**, the names and pathnames of the source-code modules are also displayed. The PROGRAM screen appears below the **MODIFY PROGRAM** menu, rather than below the **NEW PROGRAM** menu. (Both menus list the same options.)



4. Identify the files that make up your program:
   - To specify new 4GL modules or edit the list of 4GL modules, choose the **4GL** option.

     You can enter or edit the name of a module under the heading **4GL Source**; the **.4gl** file extension is optional. Repeat this step for every module. If the module is not in the current directory or in a directory specified by the **DBPATH** environment variable, enter the pathname to the directory where the module resides.

     The name of the runner (and of the Debugger, if you have the Debugger) are usually as illustrated in the PROGRAM screen, unless your 4GL program calls C functions. For information on calling C functions, see "RDS Programs That Call C Functions" on page 1-83.
   - To enter or edit the name or pathname of a **Globals** module, choose the **Globals** option and provide the corresponding information.
   - To enter or edit the file or pathname of any **.4go** modules that you have already compiled, choose the **Other** option.
5. After you correctly list all of the modules of your 4GL program, choose the **Exit** option to return to the **PROGRAM** design menu.
6. Choose the **Compile** option of the **PROGRAM** design menu.

   This option produces a file that combines all of your **.4gl** source files into an executable program. Its filename is the program name that you specified, with extension **.4gi**. The screen lists the names of your **.4gl** source modules and displays the **PROGRAM** design menu with the **Run** option highlighted.

Executing a Compiled RDS Program

After compiling your program modules, you can execute your program. To do so, choose the **Run** option from the **MODULE** design menu. This option executes the compiled 4GL program.

Menus, screen forms, windows, or other screen output are displayed, according to your program logic and the keyboard interaction of the user with the program.

Invoking the Debugger

If you are developing or modifying a 4GL program, you have much greater control over program execution by first invoking the Debugger. If you have purchased the Debugger, you can invoke it from the **MODULE** design menu or **PROGRAM** design menu of the Programmer's Environment by choosing the **Debug** option. For information on using the Debugger, see the *Guide to the INFORMIX-4GL Interactive Debugger*.

Creating Programs at the Command Line

You can also create **.4gl** source files and compiled **.4go** and **.4gi** p-code files at the operating system prompt. Figure 1-5 shows the commands for creating, compiling, and running or debugging a single-module program.

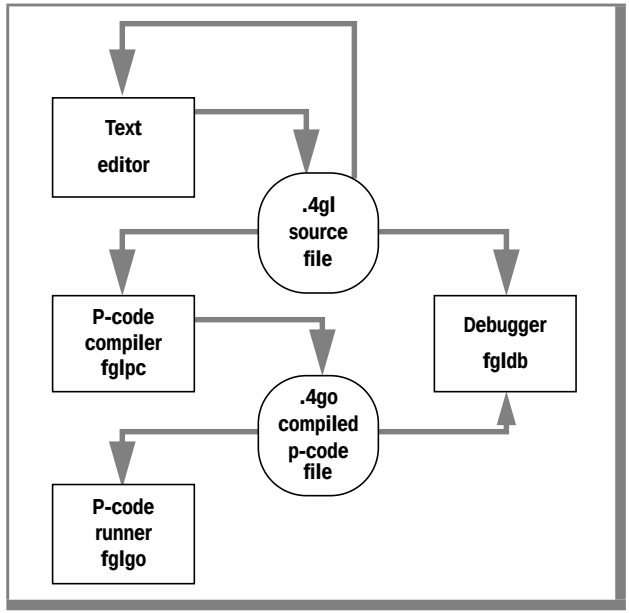


Figure 1-5
Creating and Running a Single-Module Program

In Figure 1-5, the rectangles represent processes controlled by specific commands, and the circles represent files. Arrows indicate whether a file serves as input or output for a process.

(For the sequence of operating system commands to create multi-module 4GL programs, see Figure 1-6 on page 1-79.)

This diagram is simplified and ignores the similar processes by which forms, help messages, and any other components of 4GL applications are compiled and executed. The diagram outlines the following process:

- The cycle begins in the upper-left corner with a text editor, such as **vi**, to produce a 4GL source module.
- The program module can then be compiled, using the **fglpc** p-code compiler. (If error messages are produced by the compiler, find them in the **.err** file, and edit the **.4gl** file to correct the errors. Then recompile the corrected **.4gl** file.)
- The following command line invokes the p-code runner:

  ```
  fglgo filename
  ```

  where *filename* specifies a compiled 4GL file to be executed.

Executing a program that is undergoing development or modification sometimes reveals the existence of runtime errors. If you have licensed the Debugger, you can invoke it to analyze and identify runtime errors in your program by entering the command:

```
fgldb filename
```

where filename specifies your compiled 4GL file. You can then recompile and retest the program. When it is ready for use by others, they can use the **fglgo** runner to execute the compiled program.

A correspondence between commands and menu options of the RDS Programmer's Environment is summarized by the following list.

Command	Invokes	Menu Option
vi	UNIX System Editor	**Module New / Modify**
fglpc	4GL P-Code Compiler	**Compile**
fglgo	4GL P-Code Runner	**Run**
fgldb	4GL Interactive Debugger	**Debug**

Subsequent sections of this chapter describe how to use the Rapid Development System to compile and execute 4GL programs that call C functions. (This requires a C language compiler and linker, which are unnecessary for 4GL applications in p-code that do not call programmer-defined C functions.)

Creating or Modifying a 4GL Source File

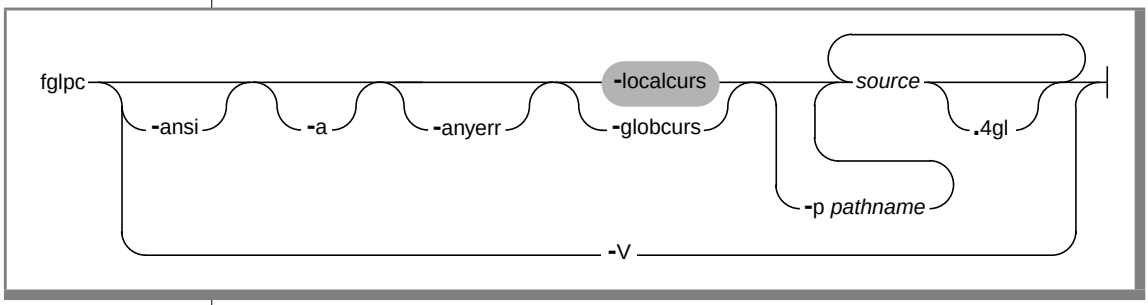
Use your system editor or another text-editing program to create a **.4gl** source file, or to modify an existing file. For information on the statements that you can include in a 4GL program, see Chapter 4.

Compiling an RDS Source File

You cannot execute a 4GL program until you compile each source module into a **.4go** file. Do this at the system prompt by entering the **fglpc** command, which compiles your 4GL source code and generates a file containing tables of information and blocks of p-code. You can then run this compiled code by using the 4GL p-code runner (or the Debugger if you have it). The 4GL source-code module to be compiled should have the file extension **.4gl**.

fglpc Command

The **fglpc** command supports the following syntax.



Element	Description
pathname	is the pathname of a directory to hold object and error files.
source	is the name of a 4GL source module. The **.4gl** extension is optional.

The **fglpc** command reads *source*.**4gl** files and creates a compiled version of each, with the filename *source*.**4go**. You can specify any number of source files, in any order, with or without their **.4gl** filename extensions.

To instruct the compiler to check all SQL statements for compliance with the ANSI/ISO standard for SQL, use the **-ansi** option. If you specify the **-ansi** option, it must appear first among **fglpc** command arguments. Including the **-ansi** option asks for compile-time and runtime warning messages if your source code includes Informix extensions to the ANSI/ISO standard for SQL.

If an error or warning occurs during compilation, 4GL creates a file called *source*.**err**. Look in *source*.**err** to find where the error or warning occurred in your code.

If you specify the **-anyerr** option, 4GL sets the **status** variable after evaluating expressions (in addition to setting it after each SQL statement executes, and after errors in 4GL screen I/O or validation statements. The **-anyerr** option overrides any WHENEVER ERROR directives in your program.

You can use the **-p** *pathname* option to specify a nondefault directory for the object (**.4go**) and error (**.err**) files. Otherwise, any files produced by **fglpc** are stored in your current working directory.

To have your compiled program check array bounds at runtime, specify the **-a** option. The **-a** option requires additional processing, so you might prefer to use this option only for debugging during development.

If you typically compile with the same options, you can set the **FGLPCFLAGS** environment variable to supply those options implicitly. See the section "FGLPCFLAGS" on page D-43 for details of this feature.

The **-globcurs** option lets you make the names of cursors and of prepared objects global to the entire program. The compilers still require you to declare the cursor before using it for any other purpose in the module, so this option is seldom useful. This option might help in debugging, however, because the cursor names are not modified. See the section "The -globcurs and -localcurs Options" on page 1-40 for more information about the scope of cursor names.

The **-localcurs** option can override the **-globcurs** option if that was set in the **C4GLFLAGS** environment variable, and makes the names of cursors and prepared objects local to the module in which they were declared.

To display the version number of the software, specify the **-V** option. The version number of your SQL and p-code compiler software appears on the screen. Any other command options are ignored. After displaying this information, the program terminates without compiling.

Examples

The following command compiles a 4GL source file **single.4gl**, and creates a file called **single.4go** in the current directory:

```
fglpc single.4gl
```

The next command line compiles two 4GL source files:

```
fglpc -p /u/ken fileone filetwo
```

This command generates two compiled files, **fileone.4go** and **filetwo.4go**, and stores them in subdirectory **/u/ken**. Any compiler error messages are saved in file **fileone.err** or **filetwo.err** in the same directory.

Concatenating Multi-Module Programs

If a program has several modules, the compiled modules must all be concatenated into a single file, as represented in Figure 1-6.

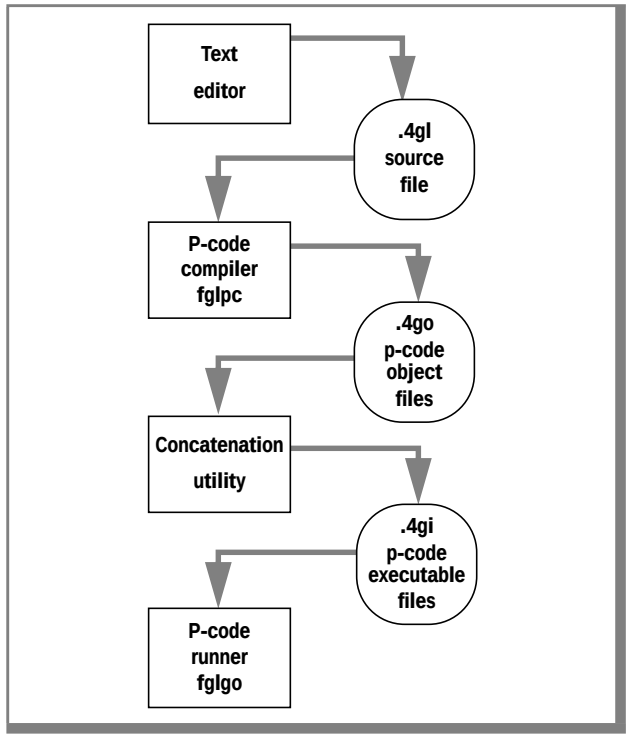


Figure 1-6
Creating and Running a Multi-Module Program

The UNIX **cat** command combines the listed files into the file specified after the redirect (>) symbol. For example, the following command combines a list of **.4go** files into a new file called **new.4gi**:

```
cat file1.4go file2.4go ... fileN.4go new.4gi
```

The new filename of the combined file must have either a **.4go** or a **.4gi** extension. The extension **.4gi** designates runable files that have been compiled (and concatenated, if several source modules make up the program). You might wish to follow this convention in naming files, because only **.4gi** files are displayed from within the Programmer's Environment. This convention is also a convenient way to distinguish complete program files from object files that are individual modules of a multi-module program.

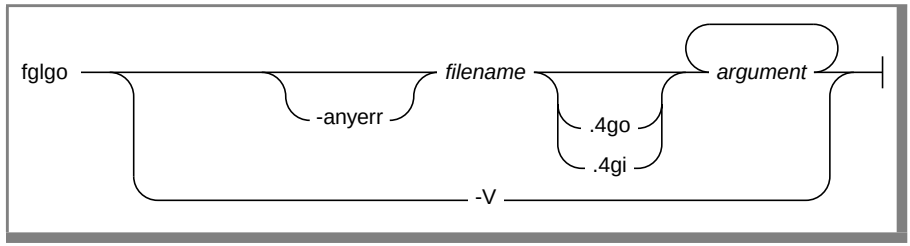
If your 4GL program calls C functions or ESQL/C functions, you must follow the procedures described in "RDS Programs That Call C Functions" on page 1-83 before you can run your application.

Running RDS Programs

To execute a compiled 4GL program from the command line, you can invoke the p-code runner, **fglgo**.

fglgo Command

The **fglgo** command supports the following syntax.



Element	Description
filename	is the name of a compiled 4GL file. The *filename* must have a **.4go** or **.4gi** extension. You do not need to enter this extension.
argument	are any arguments required by your 4GL program.

If you do not specify a filename extension, **fglgo** looks first for the *filename* with a **.4gi** extension, and then for the *filename* with a **.4go** extension.

Unlike **fglpc**, the **fglgo** command needs no **-a** option to check array bounds at runtime, because array bounds are always checked.

If you specify the **-anyerr** option, 4GL sets the **status** variable after evaluating expressions. This option overrides any WHENEVER ERROR statements.

To display the version number of the software, specify the **-V** option. The version number of your SQL and p-code software appears on the screen. Any other command options are ignored. After displaying this information, the program terminates without invoking the p-code runner. 4GL runners earlier than Version 7.3 cannot run programs that use 7.3 or later compilers, and you must recompile programs compiled with earlier versions of the 4GL compiler before a 7.3 or later runner can interpret them.

Important: *To run a 4GL program that calls programmer-defined C functions, you cannot use* ***fglgo****. You must instead use a customized p-code runner.* *"RDS Programs That Call C Functions" on page 1-83* *describes how to create a customized runner.*

Examples

To run a compiled program named **myprog.4go**, enter the following command line at the operating system prompt:

```
fglgo myprog
```

or:

```
fglgo myprog.4go
```

Running Multi-Module Programs

To run a program with multiple modules, you must compile each module and then combine them by using an operating system concatenation utility, as described in an earlier section. For example, if **mod1.4go**, **mod2.4go**, and **mod3.4go** are compiled 4GL modules that you wish to run as one program, you must first combine them as in the following example:

```
cat mod1.4go mod2.4go mod3.4go > mods.4gi
```

You can then run the **mods.4gi** program by using the command line:

```
fglgo mods
```

or:

```
fglgo mods.4gi
```

Running Programs with the Interactive Debugger

You can also run compiled 4GL programs with the Debugger. This 4GL source-code debugger is a p-code runner with a rich command set for analyzing 4GL programs. You can use the Debugger to locate logical and runtime errors in your 4GL programs and to become more familiar with 4GL programs. The Debugger must be purchased separately.

If you have the Debugger, you can invoke it at the system prompt with a command line of the form:

```
fgldb filename
```

where filename is any runable 4GL file that you produced by an **fglpc** command.

For the complete syntax of the **fgldb** command, see the *Guide to the INFORMIX-4GL Interactive Debugger*.

RDS Programs That Call C Functions

If your Rapid Development System program calls programmer-defined C functions, you must create a customized runner to execute the program.

To create a customized runner

1. Edit a structure definition file to contain information about your C functions.

   This file is named **fgiusr.c** and is supplied with 4GL.

2. Compile and link the **fgiusr.c** file with the files that contain your C functions.

   To do this, use the **cfglgo** command.

You can then use the runner produced by the **cfglgo** command to run the 4GL program that calls your C functions. Both the **fgiusr.c** file and the **cfglgo** command are described in the sections that follow.

For an example of how to call C functions from a 4GL program, see *INFORMIX-4GL by Example*.

Important: *To create a customized runner, you must have a C compiler installed on your system. If the only functions that your Rapid Development System program calls are 4GL or ESQL/C library functions, or functions written in the 4GL language, you do not need a C compiler and you do not need to follow the procedures described in this section.*

Editing the fgiusr.c File

The **fgiusr.c** file is located in the **/etc** subdirectory of the directory in which you installed 4GL (that is, in **$INFORMIXDIR/etc**). The following listing shows the **fgiusr.c** file in its unedited form:

```
/*********************************************************
*                                                        *
*           INFORMIX SOFTWARE, INC.                      *
*                                                        *
* Title: fgiusr.c                                        *
* Sccsid: @(#)fgiusr.c   4.2   8/26/87  10:48:37         *
* Description:                                           *
*        definition of user C functions                  *
*                                                        *
*********************************************************
*/

/*****************************************************
* This table is for user-defined C functions.
*
* Each initializer has the form:
*
*  "name", name, nargs
*
* Variable # of arguments:
*
*  set nargs to -(maximum # args)
*
* Be sure to declare name before the table and to leave the
* line of 0's at the end of the table.
*
* Example:
*
*  You want to call your C function named "mycfunc" and it expects
*  2 arguments.  You must declare it:
*
*     int mycfunc();
*
*  and then insert an initializer for it in the table:
*
*     "mycfunc", mycfunc, 2
*****************************************************
*/

#include "fgicfunc.h"

cfunc_t usrcfuncs[] =
    {
    0, 0, 0
    };
```

The **fgiusr.c** file is a C language file that you can edit to declare any number of programmer-defined C functions.

To edit **fgiusr.c**, you can copy the file to any directory. (Unless this is your working directory at compile time, you must specify the full pathname of the edited **fgiusr.c** file when you compile.) Edit **fgiusr.c** to specify the following:

- A declaration for each function:

```
int function-name( int nargs);
```

- Three initializers for each function:

```
" function-name ", function-name, [ - ] integer,
```

In the declaration of the function, the parenthesis symbols ( ) must follow the *function-name*.

The first initializer is the function name between double quotation marks and is a character pointer.

The second initializer is the function name (without quotation marks) and is a function pointer. It cannot include a parentheses.

The third initializer is an *integer* representing the number of arguments expected by the function. If the number of arguments expected by the function can vary, you must make the third argument the maximum number of arguments, prefixed with a minus (-) sign.

You must use a comma (,) symbol to separate each of the three initializers. Insert a set of initializers for each C function that you declare. A line of three zeroes indicates the end of the structure.

Here is an example of an edited **fgiusr.c** file:

```
#include "fgicfunc.h"

int function-name();

cfunc_t usrcfuncs[] =
{
   {"function-name",function-name,1},
   { 0,0,0 }
};
```

Here the 4GL program will be able to call a single C function called *function-name* that has one argument.

If you have several 4GL programs that call C functions, you can use **fgiusr.c** in either of two ways:

- You can create one customized p-code runner.

  In this case, you can edit **fgiusr.c** to specify all the C functions called from all your 4GL programs. After you create one comprehensive runner, you can use it to execute all your 4GL applications.

- You can create several application-specific runners.

  In this case, you can either make a copy of the **fgiusr.c** file (with a new name) for each customized runner, or you can re-edit **fgiusr.c** to contain information on the C functions for a specific application before you compile and link. If you create several runners, you must know which customized runner to use with each 4GL application.

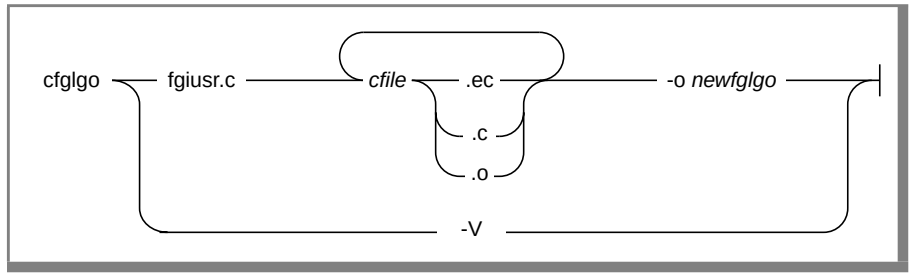
In some situations the first method is more convenient, because users do not need to keep track of which runner supports each 4GL application.

Creating a Customized Runner

You can use the **cfglgo** command to create a customized runner. You can use **cfglgo** to compile C modules and ESQL/C modules that contain functions declared in an edited **fgiusr.c** file. The customized runner can also run 4GL programs that do not call C functions.

cfglgo Command

The **cfglgo** command supports the following syntax.



Element	Description
cfile	is the name of a source file containing ESQL/C or C functions to be compiled and linked with the new runner, or the name of an object file previously compiled from a **.c** or **.ec** file.
newfglgo	specifies the name of the customized runner.

You need the ESQL/C product to compile **.ec** files with **cfglgo**.

The **cfglgo** command compiles and links the edited **fgiusr.c** file with your C program files into an executable program that can run your 4GL application. Here **fgiusr.c** is the name of the file that you edited to declare C or ESQL/C functions. If the **fgiusr.c** file to be linked is not in the current directory, you must specify a full pathname. You can also rename the **fgiusr.c** file.

You can specify any number of uncompiled or compiled C or ESQL/C files in a **cfglgo** command line. When you create a customized runner with **cfglgo**, however, you cannot include among the *cfile***.o** object files any file that was compiled from a 4GL program. (The **cfglgo** script has no syntax for linking the 4GL libraries that would be required in this context.)

If you do not specify the **-o** *newfglgo* option, the new runner is given the default name **a.out**.

To display the version number of the software, specify the **-V** option. The version number of your SQL and p-code software appears on the screen. Any other command options are ignored. After displaying this information, the program terminates without creating a customized p-code runner.

Examples

The following example 4GL program calls the C function **prdate( )**:

prog.4gl:

```
main
. . .
call prdate()
. . .
end main
```

The function **prdate( )** is defined in file **cfunc.c**, as shown here:

cfunc.c:

```
#include <errno.h>
#include <stdio.h>
#include <time.h>

int prdate(int nargs)
{
/* This program timestamps file FileX */

  long cur_date;
  FILE *fptr;

  time(&cur_date);
  fptr = fopen("time_file","a");
  fprintf(fptr,"FileX was accessed %s", ctime(&cur_date));
  fclose(fptr);
  return(0);
}
```

The C function is declared and initialized in the following **fgiusr.c** file:

fgiusr.c:

```
1 #include "fgicfunc.h"
2
3 int prdate();
4 cfunc_t usrcfuncs[] =
5    {
6   { "prdate", prdate, 0 },
7   { 0, 0, 0 }
8    };
```

An explanation of this example of an **fgiusr.c** file follows.

Line	Description
1	The file **fgicfunc.h** is always included. This line already exists in the unedited **fgiusr.c** file.
3	This is the declaration of the function **prdate( )**. You must add this line to the file.
4	This line already exists in the unedited file. It declares the structure array *usrcfuncs*.
6	This line contains the initializers for function **prdate( )**. Because it expects no arguments, the third value is zero.
7	The line of three zeros indicates that no more functions are to be included.

In this example, you can use the following commands to compile the 4GL program, to compile the new runner, and to run the program:

- To compile the example 4GL program:

  ```
  fglpc prog.4gl
  ```

- To compile the new runner:

  ```
  cfglgo fgiusr.c cfunc.c -o newfglgo
  ```

- To run the 4GL program:

  ```
  newfglgo prog.4go
  ```

Running Programs That Call C Functions

After you create a customized runner, you can use it to execute any 4GL program whose C functions you correctly specified in the edited **fgiusr.c** file. The syntax of a customized runner (apart from its name) is the same as the syntax of **fglgo**, as described in "Running RDS Programs" on page 1-80.

You can also create a customized Debugger to run a 4GL program that calls C functions. See the *Guide to the INFORMIX-4GL Interactive Debugger* for details and an example of how to create a customized debugger.

Important: *You cannot create a customized runner or debugger from within the Programmer's Environment. You must work from the system prompt and follow the procedures described in "Creating Programs at the Command Line" on page 1-75 if you are developing a 4GL program that calls user-defined C functions. Then you can return to the Programmer's Environment and use the* ***Program_Runner*** *option of the* ***MODIFY PROGRAM*** *menu or the* ***NEW PROGRAM*** *menu to specify the name of a customized runner or debugger.*

Program Filename Extensions

Source, runable, error, and backup files generated by 4GL are stored in the current directory and are labeled with the appropriate filename extensions, as described in the following table. These files are produced during the normal course of using the Rapid Development System.

File	Description
file.**4gl**	4GL source file
file.**4go**	4GL file that has been compiled to p-code
file.**4gi**	4GL file that has been compiled to p-code
file.**err**	4GL source error file, created when an attempt to compile a module fails or produces a warning. (The file contains the 4GL source code plus compiler syntax warnings or error messages.)
file.**erc**	4GL object error file, created when an attempt to compile or link a non-4GL source-code or object module fails (The file contains 4GL source code and annotated compiler errors.)

(1 of 2)

File	Description
file.**per**	FORM4GL source file
file.**frm**	FORM4GL object file
file.**err**	FORM4GL source error file

(2 of 2)

The last three files do not exist unless you create or modify a screen form specification file, as described in Chapter 6.

The following table lists backup files that are produced when you use 4GL from the Programmer's Environment.

File	Description
file.**4bl**	4GL source backup file, created during the modification and compilation of a **.4gl** program module
file.**4bo**	Object backup file, created during the compilation of a **.4go** program module
file.**4be**	Object backup file, created during the compilation of a **.4gi** program module
file.**pbr**	FORM4GL source backup file
file.**fbm**	FORM4GL object backup file

Under normal conditions, 4GL creates the backup files and intermediate files as necessary, and deletes them when a compilation is successful. If you interrupt a compilation, you might find one or more of the files in your current directory.

If you compile with a **fglpc** command line that includes the **p** *pathname* option, 4GL creates the **.4gi, .4go, .err,** and corresponding backup files in the directory specified by *pathname*, rather than in your current directory.

During the compilation process, 4GL stores a backup copy of the *file*.**4gl** source file in *file*.**4bl**. The time stamp is modified on the (original) *file*.**4gl** source file, but not on the backup *file*.**4bl** file. In the event of a system crash, you might need to replace the modified *file*.**4gl** file with the backup copy contained in the *file*.**4bl** file.

The Programmer's Environment does not allow you to begin modifying a **.4gl** or **.per** source file if the corresponding backup file already exists in the same directory. After an editing session terminates abnormally, for example, you must delete or rename any backup file before you can resume editing your 4GL module or form specification from the Programmer's Environment.

Warning: *INFORMIX-4GL is not designed to support two or more programmers working concurrently in the same directory. If several developers are working on the same 4GL application, make sure that they do their work in different directories.*

Chapter

2

The INFORMIX-4GL Language

In This Chapter

An INFORMIX-4GL program consists of at least one source file that contains a series of English-like statements. These obey a well-defined syntax that this book describes.

This chapter presents a brief overview of the 4GL language. Its theory, application, constructs, and semantics are described in detail in *INFORMIX-4GL Concepts and Use*, a companion volume to this manual.

This manual assumes that you are using Informix Dynamic Server as your database server. Features specific to INFORMIX-SE are noted.

Language Features

4GL is an English-like C or COBOL-replacement programming language that Informix Software, Inc., introduced in 1986 as a tool for creating relational database applications. Its statement set (see Chapter 4, "INFORMIX-4GL Statements") includes the industry-standard SQL language for accessing and manipulating a relational database. The 4GL development environment provides a complete environment for writing 4GL programs.

Lettercase Insensitivity

4GL is case insensitive, making no distinction between uppercase and lowercase letters, except within quoted strings. Use pairs of double (") or single (') quotation marks in 4GL code to preserve the lettercase of character literals, filenames, and names of database entities, such as cursor names.

You can mix uppercase and lowercase letters in the identifiers that you assign to 4GL entities, but any uppercase letters in 4GL identifiers are automatically shifted to lowercase during compilation.

Whitespace, Quotation Marks, Escape Symbols, and Delimiters

4GL is free-form, like C or Pascal, and generally ignores TAB characters, LINEFEED characters, comments, and extra blank spaces between statements or statement elements. You can freely use these whitespace characters to make your 4GL source code easier to read.

Blank (ASCII 32) characters act as delimiters in some contexts. Blank spaces must separate successive keywords or identifiers, but cannot appear within a keyword or identifier. Pairs of double (") or single (') quotation marks must delimit any character string that contains a blank (ASCII 32) or other whitespace character, such as LINEFEED or RETURN.

Do not mix double and single quotation marks as delimiters of the same string. For example, the following is not a valid character string:

```
'Not A valid character string"
```

If you are using Informix DRDA software to access a non-Informix relational database, such as a DB2 database from IBM, double quotation marks might not be recognized as delimiters by the non-Informix database.

Similarly, most 4GL statements require single quotation marks if the database supports delimited SQL identifiers, and in this special case cannot use double quotation marks in most contexts, because when the **DELIMIDENT** environment variable is set, double quotation marks are reserved for SQL identifiers.

To include literal quotation marks within a quoted string, precede each literal quotation mark with the backslash (\), or else enclose the string between a pair of the opposite type of quotation marks:

```
DISPLAY "Type 'Y' if you want to reformat your disk."
DISPLAY 'Type "Y" if you want to reformat your disk.'
DISPLAY 'Type \'Y\' if you want to reformat your disk.'
```

The 4GL compiler treats a backslash as the default escape symbol, and treats the immediately following symbol as a literal, rather than as having special significance. To specify anything that includes a literal backslash, enter double (\\ ) backslashes wherever a single backslash is required. Similarly, use \\\\ to represent a literal double backslash.

Except in some PREPARE and PRINT statements, and the END SQL keywords in SQL blocks, 4GL requires no statement terminators, but you can use the semicolon (;) as a statement terminator.

Statements of the SQL language, however, that include syntax later than what Informix 4.10 database servers support require SQL...END SQL delimiters, unless the post-4.10 SQL statement appears as text in a PREPARE statement.

Character Set

4GL requires the ASCII character set, but also supports characters from the client locale in data values, identifiers, form specifications, and reports. For more information, see "Naming Rules for 4GL Identifiers" on page 2-14.

4GL Statements

4GL source-code modules can contain statements and comments:

- A *statement* is a logical unit of code within 4GL programs. See Chapter 4 for a list of the statements in the 4GL statement set.
- A *comment* is a specification that 4GL disregards. For more information, see "Comment Indicators" on page 2-8.

A compilation error occurs if a program (or one of its modules or statement blocks) includes part of a statement but not all of the required elements.

Statements of 4GL can contain identifiers, keywords, literal values, constants, operators, parentheses, and expressions. These terms are described in subsequent sections of this chapter, and in Chapter 4.

For the purposes of this manual, 4GL supports two types of statements:

- SQL (Structured Query Language) statements
- Other 4GL language statements

This distinction among statements reflects whether they provide instructions to the database server (SQL statements) or instructions to the client application (other 4GL statements). *INFORMIX-4GL Concepts and Use* describes the process architecture of 4GL applications. See the documentation of your Informix database server for the syntax of SQL statements. Chapter 4 of this manual describes the syntax of 4GL statements that are not SQL statements.

Where a given statement can appear within a 4GL program, how the Interactive Debugger treats it, and whether the statement has its effect at compile time or at runtime all depend on whether the statement is executable.

The following statements of 4GL are *non-executable*. They define program blocks, declare 4GL identifiers, or act as compiler directives.

DEFER	FUNCTION	LABEL	REPORT
DEFINE	GLOBALS	MAIN	WHENEVER

The following statements of 4GL are executable.

CALL	FOREACH	OPTIONS
CASE	FREE	OUTPUT TO REPORT
CLEAR	GOTO	PAUSE
CLOSE FORM	IF	PRINT
CLOSE WINDOW	INITIALIZE	PROMPT
CONSTRUCT	INPUT	RETURN
CONTINUE	INPUT ARRAY	RUN
CURRENT WINDOW	LET	SCROLL
DISPLAY	LOAD	SKIP
DISPLAY ARRAY	LOCATE	SLEEP
DISPLAY FORM	MENU	START REPORT
ERROR	MESSAGE	TERMINATE REPORT
EXIT	NEED	UNLOAD
FINISH REPORT	OPEN FORM	VALIDATE
FOR	OPEN WINDOW	WHILE

4GL statements (individually described in Chapter 4) begin with keywords. Some statements can include delimiters (as described earlier in this section) and expressions (as described in Chapter 3, “Data Types and Expressions”).

Within the broad division into SQL statements and other statements, the 4GL statement set can be further classified into functional categories, whose component statements are listed in "The 4GL Statement Set" on page 4-9.

Types of SQL Statements	Other Types of 4GL Statements
Data definition statements	Definition and declaration statements
Data manipulation statements	Program flow control statements
Cursor manipulation statements	Compiler directives
Dynamic management statements	Storage manipulation statements
Query optimization statements	Screen interaction statements
Data access statements	Report execution statements
Data integrity statements	
Stored procedure statements	
Optical statements	

"The 4GL Statement Set" on page 4-9 identifies the SQL statements and other 4GL statements that make up these functional categories.

Some statements, called *compound* statements (described in "Statement Blocks" on page 2-12), can contain other 4GL statements. A set of nested statements within a compound statement is called a *statement block*. When necessary, 4GL uses END (with another keyword to indicate a specific statement) to terminate a compound statement.

Except in a few special cases, like multiple-statement prepared entities, 4GL requires no statement terminators, but you can use the semicolon as a statement terminator. If you have difficulty interpreting a compilation error, you might want to insert semicolons to separate the statements that precede the error message, to indicate to the 4GL compiler where each statement ends. (The PRINT statement in 4GL reports can use semicolons to control the format of output from a report by suppressing LINEFEED.)

Screen forms of 4GL are manipulated by form drivers but are defined in form specification files. These ASCII files use a syntax that is distinct from the syntax of other 4GL features. See Chapter 6, "Screen Forms," for details of the syntax of 4GL form specification files.

Comments

A comment is text in 4GL source code to assist human readers, but which 4GL ignores. (This meaning of *comment* is unrelated to the COMMENTS attribute in a form, or to the OPTIONS...COMMENT LINE statement, both of which control on-screen text displays to assist users of the 4GL application.)

Comment Indicators

You can indicate comments in any of several ways:

- A comment can begin with the left-brace (**{**) and end with the right-brace (**}**) symbol. These can be on the same line or on different lines.
- The pound (**#**) symbol (sometimes called the "sharp symbol") can begin a comment that terminates at the end of the same line.
- You can use a pair of hyphens or minus signs (**--**) to begin a comment that terminates at the end of the current line. (This comment indicator conforms to the ANSI standard for SQL.)

4GL ignores all text between braces (or from the **#** or **--** comment indicator to the end of the same line).

For clarity and to simplify program maintenance, it is recommended that you document your 4GL code by including comments in your source files. You can also use comment indicators during program development to disable statements without deleting them from your source-code modules.

Restrictions on Comments

When using comments, keep the following restrictions in mind:

- Within a quoted string, 4GL interprets comment indicators as literal characters, rather than as comment indicators.
- Comments cannot appear in the SCREEN section of a form specification file.
- The **#** symbol cannot indicate comments in a form specification, in an SQL statement block, nor in the text of a prepared statement.
- You cannot use braces (**{ }**) to nest comments within comments.

- You cannot specify consecutive minus signs (--) in arithmetic expressions, because 4GL interprets what follows as a comment. Instead, use a blank space or parentheses to separate consecutive arithmetic minus signs. For example:

```
LET x = y --3 # Now variable x evaluates as y
           # because 4GL ignores text after --

LET x = y -(-3) # Now variable x evaluates as (y + 3).
```

- The symbol that immediately follows the -- comment indicator must not be the sharp (#) or at (@) symbols, unless you intend to compile the same 4GL source file with the Dynamic 4GL product. For details of the special significance of the # or @ symbol after the -- symbols, see the next section, "Conditional Comments" on page 2-9.

Conditional Comments

Another Informix product, Informix Dynamic 4GL, treats the --# characters as a whitespace character, rather than as the beginning of a comment. This feature provides backward compatibility for source-code modules that begin lines containing Dynamic 4GL extensions to 4GL syntax with those symbols. When compiled by Dynamic 4GL, lines so marked are treated as statements in a graphical user interface (GUI) environment. 4GL, however, interprets --# as a comment symbol so that Dynamic 4GL syntax extensions (or anything else) that follow in the same line are ignored.

Conversely, the --@ symbols act as a conditional comment indicator. The 4GL compiler interprets this indicator as a single whitespace character and reads the rest of the line. In contrast, the Dynamic 4GL compiler interprets this as a comment and ignores the rest of the line.

These symbols are called *conditional* comment indicators because their interpretation depends on which compiler you use. Together, these features enable the same source file to support different features, depending on whether you compile with Dynamic 4GL or with 4GL.

Conditional comments are supported both in source (**.4gl**) files, in form specification (**.per**) files, and in INFORMIX-SQL report (**.ace**) files.

For example, you could use conditional comments to specify a Help file:

```
--# OPTIONS HELP FILE "d4gl_help.42h"  --Line is ignored by I-4GL
--@ OPTIONS HELP FILE "i4gl_help.iem"  --Line is ignored by D-4GL
```

The following example shows a fragment of a form specification:

```
ATTRIBUTES
            f0 = FORMONLY.name, --#char_var ;
                                --@REVERSE ;
```

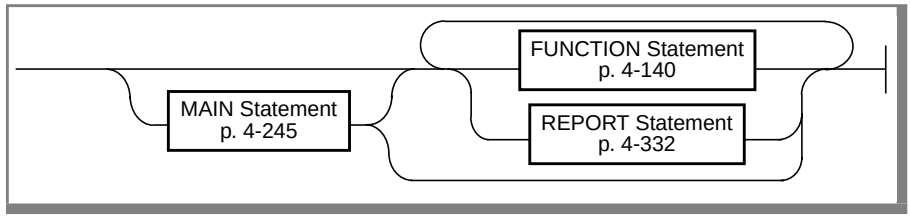
Conditional comments are also valid within SQL statement blocks, but not within the text of a PREPARE statement.

Do not put both forms of conditional comments in the same line.

Source-Code Modules and Program Blocks

When you create a 4GL program, enter statements and comments into one or more source code files, called modules, whose filenames can have no more than 10 characters, excluding any file extensions.

Because 4GL is a structured language, executable statements are organized into larger units, called *program blocks* (sometimes called *routines*, *sections*, or *functions*). 4GL modules can include three different kinds of program blocks: MAIN, FUNCTION, and REPORT.



Each block begins with the keyword after which it is named, and ends with the corresponding END *statement* keywords (END MAIN, END FUNCTION, or END REPORT). Program blocks can support 4GL applications in several ways:

- As part of a complete 4GL program (one that includes a MAIN block)
- As a FUNCTION or REPORT block that is invoked by a 4GL program
- As a 4GL FUNCTION block called by a C or ESQL/C program (INFORMIX-ESQL/C requires a separate license.)

The following rules apply to 4GL program blocks:

- Every 4GL program must contain exactly one MAIN block. This must be the first program block of the module in which it appears.
- Except for certain declarations (DATABASE, DEFINE, GLOBALS), no 4GL statement can appear outside a program block.
- Variables that you declare within a program block have a scope of reference (described in "Scope of Reference of 4GL Identifiers" on page 2-17) that is *local* to the same program block. They cannot be referenced from other program blocks. (Variables that you declare outside any program block have a scope of reference extending from their declaration until the end of the same source module.)
- The GO TO or GOTO keywords cannot reference a statement label in a different program block. (For more information about statement labels, see the GOTO and LABEL statements in Chapter 4.)
- Program blocks cannot be nested; neither can any program block be divided among more than one source-code module.
- The DATABASE statement (described in Chapter 4) has a compile-time effect when it appears before the first program block of a source module. Within a program block, it has a runtime effect.
- The scope of the WHENEVER statement extends from its occurrence to the next WHENEVER statement that specifies the same exceptional condition, or to the end of the same module (whichever comes first), but WHENEVER cannot occur outside a program block.

CALL, RETURN, EXIT REPORT, START REPORT, OUTPUT TO REPORT, FINISH REPORT, and TERMINATE REPORT statements, and any 4GL expression that includes a programmer-defined function as an operand, can transfer control of program execution between program blocks. These statements are all described in Chapter 4, "INFORMIX-4GL Statements"; expressions of 4GL are described in Chapter 3, "Data Types and Expressions.".

Chapter 5, "Built-In Functions and Operators," describes FUNCTION blocks, and Chapter 7, "INFORMIX-4GL Reports," describes REPORT blocks. (See Chapter 1, "Compiling INFORMIX-4GL Source Files," for details of how source-code modules are compiled and linked to create applications, and for details about the naming conventions for filenames and for file extensions of 4GL modules.)

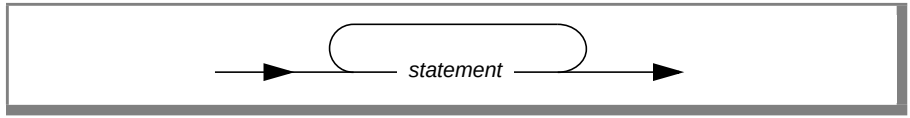
Statement Blocks

The MAIN, FUNCTION, and REPORT statements are special cases of *compound statements*, the 4GL statements that can contain other statements.

CASE	FOREACH	INPUT	PROMPT
CONSTRUCT	FUNCTION	INPUT ARRAY	REPORT
DISPLAY ARRAY	GLOBALS	MAIN	SQL
FOR	IF	MENU	WHILE

Every compound statement of 4GL supports the END keyword to mark the end of the compound statement construct within the source-code module. Most compound statements also support the EXIT *statement* keywords, to transfer control of execution to the statement that follows the END *statement* keywords, where *statement* is the name of the compound statement.

By definition, every compound statement can contain at least one statement block, a group of one or more consecutive SQL statements or other 4GL statements. In the syntax diagram of a compound statement, a statement block always includes this element.



(Some contexts permit or require semicolon (;) delimiters. Any SQL statement that is not prepared but includes syntax later than what Informix 4.10 database servers support must be enclosed between SQL and END SQL keywords.)

These are examples of statement blocks within compound 4GL statements:

- The WHEN, OTHERWISE, THEN, or ELSE blocks of CASE and IF statements
- Statements within FOR, FOREACH, or WHILE loops
- CONSTRUCT, DISPLAY ARRAY, INPUT, or INPUT ARRAY control blocks
- Statements following the COMMAND clauses of MENU statements
- Statements within the ON KEY blocks of PROMPT statements
- FORMAT section control blocks of REPORT statements
- All the statements in MAIN, FUNCTION, or REPORT program blocks

4GL permits any statement block to be empty, even if it appears as a required element in its syntax diagram. This feature enables you to compile and execute applications that contain empty (or *dummy*) functions or reports, to test the behavior of a not-yet-complete program.

Unlike program blocks, which cannot be nested, 4GL statement blocks can contain other statement blocks. This recursion can be static, as when a function includes a FOREACH loop that contains an IF statement. Blocks can also be recursive in a dynamic sense, as when a CALL statement invokes a function only if some specified condition occurs.

Although most 4GL statements can appear within statement blocks, and most compound statements can be nested, some restrictions apply. In some cases, you can circumvent these restrictions by invoking a function to execute a statement that cannot appear directly within a given compound statement.

The GLOBALS *filename* statement can incorporate statement blocks indirectly by referencing a file. The statements in the specified file are incorporated into the current module during compilation.

Statement Segments

Any subset of a 4GL statement, including the entire statement, is called a *statement segment*. For clarity, many syntax diagrams in this book use rectangles to represent statement segments (for example, MAIN, FUNCTION, and REPORT in "Source-Code Modules and Program Blocks" on page 2-10). These are expanded into syntax diagrams on the page referenced in the rectangle, or elsewhere on the same page, if the rectangle indicates no page number. For your convenience, the diagrams of a few important segments are repeated on different pages.

Chapter 4 describes certain statement segments that can appear as elements of various 4GL statements.

4GL Identifiers

Statements and form specifications can reference some 4GL program entities by name. To create a named program entity, you must declare a 4GL *identifier*. When you create any of the following program entities, you must follow the guidelines in "Naming Rules for 4GL Identifiers" on page 2-14 and adhere to the declaration procedures of 4GL to declare a valid identifier.

Named Program Entity	How Name Is Declared
4GL function or its argument	FUNCTION statement
4GL program variable	DEFINE and GLOBAL statements
4GL report or its argument	REPORT statement
4GL screen array	ATTRIBUTES section of form specification
4GL screen form	OPEN FORM statement
4GL screen record	INSTRUCTIONS section of form specification
4GL statement label	LABEL statement
4GL table alias	TABLES section of form specification
4GL window	OPEN WINDOW statement

This list excludes columns, constraints, cursors, databases, indexes, prepared statements, stored procedures, synonyms, tables, triggers, views, and other database objects, because those are SQL entities, not 4GL entities. It also omits filenames, pathnames, and user names, which must conform to the naming rules of your operating system or network.

Naming Rules for 4GL Identifiers

A 4GL identifier is a character string that is declared as the name of a program entity. In the default (U.S. English) locale, every 4GL identifier must conform to the following rules:

- It must include at least one character, but no more than 128.
- Only ASCII letters, digits, and underscore (_) symbols are valid. Blanks, hyphens, and other non-alphanumerics are not allowed.
- The initial character must be a letter or an underscore.
- 4GL identifiers are not case sensitive, so **my_Var** and **MY_vaR** both denote the same identifier.

GLS

Within non-English locales, however, 4GL identifiers can include non-ASCII characters in identifiers, if those characters are defined in the code set of the locale that **CLIENT_LOCALE** specifies. In multibyte East Asian locales that support languages whose written form is not alphabet-based, such as Chinese, Japanese, or Korean, a 4GL identifier need not begin with a letter, but the storage length of a 4GL identifier cannot exceed 128 bytes. ♦

Important: *You might get unexpected results if you declare as an identifier certain keywords of SQL, the C and C++ languages, or your operating system or network. (Appendix G, "Reserved Words," lists some keywords and predefined identifiers of 4GL that should not be declared as identifiers of programmer-defined entities.) If you receive an error message that seems unrelated to the SQL or other 4GL statement that elicits the error, see if the statement references a reserved word as an identifier.*

In releases of 4GL earlier than 7.3, the total length of all names of functions, reports, and variables in an 4GL program that was compiled to p-code could not exceed 65,535 bytes. This release is not subject to that restriction; the upper limit on what is called your *global string space* is now two gigabytes (unless some smaller limit is imposed by the memory capacity of the system on which the 4GL program is running).

If you are using the C Compiler version of 4GL, your C compiler might only recognize the first 31 characters (or the first 69) of a 4GL identifier. In this case, or if your application must be portable to all C compilers, keep the first 31 characters unique among similar program entities that have the same scope of reference. (Scope of reference is explained later in this section.)

Naming Rules for SQL Identifiers

The rules for SQL identifiers resemble those of 4GL, with these exceptions:

- For most Informix database servers, SQL identifiers are limited to no more than 18 characters. (But *database* names might be limited to 8, 10, or 14 characters, depending on the database server and operating system environment.)
- Informix 9.2 and later databases can have a limit of 128-bytes.
- SQL identifiers within quoted strings are case-sensitive.
- You can use reserved words as SQL identifiers (but such usage might require qualifiers, and can make your code difficult to maintain).

The 4GL identifiers can be the same as SQL identifiers, but this might require special attention within the scope of the 4GL identifier. For more information, see "Scope and Visibility of SQL Identifiers" on page 2-19.

If your 4GL application is the client of an Informix database server on which the **IFX_LONGID** environment variable has been set to 1, then 4GL code can reference database objects with SQL identifiers up to 128 bytes in length.

GLS

If the database and its connectivity software accept non-ASCII characters, 4GL can recognize characters valid for the locale in these SQL identifiers.

Column name	Index name	Synonym
Connection name	Log file name	Table name
Constraint name	Role name	Trigger name
Database name	Stored procedure name	View name

For INFORMIX-SE database servers, whether non-English characters are permitted in the names of databases, tables, or log files depends on whether the operating system permits such characters in filenames. What characters are valid in SQL identifiers depends on the database locale. See the *Informix Guide to GLS Functionality* for additional details of SQL identifiers.

When the client locale and the database locale are different, 4GL does not support use of the LIKE keyword in declarations of 4GL records that assign default names to record members, if LIKE references a *table* or *column* whose name includes any characters outside the code set of the client locale. ♦

Warning: *Informix database servers (Version 7.0 and later) support DELIMIDENT, an environment variable that can extend the character set of SQL. 4GL cannot, however, reference database objects whose names include any non-alphanumeric characters (such as blank spaces) outside the character set of 4GL identifiers.*

ANSI

The ANSI standard for SQL requires that all identifiers, including user names, be in uppercase letters. Before passing to the database server a user name that includes any lowercase letters, 4GL converts the name to uppercase letters, if the compile-time database is ANSI/ISO-compliant, or if the **DBANSIWARN** variable is set, or if the **-ansi** compilation flag is used.

To specify a user name that is all lowercase or a combination of uppercase and lowercase letters, you must enclose the name in quotation marks. 4GL passes to the database server any user name enclosed in quotation marks with the case of the letters intact. When a user specifies this name, quotation marks must be placed around the name. The only situations in which 4GL does not convert an unquoted, lowercase user name to uppercase is when the user name is **informix** or **public**.

The following example specifies the name **james** as the owner of the table **custnotes**:

```
CREATE TABLE "james".custnotes
   (customer_num INTEGER, notes CHAR(240))
```

You can access the table as shown in the following example:

```
SELECT * FROM "james".custnotes. ♦
```

Scope of Reference of 4GL Identifiers

Any 4GL identifier can be characterized by its scope of reference, sometimes called its name scope, or simply its scope. A point in the program where an entity can be referenced by its identifier is said to be *in* the scope of reference of that identifier. Conversely, any point in the program where the identifier cannot be recognized is said to be *outside* its scope of reference.

Identifiers of Variables

The scope of reference of a variable is determined by where in the **.4gl** source module the DEFINE statement appears that declares the identifier. Identifiers of variables can be local, module, or (in some cases) global in their scope:

- Local 4GL variables are declared within a program block. These variables cannot be referenced by statements outside the same program block.
- Module variables (sometimes called modular or static) must be declared outside any MAIN, REPORT, or FUNCTION program block. These identifiers cannot be referenced outside the same **.4gl** module.

If the GLOBALS…END GLOBALS statement declares variables in one module, you can extend the scope of those variables to any other module that includes a GLOBALS *filename* statement, where *filename* specifies the file that contains the GLOBALS…END GLOBALS statement.

Module identifiers whose scope has been extended to additional modules by this mechanism are sometimes said to have *global* scope, even if they are out of scope in some modules. Truly global in scope, however, are the names of the constants NOTFOUND, TRUE and FALSE, and *built-in* variables like status, **int_flag**, **quit_flag**, and the **SQLCA** record. These predefined identifiers do not need to be declared. Unless you declare a conflicting identifier, they are visible in any 4GL statement, and can be referenced from any 4GL module, as can the names of the built-in functions and operators like LENGTH( ) and INFIELD( ) that Chapter 4 describes.

Other 4GL Identifiers

Also global in scope are names of 4GL windows, forms, reports, and functions. The scope of the identifiers of form entities (like screen fields, screen arrays, screen records, or table aliases) includes all the 4GL statements that are executed while the form is open.

The following table summarizes the scope of reference of 4GL identifiers for various types of 4GL program entities.

Named 4GL Program Entity	Scope of Reference of 4GL Identifier
Constant	Global (for TRUE, FALSE, and NOTFOUND)
Formal argument	Local to its function or report definition
Function or report	Global
Variable	Module (if declared outside a program block) or local (if declared inside a program block)
Screen field, array, or record	While the form that declares it is displayed
Screen form or window	Global (after it has been declared)
Statement label	Local (to the program block in which it appears)
Table alias	While the form that declares it is displayed

Here each line represents a separate *name space*. With each name space, 4GL identifiers that have the same scope of reference must be unique. (In addition to these restrictions, a formal argument cannot have the same identifier as its own function or report.) For details of how 4GL resolves conflicts between non-unique identifiers within the same name space, see "Visibility of Identical Identifiers" on page 2-19.

In C, global variables and functions share the same name space. Unless you compile your 4GL source code to p-code, a compilation error results if a global variable has the same identifier as a 4GL function or report.

Scope and Visibility of SQL Identifiers

By default, the scope of a cursor or prepared object name is from its DECLARE or PREPARE declaration until the module ends, or until a FREE statement specifies its name. (The **-globcurs** compiler flag makes their scope global.) After FREE, subsequent DECLARE or PREPARE statements in the same module cannot reassign the same name, even to an entity that is identical to whatever FREE referenced. All other SQL identifiers have global scope.

Statements cannot reference the name of a global database entity like a table, column, or index after an SQL data definition statement to DROP the entity is executed, or if the database that contains the entity is not open.

If you assign to a 4GL entity the name of an SQL entity, the 4GL name takes precedence within its scope. To avoid ambiguity in DELETE, INSERT, SELECT, and UPDATE statements (and only in these statements), prefix an @ symbol to the name of any table or column that has the same name as a 4GL variable. Otherwise, only the 4GL identifier is visible in these ambiguous contexts.

Visibility of Identical Identifiers

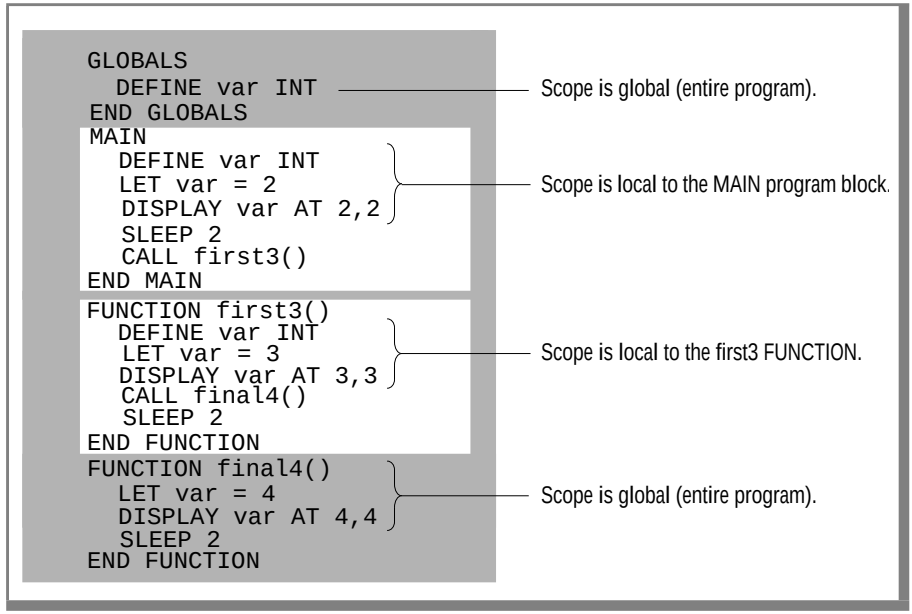
A compile-time error occurs if you declare the same name for two variables that have the same scope. You can, however, declare the same name for variables that differ in their scope. For example, you can use the same identifier to reference different local variables in different program blocks.

You can also declare the same name for two or more variables whose scopes of reference are different but overlapping. Within their intersection, 4GL interprets the identifier as referencing the variable whose scope is smaller, and therefore the variable whose scope is a superset of the other is not visible.

Non-Unique Global and Local Variables

If a local variable has the same identifier as a global variable, then the local variable takes precedence inside the program block in which it is declared. Elsewhere in the program, the identifier references the global variable, as illustrated in the 4GL program in Figure 2-1.

Figure 2-1
Precedence of Local Variables over Global Variables



The shaded area indicates where the global identifier called var is visible. This is superseded in the MAIN statement and in the first FUNCTION program block by local variables that have the same name. Only the last DISPLAY statement references the global variable; the first two display local variables.

Non-Unique Global and Module Variables

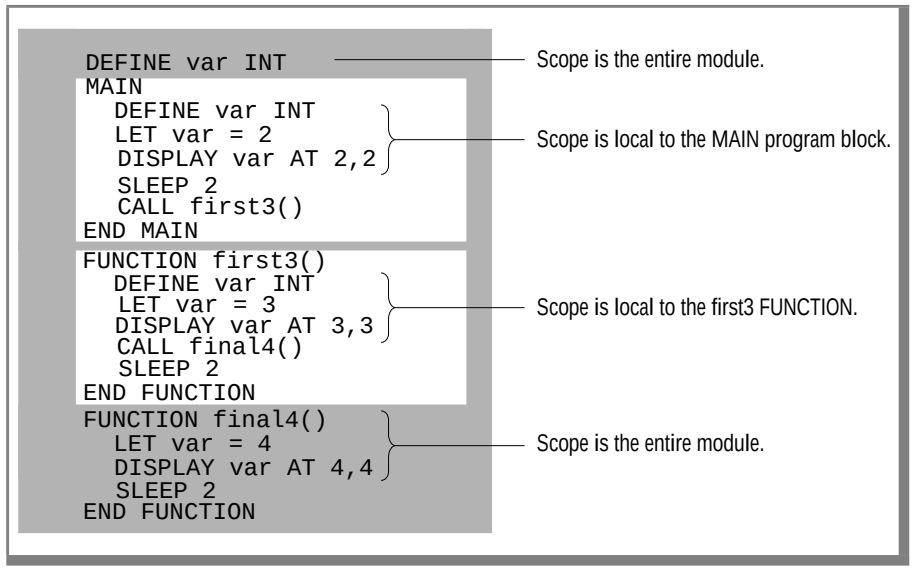
A module variable can have the same name as a global variable that is declared in a different module. Within the module where it is declared, the module variable takes precedence over the global variable. Statements in that module cannot reference the global variable.

A module variable cannot have the same name as a global variable that is declared in the same module.

Non-Unique Module and Local Variables

If a local variable has the same identifier as a module variable, then the local identifier takes precedence inside the program block in which it is declared. Elsewhere in the same source-code module, the name references the module variable, as illustrated in Figure 2-2.

Figure 2-2
Precedence of Local Variables over Module Variables



The shaded area indicates where the module variable called **var** is visible. This is superseded in the MAIN block and in the first FUNCTION program block by the identifiers of local variables called **var**. The first two DISPLAY statements show values of local variables; the last displays the module variable.

In the portion of a program where more than one variable has the same identifier, 4GL gives precedence to a module identifier over a global one, and to a local identifier over one with any other scope. Assign unique names to variables if you wish to avoid masking part of the scope of non-unique module identifiers.

Interacting with Users

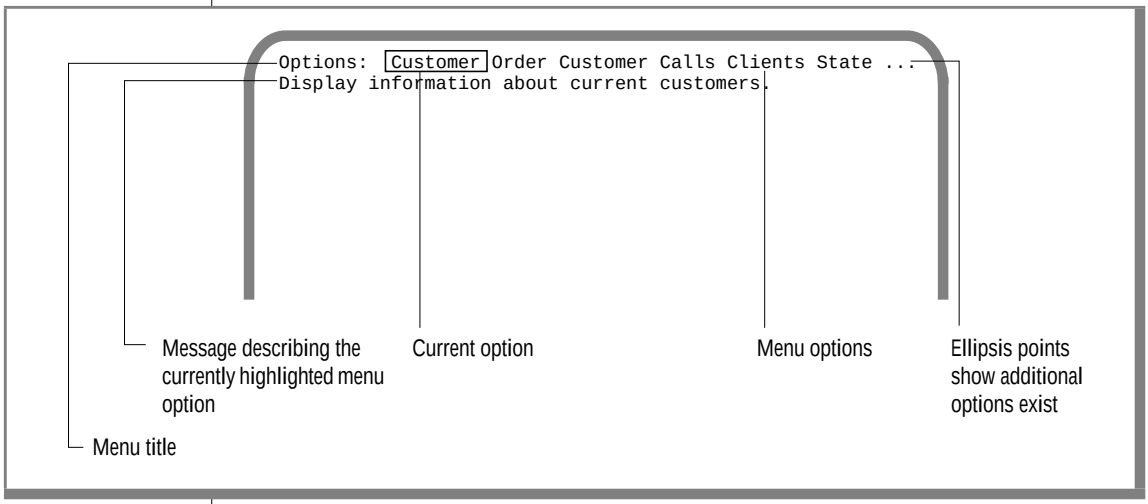
You can use 4GL to create applications composed of the following interface elements:

- Menus
- Screen forms
- 4GL windows
- Help messages
- Reports based on data retrieved from an SQL database (described in Chapter 7, "INFORMIX-4GL Reports")

Ring Menus

You can use the MENU statement of 4GL to create and display a *ring menu* of command options, so that users can perform the tasks that you specify. The menu of a typical 4GL program, for example, might look like Figure 2-3.

Figure 2-3
The Format of a Typical 4GL Ring Menu



Option names are not 4GL identifiers, and can include embedded blank characters and other printable characters. By default, an option is chosen when the user types its initial character, but you can specify additional activation keys. Different menus can have the same option names.

If a menu has more options than can fit on one line, ellipsis points automatically indicate that more options appear on another page of the menu. In this example, the ellipsis indicates that additional menu options are on one or more pages to the right. Similarly, an ellipsis on the left means that additional menu options are on one or more menu pages to the left.

The user can scroll to the right to display the next page of options by using the RIGHT ARROW or SPACEBAR, or scroll to the left with the LEFT ARROW.

You can nest MENU statements within other MENU statements, so that the menus form a hierarchy. A nested MENU statement can appear directly within a menu control block, or else in a function that is called directly or indirectly when the user chooses an option of the enclosing menu.

Selecting Menu Options

By pressing RETURN, the user can select the menu option that is currently highlighted in reverse video. In the previous example, **Customer** would be selected. The highlight that indicates the current option is called the *menu cursor*.

The menu in Figure 2-3 on page 2-23 is called a *ring menu* because the menu cursor behaves as if the list of options were cyclic; if the user moves the cursor to the right, past the last option, then the first option is highlighted. Similarly, moving the menu cursor to the left, past the first option, highlights the last option.

Pressing the key that matches the initial character of a menu option, such as O (for **Order**) in the preceding illustration, selects the corresponding option.

All other options are disabled until the associated COMMAND block completes its execution. Disabled menu options cannot be selected.

Ambiguous Keyboard Selections

If the user makes an ambiguous menu option selection (for example, by typing C in the 4GL menu containing **Customer**, **Customer Calls**, and **Clients** in the previous example, 4GL clears the second line of the menu and prompts the user to clarify the choice. 4GL displays each keystroke, followed by the names of the menu options that begin with the typed letters. When 4GL identifies a unique option, it closes this prompt line and executes the statements associated with the selected menu option. Pressing BACKSPACE undoes the most recently typed key.

Hidden Options and Invisible Options

You can suppress the display of any subset of the menu options, disabling these *hidden options*. "The HIDE OPTION and SHOW OPTION Keywords" on page 4-260 describes how the MENU statement can programmatically control whether a menu option is hidden or accessible.

Menus can also include *invisible options*. An invisible option does not appear in the menu, but it performs the specified actions when the user presses the activation key. For a description of how to create invisible options, see "Invisible Menu Options" on page 4-257.

Disabled Menus

Menus themselves are not always accessible. During screen interaction statements like INPUT, CONSTRUCT, INPUT ARRAY, and DISPLAY ARRAY, errors would be likely to result if the user could interrupt the interaction with menu choices. 4GL prevents these errors by disabling the entire menu during the execution of these statements. The menu does not change its appearance when it is disabled.

Reserved Lines for Menus

The first line (called the *Menu line*) lists a title and options of the menu. A *menu cursor* (a double border) highlights the current option. For each option, a *menu control block* specifies statements to execute if the user chooses the option. (For more information, see "The MENU Control Blocks" on page 4-250).

The next line (called the *Menu Help line*) displays a prompt for the currently highlighted option. If the user moves the menu cursor to another option, this prompt is replaced by one for the new current option.

Screen Forms

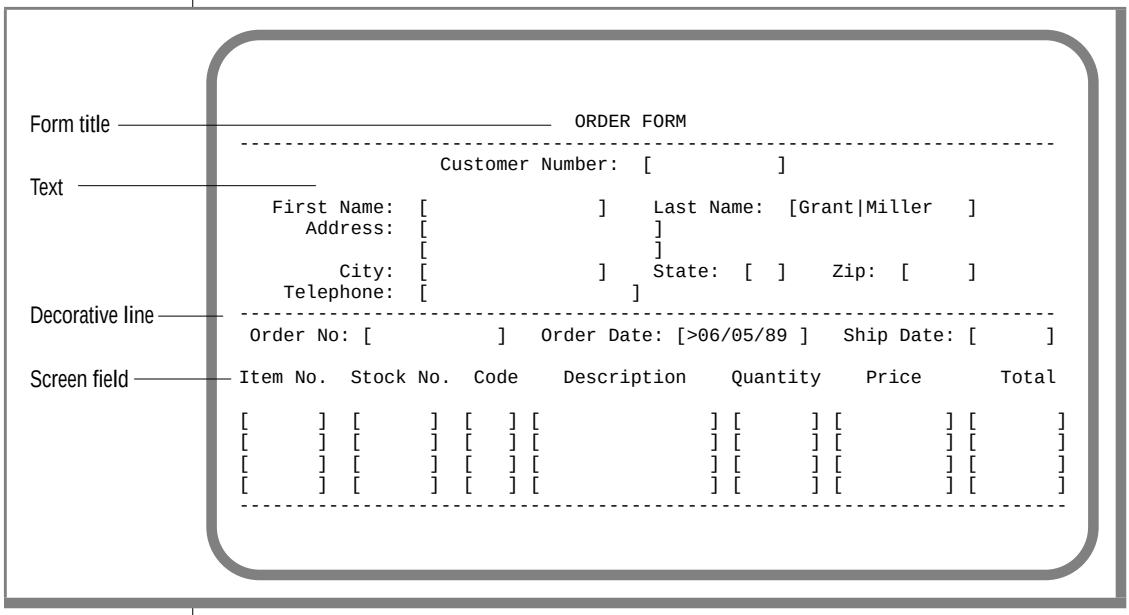
A *screen form* is a 4GL display in which the user can view, enter, or edit data. Chapter 6, "Screen Forms," describes how to create screen forms.

The following visual elements (described in greater detail in Chapter 6) can appear in a 4GL screen form:

- **Fields.** Also called form fields or screen fields, these areas are where the user enters or edits data, or the 4GL program displays a value.
- **Field delimiters.** Fields are usually enclosed within brackets.
- **Screen records.** These are logically related sets of fields.
- **Screen arrays.** These are scrollable arrays of fields or records.
- **Decorative rectangles.** These can ornament the form.
- **Text.** Anything else in the form is called text. Text always appears while the form is visible.

Text can include labels and titles, as in Figure 2-4.

Figure 2-4
Visual Elements on a Screen Form



Visual Cursors

4GL marks the user's current location (if any) in the current menu, form, or field with a *visual cursor.* Usually, each of these is simply called the cursor:

- **Menu cursor.** Reverse video marks the option chosen if you press RETURN.
- **Field cursor.** This pipe symbol (|) marks the current character position in the current field.

Field Attributes

Several 4GL statements can set display attributes (as described in "ATTRIBUTE Clause" on page 3-96). A form specification file can also specify *field attributes*. These optional descriptors can control the display when the cursor is in the field, or can supply or restrict field values during data entry. Field attributes can have effects like these:

- They control cursor movement among fields.
- They set validation and default value field attributes.
- They set formatting attributes or automatically invoke a multiple-line editor for character data, or an external editor to view or modify TEXT or BYTE data.
- They set screen display color and intensity attributes.

Reserved Lines

On the 4GL screen, certain lines are reserved for output from specific 4GL statements or from other sources. By default, these *reserved lines* appear in the following positions on the 4GL screen:

- **Menu** line. Line 1 displays the menu title and options list from MENU.
- **Prompt line.** Line 1 also displays text specified by the PROMPT statement.
- **Menu Help line.** Line 2 displays text describing MENU options. You cannot reposition this line independently of the Menu line.
- **Message line.** Line 2 also displays text from the MESSAGE statement. You can reposition this line with the OPTIONS statement.
- **Form line.** Line 3 begins a form display when DISPLAY FORM executes.
- **Comment line.** The next-to-last line of the 4GL screen (or the last line in a 4GL window) displays COMMENTS attribute messages.
- **Error line.** The last line of the 4GL screen displays output from the ERROR statement.

If you display the form in a named 4GL window, these default values apply to that window, rather than to the 4GL screen, except for the Error line. (The position of the Error line is defined relative to the entire screen, rather than to any 4GL window.)

The OPTIONS statement can change these default positions for all the 4GL windows of your application. The OPEN WINDOW statement can reposition all of these reserved lines (except the Error line) within the specified 4GL window. (These 4GL statements are described in detail in Chapter 4.)

4GL Windows

A 4GL *window* is a named rectangular area on the 4GL screen. When a 4GL program starts, the entire 4GL screen is the current window. Some 4GL statements can reference this default window as SCREEN. The OPEN WINDOW statement can create additional 4GL windows, dimensions, position, and attributes of each window. In DBMS applications that perform various tasks, displaying distinct activities in different 4GL windows is a good design strategy.

No 4GL window can display more than one 4GL form. The CURRENT WINDOW statement can transfer control from one 4GL window to another.

The Current Window

4GL maintains a list of all open 4GL windows, called the *window stack*. When you open a new 4GL window, it is added to the top of this stack. The window at the top of the stack is the current window.

The current 4GL window is always completely visible, and can obscure all or part of other windows. When you specify a new current window, 4GL adjusts the window stack by moving that window to the top, and closing any gap in the stack left by the window. When you close a window, 4GL removes that window from the window stack. The top-most window among those that remain on the screen becomes the current window. All this takes place within the 4GL *screen*.

All input and output is done in the current window. If that window contains a screen form, the form becomes the *current form*. The DISPLAY ARRAY, INPUT, INPUT ARRAY, and MENU statements all run in the current window. If a user displays a form in another window from within one of these statements (for example, by activating an ON KEY block), the window containing the new form becomes the current window. When the enclosing statement resumes execution, the original window is restored as the current window.

Programs with multiple windows might need to switch to a different current window unconditionally, so that input and output occur in the appropriate window. The CURRENT WINDOW statement makes a specified window (or SCREEN) the current window. When a window becomes the current window, 4GL restores its values for the positions of the Prompt, Menu, Message, Form, and Comment lines.

On-Line Help

4GL includes two distinct facilities for displaying help messages:

- **Development help.** The developer can request help from the Programmer's Environment regarding features of the 4GL language. Use the Help key (typically CONTROL-W) to display help on the currently selected menu option.
- **Runtime help.** The user of a 4GL application can display programmer-defined help messages.

Runtime help is displayed when the user presses a designated Help key (CONTROL-W). The 4GL statements that can include a HELP clause are these:

- CONSTRUCT (during a query by example)
- INPUT and INPUT ARRAY (during data entry)
- MENU (for each menu option)
- PROMPT (when the user must supply keyboard input)

The HELP clause specifies a single help message for the entire 4GL statement. To provide field-level help in these interactive statements, you can use an ON KEY clause with the INFIELD( ) operator and the SHOW_HELP( ) function, which are described in Chapter 5, "Built-In Functions and Operators."

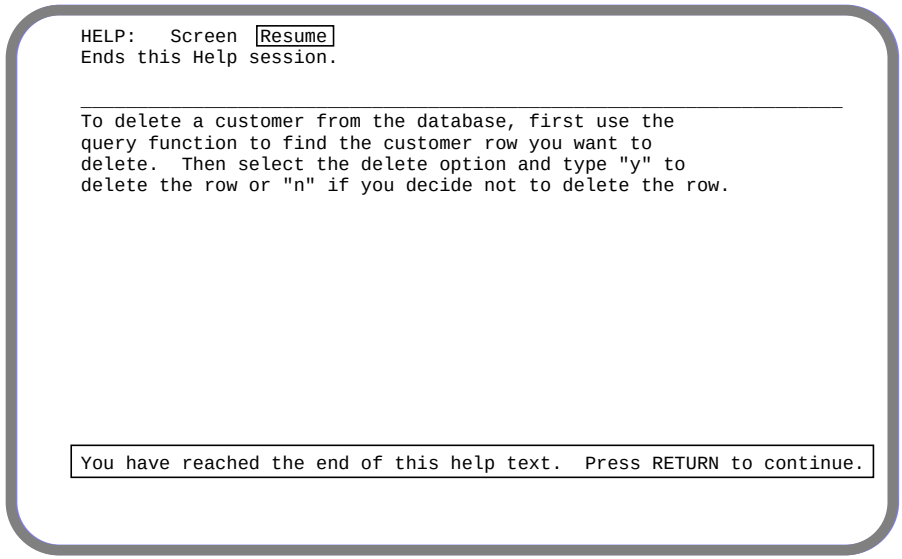
The Help Key and the Message Compiler

By default, CONTROL-W acts as the Help key. To specify a nondefault Help key, or to identify a file that contains help messages, use the OPTIONS statement. If you specify a file of help messages, 4GL displays the messages in the Help window.

While the MENU statement of 4GL is executing, the question (?) mark key also acts as the Help key.

The Help Window

Here is an example of a typical Help window display.



When the user presses the Help key in a context for which you have prepared a help message, that message appears in a Help window. The 4GL screen is hidden while this window is open.

The Help window has a 4GL ring menu containing **Screen** and **Resume** menu options. **Screen** displays the next page of help text. **Resume** closes the Help window and redisplays the 4GL screen.

You must create these help messages and store them in an ASCII file. Each message begins with a unique whole number that has an absolute value no greater than 32,767 and is prefixed by a period (.). Statements of 4GL can reference a help message by specifying its number in a HELP clause.
You must compile help messages from the ASCII source file to a runtime format by using the **mkmessage** utility. For more information about creating help messages, see "The mkmessage Utility" on page B-2.

The Help window persists until the user closes it. The user can dismiss the Help window by using the **Resume** menu option or by pressing RETURN.

Nested and Recursive Statements

The form-based 4GL statements CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, and PROMPT are collectively referred to as input control block (ICB) statements, because they use a common underlying data structure called the input control block. 4GL supports nested and recursive ICB statements in which the parent and the child ICB statements use the same screen form for accepting and displaying data. In the case of recursive ICB statements, the parent and the child statements can be the same:

- *Nested input* occurs when two ICB statements (such as INPUT, INPUT ARRAY, DISPLAY ARRAY, or CONSTRUCT) operate concurrently on the same fields of the same screen form. *Concurrently*, in this context, means that one ICB statement is suspended while using the same form field that another ICB statement is using.
- *Recursive input* occurs when two instances of the same ICB statement operate concurrently, and a BEFORE or AFTER or ON KEY control block is activated that invokes a function that executes the same ICB statement recursively, re-executing that ICB statement. It does not matter whether the recursion is direct (the function calls itself) or indirect (the function calls another function, which in turn calls the first function again).

Opening and closing screen forms precludes nesting and recursion of ICB statements, which can only occur when forms are not being opened and closed while the outer ICB statement is still processing.

Recursive and nested input imply the use of the same 4GL window. Changing the current window inherently changes the form (or the instance of the form) that is currently in use.

4GL supports both direct and indirect nesting of the ICB statements; the level of nesting is limited only by the availability of the resources. *Direct nesting* involves embedding the child ICB statement in a control block, such as ON KEY, BEFORE FIELD, AFTER FIELD, and so forth, of the parent ICB statement.

The following code example illustrates direct nesting of INPUT statements:

```
MAIN
    DEFINE r1, r2 RECORD
       f1, f2, f3 CHAR(30)
              END RECORD
   DATABASE nestedinputDB
   DEFER INTERRUPT
   OPEN FORM nest_form FROM "nested_input"
   DISPLAY FORM nest_form
    LET INT_FLAG = FALSE
   IF NOT INT_FLAG THEN
    INPUT BY NAME r1.* WITHOUT DEFAULTS   --Parent INPUT stmt
   BEFORE INPUT
   MESSAGE "BEFORE INPUT STATEMENT 1" SLEEP 1
   BEFORE FIELD f1
   MESSAGE "Input Statement 1 -- BF1" SLEEP 1
   AFTER FIELD f2
   MESSAGE "Input Statement 1 -- AF1" SLEEP 1
   --Child INPUT Statement
    INPUT BY NAME r2.* WITHOUT DEFAULTS  --Child INPUT stmt
   BEFORE INPUT
   MESSAGE "BEFORE INPUT STATEMENT 2" SLEEP 1
   AFTER INPUT
   MESSAGE "AFTER INPUT STATEMENT 2" SLEEP 1
   ON KEY (CONTROL-W)
   MESSAGE "IP2: HELP IS NOT AVAILABLE" SLEEP 1
   END INPUT   --End of Child INPUT stmt
   LET r2.f1 = "Child INPUT Statement -- ", r2.f1 CLIPPED
   INSERT INTO table_2 VALUES(r2.*)
   AFTER INPUT
   MESSAGE "AFTER INPUT STATEMENT 1"  SLEEP 1
   ON KEY (CONTROL-W)
   MESSAGE "IP1: HELP IS NOT AVAILABLE" SLEEP 1
   END INPUT   --End of Parent INPUT stmt
   END IF
   LET r.f1 = "Parent INPUT Statement -- ", r1.f1 CLIPPED
   INSERT INTO table_1 VALUES(r1.*)
    MESSAGE "Input Statement  -- complete" SLEEP 1
END MAIN
```

In this example of nested INPUT statements, the parent and the child statements use the same form for entering data. After the user enters data in fields **f1** and **f2** for the parent INPUT statement, control transfers to the child INPUT statement. Once the child INPUT statement is executed, data entered for the child is inserted in **table_2**. Then control returns to the parent and fields **f1** and **f2** are restored to their original values.

Indirect nesting invokes a function that contains the child ICB statement, from the parent ICB statement. Once again, the parent and the child ICB statements use the same form for data entry.

The following code illustrates indirect nesting of INPUT statements:

```
MAIN
   DEFINE r1 RECORD
   f1, f2, f3 CHAR(30)
   END RECORD
   DATABASE nestedinputDB
   DEFER INTERRUPT
   OPEN FORM nest_form FROM "nested_input"
   DISPLAY FORM nest_form
   LET INT_FLAG = FALSE
      IF NOT INT_FLAG THEN
         INPUT BY NAME r1.* WITHOUT DEFAULTS--Parent INPUT stmt
         BEFORE INPUT
         MESSAGE "BEFORE INPUT STATEMENT 1" SLEEP 1
         BEFORE FIELD f1
         MESSAGE "Input Statement 1 -- BF1" SLEEP 1
         AFTER FIELD f2
         MESSAGE "Input Statement 1 -- AF1" SLEEP 1
          --Indirect Nesting
         CALL child_inputstmt()
         AFTER INPUT
         MESSAGE "AFTER INPUT STATEMENT 1"  SLEEP 1
          ON KEY (CONTROL-W)
         MESSAGE "IP1: HELP IS NOT AVAILABLE" SLEEP 1
         END INPUT    --End of Parent INPUT stmt
      END IF
   LET r.f1 = "Parent INPUT Statement -- ", r1.f1 CLIPPED
   INSERT INTO table_1 VALUES(r1.*)
   MESSAGE "Input Statement  -- complete" SLEEP 1
END MAIN
FUNCTION child_inputstmt()
   DEFINE r2 RECORD
   f1, f2, f3 CHAR(30)
   END RECORD
   LET INT_FLAG = FALSE
      IF NOT INT_FLAG THEN
         INPUT BY NAME r2.* WITHOUT DEFAULTS  --Child INPUT stmt
         BEFORE INPUT
         MESSAGE "BEFORE INPUT STATEMENT 2" SLEEP 1
         AFTER INPUT
         MESSAGE "AFTER INPUT STATEMENT 2" SLEEP 1
          ON KEY (CONTROL-W)
         MESSAGE "IP2: HELP IS NOT AVAILABLE" SLEEP 1
         END INPUT   --End of Child INPUT stmt
      END IF
   LET r2.f1 = "Child INPUT Statement -- ", r2.f1 CLIPPED
   INSERT INTO table_2 VALUES(r2.*)
END FUNCTION
```

Performing heterogeneous nesting is valid where the parent and the child statements are entirely different. For example, where CONSTRUCT is the parent statement, INPUT can be the child statement.

The recursive ICB statements feature provides extra flexibility to 4GL programmers. The following example illustrates the use of the recursive INPUT statement:

```
MAIN
   DEFINE r RECORD
   f1, f2, f3 CHAR(30)
   END RECORD
   DEFINE z INTEGER
   DATABASE recinputDB
   DEFER INTERRUPT
   OPEN FORM recurs_form FROM "recursive_input"
   DISPLAY FORM recurs_form
   LET z = 0
   CALL recursive_input(z, r)
END MAIN
FUNCTION recursive_input(z1, r1)
   DEFINE r1 RECORD
   f1, f2, f3 CHAR(30)
   END RECORD
   DEFINE z1 INTEGER
   LET INT_FLAG = FALSE
   LET z1 = z1 + 1
   IF z1 > 3 THEN
   RETURN
   END IF
   MESSAGE "Recursive Cycle: ", z1 SLEEP 1
      IF NOT INT_FLAG THEN
         INPUT BY NAME r1.* WITHOUT DEFAULTS
         BEFORE INPUT
         MESSAGE "BEFORE INPUT STATEMENT 1" SLEEP 1
         BEFORE FIELD f1
         MESSAGE "Input Statement 1 -- BF1" SLEEP 1
         AFTER FIELD f2
         MESSAGE "Input Statement 1 -- AF1" SLEEP 1
         CALL recursive_input(z1, r1)  --Recursive call
         AFTER INPUT
         MESSAGE "AFTER INPUT STATEMENT 1"  SLEEP 1
         ON KEY (CONTROL-W)
         MESSAGE "IP1: HELP IS NOT AVAILABLE" SLEEP 1
         END INPUT
      END IF
   LET r.f1 = "Recursive Cycle -- ", z1, r1.f1 CLIPPED
   INSERT INTO table_1 VALUES(r1.*)
   MESSAGE "Input Statement  -- complete" SLEEP 1
END FUNCTION
```

For every recursive invocation of the function **recursive_input( )**, the same form is used for data entry. A recursive call is made after you enter data in fields **f1** and **f2** for each invocation. Before making a recursive call, the context is stored in dynamically allocated variables and pushed onto a stack. After the terminating condition for recursion is satisfied, the context is popped out of the stack, and the buffers for fields **f1** and **f2** revert to their original values.

It is valid to combine all the different types of nesting, such as direct, indirect, heterogeneous, and homogeneous, with recursive ICB statements. To avoid anomalous behavior, it is advisable to recompile and relink pre-6.x 4GL applications that use ICB-related statements with current 4GL software.

Early Exits from Nested and Recursive Operations

These nested input features allow some operations that can cause 4GL to clean up incorrectly during INPUT, INPUT ARRAY, DISPLAY ARRAY, CONSTRUCT, MENU, and PROMPT statements, and in FOREACH loops.

For any nested statements, an early exit from inside one of the inner statements to one of the outer statements, or a RETURN, means that 4GL does not clean up any statement except the innermost statement.

The following code example, though of no practical use, illustrates nested INPUT statements. There are nine levels of nested statements in the code.

```
MAIN
   CALL f()
END MAIN
FUNCTION f()
   DEFINE s CHAR(300)
   DEFINE y INTEGER
   DEFINE i INTEGER
   DEFINE t INTEGER
   DEFINE a ARRAY[10] OF INTEGER
   DECLARE c CURSOR FOR
   SELECT Tabid FROM Systables
   OPEN WINDOW w AT 1, 1 WITH FORM "xxx"
   LET y = 0
   FOREACH c INTO t
      FOR i = 1 TO 10
      WHILE y < 1000
      MENU "ABCDEF"
      BEFORE MENU
      HIDE OPTION "B"
      COMMAND "A" "Absolutely"
      SHOW OPTION "B"
      IF a[1] THEN EXIT     MENU END IF
      IF a[1] THEN CONTINUE MENU END IF
      NEXT OPTION "E"
      COMMAND "B" "Beautiful"
      MESSAGE "Thank you"
      COMMAND "C" "Colourful"
      MESSAGE "Thank you"
      COMMAND "D" "Delicious"
      MESSAGE "Thank you"
      COMMAND "E" "Exit"
      EXIT MENU
      COMMAND "F"
      MENU "XYZ"
      COMMAND "X"
      EXIT MENU
      COMMAND "Y"
      INPUT BY NAME y WITHOUT DEFAULTS
      AFTER FIELD y
      IF a[1] THEN EXIT     FOR       END IF
      IF a[1] THEN CONTINUE FOR       END IF
      IF a[1] THEN EXIT     FOREACH   END IF
      IF a[1] THEN CONTINUE FOREACH   END IF
      IF a[1] THEN EXIT     WHILE     END IF
      IF a[1] THEN CONTINUE WHILE     END IF
      IF a[1] THEN RETURN             END IF
      IF a[1] THEN EXIT     MENU      END IF
      IF a[1] THEN CONTINUE MENU      END IF
      IF a[1] THEN EXIT     INPUT     END IF
      IF a[1] THEN CONTINUE INPUT     END IF
      IF a[1] THEN GOTO End_Label     END IF
```

```
   IF a[1] THEN GOTO Mid_Label     END IF
   CONSTRUCT BY NAME s ON y
   AFTER FIELD y
   IF a[1] THEN EXIT     FOR       END IF
   IF a[1] THEN CONTINUE FOR       END IF
   IF a[1] THEN EXIT     FOREACH   END IF
   IF a[1] THEN CONTINUE FOREACH   END IF
   IF a[1] THEN EXIT     WHILE     END IF
   IF a[1] THEN CONTINUE WHILE     END IF
   IF a[1] THEN RETURN             END IF
   IF a[1] THEN EXIT     MENU      END IF
   IF a[1] THEN CONTINUE MENU      END IF
-- EXIT INPUT is not allowed by the compiler (error 4488)
-- IF a[1] THEN EXIT     INPUT     END IF
-- CONTINUE INPUT is not allowed by the compiler
-- (error 4488)
-- IF a[1] THEN CONTINUE INPUT     END IF
   IF a[1] THEN EXIT     CONSTRUCT END IF
   IF a[1] THEN CONTINUE CONSTRUCT END IF
   IF a[1] THEN GOTO End_Label     END IF
IF a[1] THEN GOTO Mid_Label     END IF
   CALL SET_COUNT(3)
   DISPLAY ARRAY a TO a.*
      ON KEY (F3)
      IF a[1] THEN EXIT     FOR       END IF
      IF a[1] THEN CONTINUE FOR       END IF
      IF a[1] THEN EXIT     FOREACH   END IF
      IF a[1] THEN CONTINUE FOREACH   END IF
      IF a[1] THEN EXIT     WHILE     END IF
      IF a[1] THEN CONTINUE WHILE     END IF
      IF a[1] THEN RETURN             END IF
      IF a[1] THEN EXIT     MENU      END IF
      IF a[1] THEN CONTINUE MENU      END IF
      IF a[1] THEN EXIT     DISPLAY   END IF
-- CONTINUE DISPLAY is not allowed by the compiler
-- IF a[1] THEN CONTINUE DISPLAY   END IF
-- EXIT INPUT is not allowed by the compiler (error 4488)
-- IF a[1] THEN EXIT     INPUT     END IF
-- CONTINUE INPUT is not allowed by the compiler (error
-- 4488)
-- IF a[1] THEN CONTINUE INPUT     END IF
-- EXIT CONSTRUCT is not allowed by the compiler (error
-- 4488)
-- IF a[1] THEN EXIT     CONSTRUCT END IF
-- CONTINUE CONSTRUCT is not allowed by the compiler
-- (error 4488)
-- IF a[1] THEN CONTINUE CONSTRUCT END IF
      IF a[1] THEN GOTO End_Label     END IF
      IF a[1] THEN GOTO Mid_Label     END IF
      INPUT ARRAY a FROM a.*
      AFTER FIELD y
      IF a[1] THEN EXIT     FOR   END IF
      IF a[1] THEN CONTINUE FOR   END IF
      IF a[1] THEN EXIT     FOREACH END IF
```

```
            IF a[1] THEN CONTINUE FOREACH END IF
            IF a[1] THEN EXIT     WHILE END IF
            IF a[1] THEN CONTINUE WHILE END IF
            IF a[1] THEN RETURN         END IF
            IF a[1] THEN EXIT     MENU  END IF
            IF a[1] THEN CONTINUE MENU  END IF
            IF a[1] THEN EXIT     INPUT END IF
            IF a[1] THEN CONTINUE INPUT END IF
      -- EXIT DISPLAY *is* allowed by the compiler (despite
      -- error 4488)
            IF a[1] THEN EXIT     DISPLAY END IF
      -- CONTINUE DISPLAY is not allowed by the compiler
      -- IF a[1] THEN CONTINUE DISPLAY END IF
      -- EXIT CONSTRUCT is not allowed by the compiler (error
      -- 4488)
      -- IF a[1] THEN EXIT  CONSTRUCT END IF
      -- CONTINUE CONSTRUCT is not allowed by the compiler
      -- (error 4488)
      -- IF a[1] THEN CONTINUE CONSTRUCT END IF
            IF a[1] THEN GOTO End_Label END IF
            IF a[1] THEN GOTO Mid_Label END IF
            LABEL Mid_label:
            MESSAGE "You got here?  How?"
            NEXT FIELD y
         END INPUT
      END DISPLAY
      END CONSTRUCT
      END INPUT
      COMMAND "Z"
      MESSAGE "Sucker!"
      CONTINUE MENU
      END MENU
      END MENU
      END WHILE
      END FOR
   END FOREACH
   LET y = 0

       LABEL End_label:
           CLOSE WINDOW w

   END FUNCTION
```

This example illustrates some problems that early exits from nested ICB and MENU statements can cause. For example, when EXIT FOREACH executes from within an INPUT statement that itself is nested within two MENU statements, a FOR loop, and a WHILE loop (as in the first EXIT FOREACH statement of the previous example), then the intervening menus (**ABCDEF** and **XYZ**) are not cleaned up correctly. Here the INPUT statement itself is handled correctly, however, and the database cursor in the FOREACH loop closes correctly.

Jump statements, including GOTO and WHENEVER...GOTO also are not dealt with properly. If this is a concern, do not use GOTO in contexts like this.

Exception Handling

4GL provides facilities for issuing compile-time and link-time detection of compilation errors, and for detection and handling of warnings and runtime errors and that occur during program execution.

Compile-Time Errors and Warnings

This manual describes 4GL language syntax, violations of which can cause compilation errors. Compilation and link-time errors can also result from improper settings of environment variables, or from invalid SQL syntax. These topics are described in the *Informix Guide to SQL* manuals.

If you do not understand an error or warning message, look up its numeric code with the **finderr** utility, or look in the on-line documentation of your Informix database server, or in your network software manual.

Runtime Errors and Warnings

INFORMIX-4GL Concepts and Use provides an overview of runtime errors and the 4GL facilities for handling them. (See also the built-in STARTLOG( ) and ERRORLOG( ) functions described in Chapter 5, "Built-In Functions and Operators,"of the present manual; these can support automatic or explicit logging of runtime errors in a file.)

By default, runtime errors and warnings are written to **stderr**. (Versions of 4GL earlier than 6.0 wrote runtime errors and warnings to **stdout**.)

There are several types of runtime errors and warnings:

- SQL errors, warnings, or NOTFOUND conditions that the database server detects and records in the SQLCA area; see "Error Handling with SQLCA" on page 2-45
- SPL (Stored Procedure Language) errors that the database server reports during execution of a stored procedure; see *Informix Guide to SQL: Syntax*
- Interrupt, Quit, or other signals from the user or from other sources
- Runtime errors and warnings that 4GL issues

For errors that occur in stored procedures, you can use the ON EXCEPTION statement of SPL to trap errors within the procedure. You can use the DEFER statement of 4GL to trap and handle Interrupt and Quit signals, and you can use WHENEVER to trap and handle SQL and 4GL errors and warnings.

The WHENEVER statement can control the processing of exceptional conditions of several kinds: SQL warnings, SQL end-of-data errors, errors in SQL or screen I/O operations, or errors in evaluating 4GL expressions.

The DEFER statement can instruct 4GL not to terminate the program when a user presses the Interrupt or Quit key. The DEFER statement has dynamic scope, as opposed to the lexical scope of WHENEVER. When Quit or Interrupt is deferred, the signal is ignored globally (that is, in all modules).

See the descriptions of DEFER and WHENEVER in Chapter 4 for details of how to use these statements to handle signals, errors, and warnings.

Normal and AnyError Scope

The term *error scope* refers to whether the scope of error handling includes expression errors. Normal error scope includes SQL errors, validation errors discovered by the VALIDATE statement, and screen interaction statement errors, but it ignores expression errors. Expression errors neither initiate any action by a WHENEVER statement, nor do they set the **status** variable when Normal error scope is in effect.

In contrast, AnyError error scope means that expression errors also activate error logic, and **status** is set to an appropriate negative value after the statement that produced the expression error. This was the only error scope available with Version 4.0 of 4GL. In the current release, however, you must request it explicitly with the WHENEVER ANY ERROR directive or with the -anyerr flag when you compile or in the RDS runner or Debugger command line.

For maximum ease in preventing, locating, and correcting expression errors, Informix recommends that you write all new 4GL code to work under AnyError error scope. With 4GL programs that you compile to C, you can achieve this automatically by setting the **C4GLFLAGS** environment variable to include -anyerr. With RDS, you can achieve this by setting the **FGLPCFLAGS** environment variable to include **-anyerr**.

A Taxonomy of Runtime Errors

The WHENEVER statement classifies 4GL runtime errors into four disjunct categories, as shown in Figure 2-5.

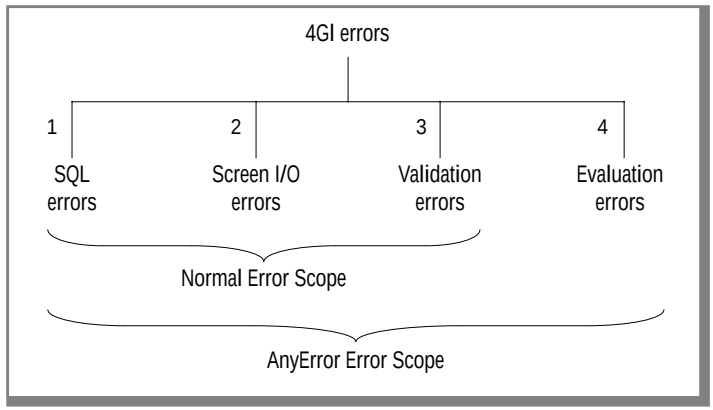


Figure 2-5
Categories of Runtime Errors

Here errors of type 1 occur on the database server, and reset the built-in **SQLCA**.**SQLCODE** variable to a negative value. (See "Error Handling with SQLCA" on page 2-45 for a description of **SQLCA**.**SQLCODE**.)

To trap errors of types 1, 2, and 3, which correspond to Normal error scope, specify WHENEVER ERROR. As Figure 2-5 indicates, Normal error scope is a logical subset of AnyError error scope.

To use AnyError error scope, which can also trap errors of type 4, you must specify WHENEVER ANY ERROR, or else compile with the **-anyerr** option. Errors of type 4 can include arithmetic, Boolean, or conversion errors that occur when 4GL evaluates an expression.

Important: *Earlier releases of 4GL supported fatal (or "untrappable") runtime errors, which could not be trapped by WHENEVER statements, but such errors are not a feature of this release. There are no 4GL errors outside AnyError error scope.*

4GL runtime errors are included in *Informix Error Messages* in Answers OnLine.

ANSI

Default Error Behavior and ANSI Compliance

The default responses to error conditions differ between the ANSI-compliant method and the non-ANSI-compliant method as follows:

1. If ANSI compliance is requested and no WHENEVER statement is in effect, the default action after an error is CONTINUE.

   ANSI compliance is in effect if any of the following conditions exists:

   - There is a default compile-time database that is ANSI-compliant.
   - The **-ansi** compilation flag is specified.
   - The **DBANSIWARN** environment variable is set.

   In releases of RDS earlier than 6.0, the default error action was STOP (rather than CONTINUE) for the last two of these three conditions.

2. If neither ANSI compliance nor any WHENEVER statement is in effect, the following factors determine the default error action:

   - If the **-anyerr** flag is used, the default action is STOP.
   - If the **-anyerr** flag is not used, the default action after an expression or data type conversion error is CONTINUE. After other categories of errors, it is STOP.

The error behavior depends on the database that the nonprocedural DATABASE statement references when you compile the program. If you compile with a non-ANSI-compliant default database, but run with an ANSI-compliant current database, the error behavior is as if the database were not an ANSI-compliant database. The converse also applies: if you compile against an ANSI-compliant database but run against a non-ANSI-compliant database, then the error behavior is the same as if the runtime database were ANSI-compliant.

If you compile part of the application against an ANSI-compliant database and part of it against a non-ANSI-compliant database, then those parts of the application compiled against the ANSI-compliant database have the default error action of CONTINUE and those parts compiled against the non-ANSI-compliant database have the default error action of STOP.

Changes to 4GL Error Handling

In this version of 4GL (and in contrast to versions earlier than 6.01), the following error handling features are in effect:

- The **status** variable is set for expression errors only if AnyError error scope is in effect. AnyError behavior is no longer the default for RDS.

  This is a significant backward-compatibility concern for RDS users who do not explicitly use AnyError error scope. Use the **-anyerr** flag when compiling your p-code modules to prevent unexpected failure of expression errors to set **status** (and recompile other p-code modules in the same program, for consistency within each program).
- Error and warning messages are no longer written to the UNIX standard error file. They are written to the 4GL error log file only.

This change primarily affects 4GL programs that are compiled to C and that use Normal error scope (because most RDS expression or conversion errors have used **-anyerr** behavior, regardless of the requested error scope). Use 4GL error logs in order to recognize and isolate errors.

Error Handling with SQLCA

Proper database management requires that all logical sequences of statements that modify the database continue successfully to completion. Suppose, for example, that an UPDATE operation on a customer account shows a reduction of $100.00 in the payable balance, but for some reason the next step, an UPDATE of the cash balance, fails; now your books are out of balance. It is prudent to verify that every SQL statement executes as you anticipated. 4GL provides two ways to do this:

- The global variable **status**, which can indicate errors, both from SQL statements and from other 4GL statements
- The global record **SQLCA**, which indicates the success of SQL statements, and provides other information about actions of the database server

Compared to **status**, the **SQLCA.SQLCODE** variable is typically easier to use for monitoring the success or failure of SQL statements because it ignores exceptional conditions that might be encountered in other 4GL statements.

4GL returns a result code into the **SQLCA** record after executing every SQL statement. Because it is automatically defined, you do not need to (and must not) declare the **SQLCA** record, which has this structure:

```
DEFINE  SQLCA  RECORD
   SQLCODE INTEGER,
   SQLERRM CHAR(71),
   SQLERRP CHAR(8),
   SQLERRD ARRAY [6]  OF  INTEGER,
   SQLAWARN CHAR (8)
END  RECORD
```

The members of **SQLCA** have the following semantics:

SQLCODE indicates the result of any SQL statement. It is set to zero for a successful execution, and to NOTFOUND (= 100) for a successfully executed query that returns zero rows, or for a FETCH that seeks beyond the end of the current active set. **SQLCODE** is negative after an unsuccessful SQL operation.

At runtime, 4GL sets the variable **status** equal to **SQLCODE** after each SQL statement. (See also the description of the ANY keyword of WHENEVER in "A Taxonomy of Runtime Errors" on page 2-42.) The **finderr** utility can provide explanations of SQL and 4GL error codes.

SQLERRM is not used at this time.

SQLERRP is not used at this time.

SQLERRD is an array of six variables of data type INTEGER:

- **SQLERRD[1]** is not used at this time.
- **SQLERRD[2]** is a SERIAL value returned or ISAM error code.
- **SQLERRD[3]** is the number of rows inserted or updated.
- **SQLERRD[4]** is the estimated CPU cost for query.
- **SQLERRD[5]** is the offset of the error into the SQL statement.
- **SQLERRD[6]** is the row ID of the last row that was processed; whether it was returned is server dependent.

SQLAWARN is an 8-byte string whose characters signal any warnings (as opposed to errors) after any SQL statement executes. All characters are blank if no problems were detected.

- **SQLAWARN[1]** is set to `W` if any of the other warning characters were set to `W`. If **SQLAWARN[1]** is blank, you do not have to check the other warning characters.

SQLAWARN[2] is set to W if one or more values were truncated to fit into a CHAR variable, or if a DATABASE statement selected a database with transactions.

SQLAWARN[3] is set to W if an aggregate like SUM( ), AVG( ), MAX( ), or MIN( ) encountered a null value in its evaluation, or if the DATABASE statement specified an ANSI/ISO-compliant database.

SQLAWARN[4] is set to W if a DATABASE statement selected an Informix Dynamic Server database, or when the number of items in the *select-list* of a SELECT clause is not the same as the number of program variables in the INTO clause. (The number of values returned by 4GL is the smaller of these two numbers.)

SQLAWARN[5] is set to W if float-to-decimal conversion is used.

SQLAWARN[6] is set to W if your program executes an Informix extension to the ANSI/ISO standard for SQL syntax while the **DBASIWARN** variable is set, or after the **-ansi** compilation flag was used.

SQLAWARN[7] is set to W if a query skips a table fragment, or if the database and client have different locales.

SQLAWARN[8] is not used at present.

If a multi-row INSERT or UPDATE statement of SQL fails, then **SQLERRD[3]** is set to the number of rows that were processed before the error was detected. If a LOAD operation fails with error -846, however, **SQLERRD[3]** is always set to 1, regardless of how many rows (if any) were successfully inserted.

For a complete description of **SQLCA**, see the *Informix Guide to SQL: Tutorial* in your Informix database server documentation.

Chapter

3

Data Types and Expressions

In This Chapter

This chapter describes how 4GL programs represent data values. The first part of this chapter describes the 4GL data types; the latter part describes 4GL expressions, which can return specific values of these data types.

Data Values in 4GL Programs

This section identifies and describes the data types of 4GL. In general, data values in 4GL must be represented as some data type. A *data type* is a named category that describes what kind of information is being stored, and that implies what kinds of operations on the data value make sense.

In some cases, a value stored as one data type can be converted to another. "Data Type Conversion" on page 3-42 identifies the pairs of 4GL data types for which automatic conversion is supported and describes ways in which information can be lost or modified by such conversion.

A related fundamental concept of 4GL is the notion of an expression. Just as a data type (such as INTEGER) characterizes a general category of values, an *expression* defines a specific data value. Later sections of this chapter describe how data values are represented in 4GL source code by expressions, and classify expressions according to the data type of the returned value.

Data Types of 4GL

You must declare a data type for each variable, FORMONLY field, formal argument of a function or report, or value returned by a function. Function and report arguments can be any 4GL data types except ARRAY.

The following data types are valid in declarations of program variables.

Data Type	Kinds of Values Stored
ARRAY OF *type*	Arrays of values of any other single 4GL data type
BYTE	Any kind of binary data, of length up to 2^{31} bytes
CHAR(*size*)	Character strings, of size up to 32,767 bytes in length
CHARACTER	This keyword is a synonym for CHAR
DATE	Points in time, specified as calendar dates
DATETIME	Points in time, specified as calendar dates and time-of-day
DEC	This keyword is a synonym for DECIMAL
DECIMAL(*p*,*s*)d	Fixed-point numbers, of a specified scale and precision
DECIMAL(*p*)	Floating-point numbers, of a specified precision
DOUBLE PRECISION	These keywords are a synonym for FLOAT
FLOAT	Floating-point numbers, of up to 32-digit precision
INT	This keyword is a synonym for INTEGER
INTEGER	Whole numbers, from -2,147,483,647 to +2,147,483,647
INTERVAL	Spans of time in years and months, or in smaller time units
MONEY	Currency amounts, with definable scale and precision
NCHAR(*size*)	Character strings, of size up to 32,767 bytes in length
NUMERIC	This keyword is a synonym for DECIMAL
NVARCHAR(*size*)	Character strings of varying length, for *size* ≤ 255 bytes

(1 of 2)

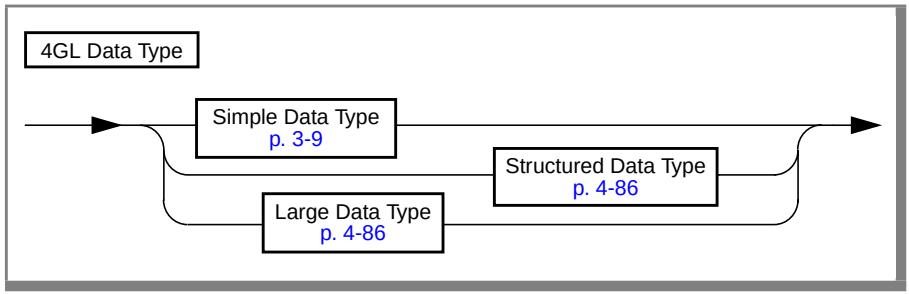
Data Type	Kinds of Values Stored
REAL	This keyword is a synonym for SMALLFLOAT
RECORD	Ordered sets of values, of any combination of data types
SMALLFLOAT	Floating-point numbers, of up to 16-digit precision
SMALLINT	Whole numbers, from -32,767 to +32,767
TEXT	Character strings of up to 2^{31} bytes
VARCHAR(*size*)	Character strings of varying length, for *size* ≤ 255 bytes

(2 of 2)

Except for ARRAY and RECORD, the 4GL data types correspond to built-in SQL data types of Informix database servers. The data types of 4GL approximate a superset of the SQL data types that 7.x Informix database servers recognize, but with the following restrictions:

- The SERIAL data type of SQL is not a 4GL data type. (Use the INTEGER data type to store SERIAL values from a database.) You cannot use the SERIAL keyword in 4GL statements that are not SQL statements.
- 4GL does not recognize the BITFIXED, BITVARYING, BLOB, BOOLEAN, CLOB, DISTINCT, INT8, LIST, LVARCHAR, MULTISET, OPAQUE, REFERENCE, ROW, SERIAL8, SET, or user-defined data types of Informix database servers.

Declarations of 4GL variables, formal arguments, and returned values use the following syntax to specify data types directly.

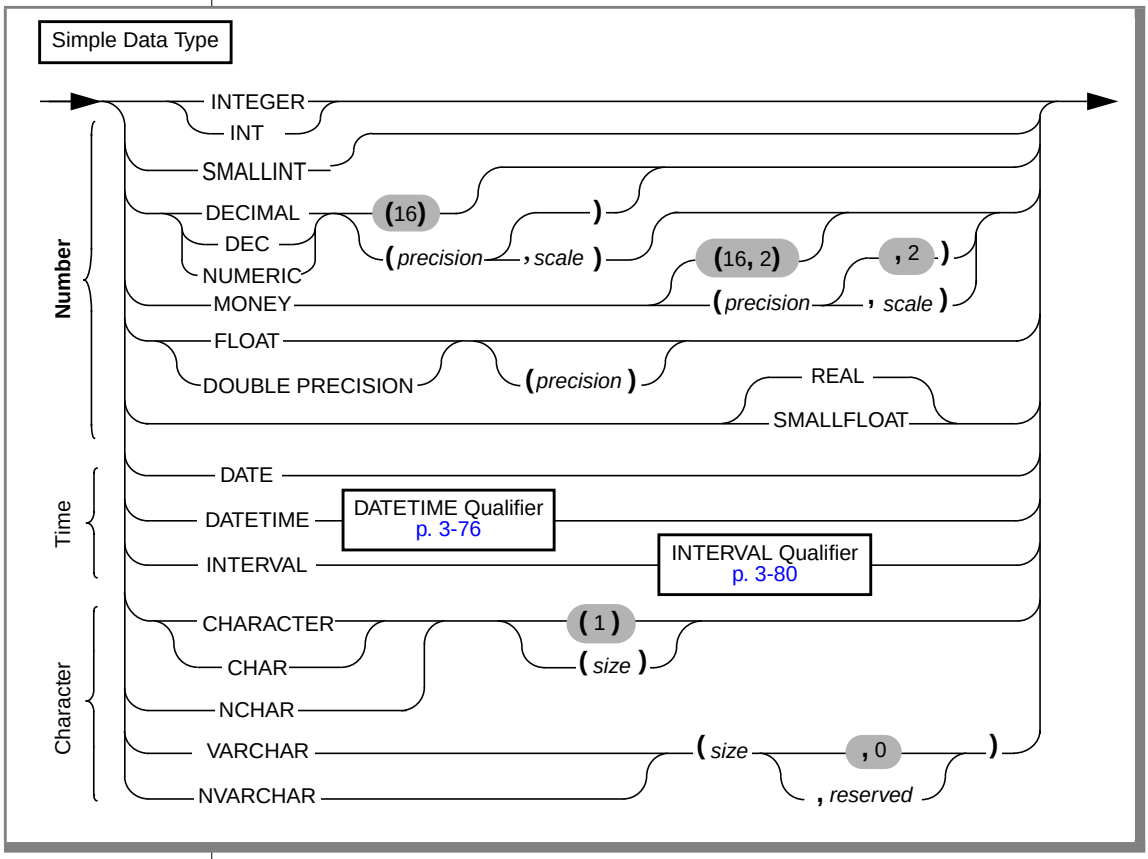


Declarations of variables can also use the LIKE keyword, as described in "Indirect Typing" on page 4-83. TEXT and BYTE are called the *large* data types; ARRAY and RECORD are called the *structured* data types. All other 4GL data types are *simple* data types.

Sections that follow describe the 4GL data types in each of these categories.

Simple Data Types

Each *simple* data type of 4GL can store a single value whose maximum storage requirement is specified or implied in the data type declaration.



Element	Description
precision	is the number of significant digits. For FLOAT, 1 ≤ *precision* ≤ 14. For DECIMAL and MONEY, the range is 1 ≤ *precision* ≤ 32.
reserved	is an SQL parameter not used by 4GL; 0 ≤ *reserved* ≤ *size* ≤ 255.
scale	is the number of digits (≤ 32) in the fractional part of the number, where 0 ≤ *scale* ≤ *precision*. The actual scale may be less than 32.
size	is the maximum number of bytes that the data type can store. For CHAR: 1 ≤ *size* ≤ 32,767; for VARCHAR: 1 ≤ *size* ≤ 255.

All parameters in data type declarations must be specified as literal integers. (For more information, see "Literal Integers" on page 3-65.) The *precision* of FLOAT and DOUBLE PRECISION data type declarations, and *reserved* in VARCHAR data type declarations, are accepted by 4GL but are ignored.

Each simple data type can be classified as a *number*, *time*, or *character* type. (*Numeric*, *chronological*, and *string* are synonyms for these categories.)

Number Data Types

4GL supports seven simple data types to store various kinds of numbers.

	Number Types	Description
Whole Number	SMALLINT	Integers, ranging from -32,767 to +32,767
	INTEGER, INT	Integers, ranging from -2,147,483,647 to +2,147,483,647 (that is, $-(2^{31}-1)$ to $(2^{31}-1)$)
Fixed-Point	DECIMAL (*p*, *s*), DEC(*p*, *s*), NUMERIC(*p*, *s*)	Fixed-point numbers, of scale *s* and precision *p*
	MONEY (*p*, *s*)	Currency values, of scale s and precision *p*
Floating-Point	DECIMAL (*p*), DEC(*p*), NUMERIC(*p*)	Floating-point numbers of precision *p* (but see "DECIMAL (p)" on page 3-24 for information about ANSI-compliant databases)
	FLOAT, DOUBLE PRECISION	Floating-point, double-precision numbers
	SMALLFLOAT, REAL	Floating-point, single-precision numbers

Character Data Types

4GL supports four simple data types for storing *character string* values.

Character Types	Description
CHAR (*size*), CHARACTER (*size*)	Strings of length *size*, for *size* up to 32,767 bytes
VARCHAR (*size*, *reserved*)	Strings of length ≤ *size*, for *size* ≤ 255 bytes
NCHAR (*size*)	Strings of length *size*, for *size* up to 32,767 bytes
NVARCHAR (*size*, *reserved*)	Strings of length ≤ *size*, for *size* ≤ 255 bytes

The TEXT data type (see "TEXT" on page 3-39) can store text strings of up to two gigabytes (= 2^{31} bytes). However, TEXT is not classified here as a character data type, because 4GL manipulates TEXT values in a different way from how CHAR, VARCHAR, NCHAR, or NVARCHAR values are processed.

Time Data Types

4GL supports three simple data types for values in chronological units. Two store *points in time*, and the third stores *spans of time* (positive or negative).

Time Types	Description
DATE	Calendar dates (*month*, *day*, *year*) with a fixed scale of days, in the range from January 1 of the year 1 up to December 31, 9999
DATETIME	Calendar dates (*year*, *month*, *day*) and time-of-day (*hour*, *minute*, *second*, and *fraction* of second), in the range of years 1 to 9999
INTERVAL	Spans of time, in years and months, or in smaller units

Structured Data Types

4GL supports two structured data types for storing sets of values.

Structured Types	Description
ARRAY	Arrays of up to 32,767 values (in up to three dimensions) of any single data type except ARRAY
RECORD	Sets of values of any data type, or any combination of data types

Large Data Types

Large data types store pointers to binary large objects up to 2^{31} bytes (two gigabytes) or up to a limit imposed by your system resources.

Large Types	Description
TEXT	Strings of printable characters
BYTE	Anything that can be digitized and stored on your system

Descriptions of the 4GL Data Types

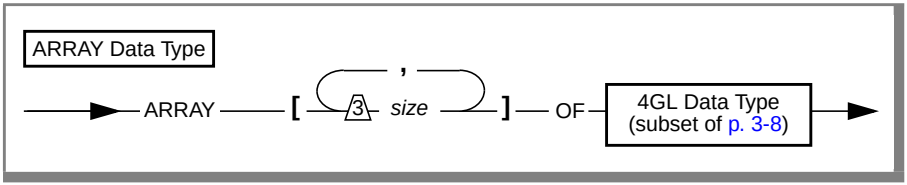
Sections that follow describe each of the 4GL data types, in alphabetical order.

GLS

Unless otherwise noted, descriptions of formats for data entry and display are for the default (U.S. English) locale. Some locales support other character sets and different entry and display formats for number, currency, and date values, as specified in the locale files and in environment variables. ♦

ARRAY

This structured data type stores a one-, two-, or three-dimensional array of variables, all of the same data type. These variables can be any 4GL data type except ARRAY. This is the syntax for declaring an array of variables:



Element	Description
size	is a positive integer number (ranging from 1 to 32,767) of elements within a dimension. Each dimension can have a different size.

Array Elements

Data types are here a subset of those listed in "Data Types of 4GL" on page 3-6, because ARRAY is not valid.

A variable in an array is called an *element*. You can use bracket ([ ]) symbols and comma-separated integers to reference a single element of an array. For example, if **xray** is the identifier of an ARRAY variable:

- **xray** [*i*] is the *i*th element of a one-dimensional array.
- **xray** [*i*, *j*] is the *j*th element in the *i*th row of a two-dimensional array.
- **xray** [*i*, *j*, *k*] is the *k*th element in the *j*th column of the *i*th row of a three-dimensional array.

Here the coordinates *i*, *j*, and *k* can be variables or other integer expressions that return positive whole numbers in the range 1 to 32,767. Within an SQL statement delimited by SQL…END SQL, an ARRAY host variable needs the dollar sign ($) prefix, but a variable used as an index to an array requires no prefix.

Some C compilers impose a limit lower than 32,767 on the number of array elements in a dimension, or on the total number. (For an array of two or more dimensions, this total is the product of all the declared *size* specifications.)

Because you cannot manipulate an array as a single unit, statements must refer to individual elements of an array. You reference a single array element by specifying its coordinates in each dimension of the array. For an array with two dimensions, for example, you must specify two coordinates to reference any element. In expressions, 4GL expands the identifier of a record to a list of its members but 4GL does not expand the name of an array to its elements.

You cannot pass an array to (or from) a function or report. You can, however, pass individual array elements. If these elements are records, 4GL expands each element to its members in the ordinary way. Similarly, if *record* is a RECORD having an ARRAY member, *record.** is not a valid argument to a function or to a report, nor (in this case) is *record.** a valid specification in a list of returned values.

Substrings of Character Array Elements

If *char_array* [*i*, *j*, *k*] is an element of an array of a character data type, you can use a comma-separated pair of integer expressions between trailing bracket ([ ]) symbols to specify a *substring* within its string value:

char_array **[***i***,** *j***,** *k***] [***m***,** *n***]**

Here $m \leq n$, for *m* and *n* expressions that return positive whole numbers to specify the respective positions of the first and last characters of a substring within the array element whose coordinates in *char_array* are *i*, *j*, and *k*.

BYTE

The BYTE data type stores any kind of binary data in a structureless byte stream. Binary data typically consists of saved spreadsheets, program load modules, digitized voice patterns, or anything that can be stored as digits.

The DEFINE statement can use the LIKE keyword to declare a 4GL variable like a BYTE column in an Informix database. The INFORMIX-SE database server does not support BYTE columns, but the 4GL application program can declare program variables of the BYTE data type.

The data type BYTE has no maximum size; the theoretical limit is 2^{31} bytes, but the practical limit is determined by the storage capacity of your system.

You can use a BYTE variable to store, retrieve, or update the contents of a BYTE database column, or to reference a file that you wish to display to users of the 4GL program through an external editor. After you declare a BYTE data type, you must use the LOCATE statement to specify the storage location.

When you select a BYTE column, you can assign all or part of it to a variable of type BYTE. You can use brackets ([ ]) and subscripts to reference only part of a BYTE value, as shown in the following example:

```
SELECT cat_picture [1,75] INTO cat_nip FROM catalog
      WHERE catalog_num = 10001
```

This statement reads the first 75 bytes of the **cat_picture** column of the row with the catalog number **10001**, and stores this data in the **cat_nip** variable. (Before running this query, the LOCATE statement must first be executed to allocate memory for the **cat_nip** BYTE variable.)

Restrictions on BYTE Variables

Built-in functions of 4GL cannot have BYTE arguments. Among the operators of 4GL, only IS NULL and IS NOT NULL can use BYTE variables as operands. The DISPLAY statement and PRINT statements cannot display BYTE values. Neither can the LET statement or INITIALIZE statement assign any value (except null) to a BYTE variable. The CALL and OUTPUT TO REPORT statements pass any BYTE arguments by reference, not by value.

A form field linked to a BYTE column (or a FORMONLY field of type BYTE) displays the character string <BYTE value> rather than actual data. You must use the PROGRAM attribute to display a BYTE value. No other field attributes (except COLOR) can reference the value of a BYTE field. The **upscol** utility cannot set default attributes or default values for a BYTE field.

CHAR

The CHAR data type of 4GL can store a character string, up to a number of bytes specified between parentheses in the *size* parameter of the data type declaration. These can be printable or non-printable characters (see "Non-Printable Characters" on page 3-71), as defined for the locale. The size can range from 1 to 32,767 bytes.

For example, the variable **keystrokes** in the following declaration can hold a character string of up to 78 bytes:

```
DEFINE keystrokes CHAR(78)
```

If the *size* is not specified, the resulting default CHAR data type can store only a single byte. In a form, you cannot specify the size of a FORMONLY CHAR field; the size defaults to the field length from the screen layout.

A character string returned by a function can contain up to 32,767 bytes. On INFORMIX-SE servers, the maximum data string length that a CHAR column can store is 32,511 bytes, but CHAR variables are not constrained by this.

When a value is passed between a CHAR variable and a CHAR database column, or between two CHAR variables, exactly *size* bytes of data are transferred, where *size* is the declared length of the 4GL variable or the database column that receives the string. If the length of the data string is shorter than *size*, the string is padded with trailing blanks (ASCII 32) to fill the declared size. If the string is longer than *size,* the stored value is truncated.

The ASCII 0 end-of-data character terminates every CHAR value; in most contexts, any subsequent characters in a data string cannot be retrieved from or entered into CHAR database columns. Use the CLIPPED operator of 4GL if you wish to convert CHAR variables with NULL values to empty strings.

To perform arithmetic on numbers stored in variables, use a *number* data type. CHAR variables can store digits, but you might not be able to use them in some calculations. Conversely, leading zeros (in some postal codes, for example) are stripped from values stored as number data types INTEGER or SMALLINT. To preserve leading zeros, store such values as CHAR data types.

GLS

In most locales, CHAR data types require one byte of storage per character, or *size* bytes for *size* characters. In some East Asian locales, however, more than one byte may be required to store an individual logical character, and some white space characters can occupy more than one byte of storage.

By default, when character strings (whether of data type CHAR or VARCHAR) are sorted by 4GL in a nondefault client locale, collation is in code-set order. If the **COLLATION** setting in the locale files defines a localized collation order, 4GL uses this order to sort CHAR and VARCHAR values, provided that the **DBNLS** environment variable is set to 1. (Even if the database locale defines a nondefault collation sequence, in most contexts the database server uses the code-set order to sort CHAR and VARCHAR column values.)

The database server generally sorts strings by code-set order, even if the **COLLATION** category of the locale defines a nondefault order, except for values in NCHAR or NVARCHAR columns. (**COLLATION** *functionally replaces the XPG3 category* **LC_COLLATE** in earlier Informix GLS products.) If the database and 4GL client have different locales, the order of collation can depend on whether 4GL or the database server performs the sort operation.

If the database has NCHAR or NVARCHAR columns, you must set the **DBNLS** environment variable to 1 if you want to store values from such columns in CHAR or VARCHAR variables of 4GL, or if you want to insert values of CHAR or VARCHAR variables into NCHAR or NVARCHAR database columns.

For more information on using non-ASCII values in CHAR columns, see Appendix E, "Developing Applications with Global Language Support." ♦

CHARACTER

The CHARACTER keyword is a synonym for CHAR.

DATE

The DATE data type stores calendar dates. The date value is stored internally as an integer that evaluates to the count of days since December 31, 1899. The default display format of a DATE value depends on the locale. In the default (U.S. English) locale, the default format is

```
mm/dd/yyyy
```

where *mm* is a month (1 to 12), *dd* is a day of the month (1 to 31 or less), and *yyyy* is a year (0001 to 9999). The **DBDATE** environment variable can change the separator or the default order of time units for data entry and display.

GLS

In some East-Asian locales, the **GL_DATE** environment variable can specify Japanese or Taiwanese eras for the entry and display of DATE values. (In any locale, **GL_DATE** can specify formats beyond what **DBDATE** can specify.) ♦

By default, if the user of a 4GL application enters from the keyboard a single-digit or double-digit value for the year, as in `3` or `03`, 4GL uses the setting of the **DBCENTURY** environment variable to supply the first two digits of the year. (Users must pad the year value with one or two leading zeros to specify years in the First Century; for example, `093` or `0093` for the year 93 A.D.)

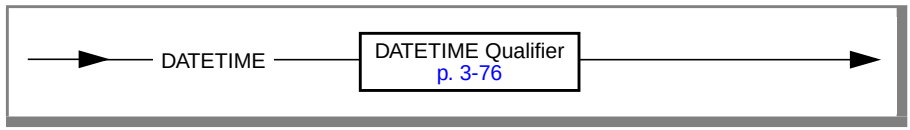
Dates that are stored as CHAR or VARCHAR strings are not affected by **DBCENTURY**; see Appendix D, "Environment Variables." You can also set the CENTURY attribute for individual DATE fields to override the global **DBCENTURY** expansion rule in that field; see "CENTURY" on page 6-35 for details of setting this field attribute.

Because DATE values are stored as integers, you can use them in arithmetic expressions, such as the difference between two DATE values. The result, a positive or negative INT value, is the number of days that have elapsed between the two dates. The UNITS DAY operator can convert this to an INTERVAL data type. (DATE operands in division, multiplication, or exponentiation operations generally cannot produce meaningful results.)

The FORMAT attribute specifies DATE display formats in forms. For month, 4GL accepts the value `1` or `01` for January, `2` or `02` for February, and so on. For days, it accepts a value `1` or `01` for the first day of the month, `2` or `02` for the second, and so on, up to the maximum number of days in a given month.

DATETIME

The DATETIME data type stores an instance in time, expressed as a calendar date and time-of-day. You specify the time units that the DATETIME value stores; the precision can range from a year through a fraction of a second. Data values are stored as DECIMAL formats representing a contiguous sequence of values for units of time.



GLS

In some East-Asian locales, the **GL_DATETIME** environment variable can specify Japanese or Taiwanese eras for the entry and display of DATETIME values. (In any locale, however, **GL_DATETIME** can specify nondefault data entry and data display formats for DATETIME values.) ♦

The scale and precision specification is called the *DATETIME qualifier*. It uses a "*first* TO *last*" format to declare variables and screen fields. You must substitute one or two of these keywords for the first and last terms.

Keyword	Corresponding Time Unit and Range of Values
YEAR	A year, numbered from 0001 (A.D.) to 9999
MONTH	A month, numbered from 1 to 12
DAY	A day, numbered from 1 to 31, as appropriate for its month
HOUR	An hour, numbered from 0 (midnight) to 23
MINUTE	A minute, numbered from 0 to 59
SECOND	A second, numbered from 0 to 59
FRACTION (*scale*) or FRACTION	A decimal fraction of a second, with a scale of up to five digits; the default scale is three digits (thousandth of a second)

The keyword specifying *last* in "*first* TO *last*" cannot represent a larger unit of time than *first*. Thus, YEAR TO SECOND or HOUR TO HOUR are valid, but DAY TO MONTH results in a compiler error, because the value for *last* (here MONTH) specifies a larger unit of time than DAY, the first keyword.

Unlike INTERVAL qualifiers, DATETIME qualifiers cannot specify nondefault precision (except for FRACTION, if that is the last keyword in the qualifier). The following are examples of valid DATETIME qualifiers:

- DAY TO MINUTE
- FRACTION TO FRACTION(4)
- YEAR TO MINUTE
- MONTH TO SECOND

Operations with DATETIME values that do not include YEAR in their qualifier use the system date to supply any additional precision. If the first term is DAY and the current month has fewer than 31 days, unexpected results can occur.

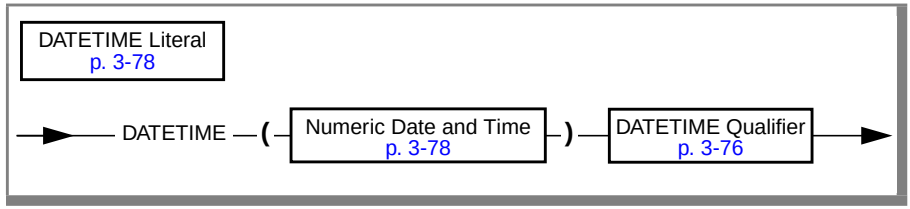
For example, assume that it is February, and you wish to store data from January 31 in the **sometime** variable that is declared in this statement:

```
DEFINE sometime DATETIME DAY TO MINUTE
CREATE TABLE mytable (mytime DATETIME DAY TO MINUTE)
LET sometime = DATETIME(31 12:30) DAY TO MINUTE
INSERT INTO mytable VALUES (sometime)
```

Because the column **mytime** does not store the month or year, the current month and year are used to evaluate whether the inserted value is within acceptable bounds. February has only 28 (or 29) days, so no value for DAY can be larger than 29. The INSERT statement in this case would fail, because the value 31 for day is out of range for February. To avoid this problem, qualify DATETIME data types with YEAR or MONTH as the first keyword, and do not enter data values with DAY as the largest time unit.

DATETIME Literals and Delimiters

Statements of 4GL can assign values to DATETIME data types. The simplest way to do this is as a DATETIME literal or as a character string. Both formats represent a specific DATETIME value as a numeric DATETIME value.



The *DATETIME literal* format begins with the DATETIME keyword, followed by a pair of parentheses that enclose unsigned whole numbers (separated by delimiters) to represent a consecutive sequence of year through fraction values, or a subset thereof. This must be followed by a DATETIME qualifier, specifying the "*first* TO *last*" keywords for the set of time units within the numeric DATETIME value.

The required delimiters must separate every time unit value in a literal. DATETIME YEAR TO FRACTION(3) values, for example, require six delimiters, as shown in Figure 3-1.

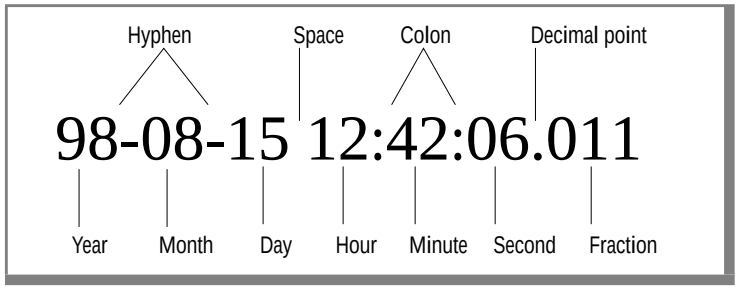


Figure 3-1
Numeric Date and Time

These are the delimiters that are required for separating successive units of time within DATETIME literal values.

Delimiter	Position Within Numeric DATETIME Value
Hyphen (-)	Between the year, month, and day portions of the value
Blank space	Between the day and hour portions
Colon (:)	Between the hour, minute, and second portions
Decimal point (.)	Between the second and fraction portions

DATETIME literals can specify every time unit from the data type declaration, or only the units of time that you need. For example, you can assign a value qualified as MONTH TO HOUR to a variable declared as YEAR TO MINUTE if the value contains information for a contiguous sequence of time units. You cannot, however, assign a value for just *month* and *hour*; in this case, the DATETIME literal must also include a value (and delimiters) for *day*.

A DATETIME literal that specifies fewer units of time than in the declaration is automatically expanded to fill all the declared time units. If the omitted value is for the first unit of time, or for this and for other time units larger than the largest unit that is supplied, the missing units are automatically supplied from the system clock-calendar. If the value omits any smaller time units, their values each default to zero (or to 1 for month and day) in the entry. To specify a year between 1 and 99, you must pad the year value with leading zeros.

The **DBCENTURY** environment variable determines how single-digit and double-digit *year* values are expanded; see Appendix D, "Environment Variables." The CENTURY attribute for an individual DATETIME field can override the global **DBCENTURY** expansion rule within that field.

Character Strings as DATETIME Values

You can also specify a DATETIME value as a *character string*, indicating the numeric values of the date, the time, or both. In a 4GL source module, this must be enclosed between a pair of quotation (") marks, without the DATETIME keyword and without qualifiers, but with all the required delimiters. A pair of single (') quotation marks is also valid as delimiters. Unlike DATETIME literals, the character string must include information for all the units of time declared in the DATETIME qualifier. For example, the following LET statement specifies a DATETIME value entered as a character string:

```
LET call_dtime = "2001-08-14 08:45"
```

In this case, the **call_dtime** variable was declared as DATETIME YEAR TO MINUTE, so the character string must specify values for year, month, day, hour, and minute time units. If the character string does not contain information for all the declared time units, an error results. Similarly, an error results if a delimiter is omitted, or if extraneous blanks appear within the string.

When a user of the 4GL program enters data in a DATETIME field of a screen form, or during a PROMPT statement that expects a DATETIME value, the only valid format is a numeric DATETIME value, entered as an unquoted string. Any entry in a DATETIME field must be a contiguous sequence of values for units of time and delimiters, in the following format (or in some subset of it):

```
year-month-day  hour:minute:second.fraction
```

Depending on the data type declaration of the DATETIME field, each of these units of time can have values that combine traditional base-10, base-24, base-60, and lunar calendar values from clocks and calendars.

Values that users enter in a DATETIME field of the 4GL form need not include all the declared time units, but users cannot enter data as DATETIME literals, a format that is valid only within 4GL statements and in the data type declarations of FORMONLY DATETIME fields of form specification files.

DATETIME values are stored internally in the format of fixed-point DECIMAL values (as described in "DECIMAL (p, s)" on page 3-23), but with sets of consecutive digits representing the declared time units. All time-unit values of a DATETIME data type are two-digit numbers, except for the year and fraction values. The *year* is stored as four digits. The fraction requires *n* digits, for $1 \leq n \leq 5$, rounded up to an even number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes required to store a DATETIME value:

```
( ( total_number_of_digits_for_all_time_units) /2)+1
```

For example, a YEAR TO DAY qualifier requires a total of 8 digits (4 for year, 2 for month, and 2 for day), or ((8 / 2) + 1) = 5 bytes of storage. For information about DATETIME values in expressions, see "Time Expressions" on page 3-72.

DEC

The DEC keyword is a synonym for DECIMAL.

DECIMAL (p, s)

The DECIMAL(*p*,*s*) data type stores values as fixed-point decimal numbers, of up to 30 (and for some data values, up to 32) significant digits. As the syntax diagram in "Simple Data Types" on page 3-9 indicates, you can optionally specify *precision* (the number of significant digits) and *scale* (the number of digits to the right of the decimal point). For example, DECIMAL (14,2) specifies a total of 14 significant digits, 2 of which describe the fractional part of the value.

The largest absolute value that a DECIMAL(*p*,*s*) data type can store without an error is $10^{p-s} - 10^{-s}$, where *p* is the precision and *s* is the scale. Values with an absolute value less than $0.5 ¥ 10^{-s}$ are stored as zero. You cannot specify *p* or *s* for a FORMONLY DECIMAL field in a form; its precision is the smaller of 32 and (*length* - 2), where *length* is the field width in the screen layout.

DECIMAL (*p*,*s*) data types are useful for storing numbers with fractional parts that must be calculated exactly, such as rates or percentages. Unless you are developing a scientific or engineering application that explicitly controls for measurement error, store floating-point numbers as DECIMAL(*p*,*s*) values.

When a user enters data in a SMALLFLOAT or FLOAT field, 4GL converts the base-10 value to binary format for storage. Likewise, to display a FLOAT or SMALLFLOAT number, 4GL reformats it from binary to base-10. Both conversions can lead to inaccuracy. Thus, if `10.7` is entered into a FLOAT field, it might be stored as `10.699999` or as `10.700001`, depending on the magnitude and the sign of the binary-to-decimal conversion error. This limitation is a feature of digital computers, rather than of 4GL, but it might motivate you to use DECIMAL(*p*,*s*) rather than FLOAT in contexts requiring high precision.

DECIMAL(*p*,*s*) values are stored internally with the first byte representing a sign bit and a 7-bit exponent in excess-65 format; the rest of the bytes express the mantissa as base-100 digits. This implies that DECIMAL(32,*s*) data types store only *s*-1 decimal digits to the right of the decimal point, if *s* is an odd number. The stored value can have up to 30 significant decimal digits in its fractional part, or up to 32 digits to the left of the decimal point. The following formulae calculate storage (in bytes) needed for DECIMAL(*p*,*s*) values, with any fractional part of the result discarded:

When scale is even: *(precision* + 3) / 2)
When scale is odd: *(precision* + 4) / 2)

For example, DECIMAL(14,2) requires ((14 + 3) / 2), or 8 bytes of storage.

DECIMAL (p)

When you specify both the precision and the scale, the 4GL program can manipulate the DECIMAL (*p*,*s*) value with fixed-point arithmetic. If the data type declaration specifies no precision or scale, however, the default is DECIMAL(16), a floating-point number with a precision of 16 digits.

If only one parameter is specified, this is interpreted as the precision of a floating-point number whose exponent can range from 10^{-130} to 10^{126}. The range of absolute data values is approximately from 1.0E-130 to 9.99E+126.

In an ANSI-compliant database, declaring a column as DECIMAL(*p*) results in a fixed-point data type with a precision of *p* and a scale of zero. If you declare a 4GL data type LIKE a DECIMAL(*p*) column in an ANSI-compliant Informix database, the resulting data type is restricted to a scale of zero, which is equivalent to an integer. For DECIMAL(*p*,0) data types, any significant digits (within the declared precision) in the fractional part of a data value are stored internally, but are not displayed. For example, if a data value of 0.25 is stored in the DECIMAL(25,0) variable **Q**, the value of **Q** is displayed as zero, but the Boolean expression `(Q > 0)` evaluates as TRUE.

DOUBLE PRECISION

The DOUBLE PRECISION keywords are a synonym for FLOAT.

FLOAT

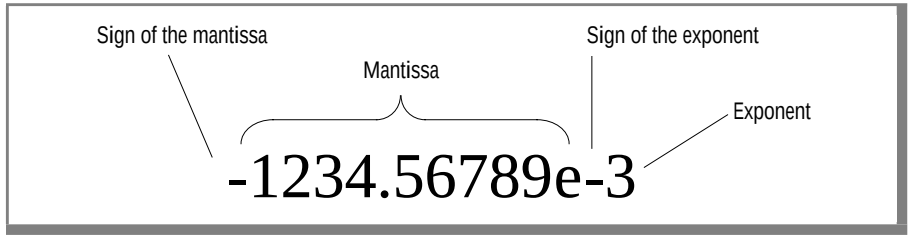
The FLOAT data type stores values as double-precision floating-point binary numbers with up to 16 significant digits. FLOAT corresponds to the **double** data type in the C language. Values for the FLOAT data type have the same range of values as the C **double** data type on your C compiler. FLOAT data types usually require 8 bytes of memory storage.

For compatibility with the ANSI standard for embedded SQL, you can declare a whole number between 1 and 14 as the *precision* of a FLOAT data type, but the actual precision is data-dependent and compiler-dependent.

A variable of the FLOAT data type typically stores scientific or engineering data that can be calculated only approximately. Because floating-point numbers retain only their most significant digits, a value that is entered into A FLOAT variable or database column can differ slightly from the numeric value that a 4GL form or report displays.

This rounding error arises because of the way computers store floating-point numbers internally. For example, you might enter a value of `1.1` into a FLOAT field. After processing the 4GL statement, the program might display this value as `1.09999999`. This occurs in the typical case where the exact floating-point binary representation of a base-10 value requires an infinite number of digits in the mantissa. The computer stores a finite number of digits, so it stores an approximate value, with the least significant digits treated as zeros.

Statements of 4GL can specify FLOAT values as *floating-point literals.*



You can use uppercase or lowercase E as the exponent symbol; omitted signs default to + (positive). If a number in another format (such as an integer or a fixed-point decimal) is supplied in a **.4gl** file or from the keyboard when a FLOAT value is expected, 4GL attempts data type conversion.

In reports and screen displays, the USING operator can format FLOAT values. Otherwise, the default scale in output is two decimal digits.

INT

The INT keyword is a synonym for INTEGER.

INTEGER

The INTEGER data type stores whole numbers in a range from -2,147,483,647 to +2,147,483,647. The negative number -2,147,483,648 is a reserved value that cannot be used. Values are stored as signed 4-byte binary integers, with a scale of zero, regardless of the word length of your system. INTEGER can stores counts, quantities, categories coded as natural numbers, and the like.

Arithmetic operations on binary integers are typically without rounding error; these operations and sort comparisons are performed more efficiently than on FLOAT or DECIMAL data. INTEGER values, however, can only store data whose absolute value is less than 2^{31}. Any fractional part of the value is discarded. If a value exceeds this numeric range, neither 4GL nor the database can store the data value as an INTEGER data type.

INTEGER variables can store SERIAL values of the database. If a user inserts a new row into the database, 4GL automatically assigns the next whole number in sequence to any field linked to a SERIAL column. Users do not need to enter data into such fields. Once assigned, a SERIAL value cannot be changed. (See the description of the SERIAL data type in the *Informix Guide to SQL: Reference*.)

INTERVAL

This data type stores *spans* of time, the differences between two points in time. You can also use it to store quantities that are naturally measured in units of time, such as age or sums of ages, estimated or actual time required for some activity, or person-hours or person-years of effort attributed to some task.

An INTERVAL data value is stored as a DECIMAL number that includes a contiguous sequence of values representing units of time. The INTERVAL data types of 4GL fall into two classes, based on their declared precision:

- A *year-month* interval represents a span of years, months, or both.
- A *day-time* interval represents a span of days, hours, minutes, seconds, and fractions of a second, or a contiguous subset of those units.

Automatic data type conversion between these two categories of INTERVAL data types is not a feature of 4GL.

Unlike DATETIME data types, which they somewhat resemble in their format, INTERVAL data types can assume zero or negative values. The declaration of an INTERVAL data type uses the following syntax.

INTERVAL Qualifiers

The *INTERVAL qualifier* specifies the precision and scale of an INTERVAL data type, using a *first* TO *last* format to declare 4GL variables, formal arguments of functions and reports, and screen fields. It has the same syntax in declarations of 4GL variables and FORMONLY fields as for INTERVAL columns of the database. You must substitute one or two keywords from only one the following lists for *first* and *last* keywords of an INTERVAL qualifier.

Year-Month INTERVAL Keywords	Day-Time INTERVAL Keywords
YEAR, MONTH	DAY, HOUR, MINUTE, SECOND, FRACTION

As with DATETIME data types, you can declare INTERVAL data types to include only the units that you need. INTERVAL represents a span of time independent of an actual date, however, so you cannot mix keywords from both lists in the same INTERVAL qualifier. Because the number of days in a month depends on the month, an INTERVAL data value cannot combine both months and days as units of time. For example, specifying "MONTH TO MINUTE" as an INTERVAL qualifier produces a compile-time error.

Arithmetic expressions that combine *year-month* INTERVAL values with DATE values (or with DATETIME values that include smaller time units than month) can return an invalid date, such as February 30. (In general, adding or subtracting an interval of months from any calendar date later than the 28th day of any month can produce similar errors in SQL and 4GL expressions.)

For any keyword specifying *first* except FRACTION, you have the option of specifying a precision of up to 9 digits; otherwise the default precision is 2 digits, except for YEAR, which defaults to 4 digits of precision. If an INTERVAL qualifier specifies only a single unit of time, the keywords specifying *first* and *last* are the same. When *first* and *last* are both FRACTION, you can only specify the scale after the *last* keyword.

When *last* is FRACTION, you can specify a scale of 1 to 5 digits; otherwise, the scale defaults to 3 digits (thousandth of a second). For example, the following are valid INTERVAL qualifiers:

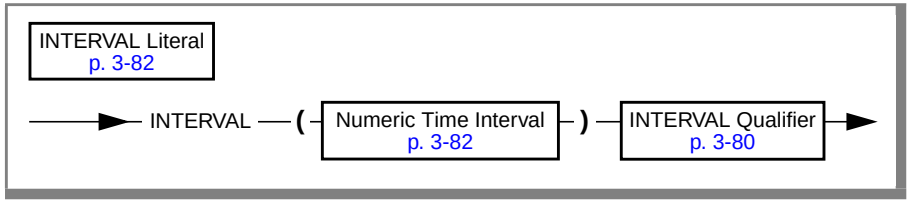
```
HOUR TO MINUTE          MONTH(5) TO MONTH
FRACTION TO FRACTION(4) HOUR(9) TO FRACTION(5)
```

The keyword specifying the last term cannot represent a larger unit of time than that specifying first term. Thus, YEAR TO MONTH and HOUR TO HOUR are valid, but HOUR TO DAY results in a compiler error, because the first keyword (here HOUR) specifies a smaller unit of time than DAY, the last keyword.

After you declare an INTERVAL data type, a 4GL statement can assign it the value of a *time expression* (as described in "Time Expressions" on page 3-72) that specifies an INTERVAL value. The simplest way to do this is as an INTERVAL literal or as a character string. Both formats require that you specify a numeric INTERVAL value.

INTERVAL Literals and Delimiters

The *INTERVAL literal* format begins with the INTERVAL keyword, followed by a pair of parentheses that enclose unsigned whole numbers (separated by delimiters) to represent a consecutive sequence of YEAR through FRACTION values, or as a portion thereof. This must be followed by a valid INTERVAL qualifier, specifying the "*first* TO *last*" keywords for the set of time units.

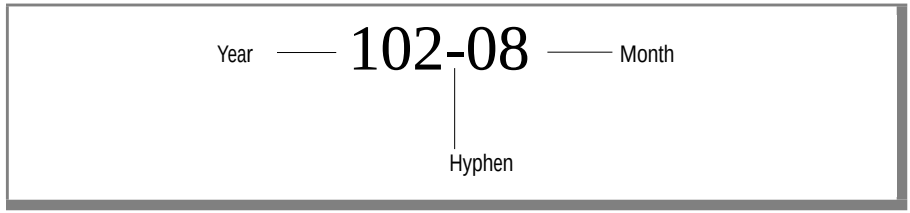


A numeric INTERVAL uses the same delimiters as DATETIME values, except that *month* and *day* need no separator, because they cannot both appear in the same INTERVAL value. The following delimiters are required for separating successive units of time within literal INTERVAL values.

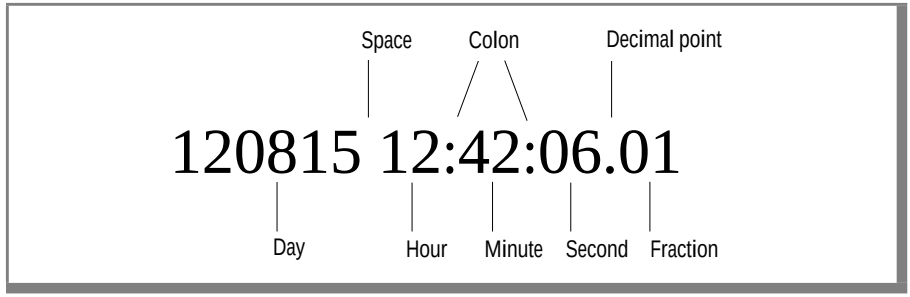
Delimiter	Position Within Numeric INTERVAL Value
Hyphen (-)	Between the year, month, and day portions of the value
Blank space	Between the day and hour portions
Colon (:)	Between the hour, minute, and second portions
Decimal point (.)	Between the second and fraction portions

Time unit values in a numeric INTERVAL have a default precision of 2 digits, except for year (4 digits), and fraction (3). The qualifier can override these defaults for the *first* time unit, and for the *scale* of the fraction.

For example, INTERVAL YEAR(3) TO MONTH values require one delimiter.



Similarly, INTERVAL DAY(6) TO FRACTION(2) values require four delimiters.



INTERVAL literals can specify all the time units from the data type declaration, or only the units that you need. For example, you can assign a value qualified as HOUR TO MINUTE to a variable declared as DAY TO SECOND if the entered value contains information for a contiguous sequence of time units. You cannot, however, assign only *day* and *minute* values; in this case, the INTERVAL literal must also include a value (and delimiters) for *hour*.

The value for the *first* time units in an INTERVAL literal can have up to nine digits of precision (except for FRACTION, which cannot include more than five digits). If a *first* unit value to be entered is greater than the default number of digits for that time unit, however, you must explicitly identify the number of significant digits that you are entering. For example, an INTERVAL of DAY TO HOUR that spans 162.5 days can use the following format:

```
INTERVAL (162 12) DAY(3) TO HOUR
```

An INTERVAL literal in a 4GL statement must include numeric values for both the *first* and *last* time units from the qualifier, and values for any intervening time units. You can optionally specify the precision of the *first* time unit (and also a *scale*, if the last keyword of the INTERVAL qualifier is FRACTION).

Character Strings as INTERVAL Values

You can also specify an INTERVAL value as a *character string*, indicating the numeric values of the time units. In a 4GL source code module, this must be enclosed between a pair of quotation (") marks, without the INTERVAL keyword and without qualifiers, but with all the required delimiters. Unlike INTERVAL literals, the character string must include information for all the units of time declared in the INTERVAL qualifier. For example, the character string in the next statement specifies a span of five years and six months:

```
LET long_time = "5-06"
```

Similarly, values entered as character strings into INTERVAL columns of the database must include information for all time units that were declared for that column. For example, the following INSERT statement shows an INTERVAL value entered as a character string:

```
INSERT into manufact (manu_code, manu_name, lead_time)
   VALUES ("BRO", "Ball-Racquet Originals", "160")
```

Because the **lead_time** column is defined as INTERVAL DAY(3) TO DAY, this INTERVAL value requires only one value, indicating the number of days required. If the character string does not contain information for all the declared time units, the database server returns an error.

Data Entry by Users

When a user of the 4GL program enters data in an INTERVAL field of a form, the only valid format is as an unquoted character string. Any entry into an INTERVAL field must be a contiguous sequence of values for units of time and separators, in one of these two formats (or in some subset of one):

```
year-month
day  hour:minute:second.fraction
```

Depending on the data type declaration of the field, each of these units of time (except the first) is restricted to values that combine traditional base-10, base-24, base-60, and lunar calendar values from clocks and calendars. The first value can have up to nine digits, unless FRACTION is the first unit of time. (If FRACTION is the first time unit, the maximum scale is 5 digits.)

Values that users enter in an INTERVAL field of a 4GL form need not include all the declared time units, but users cannot enter data as INTERVAL literals, a format that is valid only within 4GL statements and in data type declarations of FORMONLY fields of data type INTERVAL in form specification files.

By default, all values for time units in a numeric INTERVAL are two-digit numbers, except for the year and fraction values. The year value is stored as four digits. The fraction value requires *n* digits where $1 \leq n \leq 5$, rounded up to an even whole number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes required for an INTERVAL value:

```
( ( total_number_of_digits_for_all_time_units)/2) + 1
```

For example, a YEAR TO MONTH qualifier requires a total of 6 digits (4 for year and 2 for month), or ((6 / 2) + 1) = 4 bytes of storage.

For information on using INTERVAL data in arithmetic and relational operations, see "Expressions of 4GL" on page 3-49.

MONEY

The MONEY data type stores currency amounts. Like the DECIMAL data type, the MONEY data type stores fixed-point numbers, up to a maximum of 32 significant digits. As the syntax diagram in "Simple Data Types" on page 3-9 indicates, you can optionally include one or two whole numbers to specify the precision (the number of significant digits) and the scale (the number of digits to the right of the decimal point).

Unlike the DECIMAL data type (described in "DECIMAL (p, s)" on page 3-23), which stores floating-point numbers if its data type declaration specifies neither scale nor precision, MONEY values are always stored as fixed-point decimal numbers. If you declare a MONEY data type with only one parameter, 4GL interprets that parameter as the precision. By default, the scale is 2, so the data type MONEY(*p*) is stored internally as DECIMAL(*p*,*2*), where *p* is the precision ($1 \leq p \leq 32$).

If no parameters are specified, MONEY is interpreted as DECIMAL(16,2). This stores 16 significant digits, 2 of which describe the fractional part of the currency value. The largest absolute value that you can store without error as a MONEY data type is $10^{p-s} - 10^{-s}$. Here *p* is the precision, and *s* is the scale.

Values with an absolute value less than 0.5×10^{-s} are stored as zero. You cannot specify the precision or the scale of a FORMONLY MONEY field in a 4GL form; here the precision defaults to the smaller of 32 or (*length* - 2), where *length* is the field length from the SCREEN section layout.

On the screen, MONEY values are displayed with a currency symbol, by default, a dollar sign ($), and a decimal point (.) symbol. You can change the display format for MONEY values in the **DBMONEY** or **DBFORMAT** environment variable. The settings of these variables take precedence over the default currency format of the locale. 4GL statements and keyboard input by users to fields of screen forms do not need to include currency symbols in literal MONEY values.

The same formulae as for DECIMAL values apply to MONEY data types, with any fractional part of the result discarded:

When scale is even: (*precision* + 3) / 2
When scale is odd: (*precision* + 4) / 2

For example, a MONEY(13,2) variable has a precision of 13 and a scale of 2. This requires ((13 + 3) / 2) = 8 bytes of storage.

NCHAR

The NCHAR data type stores character data in a fixed-length field. This data can be a sequence of single-byte or multibyte letters, numbers, and symbols. However, the code set of your database locale must support the characters. NCHAR columns of the database can support localized collation, if the locale specifies a localized collating sequence, but 4GL treats NCHAR variables like CHAR variables.

NCHAR stores the number of bytes specified between parentheses in the *size* parameter of the data type declaration. The *size* can range from 1 to 32,767 bytes.

SE

For INFORMIX-SE, the *size* can range from 1 to 32,511 bytes. ♦

If you do not specify *size* in a DEFINE declaration, the default is NCHAR(1).

When the database server retrieves or sends an NCHAR value, it transfers exactly *size* bytes of data. If the length of a character string is shorter than *size*, the database server extends the string with spaces to make up the *size* bytes. If the string is longer than *size* bytes, the database server truncates the string.

NVARCHAR

The NVARCHAR data type stores character data in a variable-length field. This data can be a sequence of single-byte or multibyte letters, numbers, and symbols. However, the code set of your database locale must support the characters. NVARCHAR columns of the database can support localized collation, if the locale specifies a localized collating sequence, but 4GL treats NVARCHAR variables like VARCHAR variables.

The database server does not strip an NVARCHAR object of any user-entered trailing white space, nor does it pad the NVARCHAR object to the full length of the column. However, if you specify a minimum reserved space (*reserve*) and some of the data values are shorter than that amount, some of the space that is reserved for rows goes unused.

NUMERIC

The NUMERIC keyword is a synonym for DECIMAL. (When the word *numeric* appears in lowercase letters in this manual, it is always the adjective formed from the noun *number,* rather than the name of a data type.)

The NUMERIC keyword is a synonym for DECIMAL. (When the word *numeric* appears in lowercase letters in this manual, it is always the adjective formed from the noun *number,* rather than the name of a data type.)

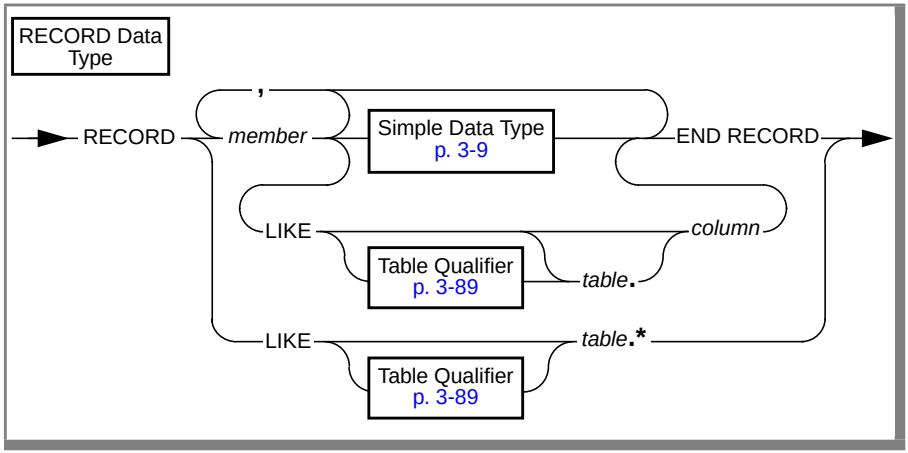
REAL

The REAL keyword is a synonym for SMALLFLOAT. (When the phrase *real number* appears in lowercase letters in this manual, it denotes a number that is neither imaginary nor transfinite, rather than the name of a data type.)

RECORD

The RECORD data type is an ordered set of variables. Within each of these sets (called a *program record*), the component variables (called *members*) can be of any 4GL data type, or any combination of data types in a fixed order.

Valid data types for member variables of records include all the simple data types (listed in "Simple Data Types" on page 3-9), the large data types (BYTE, TEXT), and the structured data types (ARRAY, RECORD). The following is the data type declaration syntax for RECORD variables.



Element	Description
column	is the name of a column in the default (or specified) database.
member	is a name that you declare for a member variable of the record.
table	is the SQL identifier of a database table, synonym, or view.

You can use the LIKE keyword to specify that a member variable has the same data type as a database column. If you do not specify *member* names, but use an asterisk (*) after a *table* name, you declare a record whose members have the same identifiers as the columns in *table*; their data types correspond to the fixed sequence of SQL data types in an entire row of the table. (Any SERIAL column in *table* corresponds to a record member of data type INTEGER.)

This example uses the LIKE keyword to declare two program records, one of which contains a member variable called **nested** of the RECORD data type:

```
DEFINE p_customer RECORD LIKE informix.customer.*,
       p_orders RECORD
          order_num LIKE informix.orders.order_num,
          nested RECORD a LIKE informix.items.item_num,
                        b LIKE informix.stock.unit_descr
                 END RECORD
          END RECORD
```

If *table* is a view, the column cannot be based on an aggregate. You cannot specify *table.** if *table* is a view that contains an aggregate column.

In an ANSI-compliant database, you must qualify the table name with the *owner* prefix, if the program will be run by users other than *owner*. If the table is an external or external, distributed table, its name must be qualified by the name of the remote database and by the name of its database server.

GLS

If the client locale and the database locale are not identical, do not use the LIKE keyword to declare a member variable whose name is the same as a database column whose identifier includes any non-ASCII character that is not supported by the character set of the client locale. ♦

Referencing Record Members

If *record* is the identifier that you declare for a program record in a DEFINE statement, or the name of a screen record in a 4GL form, you can use the following notation to reference members of the record in 4GL statements:

- The notation *record.member* refers to an individual member of a record, where *member* is the identifier of the member.
- The notation *record.first* THRU *record.last* refers to a consecutive subset of members, from *record.first* through *record.last*. Here *first* is an identifier that was listed before *last* among the explicit or implicit member names in the RECORD declaration. You can also use the keyword THROUGH as a synonym for THRU.
- The notation *record.** refers to the entire record.

Several restrictions apply when you reference members of a record:

- You cannot use THRU or THROUGH to indicate a partial list of screen record members in 4GL statements for displaying or entering data in a screen form.
- You cannot use THRU, THROUGH, or .* to reference a program record that contains an ARRAY variable among its members. (But you can use these notations to specify all or part of a record that contains one or more other records as members.)
- You cannot use THRU, THROUGH, or .* notation in a SELECT or INSERT *variable list* in a quoted string in PREPARE statements. (You can, however, use the .* notation to specify a program record in the variable list of an INSERT or SELECT clause of a DECLARE statement.)

A program record whose members correspond in number, order, and data type compatibility to a database table or to a screen record can be useful for transferring data from the database to the screen, to reports, or to functions of the 4GL program. For more information, see "Summary of Compatible 4GL Data Types" on page 3-46 and Chapter 7, "INFORMIX-4GL Reports."

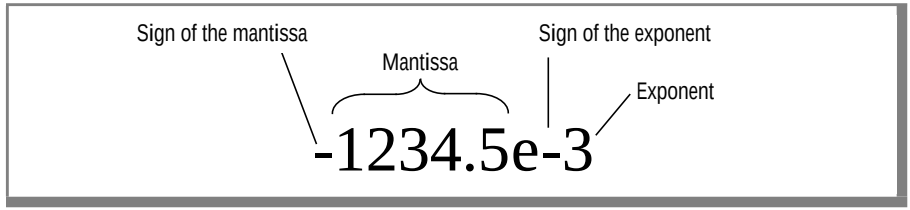
SMALLFLOAT

The SMALLFLOAT data type stores single-precision floating-point binary numbers, with up to 8 significant digits. The range of values is the same as for the **float** data type of C. The storage requirement is usually 4 bytes.

The SMALLFLOAT data type typically stores scientific or engineering data that can only be calculated approximately. Because floating-point numbers retain only their most significant digits, a value entered into a SMALLFLOAT variable or column can differ slightly from the base-10 value that an 4GL form or 4GL report displays.

This error arises from the internal storage format of binary floating-point numbers. For example, if you enter a value of `1.1` into a SMALLFLOAT field, after processing the 4GL statement, the screen might display this value as `1.1000001`. This occurs in the typical case where the exact floating-point binary representation of a base-10 value requires an infinite number of digits in the mantissa. A computer stores only a finite number of digits, so it stores an approximate value, with the least-significant digits treated as zeros.

Statements of 4GL can specify SMALLFLOAT values as floating-point literals, using the following format.



You can use uppercase or lowercase E as the exponent symbol; omitted signs default to + (positive). If a literal value in another format (such as an integer or a fixed-point decimal) is supplied from the keyboard into a SMALLFLOAT field, or in a 4GL statement, 4GL attempts data type conversion.

In reports and screen displays, the USING operator can format SMALLFLOAT values. The default scale in output, however, is two digits.

SMALLINT

The SMALLINT data type stores data as signed 2-byte binary integers. Values must be whole numbers within the range from -32,767 to +32,767. Any fractional part of the data value is discarded. If a value lies outside this range, you cannot store it in a SMALLINT variable or database column. (The negative value -32,768 is reserved; it cannot be assigned to a SMALLINT variable or database column, and it cannot be entered into a SMALLINT field of a form.)

You can use SMALLINT variables and FORMONLY fields of screen forms to store, manipulate, and display data that can be represented as whole numbers of an absolute value less than 2^{15}. This data type typically stores small whole numbers, Boolean values, ranks, or measurements that classify data into a small number of numerically-coded categories. Because the SMALLINT data type requires only 2 bytes of storage, arithmetic operations on SMALLINT operands can be done very efficiently, provided that all of the data values lie within the somewhat limited SMALLINT range.

TEXT

The TEXT data type stores character data in ASCII strings. TEXT resembles the BYTE data type, but 4GL supports features to display TEXT variables whose values are restricted to combinations of printable ASCII characters and the following white space characters:

- TAB (= CONTROL-I)
- NEWLINE (= CONTROL-J)
- FORMFEED (= CONTROL-L)

If you attempt to include other non-printable characters in TEXT values, the features of 4GL for processing TEXT data might not work correctly.

GLS

In some locales, other white space characters are supported; more than one byte is required to store some East-Asian white space characters. ♦

Strings stored as TEXT variables have a theoretical limit of 2^{31} bytes, and a practical limit determined by the available storage on your system.

SE

INFORMIX-SE database servers do not support TEXT columns, but regardless of the database server, you can declare 4GL variables of type TEXT. ♦

You can use a TEXT variable to store, retrieve, or update the contents of a TEXT database column, or to reference a file that you wish to display to users of the 4GL program through a text editor. After you declare a TEXT data type, you must use the LOCATE statement to specify the storage location.

When you retrieve a value from a TEXT column, you can assign all or part of it to a TEXT variable. Use brackets ([ ]) and comma-separated subscripts to reference only a specified part of a TEXT value, as in the following example:

```
SELECT cat_description [1,75] INTO cat_nap FROM catalog
      WHERE catalog_num = 10001
```

This reads the first 75 bytes of the **cat_description** column of the row with the catalog number 10001, and stores these data in the **cat_nap** TEXT variable.

Restrictions on TEXT Variables

In a 4GL form, a field linked to a TEXT column (or a FORMONLY field of type TEXT) only displays as many characters as can fit in the field. To display TEXT values longer than the screen field, or to edit a TEXT value, you must assign the PROGRAM attribute to the TEXT field. The WORDWRAP attribute can display the initial characters of a TEXT value, up to the last segment of the field, but cannot edit a TEXT field. No other 4GL field attribute (except COLOR) can reference the value of a TEXT field.

In a CALL, OUTPUT TO REPORT, or RETURN statement, TEXT arguments are passed by reference, rather than by value. ("Passing Arguments by Reference" on page 4-18 discusses this issue in greater detail.) The DISPLAY TO statement can display a TEXT value, but DISPLAY and DISPLAY AT cannot. The LET statement cannot assign any value (except NULL) to a TEXT variable.

Built-in functions of 4GL cannot specify TEXT arguments. Expressions of 4GL (as described in "Expressions of 4GL" on page 3-49) can reference TEXT variables only to test for NULL values, or as an operand of WORDWRAP.

VARCHAR

The VARCHAR data type stores character strings of varying lengths. You can optionally specify the maximum *size* of a data string, and the minimum storage *reserved* on disk.

SE

INFORMIX-SE database servers do not support this data type, but any 4GL application can declare VARCHAR variables. ♦

The declared *size* of VARCHAR can range from 1 to 255 bytes. If you specify no size, the default is 1. The data type can store shorter character strings than this maximum size, but not longer strings. In a form specification file, you cannot specify any parameters for a FORMONLY VARCHAR field; here the *size* defaults to the physical field length in the screen layout.

In data type declarations, the minimum *reserved* storage can range from 0 to 255 bytes, but this cannot be greater than the declared *size*. Just as 4GL accepts but ignores the *precision* specification in FLOAT or DOUBLE PRECISION data type declarations for compatibility with ANSI/ISO SQL syntax, 4GL accepts but ignores *reserved* in VARCHAR declarations. (But in SQL declarations like CREATE TABLE, the *reserved* value can affect the behavior of the database.)

The ASCII 0 end-of-data character terminates every VARCHAR value; any subsequent characters in a data string generally cannot be retrieved from or entered into VARCHAR (nor CHAR) database columns.

When you assign a value to a VARCHAR variable, only the data characters are stored, but neither 4GL nor the database server strips a VARCHAR value of user-entered trailing blanks. Unlike CHAR values, VARCHAR values are *not* padded with blank spaces to the declared maximum size, so the CLIPPED operator may not be needed in operations on VARCHAR values.

VARCHAR values are compared to CHAR values and to other VARCHAR values in 4GL Boolean expressions in the same way that CHAR values are compared: the shorter value is padded on the right with spaces until both values have equal lengths and they are compared for the full length.

GLS

In most locales, VARCHAR values require one byte of storage per character, or *size* bytes for *size* characters. In some East Asian locales, however, more than one byte may be required to store an individual logical character, and some white space characters can occupy more than one byte of storage.

If a collation order is defined by the **COLLATION** category in a locale file, the database server uses this order to sort values from NVARCHAR and NCHAR database columns in SQL statements, but uses code-set order to sort CHAR or VARCHAR values. If **DBNLS** is set to 1, 4GL uses **COLLATION** to sort CHAR or VARCHAR variables; otherwise, 4GL uses code-set order. If 4GL and the database have different locales, collation order for a sorting operation might depend on whether 4GL or the database server performs the sort.

If the database has NVARCHAR or NCHAR columns, you must set the **DBNLS** environment variable to `1` if you want to store values from such columns in VARCHAR or CHAR variables of 4GL, or if you want to insert values of CHAR or VARCHAR variables into NCHAR or NVARCHAR database columns.

For more information, see Appendix E, "Developing Applications with Global Language Support." ♦

Data Type Conversion

4GL can assign the value of a number, character string, or point in time to a variable of a different data type. 4GL performs data type conversion without objection when the process makes sense. If you assign a number expression to a CHAR variable, for example, 4GL converts the resulting number to a literal string. In an expression, 4GL attempts to convert the string representation of a number or time value to a number, time span, or point in time.

An error is issued only if 4GL cannot perform the conversion. For example, 4GL converts the string `"123.456"` to the number 123.456 in an arithmetic expression, but adding the string `"Juan"` to a number produces an error.

The global **status** variable is not reset when a conversion error occurs, unless you specify the ANY ERROR keywords (without CONTINUE) in a WHENEVER compiler directive, or include the **-anyerr** command-line argument.

Converting from Number to Number

When you pass a value from one number data type to another, the receiving data type must be able to store all of the *source* value. For example, if you assign an INTEGER value to a SMALLINT variable, the conversion will fail if the absolute value is larger than 32,767. Overflow can also occur when you transfer data from FLOAT or SMALLFLOAT variables or database columns to INTEGER, SMALLINT, or DECIMAL data types.

The kinds of errors that you might encounter when you convert values from one number data type to another are listed in "Notes on Automatic Data Type Conversion" on page 3-48. For example, if you convert a FLOAT value to DECIMAL(4,2), 4GL or the database server rounds off the floating-point value before storing it as a fixed-point number. This can sometimes result in overflow, underflow, or rounding errors, depending on the data value and on the declared precision of the receiving DECIMAL data type.

The **SQLAWARN[1]** and **SQLAWARN[5]** characters of the global **SQLCA** record are set to `W` after any FLOAT or SMALLFLOAT value is converted to a DECIMAL value.

Converting Numbers in Arithmetic Operations

4GL performs most arithmetic operations on DECIMAL values, regardless of the declared data types of the operands. (The exceptions are integers; see "Arithmetic Operators" on page 3-66.) The data type of the receiving variable determines the format of the stored or displayed result.

The following rules apply to the precision and scale of the DECIMAL variable that results from an arithmetic operation on two numbers:

- All operands, if not already DECIMAL, are converted to DECIMAL, and the result of the arithmetic operation is always a DECIMAL.

Source Operand	Converted Operand
FLOAT	DECIMAL (16)
INTEGER	DECIMAL (10, 0)
MONEY (*p*)	DECIMAL (*p*, 2)
SMALLFLOAT	DECIMAL (8)
SMALLINT	DECIMAL (5, 0)

- In addition and subtraction, 4GL adds trailing zeros to the operand with the smaller scale, until the scales are equal.
- If the data type of the result of an arithmetic operation requires the loss of significant digits, 4GL reports an error.
- Leading or trailing zeros are not considered significant digits, and do not contribute to the determination of precision and scale.
- If one operand has no scale (that is, a floating-point decimal), the result is also a floating-point decimal.
- The precision and scale of the result of an arithmetic operation depend on the precision and scale of the operands and on the arithmetic operator. The rules of 4GL are summarized in the following table for arithmetic operands that have a definite scale.

In this table, p_1 and s_1 represent the precision and scale of the first operand, and p_2 and s_2 represent the precision and scale of the second operand.

Numeric Operation	Precision and Scale of Returned Value
Addition (+) *and* Subtraction (-)	Precision: MIN(32, MAX(p_1 - s_1, p_2 - s_2) + MAX(s_1, s_2) + 1) Scale: MIN(30, MAX(s_1, s_2))
Multiplication (*)	Precision: MIN(32, p_1+ p_2) Scale: MIN(30, s_1 + s_2)
Division (/)	Precision: 32 Scale: MAX(0, 32 - p_1 + s_1 - s_2)

These values are upper limits, because the actual precision and scale of DECIMAL(*p,s*) values are data-dependent. The USING operator can override the default scale when output is displayed.

Converting Between DATE and DATETIME

You can convert DATE values to DATETIME values. If the DATETIME precision includes time units smaller than *day*, however, 4GL either ignores the time-of-day units or else fills them with zeros, depending on the context. The examples that follow illustrate how these two data types are converted; it is assumed here that the default display format for DATE values is *mm/dd/yyyy*:

- If a DATE value is specified where a DATETIME YEAR TO DAY is expected, 4GL converts the DATE value to a DATETIME value. For example, `08/15/2001` becomes `2001-08-15`.
- If a DATETIME YEAR TO DAY value is specified where a DATE is expected, `2001-08-15` becomes `08/15/2001`.
- If a DATE value is specified where a DATETIME YEAR TO FRACTION (or TO SECOND, TO MINUTE, or TO HOUR) is expected, 4GL converts the DATE value to a DATETIME value, and fills any smaller time units from the DATETIME declaration with zeros. For example, the DATE value `08/15/2001` becomes `2001-08-15 00:00:00`.
- If a DATETIME YEAR TO SECOND to DATE, value is specified where a DATE is expected, 4GL converts the DATETIME value to a DATE value, but drops any time units smaller than DAY. Thus, `2001-08-15 12:31:37` becomes `08/15/2001`.

The EXTEND( ) operator can return a DATETIME value from a DATE operand.

Converting CHAR to DATETIME or INTERVAL Data Types

You can specify DATETIME and INTERVAL data in literal formats, as described in previous sections, or as quoted strings. Values specified as suitably formatted character strings are automatically converted into DATETIME or INTERVAL values, if the string specifies a value for every time unit declared for that DATETIME or INTERVAL variable. The next examples illustrate both formats for December 5, 1974, and for a time interval of nearly 18 days:

```
DEFINE mytime DATETIME YEAR TO DAY,
       myval INTERVAL DAY TO SECOND

LET mytime = DATETIME(74-12-5) YEAR TO DAY
LET mytime = "74-12-5"     --same effect as previous line

LET myval = INTERVAL(17 21:15:30) DAY TO SECOND
LET myval = "17 21:15:30" --same effect as previous line
```

When a character string is converted into a DATETIME or INTERVAL value, 4GL assumes that the character string includes information about all the declared time units. You cannot use character strings to enter DATETIME or INTERVAL values for a subset of time units, because this produces ambiguous values. If the character string does not contain information for all the declared time units, 4GL returns an error, as in some of these examples:

```
DEFINE tyme DATETIME YEAR TO DAY,
       mynt INTERVAL DAY TO SECOND

LET tyme = DATETIME(5-12) MONTH TO DAY    --Valid
LET tyme = "5-12"                         --Error!

LET mynt = INTERVAL(11:15) HOUR TO MINUTE  --Valid
LET mynt = "11:15"                        --Error!
```

The previous DATETIME example (variable **tyme**) assigns a MONTH TO DAY value to a variable declared as YEAR TO DAY. Entering only these values is valid in the first LET statement because the qualifier of the DATETIME literal specifies no *year*, so 4GL automatically supplies the value of the current year.

In the example of the character string, however, 4GL does not know whether the "5-12" refers to *year* and *month*, or *month* and *day*, so it returns an error.

The previous INTERVAL example (variable **mynt**) assigns an HOUR TO MINUTE value to a variable declared as DAY TO SECOND. The first LET statement simply pads the value with zeros for *day* and *second*. The second LET statement produces a conversion error, however, because 4GL does not know whether "11:15" specifies HOUR TO MINUTE or MINUTE TO SECOND.

Empty or blank strings are converted to null time (or number) values.

Converting Between Number and Character Data Types

You can store a CHAR or VARCHAR value in a number variable and vice versa. But if the CHAR or VARCHAR value contains any characters that are not valid in a number data type (for example, the letters `l` or `O` instead of the digits `1` or `0`), 4GL returns a data type conversion error.

In a locale that is not U.S. English, if number or currency values are converted to character strings during the LET statement, the conversion process inserts locale-specific separators and currency symbols into the converted strings, rather than the default U.S. English separators and currency symbols. ♦

Converting Large Data Types

You can store a TEXT value in a BYTE data type. No other data type conversions involving large binary data types are supported directly by 4GL.

Summary of Compatible 4GL Data Types

Some ordered pairs of 4GL data types are said to be *compatible*, in the sense that automatic data type conversion is possible for some non-null data values. As "Notes on Automatic Data Type Conversion" on page 3-48 indicates, however, whether conversion occurs without error is in many cases data-dependent, and is typically sensitive to the declared length, precision, or scale of the receiving data type.

The table that follows shows which pairs of 4GL data types are compatible.

These relationships apply to values of simple data types, to simple members of RECORD data types, and to simple elements of ARRAY data types:

- Unshaded cells show the types of values (listed in the top row) that 4GL can assign to each type of variable (listed on the left).
- Shaded cells indicate incompatible pairs of data types, for which 4GL does not support automatic data type conversion.

		Data Type of Value to Be Passed										
		CHAR	**VARCHAR**	**INTEGER**	**SMALLINT**	**FLOAT**	**SMALLFLOAT**	**DECIMAL**	**MONEY**	**DATE**	**DATETIME**	**INTERVAL**
Receiving Data Type	**CHAR**	①	①	①	①	①	①	①	①❶	①❷	①	①
	VARCHAR	①	①	①	①	①	①	①	①❶	①❷	①	①
	INTEGER	②③	②③			③ ④	③④	③④	③④	❹		
	SMALLINT	②③	②③	③		③ ④	③④	③④	③④	③❹		
	FLOAT	②③⑤	②③⑤	❸	❸			③	③	❹		
	SMALLFLOAT	②③⑤	②③⑤	❸⑤	❸	⑤		③ ⑤	③⑤	❹⑤		
	DECIMAL	②③⑥	②③⑥	③	③	③⑥	③⑥	③⑥	③⑥	③❹		
	MONEY	②③⑥	②③⑥	③	③	③⑥	③⑥	③⑥	③⑥	③❹		
	DATE	②	②	❹	❹	③④❹	③④❹	③④❹	③④❹		❺❼	
	DATETIME	②	②							❻❼	⑦❼	
	INTERVAL	②	②									③⑦

Symbols in the cells refer to notes in the next section. These apply when the data types of the passed value and of the receiving variable are not identical:

- Light circles (①) indicate the possibility of conversion failure, or discrepancies between the passed value and the receiving data type.
- Dark circles (❶) mark features that do not usually cause conversion errors, but that can produce unexpected data formats or values.

Notes on Automatic Data Type Conversion

In the previous table, numbers within light circles (①) indicate restrictions that can cause the data type conversion to fail, or that can sometimes result in discrepancies between the passed value and the receiving variable.

① If the result of converting a value to a character string is longer than the receiving variable, the character string is truncated from the right.

② Character string values must depict a literal of the receiving data type.

③ If the value exceeds the range of the receiving data type, an overflow error occurs.

④ Any fractional part of the value is truncated.

⑤ If the passed value contains more significant digits than the receiving data type supports, low-order digits are discarded.

⑥ If the passed value contains more fractional digits than the receiving data type supports, low-order digits are discarded.

⑦ Differences in qualifiers can cause truncation from the left or right.

Numbers in dark circles (❶) indicate less critical conversion features. These can result in the assignment of unexpected values, or unexpected formats.

❶ **DBMONEY** or **DBFORMAT** controls the format of the converted string.

❷ **DBFORMAT**, **DBDATE**, or **GL_DATE** controls the format of the result.

❸ Rounding errors can produce an assigned value with a fractional part.

❹ An integer value corresponding to a count of *days* is assigned.

❺ An implicit EXTEND (*value*, YEAR TO DAY) is performed by 4GL.

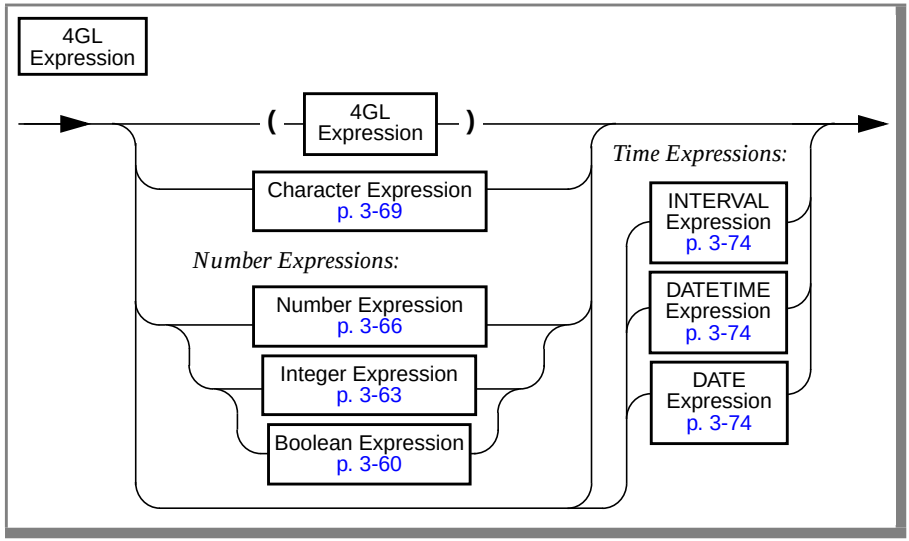
❻ The DATE becomes a DATETIME YEAR TO DAY literal before assignment.

❼ If the passed value has less precision than the receiving variable, any missing time unit values are obtained from the system clock.

You may wish to avoid writing code that applies automatic conversion to DATETIME variables declared with time units smaller than YEAR as the first keyword of the qualifier unless default values that feature (❼) assigns from the system clock are useful in your application. For more information, see Chapter 2, "The INFORMIX-4GL Language."

Expressions of 4GL

A 4GL *expression* is a sequence of operands, operators, and parentheses that 4GL can evaluate as a single value.



Usage

Statements, functions, form specifications, operators, and expressions can have expressions as arguments, components, or operands. The context where an expression appears, as well as its syntax, determines the data type of its returned value. It is convenient to classify 4GL expressions into the following five categories, based on the data type of the value that they return.

Expression Type	What the Expression Returns
Boolean	A value that is either TRUE or FALSE (or null in some contexts)
Integer	A whole-number value of data type INT or SMALLINT

(1 of 2)

Expression Type	What the Expression Returns
Number	A value of any of the number data types. For more information, see "Number Data Types" (page 3-10).
Character	A character string of data type CHAR or VARCHAR
Time	A value of data type DATE, DATETIME, or INTERVAL

(2 of 2)

In this manual, if the term *4GL expression* is not qualified as one of these five data type categories, the expression can be any 4GL data type.

As the diagram suggests, 4GL Boolean expressions are special cases of integer expressions, which in turn are a logical subset of number expressions. You can substitute a 4GL Boolean or integer expression where a number expression is valid (unless this results in an attempt to divide by zero).Topics that are discussed in this section include the following:

- "Components of 4GL Expressions" on page 3-52
- "Boolean Expressions" on page 3-60
- "Integer Expressions" on page 3-63
- "Number Expressions" on page 3-66
- "Character Expressions" on page 3-69
- "Time Expressions" on page 3-72

Differences Between 4GL and SQL Expressions

Expressions in SQL statements (and in SPL statements) are evaluated by the database server, not 4GL. The set of operators that can appear in SQL or SPL expressions resembles the set of 4GL operators, but they are not identical.

A 4GL program can include SQL operators, but these are restricted to SQL statements. Similarly, most SQL (and all SPL) operands are not valid in 4GL expressions. The SQL identifiers of databases, tables, or columns can appear in a LIKE clause or field name in 4GL statements (as described in "Field Clause" on page 3-86), provided that these SQL identifiers comply with the naming rules of 4GL, but the following SQL and SPL operands and operators cannot appear in other 4GL expressions:

- SQL identifiers, such as column names
- The names of SPL variables
- The SQL keywords USER and ROWID
- Built-in or aggregate SQL functions that are not part of 4GL
- The BETWEEN and IN operators (except in form specifications)
- The EXISTS, ALL, ANY, or SOME keywords of SQL expressions

Conversely, you *cannot* include the following 4GL operators in SQL or SPL expressions:

- Arithmetic operators for exponentiation (**) and modulus (MOD)
- String operators ASCII, COLUMN, SPACE, SPACES, and WORDWRAP
- Field operators FIELD_TOUCHED( ), GET_FLDBUF( ), and INFIELD( )
- The report operators LINENO and PAGENO
- The time operators DATE( ) and TIME

These and other built-in functions and operators of 4GL are described in Chapter 5, "Built-In Functions and Operators."

See the *Informix Guide to SQL: Syntax* for the syntax of SQL expressions and SPL expressions.

Components of 4GL Expressions

An expression of 4GL can include the following components:

- Operators, as listed on the next page
- Parentheses, to override the default precedence of operators
- Operands, including the following:
  - Named values
  - Function calls returning a single value
  - Field names
  - Literal values
  - Other 4GL expressions

Parentheses in 4GL Expressions

You can use parentheses as you would in algebra to override the default order of precedence of 4GL operators. In mathematics, this use of parentheses represents the "associative" operator. It is, however, a convention in computer languages to regard this use of parentheses as delimiters rather than as operators. (Do not confuse this use of parentheses to specify *operator precedence* with the use of parentheses to enclose arguments in function calls or to delimit other lists.)

In the following example, the variable **y** is assigned the value of 2:

```
LET y = 15 MOD 3 + 2
```

In the next example, however, **y** is assigned the value of 0 because the parentheses change the sequence of operations:

```
LET y = 15 MOD (3 + 2)
```

Chapter 4, "INFORMIX-4GL Statements," describes the LET statement of 4GL, which can assign the value of an expression to a variable of a compatible data type.

Operators in 4GL Expressions

The operators listed in Figure 3-2 on page 3-54 can appear in 4GL expressions. Expressions with several operators are evaluated according to their precedence, from highest (16) to lowest (1), as indicated in the left-most (P) column. In the previous example, although the modulus operator (MOD) had a higher precedence than the addition (+) operator, the parentheses instructed 4GL to perform the addition first, contrary to the default order of operator precedence.

The **P** values that indicate precedence in Figure 3-2 are ordinal numbers; they may change if future releases add new operators.

The fourth column (A) indicates the direction of associativity, if any, of each operator. See the page references in the right-most column of Figure 3-3 on page 3-55 for additional information about individual operators of 4GL.

Figure 3-2
Precedence (P) and Associativity (A) of 4GL Operators

P	Operator	Description	A	Example	Page
16	.	Record membership	Left	**myrec.memb**	3-35
	[ ]	Array index or substring	Left	**ar**[**i**,6,**k**][2,(*int-expr*)]	3-13
	( )	Function call	None	**myfun**(**var1**,*expr*)	3-58
15	UNITS	Single-qualifier interval	Left	(*int-expr*) UNITS DAY	5-26
14	+	Unary plus	Right	+ (*number-expr*)	3-65
	-	Unary minus	Right	**- numbarray_var3**[**i**,**j**,**k**]	3-65
13	**	Exponentiation (by integer)	Left	(*number-expr*) ** (*int-expr*)	3-64
	MOD	Modulus (of integer)	Left	(*int-expr*) MOD (*int-expr*)	3-64
12	*	Multiplication	Left	**x** * (*number-expr*)	3-64
	/	Division	Left	(*number-expr*) / **arr**[**y**]	3-64
11	+	Addition	Left	(*number-expr*) + (*number-expr*)	3-64
	-	Subtraction	Left	(**x** - **y**) - (*number-expr*)	3-64
10	||	Concatenation	Left	"str" || "ing"	5-50
9	LIKE	String comparison	Right	(*character-expr*) LIKE "%z_%"	5-38
	MATCHES	String comparison	Right	(*character-expr*)MATCHES"*z?"	5-38
8	<	*Test for*: less than	Left	(*expr1*) < (*expr2*)	5-35
	<=	Less than or equal to	Left	**x** <= **yourfun**(**y**,**z**)	5-35
	= *or* ==	Equal to	Left	**x** = *expr*	5-35
	>=	Greater than or equal to	Left	**x** >= FALSE	5-35
	>	Greater than	Left	**var1** > *expr*	5-35
	!= *or* <>	Not equal to	Left	**myrec.memb**<>LENGTH(**var1**)	5-35
7	IN( )	*Test for*: set membership	Right	*expr1* NOT IN (**x**,3,*expr2*)	5-40
6	BETWEEN ... AND	*Test for*: range	Left	BETWEEN (*integer-expr*) AND 9	5-40
5	IS NULL	*Test for*: NULL	Left	**x** IS NULL OR **y** IS NOT NULL	5-37
4	NOT	Logical inverse	Left	NOT ((*expr*) IN (**y**,DATE))	3-61
3	AND	Logical intersection	Left	*expr1* AND **fun**(*expr2*,-**y**)	3-61
2	OR	Logical union	Left	LENGTH(*expr1*,**j**) OR *expr2*	3-61
1	ASCII	Return ASCII character	Right	LET **x** = ASCII (*int-expr*)	5-31
	CLIPPED	Delete trailing blanks	Right	DISPLAY **poodle** CLIPPED	5-45
	COLUMN	Begin line-mode display	Right	PRINT COLUMN 58, "30"	5-47
	ORD	Logical inverse of ASCII	Right	LET **key** = ORD(*character-expr*)	5-100
	SPACES	Insert blank spaces	Right	DISPLAY (*int-expr*) SPACES	5-108
	USING	Format character string	Right	TODAY USING "yy/mm/dd"	5-123
	WORDWRAP	Multiple-line text display	Right	PRINT **odyssey** WORDWRAP	5-135

Figure 3-3
Data Types of Operands and of Returned Values

P	Expression	Left (= x)	Right (= y)	Returned Value	Page
16	**x . y**	RECORD	Any	Same as **y**	5-97
	w [x , y]	INT or SMALLINT	INT or SMALLINT	Any or Character	5-113
	(y)		Any or Large	Any	3-58
15	**x UNITS**	INT or SMALLINT		INTERVAL	5-119
14	**+ y**		Number or INTERVAL	Same as **y**	5-22
	- y		Number or INTERVAL	Same as **y**	5-22
13	**x * * y**	Number	INT or SMALLINT	Number	5-25
	x MOD y	INT or SMALLINT	INT or SMALLINT	INT or SMALLINT	5-25
12	**x * y**	Number or INTERVAL	Number	Number or INTERVAL	5-25
	x / y	Number or INTERVAL	Number	Number or INTERVAL	5-25
11	**x + y**	Number or Time	Number or Time	Number or Time	5-26
	x - y	Number or Time	Number or Time	Number or Time	5-26
10	**x || y**	Any	Any	Character	5-50
9	**x LIKE y**	Character	Character	Boolean	5-38
	x MATCHES y	Character	Character	Boolean	5-38
8	**x < y**	Any	Same as x	Boolean	5-34
	x <= y	Any	Same as x	Boolean	5-34
	x = y *or* **x == y**	Any	Same as x	Boolean	5-34
	x >= y	Any	Same as x	Boolean	5-34
	x > y	Any	Same as x	Boolean	5-34
	x != y *or* **x <> y**	Any	Same as x	Boolean	5-34
7	**x IN (y)**	Any	Same as **x**	Boolean	5-40
6	**BETWEEN x AND y**	Any	Same as x	Boolean	5-40
5	**x IS NULL**	Any or Large		Boolean	5-37
4	**NOT y**		Boolean	Boolean	5-33
3	**x AND y**	Boolean	Boolean	Boolean	5-33
2	**x OR y**	Boolean	Boolean	Boolean	5-33
1	**ASCII y**		INT or SMALLINT	Character	7-62
	x CLIPPED	Character		Character	5-45
	COLUMN y		INT or SMALLINT	Character	7-62
	ORD(y)		Character	INT or SMALLINT	5-100
	x SPACES	INT or SMALLINT		Character	7-64
	x USING "y"	Character,DATE,MONEY	Character	Character	5-123
	x WORDWRAP	Character or TEXT		Character	7-65

Also of lowest precedence (**P** = 1) are the following operators:

- Field operators FIELD_TOUCHED( ), GET_FLDBUF( ), and INFIELD( )
- Report operators SPACE, LINENO, and PAGENO
- Time operators CURRENT, DATE( ), DAY( ), EXTEND( ), MDY( ), MONTH( ), TIME, TODAY, WEEKDAY( ), and YEAR( )

Where no data type is listed in the previous table, the operator has no left (or else no right) operand. If an operand is not of the data types listed here, 4GL attempts data type conversion, as described in "Summary of Compatible 4GL Data Types" on page 3-46.

Most 4GL operators do not support RECORD nor ARRAY operands, but they can accept as an operand a variable (of a simple data type) that is an element of an array, or that is a member of a record.

Operands in 4GL Expressions

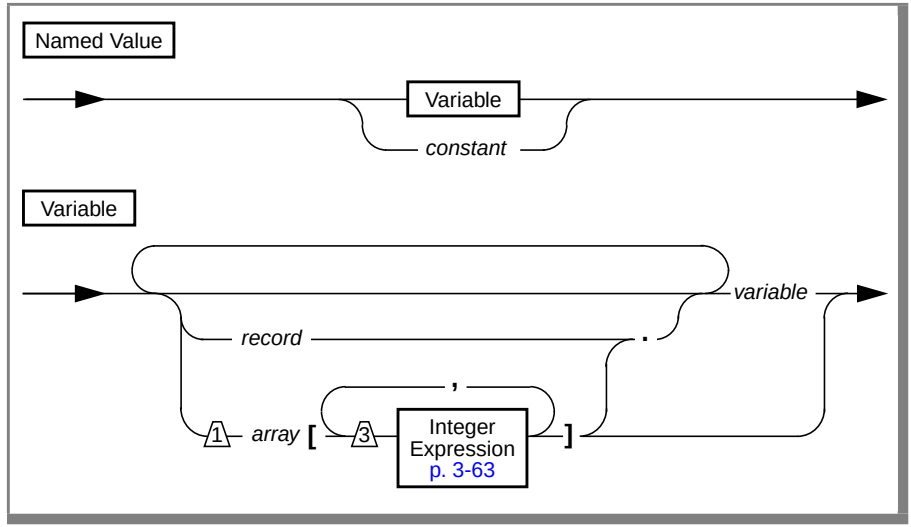
Operands of 4GL expressions can be any of the following:

- Named values
- Function calls that return one value
- Literal values
- Other 4GL expressions

Sections that follow describe these operands of 4GL expressions.

Named Values as Operands

A 4GL expression can include the name of a variable of any simple data type (as identified in "Simple Data Types" on page 3-9) or the constants TRUE, FALSE, or NOTFOUND. The variable can also be a simple member of a record or a simple element of an array.



Element	Description
array	is the name of a structured variable of the ARRAY data type. The comma-separated expression list specifies the index of an element within the declared size of the array.
constant	is one of the built-in constants TRUE, FALSE, or NOTFOUND.
record	is the name of a structured variable of the RECORD data type.
variable	is the name of a 4GL program variable of a simple data type.

In three special cases, other identifiers can be operands in 4GL expressions:

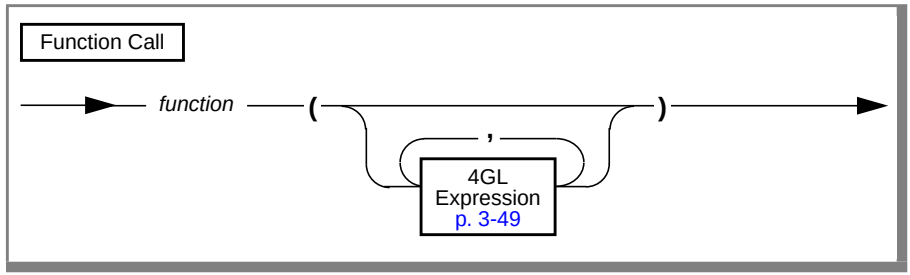
- Conditional COLOR attributes in form specification files can use a *field tag* where a named value is valid in the syntax of a 4GL Boolean expression.
- The built-in FIELD_TOUCHED( ), GET_FLDBUF( ), and INFIELD( ) operators can take field names (as described in "Field Clause" on page 3-86) as operands. See Chapter 5, "Built-In Functions and Operators," for the syntax of these field operators.
- The identifier of a BYTE or TEXT variable can be the operand of the IS NULL and IS NOT NULL Boolean operators.

If the variable is a member of a record, qualify it with the record name prefix, separated by a period (.) as the record membership operator.

Variables of the BYTE or TEXT data types cannot appear in expressions, except as operands of the IS NULL or IS NOT NULL operators or (for TEXT variables only) the WORDWRAP operator. These operators are described in Chapter 5.

Function Calls as Operands

A 4GL expression can include calls to functions that return exactly one value.



Element	Description
function	is the name of a function. The parentheses are required, regardless of whether the function takes any arguments.

The function can be a programmer-defined or built-in function, provided that it returns a single value of a data type that is valid in the expression. (Function calls as arguments can return multiple values.)

For information about this statement, see "FUNCTION" on page 4-140, and see Chapter 5 for more information about declaring, defining, and invoking 4GL functions.

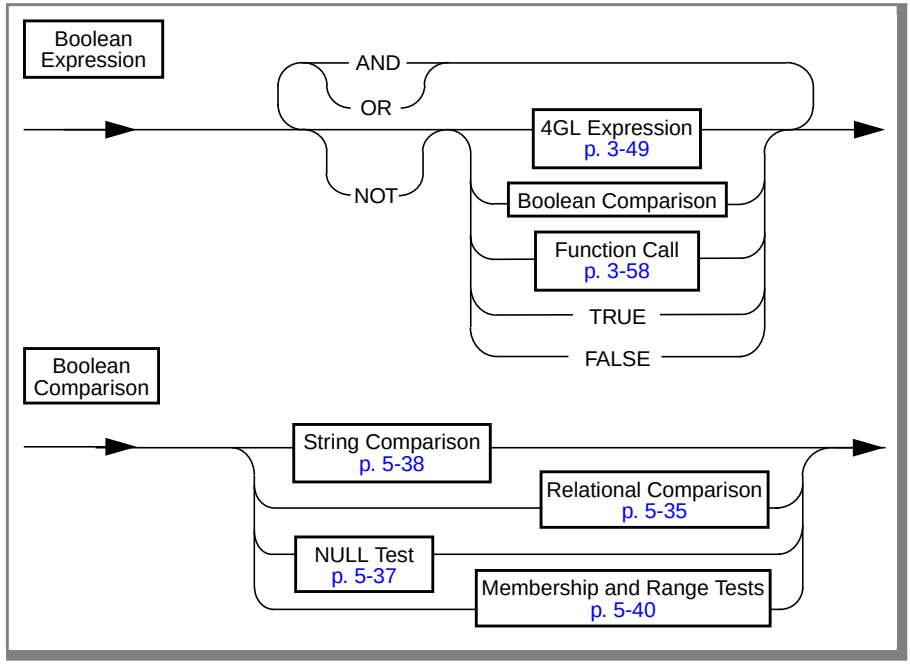
Expressions as Operands

Two expressions cannot appear consecutively without some separator, but you can nest expressions within expressions. In any context, however, the complexity of a 4GL expression is restricted. If an error message indicates that an expression is too complex, you should substitute two or more simpler expressions that 4GL can evaluate, and then combine these values.

If an expression returns a different data type from what 4GL expects in the context, 4GL attempts data type conversion, as described in "Summary of Compatible 4GL Data Types" on page 3-46.

Boolean Expressions

In 4GL, a *Boolean expression* is one that returns either TRUE (defined as 1) or FALSE (defined as 0) or (in some contexts) null. The syntax of Boolean expressions in 4GL statements is not identical to that of *Boolean conditions* in SQL statements. Boolean expressions of 4GL have the following syntax.



Logical Operators and Boolean Comparisons

The following *Boolean operators* can appear in 4GL Boolean expressions:

- The *logical operators* AND, OR, and NOT combine one or more Boolean values into a single Boolean expression.
- *Boolean comparisons* can test operands and return Boolean values:
  - Relational comparisons test for equality or inequality.
  - The IS NULL operator tests for null values.
  - The MATCHES or LIKE operators compare *character strings*.
  - The BETWEEN...AND operator compares a value to a range.
  - The IN( ) operator tests a value for set membership.

Data Type Compatibility

Any type of 4GL expression can also be a Boolean expression. You can use an INT or SMALLINT variable to store the returned TRUE, FALSE, or NULL value.

You may get unexpected results, however, from Boolean comparisons of operands of dissimilar data types. In general, you can compare numbers with numbers, character strings with strings, and time values with time values.

If a *time expression* operand of a Boolean expression is of the INTERVAL data type, any other time expression to which it is compared by a relational operator must also return an INTERVAL value. You cannot compare a span of time (an INTERVAL value) with a point in time (a DATE or DATETIME value). See "Summary of Compatible 4GL Data Types" on page 3-46 for additional information about data type compatibility in 4GL expressions.

Evaluating Boolean Expressions

In contexts where a Boolean expression is expected, 4GL applies the following rules after it evaluates the expression:

- If the value is a non-zero real number *or* any of the following items:
  - Character string representing a non-zero number
  - Non-zero INTERVAL
  - Any DATE or DATETIME value
  - A TRUE value returned by a Boolean function like INFIELD( )
  - The built-in integer constant TRUE

  then the Boolean expression returns TRUE.
- If an expression that returns NULL is the operand of the IS NULL operator, the value of the Boolean expression is TRUE.
- If the value is NULL *and* the expression does not appear in any of the following contexts:
  - The NULL Test (page 5-37)
  - Boolean Comparisons (page 5-34)
  - Any conditional statement of 4GL (IF, CASE, WHILE)

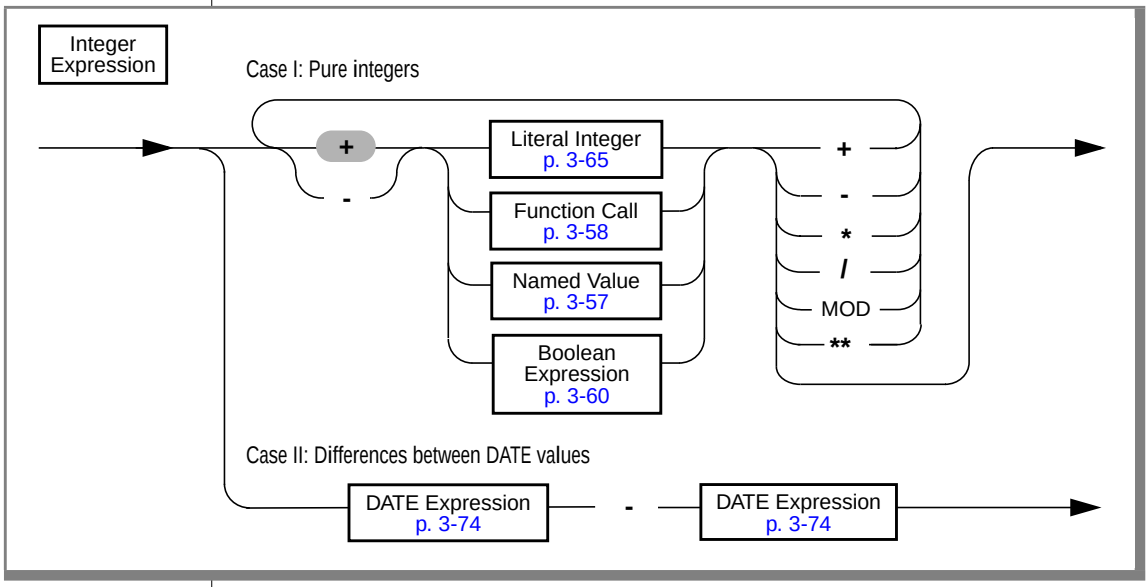
  then the Boolean expression returns NULL.
- Otherwise, the Boolean expression is evaluated as FALSE.

Boolean expressions in CASE, IF, or WHILE statements return FALSE if any element of the comparison is NULL, except for operands of the IS NULL and the IS NOT NULL operator. See "Boolean Operators" on page 5-33 for more information about individual Boolean operators and Boolean expressions.

If you include a Boolean expression in a context where 4GL expects a number, the expression is evaluated, and is then converted to an integer by the rules: TRUE = 1 and FALSE = 0.

Integer Expressions

An *integer expression* returns a whole number. It has the following syntax.



Here any *function call* or *named value* must return an integer. Logical restrictions on using DATE values as integer expressions are discussed in "Arithmetic Operations on Time Values" on page 3-83.

Integer expressions can be components of expressions of every other type. Like Boolean expressions, integer expressions are a logical subset of *number* expressions, but they are separately described here because some 4GL operators, statements, form specifications, operators, and built-in functions are restricted to integer values, or to positive integers.

Binary Arithmetic Operators

Six binary arithmetic operators can appear in an integer expression, and can take integer expressions as both the right-hand and left-hand operands.

Operator Symbol	Operator Name	Name of Result	Precedence
**	Exponentiation	Power	12
mod	Modulus	Integer remainder	12
*	Multiplication	Product	11
/	Division	Quotient	11
+	Addition	Sum	10
-	Subtraction	Difference	10

All arithmetic calculations are performed after converting both operands to DECIMAL values (but MOD operands are first converted to INTEGER).

If an expression has several operators of the same precedence, 4GL processes them from left to right. For the complete precedence scale for 4GL operators, see Figure 3-2 on page 3-54. If any operand of an arithmetic expression is a NULL value, the entire expression returns NULL.

An integer expression specifying an array element or the right-hand MOD operand cannot include exponentiation (**) or modulus (MOD) operators, and cannot be zero. The right-hand integer expression operand of the exponentiation operator (**) cannot be negative. You cannot use **mod** as a 4GL identifier.

If both operands of the division operator (/) have INT or SMALLINT data types, 4GL discards any fractional portion of the quotient. An error occurs if the right-hand operand of the division operator evaluates to zero. With some restrictions, 4GL also supports these binary arithmetic operators in number expressions (as described in "Number Expressions" on page 3-66) and in some time expressions (as described in "Time Expressions" on page 3-72).

Differences between two DATE values are integer expressions. To convert these to type INTERVAL, apply the UNITS DAY operator explicitly.

As noted earlier in this chapter, if you include a Boolean expression in a context where 4GL expects a number, the expression is evaluated, and is then converted to an integer by the rules: TRUE = 1 and FALSE = 0. An error results if you attempt to divide by zero.

Unary Arithmetic Operators

You can use plus (+) and minus (-) symbols at the left are *unary operators* to indicate the sign of the expression, or the sign of a component number. For unsigned values, the default is positive (+). Use parentheses to separate the subtraction operator (-) from any immediately following unary minus sign, as in the following:

minuend -(-*subtrahend*)

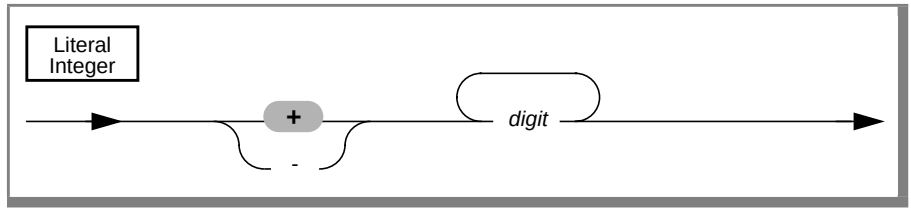
unless you want 4GL to interpret the "- -" symbols as a comment indicator.

The same rules apply to plus and minus unary operators used with number expressions, and with time expressions that return INTERVAL values.

The unary plus and minus operators are recursive.

Literal Integers

You must write *literal integers* in base-10 notation, without embedded blank spaces or commas, and without a decimal point.



Element	Description
digit	is any of the digits 1, 2, 3, 4, 5, 6, 7, 8, 9, or 0.

GLS

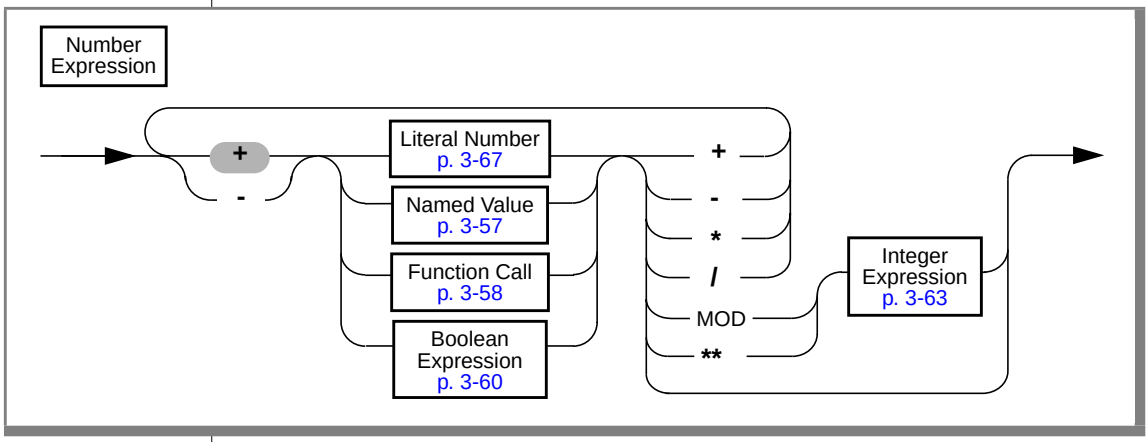
This release of 4GL does not support non-ASCII digits in number expressions, such as the Hindi numbers that some Middle-Eastern locales recognize. ♦

You can precede the integer with unary minus or plus signs:

```
15        -12        13938        +4
```

Number Expressions

A *number expression* is a specification that evaluates to a real number.



Here the function call or named value must return a real number of data type DECIMAL, FLOAT, INTEGER, MONEY, SMALLFLOAT, or SMALLINT.

If any operand of an arithmetic operator in a number expression is a null value, 4GL evaluates the entire expression as a NULL value. The range of values in a number expression is that of the receiving data type.

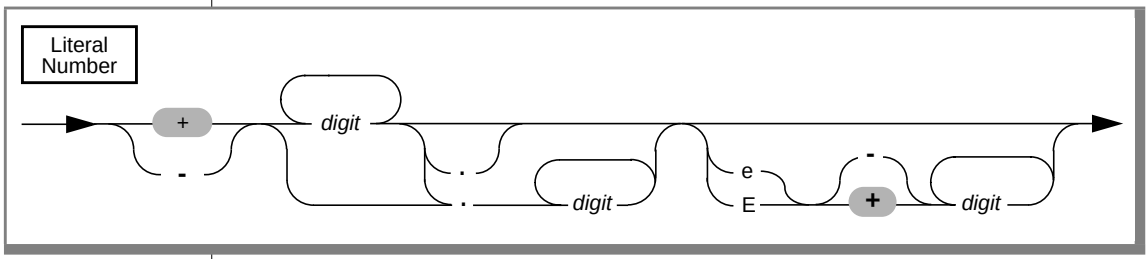
Arithmetic Operators

The sections "Binary Arithmetic Operators" on page 3-64 and "Unary Arithmetic Operators" on page 3-65 apply to number expressions. 4GL converts any modulus (MOD) operand or right-hand operand of the exponentiation operator (**) to INTEGER before conversion to DECIMAL for evaluation; this feature has the effect of discarding any fractional part of the operands.

If both operands are INTEGER, SMALLINT, or DATE data types, the result of any arithmetic operation (including division) is a whole number. If either operand is of data type DECIMAL, FLOAT, MONEY, or SMALLFLOAT, the returned value may include a fractional part, except in MOD operations.

Literal Numbers

A *literal number* is the base-10 representation of a real number, written as an integer, as a fixed-point decimal number, or in exponential notation.



Element	Description
digit	is any of the digits 1, 2, 3, 4, 5, 6, 7, 8, 9, or 0.

This cannot include a comma (,) or blank space (ASCII 32). The unary plus or a minus sign can precede a literal number, mantissa, or exponent.

GLS

This release of 4GL does not support non-ASCII digits in literal numbers, such as the Hindi numbers that some Middle-Eastern locales recognize. ♦

There are three kinds of literal numbers:

- Integer literals can exactly represent INTEGER and SMALLINT values. Literal integers have no decimal points, as in this example:

```
10    -27   25567
```

- Fixed-point decimal literals can exactly represent DECIMAL(*p*,*s*) and MONEY values. These can include a decimal point:

```
123.456.00123456  -123456.0
```

- Floating-point literals can exactly represent FLOAT, SMALLFLOAT, and DECIMAL(*p*) values that contain a decimal point or exponential notation, or both. These are examples of floating-point literals:

```
123.456e4   -1.23456e2  -123456.0e-3
```

When you use a literal number to represent a MONEY value, do not precede it with a currency symbol. Currency symbols are displayed by 4GL when MONEY values appear in a form or in a report, using whatever the **DBMONEY** or **DBFORMAT** environment variable specifies, or else the default symbol, which in the default (U.S. English locale) is the dollar sign ($).

GLS

In other locales, **DBMONEY** or **DBFORMAT** can specify number and currency display and data entry formats to conform with local cultural conventions. ♦

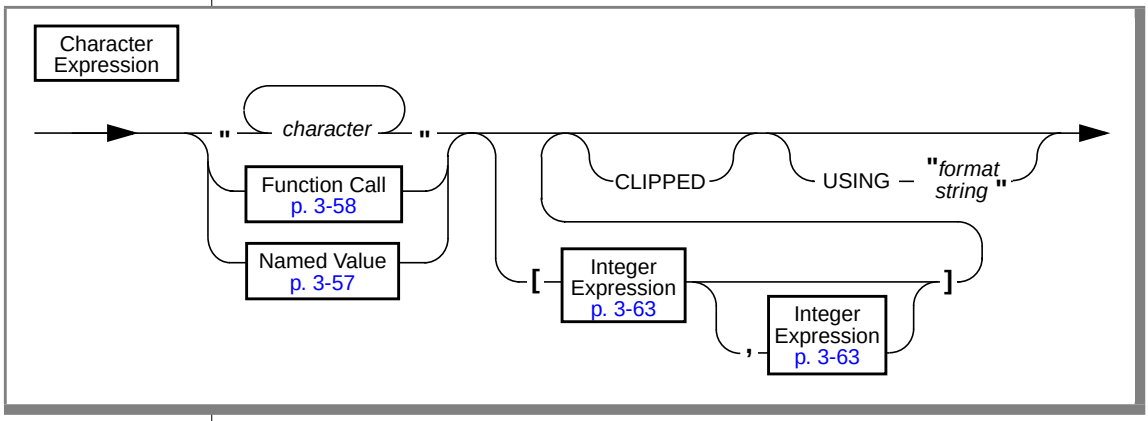
4GL automatically attempts data type conversion when a literal number is in a different format from the expected data type. If you include a character value in a context that requires a number expression, 4GL attempts to convert the string to a number. (For more information, see "Summary of Compatible 4GL Data Types" on page 3-46.)

You may get unexpected results, however, if a literal number in a 4GL Boolean expression is not in a format that can exactly represent the data type of another value with which it is compared by a relational operator. Because of rounding errors, for example, relational operators generally cannot return TRUE if one operand returns a FLOAT value and the other an INTEGER.

Similarly, you will get unpredictable (but probably useless) results if you use literal binary, hexadecimal, or other numbers that are not base-10 where 4GL expects a number expression. You must convert such numbers to a base-10 format before you can use them in a number expression.

Character Expressions

A *character expression* is a specification that evaluates to a character string.



Element	Description
character	is one or more characters enclosed between two single (') or double (") quotation marks. (This is sometimes called a character string, a quoted string, or a string literal.)
format string	is a formatting mask to specify how 4GL displays the returned character value. For details, see "USING" on page 5-123.

Here the function call or named value returns a CHAR or VARCHAR value. No variable in a character expression can be of the TEXT data type, except in a NULL test (as described in "The NULL Test" on page 5-37), or as a WORDWRAP operand in a PRINT statement of a 4GL report. As in any 4GL statement or expression, you cannot reference a named value outside its scope of reference. (See "Scope of Reference of 4GL Identifiers" on page 2-17.)

If a character expression includes a 4GL variable or function whose value is neither a CHAR nor VARCHAR data type, 4GL attempts to convert the value to a character string. For example, the following program fragment stores the character string "FAX32" in the CHAR variable **K**:

```
VARIABLE I INTEGER,
   J, K  CHAR(5)
LET I = 4*8
LET J = "FAX"
LET K =  J CLIPPED,I
```

The maximum length of a string value is the same as for the declared data type: up to 32,767 bytes for CHAR values, and up to 255 bytes for VARCHAR.

If character expressions are operands of a relational operator, 4GL evaluates both character expressions, and then compares the returned values according to their position within the collating sequence of the locale. For more information, see "Relational Operators" on page 5-35.

Arrays and Substrings

Any integer expression in brackets that follows the name of an array must evaluate to a positive number within a range from 1 to the declared size of the array. For example, **SQLCA.SQLCAWARN[6]** specifies the sixth element of character array **SQLCAWARN** within the **SQLCA** global record.

The pair of integer expressions that can follow a character expression specify a substring. The first value cannot be larger than the second. Both must be positive, and no larger than the string length (or the receiving data type). For example, **name[1,4]** specifies the first four characters of a program variable called **name**.

Neither the exponentiation (**) nor the modulus (MOD) operators can appear in an integer expression that specifies an array element or a substring, but parentheses and the other arithmetic operators (+, -, *, /) are permitted.

String Operators

You can use the USING keyword, followed by a format string, to impose a specific format on the character string to which an expression evaluates, or upon any components of a concatenated character expression. (4GL forms and reports support additional features for formatting character values.)

To discard trailing blanks from a character value, apply the CLIPPED operator to the expression, or to any components of a concatenated character expression. For more information about handling blank characters in character values, see the sections of Chapter 5 that describe the WORDWRAP field attribute in forms and the WORDWRAP operator in 4GL reports, and see "The WORDWRAP Operator" on page 7-65.

You can insert blanks in DISPLAY or PRINT statements by using the SPACE or COLUMN operators; these are described in Chapter 5. The keyword SPACES is a synonym for SPACE.

You can use the ASCII operator in DISPLAY or PRINT statements. This takes an integer expression as its operand, and returns a single-character string, corresponding to the specified ASCII character. See Chapter 5 for details.

Non-Printable Characters

In the default (U.S. English) locale, 4GL regards the following as the *printable* ASCII characters:

- TAB (= CONTROL-I)
- NEWLINE (= CONTROL-J)
- FORMFEED (= CONTROL-L)
- ASCII 32 (= blank) through ASCII 126 (= ~)

For information about the ASCII characters and their numeric codes, see Appendix G, "Reserved Words." Any other characters are *non-printable*. Character strings that include one or more non-printable characters (for example, packed fields) can be operands or returned values of character expressions. They can be stored in 4GL variables or in database columns of the CHAR, VARCHAR, and TEXT data types.

You should be aware, however, that many 4GL features for manipulating character strings were designed for printable characters only. If you create 4GL applications that use character expressions, character variables, or character columns to manipulate non-printable characters, you may encounter unexpected results. The following are examples of problems that you risk when CHAR, TEXT, and VARCHAR values include non-printable characters.

- Behavior of I/O and formatting features like the WORDWRAP attribute or the DISPLAY or PRINT statements is designed and documented for printable characters only. It may be difficult to describe or to predict the effects of data with non-printable characters with these I/O features, but the users of your application are unlikely to enjoy the results.
- Strings with non-printable characters can have unpredictable results when output to I/O devices. For example, some sequences of non-printable characters can cause terminals to position the cursor in the wrong place, clear the display, modify terminal attributes, or otherwise make the screen unreadable.

- For another example, CONTROL-D (= ASCII 4) and CONTROL-Z (= ASCII 26) in output from a report can be interpreted as logical end-of-file, causing the report to stop printing prematurely.
- If you store a zero byte (ASCII 0) in a CHAR or VARCHAR variable or column, it might be treated as a string terminator by some operators, but as data by others, and this behavior might vary between the Rapid Development System and the C Compiler implementation of 4GL, or even between database servers. The workaround is to not use the ASCII 0 character within CHAR or VARCHAR data strings.

If you encounter these or related difficulties in processing non-printable characters, you might consider storing such values as BYTE data types.

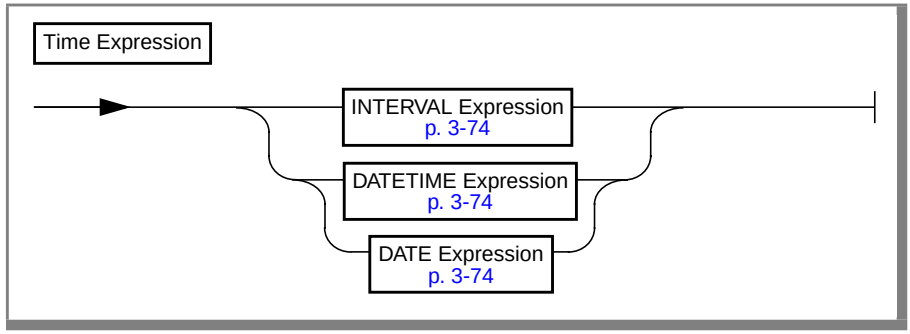
GLS

Nondefault locales can define other non-printable characters. The **DBAPICODE** environment variable lets computer peripherals that use a character set that is different from that of the database communicate with the database. **DBAPICODE** specifies the character-mapping file between the character set of the peripheral device and the character set of the database.

For more information about nondefault locales, see Appendix E. ♦

Time Expressions

A *time expression* is a specification that 4GL can evaluate as a DATE, DATETIME, or INTERVAL value.



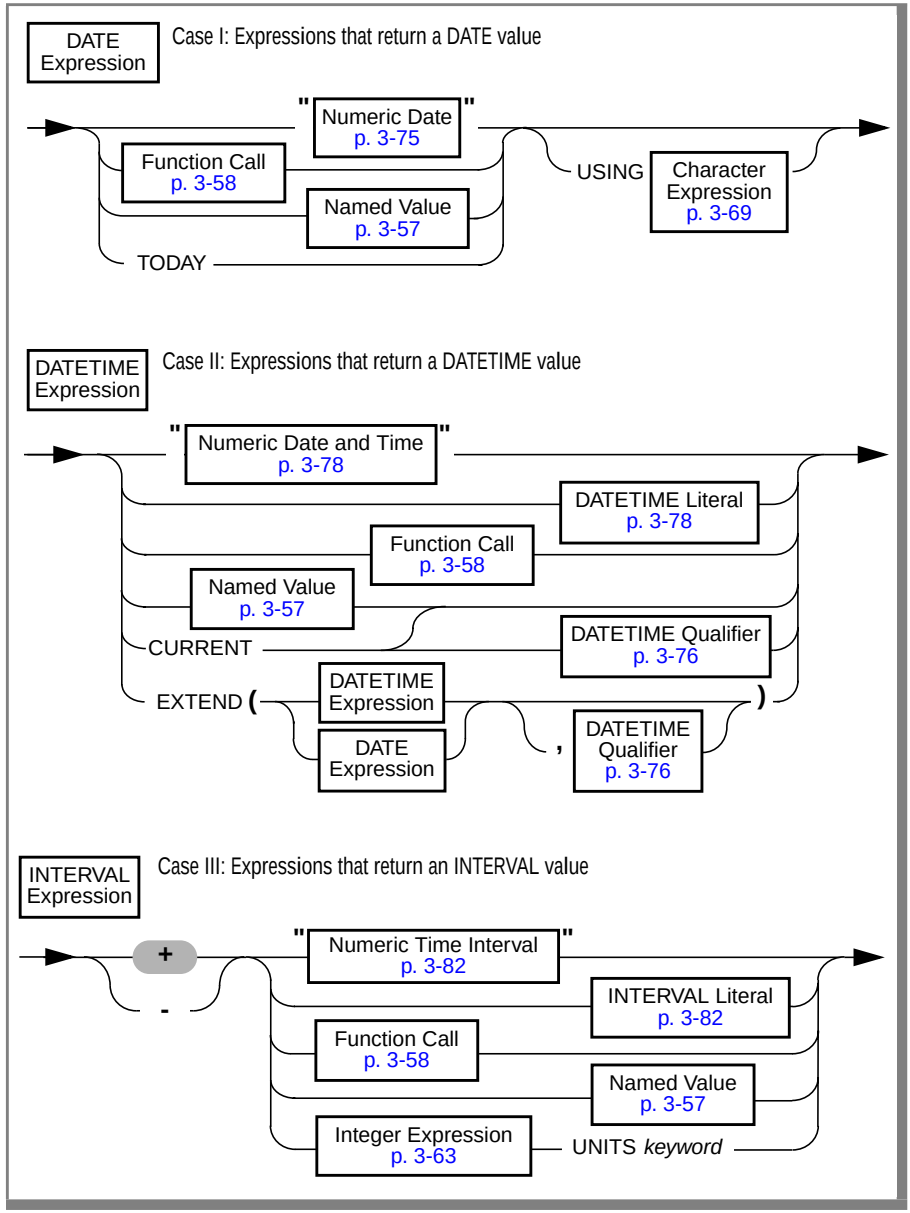
As the diagram suggests, the DATE data type is a logical subset of DATETIME. 4GL rules for arithmetic, however, are not identical for DATE and DATETIME operands, and the internal storage formats of DATE and DATETIME values are completely different. (For more information, see "Arithmetic Operations on Time Values" on page 3-83.)

Formatting features, such as USING (described in Chapter 5) and the FORMAT and PICTURE attributes (described in Chapter 6, "Screen Forms"), also treat DATETIME and DATE values differently or support only DATE.

These three data types are logically related, because they express values in units of time. But unlike the number and character data types that were described earlier in this chapter, for which 4GL supports automatic data type conversion (aside from restrictions based on truncation, overflow, or underflow), conversion among time data types is more limited. In contexts where a time expression is required, DATETIME or DATE values can sometimes be substituted for one another. INTERVAL values, however, which represent one-dimensional spans of time, cannot be converted to DATETIME or DATE values, which represent zero-dimensional points in time.

In addition, if the declared precision of an INTERVAL value includes years or months, automatic conversion to an INTERVAL having smaller time units (like days, hours, minutes, or seconds) is not available. See also "Summary of Compatible 4GL Data Types" on page 3-46.

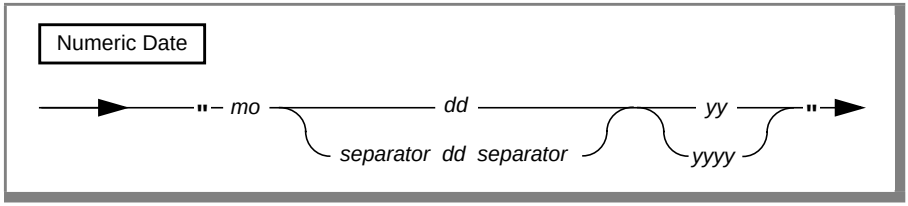
Each of the three types of time expressions has its own syntax.



Here the *keyword* can be YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION. The *function call* or *named value* must return a single value of the corresponding data type. Other operators besides those listed here can also appear in time expressions. Chapter 5 describes built-in operators like UNITS. Sections that follow show the syntax of component segments of DATE, DATETIME, and INTERVAL expressions.

Numeric Date

A *numeric date* represents a DATE value as a quoted string of digits and optional separator symbols. For the default locale, it has this format.



Element	Description
dd	is the number of the day of the month, from 1 to 31.
mo	is a number from 1 to 12, representing the month.
separator	is any character that is not a digit.
yy	is an integer from 1 to 99, abbreviating the year. If only a single digit is supplied, 4GL automatically prefixes it with a leading zero.
yyyy	is an integer from 1 to 9999, representing the year.

The digits must represent a valid calendar date. The default locale supports 6 digits (*moddyy*) or 8 digits (*moddyyyy*), with blank, slash (/), hyphen (-), no symbol, or any character that is not a digit as *separator*. Here *mo*, *dd*, and *yyyy* have the same meanings as in "DATETIME Literal" on page 3-78.

The **DBDATE** environment variable can change the order of time units and can specify other separators. Like the USING operator or the FORMAT field attribute, **DBDATE** can also specify how 4GL displays DATE values.

GLS

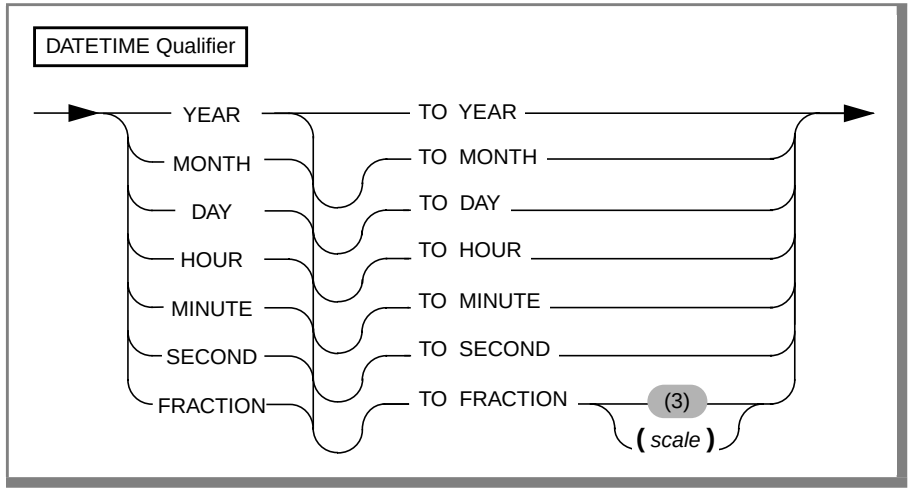
In some East-Asian locales, the **GL_DATE** environment variable can specify Japanese or Taiwanese eras for the entry and display of DATE values. (In any locale, **GL_DATE** can specify formats beyond what **DBDATE** can specify.) ♦

The **DBCENTURY** environment variable, or CENTURY attribute, determines how to expand abbreviated *year* values in DATE (or DATETIME) fields of 4GL forms. These features can also expand a two-digit *year* in a PROMPT statement.

If you omit the quotation marks where a DATE value is expected, 4GL attempts to evaluate your specification as a literal integer or as an integer expression specifying a count of days since December 31, 1899. If slash (/) is the *separator*, the quotient of *month* and *day* is divided by the *year* value, producing a value that usually rounds to zero, or December 31, 1899. This result may not be useful, if the logical context requires a more recent date.

DATETIME Qualifier

The *DATETIME qualifier* specifies the precision and scale of a DATETIME value. It has the same syntax as the qualifiers of DATETIME database columns.



Element	Description
scale	is an integer ($1 \leq scale \leq 5$), enclosed between parentheses.

Specify the largest time unit in the DATETIME value as the first keyword. After the TO, specify the smallest time unit as the last keyword. These time units can be recorded in the numeric DATETIME value.

YEAR	is a year; in numeric values, this can range from 1 to 9999.
MONTH	is a month, ranging from 1 to 12.
DAY	is a day, ranging from 1 to 31, as appropriate to its month.
HOUR	is an hour, ranging from 0 (midnight) to 23.
MINUTE	is a minute, ranging from 0 to 59.
SECOND	is a second, ranging from 0 to 59.
FRACTION	is a fraction of a second, with up to 5 decimal places.

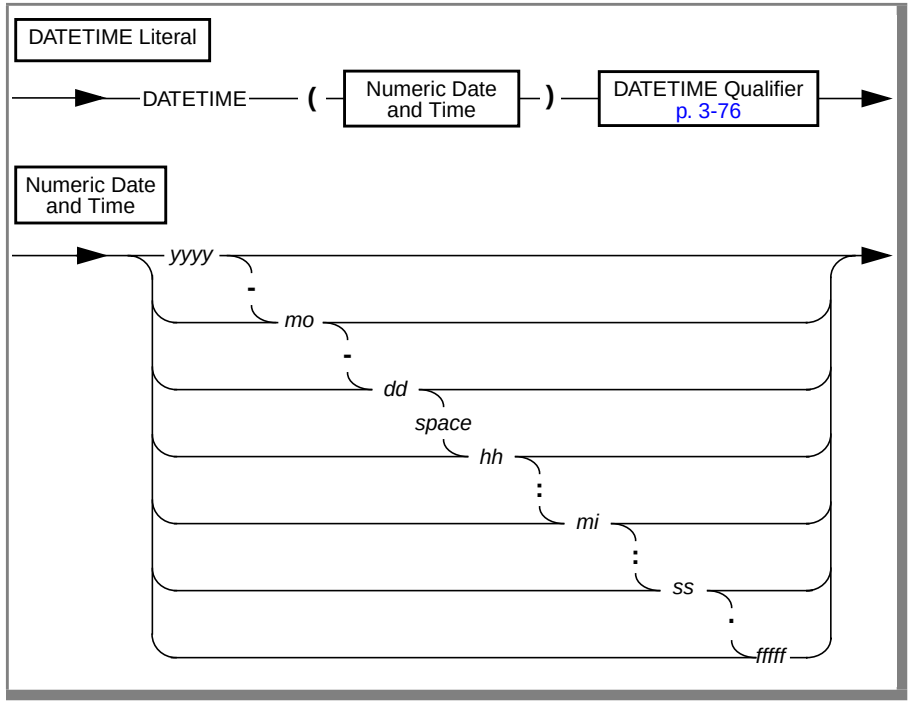
Unlike INTERVAL qualifiers, DATETIME qualifiers cannot specify nondefault precision (except for FRACTION when it is the smallest unit in the qualifier). Here are some examples of DATETIME qualifiers:

```
YEAR TO MINUTE          MONTH TO MONTH
DAY TO FRACTION(4)      MONTH TO DAY
```

An error results if the first keyword represents a smaller time unit than the last, or if you use the plural form of a keyword (such as "MINUTES").

DATETIME Literal

A *DATETIME literal* is the representation of a DATETIME value as the numeric date and time, or a portion thereof, followed by a DATETIME qualifier.



Element	Description
dd	is the number of the day of the month, from 1 to 31.
fffff	is the fraction of a second, up to 5 digits, as set by the precision specified for the FRACTION time units in the DATETIME qualifier.
hh	is the hour (from a 24-hour clock), from 0 (midnight) to 23.
mi	is the minute of the hour, from 0 to 59.
mo	is a number from 1 to 12, representing the month.
space	is a blank space (ASCII 32), entered by pressing SPACEBAR.
ss	is the second of the minute, from 0 to 59.
yyyy	is a number from 1 to 9999, representing the year. If it has fewer than 3 digits, 4GL expands the year value to 4 digits (according to the **DBCENTURY** setting, unless the CENTURY attribute is specified).

An error results if you omit any required separator or include values for units outside the range specified by the qualifier. Here are some examples:

```
DATETIME (99-3-6) YEAR TO DAY
DATETIME (09:55:30.825) HOUR TO FRACTION
DATETIME (01-5) YEAR TO MONTH
```

Here is an example of a DATETIME literal used in an arithmetic expression as an operand of the EXTEND operator:

```
EXTEND (DATETIME (2000-8-1) YEAR TO DAY, YEAR TO MINUTE)
   - INTERVAL (720) MINUTE (3) TO MINUTE
```

The **DBCENTURY** environment variable setting determines how single-digit and two-digit year specifications in DATETIME fields of 4GL forms are expanded. For more information, see Appendix D, "Environment Variables."

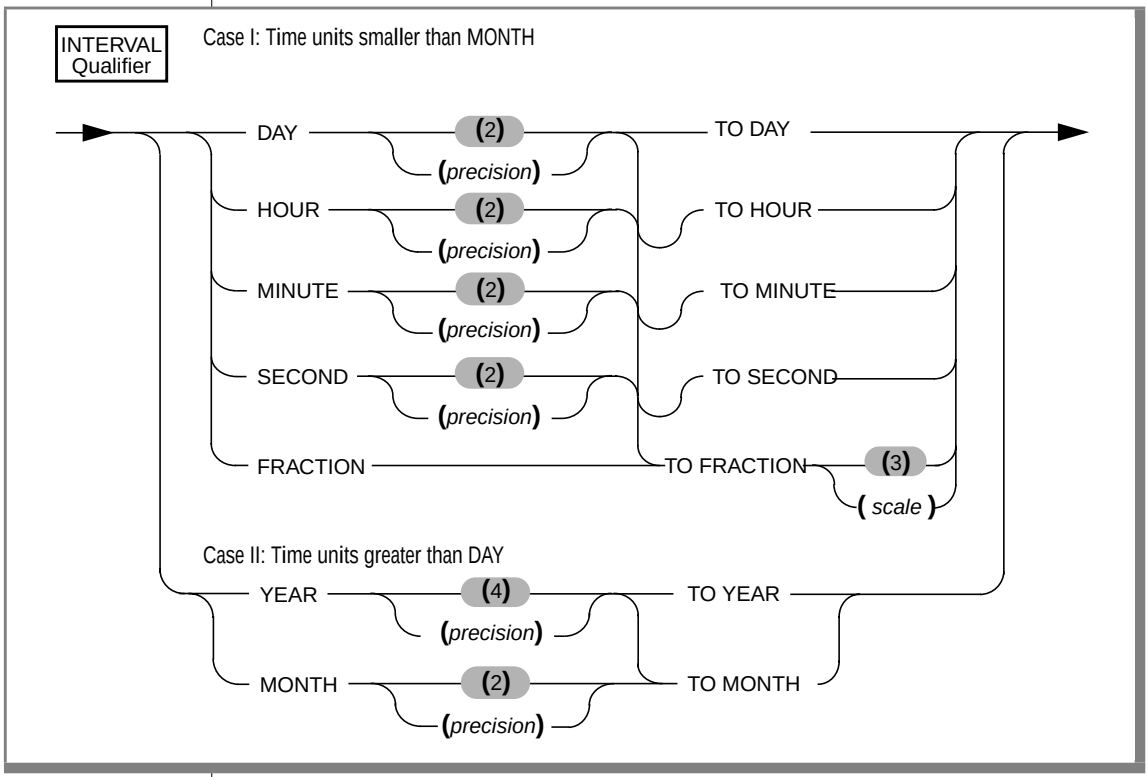
The CENTURY attribute of 4GL forms can specify the same algorithms as **DBCENTURY** for expanding abbreviated year values. Unlike **DBCENTURY** , however, which specifies a single default algorithm for the entire application, CENTURY specifies the expansion rule for a single field. If the two settings are different, the CENTURY setting takes precedence (within its field) over the **DBCENTURY** setting. For more information, see "DBCENTURY" on page D-15 and "CENTURY" on page 6-35. The PROMPT statement also supports a CENTURY attribute for DATETIME (or DATE) values that the user enters with the year abbreviated.

GLS

The **GL_DATETIME** environment variable can specify display and data entry formats for DATETIME values to conform with local cultural conventions, such as Japanese or Taiwanese eras for *year* values in some East-Asian locales. ♦

INTERVAL Qualifier

The *INTERVAL qualifier* specifies the precision and scale of an INTERVAL value. It has the same syntax in 4GL as for INTERVAL database columns.



Element	Description
scale	is the number of decimal digits to record fractions of a second in a span of time. The default is 3 digits; the maximum is 5.
precision	is the number of digits in the largest time units that the interval can include. The maximum number of digits is 9; the default is 2 (except for the number of years, whose default precision is 4).

Any intermediate time units between the first and last keywords that you specify in an INTERVAL qualifier have the default precision of 2 digits.

If the INTERVAL value can include more than one different time unit, specify the largest time unit in the INTERVAL as the first keyword. After the TO, specify the smallest time unit as the last keyword. If the first time unit keyword is YEAR or MONTH, the last cannot be smaller than MONTH.

The following examples of an INTERVAL qualifier are both YEAR TO MONTH. The first example can record a span of up to 999 years, because 3 is the precision of the YEAR units. The second example uses the default precision for the YEAR units; it can record a span of up to 9999 years and 11 months.

```
YEAR (3) TO MONTH
YEAR TO MONTH
```

When you intend for a value to contain only one kind of time unit, the first and last keywords in the qualifier are the same. For example, an interval of whole years that is qualified as YEAR TO YEAR can record a span of up to 9999 years, the default precision. Similarly, the qualifier YEAR (4) TO YEAR can record a span of up to 9,999 years.

The following examples show several forms of INTERVAL qualifiers:

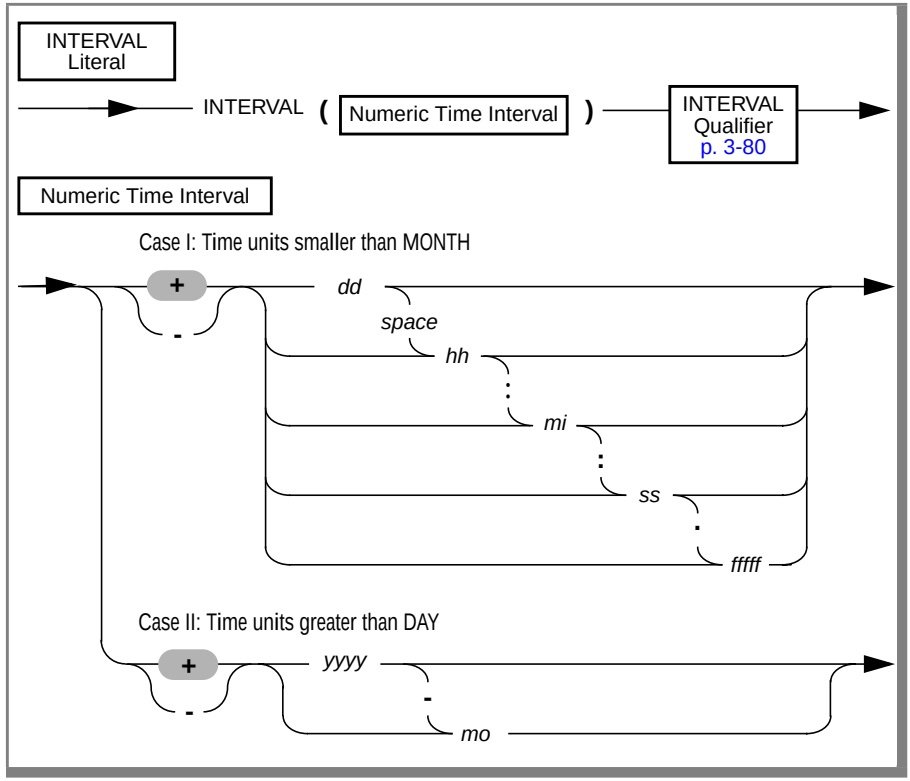
```
YEAR(5) TO MONTH
DAY (5) TO FRACTION(2)
DAY TO DAY
FRACTION TO FRACTION (4)
```

The option to specify a nondefault precision or *y*-precision (as distinct from the scale) is a feature that INTERVAL variables do not share with DATETIME variables. An error results if you attempt to do this when you declare a DATETIME variable, specify DATETIME literal, or call the EXTEND operator.

An error also results if the first keyword represents a smaller time unit than the last, or if you use the plural form of a keyword (such as "MONTHS").

INTERVAL Literal

An *INTERVAL literal* represents a span of time as a numeric representation of its chronological units, followed by an INTERVAL qualifier.



Element	Description
dd	is the number of days.
fffff	is the fraction of a second, up to 5 digits, depending on the precision of the fractional portion in the INTERVAL qualifier.
hh	is the number of hours.
mi	is the number of minutes.
mo	is the number of months, in 2 digits.
space	is a blank space (ASCII 32), entered by pressing SPACEBAR.
ss	is the number of seconds.
yyyy	is the number of years.

For all time units except years and fractions of a second, the maximum number of digits allowed is two, unless this is the first time unit, and the precision is specified differently by the INTERVAL qualifier. (For *years*, the default maximum number of digits is four, unless some other precision is specified by the INTERVAL qualifier.

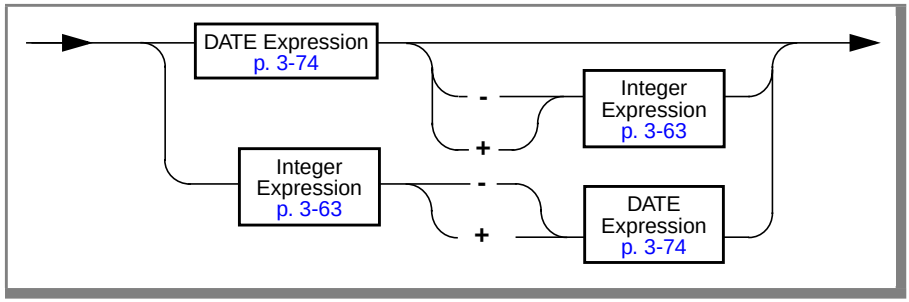
Neither the numeric values nor the qualifier can combine units of time that are smaller than month with month or year time units.

An error results if an INTERVAL literal omits any required field separator, or includes values for units outside the range specified by the field qualifiers. Some examples of INTERVAL literal values follow:

```
INTERVAL (3-6) YEAR TO MONTH
INTERVAL (09:55:30.825) HOUR TO FRACTION
INTERVAL (40-5) DAY TO HOUR
```

Arithmetic Operations on Time Values

Time expressions can be operands of some arithmetic operators. If the result is within the range of valid DATE values, these expressions return a DATE value.

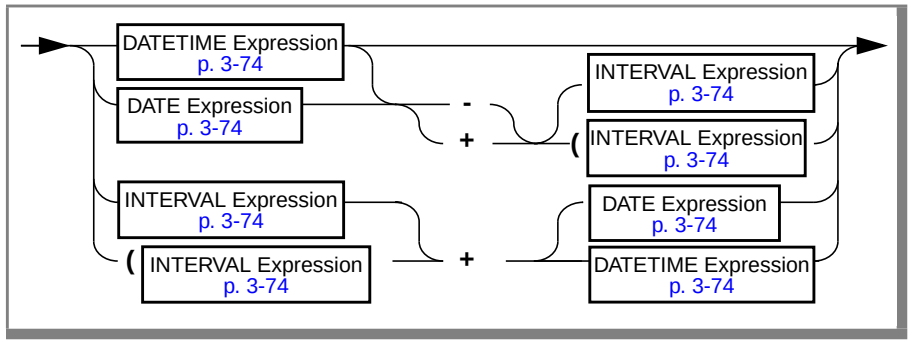


All the other binary arithmetic operators (listed in "Arithmetic Operators" on page 3-66) also accept DATE operands, equivalent to the count of days since December 31, 1899, but the values returned (except from a DATE expression as the left-hand MOD operand) are meaningless in most applications.

DATE and DATETIME values have no true zero point; they lie on *interval* scales. Such scales can logically support addition and subtraction, as well as relational operators (as described in "Relational Operators and Time Values" on page 3-85), but multiplication, division, and exponentiation are undefined.

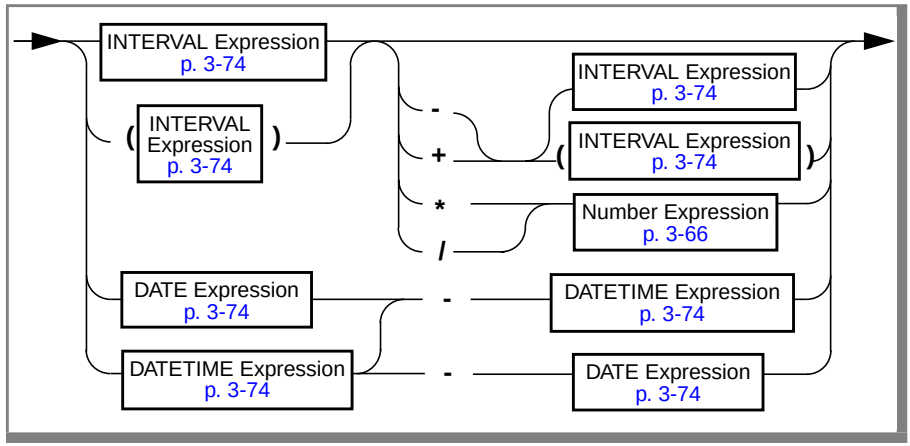
The difference between two DATE values is an INTEGER value, representing the positive or negative number of days between the two calendar dates. You must explicitly apply the UNITS DAY operator to the difference between DATE values if you wish to store the result as an INTERVAL value.

This is the syntax for arithmetic expressions that return a DATETIME value.



Do not write expressions that specify the sum (+) of two DATE or DATETIME values, or a difference (-) whose second operand is a DATE or DATETIME value, and whose first operand is an INTERVAL value.

This is the syntax for arithmetic expressions that return an INTERVAL value.



The difference between two DATETIME values (or a DATETIME and a DATE value, but *not* two DATE values) is an INTERVAL value.

An expression cannot combine an INTERVAL value of precision in the range YEAR TO MONTH with another of precision in the DAY TO FRACTION range. Similarly, you cannot combine an INTERVAL value with a DATETIME or DATE value that has different qualifiers. You must use EXTEND to change the DATE or DATETIME qualifier to match that of the INTERVAL value.

If the first operand of an arithmetic expression includes the UNITS operator (page 5-119), you must enclose that operand in parentheses.

If any component of a time expression is a NULL value, 4GL evaluates the entire expression as NULL.

DATETIME or INTERVAL operands in arithmetic cannot be quoted strings representing numeric date and time (page 3-78) or numeric time interval (page 3-82) values. Use instead DATETIME or INTERVAL literals that also include appropriate DATETIME or INTERVAL qualifiers. For example, the following LET statement that attempts to include an arithmetic expression

```
LET totalsec = "2002-01-01 00:00:00.000" - "1993-01-01 00:00:00.000"
```

in fact assigns a NULL value to the INTERVAL variable **totalsec**, rather than an interval of 9 years, because the two operands have no qualifiers. Better is

```
LET totalsec = DATETIME (2002-01-01 00:00:00.000) YEAR TO FRACTION
             - DATETIME (1993-01-01 00:00:00.000) YEAR TO FRACTION
```

Arithmetic with the UNITS operator or INTERVAL operands can return invalid dates. For example, `(5 UNITS MONTH) + DATETIME (1999-9 30) YEAR TO DAY` produces an error `(-1267: The result of a datetime computation is out of range,)` because the returned value `(2000-2 30)` is not a valid calendar date.

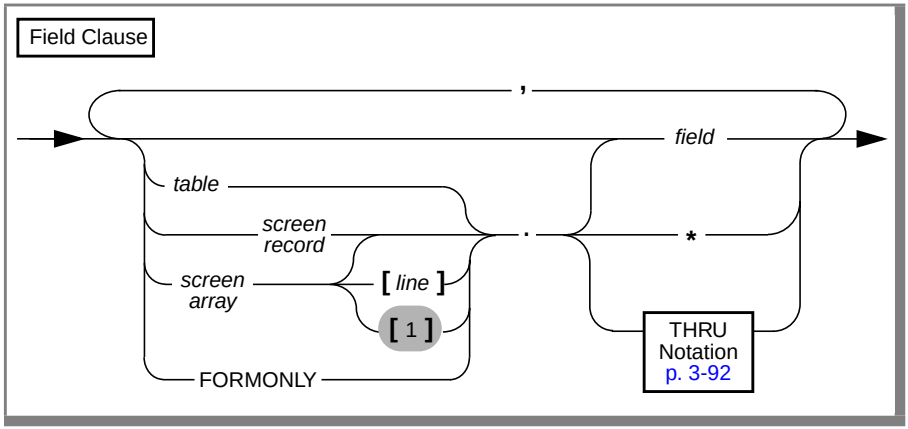
Relational Operators and Time Values

Time expression operands of *relational operators* follow these rules:

- Comparison $x < y$ is TRUE when x is a briefer INTERVAL span than y, or when x is an earlier DATE or DATETIME value than y.
- Comparison $x > y$ is TRUE when x is a longer INTERVAL span than y, or when x is a later DATE or DATETIME value than y.
- You cannot mix INTERVAL with DATE or DATETIME operands; but you can compare DATE and DATETIME values with each other.

Field Clause

The field clause specifies one or more screen fields or screen records.



Element	Description
field	is a field name, as declared in the ATTRIBUTES section of the form specification file.
line	is an integer expression, enclosed within brackets, to specify a record within the screen array. Here $1 \leq line \leq size$, where *size* is the array size that is declared in the INSTRUCTIONS section. If you omit the [*line*] specification, the default is the first record.
screen array	is the 4GL identifier that you declared for a screen array in the INSTRUCTIONS section of the form specification file.
screen record	is the 4GL identifier that you declared for a screen record, or else a table reference (as the name of a default screen record).
table	is the name, alias, or synonym of a database table or view.

Usage

A table reference cannot include table qualifiers. You must declare an alias in the form specification file, as described in "Table Aliases" on page 6-24, for any table reference that requires a qualifying prefix (such as *database*, *server*, or *owner*). Here the FORMONLY keyword acts like a table reference for fields that are not associated with a database column. For more information, see "FORMONLY Fields" on page 6-29.

You can use an asterisk (*) to specify every field in a screen record.

Some contexts, such as the NEXT FIELD clause, support only a single-field subset of this syntax. In these contexts, the THRU or THROUGH keyword, asterisk notation, and comma-separated list of field names are not valid.

You can specify one or more of the following in the field clause:

- A field name without qualifiers (*field*) if this name is unique in the form
- A field name, qualified by a table reference (FORMONLY . *field* or *table.field*)
- An individual member of a screen record (*record* . *field*)
- An individual field within a screen array (*array* [*line*] . *field*)
- A set of consecutive fields in a screen record (by the THRU notation)
- An entire screen record (*record* . *)
- The first screen record in a screen array (*array* . *)
- Any entire record within a screen array (*array* [*line*] . *)

Important: *Some 4GL statements support only a subset of these features. For example, CONSTRUCT cannot specify a screen array line below the first. Similarly, the FIELD_TOUCHED( ) operator in a CONSTRUCT or INPUT statement does not support the* [*line*] *notation to specify a screen record within a screen array.*

The field list of a SCREEN RECORD specification in the INSTRUCTIONS section of a screen form can include the THRU or THROUGH keywords. (For details, see "THRU or THROUGH Keywords and .* Notation" on page 3-92.) Chapter 6 describes how to declare screen records and screen arrays.

The following INPUT statement illustrates how to specify a field name:

```
INPUT p_customer.fname, p_customer.lname FROM fname, lname
```

The following SCROLL statement moves the displayed values in all the fields of the **s_orders** screen array downwards by two lines. Any values are cleared from the first two screen records; any values in the two screen records that are closest to the bottom of the 4GL screen or other 4GL windows are no longer visible:

```
SCROLL s_orders.* DOWN 2
```

The next SCROLL statement moves the displayed values in two of the fields of the **s_orders** screen array towards the top of the 4GL screen for every screen record. Any other fields of the **s_orders** array are not affected:

```
SCROLL s_orders.stock_num, s_orders.unit_descr UP 2
```

The following CLEAR statement clears one record of a screen array. In this example, the integer value of the **idx** variable determines which screen record is cleared:

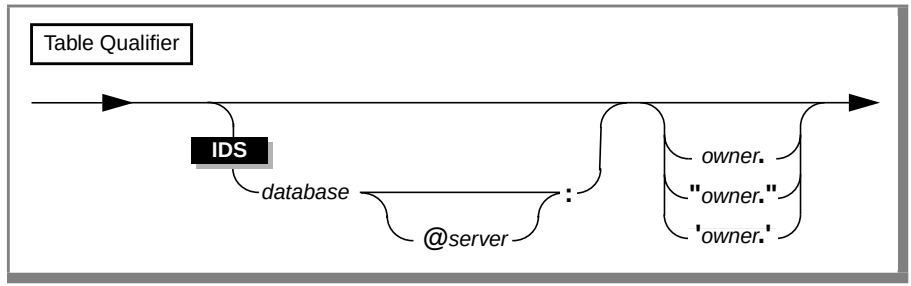
```
CLEAR s_items[idx].*
```

References

CLEAR, CONSTRUCT, DISPLAY, INPUT, INPUT ARRAY, SCROLL, THRU

Table Qualifiers

Statements that reference database tables, views, or synonyms (either alone or as qualifiers of database column names) can include *table qualifiers*.



Element	Description
database	is the name of a database containing the table, view, or synonym.
owner	is the login name of the owner of the table, view, or synonym whose identifier immediately follows the table qualifier.
server	is the name of the host system where *database* resides. Blank spaces are not valid after the @ symbol.

Usage

Table qualifiers can appear in SQL and other 4GL statements and in table alias declarations in the TABLES section of form specifications. You cannot, however, prefix a table alias or a field name with a table qualifier. Except in table alias declarations within the TABLES section, you cannot include table qualifiers anywhere in a form specification file.

Owner Naming

The qualifier can specify the login name of the owner of the table. You must specify *owner* if *table.column* is not a unique identifier within its database.

ANSI

In an ANSI-compliant database, you must qualify each table name with that of the owner of the table (*owner.table*). The only exception is that you can omit the *owner* prefix for any tables that you own.

If the current database is ANSI-compliant, a runtime error results if you attempt to query a remote database that is not ANSI-compliant. ♦

For example, if Les owns table **t1**, you own table **t2**, and Sasha owns table **t3**, you could use the following statement to reference three columns in those tables:

```
VALIDATE var1, var2, var3 LIKE les.t1.c1, t2.c2, sasha.t3.c3
```

You can include the owner name in a database that is not ANSI-compliant. If *owner* is incorrect, however, 4GL generates an error. For more information, see the discussion of the Owner Name segment in the *Informix Guide to SQL: Syntax*.

Database References

The LIKE clause of 4GL statements like DEFINE, INITIALIZE, and VALIDATE can use this *database*: or *database@server*: notation in table qualifiers to specify tables in a database other than the default database (as described in "The Default Database at Compile Time" on page 4-73). Without such qualifiers, 4GL looks for the table in the default database. Even if the table qualifier includes a database reference, however, the LIKE clause will fail unless you also include a DATABASE statement before the first program block in the same module to specify a default database.

The current database is the database specified by the most recently executed DATABASE statement in a MAIN or FUNCTION program block in the same module. 4GL programs can include SELECT statements that query a table in an Informix Dynamic Server database that is not the current database, but they cannot insert, update, or delete rows from any table that is not in the current database.

If the current database is supported by Informix Dynamic Server, a table reference can also include *@server* to specify the name of another host system on which a table resides.

```
LOAD FROM "fyl" INSERT INTO dbas@hostile:woody.table42
```

SE

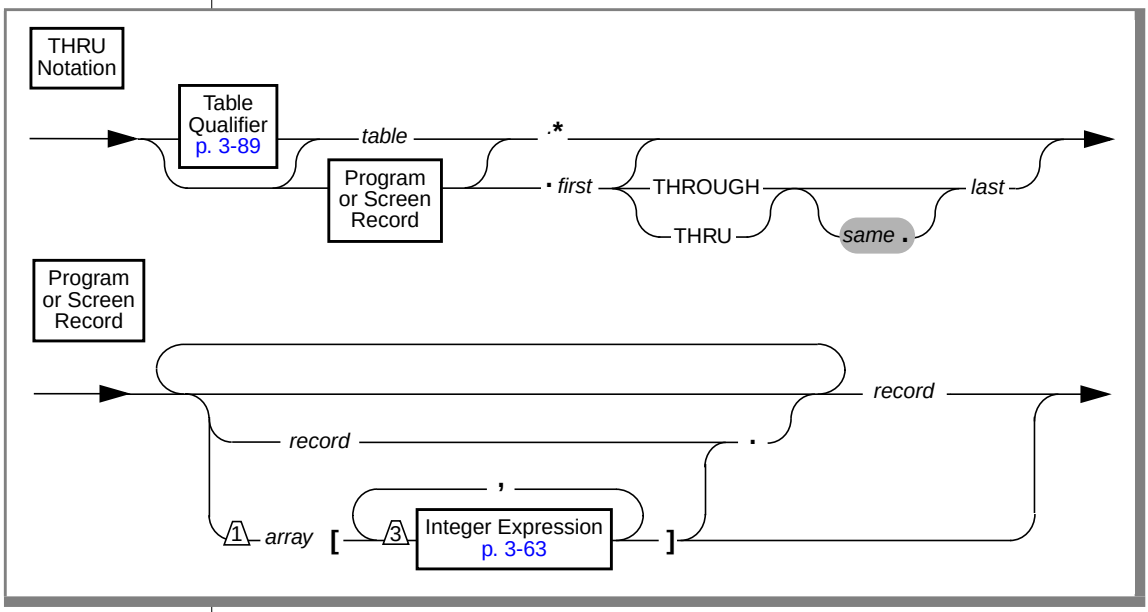
Only the databases stored in your current directory, or in a directory specified in your **DBPATH** environment variable, are recognized. Table qualifiers cannot include references to an INFORMIX-SE database. ♦

References

DATABASE, DEFINE, INITIALIZE, LOAD, VALIDATE, UNLOAD

THRU or THROUGH Keywords and .* Notation

To list consecutive set members in the order of their declaration, you can use the .* notation to specify the entire set, or you can use the keyword THRU (or THROUGH, its synonym) to specify a subset of consecutive members.



Element	Description
array	is the name of an ARRAY variable or screen array, or the keyword FORMONLY. (But if *array* is not a variable, no loop is allowed.)
first	is the name of some member variable or field of *record*.
last	is a variable or field that was declared later than *first*.
record	is the name of a program record or screen record.
same	is the name of the same record that qualified *first*.
table	is the name, alias, or synonym of a database table.

Usage

These notational devices in 4GL statements can simplify lists of structured sets of fields of a screen record or member variables of a program record, or can indicate all of the columns of a database table or view.

The columns of a database table can be referenced by the asterisk notation, but you cannot use THRU or THROUGH to specify a partial list of columns.

The notation *record.member* refers to an individual member variable of a 4GL program record, or a field of a 4GL screen record. The *record*.* notation refers to the entire program record or screen record. Here *record* can be the name, alias, or synonym of a table or view, or the name of a program record or screen record, or the FORMONLY keyword.

The THRU (or equivalently, THROUGH) notation can specify a partial list of the members of a program record or screen record. The notation *record.first* THRU *record.last* refers to a consecutive subset of members of the record, from *first* through *last*, where *first* appears sooner than *last* in the data type declaration of a program record, or else in the ATTRIBUTES section of the form specification file (for screen records).

These notations are a shorthand for writing out a full or partial list of set members with commas separating individual items in the list; this is the form to which 4GL expands these notations. Here are two examples:

```
INITIALIZE pr_rec.member4 THRU pr_rec.member8 TO NULL
DISPLAY pr_rec.* TO sc_rec.*
```

This INITIALIZE statement sets to null the values of 4GL variables **pr_rec.member4**, **pr_rec.member5**, **pr_rec.member6**, **pr_rec.member7**, and **pr_rec.member8**. The DISPLAY statement lists the entire record **pr_rec** in the screen fields that make up the screen record **sc_rec**.

The order of record members within the expanded list is the same order in which they were declared, from *first* to *last*. For a screen record, this is the order of their field descriptions in the ATTRIBUTES section. For example, suppose that the following lines appeared in the form specification file:

```
ATTRIBUTES
...
f002=tab3.aa;
f003=tab3.bb;
f004=tab3.cc;
f005=tab2.aa;
f006=tab2.bb;
```

```
f007=tab1.aa;
f008=tab1.bb;
f009=tab1.cc;
...
INSTRUCTIONS
SCREEN RECORD sc_rec (tab3.cc THRU tab1.bb)
```

This implies the following order of field names within screen record sc_rec:

```
tab3.cc  tab2.aa  tab2.bb  tab1.aa  tab1.bb
```

The order of fields in the screen record depends on the physical order of field descriptions in the ATTRIBUTES section and on the SCREEN RECORD specification. The form compiler ignores the physical arrangement of fields in the screen layout, the order of table names in the TABLES section, the CONSTRAINED and UNCONSTRAINED keywords of the OPTIONS statement, and the lexicographic order of the table names or field names when it processes the declaration of a screen record. For more information about default and nondefault screen records, see "Screen Records" on page 6-74.

The THRU, THROUGH, or .* notation can appear in any list of columns, fields, or member variables, with the following exceptions:

- THRU or THROUGH cannot reference columns of database tables. There is no shorthand for a partial listing of columns of a table.
- You cannot use THRU or THROUGH to indicate a partial list of screen record members while the program displays or enters data in a form.
- You cannot use THRU, THROUGH, or .* in a quoted string to specify variables of a SELECT or INSERT clause in the PREPARE statement.
- You cannot use THRU, THROUGH, or the .* notation to reference a program record that contains an array member. (But these notations can specify all or part of a record that contains records as members, or a record that is an element of an array of records.)
- An exception to the general rule of .* expanding to a list of all column names occurs when .* appears in an UPDATE statement. Here any columns of the SERIAL data type are excluded from the expanded list. For example, the following UPDATE statement:

  ```
  UPDATE table1 SET table1.* = program_rec.*
  ```

  is equivalent to the expanded syntax:

  ```
  UPDATE table1 SET table1.col1 = program_rec.member1,
     table1.col2 = program_rec.member2, ...
  ```

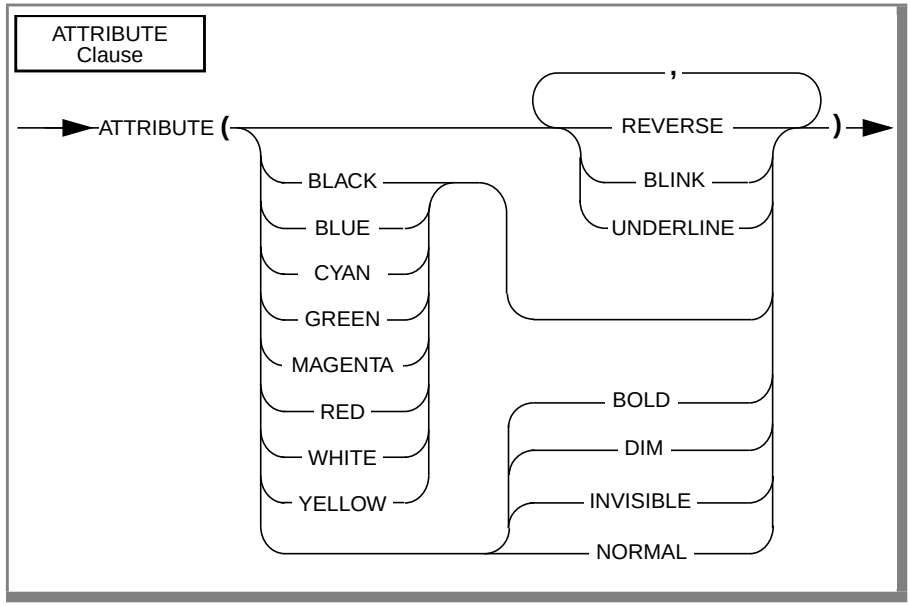
  and so forth to the last column, but with any SERIAL column omitted.

References

CLEAR, CONSTRUCT, DISPLAY, INPUT, INPUT ARRAY, REPORT, SCROLL

ATTRIBUTE Clause

The ATTRIBUTE clause assigns visual attributes in some 4GL statements.



Usage

Keywords listed at the left of this diagram specify *color*; those at the right specify *intensity*. The ATTRIBUTE clause can appear in the following 4GL statements:

CONSTRUCT	DISPLAY FORM	INPUT ARRAY
DISPLAY ARRAY	DISPLAY TO	MESSAGE
DISPLAY AT	ERROR	PROMPT
DISPLAY BY NAME	INPUT	

Besides these statements, both the OPEN WINDOW statement ("OPEN WINDOW" on page 4-280) and the OPTIONS statement ("OPTIONS" on page 4-291) can include ATTRIBUTE clauses that support additional keywords, as described in the sections about those statements in Chapter 4.

The DISPLAY ARRAY, INPUT ARRAY, and PROMPT statements support additional attributes that are described in the next chapter.

You must include at least one keyword in the ATTRIBUTE clause. An attribute clause in any statement except OPEN WINDOW or OPTIONS can specify zero or more of the BLINK, REVERSE, and UNDERLINE attributes, and zero or one of the other attributes. That is, all of the attributes except BLINK, REVERSE, and UNDERLINE are mutually exclusive.

Color and Monochrome Attributes

Support for the REVERSE and INVISIBLE attributes does not depend on the color *versus* monochrome status of the monitor. On any monitor, for example, specifying INVISIBLE in an ATTRIBUTE clause prevents its 4GL statement from displaying output on the screen, or else from echoing the user's keystrokes during data entry. (But the screen shows the *character positions* to which the screen cursor moves while the user types.)

For other attributes, 4GL supports either color or monochrome monitors, but not both. If you have a color monitor, you cannot display the *monochrome* attributes (such as BOLD or DIM). If you have a monochrome monitor, you cannot display the *color* attributes (such as RED or BLUE).

For all ATTRIBUTE clauses and field attributes, the following table shows the effects of the color attributes on a monochrome monitor, as well as the effects of the intensity attributes on a color monitor.

Color Attribute	Monochrome Display	Intensity Attribute	Color Display
WHITE	Normal	NORMAL	White
YELLOW	Bold		
RED	Bold	BOLD	Red
MAGENTA	Bold		
BLUE	Dim	DIM	Blue
GREEN	Dim		
CYAN	Dim		
BLACK	Dim		

If you specify the INVISIBLE attribute, 4GL does not display the data that the user enters in the field. The data value, however, is stored in the input buffer, and is also available by using the **get_fldbuf( )** function.

The following example demonstrates using the ATTRIBUTE clause in an ERROR statement. If the **insert_items( )** function returns FALSE, 4GL rolls back the changes to the database and displays the error message:

```
IF NOT insert_items( ) THEN
   ROLLBACK WORK
   ERROR "Unable to insert items."
      ATTRIBUTE(RED, REVERSE, BLINK)
   RETURN
END IF
```

If the terminal supports color, 4GL displays the error message in red, blinking, reverse video. If the terminal screen is monochrome, 4GL displays the error message in bold, blinking, reverse video.

Within its scope (which may be while a field, a form, or a 4GL window is displayed, or while a statement executes), a color attribute overrides any default colors specified for your terminal. (The next page describes the precedence of 4GL attributes.)

Precedence of Attributes

You can assign different attributes to the same field. During execution of field-related statements, however, 4GL uses these rules of precedence (descending) to resolve any conflicts among attribute specifications:

1. The ATTRIBUTE clause of the current statement.
2. The attributes from the field descriptions in the ATTRIBUTES section of the current form file. (See "Field Attribute Syntax" on page 6-33.)
3. The default attributes specified in the **syscolatt** table of any fields linked to database columns. To modify the **syscolatt** table, use the **upscol** utility. For information on using this utility, see Appendix B, "INFORMIX-4GL Utility Programs."
4. The ATTRIBUTE clause of the most recent OPTIONS statement.
5. The ATTRIBUTE clause of the current form in the most recent DISPLAY FORM statement.
6. The ATTRIBUTE clause of the current 4GL window in the most recent OPEN WINDOW statement.
7. The default reserved line positions and the default foreground color on your terminal.

The field-related statements of INFORMIX-4GL are these:

CONSTRUCT	DISPLAY ARRAY	INPUT
DISPLAY	DISPLAY FORM	INPUT ARRAY

You cannot override the attributes specified for the ERROR, MESSAGE, and PROMPT statements, so precedence rules do not affect these statements.

Keywords of an ATTRIBUTES clause produce their documented effects only when the **termcap** or **terminfo** files and the physical terminals support the attribute. For more information on these files, see Appendix F, "Modifying termcap and terminfo."

On UNIX systems that use **terminfo** files rather than **termcap**, 4GL does not support attributes that specify colors, and the only valid keywords are REVERSE and UNDERLINE.

Important: *Some terminal entries in **termcap** or **terminfo** include the sg#1 or xmc#1 capabilities. If you are using one of these terminals and if the attributes specified for the INPUT ARRAY statement are different than the attributes of the current form or window, 4GL replaces the right and left square brackets that indicate the input fields with blank characters. 4GL uses the blank character as a transition character between the different attributes.*

References

CONSTRUCT, DATABASE, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, ERROR, INPUT, INPUT ARRAY, MESSAGE, OPEN WINDOW, OPTIONS, PROMPT

Chapter

4

INFORMIX-4GL Statements

In This Chapter

This chapter describes the INFORMIX-4GL statements, classifying them by functional category and also providing alphabetized descriptions of the individual statements of 4GL that are not SQL statements.

The 4GL Statement Set

4GL supports the SQL language, but it is sometimes convenient to distinguish between SQL statements and other 4GL statement, as follows:

- SQL statements operate on tables and their columns in a database.
- 4GL statements operate on variables in memory.

Types of SQL Statements

SQL statements of 4GL can be classified among these functional categories:

- Cursor manipulation statements
- Data definition statements
- Data manipulation statements
- Dynamic management statements
- Query optimization statements
- Data access statements
- Data integrity statements
- Stored procedure statements
- Client / server connection statements
- Optical statements

The SQL statements in each of these categories are listed in sections that follow. SQL statements that are not listed here are not available in 4GL.

For syntax and usage information about SQL statements, see the *Informix Guide to SQL: Syntax*. To use the SQL statements identified by the SQL icon in a 4GL program, you must either put the statement in an SQL block, as described in "SQL" on page 4-349, or else prepare the statement, as described in "PREPARE" on page 4-312.

You must also prepare (or delimit with SQL...END SQL) any other SQL statements that specify syntax that was not supported by Informix 4.1 database servers. SQL statements that include only Informix 4.1 syntax can be directly embedded or delimited with SQL...END SQL, but most SQL statements can also be prepared and can appear in SQL blocks.

SQL Cursor Manipulation Statements

CLOSE
DECLARE
FETCH
FLUSH
FREE
OPEN
PUT
SQL SET AUTOFREE

SQL Data Definition Statements

SQL ALTER FRAGMENT
ALTER INDEX
ALTER TABLE
CLOSE DATABASE
CREATE DATABASE
SQL CREATE EXTERNAL TABLE
CREATE INDEX
CREATE PROCEDURE FROM
SQL CREATE ROLE
SQL CREATE SCHEMA
CREATE SYNONYM
CREATE TABLE
SQL CREATE TRIGGER
CREATE VIEW
DATABASE
DROP DATABASE
DROP INDEX
DROP PROCEDURE
SQL DROP ROLE
DROP SYNONYM
DROP TABLE
SQL DROP TRIGGER
DROP VIEW
RENAME COLUMN
SQL RENAME DATABASE
RENAME TABLE

SQL Data Manipulation Statements

INSERT
DELETE
LOAD
SQL OUTPUT
SELECT
UNLOAD
UPDATE

SQL Dynamic Management Statements

EXECUTE
EXECUTE IMMEDIATE
FREE
PREPARE
SQL SET DEFERRED_PREPARE

SQL Query Optimization Statements

	SET EXPLAIN	SQL	SET RESIDENCY
SQL	SET OPTIMIZATION	SQL	SET SCHEDULE LEVEL
SQL	SET PDQPRIORITY		UPDATE STATISTICS

SQL Data Access Statements

	GRANT		SET LOCK MODE
SQL	GRANT FRAGMENT	SQL	SET ROLE
	LOCK TABLE	SQL	SET SESSION
	REVOKE	SQL	SET TRANSACTION
SQL	REVOKE FRAGMENT		UNLOCK TABLE
	SET ISOLATION		

SQL Data Integrity Statements

	BEGIN WORK	SQL	SET PLOAD FILE
	COMMIT WORK	SQL	SET TRANSACTION MODE
	ROLLBACK WORK	SQL	START VIOLATIONS TABLE
SQL	SET DATABASE OBJECT MODE	SQL	STOP VIOLATIONS TABLE
SQL	SET LOG		WHENEVER

SQL Stored Procedure Statements

SQL	EXECUTE PROCEDURE	SQL	SET DEBUG FILE TO

SQL Client/Server Connection Statements

	CONNECT		SET CONNECTION
	DISCONNECT		

SQL Optical Subsystems Statements

SQL	ALTER OPTICAL CLUSTER	SQL	RELEASE
SQL	CREATE OPTICAL CLUSTER	SQL	RESERVE
SQL	DROP OPTICAL CLUSTER	SQL	SET MOUNTING TIMINOUT

Optical statements are only valid on Informix database servers that support optical storage. No INFORMIX-SE database server, for example, supports these features.

Other Types of 4GL Statements

Six other types of 4GL statements are available. (These are sometimes called simply "4GL statements," to distinguish them from SQL statements.)

- Definition and declaration statements
- Storage manipulation statements
- Program flow control statements
- Compiler directives
- Screen interaction statements
- Report execution statements

4GL Definition and Declaration Statements

DEFINE	LABEL
FUNCTION	MAIN
GLOBALS…END GLOBALS	REPORT

4GL Storage Manipulation Statements

CLOSE FORM	LET
FREE	LOCATE
INITIALIZE	VALIDATE

4GL Program Flow Control Statements

CALL
CASE
CONTINUE
DATABASE
END
EXIT
FINISH REPORT
FOR
FOREACH
GOTO
IF
OUTPUT TO REPORT
RETURN
RUN
START REPORT
TERMINATE REPORT
WHILE

4GL Compiler Directives

DEFER
GLOBALS *filename*
SQL…END SQL
WHENEVER

4GL Screen Interaction Statements

CLEAR
CLOSE WINDOW
CONSTRUCT
COPY ARRAY
CURRENT WINDOW
DISPLAY
DISPLAY ARRAY
DISPLAY FORM
ERROR
INPUT
INPUT ARRAY
MENU
MESSAGE
OPEN FORM
OPEN WINDOW
OPTIONS
PROMPT
REMOVE ARRAY
SCROLL
SLEEP

4GL Report Execution Statements

NEED
PAUSE
PRINT
SKIP

Most 4GL statements are not sensitive to whether INFORMIX-SE or Informix Dynamic Server supports the application. INFORMIX-SE cannot store values in BYTE, TEXT, or VARCHAR columns, but any 4GL program can declare variables of these data types.

Statement Descriptions

The following sections describe the 4GL statements that are not SQL statements, and certain SQL statements. Each description includes these elements:

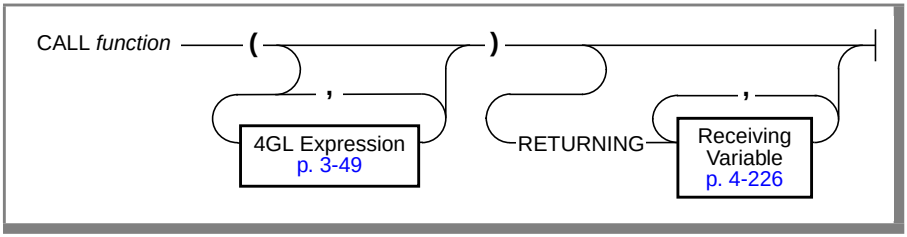
- The name and a terse summary of the effect of the 4GL statement
- A syntax diagram
- Notes on usage, typically arranged by syntax elements

If a description is longer than a few pages, a bulleted list identifies the major topical headings and their page numbers.

A list of related statements concludes most of these statement descriptions.

CALL

The CALL statement invokes a specified function.



Element	Description
function	is the identifier of the function to be invoked.

Usage

CALL can invoke the following types of functions from a 4GL application:

- Programmer-defined 4GL functions
- 4GL built-in functions
- C language functions

It can also invoke ESQL/C functions. Programmer-defined 4GL functions are defined in FUNCTION statements. These functions can appear in the same source file as the MAIN statement, or they can be compiled in separate **.4gl** modules (individually, or with other function and report definitions) and linked later to the MAIN program block.

When 4GL encounters a CALL statement at runtime, it locates the specified FUNCTION program block and executes its statements in sequence. If the function is not a built-in function, a link-time error occurs unless exactly one definition of that function exists in the modules that make up the program.

The program block containing the CALL statement is called the *calling routine*. The RETURNING clause can specify the name of one or more variables that *function* returns to the calling routine. This variable (or list of variables) has the same syntax as a *receiving variable* in the LET statement.

Tip: *Unlike 4GL identifiers, the names of C functions are case sensitive and must typically appear in lowercase letters within the function call. For more information, see Appendix C, "Using C with INFORMIX-4GL."*

In this example, the CALL statement invokes the **show_menu( )** function:

```
MAIN
   ...
   CALL show_menu()
   ...
END MAIN
FUNCTION show_menu()
   ...
END FUNCTION
```

Chapter 5, "Built-In Functions and Operators," provides more information about functions. The following sections describe these topics:

- "Arguments" on page 4-17
- "The RETURNING Clause" on page 4-19
- "Restrictions on Returned Character Strings" on page 4-20
- "Invoking a Function Without CALL" on page 4-21

Arguments

The argument list after the function name specifies values for CALL to pass as *actual arguments* to the function. These actual arguments can be any 4GL expression (as described in "Expressions of 4GL" on page 3-49) if the returned data types are compatible with the corresponding *formal arguments* in the FUNCTION definition. Statements in the FUNCTION definition are executed with the values of actual arguments substituted for the corresponding formal arguments. (Parentheses are always required around the list of argument, even if the list is empty because the function accepts no arguments.)

For example, the following program fragment passes the current values of **p_customer.fname** and **p_customer.lname** to the **print_name( )** function:

```
MAIN
   ...
   CALL print_name(p_customer.fname, p_customer.lname)
   ...
END MAIN
FUNCTION print_name(fname, lname)
   DEFINE fname, lname CHAR(15)
   ...
END FUNCTION
```

When passing arguments to a function, keep these considerations in mind:

- Values in the argument list must correspond in number and position (within the list) to the formal arguments that were specified in the FUNCTION statement.
- Data types of values must be compatible, but need not be identical, to those of the formal arguments in the FUNCTION statement.
- An argument can be an expression that contains variables of simple data types, or simple members of records, or simple elements of arrays. An argument can also be a BYTE or TEXT variable.
- Results can be unpredictable if a variable that has not yet been assigned a value is used as an argument in a CALL statement.

Passing Arguments by Value

How 4GL passes an argument between the calling routine and the function depends on the data type of the argument. Except for variables of data type BYTE or TEXT, arguments are passed to the function *by value*. That is, a copy of the argument is passed. (In this case, changing the value of a formal argument within the function has no effect in the calling routine.)

Passing Arguments by Reference

4GL passes arguments of data type BYTE and TEXT *by reference*. In this case, the function works directly with the actual variable, rather than with a copy. That is, changing a reference to a formal argument in a function changes the corresponding variable in the calling routine. You can use this as a substitute for the RETURNING clause, which does not permit BYTE or TEXT variables.

This example shows how to pass a BYTE or TEXT argument to a 4GL function:

```
MAIN
   DEFINE resume TEXT
   ...
   LOCATE resume IN MEMORY
   CALL get_resume(resume)
END MAIN
FUNCTION get_resume(parm)
   DEFINE parm TEXT
   ...
END FUNCTION
```

In this example, the LOCATE statement allocates memory for the TEXT variable, and places a pointer to this variable in **resume**. Any change to **parm** within the **get_resume( )** function also changes the TEXT variable in MAIN.

The RETURNING Clause

The RETURNING clause assigns values returned by the function to variables in the calling routine. To use this feature, you must do the following:

- Determine how many values *function* returns. In the CALL statement, specify that number of variables in the RETURNING clause.
- If you write the function definition, include expressions in a RETURN statement to specify values returned by the function. (For more information, see "RETURN" on page 4-337.)

When returning values to the CALL statement, keep the following considerations in mind:

- The values in the RETURN statement of the FUNCTION definition must correspond in number and position to the variables specified in the RETURNING clause of the CALL statement. Data types of the RETURNING variables must be compatible with the RETURN values, but they need not be identical. (For more information, see "Summary of Compatible 4GL Data Types" on page 3-46.)
- It is an error to specify more variables in the RETURNING clause than the number of values in the RETURN statement of the FUNCTION definition. (If the RETURNING clause specifies fewer variables, any additional returned values are ignored by the calling routine.)

- You can return simple or RECORD variables from a function. You cannot, however, return RECORD members of ARRAY, BYTE, and TEXT data types.
- The RETURNING clause passes information by value. Because variables of the BYTE and TEXT data types are passed by reference, they cannot be included in the RETURNING clause. (For more information, see "Passing Arguments by Reference" on page 4-18.)

Important: *It is an error to specify a RETURNING clause in the CALL statement if the function does not return anything. It is not an error to omit the RETURNING clause when you invoke a function that returns values if no statement in the calling routine references the returned values.*

In the next example, the **get_cust( )** function returns values of **whole_price** and **ret_price** to the CALL statement. 4GL then assigns the **whole_price** and **ret_price** variables to the **wholesale** and **retail** variables in the **price** record:

```
MAIN
   DEFINE price RECORD
      wholesale, retail MONEY
      END RECORD
   ...
   CALL get_cust() RETURNING price.*
   ...
END MAIN

FUNCTION get_cust()
   DEFINE whole_price, ret_price MONEY
   ...
   RETURN whole_price, ret_price
END FUNCTION
```

Restrictions on Returned Character Strings

A returned value that a CHAR variable receives cannot be longer than 32,767 bytes. (Earlier releases of 4GL allocated 5 kilobytes of memory to store character strings returned by functions, in 10 blocks of 512 bytes, but this restriction has been replaced in current releases by the 32,767 byte limit.)

You can also use TEXT variables to pass longer character values by reference, rather than using the RETURNING clause.

Invoking a Function Without CALL

If a function returns a value, you can invoke it without using CALL by simply including it (and any arguments) within an expression in contexts where the returned value is valid. In the following example, the value returned by a function call appears in an IF statement as a Boolean expression, and a LET statement uses a function call as an operand within an arithmetic expression:

```
IF get_order() THEN
   LET total = total + get_items()
END IF
```

For more information, see "Function Calls as Operands" on page 3-58.

The Comma and Double-Pipe Symbols

As the syntax diagram for "CALL" on page 4-16 indicates, if a function has more than one actual argument, a comma (,) symbol is required between successive arguments in the argument list.

The concatenation operator (| |) can appear within character expressions in the argument list, combining two operands (of any simple data type) as all or part of a single argument to the function. The following example calls the **vitamin( )** function, which takes two character expressions as its arguments:

```
CALL vitamin(var1 || "A", "E") RETURNING varB[2]
```

This use of the concatenation operator is valid in calls to functions, regardless of whether the CALL statement or an expression invokes the function.

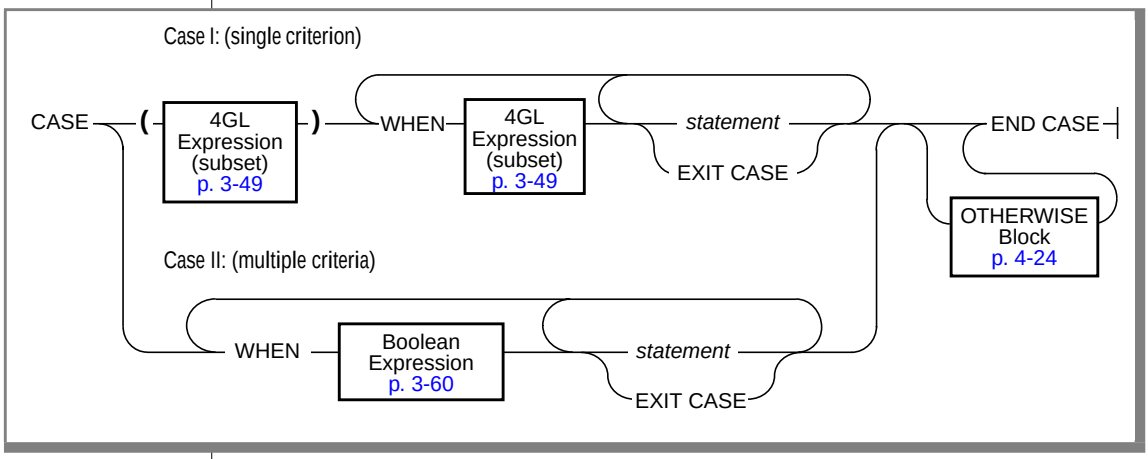
Unlike in the LET statement, the comma and the concatenation operator are not interchangeable in the argument list of a function call. In this context, comma is the required separator and has no concatenation semantics.

References

DEFINE, FUNCTION, RETURN, WHENEVER

CASE

The CASE statement specifies statement blocks to be executed conditionally, depending on the value of an expression. Unlike IF statements, CASE does not restrict the logical flow of control to only two branches.



Element	Description
statement	is an SQL statement or other 4GL statement.

Usage

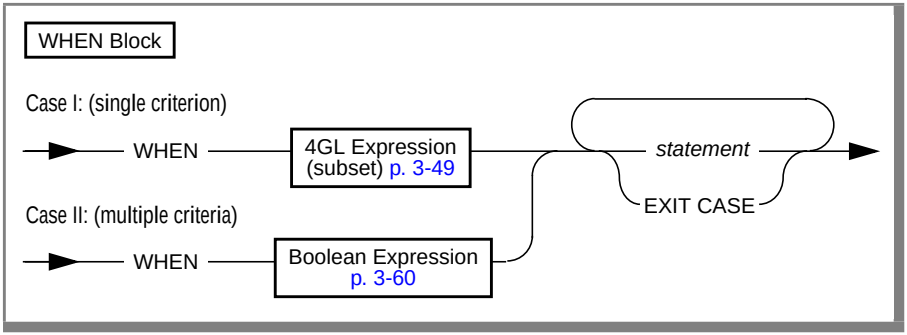
The CASE statement is equivalent to a set of nested IF statements. You can specify two types of CASE statements:

- If an expression follows the CASE keyword, you must specify INT, SMALLINT, DECIMAL, CHAR(1), or VARCHAR(1) expressions in the WHEN block. (The syntax diagram indicates a subset of general 4GL expressions because of these data type restrictions.) 4GL executes the statement block if both expressions return the same non-NULL value.
- If no expression follows the CASE keyword, the WHEN block must specify a Boolean expression; if this returns TRUE, the WHEN block is executed. (See "Boolean Expressions" on page 3-60.) This form of CASE typically executes more quickly than the other.

There is an implicit EXIT CASE statement at the end of each WHEN block of statements. An implicit or explicit EXIT CASE statement transfers program control to the statement that immediately follows the END CASE keywords.

The WHEN Blocks

Each WHEN block specifies an expression and a block of one or more associated statements. The WHEN block has the following syntax.



Element	Description
statement	is an SQL statement or other 4GL statement.

What data type can be returned by *expression* depends on what follows the CASE keyword. If CASE (*expression*) precedes the first WHEN block as a single criterion, an INTEGER, SMALLINT, DECIMAL, CHAR(1), or VARCHAR(1) expression must follow the WHEN keyword, returning a data type that is compatible with the (*expression*) term after CASE.

If a WHEN *expression* matches the value of CASE (*expression*), 4GL executes the statements in that WHEN block and exits from the CASE statement.

In the following example, both **customer_num** and the WHEN *expression* values are of data type SMALLINT:

```
CASE (p_customer.customer_num)
   WHEN 101
   ...
   WHEN 102
   ...
END CASE
```

If no (*expression*) term follows CASE, 4GL treats *expression* as a Boolean (returning TRUE or FALSE) in each of the WHEN blocks. If this Boolean expression returns TRUE (that is, neither zero nor NULL), 4GL executes the corresponding block of statements, as in the following CASE statement:

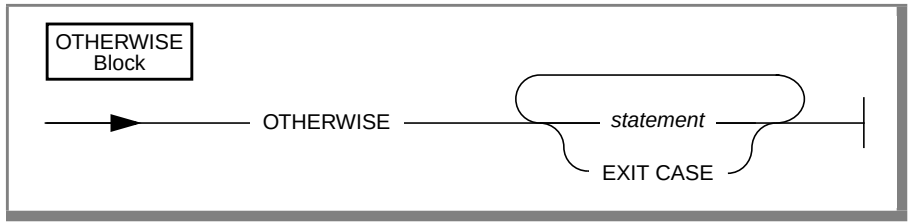
```
CASE
   WHEN total_price < 1000
   ...
   WHEN total_price = 1000
   ...
   WHEN total_price > 1000
   ...
END CASE
```

When more than one WHEN clause can satisfy your criteria, only the first of these causes its statement block to be executed. In these cases, the lexical order of the WHEN clauses in effect prioritizes your logical criteria.

4GL does not execute the statement block if the expression in the WHEN block returns FALSE or NULL, or if CASE *expression* returns NULL. (The IF and WHILE statements and the WHERE clause of a COLOR attribute also treat any NULL value returned from a 4GL Boolean expression as FALSE.)

The OTHERWISE Block

The OTHERWISE keyword specifies statements to be executed when 4GL does not find a matching WHEN block to execute. It has this syntax.



Element	Description
statement	is an SQL statement or other 4GL statement.

4GL executes the OTHERWISE block only if it cannot execute any of the WHEN blocks. If you include the OTHERWISE block, it must follow the last WHEN block.

In the next example, if neither 4GL Boolean expression in the WHEN blocks returns TRUE, 4GL invokes the retry( ) function:

```
WHILE question ...
   CASE
      WHEN answer MATCHES "[Yy]"
         CALL process()
         LET question = FALSE
      WHEN answer MATCHES "[Nn]"
         CALL abort()
      OTHERWISE
         CALL retry()
   END CASE
END WHILE
```

An implied EXIT CASE statement follows the OTHERWISE block. Unless the OTHERWISE block contains a valid GOTO statement, program control passes to the statement that follows the END CASE statement. But the use of GOTO to leave a WHEN block, rather than an implicit or explicit EXIT CASE statement, can cause runtime error -4518, as described in the next section.

The EXIT CASE Statement and the END CASE Keywords

The EXIT CASE statement terminates processing of the WHEN or OTHERWISE block. When it executes an EXIT CASE statement, 4GL skips any statements between EXIT CASE and the END CASE keywords and resumes execution at the first statement following the END CASE keywords.

The END CASE keywords indicate the end of the CASE statement. They must follow either the last WHEN block or else the OTHERWISE block. In the next example, **quantity** has a SMALLINT value. When **quantity** equals **min_qty**, 4GL executes the statement in the **min_qty** block. When **quantity** equals **max_qty**, 4GL executes the statements in the **max_qty** block.

```
CASE (quantity)
   WHEN min_qty
   ...
   WHEN max_qty
   ...
END CASE
```

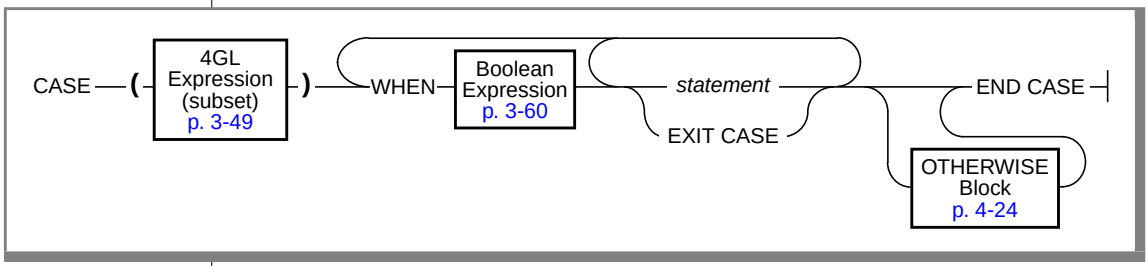
In the following example, **print_option** is a CHAR(1) variable that determines the destination of output from a REPORT program block:

```
CASE (print_option)
   WHEN "f"
      PROMPT " Enter file names for labels >"
         FOR file_name
      IF file_name IS NULL THEN
         LET file_name = "labels.out"
      END IF
      MESSAGE "Printing mailing labels to ",
         file_name CLIPPED," -- Please wait"
      START REPORT labels_report TO file_name
   WHEN "p"
      MESSAGE "Printing mailing labels -- Please wait"
      START REPORT labels_report TO PRINTER
   WHEN "s"
      START REPORT labels_report
      CLEAR SCREEN
END CASE
```

Because WHEN blocks are logically disjunct, exactly one of the START REPORT statements is executed within the CASE statement in this example.

Improper Use of Boolean Expressions with CASE

The 4GL compiler generally cannot detect the following improper syntax.



This is not valid because the expression that follows the WHEN keyword must return a value of data type INT, SMALLINT, DECIMAL, CHAR(1), or VARCHAR(1). Substituting a Boolean expression in this context tends to produce unexpected runtime results. Boolean expressions are valid only in Case II (multiple logical criteria).

For example, CASE statements with WHEN clauses of the following form produce a false result.

CASE Statement	Translates to:
CASE *variable* WHEN “A” OR “B”	if (*variable* == (“A” or “B”))
CASE *variable* WHEN (*variable* = “A” OR *variable* = “B”)	if (*variable* == (*variable* == “A” or *variable* == “B”))

To produce the intended result when the appropriate action depends on the value in the WHEN clause, omit the expression that immediately follows the CASE keyword. Use simplified logic, as in the following code, typically by associating each WHEN expression value with a function call:

```
CASE variable
WHEN "A"
   ...
WHEN "B"
   ...
```

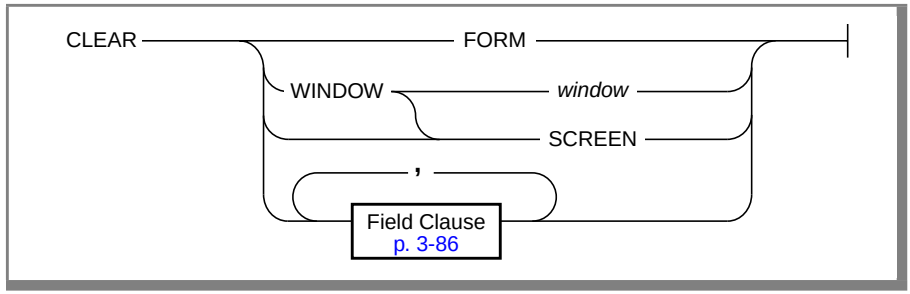
References

FOR, IF, WHILE

CLEAR

The CLEAR statement can clear any of these portions of the screen display:

- The 4GL screen (excluding any open 4GL windows within it)
- Any specified 4GL window
- All of the fields in the current screen form
- A list of one or more specified fields in the current screen form



Element	Description
window	is the name of the 4GL window to be cleared.

Usage

The CLEAR statement clears the specified portion of the display. It does *not* change the value of any 4GL variable.

The CLEAR FORM Option

Use CLEAR FORM to clear all fields of the form in the current 4GL window:

```
CLEAR FORM
```

The CLEAR FORM option has no effect on other parts of the screen display.

The CLEAR WINDOW Option

Use CLEAR WINDOW *window* to clear a specified 4GL window, where *window* is a 4GL identifier that was declared in an OPEN WINDOW statement:

```
CLEAR WINDOW threshold
```

If the window that you specify has a border, the CLEAR WINDOW statement does not erase the border. You can specify any 4GL window, including one that is not the current window, but the CLEAR WINDOW statement does not affect which 4GL window is the current 4GL window in the window stack.

The CLEAR WINDOW SCREEN Option

If you specify CLEAR WINDOW SCREEN, 4GL takes the following actions:

- Clears the 4GL screen, except for the area occupied by any open 4GL windows
- Leaves any information in the open 4GL windows untouched
- Does not change the current 4GL window setting

As in several other 4GL statements, the keyword SCREEN here specifies the 4GL screen.

The CLEAR SCREEN Option

Use the CLEAR SCREEN option to make the 4GL screen the current 4GL window and to clear everything on it, including the Prompt, Message, and Error lines. In the next example, choosing the **Exit** option clears the screen and terminates the MENU statement:

```
MENU "ORDERS"
   COMMAND "Add-order"
         "Enter new order into database and print invoice"
         HELP 301
      CALL add_order( )
      ...
   COMMAND "Exit"
         "Return to MAIN MENU"
         HELP 305
      CLEAR SCREEN
      EXIT MENU
END MENU
```

The CLEAR Field Option

Use the CLEAR *field* option to clear the specified field or fields in a form that the current 4GL window displays. For the syntax of the field clause, see "Field Clause" on page 3-86. The next example clears the fields named **fname**, **lname**, **address1**, **city**, **state**, and **zipcode**:

```
CLEAR fname, lname, address1, city, state, zipcode
```

If you specify *table*.* (where *table* is a name or alias from the TABLE section of the form specification file), CLEAR clears all the fields associated with columns of that table. (See "INSTRUCTIONS Section" on page 6-74 for a description of screen records and screen arrays that the *record*.* notation can reference.)

For example, the following program fragment clears the **orders** screen record and the first four records of the **s_items** screen array:

```
FOREACH order_list INTO p_orders.*
   CLEAR s_orders
   FOR idx = 1 TO 4
      CLEAR s_items[idx].*
   END FOR
   DISPLAY p_orders.* TO orders.*
   ...
END FOREACH
```

If a screen form is in the current 4GL window, the following statement clears all the screen fields that are not associated with database columns:

```
CLEAR FORMONLY.*
```

Any fields that you associated with database columns in the ATTRIBUTES section of the form specification file are *not* affected by this statement.

References

CLOSE FORM, CLOSE WINDOW, CURRENT WINDOW, DISPLAY, DISPLAY ARRAY, INPUT, INPUT ARRAY, OPEN FORM, OPEN WINDOW, OPTIONS

CLOSE FORM

The CLOSE FORM statement releases the memory required for a form.

Element	Description
form	is the name of the 4GL screen form to be cleared from memory.

Usage

When it executes the OPEN FORM statement, 4GL loads the compiled screen form into memory. The CLOSE FORM statement frees memory allocated to a form. For example, this program fragment opens and displays the **o_cust** form and then closes both the form and the 4GL window **cust_w**:

```
OPEN WINDOW cust_w AT 3,5 WITH 19 ROWS, 72 COLUMNS
OPEN FORM o_cust FROM "custform"
DISPLAY FORM o_cust ATTRIBUTE(MAGENTA)
...
CLOSE FORM o_cust
CLOSE WINDOW cust_w
```

If you open the form using the WITH FORM option of the OPEN WINDOW statement, you do not need to use CLOSE FORM before closing the 4GL window. In this case, CLOSE WINDOW closes both the form and the 4GL window, releasing the memory allocated to the form and to the 4GL window.

CLOSE FORM affects memory use only, not the logic of the 4GL program. After you use CLOSE FORM to release the memory allocated to a form, its name is no longer associated with the form. If you subsequently try to redisplay the form, an error message results. If you execute a new OPEN FORM or OPEN WINDOW statement that specifies the same form name that an OPEN FORM or OPEN WINDOW statement referenced previously, 4GL automatically closes the previously opened form before opening the new form.

References

CLOSE WINDOW, DISPLAY FORM, OPEN FORM, OPEN WINDOW

CLOSE WINDOW

The CLOSE WINDOW statement closes a specified 4GL window.

Element	Description
window	is the identifier of the 4GL window to be closed.

Usage

The CLOSE WINDOW statement causes 4GL to take the following actions:

- Clears the specified 4GL window from the 4GL screen and restores any underlying display
- Frees all resources used by the 4GL window and deletes it from the 4GL window stack
- If the OPEN WINDOW statement included the WITH FORM clause, closes both the form and the 4GL window

4GL maintains an ordered list of open 4GL windows, called the *window stack*. When you open a new 4GL window, it is added to the stack and becomes the current window, occupying the top of the stack. Closing the current window makes the next 4GL window on the stack the new current window. If you close any other window, 4GL deletes it from the stack, leaving the current window unchanged. Closing a window has no effect on variables that were set while the window was open. The following program fragment opens and closes a 4GL window called **stock_w**:

```
OPEN WINDOW stock_w AT 7, 3 WITH 6 ROWS, 70 COLUMNS
CLOSE WINDOW stock_w
```

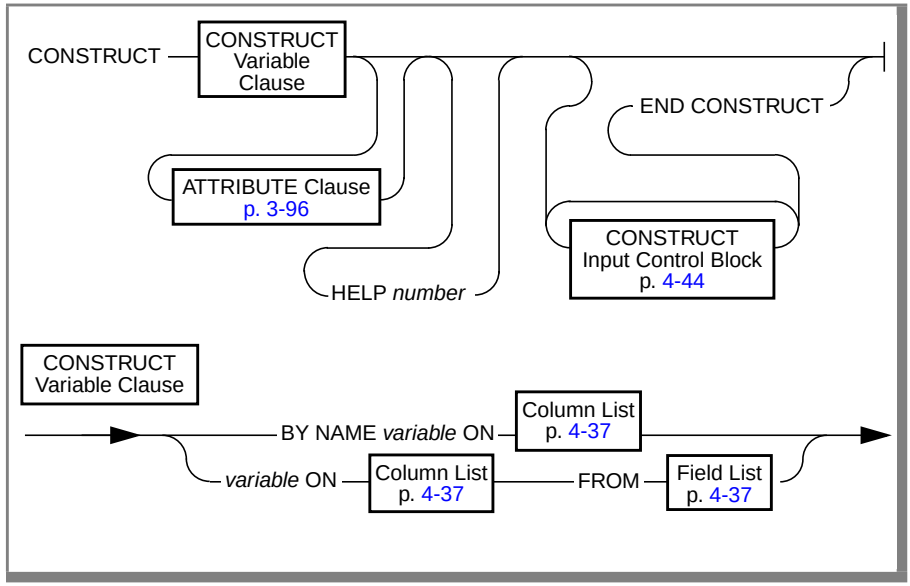
You cannot specify CLOSE WINDOW SCREEN. If *window* is currently being used for input, CLOSE WINDOW generates a runtime error. For example, you cannot close the current 4GL window while a CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or MENU statement is executing.

References

CLEAR, CLOSE FORM, CURRENT WINDOW, OPEN WINDOW, OPTIONS

CONSTRUCT

The CONSTRUCT statement stores in a character variable a 4GL Boolean expression that corresponds to query by example criteria that a user specifies. You can use this variable in the WHERE clause of a SELECT statement.



Element	Description
number	is a literal integer (as described in "Literal Integers" on page 3-65), specifying a help message number.
variable	is the identifier of a CHAR or VARCHAR variable that stores a 4GL Boolean expression summarizing the user-entered search criteria.

Usage

The CONSTRUCT statement is designed to enable users to perform a query by example. *Query by example* enables a user to query a database by specifying values (or ranges of values) for screen fields that correspond to database columns. 4GL converts these values into a Boolean expression that specifies search criteria that can appear in the WHERE clause of a prepared SELECT statement.

The CONSTRUCT statement can also control the environment in which the user enters search criteria, and can restrict the values that the user enters. To use the CONSTRUCT statement, you must do the following:

- Define fields linked to database columns in a form specification file.
- Declare a character variable with the DEFINE statement.
- Open and display the screen form with either of the following:
  - OPEN FORM and DISPLAY FORM statements
  - OPEN WINDOW statement with a WITH FORM clause
- Use CONSTRUCT to store in the character variable a Boolean expression that is based on criteria that the user enters in the fields.

The CONSTRUCT statement activates the current form. This is the form most recently displayed or, if you are using more than one 4GL window, the form currently displayed in the current window. You can specify the current window by using the CURRENT WINDOW statement. When the CONSTRUCT statement completes execution, the form is deactivated.

When it encounters the CONSTRUCT statement, 4GL takes the following actions at runtime:

1. Clears all the screen fields of the CONSTRUCT field list
2. Executes the statements in the BEFORE CONSTRUCT control block, if the CONSTRUCT statement includes that control block
3. Moves the screen cursor to the first screen field in that list
4. Waits for the user to enter some value as search criteria in the field

   For fields where the user enters no value, *any* value in the corresponding database column satisfies the search criteria.

After the user presses the Accept key (typically ESCAPE), CONSTRUCT uses AND operators to combine field values as search criteria in a Boolean expression, and stores this in the character variable. If no criteria were entered, the TRUE expression `' 1=1'` is assigned to the character variable.

By performing the following steps, you can use this variable in a WHERE clause to search the database for matching rows:

1. Concatenate the variable that contains the Boolean expression with other strings to create a string representation of an SQL statement to be executed.

   The Boolean expression generated by the CONSTRUCT statement is typically used to create SELECT statements.

2. Use the PREPARE statement to create an executable SQL statement from the character string that was generated in the previous step.

3. Execute the prepared statement in one of the following ways:

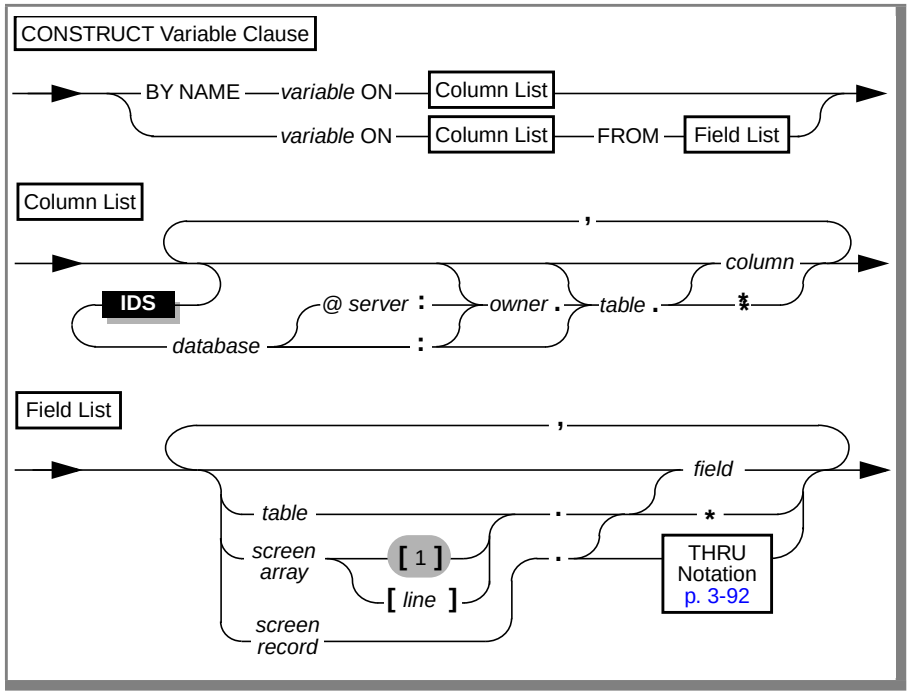
   - Use an SQL cursor with DECLARE and FOREACH statements (or else OPEN and FETCH statements) to execute a prepared SELECT statement that includes no INTO clause.
   - Use the EXECUTE statement to execute an SQL statement other than SELECT, or to execute a SELECT…INTO statement.

When the CONSTRUCT statement completes execution, the form is cleared. Environment variables that format data values, such as **DBDATE**, **DBTIME**, **DBFORMAT**, **DBFLTMASK**, and **DBMONEY**, have no effect on the contents of the Boolean expression.

The following topics are described in this section:

The CONSTRUCT Variable Clause

The CONSTRUCT *variable* clause specifies a character variable to store search criteria that the user can enter in screen fields for database columns.



Element	Description
column	is the unqualified identifier of a database column in *table*.
database	is the identifier of the database in which the table resides.
field	is the identifier of a screen field.
line	is an integer expression, identifying a record in screen array.
owner	is the user name of the owner of the table containing the column.
screen array	is the 4GL identifier of a screen array in the current form.
screen record	is the 4GL identifier of a screen record or else a *table* reference (as the name of a default screen record).
server	is the name of the host system where database resides.
table	is the name, alias, or synonym of a database table or view.
variable	is the identifier of a CHAR or VARCHAR variable.

The CONSTRUCT *variable* clause temporarily binds the specified screen fields to database columns and specifies the database columns for which the user can enter search criteria. You can map the fields implicitly (with the BY NAME keywords) or explicitly (with the FROM keyword and field list). With either method, each field and corresponding column must be of the same or compatible data types. The order of fields in the FROM clause determines the default sequence in which the screen cursor moves from field to field in the form. Within a screen array, you can specify only one screen record.

Here the field list is a restricted subset of the field clause that some 4GL screen interaction statements support (as described in "Field Clause" on page 3-86). In a field list within the CONSTRUCT *variable* clause, a *table* reference cannot include table qualifiers. You must declare an alias in the form specification file, as described in "Table Aliases" on page 6-24, for any *table* reference that requires a qualifying prefix (such as *database*, *server*, or *owner*).

4GL *constructs* a character variable by associating each column name in the ON clause with search criteria that the user enters into the corresponding field (as specified in the FROM clause, or implied by the BY NAME keywords). You can use the information stored in character variable in the WHERE clause of a prepared SELECT statement to retrieve rows from the database. To avoid overflow, declare the length of *variable* as several times the total length of all the fields, because the Boolean expression includes additional operators.

The BY NAME Keywords

You can use the BY NAME clause when the fields on the screen form have the same names as the corresponding columns in the ON clause. The BY NAME clause maps the form fields to columns implicitly. The user can query only the screen fields implied in the BY NAME clause. The following CONSTRUCT statement, for example, assigns search criteria to the variable **query_1**:

```
CONSTRUCT BY NAME query_1 ON company, address1, address2,
   city, state, zipcode
```

The user can enter search criteria in the fields named **company**, **address1**, **address2**, **city**, **state**, and **zipcode**. Because these fields have the same names as the columns specified after the ON keyword, the statement uses the BY NAME clause. If the field names do not match the column names, you must use the FROM clause instead of the BY NAME clause.

This functionally equivalent CONSTRUCT statement uses the FROM clause:

```
CONSTRUCT query_1
   ON company, address1, address2, city, state, zipcode
   FROM company, address1, address2, city, state, zipcode
```

If the column names in a CONSTRUCT BY NAME statement are associated with field names in a screen array, the construct takes place in the first row of the screen array. If you want the CONSTRUCT to take place in a different row of the screen array, you must use the FROM clause, not the BY NAME clause.

You cannot preface column names with a qualifier that includes an owner name, a database server name, or a pathname when you use the BY NAME clause. Use the FROM clause to specify table aliases in the field list when the qualifier of any column name requires an owner name, a database server name, or a pathname.

The ON Keyword and Columns List

The ON clause specifies a list of database columns for which the user will enter search criteria. Columns do not have to be from the same table. The table can be in the specified database, in **DBPATH**, or in the current database as specified by a DATABASE statement in this program block. (For more information, see "The Current Database at Runtime" on page 4-74.)

If the CONSTRUCT statement includes the BY NAME keywords, be sure that the fields on the screen form have the same names as the columns listed after the ON keyword. If the CONSTRUCT statement includes a FROM clause, the expanded list of columns in the ON clause must correspond in order and in number to the expanded list of fields in the FROM clause.

You can use the notation *table*.* (as described in "THRU or THROUGH Keywords and .* Notation" on page 3-92), meaning "every column in *table*," for all or part of the column list. The order of columns within *table* depends on their order in the **syscolumns** system catalog table when you compile your program. If the ALTER TABLE statement has changed the order, the names, the data types, or the number of the columns in *table* since you compiled your program, you might need to modify your program and its screen forms that reference that table.

The following example uses the **customer.*** notation as a macro for listing all columns in the **customer** table and **cust.*** as a macro for all the fields in the **customer** screen record:

```
CONSTRUCT query_1 ON customer.* FROM cust.*
```

The FROM Keyword and Field List

The FROM clause specifies a list of screen fields or screen records in the form. You cannot use the FROM clause if you include the BY NAME clause, but you *must* use the FROM clause if any of the following conditions is true:

- The names of fields on the screen form are different from the names of the corresponding database columns in the ON clause.
- You want to reference fields in a screen array beyond the first record.
- You specify additional qualifiers for *table.column* in the ON clause (for example, for external or non-unique table names, or to reference the owner of a table if the database is ANSI-compliant).
- You want to specify an order for the screen fields other than the default order. (The order of the fields in the screen record determines the default order of the screen fields.)

The user can position the cursor only in fields specified in the FROM clause. The list of fields in the FROM clause must correspond in order and in number to the list of database columns in the ON clause, as in this example:

```
CONSTRUCT query_1 ON stock_num, manu_code, description
   FROM stock_no, m_code, descr
```

If you use the *record.** notation in a field list, be sure that the implied order of fields corresponds to the order of columns in the ON clause. (The order of fields in a screen record depends on its definition in the form specification.)

In the following CONSTRUCT statement, the field list includes the **stock_num** and **manu_code** fields, as well as the screen record **s_stock.*** that corresponds to the remaining columns in the **stock** table:

```
CONSTRUCT query_1 ON stock.*
   FROM stock_num, manu_code, s_stock.*
```

The FROM clause is required when the field list includes an alias representing a table, view, or synonym name that includes any qualifier. For example, in the following CONSTRUCT statement, **cust** is a table alias declared in the form specification file for the **actg.customer** table, where **actg** is an *owner* prefix. This table alias must be prefixed to each field name in the FROM clause, because the column qualifiers in the ON clause include an owner name:

```
CONSTRUCT query_1 ON
   actg.customer.fname, actg.customer.lname,
   actg.customer.company
   FROM cust.fname, cust.lname, cust.company
```

To use screen-array field names in the FROM clause, you must use the notation *screen-record* [*line*].*field-name* to specify the row in which the construct takes place. Here *line* must be greater than zero, and the CONSTRUCT takes place on the *line*th record of the screen array. For example, the following CONSTRUCT statement allows you to enter search criteria in the third line of the screen array **s_items**:

```
CONSTRUCT query_1 ON items.* FROM s_items[3].*
```

If you reference a screen array in the field list with no **[***line***]** specification, the default is the first screen record of the array.

The ATTRIBUTE Clause

For general information about the color and intensity attributes that can be specified for the attribute clause, see "ATTRIBUTE Clause" on page 3-96.

Attributes in CONSTRUCT Statements

The ATTRIBUTE clause in a CONSTRUCT statement applies display attributes to all the fields specified implicitly in the BY NAME clause or explicitly in the FROM clause. If you use the ATTRIBUTE clause, the following attributes do not apply to the fields:

- Default attributes listed in **syscolatt** table

  (See the description of "The upscol Utility" on page B-5.)
- Default attributes in the form specification
- The NOENTRY and AUTONEXT attribute in the form specification

Whether or not the CONSTRUCT statement includes the ATTRIBUTE clause, when the user enters criteria into a field that has the AUTONEXT attribute and keys past the field delimiter, the cursor does not enter the next field.

The CONSTRUCT statement ignores the AUTONEXT attribute so that users can query for large ranges, alternatives, and so forth.

The CONSTRUCT attributes temporarily override any display attributes set by the INPUT ATTRIBUTE clause of an OPTIONS or OPEN WINDOW statement. Attributes in the ATTRIBUTE clause of CONSTRUCT apply to all the fields in the field list, but only during the current activation of the form. When the user deactivates the form, the form reverts to its previous attributes.

The following CONSTRUCT statement includes an ATTRIBUTE clause that specifies CYAN and REVERSE for values entered in screen fields that have the same names as the columns in the **customer** table:

```
CONSTRUCT BY NAME query_1 ON customer.*
   ATTRIBUTE (CYAN, REVERSE)
```

These keywords can produce the effects indicated only when the **termcap** or terminfo files and the physical terminals support the specified attribute. For more information on using these files, see Appendix F, "Modifying termcap and terminfo." On UNIX systems that use terminfo files rather than **termcap**, 4GL does not support attributes that specify colors, and the only valid keywords are REVERSE and UNDERLINE.

Some terminal entries in **termcap** or **terminfo** can include ***sg#1*** or ***xmc#1*** capabilities. If you are using one of these terminals and if the attributes specified for the CONSTRUCT statement are different from the attributes of the current form or window, 4GL replaces the right and left ([ ]) brackets that indicate the input fields with blank characters. 4GL uses the blank character as a transition character between the different attributes.

The HELP Clause

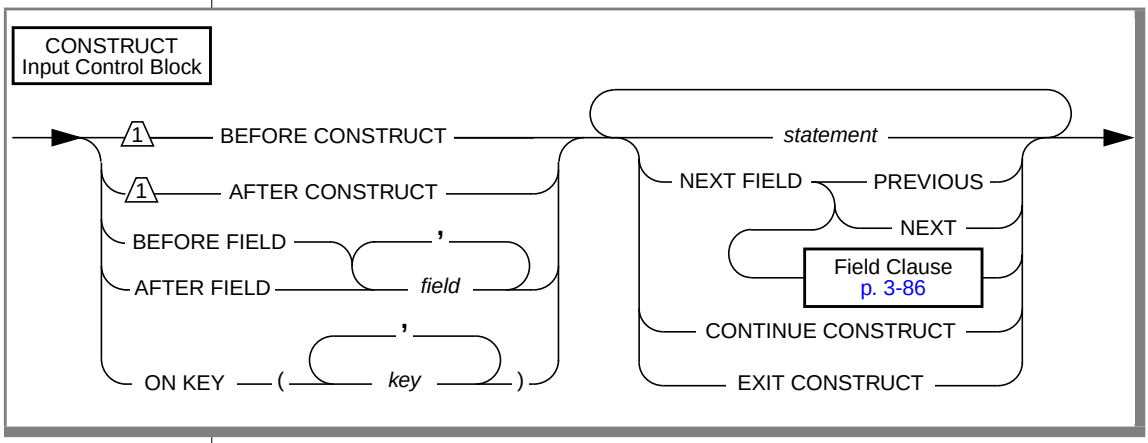
The optional HELP clause specifies the number of a help message associated with this CONSTRUCT statement. This might describe, for example, the role of the user in query by example. The message appears in the Help window when the user presses the Help key from any field in the field list. By default, the Help key is CONTROL-W, but the OPTIONS statement can assign a different physical key as the Help key.

You create help messages in an ASCII file. The *number* identifies the message in the help file. The help file is a compiled message file whose name you specify in the HELP FILE clause of the OPTIONS statement. The **mkmessage** utility can create a compiled version of the help file. For details of how to create a runtime version of the help file, see "The mkmessage Utility" on page B-2. An error occurs if 4GL cannot open the help file, or if *number* is not in the help file, or is greater than 32,767.

To provide field-level help, use an ON KEY clause (as described in "The ON KEY Blocks" on page 4-47) with the INFIELD( ) operator and SHOWHELP( ) function; both are described in Chapter 5. If you provide messages to assist the user through an ON KEY clause, rather than by the HELP clause, the message must be displayed in a 4GL window within the 4GL screen, rather than in the separate Help window.

The CONSTRUCT Input Control Blocks

Each CONSTRUCT input control block includes a statement block of at least one statement, and an activation clause that specifies when to execute the block. The activation clause and statement block correspond respectively to the left-hand and right-hand syntax elements in the following diagram.



Element	Description
field	is the name of a field that was either explicitly or implicitly referenced in the CONSTRUCT *variable* clause (page 4-37).
key	is a keyword listed in "The ON KEY Blocks" on page 4-47.
statement	is an SQL statement or other 4GL statement.

You can use CONSTRUCT input control blocks to specify the following:

- Statements to execute before and after the query by example
- Statements to execute before and after a given field
- Statements to execute if a user presses some key sequence
- The next field to which to move the screen cursor
- When to exit from the CONSTRUCT statement

4GL executes the statements in the block according to the following events:

- The fields into which and from which the user moves the cursor
- The keys that the user presses

Statements can include CONTINUE CONSTRUCT and EXIT CONSTRUCT, the NEXT FIELD clause, and most 4GL and SQL statements. See "Nested and Recursive Statements" on page 2-31 for details of including CONSTRUCT, PROMPT, INPUT, and INPUT ARRAY statements within an input control block.

4GL temporarily deactivates the form while executing statements in an input control block. After executing the statements, 4GL reactivates the form, allowing the user to continue modifying values in fields.

The Precedence of Input Control Blocks

The CONSTRUCT statement can list input control blocks in any order. You should develop some consistent ordering, however, so that your code is more readable than if the blocks were randomly ordered. When you use one or more input control blocks, you must include the END CONSTRUCT statement to terminate the CONSTRUCT statement. If you include several input control blocks, 4GL processes them in the following sequence, regardless of the order in which they appear in the CONSTRUCT statement:

1. BEFORE CONSTRUCT (executed before the user begins data entry)
2. BEFORE FIELD (executed before the user enters values in a specified field)
3. ON KEY (executed after the user presses a specified key)
4. AFTER FIELD (executed after the user enters values in a specified field)
5. AFTER CONSTRUCT (executed after the user has finished data entry)

If you include no input control blocks, the program waits while the user enters values in the fields. When the user accepts the values in the form, the CONSTRUCT statement terminates, the form is cleared, and control passes to the next statement.

The BEFORE CONSTRUCT Block

The BEFORE CONSTRUCT control block specifies a series of actions to perform before the user begins entering criteria into the screen fields. 4GL sets the values of all the screen fields to blank spaces when the CONSTRUCT statement begins execution. You can use the BEFORE CONSTRUCT block to supply different initial default values for the fields.

CONSTRUCT executes the statements in the BEFORE CONSTRUCT block once before it allows the user to perform the query by example. You can use DISPLAY statements in the BEFORE CONSTRUCT block to populate the fields; DISPLAY initializes the field buffers to the displayed values. (To specify the first field where criteria can be entered, you can use the NEXT FIELD clause.) For example, the following DISPLAY statement assigns the values in the **rec** program record to the field buffers associated with the **srec** screen record:

```
BEFORE CONSTRUCT
   DISPLAY rec.* TO srec.*
   NEXT FIELD lname
```

The following CONSTRUCT statement displays the value HRO in the **manu_code** field. If the user does not change the HRO value before pressing the Accept key, the query by example selects rows of the database that have the value HRO in the **manu_code** column:

```
CONSTRUCT query_1 ON stock.* FROM s_stock.*
   BEFORE CONSTRUCT
      LET p_stock.manu_code = "HRO"
      DISPLAY p_stock.manu_code TO stock.manu_code
END CONSTRUCT
```

No more than one BEFORE CONSTRUCT block can appear in a CONSTRUCT statement. The FIELD_TOUCHED( ) operator is not valid in this control block.

The BEFORE FIELD Blocks

This control block specifies a series of actions to execute before the cursor is placed in a specified screen field. 4GL executes the BEFORE FIELD block of statements whenever the screen cursor enters the field, and before the user types search criteria. A field can have no more than one BEFORE FIELD block.

You can use a NEXT FIELD clause within a BEFORE FIELD block to restrict access to a field. You can also use a DISPLAY statement within a BEFORE FIELD block to display a default value in a field.

The following example uses the BEFORE FIELD clause to display a message when the cursor enters the **state** field:

```
BEFORE FIELD state
   MESSAGE "Press F2 or CTRL-B to display a list of states"
```

The following program fragment defines two BEFORE FIELD blocks. The first block uses the NEXT FIELD clause to limit access to the **salary** field to certain users. The second block displays the current date in the **q_date** field:

```
CONSTRUCT BY NAME query_1 ON employee.*
   BEFORE FIELD salary
      IF (username <> "manager") AND (username <> "admin")
         THEN NEXT FIELD NEXT
      END IF
   BEFORE FIELD q_date
      LET query_date = TODAY
      DISPLAY query_date TO q_date
END CONSTRUCT
```

The ON KEY Blocks

The ON KEY control blocks specify actions to take when the user presses certain function or control keys. The statements in the appropriate ON KEY block are executed if the user presses the activation key corresponding to one of your key specifications.

The next example uses the ON KEY clause to call a help message. Here the BEFORE CONSTRUCT clause informs you how to access help:

```
BEFORE CONSTRUCT
   DISPLAY "Press F1 or CTRL-W for help"
ON KEY (f1, control-w)
   CALL customer_help()
```

The following table lists the keywords that you can specify for *key*.

ACCEPT	HELP	NEXT or NEXTPAGE
DELETE	INSERT	PREVIOUS or PREVPAGE
DOWN	INTERRUPT	RETURN
ESC or ESCAPE	LEFT	TAB
F1 through F64	RIGHT	UP
CONTROL-*char* (except A, D, H, I, J, K, L, M, R, or X)		

Like other keywords of 4GL, you can specify these in uppercase or lowercase letters.

Some keys need special consideration if you assign them in an ON KEY block.

Key	Special Considerations
ESC or ESCAPE	You must use the OPTIONS statement to specify another key as the Accept key, because ESCAPE is the default Accept key.
F3	You must use the OPTIONS statement to specify another key as the Next key, because F3 is the default Next key.
F4	You must use the OPTIONS statement to specify another key as the Previous key, because F4 is the default Previous key.
Interrupt	You must execute a DEFER INTERRUPT statement. When the user presses the Interrupt key under these conditions, 4GL executes the ON KEY block and sets **int_flag** to non-zero but does not terminate the CONSTRUCT statement.
Quit	Similarly, 4GL executes the statements in the ON KEY block and sets **quit_flag** to non-zero if the DEFER QUIT statement has been executed when the user presses the Quit key.
CTRL-*char* A, D, H, L, R, X	4GL reserves these control keys for field editing; see "Positioning the Screen Cursor" on page 4-61.
I, J, M	The standard meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is not available to the user. Instead, the key is trapped by 4GL and activates the commands in the ON KEY block. For example, if CONTROL-M appears in an ON KEY block, the user cannot press RETURN to advance the cursor to the next field. If you specify one of these keys in an ON KEY block, be careful to restrict the scope of the statement.

You might not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S respectively send the Interrupt, XON, and XOFF signals on many systems.

If an ON KEY block is activated during data entry, 4GL takes these actions:

1. Suspends the input of the current field
2. Preserves the input buffer that holds the characters the user typed
3. Executes the statements in the corresponding ON KEY clause

4. Restores the input buffer for the current screen field
5. Resumes input in the same field, with the cursor at the end of the buffered list of characters

You can change this default behavior by including statements to perform the following tasks in the ON KEY block:

- Resume input in another field by using the NEXT FIELD keywords
- Change the input buffer value for the current field by assigning a new value to the corresponding variable and displaying the value

Version 4.12 of 4GL introduced a change in the output of CONSTRUCT statements interrupted by the user pressing the Interrupt key (usually CTRL-C or DEL) or the Quit key (usually CTRL-\). This applies only to programs that have executed DEFER INTERRUPT and DEFER QUIT; otherwise, an Interrupt or Quit signal terminates the 4GL application immediately.

In 4.10 and earlier releases, an Interrupt or Quit keystroke in a CONSTRUCT statement produced an output query string that contained the contents of the field buffer at the time the Interrupt keystroke was pressed. Therefore, if the program did not carefully check the value of **int_flag** before proceeding, it could miss the fact that the CONSTRUCT had been interrupted and proceed with a query that was based on defective search criteria.

In Version 4.12 and later, a CONSTRUCT statement interrupted by an Interrupt or Quit keystroke produces a NULL query string. This reduces the risk of an Interrupt condition being undetected. The CONSTRUCT statement can only produce a NULL query string if it was interrupted; thus, you can detect an Interrupt or Quit without checking both **int_flag** and **quit_flag**. (A successful CONSTRUCT statement for which no criteria were entered before the **Accept** key was pressed produces the string `' 1=1'` , not a NULL string.)

The AFTER FIELD Blocks

4GL executes the AFTER FIELD block associated with a field when the screen cursor leaves the field. The user can move the cursor from a field by pressing any of the following keys:

- Any arrow key
- RETURN key
- Accept key
- TAB key

When the NEXT FIELD keywords appear in an AFTER FIELD block, 4GL places the cursor in the specified field and ignores the Accept keystroke. If an AFTER FIELD block exists for each field, and if a NEXT FIELD clause appears in every AFTER FIELD block, the user is unable to leave the form.

The following program fragment checks for the Accept key and terminates execution of CONSTRUCT if the Accept key was pressed:

```
AFTER FIELD status
   IF NOT GET_LASTKEY( ) = ACCEPT_KEY THEN
      LET p_stat = GET_FLDBUF(status)
      IF p_stat MATCHES "married" THEN
         NEXT FIELD spouse_name
      END IF
   END IF
END CONSTRUCT
```

The following AFTER FIELD control block displays a message after the cursor leaves the **state** field, prompting the user to enter search criteria:

```
AFTER FIELD state
   MESSAGE "Press ESC to begin search"
```

As noted in "Completing a Query" on page 4-63, the user can terminate the CONSTRUCT statement by using Accept, Interrupt, or Quit, or by pressing the TAB or RETURN key after the last form field. You can use the AFTER FIELD clause with the NEXT FIELD keywords on the last field to override this default termination. (Alternatively, you can specify INPUT WRAP in an OPTIONS statement to achieve the same effect.)

A field can have no more than one AFTER FIELD control block.

The AFTER CONSTRUCT Block

This control block specifies statements to execute after the user presses Accept and before 4GL constructs the string containing the Boolean expression. You can use the AFTER CONSTRUCT block to validate, save, or alter the values of the screen field buffers. "Using Built-In Functions and Operators" on page 4-55 describes some built-in functions and operators of 4GL that commonly appear in the AFTER CONSTRUCT block.

You can specify CONTINUE CONSTRUCT or NEXT FIELD in this block to return the cursor to the form. If you include these keywords in the AFTER CONSTRUCT block, be sure that they appear within a conditional statement. Otherwise, the user cannot exit from the CONSTRUCT statement and leave the form.

In the following program fragment, a CONTINUE CONSTRUCT statement appears in an IF statement. If the user does not specify any selection criteria, 4GL returns the screen cursor to the form.

```
AFTER CONSTRUCT
   IF NOT FIELD_TOUCHED(orders.*) THEN
      MESSAGE "You must indicate at least one ",
         "selection criteria."
      CONTINUE CONSTRUCT
   END IF
```

For more information, see "Searching for All Rows" on page 4-60.

4GL executes the statements in the AFTER CONSTRUCT block when the user presses any of the following keys:

- The Accept key
- The Interrupt key (if DEFER INTERRUPT has executed)
- The Quit key (if the DEFER QUIT statement has executed)

The AFTER CONSTRUCT block is *not* executed in the following situations:

- The user presses the Interrupt or Quit key and the DEFER INTERRUPT or DEFER QUIT statement, respectively, has not been executed. In either case, the program terminates immediately, and no query is performed.
- The EXIT CONSTRUCT statement terminates the CONSTRUCT statement.

The CONSTRUCT statement can include only one AFTER CONSTRUCT block.

The NEXT FIELD Clause

While the CONSTRUCT statement is executing, 4GL moves the screen cursor from field to field in the order specified in the FROM clause, or in the order implied by the ON clause of the CONSTRUCT BY NAME statement. You can use the NEXT FIELD keywords, however, to override the default sequence of cursor movement.

You can specify any of the following fields in the NEXT FIELD clause:

- The next field, as defined by the explicit (FROM clause) or implicit (BY NAME) order of fields in the field list of the CONSTRUCT *variable* clause. In this case, specify the NEXT keyword.
- The previous field, as defined by the same order of fields. In this case, specify the PREVIOUS keyword.
- Any other field in the current form. In this case, specify the name of the field (from the ATTRIBUTES section of the form specification file).

The NEXT FIELD keywords can appear in a BEFORE CONSTRUCT block (for example, to position the cursor at a different starting field) and in a BEFORE FIELD block (for example, to restrict access to a field), but they are more commonly used in AFTER FIELD, ON KEY, or AFTER CONSTRUCT blocks.

Use NEXT FIELD only if you want the cursor to deviate from the default field order. 4GL immediately positions the cursor in the form when it encounters the NEXT FIELD clause, without executing any statements that immediately follow the NEXT FIELD clause in the same statement block.

In the following program fragment, function **qty_help( )** cannot be invoked because its CALL statement is positioned after the NEXT FIELD clause:

```
ON KEY (CONTROL_B, F4)
   IF INFIELD(stock_num) OR INFIELD(manufact) THEN
      CALL stock_help( )
      NEXT FIELD quantity
      CALL qty_help( ) -- function is never called
   END IF
```

The following program fragment includes NEXT FIELD clauses in ON KEY and AFTER FIELD blocks. The user triggers the ON KEY block by pressing CONTROL-B or F4. If the cursor is in the **stock_num** field or **manufact** field, 4GL calls the **stock_help( )** function. When 4GL returns from the **stock_help( )** function, the NEXT FIELD clause moves the cursor to the **quantity** field.

The user executes the AFTER FIELD block by moving the cursor out of the **zipcode** field. The FIELD_TOUCHED( ) operator checks whether the user entered a value into the field. If this returns TRUE, GET_FLDBUF( ) retrieves the value entered into the field during a query, and assigns it to the **p_zipcode** variable. If the first character in the **p_zipcode** variable is not a 5, 4GL displays an error, clears the field, and returns the cursor to the field.

```
ON KEY (CONTROL_B, F4)
   IF INFIELD stock_num) OR INFIELD(manufact) THEN
      CALL stock_help( )
      NEXT FIELD quantity
   END IF

AFTER FIELD zipcode
   IF FIELD_TOUCHED(zipcode) THEN
      LET p_zipcode = GET_FLDBUF(zipcode)
      IF p_zipcode[1,1] <> "5" THEN
         ERROR "You can only search area 5."
         CLEAR zipcode
         NEXT FIELD zipcode
      END IF
   END IF
```

Do not use NEXT FIELD clauses to move the cursor across every field in a form. If you want the cursor to move in a specific order, list the fields in the CONSTRUCT statement in the desired order. In most situations, NEXT FIELD appears in a conditional statement. The NEXT FIELD clause *must* appear in a conditional statement when it appears in an AFTER CONSTRUCT block; otherwise, the user cannot exit from the query.

The CONTINUE CONSTRUCT Statement

You can use the CONTINUE CONSTRUCT statement to exit from a BEFORE CONSTRUCT, AFTER CONSTRUCT, BEFORE FIELD, AFTER FIELD, or ON KEY control block and return the cursor to the form, with the same CONSTRUCT statement still in effect. The CONTINUE CONSTRUCT statement skips all subsequent statements in the CONSTRUCT statement, and returns the cursor to the screen form at the last field occupied.

This statement is useful where program control is nested within multiple conditional statements and you want to return control to the user. It is also useful in an AFTER CONSTRUCT block, where you can examine field buffers and, depending on their contents, return the cursor to the form.

In this example, CONTINUE CONSTRUCT appears in an AFTER CONSTRUCT clause. If the user enters N or n at the prompt, the cursor returns to the form:

```
CONSTRUCT BY NAME query1 ON customer.*
...
   AFTER CONSTRUCT
      IF NOT FIELD_TOUCHED(customer.*) THEN
         PROMPT "Do you really want to see ",
            "all customer rows? (y/n)" FOR CHAR answer
         IF answer MATCHES "[Nn]" THEN
            MESSAGE "Enter search criteria; ",
               "press ESC to begin search."
            CONTINUE CONSTRUCT
         END IF
      END IF
END CONSTRUCT
```

If no criteria are entered, the user is prompted to confirm that all customer records are requested. If the user types N or n, CONTINUE CONSTRUCT positions the cursor in the form, giving the user another chance to enter selection criteria in the last field occupied. If the user types any other key, the IF statement terminates, and control passes to the END CONSTRUCT statement. Compare this method of detecting and handling the absence of search criteria to the examples in "The AFTER CONSTRUCT Block" on page 4-51 and "Searching for All Rows" on page 4-60.

When a test in an AFTER CONSTRUCT clause identifies a field that requires action by the user, specify NEXT FIELD, rather than CONTINUE CONSTRUCT, to position the cursor in the field.

The EXIT CONSTRUCT Statement

The EXIT CONSTRUCT statement causes 4GL to take the following actions:

- Skip all statements between EXIT CONSTRUCT and END CONSTRUCT.
- Terminate the process of constructing the query by example.

- Create the Boolean expression and store it in the character variable.
- Resume execution at the statement following the END CONSTRUCT keywords.

If it encounters the EXIT CONSTRUCT statement, 4GL does not execute the statements in the AFTER CONSTRUCT control block, if that block is present.

The END CONSTRUCT Keywords

The END CONSTRUCT keywords indicate the end of the CONSTRUCT statement. These keywords should follow the last BEFORE CONSTRUCT, AFTER CONSTRUCT, BEFORE FIELD, AFTER FIELD, or ON KEY control block. If you do not include any of these control blocks, however, you do not need to include the END CONSTRUCT keywords.

Using Built-In Functions and Operators

The CONSTRUCT statement supports the built-in functions and operators of 4GL. (For more information about the built-in 4GL functions and operators, see Chapter 5.) These features access field buffers and keystroke buffers.

Feature	Description
FIELD_TOUCHED( )	Returns TRUE when the user has *touched* (made a change to) a screen field whose name is passed as an operand. Moving the screen cursor through a field (with the RETURN, TAB, or arrow keys) does not mark a field as touched. This feature also ignores the effect of statements that appear in the BEFORE INPUT control block. For example, you can assign values to fields in the BEFORE INPUT control block without having the fields marked as touched.
GET_FLDBUF( )	Returns the character values of the contents of one or more fields in the currently active form.

(1 of 2)

Feature	Description
FGL_GETKEY( )	Waits for a key to be pressed, and then returns an INTEGER value corresponding to the raw value of the key that was pressed.
FGL_LASTKEY( )	Returns an INTEGER value corresponding to the most recent keystroke executed by the user in the screen form.
INFIELD( )	Returns TRUE if the name of the field that is specified as its operand is the name of the current field.

(2 of 2)

Each field in a form has only one field buffer, and a buffer cannot be used by two statements simultaneously. If a CONSTRUCT statement calls a function that includes an INPUT, INPUT ARRAY, or CONSTRUCT statement, and both statements use the same form, they might overwrite one or more of the field buffers, unless you first close the form or window. (See, however, "Nested and Recursive Statements" on page 2-31.)

If you plan to display the same form more than one time and will access the form fields, open a new window and open and display a second copy of the form. 4GL allocates a separate set of buffers to each form, and you can then be certain that your program is processing the correct field values.

Search Criteria for Query by Example

The CONSTRUCT statement allows users of your application to specify search criteria for retrieving rows from the database. The user does this by entering values (or ranges of values) into the fields of the screen form. This process is called a *query by example*. The user can use symbols to search for data values less than, equal to, greater than, or within a range.

CONSTRUCT supports the following relational operator symbols.

Symbol	Meaning	Data Type Domain	Pattern
= or ==	Equal to	All simple SQL types	==*x*, =*x*, =
>	Greater than	All simple SQL types	>*x*
<	Less than	All simple SQL types	<*x*

(1 of 2)

Symbol	Meaning	Data Type Domain	Pattern
>=	Not less than	All simple SQL types	>=*x*
<=	Not greater than	All simple SQL types	<=*x*
<> or !=	Not equal to	All simple SQL types	!=*x*, <>*x*
: or . .	Range	All simple SQL types	*x*:*y*, *x*..*y*,
*	Wildcard for any string	CHAR, VARCHAR	**x*, *x**, **x**
?	Single-character wildcard	CHAR, VARCHAR	?*x*, *x*?, ?*x*?, *x*??
|	Logical OR	All simple SQL types	*a* | *b*...
[ ]	List of values (see next page)	CHAR, VARCHAR	[*xy*]*, [*xy*]?

(2 of 2)

The ". . " form of the range operator is required for ranges of DATETIME or INTERVAL literal values that include ":" symbols as time-unit separators. Users cannot perform a query by example on BYTE, TEXT, or FORMONLY fields. If the search criteria exceed the length of a field, 4GL opens a work space on the Comment line. This action erases any comments present.

The following list explains the symbols in the preceding table:

x The x means any value appropriate to the data type of the field. The value must immediately follow any of the first six symbols in the preceding table. Do not leave a space between a symbol and a value.

=*x* The equal sign (=) is the default symbol for non-character fields, and for character fields in which the search value contains no wildcards. If the user enters a character value that does not contain a wildcard character, CONSTRUCT produces the following Boolean expression:

```
char-column = "value"
```

= The equal sign (=) with no *value* searches for a NULL value. The user must explicitly enter the equal sign to find any character value that is also used as a search criteria symbol.

>, <, >=, <=, <>
These symbols imply an ordering of the data. For character fields, "greater than" means *later* in the ASCII sequence (where a > A > 1), as listed in Appendix G, "Reserved Words." For DATE or DATETIME data, "greater than" means *after*. For INTERVAL data, it means a *longer* span of time.

A query by example cannot combine these relational operators with the range, wildcard, or logical operators that are described later. Any characters that follow a relational operator are interpreted as literals.

In Version 4.11 and earlier releases, 4GL would put double quotation marks(") characters around the field values entered during the CONSTRUCT statement regardless of the data types. The resulting WHERE clause for the SQL statement might not be compatible with non-Informix database servers because of the double quotation marks.

In Version 4.12 and later releases, the CONSTRUCT statement puts single quotation marks (') around values of the character and time data types: CHAR, DATE, DATETIME, INTERVAL, and VARCHAR. Values of the number data types are not enclosed in quotation marks: FLOAT, SMALLFLOAT, DECIMAL, MONEY, INTEGER, SMALLINT, and SERIAL.

Double quotation marks are not used as delimiters in the constructed variable to avoid incompatibility with non-Informix databases.

These changes allow 4GL programs to work with Informix Enterprise Gateway for interoperability with DB2/400, DB2/MVS (also called, simply, DB2) and DB2/VM (also called SQL/DS). These changes are known colloquially as the DRDA changes. The latest versions of the Informix Enterprise Gateway product automatically convert the keyword MATCHES to LIKE and generate an error if the MATCHES string contains a character range enclosed in square brackets (for example, [a-z]).

Besides the relational operators, the user can specify a range, or use syntax like that of the MATCHES operator to search for patterns in character values:

- **Colon.** The colon in *x*: *y* searches for all values between the *x* and *y* value, inclusive. The *y* value must be larger than the *x* value. The search criterion 1: 10 would find all rows with a value in that column from 1 through 10. For character data, the range specifies values in the ASCII collating sequence between *x* and *y*. (For DATETIME and INTERVAL fields, use instead the .. symbol to specify ranges.)
- **Two periods.** Sometimes you must substitute two periods (..) as a synonym for the colon (:) in DATETIME and INTERVAL ranges to avoid ambiguity with time-unit separators in *hh*:*mm*:*ss* values.
- **Asterisk.** The asterisk (*) is a character string wildcard, representing zero or more characters. Use the asterisk character as follows:
  - The search value `*ts*` in a field specifies all strings containing the letters *ts*, such as the strings "`Watson`" and "`Albertson`".
  - The search value `S*` specifies all strings beginning with the letter *s*, including the strings "`S`", "`Sadler`", and "`Sipes`".
  - The search value `*er` specifies all strings that end in the letters *er*, such as the strings "`Sadler`" and "`Miller`".
- **Question mark.** The question mark (?) is the single-character wildcard. The user can use the question mark to find values matching a pattern in which the number of characters is fixed, as in the following examples:
  - Enter `Eriks?n` to find names like "`Erikson`" and "`Eriksen`".
  - Enter `New??n` to find names like "`Newton`", "`Newman`", and "`Newson`", but not "`Newilsson`".
- **Pipe.** The pipe symbol between values *a* and *b* represents the logical OR operator. The following entry specifies any of three numbers:

  ```
  102|105|118
  ```
- **Brackets.** The brackets ([ ]) delimit a set of values. When used in conjunction with the * and ? wildcard characters, the brackets enclose a list of characters, including ranges, to be matched.

- **Caret.** A caret (^) as the first character within the brackets specifies the logical complement of the set, and matches any character that is not listed. For example, the search value `[^AB]*` specifies all strings beginning with characters other than *A* or *B*.
- **Hyphen.** A hyphen between characters within brackets specifies a range. The search value `[^d-f*]` specifies all strings beginning with characters other than lowercase *d*, *e*, or *f*. If you omit the * or ? wildcard, 4GL treats the brackets as literal characters, not as logical operators.

Searching for All Rows

If *none* of the fields contains search values when the user completes an entry for the CONSTRUCT statement, 4GL uses `' 1=1'` as the Boolean expression. (Notice that this string begins with a blank character.) In a WHERE clause, this search criterion causes 4GL to select *all* the rows of the specified tables.

Place a conditional statement after the CONSTRUCT statement to check for this expression, or to examine the field buffers in the AFTER CONSTRUCT control block, if you want to prevent users from selecting all rows. The next fragment, for example, tests for the `' 1=1'` expression. If this expression is found, the LET statement limits the resulting query list by creating a Boolean expression that searches only for customers with numbers less than or equal to 110:

```
CONSTRUCT BY NAME query_1 ON customer.*
IF query_1 = ' 1=1' THEN
   LET query_1 = " customer_num <= 110"
   MESSAGE "You entered nothing. Here are customers ",
         "with codes less than 111."
   SLEEP 3
END IF
```

The FIELD_TOUCHED( ) operator describes an equivalent test.

Positioning the Screen Cursor

When the user presses RETURN or TAB, the screen cursor moves from one field to the next in the order specified in the FROM clause (as described in "The FROM Keyword and Field List" on page 4-40), or in the order implied by the column list in the BY NAME clause (as described in "The BY NAME Keywords" on page 4-38). The user can also press the following *arrow keys* at runtime to alter this behavior and to position the cursor explicitly.

Key	Effect on Cursor
↓	By default, DOWN ARROW moves to the *next* field. If you specify the FIELD ORDER UNCONSTRAINED option of the OPTIONS statement, this key moves the cursor to the field *below* the current field.
↑	By default, UP ARROW moves to the *previous* field. If you specify the FIELD ORDER UNCONSTRAINED option of the OPTIONS statement, this key moves the cursor to the field *above* the current field.
←	LEFT ARROW moves one space to the *left* within a field. It does not erase the contents of the field. If this is the beginning of the field, the cursor moves to the beginning of the *previous* field.
→	RIGHT ARROW moves one space to the *right* within a field. It does not erase the contents of the field. If this is the end of the field, 4GL creates a workspace at the bottom of the screen and places the cursor there, so the user can continue entering values.

These arrow keys all operate non-destructively. That is, they move the screen cursor without erasing any underlying character.

When the cursor moves to a new field, the CONSTRUCT statement clears the Comment line and the Error line. The Comment line displays the text defined with the COMMENTS attribute in the form specification file. The Error line displays system error messages, output from the built-in ERR_PRINT( ) and ERR_QUIT( ) functions, and ERROR statement messages.

If the user enters search criteria that exceed the length of the screen field, 4GL automatically moves the cursor down to the Comment line and allows the user to continue entry. When the user presses RETURN or TAB, 4GL clears the Comment line. The field buffer contains all the criteria that the user entered, even though only a portion is visible in the screen display.

Using WORDWRAP in CONSTRUCT

In Version 4.11 and earlier releases of 4GL, when CONSTRUCT operated on a multi-segment field with the WORDWRAP attribute set, the initial input was displayed in the first segment of the field. When the input no longer fit in the first segment, it overflowed into the second and subsequent segments of the field. This also cleared the overflow line at the bottom of the screen, however, even though no data appeared in that line.

In Version 4.12 and later releases, multi-segment fields behave in the same way as single-segment fields. When the first segment is full, the overflow line displays the extra data. The CONSTRUCT statement uses only the first segment and the overflow line of a WORDWRAP field; it does not use the extra segments. This should provide sufficient space for query input.

A form field with the WORDWRAP attribute can span several lines. If the segments of a multi-segment field are not aligned in a single column, moving backwards from a later segment can cause the cursor to skip some segments entirely, or leave the cursor in arbitrary locations inside the segment. (For explanations of field segments and WORDWRAP fields, see Chapter 6, "Screen Forms.")

Avoid overly complex placement of multiple-segment fields, such as:

```
[f001    ][f001    ]
     [f001    ]
```

A rectangular field configuration, however, produces predictable results:

```
[f001    ]
[f001    ]
[f001    ]
```

In a multiple-segment field (that is, one with the WORDWRAP attribute), 4GL ignores any values that the user enters in any segment beyond the *first* segment of the field. Similarly, in a screen array, the user can enter criteria only in the first screen record of the array during a CONSTRUCT statement.

Editing During a CONSTRUCT Statement

The user can employ these keys for editing during a CONSTRUCT statement.

Key	Effect
CONTROL-A	Toggles between insert and type-over mode
CONTROL-D	Deletes characters from the cursor position to the end of the field
CONTROL-H	Moves the cursor non-destructively one space to the left within a field (This is equivalent to pressing left arrow.)
CONTROL-L	Moves the cursor non-destructively one space to the right within a field (This is equivalent to pressing right arrow.)
CONTROL-R	Redisplays the screen
CONTROL-X	Deletes the character beneath the cursor

Completing a Query

The following actions terminate the CONSTRUCT statement:

- The user presses one of the following keys:
  - The Accept key
  - The RETURN or TAB key from the last field (when INPUT WRAP is not set in the OPTIONS statement)
  - The Interrupt or Quit key
- 4GL executes the EXIT CONSTRUCT statement.

The user must press the Accept key to complete the query under these conditions:

- INPUT WRAP is specified in the OPTIONS statement.
- An AFTER FIELD block for the last field includes a NEXT FIELD clause.

By default, the Accept, Cancel, Interrupt, and Quit keys terminate both the query and the CONSTRUCT statement. (But pressing the Interrupt or Quit key can also immediately terminate the program, unless the program has also executed the DEFER INTERRUPT and DEFER QUIT statements.)

If 4GL previously executed a DEFER INTERRUPT statement in the program, pressing the Interrupt key while CONSTRUCT is awaiting input causes 4GL to take the following actions:

- Set the global variable **int_flag** to TRUE.
- Terminate the CONSTRUCT statement (except the AFTER CONSTRUCT block, if any) but not the 4GL program.

If 4GL previously executed a DEFER QUIT statement in the program, pressing the Quit key while CONSTRUCT is awaiting input causes 4GL to take the following actions:

- Set the global variable **quit_flag** to TRUE.
- Terminate the CONSTRUCT statement (except the AFTER CONSTRUCT block, if any) but not the 4GL program.

In both cases, the variable that stores the query criteria is set to NULL, a value that causes an SQL error if you attempt to use it as the WHERE clause. To avoid this problem, set any non-zero value of **int_flag** or **quit_flag** to zero (FALSE) before the CONSTRUCT statement begins execution.

When the user terminates a CONSTRUCT statement, 4GL executes the statements in the AFTER CONSTRUCT clause, unless the CONSTRUCT statement is terminated by an EXIT CONSTRUCT statement. In this case, the statements in the AFTER CONSTRUCT clause and in the AFTER FIELD clause of the current field are not executed. When NEXT FIELD appears in either of these clauses, 4GL ignores the Accept keystroke, and focus moves to the specified field.

The following program segment uses a simple CONSTRUCT statement to specify the search condition of a WHERE clause. The variable query_1 is declared as CHAR(250), and the cursor_1 cursor executes the query.

```
CONSTRUCT BY NAME query_1
   ON order_num, customer_num, order_date, ship_date
   ATTRIBUTE(BOLD)

LET s1 = "SELECT * FROM orders
   WHERE ", query_1 CLIPPED,
   " ORDER BY order_date, order_num"
PREPARE s_1 FROM s1
DECLARE cursor_1 CURSOR FOR s_1
FOREACH cursor_1 INTO order_rec.*
   ...
END FOREACH
```

The following program fragment demonstrates six CONSTRUCT input control blocks:

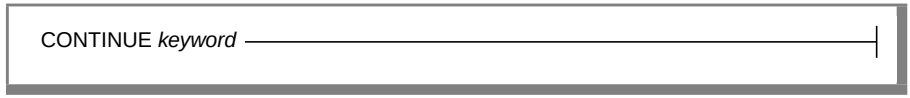
```
CONSTRUCT BY NAME query_1 ON customer.*

   BEFORE CONSTRUCT
      MESSAGE "Enter search criteria; ",
         "press ESC to begin search."
      DISPLAY "Press F1 or CTRL-W for field help." AT 2,1
   ON KEY (F1, CONTROL-W)
      CALL customer_help()  -- display field level help
   BEFORE FIELD state
      MESSAGE "Press F2 or CTRL-B ",
         "to display a list of states."
   ON KEY (F2, CONTROL-B)
      IF INFIELD(state) THEN
         CALL statehelp()   -- display list of states
      END IF
   AFTER FIELD state
      MESSAGE "Enter search criteria; ",
         "press ESC to begin search."
   AFTER CONSTRUCT     -- check for blank search criteria
      IF NOT FIELD_TOUCHED(customer.*) THEN
         PROMPT "Do you really want to see ",
            "all customer rows? (y/n) "
            FOR CHAR answer

         IF answer MATCHES "[Nn]" THEN
            MESSAGE "Enter search criteria; ",
               "press ESC to begin search."
         CONTINUE CONSTRUCT -- reenter query by example
         END IF
      END IF
END CONSTRUCT

LET s1 = "SELECT * FROM customer WHERE ", query_1 CLIPPED
PREPARE s_1 FROM s1
DECLARE q_curs CURSOR FOR s_1
DISPLAY "" AT 2,1               -- clear line 2 of text
LET exist = 0
```

References

DECLARE, DEFER, DISPLAY FORM, EXECUTE, LET, OPEN FORM, OPEN WINDOW, OPTIONS, SELECT, PREPARE

CONTINUE

The CONTINUE statement transfers control of execution from a statement block to another location in the currently executing compound statement.



Element	Description
keyword	specifies the current 4GL statement. You can choose from the keywords CONSTRUCT, FOR, FOREACH, INPUT, MENU, and WHILE.

Usage

You can use CONTINUE within a statement block of the currently executing compound statement that *keyword* specifies. This is a runtime instruction to transfer control within the current statement. (Use the EXIT keyword, rather than CONTINUE, to terminate the compound statement unconditionally.)

The use of CONTINUE in WHENEVER statements is described in "The CONTINUE Option" on page 4-381.

CONTINUE in CONSTRUCT, INPUT, and INPUT ARRAY Control Blocks

The CONTINUE CONSTRUCT and CONTINUE INPUT statements cause 4GL to skip all subsequent statements in the current control block. The screen cursor returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field. For more information, see "The EXIT CONSTRUCT Statement" on page 4-54 and "The CONTINUE INPUT Statement" on page 4-214.

CONTINUE INPUT is valid in INPUT and INPUT ARRAY statements.

CONTINUE in FOR, FOREACH, and WHILE Loops

The CONTINUE FOR, CONTINUE FOREACH, or CONTINUE WHILE keywords cause the current FOR, FOREACH, or WHILE loop (respectively) to begin a new cycle immediately. If conditions do not permit a new cycle, however, the looping statement terminates. For more information, see "The CONTINUE FOR Statement" on page 4-129, "The CONTINUE FOREACH Keywords" on page 4-137, and "The CONTINUE WHILE Statement" on page 4-383.

CONTINUE in MENU Control Blocks

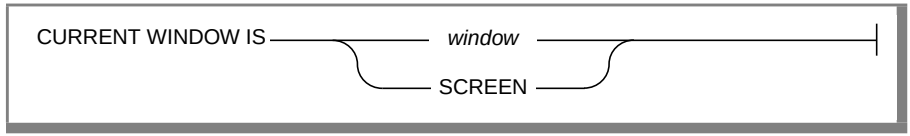
The CONTINUE MENU statement causes 4GL to ignore the remaining statements in the current MENU control block and redisplay the menu. The user can then choose another menu option. (For more information, see "MENU" on page 4-248.)

References

CONSTRUCT, FOR, FOREACH, GOTO, INPUT, INPUT ARRAY, MENU, WHILE, WHENEVER

CURRENT WINDOW

The CURRENT WINDOW statement makes a specified 4GL window the current window (that is, the top-most 4GL window in the window stack).



Element	Description
window	is the name of the open 4GL window to be made current.

Usage

4GL maintains a list or *stack* of all open 4GL windows in the 4GL screen. The OPEN WINDOW statement creates a new 4GL window that is added to the top of this window stack, becoming the *current window*. When you close a 4GL window, that 4GL window is removed from the stack. The top-most 4GL window among those that remain open becomes the new current window. Its values take effect for the positions of reserved lines like Prompt, Message, Form, and Comment lines.

The current 4GL window is always completely visible and can obscure all or part of any inactive 4GL windows. When you specify a new current window, 4GL adjusts the window stack by moving the new current 4GL window to the top and closing the gap in the stack left by this 4GL window.

Programs with multiple 4GL windows might need to switch to a different open window so that input and output occur in the appropriate 4GL window. To make a 4GL window the current window, use the CURRENT WINDOW statement. For example, this statement makes **win1** the current 4GL window:

```
CURRENT WINDOW IS win1
```

When a program starts, the 4GL screen is the current 4GL window. Its name is SCREEN. To make this the current 4GL window, specify the keyword SCREEN instead of a *window* identifier:

```
CURRENT WINDOW IS SCREEN
```

If a 4GL window contains a form, that form becomes the current form when a CURRENT WINDOW statement specifies the name of that 4GL window.

The CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, and MENU statements use only the current 4GL window for input and output. If the user displays another form (for example, through an ON KEY clause) in one of these statements, the 4GL window containing the new form becomes the current window. When the CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or MENU statement resumes, its original 4GL window becomes the current window.

The next program fragment opens multiple 4GL windows, including one called **w2**. Interactive statements that use the 4GL window **w2** can follow the CURRENT WINDOW statement within the **do2( )** function. If the function **do2( )** terminates after assigning the modular variable **next_win** any value but 2, the CALL statement in the WHILE loop invokes a different function. The **w2** 4GL window remains current until another CURRENT WINDOW statement specifies some other 4GL window, or until CLOSE WINDOW **w2** is executed.

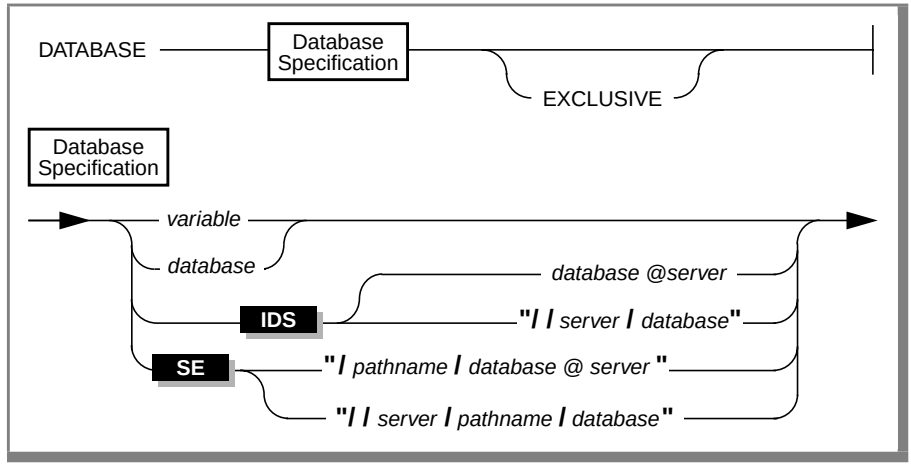
```
DEFINE next_win INTEGER
MAIN
OPEN WINDOW w1 AT 3,3 WITH FORM "cust1"
OPEN WINDOW w2 AT 9,15 WITH FORM "cust2"
OPEN WINDOW w3 AT 15,27 WITH FORM "cust3"
   . . .
   LET next_win = 1
   WHILE next_win IS NOT NULL
      CASE (next_win)
       WHEN 1 CALL do1()
       WHEN 2 CALL do2()
       WHEN 3 CALL do3()
   . . .
      END CASE
   END WHILE
CLOSE WINDOW w1
CLOSE WINDOW w2
CLOSE WINDOW w3
   . . .
END MAIN
FUNCTION do2()
   LET next_win = NULL
   CURRENT WINDOW IS w2
    . . .
END FUNCTION
```

References

CLEAR, CLOSE WINDOW, DISPLAY ARRAY, INPUT, INPUT ARRAY, MENU, OPEN WINDOW, OPTIONS

DATABASE

The DATABASE statement opens a default database at compile time, or a current database at runtime. (See also the description of the DATABASE statement in the *Informix Guide to SQL: Syntax*.)



Element	Description
database	is the name of a database. Blank spaces are not valid between quotation marks or after the @ symbol.
pathname	is the path to the parent directory of the **.dbs** directory.
server	is the name of the host system where database resides.
variable	is a variable that contains the database specification (as described in "The Database Specification" on page 4-72). You can specify *variable* only in a MAIN or FUNCTION block.

Usage

This statement is not required if your 4GL program does not reference entities in a database. You can use the DATABASE statement in two distinct ways, depending on the context of the statement within its source code module:

- You can specify the default database (as described in "The Default Database at Compile Time" on page 4-73) for the compiler to use in declaring data types indirectly in DEFINE statements, or for INITIALIZE or VALIDATE to access **syscolatt** or **syscolval**. (The default database is opened automatically at runtime.)
- You can specify the current database (as described in "The Current Database at Runtime" on page 4-74), so that SQL statements can access data and other entities in that database at runtime.

The Database Specification

The DATABASE statement can specify any accessible database on the current Informix database server (or on another server, if you also specify its name). This becomes the default database at compile time, and the current database at runtime. To reference entities in any other database, you must use the CLOSE DATABASE statement and then another DATABASE statement or table qualifiers. (For more information, see "Table Qualifiers" on page 3-89.)

The DATABASE statement closes any other open database on the same database server. If a database is open on another server, you must first use CLOSE DATABASE explicitly to close that current database, or an error occurs. An error results if you specify a database that 4GL cannot locate or cannot open or for which the user of your program does not have access privileges.

SE

Only the databases stored in your current directory, or in a directory specified in your **DBPATH** environment variable, are recognized.

To specify a database that does not reside in your current directory or in a **DBPATH** directory, you must follow the DATABASE keyword with a complete pathname, or with a program variable that evaluates to the full pathname of the database (excluding the **.dbs** extension). ♦

GLS

When the DATABASE statement establishes the database connection between 4GL and the database server, the locale categories for COLLATION and CTYPE on the client system are transmitted with the request for database service. The database server uses these settings to compare the user locale and the database locale. If the user locale and database locale do not match, the request for database service is rejected. This process is referred to as *locale consistency checking*.

The CTYPE and COLLATION categories at the time of database creation are stored with the database in a system table. These values are kept unchanged throughout the life of the database to ensure the consistent use of collating sequences, code sets, and formatting rules. You cannot change the character set and collation settings for a database; you must unload and reload all the data into a different database to change locales. For more information on nondefault locales and locale-defined collation, see Appendix E, "Developing Applications with Global Language Support." ♦

The Default Database at Compile Time

The DEFINE statement can specify that a record is LIKE a table, or that a variable is LIKE a column in a database table. (For details, see "Indirect Typing" on page 4-83.) Even if you qualify the name of the table with a database name, this requires a DATABASE statement to specify a default database at compile time. The compiler looks in this default database for the schema of tables whose columns are to be used as templates for declaring variables indirectly through the LIKE keyword.

To declare variables indirectly, the DATABASE statement must precede any program block in each module that includes a DEFINE…LIKE declaration, and must precede any GLOBALS…END GLOBALS statement (described in "GLOBALS" on page 4-145). It must also precede any DEFINE…LIKE declaration of module variables. Here the *database* name must be expressed explicitly, and not as a variable. The EXCLUSIVE keyword is not valid in this context. (The INITIALIZE…LIKE and VALIDATE…LIKE statements likewise require that DATABASE specify a default database before the first program block in the same module.)

If you want different program blocks to use the same database, you can repeat the same DATABASE statement in each program block in which entities in the database are referenced or created. Alternatively, you can create a file that includes only the DATABASE and the GLOBALS…END GLOBALS statements, and then include GLOBALS "*filename*" statements at the beginning of each module that requires the DATABASE statement.

The next example shows the contents of a file in which no global variables are declared, but the **zeitung** database can be accessed by statements in any other source modules that include the GLOBALS "*filename*" statement:

```
DATABASE zeitung
GLOBALS
END GLOBALS
```

Here GLOBALS…END GLOBALS can also include DEFINE statements.

The Current Database at Runtime

If your 4GL program is designed to interact with a database at runtime, the DATABASE statement must specify a current database that subsequent SQL statements can reference, until it is closed (by CLOSE DATABASE or by another DATABASE statement, for example), or until the program terminates.

In this case, the DATABASE statement must occur in a FUNCTION or the MAIN program block, and must follow any DEFINE statements in that block, or else it must precede the MAIN program block. When DATABASE specifies the current database, the database specification can be in a 4GL variable, and you can include the EXCLUSIVE keyword. (For more information, see "The EXCLUSIVE Keyword" on page 4-75.)

If a DATABASE statement (or a GLOBALS "*filename*" statement, where *filename* includes the DATABASE statement) precedes the MAIN statement, then the 4GL compiler (in effect) inserts the same DATABASE statement into the beginning of the MAIN program block, before the first executable statement, if no other DATABASE statement precedes MAIN. In this special case, the same DATABASE statement produces both compile-time and runtime effects.

You cannot include the DATABASE statement within a REPORT program block. If a 4GL report definition requires a two-pass report (as described in "The EXTERNAL Keyword" on page 7-27), an error occurs if no database is open when the report is run, or if the report driver issues a DATABASE statement while the report is running.

You cannot include the DATABASE statement in a multiple-statement PREPARE operation. (See also the descriptions of the PREPARE statement and the CLOSE DATABASE statement in the *Informix Guide to SQL: Syntax*.)

The EXCLUSIVE Keyword

The DATABASE statement with the EXCLUSIVE keyword opens the database in exclusive mode but prevents access by anyone but the current user. It is valid only in a FUNCTION or MAIN program block. To allow others to access a database that was opened in EXCLUSIVE mode, you must execute the CLOSE DATABASE statement. Then use DATABASE without the EXCLUSIVE keyword to reopen the database, if appropriate.

The following statement opens the **stores7** database on the **mercado** server in exclusive mode:

```
DATABASE stores7@mercado EXCLUSIVE
```

If another user already has the specified database open, exclusive access is denied, an error is returned, and no database is opened.

Testing SQLCA.SQLAWARN

You can determine the type of database that the DATABASE statement opens by examining the built-in **SQLCA.SQLAWARN** variable (as described in "Error Handling with SQLCA" on page 2-45) after the DATABASE statement has executed successfully:

- If the specified database uses transactions, **SQLCA.SQLAWARN[2]**, the second element of the **SQLCA.SQLAWARN** global record, contains a W.
- If the database is ANSI-compliant, **SQLCA.SQLAWARN[3]**, the third element of the **SQLCA.SQLAWARN** global record, contains a W.
- If Informix Dynamic Server is the database server, **SQLCA.SQLAWARN[4]**, the fourth element of the **SQLCA.SQLAWARN** global record, contains a W.

Effects of the Default Database on Error Handling

The database specified in a DATABASE statement that appears outside of any program block is the default database (as described in "The Default Database at Compile Time" on page 4-73). If you specify a default database, then runtime error handling is affected by whether or not this default database complies with the ANSI/ISO standard for SQL.

Error behavior depends on what kind of database the DATABASE statement references during compilation, rather than on the ANSI-compliant status at runtime. For example, if you compile against a database that is not ANSI-compliant but run against an ANSI-compliant database, the error behavior is as though the current database were not an ANSI-compliant database.

The default responses to error conditions differ between the ANSI-compliant method and the non-ANSI-compliant method as follows:

1. If ANSI compliance is requested and no WHENEVER ERROR statement is in effect, the default action after an error is CONTINUE.

   ANSI compliance is in effect if one of the following conditions exists:

   - There is a stated default database and it is ANSI-compliant.
   - The **-ansi** compilation flag is specified.
   - The **DBANSIWARN** environment variable is set.

   In releases of RDS before 7.2, for the last two of these conditions, the default error action was (improperly) STOP instead of CONTINUE.

2. If ANSI compliance is not in effect and no WHENEVER ERROR statement is in effect:

   - If the **-anyerr** flag is used, the default action is STOP.
   - If, instead, the **-anyerr** flag is not used, the default action after expression or data type conversion errors is CONTINUE; after other categories of errors, it is STOP.

If you compile part of the application against an ANSI-compliant database and part of it against a non-ANSI-compliant database, the parts of the application compiled against the ANSI-compliant database have the default error action of CONTINUE, and the parts compiled against the non-ANSI-compliant database have the default error action of STOP.

Additional Facts About Connections

4GL can also directly embed the CONNECT, SET CONNECT, and DISCONNECT statements of SQL. You cannot prepare these statements.

You must use a network connection, rather than a shared-memory connection, to connect a 32-bit 4GL client to a 64-bit database server.

4GL supports the stream-pipe interprocess connection mechanism to local hosts. You can use this mechanism to do distributed communication, if both systems are on the same computer. Unlike shared-memory connections, stream pipes do not pose the risk of being overwritten or being read by other programs that explicitly access the same part of shared memory. Stream-pipe connections, however, are slower than shared-memory connections and are not available on some computers. A stream-pipe connection requires **onipcstr** as the entry in the **nettype** field of the **sqlhosts** file.

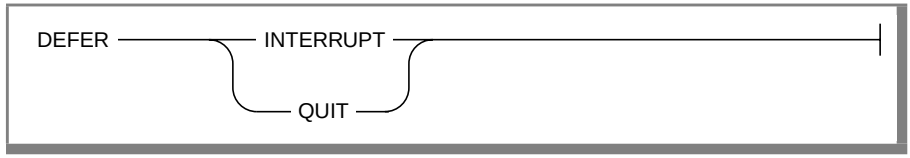
Server	nettype	hostname	Service
alpha	onipcstr	idcsun33	service1

References

DEFINE, FUNCTION, GLOBALS, INITIALIZE, MAIN, REPORT, VALIDATE

DEFER

The DEFER statement prevents 4GL from terminating program execution when the user presses the Interrupt key or the Quit key.



Usage

DEFER is a method of intercepting asynchronous signals from outside the program. Unless it includes the DEFER statement, the 4GL application terminates whenever the user presses the Interrupt or Quit key. The Interrupt key is CONTROL-C, and the Quit key is CONTROL-\. The DEFER statement tells 4GL to set a built-in global variable to a non-zero value, rather than terminate, when the user presses one of these keys:

- If the user presses the Interrupt key when DEFER INTERRUPT has been specified, 4GL sets the built-in global variable **int_flag** to TRUE.
- If the user presses the Quit key when DEFER QUIT has been specified, 4GL sets the built-in global variable **quit_flag** to TRUE.

The DEFER INTERRUPT and DEFER QUIT statements can appear only in the MAIN program block, and only once in any program. Once executed, the DEFER statement remains in effect for the duration of the program; you cannot restore the original function of the Interrupt key or the Quit key.

4GL programs can include code to check whether **int_flag** or **quit_flag** is TRUE, and if so, to take appropriate action. Be sure also to reset **int_flag** or **quit_flag** to FALSE (that is, to zero) so that subsequent tests are valid.

Interrupting Screen Interaction Statements

If DEFER INTERRUPT has executed, you can specify INTERRUPT to make the Interrupt key the activation key in an ON KEY clause of CONSTRUCT, INPUT ARRAY, and INPUT statements. If the user presses the Interrupt key, control returns to the same field, unless the statement block includes the EXIT or NEXT FIELD keywords.

Without the ON KEY (INTERRUPT) specification, an Interrupt signal transfers control to the AFTER INPUT or AFTER CONSTRUCT control block, if these are present, or else to END INPUT or END CONSTRUCT. Any AFTER FIELD clause for the current field is ignored, and the **int_flag** is reset to TRUE. (After DEFER QUIT, pressing the Quit key resets the **quit_flag** to TRUE, but the Quit key has no effect on CONSTRUCT, INPUT ARRAY, and INPUT statements.)

To make sure that **int_flag** or **quit_flag** is reset, you can use the LET statement to set both variables to FALSE immediately before the CONSTRUCT, DISPLAY ARRAY, INPUT, MENU, and PROMPT statements. After DEFER INTERRUPT has executed, if the user presses the **Interrupt** key during any DISPLAY ARRAY or PROMPT statement, program control leaves the current statement, and 4GL sets the **int_flag** to a non-zero value. (When a MENU statement is executing, however, program control remains in the MENU statement.)

To have the user terminate a statement with a key other than the Interrupt key, use the ON KEY clause to define the action of the desired key sequence.

The next program fragment executes the DEFER INTERRUPT statement in the MAIN program block, and then calls a function that prompts the user to enter criteria for retrieving data from the **stock** table.

```
MAIN
   . . .
   DEFER INTERRUPT
   . . .
   CALL find_stock()
   . . .
END MAIN

FUNCTION find_stock()
   DEFINE
      where_clause CHAR(200)
   . . .
   DISPLAY "Enter selection criteria for ",
         "the stock items you want." AT 10,1
   LET int_flag = FALSE
```

```
    CONSTRUCT BY NAME where_clause
          ON stock.* FROM s_stock.*
    IF int_flag THEN
       ERROR "Query cancelled."
       RETURN
    END IF
    . . .
END FUNCTION
```

If the user decides not to enter any selection criteria, pressing the Interrupt key terminates the CONSTRUCT statement without executing the query.

If **int_flag** flag is set to a non-zero value (TRUE), the program terminates the function by executing a RETURN statement. Notice that the function resets the value of **int_flag** to FALSE (zero) before beginning the CONSTRUCT statement.

Here if **int_flag** is set to a non-zero value (evaluates to TRUE), a RETURN statement terminates the function. Notice that in this example, the **find_stock( )** function explicitly resets the value of **int_flag** to FALSE (zero) before beginning the CONSTRUCT statement.

Interrupting SQL Statements

To enable the Interrupt key to interrupt SQL statements, your program must contain:

- the DEFER INTERRUPT statement.
- the OPTIONS statement with the SQL INTERRUPT ON option.

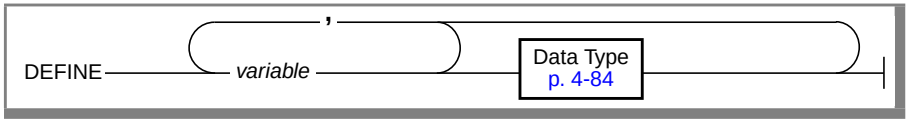
The keywords SQL INTERRUPT OFF restore the default of uninterruptable SQL statements. "Interrupting SQL Statements" on page 4-301 describes this feature in detail and its effect on the current database transaction.

References

CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, MAIN, MENU, OPTIONS, PROMPT, WHENEVER

DEFINE

The DEFINE statement declares the names and data types of 4GL variables.



Element	Description
variable	is a name that you declare here as the identifier of a variable.

Usage

A *variable* is a named location in memory that can store a single value, or an ordered set of values. Except for predefined global variables like **status**, **int_flag**, **quit_flag**, or the **SQLCA** record, you cannot reference any program variable before it has been declared by the DEFINE statement.

Releases of 4GL prior to 7.3 supported a total of no more than 64,535 bytes in all the names of variables in a single 4GL program, including record members and redefined variables. In this release, however, the upper limit on global string space (which includes variables, regardless of their scope, and certain other named 4GL program entities) is now 2 gigabytes (= 2,048 megabytes). Your available system resources might impose a lower limit. In programs that are compiled to p-code, however, a single 4GL function or report can have a total of no more than 32,767 bytes in the names of all its variables.

The GLOBALS "*filename*" statement can extend the visibility of module-scope variables that you declare in *filename* to additional source code modules.

The following sections describe these topics:

The Context of DEFINE Declarations

The DEFINE statement declares the identifier of one or more 4GL variables. There are two important things to know about these identifiers:

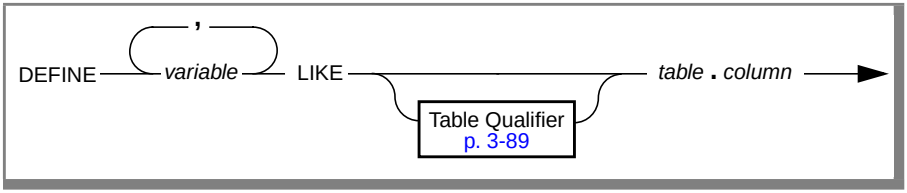
- Where in the program can they be used? The answer defines the scope of reference of the variable. A point in the program where an identifier can be used is said to be *in* the scope of the identifier. A point where the identifier is not known is *outside* the scope of the identifier.
- When is storage for the variable allocated? Storage can be allocated either statically, when the program is loaded to run (at load time), or dynamically, while the program is executing (at runtime).

The context of its declaration in the source module determines where a variable can be referenced by other 4GL statements, and when storage is allocated for the variable in memory. The DEFINE statement can appear in only two contexts:

1. Within a FUNCTION, MAIN, or REPORT program block, DEFINE declares *local* variables, and causes memory to be allocated for them. These DEFINE declarations of local variables must precede any executable statements within the same program block.
   - The scope of reference of a local variable is restricted to the same program block. Elsewhere, the variable is not visible.
   - Storage for local variables is allocated when its FUNCTION, REPORT, or MAIN block is entered during execution. Functions can be called recursively, and each recursive entry creates its own set of local variables. The variable is unique to that invocation of its program block. Each time the block is entered, a new copy of the variable is created.
2. Outside any FUNCTION, REPORT, or MAIN program block, the DEFINE statement declares names and data types of *module* variables, and causes storage to be allocated for them. These declarations must appear before any program blocks.
   - Scope of reference is from the DEFINE statement to the end of the same module (but the variable is not visible within this scope in program blocks where a local variable has the same identifier).
   - Memory storage for variables of module scope is allocated statically, in the executable image of the program.

Indirect Typing

You can use the LIKE keyword to declare a variable that has the same simple, BYTE, or TEXT data type as a specified column in a database table.



Element	Description
column	is the identifier of some column in *table*, as it appears in the **syscolumns** table of the system catalog.
table	is the identifier or synonym of a table or view in the default database that was specified in the DATABASE statement.
variable	is the 4GL identifier of a variable that you declare here.

If *table* is a view, then *column* cannot be based on an aggregate. If LIKE references a SERIAL column, the new variable is of the INTEGER data type.

The DATABASE statement must specify a default database before the first program block (or before the first DEFINE statement that uses LIKE to define module-scope or global variables) in the current module. (For more information, see "The Default Database at Compile Time" on page 4-73.) At compile time, 4GL substitutes a data type for the LIKE declaration, based on the schema of *table*. (If that schema is subsequently modified, recompile the module to restore the correspondence between variables and columns.)

Any column in the LIKE declaration has either a simple or a large data type. (These data types are described in sections that follow.) The table qualifier must specify *owner* if *table.column* is not a unique column identifier within its database, or if the database is ANSI-compliant and any user of your 4GL application is not the owner of *table*.

In the demonstration database, the **manufact** table has three columns:

- **manu_code** of data type CHAR(3)
- **manu_name** of data type CHAR(15)
- **lead_time** of data type INTERVAL DAY(3) TO DAY

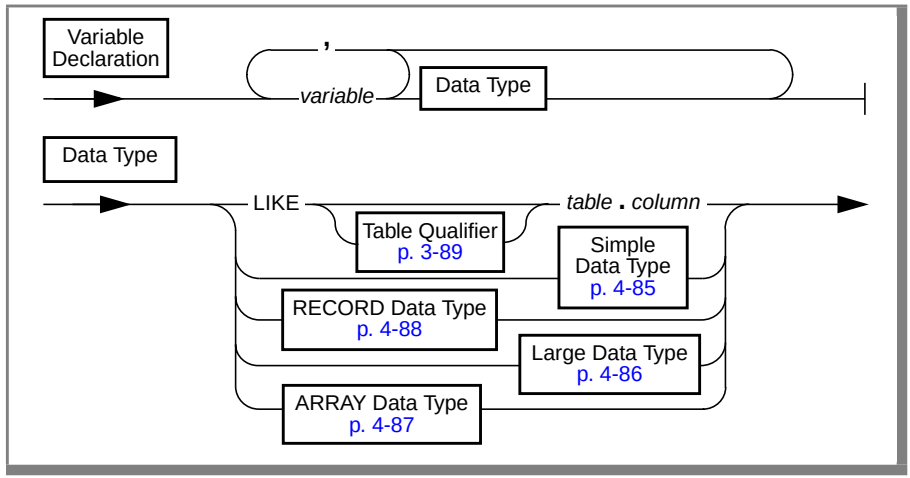
The following declarations of variables are based on the **manufact** table:

```
DATABASE demo5
DEFINE codename RECORD LIKE manufact.*
   -- equivalent to "manu_code char(3), manu_name char(15),
   --                lead_time interval day(3) to day"
   DEFINE hidden LIKE manufact.manu_code
   -- equivalent to "hidden char(3)"
   DEFINE leaden LIKE manufact.lead_time
   -- equivalent to "lead_time interval day(3) to day"
```

The LIKE keyword cannot reference column names that violate the naming rules for 4GL identifiers, such as restrictions on the character set or length.

Declaring the Names and Data Types of Variables

The DEFINE statement must declare the name and the data type of each new variable, either explicitly or else implicitly (by using the LIKE keyword).

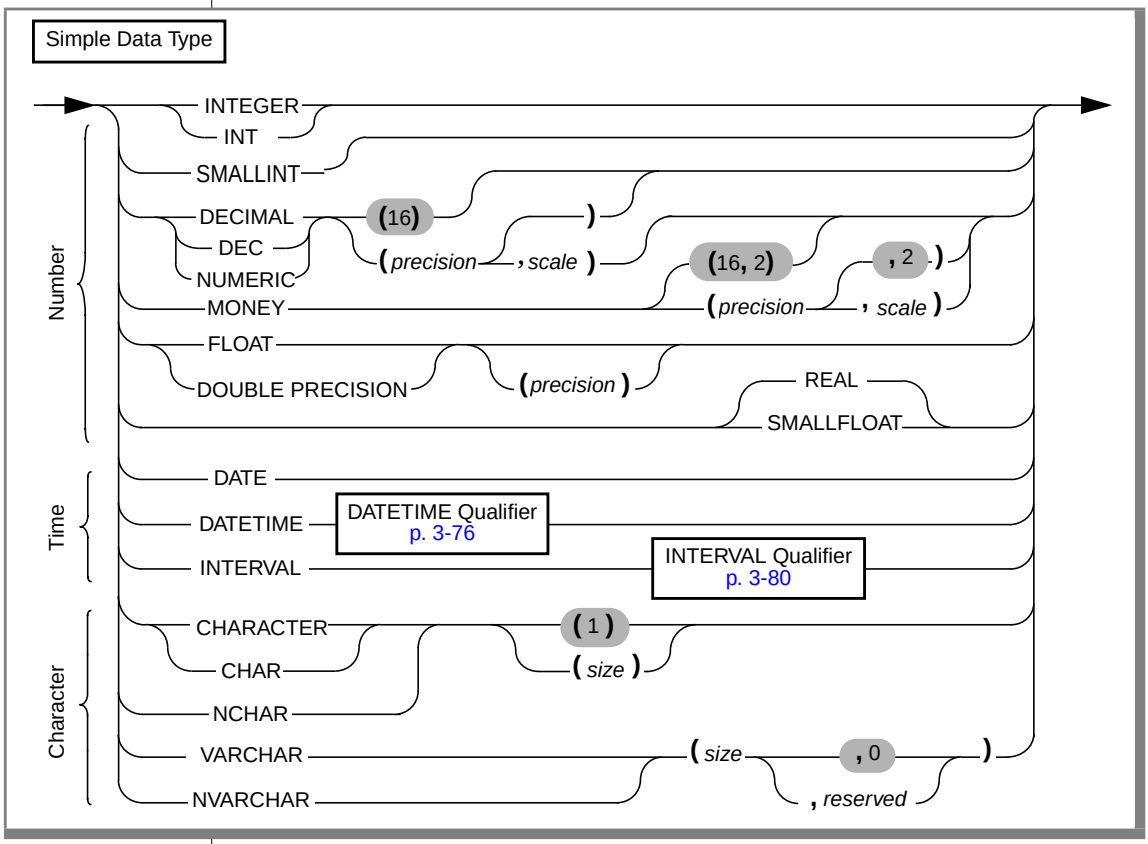


Element	Description
column	is the name of a database column.
table	is the name or synonym of a database table or view.
variable	is the name of the variable. This name must be unique among variables within the same scope of reference.

See "Data Types of 4GL" on page 3-6 for details of the various data types that you can specify when you declare 4GL variables.

Variables of Simple Data Types

The simple data types of 4GL have the following syntax.



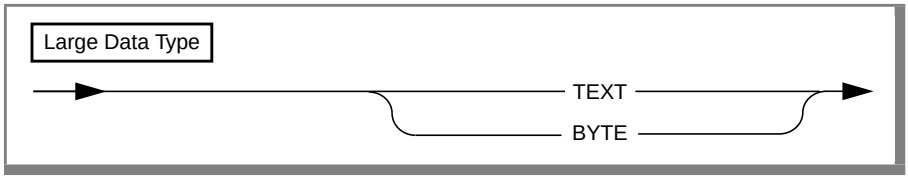
Element	Description
precision	is the number of significant digits. For FLOAT, the range is 1 ≤ *precision* ≤ 14. For DECIMAL and MONEY, the range is 1 ≤ *precision* ≤ 32.
reserved	is an SQL parameter that 4GL does not use.
scale	is the number of digits (≤ 32) in the fractional part of the number, where 0 ≤ *scale* ≤ *precision*. The actual scale can be less than 32.
size	is the maximum bytes that the data type can store. For CHAR, the range is 1 ≤ *size* ≤ 32,766. For VARCHAR, the range is 1 ≤ *size* ≤ 255.

All of these declaration parameters must be literal integers.

Variables of Large Data Types

4GL supports two data types for storing *binary large object* values, up to 2^{31} bytes in size (or up to a limit imposed by your system resources):

- TEXT, for character strings
- BYTE, for any data that can be stored on your system



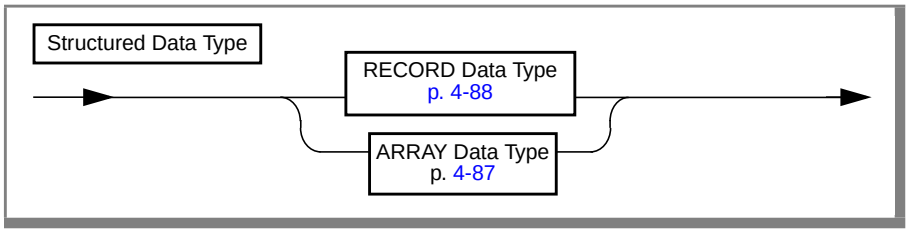
Unlike BYTE and TEXT declarations in SQL, DEFINE has no IN clause; in 4GL the LOCATE statement supports the functionality of the IN clause.

The CALL and RUN statements cannot include the BYTE or TEXT keyword in their RETURNING clauses. For more information, see "BYTE" on page 3-14 and "TEXT" on page 3-39.

Variables of Structured Data Types

4GL supports two *structured* data types for storing sets of values:

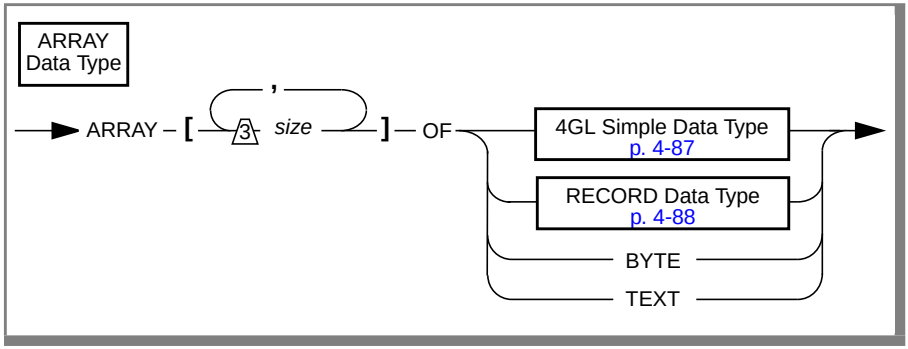
- ARRAY, for arrays of values of any single data type except ARRAY
- RECORD, for sets of values of any combination of data types



A database table cannot include a column of the ARRAY or RECORD data types because these 4GL data types are not part of the SQL language. For information on RECORD and ARRAY data types, see "ARRAY" on page 3-13 and "RECORD" on page 3-35. For information on using program arrays of records in interactive statements, see "INPUT ARRAY" on page 4-187 and "DISPLAY ARRAY" on page 4-102.

ARRAY Variables

The ARRAY keyword declares a structured variable that can store a 1-, 2-, or 3-dimensional array of values, all of the same data type.



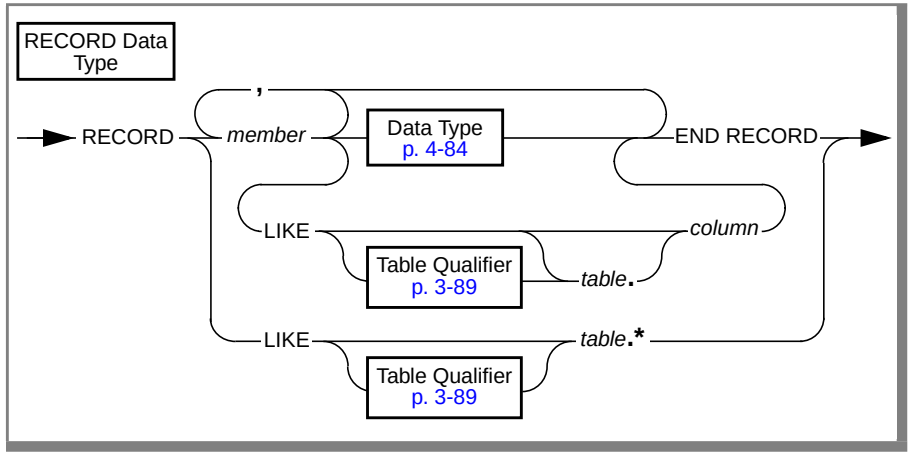
Element	Description
size	is the number (up to 32,767) of elements in a dimension. Dimensions can be different sizes, up to the limit of your C compiler.

The elements of an ARRAY variable can be of any data type except ARRAY, but an element can be a record that contains an array member.

You cannot specify an ARRAY data type as an argument or as a returned value of a 4GL function. The CALL and RUN statements cannot include the ARRAY keyword in their RETURNING clauses. In the DEFINE section of a REPORT statement, formal arguments cannot be declared as ARRAY data types, nor as RECORD variables that contain ARRAY members. (Data types of local variables that are not formal arguments are unrestricted.)

RECORD Variables

A 4GL program record is a collection of members, each of which is a variable. The member variables of a record can be of any 4GL data type, including the simple data types (described in "Declaring the Names and Data Types of Variables" on page 4-84), the structured (ARRAY and RECORD) data types, and the large (BYTE and TEXT) data types.



Element	Description
column	is a name of a column whose data type is the same as *member*.
member	is a name that you declare here for a member variable of the new record; this identifier must be unique within the record.
table	is the identifier or synonym of a table or view in the default database that was specified in the DATABASE statement.

The DATABASE statement must specify a default database before the first program block (or before the first DEFINE statement that uses LIKE to define module-scope or global variables) in the current module. (For more information, see "The Default Database at Compile Time" on page 4-73.)

Specify LIKE *table*.* to declare the record members implicitly, with identifiers and data types that correspond to all the non-SERIAL columns of *table*.

You do not need the END RECORD keywords to declare a single record whose members correspond to all the non-SERIAL columns of *table*:

```
recordname RECORD LIKE table.*
```

In this context, *table*.* cannot be a view containing an aggregate column.

You can use multiple LIKE clauses in the same RECORD declaration, provided that the LIKE keyword does not immediately follow the keyword RECORD:

```
DEFINE cust_ord_item
   RECORD
      cust_no LIKE customer.customer_num,
      ord RECORD LIKE orders.*,-- row from "orders" table
      it1 RECORD a1 LIKE items.item_num, -- subset of row
               b1 LIKE items.order_num -- in "items"
         END RECORD
      item_quantity LIKE items.quantity, --an "items" column
      it2 RECORD a2 LIKE items.total_price -- columns from
               b2 LIKE stock.unit,       -- various tables
               c2 LIKE manufact.manu_name
         END RECORD
   END RECORD
```

A compilation error occurs, however, if a LIKE clause begins the declaration of a record that is terminated by the END RECORD keywords. To declare a record with members that mirror the data types of a database table, but that also contains other members, declare one or more of the other members first. Then you can mix LIKE clauses and explicit variable declarations to the end of the record, as in the previous example.

Join columns often have the same name, but you must avoid the repetition of column names when using two or more LIKE clauses in the same scope of reference, so that both variables do not have the same name. In the demonstration database, both the **orders** and **items** tables include a column **order_num** that can join them. In the previous example, the record members declared LIKE the columns of **items** appear in the same order as in the table, but the record member that is declared like the second **order_num** column is called **item_order_num**.

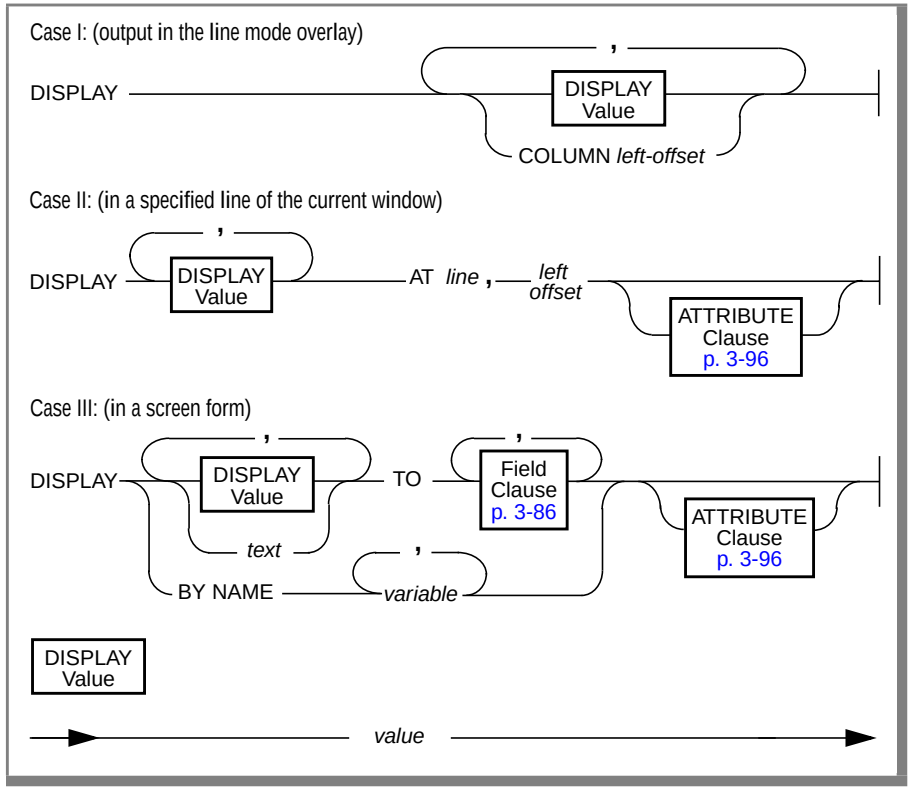
Important: *A scroll cursor cannot be used with a record that includes a member that is declared LIKE a BYTE or TEXT column.*

References

DATABASE, FUNCTION, GLOBALS, MAIN, REPORT

DISPLAY

The DISPLAY statement displays data values on the screen in line mode overlay, in a specified line of the current 4GL window, or in a form.



Element	Description
left offset	is an integer variable or a literal integer, specifying the horizontal coordinate of the first character of the next item of output.
line	is an integer variable or a literal integer, specifying the vertical coordinate of a line of the screen or of the current window.
text	is the name of a variable of the TEXT data type.
value	is a quoted string, a simple variable, a literal value, or a character string returned by a CLIPPED or USING expression.
variable	is the name of a variable that is also the name of a field.

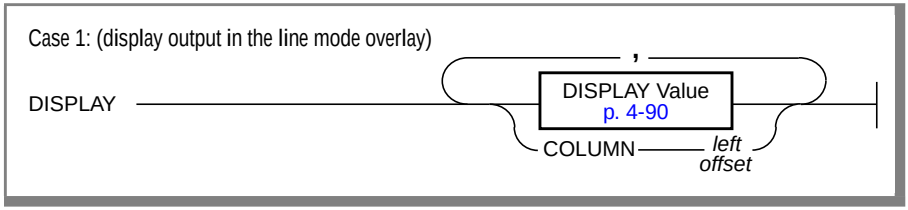
Usage

DISPLAY sends output directly to the screen, or to fields of a screen form.

DISPLAY cannot reference ARRAY or BYTE data types. After DISPLAY is executed, changing the value of a displayed variable has no effect on the current display until you execute the DISPLAY statement again. (To produce output within a REPORT, you must use PRINT rather than DISPLAY.) The following topics are described in this section:

Sending Output to the Line Mode Overlay

The DISPLAY statement without a qualifying TO, AT, or BY NAME clause (or with the COLUMN operator) sends output to the line mode overlay.



Element	Description
left offset	specifies the position of the first character of the next item of output within the line mode overlay.

Interactive 4GL statements produce screen output in either of two modes:

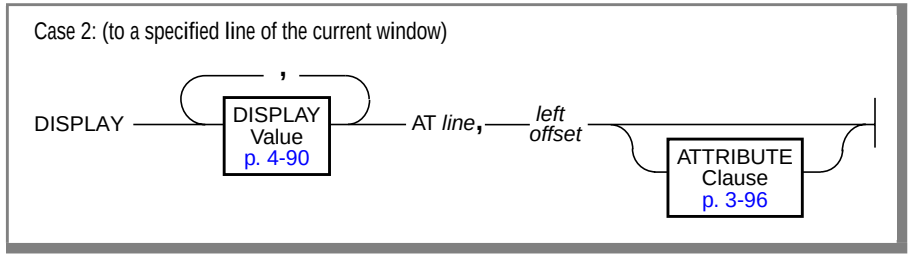
- *Formatted mode* statements: INPUT, INPUT ARRAY, CONSTRUCT, ERROR, MESSAGE, DISPLAY ARRAY, and DISPLAY (with any clause)
- *Line mode* statements: DISPLAY (without any clause)

The PROMPT statement produces output in whichever mode is current. When 4GL executes a DISPLAY statement that has no qualifying clause, a new 4GL window opens, called the *line mode overlay*, that covers the entire 4GL screen until another interactive statement produces formatted mode output.

If the next interactive statement is neither a line mode DISPLAY nor a PROMPT statement, the line mode overlay disappears, revealing the 4GL screen. Otherwise, any line mode DISPLAY statement continues the display in the next line of the line mode overlay.

Sending Output to the Current 4GL Window

In Case 2, you can include the AT keyword and specify coordinates to display output, beginning in the specified location in the current 4GL window.



Element	Description
left offset	is a literal integer that specifies the position of the first character of the next item of output within the specified *line*.
line	is an integer expression that returns a line number of the current 4GL window (or the 4GL screen itself, if no other 4GL window is current).

Formatting Screen Output

The DISPLAY statement supports only a subset of the syntax of character expressions (as described in "Character Expressions" on page 3-69). You can use the **record**.* or the THROUGH or THRU notation to reference the member variables of a record. (For more information, see "THRU or THROUGH Keywords and .* Notation" on page 3-92.)

You can refer to substrings of CHAR, VARCHAR, and TEXT variables by following the identifier with the starting and ending positions of the substring, separated by a comma and enclosed in brackets. For example, this statement displays characters 8 through 20 of the **full_name** variable:

```
DISPLAY "name", full_name[8,20],"added to database" AT 9, 2
```

You can use the following keywords to format the screen output:

- ASCII *number* (to display any ASCII character)
- CLIPPED (to truncate trailing blanks)
- COLUMN *number* (to begin output at a specified character position)
- USING "*string*" (to format values of number or DATE data types)

Important: *You cannot use the AT, ATTRIBUTE, BY NAME, or TO clause with the COLUMN operator.*

These operators are described in Chapter 5. No others are supported. If you want to display the current time, for example, you must assign the value of CURRENT to a program variable and then display that variable, rather than include the CURRENT operator among the list of DISPLAY values.

The following statement displays the values of two character variables in the format **lname**, **fname** on the next line, using the CLIPPED operator:

```
DISPLAY p_customer.lname CLIPPED, ", ", p_customer.fname
```

Unless you use the CLIPPED or USING operator, the DISPLAY statement formats character representations of the values of program variables and constants with display widths (including any sign) that depend on their declared data types, as the following table indicates.

Data Type	Default Display Width (in Characters)
CHAR	The length from the data type declaration
DATE	10
DATETIME	From 2 to 25, as implied in the data type declaration
DECIMAL	(2 + *m*), where *m* is the precision from the data type declaration
FLOAT	14
INTEGER	11
INTERVAL	From 3 to 25, as implied in the data type declaration

(1 of 2)

Data Type	Default Display Width (in Characters)
MONEY	(3 + *m*), where *m* is the precision from the data-type declaration
NCHAR	The length from the data type declaration
NVARCHAR	The maximum length from the data type declaration
SMALLFLOAT	14
SMALLINT	6
VARCHAR	The maximum length from the data type declaration

(2 of 2)

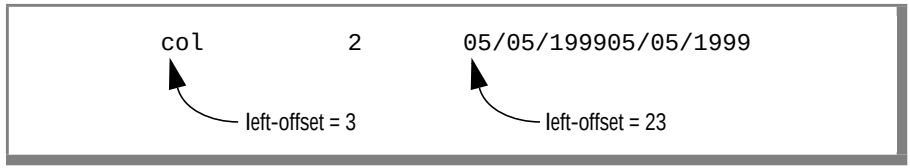
When no field is referenced by the TO or BY NAME keywords, output begins on the screen where the AT *line*, *left-offset* coordinates position it, or (if the AT clause is omitted) it defaults to the line below the current cursor position.

Unless the COLUMN operator specifies a nondefault character position, the output begins in the first character position, and successive output items within the same DISPLAY statement are *not* separated by blank spaces.

For example, suppose the following program fragment runs on May 5, 1999:

```
DEFINE col INTEGER, cow DATE
LET col = 2
LET cow = CURRENT
DISPLAY COLUMN 3, "col", col, COLUMN 23, cow, cow
   SLEEP 5
```

This DISPLAY statement would produce one line of output on May 5, 1999.



- Two blank spaces (the `COLUMN 3` specification)
- The string `col` (the `"col"` string specification)
- Ten blank spaces, followed by the character `2` (the `col` INTEGER variable)
- Seven blank spaces (the `COLUMN 23` specification)
- The string `05/05/199905/05/1999` (the `cow, cow` DATE variables)

Each DISPLAY statement begins its output on a new line. You can also use the AT clause to position output when no screen fields are specified by the TO or BY NAME clause. If no fields are specified, you cannot include an ATTRIBUTES clause in the DISPLAY statement, unless you also include the AT clause.

The AT Clause

You can use the AT clause to display text at a specified location in the current 4GL window, which can be the 4GL screen. The CLIPPED or USING operator can format the displayed values. You *cannot*, however, include the COLUMN operator in a DISPLAY statement that includes the AT clause.

The coordinates start with line 1 and character position 1 in the upper-left corner of the 4GL screen or the current 4GL window. The *line* values increase as you go down, and the character position values increase as you move from left to right. An error occurs if either coordinate value exceeds the dimensions of the 4GL screen or the current 4GL window. For example, the following DISPLAY statement displays the value of record member **total_price**, starting in line 22, at character position 5:

```
DISPLAY "TOTAL:  ", p_items.total_price AT 22, 5
```

Text that you display remains on the screen until you overwrite it. If you use the AT clause when the last variable is a NULL value of the CHAR data type, 4GL clears to the end of the line. If you execute a formatted-mode statement when line mode output from a DISPLAY statement with no clause is visible, 4GL clears the screen or the current 4GL window before producing formatted-mode display. (Formatted mode statements include ERROR, MESSAGE, PROMPT, and DISPLAY with any AT, BY NAME, or TO clause.)

Do not use DISPLAY AT to display text where it could overwrite useful data. Because INPUT clears the Comment line and the Error line when the cursor moves between fields, it is often a good idea *not* to display text in the following positions of the current 4GL window or the 4GL screen:

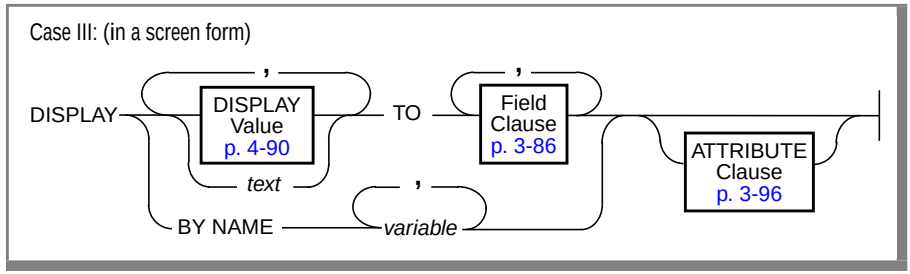
- The last line of the current 4GL window (the default Comment line)
- The last line of the 4GL screen (the default Error line)

To use these lines for text display, you must reposition the Comment and Error lines. The OPEN WINDOW and OPTIONS statements can position the Comment line, and `OPEN WINDOW...COMMENT LINE OFF` can hide the Comment line. The OPTIONS statement can position the Error line.

If the displayed text exceeds the size of the current 4GL window, 4GL truncates the text to fit the available space.

Sending Output to a Screen Form

You can use the TO clause or the BY NAME clause to display output in the fields of a screen form, using the formatted mode of display.



Element	Description
text	is the name of a variable of the TEXT data type.
variable	is the name of a variable that is also the name of a field.

Here you cannot use the COLUMN operator or the AT keyword to position output because the locations of fields within the form are fixed.

If 4GL was in line mode, this form of the DISPLAY statement first clears the screen before sending output to the fields of the form.

Character representations of values are displayed according to data type.

Type of Value	Display
Number	Right-justified. If the number does not fit in the field, 4GL fills the field with asterisks (*) to indicate an overflow.
Literal string, TEXT	Left-justified. If a character string does not fit in the field, 4GL truncates the display of the value.
BYTE	The field displays the message <byte value>, but actual BYTE values do not appear in the field. (The PROGRAM attribute, as described in Chapter 6, can display BYTE and TEXT values.)

Field attributes can change some of these default formats. For example, the LEFT attribute (described in Chapter 6) left-justifies numbers, and the FORMAT attribute can format DATE, DECIMAL, FLOAT, and SMALLFLOAT values. See also the PICTURE attribute (in Chapter 6) and the USING operator (in Chapter 5).

The BY NAME Clause

If the variables to be displayed have the same name as screen fields, you can use the BY NAME clause. The BY NAME clause binds the fields to variables implicitly. To use this clause, you must define variables with the same name as the screen fields where they will be displayed. 4GL ignores any record name prefix when matching the names. The names must be unique and unambiguous. If not, this option results in an error, and 4GL sets `status < 0`.

For example, the following statement displays the values for the specified variables in the screen fields with corresponding names (**company**, **address1**, **address2**, **city**, **state**, and **zipcode**):

```
DISPLAY BY NAME p_customer.company, p_customer.address1,
   p_customer.address2, p_customer.city, p_customer.state,
   p_customer.zipcode
```

You can produce the same result by using the THRU or THROUGH notation when listing the fields of the screen record:

```
DISPLAY BY NAME p_customer.company THRU p_customer.zipcode
```

This BY NAME clause displays data to the screen fields of the default screen records. The default screen records are those having the names of the tables defined in the TABLES section of the form specification file. To use a screen array, you define a screen array in addition to the default screen record. This default screen record holds only the first line of the screen array.

For example, the following DISPLAY statement displays the **ordno** variable only in the first line of the screen array (the default screen record):

```
DISPLAY BY NAME p_stock[1].ordno
```

To display **ordno** in all elements of the array, you can use the DISPLAY ARRAY statement, or DISPLAY and the TO clause, as in the next example:

```
FOR i = 1 TO 10
   DISPLAY p_stock[i].ordno TO sc.stock[i].ordno
   ...
END FOR
```

The TO Clause

If the variables do not have the same names as the screen fields, the BY NAME clause is not valid. Instead, you must use the TO clause to map variables to fields explicitly. You can list the fields individually, or you can use the *screen record*.* or *screen record*[*n*].* notation, where *screen record*[*n*].* specifies all the fields in line *n* of a screen array.

In a DISPLAY TO statement, any screen attributes specified in the ATTRIBUTE clause apply to all the fields that you specify after the TO keyword.

You can use the SCROLL statement to move such values up or down, but the DISPLAY ARRAY statement is generally more convenient to use with screen arrays. In the following example, the values in the **p_items** program record are displayed in the first row of the **s_items** screen array:

```
DISPLAY p_items.* TO s_items[1].*
```

The expanded list of screen fields must correspond in order and in number to the expanded list of identifiers after the DISPLAY keyword. Identifiers and their corresponding fields must have the same or compatible data types. For example, the next DISPLAY statement displays the values in the **p_customer** program record in fields of the **s_customer** screen record:

```
DISPLAY p_customer.* TO s_customer.*
```

For this example, the **p_customer** program record and the **s_customer** screen record require compatible declarations. The following DEFINE statement declares the **p_customer** program record:

```
DEFINE   p_customer RECORD
         customer_num LIKE customer.customer_num,
         fname        LIKE customer.fname,
         lname        LIKE customer.lname,
         phone        LIKE customer.phone
         END RECORD
```

This fragment of a form specification declares the **s_customer** screen record:

```
ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.phone;
END
```

```
INSTRUCTIONS
SCREEN RECORD s_customer (customer.customer_num,
                          customer.fname,
                          customer.lname,
                          customer.phone)
END
```

The ATTRIBUTE Clause

For general information, see "The ATTRIBUTE Clause" on page 4-41. This section describes the ATTRIBUTE clause within the DISPLAY statement. The ATTRIBUTE clause is valid only if you also use the BY NAME, TO, or AT clause. At least one of the following keywords (no more than one of which can be a *color* keyword) must appear in the ATTRIBUTE clause.

Intensity Keyword	Interpretation	Color Keyword	Interpretation
NORMAL	White	WHITE	Normal
BOLD	Red	YELLOW	Bold
DIM	Blue	MAGENTA	Bold
INVISIBLE	Non-printing	RED	Bold
REVERSE	Reverse	CYAN	Dim
BLINK	Blink	GREEN	Dim
UNDERLINE	Underline	BLUE	Dim
		BLACK	Dim

The ATTRIBUTE clause temporarily overrides any default display attributes or any attributes specified in the OPTIONS or OPEN WINDOW statements for the fields. When the DISPLAY statement completes execution, the default display attributes are restored.

The column labeled "Interpretation" indicates how an attribute appears on a color terminal (for the first four keywords) or on a monochrome terminal (for the subsequent keywords). For example, on color terminals, NORMAL is interpreted as WHITE, and BOLD is interpreted as RED.

The REVERSE, BLINK, INVISIBLE, and UNDERLINE attributes are not sensitive to the color or monochrome status of the terminal, if the terminal is capable of displaying these intensity modes.

The ATTRIBUTE clause can include zero or more of the BLINK, REVERSE, and UNDERLINE attributes, and zero or one of the other attributes.

That is, all of the attributes except BLINK, REVERSE, and UNDERLINE are mutually exclusive. For information about additional field attributes and other form specifications, see Chapter 6.

These interpretations also apply to the ATTRIBUTE clause of the CONSTRUCT, DISPLAY ARRAY, DISPLAY FORM, INPUT, and INPUT ARRAY statements. The following DISPLAY statement specifies the attributes REVERSE and BLUE for the message that will be displayed on line 12, starting in the first column:

```
DISPLAY " There are ", num USING "#####",
   " items in the list" AT 12,1
   ATTRIBUTE(REVERSE, BLUE)
```

While the DISPLAY statement is executing, 4GL ignores the INVISIBLE attribute, regardless of whether you specify it in the ATTRIBUTE clause.

Displaying Numeric and Monetary Values

The MONETARY and NUMERIC categories of the locale files (respectively) specify default display formats of number and currency data values; see the descriptions of MONETARY and NUMERIC in the *Informix Guide to GLS Functionality*. The **DBFORMAT** and **DBMONEY** environment variables (and the USING operator) can affect the display of numeric and monetary data values as follows:

- A leading currency symbol (as set by **DBFORMAT** or **DBMONEY**) can precede MONEY values. If the FORMAT attribute specifies a leading currency symbol for other data types, 4GL displays that symbol.
- 4GL omits the *thousands* separators in DISPLAY statements, unless they are specified by a FORMAT attribute or by the USING operator.
- 4GL displays the decimal separator, except for INT or SMALLINT values.
- 4GL displays the trailing currency symbol (as set by **DBFORMAT** or **DBMONEY**) for MONEY values, unless you specify a FORMAT attribute or the USING operator. In this case, 4GL ignores the trailing currency symbol; the user cannot enter a trailing currency symbol, and 4GL does not display it.

GLS

- The MONETARY and NUMERIC categories of the locale files can specify default display formats that are distinct for currency and number values. In some locales, such as those that support the Italian or Portuguese languages, it is conventional for currency values to be displayed in a different format from other numeric values. ♦

For more information on **DBFORMAT** and **DBMONEY**, refer to Appendix D, "Environment Variables."

Displaying Time Values

The **DBDATE**, **GL_DATE**, and **GL_DATETIME** environment variables, and the USING operator, can affect the display of DATE and DATETIME values.

For more information on **DBDATE**, refer to Appendix D, "Environment Variables." The **GL_DATE**, and **GL_DATETIME** environment variables are described in *"Informix Guide to GLS Functionality."*

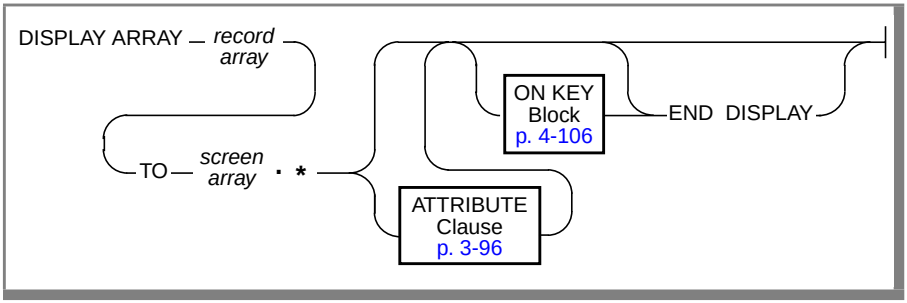
The **DBTIME** environment variable is supported by some Informix products, but has no effect on 4GL applications.

References

INPUT, DISPLAY ARRAY, DISPLAY FORM, OPEN WINDOW, OPTIONS, PRINT

DISPLAY ARRAY

The DISPLAY ARRAY statement displays program array values in a screen array, so that the user can scroll through the screen array.



Element	Description
record array	is the identifier of a program array of RECORD variables.
screen array	is the identifier of a screen array. (For more information, see "Screen Arrays" on page 6-77.)

Usage

The following steps describe how to use the DISPLAY ARRAY statement:

1. Define a screen array in the form specification file.
2. Use DEFINE to declare an array of program records whose members correspond in name, data type, and order to the screen array fields.
3. Open and display the screen form with either of the following statements:
   - The OPEN FORM and DISPLAY FORM statements
   - The OPEN WINDOW statement with the WITH FORM clause
4. Fill the program array with data to be displayed, counting the number of program records being filled with retrieved data.

5. Call the SET_COUNT(*x*) function, where *x* is the number of filled records.
6. Use the DISPLAY ARRAY statement to display the program array values in the screen array fields.

The SET_COUNT( ) function sets the initial value of the ARR_COUNT( ) function. If you do not call SET_COUNT( ), 4GL cannot determine how much data to display, and so the screen array remains empty. For a description of the syntax of the built-in SET_COUNT( ) function, see Chapter 5.

The DISPLAY ARRAY statement binds the screen array fields to the member records of the program array. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array). Each mapped variable must have the same data type or a compatible data type as the corresponding field.

The size of the screen array (from the form specification file) determines the number of program records that 4GL displays at one time on the screen. The size of the program array determines how many retrieved rows of data the program can store. The size of the program array can exceed the size of the screen array. In this case, the user can scroll through the rows on the form.

When 4GL encounters a DISPLAY ARRAY statement, it takes the following actions:

1. Displays the program array values in the screen array fields
2. Moves the cursor to the first field in the first screen record
3. Waits for the user to press a scroll key (by default, F3 or PAGE DOWN to scroll forward, or F4 or PAGE UP to scroll backwards) or the Accept key (ESCAPE by default)

Because the DISPLAY ARRAY statement does not terminate until the user presses the Accept or Interrupt key, you might want to display a message informing the user. By default, 4GL displays variables and constants as follows:

- Right-justifies number values in a screen field
- Left-justifies character values in a screen field
- Truncates the displayed value, if a character value is longer than the field

- Fills the field with asterisks (*) to indicate an overflow, if a number value is larger than the field can display
- If the field contains a BYTE value, displays <byte value> in the field

The following topics are described in this section:

The ATTRIBUTE Clause

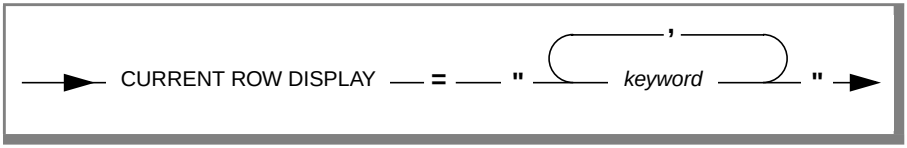
The ATTRIBUTE clause resembles the ATTRIBUTE clause of other form-based statements like CONSTRUCT. Except for CURRENT ROW DISPLAY, as described in the next section, attributes that you specify apply to all of the fields in *screen array*. For example, the following DISPLAY ARRAY statement displays items in RED:

```
DISPLAY ARRAY p_items TO s_items.* ATTRIBUTE (RED)
```

The ATTRIBUTE clause specifications override all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. While the DISPLAY ARRAY statement is executing, 4GL ignores the INVISIBLE attribute.

Highlighting the Current Row of the Screen Array

Besides the color and intensity attributes that are described in "ATTRIBUTE Clause" on page 3-96, the ATTRIBUTE clause of the DISPLAY ARRAY statement also supports the following syntax.



Element	Description
keyword	is zero or one of the *color* attribute keywords, and zero or more of the *intensity* attribute keywords (except DIM, INVISIBLE, and NORMAL) from the syntax diagram of "The ATTRIBUTE Clause" on page 4-41.

The comma-separated list of attributes within the quoted string is applied to the current row of *screen array*.

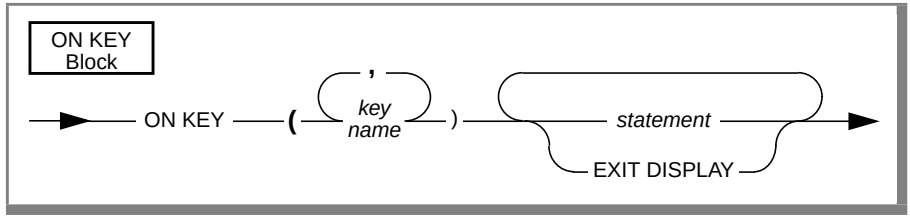
For example, the following specification displays *screen array* as in the previous example, but with the current row (the row that contains the screen cursor) in reverse video and green:

```
DISPLAY ARRAY p_items TO s_items.*
    ATTRIBUTE (RED, CURRENT ROW DISPLAY = "GREEN, REVERSE")
```

If the quoted string includes no *keyword*, an error is issued. If *screen array* has only one row, the CURRENT ROW DISPLAY attributes are applied to that row.

The ON KEY Blocks

The ON KEY keywords specify a block of statements to be executed when the user presses one of the specified keys.



Element	Description		
key name	is one or more of these keywords, in uppercase or lowercase letters, separated by commas, to specify a key:		
	ACCEPT	HELP	NEXT or NEXTPAGE
	DELETE	INSERT	PREVIOUS or PREVPAGE
	DOWN	INTERRUPT	RETURN
	ESC or ESCAPE	LEFT	TAB
	F1 through F64	RIGHT	UP
	CONTROL-char (except A, D, H, I, J, L, M, R, or X)		
statement	is an SQL statement or some other 4GL statement.		

For *key name*, you can substitute the NEXTPAGE keyword as a synonym for NEXT, and PREVPAGE as a synonym for PREVIOUS.

4GL executes the statements specified in the ON KEY block when the user presses one of the keys that you specify. 4GL deactivates the form while executing statements in an ON KEY block. After executing the statements, 4GL re-activates the form, allowing the user to continue viewing the fields.

You can enter uppercase or lowercase *key* specifications. The keys in the following table require special consideration before you reference them in an ON KEY clause.

Key	Special Considerations
ESC or ESCAPE	Specify another key as the Accept key in the OPTIONS statement, because ESC is the default Accept key.
Interrupt	You must execute a DEFER INTERRUPT statement, so that when the user presses the Interrupt key, 4GL executes the statements in the ON KEY clause and sets **int_flag** to nonzero for the current task, but does not terminate the DISPLAY ARRAY statement.
Quit	4GL also executes the statements in this ON KEY clause if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, 4GL sets **quit_flag** to non-zero for the current task.
CONTROL-*char*	
A, D, H, L, R, and X	4GL reserves these keys for field editing.
I, J, and M	If you specify these keys in the ON KEY clause, the key is "trapped" by 4GL to activate the ON KEY clause. The standard effect of these keys (TAB, LINEFEED, and RETURN, respectively) is not available to the user. For example, if CONTROL-M appears in an ON KEY clause, the user cannot press RETURN to advance the cursor to the next field.

You might not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

After executing the statements in the ON KEY block, 4GL resumes the display with the cursor in the same location as before the ON KEY block, unless it encounters EXIT DISPLAY within the block. (In this case, program execution resumes at the statement following the DISPLAY ARRAY statement.)

The following ON KEY clause specifies two keys to display a help message:

```
ON KEY (f1, control-w) CALL customer_help()
```

The EXIT DISPLAY Statement

The EXIT DISPLAY statement terminates the DISPLAY ARRAY statement. When it encounters an EXIT DISPLAY statement, 4GL takes the following actions:

1. Skips all subsequent statements between the EXIT DISPLAY keywords and the END DISPLAY keywords
2. Resumes execution at the statement after the END DISPLAY keywords

For example, the EXIT DISPLAY statement terminates the following DISPLAY ARRAY statement if the user presses F5 and the value of **amt_received** in the current program array record is greater than 1000:

```
DISPLAY ARRAY p_receipts TO s_receipts.*
   ON KEY (F5)
      LET x = arr_curr()
      IF p_receipts[x].amt_received > 1000 THEN
         CALL get_allocation(p_receipts[x].receipt_num)
         EXIT DISPLAY
      END IF
END DISPLAY
```

The END DISPLAY Keywords

The END DISPLAY keywords terminate the DISPLAY ARRAY statement. Each of these conditions requires that you include the END DISPLAY keywords:

- The DISPLAY ARRAY statement includes one or more ON KEY blocks.
- The DISPLAY ARRAY statement is specified in a form management block of a CONSTRUCT, INPUT, or INPUT ARRAY statement, and an ON KEY block of the enclosing statement follows the DISPLAY ARRAY statement.
- The DISPLAY ARRAY statement is specified within an ON KEY block in a PROMPT statement or in another DISPLAY ARRAY statement.

The following DISPLAY ARRAY statement must include the END DISPLAY keywords because it immediately precedes an ON KEY block that belongs to an INPUT statement:

```
INPUT BY NAME p_customer.*
   AFTER FIELD company
      ...
      DISPLAY ARRAY pa_array TO sc_array.*
      END DISPLAY
   ON KEY (CONTROL_B)
      ...
END INPUT
```

Otherwise, it would be ambiguous whether the ON KEY block were part of the INPUT statement or part of the DISPLAY ARRAY statement.

Here the END DISPLAY keywords are required because of the ON KEY clause:

```
DISPLAY ARRAY p_items TO s_items.*ON KEY (CONTROL_W)
      CALL get_help()
END DISPLAY
```

Using Built-In Functions and Operators

4GL provides several built-in functions to use in a DISPLAY ARRAY statement. These functions are described in Chapter 5 and are summarized here.

You can use the following built-in functions to keep track of the relative states of the screen cursor, the program array, and the screen array.

Function	Description
ARR_CURR( )	Returns the number of the *current record* of the program array. This corresponds to the position of the screen cursor at the beginning of the ON KEY control block, not the line to which the screen cursor moves after execution of the block
ARR_COUNT( )	Returns the current number of records in the program array.
SCR_LINE( )	Returns the number of the current line within the screen array. This number can be different from the value returned by ARR_CURR( ) if the program array is larger than the screen array.
SET_COUNT( )	Takes the number of rows currently in the program array as an argument, and sets the initial value of ARR_COUNT( ).

DISPLAY ARRAY also supports the following built-in functions and operators that allow you to access field buffers and keystroke buffers.

Feature	Description
FIELD_TOUCHED( )	Returns TRUE when the user has *touched* (made a change to) a screen field whose name is passed as an operand. Moving the screen cursor through a field (with the RETURN, TAB, or arrow keys) does not mark a field as touched.
GET_FLDBUF( )	Returns the character values of the contents of one or more fields in the currently active form.
FGL_GETKEY( )	Waits for a key to be pressed, and then returns an INTEGER corresponding to the raw value of the key that was pressed.
FGL_LASTKEY( )	Returns an INTEGER value corresponding to the most recent keystroke executed by the user while in the screen form.
INFIELD( )	Returns TRUE if the name of the field that is passed as its operand is the name of the current field.

For more about these built-in 4GL functions and operators, see Chapter 5. Each field in a form has only one field buffer, and a buffer cannot be used by two different statements simultaneously. If you plan to display the same form with data entry fields more than once, you can open a new 4GL window, and then open and display in it a second copy of the form. 4GL allocates a separate set of buffers to each form, and you can be certain that your program is retrieving the correct field values.

Scrolling During the DISPLAY ARRAY Statement

Users can select these keys to scroll through the screen array.

Key	Effect
↓, →	Moves the cursor down one row at a time. If the cursor was on the last row of the screen array before the user pressed one of these arrow keys, 4GL scrolls the program array data up one row. If the last row in the program array is already in the last row of the screen array, pressing one of these keys generates a message that says there are no more rows in that direction.
↑, ←	Moves the cursor up one row at a time. If the cursor was on the first row of the screen array before the user pressed one of these arrow keys, 4GL scrolls the program array data down one row. If the first row in the program array is already in the first row of the screen array, pressing one of these keys generates a message that says there are no more rows in that direction.
F3	Scrolls the display to the next full page of program array records. You can reset this key by using the NEXT KEY option of the OPTIONS statement.
F4	Scrolls the display to the previous full page of program array records. You can reset this key by using the PREVIOUS KEY option of the OPTIONS statement.

Completing the DISPLAY ARRAY Statement

The following conditions terminate the DISPLAY ARRAY statement:

- The user chooses any of the following keys:
  - The Accept key
  - The Interrupt key
  - The Quit key
- 4GL executes the EXIT DISPLAY statement.

By default, the Accept, Interrupt, and Quit keys terminate the DISPLAY ARRAY statement. Each of these actions also deactivates the form. (But pressing the Interrupt or Quit key can immediately terminate the program, unless the program also includes the DEFER INTERRUPT and DEFER QUIT statements.)

If 4GL previously executed a DEFER INTERRUPT statement in the program, pressing the Interrupt key causes 4GL to take the following actions:

1. Set the global variable **int_flag** to a nonzero value.
2. Terminate the DISPLAY ARRAY statement but not the 4GL program.

If 4GL previously executed a DEFER QUIT statement in the program, pressing the Quit key causes 4GL to take the following actions:

1. Set the global variable **quit_flag** to a nonzero value.
2. Terminate the DISPLAY ARRAY statement but not the 4GL program.

The following program fragment displays a program array **p_customer** in the fields of a screen array called **s_customer**:

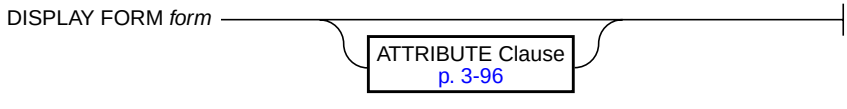
```
OPEN FORM f_customer FROM "f_customer"
DISPLAY FORM f_customer
...
DECLARE c_custs CURSOR FOR
   SELECT customer_num, company
   FROM customer
   WHERE state = "CA"
LET counter = 1
FOREACH c_custs INTO p_customers[counter].*
   LET counter = counter + 1
END FOREACH
...
CALL SET_COUNT(counter - 1)
DISPLAY ARRAY p_customers TO s_customers.*
```

References

ATTRIBUTE, DISPLAY, INPUT ARRAY, OPEN WINDOW, OPTIONS, SCROLL

DISPLAY FORM

The DISPLAY FORM statement displays a compiled 4GL screen form.



Element	Description
form	is the identifier of a 4GL screen form.

Usage

Before you can use a compiled form, you must take these steps:

1. Use OPEN FORM to declare the name of the form.
2. Use DISPLAY FORM to display the form on the screen.

The *form* name specifies which screen form to display. DISPLAY FORM is not required if you display a form by using the WITH FORM option of the OPEN WINDOW statement (see "The WITH FORM Clause" on page 4-283). An error occurs if the current 4GL window is too small to display the form.

Form Attributes

The DISPLAY FORM statement ignores the INVISIBLE attribute. 4GL applies any other display attributes that you specify in the ATTRIBUTE clause to any fields that have not been assigned attributes by the ATTRIBUTES section of the form specification file, or by the **syscolatt** table, or by the OPTIONS statement. If the form is displayed in a 4GL window, color attributes from the DISPLAY FORM statement supersede any from the OPEN WINDOW statement. If subsequent CONSTRUCT, DISPLAY, or DISPLAY ARRAY statements that include an ATTRIBUTE clause reference the form, however, their attributes take precedence over those specified in the DISPLAY FORM statement.

Reserved Lines

DISPLAY FORM displays the specified form in the current 4GL window, or in the 4GL screen itself, if no other 4GL window is open.

The form begins in the line that was indicated by the FORM LINE specification of the OPEN WINDOW or OPTIONS statement. This specification positions the first line of the form relative to the top of the current 4GL window. If you provided no FORM LINE specification, the default Form line is 3. On a default screen display, the reserved lines are positioned as follows.

Default Location	Reserved for
First line	Prompt line (output from PROMPT statement); also Menu line (*command value* from MENU statement)
Second line	Message line (output from MESSAGE statement; also the *description value* output from MENU statement)
Third line	Form line (output from DISPLAY FORM statement)
Second-to-last line	Comment line (output from COMMENT attribute) when SCREEN is the current 4GL window
Last line	Error line (output from ERROR statement); also Comment line in any 4GL window except SCREEN

For example, the following statements display the **cust_form** form in the 4GL screen (or in the current 4GL window):

```
OPEN FORM cust_form FROM "customer"
DISPLAY FORM cust_form
```

The OPTIONS statement can change the default position of all the reserved lines, including that of the Form line, for all 4GL windows, including the entire 4GL screen (specified as SCREEN). You can also reposition the Form line for a specific 4GL window only, by using an ATTRIBUTE clause in the OPEN WINDOW statement.

The following statements make line 6 the Form line for all 4GL windows, and then displays **cust_form**:

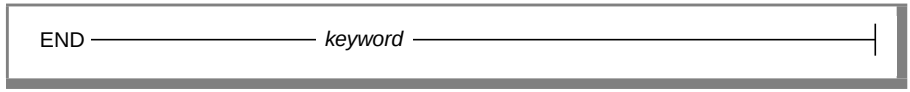
```
OPTIONS FORM LINE 6
OPEN FORM cust_for FROM "customer"
DISPLAY FORM cust_form
```

References

CLEAR, CLOSE FORM, OPEN FORM, OPEN WINDOW, OPTIONS

END

The END keyword marks the end of a compound 4GL statement.



Element	Description
keyword	is a keyword that specifies the name of the 4GL statement to be delimited, from among the keywords listed in this section.

Usage

The END keyword marks the last line of a compound 4GL statement. This is a compile-time indicator of the end of the statement construct. (Use the EXIT keyword, rather than END, to terminate execution of a compound statement.) The following compound statements of 4GL support END keywords to mark the end of the statement construct within the source module.

CASE	FOREACH	INPUT	PROMPT
CONSTRUCT	FUNCTION	INPUT ARRAY	REPORT
DISPLAY ARRAY	GLOBALS	MAIN	SQL
FOR	IF	MENU	WHILE

The END DISPLAY keywords delimit the DISPLAY ARRAY statement, and END INPUT delimits both INPUT and INPUT ARRAY. Unlike EXIT *statement* clauses, no more than one END *statement* clause can appear within the specified statement, but most compound statements of 4GL can be nested.

This statement fragment uses END MENU to delimit a MENU statement:

```
MENU "MAIN"
   ...
END MENU
```

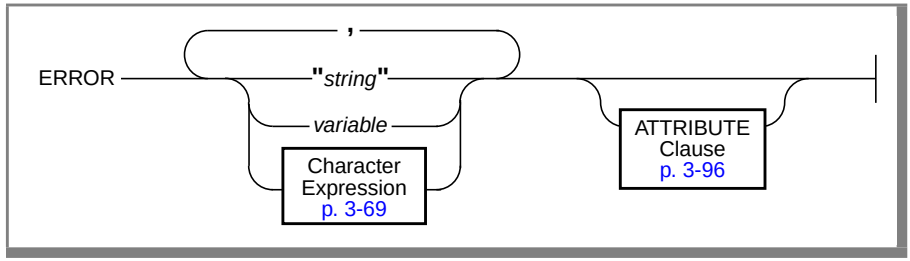
The END keyword can also delimit RECORD declarations (as described in "RECORD Variables" on page 4-88).

References

CASE, DISPLAY ARRAY, FOR, FOREACH, FUNCTION, GLOBALS, IF, INPUT, INPUT ARRAY, MAIN, MENU, PROMPT, REPORT, SQL, WHILE

ERROR

The ERROR statement displays an error message on the Error line and rings the terminal bell.



Element	Description
string	is a quoted string no longer than the number of characters that the Error line of the current 4GL window can display.
variable	is the name of a CHAR or VARCHAR variable whose contents are to be displayed on the Error line of the 4GL screen.

Usage

The *string* or *variable* value specifies all or part of the text of a screen message to be displayed on the Error line.

You can specify any combination of character variables and literal character strings for the message. 4GL generates the message to display by replacing any variables with their values, and concatenating the returned strings. The total length of this message must not be greater than the number of characters that the Error line can display in a single line of the 4GL screen. The message text remains on the screen until the user presses the next key.

The Error Line

The error message text appears in a borderless single-line 4GL window on the Error line. This 4GL window opens to display your text when ERROR is executed, and closes at the next keystroke by the user. When this 4GL window closes, any underlying display on the same line becomes visible again.

The position of the Error line is determined by the most recently executed ERROR LINE specification in the OPTIONS statement. Otherwise, the default Error line position is the last line of the screen. Because the Error line is positioned relative to the screen, rather than to the current window, you cannot use the OPEN WINDOW statement to reposition the Error line.

See "Reserved Lines" on page 4-114 for more information about the Error line and its relationship to the other reserved lines of 4GL.

You can use the CLIPPED and USING operators in the ERROR statement, as illustrated in the following examples:

```
ERROR p_orders.order_num USING "#####", " is not valid."

ERROR pattern CLIPPED, " has no match."
```

You can also use the ASCII and COLUMN operators, and other features of 4GL character expressions. (For more information on the built-in functions and operators of 4GL, see Chapter 5.)

The ATTRIBUTE Clause

The ATTRIBUTE clause syntax is described in "The ATTRIBUTE Clause" on page 4-41. The default display attribute for the Error line is REVERSE. You can use the ATTRIBUTE clause to specify some other attribute. 4GL ignores the INVISIBLE attribute if you include it with another attribute in the ATTRIBUTE clause of the ERROR statement. If the INVISIBLE attribute is the only attribute that you specify, 4GL displays the ERROR text as NORMAL.

In the following example, if the **insert_items( )** function returns FALSE, then 4GL rolls back the changes to the database and displays an error message:

```
IF NOT insert_items( ) THEN
   ROLLBACK WORK
   ERROR "Unable to insert items."
      ATTRIBUTE(RED, REVERSE, BLINK)
   RETURN
END IF
```

If the terminal supports color, then 4GL displays this error message in red, blinking, reverse video. If the terminal screen is monochrome, then 4GL displays the error message in bold, blinking, reverse video.

The next example specifies BLUE and BLINK attributes for the ERROR text:

```
ERROR "Unable to insert items" ATTRIBUTE(BLUE, BLINK)
```

System Error Messages

The Error line also displays *system error messages*. These can provide you with useful diagnostic information while you are developing 4GL programs, but they might not be helpful to users of your application.

One way to avoid displaying system error messages is to use the WHENEVER statement to trap runtime errors. The WHENEVER statement can call a function that executes an ERROR statement, displaying a screen message that is more suitable for your users.

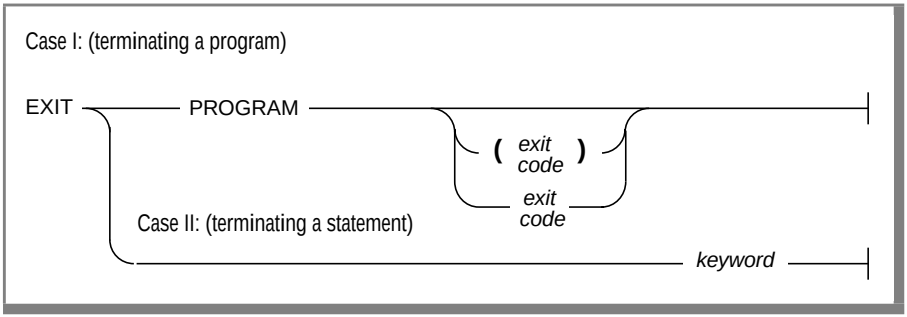
Some runtime errors cannot be trapped by the WHENEVER statement. "Exception Handling" on page 2-40 includes a list of error messages that are currently untrappable.

References

DISPLAY, MESSAGE, OPTIONS, PROMPT, WHENEVER

EXIT

The EXIT statement transfers control out of a control structure (a block, a loop, a CASE statement, an interface statement) or out of the program itself.



Element	Description
exit code	is an integer expression. For a description of this term, see "The Exit Code Value" on page 4-123.
keyword	is a keyword that specifies the current statement from which control of execution is to be transferred, from among those in the list that appears later in this section.

Usage

The EXIT PROGRAM statement terminates the program that is currently executing. Other forms of EXIT transfer control from the current control structure to whatever statement follows the corresponding END *keyword* keywords.

Leaving a Control Structure

Some compound statements support EXIT *statement* to terminate execution of the current statement and pass control of execution to the next statement.

EXIT CASE	EXIT FOR	EXIT MENU
EXIT CONSTRUCT	EXIT FOREACH	EXIT REPORT
EXIT DISPLAY	EXIT INPUT	EXIT WHILE

Here EXIT DISPLAY exits from DISPLAY ARRAY (but not DISPLAY) statements, and EXIT INPUT can exit from both INPUT ARRAY and INPUT statements.

Unlike EXIT PROGRAM, these other EXIT statements can only appear within the specified statement. For example, EXIT FOR can occur only in a FOR loop; if it is executed, it transfers control to the statement following the END FOR keywords that mark the end of that FOR statement.

Similarly, EXIT MENU can appear only within a control block of a MENU statement, where it transfers control to the first statement that follows the END MENU keywords of the same MENU statement.

Leaving a Function

The RETURN statement exits from a FUNCTION definition. There is no EXIT PROGRAM statement, because RETURN supports this functionality (and can also pass zero or more values from the FUNCTION program block to the calling statement).

You cannot use the GO TO or WHENEVER GO TO statements to transfer control of execution from the currently executing function. (These statements can only transfer control within the same program block.)

Leaving a Report

The EXIT REPORT statement exits from a REPORT definition. An error is issued if RETURN is encountered within a REPORT definition. Unlike a function, a report does not return anything to the calling routine, but a report normally sends formatted output to some specified destination. Within the report driver, you can terminate processing of a report by executing the FINISH REPORT or TERMINATE REPORT statements.

You cannot use the GO TO or WHENEVER GO TO statements to transfer control of execution from the currently executing report. (These statements can only transfer control within the same program block.)

Leaving the Program

The EXIT PROGRAM statement terminates execution of the 4GL program. After 4GL encounters the EXIT PROGRAM statement anywhere within the program, no subsequent statements are executed, and control returns to the operating system (or to whatever process invoked the 4GL program).

For example, here EXIT PROGRAM appears in a MENU statement:

```
MENU "MAIN"
   ...
   COMMAND "Quit" "Exit from the program"
      CLEAR SCREEN
      EXIT PROGRAM
END MENU
```

If 4GL encounters the END MAIN keywords in the MAIN block, END MAIN terminates the program, as if you had specified EXIT PROGRAM (0). If you are using the INFORMIX-4GL Interactive Debugger, a program that EXIT PROGRAM terminates can be examined subsequently by the WHERE or STACK commands of the Debugger, as if an abnormal termination had occurred.

The Exit Code Value

The *exit code* value returns the status code when a process terminates. The status code is a whole-number value, usually less than 256. The RETURNING clause of the RUN statement instructs 4GL to save the *exit code* value from the EXIT PROGRAM statement, if RUN invokes a 4GL program that EXIT PROGRAM terminates. When the 4GL program that RUN specifies completes execution, RUN can return an integer variable that contains two bytes of termination status information:

- The low byte contains the termination status of whatever RUN executes. You can recover this by calculating the value of (*integer value* modulo 256).
- The high byte contains the low byte from the EXIT PROGRAM statement of the 4GL program that RUN executes. You can recover this returned code by dividing *integer value* by 256.

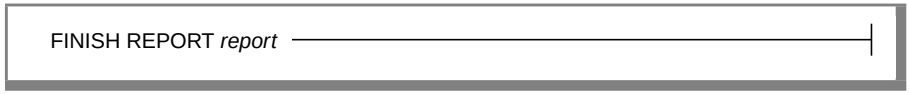
See "The RETURNING Clause" on page 4-341 for an example of using RUN and EXIT PROGRAM to examine termination status and exit codes from 4GL programs that RUN invoked and EXIT PROGRAM terminated.

References

CONTINUE, END, GOTO, LABEL, MAIN, RETURN, RUN

FINISH REPORT

The FINISH REPORT statement completes processing of a 4GL report.



Element	Description
report	is the name of a 4GL report, as declared in the REPORT statement.

Usage

This statement indicates the end of a *report driver* and complete processing of the report. (For more information, see "The Report Driver" on page 7-5.) FINISH REPORT must follow a START REPORT statement and at least one OUTPUT TO REPORT statement that reference the same report.

If the REPORT definition includes an ORDER BY section with no EXTERNAL keyword, or specifies aggregates based on all the input records, 4GL makes two passes through the input records. During the first pass, it uses the database server to sort the data, and then stores the sorted values in a temporary file. During the second pass, it calculates any aggregate values, and produces output from data in the temporary files. For more information, see "The EXTERNAL Keyword" on page 7-27 and "Aggregate Report Functions" on page 7-60.

The FINISH REPORT statement performs the following actions:

- Completes the second pass, if *report* is a two-pass report. These "second pass" activities handle the calculation and output of any aggregate values that are based on all the input records in the report, such as COUNT(*) or PERCENT(*) with no GROUP qualifier.
- Executes any AFTER GROUP OF control blocks (described in Chapter 7, "INFORMIX-4GL Reports").
- Executes any PAGE HEADER, ON LAST ROW, and PAGE TRAILER control blocks to complete the report, as described in Chapter 7.

- Copies data from the output buffers of the report to the destination in START REPORT or in the OUTPUT section of the report definition. If no destination is specified, output goes to the Report window (as described in "Sending Report Output to the Screen" on page 7-19).
- Closes the Select cursor on any temporary table that was created to order the input records or to perform aggregate calculations.
- Deallocates memory for local BYTE or TEXT variables of the report.
- Terminates processing of the 4GL report, and deletes from the database any files that held temporary tables for a two-pass report.

The following program creates a report based on data in the **orders** table:

```
DATABASE stores7
MAIN
   DEFINE p_orders RECORD LIKE orders.*
   DECLARE q_ordcurs CURSOR FOR SELECT * FROM orders
   START REPORT ord_list TO "ord_listing"
   FOREACH q_ordcurs INTO p_orders
      OUTPUT TO REPORT ord_list(p_orders)
   END FOREACH
   FINISH REPORT ord_list
END MAIN
REPORT ord_list(r_orders)
   DEFINE r_orders RECORD LIKE orders.*
   FORMAT EVERY ROW
END REPORT
```

The temporary tables that 4GL reports use for sorting input records or for calculating aggregates in two-pass reports are stored in the current database. If you do not open any database, or if the CLOSE DATABASE statement closes the current database, then a runtime error occurs when 4GL cannot create or access the temporary tables that are required for a two-pass report.

Similarly, the FINISH REPORT statement cannot access temporary tables in more than one database. An error can occur if the DATABASE statement opens a different database while a two-pass 4GL report is being processed. The following program fragment, for example, produces a runtime error if the **produce** report requires two passes:

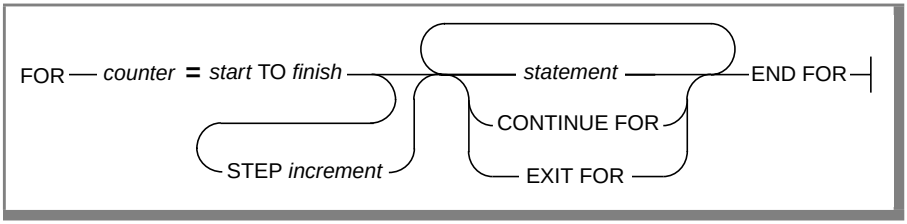
```
DATABASE apples
 . . .
START REPORT produce          --database is apples
 . . .
OUTPUT TO REPORT produce(input_rex)
 . . .
DATABASE oranges           --new database is oranges
FINISH REPORT produce --cannot access files in apples database
```

References

OUTPUT TO REPORT, REPORT, START REPORT, TERMINATE REPORT

FOR

The FOR statement executes a statement block a specified number of times.



Element	Description
counter	is a variable of type INTEGER or SMALLINT that serves as an index for the *statement* block.
finish	is an integer expression to specify an upper limit for *counter*.
increment	is an integer expression whose value is added to *counter* after each iteration of the *statement* block.
start	is an integer expression to set an initial *counter* value.
statement	is an SQL statement or other 4GL statement. (This statement block is sometimes called the FOR loop.)

Usage

The FOR statement executes the statements up to the END FOR statement a specified number of times, or until EXIT FOR terminates the FOR statement. (Use the WHILE statement, rather than FOR, if you cannot specify an upper limit on how many times the program needs to repeat a statement block, but you can specify a Boolean condition for leaving the block.)

The TO Clause

4GL maintains an internal *counter*, whose value changes on each pass through the statement block. On the first iteration through the loop, this counter is set to the initial expression at the left of the TO keyword. Thereafter, the value of the *increment* expression in the STEP clause specification (or by default, 1) is added to *counter* in each pass through the block of statements.

When the sign of the difference between the values of *counter* and the *finish* expression at the right of the TO keyword changes, 4GL exits from the FOR loop. Execution resumes at the statement following the END FOR keywords. For example, this statement clears four records of the **s_items** screen array:

```
FOR counter = 1 TO 4
   CLEAR s_items[counter].*
END FOR
```

The FOR loop terminates after the iteration for which the left- and right-hand expressions are equal. If either returns NULL, the loop cannot terminate, because here the Boolean expression `"left = right"` cannot become TRUE.

The STEP Clause

Use the STEP clause to tell 4GL the number by which to increment the counter. For example, this FOR statement increments the counter by 2:

```
FOR idx = 1 TO 12 STEP 2
   DISPLAY month_names[idx] TO sc_month[i]
   LET i = i + 1
END FOR
```

If you use a negative STEP value, specify the second expression in the TO clause as smaller than the first value in the range.

Before processing the block of statements, 4GL first tests the counter value against the terminating value. For example, if the STEP value is positive and the counter value is greater than the last value in the range, 4GL skips over the statements in the loop without executing them.

The CONTINUE FOR Statement

Use the CONTINUE FOR statement to interrupt the current iteration and start the next iteration of the statement block. To execute a CONTINUE FOR statement, 4GL does the following:

- Skips the remaining statements between the CONTINUE FOR and END FOR keywords
- Increments the counter variable and tests it
- If the counter does not exceed the final value, goes back to the beginning of the loop and performs another iteration; otherwise, continues execution after the END FOR keywords

The EXIT FOR Statement

Use the EXIT FOR statement to terminate the statement block. When 4GL encounters this statement, it skips any statements between the EXIT FOR and END FOR keywords. Execution resumes at the first statement immediately after the END FOR keywords.

The END FOR Keywords

Use END FOR to indicate the end of the FOR loop. Upon encountering the END FOR keywords, 4GL increments the counter and compares it with the expression that immediately follows the TO keyword. If the counter exceeds this value, then 4GL terminates the FOR loop and executes the statement following the END FOR keywords.

Databases with Transactions

If your database has transaction logging, and the FOR loop includes one or more SQL statements that modify the database, then it is advisable that the entire FOR loop be within a transaction. Otherwise, if an error occurs after some of the SQL statements within the FOR loop have executed, but before the loop has terminated, the user might face two potential problems:

- It might be difficult to determine the extent to which the integrity of the database has been compromised.
- If the database has been corrupted, it might be difficult to restore it to its condition prior to the execution of the FOR loop.

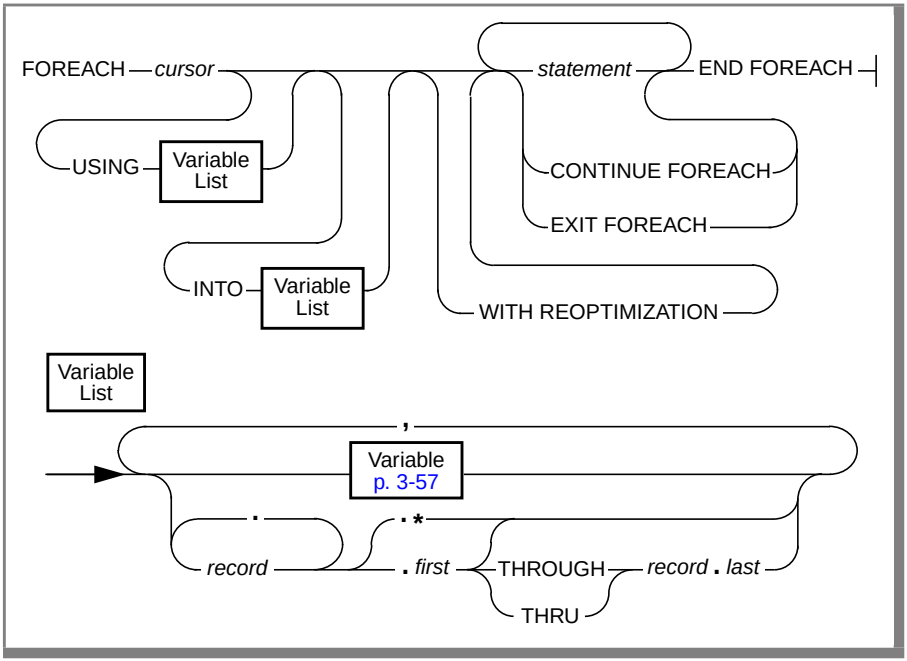
The same data integrity considerations also apply to FOREACH and WHILE loops that include SQL statements in 4GL programs. (See the *Informix Guide to SQL: Tutorial* for more information about the SQL concepts and statements that support data integrity through transactions.)

References

CONTINUE, FOREACH, WHILE

FOREACH

The FOREACH statement applies a series of actions to each row of data that is returned from a query by a cursor.



Element	Description
cursor	is the name of a previously declared SQL cursor.
first	is the name of a member variable in which to store a value.
last	is a member of *record* that was declared later than first.
record	is the name of a variable of the RECORD data type.
statement	is an SQL statement or other 4GL statement.

Usage

Use the FOREACH statement to retrieve and process database rows that were selected by a query. The FOREACH statement is equivalent to using the OPEN, FETCH, and CLOSE statements.

The FOREACH statement has these effects:

1. Opens the specified cursor
2. Fetches the rows selected
3. Closes the cursor (after the last row has been fetched)

You must declare the cursor (by using the DECLARE statement) before the FOREACH statement can retrieve the rows. A compile-time error occurs unless the cursor was declared prior to this point in the source module. You can reference a sequential cursor, a scroll cursor, a hold cursor, or an update cursor, but FOREACH only processes rows in sequential order.

The FOREACH statement performs successive fetches until all rows specified by the SELECT statement are retrieved. Then the cursor is automatically closed. It is also closed if a WHENEVER NOT FOUND statement within the FOREACH loop detects a NOTFOUND condition (that is, `status = 100`).

Implicit FETCH statements that FOREACH executes with a FOR UPDATE cursor can support promotable locks. (See the *Informix Guide to SQL: Syntax*.)

The following topics are described in this section:

Cursor Names

You must follow the FOREACH keyword with a *cursor name* that a DECLARE statement declared earlier in the same module. A runtime error can occur if the FOREACH statement does not specify a previously declared cursor.

The next example fetches values retrieved by the **c_orders** cursor. For each retrieved row, 4GL increments the **counter** variable by 1, invokes a function called **scan( )**, and passes the values of **ord_num**, **cust_num**, and **comp**. If the query does not return any rows, 4GL ignores the FOREACH loop and resumes processing with the statement that immediately follows the END FOREACH keywords. This IF statement examines the **counter** variable, and displays a message on the Error line if the query returns no rows.

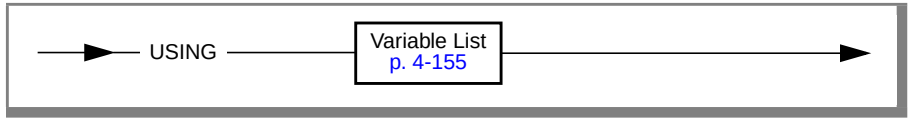
```
PROMPT "Enter cut-off date for orders: " FOR o_date
DECLARE c_orders CURSOR FOR
   SELECT order_num, orders.customer.num, company
   INTO ord_num, cust_num, comp FROM orders o, customer c
   WHERE o.customer_num = c.customer_num
      AND order_date < o_date
LET counter = 1
FOREACH c_orders
   LET counter = counter + 1
   CALL scan(ord_num, cust_num, comp)
END FOREACH
IF counter = 0 THEN
   ERROR "No orders before ", o_date
END IF
```

FOREACH internally generates an OPEN statement, a FETCH loop (which normally exits when NOTFOUND is returned), and a CLOSE statement. If a FETCH returns an error other than NOTFOUND (error `100`, the normal end-of-data indication) and WHENEVER ERROR GOTO is in effect, the implicit CLOSE statement is not executed. (Also, if you use WHENEVER ERROR CALL and the called function terminates the program, the cursor is not closed before entering the called function.) If you use WHENEVER ERROR GOTO to resume execution elsewhere in the program, the cursor might remain open and would need to be explicitly closed.

Because the internally generated CLOSE statement can change the values in the **SQLCA** structure, the value of **SQLCA.SQLERRD[3]** after the END FOREACH keywords are encountered does not represent the number of rows fetched by the FOREACH. If you need to know the number of rows fetched, you must maintain your own row counter.

The USING Clause

The USING clause specifies a variable (or a comma-separated list of variables) to provide values to be used as search criteria by the query.



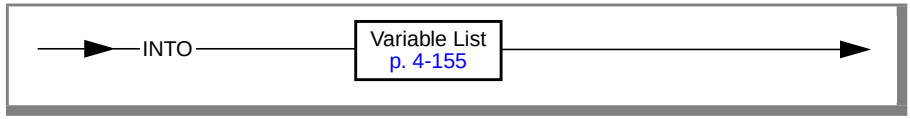
The USING clause is required only if the cursor expects user-supplied values to replace question (?) mark placeholders.

Just as with the OPEN statement of SQL, the number and data types of placeholders in the prepared SELECT statement must correspond exactly to the number and data types of the variables in the USING clause.

If both the USING and INTO clauses are used, the USING clause must precede the INTO clause. (In embedded EXECUTE statements of SQL, however, 4GL supports both the EXECUTE…INTO…USING and EXECUTE…USING…INTO sequences of clauses, as a convenience to the programmer.)

The INTO Clause

The INTO clause specifies a variable (or a comma-separated list of variables) in which to store values from each row that is returned by the query.



The number and order of variables in the INTO clause must match the number and order of the columns in the active set of rows that are retrieved by the cursor, and must be of compatible data types.

For example, the following FOREACH statement stores the retrieved rows in the **p_items** program array:

```
LET counter = 1
FOREACH my_curs INTO p_items[counter].*
   LET counter = counter + 1
   IF counter > 10 THEN
      CALL mess ("Ten or more items.")
      EXIT FOREACH
   END IF
END FOREACH
```

You can include the INTO clause in the SELECT statement associated with the cursor, or in the FOREACH statement, but not in both. To retrieve rows into a program array, however, you must place the INTO clause in the FOREACH statement, rather than in the SELECT-*statement* of a DECLARE statement.

The WITH REOPTIMIZATION Keywords

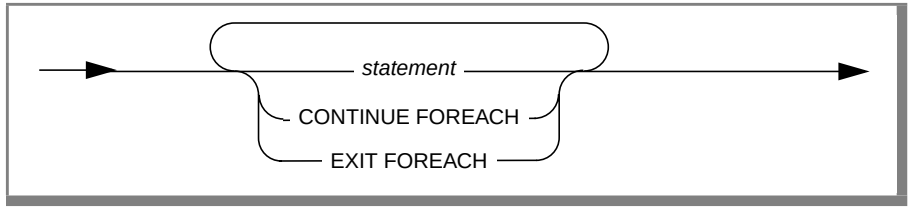
The WITH REOPTIMIZATION keywords enable you to reoptimize your query-design plan. When you prepare a SELECT or EXECUTE PROCEDURE statement, the Informix database server uses a query-design plan to optimize the performance of that query.

If you subsequently modify the data values that are associated with the SELECT or EXECUTE PROCEDURE statement, the plan might no longer be efficient. To avoid this, you can prepare the SELECT or EXECUTE PROCEDURE statement again, or you can use FOREACH or OPEN with the WITH REOPTIMIZATION keywords, so that a new query design plan can take into account the modified data values.

Informix recommends that you specify WITH REOPTIMIZATION, rather than reprepare the statement, because WITH REOPTIMIZATION rebuilds only the query-design plan, rather than the entire statement. This process takes less time and requires fewer resources than preparing the statement again.

The FOREACH Statement Block

These statements are executed after each row of the active set is fetched.



Element	Description
statement	is an SQL statement or other 4GL statement.

This block is sometimes called the FOREACH loop. If the cursor returns no rows, then no statements in this loop are executed, and program control passes to the first statement that follows the END FOREACH keywords. If the specified cursor is FOR UPDATE, the statement block can include statements to modify retrieved rows. See the *Informix Guide to SQL: Syntax*.

Databases with Transactions

If your database has *transaction logging*, then it is advisable to put the entire FOREACH statement block in a transaction. Otherwise, if an error occurs after some of the SQL statements within the FOREACH statement block have executed, but before the loop has terminated, the user might face two potential problems:

- It might be difficult to determine the extent to which the integrity of the database has been compromised.
- If the database has been corrupted, it might be difficult to restore it to its condition prior to the execution of the FOREACH loop.

These considerations apply to FOR and WHILE loops that can change the database. (See *Informix Guide to SQL: Tutorial* for information about the SQL concepts and statements that support data integrity through transactions.)

If your database has transactions and the cursor was declared by DECLARE FOR UPDATE but not DECLARE WITH HOLD, the FOREACH statement must be executed within a transaction. (You can open an update cursor that was declared with a DECLARE WITH HOLD via a FOREACH statement outside a transaction, but you cannot roll back any changes that the cursor performs outside the transaction. In this situation, each UPDATE WHERE CURRENT OF is automatically committed as a singleton transaction.)

The CONTINUE FOREACH Keywords

CONTINUE FOREACH interrupts processing of the current row and starts processing the next row. 4GL fetches the next row and resumes processing at the first statement in the FOREACH statement block. For example, if **total_price** is less than 1000 in the next example, 4GL increments **smallOrders**, fetches the next row, and executes the IF statement. If **total_price** is equal to or greater than 1000, 4GL proceeds to the next statement in the FOREACH block, in this case, the OUTPUT TO REPORT statement:

```
LET smallOrders = 1
FOREACH orderC
   IF orderP.total_price < 1000 THEN
      LET smallOrders = smallOrders + 1
      CONTINUE FOREACH
   END IF
   OUTPUT TO REPORT order_list (orderR.*, smallOrders)
   ...
END FOREACH
```

The EXIT FOREACH Statement

Use the EXIT FOREACH statement to interrupt processing and ignore the remaining rows of the active set. Upon encountering EXIT FOREACH, 4GL skips the statements between the EXIT FOREACH and the END FOREACH keywords. Execution resumes at the statement that follows the END FOREACH keywords.

The next section provides a code example in which a message is displayed on the screen and EXIT FOREACH is executed when a report driver detects an error condition within a FOREACH statement block.

The END FOREACH Keywords

Use the END FOREACH keywords to indicate the end of the FOREACH loop. When 4GL encounters the END FOREACH keywords, it re-executes the loop until no more rows returned by the query remain. Otherwise, it executes the statement that follows the END FOREACH keywords.

For example, if the **status** variable is not equal to 0 in the following program fragment, 4GL displays a message and exits from the FOREACH loop:

```
DECLARE orderC CURSOR FOR
   SELECT * INTO orderR.* FROM orders
   WHERE order_date BETWEEN start_date AND end_date
START REPORT order_list
LET smallOrders = 0
FOREACH orderC
   IF orderR.total_price < 1000 THEN
      LET smallOrders = smallOrders + 1
      CONTINUE FOREACH
   END IF
   OUTPUT TO REPORT order_list (orderR.*, smallOrders)
   IF status != 0 THEN
      MESSAGE "Error on output to report."
      EXIT FOREACH
   END IF
END FOREACH
FINISH REPORT order_list
```

The next example creates a cursor **c_query**, based on search criteria entered by the user.

For each row retrieved by the SELECT statement of the cursor, this example displays the row on the screen and waits for the user to request the next row. If no rows are selected, then 4GL displays a message.

```
DEFINE stmt1, query1 CHAR(300),
      p_customer RECORD LIKE customer.*
CONSTRUCT BY NAME query1 ON customer.*
LET stmt1 = "SELECT * FROM customer ",
   "WHERE ", query1 CLIPPED
PREPARE stmt_1 FROM stmt1
DECLARE c_query CURSOR FOR stmt_1
LET exist = 0
FOREACH c_query INTO p_customer.*
   LET exist = 1
   DISPLAY BY NAME p_customer.*
   PROMPT "Do you want to see the next customer (y/n): "
      FOR answer
   IF answer MATCHES "[Nn]" THEN
```

```
      EXIT FOREACH
   END IF
END FOREACH
IF exist = 0 THEN
   MESSAGE "No rows found."
END IF
```

If a query returns no rows, then none of the statements in the FOREACH block is executed, and program control passes immediately to the first statement following END FOREACH. If you need to know whether any rows were returned, you can set up a flag or a counter as in the example that follows:

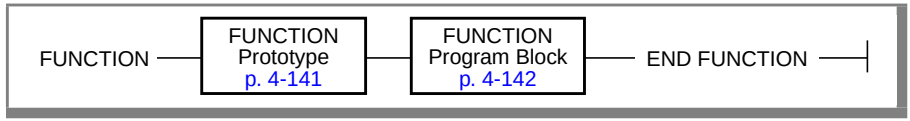
```
PROMPT "Enter cut-off date for query: "FOR o_date
DECLARE q_curs CURSOR FOR
   SELECT order_num, o.customer_num, company
      FROM orders o, customer c
      WHERE o.customer_num = c.customer_num
         AND order_date < o_date
LET counter = 0
FOREACH q_curs INTO ord_num, cust_num, comp
   LET counter = counter + 1
   CALL scan (ord_num, cust_num, comp)
END FOREACH
IF counter = 0 THEN
   ERROR "No orders before ", o_date
END IF
```

References

CONTINUE, FETCH, FOR, OPEN, WHILE, WHENEVER

FUNCTION

The FUNCTION statement defines a FUNCTION program block.



Usage

As Chapter 5 explains, a 4GL *function* is a named block of statements. The FUNCTION statement defines a 4GL function that can be invoked from any module of your program. The FUNCTION statement has two effects:

- It *declares* the name of a function and any formal arguments. 4GL imposes no limit on the number or size of formal arguments.
- It *defines* the corresponding FUNCTION program block.

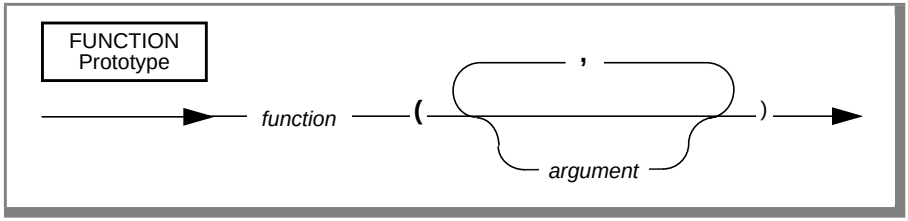
The FUNCTION statement cannot appear within the MAIN statement, in a REPORT statement, nor within another FUNCTION statement. If the function returns a single value, it can be invoked as an operand within a 4GL expression (as described in "Function Calls as Operands" on page 3-58). Otherwise, you must invoke it with the CALL statement.

An error results if the list of returned values in the RETURN statement conflicts in number or in data type with the RETURNING clause of the CALL statement that invokes the function (as described in "The RETURNING Clause" on page 4-19).

The following topics are described in this section:

The Prototype of the Function

The FUNCTION statement both declares and defines a 4GL function. The function declaration specifies the identifier of the function and the identifiers of its formal arguments (if any). These specifications are sometimes called the *function prototype*, as distinct from the *function definition*.



Element	Description
argument	is the name of a formal argument to this function. This can be of any data type except ARRAY. (See "Declaring the Names and Data Types of Variables" on page 4-84 and "ARRAY Variables" on page 4-87.)
function	is the identifier that you declare for this 4GL function.

The Identifier of the Function

The function name must follow the rules for 4GL identifiers (as described in "4GL Identifiers" on page 2-14) and must be unique among all the names of functions or reports in the same program. If the name is also the name of a built-in 4GL function, an error occurs at link time, even if the program does not reference the built-in function. Like all 4GL identifiers, the name is not case sensitive. For example, the function names **unIonized( )** and **Unionized( )** are identical to 4GL.

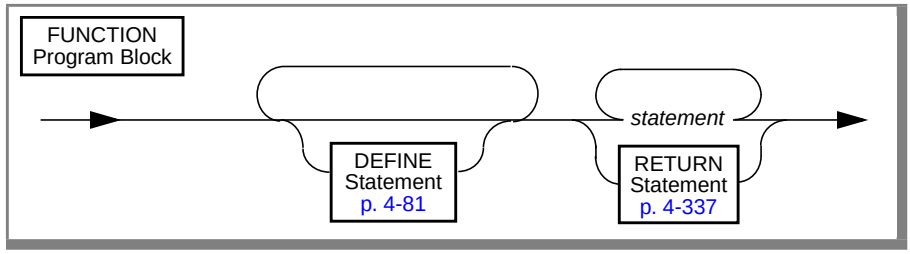
The Argument List of the Function

The names specified (between parentheses) in the argument list define the formal arguments, if any, as they will be received when the FUNCTION program block is executed. Argument names must be unique within the argument list of the current FUNCTION declaration. Their scope of reference is local to the function; that is, they are not visible in other program blocks.

Important: *If no argument is specified, an empty argument list must still be supplied, enclosed between the parentheses.*

The FUNCTION Program Block

The statements between the argument list and the END FUNCTION keywords make up the FUNCTION program block. These statements are executed whenever the function is successfully invoked.



Element	Description
statement	is any SQL statement or other 4GL statement (but not GLOBALS, DEFINE, DEFER, MAIN, FUNCTION, REPORT, EXIT REPORT, NEED, PAUSE, PRINT, nor SKIP).

You can define a function whose statement block is empty. This enables you to test other parts of a program before a function definition is written.

Executable Statements

Any executable statements in the statement block are executed when the function is called. Here is a simple example of a function definition:

```
FUNCTION state_abbrev(state)
   DEFINE st LIKE state.code,
         state LIKE state.sname
   SELECT state.code INTO st FROM state
            WHERE state.sname MATCHES state
   RETURN st
END FUNCTION
```

In this example, the function definition contains two executable statements:

- DEFINE is a declarative statement that allocates storage in memory for the local variables **st** and **state**.
- SELECT is an executable SQL statement.
- RETURN returns control (and the value of **st**) to the calling routine.
- END FUNCTION marks the end of the program block.

Here DEFINE and END FUNCTION are not executable statements, but they are needed to declare the formal argument and another local variable, and to delimit the function definition.

Data Type Declarations

The data type of each formal argument of the function must be specified by a DEFINE statement that immediately follows the argument list. Any DEFINE declarations within a function definition must occur before any other statements within the FUNCTION program block. Just as in a MAIN or REPORT program block, a compile-time error occurs if any executable statement precedes a DEFINE declaration in the FUNCTION definition.

The actual argument in a call to the function need not be of the declared data type of the formal argument. If both are of compatible data types, 4GL converts the actual argument to the data type that the function requires. If data type conversion is impossible, a runtime error occurs. For a discussion of compatible data types, see "Data Type Conversion" on page 3-42.

Here is an example of a call for which data-type conversion is necessary. The actual argument, the character string "105", must be converted to INTEGER.

```
DEFINE getStat INTEGER
LET getStat = getCustRec("105")
   . . .
FUNCTION getCustRec(cno)
   DEFINE cno, dno INTEGER
   . . .
   RETURN dno
END FUNCTION
```

The Function as a Local Scope of Reference

The same or a subsequent DEFINE statement must also declare any other local variable that is referenced in the same FUNCTION definition. Two local variables are declared in the previous example, the function argument **cno**, and the variable named **dno**. The identifiers of local variables must be unique among the variables that are declared in the same FUNCTION definition. They are not visible in other program blocks.

Just as within MAIN or REPORT program blocks, statements in the function can reference previously declared module or global variables.

Any global or module variable that has the same identifier as a local variable, however, is not visible within the scope of the local variable.

For information about using the LIKE keyword during compilation to declare the data types of local variables indirectly, see the description of the DATABASE statement (in "The Default Database at Compile Time" on page 4-73) .

You can also use DATABASE within a FUNCTION definition to specify a new current database at runtime (as described in "The Current Database at Runtime" on page 4-74).

Any GOTO or WHENEVER...GOTO statement in a function must reference a statement label (described in "LABEL" on page 4-224) within the same FUNCTION program block.

Returning Values to the Calling Routine

Any programmer-defined 4GL function that returns one or more values to the calling routine must include the RETURN statement. Values specified in RETURN must correspond in number and position, and must be of the same or of compatible data types, to the variables in the RETURNING clause of the CALL statement. (For more information, see "Summary of Compatible 4GL Data Types" on page 3-46 and "The RETURNING Clause" on page 4-19.)

Unless it has the same name as a built-in operator (see Chapter 5), any built-in or programmer-defined function that returns a single value of a simple data type can appear in 4GL expressions (with its arguments, if any) if the returned value is of a range and data type that is valid in the expression:

```
DISPLAY AT 2,2 ERR_GET(SQLCA.SQLCODE)
```

The END FUNCTION Keywords

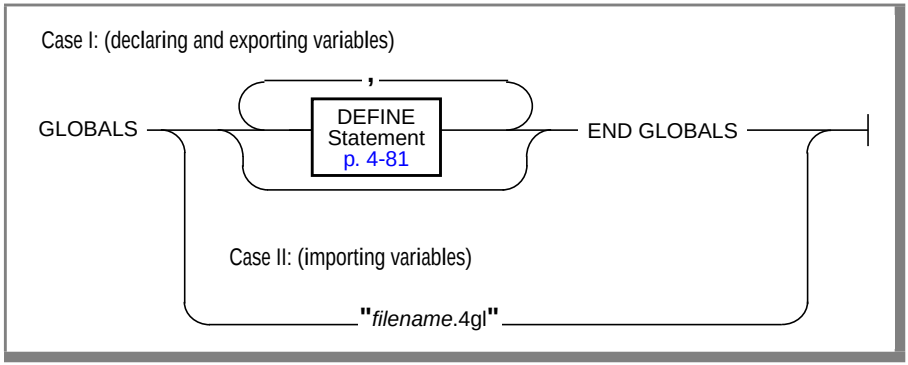
The END FUNCTION keywords mark the end of the FUNCTION program block. Only another FUNCTION definition or the REPORT statement can follow the END FUNCTION keywords in the same source code module.

References

CALL, DEFINE, RETURN, WHENEVER

GLOBALS

The GLOBALS statement declares modular variables that can be exported to other program modules. It can also import variables from other modules.



Element	Description
filename	is a quoted string that specifies the name of a file that contains the GLOBALS…END GLOBALS statement (and optionally the DATABASE statement) but no executable statements. The *filename* can include a pathname. The **.4gl** file extension is required.

Usage

In general, a program variable is in scope only in the same FUNCTION, MAIN, or REPORT program block in which it was declared. To make its scope of reference the entire source module, you must specify a modular declaration, by locating the DEFINE statement outside of any program block.

To extend the visibility of one or more module variables beyond the source module in which they are declared, you must take the following steps:

- Declare variables in GLOBALS…END GLOBALS declarations (in files containing only GLOBALS, DEFINE, and DATABASE statements).
- Specify the files in GLOBALS *"filename"* statements in each additional source module that includes statements referencing the variables.

These files must also be compiled and linked with the 4GL application. (Earlier 4GL releases permitted no more than one GLOBALS…END GLOBALS statement, but the number of globals files is now unrestricted.)

Declaring and Exporting Global Variables

To declare global variables, the GLOBALS statement must appear before the first MAIN, FUNCTION, or REPORT program block, so that variables that you declare in the GLOBALS statement are modular in their scope of reference. You can include one or more DEFINE statements after the GLOBALS keyword. The END GLOBALS keywords must follow the last DEFINE declaration.

If you use the LIKE keyword in the DEFINE declaration, a DATABASE statement must precede the GLOBALS statement within the same module.

The following program fragment declares a global record, a global array, and a simple global variable that are referenced by built-in and programmer-defined functions within the same source code module:

```
DATABASE stores7
GLOBALS
      DEFINE p_customer RECORD LIKE customer.*,
         p_state ARRAY[50] OF RECORD LIKE state.*,
         fifty, state_cnt  SMALLINT
END GLOBALS

MAIN
   ...
END MAIN

FUNCTION get_states()
   ...
   FOREACH c_state INTO p_state[state_cnt].*
      LET state_cnt = state_cnt + 1
      IF state_cnt > fifty THEN
         EXIT FOREACH
      END IF
   END FOREACH
   ...
END FUNCTION
FUNCTION statehelp()
   DEFINE idx SMALLINT
   ...
   CALL SET _COUNT(state_cnt)
   DISPLAY ARRAY p_state TO s_state.*
```

```
    LET idx = ARR_CURR()
    CLOSE WINDOW w_state
    LET p_customer.state = p_state[idx].code
    DISPLAY BY NAME p_customer.state
    RETURN
END FUNCTION
```

GLOBALS *"filename"* statements cannot reference this file because it includes executable statements (besides GLOBALS, DEFINE, and DATABASE).

A compile-time error would occur if you declared a 4GL variable of *modular* scope called **fifty**, **p_customer**, **p_state**, or **state_cnt** in the same module as this GLOBALS statement. If you want, however, you can declare *local* variables whose names match those of variables from GLOBALS declarations.

Although you can include multiple GLOBALS ... END GLOBALS statements in the same 4GL application, do not declare the same identifier as the name of a variable within the DEFINE statements of more than one GLOBALS declaration. Even if several declarations of a global variable defined in multiple places are identical, declaring any global variable more than once can result in compilation errors, or in unpredictable runtime behavior.

The GLOBALS *"filename"* statement must occur earlier in every file than any function that makes reference to a global variable. Within its source code file, the GLOBALS statement must be outside the MAIN program block (and also outside any FUNCTION or REPORT definition).

Importing Global Variables

A *globals file* is a source module that contains a GLOBALS ... END GLOBALS statement. This can also contain a DATABASE statement (as described in "The Default Database at Compile Time" on page 4-73), but no executable statements. The scope of reference of variables declared in that file can be extended to all the program blocks of any 4GL program module that includes a GLOBALS *"filename"* statement.

To import global variables into other modules

1. Create a globals file called *filename*.**4gl** that includes the following items:

   - If necessary, a DATABASE statement

     This is required only if you use the LIKE keyword in the DEFINE declaration. If present, the DATABASE statement must precede the GLOBALS statement. For the syntax of LIKE in declarations of variables, see "Indirect Typing" on page 4-83.

   - The GLOBALS keyword, followed by as many DEFINE statements as necessary to declare your global variables

     You cannot include any DEFINE statements if the GLOBALS "*filename*" statement is used only to apply a DATABASE statement to several modules.

   - The END GLOBALS keywords

2. In any other module of the program that includes statements referencing the global variables, include a GLOBALS "*filename*" statement before the first MAIN, FUNCTION, or REPORT program block.

   To import global variables, you specify the *filename* of the globals file, but do not include the END GLOBALS keywords.

These two steps correspond, respectively, to Case I and Case II in the syntax diagram at the beginning of this section. For example, the globals file **d4_glob.4gl** in the **stores7** demonstration application includes the following DATABASE and GLOBALS statements:

```
DATABASE stores7
GLOBALS
   DEFINE
      p_customer RECORD LIKE customer.*,
      p_orders RECORD
         order_num LIKE orders.order_num,
         order_date LIKE orders.order_date,
         po_num LIKE orders.po_num,
         ship_instruct LIKE orders.ship_instruct
      END RECORD,
      p_items ARRAY[10] OF RECORD
         item_num LIKE items.item_num,
         stock_num LIKE items.stock_num,
         manu_code LIKE items.manu_code,
         description LIKE stock.description,
         quantity LIKE items.quantity,
         unit_price LIKE stock.unit_price,
         total_price LIKE items.total_price
```

```
        END RECORD,
        p_stock ARRAY[30] OF RECORD
           stock_num LIKE stock.stock_num,
           manu_code LIKE manufact.manu_code,
           manu_name LIKE manufact.manu_name,
           description LIKE stock.description,
           unit_price LIKE stock.unit_price,
           unit_descr LIKE stock.unit_descr
        END RECORD,
        p_state ARRAY[fifty] OF RECORD LIKE state.*,
        fifty, state_cnt, stock_cnt INTEGER,
        print_option CHAR(1)
END GLOBALS
```

The next program fragment include a GLOBALS statement that specifies **d4_glob.4gl** as the globals file that declares global variables:

```
GLOBALS "d4_glob.4gl"
MAIN
   DEFER INTERRUPT
   ...
   CALL get_states()
   CALL get_stocks()
   ...
END MAIN
```

Here the database specified by the DATABASE statement in the globals file is both the default database at compile time and the current database at runtime, because the `GLOBALS "d4_glob.4gl"` statement includes the DATABASE statement before the MAIN program block. (For more information, see "The Default Database at Compile Time" on page 4-73 and "The Current Database at Runtime" on page 4-74.)

If a *local* variable has the same name as another variable that you declare in the GLOBALS statement, only the local variable is visible within its scope of reference. Similarly, a *modular* variable takes precedence in the module where it is declared over any variable of the same name whose declaration is in the filename referenced by a GLOBALS statement. (A compile-time error occurs if you declare another module variable with the same identifier as another variable that the GLOBALS…END GLOBALS statement declares in the same module.) For more information about the scope and visibility of 4GL identifiers, see "4GL Identifiers" on page 2-14.

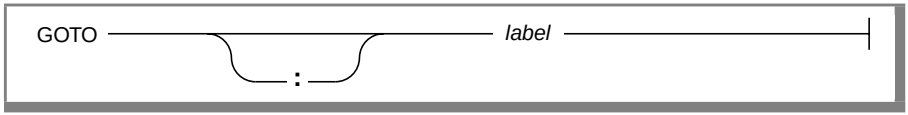
4GL does not check for the name conflicts between global variables and system function calls. To avoid errors at runtime, do not use system function names such as **read( )**, **open( )**, or **stat( )**, as identifiers of global variables.

References

DATABASE, DEFINE, FUNCTION, INCLUDE, MAIN, REPORT

GOTO

The GOTO statement transfers program control to a labeled line within the same program block.



Element	Description
label	is a *statement label* that you declare in a LABEL statement.

Usage

The GOTO statement transfers control of execution within a program block. Upon encountering this statement, 4GL jumps to the statement immediately following the specified LABEL statement, and resumes execution there, skipping any intervening statements that lexically follow the GOTO statement. These rules apply to the use of the GOTO and LABEL statements:

- To transfer control to a labeled line, the GOTO statement must use the same label name as the LABEL statement above the desired line.
- Both statements must be in the same MAIN, FUNCTION, or REPORT block. GOTO cannot transfer control into or out of a program block.

Excessive use of GOTO statements in 4GL (or any programming language) can make your code difficult to read or to maintain, or can result in a loop that has no termination. Many situations in which you need to transfer control of program execution can be solved by using one of the following alternatives to the GOTO statement:

- Boolean expressions and the CASE, FOR, IF, and WHILE statements
- The CALL, OUTPUT TO REPORT, or WHENEVER statement
- The EXIT keyword in blocks within the following statements:

CASE	FOR	INPUT ARRAY
CONSTRUCT	FOREACH	MENU
DISPLAY ARRAY	INPUT	WHILE

- The CONTINUE keyword in blocks within the following statements:

CONSTRUCT	FOR FOREACH	INPUT INPUT ARRAY	MENU WHILE

It is convenient to use the GOTO and LABEL statements in some situations; for example, to exit from deeply nested code:

```
FOR i = 1 TO 10
   FOR j = 1 TO 20
      FOR k = 1 To 30
         ...
         IF pa_array3d[i,j,k] IS NULL THEN
            GOTO :done
         ELSE
            ...
         END IF
         ...
      END FOR
   END FOR
END FOR

LABEL done:
ERROR "Cannot complete processing."
ROLLBACK WORK
```

More important than avoiding the GOTO statement, however, is to adhere to the design principle that any block of statements (such as a function or a loop) have only *one entry point* and *one exit point*, as in this program fragment:

```
CALL do_things(value)       --invokes a FUNCTION block
...
FUNCTION do_things(arglist)--unique entry point
      ...
   IF (exit_condition) THEN
      GOTO :outofhere   --jump within same program block
   END IF
   ...
   LABEL outofhere:
   CALL clean_up()
   RETURN ret_code      --unique exit point
END FUNCTION            --marks end of FUNCTION construct
```

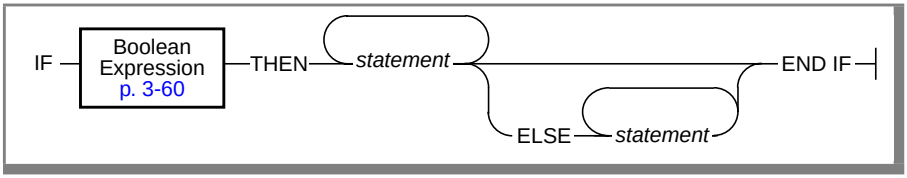
You can optionally place a colon before *label name* in the GOTO statement. This conforms to the ANSI/ISO standard for embedded SQL syntax.

References

CASE, FOR, IF, FOR, FUNCTION, LABEL, MAIN, REPORT, WHENEVER, WHILE

IF

The IF statement executes a group of statements conditionally. It can switch program control conditionally between two blocks of statements.



Element	Description
statement	is an SQL statement or other 4GL statement.

Usage

If the Boolean expression is TRUE, then 4GL executes the block of statements following the THEN keyword, until it reaches either the ELSE keyword or the END IF keywords. 4GL then resumes execution after the END IF keywords.

If the Boolean expression is FALSE, 4GL executes the block of statements between the ELSE keyword and the END IF statement. If ELSE is absent, execution after the END IF keywords. The Boolean expression returns FALSE if it contains a NULL value (except as the operand of the IS NULL operator).

You can nest IF statements up to a limit (around 20) that also depends on the number of FOR and WHILE loops. If nested IF statements all test the same value, consider using the CASE statement. In the next example, if the value of **direction** matches the string "BACK", 4GL decrements **p_index** by one. If **direction** matches the string "FORWARD", 4GL increments **p_index** by one.

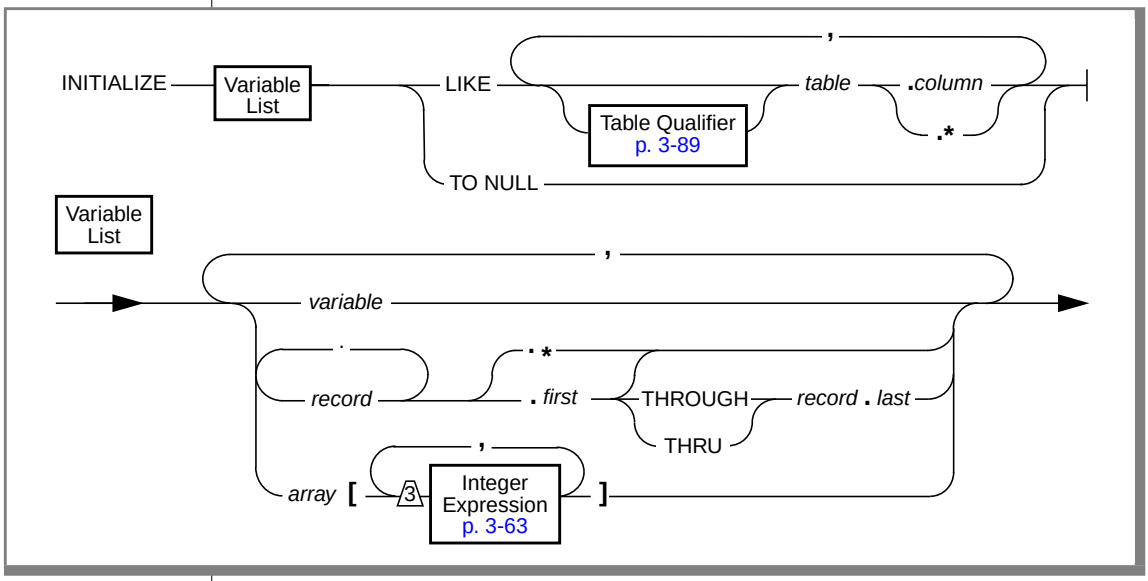
```
IF  direction = "BACK" THEN
   LET p_index = p_index - 1
   DISPLAY dp_stock[p_index].* TO s_stock.*
   ELSE IF direction = "FORWARD" THEN
         LET p_index = p_index + 1
         DISPLAY dp_stock[p_index].* TO s_stock.*
      END IF
END IF
```

References

CASE, FOR, WHENEVER, WHILE

INITIALIZE

The INITIALIZE statement assigns initial NULL or default values to variables.



Element	Description
array	is the name of a variable of the ARRAY data type.
column	is the name of a column of *table* for which a DEFAULT value exists.
first	is the name of a member variable to be initialized.
last	is another member of *record* that was declared later than *first*.
record	is the name of a variable of the RECORD data type.
table	is the name or synonym of the table or view that contains *column*.
variable	is the name of a variable of a simple data type.

Usage

After you declare a variable with a DEFINE statement, the compiler allocates memory to that variable. The contents of the variable, however, is whatever occupies that memory location.

INITIALIZE can specify initial values for 4GL variables in either of two ways:

- The LIKE keyword assigns the default values of a specified database column, using default values from the **syscolval** table.
- You can use the TO NULL keywords to assign NULL values, using the representation of NULL for the declared data type of each variable.

The LIKE Clause

The LIKE clause specifies default values from one or more **syscolval** columns. Just as in the DEFINE or VALIDATE statement, the LIKE clause requires a DATABASE statement to specify a default database (as described in "The Default Database at Compile Time" on page 4-73). The DATABASE statement to specify a default database must precede the first program block in the same module as the INITIALIZE statement.

When initializing variables with the default values of database columns, the variables must match the columns in order, number, and data type. You must prefix the name of each column with the name of its table. For example, the following statement assigns to three variables the default values from three database columns in table **tab1**:

```
INITIALIZE  var1,  var2,  var3
   LIKE  tab1.col1,  tab1.col2,  tab1.col3
```

The *table*.* notation specifies every column in the specified table. If **tab1** has only the three columns (**col1**, **col2**, and **col3**), the following statement is equivalent to the previous one:

```
INITIALIZE  v_cust.*  LIKE  customer.*
```

ANSI

In an ANSI-compliant database, you must qualify each table name with that of its owner (*owner. table*), if the application will be run by a user who does not own the table. For example, if you own **tab1**, and Lydia owns **tab2**, and Boris owns **tab3**, the following statement is valid:

```
INITIALIZE  var1,  var2,  var3
   LIKE  tab1.var1,  lydia.tab2.var2,  boris.tab3.var3 ♦
```

You can include the owner name as a prefix in a database that is not ANSI-compliant, but if the owner name that you specify is incorrect, you receive an error. For additional information, see the *Informix Guide to SQL: Syntax*. The INITIALIZE statement looks up the default values for database columns in the DEFAULT column of the **syscolval** table in the default database.

Any changes to **syscolval** after compilation have no effect on the 4GL program, unless you recompile the program. To enter default values in this table, use the **upscol** utility, as described in Appendix B. If a column has no default value in the **syscolval** table, 4GL assigns NULL values to any variables initialized from that column. If the database is not ANSI-compliant, **upscol** creates a single **syscolval** table.

ANSI

In an ANSI-compliant database, each user can create an ***owner*.syscolval** table, which sets the default values only for the tables owned by that user. If you omit the owner of the table and you own the table, your **syscolval** table becomes the source for the defaults when you compile the program. If the ***owner*.syscolval** table does not exist, the LIKE clause of the INITIALIZE statement sets the values of the specified variables to NULL. ♦

You cannot use **upscol** to specify attributes or validation criteria for TEXT or BYTE columns. Therefore, you cannot use the LIKE clause of the INITIALIZE statement to assign non-NULL values to variables of these large data types.

Use the TO NULL clause to assign a NULL value to a variable. The following statement initializes all variables in the **v_orders** record to NULL:

```
INITIALIZE  v_orders.*  TO  NULL
```

You might wish to initialize variables to NULL for the following reasons:

- To assign an initial value to a variable that has no assigned value.
- To discard some existing value of a variable, which might be convenient if you want to reuse the same variable later in a program

To optimize performance, you might wish to limit the use of this statement. For example, the next program fragment uses INITIALIZE once to create a NULL record, and then uses the LET statement to initialize another record:

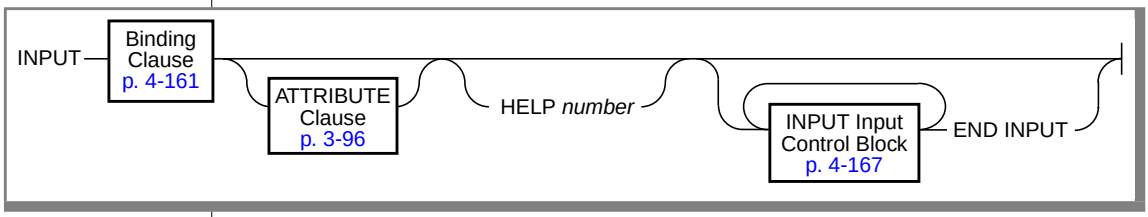
```
DATABASE stores2
MAIN
DEFINE p_customer, n_customer RECORD LIKE customer.*
   INITIALIZE n_customer.* TO NULL
   LET p_customer.* = n_customer.*
```

References

DATABASE, DEFINE, GLOBALS, LET, VALIDATE

INPUT

The INPUT statement supports data entry into fields of a screen form.



Element	Description
number	is a literal integer to specify a help message number.

Usage

The INPUT statement assigns to one or more variables the values that users enter into the fields of a screen form. INPUT can include statement blocks to be executed under conditions that you specify, such as screen cursor movement, or other user actions. The following steps describe how to use this statement:

1. Specify fields in a form specification file, and compile the form.
2. Declare variables with the DEFINE statement.
3. Open and display the screen form in either of the following ways:
   - The OPEN FORM and DISPLAY FORM statements
   - An OPEN WINDOW statement that uses a WITH FORM clause
4. Use the INPUT statement to assign values to the variables from data that the user enters into fields of the screen form.

When the INPUT statement is encountered, 4GL takes the following actions:

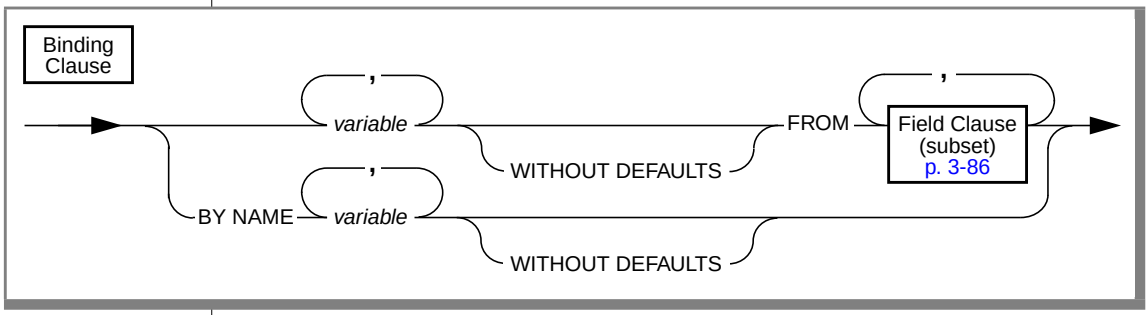
1. Displays any default values in the screen fields, unless you specify the WITHOUT DEFAULTS keywords (as described in "The WITHOUT DEFAULTS Keywords" on page 4-163)
2. Moves the cursor to the first field explicitly or implicitly referenced in the binding clause, and waits for the user to enter data in the field
3. Assigns the user-entered field value to a corresponding program variable when the user moves the cursor from the field or presses the Accept key

The INPUT statement activates the current form (the form that was most recently displayed, or the form in the current 4GL window). When the INPUT statement completes execution, the form is deactivated. After the user presses the Accept key, the INSERT statement of SQL can insert values from the program variables into the appropriate database tables.

The following topics are described in this section:

The Binding Clause

The *binding clause* temporarily associates form fields with 4GL variables, so that the 4GL program can manipulate values that the user enters in the form.



Element	Description
variable	is the name of a variable to store values entered in the field.

Here *variable* supports the syntax of a receiving variable in the LET statement, but you can also use *record*.* or the THRU or THROUGH notation to specify all or some of the members of a program record.

The field names are declared in the ATTRIBUTES section of the form specification. These can be simple fields, members of screen records, WORDWRAP fields, and FORMONLY fields, but cannot include records from screen arrays.

INPUT statements supports two types of binding clauses:

- In the special case where all of the variables have names that are identical (apart from qualifiers) to the names of fields, you can specify INPUT BY NAME *variable list* to bind the specified variables to their namesake fields implicitly. (See also "The BY NAME Clause" on page 4-164.)
- In the general case, you can specify INPUT *variable list* FROM *field list* to bind variables explicitly to fields.

The Correspondence of Variables and Fields

The total number of variables in the variable list must equal the total number of fields that the FROM clause specifies (or that the BY NAME clause implies).

The order in which the screen cursor moves from field to field in the form is determined by the order of the field names in the FROM clause, or else by the order of variable names in the BY NAME clause. (See also "The NEXT FIELD Keywords" on page 4-176, and the WRAP and FIELD ORDER options of the OPTIONS statement described in "Cursor Movement in Interactive Statements" on page 4-296.)

Each screen field and its corresponding variable must have the same (or a compatible) data type. When the user enters data in a field, 4GL checks the value against the data type of the variable, not that of the field. You must first declare all the variables before using the INPUT statement.

The binding clause can specify variables of any 4GL data type. If a variable is declared LIKE a SERIAL column, however, 4GL does not allow the screen cursor to stop in the field. (Values in SERIAL columns are maintained by the database server, not by 4GL.)

Displaying Default Values

If you omit the WITHOUT DEFAULTS keywords, 4GL displays default values from the program array when the form is activated. 4GL determines the default values in the following way, in descending order of precedence:

1. The DEFAULT attribute (from the form specification file)
2. The DEFAULT column value (from the **syscolval** table)

4GL assigns NULL values to all variables for which no default is set. But if you include the WITHOUT NULL INPUT option in the DATABASE section of the form specification file, 4GL assigns the following default values.

Field Type	Default	Field Type	Default
Character	Blank (= ASCII 32)	INTERVAL	0
Number	0	MONEY	$0.00
		DATE	12 / 31 / 1899
		DATETIME	1899-12-31 23:59:59.99999

The WITHOUT DEFAULTS Keywords

If you specify the WITHOUT DEFAULTS option, however, the screen displays the current values of the variables when the INPUT statement begins. This option is available with both the BY NAME and the FROM binding clauses.

The following outline describes how to display initialized values, rather than defaults:

1. Initialize the variables with whatever values you want to display.
2. Call the built-in SET_COUNT( ) function so that 4GL can determine how many rows of data are currently stored in the program array.
3. Use INPUT…WITHOUT DEFAULTS to display the current values of the variables and to allow the user to change those values.

The following INPUT statement causes 4GL to display the character string "Send via air express" in the **ship_instruct** field:

```
LET pr_orders.ship_instruct = "Send via air express"
INPUT BY NAME pr_orders.order_date THRU pr_orders.paid_date
   WITHOUT DEFAULTS
END INPUT
```

The WITHOUT DEFAULTS option is useful when you want the user to be able to make changes to existing rows of the database. You can display the existing database values on the screen before the user begins editing the data. The FIELD_TOUCHED( ) operator (described briefly in "Using Built-In Functions and Operators" on page 4-178, and in detail on "FIELD_TOUCHED( )" on page 5-84) can help you to determine which fields have been altered and which ones therefore require updates to the database.

If you omit the WITHOUT DEFAULTS clause, 4GL determines default values by looking in the following sources of information, in the order indicated:

1. The DEFAULT attribute from the form specification
2. The DEFAULT column as stored in the **syscolval** table

4GL assigns NULL values for all variables for which no default is set.

The BY NAME Clause

The BY NAME clause implicitly binds the fields to the 4GL variables that have the same identifiers as field names. You must first declare variables with the same names as the fields from which they accept input. 4GL ignores any *record name* prefix when making the match.

The unqualified names of the variables and of the fields must be unique and unambiguous within their respective domains. If they are not, 4GL generates a runtime error, and sets the status variable to a negative value. (To avoid this error, use the FROM clause instead of the BY NAME clause when the screen fields and the variables have different names.)

The user can enter values only into fields that are implied in the BY NAME clause. For example, the INPUT statement in the following example specifies variables for all the screen fields except **customer_num**:

```
DEFINE pr_customer RECORD LIKE customer.*
...
INPUT BY NAME pr_customer.fname, pr_customer.lname,
   pr_customer.company, pr_customer.address1,
   pr_customer.address2, pr_customer.city, pr_customer.state,
   pr_customer.zipcode, pr_customer.phone
```

Because **pr_customer.customer_num** does not appear in the list of variables, the user cannot enter a value for it. A functionally equivalent statement is:

```
DEFINE pr_cust RECORD LIKE customer.*
...
INPUT BY NAME pr_cust.fname THRU pr_cust.phone
```

The FROM Clause

When variables and fields do not have the same names, you must use the FROM clause to bind the screen fields to program variables of a program array of records. The user can position the cursor only in fields that are listed explicitly or implicitly in the FROM clause. These fields must correspond both in order and in number to the list of variables, and must be of the same or compatible data types as the corresponding variables:

```
DEFINE pr_cust RECORD LIKE customer.*
...
INPUT pr_cust.fname, pr_cust.lname FROM fname, lname
```

The THRU (or THROUGH) keyword implicitly includes the variables between two specified member variables of a program record. For example, the next statement maps fields to all member variables from **fname** to **phone**:

```
INPUT pr_cust.fname THRU pr_cust.phone
   FROM fname, lname, company, address1,
      address2, city, state, zipcode, phone
```

If the form specification file declared a screen record as **fname** THRU **phone**, you can abbreviate this statement even further:

```
INPUT pr_cust.fname THRU pr_cust.phone FROM sc_cust.*
```

You cannot use the THRU or THROUGH keywords in the FROM clause.

The ATTRIBUTE Clause

For the syntax of the ATTRIBUTE clause, see "The ATTRIBUTE Clause" on page 4-41. This section describes the use of the ATTRIBUTE clause within an INPUT statement.

If you specify form attributes with the INPUT statement, the new attributes apply only during the current activation of the form. When actions of the user deactivate the form, the form reverts to its previous attributes. The following INPUT statement assigns the RED and REVERSE attributes:

```
INPUT  p_addr.* FROM sc_addr.* ATTRIBUTE (RED, REVERSE)
```

This statement assigns the WHITE attribute:

```
INPUT BY NAME p_items ATTRIBUTE (WHITE)
```

The ATTRIBUTE clause overrides display attributes specified in a DISPLAY FORM, OPTIONS, or OPEN WINDOW statement, and suppresses any default attributes specified in the **syscolatt** table of the **upscol** utility.

The HELP Clause

The HELP clause includes a literal integer to specify the *number* of the help message to display. (For more information, see "Literal Integers" on page 3-65.) The help message is displayed in the Help window, as described in "The Help Window" on page 2-30. This window appears if the user presses the Help key while the screen cursor is in any field that you listed in the FROM clause, or that you implied in the BY NAME clause.

The default Help key is CONTROL-W, but you can specify a different Help key by using the OPTIONS statement (as described in "The OPTIONS ATTRIBUTE Clause" on page 4-297).

This example specifies help message 311 if the user requests help from any field in the **s_items** screen array:

```
INPUT p_items.* FROM s_items.* HELP 311
```

The next example tells 4GL to display message 12 if the user presses the Help key when the screen cursor is in either of two fields:

```
INPUT cust.fname, cust.lname FROM fname, lname HELP 12
```

You create help messages in an ASCII file whose filename you specify in the HELP FILE clause of the OPTIONS statement. Use the **mkmessage** utility, as described in Appendix B, to create a runtime version of the help file. A runtime error occurs in the following situations:

- 4GL cannot open the help file.
- You specify a number that is not in the help file.
- You specify a number outside the range from -32,767 to 32,767.

The help message corresponding to your HELP clause specification applies to the entire INPUT statement. To override this with field-level help messages, specify the Help key in an ON KEY block that invokes the INFIELD( ) operator and SHOWHELP( ) function. (For more information, see "The ON KEY Block" on page 4-171.)

If you provide messages to assist the user through an ON KEY clause, rather than by the HELP clause, the messages must be displayed in a 4GL window within the 4GL screen, rather than in the separate Help window.

The INPUT Control Block

Each INPUT control block includes a statement block of at least one statement, and an activation clause that specifies when to execute the statement block. An input control block can specify any of the following items:

- The statements to execute before or after visiting specific screen fields
- The statements to execute when the user presses a key sequence
- The statements to execute before or after the INPUT statement
- The next field to which to move the screen cursor
- When to terminate execution of the INPUT statement

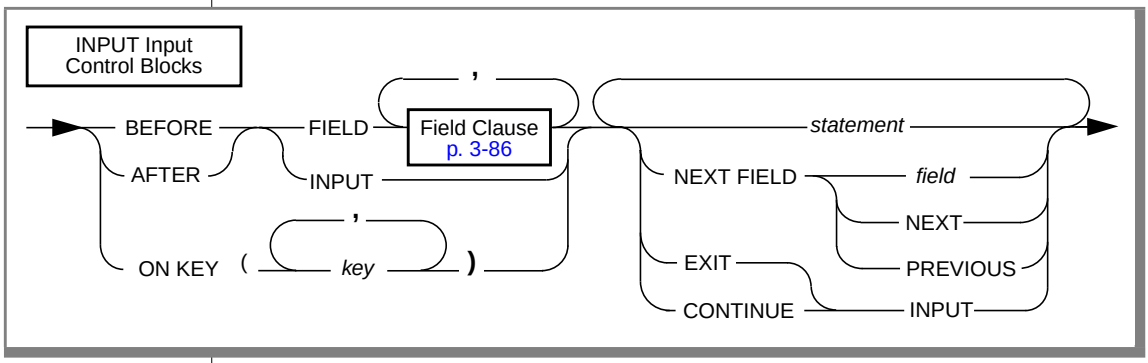
The activation clause can specify any one of the following items:

- Pre- and post-INPUT actions (the BEFORE or AFTER INPUT clause)
- Keyboard sequence conditions (the ON KEY clause)
- Cursor movement conditions (the BEFORE or AFTER FIELD clause)

The statement block can include any SQL or 4GL statements, as well as the following items:

- Cursor movement instructions (the NEXT FIELD clause)
- Termination of the INPUT statement (the EXIT INPUT statement)
- Returning control to the user without terminating the INPUT statement (the CONTINUE INPUT statement)

The activation clause and the statement block correspond respectively to the left-hand and right-hand elements in the following syntax diagram.



Element	Description
field	is the name of a field (as described in "Field Clause" on page 3-86) in the current form.
key	is one or more keywords to specify physical or logical keys. For details, see "The ON KEY Block" on page 4-171.
statement	is an SQL statement or other 4GL statement.

After BEFORE FIELD, AFTER FIELD, or NEXT FIELD, the field clause specifies a field that the binding clause referenced implicitly (in the BY NAME clause, or as *record*.* or *array* [*line*].*) or explicitly. You can qualify a field name by a table reference, or the name of a screen record or a screen array or *array* [*line*].

If you include one or more control blocks, the END INPUT keywords must terminate the INPUT statement. If no control block is included, 4GL waits while the user enters values into the fields. When the user accepts the values in the form, the INPUT statement terminates.

If you include a control block, 4GL executes or ignores the statements in that statement block, depending on the following items:

- Whether you specify the BEFORE INPUT or AFTER INPUT keywords
- The fields to which and from which the user moves the screen cursor
- The keys that the user presses

4GL deactivates the form while executing statements in a control block. After executing the statements, 4GL reactivates the form, allowing the user to continue entering or modifying the data values in fields.

The Precedence of Input Control Blocks

This is the order in which 4GL executes the statements from control blocks:

1. BEFORE INPUT
2. BEFORE FIELD
3. ON KEY
4. AFTER FIELD
5. AFTER INPUT

You can list these blocks in any order. If you develop some consistent ordering, however, your code will be easier to read.

Within these control blocks, you can include the NEXT FIELD keywords and the CONTINUE INPUT and EXIT INPUT statements, as well as most 4GL and SQL statements. See "Nested and Recursive Statements" on page 2-31 for information about including CONSTRUCT, PROMPT, INPUT, and INPUT ARRAY statements within an input control block.

The activation clauses that you can specify in control blocks are described in their order of execution by 4GL. Descriptions of NEXT FIELD and EXIT INPUT follow the discussions of these activation clauses. No subsequent INPUT control block statements are executed if EXIT INPUT executes.

The BEFORE INPUT Block

You can use the BEFORE INPUT block to display messages on how to use the INPUT statement. For example, the following INPUT statement fragment displays a message informing the user how to enter data into the table:

```
INPUT BY NAME p_customer.*
   BEFORE INPUT
      DISPLAY "Press ESC to enter data" AT 1,1
```

4GL executes the BEFORE INPUT block after displaying the default values in the fields and before letting the user enter any values. (If you included the WITHOUT DEFAULTS clause, 4GL displays the current values of the variables, not the default values, before executing the BEFORE INPUT block.)

An INPUT statement can include no more than one BEFORE INPUT block. You cannot include the FIELD_TOUCHED( ) operator in the BEFORE INPUT block.

The BEFORE FIELD Block

4GL executes the statements in the BEFORE FIELD block associated with a field whenever the cursor moves into the field, but before the user enters a value. You can specify no more than one BEFORE FIELD block for each field.

The following program fragment defines two BEFORE FIELD blocks. When the cursor enters the **fname** or **lname** field, 4GL displays a message:

```
BEFORE FIELD fname
   MESSAGE  "Enter first name of customer"
BEFORE FIELD lname
   MESSAGE  "Enter last name of customer"
```

You can use a NEXT FIELD clause within a BEFORE FIELD block to restrict access to a field. You can also use a DISPLAY statement within a BEFORE FIELD block to display a default value in a field.

The following statement fragment causes 4GL to prompt the user for input when the cursor is in the **stock_num**, **manu_code**, or **quantity** field:

```
INPUT p_items.* FROM s_items.*
   BEFORE FIELD stock_num
      MESSAGE "Enter a stock number."
   BEFORE FIELD manu_code
      MESSAGE "Enter the code for a manufacturer."
   BEFORE FIELD quantity
      MESSAGE "Enter a quantity."
   ...
END INPUT
```

The ON KEY Block

Statements in the ON KEY block are executed if the user presses some key that you specify by the keywords in the following table (in lowercase or uppercase letters).

ACCEPT	HELP	NEXT or NEXTPAGE
DELETE	INSERT	PREVIOUS or PREVPAGE
DOWN	INTERRUPT	RETURN
ESC or ESCAPE	LEFT	TAB
F1 through F64	RIGHT	UP
CONTROL-*char* (except A, D, H, I, J, K, L, M, R, or X)		

For example, the following ON KEY block displays a help message. The BEFORE INPUT clause informs the user how to access help:

```
BEFORE INPUT
   DISPLAY "Press CONTROL-W or CTRL-F for Help"
ON KEY (CONTROL-W, CONTROL-F)
   CALL customer_help()
```

The next statement defines an ON KEY block for the CONTROL-B key. Whenever the user presses CONTROL-B, 4GL determines if the screen cursor is in the **stock_num** or **manu_code** field. If it is in either one of these fields, 4GL calls the **stock_help( )** function and sets **quantity** as the next field.

```
INPUT p_items.* FROM s_items.*
   ON KEY (CONTROL-B)
      IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
         CALL stock_help()
         NEXT FIELD quantity
      END IF
```

Some keys require special consideration if specified in an ON KEY block.

Key	Special Considerations
ESC or ESCAPE	You must specify another key as the Accept key in the OPTIONS statement, because this is the default Accept key.
Interrupt	You must execute a DEFER INTERRUPT statement. If the user presses the Interrupt key under these conditions, 4GL executes the statements in the ON KEY block and sets **int_flag** to non-zero, but does not terminate the INPUT statement.
Quit	4GL also executes the statements in this ON KEY block if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, 4GL sets **quit_flag** to non-zero.
CTRL-*char* A, D, H, K, L, R, X	4GL reserves these control keys for field editing; see "Cursor Movement in Simple Fields" on page 4-180.
I, J, M	The standard meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is not available to the user. Instead, the key is trapped by 4GL and activates the commands in the ON KEY block. For example, if CONTROL-M appears in an ON KEY block, the user cannot press RETURN to advance the cursor to the next field. If you specify one of these keys in an ON KEY block, be careful to restrict the scope of the statement.

You might not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

If you use the OPTIONS statement to redefine the Accept or Help key, the keys assigned to these sequences cannot be used in an ON KEY clause. For example, if you redefine the Accept key by using the following statement, you should not define an ON KEY block for the key sequence CONTROL-B:

```
OPTIONS ACCEPT KEY (CONTROL-B)
```

When the user presses CONTROL-B, 4GL will always perform the Accept key function, regardless of the presence of an ON KEY (CONTROL-B) block.

If the user activates an ON KEY block while entering data in a field, 4GL takes the following actions:

1. Suspends input to the current field
2. Preserves the input buffer that contains the characters the user has typed
3. Executes the statements in the current ON KEY block
4. Restores the input buffer for the current screen field
5. Resumes input in the same field, with the screen cursor at the end of the buffered list of characters

You can change this default behavior by performing the following tasks in the ON KEY block:

- Resuming input in another field by using the NEXT FIELD statement
- Changing the input buffer value for the current field by assigning a new value to the corresponding variable, and then displaying this value

This block can support *accelerator keys* for common functions, such as saving and deleting. You can use the INFIELD( ) operator in the ON KEY clause to support field-specific actions. For example, you can implement field-level help by using the INFIELD( ) operator and the built-in SHOWHELP( ) function.

The AFTER FIELD Block

4GL executes the statements in the AFTER FIELD block associated with a field every time the cursor leaves the specified field. Any of the following keys can cause the cursor to leave the field:

- The HOME or END key
- Any arrow key
- The RETURN or TAB key
- The Accept key
- The Interrupt or Quit key (if a supporting DEFER statement was included)

You can specify only one AFTER FIELD block for each field.

This AFTER FIELD block checks if the **stock_num** and **manu_code** fields contain values. If they contain values, 4GL calls the **get_item( )** function:

```
AFTER FIELD stock_num, manu_code
   LET pa_curr = ARR_CURR()
   IF p_items[pa_curr].stock_num IS NOT NULL
      AND p_items[pa_curr].manu_code IS NOT NULL THEN
         CALL get_item()
         IF p_items[pa_curr].quantity IS NOT NULL THEN
            CALL get_total()
         END IF
   END IF
```

The following INPUT statement performs a NULL test to determine whether the user entered a value in the **address1** field, and returns to that field if no value was entered:

```
INPUT p_addr.* FROM sc_addr.*
   AFTER FIELD address1
      IF p_addr.address1 IS NULL THEN
         NEXT FIELD address1
      END IF
END INPUT
```

The user terminates the INPUT statement by pressing the Accept key when the cursor is in any field, or by pressing the TAB or RETURN key after the *last* field. You can use the AFTER FIELD block on the last field to override this default termination. (Including the INPUT WRAP in the OPTIONS statement produces the same effect.)

When the NEXT FIELD keywords appear in an AFTER FIELD block, the cursor moves to in the specified field. If an AFTER FIELD block appears for each field, and NEXT FIELD keywords are in each block, the user cannot leave the form.

The AFTER INPUT Block

4GL executes the AFTER INPUT block when the user presses the Accept key. You can use the AFTER INPUT block to validate, save, or alter the values the user entered by using the built-in GET_FLDBUF( ) or FIELD_TOUCHED( ) operator within the AFTER INPUT clause. (Use of these operators in an INPUT statement is described in"Using Built-In Functions and Operators" on page 4-178.)

The next example uses the AFTER INPUT block to require that a first name be specified for any customers with the last name Smith:

```
INPUT BY NAME p_customer.fname THRU p_customer.phone
   AFTER INPUT
      IF p_customer.lname="Smith" THEN
         IF NOT FIELD_TOUCHED(p_customer.fname) THEN
            CALL mess("You must enter a first name.")
            NEXT FIELD fname
         END IF
      END IF
   END INPUT
```

4GL executes the AFTER INPUT block only when the INPUT statement is terminated by the user pressing one of the following keys:

- The Accept key
- The Interrupt key (if the DEFER INTERRUPT statement has executed)
- The Quit key (if the DEFER QUIT statement has executed)

The AFTER INPUT clause is not executed in the following situations:

- The user presses the Interrupt or Quit key when the DEFER INTERRUPT or DEFER QUIT statement, respectively, has not executed. In either case, the program terminates immediately.
- The EXIT INPUT statement terminates the INPUT statement.

You can place the NEXT FIELD clause in this block to return the cursor to the form. If you place a NEXT FIELD clause in the AFTER INPUT block, use it in a conditional statement. Otherwise, the user cannot exit from the form.

No more than one AFTER INPUT block can appear in an INPUT statement.

The NEXT FIELD Keywords

The NEXT FIELD keywords specify the next field to which 4GL moves the screen cursor. If you do not specify a NEXT FIELD clause, by default the cursor moves among the screen fields according to the explicit or implicit order of fields in the INPUT binding clause. The user can control movement from field to field by using the arrow keys, TAB, and RETURN. By using the NEXT FIELD keywords, however, you can explicitly position the screen cursor.

You must specify one of the following options with the NEXT FIELD keywords.

Clause	Effect
NEXT FIELD NEXT	Advances the cursor to the next field
NEXT FIELD PREVIOUS	Returns the cursor to the previous field
NEXT FIELD *field-name*	Moves the cursor to *field-name*

For example, this NEXT FIELD clause places the cursor in the previous field:

```
NEXT FIELD PREVIOUS
```

The following INPUT statement includes a NEXT FIELD clause in an ON KEY block. If the user presses CONTROL-B when the cursor is in the **stock_num** or **manu_code** field, 4GL moves the cursor to **quantity** as the next field:

```
INPUT p_items.* FROM s_items.*
   ON KEY (CONTROL-B)
      IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
         CALL stock_help()
         NEXT FIELD quantity
      END IF
   ...
END INPUT
```

4GL immediately positions the cursor in the form when it encounters the NEXT FIELD clause; it does not execute any statements that follow the NEXT FIELD clause in the control block.

For example, 4GL cannot invoke function **qty_help( )** in the next example:

```
ON KEY (CONTROL-B, F4)
   IF INFIELD(stock_num) OR INFIELD(manufact) THEN
      CALL stock_help()
      NEXT FIELD quantity
      CALL qty_help() -- function is never called
   END IF
```

You can use the NEXT FIELD clause in any INPUT input control block. The NEXT FIELD clause typically appears in a conditional statement. In an AFTER INPUT clause, the NEXT FIELD statement *must* appear in a conditional statement; otherwise, the user cannot exit from the form. To restrict access to a field, use the NEXT FIELD statement in a BEFORE FIELD clause.

The following example demonstrates using the NEXT FIELD clause in an ON KEY control block. 4GL executes the ON KEY block if the user presses CONTROL-W. If the cursor is in the **city** field, 4GL displays `San Francisco` in the **city** field and `CA` in the **state** field, and then moves the cursor to the **zipcode** field.

```
ON KEY (CONTROL-W)
   IF INFIELD(city) THEN
      LET p_addr.city = "San Francisco"
      DISPLAY p_addr.city TO city
      LET p_addr.state = "CA"
      DISPLAY p_addr.state TO state
      NEXT FIELD zipcode
   END IF
```

To wrap from the last field of a form to the first field of a form, use the NEXT FIELD statement after an AFTER FIELD clause for the last field of the form. (The INPUT WRAP option of the OPTIONS statement has the same effect.)

The CONTINUE INPUT Statement

The CONTINUE INPUT statement causes 4GL to skip all subsequent statements in the current control block. The screen cursor returns to the most recently occupied field in the current form.

The CONTINUE INPUT statement is useful when program control is nested within multiple conditional statements, and you want to return control to the user. It is also useful in an AFTER INPUT control block that examines the field buffers; depending on their contents, you can return the cursor to the form.

The EXIT INPUT Statement

The EXIT INPUT statement terminates input. 4GL performs the following tasks:

- Skips all statements between the EXIT INPUT and END INPUT keywords
- Deactivates the form
- Resumes execution at the first statement after the END INPUT keywords

4GL ignores any statements in an AFTER INPUT control block if the EXIT INPUT statement is executed.

The END INPUT Keywords

The END INPUT keywords indicate the end of the INPUT statement. These keywords should follow the last control block. If you do not include any control blocks, the END INPUT keywords are not required.

Using Built-In Functions and Operators

The INPUT statement supports built-in functions and operators of 4GL. (For more about these built-in 4GL functions and operators, see Chapter 5.) The following features allow you to access field buffers and keystroke buffers.

Feature	Description
FIELD_TOUCHED( )	Returns TRUE if the user has "touched" (made a change to) a screen field whose name is passed as an operand. Moving the screen cursor through a field (with the RETURN, TAB, or arrow keys) does not mark a field as touched. This operator also ignores the effect of statements that appear in the BEFORE INPUT control block. For example, you can assign values to fields in the BEFORE INPUT control block without having the fields marked as touched.
GET_FLDBUF( )	Returns the character values of the contents of one or more fields in the currently active form

(1 of 2)

Feature	Description
FGL_GETKEY( )	Waits for a key to be pressed, and then returns an INTEGER corresponding to the raw value of the key that was pressed.
FGL_LASTKEY( )	Returns an INTEGER corresponding to the most recent keystroke executed by the user while in the screen form.
INFIELD( )	Returns TRUE if the name of the field that is specified as its operand is the name of the current field.

(2 of 2)

Each field has only one field buffer, and a buffer cannot be used by two different statements simultaneously. If you plan to display the same form with data entry fields more than once, you should open a new 4GL window and open and display a second copy of the form. 4GL allocates a separate set of buffers to each form, so this avoids overwriting field buffers when more than one INPUT, INPUT ARRAY, or CONSTRUCT statement accepts input.

The next example of an INPUT statement uses the INFIELD( ) operator to determine if the cursor is in the **stock_num** or **manu_code** field.

If the cursor is in one of these fields, 4GL calls the **stock_help( )** function and sets **quantity** as the next field:

```
INPUT p_items.* FROM s_items.*
   ON KEY (CONTROL-B)
      IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
         CALL stock_help()
         NEXT FIELD quantity
      END IF
```

The INFIELD(*field*) expression returns TRUE if the current field is *field* and FALSE otherwise. Use this function for field-dependent actions when the user presses a key in the ON KEY block. In the following INPUT statement, the BEFORE FIELD clause for the **city** field displays a message advising the user to press a control key to enter the value `San Francisco` into the field:

```
INPUT p_customer.fname THRU p_customer.phone
   FROM sc_cust.* ATTRIBUTE(REVERSE)
   BEFORE FIELD city
      MESSAGE "Press CONTROL-F to enter San Francisco"
   ON KEY (CONTROL-F)
      IF INFIELD(city) THEN
         LET p_customer.city = "San Francisco"
```

```
            DISPLAY p_customer.city TO city
            LET p_customer.state = "CA"
            DISPLAY p_customer.state TO state
            NEXT FIELD zipcode
         END IF
END INPUT
```

If the user presses CONTROL-F while the cursor is in the **city** field, the ON KEY clause in this example changes the screen display in three ways:

1. Displays the value `San Francisco` in the **city** field
2. Displays `CA` in the **state** field
3. Moves the cursor to the first character position in the **zipcode** field

Keyboard Interaction

The user of the 4GL application can position the visual cursor during the INPUT statement by keyboard actions.

Some keys are sensitive to what kind of field the cursor occupies:

- A simple field
- A segment of a multiple-segment field

Subsequent sections describe cursor movement in both environments.

Cursor Movement in Simple Fields

In a simple field, when the user presses TAB or RETURN, the cursor moves from one screen field to the next in an order based on the binding clause:

- For INPUT BY NAME, 4GL uses the order implied by the sequence of program variables specified in the binding clause.
- Otherwise, 4GL uses the order of the screen fields specified in the FROM clause of the INPUT statement.

The user can press the arrow keys to position the screen cursor.

Arrow	Effect
↓	By default, DOWN ARROW moves the cursor to the next field. If you specify FIELD ORDER UNCONSTRAINED in the OPTIONS statement, this key moves the cursor to the field below the current field. If no field is below the current field and a field exists to the left of the current field, 4GL moves the cursor to the field to the left.
↑	By default, UP ARROW moves the cursor to the previous field. If you specify the FIELD ORDER UNCONSTRAINED option of the OPTIONS statement, this key moves the cursor to the field above the current field. If no field is above the current field and a field exists to the left of the current field, 4GL moves the cursor to the field to the left.
→	RIGHT ARROW moves the cursor one space to the right inside a screen field, without erasing the current character. At the end of the field, 4GL moves the cursor to the first character position of the next screen field. RIGHT ARROW is equivalent to the CONTROL-L editing key.
←	LEFT ARROW moves the cursor one space to the left inside a screen field without erasing the current character. At the beginning of the field, 4GL moves the cursor to the first character position of the previous field. LEFT ARROW is equivalent to the CONTROL-H editing key.

Unless a field has the NOENTRY attribute, the user can press the following keys during an INPUT statement to edit values in a screen field.

Editing Key	Effect
CONTROL-A	Toggles between insert and type-over mode
CONTROL-D	Deletes characters from the current cursor position to the end of the field
CONTROL-H	Moves the cursor nondestructively one space to the left; equivalent to pressing left arrow

(1 of 2)

Editing Key	Effect
CONTROL-L	Moves the cursor nondestructively one space to the right; equivalent to pressing right arrow
CONTROL-R	Redisplays the screen
CONTROL-X	Deletes the character beneath the cursor

(2 of 2)

Multiple-Segment Fields

For an explanation of how you can create a *multiple-segment* field to display long character strings, see "Multiple-Segment Fields" on page 6-31. These fields superficially resemble a screen array, but the successive lines are segments of the same field, rather than screen records.

If the data string is too long to fit in the first segment, 4GL divides it at a blank character (if possible), padding the rest of the segment on the right with blank (ASCII 32) characters, and continues the display in the next field segment. If necessary, this process is repeated until all of the segments are filled, or until the last text character is displayed (whichever happens first).

If the user inserts or deletes characters while editing a multiple-segment field, the WORDWRAP attribute can move down subsequent characters as needed. Blank characters that the WORDWRAP editor uses as padding are called *editor blanks*. The COMPRESS keyword in the form specification can prevent storage of editor blanks in the database. Characters that users enter or that 4GL retrieves from the database are called *intentional* characters.

If the cursor enters a multiple-segment field, additional features of a multiple line editor become available to the user. The user must press CONTROL-M for NEWLINE, because RETURN moves the cursor to the next field.

WORDWRAP Editing Keys

When values are entered or updated in a multiple-segment field, the user can press keys to move the screen cursor over the data, and to insert, delete, and type over the data. The cursor never pauses on editor blanks.

The WORDWRAP editor has two modes, *insert* (to add data at the cursor) and *type-over* (to replace the displayed data with entered data). Users cannot overwrite a NEWLINE. If the cursor is in type-over mode and encounters a NEWLINE character, the mode automatically changes to insert, "pushing" the NEWLINE character to the right. Some keystrokes behave differently in the two modes.

When it first enters a multiple-segment field, the cursor is positioned on the first character of the first field segment, and the editing mode is set to type-over. The cursor movement keys are as follows.

Key	Effect
RETURN	Leaves the entire multiple-segment field, and goes to the first character of the next field.
BACKSPACE or LEFT ARROW	Moves left one character, unless at the left edge of a field segment. From the beginning of the first segment, these move to the first character of the preceding field (if INPUT WRAP is in effect), or beep (if INPUT NO WRAP; see the OPTIONS statement). From the left edge of a lower field segment, these keys move to the last intentional character of the previous field segment.
RIGHT ARROW	Moves right one character, unless at the right-most intentional character in a segment. From the right-most intentional character of the last segment, this either moves to the first character of the next field, or only beeps, depending on INPUT WRAP mode. From the last intentional character of a higher segment, this moves to the first intentional character in a lower segment.
UP ARROW	Moves from the top-most segment to the first character of the preceding field. From a lower segment, this moves to the character in the same column of the next higher segment, jogging left, if required, to avoid editor blanks, or if it encounters a tab.

(1 of 2)

Key	Effect
DOWN ARROW	Moves from the lowest segment to the first character of the next field. From a higher segment, moves to the character in the same column in the next lower segment, jogging left if required to avoid editor blanks, or if it encounters a tab.
TAB	Enters a tab character, in insert mode, and moves the cursor to the next tab stop. This can cause following text to jump right to align at a tab stop. In type-over mode, this moves the cursor to the next tab stop that falls on an intentional character, going to the next field segment if required.

(2 of 2)

The character keys enter data. Any following data shifts right, and words can move down to subsequent segments. This can result in characters being discarded from the final field segment. These keystrokes can also alter data.

Key	Effect
CONTROL-A	Switches between type-over and insert mode.
CONTROL-X	Deletes the character under the cursor, possibly causing words to be pulled up from subsequent segments.
CONTROL-D	Deletes all text from the cursor to the end of the multiple-line field (not merely to the end of the current field segment).
CONTROL-N	Inserts a NEWLINE character, causing subsequent text to align at the first column of the next segment of the field, and possibly moving words down to subsequent segments. This can result in characters being discarded from the final segment of the field.

The editing keys (described in "Editing Keys" on page 4-220) have the same effect in a multiple-segment field, except that CONTROL-H can move to the last intentional character of the previous segment of the same field, if the cursor is on the first intentional character. Also, CONTROL-L can move to the first intentional character of the next segment of the same field from the last intentional character of a segment.

Using Large Data Types

4GL displays values of large data types (BYTE or TEXT) as follows.

Field Type	Screen Display
TEXT	As much of the TEXT data as fit within the screen field.
BYTE	The string "<BYTE value>". 4GL cannot display the actual BYTE value in a screen field.

Use a simple field. (You can display part of a TEXT value in a multiple-segment field, but the WORDWRAP editor cannot process a TEXT value.)

If the form specification file assigns an appropriate attribute to a BYTE or TEXT field, the user can invoke an external program by pressing the exclamation point (!) key when the cursor is in the field. This external program is typically an editor to allow the user to edit character (TEXT) or graphics (BYTE) data. To implement this feature, specify the PROGRAM attribute as part of the field description in the form specification file, identifying the external program to execute. (For more information on using the PROGRAM attribute, see the description of that field attribute in Chapter 6.)

The external program takes over the entire screen. Any key sequence that you have specified in the ON KEY clause is ignored by the external program. When the external program terminates, 4GL performs the following tasks:

1. Restores the screen to its state before the external program began
2. Resumes the INPUT statement at the BYTE or TEXT field
3. Reactivates any key sequences specified in the ON KEY clause

Completing the INPUT Statement

The following actions can terminate the INPUT statement:

- The user chooses one of the following keys:
  - The Accept, Interrupt, or Quit key
  - The RETURN or TAB key from the last field (and INPUT WRAP is not currently set by the OPTIONS statement)
- 4GL executes the EXIT INPUT statement.

By default, the Accept, Interrupt, and Quit keys terminate the INPUT statement. Each of these actions also deactivates the form. (But pressing the Interrupt or Quit key can immediately terminate the program, unless the program also includes the DEFER INTERRUPT and DEFER QUIT statements.)

The user must press the Accept key explicitly to complete the INPUT statement under the following conditions:

- INPUT WRAP is specified in the OPTIONS statement.
- An AFTER FIELD block for the last field includes a NEXT FIELD clause.

If 4GL previously executed a DEFER INTERRUPT statement in the program, the Interrupt key causes 4GL to take the following actions:

- Set the global variable **int_flag** to a non-zero value.
- Terminate the INPUT statement, but not the 4GL program.

If 4GL previously executed a DEFER QUIT statement in the program, a Quit signal causes 4GL to take the following actions:

- Set the global variable **quit_flag** to a non-zero value.
- Terminate the INPUT statement, but not the 4GL program.

Executing Control Blocks When INPUT Terminates

When INPUT terminates, these blocks are executed in the order indicated:

1. The AFTER FIELD clause for the current field
2. The AFTER INPUT clause

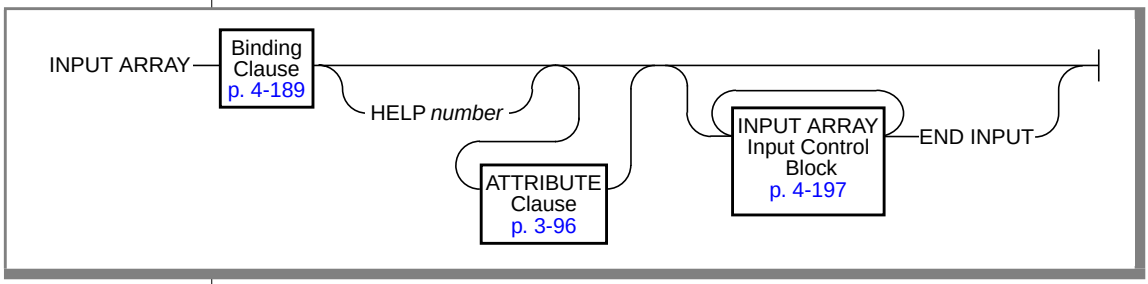
If INPUT terminates by an EXIT INPUT statement, or by pressing the Interrupt or Quit key, 4GL does not execute any of these clauses. If a NEXT FIELD statement appears in one of these clauses, 4GL places the cursor in the specified field and returns control to the user.

References

DEFER, DISPLAY ARRAY, INPUT ARRAY, OPEN WINDOW, OPTIONS

INPUT ARRAY

The INPUT ARRAY statement supports data entry by users into a screen array, and stores the entered data in a program array of records.



Element	Description
number	is a literal integer to specify a help message number.

Usage

The INPUT ARRAY statement assigns to variables in one or more program records the values that the user enters into the fields of a screen array. This statement can include statement blocks to be executed under conditions that you specify, such as screen cursor movement, or other user actions. The following outline describes how to use the INPUT ARRAY statement:

1. Create a screen array in the form specification, and compile the form.
2. Declare an ARRAY OF RECORD with the DEFINE statement.
3. Open and display the screen form in either of the following ways:
   - Using the OPEN FORM and DISPLAY FORM statements
   - Using an OPEN WINDOW statement with the WITH FORM clause
4. Use the INPUT ARRAY statement to assign values to the program array from data that the user enters into fields of the screen array.

When the INPUT ARRAY statement is encountered, 4GL performs the following tasks:

1. Displays any default values in the screen fields, unless you specify the WITHOUT DEFAULTS keywords (as described on page 4-191)
2. Moves the cursor to the first field and waits for input from the user
3. Assigns the user-entered value to a corresponding program variable

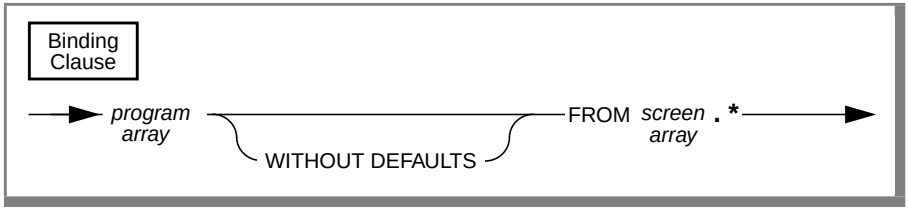
Assignment of the entered value to the variable occurs when the cursor moves from the field or the user presses the Accept key (typically ESCAPE).

The INPUT ARRAY statement activates the *current form* (the form that was most recently displayed or the form in the current 4GL window). When the INPUT ARRAY statement completes execution, the form is deactivated. After the user presses the Accept key, the INSERT statement of SQL can insert the values of the program variables into the appropriate database tables.

The following topics are described in this section:

The Binding Clause

The *binding clause* temporarily associates the member variables in an array of program records with fields in the member records of a screen array, so the 4GL program can manipulate values that the user enters in the screen array.



Element	Description
program array	is the name of an array of program records.
screen array	is the name of an array of screen records.

You must declare the program array in a DEFINE statement within your 4GL program; the screen array must be declared in the form specification file.

The Correspondence of Variables and Fields

The FROM clause binds the screen records in the screen array to the program records of the program array. The form can include other fields that are not part of the specified screen array, but the number of member variables in each record of *program array* must equal the number of fields in each row of *screen array*. Each variable must be of the same (or a compatible) data type as the corresponding screen field. When the user enters data, 4GL checks the entered value against the data type of the variable, not the data type of the screen field.

The member variables of the records in *program array* can be of any 4GL data type. If a variable is declared LIKE a SERIAL column, however, 4GL does not allow the screen cursor to stop in the field. (Values in SERIAL columns are maintained by the database server, not by 4GL.)

The number of screen records in *screen array* determines how many rows the form can display at one time. The size of *record array* determines how many RECORD variables your program can store. If the size of a program array exceeds the size of its screen array, users can press the Next Page or Previous Page keys to scroll through the screen array. (For more information, see "Keyboard Interaction" on page 4-219.)

The default order in which the screen cursor moves from field to field in the screen array is determined by the declared order of the corresponding member variables, beginning in the first screen record. (See also the NEXT FIELD keywords in "The NEXT FIELD Keywords" on page 4-176, and the WRAP and FIELD ORDER options of the OPTIONS statement, as described in "Cursor Movement in Interactive Statements" on page 4-296.)

Displaying Default Values

If you omit the WITHOUT DEFAULTS keywords, 4GL displays default values from the program array when the form is activated. 4GL determines the default values in the following way, in descending order of precedence:

1. The DEFAULT attribute (from the form specification file)
2. The DEFAULT column value (from the **syscolval** table)

4GL assigns NULL values to all variables for which no default is set. But if you include the WITHOUT NULL INPUT option in the DATABASE section of the form specification file, 4GL assigns these non-NULL default values.

Field Type	Default	Field Type	Default
Character	Blank (= ASCII 32)	INTERVAL	0
Number	0	MONEY	$0.00
		DATE	12 / 31 / 1899
		DATETIME	1899-12-31 23:59:59.99999

The WITHOUT DEFAULTS Keywords

If you specify the WITHOUT DEFAULTS option, however, the screen displays *current* values of the variables when the INPUT ARRAY statement begins. This option is available with both the BY NAME and the FROM binding clauses. The following steps describe how to display initialized values:

1. Initialize the variables with whatever values you want to display.
2. Call the built-in SET_COUNT( ) function to tell 4GL how many rows are currently stored in the program array.
3. Specify INPUT ARRAY…WITHOUT DEFAULTS to display current values, and to allow the user to change those records.

The WITHOUT DEFAULTS clause is useful when you want the user to be able to make changes to existing rows of the database. You can display the existing database values on the screen before the user begins editing the data. The FIELD_TOUCHED( ) operator can help you to determine which fields have been altered, and which ones therefore require updates to the database. (This operator is described briefly in "Using Built-In Functions and Operators" on page 4-215, and in detail in "FIELD_TOUCHED( )" on page 5-84.)

The ATTRIBUTE Clause

This resembles the ATTRIBUTE clause of other form-based statements like CONSTRUCT. Except for CURRENT ROW DISPLAY, as described in the next section, attributes that you specify apply to all of the fields in *screen array*.

For the syntax of this clause in specifying color and intensity attributes in screen interaction statements, see "The ATTRIBUTE Clause" on page 4-41.

If you specify form attributes with the INPUT ARRAY statement, the new attributes apply only during the current activation of the form. When actions of the user deactivate the form, the form reverts to its previous attributes. The following INPUT ARRAY statement assigns the RED and REVERSE attributes:

```
INPUT ARRAY p_addr FROM sc_addr.* ATTRIBUTE (RED, REVERSE)
```

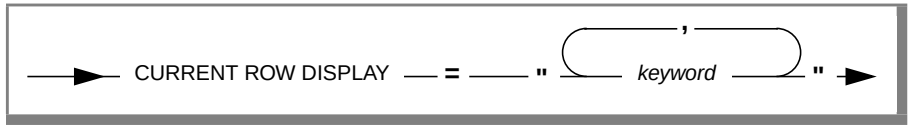
This statement assigns the WHITE attribute:

```
INPUT ARRAY p_items FROM sc_items.* ATTRIBUTE (WHITE)
```

The ATTRIBUTE clause temporarily overrides any default display attributes that were specified in an OPTIONS, DISPLAY FORM, or OPEN WINDOW statement for these fields. It also suppresses any default attributes that were specified in the **syscolatt** table by the **upscol** utility.

Highlighting the Current Row of the Screen Array

Besides color and intensity attributes that "ATTRIBUTE Clause" on page 3-96 describes, the ATTRIBUTE clause of the INPUT ARRAY statement also supports this syntax.



Element	Description
keyword	is zero or one of the *color* attribute keywords, and zero or more of the *intensity* attribute keywords (except DIM, INVISIBLE, and NORMAL) from the syntax diagram of "The ATTRIBUTE Clause" on page 4-41.

The comma-separated list of attributes within the quoted string is applied only to the current row of *screen array*. For example, the specification

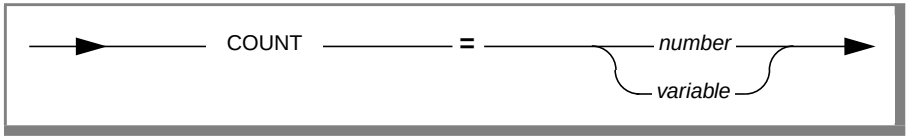
```
INPUT ARRAY p_items FROM s_items.*
    ATTRIBUTE (RED, CURRENT ROW DISPLAY = "GREEN, REVERSE")
```

displays *screen array* in red, but with the current row (the row that contains the screen cursor) in reverse video and green. When the cursor moves to another row, the previously highlighted row reverts to red, and the attributes list is applied to the new current row. If *screen array* has only one row, the CURRENT ROW DISPLAY attribute list is applied to that row.

If the quoted string includes no *keyword*, an error is issued.

The COUNT Attribute

The COUNT attribute can specify the number of records within a program array that contain data. It is valid only within the ATTRIBUTE clause of the INPUT ARRAY statement, where it has this syntax.



Element	Description
number	is a non-negative literal integer, specifying how many records in the program array contain data.
variable	is an INT or SMALLINT variable that contains the value of *number*.

The specification

```
COUNT = 5
```

is equivalent to the 4GL statement

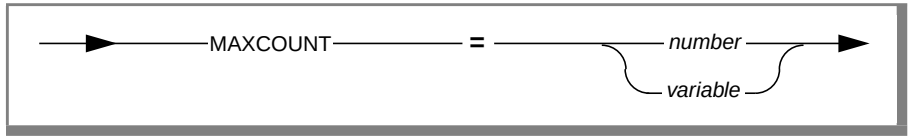
```
CALL SET_COUNT(5)
```

Both of these specifications restrict the number of screen records that can be displayed in the current screen array to 5.

You can use the COUNT attribute to control the screen array dynamically, as the next section illustrates.

The MAXCOUNT Attribute

The MAXCOUNT attribute can specify the dynamic size of a screen array. This size can be less than the declared size that the INSTRUCTIONS section of the **.per f**ile specifies for the screen array. MAXCOUNT is valid only within the ATTRIBUTE clause of the INPUT ARRAY statement. It has this syntax.



Element	Description
number	is a non-negative literal integer, specifying how many records in the screen array can display data.
variable	is an INT or SMALLINT variable that contains the value of *number*.

The following example of an INPUT ARRAY statement specifies both the MAXCOUNT and COUNT attributes:

```
INPUT ARRAY prog_array WITHOUT DEFAULTS
FROM scr_array.* ATTRIBUTE( MAXCOUNT = x, COUNT = y)
```

Here x and y are literal integers or integer variables. In this example, y is the number of records that contain data within the program array. The MAXCOUNT value of x determines the dynamic size of the screen array that displays the program array.

If MAXCOUNT is specified as less than one or greater than the declared program array size, the original program array size is used as the MAXCOUNT value.

You can specify both COUNT and MAXCOUNT in the same ATTRIBUTE clause:

```
CALL SET_COUNT(5)
INPUT ARRAY prog_array WITHOUT DEFAULTS
FROM scr_array.* ATTRIBUTE( MAXCOUNT = 10, COUNT = 6)
```

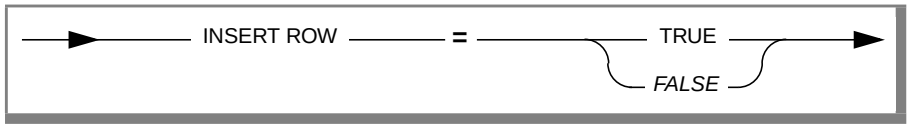
In this example, the COUNT attribute overrides the SET_COUNT( ) value. The number of rows displayed will be 6.

Except for the new COUNT and MAXCOUNT attributes, ATTRIBUTE lists of 4GL can only support fixed keywords or literal integers.

INSERT ROW Attribute

The ATTRIBUTE clause supports a feature by which the programmer can enable or disable the Insert key for the entire form during INPUT ARRAY statements.

The INSERT ROW attribute can be set to TRUE or FALSE in the ATTRIBUTE clause that follows the INPUT ARRAY binding clause. It has this syntax.



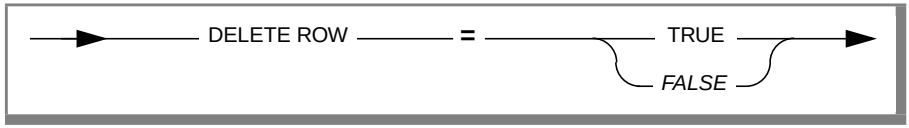
When INSERT ROW = FALSE is specified, the user cannot use the Insert key to perform insert actions within the INPUT ARRAY statement. (The user can still perform insert actions by using the TAB, ARROW, and RETURN keys in the last initialized row.)

When INSERT ROW = TRUE is specified, the user is not prevented from using the Insert key to enter data. The default is TRUE, which corresponds to the behavior of previous 4GL releases.

DELETE ROW Attribute

The DELETE ROW attribute provides similar functionality by which the programmer can enable or disable the Delete key for the entire form during INPUT ARRAY statements.

The DELETE ROW attribute can be set to TRUE or FALSE in the ATTRIBUTE clause that follows the INPUT ARRAY binding clause. It has this syntax.



When DELETE ROW = FALSE is specified, the user cannot perform any DELETE actions within the INPUT ARRAY statement.

When DELETE ROW = TRUE is specified, the user is not prevented from using the Delete key to delete data. The default is TRUE, which corresponds to the behavior of previous 4GL releases.

The following example disables the Insert and Delete keys on rows of the screen array:

```
INPUT ARRAY arrayname WITHOUT DEFAULTS FROM s_array.*
ATTRIBUTE(INSERT ROW = FALSE, DELETE ROW = FALSE)
```

The HELP Clause

The HELP clause specifies the number of a help message to display if the user presses the Help key while the screen cursor is in any field of the screen array. The default Help key is CONTROL-W, but you can assign a different key as the Help key by using the HELP KEY clause of the OPTIONS statement.

The following program fragment specifies help message 311 if the user requests help from any field in the **s_items** screen array:

```
INPUT ARRAY p_items FROM s_items.*
   HELP 311
```

You create help messages in an ASCII file whose filename you specify in the HELP FILE clause of the OPTIONS statement (see "The HELP FILE Option" on page 4-299). Use the **mkmessage** utility to create a compiled version of the help file. A runtime error occurs in the following situations:

- 4GL cannot open the help file.
- You specify a number that is not in the help file.
- You specify a number outside the range from -32,767 to 32,767.

The help message specified in your HELP clause applies to the entire INPUT ARRAY statement. To override this with field-level help messages, specify an ON KEY block (see "The ON KEY Block" on page 4-171) that invokes the INFIELD( ) operator and SHOW_HELP( ) function, as described in Chapter 5. If you do this, the messages must be displayed in a 4GL window within the 4GL screen, rather than in the separate Help window.

The INPUT ARRAY Input Control Blocks

Each INPUT ARRAY control block includes a statement block of at least one statement and an *activation clause* that specifies when to execute the statement block. An INPUT ARRAY control block can specify any of the following items:

- The statements to execute before or after visiting specific screen fields
- The statements to execute when the user presses a key sequence
- The statements to execute before or after the INPUT ARRAY statement
- The next field to which to move the screen cursor
- When to terminate execution of the INPUT ARRAY statement

The activation clause can specify any one of the following items:

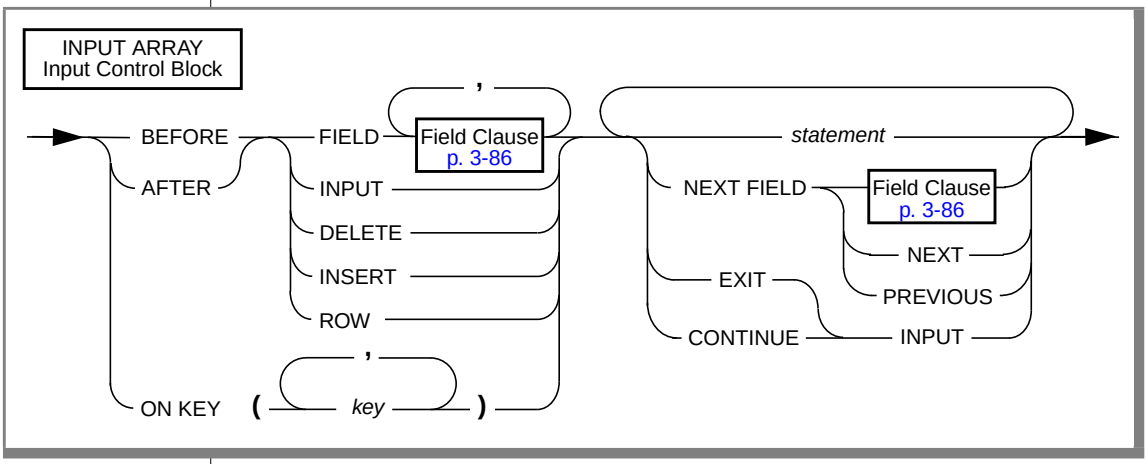
- Pre- and post-INPUT actions (the BEFORE or AFTER INPUT clause)
- Pre- and post-INSERT actions (the BEFORE or AFTER INSERT clause)
- Pre- and post-DELETE actions (the BEFORE or AFTER DELETE clause)
- Keyboard sequence conditions (the ON KEY clause)
- Cursor movement conditions (the BEFORE or AFTER FIELD clause, and the BEFORE or AFTER ROW clause)

The statement block can include any SQL or 4GL statements, as well as the following items:

- Cursor movement instructions (the NEXT FIELD or NEXT ROW clause)
- Termination of the INPUT ARRAY statement (the EXIT INPUT statement)
- Returning control to the user without terminating the INPUT ARRAY statement (the CONTINUE INPUT statement)

If you include one or more control blocks, the END INPUT keywords must terminate the INPUT ARRAY statement. If no control block is included, 4GL waits while the user enters values into the fields. When the user presses the Accept key, the INPUT ARRAY statement terminates.

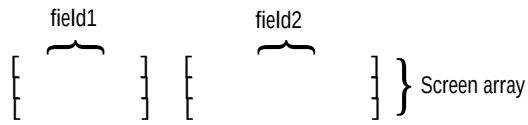
The activation clause and the statement block correspond respectively to the left-hand and right-hand elements in the following syntax diagram.



Element	Description
key	is one or more keywords to specify physical or logical keys. For details, see "The ON KEY Block" on page 4-171.
statement	is an SQL statement or other 4GL statement.

After BEFORE FIELD, AFTER FIELD, or NEXT FIELD, the field clause specifies a field that the binding clause referenced implicitly (in the BY NAME clause, or as *record*.* or *array* [*line*].*) or explicitly. You can qualify a field name by a table reference, or the name of a screen record or a screen array or *array* [*line*].

The BEFORE FIELD *screen-array* or AFTER FIELD *screen-array* activation clause applies to the entire screen array. BEFORE FIELD *screen-array.field* or AFTER FIELD *screen-array*. *field* applies to the specified field in the screen array, as in the following example, which represents part of a screen form.



If you specify BEFORE FIELD *screen-array*.**field1**, 4GL executes the statement block if the cursor moves into the **field1** field of any screen record of *screen-array*, but not after movement to a **field2** field. You can specify BEFORE FIELD *screen-array* if you want the statement block to be executed if the cursor enters any field of *screen-array*.

If you include a control block, 4GL executes or ignores the statements in a control block, depending on:

- whether you specify the BEFORE INPUT or AFTER INPUT keywords.
- the fields to which and from which the user moves the screen cursor.
- the keys that the user presses.

4GL deactivates the form while executing statements in a control block. After executing the statements, 4GL reactivates the form, allowing the user to continue entering or modifying the data values in fields.

The Precedence of Input Control Blocks

4GL executes the statements in control blocks in the following order:

1. BEFORE INPUT
2. BEFORE ROW
3. BEFORE INSERT, BEFORE DELETE
4. BEFORE FIELD *screen-array*
5. BEFORE FIELD *screen-array*. *field*
6. ON KEY
7. AFTER FIELD *screen-array*. *field*
8. AFTER FIELD *screen-array*
9. AFTER INSERT, AFTER DELETE
10. AFTER ROW
11. AFTER INPUT

These blocks are described in the sections that follow. You can list these blocks in any order. If you develop some consistent ordering, however, your code might be easier to read.

Within these blocks, you can include the NEXT FIELD keywords (as described in "The NEXT FIELD Keywords" on page 4-176) and EXIT INPUT statement (described in "The EXIT INPUT Statement" on page 4-178), as well as most 4GL and SQL statements. See "Nested and Recursive Statements" on page 2-31 for information about including CONSTRUCT, PROMPT, INPUT, and INPUT ARRAY statements within an input control block.

The activation clauses of INPUT ARRAY control blocks are described in their order of execution by 4GL. Descriptions of NEXT FIELD and EXIT INPUT follow the discussions of these activation clauses. No subsequent control block statements are executed if EXIT INPUT executes.

The BEFORE INPUT Block

You can use the BEFORE INPUT block to display messages describing how to use the INPUT ARRAY statement. For example, the following statement fragment displays a message that tells the user how to enter data into the table:

```
INPUT ARRAY p_customer FROM s_customer.*
   BEFORE INPUT
      DISPLAY "Press ESC to enter data" AT 1,1
```

4GL executes the BEFORE INPUT block after displaying the default values in the fields and before allowing the user to enter values. (If you include the WITHOUT DEFAULTS clause, 4GL displays the current values of the variables, not the default values, before executing the BEFORE INPUT block.)

The following example displays the value 2 in the **stock_num** field:

```
CALL SET_COUNT(1)
INPUT ARRAY p_items WITHOUT DEFAULTS FROM s_items.*
   BEFORE INPUT
      LET pa_curr = ARR_CURR()
      LET s_curr = SCR_LINE()
      LET p_items[pa_curr].stock_num = 2
      DISPLAY p_items[pa_curr].stock_num TO
         s_items[s_curr].stock_num
      NEXT FIELD manu_code
END INPUT
```

The following list describes how this program fragment uses the DISPLAY statement in the BEFORE INPUT block to populate the fields of a single screen array:

1. Call SET_COUNT(1) to initialize the array with a nondefault record.
2. Include WITHOUT DEFAULTS in the INPUT ARRAY binding clause.
3. Within the BEFORE INPUT block, use LET statements to assign values to the variables.
4. Use DISPLAY to display the variable to the screen.

An INPUT ARRAY statement can include no more than one BEFORE INPUT control block. You cannot include the FIELD_TOUCHED( ) operator in the BEFORE INPUT block.

The BEFORE ROW Block

Here ROW means a screen record; it need not be linked to a database row. You can specify no more than one BEFORE ROW block. 4GL executes the BEFORE ROW block statements in the following cases:

- The cursor moves into a new line of the screen form.
- An INSERT statement fails because of lack of space.
- An INSERT statement is terminated by the Interrupt or Quit key.
- The user presses the Delete key.

The BEFORE DELETE Block

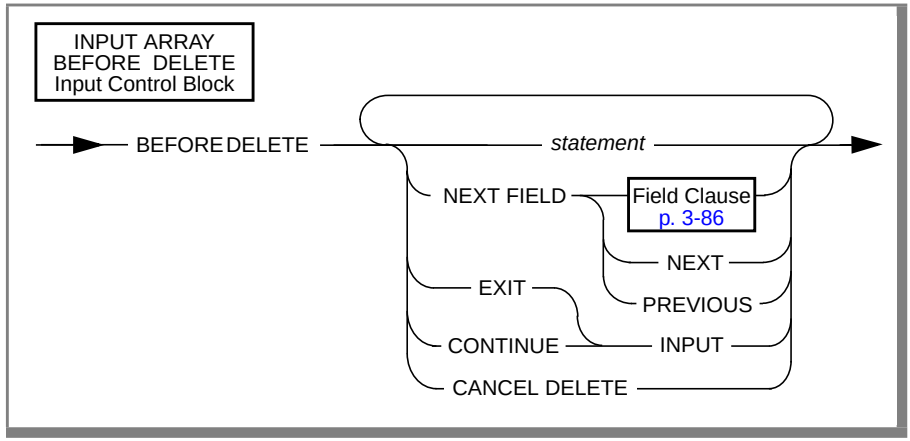
This statement block is executed after the user presses the Delete key while the cursor is in a screen array, but before 4GL actually deletes the record. You can specify no more than one BEFORE DELETE block.

If you want to prevent the record from being deleted (for example, if some Boolean condition is not satisfied), specify EXIT INPUT, rather than CONTINUE INPUT, within the BEFORE DELETE block.

Cancelling Delete Operations

If you include the CANCEL DELETE keywords within the BEFORE DELETE control block, delete operations by the user can also be cancelled programmatically for individual screen records of the current 4GL form.

The syntax of CANCEL DELETE within the BEFORE DELETE control block of INPUT ARRAY statements follows.



Element	Description
statement	is an SQL statement or other 4GL statement that is valid within a BEFORE DELETE control block of INPUT ARRAY.

The cancelled Delete operation has no effect on the active set of rows that INPUT ARRAY is processing.

See also "Cancelling Insert Operations" on page 4-204 for the parallel syntax of CANCEL INSERT within the BEFORE INSERT control block.

If CANCEL INSERT or CANCEL DELETE is executed, the current BEFORE INSERT or BEFORE DELETE control block is terminated, and control of program execution passes to the next statement that follows the terminated control block.

An error is issued if you specify CANCEL DELETE outside the context of the BEFORE DELETE control block.

Similarly, an error is issued if you specify CANCEL INSERT outside the context of the BEFORE INSERT control block.

As an example, the programmer might want to implement a system where the user is allowed to delete all but one of the rows, but once a row is deleted, a replacement row cannot be inserted in its place. The following code implements this design:

```
DEFINE n_rows INTEGER
         DEFINE arrayname ARRAY[100] OF RECORD
        . . .
         INPUT ARRAY arrayname WITHOUT DEFAULTS FROM s_array.*
             ATTRIBUTES(COUNT = n_rows, MAXCOUNT = n_rows,
                        INSERT ROW = FALSE, DELETE ROW = TRUE
         BEFORE INSERT
             CANCEL INSERT
         BEFORE DELETE
             LET n_rows = n_rows - 1
             IF n_rows <= 0 THEN
                 CANCEL DELETE
             END IF
         END INPUT
```

The BEFORE INSERT Block

Statements in the BEFORE INSERT block are executed in the following cases:

- The user begins entering new records into the array.
- The user presses the Insert key to insert a new record between existing records of a screen array, but before the record is added to the array.
- The user moves the cursor to a blank record at the end of an array.

4GL executes the statements in this block before the user enters data for each successive screen record that the Insert key creates.

The following BEFORE INSERT block calls the **get_item_num( )** function before inserting a new empty record into the screen array:

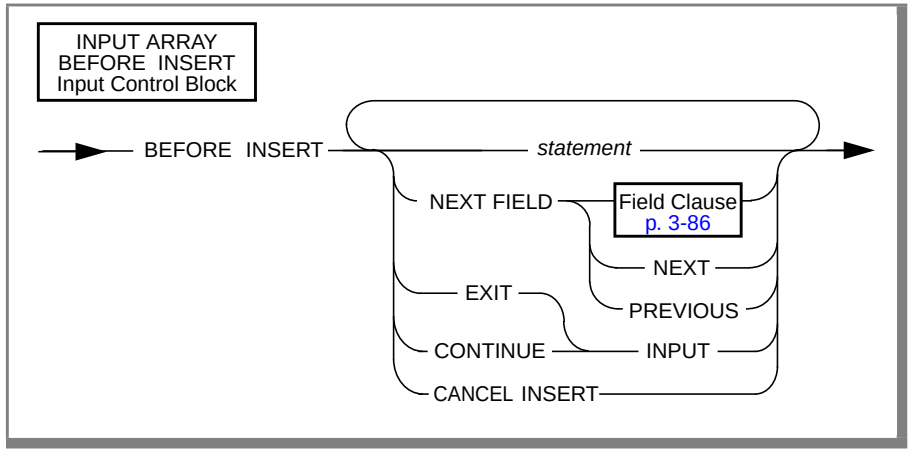
```
BEFORE INSERT
   CALL get_item_num()
```

You can specify no more than one BEFORE INSERT block.

Cancelling Insert Operations

You can include the CANCEL INSERT keywords within the BEFORE INSERT control block to programmatically cancel insert operations by the user for individual screen records of the current 4GL form. The cancelled insert operation has no effect on the active set of rows that INPUT ARRAY is processing.

The syntax of CANCEL INSERT within the BEFORE INSERT control block of INPUT ARRAY statements follows.



Element	Description
statement	is an SQL statement or other 4GL statement that is valid within a BEFORE INSERT control block of INPUT ARRAY.

If CANCEL INSERT is specified, the user is prevented from entering rows by using the Insert key. This feature also prevents the user from entering rows by using an arrow key, TAB, RETURN, or (in Dynamic 4GL) ENTER to move the screen cursor past the last initialized row.

The following example disables the Insert key for only the third row:

```
INPUT ARRAY ...
BEFORE INSERT
        IF ARR_CURR() == 3
        THEN
                CANCEL INSERT
        END IF
END INPUT
```

The BEFORE FIELD Block

This statement block is executed whenever the screen cursor moves into the specified field, but before the user enters a value. You can specify no more than one BEFORE FIELD block for each field.

The following program fragment defines two BEFORE FIELD blocks. When the cursor enters the **fname** or **lname** field, 4GL displays a message:

```
BEFORE  FIELD  fname
   MESSAGE  "Enter  first  name  of  customer"
BEFORE  FIELD  lname
   MESSAGE  "Enter  last  name  of  customer"
```

You can use a NEXT FIELD clause within a BEFORE FIELD block to restrict access to a field. You can also use a DISPLAY statement within a BEFORE FIELD block to display a default value in a field.

The following statement fragment causes 4GL to prompt the user for input when the cursor is in the **stock_num**, **manu_code**, or **quantity** field:

```
INPUT ARRAY p_items FROM s_items.*
   BEFORE FIELD stock_num
      MESSAGE "Enter a stock number."
   BEFORE FIELD manu_code
      MESSAGE "Enter the code for a manufacturer."
   BEFORE FIELD quantity
      MESSAGE "Enter a quantity."
   ...
END INPUT
```

The ON KEY Block

Statements in the ON KEY block are executed if the user presses some key that you specify by the keywords in the following table (in lowercase or uppercase letters).

ACCEPT	HELP	NEXT or NEXTPAGE
DELETE	INSERT	PREVIOUS or PREVPAGE
DOWN	INTERRUPT	RETURN
ESC or ESCAPE	LEFT	TAB
F1 through F64	RIGHT	UP
CONTROL-*char* (except A, D, H, I, J, K, L, M, R, or X)		

The list of keys that can activate the ON KEY control block must be enclosed in parentheses, with commas separating the names of keys.

The next example defines an ON KEY block for CONTROL-B. When the user presses CONTROL-B, 4GL determines if the screen cursor is in the **stock_num** or **manu_code** field. If it is in either one of these fields, 4GL calls the **stock_help( )** function and sets **quantity** as the next field.

```
INPUT ARRAY p_items FROM s_items.*
   ON KEY (CONTROL-B)
      IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
         CALL stock_help()
         NEXT FIELD quantity
      END IF
```

The following ON KEY block displays a help message. The BEFORE INPUT clause informs the user how to access help:

```
BEFORE INPUT
   DISPLAY "Press CTRL-W for help"
ON KEY (CONTROL-W, CONTROL-F)
   CALL customer_help()
```

Keys in the following table require special consideration in an ON KEY block.

Key	Special Considerations
ESC or ESCAPE	You must specify another key as the Accept key in the OPTIONS statement, because this is the default Accept key.
Interrupt	You must execute a DEFER INTERRUPT statement. If the user presses the Interrupt key under these conditions, 4GL executes the statements in the ON KEY block and sets **int_flag** to non-zero, but does not terminate the INPUT statement.
Quit	4GL also executes the statements in this ON KEY block if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, 4GL sets **quit_flag** to non-zero.
F1	You must specify another key as the Insert key in the OPTIONS statement because CONTROL-W is the default Insert key.
F2	You must specify another key as the Delete key in the OPTIONS statement, because F2 is the default Delete key.
F3	You must specify another key as the Next Page key in the OPTIONS statement, because F3 is the default Next Page key.
F4	You must specify another key as the Previous Page key in the OPTIONS statement, because F4 is the default for that key.

(1 of 2)

Key	Special Considerations
CONTROL-*char*	
A, D, H, K, L, R, and X	4GL reserves these keys for field editing.
I, J, and M	The standard meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is not available to the user. Instead, the key is trapped by 4GL and used to activate the ON KEY block. For example, if CONTROL-M appears in an ON KEY block, the user cannot press RETURN to advance the cursor to the next field. If you include one of these keys in an ON KEY block, be careful to restrict the scope of the block to specific fields.
W	This is the default Help key, so use OPTIONS to declare another.

(2 of 2)

You might not be able to use other keys that have special meaning to your operating system, such as CONTROL-Z on many BSD UNIX systems. Similarly, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many UNIX systems.

If you use the OPTIONS statement to redefine the Accept or Help key, the keys assigned to these sequences cannot be used in an ON KEY clause. For example, if you redefine the Accept key by using the following statement, you should not define an ON KEY block for the key sequence CONTROL-B:

```
OPTIONS ACCEPT KEY (CONTROL-B)
```

When the user presses CONTROL-B, 4GL will always perform the Accept key function, regardless of the presence of an ON KEY (CONTROL-B) block.

If the user activates an ON KEY block while entering data in a field, 4GL takes the following actions:

1. Suspends input to the current field
2. Preserves the input buffer containing characters that the user typed
3. Executes the statements in the current ON KEY block

4. Restores the input buffer for the current screen field
5. Resumes input in the same field, with the screen cursor at the end of the buffered list of characters

You can change this default behavior by performing the following tasks in the ON KEY block:

- Resuming input in another field by using the NEXT FIELD statement
- Changing the input buffer value for the current field by assigning a new value to the corresponding variable and then displaying this value

You can also use this block to provide *accelerator keys* for common functions, such as saving and deleting. The INFIELD( ) operator can control field-specific responses in the action for an ON KEY clause. You can implement field-level help by using the INFIELD( ) operator and SHOWHELP( ) function.

The AFTER FIELD Block

4GL executes the statements in the AFTER FIELD block associated with a field every time the cursor leaves the field. Any of the following keys can cause the cursor to leave the field:

- Any arrow key
- The RETURN or TAB key
- The Accept key
- The Interrupt or Quit key (if a supporting DEFER statement was executed)

You can specify only one AFTER FIELD block for each field.

This AFTER FIELD block checks if the **stock_num** and **manu_code** fields contain values. If they contain values, 4GL calls the **get_item( )** function:

```
AFTER FIELD stock_num, manu_code
   LET pa_curr = ARR_CURR()
   IF p_items[pa_curr].stock_num IS NOT NULL
      AND p_items[pa_curr].manu_code IS NOT NULL THEN
         CALL get_item()
         IF p_items[pa_curr].quantity IS NOT NULL THEN
            CALL get_total()
         END IF
   END IF
```

The following statement makes sure that the user enters an address line:

```
INPUT ARRAY p_addr FROM sc_addr.*
   AFTER FIELD address1
      IF p_addr.address1 IS NULL THEN
         NEXT FIELD address1
      END IF
END INPUT
```

The user terminates the INPUT ARRAY statement by pressing the Accept key when the screen cursor is in any field, or by pressing RETURN or TAB after the *last* field. You can use the AFTER FIELD block on the last field to override this default termination. (Including INPUT WRAP in the OPTIONS statement produces the same effect.)

When the NEXT FIELD keywords appear in an AFTER FIELD block, the cursor moves to the specified field. If an AFTER FIELD block appears for each field, and the NEXT FIELD keywords are in each block, the user cannot leave the form.

The AFTER INSERT Block

This block has no effect unless the BY NAME or FROM clause references a screen array. 4GL executes the AFTER INSERT block after the user inserts a record into the screen array. A user inserts a record by following these steps:

1. Entering information in all the required fields of the current record
2. Moving the cursor out of the last input field by using one of these keys:
   - Any arrow key
   - The RETURN or TAB key
   - The Accept key
   - The HOME or END key

Tip: *The Insert key does not by itself activate the AFTER INSERT block; the user must also move the cursor from the newly inserted record.*

The following AFTER INSERT block calls the **renum_items()** function after the user inserts a new blank screen record into the **items** screen array:

```
AFTER INSERT OF items
   CALL renum_items()
```

An INPUT statement can include only one AFTER INSERT block.

The AFTER DELETE Block

4GL executes the AFTER DELETE block after the user deletes the values from a screen record by using the Delete key. If this block is present, 4GL takes the following actions when the user presses the Delete key:

1. Deletes the record from the screen array
2. Executes the statements in the AFTER DELETE block
3. Executes the statements in the AFTER ROW block, if one is specified

The user must also press the Accept key to make corresponding changes to the variables in the array of program records. The following AFTER DELETE block calls the **renum_items( )** function:

```
AFTER DELETE OF items
   CALL renum_items()
```

An INPUT statement can include only one AFTER DELETE block.

The AFTER ROW Block

Here ROW means a screen record; this need not be linked to a database row. 4GL executes the statements in the AFTER ROW block in these cases:

- The cursor leaves the current row by using one of these keys:
  - Any arrow key
  - The RETURN or TAB key
  - The Accept key
  - The Interrupt key (if DEFER INTERRUPT was also executed)
- A new screen record is inserted by the Insert key.

The INPUT ARRAY statement can specify only one AFTER ROW block. If you specify both an AFTER ROW and an AFTER INSERT block, 4GL executes the AFTER ROW block immediately after executing the AFTER INSERT block.

The following AFTER ROW block calls the **order_total( )** function after the screen cursor leaves a row, and the row is inserted:

```
AFTER ROW
   CALL order_total()
```

If you include a NEXT FIELD statement in an AFTER ROW block, 4GL moves the cursor to the next field of the next row, not to the row which the cursor has just left. For example, if values in the fields in that row are in conflict, the developer could detour the user back to the conflicting fields before allowing the INPUT statement to complete, by having conditional NEXT FIELD statements in the AFTER INPUT block. For example:

```
INPUT ARRAY p_items from s_items.*
   ...
   AFTER ROW
   LET pa_curr = arr_curr()
      IF p_items[pa_curr].manu_code = "PNG"  THEN
         MESSAGE "NOTE: PNG products are currently on hold"
         NEXT FIELD manu_code
      END IF
   ...
END INPUT
```

The NEXT FIELD statement in the AFTER ROW control block, if executed, now keeps the user entry in the current row, with the cursor in the **manu_code** field, regardless of what navigation key you use to leave that row (for example UP ARROW, DOWN ARROW, TAB, RETURN, or ACCEPT).

The AFTER INPUT Block

The statements in the AFTER INPUT block are executed when the user terminates the INPUT ARRAY statement without terminating the 4GL program.

4GL executes the AFTER INPUT block only when the INPUT ARRAY statement is terminated by the user pressing one of the following keys:

- The Accept key
- The Interrupt key (if the DEFER INTERRUPT statement has executed)
- The Quit key (if the DEFER QUIT statement has executed)

The AFTER INPUT clause is not executed in the following situations:

- The user presses the Interrupt or Quit key and the DEFER INTERRUPT or DEFER QUIT statement, respectively, has not executed. In either case, the program terminates immediately.
- The EXIT INPUT statement terminates the INPUT ARRAY statement.

By using the GET_FLDBUF( ) or FIELD_TOUCHED( ) built-in operators within the AFTER INPUT block, you can use the AFTER INPUT block to validate, save, or alter values that the user entered.

The NEXT FIELD statement in the AFTER INPUT control block gives the 4GL developer the ability to prevent the user from completing the INPUT statement if some programmer-defined semantic criteria are not satisfied. The following example uses this block to require that a first name be specified for any customers with the last name Smith:

```
CALL SET_COUNT(1)
INPUT ARRAY p_customer FROM sc_customer.*
   AFTER INPUT
      IF p_customer.lname="Smith" THEN
         IF NOT FIELD_TOUCHED(p_customer.fname) THEN
            CALL mess("You must enter a first name.")
            NEXT FIELD fname
         END IF
      END IF
   END INPUT
```

You can place the NEXT FIELD clause in this block to return the cursor to the form. If you place a NEXT FIELD clause in the AFTER INPUT block, use it in a conditional statement. Otherwise, the user cannot exit from the form.

You can include no more than one AFTER INPUT control block.

The NEXT FIELD Keywords

The NEXT FIELD keywords specify the next field to which 4GL moves the screen cursor. If you omit this clause, by default the cursor moves among the screen fields according to the explicit or implicit order of fields in the INPUT ARRAY binding clause. The user can control movement from field to field by using the arrow keys, TAB, and RETURN. By using the NEXT FIELD keywords, however, you can explicitly position the screen cursor. You must specify one of the following options with the NEXT FIELD keywords.

Clause	Effect
NEXT FIELD NEXT	Advances the screen cursor to the next field
NEXT FIELD PREVIOUS	Returns the screen cursor to the previous field
NEXT FIELD *field-name*	Moves the screen cursor to *field-name*

For example, this NEXT FIELD clause places the cursor in the previous field:

```
NEXT FIELD PREVIOUS
```

The following INPUT ARRAY statement includes a NEXT FIELD clause in an ON KEY block. If the user presses CONTROL-B when the screen cursor is in the **stock_num** or **manu_code** field, 4GL sets **quantity** as the next field:

```
INPUT ARRAY p_items FROM s_items.*
   ON KEY (CONTROL-B)
      IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
         CALL stock_help()
         NEXT FIELD quantity
      END IF
   ...
END INPUT
```

In releases of 4GL prior to 6.03, the NEXT FIELD statement was ignored except to choose the active field in the target row. If the ACCEPT key was used, the NEXT FIELD was ignored because the INPUT ARRAY statement had been completed (except for AFTER INPUT processing, if any).

In most situations, the NEXT FIELD should appear in a conditional statement. The NEXT FIELD statement must appear in a conditional statement when it appears in an AFTER INPUT clause; otherwise, the user cannot exit the form.

4GL immediately positions the screen cursor in the form when it encounters the NEXT FIELD clause; it does not execute any statements that follow the NEXT FIELD clause in the control block. For example, 4GL does not invoke the **qty_help( )** function in the next example:

```
ON KEY (CONTROL-B, F4)
   IF INFIELD(stock_num) OR infield(manufact) THEN
      CALL stock_help()
      NEXT FIELD quantity
      CALL qty_help() -- function is never called
   END IF
```

You can use the NEXT FIELD clause in any INPUT ARRAY control block. The NEXT FIELD clause typically appears in a conditional statement. In an AFTER INPUT clause, the NEXT FIELD statement *must* appear in a conditional statement; otherwise, the user cannot exit from the form. To restrict access to a field, use the NEXT FIELD statement in a BEFORE FIELD clause.

The following example demonstrates using the NEXT FIELD clause in an ON KEY control block, which 4GL executes if the user presses CONTROL-W. If the cursor is in the **city** field, 4GL displays San Francisco in the **city** field and CA in the **state** field, and then moves the cursor to the **zipcode** field.

```
ON KEY (CONTROL-W)
   IF INFIELD(city) THEN
      LET p_addr.city = "San Francisco"
      DISPLAY p_addr.city TO city
      LET p_addr.state = "CA"
      DISPLAY p_addr.state TO state
      NEXT FIELD zipcode
   END IF
```

To wrap from the last field of a form to the first field of a form, use the NEXT FIELD statement after an AFTER FIELD clause for the last field of the form. (The INPUT WRAP option of the OPTIONS statement has the same effect.)

The CONTINUE INPUT Statement

The CONTINUE INPUT statement causes 4GL to skip all subsequent statements in the current control block. The screen cursor returns to the most recently occupied field in the current form.

The CONTINUE INPUT statement is useful when program control is nested within multiple conditional statements, and you want to return control to the user. It is also useful in an AFTER INPUT control block that examines the field buffers; depending on their contents, you can return the cursor to the form.

The EXIT INPUT Statement

The EXIT INPUT statement terminates input. 4GL does the following:

- Skips all statements between EXIT INPUT and END INPUT
- Deactivates the form
- Resumes execution at the first statement after END INPUT

4GL ignores any statements in an AFTER INPUT control block if the EXIT INPUT ARRAY statement is executed.

The END INPUT Keywords

The END INPUT keywords indicate the end of the INPUT ARRAY statement. These keywords should follow the last control block. If you do not include any control blocks, the END INPUT keywords are not required.

Using Built-In Functions and Operators

You can use the following built-in functions to keep track of the relative states of the screen cursor, the program array, and the screen array.

Function	Description
ARR_CURR( )	Returns the number of the current record of the program array. This indicates the position of the screen cursor at the beginning of the BEFORE or AFTER ROW control block, rather than the line to which the cursor moves after execution of the block.
ARR_COUNT( )	Returns the current number of records in the program array.
FGL_SCR_SIZE( )	Returns an INTEGER value corresponding to the declared number of screen records in a specified screen array in the currently active form.
SCR_LINE( )	Returns the number of the current line in the screen array. This is the line containing the screen cursor at the beginning of the BEFORE ROW or AFTER ROW control block, rather than the line to which the cursor moves after execution of the block. This can be different from the value returned by ARR_CURR( ) if the program array is larger than the screen array.
SET_COUNT( )	Takes the number of records currently in the program array as an argument and sets the initial value of ARR_COUNT( ). You must call this function before executing the INPUT ARRAY WITHOUT DEFAULTS or DISPLAY ARRAY statement.

These functions and operators access field buffers and keystroke buffers.

Feature	Description
FIELD_TOUCHED( )	Returns TRUE when the user has "touched" (made a change to) a screen field whose name is passed as an operand. Moving the screen cursor through a field (with the RETURN, TAB, or arrow keys) does not mark a field as touched. This function also ignores the effect of statements that appear in the BEFORE INPUT control block. For example, you can assign values to fields in the BEFORE INPUT control block without having the fields marked as touched.
GET_FLDBUF( )	Returns the character values of the contents of one or more fields in the currently active form.
FGL_GETKEY( )	Waits for a key to be pressed, and then returns an INTEGER value corresponding to the raw value of the key that was pressed.
FGL_LASTKEY( )	Returns an INTEGER value corresponding to the most recent keystroke executed by the user in the screen form.
INFIELD( )	Returns TRUE if the name of the field that is specified as its operand is the name of the current field.

Each field has only one field buffer; two statements cannot use a buffer simultaneously. To display the same form with data entry fields more than once, open a new 4GL window, and open and display a second copy of the form. (4GL allocates a separate set of buffers to each form, so this avoids overwritting buffers when two or more concurrent statements accept input.)

The following statement uses the INFIELD( ) operator to determine if the cursor is in the **stock_num** or **manu_code** field. If the cursor is in one of these fields, 4GL calls the **stock_help( )** function and sets **quantity** as the next field:

```
INPUT ARRAY p_items FROM s_items.*
   ON KEY (CONTROL-B)
      IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
         CALL stock_help()
         NEXT FIELD quantity
      END IF
```

The INFIELD(*field*) expression returns TRUE if the current field is *field* and FALSE otherwise. You can use this function to control field-dependent actions when the user presses a specified key in the ON KEY control block. In the following INPUT ARRAY statement, the BEFORE FIELD control block for the **city** field displays a message identifying a key that the user can press to enter the character string "San Francisco" into the field:

```
INPUT ARRAY pr_customer FROM sc_cust.* ATTRIBUTE(REVERSE)
   BEFORE FIELD city
      MESSAGE "Press CTRL-F for default city, San Francisco"
   ON KEY (CONTROL-F)
      IF INFIELD(city) THEN
         LET p_customer.city = "San Francisco"
         DISPLAY p_customer.city TO city
         LET p_customer.state = "CA"
         DISPLAY p_customer.state TO state
         NEXT FIELD zipcode
      END IF
END INPUT
```

If the user presses CONTROL-F while the cursor is in the **city** field, the ON KEY clause in this example changes the screen display in three ways:

1. Displays the value San Francisco in the **city** field
2. Displays CA in the **state** field
3. Moves the cursor to the first character position in the **zipcode** field

For more about the built-in 4GL functions and operators, see Chapter 5.

Using Large Data Types

Within a field of a screen array, 4GL displays any value of a large data type (BYTE or TEXT) in the following way.

Type	Screen Display
TEXT	As much of the TEXT data as can fit in the screen field
BYTE	The string `<BYTE value>` (4GL cannot display the data value in a field.)

If the form specification file assigns an appropriate attribute to a BYTE or TEXT field, the user can invoke an external program by pressing the exclamation point (!) key when the cursor is in the field. This external program is typically an editor that allows the user to edit large string (TEXT) or graphics (BYTE) data. To implement this feature, specify the PROGRAM attribute as part of the field description in the form specification file, identifying the external program to execute. (For more information on using the PROGRAM attribute, see the description of that field attribute in Chapter 6.)

The external program takes over the entire screen. Any key sequence that you have specified in the ON KEY clause is ignored by the external program. When the external program terminates, 4GL takes the following actions:

1. Restores the screen to its state before the external program began
2. Resumes the INPUT statement at the BYTE or TEXT field
3. Reactivates any key sequences specified in the ON KEY clause

Keyboard Interaction

The user of your 4GL application can use the keyboard to position the cursor during an INPUT ARRAY statement, to scroll the screen array, and to edit data in screen records.

By default, the user can move the cursor within a screen array and scroll the displayed rows by clicking the arrow keys, the PAGE UP or PAGE DOWN key, and the F3 and F4 function keys. The following table describes these keys.

Key	Effect
→	RIGHT ARROW moves the cursor one space to the right inside a screen field without erasing the current character. At the end of the field, 4GL moves the cursor to the first character position of the next screen field. This key is equivalent to the CONTROL-L editing key.
←	LEFT ARROW moves the cursor one character position to the left in a screen field without erasing the current character. At the end of the field, the cursor moves to the first character position of the previous screen field. This key is equivalent to the CONTROL-H editing key.
↓	DOWN ARROW moves the cursor to the same display field one line down on the screen. If the cursor was on the last line of the screen array before DOWN ARROW was used, 4GL scrolls the program array data up one line. If the last program array record is already on the last line of the screen array, DOWN ARROW generates a message indicating that there are "no more rows in that direction."
↑	UP ARROW moves the cursor to the same field one line up on the screen. If the cursor is on the first line of the screen array, 4GL scrolls the program array data down one line. If the first program array record is already on the first screen array line, UP ARROW generates a message indicating that there are "no more rows in that direction."
F3	F3 scrolls the display to the next full page of program records. The NEXT KEY clause of the OPTIONS statement can reset this key.
F4	F4 scrolls the display to the previous full page of program records. The PREVIOUS KEY clause of the OPTIONS statement can reset this key.

Clearing Reserved Lines

When moving the cursor to a new field of an array, the INPUT ARRAY statement clears the Comment line and the Error line. The Comment line displays text defined with the COMMENTS attribute in the form specification file. The Error line displays system error messages and ERROR statement text.

Editing Keys

Unless a field has the NOENTRY attribute, the user can press the following keys during an INPUT ARRAY statement to edit values in a field.

Key	Effect
CONTROL-A	Toggles between insert and type-over mode
CONTROL-D	Deletes characters from current cursor position to the end of the field
CONTROL-H	Moves the cursor nondestructively one space to the left. It is equivalent to pressing left arrow
CONTROL-L	Moves the cursor nondestructively one space to the right. It is equivalent to pressing right arrow
CONTROL-R	Redisplays the screen
CONTROL-X	Deletes the character beneath the cursor

Inserting and Deleting Records from an Array

The user can insert and delete records within screen arrays by pressing CONTROL-W (the default Insert key) and F2 (the default Delete key).

Key	Effect
CONTROL-W	Inserts a new blank screen record into the screen array at the line below the cursor. Any displayed values in lower records move down one line, and the cursor moves to the beginning of the first field of the new blank record. This key is not needed to insert rows at the end of the screen array. If the user attempts to insert more rows than the declared size of the program array, 4GL displays a message that the array is full. The OPTIONS statement can specify a different physical key as the Insert key.
F2	Deletes the current record from the screen array. 4GL adjusts any subsequent rows to fill the gap. The OPTIONS statement can specify a different physical key as the Delete key.

Pressing the Accept key makes corresponding changes in the program array. You can then use the ARR_COUNT( ) function to determine how many records (possibly including blank records) remain in the program array after the user has pressed the Insert or Delete key and the Accept key.

See also "The BEFORE DELETE Block" on page 4-201, "The BEFORE INSERT Block" on page 4-203, "The AFTER INSERT Block" on page 4-209, and "The AFTER DELETE Block" on page 4-210.

Completing the INPUT ARRAY Statement

The following actions can terminate the INPUT ARRAY statement:

- The user presses one of the following keys:
  - The Accept, Interrupt, or Quit key
  - The RETURN or TAB key from the last field (and INPUT WRAP is not currently set by the OPTIONS statement)
- 4GL executes the EXIT INPUT statement.

All of these conditions deactivate the form. Unlike the INPUT statement, the INPUT ARRAY statement is not terminated when the user presses the RETURN or TAB key in the last screen field.

By default, the Accept, Interrupt, and Quit keys terminate the INPUT ARRAY statement. Each of these actions also deactivates the form. (But pressing the Interrupt or Quit key can immediately terminate the program, unless the program also includes the DEFER INTERRUPT and DEFER QUIT statements.)

The user must press the Accept key explicitly to complete the INPUT ARRAY statement under the following conditions:

- INPUT WRAP is specified in the OPTIONS statement.
- An AFTER FIELD block for the last field includes a NEXT FIELD clause.

If 4GL previously executed a DEFER INTERRUPT statement in the program, an Interrupt signal causes 4GL to take the following actions:

- Sets the global variable **int_flag** to a non-zero value
- Terminates the INPUT ARRAY statement but not the 4GL program

If 4GL previously executed a DEFER QUIT statement in the program, a Quit signal causes 4GL to take the following actions:

- Sets the global variable **quit_flag** to a non-zero value
- Terminates the INPUT ARRAY statement but not the 4GL program

Executing Control Blocks when INPUT ARRAY Terminates

When INPUT ARRAY terminates, control blocks are executed in this order:

1. The AFTER FIELD clause for the current field
2. The AFTER ROW clause
3. The AFTER INPUT clause

If INPUT ARRAY is terminated by the EXIT INPUT keywords, or by pressing the Interrupt or Quit key, 4GL does not execute any of these clauses. If a NEXT FIELD statement appears in one of these clauses, 4GL places the cursor in the specified field and returns control to the user.

The INPUT ARRAY statement in the example that follows supports data entry into a screen form.

The BEFORE FIELD clauses display messages telling the user what to enter in the **stock_num**, **manu_code**, and **quantity** fields. The AFTER FIELD clauses check that user entered values for the **stock_num**, **manu_code**, and **quantity** fields. When the user enters item values for the **stock_num** and **manu_code** fields, 4GL calls **get_item( )** to display a description and price of the item. When all three fields are specified, 4GL displays the total cost.

In this example, the BEFORE INSERT, AFTER INSERT, and AFTER DELETE clauses call functions that ensure that the numbering of the items is accurate. Accurate numbering is necessary because the user can press the Insert and Delete keys at runtime to insert and to delete items within the screen form.

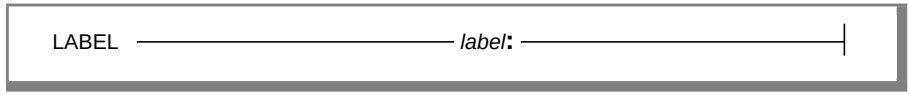
```
CALL SET_COUNT(1)
INPUT ARRAY p_items FROM s_items.*
   BEFORE FIELD stock_num
      MESSAGE "Enter a stock number."
   BEFORE FIELD manu_code
      MESSAGE "Enter the code for a manufacturer."
   BEFORE FIELD quantity
      MESSAGE "Enter a quantity"
   AFTER FIELD stock_num, manu_code
      MESSAGE ""
      LET pa_Curr = arr_curr()
      IF p_items[pa_curr].stock_num IS NOT NULL
         AND p_items[pa_curr].manu_code IS NOT NULL THEN
            CALL get_item()
            IF p_items[pa_curr].quantity IS NOT NULL THEN
               CALL get_total()
            END IF
      END IF
   AFTER FIELD quantity
      MESSAGE ""
      LET pa_curr = arr_curr()
      IF p_items[pa_curr].stock_num IS NOT NULL
         AND p_items[pa_curr].manu_code IS NOT NULL
         AND p_items[pa_curr].quantity IS NOT NULL THEN
            CALL get_total()
      END IF
   BEFORE INSERT
      CALL get_item_num()
   AFTER INSERT
      CALL renum_items()
   AFTER DELETE
      CALL renum_items()
END INPUT
```

References

DEFER, DISPLAY ARRAY, INPUT, OPEN WINDOW, OPTIONS, SCROLL

LABEL

The LABEL statement declares a *statement label*, marking the next statement as one to which a WHENEVER or GOTO statement can transfer program control.



Element	Description
label	is a statement label. A colon (:) symbol follows the last character.

Usage

The LABEL statement indicates where to transfer control of program execution within the same program block. Upon executing a GOTO or WHENEVER statement that references the *label* identifier, 4GL jumps to the statement immediately following the LABEL statement, skipping any intervening statements. (See also "GOTO" on page 4-151.)

The following restrictions apply to the LABEL statement:

- The identifier must be unique among labels in the program block.
- To jump to a label, the GOTO or WHENEVER statement must specify the same *label* identifier as the LABEL statement.
- The GOTO (or WHENEVER) and LABEL statements must both be in the same MAIN, FUNCTION, or REPORT program block.

A colon (:) symbol must follow the last character in the *label* identifier. This syntax contrasts with GOTO, where the colon is optional, and with WHENEVER, where the colon precedes the identifier but is not required. You might wish to declare a meaningful name to indicate something about the purpose of the jump:

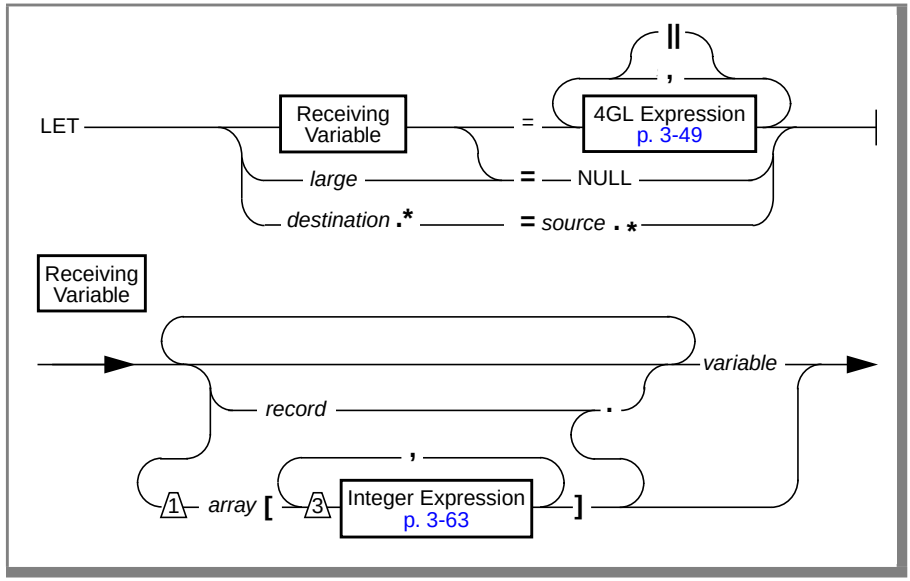
```
WHENEVER ERROR GO TO :l_error
   ...
LABEL l_error:
ERROR "Cannot complete processing."
ROLLBACK WORK
```

References

GOTO, WHENEVER

LET

The LET statement assigns a value to a variable, or a set of values to a record.



Element	Description
array	is a variable of the ARRAY data type.
destination	is a program record to be assigned values.
large	is a variable of the BYTE or TEXT data type.
record	is a variable of the RECORD data type.
source	is a program record from which to copy values.
variable	is a variable of a simple data type, a simple member of a record, or a simple element of an array.

Usage

After DEFINE declares a variable, the 4GL compiler allocates memory to that variable, and (for RDS) initializes the value to NULL. For the 4GL C Compiler, however, the contents of the variable is whatever happens to occupy that memory location.

Do not use uninitialized variables in 4GL expressions. If you reference any variable without first initializing it with an INITIALIZE or LET statement, the results are unlikely to be useful. The LET statement can assign a single value to a single variable, or it can assign all the values from a RECORD variable to another program record.

To execute a LET statement, 4GL evaluates the expression on the right of the equal (=) sign and assigns the resulting value to the variable on the left. For example, the statements in the following example create a SELECT statement as text for a PREPARE statement. Here slash (/) embeds a quotation (") mark within a string; the comma (,) and double pipe (||) symbols concatenate successive elements of the statement text:

```
DEFINE sel_stmt CHAR(80)
LET sel_stmt = "SELECT * FROM customer" ||
     "WHERE lname MATCHES \"" || last_name CLIPPED, "\""
PREPARE s1 FROM sel_stmt
```

The next example assigns a NULL value to a MONEY variable:

```
DEFINE total_price MONEY
LET total_price = NULL
```

You can use most of the 4GL built-in functions and character operators like CLIPPED and USING within the LET statement. For example, these statements use the ASCII operator to ring the terminal bell (= the ASCII value for 7):

```
DEFINE bell CHAR(1)
LET bell = ASCII 7
DISPLAY bell
```

You cannot assign individual values to an entire program record or to a program array. You cannot use the THRU or THROUGH notation (described in "THRU or THROUGH Keywords and .* Notation" on page 3-92) in the LET statement, but you can assign all the values of one program record to another program record of the same size by using the asterisk (*) notation:

```
LET  x.*  =  y.*
```

This example copies the value of each member of the **y** record to consecutive members of the **x** record. The two records must have the same number of members, and corresponding members must be of compatible data types. (See "Summary of Compatible 4GL Data Types" on page 3-46.) No member of the record can be of the BYTE or TEXT data types.

To reference substrings of CHAR or VARCHAR variables, specify the starting and ending character positions as integers. These substring positions must be enclosed within brackets and separated by a comma, as in this example:

```
DEFINE full_name CHAR(20), first_name CHAR(10)
LET full_name[1,10] = new_first
```

For TEXT and BYTE variables, LET can assign only NULL values. To assign other values to TEXT and BYTE variables, you can take one of the following actions:

- Use the INTO clause of the SELECT, FOREACH, OPEN, or FETCH statement
- Pass the name of the variable as an argument to a function

The Comma and Double-Pipe List Separator Symbols

A comma (,) between values to the right of the equal (=) sign has concatenation semantics. Unlike the double pipe (||) concatenation operator, however, a NULL value in a comma-separated list has no effect, unless every value in the list is NULL (in which case, the list evaluates to a single NULL). In contrast, the || operator returns NULL if any operand is NULL; this behavior conforms to the ANSI/ISO standard for SQL in concatenating NULL strings.

The following program illustrates the difference between these separators:

```
MAIN
   DEFINE c1, c2, c3 CHAR(5)
   DEFINE c4, c5 CHAR(14)
   LET c1 = "SOME"
   LET c2 = NULL
   LET c3 = "THING"
   LET c4 = c1, c2, c3
   LET c5 = c1 || c2 || c3
   DISPLAY "c4 = "<<", c4, ">>"
   DISPLAY "c5 = "<<", c5, ">>"
END MAIN
```

The DISPLAY statements in this example produce the following output:

```
c4 = <<SOME THING>>
c5 = <<          >>
```

Which separator you use should depend on how you want NULL strings handled when the list can include both NULL and non-NULL values.

4GL performs data type conversion on compatible data types (as described in "Data Type Conversion" on page 3-42).

GLS

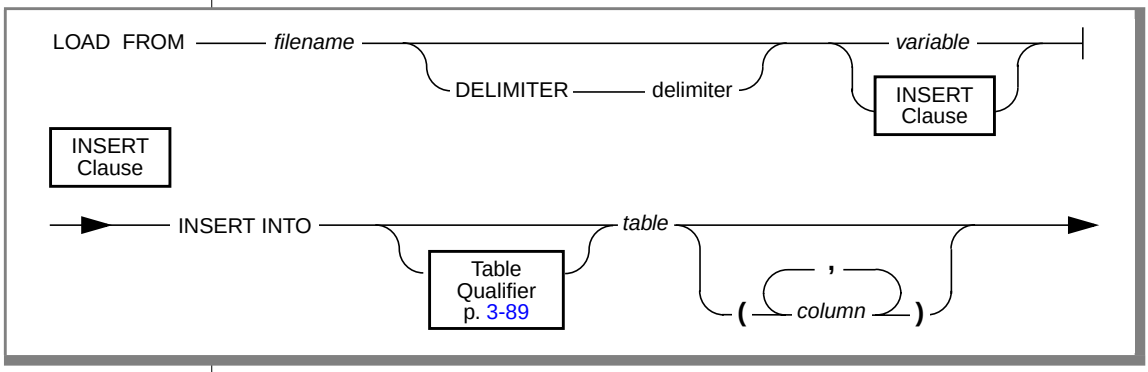
In nondefault locales, if you assign number or monetary values in the LET statement, the conversion process inserts locale-specific separators and currency symbols into the created strings, rather than U.S. English separators and currency symbols. This result occurs regardless of whether you include the USING operator in the LET statement. If **DBFORMAT** or **DBMONEY** is set, however, their settings override the default settings in the locale files. ♦

References

FOREACH, GLOBALS, INITIALIZE

LOAD

The LOAD statement inserts data from a file into an existing table.



Element	Description
column	is the name of a column in table, in parentheses. If you omit the list of column names, the default is all the columns of *table*.
delimiter	is a quoted string (or a CHAR or VARCHAR variable) containing a delimiter symbol.
filename	is a quoted string (or a CHAR or VARCHAR variable) specifying a file that contains the input data. This can include a pathname.
table	is the name of a table, synonym, or view in the current database, or in a database specified in the table qualifier.
variable	is a CHAR or VARCHAR variable containing an INSERT clause.

Usage

The LOAD statement must include an INSERT statement (either directly or as text in a variable) to specify where to store the data. LOAD appends the new rows to the specified table, synonym, or view, but does not overwrite existing data. It cannot add a row that has the same key as an existing row. You cannot use the PREPARE statement to preprocess a LOAD statement.

The user who executes LOAD must have Insert privileges for *table*. (Database and table-level privileges are described in *Informix Guide to SQL: Syntax*.) For readers familiar with the ACE reports of INFORMIX-SQL, LOAD provides flat-file input functionality, similar to that of the READ command of ACE.

The Input File

The variable or string following the LOAD FROM keywords must specify the name of a file of ASCII characters (or characters that are valid for the client locale) that represent the data values that are to be inserted. How data values in this input file should be represented by a character string depends on the SQL data type of the receiving column in *table*.

Data Type	Input Format
CHAR, VARCHAR, TEXT	Values can have more characters than the declared maximum length of the column, but any extra characters are ignored. A backslash (\ ) is required before any literal backslash or any literal delimiter character, and before any NEWLINE character anywhere in a VARCHAR value, or as the last character in a TEXT value. Blank values can be represented as one or more blank characters between delimiters, but leading blanks must not precede other CHAR, VARCHAR, or TEXT values.
DATE	In the default locale, values must be in *month* / *day* / *year* format (see "Numeric Date" on page 3-75) unless another format is specified by **DBDATE** or **GL_DATE** environment variables. You must represent the month as a 2-digit number. You can use a 2-digit number for the year if you are satisfied with the **DBCENTURY** setting. Values must be actual dates; for example, February 30 is invalid.
DATETIME, INTERVAL	INTERVAL values must be formatted *year-month* or else *day hour*:*minute*:*second.fraction*, or a contiguous subset thereof; DATETIME values must be in the format *year-month-day hour:minute:second.fraction*, or a contiguous subset, without DATETIME or INTERVAL keywords or qualifiers. Time units outside the declared column precision can be omitted. DATETIME *year* must be a four-digit number; all other time units (except *fraction*) require two digits.
MONEY	Values can include currency symbols, but these are not required.

(1 of 2)

Data Type	Input Format
SERIAL	Values can be represented as 0 to tell the database server to supply a new SERIAL value. You can specify a literal integer greater than zero, but if the column has a unique index, an error results if this number duplicates an existing value.
BYTE	Values must be ASCII-hexadecimals; no leading or trailing blanks.

(2 of 2)

Each set of data values in *filename* that represents a new row is called an input record. The NEWLINE character must terminate each input record in *filename*. Specify only values that 4GL can convert to the data type of the database column. For database columns of character data types, inserted values are truncated from the right if they exceed the declared length of the column.

NULL values of any data type must be represented by consecutive delimiters in the input file; you cannot include anything between the delimiter symbols.

This example shows two records in a hypothetical input file called **nu_cus**:

```
0|Jeffery|Padgett|Wheel Thrills|3450 El Camino|Suite 10|Palo
Alto|CA|94306||
0|Linda|Lane|Palo Alto Bicycles|2344 University||Palo
Alto|CA|94301|(415)323-6440
```

This **nu_cus** data file illustrates the following features of LOAD:

- The first data value in each record is zero, because the database server should supply a value for a SERIAL column in the row of the database table.
- The pipe symbol (|), the default delimiter, separates consecutive values.
- LOAD uses adjacent delimiters to assign NULL values to the **phone** column in the first record and to the **address2** column for the second record.

The following LOAD statement inserts all the values from the **nu_cus** file into a **customer** table that is owned by the user whose login is **krystl**:

```
LOAD FROM "nu_cus" INSERT INTO krystl.customer
```

Each input record must contain the same number of delimited data values. If the INSERT clause has no list of columns, the sequence of values in each input record must match the columns of *table* in number and order. Each value must have the literal format of the column data type, or of a compatible data type. (See "Summary of Compatible 4GL Data Types" on page 3-46.)

A file created by the UNLOAD statement can be used as input for the LOAD statement if its values are compatible with the schema of *table*.

The **onload** and **dbload** utilities give you more flexibility for the format of the input file. See your database server documentation for a description of **onload**, and see the *INFORMIX-SE Administrator's Guide* for a description of **dbload**.

GLS

The LOAD statement expects incoming data in the format specified by environment variables **DBFORMAT**, **DBMONEY**, **DBDATE**, **GL_DATE**, and **GL_DATETIME**. The precedence of these format specifications is consistent with forms and reports. If there is an inconsistency, an error is reported and the LOAD is cancelled. For more information, see Appendix E, "Developing Applications with Global Language Support." ♦

The DELIMITER Clause

The DELIMITER clause specifies the symbol that must separate consecutive data values in each input record, and must terminate any record whose last value is NULL. The next example uses the caret (^) symbol as the delimiter:

```
LOAD FROM "/a/data/ord.loadfile" DELIMITER "^"
   INSERT INTO orders
```

If you omit the DELIMITER clause, the default delimiter symbol is the value of the **DBDELIMITER** environment variable, or else a pipe (|) symbol (ASCII 124) if **DBDELIMITER** is not set. For details, see Appendix D.

- Hexadecimal numbers (0 through 9, *a* through *f*, or *A* through *F*)
- NEWLINE or CONTROL-J
- Backslash (\ )

The backslash serves as an escape character to indicate that the next character is to be interpreted literally as part of the data, rather than as a delimiter or record separator or escape character. If any character value in the input file includes the delimiter or NEWLINE symbols without backslashes, the LOAD statement produces error -846; see *Informix Error Messages* in Answers OnLine.

Important: *When this error occurs, the* ***SQLCA.SQLERRD[3]*** *character is always set to 1, regardless of how many records (if any) were successfully loaded into the database. For this reason, unless the LOAD operation occurs within a transaction, recovery of the database after this error is not trivial.*

The UNLOAD statement automatically inserts a backslash before any literal delimiter or NEWLINE symbol in character values. When LOAD (or the **onload** or **dbload** utility) inserts output from UNLOAD into a database table, these escapist backslash symbols are automatically stripped from the data.

The INSERT Clause

The INSERT clause specifies the table and columns in which to store the new data. This clause supports a subset of the syntax of the INSERT statement, which is described in the *Informix Guide to SQL: Syntax*. You cannot, however, include the VALUES, SELECT, or EXECUTE PROCEDURE clause of the INSERT statement within the INSERT clause of the LOAD statement. You must specify explicit column names if either of these conditions is true:

- You are not inserting data into all of the columns of *table*.
- The input file does not match the default order of columns, as listed in the **syscolumns** table of the system catalog.

The following example identifies the **price** and **discount** columns as the only columns into which to insert non-NULL data values:

```
LOAD FROM "/tmp/prices" DELIMITER ","
   INSERT INTO maggie.worktab(price,discount)
```

Data Integrity Issues with LOAD

If LOAD is executed within a transaction, the inserted rows are locked, and they remain locked until the COMMIT WORK or ROLLBACK WORK statement terminates the transaction. If no other user is accessing the table, you can avoid locking limits and reduce locking overhead by locking the table with the LOCK TABLE statement after the transaction begins. This exclusive table lock is released when the transaction terminates. (Transactions, row locking, and table locking are described in *Informix Guide to SQL: Tutorial*.)

Consult the documentation for your database server about the limit on the number of locks available during a single transaction.

Important: *In a database that is not ANSI-compliant, but that supports transaction logging, it is recommended that you use the BEGIN WORK statement to initiate a transaction prior to any LOAD statement. Otherwise, if the LOAD statement fails after inserting some rows, it might be difficult to restore the database to its condition before the LOAD statement began to execute outside any transaction.*

If the current database has no transaction log, a failing LOAD statement cannot remove any rows that were loaded before the failure occurred. You must manually remove the already loaded records from either the load file or from the receiving table, repair the erroneous records, and rerun LOAD. This is true for all versions of 4GL.

In versions of 4GL prior to 6.01, if the database has transaction logging and a transaction is in effect before the LOAD statement executes, 4GL always commits the transaction when the LOAD statement completes. You can neither control nor prevent this internal COMMIT WORK operation.

Beginning with Version 6.01, however, you can take one of the following actions when the database has a transaction log:

- Run LOAD as a singleton transaction, so that any error causes the entire LOAD statement to be automatically rolled back.
- Run LOAD within an explicit transaction, so that a data error merely stops the LOAD statement in place with the transaction still open.

LOAD does not execute a COMMIT WORK or ROLLBACK WORK automatically unless LOAD was the first statement of a new (implicit) transaction. For example, if the database is ANSI-compliant, a transaction is always in effect. LOAD would be the first statement of a new transaction only if the immediately preceding SQL statement was a DATABASE statement, a COMMIT WORK statement, or a ROLLBACK WORK statement.

If the database is not ANSI-compliant but has a transaction log, a new transaction is indicated by a BEGIN WORK statement. LOAD is the first statement of a new transaction only if the immediately preceding SQL statement was a BEGIN WORK statement.

In either type of logged database, if LOAD is the first statement of a new transaction, 4GL automatically commits the transaction if the load completes without errors. If errors are found, a rollback is automatically performed.

If LOAD is *not* the first statement of a new transaction, 4GL leaves the transaction open. This allows you to take one of the following actions:

- COMMIT the successfully loaded rows, fix the load records, and rerun LOAD with the balance of the records.
- Abort the LOAD altogether by executing ROLLBACK WORK, followed by repeating the LOAD operation from the beginning.

The following 4GL script fragment illustrates a typical LOAD statement series using an explicit transaction in a database that is not ANSI-compliant:

```
create database mine with log in "/db/mine/trans.log";
create table mytab1 (col1 serial, col2 char(20), col3 date);
create table loadlog (tabname char(18), loaddate date);
begin work;
insert into  loadlog values ("mytab1", today);
load from "mytab1.unl" insert into mytab1;
```

If the LOAD is successful, at this point you can execute COMMIT WORK or ROLLBACK WORK as appropriate.

The next 4GL script fragment illustrates the same steps using an explicit transaction in an ANSI-compliant database:

```
create database mine_ansi with log in "/db/mine/trans.log" MODE ANSI;
create table "user1".mytab1 (col1 serial, col2 char(20), col3 date);
create table "user1".loadlog (tabname char(18), loaddate date);
commit work;
insert into "user1".loadlog values ("mytab1", today);
load from "mytab1.unl" insert into "user1".mytab1;
```

If the LOAD is successful, at this point you can execute COMMIT WORK or ROLLBACK WORK as appropriate.

The third 4GL script fragment illustrates a typical LOAD statement (with an implicit transaction) in a database that is not ANSI-compliant:

```
create database mine with log in "/db/mine/trans.log";
create table mytab1 (col1 serial, col2 char(20), col3 date);
close database;
database mine;
load from "mytab1.unl" insert into mytab1;
```

If the LOAD has no errors, the changes are committed. If error messages appear, the rows that were loaded before the error occurred are rolled back automatically.

The final 4GL script fragment illustrates a typical LOAD statement (with implicit transaction) in a database that is ANSI compliant:

```
create database mine_ansi with log in "/db/mine/trans.log" MODE ANSI;
create table "user1".mytab1 (col1 serial, col2 char(20), col3 date);
close database;
database mine_ansi;
load from "mytab1.unl" insert into "user1".mytab1;
```

If the LOAD has no errors, the changes are committed. If error messages appear, the rows that were loaded before the error occurred are rolled back automatically.

Performance Issues with LOAD

For valid input files, LOAD provides better performance when the *table* that the INSERT INTO clause references has no index, no constraint, and no trigger.

If one or more trigger, constraint, or index exists on the table, however, it is recommended that you follow these steps:

1. Use SET INDEX…DISABLED to disable any indexes on the table.
2. Use SET CONSTRAINT…DISABLED to disable any constraints.
3. Use SET TRIGGER…DISABLED to disable any triggers.
4. Use LOAD to insert data into the table.
5. Use SET INDEX… ENABLED to restore any indexes on the table
6. Use SET CONSTRAINT…ENABLED to restore any constraints.
7. Use SET TRIGGER...ENABLED to restore any triggers.

(See *Informix Guide to SQL: Syntax* for the syntax of the SET INDEX, SET CONSTRAINT, and SET TRIGGER statements.) It is more efficient to follow these steps than to drop the indices, constraints, and triggers; perform the LOAD; and then recreate the indices, constraints, and triggers.

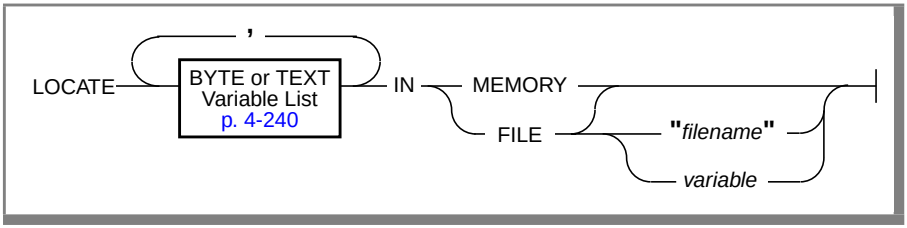
Because the SET statement was introduced in the Informix implementation of the SQL language after the 4.10 release, these SET statements must be prepared, or else must be delimited by the SQL and END SQL keywords.

References

DATABASE, UNLOAD

LOCATE

The LOCATE statement specifies where to store a TEXT or BYTE value.



Element	Description
filename	is the name of a file in which to store the TEXT or BYTE value. This specification can include a pathname and file extension.
variable	is the name of a CHAR or VARCHAR variable containing a *filename* specification.

Usage

The TEXT or BYTE data type can stores a large binary value. You must specify whether you want to store the value of the variable in memory or in a file. You can access a value from memory faster than from a file. If your program exceeds the available memory, however, 4GL automatically stores part of the TEXT or BYTE value in a file. To use a large variable, your program must include the following steps:

1. Declare the variable with a DEFINE statement.
2. Use the LOCATE statement to specify the storage location.

   The LOCATE statement must appear within the scope of reference of the variable.

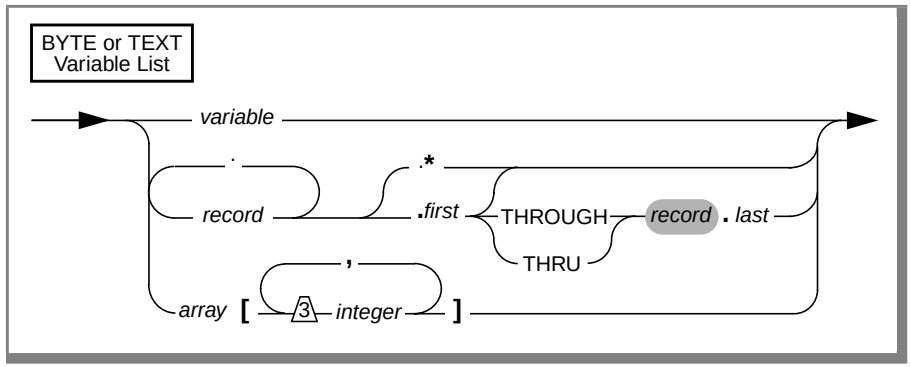
The following topics are described in this section:

- "Passing Large Variables to Functions" on page 4-243
- "Freeing the Storage Allocated to a Large Data Type" on page 4-243

The LOCATE statement must follow any DEFINE statement that declares TEXT or BYTE variables, and it must appear in the same program block as a local TEXT or BYTE variable. If you try to access a TEXT or BYTE value before initializing its variable with a LOCATE statement, 4GL generates a runtime error.

The List of Large Variables

Use the following syntax to specify the large variables to be initialized.



Element	Description
array	is the name of a structured variable of the ARRAY data type.
first	is the name of a large member variable to be initialized.
integer	is a literal integer between 0 and the declared size of the array.
last	is another member of *record* that was declared later than *first*.
record	is the name of a structured variable of the RECORD data type.
variable	is the name of a large variable of the TEXT or BYTE data type.

As in all 4GL statements that do not declare variables, any identifier of an array, record, record member, or variable must have been previously declared and must be within its scope of reference. (The only 4GL statements that can declare variables are DEFINE and GLOBALS.)

You can then use most 4GL statements to access the variable. The LET statement can assign a NULL value to a TEXT or BYTE variable, but it cannot assign non-NULL values. The INTO clause of the SELECT statement can assign to a specified variable a TEXT or BYTE value from the database.

The IN MEMORY Option

Use the IN MEMORY option to allocate storage in memory for TEXT and BYTE values. The following example declares the variable **quarter** as the same data type as the database column **analysis**, and stores the variable in memory:

```
DEFINE quarter LIKE stock.analysis
LOCATE quarter IN MEMORY
```

If the TEXT or BYTE variable has already been stored in memory, you can use the LOCATE statement again to reinitialize the variable.

If a TEXT or BYTE variable has been initialized to memory or to a temporary file, you can use LOCATE to reinitialize the variable. You *cannot* reinitialize a TEXT or BYTE variable that is stored in a named file.

The IN FILE Option

The IN FILE option stores the TEXT or BYTE value in a file, whose name can be specified as a quoted string, as a CHAR or VARCHAR variable, or as a CHAR or VARCHAR member of a record or element of an array. 4GL opens and closes the file each time that you use the variable in an SQL or other 4GL statement.

When you retrieve a row containing a TEXT or BYTE column, the value of the column overwrites the current contents of the file. Similarly, when you update a row, 4GL reads and stores the entire contents of the file in the database column.

As with storage in memory, the file contains only the value most recently assigned to the variable. You have several options with the IN FILE clause:

- Omit a filename, so that 4GL places the value in a temporary file.
- Specify a variable that contains the name of a file in which to store the TEXT or BYTE value. The filename can include a pathname.

Using a Temporary File

If you omit the filename, 4GL places the TEXT or BYTE value in a temporary file. 4GL creates the temporary file at runtime in the directory identified by the **DBTEMP** environment variable. If **DBTEMP** is not set, 4GL puts the temporary files in the **/tmp** directory. If no temporary directory exists, a runtime error occurs.

The following example omits the filename. It also shows that TEXT and BYTE types can be declared as components of RECORD variables:

```
DEFINE stock RECORD
      n INTEGER, analysis TEXT, graph BYTE
   END RECORD
LOCATE stock.analysis IN FILE
LOCATE stock.graph IN FILE
```

If the TEXT or BYTE variable has already been located in a temporary file, you can use the LOCATE statement again reinitializes the variable.

You can specify multiple filenames by declaring an array of character variables. This example stores an array of filenames in an array of TEXT variables:

```
DEFINE flnames ARRAY[10] OF char(20),
   t_holds ARRAY[10] of TEXT
   i INTEGER
FOR i = 1 TO 5
   LET flnames[i] = "/u/db/profile", i, USING "<<&"
   LOCATE t_holds[i] IN FILE flnames[i]
END FOR
```

Specifying a Filename

To place the TEXT or BYTE value in a specific file, the LOCATE statement can include either a literal filename, or else a character variable that contains the filename. This example uses a quoted string to specify the filename:

```
DEFINE analysis TEXT
LOCATE analysis IN FILE "/u/db/analysis"
```

The next example uses a CHAR variable to specify the filename:

```
DEFINE flname CHAR(20),t_hold TEXT
LET flname = "/tmp/TodaysReport"
LOCATE t_hold IN FILE flname
```

Passing Large Variables to Functions

If you specify a variable in the argument list of a function or report, 4GL ordinarily passes it by value. The function or report can modify the passed value without affecting the variable in the calling function.

4GL handles large data types differently. It passes large variables *by reference*. If a function modifies a TEXT or BYTE variable, the change is apparent to the variable in the calling routine. The CALL statement need not include a RETURNING clause for a TEXT or BYTE value.

Freeing the Storage Allocated to a Large Data Type

If you no longer need a TEXT or BYTE variable, you can use the following statements to release the memory that stored its value.

Statement	Description
FREE	If you stored the TEXT or BYTE variable in a file, you can reference the variable in the FREE statement to delete the file. If you stored the TEXT or BYTE variable in memory, the FREE statement releases all memory associated with the variable.
LOCATE	The LOCATE statement for the same variable releases memory and removes temporary files, but does not remove named files.

When it encounters the RETURN statement or the END FUNCTION or END REPORT keywords, 4GL frees any *local* TEXT or BYTE variables that are stored in memory or in a temporary file. 4GL does not, however, deallocate storage for TEXT and BYTE variables that are passed by reference as arguments to a function or to a report. Storage for such variables is deallocated when EXIT PROGRAM or END MAIN terminates the program. 4GL does not automatically remove a named file that is associated with a TEXT or BYTE variable.

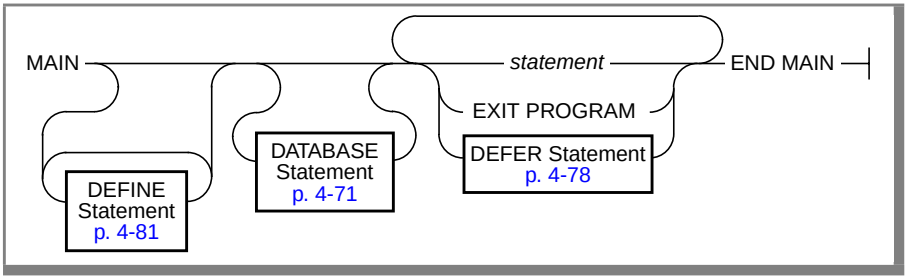
After you release the storage, you cannot access the TEXT or BYTE variable without executing a new LOCATE statement to initialize it. If you have named the file for the TEXT or BYTE value, and you want to retain the file, do not use the FREE statement. For information on the FREE statement, see the *Informix Guide to SQL: Syntax*.

References

DEFINE, EXIT, FUNCTION, GLOBALS, MAIN, INITIALIZE, REPORT, RETURN

MAIN

The MAIN statement defines the MAIN program block.



Element	Description
statement	is any SQL or other 4GL statement (except MAIN, FUNCTION, GLOBALS, NEED, PAUSE, PRINT, REPORT, RETURN, or SKIP).

Usage

Every 4GL program must have exactly one MAIN statement, which typically calls functions or reports to do the work of the application. The following fragment calls functions defined in the same module as the MAIN statement:

```
MAIN
   ...
   CALL get_states()
   CALL ring_menu()
   ...
END MAIN

FUNCTION get_states()
   ...
END FUNCTION

FUNCTION ring_menu()
   ...
END FUNCTION
```

The MAIN statement cannot appear within another statement. It must be the first program block of the module in which it appears, as in this example.

The END MAIN keywords mark the end of the MAIN program block. The program terminates when it encounters these keywords.

If it encounters the EXIT PROGRAM statement, the program terminates before END MAIN. The Interactive Debugger treats this as an abnormal termination.

Variables Declared in the MAIN Statement

You can declare variables by including DEFINE statements within the MAIN program block. Variables that you declare here are *local* to the MAIN block; you cannot reference their names in any FUNCTION or REPORT definition.

If you include a DEFINE statement before the MAIN statement, however, and outside of any FUNCTION or REPORT statement, its *module* variables are visible to subsequent statements in any program block of the same source module. The GLOBALS statement can extend the visibility of a module variable beyond the module where it is declared. If you assign the same identifier to variables that differ in scope of reference, in any portion of your program where the scopes of their names overlap, the following rules of precedence apply:

- A local variable has the highest precedence, so that within its scope, no identical identifier of a global or module variable can be visible.
- Within the module in which it was declared, a module variable takes precedence over another with the same identifier whose scope has been extended by the GLOBALS "*filename*" statement.

You should assign unique names to global and module variables that you intend to reference within the MAIN program block.

DEFER and DATABASE Statements and the MAIN Program Block

DEFER statements can appear only within the MAIN statement.

Any DATABASE statement that appears before the MAIN statement (but in the same module) specifies the default database at compile time. This database also becomes the current database at runtime, unless another DATABASE statement specifies a different database. Any DATABASE statement in the MAIN statement must follow the last DEFINE declaration.

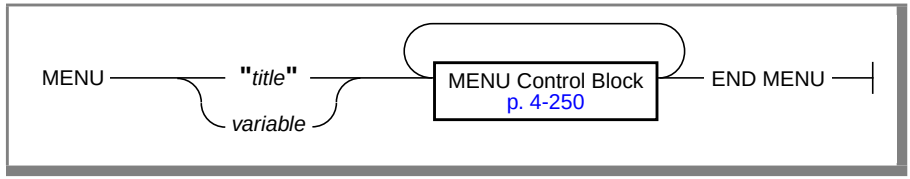
This database becomes the current database for subsequent SQL statements until the program ends, or until another DATABASE statement is encountered.

References

DATABASE, DEFER, DEFINE, EXIT PROGRAM, FUNCTION, GLOBALS, REPORT

MENU

The MENU statement creates and displays a ring menu. From the keyboard, the user can choose menu options that execute blocks of statements.



Element	Description
title	is a quoted string that specifies a display label for the ring menu. If the string is empty, no menu title is displayed.
variable	is a CHAR or VARCHAR variable containing a display label.

Usage

This statement specifies and displays a ring menu that occupies two lines in the current 4GL window. You can use the MENU statement to accomplish these tasks:

- Create and display a screen menu, including a title of the menu
- Define and display a list of menu options that the user can select
- Specify a single-line description to display for each menu option
- Designate help message numbers for each menu option
- Specify a block of 4GL statements to execute for each menu option

When it encounters a MENU statement, 4GL performs these actions:

1. Displays the menu title (and as many menu options as can fit) in the Menu line of the current 4GL window.
2. Moves the *menu cursor* to the first option and displays its description

   The order of options is determined by the order of control blocks. The menu cursor marks the current menu option with a double border. (The term *ring menu* refers to a menu where moving the menu cursor beyond the last option returns it to the first option.)

3. Waits for the user to press the activation key for a MENU control block, or to terminate the MENU statement by pressing the Quit key or Interrupt key
4. If the user presses an activation key, executes statements in the corresponding control block, until it encounters one of these statements:
   - **EXIT MENU statement.** 4GL then exits from the menu.
   - **CONTINUE MENU statement.** 4GL skips any remaining statements in the MENU control block, and redisplays the menu.
   - **Last statement in the MENU control block.** 4GL redisplays the menu so that the user can choose another option.

A menu can appear above or below a screen form, but not within a form. 4GL displays the menu title and the menu options on the Menu line. This reserved line is positioned by the most recent MENU LINE specification in the OPTIONS or OPEN WINDOW statement. The default position is the first line of the current 4GL window.

Unless the title and at least one option can fit on the screen or in the current 4GL window, a runtime error occurs. For information on multiple-page menus, and how the set of menu options acts like a *ring* for the menu cursor, see "Scrolling the Menu Options" on page 4-266.

The title of a menu is just a display label; your program cannot reference a menu by name. To repeat the same menu and all its behavior in different parts of a program, you can include its MENU statement in a FUNCTION definition, so that you can invoke the function when you want the menu to appear. The following topics are described in this section:

- "Completing the MENU Statement" on page 4-268
- "COMMAND KEY Conflicts" on page 4-271

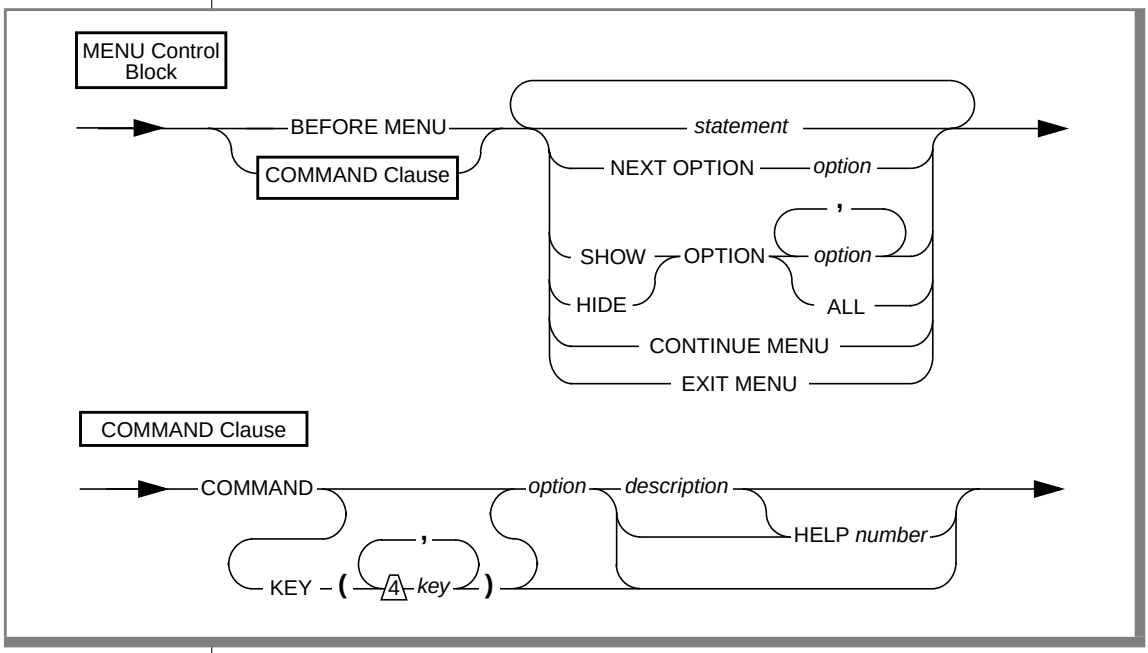
The MENU Control Blocks

Each control block includes a statement block of at least one statement, and an *activation clause* that specifies when to execute the statement block. Any of three types of activation clauses can appear within MENU control blocks:

- BEFORE MENU clause (statements to execute before the menu is displayed)
- COMMAND clause (to specify the name and description of an option, an optional activation key to choose the option, and an optional help message number, identifying a help message to display if the user presses the Help key while this is the current option; 4GL executes the statements in this block when the user chooses the menu option)
- Hidden option (a COMMAND clause that only specifies activation keys to execute a statement block if the key is pressed; no option name, description, nor help message number is specified)

The statement block can specify SQL or other 4GL statements to execute when a user presses a key sequence, as well as special MENU instructions:

- The next menu option to highlight with the menu cursor
- Whether to suppress or restore the display of one or more menu options
- Whether to exit from the MENU statement



Element	Description
description	is a quoted string or the name of a CHAR or VARCHAR variable that contains an option description for the Menu help line.
key	is a letter, a literal symbol in quotation marks, or a keyword. (Quotation marks are not required if *key* is a single letter of the alphabet.) This list of up to four activation keys must be enclosed in parentheses; see "The KEY Clause" on page 4-255.
number	is an integer that identifies the help message for this menu option. You must have used the OPTIONS statement previously to identify the help file containing the message.
option	is a quoted string or the name of a CHAR or VARCHAR variable that contains the name of the menu option. This name cannot be longer than the width of the current 4GL window.
statement	is an SQL statement or other 4GL statement.

The screen displays a ring menu of *option* names as menu options. The menu options appear in the same order in which you specify them in COMMAND clauses within the MENU statement.

You must include at least one non-hidden option (that is, one COMMAND clause with a non-null *option*) for each menu. Within the MENU control block that includes the COMMAND clause, you can include statements that perform the activity specified by the menu option.

The *description* appears on the line below the menu when the *option* is current. The string length must not be longer than the width of the screen or 4GL window. See also "Identifiers in the MENU Statement" on page 4-263.

The BEFORE MENU Block

Before displaying the menu, 4GL executes any statements in the statement block that follows the optional BEFORE MENU keywords. Use statements in this control block to perform preliminary tasks, such as:

- Specifying values for variables used for the menu name, the names of options, and the strings containing descriptions of options
- Hiding or showing individual menu options
- Checking user access privileges

If 4GL encounters the EXIT MENU statement here, no menu is displayed.

The following program fragment includes statements that specify the name of the menu, the name of an option, and the option description at runtime:

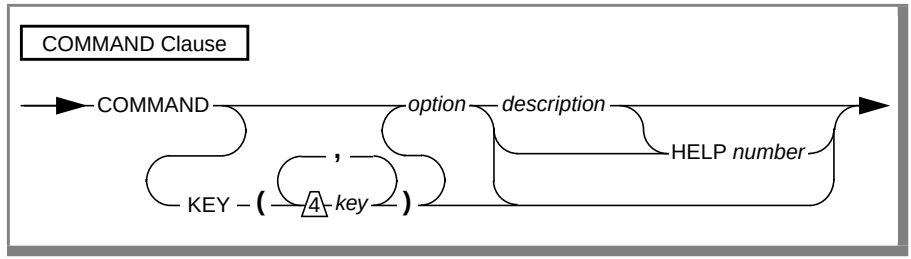
```
DEFINE menu_name, opt_name CHAR(20)
   opt_desc CHAR(40), priv_flag SMALLINT
LET menu_name = "SEARCH"
LET opt_name = "Query"
LET opt_desc = "Query for customers."
IF ...
   LET priv_flag = 1
END IF
MENU menu-name
   BEFORE MENU
      IF priv_flag THEN
         LET menu_name = "POWER SEARCH"
         LET opt_name = "Power Query"
      END IF
   COMMAND opt_name opt_desc HELP 12
      IF priv_flag THEN
         CALL cust_find(1)
      ELSE
         CALL cust_find(2)
      END IF
   ...
END MENU
```

The next BEFORE MENU clause initially hides all menu options. If the user is privileged, 4GL then displays all the menu options. If the user is not privileged, 4GL displays only the Query, Detail, Switch, and Exit menu options:

```
MENU menu_name
   BEFORE MENU
      HIDE OPTION ALL
      IF priv THEN
         LET menu_name = "PRIVILEGED SEARCH"
         SHOW OPTION ALL
      ELSE SHOW OPTION "Query", "Detail", "Switch", "Exit"
      END IF
   ...
END MENU
```

The COMMAND Clause

The COMMAND clause can define a menu *option* that appears after the menu title in the Menu line, as well as its description that appears in the following line when the menu cursor is on the option.



For definitions of terms, see "The MENU Control Blocks" on page 4-250. Each COMMAND clause is part of a MENU control block whose statements perform the activity specified by the option description. To nest menus, you can include another MENU statement. The MENU control blocks might be easier to read if you use function calls to group statements.

The COMMAND clause can optionally include a HELP clause to associate a help message *number* with the menu *option*. It can also include a KEY clause, to specify up to four activation keys that the user can press to choose the option; otherwise, 4GL recognizes default activation keys, based on the initial character of *option*. By default, when OPTION is highlighted, pressing the RETURN key has the same effect as pressing an activation key.

Optionally, you can include a *description* of the menu option in a COMMAND clause. The description appears on the line below the menu and is displayed when the option is highlighted.

4GL produces a runtime error if a menu *option* or its *description* exceeds the width of the screen or the width of the current 4GL window.

If the name and description of the menu option are omitted, the COMMAND clause produces no visual display, as described in "Invisible Menu Options" on page 4-257.

The HELP Clause

The HELP clause specifies the *number* of a help message to display for *option*. 4GL displays this help message when the corresponding menu option is current and a user presses the Help key. The default Help key is CONTROL-W. You can use the OPTIONS statement to assign a different Help key.

The following MENU statement specifies different help message numbers for each of two menu options:

```
MENU "MAIN"
   COMMAND "Customer" "Enter and maintain customer data"
         HELP 101
      CALL customer( )
      CALL ring_menu( )
   COMMAND "Orders" "Enter and maintain orders" HELP 102
      CALL orders( )
      CALL ring_menu( )
   ...
END MENU
```

You can specify help messages (and their numbers) in an ASCII file whose filename appears in the HELP FILE clause of the OPTIONS statement. Use the **mkmessage** utility, as described in "The mkmessage Utility" on page B-2, to create a runtime version of the help file. A runtime error occurs if the help file cannot be opened, or if you specify a help number that is not defined in the help file, or that is greater than 32,767.

An invisible menu option cannot have a Help clause.

The KEY Clause

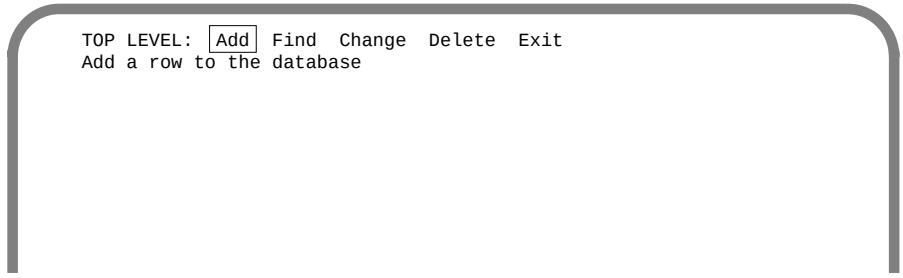
The KEY clause in a MENU control block specifies one or more activation keys that users can press to choose the option (if an option name is specified) and to execute the statements in the MENU control block. If you omit the KEY clause, the first character in *option* name is the default activation key to choose the option. Conversely, if the KEY clause assigns a key to an option, the first letter no longer activates the option.

If a user chooses the option, 4GL executes the statements in the MENU control block that includes the COMMAND clause. If EXIT MENU is not among these statements, 4GL redisplays the menu, so the user can choose another option.

This MENU statement, for example, creates a menu entitled **TOP LEVEL** that displays five options. The default activation keys are A, F, C, D, and E:

```
MENU "TOP LEVEL"
   COMMAND "Add" "Add a row to the database."
      ...
   COMMAND "Find" "Find a row in the database."
      ...
   COMMAND "Change" "Update a row in the database."
      ...
   COMMAND "Delete" "Delete a row from the database."
      ...
   COMMAND "Exit" "Return to the operating system."
      EXIT MENU
END MENU
```

This MENU statement produces the following initial display:



One option is always marked as the current option. This option is marked by a double border, called the menu cursor.

The line under the menu options (the Menu help line) displays a description of the menu option, as specified in the COMMAND clause for that menu option. If the menu cursor moves to another option, the display in this line changes, unless you specify the same description for both menu options.

4GL executes the statements in the MENU control block if the user presses an activation key that you specify by *key* specification in the KEY clause:

- Letters (Both upper- and lowercase letters are valid, but 4GL does not distinguish between them.)
- Symbols (such as !, @, or #) enclosed between quotation marks
- Any of the following keywords (in uppercase or lowercase letters):

DOWN	INTERRUPT	RETURN or ENTER	TAB
ESC or ESCAPE	LEFT	RIGHT	UP
F1 through F64			
CONTROL-*char* (except A, D, H, I, J, K, L, M, R, or X)			

The following keys deserve special consideration before you assign them as activation keys in the KEY clause of a MENU statement.

Key	Special Considerations
ESC or ESCAPE	You must use the OPTIONS statement to specify another key as the Accept key, because this is the default Accept key.
Interrupt	You must include a DEFER INTERRUPT statement. When the user presses the Interrupt key under these conditions, **4GL** executes the statements in the MENU control block and sets **int_flag** to non-zero, but does not terminate the MENU statement.
QUIT	4GL also executes the statements in the control block if DEFER QUIT has been executed and the user presses the Quit key. In this case, 4GL sets **quit_flag** to non-zero.

(1 of 2)

Key	Special Considerations
CONTROL-*char*	
A, D, H, K L, R, and X	4GL reserves these keys for field editing.
I, J, and M	The usual meanings of these keys (TAB, LINEFEED, and RETURN, respectively) are not available to the user. Instead, the key is trapped by 4GL and used to trigger the menu option. For example, if CONTROL-M appears in the KEY clause, the user cannot press RETURN to advance the cursor to the next field.

(2 of 2)

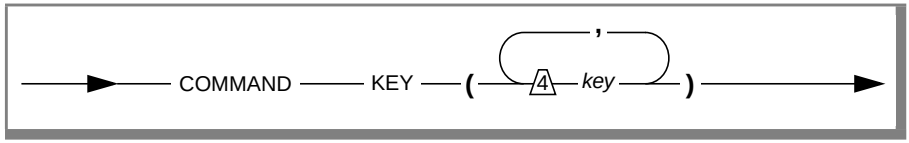
Some other control keys, such as S, Q, or Z also might not be allowed, depending on your implementation of UNIX.

The *key* must be unique among all KEY clauses of the same MENU statement. You might not be able to specify other keys that have special meaning to your operating system. The *key* value also cannot be the default activation key of any other COMMAND clause. If you specify a letter here as the activation key, it must be different from the first letter of any option of the same menu.

See also "COMMAND KEY Conflicts" on page 4-271.

Invisible Menu Options

You can add an *invisible* option (an option that is never displayed) to a menu by including a KEY clause in the COMMAND clause of the MENU control block, but not specifying an option name or an option description.

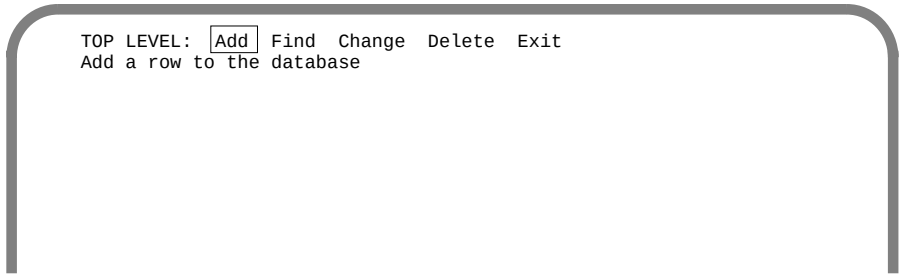


Just as with visible options, the *key* value cannot be the activation key of any other COMMAND clause. If you specify a letter here as the activation key, it must be different from the first letter of any option of the same menu.

The following MENU statement creates a menu named **TOP LEVEL** with six options, of which only five appear in the menu display. The exclamation point (!) key chooses an invisible option that is not displayed on the menu. Here a *description* and a help *number* are associated with each visible *option*:

```
MENU "TOP LEVEL"
   COMMAND "Add" "Add a row to the database" HELP 12
      ...
   COMMAND "Find" "Find a row in the database" HELP 13
      ...
   COMMAND "Change" "Update a row in the database" HELP 14
      ...
   COMMAND "Delete" "Delete a row from the database" HELP 15
      ...
   COMMAND KEY ("!")
      CALL bang()
      ...
   COMMAND "Exit" "Return to operating system" HELP 16
      EXIT PROGRAM
END MENU
```

These statements produce the following menu display:



At least one COMMAND clause, however, must include an *option*. You cannot specify a menu in which every option is invisible. If your application requires such a menu, the MENU statement can include a COMMAND clause in which the option is null (that is, an empty string), as in the following example:

```
MENU ""
COMAND ""
COMMAND KEY(UP)    DISPLAY "Up   " AT 3,1
COMMAND KEY(DOWN)  DISPLAY "Down " AT 3,1
COMMAND KEY(LEFT)  DISPLAY "Left " AT 3,1
COMMAND KEY(RIGHT) DISPLAY "Right" AT 3,1
COMMAND KEY(INTERRUPT) EXIT MENU
END MENU
```

This example would not be valid if the first COMMAND clause were omitted.

The CONTINUE MENU Statement

The CONTINUE MENU statement causes 4GL to ignore the remaining statements in the current MENU control block, and redisplay the menu. The user can then choose another menu option, as in the following program fragment.

In this example, the **Yearly Report** option first cautions the user that a report takes several hours to create. If the user types Y to create the report, 4GL calls the **calc_yearly( )** function. Otherwise, 4GL executes the CONTINUE MENU statement and redisplays the **YEAR END** menu:

```
MENU "YEAR END"
   COMMAND "Yearly Report" "Compile Yearly Statistics Report"
      PROMPT "This report takes several hours to create." ,
         "Do you want to continue? (y/n)" FOR answer
      IF answer MATCHES "[Yy]" THEN
         CALL calc_yearly()
      ELSE
         CONTINUE MENU
   ...
END MENU
```

The EXIT MENU Statement

The EXIT MENU statement terminates the MENU statement without executing any remaining statements in the menu control blocks. Use this statement at any point where you want the user to leave the menu instead of redisplaying it. You must specify this statement for at least one menu option in each 4GL menu. Otherwise, the user will have no way to leave the menu.

If it encounters the EXIT MENU statement, 4GL takes the following actions:

- Skips all statements between the EXIT MENU and END MENU keywords
- Deactivates the menu and erases the menu from the screen.
- Resumes execution at the first statement after the END MENU keywords

The following example demonstrates using the EXIT MENU keywords in the MENU block of a menu option named **Exit**:

```
MENU "CUSTOMER"
   ...
   COMMAND "Exit" "leave the CUSTOMER menu." HELP 5
      EXIT MENU
END MENU
```

(To exit from the current MENU control block without exiting from the MENU statement, use the CONTINUE MENU keywords rather than EXIT MENU.)

The NEXT OPTION Clause

When 4GL finishes executing the statements in a control block that includes a COMMAND clause, the option just executed remains as the current option. If you want a different option to be the current option, use the NEXT OPTION keywords. The NEXT OPTION clause identifies the name of a menu option to make current. This clause does *not* choose the next menu option; rather, it identifies the next menu option that will be highlighted as the current option. The user can simply press RETURN to choose the current option.

In the following MENU statement, if the user selects the **Query** option, 4GL calls the function **query_data( )**, and redisplays the menu with **Modify** as the current option. To choose the **Modify** option, the user presses RETURN.

```
MENU "CUSTOMER"
   COMMAND "Query" "Search for a customer"
      CALL query_data( )
      NEXT OPTION "Modify"
   ...
   COMMAND "Modify" "Modify a customer"
      ...
END MENU
```

Without NEXT OPTION, 4GL would display **Query** as the current option; the user would have to make **Modify** the current option and then press RETURN.

If you want the cursor to move among menu options in a certain order, list their defining COMMAND clauses in the desired order. Use the NEXT OPTION keywords only if you want to deviate from the default left-to-right order of the ring menu.

The HIDE OPTION and SHOW OPTION Keywords

When you want to display a subset of the menu options, use the HIDE OPTION and SHOW OPTION keywords to specify which options appear on a menu. The HIDE OPTION keywords can conceal some menu options from users. 4GL does not display a *hidden option* in the menu, and does not recognize as valid any keystroke that would otherwise select the option (if it were visible). Such options remain hidden and disabled, until 4GL executes a SHOW OPTION clause that references their *option* name.

The following MENU statement creates a menu with seven options. The Long_menu option shows all options; the **Short_menu** options shows only the **Query**, **Long_menu**, and **Exit** options:

```
MENU "Order Management"
   COMMAND "Query" "Search for orders"
      CALL get_orders( )
   COMMAND "Add" "Add a new order"
      CALL add_order( )
   COMMAND "Update" "Update the current order"
      CALL upd_order( )
   COMMAND "Delete" "Delete the current order"
      CALL del_order( )
   COMMAND "Long_menu" "Display all menu options"
      SHOW OPTION ALL
   COMMAND "Short_menu" "Display a short menu"
      HIDE OPTION ALL
      SHOW OPTION "Query", "Long_menu", "Exit"
   COMMAND "Exit" "Exit from the Order Management Form"
      EXIT MENU
END MENU
```

If you specify the options to hide by listing them in character variables, you must assign values to the variables before you can include the variables in a HIDE OPTION clause. (For more information about variables, see "Identifiers in the MENU Statement" on page 4-263.)

The ALL keyword in a SHOW OPTION or HIDE OPTION clause specifies all of the menu options that you created in any COMMAND clause.

Use the SHOW OPTION keywords to restore a list of menu options that the HIDE OPTION keywords disabled. By default, 4GL displays all menu options. You only need to use this statement if you have previously specified the HIDE OPTIONS keywords to disable at least one menu option.

4GL displays menu options in the same order in which their COMMAND clauses defined them. The order in which a SHOW OPTION clause lists options has no effect on the order of their subsequent appearance in the menu.

Do not confuse *hidden options* with *invisible options*. Neither appears on the menu, but hidden options cannot be accessed by the user until after they have been enabled by the SHOW OPTION keywords. Invisible options have an activation key, but no command name. Their statement blocks can be accessed by pressing an activation key, but they do not appear in the menu.

The HIDE OPTION and SHOW OPTION keywords cannot affect invisible options, because (as their name suggests) invisible options are never displayed. Use some other approach to enable and disable invisible options; for example, you might specify their actions within a conditional statement.

The following example MENU statement populates a menu with eight options. The **Long_menu** option shows all options; the **Short_menu** option shows only the **Query**, **Details**, **Long_menu**, and **Exit** options.

```
MENU "Order Management"
   COMMAND "Query" "Search for orders"
      CALL get_orders()
   COMMAND "Add" "Add a new order"
      CALL add_order()
   COMMAND "Update" "Update the current order"
      CALL upd_order()
   COMMAND "Delete" "Delete the current order"
      CALL del_order()
   COMMAND "Details" "Display details about current order"
      CALL det_order()
   COMMAND "Long_menu" "Display all menu options"
      SHOW OPTION ALL
   COMMAND "Short_menu" "Display a short menu"
      HIDE OPTION ALL
      SHOW OPTION "Query", "Details", "Long_menu", "Exit"
   COMMAND "Exit" "Exit the Order Management Form"
      EXIT MENU
END MENU
```

The HIDE OPTION and SHOW OPTION keywords are valid in a BEFORE MENU clause or in a COMMAND clause.

You must assign a value to a variable used to specify a menu option before you can include the variable in a HIDE OPTION statement.

Nested MENU Statements

You can nest MENU statements within MENU control blocks, so that the menus form a *tree* hierarchy. Nested MENU statements can appear either directly in a statement block or in 4GL functions that are called directly or indirectly when the user chooses options of the enclosing menu.

The END MENU Keywords

Use the END MENU keywords to indicate the end of the MENU statement. The END MENU keywords must follow the last statement in the last MENU control block. These keywords are required in every MENU statement. If you are nesting menus within menus, you must include a separate set of END MENU keywords to mark the end of each MENU statement construct.

If 4GL encounters the EXIT MENU statement within any MENU control block, control of execution is immediately transferred to the first statement that follows the END MENU keywords. (To terminate the current MENU control block without exiting from the MENU statement, use the CONTINUE MENU keywords, rather than END MENU or EXIT MENU.)

Identifiers in the MENU Statement

You can specify a character variable for the following items:

- The menu title
- The option name
- The option description
- The NEXT OPTION option name
- The SHOW OPTION or HIDE OPTION option name

Assignment statements can appear before 4GL executes the MENU statement or within the MENU statement. You can specify variable values in the BEFORE MENU block and in one or more of the subsequent MENU control blocks. Make sure, however, that a variable has a value before you include it in the MENU statement.

Keep the following considerations in mind if you change the value of a variable that was used as the menu title or as an option name in a MENU statement:

- 4GL determines the length of the menu title and of each option name when it first displays the menu. This length does not change during the MENU statement. If you subsequently assign a new value to a variable, 4GL displays as much of the new value as can fit in the existing space.

  For example, suppose that you assign the string **Short_Menu** (10 characters) to a variable, and later specify that variable as a menu title. If a subsequent statement in a control block of the same MENU statement assigns the new value **Very_Long_Menu** (14 characters) to the variable, 4GL displays only the first 10 characters of the new title.

  Similarly, if a second MENU control block assigns the value `Menu` (4 characters) to the variable that you specified as the menu title, 4GL displays the new title with 6 trailing blank spaces. For examples of using a variable as a menu title, an option name, and an option description in the MENU statement, see the program fragment in "Completing the MENU Statement" on page 4-268.

- If you use an array element (for example, **p_array[i]**) as a variable in a MENU statement, be aware that 4GL calculates the value of the index variable only once, *before* it first displays the menu. To index into the array, 4GL uses the value of the index variable after executing the BEFORE MENU block (if that block is included). Any subsequent changes to the index variable made in subsequent MENU CONTROL BLOCKS do *not* affect the way that 4GL evaluates the array element variable.

4GL produces a runtime error if the length of a variable or quoted string that specifies a menu name, an option name, or an option description exceeds the width of the current 4GL window.

Choosing a Menu Option

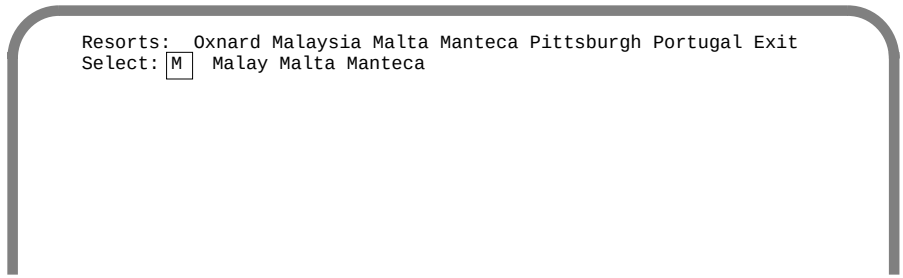
The user can choose a menu option in any of the following ways:

- Using the arrow keys to position the menu cursor on the option and pressing RETURN (See also "Scrolling the Menu Options" on page 4-266.)
- Typing a key sequence that the KEY clause associated with the option
- Typing the first letter or letters of the option name (regardless of whether the option is currently displayed on the screen)

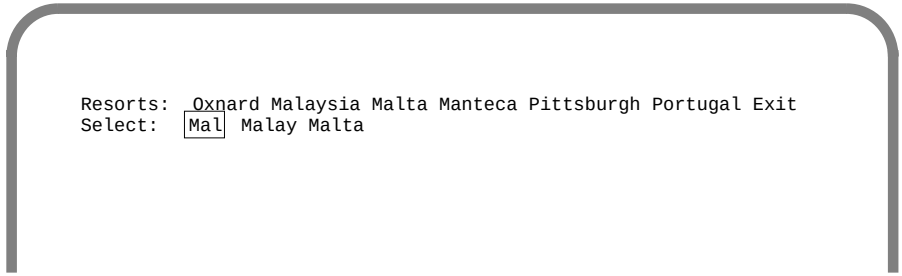
When the user types a letter, 4GL looks for a unique match among options:

- If only one option begins with the letter, or only one option is associated in a KEY clause with the letter, the choice is unambiguous. 4GL executes the commands associated with the option.
- If more than one option begins with the same letter, 4GL clears the second line of the menu and prompts the user to clarify the choice. 4GL displays each keystroke, followed by the names of the menu options that begin with the typed letters. When 4GL identifies a unique option, it closes this prompt line and executes the statements associated with the selected menu option.

For example, the next menu includes three options that begin with the letters `Ma`. The following screen is displayed when the user types the letter `M`:



When the user types Mal, 4GL drops **Manteca** from the list and displays the two remaining options:



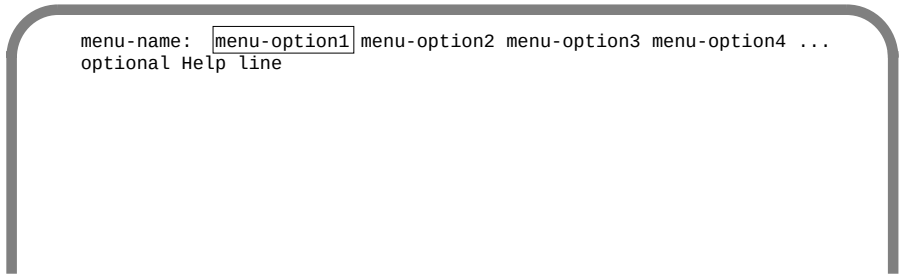
At this point, the user can type an a to select **Malay** or a t to select **Malta**.

The arrow keys have no effect when choosing among menu options that begin with the same letters. Pressing BACKSPACE deletes the keystroke to the left of the cursor.

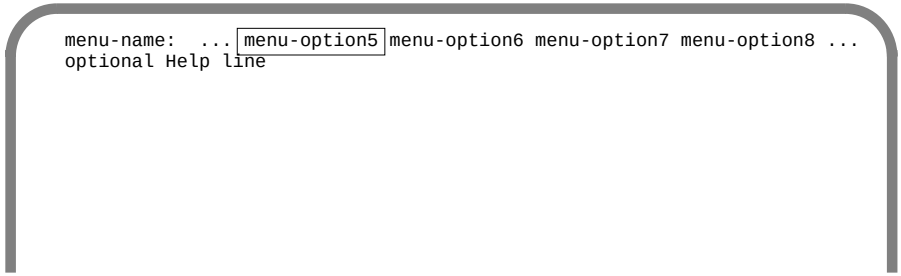
Scrolling the Menu Options

When 4GL displays a menu, it adds a colon (:) symbol and a blank space after the menu name, and a blank space before and after each menu option. If the width of the menu exceeds the number of characters that the screen or a 4GL window can display on a single line, 4GL displays the first page of options followed by ellipsis (...) points. This indicates that additional options exist.

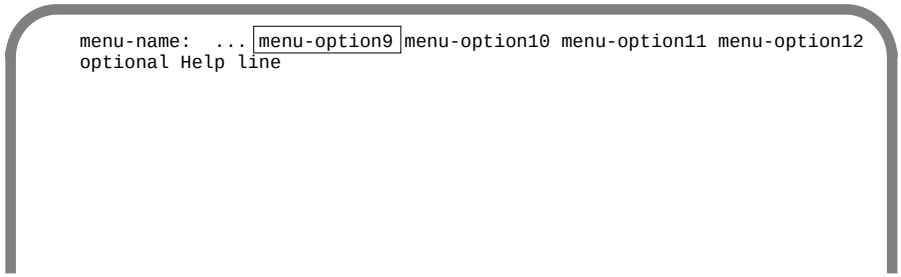
For example, the following menu displays an ellipsis:



If the user presses SPACEBAR or RIGHT ARROW to move past the right-most option (**menu-option4** in this case), 4GL displays the next page of menu options. In the following example, the ellipses at both ends of the menu indicate that more menu options exist in both directions:



If the user moves the highlight to the right past **menu-option8** in this example, 4GL displays a page of menu options:



Here no ellipsis appears at the right of the menu, because the user has come to the last page of the menu options. The user can display the previous page of menu options again by using ← to move the highlight past the left-most menu option, or can press → to move past the right-most option to display the first page, as if the first option followed the last. (This is why 4GL menus are called *ring menus*.)

The following keys can move through a menu.

Key	Effect
→, SPACEBAR	Moves the menu cursor to the next option. If the menu displays an ellipsis (…) on the right, pressing RIGHT ARROW from the right-most option displays the next page of menu options. If the last menu option is current and no ellipsis is on the right, RIGHT ARROW returns to the first option in the first page of menu options.
←	Moves the menu cursor to the previous option. If the menu displays an ellipsis (…) on the left, pressing LEFT ARROW from the left-most option displays the previous page of menu options. If the first menu option is current and no ellipsis is on the left, pressing LEFT ARROW returns to the last option on the last page of menu options.
↑	Moves the menu cursor to the first option on the previous menu page.
↓	Moves the menu cursor to the first option on the next page of options.

During interactive statements like INPUT, CONSTRUCT, or INPUT ARRAY, errors would be likely to result if the user could interrupt the interaction with menu choices. 4GL prevents this possibility by disabling the entire menu during the execution of these statements. The menu does not change its appearance when it is disabled.

Completing the MENU Statement

Any of the following actions can terminate the MENU statement:

- The user presses the Interrupt key.
- 4GL encounters the EXIT MENU statement.

By default, pressing the Interrupt key terminates program execution immediately. Unlike the CONSTRUCT, DISPLAY ARRAY, and INPUT statements, the MENU statement is not terminated by the Interrupt key if 4GL has executed the DEFER INTERRUPT statement. In these cases, an Interrupt signal causes 4GL to take the following actions:

- Set the global variable **int_flag** to a non-zero value.
- Remain in the MENU statement until EXIT MENU is encountered.

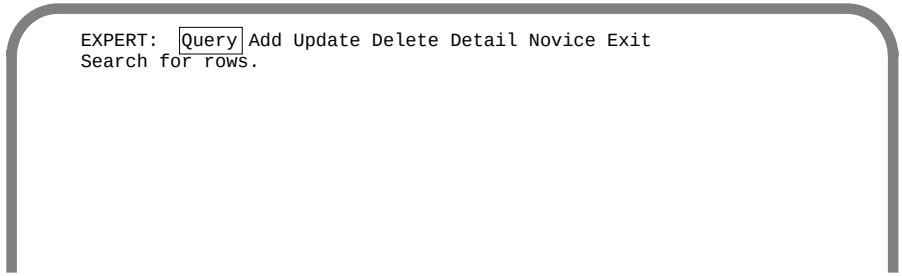
The EXIT MENU statement is typically included in a MENU control block that is activated when the user chooses an **Exit** or **Quit** option, as in the next example. If menus are *nested*, EXIT MENU terminates only the current MENU statement, passing control to the innermost enclosing MENU statement.

In the following program fragment, the MENU statement uses variables for the menu name, command name, and option description:

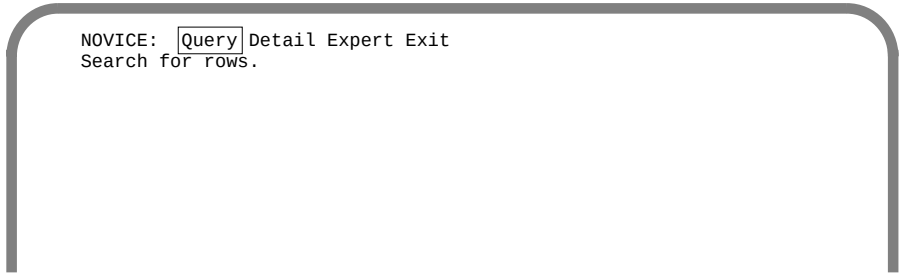
```
DEFINE menu_name, command_name CHAR(10),
      option_desc CHAR(30),
      priv_flag SMALLINT

LET menu_name = "NOVICE"
LET command_name = "Expert"
LET option_desc = "Display all menu options."

IF ... THEN
   LET priv_flag = 1
END IF

MENU menu_name
   BEFORE MENU
      HIDE OPTION ALL
      IF priv_flag THEN            -- expert user
         LET menu_name = "EXPERT"
         LET command_name = "Novice"
         LET option_desc = "Display a short menu."
         SHOW OPTION ALL
      ELSE                         -- novice user
         SHOW OPTION "Query", "Detail", "Exit", command_name
      END IF

   COMMAND "Query" "Search for rows." HELP 100
      CALL get_cust()
   COMMAND "Add" "Add a new row." HELP 101
      CALL add_cust()
   COMMAND "Update" "Update the current row." HELP 102
      CALL upd_cust()
      NEXT OPTION "Query"

   COMMAND "Delete" "Delete the current row." HELP 103
      CALL del_cust()
      NEXT OPTION "Query"
   COMMAND "Detail" "Get details." HELP 104
      CALL det_ord()
      NEXT OPTION "Query"
   COMMAND command_name option_desc HELP 105
      IF priv_flag THEN            -- EXPERT menu visible
         LET menu_name = "NOVICE"
         LET command_name = "Expert"
         LET option_desc = "Display all menu options."
         HIDE OPTION ALL
         SHOW OPTION "Query", "Detail", "Exit", command_name
         LET priv_flag = 0
      ELSE                         -- NOVICE menu visible
         LET menu_name = "EXPERT"
         LET command_name = "Novice"
         LET option_desc = "Display a short menu."
```

```
            SHOW OPTION ALL
            LET priv_flag = 1
         END IF
      COMMAND KEY ("!")
         CALL bang()
      COMMAND "Exit" "Leave the program." HELP 106
         EXIT MENU
   END MENU
```

These statements produce two menus. This is the EXPERT menu:



This is the simpler NOVICE menu:



COMMAND KEY Conflicts

In 4GL releases earlier than 6.x, the runtime MENU code gave inconsistent visual results and hung menus if a conflict existed between COMMAND KEY clauses of the same menu or between a COMMAND KEY clause and the default activation key of a COMMAND clause.

Because a single keystroke immediately activates the statements in a COMMAND KEY code structure (without waiting for you to press RETURN), no *key* specification in a COMMAND KEY clause can logically appear more than once in a given menu. The runtime code, however, did not check for such a programming error, and confusing prompts might be issued to the user if such an error existed in a menu.

The following example illustrates the improper coding methods and typical runtime results. Here is a conflict between two COMMAND KEY clauses:

```
MENU "main 1"
   COMMAND KEY (F3, "a")
      MESSAGE "This is F3 or <a> only"
      SLEEP 1
   COMMAND KEY (F3, CONTROL-F)
      MESSAGE "This is F3 or CONTROL-F"
      SLEEP 1
```

If the F3 key were pressed while this menu was active, the program entered an error state from which it could not recover. A submenu of the style used to resolve conflicts of default activation keys in COMMAND *option* clauses (as illustrated in "Choosing a Menu Option" on page 4-265) appeared inappropriately with two *invisible* prompts:

```
Select: \072  (invisible) (invisible)
```

Such conflicts between COMMAND KEY clauses always produced two (invisible) prompts, regardless of how many keys were acceptable to the COMMAND KEY clause. Any subsequent keystroke only rang the terminal bell, and so for the program to terminate, it had to be killed with a signal (such as the Interrupt or Quit key).

A related problem in 4GL releases earlier than 6.x is conflict between a COMMAND clause and a COMMAND KEY clause, where the first character of the COMMAND *option*, whether specified as a literal or as a variable, conflicts with a printable character that can activate a COMMAND KEY clause. If the user pressed an ambiguous key (b in the following example), a submenu appeared with one side showing invisible:

```
MENU "main 1"
   COMMAND KEY (F3, "a", F22, F23)
      MESSAGE "This is F3, <a>, F22, or F23 only"
      SLEEP 1
   COMMAND KEY (F4, "b")
      MESSAGE "This is F4 or <b> only"
      SLEEP 1
   COMMAND "bark"  "collides with command key (f4, b)"
      MESSAGE "This collides with command key (f4, b)"
      SLEEP 1
```

If the proper second letter was pressed (in this case), the menu proceeded normally; any other keystroke simply activated the terminal bell.

In this release, the runtime menu library detects such collisions. If such a conflict occurs, error -1176 is issued:

```
,A COMMAND KEY value occurs elsewhere in the current menu
```

and the program terminates.

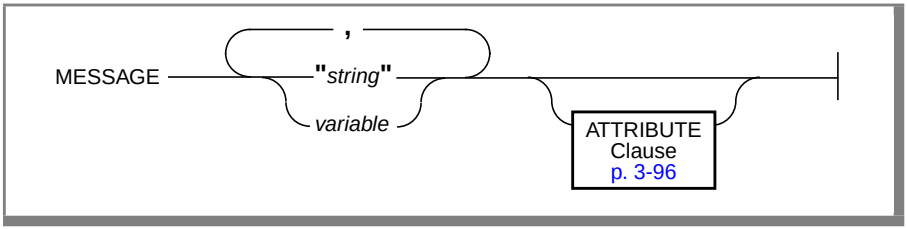
If you encounter error -1176, it means that a COMMAND KEY conflict already exists in that menu. You need to revise the offending COMMAND KEY clauses to remove the conflict.

References

CONTINUE, DEFER, OPEN WINDOW, OPTIONS

MESSAGE

The MESSAGE statement displays a character string on the Message line.



Element	Description
string	is a quoted string that contains message text.
variable	is a CHAR or VARCHAR variable that contains message text.

Usage

You can specify any combination of variables and strings for the message text. 4GL generates the message to display by replacing any variables with their values and concatenating the strings. If the length of the message text exceeds the width of the screen or 4GL window, the text is truncated to fit.

The Message Line

4GL displays message text in the Message line. 4GL positions this reserved line according to default or explicit Message line specification for the program or for the current 4GL window, in this order of descending precedence:

1. A MESSAGE LINE specification in the most recent OPTIONS statement
2. A MESSAGE LINE specified in the most recent OPEN WINDOW statement
3. The default Message line, or the *second* line of the current 4GL window

The message remains on the screen until you display a menu or another message.

To clear the Message line, you can display a blank message, like this:

```
MESSAGE " "
```

You can include the CLIPPED and USING operators in a MESSAGE statement. For example, the following MESSAGE statement uses the CLIPPED operator to remove any trailing blanks from the string in the variable **file_name**:

```
DEFINE file_name CHAR(20)
...
MESSAGE "Printing mailing labels to", file_name CLIPPED,
   " -- Please wait"
```

You can also use the ASCII and COLUMN operators. For information on using the 4GL built-in functions and operators, see Chapter 5.

If you position the Message line so that it coincides with the Comment line, Menu line, or fields of a form, output from the MESSAGE statement is not visible. For example:

```
DATABASE stores
MAIN
   DEFINE p_customer RECORD LIKE customer.*
   OPEN WINDOW r1 AT 4,1 WITH FORM "platonic"
      ATTRIBUTE (MESSAGE LINE LAST)
   MESSAGE "This is a word to the wise."
   INPUT BY NAME p_customer.*
   CLOSE WINDOW r1
END MAIN
```

This program does not display the text of the MESSAGE statement, because the default position of the Comment line is also the last line. If the ATTRIBUTE clause of OPEN WINDOW in the same example were revised to specify

```
ATTRIBUTE (MESSAGE LINE LAST, COMMENT LINE FIRST)
```

so that there was no conflict between those reserved lines, the message text would appear when the MESSAGE statement was executed. For a description of the syntax used to position reserved lines, see the "Positioning Reserved Lines" sections of the OPEN WINDOW and OPTIONS statements.

The ATTRIBUTE Clause

For general information about syntax, see "The ATTRIBUTE Clause" on page 4-41. This section describes specific information about using the ATTRIBUTE clause within a MESSAGE statement.

The default display attribute for the Message line is the NORMAL display. You can use the ATTRIBUTE clause to alter the default display attribute of the Message line. For example, the following statement changes the display attribute of the message text to reverse video:

```
MESSAGE "Please enter a value " ATTRIBUTE (REVERSE)
```

4GL ignores the INVISIBLE attribute if you include it in the ATTRIBUTE clause of the MESSAGE statement.

You can refer to substrings of CHAR, VARCHAR, and TEXT type variables by following the variable name with a pair of integers to indicate the starting and ending position of the substring, enclosed between brackets ([ ]) and separated by a comma. For example, the following MESSAGE statement displays a 10-character substring of the **full_name** variable:

```
MESSAGE "Customer ", full_name[11,20]
   CLIPPED, " added to the database"
```

Statements in the next program fragment perform the following tasks:

1. Use a MESSAGE statement to clear the Message line of any text.
2. Clear all the fields of the current form.
3. Use a PROMPT statement to instruct the user to type a name.
4. Assign the value of the entered string to the variable **last_name**.
5. Use another MESSAGE statement to indicate to the user that the program is retrieving rows.
6. Clear the second message after a three -second delay:

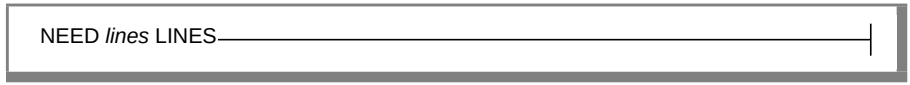
```
MESSAGE ""
CLEAR FORM
PROMPT "Enter a last name:" FOR last_name
MESSAGE "Selecting rows for customer with last name ",
   last_name, ". . ." ATTRIBUTE (YELLOW)
SLEEP 3
MESSAGE ""
```

References

DISPLAY, ERROR, OPEN WINDOW, OPTIONS, PROMPT

NEED

NEED is a conditional statement to control output from the PRINT statement. (The NEED statement can appear only in a REPORT program block.)



Element	Description
lines	is an expression, as described in "Integer Expressions" on page 3-63, that specifies how many lines must remain in the current page between the line above the current character position and the bottom margin.

Usage

The NEED statement causes subsequent report output from the PRINT statement to start on the next page of the report, if fewer than the specified number of available lines remain between the current line of the page and the bottom margin. NEED has the effect of a conditional SKIP TO TOP OF PAGE, the condition being that the number returned by the integer expression must be greater than the number of lines that remain on the current page.

The NEED statement can prevent 4GL from separating parts of the report that you want to keep together on a single page. In the following example, the NEED statement causes the PRINT statement to send output to the next page, unless at lease six lines remain on the current page:

```
AFTER GROUP OF r.order_num
   NEED 6 LINES
   PRINT " ",r.order_date, 7 SPACES,
      GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
```

NEED does not include the BOTTOM MARGIN value in calculating the lines available. If the number of lines remaining above the bottom margin on the page is less than *lines*, both the PAGE TRAILER and the PAGE HEADER are printed before the next PRINT statement is executed. You cannot include the NEED statement in the PAGE HEADER or PAGE TRAILER control blocks.

References

PAUSE, PRINT, REPORT, SKIP

OPEN FORM

The OPEN FORM statement declares the name of a compiled 4GL form.

Element	Description
filename	is a quoted string that specifies the name of a file that contains the compiled screen form. This can also include a pathname.
form	is a 4GL identifier that you assign here as the name of the form.

Usage

The following steps describe how to display a form:

1. Create a form specification file (with a **.per** extension).
2. Compile the form by using the **Compile** option of the **Form** menu in the Programmer's Environment or by using the **form4gl** command.

   The compiled form file has **.frm** as its file extension.
3. Declare the form name by using the OPEN FORM statement.
4. Display the form by using the DISPLAY FORM statement.

Once 4GL displays the form, you can activate the form by executing the CONSTRUCT, DISPLAY ARRAY, INPUT, or INPUT ARRAY statement.

When it executes the OPEN FORM statement, 4GL loads the compiled form into memory. (The CLOSE FORM statement is a memory-management feature to recover memory from forms that 4GL no longer displays on the screen.)

Specifying a Filename

The quoted string that follows the FROM keyword must specify the name of the file that contains the compiled screen form. This filename can include a pathname. You can omit or include the **.frm** extension:

```
OPEN FORM frmofmor FROM "/fomr/fmro.frm"
```

The Form Name

The form name need not match the name of the form specification file, but it must be unique among form names in the program. Its scope of reference is the entire program. For more information, see "4GL Identifiers" on page 2-14.

Displaying a Form in a 4GL Window

To position the form in a 4GL window, precede the OPEN FORM statement with the OPEN WINDOW statement. The following program fragment opens the **w_cust1** window, opens and displays the **o_cust** form in that 4GL window, and calls the **cust_order( )** function. When the function returns, the CLOSE WINDOW statement closes both the form and the 4GL window:

```
MAIN
   OPEN WINDOW w_cust1 AT 10,15
      WITH 11 ROWS, 63 COLUMNS
      ATTRIBUTE (BORDER)
   OPEN  FORM  o_cust  FROM  "custorder"
   DISPLAY FORM o_cust
   CALL cust_order()
   CLOSE WINDOW w_cust1
END MAIN
```

If you execute an OPEN FORM statement with the name of an open form, 4GL first closes the existing form before opening the new form.

The WITH FORM keywords of OPEN FORM both open and display a form in a 4GL window. You do not need to execute the OPEN FORM, DISPLAY FORM, and CLOSE FORM statements if you use the OPEN WINDOW statement to display the form. You also do not need to use the CLOSE FORM statement to release the memory allocated to the form. Instead, you can use the CLOSE WINDOW statement to close both the form and the 4GL window, and to release the memory. For example, the following statements open 4GL window **w_cust2**, call function **cust_order( )**, and then close the 4GL window:

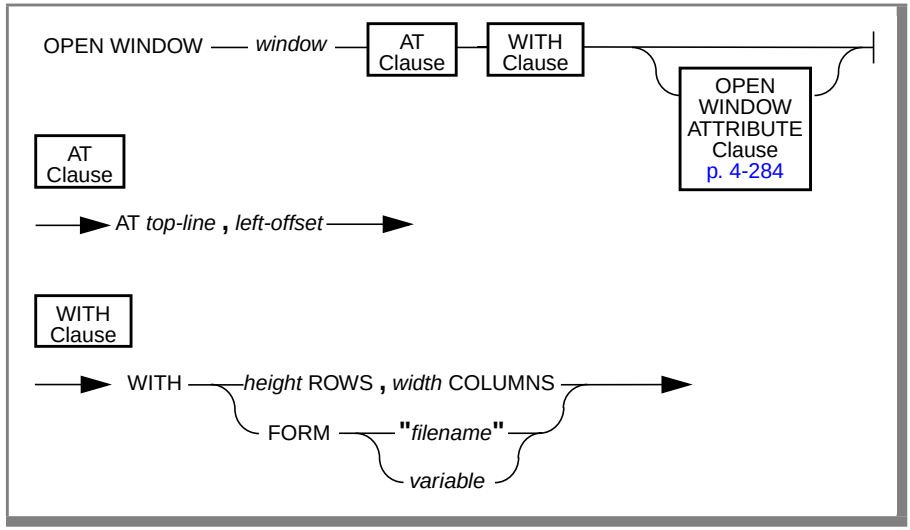
```
OPEN WINDOW w_cust2 AT 10,15 WITH FORM "custorder"
CALL cust_order()
CLOSE WINDOW w_cust2
```

References

CLEAR, CLOSE FORM, CLOSE WINDOW, CURRENT WINDOW, OPEN WINDOW, OPTIONS

OPEN WINDOW

The OPEN WINDOW statement declares and opens a 4GL window.



Element	Description
filename	is a quoted string that specifies the file containing a compiled 4GL form. This can include a pathname and file extension.
height	is an integer expression to specify the height, in lines.
left-offset	is an integer expression to specify the position of the left margin, in characters, where 0 = the left edge of the 4GL screen.
top-line	is an integer expression to specify the position of the top line of the 4GL window, where 0 = the top of the 4GL screen.
variable	is a CHAR or VARCHAR variable that specifies the *filename*.
width	is an integer expression to specify the width, in characters.
window	is the identifier declared here for the 4GL window to be opened.

Usage

A *4GL window* is a rectangular area in the 4GL screen that can display a form, a menu, or output from the DISPLAY, MESSAGE, or PROMPT statement. The 4GL screen can display one or more 4GL windows concurrently.

An OPEN WINDOW statement can have the following effects:

- Declares a name for the 4GL window
- Specifies the position of the 4GL window on the 4GL screen
- Defines the dimensions of the 4GL window, in lines and characters
- Specifies the display attributes of the 4GL window

The *window* identifier must follow the rules for 4GL identifiers (as described in "4GL Identifiers" on page 2-14) and be unique among 4GL windows in the program. Its scope is the entire program. You can use this identifier to reference the same 4GL window in other statements (for example, CLEAR, CURRENT WINDOW, and CLOSE WINDOW).

The following topics are described in this section:

The 4GL Window Stack

4GL maintains a *window stack* of all open 4GL windows. If you execute OPEN WINDOW to open a new 4GL window, 4GL takes the following actions:

- Saves any changes made to the current 4GL window
- Adds the new 4GL window to the window stack
- Makes the new 4GL window the current 4GL window

Other statements that can modify the window stack are CURRENT WINDOW and CLOSE WINDOW.

The AT Clause

The AT clause specifies the location of the top-left corner of the 4GL window. The location is relative to the entire 4GL screen and is independent of the position of any other 4GL windows.

You must specify these coordinates as expressions that return positive integers within the following ranges:

- The first expression must return an integer between 1 and (*max* - *lines*), where *max* is the maximum number of lines in the 4GL screen, and *lines* is the ROWS specification. The window begins on this line.
- The second expression must return a whole number between 1 and (*length* - *characters*), where *length* is the maximum number of characters that the 4GL screen can display on one line, and *characters* is the COLUMNS specification. This is the left margin.

A comma separates the two expressions in the AT clause. For example, the following statement opens a 4GL window with the top-left corner at the third line and the fifth character position of the 4GL screen:

```
OPEN WINDOW o1 AT LENGTH("Mom"), 5 WITH 10 ROWS, 40 COLUMNS
```

The WITH ROWS, COLUMNS Clause

The WITH *lines* ROWS, *characters* COLUMNS clause specifies explicit vertical and horizontal dimensions for the 4GL window:

- The expression at the left of the ROWS keyword specifies the height of the 4GL window, in lines. This must be an integer between 1 and *max*, where *max* is the maximum number of lines that the 4GL screen can display.
- The integer expression after the comma at the left of the COLUMNS keyword specifies the width of the 4GL window, in characters. This must return a whole number between 1 and *length*, where *length* is the number of characters that your monitor can display on one line.

This statement opens a 4GL window 5 lines high and 74 characters wide:

```
OPEN WINDOW w2 AT 10, 12 WITH 5 ROWS, 74 COLUMNS
```

In addition to the lines needed for a form, allow room for the following reserved lines:

- The Comment line. (By default, this is the last line of the 4GL window.)
- The Form line. (By default, this is line 3 of the 4GL window.)
- The Error line. (By default, this is the last line of the 4GL screen, not of the 4GL window.)

4GL issues a runtime error if the 4GL window does not include sufficient lines in to display both the form and these additional reserved lines. To reduce the number of lines required by 4GL, you can define the Form line as line 1 or 2, and change other reserved lines accordingly, such as the Prompt and Menu lines. For information on how to make these changes, see "The OPEN WINDOW ATTRIBUTE Clause" on page 4-284.

The minimum number of lines required to display a form in a 4GL window is the number of lines in the form, plus an additional line below the form for prompts, messages, and comments.

The WITH FORM Clause

As an alternative to specifying explicit dimensions, the WITH FORM clause can specify a quoted string or a character variable that specifies the name of a file that contains the compiled screen form. You can omit or include the **.frm** file extension. 4GL automatically opens a 4GL window sized to the screen layout of the form (as described in "The Screen Layout" on page 6-17) and displays the form.

If you include a WITH FORM clause, the width of the 4GL window is from the left-most character on the screen form (including leading blank spaces) to the right-most character on the screen form (truncating trailing blank spaces).

The length of the 4GL window is the following sum:

(*form line*) + (*form length*)

Here *form line* is the reserved line position on which to display the first line of the form (by default, line 3) and *form length* is the number of lines in the screen layout of the SCREEN section of the form specification file. 4GL adds one line for the Comment line. Unless you specify FORM LINE in an ATTRIBUTE clause or in the OPTIONS statement, the default value of this sum is *form length* + 2. (For more information on screen layouts in 4GL forms, see "SCREEN Section" on page 6-15.)

For example, the following statement opens a 4GL window called **w1** and positions its top-left corner at the fifth row and fifth column of the 4GL screen. The WITH FORM clause opens and displays the **custform** form in this 4GL window. If **custform** were 10 lines long and the FORM LINE option were the default value (3), the height of **w1** would be (10 + 3) = 13 lines:

```
OPEN WINDOW w1 AT 5, 5 WITH FORM "custform"
```

The WITH FORM clause is convenient if the 4GL window always displays the same form. If you use this clause, you do not need the OPEN FORM, DISPLAY FORM, or CLOSE FORM statement to open and close the form. The OPEN WINDOW WITH FORM statement opens and displays the form. The CLOSE WINDOW statement closes the 4GL window and the form.

You *cannot* use the WITH FORM clause for the following purposes:

- To display more than one form in the same 4GL window
- To display a 4GL window larger than the default dimensions (as described earlier) when 4GL executes the WITH FORM clause

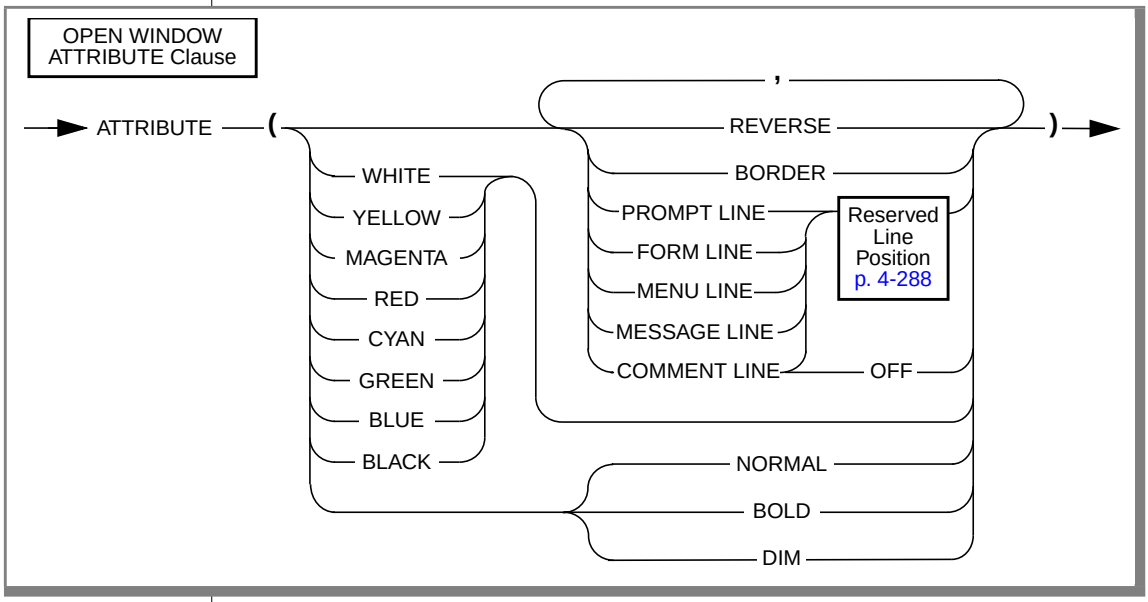
In these cases, you must specify explicit dimensions by using the WITH *lines* ROWS, *characters* COLUMNS clause. You must also execute the OPEN FORM, DISPLAY FORM, and CLOSE FORM statements to open, display, and close the form or forms explicitly. (You typically are not required to use the CLOSE FORM statement, which affects memory management, rather than the visual interface of your program.)

The OPEN WINDOW ATTRIBUTE Clause

Use the OPEN WINDOW ATTRIBUTE clause to perform the following tasks:

- Specify a border for the 4GL window
- Display the 4GL window in reverse video or in a color
- Reposition the Prompt, Message, Menu, Form, and Comment lines

The OPEN WINDOW ATTRIBUTE clause has the following syntax.



The color attributes are listed in the left-hand portion of the diagram. Besides these, you can also specify INVISIBLE as a color, but this specification has no effect in the OPEN WINDOW ATTRIBUTE clause. Without this clause, the attributes and reserved line positions have the following default values.

Attribute	Default Setting
Color	The default foreground color on your terminal
REVERSE	No reverse video
BORDER	No border
PROMPT LINE *line value*	FIRST (=1)
MESSAGE LINE line value	FIRST + 1 (=2)
MENU LINE line value	FIRST (=1)
FORM LINE line value	FIRST + 2 (=3)
COMMENT LINE line value	LAST - 1 (for the 4GL screen) LAST (for all other 4GL windows)

For more information on valid *reserved line* values, see "Positioning Reserved Lines" on page 4-288. For more information about color and intensity attributes, see Chapter 3.

If you specify a color or the REVERSE attribute in the ATTRIBUTE clause of an OPEN WINDOW statement, it becomes the default attribute for displays in the 4GL window, except for menus. You can override this default by specifying a different attribute in the ATTRIBUTE clause of the CONSTRUCT, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, INPUT, or INPUT ARRAY statement.

The Color and Intensity Attributes

Display attributes can be classified as color and intensity (or monochrome) attributes. The color attributes described earlier override the default foreground color on your terminal. On monochrome monitors, all color attributes except BLACK are displayed as WHITE.

4GL displays the intensity attributes as follows on color monitors.

Attribute	Displayed As
NORMAL	WHITE
BOLD	RED
DIM	BLUE

For example, if you have a color monitor, the 4GL window specified in the following statement is displayed with the BLUE attribute:

```
OPEN WINDOW w2 AT 10, 12 WITH 5 ROWS, 40 COLUMNS ATTRIBUTE (BLUE)
```

On a monochrome display, the BLUE attribute produces a *white* 4GL window.

The REVERSE Attribute

Use the REVERSE attribute to display the foreground of the 4GL window in reverse video (sometimes called *inverse video*). The following statement assigns the BLUE and REVERSE attributes to the **w2** window:

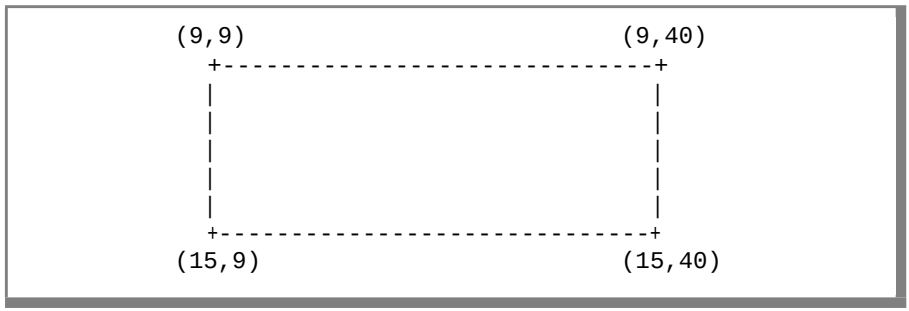
```
OPEN  WINDOW w2 AT 10, 12 WITH 5 ROWS, 40 COLUMNS
   ATTRIBUTE (BLUE,REVERSE)
```

The BORDER Attribute

The BORDER attribute draws a border *outside* the specified 4GL window. The border requires two lines on the screen (one above and another below the window) and two character positions (one to the left and one to the right of the window). Make sure to account for this space when you specify coordinates in the AT clause. For example, the following statement opens a 4GL window and displays a border around it:

```
OPEN WINDOW w1 AT 10,10 WITH 5 ROWS, 30 COLUMNS ATTRIBUTE (BORDER)
```

The following diagram indicates the coordinates of the border enclosing the 5 x 30 4GL window that was specified in the preceding example:



The coordinates of the top-left corner of the window border are 9, 9. The 4GL window itself starts at 10, 10.

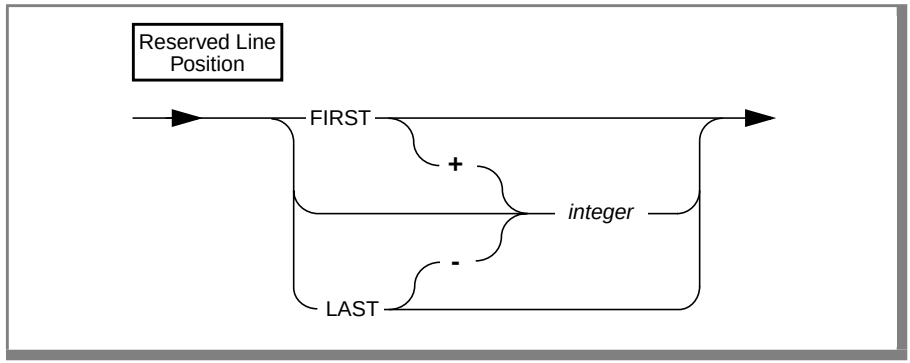
4GL draws the window with the characters defined in the **termcap** or **terminfo** file. You can specify alternative characters in these files. Otherwise, 4GL uses the hyphen (-) for horizontal lines, the vertical bar (|) for vertical lines, and the plus sign (+) for corners. Some **termcap** or **terminfo** files have settings that require additional rows and columns to display windows. For more information, see Appendix F, "Modifying termcap and terminfo."

If a window and its border exceed the physical limits of the screen, a runtime error occurs.

See also the built-in FGL_DRAWBOX( ) function, which displays rectangles (in Chapter 5).

Positioning Reserved Lines

The Reserved Line Position segment has the following syntax.



Line values specified in the OPTIONS ATTRIBUTE clause of the most recently executed OPTIONS statement can position the Form, Prompt, Menu, Message, Comment, and Error lines. (For more information, see "Positioning Reserved Lines" on page 4-295.) If no line positions are specified in the OPTIONS ATTRIBUTE or OPEN WINDOW ATTRIBUTE clauses, the 4GL window uses the following default positions for its reserved lines.

Default Location	Reserved for
First line	Prompt line (output from PROMPT statement); also Menu line (*command value* from MENU statement)
Second line	Message line (output from MESSAGE statement; also the *description value* output from MENU statement)
Third line	Form line (output from DISPLAY FORM statement)
Last line	Comment line in any 4GL window except SCREEN

These positional values are relative to the first or last line of the 4GL window, rather than to the 4GL screen. (The Error line is always the last line of the 4GL screen.) When you open a new 4GL window, however, the OPEN WINDOW ATTRIBUTE clause can override these defaults for every reserved line (except the Error line). This disables the OPTIONS statement reserved line specifications only for the specified 4GL window.

Except for the cases that are described later in this section, you can specify any of the following *positions* for the reserved lines of 4GL:

- FIRST
- FIRST + *integer*
- *integer*
- LAST - *integer*
- LAST

Here *integer* is a literal or variable that returns a positive whole number, such that the LINE specification is no greater than the number of lines in the 4GL window. This is true for all reserved lines except:

- The Menu line: do not specify LAST

  A menu requires two lines. The menu *title* and *commands* appear on the Menu line, and *command description* appears on the next line. To display a menu at the bottom of a 4GL window, specify MENU LINE LAST - 1.
- The Form line: do not specify LAST or LAST - *integer*

FIRST is the first line of the 4GL window (line 1), and LAST is the last line. The following statement sets three reserved line positions:

```
OPEN WINDOW wcust AT 3,6 WITH 10 ROWS, 50 COLUMNS
ATTRIBUTE (MESSAGE LINE 20,
      PROMPT LINE LAST-2,
      FORM LINE FIRST)
```

If a 4GL window is not large enough to contain the specified value for one or more of these reserved lines, 4GL increases its line value to FIRST or decreases it to LAST, whichever is appropriate.

If the 4GL window is not wide enough to display all the text that you specify, 4GL truncates the message. You can use these features to display text:

- PROMPT statement
- MESSAGE statement
- DISPLAY statement
- COMMENTS attribute of a screen form

Because the position of the Error line is relative to the 4GL screen, rather than to the current 4GL window, the ATTRIBUTE clause of an OPEN WINDOW statement cannot change the location of the Error line. Use the OPTIONS statement to change the position of the Error line. (For details, see "Features Controlled by OPTIONS Clauses" on page 4-292.)

Because the INPUT statement clears both the Comment line and the Error line when moving between fields, do not use either of the following settings for the Message or Prompt line:

- The last line of the 4GL window (the default Comment line)
- The last line of the 4GL screen (the default Error line)

If you intend to use these lines for messages or prompts, be sure to redefine the Comment and Error lines too.

Hiding the Comment Line

By default, the last line of the current 4GL window is the Comment line, which can display messages that the COMMENTS attribute of a 4GL form specifies. The Comment line is a reserved line, which is cleared when the user moves the visual cursor to a new line of the current screen form.

It is typically used to send messages to the user, rather than for data entry or data display. In 4GL forms that do not use the COMMENTS attribute, the Comment line is unused space on the screen.

You can conserve display space within a 4GL window by hiding the Comment line. The syntax to do this (in the OPEN WINDOW ATTRIBUTE clause) is:

```
COMMENT LINE OFF
```

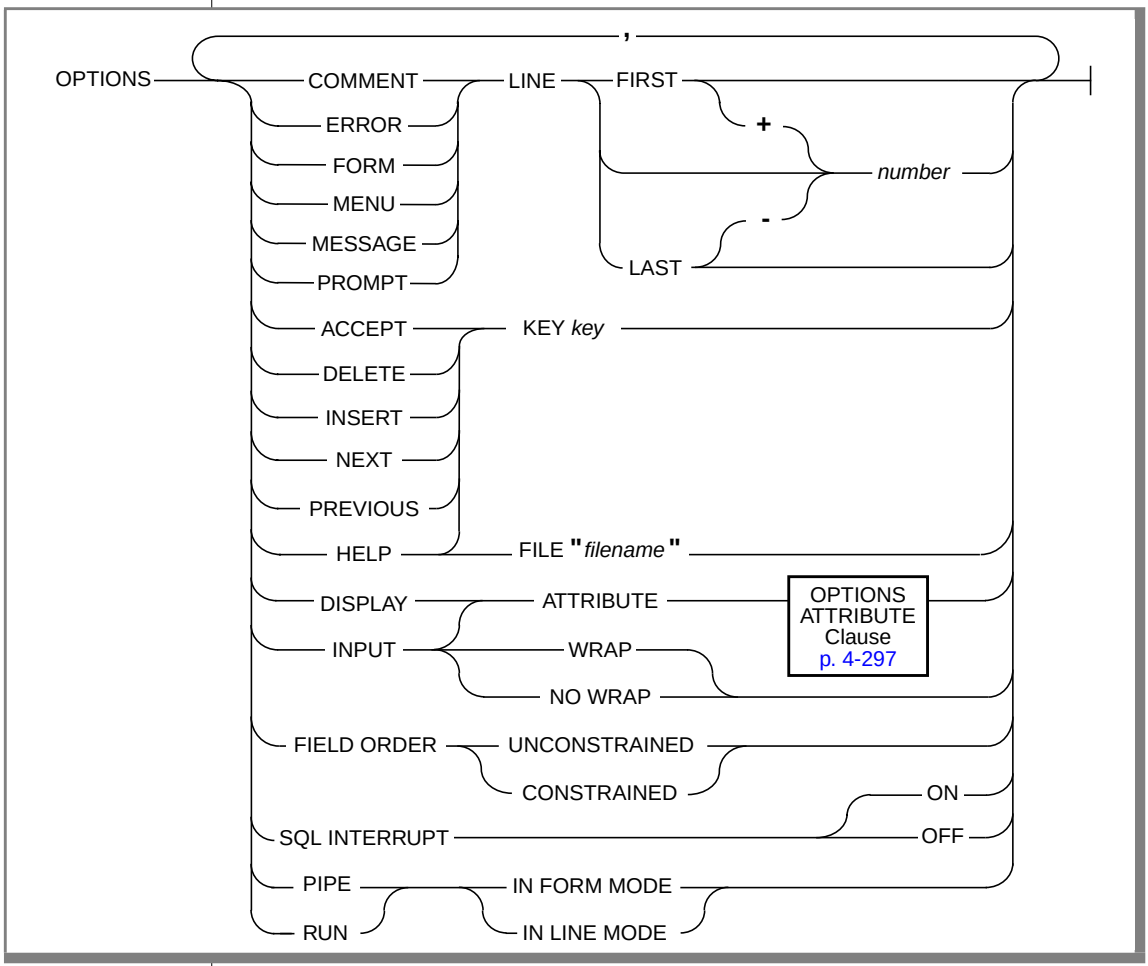
If you use this syntax, the Comment line is hidden for that 4GL window and cannot display messages from the form specification, even if some fields of a form that this window displays have the COMMENTS attribute.

References

CLEAR, CLOSE FORM, CLOSE WINDOW, CURRENT WINDOW, DISPLAY, MESSAGE, OPEN FORM, OPTIONS, PROMPT

OPTIONS

The OPTIONS statement sets default features of screen interaction statements.



Element	Description
filename	is a quoted string that specifies the name of a file that contains the compiled help messages. This can include a pathname.
key	is a keyword to specify a physical or logical key.
number	is a literal integer to specify a line number.

Usage

The OPTIONS statement specifies default features for form-related statements and for other 4GL screen-interaction statements.

The following topics are described in this section:

Features Controlled by OPTIONS Clauses

A program can include several OPTIONS statements. If these statements conflict in their specifications, the OPTIONS statement most recently encountered at runtime prevails. OPTIONS can specify the following features of other 4GL statements, including CONSTRUCT, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, ERROR, INPUT, INPUT ARRAY, MESSAGE, OPEN FORM, OPEN WINDOW, PROMPT, REPORT, RUN, and START REPORT:

- Positions of the reserved lines of 4GL
- Input and display attributes
- Logical key assignments
- The name of the Help file
- Whether SQL statements can be interrupted
- Field traversal constraints
- The default screen display mode

If you omit the OPTIONS statement, 4GL uses defaults that are described in the following table.

Clause	Effect
COMMENT LINE	Specifies the position of the Comment line. This displays messages defined with the COMMENT attribute in the form specification file. The default is (LAST - 1) for the 4GL screen, and LAST for all other 4GL windows.
ERROR LINE	Specifies the position in the 4GL screen of the Error line that displays text from the ERROR statement. The default is the LAST line of the 4GL screen.
FORM LINE	Specifies the position of the first line of a form. The default is (FIRST + 2), or line 3 of the current 4GL window.
MENU LINE	Specifies the position of the Menu line. This displays the menu name and options, as defined by the MENU statement. The default is the FIRST line of the 4GL window.
MESSAGE LINE	Specifies the position of the Message line. This reserved line displays the text listed in the MESSAGE statement. The default is (FIRST + 1), or line 2 of the current 4GL window.
PROMPT LINE	Specifies the position of the Prompt line, to display text from PROMPT statements. The default value is the FIRST line of the 4GL window.
ACCEPT KEY	Specifies the key to terminate an CONSTRUCT, INPUT, INPUT ARRAY, or DISPLAY ARRAY statement. The default is ESCAPE.
DELETE KEY	Specifies the key in INPUT ARRAY statements to delete a screen record. The default Delete key is F2.
INSERT KEY	Specifies the key to open a screen record for data entry in INPUT ARRAY. The default Insert key is F1.
NEXT KEY	Specifies the key to scroll to the next page of a program array of records in an INPUT ARRAY or DISPLAY ARRAY statement. The default Next key is F3.

(1 of 2)

Clause	Effect
PREVIOUS KEY	Specifies the key to scroll to the previous page of program records in an INPUT ARRAY or DISPLAY ARRAY statement. The default Previous key is F4.
HELP KEY	Specifies the key to display help messages. The default Help key is CONTROL-W.
HELP FILE	Specifies the file (produced by the **mkmessage** utility) containing programmer-defined help messages.
DISPLAY ATTRIBUTE	Specifies default attributes to use during a DISPLAY or DISPLAY ARRAY statement when none is specified by those statements or in the form specification file.
INPUT ATTRIBUTE	Specifies the attributes to use during a CONSTRUCT or INPUT statement when no attributes are specified by those statements or in the form specification file.
INPUT NO WRAP	Specifies that the cursor does not wrap. An INPUT or CONSTRUCT statement terminates when a user presses RETURN after the last field. This is the default value.
INPUT WRAP	Specifies that the cursor wraps between the last and first input fields during INPUT, INPUT ARRAY, and CONSTRUCT statements, until the user presses the Accept key. Pressing RETURN at the last field does not deactivate the form.
FIELD ORDER CONSTRAINED	Specifies that the UP ARROW key moves the cursor to the previous field and the DOWN ARROW key moves the cursor to the next field when users enter values for CONSTRUCT or INPUT statements.
FIELD ORDER UNCONSTRAINED	Specifies that the UP ARROW key moves the cursor to the field above the current position and the DOWN ARROW key moves the cursor to the field below the current cursor position when users enter values for CONSTRUCT or INPUT statements.
SQL INTERRUPT ON	Specifies that the user can interrupt SQL statements as well as 4GL statements.
SQL INTERRUPT OFF	Specifies that the user cannot interrupt SQL statements.

(2 of 2)

Positioning Reserved Lines

Except for the cases that are described later in this section, you can specify any of the following positions for each reserved line of 4GL:

- FIRST
- FIRST + *integer*
- *integer*
- LAST - *integer*
- LAST

Here *integer* is a variable or a literal that returns a positive whole number, such that the LINE specification is no greater than the number of lines in the 4GL window or 4GL screen, except for these reserved lines:

- The Form line: do not specify LAST or LAST - *integer*
- The Menu line: do not specify LAST

A menu requires two lines. The menu *title* and *commands* appear on the Menu line, and the *command description* appears on the following line. If you want a menu to appear at the bottom of a 4GL window, specify MENU LINE LAST - 1.

FIRST is the top line of the current 4GL window (line 1), and LAST is the last line. For example, the following statement sets three reserved line positions:

```
OPTIONS MENU LINE 20, PROMPT LINE LAST-2, FORM LINE FIRST
```

The line position for the Error line is relative to the 4GL screen, rather than to the current 4GL window. The line value of any other reserved line is relative to the first line of the current 4GL window (or to the 4GL screen, if that is the current 4GL window). If the 4GL window is not wide enough to display all the message text that you specify, 4GL truncates the message. You can use these features of 4GL to display message text:

- PROMPT statement
- MESSAGE statement
- DISPLAY statement
- ERROR statement
- COMMENTS attribute of a form specification file

Because the INPUT statement clears both the Comment line and the Error line when the cursor moves between fields, it is not a good idea to set the Message line or the Prompt line to either of the following positions:

- The last line of the current 4GL window (the default Comment line)
- The last line of the 4GL screen (the default Error line)

If a 4GL window is not large enough to contain the specified value for one or more of these reserved lines, 4GL automatically decreases the position value to FIRST or increases it to LAST, as appropriate. If the value that you specify for the Prompt line exceeds the number of rows in the current window, PROMPT LINE is set to its default value, the first row of the window.

Default line positions set by OPTIONS remain in effect until another OPTIONS statement redefines them. They can also be reset by the ATTRIBUTE clause of the OPEN WINDOW statement (as described in "Positioning Reserved Lines" on page 4-288), but only for the specified 4GL window; after it closes, the reserved line positions are restored to their values from the most recently executed OPTIONS statement.

Cursor Movement in Interactive Statements

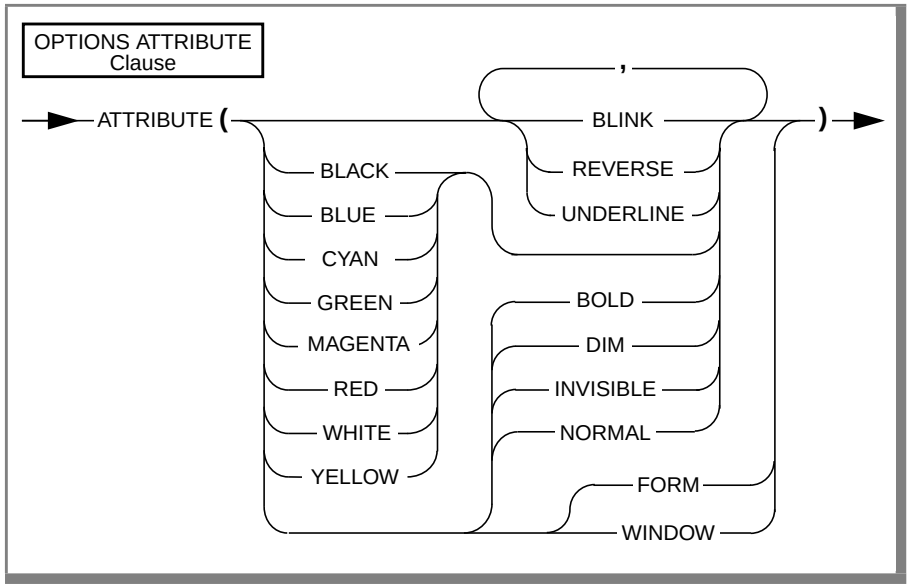
The tab order in which the screen cursor visits fields of a form is that of the *field list* of currently executing CONSTRUCT, INPUT, or INPUT ARRAY statements, except as modified by NEXT FIELD clause. By default, the interactive statement terminates if the user presses RETURN in the last field (or if entered data fills the last field, if that field has the AUTONEXT attribute).

The INPUT WRAP keywords change this behavior, causing the cursor to move from the last field to the first, repeating the sequence of fields until the user presses the Accept key. The INPUT NO WRAP option restores the default cursor behavior.

Specify FIELD ORDER UNCONSTRAINED to cause the UP ARROW and DOWN ARROW keys to move the cursor to the field above or below, respectively. Use the FIELD ORDER CONSTRAINED option to restore the default behavior of the UP ARROW and DOWN ARROW keys moving the cursor to the previous or next field, respectively.

The OPTIONS ATTRIBUTE Clause

This section describes the OPTIONS ATTRIBUTE clause. It explains the FORM keyword and the WINDOW keyword in detail. For generic information about the ATTRIBUTE clause, see Chapter 3.



This clause can specify features for input statements (CONSTRUCT, INPUT, and INPUT ARRAY) and for display statements (DISPLAY and DISPLAY ARRAY):

- The attributes of the foreground of the 4GL window
- Whether to use input attributes of the current form or 4GL window
- Whether to use display attributes of the current form or 4GL window

If this clause conflicts with another attribute specification, 4GL applies the precedence rules that are listed in "Precedence of Attributes" on page 3-98. Any attribute defined by the OPTIONS statement remains in effect until 4GL encounters an ATTRIBUTES clause that redefines the same attribute in one of the following statements:

- CONSTRUCT, INPUT, INPUT ARRAY, DISPLAY, or DISPLAY ARRAY
- Another OPTIONS statement
- An OPEN WINDOW statement

An ATTRIBUTE clause of an OPEN WINDOW, CONSTRUCT, INPUT, DISPLAY, or DISPLAY ARRAY statement only temporarily redefines the attributes. After the 4GL window closes (in the case of an OPEN WINDOW statement) or after the statement terminates (in the case of an input or display statement), 4GL restores the attributes from the most recent OPTIONS statement.

The FORM keyword in INPUT ATTRIBUTE or DISPLAY ATTRIBUTE clauses instructs 4GL to use the input or display attributes of the current form. In the following example, 4GL uses the display attributes from the form specification file:

```
OPTIONS DISPLAY ATTRIBUTE (FORM)
```

Similarly, you can use the WINDOW keyword of the same options to instruct 4GL to use the input or display attributes of the current 4GL window. You cannot combine the FORM or WINDOW attributes with any other attributes.

The HELP FILE Option

The HELP FILE clause specifies an expression that returns the filename of a help file. This filename can also include a pathname.

Messages in this file can be referenced by number in form-related statements, and are displayed at runtime when the user presses the Help key. (The **mkmessage** utility for help files is described in Appendix B.)

Assigning Logical Keys

The OPTIONS statement can specify physical keys to support 4GL logical key functions in the current task. You can specify the following keywords in uppercase or lowercase letters for *key name*.

DOWN	NEXT or NEXTPAGE	TAB
ESC or ESCAPE	PREVIOUS or PREVPAGE	UP
INTERRUPT	RETURN or ENTER	
LEFT	RIGHT	
F1 through F64		
CONTROL-*char* (except A, D, H, I, J, K, L, M, R, or X)		

For example, this statement redefines the Next Page and Previous Page keys:

```
OPTIONS NEXT KEY CONTROL-N, PREVIOUS KEY CONTROL-P
```

The keyword NEXTPAGE is a synonym for NEXT in 4GL statements (like CONSTRUCT, DISPLAY ARRAY, INPUT, MENU, OPTIONS, and PROMPT) that reference the Next Page key. Similarly, the keyword PREVPAGE is a synonym for PREVIOUS in statements that reference the Previous Page key.

The following table lists keys that require special consideration before you assign them in an OPTIONS statement.

Key	Special Considerations
ESC or ESCAPE	You must specify another key as the Accept key because ESCAPE is the default Accept key. Reassign the Accept key in the OPTIONS statement.
Interrupt	You must first execute a DEFER INTERRUPT statement. When the user presses the Interrupt key under these conditions, 4GL executes the statements in the ON KEY block and sets the global variable **int_flag** to non-zero, but does not terminate the current statement. 4GL also executes the ON KEY statement block if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, 4GL sets the **quit_flag** variable for the current task to non-zero.
CONTROL-*char*	
A, D, H, K, L, R, and X	4GL reserves these keys for field editing.
I, J, and M	The standard meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is not available to the user. Instead, the key is trapped by 4GL and used to trigger the commands in the OPTIONS statement. For example, if CONTROL-M appears in an OPTIONS statement, the user cannot press RETURN to advance the cursor to the next field. If you include one of these keys in an OPTIONS statement, also restrict the scope of the statement.

You might not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

To disable a key function, you can assign it to a control sequence that will never be executed. For example, the editing control sequences (CONTROL-A, -D, -H, -K, -L, -R, and -X) are always interpreted as field editing commands. If you assign one of these control sequences to a key function, 4GL executes the editing sequence instead of the key function. For example, the following statement disables the Delete key:

```
OPTIONS DELETE KEY CONTROL-A
```

After 4GL processes this statement, the user is no longer able to delete rows in a screen array.

Interrupting SQL Statements

The SQL INTERRUPT option specifies whether the Interrupt key interrupts SQL statements as well as 4GL statements. By default, this option is set to OFF, so pressing the Interrupt key cannot interrupt SQL statements. If the user presses the Interrupt key when an SQL statement is executing, 4GL waits for the database server to complete the SQL statement before processing the Interrupt as follows:

- If the program contains the DEFER INTERRUPT statement, 4GL sets the **int_flag** built-in variable to TRUE and continues execution.
- If the program does not contain DEFER INTERRUPT, 4GL terminates the program.

For more information on the actions of the DEFER INTERRUPT statement, see the DEFER statement on page 4-78.

To enable the Interrupt key to interrupt SQL statements, your program must contain:

- the DEFER INTERRUPT statement.
- the OPTIONS statement with the SQL INTERRUPT ON option.

When your program contains both these statements, 4GL takes the following actions when the user presses the Interrupt key:

1. Tells the database server to terminate the current SQL statement.

   The SQL statements in the following table can be terminated.

SQL Statement	Considerations
ALTER INDEX	Can be interrupted by Informix Dynamic Server only
ALTER TABLE	
CREATE INDEX	Can be interrupted by Informix Dynamic Server only
DELETE	
FETCH	Includes implicit FETCH during a FOREACH
INSERT	Includes INSERT performed during a LOAD
OPEN	If SELECT stores all the data in a temporary table
SELECT	Includes SELECT performed during an UNLOAD
UPDATE	

   If the interrupted SQL statement is within a database transaction, the database server must handle the interrupted transaction. For more information, see "Interrupting Transactions" on page 4-303.

2. Sets the built-in **int_flag** to TRUE.
3. Sets the global **SQLCA.SQLCODE** and **status** variables to error code -213.
4. Continues execution with the statement following the interrupted SQL statement, if your program has the WHENEVER ERROR CONTINUE compiler directive in effect; otherwise, the program terminates.

For SQL statements not listed in the preceding table, 4GL will allow the statement to complete before setting the built-in **int_flag** variable. It will then continue execution with the statement following the SQL statement (if your program has the WHENEVER ERROR CONTINUE compiler directive in effect).

If the DEFER QUIT statement has been executed and the user presses the Quit key (or sends a SIGQUIT signal), 4GL takes the same four actions, except that it sets the global variable **quit_flag**, rather than **int_flag**.

If you specify SQL INTERRUPT ON, but later in the program you wish to disable the SQL interruption feature, execute an OPTIONS SQL INTERRUPT OFF statement. This statement restores the default of uninterruptable SQL statements.

Interrupting Transactions

Interrupting an SQL statement has consequences for database transactions. In typical 4GL applications, the SQL INTERRUPT ON feature is of very limited value unless the database supports transaction logging. How to handle an interrupted SQL statement depends on whether the database is ANSI-compliant and on what type of transaction includes the SQL statement:

- In non-ANSI-compliant databases that support transaction logging, a transaction is either:
  - an *explicit* transaction. This starts with a BEGIN WORK statement and ends with either the COMMIT WORK (save the transaction) or ROLLBACK WORK (cancel the transaction) statement.
  - a *singleton* transaction. An SQL statement that is not within an explicit transaction (preceded by BEGIN WORK) is in a transaction of its own. The transaction ends when the SQL statement completes.
- In ANSI-compliant databases, a transaction is always in effect. Transactions in such databases are called *implicit transactions.*

In all three cases, the WORK keyword is optional in transaction management statements. Your code might be easier to read, however, if you include it.

Interrupting Implicit Transactions

ANSI

In ANSI-compliant databases, a transaction is always in effect. BEGIN WORK is not needed because any COMMIT WORK or ROLLBACK WORK statement that ends a transaction automatically marks the beginning of a new *implicit* transaction. No SQL statement can be executed outside of a transaction.

If a user interrupts an implicit transaction, no automatic ROLLBACK WORK occurs. The current transaction is still in progress. ♦

Interrupting Singleton Transactions

A *singleton* transaction occurs for every SQL statement executed outside a transaction. Singleton transactions occur only in databases that are not ANSI-compliant.

In a database that is not ANSI-compliant, and that uses transaction logging, the BEGIN WORK statement is required to begin a transaction. The database server treats any SQL statement that you execute outside of a transaction as a singleton transaction.

If an interruptable SQL statement (those listed in "Interrupting SQL Statements" on page 4-301) is within a singleton transaction and is interrupted, the database server automatically rolls back the current transaction before returning control to the 4GL program. Just as before the SQL statement was interrupted, no transaction is currently in progress.

Interrupting Explicit Transactions

An *explicit* transaction is enclosed between a BEGIN WORK and COMMIT WORK or ROLLBACK WORK statement. Explicit transactions occur only in databases that are not ANSI-compliant.

The following table summarizes what the database server does when an explicit transaction is interrupted.

Database Server	Database Server Response to Interrupt
Informix Dynamic Server	All interruptable SQL statements: automatic undo of SQL statement.
INFORMIX-SE	All interruptable SQL statements (ALTER INDEX and CREATE INDEX are not interruptable): no automatic undo for current SQL statement (interrupted statement can be in a partially completed state). The current transaction is still in progress.

Handling Interrupted Transactions

When the database server does not perform an automatic rollback, an interrupted transaction can leave the database in an unknown state. In these cases, your program should decide how to proceed.

To check for an interrupted SQL statement, a program can test the following values:

- The **int_flag** built-in variable: if your program contains the DEFER INTERRUPT statement, **int_flag** will have a value of TRUE if the user presses the Interrupt key during an interruptable SQL statement.
- The **SQLCA.SQLCODE** or **status** built-in variables, if the interruptable SQL statement is preceded by the WHENEVER ERROR CONTINUE statement. This variable will have the value of -213 if the SQL statement failed due to user interruption.

If the database is in an unknown state, your program should explicitly perform a ROLLBACK WORK statement. The ROLLBACK WORK statement reverses the current transaction while the COMMIT WORK statement commits all modifications made to the database since the beginning of the transaction. To begin a new transaction, you must use the BEGIN WORK statement.

ANSI

In ANSI-compliant databases, the ROLLBACK WORK statement reverses the current implicit transaction and automatically begins a new transaction. No BEGIN WORK statement is needed. ♦

Avoid use of the COMMIT WORK statement when the database is in an unknown state.

The following code fragment checks for an interrupted DELETE statement. This fragment assumes that the database server is not ANSI-compliant but that it supports transaction logging. Therefore the current transaction is explicit (not a singleton).

```
DEFER INTERRUPT
OPTIONS
 SQL INTERRUPT ON
...
OPEN WINDOW w_purge AT 2,2
  WITH 10 ROWS, 50 COLUMNS
  ATTRIBUTE (BORDER, PROMPT LINE 9)
DISPLAY "ACCOUNT PURGE" AT 1, 2
DISPLAY "Purging customer account of last year's info...."
  AT 3, 2
DISPLAY "Press Cancel to interrupt." AT 4, 2
```

```
LET cancelled = FALSE
LET tx_status = 0
BEGIN WORK

  UNLOAD TO filename
    SELECT *
    FROM accthistory
    WHERE customer_num = cust_num AND tx_date < start_fiscal
  IF int_flag THEN
    LET int_flag = FALSE
    IF (SQLCA.SQLCODE < 0) THEN
      IF (SQLCA.SQLCODE = -213) THEN
        LET cancelled = TRUE
      ELSE
        LET tx_status = SQLCA.SQLCODE
      END IF
    END IF
  ELSE

    DELETE FROM accthistory
    WHERE customer_num = cust_num
      AND tx_date < start_fiscal

    IF int_flag THEN
      IF (SQLCA.SQLCODE < 0) THEN
        IF (SQLCA.SQLCODE = -213) THEN
          LET cancelled = TRUE
        ELSE
          LET tx_status = SQLCA.SQLCODE
        END IF
      END IF
    END IF
  END IF

  IF (tx_status < 0) OR cancelled THEN
    ROLLBACK WORK
    IF cancelled THEN
      ERROR "Account purge terminated at user request. ",
            " No information purged."
    ELSE
      ERROR "Account purge terminated (status=",
            tx_status USING "-<<<<<<<<<<<",
            "). No information purged."
    END IF
  ELSE
    COMMIT WORK
    MESSAGE "Account purge complete."
  END IF

CLOSE WINDOW w_purge
```

For more information on data integrity, refer to the *Informix Guide to SQL: Syntax*.

Setting Default Screen Modes

4GL recognizes two screen display modes: *line mode* (IN LINE MODE) and *formatted mode* (IN FORM MODE). Besides OPTIONS, the RUN, START REPORT, and REPORT statements can explicitly specify a screen mode. The OPTIONS statement can set separate defaults for the screen mode of the RUN statement and for the screen mode of REPORT output that is sent to a pipe.

After IN LINE MODE is specified, the terminal is in the same state (in terms of **stty** options) as when the program began. This usually means that the terminal input is in *cooked mode*, with interrupts enabled, and input not becoming available until after a newline character is typed.

The IN FORM MODE keywords specify *raw mode*, in which each character of input becomes available to the program as it is typed or read.

By default, 4GL programs operate in line mode, but so many statements take it into formatted mode (including OPTIONS statements that set keys, DISPLAY AT, OPEN WINDOW, DISPLAY FORM, and other screen interaction statements), that typical 4GL programs are actually in formatted mode most of the time.

The default behavior for PIPE is IN FORM MODE (in which the screen is not cleared), for compatibility with releases earlier than INFORMIX-4GL 6.0. This mode is the opposite of the default screen mode for RUN specifications.

When the OPTIONS statement specifies RUN IN FORM MODE, the program remains in formatted mode if it currently is in formatted mode, but it does not enter formatted mode if it is currently in line mode.

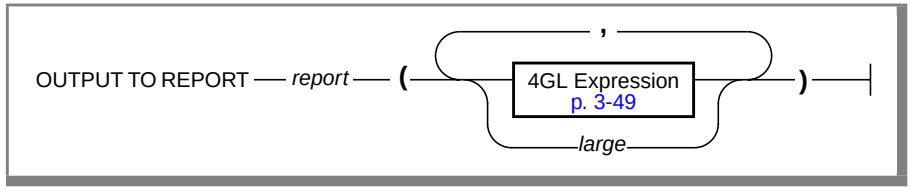
When the OPTIONS statement specifies RUN IN LINE MODE, the program remains in line mode if it is currently in line mode, and it switches to line mode if it is currently in formatted mode.

References

CONSTRUCT, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, ERROR, INPUT, INPUT ARRAY, MENU, MESSAGE, OPEN FORM, OPEN WINDOW, PROMPT

OUTPUT TO REPORT

The OUTPUT TO REPORT statement passes a single set of data values (called an input record) to a REPORT statement.



Element	Description
large	is the name of a TEXT or BYTE variable to be passed to the report.
report	is the name of a 4GL report by which to format the input record. You must also declare this identifier in a REPORT statement and invoke the report with a previous START REPORT statement.

Usage

The OUTPUT TO REPORT statement passes data to a report and instructs 4GL to process and format the data as the next input record of the report.

An *input record* is the ordered set of values returned by the expressions that you list between the parentheses. Returned values are passed to the specified report, as part of the input record. The input record can correspond to a retrieved row from the database, or to a 4GL program record, but 4GL does not require this correspondence.

The members of the input record that you specify in the expression list of the OUTPUT TO REPORT statement must correspond to elements of the formal argument list in the REPORT definition in their number and their position, and must be of compatible data types (see "Summary of Compatible 4GL Data Types" on page 3-46).

Arguments of the TEXT and BYTE data types are passed by reference rather than by value; arguments of other data types are passed by value. A report can use the WORDWRAP operator with the PRINT statement to display TEXT values. (For more information, see "The WORDWRAP Operator" on page 7-65.) A report cannot display BYTE values; the character string <byte value> in output from the report indicates a BYTE value.

You typically include the OUTPUT TO REPORT statement within a WHILE, FOR, or FOREACH loop, so that the program passes data to the report one input record at a time. The portion of the 4GL program that includes START REPORT, OUTPUT TO REPORT, and FINISH REPORT statements that reference the same report is sometimes called the report driver. (For more information about 4GL reports, see Chapter 7.)The following program fragment uses a FOREACH loop to pass input records to a report:

```
START REPORT cust_list
   ...
FOREACH q_curs INTO p_customer.lname, p_customer.company
   OUTPUT TO REPORT cust_list
      (p_customer.lname, p_customer.company,
       "San Francisco", TODAY)
END FOREACH
```

Each input record consists of four values:

- The **lname** and **company** values from columns of a database table
- The literal string constant "San Francisco"
- The DATE value returned by the TODAY operator

The following program creates a report, with default formatting, of all the customers in the **customer** table, and sends the resulting output to a file:

```
DATABASE stores7
MAIN
   DEFINE p_customer RECORD LIKE customer.*
   DECLARE q_curs CURSOR FOR SELECT * FROM customer
   START REPORT cust_list TO "cust_listing"
   FOREACH q_curs INTO p_customer.*
      OUTPUT TO REPORT cust_list(p_customer.*)
   FINISH REPORT cust_list
END MAIN
REPORT cust_list(r_customer)
   DEFINE r_customer RECORD LIKE customer.*
   FORMAT EVERY ROW
END REPORT
```

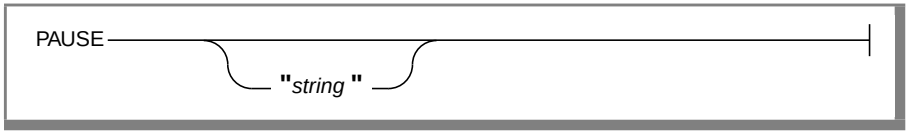
If OUTPUT TO REPORT is not executed, no control blocks of the report definition are executed, even if your report driver also includes the START REPORT and FINISH REPORT statements.

References

CALL, FINISH REPORT, PAUSE, REPORT, START REPORT, TERMINATE REPORT

PAUSE

The PAUSE statement suspends the display of output from a 4GL report to the 4GL screen. The PAUSE statement can only appear in the FORMAT section of a REPORT program block and only affects report output sent to the screen.



Element	Description
string	is a quoted string.

Usage

The PAUSE statement affects the behavior of the report output in the 4GL screen as follows. It has no effect on the formatted report output.

- If a PAUSE statement appears in the REPORT definition, the report displays a screenful of output and then pauses. The user needs to press RETURN to view the next screenful of output. If a quoted string is specified, its text appears on the 4GL screen.
- In the absence of a PAUSE statement, the report output scrolls down the 4GL screen.

The PAUSE statement has no effect if you include a REPORT TO clause in the OUTPUT section, or a TO clause in the START REPORT statement.

References

NEED, PRINT, REPORT, SCROLL, SKIP, START REPORT

PREPARE

Use the PREPARE statement to parse, validate, and generate an execution plan for SQL statements in a 4GL program at runtime.

Element	Description
statement identifier	is an SQL statement identifier. This must be unique (within its scope) among the names of prepared statements and cursors.
string	is a quoted string containing part or all of the text of one or more SQL statements to be prepared.
variable	is a variable containing text of one or more SQL statements to be prepared.

Usage

This statement assembles the text of an SQL statement at runtime and makes it executable. This *dynamic* form of SQL is accomplished in three steps:

1. PREPARE accepts SQL statement text as input, either as a quoted string or stored within a character variable; this text can contain question mark (?) placeholders to represent data values that the user must specify at runtime when the statement is executed.
2. The EXECUTE or OPEN statement can supply input values in the USING clause and can execute the prepared statement once or many times.
3. Resources allocated to the prepared statement can be released later by the FREE statement.

For more about FREE, EXECUTE, and OPEN, see the documentation of your Informix database server. See also "SQL" on page 4-349, which describes an alternative to PREPARE for using SQL statements in 4GL programs.

Important: *You cannot reference a 4GL variable in the text of a prepared statement. Use an SQL…END SQL block, rather than PREPARE, for SQL statements that cannot be embedded, but that require host variables as input or output parameters.*

The number of prepared objects in a single program is limited only by available memory. Prepared objects include both statement identifiers named in PREPARE statements and cursor declarations that incorporate SELECT, EXECUTE PROCEDURE, or INSERT statements. (To deallocate these objects, you can use a FREE statement to release some statements or cursors.)

The following topics are described in this section:

Statement Identifier

The PREPARE statement sends statement text to the database server where it is analyzed. If it contains no syntax errors, the text is converted to an internal form. This translated statement is saved for later execution in a data structure that the PREPARE statement allocates. The structure has the name specified by the *statement identifier* in the PREPARE statement. Subsequent SQL statements can refer to the prepared statement by using the statement identifier.

By default, the scope of reference of a statement identifier is the 4GL module in which it was declared. The identifier of a statement that was prepared in one 4GL module cannot be referenced from another module. To reference a statement identifier outside the module in which it was declared, compile the module in which it is declared with the **-globcurs** option.

For some database servers, unless you use the **-global** command-line option to compile your program, the name cannot be longer than nine characters. Read the documentation for your database server to see if this restriction on the length of statement identifiers affects your application.

A subsequent FREE *statement identifier* statement releases the resources allocated to the prepared statement. After FREE releases it, the statement identifier cannot be referenced by a cursor, or by the EXECUTE statement, until you prepare the statement again.

A statement identifier can represent only one SQL statement (or one sequence of statements) at a time. You can execute a new PREPARE statement with an existing statement identifier if you wish to assign the text of a different SQL statement to the statement identifier.

Statement Text

The statement text can be a quoted string or text stored in a variable. The following restrictions apply to the statement text:

- Text can contain only SQL or SPL statements. Not valid are 4GL statements that are not SQL statements, C or C++ statements, or SQL statements that cannot be prepared (as listed in "Statements That Cannot Be Prepared" on page 4-317).
- The text can contain either a single SQL statement or a sequence of statements. If the text contains more than a single statement, successive statements must be separated by semicolons.
- Comments preceded by two hyphens (- -), or enclosed in braces ({ }) are allowed in the statement text. The comment ends at the end of the line (after - -) or at the right-brace (}); see also "Comments" on page 2-8. The pound sign (#) symbol is not valid here as a comment indicator. Comment text is restricted to the code set of the locale.
- The only valid identifiers are SQL names of database entities, such as tables and columns. You cannot prepare a SELECT statement that contains an INTO *variable* clause, which requires a 4GL variable.
- The question mark (?) placeholder can indicate where a data value (but not an SQL identifier) needs to be supplied at runtime.

The following example includes placeholders for values that are to be input:

```
PREPARE nt FROM "INSERT INTO cust(fname,lname) VALUES(?,?)"
```

Preparing a SELECT Statement

You can prepare a SELECT statement. If the SELECT statement includes the INTO TEMP clause, you can only execute the prepared statement with an EXECUTE statement. If SELECT does not include the INTO TEMP clause, you must use DECLARE *cursor* and either the FOREACH statement or the OPEN *cursor* and FETCH *cursor* statements to retrieve the specified rows. You cannot use FOREACH with a prepared SELECT statement that includes a question mark placeholder.

A prepared SELECT statement can include a FOR UPDATE clause. This clause normally is used with the DECLARE statement to create an update cursor. This example shows a SELECT statement with a FOR UPDATE clause:

```
PREPARE up_sel FROM
   "SELECT * FROM customer ",
   "WHERE customer_num between ? and ? ",
   "FOR UPDATE"

DECLARE up_curs CURSOR FOR up_sel

OPEN up_curs USING low_cust, high_cust
```

Statements That Can or Must Be Prepared

The 4GL compiler supports directly embedded 4GL statements that include only the syntax of Informix 4.1 database servers. See Appendix I, "SQL Statements That Can Be Embedded in 4GL Code," for a list of the SQL 4.1 statements that can be directly embedded. Most (but not all) SQL statements can be prepared; some *must* either be prepared or else enclosed within the SQL...END SQL delimiters if they are to be used in a 4GL program.

You must prepare most SQL statements that include syntax introduced after Version 4.1 of Informix database servers. Thus, the ON DELETE CASCADE clause in the CREATE TABLE statement requires PREPARE. But you do not need to prepare the following statements, if they include only Informix 4.1 syntax.

ALTER TABLE	CREATE SYNONYM	INSERT INTO
CREATE INDEX	DROP TABLE	REVOKE
CREATE TABLE	DROP VIEW	UPDATE STATISTICS
CREATE SCHEMA	GRANT	

Statements That Cannot Be Embedded

Some SQL statements cannot be directly embedded; they must be prepared (or else enclosed in SQL...END SQL delimiters) to be used in a 4GL program. None of the statements in the following table can be directly embedded. For more information, see "Statements That Cannot Be Prepared" on page 4-317.

ALTER FRAGMENT	SET CONSTRAINT
ALTER OPTICAL CLUSTER	SET DATABASE OBJECT MODE
CREATE EXTERNAL TABLE	SET DATASKIP
CREATE OPTICAL CLUSTER	SET DEBUG FILE TO
CREATE ROLE	SET LOG
CREATE SCHEMA	SET MOUNTING TIMEOUT
CREATE TRIGGER	SET OPTIMIZATION
DROP OPTICAL CLUSTER	SET PDQPRIORITY
DROP PROCEDURE	SET PLOAD FILE
DROP ROLE	SET RESIDENCY
DROP TRIGGER	SET ROLE
EXECUTE PROCEDURE	SET SCHEDULE LEVEL
GRANT FRAGMENT	SET SESSION AUTHORIZATION
RELEASE	SET TRANSACTION
RENAME DATABASE	SET TRANSACTION MODE
RESERVE	START VIOLATIONS TABLE
REVOKE FRAGMENT	STOP VIOLATIONS TABLE

Statements That Might Need to Be Prepared

Some SQL statements require you to prepare them only if you are using a 5.0 or later syntax in the statement. For example, if you use the PUBLIC or PRIVATE clause of the CREATE SYNONYM statement, you need to prepare the CREATE SYNONYM statement. If you do not include the PUBLIC or PRIVATE clause, you do not need to prepare the statement. See Appendix I, "SQL Statements That Can Be Embedded in 4GL Code," for the syntax of SQL statements that do not require PREPARE or SQL...END SQL delimiters in 4GL programs.

Statements That Cannot Be Prepared

This release of 4GL can embed CREATE PROCEDURE FROM statements directly, unless the procedure includes SPL statements that perform I/O.

In addition, the following SQL statements, which are supported in some releases of Informix database servers, cannot appear as text in PREPARE statements. (In this release of 4GL, the SQL statements that are marked by an × symbol are not available, prepared nor otherwise.)

×	ALLOCATE COLLECTION		FREE
×	ALLOCATE DESCRIPTOR	×	GET DESCRIPTOR
×	ALLOCATE ROW	×	GET DIAGNOSTICS
×	CHECK TABLE	×	INFO
	CLOSE		LOAD
	CONNECT		OPEN
	CREATE PROCEDURE FROM	×	OUTPUT
×	DEALLOCATE COLLECTION		PREPARE
×	DEALLOCATE DESCRIPTOR		PUT
×	DEALLOCATE ROW	×	REPAIR TABLE
	DECLARE	×	SET AUTOFREE
×	DESCRIBE		SET CONNECTION
	DISCONNECT	×	SET DEFERRED_PREPARE
	EXECUTE	×	SET DESCRIPTOR
	EXECUTE IMMEDIATE		UNLOAD
	FETCH		WHENEVER
	FLUSH		

Additionally, you cannot use the following statements in prepared statement text that contains multiple SQL statements separated by semicolons.

CLOSE DATABASE	DATABASE	SELECT
CREATE DATABASE	DROP DATABASE	START DATABASE

Thus, the SELECT statement (except for SELECT INTO TEMP) is not valid within the text of a multistatement PREPARE. In addition, statements that could cause the current database to be closed during execution of the sequence of prepared statements (such as CONNECT, DISCONNECT, and SET CONNECTION) are also not valid in this context.

For general information about multistatement prepares, see "Preparing Sequences of Multiple SQL Statements" on page 4-321.

Executing Stored Procedures Using PREPARE

The following steps describe how to define and execute a stored procedure in a 4GL program:

1. Put the text of the CREATE PROCEDURE statement in a separate file.

   Use SPL statements to define the procedure.
2. Directly embed a CREATE PROCEDURE FROM *filename* statement that references the text file created in step 1.
3. Use a PREPARE statement to prepare the text of an EXECUTE PROCEDURE statement to execute the same stored procedure.
4. Use an EXECUTE statement to execute the EXECUTE PROCEDURE statement that you prepared in step 3.

Warning: *The Stored Procedure Language (SPL) is not a part of the 4GL language. Attempting to include SPL statements directly within a 4GL program, rather than through a CREATE PROCEDURE FROM filename statement, causes compile errors.*

You can also invoke a stored procedure implicitly through a reference to that procedure within the context of an SQL expression. For example, the reference to **avg_price( )** in the following SELECT statement implicitly invokes the stored procedure called **avg_price**:

```
SELECT
   manu_code, unit_price, (avg_price(1) - unit_price)
   FROM stock
   WHERE stock_num = 1
```

Such implicit references to stored procedures do not require the statement to be prepared, because the database server processes them in a manner that is transparent to the 4GL program.

The *Informix Guide to SQL: Tutorial* describes how to create and execute stored procedures. See the *Informix Guide to SQL: Syntax* for complete descriptions of the CREATE PROCEDURE and CREATE PROCEDURE FROM statements.

Using Parameters in Prepared Statements

You can pass values to a prepared statement either when you prepare the statement or at execution time.

Preparing Statements When Parameters Are Known

In some prepared statements, all needed information is known at the time the statement is prepared. Although all parts of the statement are known prior to the prepare, they also can be derived dynamically from program input. In the following example, user input is incorporated into a SELECT statement, which is then prepared and associated with a cursor:

```
DEFINE u_po LIKE orders.po_num
PROMPT "Enter p.o. number please: " FOR u_po
PREPARE sel_po FROM
   "SELECT * FROM orders ",
   "WHERE po_num = '", u_po, "'"
DECLARE get_po CURSOR FOR sel_po
```

Preparing Statements That Receive Parameters at Execution

In some statements, parameters are unknown when the statement is prepared because a different value can be inserted each time the statement is executed. In these statements, you can use a question mark placeholder where a parameter must be supplied when the statement is executed.

The PREPARE statements in the following example show some uses of question mark placeholders:

```
PREPARE s3 FROM
   "SELECT * FROM customer WHERE state MATCHES ?"

PREPARE in1 FROM
   "INSERT INTO manufact VALUES (?,?,?)"

PREPARE update2 FROM
   "UPDATE customer SET zipcode = ?",
   "WHERE CURRENT OF zip_cursor"
```

You can use a placeholder only to supply a value for an expression. You cannot use a question mark placeholder to represent an SQL identifier such as a database name, a table name, or a column name.

The USING clause is available in both OPEN statements (for statements associated with a cursor) and EXECUTE statements (for all other prepared statements). For example:

```
DEFINE zip LIKE customer.zipcode
PREPARE zip_sel FROM
   "SELECT * FROM customer WHERE zipcode MATCHES ?"
DECLARE zip_curs CURSOR FOR zip_sel
PROMPT "Enter a zipcode: " FOR zip
OPEN zip_curs USING zip
```

If the prepared SELECT statement contains a question mark placeholder, you cannot execute the statement with a FOREACH statement; you must use the OPEN, FETCH, and CLOSE group of statements.

Preparing Statements with SQL Identifiers

You cannot use question mark placeholders for SQL identifiers such as a table name or a column name; you must specify these identifiers in the statement text when you prepare it.

If these identifiers are not available when you write the statement, however, you can construct a statement that receives SQL identifiers from user input. In the following example, the name of the column is supplied by the user and inserted in the statement text before the PREPARE statement. The search value in that column also is taken from user input, but it is supplied to the statement with a USING clause:

```
DEFINE   column_name CHAR(30),
         column_value CHAR(40),
         del_str CHAR(100)

PROMPT "Enter column name: " FOR column_name

LET del_str =
   "DELETE FROM customer WHERE ",
   column_name CLIPPED, " = ?"
PREPARE de4 FROM del_str

PROMPT "Enter search value in column ",column_name, ":"
   FOR column_value

EXECUTE de4 USING column_value
```

Preparing Sequences of Multiple SQL Statements

You can execute several SQL statements as one action if you include them in the same PREPARE statement. Multistatement text is processed as a unit; actions are not treated sequentially. Therefore, multistatement text cannot include statements that depend on action that occurs in a previous statement in the text. For example, you cannot create a table and insert values into that table in the same prepared block. Avoid placing BEGIN WORK and COMMIT WORK statements with other statements in a multistatement prepare.

In most situations, 4GL returns error status information on the first error in the multistatement text. No indication exists of which statement in the sequence causes an error. You can use **SQLCA** to find the offset of the **SQLERRD[5]** errors. For more information, see "Exception Handling" on page 2-40.

The following example updates the **stores7** database by replacing existing manufacturer codes with new codes. Because the **manu_code** columns are potential join columns that link four of the tables, the new codes must replace the old codes in three tables:

```
DATABASE stores7
MAIN
   DEFINE code_chnge RECORD
      new_code LIKE manufact.manu_code,
      old_code LIKE manufact.manu_code
   END RECORD
   sqlmulti CHAR(250)

PROMPT "Enter new manufacturer code: "
   FOR code_chnge.new_code
PROMPT "Enter old manufacturer code: "
   FOR code_chnge.old_code
LET sqlmulti =
   "UPDATE manufact SET manu_code = ? WHERE manu_code = ?;",
   "UPDATE stock SET menu_code = ? WHERE manu_code = ?;",
   "UPDATE items SET manu_code = ? WHERE manu_code = ?;",
   "UPDATE catalog SET manu_code = ? WHERE manu_code = ?;"

PREPARE exmulti FROM sqlmulti
EXECUTE exmulti USING code_chnge.*, code_chnge.*, code_chnge.*
      code_chnge.*
END MAIN
```

Runtime Errors in Multistatement Texts

If an error is returned while any SQL statement within multistatement prepared text is being processed, no subsequent prepared SQL statements within the same text are executed. (If another statement in that text is also capable of producing an error, this additional error cannot be issued until after you correct the prior errors within the multistatement text.)

Thus, 4GL returns error status information on only the first error that it finds in the multistatement text, with no indication of which statement in the sequence caused the error. You can use **SQLCA** to find the offset of the **SQLERRD[5]** errors. For more information about **SQLCA** and error-status information, see “Error Handling with SQLCA” on page 2-45.

Any error or warning message that the database server returns, regardless of the severity, terminates execution of the multistatement prepared text. For example, the NOTFOUND end-of-data condition terminates execution of the prepared text.

Using Prepared Statements for Efficiency

To increase performance efficiency, you can use the PREPARE statement and an EXECUTE statement in a loop to eliminate overhead caused by redundant parsing and optimizing. For example, an UPDATE statement located within a WHILE loop is parsed each time the loop runs. If you prepare the UPDATE statement outside the loop, the statement is parsed only once, eliminating overhead and speeding statement execution. The following example shows how to prepare statements to improve performance:

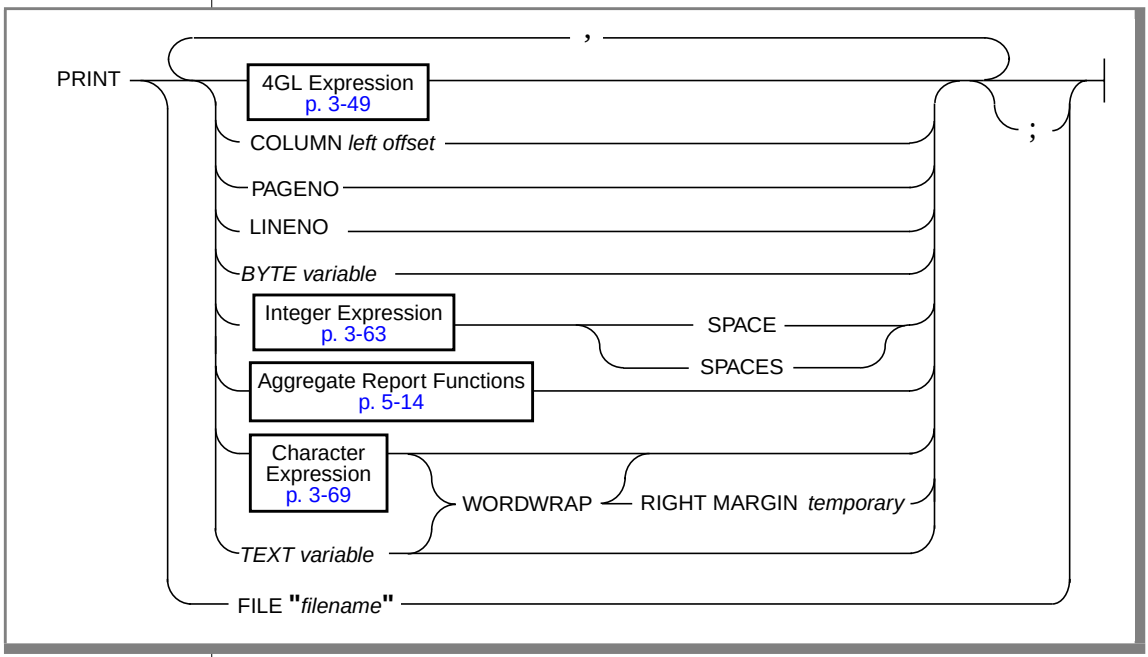
```
PREPARE up1 FROM "UPDATE customer ",
   "SET discount = 0.1 WHERE customer_num = ?"
WHILE TRUE
   PROMPT "Enter Customer Number" FOR dis_cust
   IF dis_cust = 0 THEN
      EXIT WHILE
   END IF
   EXECUTE up1 USING dis_cust
END WHILE
```

References

See the DECLARE, DESCRIBE, EXECUTE, FREE, and OPEN statements in the *Informix Guide to SQL: Syntax*. See also "SQL" on page 4-349.

PRINT

The PRINT statement produces output from a report definition.

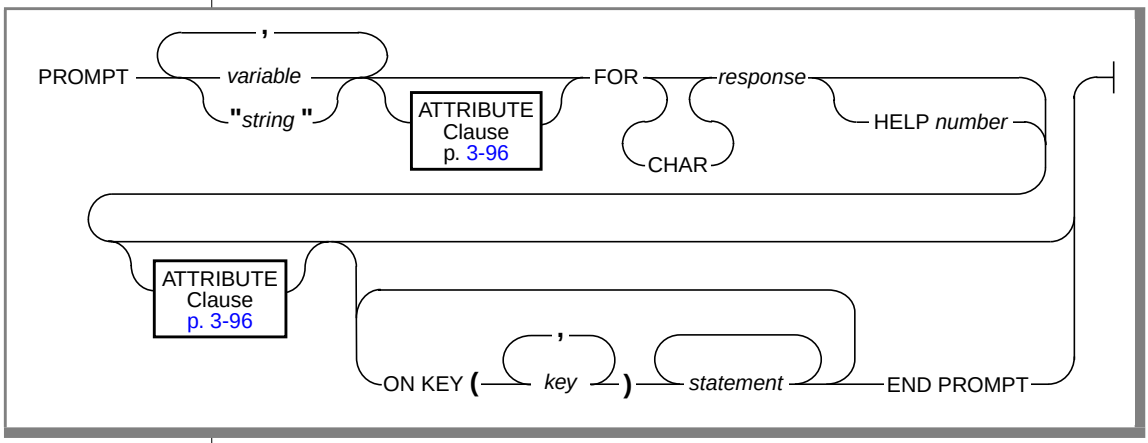


Element	Description
BYTE variable	is the identifier of a 4GL variable of data type BYTE.
filename	is a quoted string that specifies the name of a text file to include in the output from the report. This can include a pathname.
left offset	is an expression that return a positive whole number. It specifies a character position offset (from the left margin) no greater than the difference (*right margin - left margin).*
temporary	is an expression that evaluates to a positive whole number. It specifies the absolute position of a temporary right margin.
TEXT variable	is the identifier of an 4GL variable of the TEXT data type.

For details of the syntax and usage of the PRINT statement in 4GL report definitions, see "PRINT" on page 7-55.

PROMPT

The PROMPT statement can assign a user-supplied value to a variable.



Element	Description
key	is a keyword to specify an activation key. (For more information, see "The ON KEY Blocks" on page 4-329.)
number	is a literal integer (as described in "Literal Integers" on page 3-65) to specify a help message number.
response	is the name of a variable to store the response of the user to the PROMPT character string. This cannot be of data type TEXT or BYTE.
statement	is an SQL statement or other 4GL statement.
string	is a quoted string that 4GL displays on the Prompt line.
variable	is a CHAR or VARCHAR variable that contains all or part of a message to the user, typically prompting the user to enter a value.

Usage

The PROMPT statement displays the specified character string on the Prompt line, and then waits for input from the user. What the user types is saved in the *response* variable, unless what the user typed was one of the keys that an ON KEY clause specified as its activation key.

4GL takes the following actions when it executes a PROMPT statement:

1. Replaces any variables with their current values
2. Concatenates the list of values into a single *prompt string*

   (The total length of this string, plus the length of the response that the user enters, cannot exceed 80 bytes.)
3. Displays the resulting string on the Prompt line of the current form (or in the line mode overlay, if it currently covers the 4GL screen)
4. Waits for the user to enter a value
5. Reads whatever value was entered until the user presses RETURN, and then stores this value in *response variable*.

The prompt string remains visible until the user enters a response.

The following topics are described in this section:

The PROMPT String

Depending on whether the line mode overlay is visible when the PROMPT statement is executed, PROMPT can produce two types of displays:

- If PROMPT is the next interactive statement after a line mode DISPLAY statement, the prompt string appears in the bottom line of the line mode overlay. The prompt string does not scroll with any subsequent output from line mode DISPLAY statements. (See "Sending Output to the Line Mode Overlay" on page 4-91.)
- If the 4GL screen or any other 4GL window is visible, the prompt string appears on the Prompt line of the current 4GL window. (If this is not as wide as the prompt string, runtime error -1146 occurs.)

The Response Variable

The PROMPT statement returns the value entered by the user in a *response* variable, which can be of any data type except TEXT and BYTE. If it has a character data type, its returned value can include blank spaces. If 4GL cannot convert the value entered by the user to the data type of the *response* variable, a negative error code is assigned to the global **status** variable.

The FOR Clause

The FOR clause specifies the name of the response variable to store input from the user. When the user types a response and presses RETURN, 4GL saves the response in the response variable. You can optionally include the CHAR keyword to accept a single-character input without requiring that the user press RETURN. For example, the following program fragment checks the *response* input for an uppercase or lowercase *y*:

```
PROMPT "Do you want to continue: " FOR CHAR ans
IF ans MATCHES "[Yy]" THEN
   CALL next_form()
END IF
```

The ATTRIBUTE Clauses

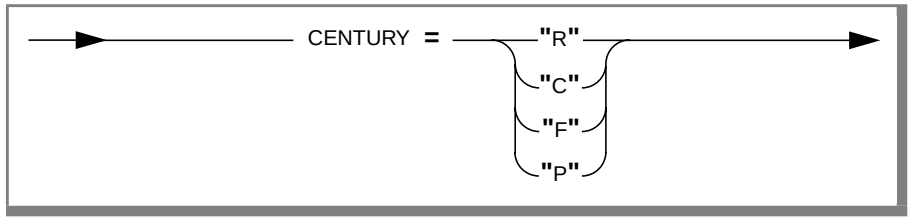
The ATTRIBUTE clauses of the PROMPT statement temporarily override any attributes specified in OPTIONS or OPEN WINDOW statements for the prompt string and for the response from the user. You can set the following attributes independently:

- The first ATTRIBUTE clause specifies display attributes of the prompt string text. The default display attribute for this text is NORMAL.
- The second ATTRIBUTE clause specifies display attributes of the response. The default is REVERSE.

The CENTURY Attribute

If the *response* variable has the DATE or DATETIME data type, this clause can include the CENTURY attribute, with the same syntax and semantics as the CENTURY field attribute (as described in "CENTURY" on page 6-35).

The CENTURY attribute specifies how to expand abbreviated one- and two-digit *year* specifications in a DATE and DATETIME field. Expansion is based on this setting (and on the year value from the system clock at runtime).



CENTURY can specify any of four algorithms to expand abbreviated years into four-digit year values that end with the same digits (or digit) that the user entered. CENTURY supports the same settings as the **DBCENTURY** environment variable, but with a scope that is restricted to a single field.

Symbol	Algorithm for Expanding Abbreviated Years
C or c	Use the past, future, or current year closest to the current date.
F or f	Use the nearest year in the future to expand the entered value.
P or p	Use the nearest year in the past to expand the entered value.
R or r	Prefix the entered value with the first two digits of the current year.

Here *past*, *closest*, *current*, and *future* are all relative to the system clock.

Unlike **DBCENTURY**, which sets a global rule for expanding abbreviated year values in DATE and DATETIME fields that do not have the CENTURY attribute, CENTURY is not case-sensitive; you can substitute lowercase letters (r, c, f, p) for these uppercase letters. If you specify anything else, an error (-2018) is issued. If the CENTURY and **DBCENTURY** settings are different, CENTURY takes precedence.

For example, the following statement prompts for a delivery date:

```
PROMPT "Enter the preferred delivery day for ",
      customer_num, " "ATTRIBUTE (YELLOW)
   FOR del_day ATTRIBUTE (BLUE, CENTURY = "F")
   ...
END PROMPT
```

Here the prompt string appears in yellow on color monitors or in bold on monochrome monitors. The delivery date that the user enters appears in blue on color monitors and in dim on monochrome monitors. If the *year* is entered as two digits, 4GL assigns to the variable the nearest future date that matches the unabbreviated portion of the response. (For more information about 4GL display attributes, see "ATTRIBUTE Clause" on page 3-96.)

The HELP Clause

This clause specifies a literal integer (as described in "Literal Integers" on page 3-65) that returns the number of a help message for the PROMPT statement. 4GL displays the help message in the Help window if the user presses the Help key from the response field. By default, the Help key is CONTROL-W. You can redefine the Help key by using the OPTIONS statement.

You create help messages in an ASCII file whose filename you specify in the HELP FILE clause of the OPTIONS statement. Use the **mkmessage** utility (as described in Appendix B) to create a runtime version of the help file. Runtime errors occur in these situations:

- 4GL cannot open the help file.
- You specify a number that is not in the help file.
- You specify a number outside the range from -32,767 to 32,767.

The ON KEY Blocks

An ON KEY block executes a series of statements when the user presses one of the specified keys. If the user presses a specified key, control passes to the statements specified in the ON KEY block. After completing the ON KEY block, 4GL passes control to the statements following the END PROMPT statement. In this case, the value of the response variable is undetermined.

You can specify the following in uppercase or lowercase letters for *key name*.

ACCEPT	HELP	NEXT or NEXTPAGE
DELETE	INSERT	PREVIOUS or PREVPAGE
DOWN	INTERRUPT	RIGHT
ESC or ESCAPE	LEFT	TAB
F1 through F64		UP
CONTROL-*char* (except A, D, H, I, J, K, L, M, R, or X)		

Here you can substitute NEXTPAGE for NEXT, and PREVPAGE for PREVIOUS.

The following table lists keys that require special consideration before you assign them in an ON KEY clause.

Key	Special Considerations
ESC or ESCAPE	You must use the OPTIONS statement to specify another key as the Accept key because ESCAPE is the default Accept key.
Interrupt	You must execute a DEFER INTERRUPT statement. When the user presses the Interrupt key under these conditions, 4GL executes the ON KEY block statements and sets **int_flag** to non-zero, but does not terminate the PROMPT statement.
Quit	4GL also executes the statements in this ON KEY clause if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, 4GL sets **quit_flag** to non-zero.
CONTROL-*char*	
A, D, H, K, L, R, and X	4GL reserves these keys for field editing.
I, J, and M	The usual meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is not available to the user, because 4GL traps the key and uses it to activate the commands in the ON KEY clause. For example, if `CONTROL-M` appears in an ON KEY clause, the user cannot press RETURN to advance the cursor to the next field. If you must include one of these keys in an ON KEY clause, be careful to restrict the scope of the clause to specific fields.

You might not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

The next example specifies two ON KEY clauses:

```
PROMPT "Enter the preferred delivery day for ", customer_num, " "
   ATTRIBUTE (YELLOW) FOR del_day
   ON KEY (CONTROL_B) LET del_day = set_day()
   ON KEY (F6, CONTROL_F) CALL delivery_help()
END PROMPT
```

In this example, if the user presses CONTROL-B, 4GL calls the **set_day( )** function and sets the **del_day** variable to the value returned by **set_day**. If the user presses F6 or CONTROL-F, the **delivery_help( )** function is invoked.

The END PROMPT Keywords

The END PROMPT keywords indicate the end of the PROMPT statement. These keywords are required only if you specify an ON KEY block. Place the END PROMPT keywords after the last statement of the last ON KEY block.

You can optionally include the END PROMPT keywords as a statement terminator for a PROMPT statement that has no ON KEY block.

The Position of the Prompt Line

Either of the following statements can change the default position of the Prompt line (the *first* line of the current window):

- A PROMPT LINE specification in the OPEN WINDOW statement
- A PROMPT LINE specification in the OPTIONS statement

If PROMPT LINE assigns a position that is outside the range of lines in the current window, the position defaults to the first line. For example, the next program increments the position of the PROMPT LINE in a WHILE loop:

```
MAIN
   DEFINE ans CHAR(1)
   DEFINE pline INTEGER
   DEFINE flag CHAR(1)
   LET pline = 7
   OPTIONS PROMPT LINE pline
   WHILE pline <> 10
      OPEN WINDOW wdw AT 4,6 WITH 7 ROWS, 60 COLUMNS ATTRIBUTE (BORDER)
      DISPLAY "  winrowsize =  7, PROMPT LINE is set to ", pline at 2, 6
      PROMPT "12345678901234567890123456789abcdef" FOR CHAR ans
      CLOSE WINDOW wdw
      LET pline = pline + 1
      OPTIONS PROMPT LINE pline
   END WHILE
END MAIN
```

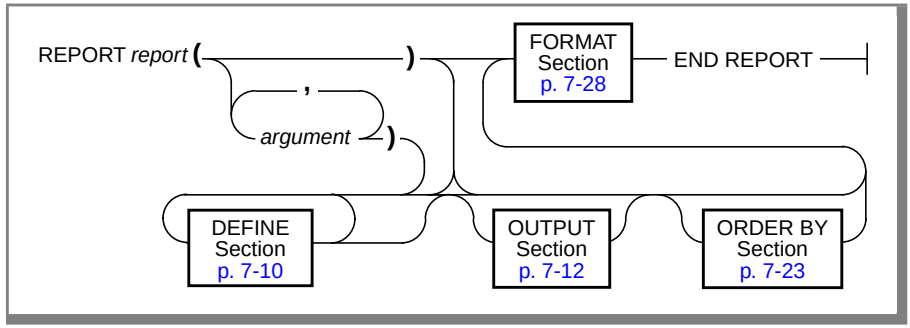
When the incremented value of the PROMPT LINE becomes 8, 4GL detects that this value is larger than the vertical size of the window and silently resets the value of PROMPT LINE to the default value (that is, to the first line of the window).

References

DISPLAY, DISPLAY ARRAY, INPUT, INPUT ARRAY, OPEN WINDOW, OPTIONS

REPORT

The REPORT statement declares the identifier and defines the format of a 4GL report. (For details of its syntax and usage, see Chapter 7.)



Element	Description
argument	is the name of a formal argument in each input record. The list can include arguments of the RECORD data type, but the *record*.* notation and ARRAY data type are not valid here.
report	is the 4GL identifier that you declare here for the report.

Usage

This statement defines a REPORT program block, just as the FUNCTION statement defines a function. You can execute a report from the MAIN program block or from a function, but the REPORT statement *cannot* appear within the MAIN statement, in a FUNCTION definition, or in another REPORT statement. Creating a 4GL report is a two-step process:

1. Use the REPORT statement to define how to format data in the report.
2. Write a *report driver* that passes data to the report.

The report driver typically uses a loop (such as WHILE, FOR, or FOREACH) with the following 4GL statements to process the report:

- START REPORT (invokes the REPORT routine)
- OUTPUT TO REPORT (sends data to the REPORT for formatting)

- FINISH REPORT (to complete execution of the REPORT routine)
- TERMINATE REPORT (to stop processing and exit from the REPORT routine, typically after an exceptional condition has been detected)

Unlike a FUNCTION program block, a REPORT routine is not reentrant. If a START REPORT statement references a report that is already running, the report is reinitialized, and any output might be unpredictable. If OUTPUT TO REPORT is not executed, no control blocks of the report are executed, even if your program includes the START REPORT and FINISH REPORT statements.

The Report Prototype

The report name must immediately follow the REPORT keyword. When assigning a name to a report, follow the guidelines described in "4GL Identifiers" on page 2-14. The name must be unique among function and report names within the 4GL program. Its scope is the entire 4GL program.

The list of formal arguments of the report must be enclosed in parentheses and separated by commas. These are local variables that store values that the calling routine passes to the report. The compiler issues an error unless you declare their data types in the subsequent DEFINE section (described in Chapter 7). You can include a program record in the formal argument list, but you cannot append the **.*** symbols to the name of the record. Arguments can be of any data type except ARRAY, or a record with an ARRAY member.

When you call a report, the formal arguments are assigned values from the argument list of the OUTPUT TO REPORT statement. These *actual arguments* that you pass must match, in number and position, the *formal arguments* of the REPORT statement. The data types must be compatible (as described in "Data Type Conversion" on page 3-42), but they need not be identical. 4GL can perform some conversions between compatible data types. The names of the actual arguments and the formal arguments do not have to match.

The Report Program Block

The REPORT definition must include a FORMAT section, and can also include DEFINE, OUTPUT, and ORDER BY sections, as described in Chapter 7. You must declare the data types of the formal arguments and of any local variables in the DEFINE section of the report, which must immediately follow the formal argument list. Within the REPORT program block, these variables take precedence over any global or module variables of the same name. Variables local to the 4GL report cannot be referenced outside of the report, and they do not retain values between invocations of the report.

You must include the following items in the list of formal arguments:

- All the values for each row sent to the report in the following cases:
  - If you include an ORDER BY section or GROUP PERCENT(*) function
  - If you use a global aggregate function (one over all rows of the report) anywhere in the report, except in the ON LAST ROW control block
  - If you specify the FORMAT EVERY ROW default format
- Any variables referenced in the following group control blocks:
  - AFTER GROUP OF
  - BEFORE GROUP OF

Two-Pass Reports

A *two-pass report* is one that creates a temporary table. The REPORT statement creates a temporary table if it includes any of the following items:

- An ORDER BY section without the EXTERNAL keyword
- The GROUP PERCENT(*) aggregate function anywhere in the report
- Any aggregate function that has no GROUP keyword in any control block other than ON LAST ROW.

The FINISH REPORT statement uses values from these tables to calculate any global aggregates, and then deletes the tables.

A two-pass report requires that the 4GL program be connected to a database when the report runs. See the DATABASE statement for information on how to specify a current database at runtime (as described in "The Current Database at Runtime" on page 4-74).

If the DEFINE section uses the LIKE keyword to declare local variables of the report indirectly, you must also include a DATABASE statement in the same module as the REPORT statement, but before the first program block, to specify a default database at compile time. (See "The Default Database at Compile Time" on page 4-73.)

An error occurs if you close the current database, or if you connect to another database, while a two-pass report is running. Even if none of the input records that the report formats are retrieved from a database, a two-pass report requires a current database to store the temporary tables.

The Exit Report Statement

This statement can appear in the FORMAT section of the report definition. It has the same effect as TERMINATE REPORT, except that EXIT REPORT must appear within the REPORT program block, while TERMINATE REPORT must appear in the report driver.

EXIT REPORT is useful after the program (or the user) becomes aware that a problem prevents the report from producing part of its intended output. EXIT REPORT has the following effects:

- Terminates the processing of the current report
- Deletes any intermediate files or temporary tables that were created in processing the REPORT statement

You cannot use the RETURN statement as a substitute for EXIT REPORT. An error is issued if RETURN is encountered within the definition of a 4GL report.

The END REPORT Keywords

The END REPORT keywords mark the end of the REPORT program block. The following program fragment briefly illustrates some of the components of the REPORT statement. This example creates a report named **simple** that displays on the screen in default format all the rows from the **customer** table:

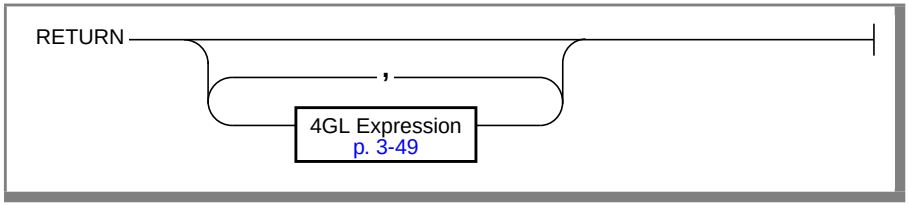
```
DECLARE simp_curs CURSOR FOR SELECT * FROM customer
START REPORT simple
FOREACH simp_curs INTO cust.*
   OUTPUT TO REPORT simple(cust.*)
END FOREACH
FINISH REPORT simple
   ...
REPORT simple (x)
   DEFINE x RECORD LIKE customer.*
   FORMAT EVERY ROW
END REPORT
```

References

DATABASE, DEFINE, FINISH REPORT, OUTPUT TO REPORT, START REPORT, TERMINATE REPORT

RETURN

The RETURN statement transfers control of execution from a FUNCTION program block. It can also return values to the calling routine. (This statement can appear only within a FUNCTION program block.)



Usage

The RETURN statement can appear only in the definition of a function. This statement tells 4GL to exit from the function and to return program control to the calling routine. (The calling routine is the MAIN, FUNCTION, or REPORT program block that contains the statement that invoked the function.)

You can use the RETURN statement in either of two ways:

- Without values, to control the flow of program execution
- With a list of one or more values, to control the flow of program execution and to return values to the calling statement

If 4GL does not encounter a RETURN statement, it exits from the function after encountering the END FUNCTION keywords.

An error is issued if RETURN appears within the MAIN statement or within a REPORT definition. To terminate execution of MAIN or REPORT from within the same program block, use PROGRAM or EXIT REPORT, respectively.

The List of Returned Values

You can specify a list of one or more expressions as values to return to the calling routine. You can use the *record*.* or the THRU or THROUGH notation to specify all or part of a list of the member variables of a record.

If the RETURN statement specifies one or more values, you can do either of the following to invoke the function:

- Explicitly execute a CALL statement with a RETURNING clause.
- Invoke the function implicitly within an expression (in the same way that you would specify a variable or a list of variables).

If the function does not return any values, you must use the CALL statement (without the RETURNING clause) to invoke the function.

The Data Types of Returned Values

4GL compares the list of expressions in the RETURN statement to arguments in the RETURNING clause of the CALL statement. A compile-time error is issued if any of these arguments do not agree with the RETURN expression list in number or position, or if data types are incompatible (see "Summary of Compatible 4GL Data Types" on page 3-46).

Similarly, if the function is invoked implicitly in an expression (as described in "Function Calls as Operands" on page 3-58), the RETURN statement is checked for agreement with the number and data types of the values that are required by the context of the calling statement.

You cannot return variables of the ARRAY data type, nor RECORD variables that contain ARRAY members. You can, however, return records that do not include ARRAY members.

The following example returns the values of **whole_price** and **ret_price** to the CALL statement. 4GL then assigns the **whole_price** and **ret_price** variables to the **wholesale** and **retail** variables in the **price** record.

```
MAIN
   DEFINE price RECORD wholesale, retail MONEY
      END RECORD
   ...
   CALL get_cust() RETURNING price.*
   ...
```

```
END MAIN
FUNCTION get_cust()
   DEFINE whole_price, ret_price MONEY
   ...
   RETURN whole_price, ret_price
END FUNCTION
```

You cannot specify variables of the BYTE or TEXT data types in the RETURN statement, just as you cannot include those data types in the RETURNING clause of a CALL statement. Because 4GL passes variables of large data types by reference, any changes made to a BYTE or TEXT variable within a function become visible within the calling routine without being returned.

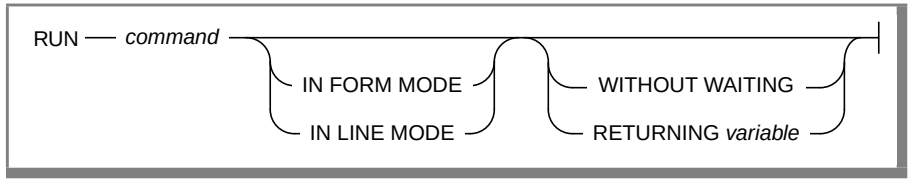
4GL allocates 5 kilobytes of memory to store character strings returned by functions, in 10 blocks of 512 bytes. A returned character value can be no larger than 511 bytes (because every string requires a terminating ASCII 0), and no more than 10 of these 511-byte strings can be returned. You can use TEXT variables to pass longer character values by reference (as described in "Passing Arguments by Reference" on page 4-18), rather than using the RETURN statement.

References

CALL, EXIT PROGRAM, EXIT REPORT, FUNCTION, WHENEVER

RUN

The RUN statement executes an operating system command line.



Element	Description
command	is a quoted string (or a CHAR or VARCHAR variable) that contains a command line for the operating system to execute.
variable	is the identifier of an INT or SMALLINT variable.

Usage

The RUN statement executes an operating system command line. You can even run a second 4GL application as a secondary process. When the command terminates, 4GL resumes execution. For example, the following statement executes the command line specified by the element i of the array variable **charval**, where **i** is an INT or SMALLINT variable:

```
RUN charval[i]
```

Unless you specify WITHOUT WAITING, RUN also has these effects:

1. Causes execution of the current 4GL program to pause
2. Displays any output from the specified command in a new 4GL window
3. After that command completes execution, closes the new 4GL window and restores the previous display in the 4GL screen

If you specify WITHOUT WAITING, all of these effects except the last are suppressed, so that the command line typically executes without any effect on the visual display. (For more information, see "The WITHOUT WAITING Clause" on page 4-343.)

Screen Display Modes

4GL recognizes two screen modes: line mode (IN LINE MODE) and formatted mode (IN FORM MODE). Besides RUN, the OPTIONS, START REPORT, and REPORT statements can explicitly specify a screen mode.

The default behavior for RUN is IN LINE MODE (so that the screen is cleared), for compatibility with releases earlier than 4GL 6.0. This mode is the opposite of the default screen mode for PIPE specifications. If no screen mode is specified, the current value from the OPTIONS statement is used.

After IN LINE MODE is specified, the terminal is in the same state (in terms of stty options) as when the program began. Usually the terminal input is in cooked mode, with interrupts enabled and input not becoming available until after a newline character is typed.

The IN FORM MODE keywords specify raw mode, in which each character of input becomes available to the program as it is typed or read.

By default, 4GL programs operate in line mode, but so many statements take it into formatted mode (including OPTIONS statements that set keys, DISPLAY AT, OPEN WINDOW, DISPLAY FORM, and other screen interaction statements) that typical 4GL programs are actually in formatted mode most of the time.

When the RUN statement specifies IN FORM MODE, the program remains in formatted mode if it currently is in formatted mode, but it does not enter formatted mode if it is currently in line mode. When the prevailing RUN option specifies IN LINE MODE, the program remains in line mode if it is currently in line mode, and it switches to line mode if it is currently in formatted mode. The same comments apply to the PIPE option.

The RETURNING Clause

The RETURNING clause saves the termination status code of what RUN executes in a 4GL variable. You can then examine this variable in your program to determine the next action to take. A status code of zero usually indicates that the command terminated normally. Non-zero exit status codes usually indicate that an error or a signal caused execution to terminate.

You can only use this clause if RUN invokes a 4GL program that contains an EXIT PROGRAM statement. When this program completes execution, the integer variable contains two bytes of termination status information:

- The low byte contains the termination status of whatever RUN executes. You can recover the value of the status code by calculating the value of the integer *variable* modulo 256.
- The high byte contains the low byte from the EXIT PROGRAM statement of the 4GL program that RUN executes. You can recover this returned code by dividing the integer *variable* by 256.

For example, suppose that a program consisted of these 4GL statements:

```
MAIN
  DEFINE ret_int INT
  LET ret_int = 5
  EXIT PROGRAM (ret_int)
END MAIN
```

The following program fragment uses RUN to invoke the compiled version of the previous program, whose filename is stored in variable **prog1**:

```
DEFINE expg_code, stat_code, ret_int INT,
       prog1 CHAR(20)
   . . .
   RUN prog1 RETURNING ret_int
   LET stat_code = (ret_int MOD 256)
   IF stat_code <> 0 THEN
      MESSAGE "Unable to run the ", prog1, " program."
   END IF
   LET expg_code = (ret_int/256)
   DISPLAY " Code from the ", prog1, " program is ", expg_code
```

Unless an error or signal terminates the program before the EXIT PROGRAM statement is encountered, the displayed value of **expg_code** is 5. Exercise caution in interpreting the integer variable, however, because under some circumstances the quotient (*variable*)/256 might not be the actual status code value that the command line returned.

If an Interrupt signal terminates the program, the integer value is 256. If a Quit signal causes the termination, the integer value is (3*256), or 758.

If a 4GL program that RUN executes can be terminated by actions of the user, you could include several EXIT PROGRAM (*number*) statements with different *number* values in different parts of the program. Examination of the code returned by RUN could indicate which EXIT PROGRAM statement (if any) was encountered during execution.

The WITHOUT WAITING Clause

The WITHOUT WAITING clause lets you execute a secondary application in the background. The syntax of WITHOUT WAITING is illustrated in the following example:

```
RUN "/$INFORMIXDIR/bin/fglgo /home/elke/sub.4gi" WITHOUT WAITING
```

Each 4GL application must have its own MAIN routine. The two programs cannot share variable scope. Each must be independently terminated, either by executing an END MAIN or EXIT PROGRAM statement in 4GL.

The WITHOUT WAITING clause is useful if you know that the command will take some time to execute, and your 4GL program does not need the result to continue. Because RUN WITHOUT WAITING executes the specified command line as a background process, it generally does not affect the visual display.

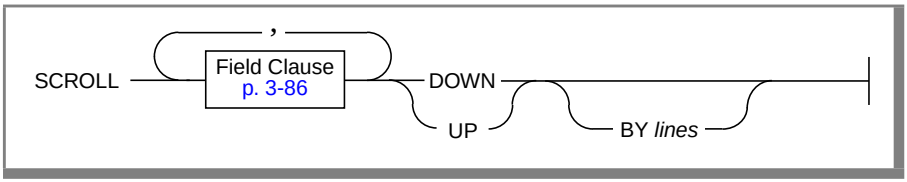
A common way to use RUN WITHOUT WAITING is to execute 4GL reports in the background.

References

CALL, FUNCTION, START REPORT

SCROLL

The SCROLL statement specifies vertical movement of displayed values in all or some of the fields of a screen array within the current form.



Element	Description
lines	is a literal integer (as described in "Literal Integers" on page 3-65), or the name of a variable containing an integer value, that specifies how far (in lines) to scroll the display.

Usage

Here $1 \leq lines \leq size$, where *size* is the number of lines in the screen array, and *lines* is the positive whole number specified in the BY clause, indicating how many lines to move the displayed values vertically in the specified fields of a screen array. If you omit the BY *lines* specification, the default is one line.

Specify UP to scroll the data toward the top of the form, or DOWN to scroll toward the bottom of the form. For example, the following statement moves *up* by one line all the displayed values in the **sc_item** screen array and fills with blanks all the fields of the last (that is, the bottom) screen record:

```
SCROLL sc_item.* UP
```

The BY clause indicates how many lines upwards or downwards to move the data; if you omit it, as in the previous example, the default is one line in the specified direction. This following example moves values in two fields *down* by three lines:

```
SCROLL stock_num, manu_code DOWN BY 3
```

The SCROLL statement ignores any bracket notation (like **sc_item[3].***) that references a single record within the array; 4GL always scrolls values in the specified fields of every screen record.

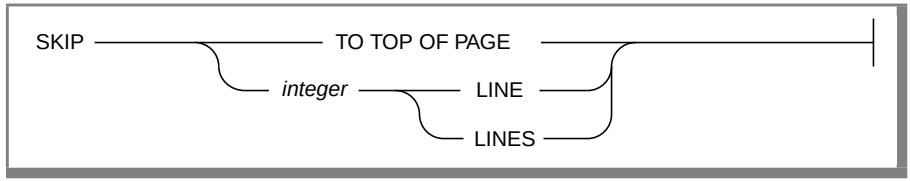
If you use SCROLL, you need to keep track of the data that is left on the screen. Many developers prefer to have the user rely on the scrolling keys of the INPUT ARRAY statement (described in "Keyboard Interaction" on page 4-219) or the DISPLAY ARRAY statement (described in "Scrolling During the DISPLAY ARRAY Statement" on page 4-111), rather than the SCROLL statement, to scroll through screen records programmatically.

References

DISPLAY ARRAY, INPUT ARRAY

SKIP

The SKIP statement inserts blank lines into a report, or finishes the current page. (It can appear only in the FORMAT section of a REPORT program block.)



Element	Description
integer	is a literal integer, specifying how many blank lines to insert.

Usage

The SKIP statement inserts blank lines into REPORT output or advances the current print position to the top of the next page. The LINE and LINES keywords are synonyms in the SKIP statement. (They are not keywords in any other statement.) Output from any PAGE TRAILER or PAGE HEADER block appears in its usual location. The next program fragment produces a list of names and addresses:

```
FIRST PAGE HEADER
   PRINT COLUMN 30, "CUSTOMER LIST"
   SKIP 2 LINES
   PRINT "Listings for the State of ", thisstate
   SKIP 2 LINES
  PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
     COLUMN 57, "ZIP", COLUMN 65, "PHONE"
  SKIP 1 LINE
PAGE HEADER
  PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
      COLUMN 57, "ZIP", COLUMN 65, "PHONE"
  SKIP 1 LINE
ON EVERY ROW
  PRINT customer_num USING "####",
  COLUMN 12, fname CLIPPED, 1 SPACE,
    lname CLIPPED, COLUMN 35, city CLIPPED, ", " , state    COLUMN 57, zipc
ode, COLUMN 65, phone
```

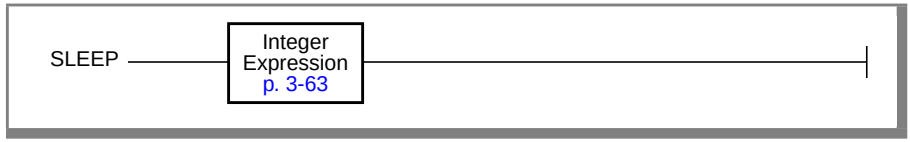
The SKIP LINES statement cannot appear within a CASE statement, a FOR loop, or a WHILE loop. The SKIP TO TOP OF PAGE statement cannot appear in a FIRST PAGE HEADER, PAGE HEADER, or PAGE TRAILER control block.

References

NEED, OUTPUT TO REPORT, PAUSE, PRINT, REPORT, START REPORT

SLEEP

The SLEEP statement suspends execution of the 4GL program for a specified number of seconds.



Usage

The SLEEP statement causes the program to pause for the specified number of seconds. This can be useful, for example, when you want a screen display to remain visible long enough for the user to read it. The following statement displays a screen message, and then waits three seconds before erasing it:

```
MESSAGE "Row has been added."
SLEEP 3
MESSAGE " "
```

In contexts where the PROMPT statement is valid, an alternative to SLEEP is the PROMPT statement. The following example suspends program execution until the user acknowledges a screen message by providing keyboard input:

```
PROMPT "Row was added. Press RETURN to continue:" FOR reply
```

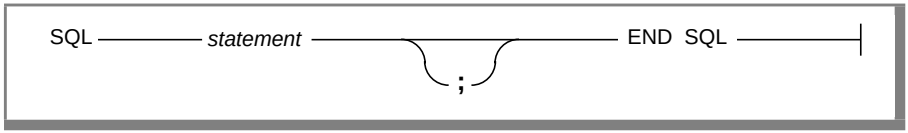
Here the screen display remains visible until the user presses RETURN (or enters anything), rather than for a fixed time interval. Entered keystrokes are stored in the **reply** variable, but their actual value can be ignored.

References

DISPLAY, EXPRESSION, MESSAGE, PROMPT

SQL

The SQL…END SQL keywords prepare, execute, and free an SQL statement.



Element	Description
statement	is a preparable SQL statement that the database server supports.

Usage

Most SQL statements that Informix 4.10 databases support can be directly embedded in 4GL source code, as well as few post-4.10 statements, such as CONNECT, CREATE PROCEDURE FROM, CREATE TRIGGER, DISCONNECT, FOREACH…WITH REOPTIMIZATION, OPEN…WITH REOPTIMIZATION, and SET CONNECTION. Other SQL statements that include syntax later than 4.10 must be prepared, if the database server can prepare and execute them.

The SQL…END SQL delimiters provide an alternative facility by which an SQL statement is automatically prepared, executed, and freed. For example, this ALTER TABLE statement includes the DISABLED keyword, which was introduced to the Informix implementation of SQL after the 4.10 release:

```
SQL
ALTER TABLE cust_fax MODIFY (lname CHAR(15)
      NOT NULL CONSTRAINT lname_notblank DISABLED)
END SQL
```

A statement like this, which has no input nor output parameters, is simply placed between the SQL and END SQL keywords. It resembles an embedded SQL statement, except that its post-4.10 syntax would have produced a compilation error if the SQL…END SQL delimiters were absent. Only one SQL statement can appear in each delimited SQL statement block.

Host Variables

Unlike the PREPARE statement, delimited SQL blocks can include SQL statements that accept host variables as input or output parameters. The 4GL host variables must be prefixed by a dollar sign ($). You can include white space between the $ symbol and the name of the variable.

In the next example, **element** is a host variable member of a RECORD variable whose index is specified by the value of SMALLINT variable **j** in the ARRAY OF RECORD **arr** in an INSERT statement, and **str** is another host variable:

```
SQL
INSERT INTO someTable(Column1, Column2)
SELECT TRIM(A.firstname) || "  " || TRIM(A.lastname),
   B.whatever FROM table1 A, Table2 B
WHERE A.PkColumn = B.FkColumn
   AND A.Filter = $arr[j].element
   AND B.Filter MATCHES $str
END SQL
```

Here the $ symbol marks **arr[j].element** and **str** as host variables, rather than database entities. (Standard 4GL notation can prefix some SQL identifiers with the @ symbol to distinguish them from 4GL identifiers.)

Returned Values

The SELECT INTO and EXECUTE PROCEDURE INTO SQL statements can return values to the 4GL program. For example, the following statement executes a stored procedure that returns two values to the 4GL program:

```
SQL
EXECUTE PROCEDURE someProcedure(12) INTO $rv1, $rv2
END SQL
```

SQL statements that have both host variables and returned values follow the same rules, as the next example of a SELECT INTO statement illustrates:

```
SQL
SELECT someProcedure(colName, $inval), otherColumn
   INTO $x, $y FROM someTable WHERE PkColumn=$pkval
END SQL
```

Although you cannot prepare EXECUTE PROCEDURE INTO and SELECT INTO *variable* statements, they can appear within an SQL block. These statements are exceptions to the rule that only preparable statements can appear in an SQL block. SQL blocks can support 4GL variables; PREPARE cannot.

Referencing and Declaring Cursors

4GL does not mangle cursor names within an SQL block. You must ensure that the application resolves cursor-name conflicts before you include statements that reference a local cursor within the SQL block.

The DECLARE statement can declare a cursor for a prepared statement, but it can also include SELECT directly (or an INSERT statement, for INSERT cursor). In these cases, the cursor declaration must precede the SQL block, as follows:

```
DECLARE c_su SCROLL CURSOR WITH HOLD FOR
SQL
SELECT TRIM(Firstname)||" " || TRIM (Lastname)
INTO $var1 FROM someTable WHERE PkColumn > $pkvar
END SQL
```

In this example, part of the DECLARE statement precedes the beginning of the SQL block, and part of it follows the END SQL delimiters.

Excluded Statements

Statements in the SQL block must be preparable. The SQL block cannot include 4GL statements that are not SQL or SPL statements. In addition, the following SQL statements, which are supported in some releases of Informix database servers, cannot appear within an SQL block.

×	ALLOCATE COLLECTION		FREE
×	ALLOCATE DESCRIPTOR	×	GET DESCRIPTOR
×	ALLOCATE ROW	×	GET DIAGNOSTICS
×	CHECK TABLE	×	INFO
	CLOSE		LOAD
	CONNECT		OPEN
	CREATE PROCEDURE FROM	×	OUTPUT
×	DEALLOCATE COLLECTION		PREPARE
×	DEALLOCATE DESCRIPTOR		PUT
×	DEALLOCATE ROW	×	REPAIR TABLE
	DECLARE	×	SET AUTOFREE
×	DESCRIBE		SET CONNECTION
	DISCONNECT	×	SET DEFERRED_PREPARE
	EXECUTE	×	SET DESCRIPTOR
	EXECUTE IMMEDIATE		UNLOAD
	FETCH		WHENEVER
	FLUSH		

The SQL statements that are marked by the **x** symbol cannot be embedded (and are also not valid in SQL blocks, nor as text within PREPARE statements).

SQL blocks do not support CREATE PROCEDURE statements. Use instead directly-embedded CREATE PROCEDURE FROM *filename* statements.

Additional Restrictions

You cannot include an SQL block within a PREPARE statement, nor a PREPARE statement within an SQL block. Question mark (?) placeholders within SQL blocks are valid in strings that are prepared, but not in other contexts. Thus, the following code generates a syntax error:

```
DECLARE cname CURSOR FOR
SQL
   SELECT * FROM SomeWhere
      WHERE SomeColumn BETWEEN ? AND ?    -- Invalid!!!
END SQL
```

Trailing semicolon (;) delimiters are valid after the SQL statement but have no effect. Semicolons that separate two statements within the SQL block cause the compilation to fail with a syntax violation error message.

Optimizer Directives and Comment Indicators

Optimizer directives and comments within delimited SQL statement blocks are passed to the database server, if you use the standard notation for these features in Version 7.3 and later Informix database servers. For example:

```
SQL SELECT {+ USE_HASH (dept/BUILD)}
       * FROM emp, dept, job WHERE loc = "Mumbai"
       AND emp.dno = dept.dno AND emp.job = job.job
END SQL
```

Here `{+` begins an optimizer directive (to use a hash join on the **dept** table) that is terminated by the } symbol at the end of the first line.

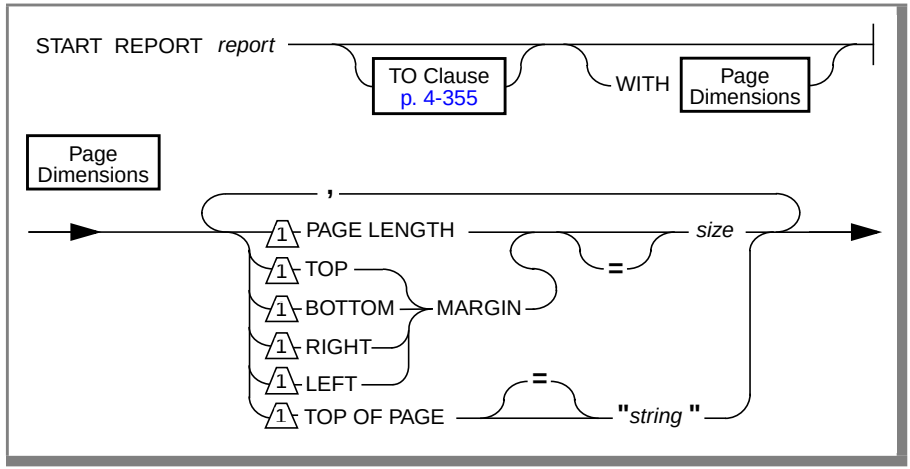
Such directives can immediately follow the DELETE, SELECT, or UPDATE keywords in SQL data manipulation statements. The + symbol must be the first character following the comment indicator that begins an optimizer directive. The # symbol is not a valid comment indicator in this context, but `{ }` or `--` comment indicators are valid within an SQL block. For more information, see the *Informix Guide to SQL: Syntax*.

References

PREPARE

START REPORT

The START REPORT statement begins processing a 4GL report and can specify the dimensions and destination of its output. (For more information, see Chapter 7.)



Element	Description
report	is the identifier of a report, as declared in a REPORT statement.
size	is an integer expression that specifies the height (in lines) or width (in characters) of a page of output from *report*, or of its margins.
string	is a quoted string that specifies the page-eject character sequence.

Usage

The START REPORT statement begins processing a report with these actions:

- Identifies a REPORT definition by which to format the input records.
- Specifies a destination and page dimensions for output of the report.
- Initializes any page headers in the FORMAT section of the report.

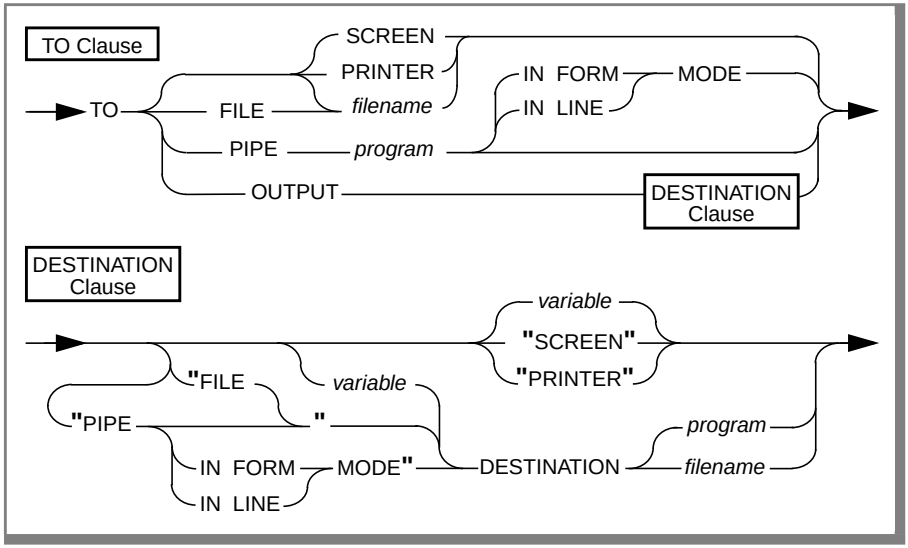
Only the name of the *report* definition is required, if you are satisfied with the default destination and page dimensions. START REPORT specifications supersede the OUTPUT section of the *report* definition, if these are different.

Do not use the START REPORT statement to reference a report that is already running. If you do, any output will be unpredictable.

START REPORT typically precedes a FOR, FOREACH, or WHILE loop in which OUTPUT TO REPORT sends input records to the report. After the loop terminates, FINISH REPORT completes the processing of output. (See "Report Drivers" on page 4-362 for more details of this topic.)

The TO Clause

The TO clause can specify a destination for output from the report. Values in this clause supersede any REPORT TO clause in the REPORT definition.



Element	Description
filename	is a quoted string (or a character variable), specifying a file to receive output from *report*. This can also include a pathname.
program	is a quoted string (or a character variable), specifying a program, a shell script, or a command line to receive output from *report*.
variable	is a character variable that specifies SCREEN, PRINTER, FILE, PIPE, PIPE IN LINE MODE, or PIPE IN FORM MODE.

If you omit the TO clause, 4GL sends report output to the destination specified in the REPORT definition, as described in "The REPORT TO Clause" on page 7-17. If neither START REPORT nor the REPORT definition specifies any destination, output is sent by default to the Report window (as described in "Sending Report Output to the Screen" on page 7-19). This default is equivalent to specifying the SCREEN keyword in the TO clause.

If the OUTPUT TO REPORT statement sends an empty set of data records to the report, the report produces no output and the TO clause has no effect, even if headers, footers, and other formatting control blocks are specified in the FORMAT section of the report definition.

The TO clause can send the report output to any of the following destinations:

- To the screen (using the SCREEN keyword)
- To a printer (using PRINTER)
- To a file (using FILE)
- To another program, command line, or shell script (using PIPE)

The following sections describe each of these options.

Dynamic Output Configuration

The TO clause is not required if you are satisfied with the default destination specifications (or default values) from the OUTPUT section of the REPORT definition. You can use the TO clause, however, to specify the destination of output from the report dynamically at runtime as follows:

- If FILE or PIPE is known as the destination at compile time, the TO FILE option can specify the *filename* (or the TO PIPE option can specify the *program)* as a character variable that is defined at runtime.
- If the destination is determined at runtime, the TO OUTPUT option can specify SCREEN, PRINTER, FILE, or PIPE as the destination by using a character variable that is defined at runtime. If this variable specifies FILE or PIPE , you can also specify a *filename* or *program* in a character variable that follows the DESTINATION keyword.

You can also specify the *program* or *filename* that follows the FILE or PIPE options as a quoted string.

Except for DESTINATION, keywords following the OUTPUT keyword within the TO clause must be delimited by quotation (") marks.

The DESTINATION keyword is not required (and is ignored, if specified) when SCREEN or PRINTER is specified by the quoted string or variable that follows the TO OUTPUT keywords.

The SCREEN Option

The TO SCREEN option sends the report output to the Report window.

The following statement specifies this option for the **cust_list** report:

```
START REPORT cust_list TO SCREEN
```

The following statement has the same effect but uses a DESTINATION clause:

```
START REPORT cust_list TO OUTPUT "SCREEN"
```

Tip: *If you intend to read output from a report that uses SCREEN as its explicit or default destination, you might want to set a PAGE LENGTH value no larger than the number of lines that the screen of your terminal can display. (Also include PAUSE statements in the FORMAT section of the report definition, so that the output remains on the screen long enough for users to examine it.)*

The PRINTER Option

The TO PRINTER option sends the report output to the device or program specified by the **DBPRINT** environment variable. If you do not set **DBPRINT**, 4GL sends output to the default printer of the system. (For information about setting **DBPRINT**, see Appendix D, “Environment Variables.”)

The following statement sends output from a report called cust_list to the printer:

```
START REPORT cust_list TO PRINTER
```

The following statement has the same effect but uses a DESTINATION clause:

```
START REPORT cust_list TO OUTPUT "PRINTER"
```

To send the output to a printer other than the default printer, you have the following options:

- Set **DBPRINT** to the desired value and use the TO PRINTER option.
- Use the TO "*filename*" option (or equivalently, TO FILE "*filename*" or TO OUTPUT "FILE" DESTINATION "*filename*") to send the report output to a file, and then send the file to a printer.
- Use the TO PIPE "*program*" option (or equivalently, TO OUTPUT "PIPE" DESTINATION "*program*") to direct output to a command line or to a shell script that sends output to a printer directly, or to send the output to a text editor for further processing before it is printed.

The FILE Option

The TO FILE *filename* option, which can include a pathname, sends the report output to a specified file. Except in a DESTINATION clause, the FILE keyword is optional, but you can include it to make your code more readable. The *filename* specification can be a quoted string or a character variable.

This example sends output from the **cu_list** report to the file **outfile**:

```
START REPORT cu_list TO "outfile"
```

The following statement has the same effect but uses a DESTINATION clause:

```
START REPORT cu_list TO OUTPUT "FILE" DESTINATION "outfile"
```

The FILE keyword is not required, but you can include it to make your code more readable. (See also "OUTPUT Section" on page 7-12 for information about how to set the default destination of output from a report within the report definition.)

The following program creates a report with default formatting, describing all the customers in the **customer** table, and saves it in the **cust_lst** file.

```
DATABASE stores7
MAIN
   DEFINE p_customer RECORD LIKE customer.*
   DECLARE q_curs CURSOR FOR SELECT * FROM customer
   START REPORT cust_list TO "cust_lst"
   FOREACH q_curs INTO p_customer.*
      OUTPUT TO REPORT cust_list(p_customer.*)
   END FOREACH
   FINISH REPORT cu _list
```

```
END MAIN
REPORT cust_lst(r_customer)
   DEFINE r_customer RECORD LIKE customer.*
   OUTPUT REPORT TO PRINTER
   FORMAT EVERY ROW
END REPORT
```

4GL ignores the OUTPUT REPORT TO PRINTER specification in the REPORT definition because the TO "*filename*" clause of the START REPORT statement overrides any default destination that REPORT specifies. But if the same **cust_list** report were referenced in another START REPORT statement that had no TO clause, its output would go to the default printer.

The PIPE Option

The TO PIPE option sends the output to a specified program, shell script, or command line and can override the screen mode (either formatted mode or line mode) that the OPTIONS statement (or the OUTPUT section of the report) specifies for any resulting screen output. You can include command-line arguments in the character string or variable specified for the TO PIPE option. The following statement pipes output from the report to the **more** program:

```
START REPORT cust_list TO PIPE "/usr/ucb/more"
```

The following statement has the same effect but uses a DESTINATION clause:

```
START REPORT cust_list
   TO OUTPUT "PIPE" DESTINATION "/usr/ucb/more"
```

Like the OUTPUT section of a REPORT definition, the TO clause can also specify whether the program is in line mode or in formatted mode when output from a report is sent to a pipe.

The following statement specifies PIPE output in formatted mode:

```
START REPORT cust_list TO PIPE "/usr/ucb/more" IN FORM MODE
```

The next example specifies line mode, using a DESTINATION clause:

```
START REPORT cust_list
   TO OUTPUT "PIPE IN LINE MODE" DESTINATION "/usr/ucb/more"
```

If neither IN LINE MODE nor IN FORM MODE is included as the screen mode specification, IN FORM MODE is the default unless a previous OPTIONS statement has set IN LINE MODE as the default. See "Screen Display Modes" on page 4-341 for more information about line mode and formatted mode in 4GL operations that produce screen output.

The WITH Clause

This clause sets the dimensions of each page of report output and overrides any conflicting OUTPUT section specifications of the report. Vertical dimensions are in lines; horizontal dimensions are in character positions, for monospace fonts. Values cannot be negative nor larger than 32,766.

You can use integer values in the WITH clause to set these values dynamically at runtime.

The five clauses that are shown in Figure 4-1 specify the physical dimensions of a page of output from the report. They can appear in any order, and each *size* specification for the dimensions can be different:

- The LEFT MARGIN clause specifies how many blank spaces to include at the start of each new line of output. The default is 5.
- The RIGHT MARGIN clause specifies the maximum number of characters in each line of output, including the left margin. If you omit this but specify FORMAT EVERY ROW, the default is 132.
- The TOP MARGIN clause specifies how many blank lines appear above the first line of text on each page of output. The default is 3.
- The BOTTOM MARGIN clause specifies how many blank lines follow the last line of output on each page. The default is 3.
- The PAGE LENGTH clause specifies the total number of lines on each page, including data, the margins, and any page headers or page trailers from the FORMAT section. The default page length is 66 lines.

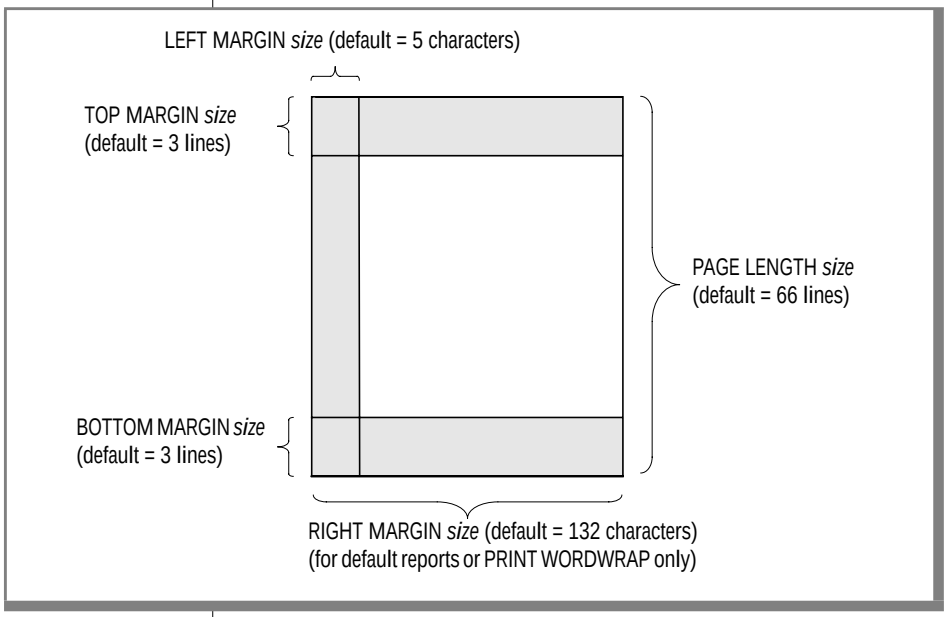


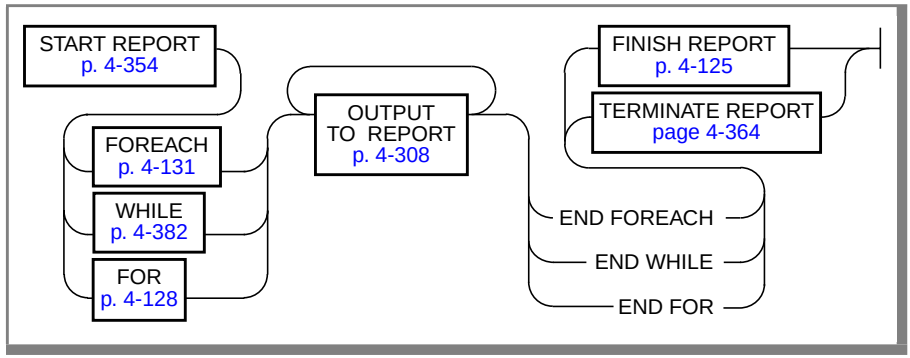
Figure 4-1
Physical Dimensions of a Page of Report Output

In addition to these five clauses, the TOP OF PAGE clause can specify a page-eject sequence for a printer. On some systems, specifying this value can reduce the time required for a large report to produce output, because SKIP TO TOP OF PAGE can substitute this value for multiple linefeeds.

Report Drivers

The START REPORT statement is logically part of a *report driver*, the portion of a 4GL application that invokes a 4GL report, retrieves data, and sends the data (as *input records*) to be formatted by the REPORT definition.

The following diagram shows the basic elements of a report driver. These elements can appear in the same or different program blocks, typically embedded within a FOR, WHILE, or FOREACH loop that uses a database cursor to pass rows from a database to the OUTPUT TO REPORT statement for formatting.



You must use START REPORT, rather than the CALL statement, to invoke a 4GL report.

If OUTPUT TO REPORT is not executed, no control blocks of the report are executed even if your program includes the START REPORT statement.

The driver also must include the FINISH REPORT statement to instruct the report when to stop processing. (You can optionally use TERMINATE REPORT to do this, as described in the next section, usually after an exception condition has been detected. Control logic within the report definition can also terminate execution of the report by using the EXIT REPORT statement.)

A report cannot invoke itself. The report driver must be in a different program block from the REPORT definition. See also "The Report Driver" on page 7-5 for additional information about report drivers.

References

DEFINE, EXIT REPORT, FINISH REPORT, FOR, FOREACH, NEED, OUTPUT TO REPORT, PAUSE, PRINT, REPORT, SKIP, TERMINATE REPORT, WHILE

TERMINATE REPORT

The TERMINATE REPORT statement stops processing a currently running 4GL report, typically because some error has been detected. (Use FINISH REPORT instead for the normal termination of report processing.)

Element	Description
report	is the name of a 4GL report, as declared in a REPORT statement.

Usage

The TERMINATE REPORT statement causes 4GL to terminate the currently executing report driver without completing the normal processing of the report. TERMINATE REPORT must be the last statement in the report driver, and it must follow a START REPORT statement that specifies the name of the same report. (For more information about 4GL report definitions and report drivers, see Chapter 7, "INFORMIX-4GL Reports.")

The TERMINATE REPORT statement has the following effects:

- Terminates the processing of the report
- Deletes any intermediate files or tables that were created in processing a REPORT statement

Unlike the FINISH REPORT statement (page 4-125), TERMINATE REPORT does not format any values from aggregate functions and does not execute any statements in the ON LAST ROW section of the specified REPORT routine.

If the report includes an ORDER BY section that sorts the input records within the report (rather than specifying the ORDER EXTERNAL BY option), the effect of the TERMINATE REPORT statement is to produce an empty report.

This statement is useful if you are not interested in output that is missing some input records, but your code detects a condition that prevents some report data from being processed. For example, an SQL statement in the report driver returns an error, the designated printer fails, or something disables a process to which the report sends output through a pipe.

In general, the TERMINATE REPORT statements should be conditional on detection of an error, as in the following example. Otherwise, you would normally execute FINISH REPORT rather than TERMINATE REPORT. The following program specifies a report on data from the **customer** table:

```
DATABASE stores
   DEFINE trouble INT
MAIN
   DEFINE p_customer RECORD LIKE customer.*
   DECLARE q_curs CURSOR FOR SELECT * FROM customer
   LET trouble = 0
   START REPORT cust_list TO "cust_listing"
   FOREACH q_curs INTO p_customer.*
      OUTPUT TO REPORT cust_list(p_customer.*)
      IF status !=0 THEN LET trouble = trouble + 1
         EXIT FOREACH
      END IF
   END FOREACH
   IF trouble > 0 THEN TERMINATE REPORT cust_list
      ELSE FINISH REPORT cust_list
   END IF
END MAIN
REPORT cust_list(r_customer)
   DEFINE r_customer RECORD LIKE customer.*
   FORMAT
       PAGE HEADER
          PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35,
          "LOCATION", COLUMN 57, "ZIP", COLUMN 65, "PHONE"
          SKIP 1 LINE
       ON EVERY ROW
          PRINT r_customer.customer_num USING "####",
          COLUMN 12, r_customer.fname CLIPPED, 1 SPACE,
          r_customer.lname CLIPPED, COLUMN 35,
          r_customer.city CLIPPED, ", " ,
          r_customer.state, COLUMN 57,
          r_customer.zipcode, COLUMN 65, r_customer.phone
       ON LAST ROW
          SKIP 1 LINE
          PRINT COLUMN 12, "TOTAL NUMBER OF CUSTOMERS:",
          COLUMN 57, COUNT(*) USING "##"
END REPORT
```

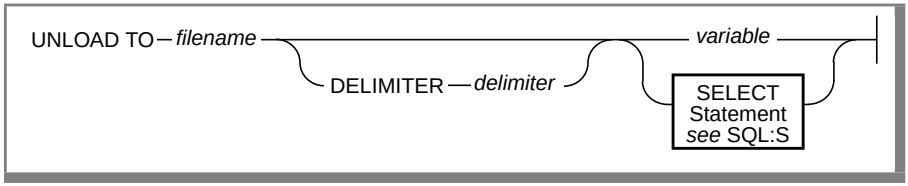
If the module variable **trouble** has a non-zero value when the FOREACH loop terminates, the TERMINATE REPORT statement is executed rather than FINISH REPORT. In this case, no statements in the ON LAST ROW control block are executed, and the aggregate COUNT(*) function is not evaluated.

References

EXIT REPORT, FINISH REPORT, OUTPUT TO REPORT, REPORT, START REPORT

UNLOAD

The UNLOAD statement copies data from the current database to a file.



Element	Description
delimiter	is a literal delimiter symbol, enclosed in quotation marks, or a CHAR or VARCHAR variable that contains a single delimiter symbol to separate adjacent columns in the character-string representation of each row from the database in the output file.
filename	is a quoted string (or CHAR or VARCHAR variable) that specifies the name of an output file in which to store the rows retrieved by the SELECT statement. This string can include a pathname.
variable	is a CHAR or VARCHAR variable that contains a SELECT statement. (See the *Informix Guide to SQL: Syntax* for the syntax of SELECT.)

Usage

The UNLOAD statement must include a SELECT statement (directly, or in a variable) to specify what rows to copy into *filename*. UNLOAD does not delete the copied data. The user must have SELECT privileges on every column specified in the SELECT statement. (For details of database-level and table-level privileges, see the GRANT statement in *Informix Guide to SQL: Syntax*.)

The DATABASE or CONNECT statement must first open the database that SELECT accesses before UNLOAD can be executed.

You cannot use the PREPARE statement to preprocess an UNLOAD statement.

The Output File

The *filename* identifies an output file in which to store the rows retrieved from the database by the SELECT statement. In the default (U.S. English) locale, this file contains only ASCII characters. (In other locales, output from UNLOAD can contain characters from the codeset of the locale.)

A set of values in output representing a row from the database is called an *output record*. A newline character (ASCII 10) terminates each output record.

The UNLOAD statement represents each value in the output file as a string of ASCII characters, according to the declared data type of the database column.

Data Type	Output Format
CHARACTER, TEXT, VARCHAR	Trailing blanks are dropped from CHAR and TEXT (but not from VARCHAR) values. A backslash (\ ) is inserted before any literal backslash or delimiter character and before a newline character in a VARCHAR value or as the last character in a TEXT value.
DECIMAL, FLOAT, INTEGER, MONEY, SMALL-FLOAT, SMALLINT	Values are written as literals with no leading blanks. (For more information, see "Number Expressions" on page 3-66.) MONEY values are represented with no leading currency symbol. Zero values are represented as `0` for INTEGER or SMALLINT columns, and as `0.00` for FLOAT, SMALLFLOAT, DECIMAL, and MONEY columns. SERIAL values are represented as literal integers (as described in "Integer Expressions" on page 3-63).
DATE	Values are written in the format *month/day/year* (as described in "Numeric Date" on page 3-75) unless some other format is specified by the **DBDATE** environment variable.
DATETIME, INTERVAL	INTERVAL values are formatted *year-month*, or else as *day hour*:*minute*:*second.fraction*, or a contiguous subsets thereof; DATETIME values must be in the format *year-month-day hour:minute:second.fraction*, or a contiguous subset, without DATETIME or INTERVAL keywords or qualifiers. Time units outside the declared precision of the database column are omitted.
BYTE	Values are written in ASCII hexadecimal form, without any added blank or newline characters. The logical record length of an output file that contains BYTE values can be very long, and thus might be very difficult to print or to edit.

Null values of any data type are represented by consecutive delimiters in the output file, without any characters between the delimiter symbols.

Quotation marks (") are required around a literal filename. The following statement copies any rows where the value of customer.customer_num is greater than or equal to 138, and stores them in a file called **cust_file**:

```
UNLOAD TO "cust_file" DELIMITER "!"
   SELECT * FROM customer WHERE customer_num> = 138
```

This produces this output file, **cust_file**:

```
138!Jeffery!Padgett!Wheel Thrills!3450 El Camino!Suite 10!Palo
Alto!CA!94306!!
139!Linda!Lane!Palo Alto Bicycles!2344 University!!Palo
Alto!CA!94301!(415)323-5400
```

GLS

UNLOAD uses the environment variables **DBFORMAT**, **DBMONEY**, **DBDATE, GL_DATE**, and **GL_DATETIME** to determine the format of the output file. The precedence of these format specifications is consistent with that of 4GL forms and reports. For additional information about environment variables that are used in global language support, see Appendix E. ♦

The DELIMITER Clause

The DELIMITER clause specifies the delimiting character that separates the data contained in each column in a row in the output file. Enclosing quotation marks are required around a literal delimiter symbol.The UNLOAD statement in the following program specifies semicolon (;) as the delimiter symbol:

```
DATABASE stores
MAIN
   DEFINE hostvar SMALLINT
   LET hostvar = 103
   UNLOAD TO "custfile" DELIMITER ";"
      SELECT * FROM customer WHERE customer_num = hostvar
END MAIN
```

If you omit the DELIMITER clause, the default delimiter symbol is the value of the **DBDELIMITER** environment variable, or else a pipe (|) symbol (= ASCII 124) if **DBDELIMITER** is not set. For details, see Appendix D.

Do not specify any of the following characters as the delimiter symbol:

- Hexadecimal numbers (0 through 9, *a* through *f*, or *A* through *F*)
- Newline character or CONTROL-J
- The backslash (\ )

Host Variables

Do not attempt to substitute question marks (?) in place of host variables to make the SELECT statement dynamic, because this usage has binding problems. The following 4GL code is an example of this (unsupported) syntax:

```
FUNCTION func_unload()



   DEFINE query CHAR(250)
   DEFINE file  CHAR(20)
   DEFINE del   CHAR(1)
   DEFINE i     INTEGER, j INTEGER, k INTEGER

   LET i     = 100
   LET j     = 30
   LET k     = 400
   LET del   = ";"
   LET file  = "/dev/tty"

   LET query = "select * from systables where tabid >= ?"
      " and ncols >= ? and rowsize >= ?"

   UNLOAD TO file DELIMITER del query

END FUNCTION
```

A corrected version of the function func_unload( ) follows:

```
FUNCTION func_unload()

   DEFINE file  CHAR(20)
   DEFINE del   CHAR(1)
   DEFINE i     INTEGER, j INTEGER, k INTEGER

   LET i     = 100
   LET j     = 30
   LET k     = 400
   LET del   = ";"
   LET file  = "/dev/tty"

   UNLOAD TO file DELIMITER del
      SELECT * FROM Systables
         WHERE tabid >= i AND ncols >= j AND rowsize >= k

END FUNCTION
```

The Backslash Escape Character

The backslash symbol (\ ) serves as an escape character in the output file to indicate that the next character in a data value is a literal. The UNLOAD statement automatically inserts a preceding backslash to prevent literal characters from being interpreted as special characters in the following contexts:

- The backslash character appears anywhere in a value from a CHAR, VARCHAR, or TEXT column.
- The delimiter character appears anywhere in a value from a CHAR, VARCHAR, or TEXT column.
- The newline character appears anywhere in a value from a VARCHAR column or as the last character in a TEXT column.

If LOAD (or **tbload** or **onload**) inserts output from UNLOAD into the database, all backslash symbols that were used as escape characters are automatically stripped from the values that are inserted into the database.

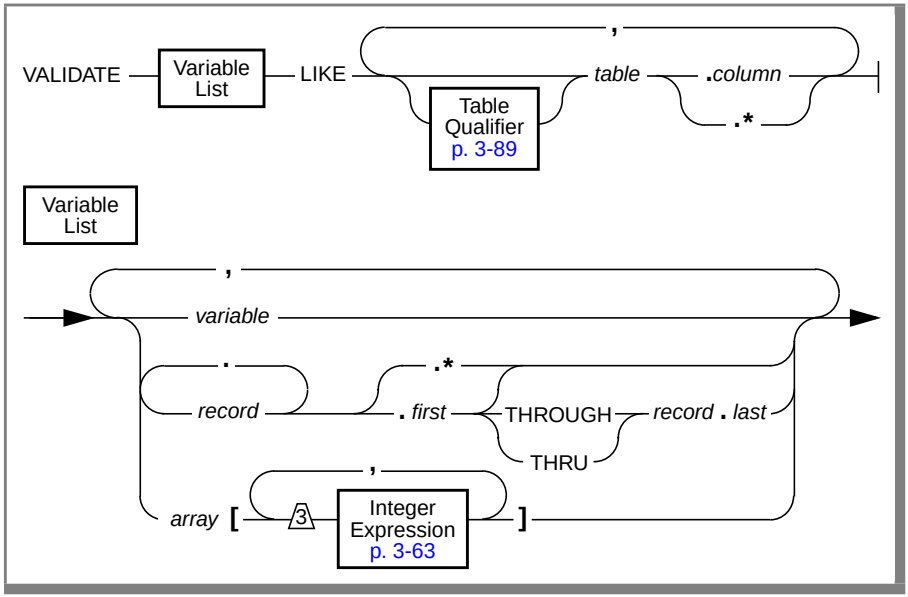
See also the description of UNLOAD in the *Informix Guide to SQL: Syntax*.

References

DATABASE, LOAD

VALIDATE

The VALIDATE statement tests whether the value of a variable is within the range of values for a corresponding column in the **syscolval** table.



Element	Description
array	is a variable of the ARRAY data type.
column	is the name of a column of *table* for which an INCLUDE value exists in the **syscolval** table of the default database.
first	is the name of a member variable of *record* to be validated.
last	is another member that was declared later than *first* in *record*.
record	is the name of a program record to be validated.
table	is the name or synonym of the table or view that contains *column*.
variable	is the name of a variable (of a simple data type) to be validated.

Usage

If your program inserts data from a screen form, 4GL automatically checks for validation criteria that the form attributes specify.

If your program inserts data into the database from sources other than a screen form, you can use the VALIDATE statement to apply validation criteria from the **syscolval** table. (For more information, see "Default Attributes" on page 6-80.)

This statement has no effect unless the **upscol** utility has assigned INCLUDE values in the **syscolval** table for at least one of the database columns in the column list of the VALIDATE statement. If the value of a variable does not conform with the INCLUDE value in the **syscolval** table, 4GL sets the **status** variable to -1321. If you specify a list of variables and receive a negative **status** value, you must test the variables individually to detect the non-conforming value.

Because INCLUDE can specify values only for columns of simple data types, the list of variables cannot include BYTE or TEXT variables. You can, however, include members of RECORD variables or elements of ARRAY variables if these members or elements are of simple data types.

The LIKE Clause

The LIKE clause specifies the database columns with which to validate the variables. The variables must match the specified columns in order and number, and must be of the same or compatible data types (as described in "Summary of Compatible 4GL Data Types" on page 3-46). You must prefix the name of each column with that of the table. For example, the following statement validates two variables against two columns in the **stock** table:

```
VALIDATE var1, var2 LIKE stock.stock_num, stock.manu_code
```

In an ANSI-compliant database, you must qualify each table name with that of the owner of the table (*owner.table*). The only exception is that you can omit the *owner* prefix for any tables that you own. For example, if you own **tab1**, Krystl owns **tab2**, and Nick owns **tab3**, you could use this statement:

```
VALIDATE var1, var2, var3
   LIKE tab1.var1, krystl.tab2.var2, nick.tab3.var3
```

You can also reference columns in tables outside the default database. See "Table Qualifiers" on page 3-89 for more information. Even if you specify the name of a database in the table qualifier, however, you must also include a DATABASE statement before the first program block in the same module to specify a default database at compile time. (For more information, see "The Default Database at Compile Time" on page 4-73.)

The syscolval Table

The VALIDATE statement looks for validation criteria in the INCLUDE column of the **syscolval** table. To enter values into this table, use the **upscol** utility, as described in Appendix B. If a column does not have any INCLUDE value in **syscolval**, 4GL takes no action. If the current database is not ANSI-compliant, **upscol** creates a single **syscolval** table for all users.

ANSI

In an ANSI-compliant database, each user of the **upscol** utility creates an *owner*.**syscolval** table, which stores validation criteria only for the tables owned by that user. If you omit the *owner* qualifier for a table that you own, your **syscolval** table becomes the source for validation criteria when you compile the program. If the *owner*.**syscolval** table does not exist, the VALIDATE statement takes no action. (You can also include the *owner* name in a database that is not ANSI-compliant. If the *owner* value is incorrect, however, 4GL issues an error.) ♦

The compiler looks in the default database for **syscolval**. Any changes to **syscolval** after compilation have no effect on the 4GL program, unless you recompile the program.

This example assumes that the **state** field in the **customer** table has validation criteria in **syscolval** that limit the valid states to those in the Western region:

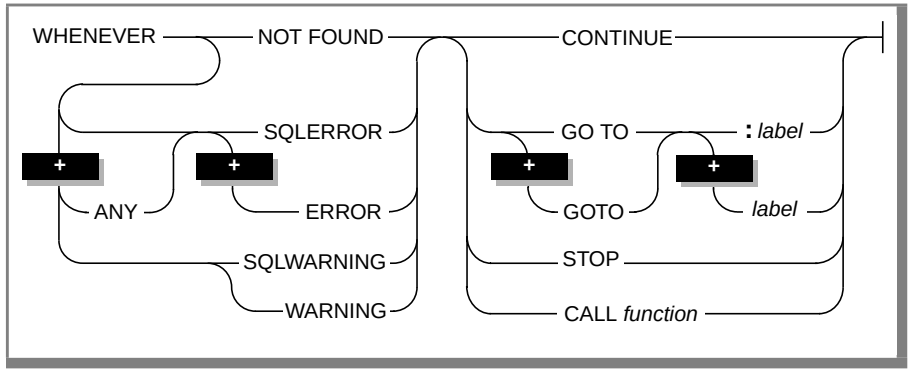
```
INPUT BY NAME p_customer.*
   ...
   AFTER FIELD zipcode
      CALL check_zip(p_customer.zipcode)
      RETURNING state_zip
      WHENEVER ERROR CONTINUE
      VALIDATE state_zip LIKE customer.state
      WHENEVER ERROR STOP
      IF (status < 0) THEN
         ERROR "This zipcode is not in the Western region."
      END IF
   ...
END INPUT
```

References

DATABASE, DEFINE, INITIALIZE, INPUT, INPUT ARRAY, WHENEVER

WHENEVER

The WHENEVER statement traps SQL and 4GL errors, warnings, and end-of-data conditions that might occur during program execution.



Element	Description
function	is the identifier of a function (with no parentheses and no argument list) to be invoked if the specified exceptional condition occurs.
label	is a statement label (in the same program block) to which 4GL transfers control when the specified exceptional condition occurs.

Usage

WHENEVER can appear only within a MAIN, REPORT, or FUNCTION program block. It can trap errors, warnings, and the NOT FOUND condition at runtime.

The WHENEVER statement must include two items of information:

- Some type of exceptional condition
- An action to take if the specified exceptional condition is detected

These specifications correspond respectively to the left-hand (*conditions*) and right-hand (*actions*) portions of the preceding syntax diagram.

Using WHENEVER is equivalent to including code after every SQL statement, and (optionally) after certain other 4GL statements, instructing 4GL to take the specified action at runtime if the exceptional condition is detected. If you use WHENEVER ERROR with any option but STOP or CONTINUE, 4GL tests for errors by polling the global variable **status**.

The following topics are discussed in this section:

The Scope of the WHENEVER Statement

The scope of a WHENEVER statement is from its location in a program block until the next WHENEVER statement with the same exceptional condition in the same module (except that both ERROR and ANY ERROR reset both ERROR and ANY ERROR). Otherwise, the WHENEVER statement remains in effect for that exceptional condition until the end of the module.

For example, the following program has three WHENEVER statements, two of which describe WHENEVER ERROR conditions. In line 4, CONTINUE is specified as the action to take; line 8 specifies STOP as the action for the same ERROR condition. Any errors that 4GL encounters after line 4 but before line 8 are ignored. After line 8, and for the rest of the program, any errors that are encountered cause the program to terminate.

```
MAIN                                                    --1
   DEFINE char_num INTEGER                              --2
   DATABASE test                                        --3
   WHENEVER ERROR CONTINUE                              --4
   PRINT "Program will now attempt first insert."       --5
   INSERT INTO test_color VALUES ("green")              --6
   WHENEVER NOT FOUND CONTINUE                          --7
   WHENEVER ERROR STOP                                  --8
```

```
    PRINT "Program will now attempt second insert." --9
    INSERT INTO test_color VALUES ("blue")          --10
    CLOSE DATABASE                                  --11
    PRINT "Program over."                           --12
END MAIN                                            --13
```

The ERROR Condition

The ERROR keyword directs 4GL to take the specified action if **sqlcode** in the **SQLCA** global record is negative after any SQL statement, or if a VALIDATE statement or screen interaction statement (described in "OPTIONS" on page 4-291) fails. For example, this statement causes SQL errors to be ignored:

```
WHENEVER ERROR CONTINUE
```

If you do not use any WHENEVER statements, the default action for WHENEVER ERROR (or for any other condition) is CONTINUE.

Besides checking for errors after SQL statements, the WHENEVER ERROR statement also checks for errors after screen interaction statements and after VALIDATE statements. (In a WHENEVER statement, and only in this context, SQLERROR is a synonym for ERROR. You cannot, for example, substitute SQLERROR for ERROR in an OPTIONS or ERROR statement.)

The ANY ERROR Condition

Without ANY, WHENEVER ERROR resets **status** to the **sqlcode** value only if the error occurs during an SQL, VALIDATE, or screen interaction statement. The ANY keyword before ERROR, however, resets **status** after evaluating any 4GL expression. The **-anyerr** command-line option is described in Chapter 1. This option can override WHENEVER statements in determining whether the **status** variable is reset when 4GL expressions are evaluated.

The NOT FOUND Condition

If you use the NOT FOUND keywords, SELECT and FETCH statements (and implicit FETCH or SELECT statements in FOREACH or UNLOAD statements) are treated differently from other SQL statements. The NOT FOUND keywords check for the end-of-data condition in the following cases:

- A FETCH statement attempts to retrieve a row beyond the first or last row in the active set.
- A SELECT statement returns no rows.

In both cases, the **sqlcode** variable is set to 100. The following statement calls the **no_rows( )** function whenever the NOT FOUND condition is detected:

```
WHENEVER NOT FOUND CALL no_rows
```

Although both NOT FOUND and NOTFOUND indicate the same condition, you cannot use them interchangeably. Use NOTFOUND (one word) in testing **status**, and use NOT FOUND (two words) in the WHENEVER statement.

The WARNING Condition

If you use the WARNING keyword (or its synonym SQLWARNING), any SQL statement that generates a warning also produces the action indicated by the WHENEVER WARNING statement. If a warning occurs, the first field of the **SQLAWARN** record is set to W. For example, the following statement causes a program to halt execution whenever a warning condition exists:

```
WHENEVER WARNING STOP
```

The GOTO Option

Use the GOTO clause to transfer control to the statement identified by the specified statement label. The keywords GO TO are a synonym for GOTO.

The label that follows the GOTO keyword must be declared by a LABEL statement in the same FUNCTION, REPORT, or MAIN program block as the current WHENEVER statement. For example, the WHENEVER statement in this program fragment transfers control to the statement labeled **missing:** whenever the NOT FOUND condition occurs:

```
FUNCTION query_data()
   ...
   FETCH FIRST a_curs INTO p_customer.*
   WHENEVER NOT FOUND GO TO :missing
   ...
   LABEL missing:
      MESSAGE "No customers found."
      SLEEP 3
      MESSAGE ""
END FUNCTION
```

If your source module contains more than one program block, you might need to redefine the error condition. For example, suppose that the module contains three functions, and the first function includes a WHENEVER ... GOTO statement and a corresponding LABEL statement. When compilation moves from the first FUNCTION definition to the next, the WHENEVER specification still specifies a jump to the label, but that label is no longer defined in the second FUNCTION block.

If the compiler processes an SQL statement within that block before you redefine the action to take for the same condition (for example, to WHENEVER ERROR CONTINUE), a compilation error results.To avoid this error, you can reset the error condition by issuing another WHENEVER statement. Alternatively, you can use the LABEL statement to define the same statement label in each function, or you can use the CALL option of WHENEVER to invoke a separate function.

The CALL Option

The CALL clause transfers program control to the specified function. Do not include parentheses after the function name. You cannot pass variables to the function. For example, the following statement executes a function called **error_recovery( )** if an error condition is detected:

```
WHENEVER ERROR CALL error_recovery
```

If you use the BEGIN WORK statement in a function called by WHENEVER, always specify WHENEVER ERROR CONTINUE and WHENEVER WARNING CONTINUE before the ROLLBACK WORK statement. This prevents the program from looping if ROLLBACK WORK encounters an error or warning.

You cannot specify the name of a stored procedure after the CALL keyword. To invoke a stored procedure, use the CALL clause to execute a function that contains an EXECUTE PROCEDURE statement for the desired procedure.

The CONTINUE Option

The CONTINUE keyword instructs program to take no action. This keyword turns off a previously specified option and is the default for all conditions.

The STOP Option

The STOP keyword exits if the specified condition occurs. The following statement terminates execution when the database server issues a warning:

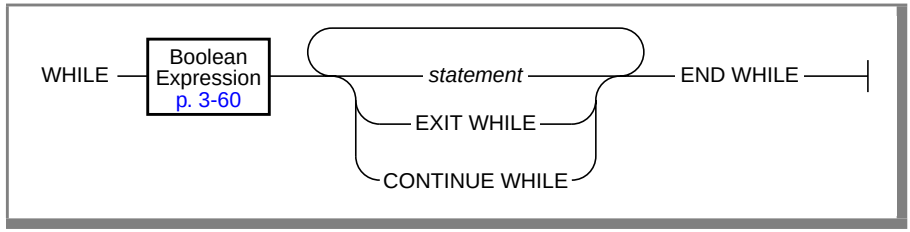
```
WHENEVER WARNING STOP
```

References

CALL, DEFER, FOREACH, FUNCTION, GOTO, IF, LABEL, VALIDATE

WHILE

The WHILE statement executes a block of statements while a condition that you specify by a Boolean expression is true.



Element	Description
statement	is an SQL statement or other 4GL statement.

Usage

If the Boolean expression is true, 4GL executes the statements that follow it, down to the END WHILE keywords. 4GL again evaluates the Boolean expression, and if it is still true, 4GL executes the same statement block. If the expression is false, however, 4GL terminates the WHILE loop and resumes execution after the END WHILE keywords. (If the expression is already false on entry to the WHILE statement, program control passes directly to the statement immediately following END WHILE.)

In the following example, if the user responds to the prompt by typing y, 4GL calls the **enter_order( )** function and displays a prompt, asking whether the user wants to enter another order. 4GL continues entering orders and prompting the user, as long as the user types y in response to the prompt:

```
LET answer = "y"
WHILE answer = "y"
   CALL enter_order()
   PROMPT "Do you want to enter another order (y/n) : "
      FOR answer
END WHILE
```

The CONTINUE WHILE or EXIT WHILE keywords can interrupt the sequence of statements in a WHILE loop as described in the following sections.

If your database has transaction logging, it is advisable that the entire WHILE loop be within a transaction. (For more information about the SQL statements that support transactions, see *Informix Guide to SQL: Tutorial*.)

The CONTINUE WHILE Statement

The CONTINUE WHILE statement interrupts a WHILE loop and causes 4GL to evaluate the Boolean expression again. If the expression is still true, 4GL begins a new iteration of the statements in the loop. If the expression is no longer true, control passes to the statement that follows END WHILE.

The EXIT WHILE Statement

Use the EXIT WHILE statement to terminate the WHILE loop. When the EXIT WHILE keywords are encountered, 4GL takes the following actions:

- Skips statements between EXIT WHILE and END WHILE
- Resumes execution at the statement following END WHILE

The following example demonstrates using the EXIT WHILE statement within a WHILE loop. If the **status** variable is not equal to zero, 4GL executes the statements that follow the END IF keywords. Otherwise, 4GL exits from the WHILE loop and executes the following DISPLAY statement:

```
WHILE TRUE
   ...
   IF status = 0 THEN EXIT WHILE
   END IF
   ...
END WHILE
DISPLAY p_customer.* TO customer.*
```

If, as in this example, statements in the WHILE loop cannot change the value of the Boolean expression to FALSE, the WHILE loop cannot terminate unless you specify EXIT WHILE, GOTO, or some other logical way out of the loop.

The END WHILE Keywords

The END WHILE keywords indicate the end of the WHILE loop and cause 4GL to evaluate the Boolean expression again. If the expression is still true, 4GL re-executes the statements in the loop. If the expression is no longer true, 4GL passes control to the statement that follows END WHILE.

References

CONTINUE, END, EXIT, FOR, FOREACH

Chapter

5

Built-In Functions and Operators

In This Chapter

This chapter describes the kinds of built-in functions you can use in INFORMIX-4GL applications. It also describes 4GL operators. Information about the syntax and usage of these built-in functions and operators appears in the sections that follow, arranged in alphabetical order according to the name of the function or operator. (Aggregate functions, arithmetic operators, and most Boolean operators are described under those headings, rather than under their individual names.)

Functions in 4GL Programs

In 4GL, a *function* is a named collection of statements that perform a task. (In some programming languages, terms like *method*, *subroutine*, and *procedure* correspond to a *function* in 4GL.) If you need to repeat the same series of operations, you can call the same function several times, rather than specify the same steps for each repetition. This construct supports the structured programming design goal of segmenting source code modules into logical units, each of which has only a single entry point and controlled exit points.

4GL programs can invoke the following types of functions:

- Programmer-defined 4GL functions
- 4GL built-in functions
- SQL built-in functions
- C functions
- ESQL/C functions (if you have INFORMIX-ESQL/C)

The FUNCTION statement defines a function. Variables that are declared within a function are local to it, but functions that are defined in the same module can reference any module-scope variables that are declared in that module. See the descriptions in "Programmer-Defined 4GL Functions" on page 5-8, and "FUNCTION" on page 4-140. The other types of functions that you can call from a 4GL program are briefly discussed on the next two pages.

Built-In 4GL Functions

The *built-in functions* of 4GL are predefined functions that support features of the INFORMIX-4GL language. Except for the fact that no FUNCTION definition is required, built-in functions behave exactly like the 4GL functions that you define with the FUNCTION statement.

Built-in 4GL functions include the following features:

- You can invoke them with the CALL statement. (If they return a single value, they can appear without CALL in 4GL expressions.)
- They require parentheses, even if the argument list is empty.
- You cannot invoke them from SQL statements.
- You can invoke them from a C program.

If you use the FUNCTION statement to define a function with the same name as a built-in function, your program cannot invoke the built-in function. Each of the following 4GL built-in functions is described in this chapter.

ARG_VAL(*int-expr*)	FGL_KEYVAL(*char-expr*)
ARR_COUNT( )	FGL_LASTKEY( )
ARR_CURR(*char-expr*)	FGL_SCR_SIZE( )
CURSOR_NAME ("*identifier*")	LENGTH(*char-expr*)
DOWNSHIFT(*char-expr*)	NUM_ARGS( )
ERR_GET(*int-expr*)	ORD(*char-expr*)
ERR_PRINT(*int-expr*)	SCR_LINE( )
ERR_QUIT(*int-expr*)	SET_COUNT(*int-expr*)
ERRORLOG(*int-expr*)	SHOWHELP(*int-expr*)
FGL_DRAWBOX(*nlines, ncols, begy, begx, color*)	STARTLOG("*filename.filetype*")
FGL_GETENV(*char-expr*)	UPSHIFT(*char-expr*)
FGL_GETKEY( )	

FGL_DRAWBOX( ) arguments are integer expressions, except for *color*, which can also be a keyword. The aggregates AVG( ), COUNT(*), MAX( ), MIN( ), PERCENT(*), and SUM( ) are valid only in reports. For more information, see "Aggregate Report Functions" on page 5-14.

Built-In and External SQL Functions and Procedures

Informix database servers support built-in SQL functions, some of which have the same names as built-in 4GL functions or operators. Some Informix database servers also support the syntax of the following statements.

CREATE FUNCTION	CREATE ROUTINE FROM
CREATE FUNCTION FROM	EXECUTE FUNCTION
CREATE PROCEDURE FROM	EXECUTE PROCEDURE

These SQL statements register and execute external functions and stored procedures, which resemble 4GL functions but are executed by the database server.)

Calls to SQL, SPL, and external functions can appear in SQL statements but not in other 4GL statements. (See the *Informix Guide to SQL: Syntax*.)

C Functions

You can use the CALL statement or an expression to invoke properly written C language functions within a 4GL program. Such functions are often helpful for specialized tasks that are not easily written in 4GL, such as processing binary I/O. For information on the application program interface (API) between 4GL and the C programming language, see Appendix C, "Using C with INFORMIX-4GL."

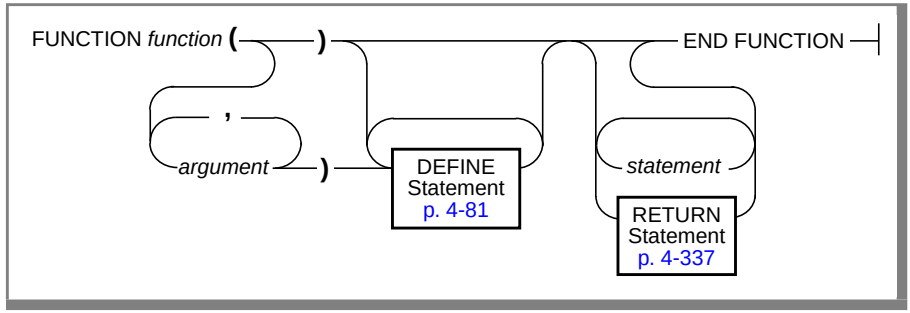
Unlike 4GL identifiers, names of C functions and their arguments are case-sensitive. They must typically appear in lowercase letters in the function call.

ESQL/C Functions

If you have INFORMIX-ESQL/C, your 4GL program can also call compiled ESQL/C functions that you write, as well as ESQL/C library functions. See "Running Programs That Call C Functions" on page 1-90.

Programmer-Defined 4GL Functions

The FUNCTION program block begins with the FUNCTION keyword and ends with the END FUNCTION keywords. These keywords enclose a program block of the 4GL statements that compose the function, and that are executed when the function is invoked.



Element	Description
argument	is the name of a formal argument to the function.
function	is the identifier that you declare here for the function.
statement	is an SQL statement or other 4GL statement.

The left-hand portion of this diagram, including the identifier of the function and the list of formal arguments, is sometimes called the *function prototype* (described in detail in "The Prototype of the Function" on page 4-141). This portion resembles the prototype of a report.

Names of functions must be unique among the names of functions, reports, and global variables within the program and cannot be the same as any of the formal arguments of the same function. See "FUNCTION" on page 4-140 for details of how to define 4GL functions.

The right-hand portion of this diagram, including the declarations of formal arguments and of local variables, and the *statement* block, is sometimes called the FUNCTION *program block*. This portion can include any executable statement of SQL or 4GL except the report execution statements (described in "4GL Report Execution Statements" on page 4-14). The entire FUNCTION program block must be defined within a single source module.

No other FUNCTION, REPORT, or MAIN program block can be included in a FUNCTION definition, but it can include statements that produce a report, call a function, or execute a RUN statement that invokes another program.

Invoking Functions

Except for SQL functions and stored procedures, which are called in SQL expressions, 4GL programs can use the CALL statement to invoke functions. In some contexts, however, you can also call functions implicitly:

- If a function returns a single value, you can invoke the function simply by specifying its name (and any required arguments) within an expression where a value of the returned data type is valid.
- Exception-handling features of 4GL can automatically invoke the function that you specify in the WHENEVER…CALL statement.

Passing Arguments and Returning Values

The program block containing the CALL statement or expression that invokes a function is called the *calling routine*. Functions can receive information from (and return values to) the calling routine. In the typical case where this is a different program block, values from the calling routine and from other program blocks are visible to the function only through global or module variables, or through the argument list of the calling statement.

For most data types, the RETURN statement in the function and RETURNING clause of the CALL statement specify any values that the function returns to the calling routine. This mechanism for communication between the function and its calling routine is called *passing by value*.

Arguments of the large data types (BYTE or TEXT) are processed in a different way, called *passing by reference*. The BYTE or TEXT variables appear in the argument list of the calling statement, but what is passed to the function is a pointer to the variables. The RETURN statement and RETURNING clause cannot include BYTE or TEXT variables. (The built-in 4GL functions that are described in this chapter all pass their arguments by value, rather than by reference.)

Invoking SQL Functions

You can invoke predefined SQL functions and operators in 4GL programs, but only within SQL statements. (For information about SQL functions, see the description of function expressions in the *Informix Guide to SQL: Syntax*.) For example, the USER operator of SQL can appear in a SELECT statement:

```
DEFINE usr_id CHAR(9)
...
SELECT USER INTO usr_id FROM systables WHERE tabid = 1
```

Important: *Some built-in 4GL functions and operators have the same names as SQL functions or operators. For example, CURRENT, DATE( ), DAY( ), EXTEND( ), LENGTH( ), MDY( ), MONTH( ), WEEKDAY( ), YEAR( ), and the relational operators are features of both 4GL and SQL. (For more information, see* *"Relational Operators" on page 5-35**.) You will generally encounter a compile-time or link-time error, however, if a statement that is not an SQL statement references an SQL function or operator that is not also a 4GL function or operator.*

Built-in SQL functions and operators like USER cannot appear in other 4GL statements, however, that are not SQL statements. If a program requires the functionality of USER in a non-SQL statement like PROMPT, for example, you must first use FUNCTION to define an equivalent 4GL function:

```
FUNCTION get_user()
   DEFINE uid LIKE informix.sysusers.username
   SELECT USER INTO uid FROM informix.systables
      WHERE tabname = "systables"
      -- row is sure to exist and to be singular
   RETURN uid
END FUNCTION
```

To require no cursor, the SELECT statement in this example must be written so that it returns only one row. Here the **get_user( )** function selects the row of **systables** that names itself because this row it is sure to exist and to be unique. (The owner name **informix** is required to reference tables of the system catalog only in an ANSI-compliant database, but it is valid in any SQL database where it is an owner name.)

Operators of 4GL

The operators of INFORMIX-4GL differ in several ways from 4GL functions:

- Except for GET_FLDBUF( ), the CALL statement cannot invoke operators.
- Some operators can take special non-alphanumeric symbols as operands.
- You cannot reference operators from a C program.

Despite these differences, 4GL operators are described in this chapter because they resemble the built-in 4GL functions in their syntax and behavior. Operators that return a single value can be operands in expressions.

The FUNCTION statement can define (and CALL can invoke) a 4GL function that has the same name as an operator. In this case, only the operator, not the function, is visible as an operand within a 4GL expression. For example:

```
let dt = mdy(1,2,3)   --built-in MDY() operator
call mdy(1,2,3)       --programmer-defined MDY() function
```

The following keyword-based operators of 4GL are described in this chapter.

AND	LINENO
ASCII *int-expr*	MATCHES *expr*
BETWEEN *expr* AND *expr*	*int-expr* MOD *int-expr*
char-expr CLIPPED	NOT
COLUMN *int-expr*	OR
CURRENT *qualifier*	PAGENO
DATE (*date-expression*)	*int-expr* SPACE
DAY(*date-expression*)	*int-expr* SPACES
EXTEND(*value*, *qualifier*)	TIME
FIELD_TOUCHED(*field-list*)	TODAY
GET_FLDBUF(*field-list*)	*int-expr* UNITS *time-keyword*
INFIELD (*field*)	*expression* USING *format-string*
IS NULL	WEEKDAY (*date-expression*)
LENGTH(*char-expression*)	*char-expr* WORDWRAP
LIKE	YEAR (*date-expression*)

These operators (and additional arithmetic and relational operators) are included in this chapter as a convenience, so that you can find syntax articles without needing to classify a given feature as a function or as an operator.

Operators that are represented by non-alphabetic symbols are grouped in this chapter under the following headings:

- "Arithmetic Operators" on page 5-20
- "Relational Operators" on page 5-35
- "Concatenation (| |) Operator" on page 5-50
- "Membership (.) Operator" on page 5-97
- "Substring ([ ]) Operator" on page 5-113

The following table lists the arithmetic and relational operators of 4GL.

Arithmetic Operators			Relational Operators		
Symbol	**Description**	**Page**	**Symbol**	**Description**	**Page**
+	Addition	5-26	<	Less than	5-34
/	Division	5-25	<=	Less than or equal to	5-35
**	Exponentiation	5-25	= *or* ==	Equal to	5-34
MOD	Modulus	5-25	!= *or* <>	Not equal to	5-34
*	Multiplication	5-25	>=	Greater than or equal to	5-35
-	Subtraction	5-26	>	Greater than	5-34
-	Unary negative	5-22			
+	Unary positive	5-22			

For a general discussion of 4GL operators, see "Operators in 4GL Expressions" on page 3-53.

Built-In Functions of Informix Dynamic 4GL

Dynamic 4GL is a separately sold product that supports the migration of character-based 4GL applications to UNIX and Windows platforms that support a graphical user interface. Dynamic 4GL extends the INFORMIX-4GL language with many additional built-in functions besides those that are described in this chapter. See the *Informix Dynamic 4GL User Guide* for the syntax and semantics of these additional built-in functions of Dynamic 4GL.

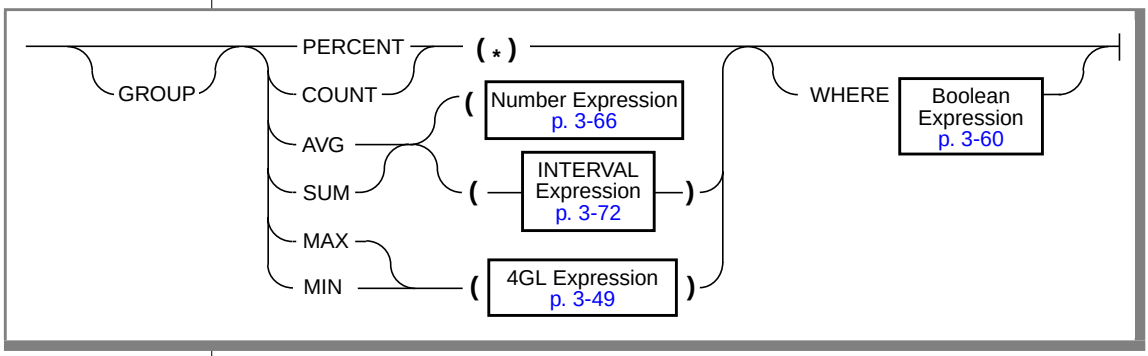
Syntax of Built-In Functions and Operators

Sections that follow describe these built-in functions and operators of 4GL.

† Valid only in the FORMAT section of a REPORT program block.
‡ Valid only in the COLOR attribute of a form specification, and in SQL statements.

Aggregate Report Functions

Each *aggregate report function* of 4GL returns a value summarizing data from all the input records or from a specified group of input records. The 4GL report aggregates are not valid outside of a REPORT program block.



Usage

Aggregate report functions return statistical aggregates (or for MAX and MIN, extrema) based on input records of a 4GL report. The GROUP keyword restricts the data set to the current AFTER GROUP OF control block; the WHERE keyword can apply a Boolean filter. Otherwise, returned values are based on the entire data set.

The 4GL report aggregates resemble the SQL aggregates that can appear in SELECT or DELETE statements, but their syntax is not identical. For details, see "Differences Between the 4GL and SQL Aggregates" on page 5-17. Aggregate report functions cannot appear as operands or as arguments in 4GL expressions, and cannot be nested. That is, no expression within a report aggregate can include a report aggregate.

An error typically occurs if you attempt to use the name of an aggregate as an identifier. Programmer-defined functions to calculate the same statistics can be invoked from within or outside of REPORT definitions, but you must declare other names for such functions.

Variables of large or structured data types cannot be arguments to these functions. You can, however, specify the name of a simple variable that is a member of a record, or that is an element of an array. For more information, see "Variables of Large Data Types" on page 4-86 and "Simple Data Types" on page 3-9.

AVG( ), SUM( ), MIN( ), and MAX( ) ignore records with null values for their argument, but each returns NULL if all records have a null value.

If an aggregate value that depends on all records of the report appears anywhere except in the ON LAST ROW control block, each variable in that aggregate or WHERE clause must also appear in the list of formal arguments of the report. (Examples of aggregates that depend on all records include using GROUP COUNT(*) anywhere in a report, or using any aggregate without the GROUP keyword anywhere outside the ON LAST ROW control block.)

4GL stores intermediate results for aggregates in temporary tables. An error results if no database is open (because the temporary table cannot be created), or if you change or close the current database while evaluating an aggregate.

The GROUP Keyword

This optional keyword causes the aggregate function to include data only for a group of records that have the same value on a variable that you specify in an AFTER GROUP OF control block.

An aggregate can include the GROUP keyword only within an AFTER GROUP OF control block. If you need the value of a GROUP report aggregate elsewhere, you must use the LET statement within the AFTER GROUP OF control block to store the value in a variable that has the appropriate scope of reference.

The WHERE Clause

The optional WHERE keyword selects among the records passed to the report, including only those for which a Boolean expression is TRUE. Conditional aggregates are calculated on the first pass, when the records are read, and printed on the second pass. You cannot use aggregates in a loop, such as FOR or WHILE, where the Boolean expression changes dynamically.

The MIN( ) and MAX( ) Functions

These return the minimum value and maximum value (respectively) of their argument among all records, or among records qualified by the WHERE clause or by the GROUP keyword. For character data, *greater than* means *after* in the ASCII collation sequence, where a> A> 1, and *less than* means *before* in the ASCII sequence, where 1< A< a. For DATE or DATETIME data, *greater than* means *later* and *less than* means *earlier* in time. See also Appendix A, "The ASCII Character Set," for a listing of the ASCII collation sequence.

GLS

In nondefault locales, 4GL sorts character operands of MIN( ) or MAX( ) in code-set order, unless the locale files define a localized collation sequence in its COLLATION category, and **DBNLS** is set to 1. This order also applies to character variables whose values were retrieved from the database.

Unlike the database, 4GL makes no distinction between character strings whose values were retrieved from CHAR or VARCHAR columns and values from NCHAR or NVARCHAR columns of the database. To sort strings by the rules of the database, rather than by these 4GL rules, write your code so that the database performs the sorting. ♦

The AVG( ) and SUM( ) Functions

These functions evaluate as the *average* (that is, the arithmetic mean value) and the total (respectively) of the expression among all records, or among records qualified by the optional WHERE clause or GROUP keyword. The operand of AVG( ) or SUM( ) must be a 4GL expression of a number or INTERVAL data type.

The COUNT (*) and PERCENT (*) Functions

These functions are evaluated, respectively, as the total number of records qualified by the optional WHERE clause, and as a *percentage* of the total number of records in the report. You must include the (*) symbols. Like the other report aggregates, PERCENT(*) and COUNT(*) cannot be used within an expression.

The following fragment of a REPORT routine uses the AFTER GROUP OF control block and GROUP keyword to form sets of records according to how many items are in each order. The last PRINT statement calculates the total price of each order, adds a shipping charge, and prints the result.

```
AFTER GROUP OF number
 SKIP 1 LINE
 PRINT 4 SPACES, "Shipping charges for the order:  ",
  ship_charge USING "$$$$.&&"
 PRINT 4 SPACES,  "Count of small orders:  ",
  COUNT(*) WHERE total_price < 200.00 USING "##,###"
 SKIP 1 LINE
 PRINT 5 SPACES, "Total amount for the order:  ",
  ship_charge + GROUP SUM(total_price) USING "$$,$$$,$$$.&&"
```

With no WHERE clause, GROUP SUM here combines every item in the group.

Differences Between the 4GL and SQL Aggregates

The *Informix Guide to SQL: Syntax* describes the syntax of the SQL aggregate functions. The major differences between the 4GL report aggregates and aggregate functions that Informix database servers support follow:

- Only 4GL report aggregates can use the PERCENT(*) and GROUP keywords.
- Only SQL aggregates can use ALL, DISTINCT, and UNIQUE as keywords.
- In 4GL reports, COUNT can only take an asterisk (*) as its argument, but in SELECT or DELETE statements of SQL, the COUNT aggregate can also use a column name or an SQL expression as its argument.

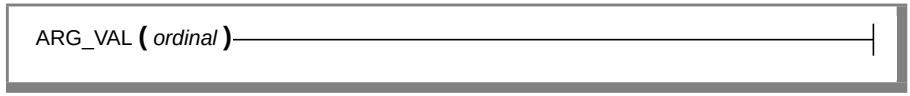
Only SQL aggregate functions can use database column names as arguments, but this syntax difference is not of much practical importance. (Operands in the expressions that you specify as arguments for 4GL report aggregates can be program variables that contain values from database columns.)

References

LINENO, PAGENO, WORDWRAP

ARG_VAL( )

The ARG_VAL( ) function returns a specified argument from the command line that invoked the current 4GL application program. It can also return the name of the current 4GL program.



Element	Description
ordinal	is an integer expression that evaluates to a non-negative whole number no larger than the number of arguments of the program. (See the syntax of "Integer Expressions" on page 3-63.)

Usage

This function provides a mechanism for passing values to the 4GL program through the command line that invokes the program. You can design a 4GL program to expect or allow arguments after the name of the program in the command line.

Like all 4GL functions, ARG_VAL( ) can be invoked from any program block. You can use it to pass values to MAIN, which cannot have formal arguments, but you are not restricted to calling ARG_VAL( ) from the MAIN statement.

You can use the ARG_VAL( ) function to retrieve individual arguments during program execution. (You can also use the NUM_ARGS( ) function to determine how many arguments follow the program name on the command line.)

If $1 \leq ordinal = n$, ARG_VAL(*n*) returns the n^{th} command-line argument (as a character string). The value of *ordinal* must be between 0 and the value returned by NUM_ARGS ( ), the number of command-line arguments.

The expression ARG_VAL(0) returns the name of the 4GL application program.

Using ARG_VAL( ) with NUM_ARGS( )

The built-in ARG_VAL( ) and NUM_ARGS( ) functions can pass data to a compiled 4GL program from the command line that invoked the program.

For example, suppose that the 4GL program called **myprog** can accept one or more login names as command-line arguments. Both of the following command lines can include the same four arguments:

`myprog.4gi joe bob sue les` (C compiler version)

`fglgo myprog joe bob sue les` (RDS version)

In either case, statements in the following program fragment use the ARG_VAL( ) function to store in an array of CHAR variables all the names that the user who invoked **myprog** entered as command-line arguments:

```
DEFINE args ARRAY[8] OF CHAR(10),
       i    SMALLINT
. . .
FOR i = 1 TO NUM_ARGS()
    LET args[i] = ARG_VAL(i)
END FOR
```

After **myprog** is invoked by these command-line arguments, the NUM_ARGS( ) function returns the value 4. Executing the LET statements in the FOR loop assigns the following values to elements of the **args** array.

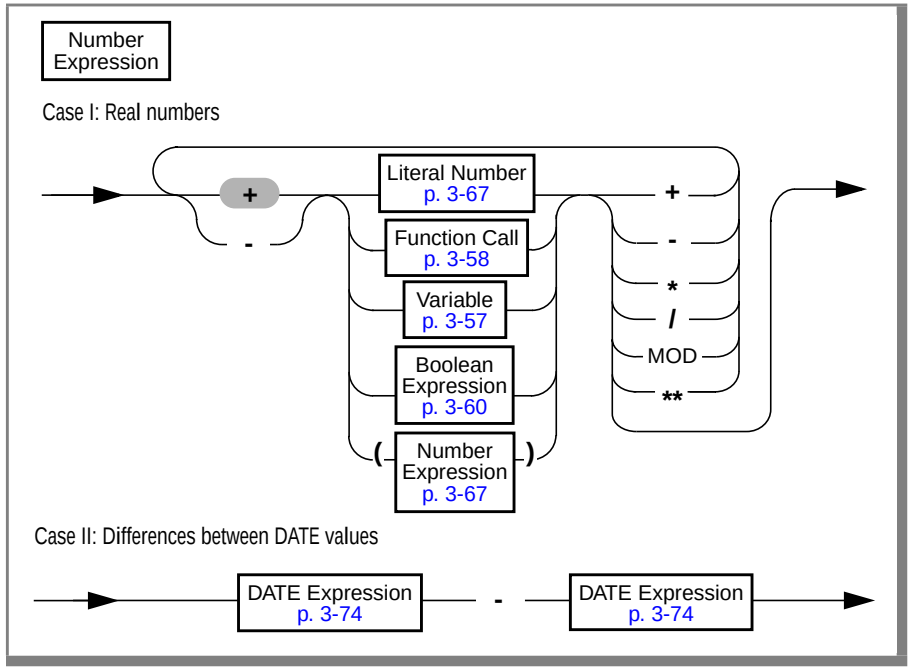
Variable	Value
args[1]	joe
args[2]	bob
args[3]	sue
args[4]	les

Reference

NUM_ARGS( )

Arithmetic Operators

The 4GL *arithmetic operators* perform arithmetic operations on operands of number data types (and in some cases, on operands of time data types). Arithmetic expressions that return a number value have this syntax.

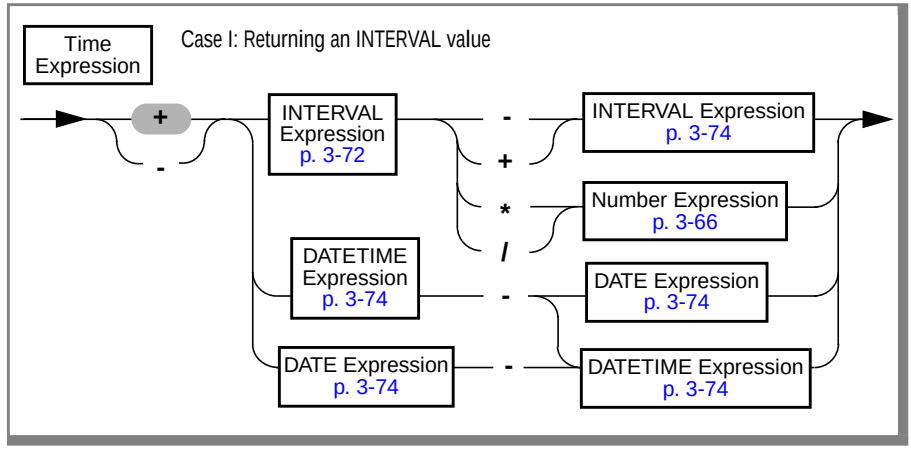


In the next three syntax diagrams, the DATETIME and INTERVAL expression segments are only a subset of time expressions, as described in "Time Expressions" on page 3-72. A DATETIME expression or an INTERVAL expression used as an arithmetic operand can be any of the following items:

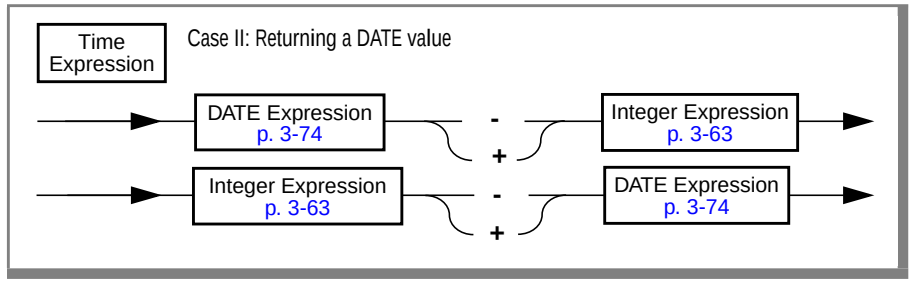
- A program variable of the DATETIME or INTERVAL data type.
- A DATETIME or INTERVAL value that a function or operator returns.
- A DATETIME literal (page 3-78) or an INTERVAL literal (page 3-82).

A DATETIME or INTERVAL expression cannot be a quoted string, or numeric DATETIME value (page 3-78), or numeric INTERVAL value (page 3-82), that omits the DATETIME or INTERVAL keyword and the DATETIME qualifier or INTERVAL qualifier.

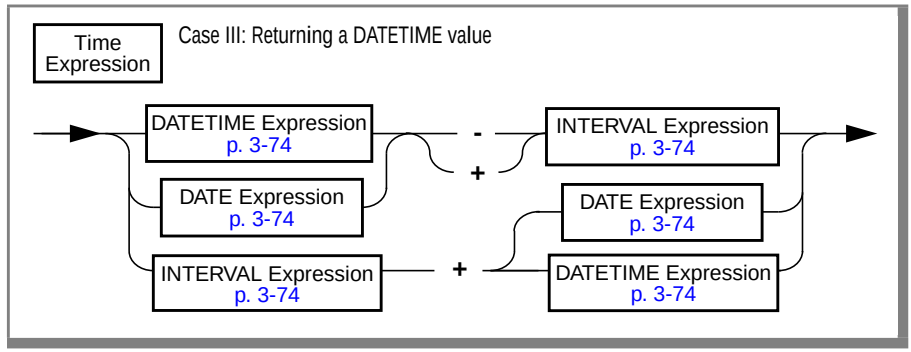
This is the syntax for arithmetic expressions that return an INTERVAL value.



This is the syntax for arithmetic expressions that return a DATE value.



This is the syntax for arithmetic expressions that return a DATETIME value.



Usage

The arithmetic operators of 4GL resemble the arithmetic functions of Informix database servers, but they are evaluated by the client system.

Arithmetic operands can only be of simple data types. Structured (ARRAY or RECORD) or large (BYTE or TEXT) data types are not valid as operands. An operand, however, can be a simple variable that is a member of a record or that is an element of an array.

The range of returned values is that of the returned data type.

In most contexts, CHAR or VARCHAR character-string representations of number values are valid as arithmetic operands, but this imposes additional overhead for data type conversion. The direct use of number data types tends to support better performance in 4GL applications that make intensive use of arithmetic operators.

If a Boolean expression is an arithmetic operand, 4GL evaluates it and converts it to an integer by these rules: TRUE = `1` and FALSE = `0`.

If any component of an expression that includes an arithmetic operator is a null value, the entire expression returns NULL, unless the null component is an operand of the IS NULL or IS NOT NULL operators.

Unary Arithmetic Operators

At the left of expressions that return a number or INTERVAL value, plus (+) and minus (-) symbols can appear as *unary operators* to specify the sign. For unsigned values, the default is positive (+). The number data types of 4GL include DECIMAL, FLOAT, INTEGER, MONEY, SMALLFLOAT, and SMALLINT.

Unary plus (+) and minus (-) operators are recursive. Parentheses must separate the subtraction operator from any immediately following unary minus sign, as in "`minuend -(-subtrahend)`" unless you want 4GL to interpret the -- symbols as a comment indicator.

Binary Arithmetic Operators

Six *binary arithmetic operators* can appear in number expressions. As the syntax diagrams at the beginning of this article indicate, four of these operators (*, /, +, and -) also can appear in time (DATE, DATETIME, and INTERVAL) expressions. The MOD and exponentiation (**) operators accept some DATE values as operands, but such expressions might return values that are difficult to interpret.

Symbol	Operator Name	Name of Result	Precedence
**	Exponentiation	Power	13
MOD	Modulus	Remainder	13
*	Multiplication	Product	12
/	Division	Quotient	12
+	Addition	Sum	11
-	Subtraction	Difference	11

4GL performs calculations with binary arithmetic operators of number data types after automatically converting both operands to DECIMAL values.

Time operands in arithmetic expressions cannot be quoted strings representing numeric date and time values (page 3-78) nor numeric time intervals (page 3-82). Use instead DATETIME literals or INTERVAL literals that include an appropriate DATETIME qualifier or INTERVAL qualifier.

For example, the following LET statement that attempts to use an arithmetic time expression as its right-hand term in fact assigns a null value to the INTERVAL variable **totalsec**, rather than an interval of 9 years.

```
LET totalsec = "2002-01-01 00:00:00.000" - "1993-01-01 00:00:00.000"
```

The desired non-null result requires:

```
LET totalsec = DATETIME (2002-01-01 00:00:00.000) YEAR TO FRACTION
             - DATETIME (1993-01-01 00:00:00.000) YEAR TO FRACTION
```

If the first operand of an arithmetic expression includes the UNITS operator ("UNITS" on page 5-119), you must enclose that operand in parentheses.

The following table shows the precedence (**P**) and data types of operands and of returned values for both unary and binary arithmetic operators. Time operands not listed here produce either errors or meaningless results.

P	Expression	Left (= x)	Right (= y)	Returned Value
13	+ y		Number or INTERVAL	Same as y
	- y		Number or INTERVAL	Same as y
12	x * * y	Number	INT or SMALLINT	Same as y
	x MOD y	INT or SMALLINT	INT or SMALLINT	Same as y
11	x * y	Number or INTERVAL	Number	Same as y
	x / y	Number or INTERVAL	Number	Same as x
10	x + y	Number	Number	Number
	x + y	INT or SMALLINT	DATE	DATE
	x + y	DATE	INT or SMALLINT	DATE
	x + y	DATE or DATETIME	INTERVAL	DATETIME
	x + y	INTERVAL	DATE or DATETIME	DATETIME
	x - y	INTERVAL	INTERVAL	INTERVAL
	x - y	Number	Number	Number
	x - y	INT or SMALLINT	DATE	DATE
	x - y	DATE	INT or SMALLINT	DATE
	x - y	DATE or DATETIME	INTERVAL	DATETIME
	x - y	DATE or DATETIME	DATETIME	INTERVAL
	x - y	DATETIME	DATE	INTERVAL
		INTERVAL	INTERVAL	INTERVAL
		DATE	DATE	INT

The precedence of all 4GL operators is listed in "Operators in 4GL Expressions" on page 3-53; the operators that are not listed on that page have a precedence of 1. These precedence (**P**) values are ordinal numbers to show relative ranks.

Important: *DATE and DATETIME values have no true zero point. Such values can support addition, subtraction, and the relational operators, but division, multiplication, and exponentiation are logically undefined for these data types.*

Exponentiation (**) Operator

The exponentiation (**) operator returns a value calculated by raising the left-hand operand to a power corresponding to the integer part of the right-hand operand. This right-hand operand cannot have a negative value.

An expression specifying the right-hand MOD operand cannot include the exponentiation operator. Before conversion to DECIMAL for evaluation, 4GL converts the right-hand operand of the exponentiation operator to an INTEGER value. Any fractional part is discarded.

Modulus (MOD) Operator

The modulus (MOD) operator returns the remainder from integer division when the integer part of the left-hand operator is divided by the integer part of the right-hand operator. For example, if `y = 7.76` and `z = 2.95`, the following expression assigns to **x** the value `1`, the integer part of the remainder of `7` divided by `2`. The syntax is:

```
LET x = y MOD z
```

In 4GL programs, MOD is a reserved word. Do not use it as a 4GL identifier.

An expression specifying the right-hand MOD operand cannot include the exponentiation or modulus operator, and it cannot be zero. Before conversion to DECIMAL for evaluation, 4GL converts any operand of MOD that is not of the INTEGER or SMALLINT data type to an INTEGER value by truncation. Any fractional part is discarded.

Multiplication (*) and Division (/) Operators

The multiplication (*) operator returns the scalar product of its left-hand and right-hand operands. The division (/) operator returns the quotient of its left-hand operand divided by its right-hand operand. An error is returned if the right-hand operand (the divisor) evaluates to zero.

If both operands of the division operator are of the INT or SMALLINT data type, 4GL discards any fractional portion of the quotient.

For multiplication and division, if the left-hand operand has an INTERVAL value, the result is an INTERVAL value of the same precision. (The right-hand operand must be an expression that returns a number data type.)

When the results of division are assigned to a fixed-point DECIMAL variable, results are rounded, rather than truncated, if the fractional part contains more decimal places than the declared scale of the receiving DECIMAL data type.

Addition (+) and Subtraction (-) Operators

The addition (+) and subtraction (-) operators return the algebraic sum and difference, respectively, between their left- and right-hand operands.

You can use DATE operands in addition and subtraction, but not the sum of two DATE values. All the other binary arithmetic operators also accept DATE operands, equivalent to the count of days since December 31, 1899; but the values returned (except from a DATE expression as the left-hand MOD operand) are meaningless in most applications.

Do not write expressions that specify the sum of two DATE or DATETIME values, or a difference whose second operand is a DATE or DATETIME value, and whose first operand is an INTERVAL value.

The difference between two DATETIME values (or a DATETIME and a DATE value, but *not* two DATE values) is an INTERVAL value. If the operands have different qualifiers, the result has the qualifier of the first operand.

The difference between two DATE values is an INTEGER value, representing the positive or negative number of *days* between the two calendar dates. You must explicitly apply the UNITS DAY operator to a difference between DATE values to store the result as an INTERVAL DAY TO DAY value.

An arithmetic expression cannot combine an INTERVAL value of precision in the range YEAR to MONTH with another in the DAY to FRACTION range. Neither can an expression combine an INTERVAL value with a DATETIME or DATE value that has different qualifiers. You must explicitly use the EXTEND operator (as described in "EXTEND( )" on page 5-67) to change the DATE or DATETIME precision to match that of the INTERVAL operand.

Arithmetic with DATE or DATETIME values sometimes produces errors. For example, adding or subtracting a UNITS MONTH operand to a date near the end of a month can return an invalid date (such as September 31).

References

Aggregate Report Functions, Boolean Operators, EXTEND, UNITS

ARR_COUNT( )

The ARR_COUNT( ) function returns a positive whole number, typically representing the number of records entered in a program array during or after execution of the INPUT ARRAY statement.

Usage

You can use ARR_COUNT( ) to determine the number of program records that are currently stored in a program array. In typical 4GL applications, these records correspond to values from the active set of retrieved database rows from the most recent query. By first calling the SET_COUNT( ) function, you can set an upper limit on the value that ARR_COUNT( ) returns.

ARR_COUNT( ) returns a positive integer, corresponding to the index of the furthest record within the program array that the screen cursor accessed. Not all the rows *counted* by ARR_COUNT( ) necessarily contain data (for example, if the user presses the DOWN ARROW key more times than there are rows of data). If SET_COUNT( ) was explicitly called, ARR_COUNT( ) returns the greater of these two values: the argument of SET_COUNT( ) or the highest value attained by the array index.

The **insert_items( )** function in the following example uses the value returned by ARR_COUNT( ) to set the upper limit in a FOR statement:

```
FUNCTION insert_items()
   DEFINE counter SMALLINT
   FOR counter = 1 TO ARR_COUNT()
      INSERT INTO items
         VALUES (p_items[counter].item_num,
            p_orders.order_num,
            p_items[counter].stock_num,
            p_items[counter].manu_code,
            p_items[counter].quantity,
            p_items[counter].total_price)
   END FOR
END FUNCTION
```

The following example makes use of ARR_COUNT( ) and the related built-in functions ARR_ CURR( ) and SCR_LINE( ) to assign values to variables within the BEFORE ROW clause of an INPUT ARRAY WITHOUT DEFAULTS statement. By calling these functions in BEFORE ROW, the respective variables are evaluated each time the cursor moves to a new line and are available within other clauses of the INPUT ARRAY statement.

```
INPUT ARRAY ga_manuf WITHOUT DEFAULTS FROM sa_manuf.*
   BEFORE ROW
      LET curr_pa = ARR_CURR()
      LET curr_sa SCR_LINE()
      LET total_pa = ARR_COUNT()
```

You can have a statement, in a later statement within INPUT ARRAY, such as the following example, which tests whether the cursor is at the last position in the screen array:

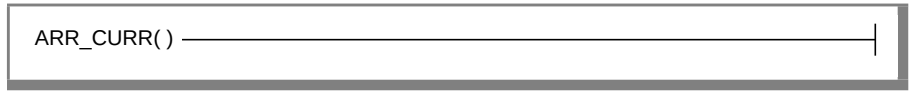
```
IF curr_pa <> total_pa THEN       ...
```

References

ARR_CURR( ), FGL_SETCURRLINE( ), SCR_LINE( ), SET_COUNT( )

ARR_CURR( )

During or immediately after the INPUT ARRAY or DISPLAY ARRAY statement the ARR_CURR( ) function returns the number of the program record within the program array that is displayed in the current line of a screen array.



Usage

The current line of a screen array is the line that displays the screen cursor at the beginning of a BEFORE ROW or AFTER ROW clause.

The ARR_CURR( ) function returns an integer value. The first row of the program array and the first line (that is, top-most) of the screen array are both numbered 1. The built-in functions ARR_CURR( ) and SCR_LINE( ) can return different values if the program array is larger than the screen array.

You can pass ARR_CURR( ) as an argument when you call a function. In this way the function receives as its argument the current record of whatever array is referenced in the INPUT ARRAY or DISPLAY ARRAY statement.

The ARR_CURR( ) function can be used to force a FOR loop to begin beyond the first line of an array by setting a variable to ARR_CURR( ) and using that variable as the starting value for the FOR loop.

The following program segment tests the user input for duplication of what should be a unique column. If the field duplicates an existing item, the program instructs the user to try again.

```
INPUT ARRAY ga_manufact FROM sa_manufact.*
    AFTER FIELD manu_code
        IF pk_check(ARR_CURR())  THEN
            ERROR "This code already exists. Re-enter",
            " or press F2 to delete this entry."
            NEXT FIELD manu_code
        END IF
END INPUT
```

In this example, the value returned by ARR_CURR( ) is then passed to function **pk_check( )**, where it is stored in the local variable **el_pa**, serving as an index to the global array **ga_manufact**:

```
FUNCTION pk_check(el_pa) --verifies primary key
  DEFINE el_pa, manu_count INT
  SELECT COUNT(*) INTO manu_count
      FROM manufact
      WHERE manufact.manu_code = ga_manufact[el_pa].manucode
  IF manu_count >= 1
      THEN RETURN TRUE
      ELSE RETURN FALSE
  END IF
END FUNCTION
```

The ARR_CURR( ) function is frequently used with a DISPLAY ARRAY statement in popup windows to return the user's selection.

The following example allows users to choose supplier codes for shoes in an order form. The user chooses among eight possibilities. The choice is returned and displayed on a form. The variables **pa_supplier** and **elem_pa** are locally defined. The variable **gr_shoes** is a global record associated with an INPUT statement.

```
OPEN WINDOW w_supplier AT 3,50
   WITH FORM "f_popscode"
   ATTRIBUTE  (BORDER, REVERSE)
DISPLAY "Press ESC to select." AT 1,1
DISPLAY "Use arrow keys to move." at 2,1
CALL SET_COUNT(8)
DISPLAY ARRAY pa_supplier TO sa_supplier.*
   LET elem_pa = ARR_CURR()
   LET gr_shoes.supply_code = pa_supplier[elem_pa].s_code
CLOSE WINDOW w_supplier
DISPLAY BY NAME gr_shoes.supply_code
```

References

ARR_COUNT( ), FGL_SETCURRLINE( ), SCR_LINE( )

ASCII

The ASCII operator converts an integer operand into its corresponding ASCII character.

Element	Description
number	is an integer expression (as described in "Integer Expressions" on page 3-63) that returns a positive whole number within the range of ASCII values.

Usage

You can use the ASCII operator to evaluate an integer to a single character. This operator is especially useful if you need to display CONTROL characters.

The following DISPLAY statement rings the terminal bell (ASCII value of 7):

```
DEFINE bell CHAR(1)
LET bell = ASCII 7
DISPLAY bell
```

The next REPORT program block fragments show how to implement special printer or terminal functions. They assume that, when the printer receives the sequence of ASCII characters 9, 11, and 1, it will start printing in red, and when it receives 9, 11, and 0, it will revert to black printing. The values used in the example are hypothetical; refer to the documentation for your printer or terminal for information about which values to use.

```
FORMAT
   FIRST PAGE HEADER
      LET red_on = ASCII 9, ASCII 11, ASCII 1
      LET red_off = ASCII 9, ASCII 11, ASCII 0
   ON EVERY ROW
     ...
      PRINT red_on,
            "Your bill is overdue.", red_off
```

Warning: *4GL cannot distinguish between printable and nonprintable ASCII characters. Be sure to account for nonprinting characters when using the COLUMN operator to format the screen or a page of report output. Because devices differ in outputting spaces with control characters, you might need to use trial and error to align columns properly when you include control characters in output.*

The ASCII Operator in PRINT Statements

To print a null character in a report, call the ASCII operator with 0 in a PRINT statement. For example, the following statement prints the null character:

```
PRINT ASCII 0
```

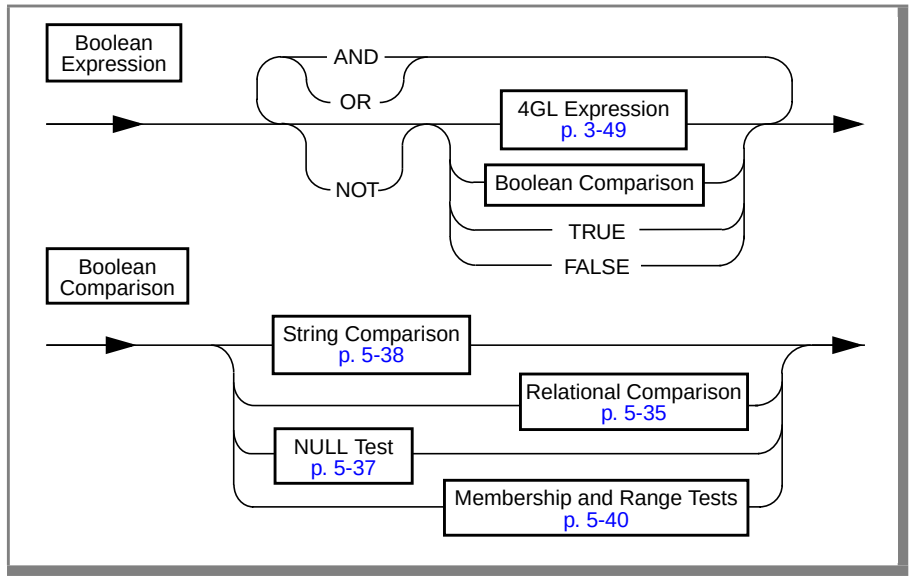
ASCII 0 only displays the null character within the PRINT statement. If you specify ASCII 0 in other contexts, it returns a blank space.

References

FGL_KEYVAL( ), FGL_LASTKEY( )

Boolean Operators

A 4GL *Boolean operator* returns TRUE (= 1), FALSE (= 0) or NULL. These 4GL operators resemble the SQL Boolean operators, but some are not identical.



The Boolean operators include the logical operators AND, OR, and NOT, and operators for Boolean comparisons, as described in the sections that follow.

Logical Operators

The *logical operators* AND, OR, and NOT combine Boolean values into a single 4GL Boolean expression. AND, OR, and NOT produce the following results (where T means TRUE, F means FALSE, and ? means NULL).

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

Values of right-hand operands appear in **boldface** below each operator; the top row also lists the possible values of left-hand operands of AND and OR.

Returned values appear in the cells where the row and column intersect. Any nonzero operand that is not null is treated as TRUE by the logical operators. For more information, see "Evaluating Boolean Expressions" on page 5-42.

When one or both arguments of a logical operator are null, the result can in some cases also be null. For example, if **var1** = 0 and **var2** = NULL, the following expression assigns a null value to variable x:

```
LET x = var1 OR var2
```

4GL attempts to evaluate both operands of AND and OR logical operators, even if the value of the first operand has already determined the returned value. The NOT operator is recursive.

Boolean Comparisons

Boolean comparisons can test any type of expression for equality or inequality, null values, or set membership, using the following Boolean operators:

- Relational operators to test for equality or inequality
- IS NULL (and IS NOT NULL) to test for null values
- LIKE or MATCHES to compare character strings
- IN to test for set membership
- BETWEEN…AND to test for range

The IN and BETWEEN…AND operators are valid only in SQL statements. They cause a compilation error if you include them in other 4GL statements. They are listed here, however, because they are valid in the WHERE clause of a form specification file that includes the COLOR attribute. For more information, see "Boolean Expressions in 4GL Form Specification Files" on page 6-38.

Boolean expressions in the CASE, IF, or WHILE statements (or in the WHERE clause of a COLOR attribute specification) return FALSE if any element of the comparison is null, unless it is the operand of the IS NULL or IS NOT NULL operator. Use a NULL test to detect and exclude null values in Boolean comparisons. In this CASE statement fragment, the value of the comparison is null if the value of **salary** or of **last_raise** is null:

```
WHEN salary * last_raise < 25000
```

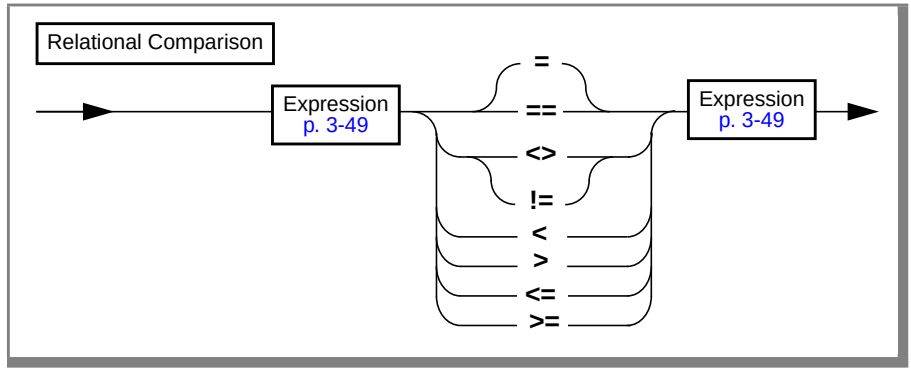
You can use any value that is not false (= 0) or null as a Boolean expression that returns TRUE. The constant TRUE, however, has the specific value of 1.

Thus, the value FALSE is returned by a comparison like:

```
(10 = TRUE)
```

Relational Operators

These operators perform relational comparisons in Boolean expressions.



Boolean expressions in 4GL statements can use these relational operators (=, ==, <, >, <=, >=, <>, or !=, as defined in "Evaluating Boolean Expressions" on page 5-42) to compare operands. For example, each of the following comparisons returns TRUE or FALSE.

Expression	Value
(2+5)* 3 = 18	FALSE
14 <= 16	TRUE
"James" = "Jones"	FALSE

For character expressions, the result depends on the position of the initial character of each operand within the collation sequence. The collation sequence is the code-set order unless the locale files define a localized collation sequence in the COLLATION category, and **DBNLS** is set to 1. If the initial characters are identical in both strings, 4GL compares successive characters, until a non-identical character is found, or until the end of a string is encountered.

For number expressions, the result of a relational comparison reflects the relative positions of the calculated values of the two operands on the real line. Relational comparisons of time expression operands follow these rules:

- Comparison x < y is true when x is a *briefer* interval span than y, or when x is an *earlier* DATE or DATETIME value than y.
- Comparison x > y is true when x is a *longer* interval span than y, or when x is a *later* DATE or DATETIME value than y.
- You cannot compare an interval with a DATE or DATETIME value, but you can compare DATE and DATETIME values with each other.

The value of the built-in constant TRUE is 1. A Boolean expression such as the following example returns FALSE unless **x** is exactly equal to 1:

```
IF (x = TRUE) THEN ...
```

To determine whether some value is not zero or null, avoid using TRUE in Boolean comparisons, and instead use expressions like:

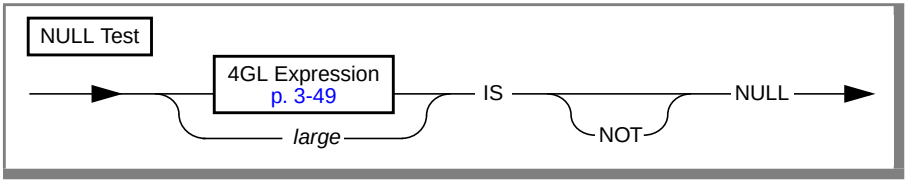
```
IF (x) THEN ...
IF (x != FALSE) THEN ...
```

Your code might be easier to read and might produce better results if you avoid using TRUE as an operand of the ==, =, !=, or <> operators.

The NULL Test

If any operand of a 4GL Boolean comparison is NULL, the value of the comparison is FALSE (rather than NULL), unless the IS NULL keywords are also included in the expression. Applying the NOT operator to a null value does not change its FALSE evaluation.

To process expressions with NULL values in a different way from other values, you can use the IS NULL keywords to test for a NULL value.



Element	Description
large	is the name of a program variable of the BYTE or TEXT data type.

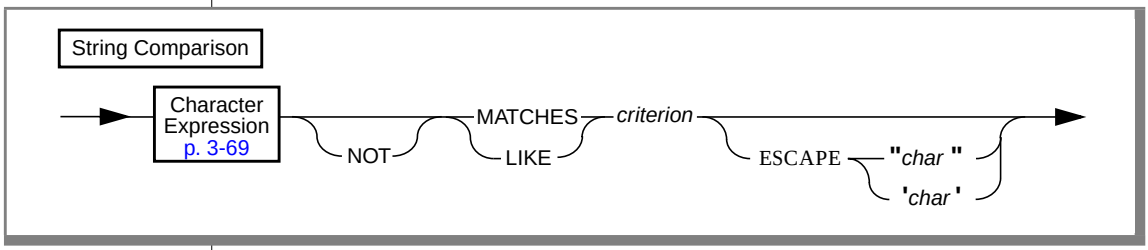
Without the NOT keyword, the comparison returns TRUE if the operand has a null value. (If you include the NOT keyword, the comparison returns TRUE if the value of the operand is not null.) Otherwise, it returns FALSE.

The NULL test (like the WORDWRAP string operator with TEXT variables) is an exception to the general rule that variables of the BYTE or TEXT data type cannot appear in 4GL expressions.

The LIKE and MATCHES Operators

The LIKE and MATCHES operators test whether a character value matches a quoted string that can include wildcard characters. If an operand has a null value, the entire string comparison returns NULL. Use a NULL test (as in the previous section) to detect and exclude null values.

You can use the following syntax to compare character strings.



Element	Description
char	is a single character, enclosed between a pair of single (') or double (") quotation marks, to specify an escape symbol.
criterion	is a character expression. The string that it returns can include literal characters, wildcards, and other symbols.

MATCHES and LIKE support different wildcards. If you use MATCHES, you can include the following wildcard characters in the right-hand operand.

Symbol	Effect in MATCHES Expression
*	An asterisk matches any string of zero or more characters.
?	A question mark matches any single character.
[ ]	Square brackets match any of the enclosed characters.
-	A hyphen between characters in brackets means a range in the collation sequence. For example, `[a-z]` matches any lowercase letter.
^	An initial caret in the brackets matches any character that is not listed. For example, `[^abc]` matches any character except a, b, or c.
\	A backslash causes 4GL to treat the next character as a literal character, even if it is one of the special symbols in this list. For example, you can match * or ? by * or \? in the string.

It is not valid to use backslash (nor any non-default escape character) within the brackets that specify a range for the MATCHES operator..

GLS

If an SQL statement uses the bracket notation of MATCHES to test for a range of values, and if the locale specifies a localized collation order, the database server sorts according to that localized order, rather than in code-set order.

This sort is true even for data values from CHAR and VARCHAR columns. However, **DBNLS** must be set to 1 for 4GL to be able to manipulate data values from NCHAR and NVARCHAR database columns (by using CHAR and VARCHAR program variables), and for nondefault collation to be supported. ♦

The following WHERE clause tests the contents of character field field007 for the string ten. Here the * wildcards specify that the comparison is true if ten is found alone or in a longer string, such as often or tennis shoe:

```
COLOR = RED WHERE field007 MATCHES "*ten*"
```

If you use the keyword LIKE to compare strings, the wildcard symbols of MATCHES have no special significance, but you can use the following wildcard characters of LIKE within the right-hand quoted string.

Symbol	Effect in LIKE Expression
%	A percent sign matches zero or more characters.
_	An underscore matches any single character.
\	A backslash causes 4GL to treat the next character as a literal (so you can match % or _ by \% or _).

The next example tests for the string ten in the character variable **string**, either alone or in a longer string:

```
IF string LIKE "%ten%"
```

The next example tests whether a substring of a character variable (or else an element of a two-dimensional array) contains an underscore symbol. The backslash is necessary because underscore is a wildcard symbol with LIKE.

```
IF horray[3,8] LIKE "%\_%" WHERE >> out.a
```

You can replace the backslash as the literal symbol. If you include an ESCAPE *char* clause in a LIKE or MATCHES specification, 4GL interprets the next character that follows *char* as a literal in the preceding character expression, even if that character corresponds to a special symbol of the LIKE or MATCHES keyword. The double quote (") symbol cannot be *char*.

For example, if you specify ESCAPE z, the characters z_ and z? in a string stand for the literal character _ and ?, rather than wildcards. Similarly, characters z% and z* stand for the characters % and *. Finally, the characters zz in the string stand for the single character z. The following expression is true if the variable **company** does not include the underscore character:

```
NOT company LIKE "%z_%" ESCAPE "z"
```

GLS

The evaluation of logical comparisons and MATCHES, LIKE, and BETWEEN expressions containing character operands is based on the code-set order of the client locale when 4GL performs the comparison, unless a localized collation is specified. When the database server performs the comparison, sorting is based on the code-set order of the database locale, or (for data values from NVARCHAR or NCHAR columns only) on a localized collation order, if one is specified in the COLLATION category of the locale files and if the **DBNLS** environment variable is set to 1. ♦

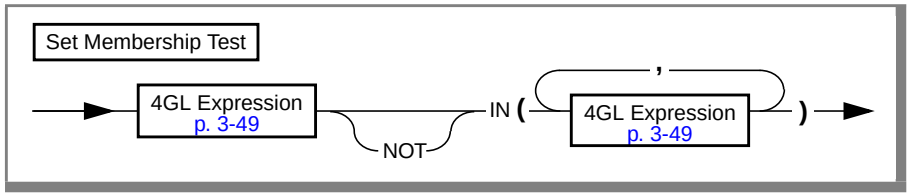
Set Membership and Range Tests

The BETWEEN...AND and IN( ) operators that test for set membership or range are supported for 4GL programs in three contexts:

- In SQL statements
- In the WHERE clause of the COLOR attribute in 4GL form specifications
- In the **condition** column of the **syscolatt** table

They are not valid in 4GL statements that are not also SQL statements.

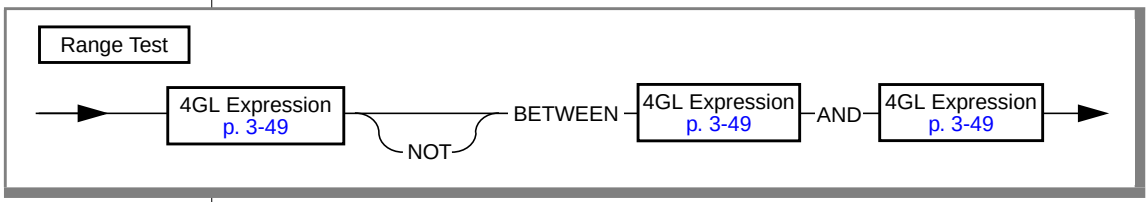
The following diagram shows the syntax for using the IN( ) operator to test for set membership.



If you omit the NOT keyword, this test returns TRUE if any expression in the list (within parentheses) at the right matches the expression on the left.

If you include the NOT keyword, the test evaluates as FALSE if no expression in the list matches the expression on the left.

The following diagram shows the syntax for the BETWEEN...AND operators to test whether a value is included within a specified range (or an inclusive interval on the real line).



Operands must return compatible data types. Values returned by the second (O2) and third (O3) operands that define the range must follow these rules:

- For number or INTERVAL values, O2 must be less than or equal to O3.
- For DATE and DATETIME values, O2 must be no later than O3.
- For character strings, O2 must be earlier than O3 in the ASCII collation sequence. (The ASCII character set is listed in Appendix A.)

GLS

In nondefault locales, the code-set order (or whatever order of sorting the COLLATION category specifies in the locale files) replaces the ASCII collation order. The **DBNLS** environment variable must also be set to 1 for 4GL to support non-code-set sorting of string values. ♦

If you omit the NOT keyword, this test evaluates as TRUE if the first operand has a value not less than the second operand or greater than the third. If you include the NOT keyword, the test evaluates as FALSE if the first operand has a value outside the specified range.

Data Type Compatibility

You might get unexpected results if you use relational operators with expressions of dissimilar data types. In general, you can compare numbers with numbers, character strings with strings, and time values with time values.

If a *time expression* operand of a Boolean expression is of the INTERVAL data type, any other time expression to which it is compared by a relational operator must also be an INTERVAL value. You cannot compare a span of time (an INTERVAL value) with a point in time (a DATE or DATETIME value). For additional information about data type compatibility in expressions, see "Summary of Compatible 4GL Data Types" on page 3-46.

Evaluating Boolean Expressions

In contexts where a Boolean expression is expected, 4GL applies the following rules after it evaluates the expression:

- The Boolean expression returns the value TRUE if the expression returns a nonzero real number *or* any of the following values:
  - A character string representing a nonzero number
  - A nonzero INTERVAL value
  - Any DATETIME or DATE value (except `31 December 1899`)
  - A true value returned by any Boolean function or operator
  - The integer constant TRUE
- The Boolean expression returns the value TRUE if the value is null *and* the expression is the operand of the IS NULL operator.

- The Boolean expression returns null if the value of the expression is null *and* the expression appears in none of the following contexts:
  - An operand of the IS NULL or IS NOT NULL operator
  - An element in a Boolean comparison (as described in "Boolean Comparisons" on page 5-34)
  - An element in a conditional statement of 4GL (IF, CASE, WHILE)
- Otherwise, the Boolean expression returns the value FALSE.

Operator Precedence in Boolean Expressions

If a Boolean expression has several operators, they are processed according to their precedence. Operators that have the same precedence are processed from left to right.

Important: *The precedence of 4GL operators is listed in "Operators in 4GL Expressions" on page 3-53. These relative precedence values are ordinal numbers.*

The following table lists the precedence (P) of the Boolean operators of 4GL and summarizes the data types of their operands.

P	Description	Expression	Left (= x)	Right (= y)	Returns
9	String comparison	x LIKE y	Character	Character	Boolean
	String comparison	x MATCHES y	Character	Character	Boolean
8	Test for: less than	x < y	Any simple data type	Same as x	Boolean
	Less than or equal to	x <= y	Any simple data type	Same as x	Boolean
	Equal to	x = y or x == y	Any simple data type	Same as x	Boolean
	Greater than or equal to	x >= y	Any simple data type	Same as x	Boolean
	Greater than	x > y	Any simple data type	Same as x	Boolean
	Not equal to	x != y or x <> y	Any simple data type	Same as x	Boolean
7	Test for: set membership	x IN (y)	Any	Any	Boolean
6	Test for: range	x BETWEEN y AND z	Any	Same as x	Boolean
5	Test for: NULL Test for: NULL	x IS NULL x IS NOT NULL	Any Any		Boolean Boolean

(1 of 2)

P	Description	Expression	Left (= x)	Right (= y)	Returns
4	Logical inverse	NOT y		Boolean	Boolean
3	Logical intersection	x AND y	Boolean	Boolean	Boolean
2	Logical union	x OR y	Boolean	Boolean	Boolean
1	Test whether field edited Test for current field	FIELD_TOUCHED(y) INFIELD(y)		Field name Field name	Boolean Boolean

(2 of 2)

Besides the Boolean operators listed in this table, the built-in 4GL operators FIELD_TOUCHED( ) and INFIELD( ) also return Boolean values. Their precedence is lower (**P** = 1) than that of the OR operator. They can use the name of a field in the current form as their operand. Both the INFIELD( ) and FIELD_TOUCHED( ) operators are described later in this chapter.

References

FIELD_TOUCHED( ), INFIELD( )

CLIPPED

The CLIPPED operator takes a character operand and returns the same character value, but without any trailing white space (such as ASCII 32).

Usage

Character expressions often have a data length less than their total size. The following DISPLAY statement, for example, would produce output that included 200 trailing blanks if CLIPPED were omitted but displays only 22 characters when CLIPPED is included:

```
DEFINE string CHAR(222)
LET string = "Two hundred characters"
DISPLAY string CLIPPED
```

The CLIPPED operator can be useful in the following kinds of situations:

- After a variable in a DISPLAY, ERROR, LET, MESSAGE, or PROMPT statement, or in a PRINT statement of a REPORT program block
- When concatenating several character expression into a single string
- When comparing two or more character expressions and one or more of them is already clipped

The CLIPPED operator can affect the value of a character variable within an expression. CLIPPED does not affect the value when it is stored in a variable (unless you are concatenating CLIPPED values together). For example, if CHAR variable **b** contains a string that is shorter than the declared length of CHAR variable **a**, the following LET statement pads **a** with trailing blanks, despite the CLIPPED operator:

```
LET a = b CLIPPED
```

However, if CHAR variable **b** contains a string value no longer than the declared maximum size of VARCHAR variable **v**, the following statement discards any trailing blanks from what it stored in **v**:

```
LET v = b
```

The following program fragment is from a REPORT that prints mailing labels:

```
FORMAT
  ON EVERY ROW
    IF (city IS NOT NULL)
       AND (state IS NOT NULL) THEN
       PRINT fname CLIPPED, 1 SPACE, lname
       PRINT company
       PRINT address1
       IF (address2 IS NOT NULL) THEN PRINT address2
    END IF
       PRINT city CLIPPED, ", " , state,
       2 SPACES, zipcode
       SKIP TO TOP OF PAGE
  END IF
```

The following program fragment is from a report driver. Here CLIPPED is used to format the text of a MESSAGE statement that includes a filename stored in a character variable:

```
DEFINE file_name CHAR(60)
PROMPT " Enter drive, pathname, ",
       "and file name for Book Report:" FOR file_name
IF (file_name IS NULL) THEN
    LET file_name = "book.out"
END IF
MESSAGE "Printing Book Report to ", file_name CLIPPED,
        " --Please wait."
```

Relative to other 4GL operators, CLIPPED has a very low precedence. This can lead to confusion in some contexts, such as specifying compound Boolean conditions. For example, **i4glc1** and **fglpc** both parse the condition:

```
IF LENGTH(f2) > 0 AND f2 CLIPPED != "customer" THEN
```

as if it were delimited with parentheses as:

```
 IF (((LENGTH(f2) > 0) AND f2) CLIPPED) != "customer" THEN
```

To achieve the required result, you can write the expression as:

```
 IF LENGTH(f2) > 0 AND (f2 CLIPPED) != "customer" THEN
```

GLS

In East Asian locales, CLIPPED can delete trailing multibyte white-space characters without creating partial characters. ♦

Reference

USING

COLUMN

COLUMN specifies the position in the current line of a report where output of the next value in a PRINT statement begins, or the position on the 4GL screen for the next value in a DISPLAY statement.

Element	Description
left-offset	is an integer expression (as described in "Integer Expressions" on page 3-63) in a report, or else a literal integer (as described in "Literal Integers" on page 3-65) in a DISPLAY statement, specifying where the next character of output will appear, in character positions from the left margin (of the screen, or page of report output).

Usage

Unless you use the keyword CLIPPED or USING, the PRINT statement and the DISPLAY statement (when no form is open) display 4GL variables with widths (including any sign) that depend on their declared data types.

Data Type	Default Display Width (in Character Positions)
CHAR	The length from the data type declaration
DATE	10
DATETIME	From 2 to 25, as implied in the data type declaration
DECIMAL	(2 + *m*), where *m* is the precision from the data type declaration
FLOAT	14
INTEGER	11
INTERVAL	From 3 to 25, as implied in the data type declaration
MONEY	(3 + *m*), where *m* is the precision from the data type declaration
SMALLFLOAT	14
SMALLINT	6
VARCHAR	The maximum length from the data type declaration

In a REPORT program block or in A DISPLAY statement that outputs data to the 4GL screen, you can use the COLUMN operator to control precisely the location of items within a line. The COLUMN operator is often a requirement for the output of tabular information, and it is convenient for many other uses.

The *left-offset* value specifies a character position offset from the left margin of the 4GL screen or the currently executing 4GL report. In a report, this value cannot be greater than the arithmetic difference (*right margin - left margin)* for explicit or default values in the OUTPUT section of the REPORT definition (for more information, see "OUTPUT Section" on page 7-12). For the syntax of 4GL expressions that return whole numbers, see "Integer Expressions" on page 3-63.

If the printing position in the current line is already beyond the specified *left-offset*, the COLUMN operator has no effect.

COLUMN in DISPLAY Statements

4GL calculates the left-offset from the first character position of the 4GL screen. For example, in the following statements both the string NAME and the 4GL variable fname are displayed with their first (left-most) character in the twelfth character position on the 4GL screen:

```
DISPLAY "NUMBER", COLUMN 12, "NAME", COLUMN 35,
  "CITY", COLUMN 57, "ZIP", COLUMN 65, "PHONE"
DISPLAY ASCII 13, customer_num, COLUMN 12, fname CLIPPED,
  ASCII 32, lname CLIPPED, COLUMN 35, city CLIPPED,
  ", ", state, COLUMN 57, zipcode, COLUMN 65, phone
```

Output from each DISPLAY statement begins on a new line.

Important: *You cannot use COLUMN to send output to a screen form. Any DISPLAY statement that includes the COLUMN operator cannot also include the AT, TO, BY NAME, or ATTRIBUTE clause. When you include the COLUMN operator in a DISPLAY statement, you must specify a literal integer as the left-offset, rather than an integer expression.*

COLUMN in PRINT Statements

When you use the PRINT statement in the FORMAT section of a report, by default, items are printed one following the other, separated by blank spaces. The COLUMN operator can override this default positioning. 4GL calculates the left-offset from the left margin. If no left margin is specified in the OUTPUT section or in START REPORT, the left-offset is counted from the left margin of the page.

If the following PRINT statements (with COLUMN and SPACE specifications) were part of a report that sent output to the 4GL screen, the output would resemble that of the DISPLAY statements in the previous example:

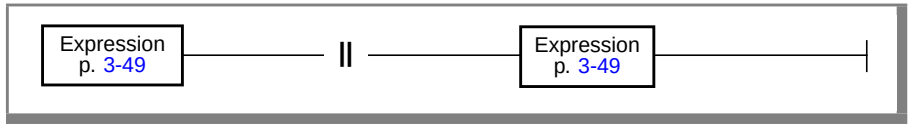
```
PAGE HEADER
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35,
     "CITY", COLUMN 57, "ZIP", COLUMN 65, "PHONE"
     SKIP 1 LINE
ON EVERY ROW
   PRINT customer_num, COLUMN 12, fname CLIPPED,
     1 SPACE, lname CLIPPED, COLUMN 35, city CLIPPED,
     ", ", state, COLUMN 57, zipcode, COLUMN 65, phone
```

References

ASCII, CLIPPED, SPACE, USING

Concatenation (||) Operator

The double bar (| |) concatenation operator joins two operands of any simple data type, returning a single string.



Usage

The concatenation (| |) operator joins two strings, with left-to-right associativity. For example, `(a || b ||c)` and `((a || b) ||c)` are equivalent expressions. The precedence of | | is greater than LIKE or MATCHES, but less than the arithmetic operators. Like arithmetic operators, | | returns a null value (as a zero-length string) if either operand has a null value.

The LET statement can use a comma (,) to concatenate strings, but with a different rule for null values: if one string operand is NULL, the result is the other string. For example, the comma separator in the right-hand expression list of the LET statement ("LET" on page 4-226) has concatenation semantics, ignoring any null values. (If all of the operands of a comma-separated list are NULL, however, LET returns a null value but represents it as a single blank space).

The following left-hand and right-hand expressions are equivalent:

```
a || b + c                    (a || (b + c))
"abcd   " || NULL           NULL
"abcd   " CLIPPED || "ZZ"   "abcdZZ"
```

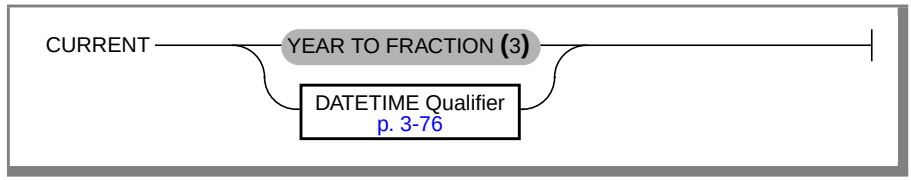
Concatenation with | | discards trailing white space from operands of integer and fixed-point number data types, but not from character or floating-point data types. The CLIPPED operator can remove trailing blanks from values before concatenation in 4GL statements, but TRIM must replace CLIPPED in preparable SQL statements (for Version 7.x and later Informix databases).

References

CLIPPED, COLUMN, Substring ([ ]) operator

CURRENT

The CURRENT operator returns the current date and time of day from the system clock as a DATETIME value of a specified or default precision.



Usage

The CURRENT operator reads the date and time from the system clock.

You can optionally specify the precision of the returned value by including a qualifier of the form *first* TO *last*, where *first* and *last* are keywords from this list.

YEAR	MONTH	DAY	
HOUR	MINUTE	SECOND	FRACTION (*n*)

The first keyword must specify a time unit that is the same as or larger than what the last keyword specifies. For example, the following qualifier is valid:

```
CURRENT YEAR TO DAY
```

But the next qualifier is not valid, because the *last* keyword specifies a time unit greater than what the *first* specifies:

```
CURRENT MINUTE TO HOUR
```

If FRACTION is the last keyword, you can include a digit *n* in parentheses to specify the scale (which can range from 1 to 5 digits) of the *seconds* value.

If no qualifier is specified, the default qualifier is YEAR TO FRACTION(3).

If CURRENT is executed more than once in a statement, identical values might be returned at each call. Similarly, the order in which the CURRENT operator is executed in a statement cannot be predicted. For this reason, do not attempt to use this operator to mark the start or end of a 4GL statement or any specific point in the execution of a statement.

You can use the CURRENT operator both in SQL statements and in other 4GL statements. The following example is from an SQL statement:

```
SELECT prog_title FROM tv_programs
      WHERE air_date > CURRENT YEAR TO DAY
```

This example is from a form specification file:

```
ATTRIBUTES-- FORM4GL field
      timestamp = FORMONLY.tmstmp TYPE DATETIME HOUR TO SECOND,
         DEFAULT = CURRENT HOUR TO SECOND;
```

The next example is from a report:

```
PAGE HEADER-- Report control block
      PRINT COLUMN 40, CURRENT MONTH TO MONTH,
            COLUMN 42, "/",
            COLUMN 43, CURRENT DAY TO DAY,
            COLUMN 45, "/",
            COLUMN 46, CURRENT YEAR TO YEAR
```

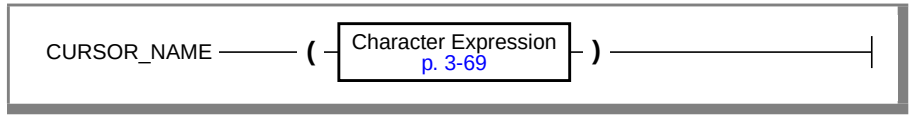
The last example would not produce the correct results if its execution spanned midnight.

References

DATE, EXTEND( ), TIME, TODAY

CURSOR_NAME( )

The CURSOR_NAME( ) function takes as its argument the SQL identifier of an UPDATE or DELETE cursor, or the identifier of a prepared statement, and returns the mangled name.



Usage

In versions of 4GL and ESQL / C earlier than 4.13 and 5.0, the scope of reference of the names of cursors (and of prepared statements) was restricted to the source module in which they were declared. Different **.4gl** source files could reference different cursors that had the same identifier without conflict. INFORMIX-ESQL / C 5.0 made the scope of cursor names global by default; the cursor **c_query** in **filea.ec** is the same as the cursor **c_query** in **fileb.ec**.

To emulate this behavior of older 4GL applications, INFORMIX-4GL p-code and C code preprocessors *mangle* (by default) all cursor identifiers and prepared statement identifiers, using the following algorithm:

```
sprintf(mangled_name, "I%08X_%08X", inode_number,
      hash_cursorname(cursor_name));
...
static unsigned long hash_cursorname(name)
char    *name;
{
   unsigned long uhash = 0x14C1BC85;
   unsigned long g;
   unsigned char *s;
   unsigned char c;
   for (s = (unsigned char *)name; (c = *s) != '\0'; s++)
   {
      uhash = (uhash << 4) + (c);
      if ((g = uhash & 0xF0000000L) != 0)
      {
         uhash = uhash ^ (g >> 24);
         uhash = uhash ^ g;
      }
   }
   return(uhash);
}
```

The mangled name is always 18 characters long. The first half is derived from the inode number of the 4GL source file, and the second half, from the user-supplied name. This strategy supports backward compatibility with release version 4.12 (and earlier) 4GL applications. In most contexts, this solution is useful with the 4GL preprocessor silently substituting the mangled name wherever the identifier of the cursor or the prepared statement appears.

These mangled names can cause an error, however, when a prepared UPDATE or DELETE statement references a mangled name as a literal string, because the preprocessor does not perform name-mangling within a quoted string. (Similarly, name-mangling is not performed within delimited SQL statement blocks, nor within the text of prepared statements.)

The CURSOR_NAME( ) function is helpful in situations where name-mangling of cursors causes failure of prepared UPDATE or DELETE statements that have the WHERE CURRENT OF clause. The problem occurs in contexts like this:

```
DECLARE c_name CURSOR FOR SELECT ... FOR UPDATE
PREPARE p_update FROM
   "UPDATE SomeTable SET SomeColumn = ? WHERE CURRENT OF c_name"
```

Here the cursor name, **c_name**, is mangled into `I01234567_87654321` (or something similar), but the name embedded in the UPDATE statement is not mangled. This code example leads to a -507 error when the **p_update** statement is executed. CURSOR_NAME( ) provides a solution to this problem by mangling the name before it is embedded in the quoted string. For example:

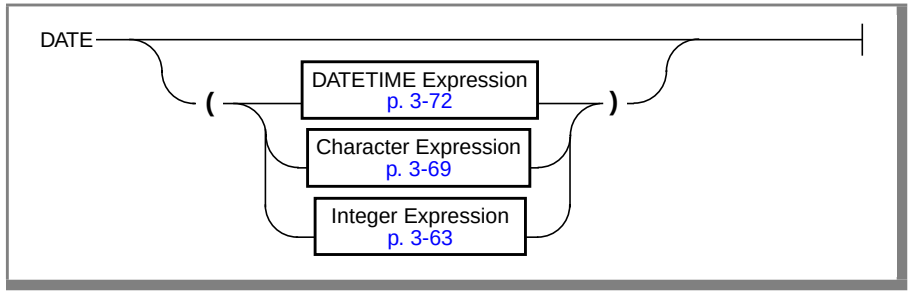
```
DECLARE c_name CURSOR FOR SELECT ... FOR UPDATE
   LET s = "UPDATE SomeTable SET SomeColumn = ? WHERE CURRENT OF ",
           CURSOR_NAME("c_name")
   PREPARE p_update FROM s
```

The name-mangling code uses, and therefore needs to know, the inode number of the source file at the time it is compiled. The fglpc and i4glc1 compilers arrange for the inode number to be pushed onto the 4GL stack before the C function is called. This is invisible to the 4GL programmer.

The **-globcurs** compilation option makes the scope of reference of cursors global and disables name-mangling. The compilers require you to declare the cursor before using it for any other purpose in the module, however, so this option is seldom useful. It might help in debugging, however, because the cursor names are not modified. You can also use **-globcurs** with **fglpc**.
Although the CURSOR_NAME( ) function is of no use to the C programmer, it must be called with two arguments on the 4GL stack; in C programs, the two arguments are the inode number and the identifier that is to be mangled.

DATE

The DATE operator converts its CHAR, VARCHAR, DATETIME, or integer operand to a DATE value. If you supply no operand, it returns a character representation of the current date.



Usage

The DATE operator can convert values of other data types to DATE values and can return the current date as a character string.

When used with no operand, the DATE operator reads the system clock-calendar and returns the current date as a string in the following format:

```
weekday month day year
```

Element	Description
weekday	is a three-character abbreviation of the name of the day of the week.
month	is a three-character abbreviation of the name of the month.
day	is a two-digit representation of the day of the month.
year	is a four-digit representation of the year.

The following example uses the DATE operator to display the current date:

```
DEFINE p_date CHAR(15)
LET p_date = DATE
. . .
DISPLAY "Today is ", p_date AT 5,14
```

On Sunday, December 5, 1999, this example would display the string:

```
Today is Sun Dec 5 1999
```

The effect of the DATE operator is sensitive to the time of execution and to the accuracy of the system clock. An alternative way to format a DATE value as a character string is to use the FORMAT field attribute in a screen form (as described in "FORMAT" on page 6-50), or to use the USING operator (as described in "Formatting DATE Values" on page 5-127).

GLS

4GL can display language-specific month-name and day-name abbreviations, if the **DBLANG** environment variable references appropriate files in the **$INFORMIXDIR/msg** file system. For information, see Appendix E, "Developing Applications with Global Language Support." ♦

The DATE operator can also perform data type conversion of a character, integer, or DATETIME operand to a DATE value, equivalent to a count of days since the beginning of the last year of the 19th century:

- Converting a properly formatted character string representation of a numeric date to a DATE value. For details, see "Numeric Date" on page 3-75. (The default format is *mm/dd/yy*, but the **DBDATE** or **GL_DATE** environment variable can change this default.)
- Converting a DATETIME value to a negative or positive integer. The returned integer corresponds to the number of days between the specified date and December 31, 1899.
- Obtaining a DATE value from a negative or positive integer that specifies the number of days between the specified date and December 31, 1899.

The following program fragment illustrates uses of the DATE operator:

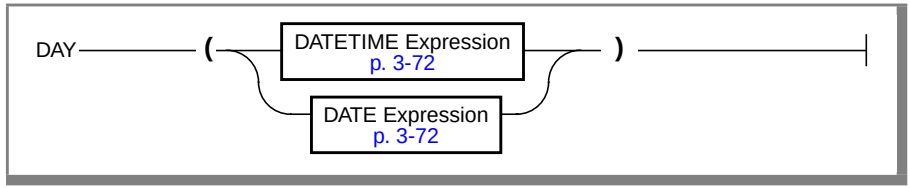
```
DEFINE d DATE
DEFINE dt DATETIME YEAR TO DAY
   LET d = DATE (" 11/20/99 ") -- this requires the default DATE format
   LET d = DATE (" 1999-11-02 ") -- this requires that DBDATE be set to Y4MD-
   LET d = DATE (" 02:99:11 ") -- this requires that DBDATE be set to DY2M:
LET d = DATE(d)  -- The operand can be a DATE variable, as here,
-- or the integer number of days since the last day of the year 1899
LET d = DATE (0)     -- result: 12/31/1899
LET d = DATE (34000) -- result: 2/1/1993
-- Or the operand can be a DATETIME type
LET dt = CURRENT
LET d = DATE (dt)     -- result is today's date
LET d = DATE(CURRENT) -- same result as previous
```

References

CURRENT, DATE, DAY, MDY( ), MONTH, TIME, UNITS, WEEKDAY, YEAR

DAY( )

The DAY( ) operator returns a positive integer, corresponding to the day portion of the value of its DATE or DATETIME operand.



Usage

The DAY( ) operator can extract an integer value for the day of the month from a DATETIME or DATE operand. This feature is helpful in some applications because INTEGER values are easier than DATETIME or DATE values to manipulate with arithmetic operators.

The following program fragment extracts the day of the month from a DATETIME literal:

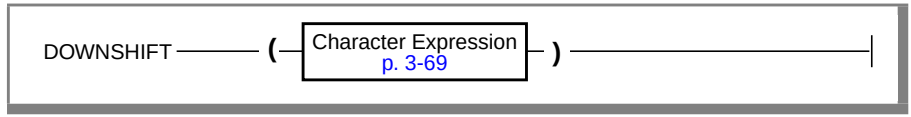
```
DEFINE d_var    INTEGER,
       date_var DATETIME YEAR TO SECOND
LET date_var =  DATETIME (89-12-09 18:47:32) YEAR TO SECOND
LET d_var = DAY(date_var)
DISPLAY "The day of the month is: ", d_var USING "##"
```

References

CURRENT, DATE, MONTH( ), TIME, TODAY, WEEKDAY( ), YEAR( )

DOWNSHIFT( )

The DOWNSHIFT( ) function returns a string value in which all uppercase characters in its argument are converted to lowercase.



Usage

The DOWNSHIFT( ) function is typically called to regularize character data. You might use it, for example, to prevent the state abbreviation entered as `TX`, `Tx`, or `tx` from resulting in different values, if these were logically equivalent in the context of your application.

Non-alphabetic or lowercase characters are not altered by DOWNSHIFT( ). The maximum data length of the argument (and of the returned character string value) is 32,766 bytes.

You can use the DOWNSHIFT( ) function in an expression (where such usage is allowed), or you can assign the value returned by the function to a variable.

In the following example, suppose that the CHAR value `GEAR_4` is stored in the program variable **p_string**. The following statement takes the value of the expression `DOWNSHIFT(p_string)`, namely `gear_4`, and assigns it to another CHAR variable called **d_str**:

```
LET d_str = DOWNSHIFT(p_string)
```

For more information, see the DOWNSHIFT field attribute in Chapter 6, "Screen Forms."

GLS

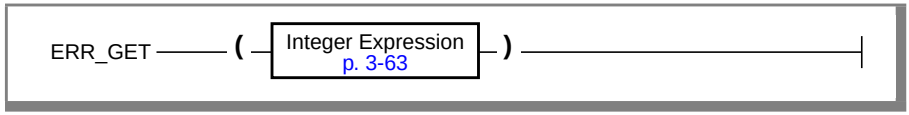
The results of conversion between uppercase and lowercase letters are based on the locale files, which identify the relationship between corresponding pairs of uppercase and lowercase letters. If the locale files do not provide this information, no case conversion occurs. DOWNSHIFT( ) has no effect on non-English characters in most multibyte locales. ♦

Reference

UPSHIFT( )

ERR_GET( )

The ERR_GET( ) function returns a character string containing the text of the 4GL or SQL error message whose numeric code you specify as its argument.



Usage

This is a possible sequence of steps for logging system error messages:

1. Call STARTLOG( ) to open or create an error log file (as described in "STARTLOG( )" on page 5-110).
2. Test the value of the global **status** variable to see if it is less than zero.
3. If **status** is negative, call ERR_GET( ) to retrieve the error text.
4. Call ERRORLOG( ) to make an entry into the error log file (as described in "ERRORLOG( )" on page 5-65).

The STARTLOG( ) function (step 1) automatically records the error text in the default error record, so the last three steps are not needed. ERR_GET( ) is most useful when you are developing a program. The message that it returns is probably not helpful to the user of your 4GL application. The argument of ERR_GET( ) is typically the value of the **status** variable, which is affected by both SQL and 4GL errors, or else a member of the global **SQLCA.SQLCODE** record. See "Exception Handling" on page 2-40. The LET statement in the following program fragment assigns the text of a 4GL error message to **errtext**, a CHAR variable:

```
LET op_status = STATUS
IF op_status < 0 THEN LET errtext = ERR_GET(op_status)
END IF
```

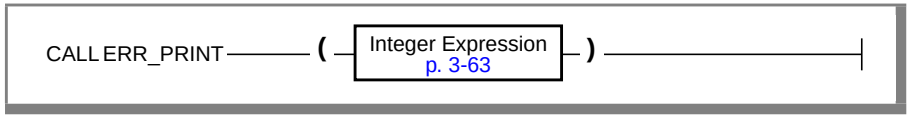
Here the value of **status** is first assigned to a variable, **op_status**, rather than testing ERR_GET(**status**) directly. Otherwise, the value of **status** normally would be zero, reflecting the success of the ERR_GET(**status**) function call.

References

ERR_PRINT( ), ERR_QUIT( ), ERRORLOG( ), STARTLOG( )

ERR_PRINT( )

The ERR_PRINT( ) function displays on the Error line the text of an SQL or 4GL error message, corresponding to a negative integer argument.



Usage

The argument of ERR_PRINT( ) specifies an error message number, which must be less than zero. It is typically the value of the global **status** variable, which is affected by both SQL and 4GL errors. For SQL errors only, you can examine the global **SQLCA.SQLCODE** record. For information on both **SQLCA.SQLCODE** and **status**, see "Error Handling with SQLCA" on page 2-45.

ERR_PRINT( ) is most useful when you are developing a 4GL program. The message that it returns is probably not helpful to the user of your application.

The following program segment sends any error message to the Error line:

```
LET op_status = STATUS
IF op_status < 0 THEN
    CALL ERR_PRINT(op_status)
END IF
```

Here the value of **status** is first assigned to a variable, **op_status**, rather than calling ERR_PRINT(**status**) directly. Otherwise, the value of **status** normally would be zero, reflecting the success of the ERR_PRINT(**status**) function call.

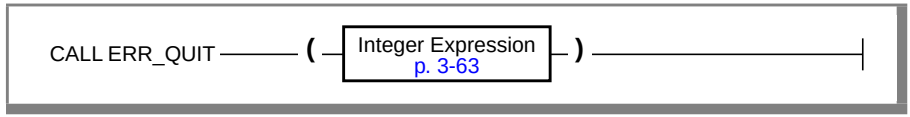
If you specify the WHENEVER ANY ERROR CONTINUE compiler directive (or the equivalent, the **anyerr** command-line flag), **status** is reset after certain additional 4GL statements, as described in "The ANY ERROR Condition" on page 4-378. For information about trapping errors, see "Exception Handling" on page 2-40.

References

ERR_GET( ), ERR_QUIT( ), ERRORLOG( ), STARTLOG( )

ERR_QUIT( )

The ERR_QUIT( ) function displays on the Error line the text of an SQL or 4GL error message, corresponding to the error code specified by its negative integer argument, and terminates the program.



Usage

The argument of ERR_QUIT( ) specifies an error message number, which must be less than zero. It is typically the value of the global **status** variable, which is reset by both SQL and 4GL errors. For SQL errors only, you can examine the global **SQLCA.SQLCODE** record. Both **SQLCA.SQLCODE** and **status** are described in "Error Handling with SQLCA" on page 2-45.

The ERR_QUIT( ) function is identical to the ERR_PRINT( ) function, except that ERR_QUIT( ) terminates execution once the message is printed. ERR_QUIT( ) is primarily useful when you are developing a 4GL program. The message that it returns is probably not helpful to the user of your application.

If an error occurs, the following statements display the error message on the Error line, and terminate program execution:

```
IF STATUS < 0 THEN
    CALL ERR_QUIT(STATUS)
END IF
```

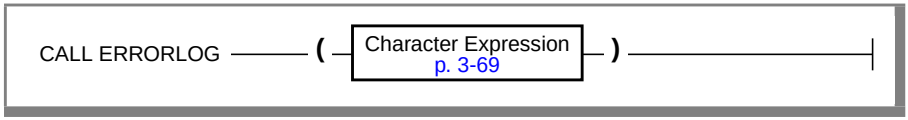
If you specify the WHENEVER ANY ERROR CONTINUE compiler directive (or the equivalent, the **anyerr** command-line flag), **status** is reset after certain additional 4GL statements, as described in "The ANY ERROR Condition" on page 4-378. For information about trapping errors, see "Exception Handling" on page 2-40.

References

ERR_GET( ), ERR_PRINT( ), ERRORLOG( ), STARTLOG( )

ERRORLOG( )

The ERRORLOG( ) function copies its argument into the current error log file.



Usage

If you simply invoke the STARTLOG( ) function, error records that 4GL appends to the error log after each subsequent error have this format:

```
Date: 07/06/99   Time: 12:20:20
Program error at "stock_one.4gl", line number 89.
SQL statement error number -239.
Could not insert new row - duplicate value in a UNIQUE INDEX
column.
SYSTEM error number -100
ISAM error:  duplicate value for a record with unique key.
```

The actual record might be incomplete if after an error the operating system fails to preserve the buffer that contains the module name or the line number. You can use the ERRORLOG( ) function to supplement default error records with additional information. Entries that ERRORLOG( ) makes in the error log file automatically include the date and time when the error was recorded.

This is a typical sequence of steps for logging system error messages:

1. Call STARTLOG( ) to open or create an error log file (as described in "STARTLOG( )" on page 5-110).
2. Test the value of the global **status** variable to see if it is negative.
3. If **status** < 0, call ERR_GET( ) to retrieve the error text (as described in "ERRORLOG( )" on page 5-65).
4. Call ERRORLOG( ) to make an entry into the error log file.

Unless you specify some other action for error conditions, WHENEVER ERROR CONTINUE is in effect by default. This default prevents the first SQL error from terminating program execution.

You can use the ERRORLOG( ) function to identify errors in programs that you are developing and to customize error handling. Even after implementation, some errors, such as those relating to permissions and locking, are sometimes unavoidable. These errors can be trapped and recorded by these logging functions.

You can use error-logging functions with other 4GL features for *instrumenting* a program, by tracking the way the program is used. This functionality is not only valuable for improving the program but also for recording work habits and detecting attempts to breach security. See *INFORMIX-4GL by Example* for a detailed example of a program with this functionality.

The following program fragment calls STARTLOG( ) in the MAIN program block. Here the ERRORLOG( ) function has a quoted string argument:

```
CALL STARTLOG("\\usr\\catherine\\error.log")
...
FUNCTION start_menu()
CALL ERRORLOG("Entering start_menu function")
```

The following example illustrates the use of ERR_GET( ) and ERRORLOG( ). It assumes that an error log file has already been created or initialized by calling the built-in STARTLOG( ) function:

```
FUNCTION add_cust()
   DEFINE errvar CHAR(80)
   WHENEVER ERROR CONTINUE
   INPUT BY NAME gr_customer.*
   INSERT INTO customer VALUES (gr_customer.*)
   IF STATUS < 0 THEN
      LET errvar = ERR_GET(STATUS)
      CALL ERRORLOG(errvar CLIPPED)
   END IF
END FUNCTION
```

If its argument is not of a character data type (for example, a DECIMAL variable), invoking ERRORLOG( ) can itself produce an error.

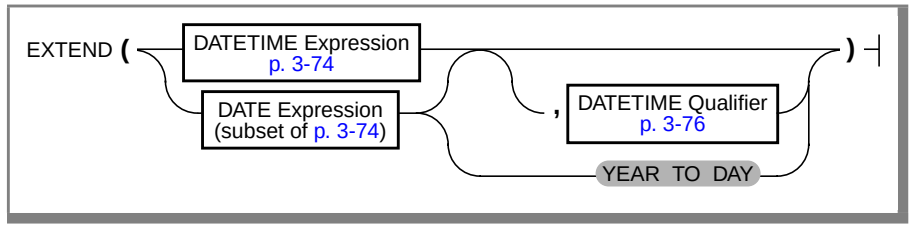
Automatic error logging increases the size of the generated executable.

References

ERR_GET( ), ERR_PRINT( ), ERR_QUIT( ), STARTLOG( )

EXTEND( )

The EXTEND( ) operator converts its DATETIME or DATE operand to a DATETIME value of a specified (or default) precision and scale.



Usage

The EXTEND( ) operator returns the value of its DATE or DATETIME operand, but with an adjusted precision that you can specify by a DATETIME qualifier. The operand can be a DATE or DATETIME expression of any valid precision. If it is a character string, it must consist of valid and unambiguous time-unit values and separators, but with these restrictions:

- It cannot be a character string in DATE format, such as `"12/12/99"`.
- It cannot be an ambiguous numeric DATETIME value, such as `"05:06"` or `"05"` whose time units are ambiguous.
- It cannot be a time expression that returns an INTERVAL value.

DATETIME Qualifiers

A qualifier can specify the precision of the result (and the scale, if FRACTION is the last keyword in the qualifier). The qualifier follows a comma and is of the form *first* TO *last*, where *first* and *last* are keywords to specify (respectively) the largest and smallest time unit in the result. Both can be the same.

If no qualifier is specified, the following defaults are in effect, based on the explicit or default precision of the DATE or DATETIME operand:

- The default qualifier that EXTEND( ) applies to a DATETIME operand is YEAR TO FRACTION(3).
- The default qualifier for a DATE operand is YEAR TO DAY.

See also "DATETIME Qualifier" on page 3-76.

The following rules are in effect for DATETIME qualifiers that you specify as EXTEND( ) operands:

- If a *first* TO *last* qualifier is specified, the first keyword must specify a time unit that is larger than (or the same as) the time unit that the last keyword specifies.
- If *first* specifies a time unit larger than any in the operand, omitted time units are filled with values from the system clock-calendar. In the following fragment, the first keyword specifies a time unit larger than any in **t_stamp**, so the value of the current year would be used:

```
DEFINE t_stamp DATETIME MONTH TO DAY
DEFINE annual DATETIME YEAR TO MINUTE
...
LET t_stamp = "1993-12-04 17"
LET annual = EXTEND(t_stamp, YEAR TO MINUTE)
```

- If *last* specifies a smaller time unit than any in the operand, the missing time units are assigned values according to these rules:
  - A missing MONTH or DAY is filled in with the value one (`01`).
  - Any missing HOUR, MINUTE, SECOND, or FRACTION is filled in with the value zero (`00`).
- If the operand contains time units outside the precision specified by the qualifier, the unspecified time units are discarded. For example, if you specify *first* TO *last* as DAY TO HOUR, any information about MONTH in the DATETIME operand is not used in the result.

Using EXTEND with Arithmetic Operators

If the precision of an INTERVAL value includes a time unit that is not present in a DATETIME or DATE value, you cannot combine the two values directly with the addition (+) or subtraction (-) binary arithmetic operators. You must first use the EXTEND( ) operator to return an adjusted DATETIME value on which to perform the arithmetic operation.

For example, you cannot directly subtract the 720-minute INTERVAL value in the next example from the DATETIME value that has a precision from YEAR to DAY. You can perform this calculation by using the EXTEND( ) operator:

```
EXTEND (DATETIME (1989-8-1) YEAR TO DAY, YEAR TO MINUTE)
   - INTERVAL (720) MINUTE(3) TO MINUTE
   --result:  DATETIME (1989-07-31 12:00) YEAR TO MINUTE
```

Here the EXTEND( ) operator returns a DATETIME value whose precision is expanded from YEAR TO DAY to YEAR TO MINUTE. This adjustment allows 4GL to evaluate the arithmetic expression. The result of the subtraction has the extended precision of YEAR TO MINUTE from the first operand.

In the next example, fragments of a report definition use DATE values as operands in expressions that return DATETIME values. Output from these PRINT statements would in fact be the numeric date and time (as described in "DATETIME Literal" on page 3-78) without the DATETIME keywords and qualifiers that are included here to show the precision of the values that the arithmetic expressions return.

```
DEFINE calendar DATE
   LET calendar = "05/18/1999"
   PRINT (calendar - INTERVAL (5-5) YEAR TO MONTH)
   --result:  DATETIME (1993-12-18) YEAR TO DAY
   PRINT (EXTEND(calendar, YEAR TO HOUR)
           - INTERVAL (4 8) DAY TO HOUR)
   --result:  DATETIME (1999-05-13 16) YEAR TO HOUR
```

You cannot directly combine a DATE with an INTERVAL value whose last qualifier is smaller than DAY. But as the previous example shows, you can use the EXTEND( ) operator to convert the value in a DATE column or variable to a DATETIME value that includes all the fields of the INTERVAL operand.

In the next example, the INTERVAL variable **how_old** includes fields that are not present in the DATETIME variable **t_stamp**, so the EXTEND( ) operator is required in the expression that calculates the sum of their values.

```
DEFINE t_stamp DATETIME YEAR TO HOUR
DEFINE age DATETIME DAY TO MINUTE
DEFINE how_old INTERVAL DAY TO MINUTE

LET t_stamp = "1989-12-04 17"
LET how_old = INTERVAL (28 9:25) DAY TO MINUTE
LET age = EXTEND(t_stamp, DAY TO MINUTE) + how_old
```

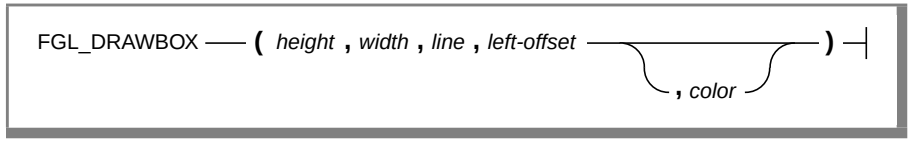
SQL statements can include a similar EXTEND( ) operator of SQL, whose first argument can be the name of a DATETIME or DATE database column.

Reference

UNITS

FGL_DRAWBOX( )

The FGL_DRAWBOX( ) function displays a rectangle of a specified size.



Element	Description
color	is an integer expression (as described in "Integer Expressions" on page 3-63) that returns a positive whole number, specifying a foreground color code.
height	is an integer expression, specifying the number of screen lines occupied by the rectangle.
left-offset	is an integer expression, specifying the horizontal coordinate (in characters) of the upper-left corner of the rectangle, where `1` is the first (or left-most) character in a line of the current 4GL window.
line	is an integer expression, specifying the vertical coordinate of the upper-left corner, where `1` means the first (or top-most) line.
width	is an integer expression, specifying the number of character positions occupied by each line of the rectangle.

Usage

The FGL_DRAWBOX( ) function draws a rectangle with the upper-left corner at (*line*, *left-offset*) and the specified *height* and *width*. These dimensions must have positive integer values, in units of lines and character positions, where (`0,0`) is the upper-left corner of the current 4GL window.

The optional *color* number must correspond to one of the following foreground colors.

Color Number	Foreground Color	Color Number	Foreground Color
0	WHITE	4	CYAN
1	YELLOW	5	GREEN
2	MAGENTA	6	BLUE
3	RED	7	BLACK

The **upscol** utility can specify these same color options in the **syscolatt** table. The default color is used when the color number is omitted. The color argument is optional.

As is the case with borders, the width of the line that draws the rectangle is fixed. This fixed width cannot be specified or modified when you invoke FGL_DRAWBOX( ). Also as with borders, 4GL draws the box with the characters defined in the **termcap** or **terminfo** files. You can specify alternative characters in these files. Otherwise, 4GL uses hyphens to create horizontal lines, pipe symbols (|) for vertical lines, and plus signs at the corners. To assign the box a color, you must use **termcap** because **terminfo** does not support color. For complete information on **termcap** and **terminfo**, see Appendix F, "Modifying termcap and terminfo."

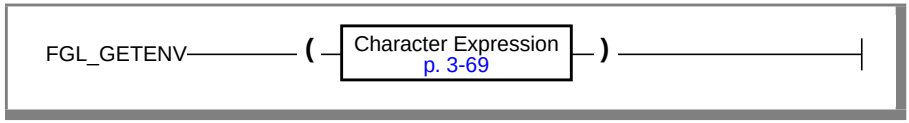
Rectangles drawn by FGL_DRAWBOX( ) are part of a displayed form. Each time that you execute the corresponding DISPLAY FORM or OPEN WINDOW ... WITH FORM statement, you must also redraw the rectangle.

If you invoke FGL_DRAWBOX( ) several times to create a display in which rectangles intersect, output from the most recent function call overlies any previously drawn rectangles. Screen fields and reserved lines, however, have a higher display priority than FGL_DRAWBOX( ) rectangles, regardless of the order in which the fields, lines, and rectangles are drawn.

Important: *In most applications, avoid drawing rectangles that intersect or overlap any field or reserved line. Reserved lines might be redrawn frequently during user interaction statements, partially erasing any rectangles at the intersections where they overlap the reserved lines. To avoid this problem, position the rectangles so they do not overlap any reserved lines or screen fields. For more information, see* *"Reserved Lines" on page 4-114**.*

FGL_GETENV( )

The FGL_GETENV( ) function returns a character string, corresponding to the value of an environment variable whose name you specify as the argument.



Usage

The argument of FGL_GETENV( ) must be a character expression that returns the name of an environment variable. To evaluate a call to FGL_GETENV( ), 4GL takes the following actions at runtime:

1. Evaluates the character expression argument of FGL_GETENV( )
2. Searches among environment variables for the returned value

If the requested value exists, the function returns it as a character string and then returns control of execution to the calling context.

The identifier of an environment variable is not a 4GL expression, so you typically specify the argument as a quoted string or character variable. For example, this call evaluates the **DBFORMAT** environment variable:

```
fgl_getenv("DBFORMAT")
```

You can assign the name of the environment variable to a character variable and use that variable as the function argument. If you declare a CHAR or VARCHAR variable called **env_var** and assign to it the name of an environment variable, a FGL_GETENV( ) function call could look like this:

```
fgl_getenv(env_var)
```

If the argument is a character variable, be sure to declare it with sufficient size to store the character value returned by the FGL_GETENV( ) function. Otherwise, 4GL truncates the returned value.

If the specified environment variable is not defined, FGL_GETENV( ) returns a NULL value. If the environment variable is defined but does not have a value assigned to it, FGL_GETENV( ) returns blank spaces.

You can use the FGL_GETENV( ) function anywhere within a 4GL program to examine the value of an environment variable. The following program segment displays the value of the **INFORMIXDIR** environment variable. The environment variable is identified in the FGL_GETENV( ) call by enclosing the name **INFORMIXDIR** between quotation marks:

```
DEFINE path CHAR(64)
...
LET path = fgl_getenv("INFORMIXDIR")
DISPLAY "Informix installed in ", path CLIPPED
```

The next example also displays the value of the **INFORMIXDIR** environment variable. In this case, the environment variable is identified by the **env_var** character variable, and its contents are stored in a variable called **path**:

```
DEFINE env_var CHAR(25),
       path CHAR(64)
...
LET env_var = "INFORMIXDIR"
LET path = fgl_getenv(env_var)
DISPLAY "Informix installed in ", path CLIPPED
```

The following example examines the environment to see if the **DBANSIWARN** environment variable is currently set:

```
DEFINE dbansi_flag SMALLINT
...
IF (fgl_getenv("DBANSIWARN") IS NOT NULL) THEN
   LET dbansi_flag = 1
END IF
```

GLS

In nondefault locales, FGL_GETENV( ) can return values that include non-ASCII characters that the code set of the locale supports. In multibyte locales, the returned value can include multibyte characters. ♦

References

For environment variables that control features of the database, see the *Informix Guide to SQL: Reference*. For descriptions of environment variables that can affect the visual displays of 4GL programs, see Appendix D, "Environment Variables."

FGL_GETKEY( )

The function FGL_GETKEY( ) waits for a key to be pressed and returns the integer code of the physical key that the user pressed.

Usage

Unlike FGL_LASTKEY( ), which can return a value indicating the logical effect of whatever key the user pressed, FGL_GETKEY( ) returns an integer representing the raw value of the physical key that the user pressed.

The FGL_GETKEY( ) function recognizes the same codes for keys that the FGL_KEYVAL( ) function returns. Unlike FGL_KEYVAL( ) , which can only return keystrokes that are entered in 4GL forms, FGL_GETKEY( ) can be invoked in any context where the user is providing keyboard entry.

GLS

Single-byte non-ASCII characters from the code set of the locale can also be returned. ♦

Here is an example of a program fragment that calls both functions, so that FGL_KEYVAL( ) evaluates what FGL_GETKEY( ) returns.

```
DEFINE key INT
PROMPT "Press the RETURN key to continue. " ||
       "Press any other key to quit."
LET key = FGL_GETKEY( )
IF key = FGL_KEYVAL("return") THEN
   CALL continue()
ELSE
   CALL quit()
END IF
```

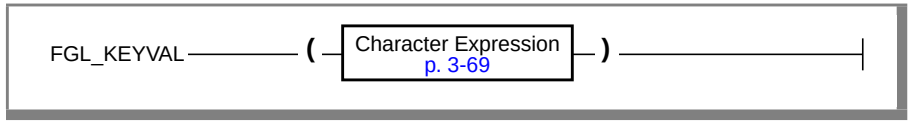
Important: *Here the term "key" refers to a physical element of the keyboard or to its logical effect rather than to the SQL construct of the same name.*

References

ASCII, FGL_KEYVAL( ), FGL_LASTKEY( ), ORD( )

FGL_KEYVAL( )

Function FGL_KEYVAL( ) returns the integer code of a logical or physical key.



Usage

The FGL_KEYVAL( ) function returns NULL unless its argument specifies one of the following physical or logical keys:

- A single letter or a digit
- Non-alphanumeric symbols (such as !, @, and #)
- Any of the keywords in the following table (in uppercase or lowercase letters)

ACCEPT	HELP	NEXT *or*	RETURN
DELETE	INSERT	NEXTPAGE	RIGHT
DOWN	INTERRUPT	PREVIOUS *or*	TAB
ESC *or* ESCAPE	LEFT	PREVPAGE	UP
F1 *through* F64			
CONTROL-character (*except* A, D, H, I, J, L, M, R, *or* X			

GLS

Single-byte non-ASCII characters from the code set of the locale can also be returned. ♦

Enclose the argument in quotation marks. If you specify a single letter, FGL_KEYVAL( ) considers the case. In all other instances, FGL_KEYVAL( ) ignores the case of its argument, which can be uppercase or lowercase letters. If the argument is invalid, FGL_KEYVAL( ) returns NULL.

Important: *Here the term "key" refers to a physical element of the keyboard or to its logical effect rather than to the SQL construct of the same name.*

Using FGL_KEYVAL( ) with FGL_GETKEY( ) or FGL_LASTKEY( )

FGL_KEYVAL( ) can be used in form-related statements to examine a value returned by the FGL_GETKEY( ) or FGL_LASTKEY( ) function. By comparing the values returned by FGL_KEYVAL( ) with what FGL_GETKEY( ) or FGL_LASTKEY( ) returns, you can determine whether the last key that the user pressed was a specified logical or physical key. Typically, you use the FGL_KEYVAL( ) function in conditional statements and Boolean comparisons:

```
DEFINE key_var INTEGER
...
INPUT BY NAME p_customer.fname THRU p_customer.phone
   ...
   AFTER FIELD phone
      IF FGL_LASTKEY() = FGL_KEYVAL("f1") THEN
         ...
      END IF
   END INPUT
```

This example displays a message and moves the cursor to the **manu_code** field if the user presses the UP ARROW key to leave the **stock_num** field:

```
CONSTRUCT query_1 ON stock.* FROM s_stock.*
   ...
   AFTER FIELD stock_num
      IF FGL_LASTKEY() = FGL_KEYVAL("up") THEN
         DISPLAY "You cannot move up from here."
         NEXT FIELD manu_code
      END IF
   ...
END CONSTRUCT
```

To determine whether the user performed some action, such as inserting a row, specify the logical name of the action (such as INSERT) rather than the name of the physical key (such as F1). For example, the logical name of the default Accept key is ESCAPE. To test if the key most recently pressed by the user was the Accept key, specify FGL_KEYVAL("ACCEPT") rather than FGL_KEYVAL("escape") or FGL_KEYVAL("ESC"). Otherwise, if a key other than ESCAPE is set as the Accept key and the user presses that key, FGL_LASTKEY( ) does not return a code equal to FGL_KEYVAL("ESCAPE"). The value returned by FGL_LASTKEY( ) is undefined in a MENU statement.

References

ASCII, FGL_GETKEY( ), FGL_LASTKEY( ), ORD( )

FGL_LASTKEY( )

The FGL_LASTKEY( ) function returns an INTEGER code, corresponding to the logical key that the user most recently typed in a field of a screen form.

Usage

The FGL_LASTKEY( ) function returns a numeric code for the user's last keystroke before FGL_LASTKEY( ) was called. For example, if the last key that the user entered was the lowercase s, the FGL_LASTKEY( ) function returns 115. Appendix A lists the numeric codes for all the ASCII characters. The value returned by FGL_LASTKEY( ) is undefined in a MENU statement.

Important: *Here the term "key" refers to a physical element of the keyboard of a terminal or to its logical effect rather than to the SQL construct of the same name.*

Using FGL_LASTKEY( ) with FGL_KEYVAL( )

You do not need to know the specific key codes to use FGL_LASTKEY( ). The built-in FGL_KEYVAL( ) function can return a code to compare with the value returned by FGL_LASTKEY( ). For more information, see "FGL_KEYVAL( )" on page 5-76. The FGL_KEYVAL( ) function lets you compare the last key that the user pressed with a logical or physical key. For example, to check if the user pressed the Accept key, compare FGL_LASTKEY( ) with the FGL_KEYVAL("accept") value.

The following CONSTRUCT statement checks the value of the last key that the user entered in each field. If the user last pressed RETURN, the program displays a message in the Error line:

```
CONSTRUCT query_1 ON stock.* FROM s_stock.*
   BEFORE CONSTRUCT
      DISPLAY "Use the TAB key to move ",
            "between the fields." AT 1,1
   AFTER FIELD stock_num, manu_code, description,
         unit_price, unit, unit_descr
      IF FGL_LASTKEY() = FGL_KEYVAL("return") THEN
         ERROR "Use the TAB key to move the cursor ",
               "between the fields."
      END IF
END CONSTRUCT
```

Here (as in ON KEY clauses), RETURN is a synonym for ENTER.

The following example demonstrates using the FGL_LASTKEY( ) function after a PROMPT statement that expects the user to respond to the prompt with a single keystroke. The FGL_LASTKEY( ) function returns the code of the key the user pressed to the program. The FGL_LASTKEY( ) function compares the code with the code for the RETURN key. If an exact match occurs, 4GL calls the **continue( )** function. If a match does not occur because the user pressed a key other than RETURN, 4GL calls the **quit( )** function:

```
DEFINE      value CHAR,
      key INTEGER

PROMPT "Press the RETURN key to continue. ",
      "Press any other key to quit." FOR CHAR value
LET key = FGL_LASTKEY()
IF key = FGL_LASTKEY("return") THEN
   CALL continue()
ELSE
   CALL quit()
END IF
```

AUTONEXT Fields

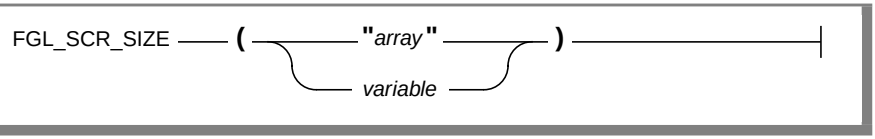
If FGL_LASTKEY( ) is invoked after the user enters a value in a field with the AUTONEXT attribute, 4GL returns the code of the last key that the user entered, regardless of any processing done in the AFTER FIELD or BEFORE FIELD clause. For more information, see "AUTONEXT" on page 6-34.

References

ASCII, FGL_GETKEY( ), FGL_KEYVAL( ), ORD( )

FGL_SCR_SIZE( )

The function FGL_SCR_SIZE( ) accepts as its argument the name of a screen array in the currently opened form and returns an integer that corresponds to the number of screen records in that screen array.



Element	Description
array	is the identifier (between quotation marks) of a screen array from the INSTRUCTIONS section of the specification of the current form.
variable	is a CHAR or VARCHAR variable containing the *array* identifier.

Usage

The built-in FGL_SCR_SIZE( ) function returns the declared size of a specified screen array at runtime. In the following example, a form specification file (called **file.per**) declares two screen arrays, called **s_rec1** and **s_rec2**:

```
DATABASE FORMONLY

SCREEN
{
[f1    ] [f2    ]
[f1    ] [f2    ]
[f1    ] [f2    ]
[f3    ] [f4    ]
[f3    ] [f4    ]
[f5    ]
}

ATTRIBUTES
f1 = FORMONLY.a ;
f2 = FORMONLY.b ;
f3 = FORMONLY.c ;
f4 = FORMONLY.d ;
f5 = FORMONLY.e ;
```

```
INSTRUCTIONS
DELIMITERS ""
SCREEN RECORD s_rec1[3] (a,b)
SCREEN RECORD s_rec2 (c,d)
```

The following 4GL program invokes the FGL_SCR_SIZE( ) function:

```
MAIN
DEFINE n1,n2 INT
   DEFINE ch CHAR(10)

   OPEN WINDOW w1 AT 2,3 WITH FORM "file" ATTRIBUTE (BORDER)
   CALL fgl_scr_size("s_rec1") RETURNING n1
   LET n1 = fgl_scr_size("s_rec1")-- Can also be called
      -- in a LET statement
   DISPLAY "n1 = ", n1

   LET ch = "s_rec2"
   CALL fgl_scr_size(ch) RETURNING n2
   LET n2 = fgl_scr_size(ch) -- Can also be called
                                          -- in a LET statement
DISPLAY "n2 = ", n2
   CLOSE WINDOW w1
END MAIN
```

This program produces the following output:

```
n1 = 3
n2 = 2
```

The proper value is returned even though the array dimension is not specified in the form file.

An error is returned if no form is open or if the specified *array* is not in the current open form.

References

ARR_CURR( ), ARR_COUNT( )

FGL_SETCURRLINE ( )

During the INPUT ARRAY or DISPLAY ARRAY statement the FGL_SETCURRLINE( ) function positions the cursor at a specified program record within the program array that is displayed in the current screen array, and displays that record.

Element	Description
num	is a literal integer, specifying the ordinal number of a program record within the program array that was specified in the current INPUT ARRAY or DISPLAY ARRAY statement.

Usage

The current line of a screen array is the line that displays the screen cursor at the beginning of a BEFORE ROW or AFTER ROW clause.

The integer argument of the FGL_SETCURRLINE( ) function specifies which program record to display in the current screen array, and moves the cursor to that line. . The first row of the program array is numbered 1.

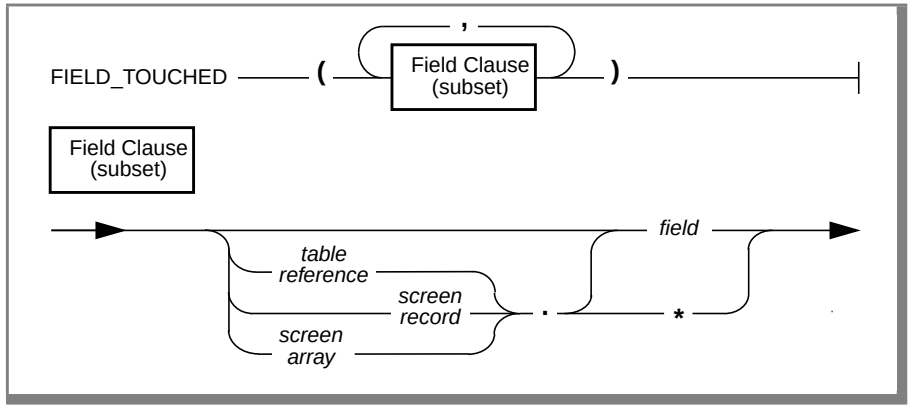
Error -4669 is issued if you attempt to call the FGL_SETCURRLINE( ) function outside a statement block of the DISPLAY ARRAY or INPUT ARRAY statement.

References

ARR_CURR( ), SCR_LINE( ), SET_COUNT( )

FIELD_TOUCHED( )

The FIELD_TOUCHED( ) operator tests whether the user has entered or edited a value in a specified field or list of fields of the current 4GL form. (This operator can only appear within CONSTRUCT, INPUT, and INPUT ARRAY statements.)



Element	Description
field	is the name of a screen field (from the ATTRIBUTES section).
screen array	is the name of a screen array (from the INSTRUCTIONS section).
screen record	is the name of a screen record (from the INSTRUCTIONS section of the form specification).
table reference	is a table name, alias, synonym, or FORMONLY keyword (from the TABLES section of the form specification).

Usage

FIELD_TOUCHED( ) returns the Boolean value TRUE (meaning that the user changed the contents of a field) after a DISPLAY statement displays data in any of the specified fields or after the user presses any of the following keys:

- Any printable character (including SPACEBAR)
- CONTROL-X (character delete)
- CONTROL-D (clear to end of field)

After any of these keystrokes, the FIELD_TOUCHED( ) operator returns TRUE, regardless of whether the keystroke actually changed the value in the field. (The locale files classify each character as printable or unprintable.)

Otherwise, the FIELD_TOUCHED( ) operator returns FALSE, indicating that none of the specified fields have been edited. Moving through a field (by pressing RETURN, TAB, or the arrow keys) does *not* mark a field as *touched*.

Important: *FIELD_TOUCHED( ) is valid only in CONSTRUCT, INPUT, and INPUT ARRAY statements. When you use it, 4GL assumes that you are referring to the current screen record rather than to a different row of the screen array.*

This operator does not register the effect of 4GL statements that appear in a BEFORE CONSTRUCT or BEFORE INPUT clause. You can assign values to fields in these clauses without marking the fields as touched.

In the following program fragment, an IF statement tests whether the user has entered a value into any field. If no field has been touched, the program prompts the user to indicate whether to retrieve all customer records. If the user types N or n, the CONTINUE CONSTRUCT statement is executed, and the screen cursor is positioned in the form, giving the user another opportunity to enter selection criteria. If the user types any other key, the program terminates the IF statement and reaches the END CONSTRUCT keywords.

```
CONSTRUCT BY NAME query1 ON customer.*
...
   AFTER CONSTRUCT
      IF NOT FIELD_TOUCHED(customer.*) THEN
         PROMPT "Do you really want to see ",
            "all customer rows? (y/n)"
            FOR CHAR answer
         IF answer MATCHES "[Nn]" THEN
            CONTINUE CONSTRUCT
         END IF
      END IF
END CONSTRUCT
```

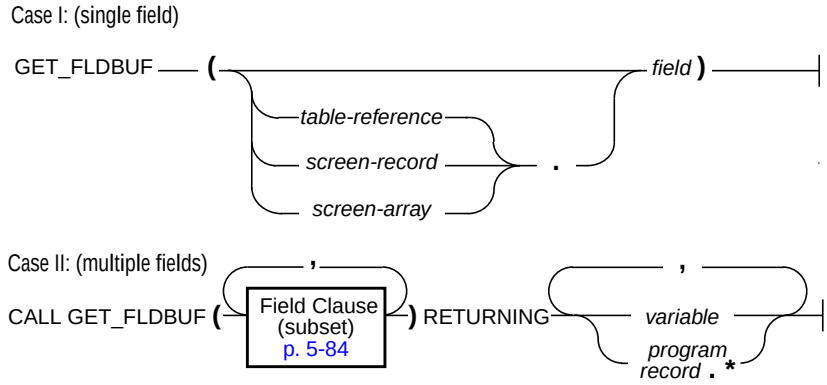
This strategy is not as dependable as testing whether `query1 = " 1=1"` after the END CONSTRUCT keywords because the user might have left all the fields blank after first entering and then deleting query criteria in some field. In that case, the resulting Boolean expression (`" 1=1"`) can retrieve all rows, but FIELD_TOUCHED( ) returns TRUE, and the PROMPT statement is not executed. For additional information, see "Searching for All Rows" on page 4-60.

References

Boolean Operators, FGL_GETKEY, FGL_KEYVAL( ), FGL_LASTKEY( ), GET_FLDBUF( ), INFIELD( )

GET_FLDBUF( )

The GET_FLDBUF( ) operator returns the character values of the contents of one or more fields in the currently active screen form. (This operator can only appear within the CONSTRUCT, INPUT, and INPUT ARRAY statements of 4GL.)



Element	Description
field	is the name of a field in the current screen form.
program record	is the name of a program record of CHAR or VARCHAR variables in which to store values from the specified fields.
screen-array	is the name of a screen array that was defined in the INSTRUCTIONS section of the form specification file.
screen-record	is the name of a screen record that is explicitly or implicitly defined in the form specification file.
table reference	is the unqualified name, alias, or synonym of a database table or view, or else the keyword FORMONLY.
variable	is a name within a list of one or more character variables, separated by commas. Variables must correspond in number and position with the list of fields in the field clause.

Usage

GET_FLDBUF( ) operates on a list of one or more fields. For example, this LET statement assigns the value in the **lname** field to the **lbuff** variable:

```
LET lbuff = GET_FLDBUF(lname)
```

To specify a list of several field names as operands of GET_FLDBUF( ), you must use the CALL statement with the RETURNING clause. Insert commas to separate successive field names and successive variables:

```
CALL GET_FLDBUF(c_num, company, lname)
   RETURNING p_cnum, p_company, p_lname
```

The following statement returns a set of character values corresponding to the contents of the **s_customer** screen record and assigns these values to the **p_customer** program record:

```
CALL GET_FLDBUF(s_customer.*) RETURNING p_customer.*
```

(The first asterisk (*) specifies all the fields in the **s_customer** screen-record; the second specifies all the members of the **p_customer** program record.)

You can use the GET_FLDBUF( ) operator to assist a user when entering a value in a field. For example, if you have an input field for last names, you can include an ON KEY clause that lets a user enter the first few characters of the desired last name. If the user calls the ON KEY clause, 4GL displays a list of last names that begin with the characters entered. The user can then choose a last name from the list. The following program fragment demonstrates this use of the GET_FLDBUF( ) operator:

```
DEFINE lname, myquery, partial_name CHAR(20),
       tw ARRAY[10] OF CHAR(20),
       a INTEGER
...
INPUT BY NAME lname
   ON KEY (CONTROL-P)
      LET partial_name = GET_FLDBUF(lname)
      LET myquery = "SELECT lname FROM teltab ",
         "WHERE lname MATCHES \"", partial_name CLIPPED, "*\""
      OPEN WINDOW w1 AT 5,5 WITH FORM "tel_form"
         ATTRIBUTE (BORDER)
      DISPLAY partial_name AT 1,1
      PREPARE mysubquery FROM myquery
      DECLARE q1 CURSOR FOR mysubquery
      LET a = 0
      FOREACH q1 INTO lname
         LET a = a + 1
         ...
      END FOREACH
      DISPLAY a TO ncount
```

```
      IF (a = 0) THEN
         PROMPT "Nothing beginning with these letters"
FOR CHAR partial_name
      ELSE
         IF (a > 10) THEN  LET a = 10
         END IF
         CALL SET_COUNT(a)
         DISPLAY ARRAY tw TO srec.*
      END IF
   ...
END INPUT
```

If you assign the character string returned by the GET_FLDBUF( ) operator to a variable that is not defined as a character data type, 4GL tries to convert the string to the appropriate data type. Conversion is not possible in these cases:

- The field contains special characters (for example, date or currency characters) that 4GL cannot convert.
- GET_FLDBUF( ) is called from a CONSTRUCT statement, and the field contains comparison or range operators that 4GL cannot convert.

GET_FLDBUF( ) is valid only in CONSTRUCT, INPUT, and INPUT ARRAY statements. When it encounters this operator in an INPUT ARRAY statement, 4GL assumes that you are referring to the current row. You cannot use a subscript within brackets to reference a different row of the screen array.

The following example uses the GET_FLDBUF( ) and FIELD_TOUCHED( ) operators in an AFTER FIELD clause in a CONSTRUCT statement. The FIELD_TOUCHED( ) operator checks whether the user has entered a value in the **zipcode** field. If FIELD_TOUCHED( ) returns TRUE, GET_FLDBUF( ) retrieves the value entered in the field and assigns it to the **p_zip** program variable. If the first character in the **p_zip** variable is not a 9, the program displays an error, clears the field, and returns the cursor to the field.

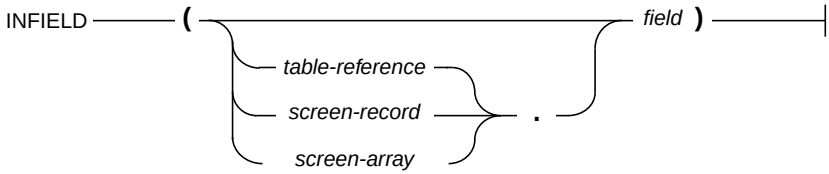
```
CONSTRUCT BY NAME query1 ON customer.*
   ...
   AFTER FIELD city
      IF FIELD_TOUCHED(zipcode) THEN LET p_zip = GET_FLDBUF(zipcode)
         IF p_zip[1,1] <> "9" THEN
            ERROR "You can only search in section 9."
            CLEAR zipcode
            NEXT FIELD zipcode
         END IF
      END IF
```

References

FIELD_TOUCHED( ), INFIELD( )

INFIELD( )

The INFIELD( ) operator in CONSTRUCT, INPUT, and INPUT ARRAY statements tests whether its operand is the identifier of the current screen field.



Element	Description
field	is the name of a field in the current screen form.
screen-array	is the name of a screen array that was defined in the INSTRUCTIONS section of the form specification file.
screen-record	is the name of a screen record that is explicitly or implicitly defined in the form specification file.
table-reference	is the unqualified name, alias, or synonym of a database table or view, or else the keyword FORMONLY.

Usage

INFIELD( ) is a Boolean operator that returns the value TRUE if *field* is the name of the current screen field. Otherwise, INFIELD( ) returns the value FALSE. (For information on assigning a name to a display field of a screen form, see "ATTRIBUTES Section" on page 6-25.)

Important: *You must specify a field name rather than a field tag as the operand.*

You can use INFIELD( ) during a CONSTRUCT, INPUT, or INPUT ARRAY statement to take field-dependent actions.

The INFIELD( ) operator is typically part of an ON KEY clause, often with the built-in function SHOWHELP( ) to display help messages to the user. The next code example is from a program that uses INFIELD( ) to determine whether to call a function:

```
ON KEY (CONTROL-F, F5)
   IF INFIELD(customer_num) THEN
      CALL cust_popup()
```

When a user presses either of two keys during the INPUT, the **cust_popup( )** function is invoked if the screen cursor is in the **customer_num** field.

In the following fragment, **call_flag** and **res_flag** are the names of fields:

```
ON KEY (F2, CONTROL-E)
   IF INFIELD(call_flag) OR INFIELD(res_flag) THEN
      IF INFIELD (call_flag) THEN
         LET fld_flag = "C"
      ELSE        --* user pressed F2 (CTRL-E) from res_flag
         LET fld_flag = "R"
      END IF
   ...
   END IF
```

Subsequent code could use these field names to determine which column of a row to edit.

In the following example, the INPUT statement uses the INFIELD( ) operator with the SHOWHELP( ) function to display field-dependent help messages.

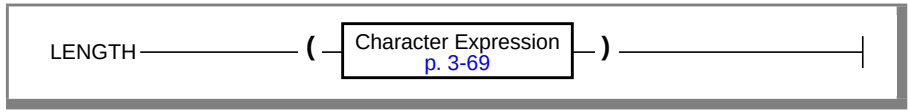
```
INPUT gr_equip.* FROM sr_equip.*
    ON KEY(CONTROL-B)
        CASE
            WHEN INFIELD(part_num)
                CALL SHOWHELP(301)
            WHEN INFIELD(part_name)
                CALL SHOWHELP(302)
            WHEN INFIELD(supplier)
                CALL SHOWHELP(303)
            ...
        END CASE
END INPUT
```

References

SCR_LINE( ), SHOWHELP( ), FIELD_TOUCHED( ), GET_FLDBUF( )

LENGTH( )

The LENGTH( ) function accepts a character string argument and returns an integer, representing the number of bytes in its argument (but disregarding any trailing blank spaces).



Usage

The LENGTH( ) function returns an integer value, based on the length (in bytes) of its character-expression argument.

Statements in the next example center a report title on an 80-column page:

```
LET title = "Invoice for ", fname CLIPPED,
                  " ", lname CLIPPED
LET offset = (80 - length(title))/2
PRINT COLUMN offset, title
```

The following are among the possible uses for the LENGTH( ) function:

- You can check whether a user has entered a database name and, if not, set a default name.
- Check whether the user has supplied the name of a file to receive the output from a report and, if not, set a default output.
- Use LENGTH(*string*) as the upper limit in a FOR loop, and check each character in *string* for a specific character. For example, you can check for a period (.) to determine whether a table name has a qualifier.

LENGTH( ) is also useful as a check on user input. In the following example, an IF statement is used to determine whether the user has responded to a displayed message:

```
IF LENGTH (ans1) = 0 THEN
   PROMPT "Press RETURN to continue: " FOR input_val
ELSE ...
```

If its argument evaluates to a NULL string, LENGTH( ) returns zero.

Using LENGTH( ) in SQL Expressions

Unlike some other built-in functions of 4GL, you can use LENGTH( ) in SQL statements as well as in other 4GL statements. LENGTH( ) can also be called from a C function. (That is, Informix database servers support a function of the same name and of similar functionality.)

In a SELECT or UPDATE statement, the argument of LENGTH( ) is the identifier of a character column. In this context, LENGTH( ) returns the number of bytes in the CLIPPED data value (for CHAR and VARCHAR columns) or the full number of bytes (for TEXT and BYTE data types).

The LENGTH( ) function can also take the name of a database column as its argument but only within an SQL statement.

GLS

LENGTH( ) in Multibyte Locales

LENGTH( ) avoids returning incorrect values when it encounters partial characters while operating in a multibyte locale. If the LENGTH( ) function encounters a partial (or otherwise invalid) character in its argument, LENGTH( ) returns a value that disregards any of the following items:

- The first invalid character (or partial character)
- All subsequent characters
- Any immediately preceding single-byte or multibyte white spaces

For example, suppose that `w` is an invalid character. The following expression evaluates to 7 because `ABCD EF` corresponds to seven bytes:

```
LENGTH(ABCD EF    wXYZ)
```

The invalid character, the blank spaces preceding it, and all subsequent characters in the argument are ignored because `w` is an invalid character. ♦

References

CLIPPED, USING

LINENO

The LINENO operator returns the number of the line within the page that is currently printing. (This operator can appear only in the FORMAT section of a REPORT program block.)

Usage

This operator returns the value of the line number of the report line that is currently printing. 4GL computes the line number by calculating the number of lines from the top of the current page, including the TOP MARGIN.

For example, the following program fragment examines the value of LINENO. If this value is less than 9, a PRINT statement formats and displays it, beginning in the 10th character position after the left margin.

```
IF (LINENO > 9) THEN
   PRINT COLUMN 10, LINENO USING "Line <<<"
END IF
```

You can specify LINENO in the PAGE HEADER, PAGE TRAILER, and other report control blocks to find the print position on the current page of a report.

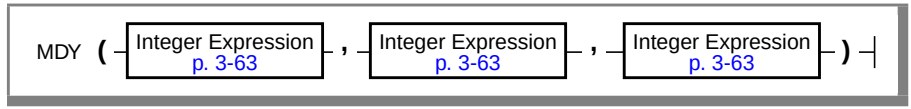
4GL cannot evaluate the LINENO operator outside the FORMAT section of a REPORT program block. The value that LINENO returns must be assigned to a variable that is not local to the report if you need to reference this value within some other program block of your 4GL application.

Reference

PAGENO

MDY( )

The MDY( ) operator returns a value of the DATE data type from three integer operands that represent the *month*, the *day* of the month, and the *year*.



Usage

The MDY( ) operator converts to a single DATE format a list of exactly three valid integer expressions. The three expressions correspond with the month, day, and year elements of a calendar date:

- The first expression must return an integer, representing the number of the month (1 through 12).
- The second must return an integer, representing the number of the day of the month (1 through 28, 29, 30, or 31, depending on the month).
- The third must return a four-digit integer, representing the year.

An error results if you specify values outside the range of days and months in the calendar or if the number of operands is not three.

You must enclose the three integer expression operands between parentheses, separated by commas, just as you would if MDY( ) were a function.

The third expression cannot be the abbreviation for the year. For example, 99 specifies a year in the first century, approximately 1,900 years ago.

The following program uses MDY( ) to return a DATE value, which is then assigned to a variable and displayed on the screen:

```
MAIN
DEFINE a_date DATE
LET a_date = MDY(12/2,3+2,1988)
DISPLAY a_date
END MAIN
```

Reference

DATE( )

Membership (.) Operator

The *membership operator*, a period (.), specifies that its right-hand operand is a member of the set whose name is its left-hand operand.

Element	Description
member	is the name of a component of structure.
structure	is the name of a RECORD variable, screen record, screen array of records, or database table, view, or synonym that has *member* as a component.

Usage

The *structure* value can specify a screen record, screen array, RECORD variable, or database table, view, or synonym.

If *member* is the name of a database column, *structure* can be qualified by a table qualifier. (For details, see "Table Qualifiers" on page 3-89.)

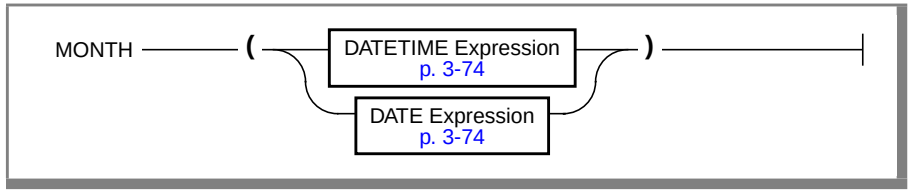
The *structure* value, *member* value, or both can be expressions that include the membership operator. For example:

```
myRec.data             --member of a program or screen record
alias.array[3].field5 --field within a screen array
```

The LET statement syntax diagram (shown in "Data Types and Expressions" on page 3-5) illustrates the syntax of members that are RECORD variables. For a description of the members of a RECORD variable, see also "THRU or THROUGH Keywords and .* Notation" on page 3-92. In contexts where more than one member is valid, you can substitute an asterisk (*) for *member* to specify every member of *structure*.

MONTH( )

The MONTH( ) operator returns a positive whole number between 1 and 12, corresponding to the *month* portion of a DATE or DATETIME operand.



Usage

The MONTH( ) operator extracts an integer value for the month in a DATE or DATETIME value. You cannot specify an INTERVAL operand.

The following program extracts the month time unit from a DATETIME literal expression. It evaluates MONTH(**date_var**) as an operand of a Boolean expression to test whether the month is earlier in the year than March.

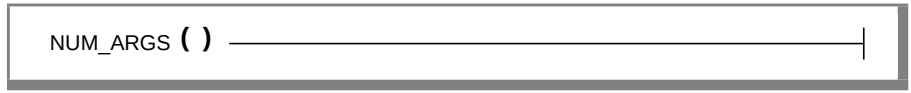
```
MAIN

DEFINE date_var DATETIME YEAR TO SECOND
DEFINE current_month CHAR(10)
DEFINE month_var INT

LET current_month = CURRENT MONTH TO MONTH
LET date_var = DATETIME(89-01-12 18:47:32) YEAR TO SECOND
LET month_var = MONTH(date_var)
DISPLAY "The current month is: ", current_month
DISPLAY "The month of interest is month number : ",
    month_var USING "##"
IF MONTH(date_var) < 3
   THEN DISPLAY "Month of interest is Feb. or Jan."
END IF

END MAIN
```

References

DATE( ), DAY( ), TIME( ), WEEKDAY( ), YEAR( )

NUM_ARGS( )

The NUM_ARGS( ) function takes no arguments. It returns an integer that corresponds to the number of command-line arguments that followed the name of your 4GL program when the user invoked it.



Usage

You can use the ARG_VAL( ) built-in function to retrieve individual arguments. By using NUM_ARGS( ) with the ARG_VAL( ) function, the program can pass command-line arguments to the MAIN statement or to whatever program block invokes the NUM_ARGS( ) and ARG_VAL( ) functions.

In the following examples, both of the command lines include three arguments:

`myprog.4gi kim sue joe` (executable compiled C version)

`fglgo myprog kim sue joe` (command for p-code runner in RDS)

After either of these command lines, NUM_ARGS( ) sets 3 as the upper limit of variable **i** in the FOR loop of the program fragment that follows:

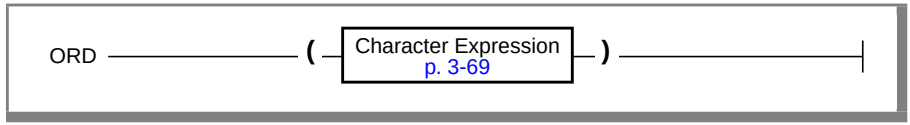
```
DEFINE pa_args ARRAY[8] OF CHAR(10),
       i    SMALLINT
FOR i = 1 TO NUM_ARGS()
       LET pa_args[i] = ARG_VAL(i)
END FOR
```

Reference

ARG_VAL( )

ORD( )

The ORD( ) function accepts as its argument a character expression and returns the integer value of the first byte of that argument.



For the default (U.S. English) locale, the ORD( ) function is the logical inverse of the ASCII operator. Only the first byte of the argument is evaluated.

The following line assigns the value 66 to the integer **ord1**:

```
LET ord1 = ORD ("Belladonna")
```

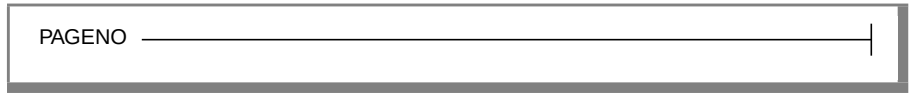
This built-in function is case sensitive; if the first character in its argument is an uppercase letter, ORD( ) returns a value different from what it would return if its argument had begun with a lowercase letter.

References

ASCII, FGL_KEYVAL( )

PAGENO

The PAGENO operator returns a positive whole number, corresponding to the number of the page of report output that 4GL is currently printing. (PAGENO is valid only in the FORMAT section of a REPORT program block.)



Usage

This operator returns a positive integer whose value is the number of the page of output that includes the current print position in the currently executing report.

For example, the following program fragment conditionally prints the value returned by PAGENO, using the USING operator to format it, if this value is less than 10,000:

```
IF (PAGENO < 10000) THEN
   PRINT COLUMN 28, PAGENO USING "page <<<<"
END IF
```

You can include the PAGENO operator in PAGE HEADER and PAGE TRAILER control blocks and in other control blocks of a report definition to identify the page numbers of output from a report.

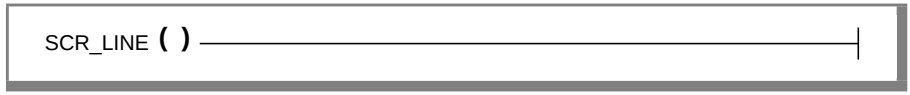
4GL cannot evaluate the PAGENO operator outside the FORMAT section of a REPORT program block. If some other program block of your 4GL application needs to reference the value that PAGENO returns, the report must assign that value to a program variable whose scope of reference is not local to the report.

Reference

LINENO

SCR_LINE( )

The SCR_LINE( ) function returns a positive integer that corresponds to the number of the current screen record in its screen array during a DISPLAY ARRAY or INPUT ARRAY statement.



Usage

The current screen record is the line of a screen array that contains the screen cursor at the beginning of a BEFORE ROW or AFTER ROW clause.

The first record of the program array and of the screen array are both numbered 1. The built-in 4GL functions SCR_LINE( ) and ARR_CURR( ) can return different values if the program array is larger than the screen array.

The following program fragment tests what the user enters and rejects it if the **state** field value indicates that the customer is not from California:

```
DEFINE pa_clients ARRAY[90] OF RECORD
    fname   CHAR(15),
    lname   CHAR(15),
    state   CHAR(2)
    END RECORD,
    curr_pa, curr_sc SMALLINT

INPUT ARRAY pa_clients FROM sa_clients.*
    AFTER FIELD state
        LET curr_pa = ARR_CURR()
        LET curr_sc = SCR_LINE()
        IF UPSHIFT(pa_clients[curr_pa].state) != "CA" THEN
            ERROR "Policy for California clients only"
            INITIALIZE pa_clients[curr_pa].* TO NULL
            CLEAR scr_array[curr_sc].*
            NEXT FIELD fname
        END IF
END INPUT
```

The following example makes use of SCR_LINE( ) and of the related ARR_CURR( ) built-in function to assign values to variables within the BEFORE ROW clause of an INPUT ARRAY statement. Because these functions are invoked in the BEFORE ROW control block, the respective **curr_pa** and **curr_sa** variables are evaluated each time that the cursor moves to a new line and are available within other clauses of the INPUT ARRAY statement.

```
INPUT ARRAY ga_items FROM sa_items.* HELP 62
   BEFORE ROW
      LET curr_pa = ARR_CURR()
      LET curr_sa = SCR_LINE()
```

In a later statement within INPUT ARRAY, you can have a statement such as the following example, which fills in the **description** and **unit_price** fields on the screen:

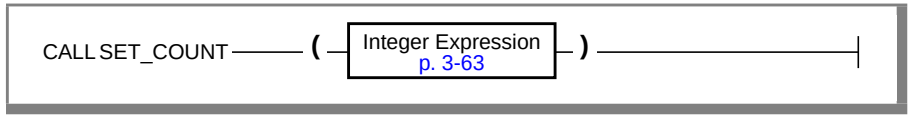
```
DISPLAY
 ga_items[curr_pa].description, ga_items[curr_pa].unit_price
TO
 sa_items[curr_sa].description, sa_items[curr_sa].unit_price
```

References

ARR_COUNT( ), ARR_CURR( )

SET_COUNT( )

The SET_COUNT( ) function specifies the number of records that contain data in a program array.



Usage

Before you use an INPUT ARRAY WITHOUT DEFAULTS statement or a DISPLAY ARRAY statement, you must call the SET_COUNT( ) function with an integer argument to specify the total number of records in the program array. In typical applications, these records contain the values in the retrieved rows that a SELECT statement returned from a database and are associated with a database cursor.

The SET_COUNT( ) built-in function sets an initial value from which the ARR_COUNT( ) function determines the total number of members in an array. If you do not explicitly call ARR_COUNT( ), a default value of zero is assigned.

In the following program fragment, the variable **n_rows** is an array index that received its value in an earlier FOREACH loop. The index was initialized with a value of 1, so the expression (n_rows -1) represents the number of rows that were fetched from a database table in the FOREACH loop. The expression SET_COUNT (n_rows - 1) tells INPUT ARRAY WITHOUT DEFAULTS how many program records containing row values from the database are in the program array, so it can determine how to control the screen array.

```
CALL SET_COUNT(n_rows - 1)
INPUT ARRAY pa_items WITHOUT DEFAULTS
    FROM sa_items.*
```

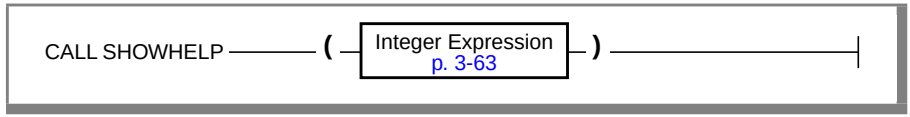
If no INPUT ARRAY statement has been executed, and you do not call the SET_COUNT ( ) function, the DISPLAY ARRAY or INPUT ARRAY WITHOUT DEFAULTS statement displays no records.

References

ARR_COUNT( ), ARR_CURR( )

SHOWHELP( )

The SHOWHELP( ) function displays a runtime help message, corresponding to its specified SMALLINT argument, from the current help file.



Usage

The argument of SHOWHELP( ) identifies the number of a message in the current help file that was specified in the most recently executed HELP FILE clause of the OPTIONS statement. For details of how to specify the current help file, see "The HELP FILE Option" on page 4-299.

The Help Menu

SHOWHELP( ) opens the Help window (as described in "The Help Window" on page 2-30) and displays the first (or only) page of the help message text below a ring menu of help options. This menu is called the **Help** menu.

If the help message is too long to fit on one page, the **Screen** option of the **Help** menu can display the next page of the message. The **Resume** option closes the Help window and returns focus to the 4GL screen.

The Help File That SHOWHELP( ) Displays

To create a help file, you must use a text editor to create an ASCII file of help messages, each identified by a message number. The message number must be an literal integer in the range from -2,147,483,647 to +2,147,483,647 and must be prefixed by a period (.) as the first character on the line containing the number. No sign is required, but message numbers must be unique within the file. Just as in other literal integers, no decimal points, commas, or other separators are allowed. The NEWLINE character (or a NEWLINE RETURN pair) must terminate each message number.

GLS

In nondefault locales, the help file can also contain printable non-ASCII characters from the code set of the locale. ♦

The help message follows the message number on the next line. It can include any printable ASCII characters, except that a line cannot begin with a period. The text of the help message should contain information useful to the user in the context where SHOWHELP( ) is called. The message is terminated by the next message number or by the end of the file.

You must then use the **mkmessage** utility to create a runtime version of the help file that users can view. See the description of the **mkmessage** utility in Appendix B for details of how to compile help files. Here is a simple example of an ASCII help file for use with SHOWHELP( ):

```
.100
You have pressed the Help key of the Megacrunch Application.
Unfortunately, all of our operators are busy at this time.
Perhaps your supervisor can tell you what to do next..200
Press CONTROL-ALT-DEL to exit from this program.
```

GLS

Help messages should be in the language of the intended user of the application. For applications that will be run in different locales, this might require translating the help messages into several languages. At runtime, the compiled help message files must exist in an appropriate subdirectory of **$INFORMIXDIR/msg** and be referenced by the **DBLANG** variable. (See also Appendix E, "Developing Applications with Global Language Support.") ♦

In interactive statements like CONSTRUCT, INPUT, INPUT ARRAY, PROMPT, and the COMMAND clause of a MENU statement, the effect of SHOWHELP( ) resembles that of the Help key. The Help key, however, displays only the message specified in the current HELP clause. The following example uses INFIELD( ) with SHOWHELP( ) to display field-dependent help messages:

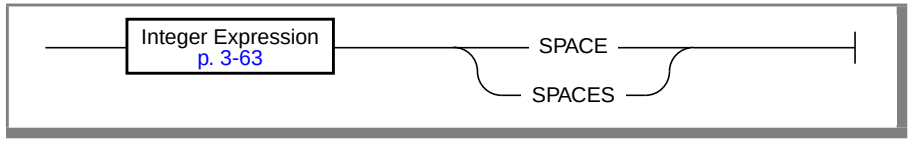
```
INPUT ARRAY gr_equip.* FROM sa_equip.*
    ON KEY(CONTROL-B)
        CASE
            WHEN INFIELD(part_num)
                CALL SHOWHELP(301)
            WHEN INFIELD(part_name)
             CALL SHOWHELP(302)
            WHEN INFIELD(supplier)
                CALL SHOWHELP(303)
            ...
        END CASE
END INPUT
```

Reference

INFIELD( )

SPACE

The SPACE operator returns a string of a specified length, containing only blank (ASCII 32) characters. The keyword SPACES is a synonym for SPACE.



Usage

This operator returns a blank string of a length corresponding to its positive integer argument, specifying a relative offset. The returned value is identical to a quoted string that contains the same number of blank spaces.

In a PRINT statement in the FORMAT section of a report definition, SPACE advances the character position by the specified number of characters.

The following statements from a fragment of a report definition use the SPACE operator to accomplish several tasks:

- To separate variables within two PRINT statements
- To concatenate six blank spaces to the string "=ZIP"
- To print the resulting string after the value of the variable **zipcode**:

```
FORMAT
  ON EVERY ROW
     LET mystring = (6 SPACES), "=ZIP"
     PRINT fname, 2 SPACES, lname
     PRINT company
     PRINT address1
     PRINT city, ", " , state, 2 SPACES, zipcode, mystring
```

In a DISPLAY statement, the SPACE operator inserts the specified number of blank characters into the output.

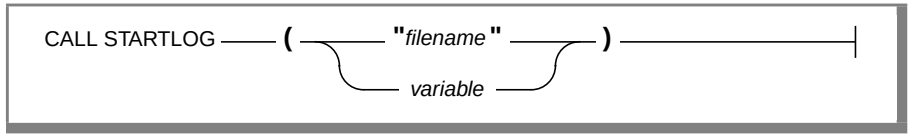
Outside PRINT statements, the SPACE (or SPACES) keyword and its operand must appear within parentheses, as in the LET statement of the previous example.

References

LINENO, PAGENO

STARTLOG( )

The STARTLOG( ) function opens an error log file.



Element	Description
filename	is a quoted string to specify a filename (and optional pathname and file extension) of the error log file.
variable	is a variable of type CHAR or VARCHAR that contains a filename (and optional pathname and file extension) of the error log file.

Usage

The following is a typical sequence to implement error logging:

1. Call STARTLOG( ) in the MAIN program block to open or create an error log file.
2. Use a LET statement with ERR_GET(**status**) to retrieve the error text and to assign this value to a program variable.
3. Use ERRORLOG( ) to make an entry into the error log file.

The last two steps are not needed if you are satisfied with the error records that are automatically produced after STARTLOG( ) has been invoked. After STARTLOG( ) has been invoked, a record of every subsequent error that occurs during the execution of your program is written to the error log file.

The default format of an error record consists of the date, time, source-module name and line number, error number, and error message. If you invoke the STARTLOG( ) function, the format of the error records that 4GL appends to the error log file after each subsequent error are as follows:

```
Date: 03/06/99   Time: 12:20:20
Program error at "stock_one.4gl", line number 89.
SQL statement error number -239.
Could not insert new row - duplicate value in a UNIQUE INDEX
column.
SYSTEM error number -100
ISAM error:  duplicate value for a record with unique key.
```

You can also write your own messages in the error log file by using the ERRORLOG( ) function. For details, see "ERRORLOG( )" on page 5-65.

With other 4GL features, the STARTLOG( ), ERR_GET( ), and ERRORLOG( ) functions can be used for *instrumenting* a program, to track how the program is used. This use is not only valuable for improving the program but also for recording work habits and detecting attempts to breach security. Example 25 in *INFORMIX-4GL by Example* contains an example of this type of functionality.

Unless you specify another option, WHENEVER ERROR CONTINUE is the default error-handling action when a runtime error condition is detected. The WHENEVER ERROR CONTINUE compiler directive can prevent the first SQL error from terminating program execution.

Specifying the Error Log File

If the argument of STARTLOG( ) is not the name of an existing file, STARTLOG( ) creates one. If the file already exists, STARTLOG( ) opens it and positions the file pointer so that subsequent error messages can be appended to this file. The following program fragment invokes STARTLOG( ), specifying the name of the error log file in a quoted string that includes a pathname and a file extension. The function definition includes a call to the built-in ERRORLOG( ) function, which adds a message to the error log file.

```
CALL STARTLOG("/usr/arik/error.log")
...
FUNCTION start_menu()
CALL ERRORLOG("Entering start_menu function")
...
END FUNCTION
```

In this example, text written to the error log file merely shows that control of program execution has passed to the **start_menu( )** function rather than indicating that any error has been issued.

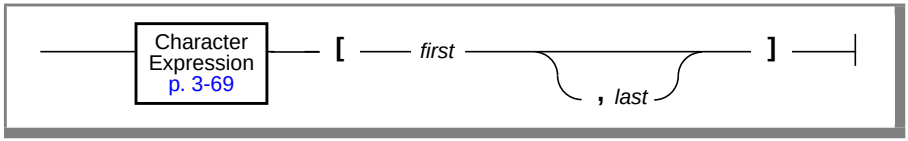
For portable programs, the filename should be a variable rather than a literal string. As in other filename specifications, any literal backslash (\ ) that is required as a pathname separator must be entered as two backslashes.

References

ERRORLOG( ), ERR_GET( ), ERR_PRINT( ), ERR_QUIT( )

Substring ([ ]) Operator

The *substring operator* ([ ]) specifies a substring of the value returned by a character expression.



Element	Description
first and *last*	are 1-based positions of the first and last bytes (respectively) of a substring within the string returned by the left-hand operand.

Usage

The brackets ([ ]) are required. If *last* is not specified, the single byte in the *first* position is returned. If *last* is specified, a comma must separate it from *first*, and a substring is returned whose first byte is *first* and whose last byte is *last*, including any intervening bytes.

The integer expressions *first* and *last* must return values greater than zero but in the range $1 \leq first \leq length$, where *length* is the length of the string returned by the character expression, and $first \leq last \leq length$. For example:

```
DEFINE diamond, spade CHAR[5], club ARRAY [3,4,5] OF CHAR[5]
LET spade =  "heart"
LET club[2,2,2] =  "heart"
LET diamond = spade[3,5]
```

Here the last statement assigns the value "art" to the variable **diamond**, using a three-character substring of the string value in **spade**. If the substring consists of a single character, the last term is not required. For example, if you modified the previous program fragment to include the following statement, the LET statement assigns the value a to variable **diamond**, from the third character of the "heart" string value in **spade**:

```
LET diamond = spade[3]
```

Expressions with character arrays as operands can specify substrings of an individual array element:

```
LET diamond = club[2,2,2] [2,4]
```

In the context of the previous program fragments, this assigns to **diamond** the substring "ear" from the value ("heart") of array element **club[2,2,2]**.

Invalid Operands in Substring Expressions

Be careful to avoid specifying invalid operands for the substring ([ ]) operator, as in the following cases:

- When *first* has a zero or negative value
- When *first* is larger than *last*
- When first or *last* cannot be converted to an integer value
- When *last* has a value greater than the number of bytes returned by the left-hand character expression
- When the left-hand expression is a CHAR or VARCHAR variable (or ARRAY element, or RECORD member) of a declared size less than *last*
- When the left-hand character expression returns an empty string

Invalid operands can produce runtime error -1332. If you are using the RDS version of 4GL, the resulting error message also reports the filename of the **.4gl** source-code module in which the error was detected and the line number within that module of the substring expression that caused the error.

If you have compiled your source code to C, however, rather than to p-code, no module name or line number is provided in the error message. This situation can make it more difficult for developers who do not have RDS to locate an invalid substring specification.

GLS

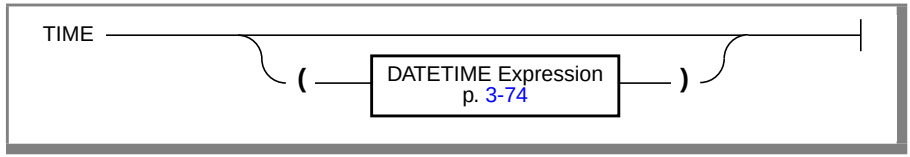
The substring operator is byte based. In East Asian locales that support multibyte characters, 4GL automatically replaces any partial characters that this operator attempts to create with single-byte white-space characters so that no partial character is returned. Informix database servers that support GLS provide functions that are based on a count of logical characters rather than on the number of bytes. ♦

Reference

CLIPPED

TIME

The TIME operator converts the time-of-day portion of its DATETIME operand to a character string. If you supply no operand, TIME reads the system clock and returns a character string value representing the current time of day.



Usage

TIME returns a character string that represents the time-of-day portion of its DATETIME operand in the format *hh:mi:ss*, based on a 24-hour clock. (Here *hh* represents the *hour*, *mi* the *minute*, and *ss* the *second* as 2-digit strings, with colons as separators.) If you do not supply an operand, TIME returns a character string that represents the current time in the format *hh:mi:ss*, based on a 24-hour clock.

In the following program fragment, the value returned by TIME is assigned to the **p_time** variable and displayed:

```
DEFINE p_time char(15)
LET p_time = TIME

DISPLAY "The time is ", p_time
```

If this code were executed half an hour before midnight, the previous DISPLAY statement would produce output in the following format:

```
The time is 23:30:00
```

Like the values returned by the CURRENT, DATE, DAY, MONTH, TODAY, WEEKDAY, and YEAR operators, the value that TIME returns is sensitive to the time of execution and to the accuracy of the system clock-calendar.

References

CURRENT, DATE, DAY, MONTH, TODAY, WEEKDAY, YEAR

TODAY

The TODAY operator reads the system clock and returns a DATE value that represents the current calendar date.

Usage

TODAY can return the current date in situations where the time of day (which CURRENT or TIME supplies) is not necessary. Like the CURRENT, DATE, DAY, MONTH, TIME, WEEKDAY, and YEAR operators, TODAY is sensitive to the time of execution and to the accuracy of the system clock-calendar. The following example uses TODAY in a REPORT definition:

```
SKIP 1 LINE
PRINT COLUMN 15, "FROM: ", begin_date USING "mm/dd/yy",
    COLUMN 35, "TO: ", end_date USING "mm/dd/yy"
PRINT COLUMN 15, "Report run date:  ",
    TODAY USING "mmm dd, yyyy"
SKIP 2 LINES
PRINT COLUMN 2, "ORDER DATE", COLUMN 15, "COMPANY",
    COLUMN 35, "NAME", COLUMN 57, "NUMBER",
    COLUMN 65, "AMOUNT"
```

TODAY is useful in setting defaults and initial values in form fields. The next code fragment initializes a field with the current date if the field is empty. This initialization takes place before the user enters data into the field:

```
INPUT gr_payord.paid_date FROM a_date
   BEFORE FIELD a_date
      IF gr_payord.paid_date IS NULL THEN
         LET gr_payord.paid_date = TODAY
      END IF
```

GLS

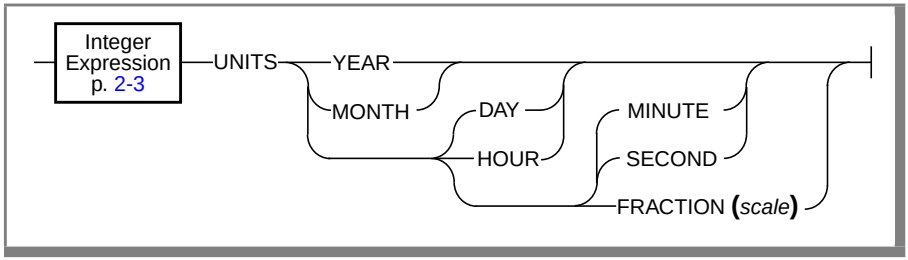
4GL can display language-specific month-name and day-name abbreviations if appropriate files exist in a subdirectory of **$INFORMIXDIR/msg** and they are referenced by the **DBLANG** variable. For example, the weekday portion of a date in a Spanish locale can translate *Saturday* to the abbreviation Sab, which stands for *Sabado* (the Spanish word for *Saturday*). ♦

References

CURRENT, DATE, DAY, MONTH, TIME, WEEKDAY, YEAR

UNITS

The UNITS operator converts an integer expression to an INTERVAL value, expressed in a single unit of time that you specify after the UNITS keyword.



Element	Description
scale	is a literal integer, greater than zero but less than 6.

Usage

The UNITS operator returns an INTERVAL value for a single unit of time, such as DAY TO DAY, YEAR TO YEAR, or HOUR TO HOUR. If you substitute a number expression for the integer operand, any fractional part of the returned value is discarded before the UNITS operator is applied.

UNITS has a higher precedence than any arithmetic or Boolean operator of 4GL. Any left-hand arithmetic operand that includes the UNITS operator must be enclosed within parentheses. The next example specifies a starting time for a meeting (DATETIME value) and a value for the duration of the meeting, which the program has already converted to a whole number of minutes (SMALLINT). The program calculates when the meeting will end (DATETIME value). UNITS in this case allows you to add the SMALLINT value to the DATETIME value and get a new DATETIME value.

```
LET end_time = (meeting_length UNITS MINUTE) + start_time
```

Because the difference between two DATE values is an integer count of days rather than an INTERVAL data type, you might want to use the UNITS operator to convert such differences explicitly to INTERVAL values:

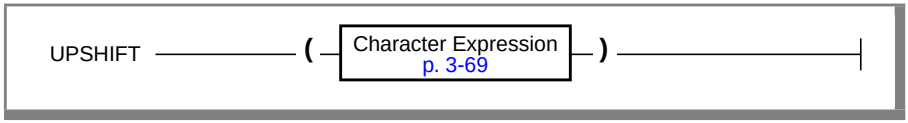
```
LET lateness = (date_due - TODAY) UNITS DAYS
```

Arithmetic operations with UNITS can return an invalid date. For example, the expression `(1 UNITS MONTH) + DATETIME (2001-1 31) YEAR TO DAY` returns February 31, 2001, and also a runtime error:

```
-1267: The result of a datetime computation is out of
range.
```

UPSHIFT( )

The UPSHIFT( ) function takes a character-string argument and returns a string in which any lowercase letters are converted to uppercase letters.



Usage

The UPSHIFT( ) function is most often used to regularize data; for example, to prevent the state abbreviation VA, Va, or va from resulting in different values if these abbreviations were logically equivalent in the context of your application.

You can use the UPSHIFT( ) function in an expression where a character string is valid, in DISPLAY and PRINT statements, and in assignment statements. (See also "UPSHIFT" on page 6-64.)

Non-alphabetic and uppercase characters are not altered by UPSHIFT( ). The maximum data length of the argument (and of the returned character string value) is 32,766 bytes.

The following example demonstrates a function that was written to merge two privilege strings. Its output preserves letters in preference to hyphens (privileges over lack of privilege) and uppercase letters in preference to lowercase (privileges WITH GRANT OPTION over those without).

```
FUNCTION merge_auth(oldauth, newauth)
   DEFINE oldauth, newauth LIKE systabauth.tabauth, k SMALLINT
   FOR k = 1 TO LENGTH(oldauth)
      IF (oldauth[k] = "-")   -- no privilege in this position
         OR (UPSHIFT(oldauth[k]) = newauth[k])
                              -- new is "with grant option"
      THEN LET oldauth[k] = newauth[k])
      END IF
   END FOR
   RETURN oldauth
END FUNCTION
```

In the next example, the CHAR variables **u_str** and **str** are equivalent, except that **u_str** substitutes uppercase letters for any lowercase letters in **str**:

```
LET u_str = UPSHIFT(str)
```

GLS

The results of conversion between uppercase and lowercase letters are based on the locale files, which specify the relationship between corresponding pairs of uppercase and lowercase letters. If the locale files do not provide this information, no case conversion occurs.

UPSHIFT( ) has no effect on non-English characters in most multibyte locales.

In multibyte locales, UPSHIFT( ) and DOWNSHIFT( ) treat the first partial (or otherwise invalid) character in the argument as if it terminated the string. For example, suppose that b is an invalid character. The following expression would return the character string "ABCD EF " with any single-byte or multibyte white-space characters that immediately precede the first invalid character being included in the returned value, rather than being discarded:

```
UPSHIFT(ABCD ef    bXYZ)
♦
```

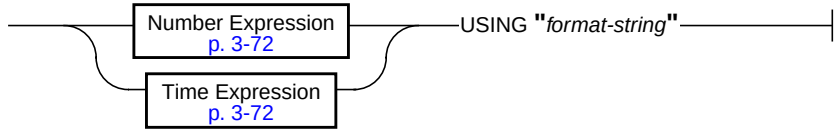
See also the UPSHIFT field attribute in Chapter 6, “Screen Forms.”

Reference

DOWNSHIFT( )

USING

The USING operator specifies a character-string format for a number, MONEY, or DATE operand and returns the formatted value.



Element	Description
format-string	is a quoted string that specifies how to format the returned character string from the number or time expression.

Usage

With a number or MONEY operand, you can use the USING operator to align decimal points or currency symbols, to right- or left-align numbers, to put negative numbers in parentheses, and to perform other formatting tasks. USING can also convert a DATE operand to a variety of formats.

USING is typically used in DISPLAY and PRINT statements, but you can also use it with LET to assign the formatted value to a character variable. If a value is too large for the field, 4GL fills it with asterisks (*) to indicate an overflow.

For information on symbols that the USING operator recognizes in *format-string*, see "The USING Formatting Symbols for Number Values" on page 5-124 (for number values) and "Formatting DATE Values" on page 5-127 (for DATE values).

Formatting Number Expressions

The USING operator takes precedence over the **DBMONEY** or **DBFORMAT** environment variables and is required to display the *thousands* separator of **DBFORMAT**. When 4GL displays a number value, it follows these rules:

- 4GL displays the leading currency symbol (as set by **DBFORMAT** or **DBMONEY**) for MONEY values. (But if the FORMAT attribute also specifies a leading currency symbol, 4GL displays that symbol for other data types.)
- 4GL omits the thousands separators, unless they are specified by a FORMAT attribute or by the USING operator.
- 4GL displays the decimal separator, except for INT or SMALLINT values.
- 4GL displays the trailing currency symbol (as set by **DBFORMAT** or **DBMONEY**) for MONEY values unless you specify a FORMAT attribute or the USING operator. In this case, the user cannot enter a trailing currency symbol, and 4GL does not display it.

The USING Formatting Symbols for Number Values

The *format-string* value can include the following characters.

Character	Description
*	This character fills with asterisks any positions in the display field that would otherwise be blank.
&	This fills with zeros any positions that would otherwise be blank.
#	This does not change any blank positions in the display field. You can use it to specify a maximum width for a field.
<	This causes numbers in the field to be left aligned.
,	This character is a literal. USING displays it as a comma (but displays no comma unless there is a number to the left of it).
.	This character is a literal. USING displays it as a period. You can only have one decimal point (or period) in a number format string.

(1 of 2)

Character	Description
-	This character is a literal. USING displays it as a minus sign when the expression is less than zero and otherwise as a blank. When you group several minus signs in a row, a single minus sign floats immediately to the left of the number being printed.
+	This character is a literal. USING displays it as a plus sign when the expression is greater than or equal to zero and as a minus sign when it is less than zero. When you group several plus signs in a row, a single plus sign floats immediately to the left of the displayed number.
$	The dollar ($) sign is a placeholder for the *front* specification of **DBMONEY** or **DBFORMAT**. (The *back* specification of **DBMONEY** or **DBFORMAT** has no effect if USING is applied to the data value.) When you group several consecutive dollar signs, a single *front* currency symbol floats immediately to the left of the number being printed.
(	This literal character is displayed as a left parenthesis before a negative number. It is the *accounting parenthesis* that is used in place of a minus sign to indicate a negative number. Consecutive left parentheses display a single left parenthesis to the left of the number being printed.
)	For the accounting parenthesis that is used in place of a minus sign to indicate a negative number, one of these characters generally closes a format string that begins with a left parenthesis.

(2 of 2)

The minus sign (-), plus sign (+), parentheses, and dollar sign ($) *float*, meaning that when you specify multiple leading occurrences of one of these characters, 4GL displays only a single character immediately to the left of the number that is being displayed. Any other character in *format-string* is interpreted as a literal.

Important: *These characters are not identical to the formatting characters that you can specify in the format-strings of the FORMAT or PICTURE field attributes, described in Chapter 6, "Screen Forms."*

For examples of using format strings for number expressions, see "Examples of the USING Operator" on page 5-129. Because format strings interact with data to produce visual effects, you might find that the examples are easier to follow than the descriptions on the previous page of USING format string characters.

The following example prints a MONEY value using a format string that allows values up to $9,999,999.99 to be formatted correctly:

```
DEFINE mon_val MONEY(8,2)
LET mon_val = 23485.23
DISPLAY "The current balance is ", mon_val
    USING "$#,###,##&.&&"
```

Executing this DISPLAY statement (with the value of **mon_val** set to 23485.23) produces the following output:

```
The current balance is $   23,485.23
```

The format string in this example specifies the currency symbol.

The previous example also uses the # and & fill characters. The # character provides blank fill for unused character positions, while the & character provides zero filling. This format ensures that even if the number is zero, any positions marked with & appear as zero, not blank.

Dollar signs can be used instead of # characters, as in the following statement:

```
DISPLAY "The current balance is ",mon_val
    USING "$$,$$$,$$&.&&"
```

In this example, the currency symbol floats with the size of the number so that it appears immediately to the left of the most significant digit in the display. This example would produce the following formatted output, if the value of the **mon_val** variable were 23485.23:

```
The current balance is    $23,485.23
```

By default, 4GL displays numbers right aligned. You can use the < symbol in a USING format string to override this default. For example, specifying

```
DISPLAY "The current balance is ",mon_val
    USING "$<<,<<<,<<&.&&"
```

produces the following output when the value of **mon_val** is 23485.23:

```
The current balance is $23,485.23
```

Formatting DATE Values

When you use it to format a DATE value, USING takes precedence over any **DBDATE** or **GL_DATE** environment variable settings. The *format-string* value for a date can be a combination of the characters m, d, and y.

Symbols	Resulting Time Unit in Formatted DATE Display
dd	Day of the month as a 2-digit number (01 through 31 or less)
ddd	Day of the week as a 3-letter abbreviation (Sun through Sat)
mm	Month as a 2-digit number (01 through 12)
mmm	Month as a 3-letter abbreviation (Jan through Dec)
yy	Year as a 2-digit number (the trailing digits, 00 through 99)
yyyy	Year as a 4-digit number (0001 through 9999)

Here lowercase is required; uppercase D, M, or Y cannot be substituted.

Any other characters within a USING formatting mask for DATE values are interpreted as literals.

The following examples show valid *format-string* masks for December 25, 1999, and the resulting display for the default U.S. English locale.

Format String	Formatted Result
"mmddyy"	122599
"ddmmyy"	251299
"yymmdd"	991225
"yy/mm/dd"	99/12/25
"yy mm dd"	99 12 25
"yy-mm-dd"	99-12-25
"mmm. dd, yyyy"	Dec. 25, 1999
"mmm dd yyyy"	Dec 25 1999

(1 of 2)

Format String	Formatted Result
"yyyy dd mm"	1999 25 12
"mmm dd yyyy"	Dec 25 1999
"ddd, mmm. dd, yyyy"	Sat, Dec. 25, 1999
"(ddd) mmm. dd, yyyy"	(Sat) Dec. 25, 1999

(2 of 2)

The following example is from a REPORT program block:

```
ON LAST ROW
  SKIP 2 LINES
  PRINT "Number of customers in ", state, " are ",
        COUNT(*) USING "<<<<<" PAGE TRAILER
  PRINT COLUMN 35, "page ", PAGENO USING "<<<<"
```

The following REPORT fragment illustrates several different formats:

```
SKIP 1 LINE
PRINT COLUMN 15, "FROM:  ", begin_date USING "mm/dd/yy",
      COLUMN 35, "TO:  ", end_date USING "mm/dd/yy"
PRINT COLUMN 15, "Report run date:  ", TODAY USING "mmm dd, yyyy"
   SKIP 2 LINES
PRINT COLUMN 2, "ORDER DATE", COLUMN 15, "COMPANY",
      COLUMN 35, "NAME", COLUMN 57, "NUMBER", COLUMN 65, "AMOUNT"
BEFORE GROUP OF days
   SKIP 2 LINES
AFTER GROUP OF number
   PRINT COLUMN 2, order_date, COLUMN 15, company CLIPPED,
         COLUMN 35, fname CLIPPED, 1 SPACE, lname CLIPPED,
         COLUMN 55, number USING "####",
         COLUMN 60, GROUP SUM(total_price)
              USING "$$,$$$,$$$.&&"
AFTER GROUP OF days
   SKIP 1 LINE
   PRINT COLUMN 21, "Total amount ordered for the day:  ",
         GROUP SUM(total_price) USING "$$$$,$$$,$$$.&&"
   SKIP 1 LINE
   PRINT COLUMN 15,
         "================================================="
ON LAST ROW
   SKIP 1 LINE
   PRINT COLUMN 15,
         "================================================="
   SKIP 2 LINES
   PRINT "Total Amount of orders:  ", SUM(total_price)
         USING "$$$$,$$$,$$$.&&"
PAGE TRAILER
   PRINT COLUMN 28, PAGENO USING "page <<<<"
```

GLS

In nondefault locales, the NUMERIC and MONETARY categories in the locale files affect how the format string of the USING operator is interpreted for formatting number and currency data values.

In *format string*, the period (.) is not a literal character but a placeholder for the decimal separator specified by environment variables. Likewise, the comma is a placeholder for the thousands separator specified by environment variables. The dollar sign ($) is a placeholder for the leading currency symbol. The @ symbol is a placeholder for the trailing currency symbol. Thus, the format string $#,###.## formats the value `1234.56` as `£1,234.56` in a U.K. English locale, but as `f1.234,56` in a French locale. Setting either **DBFORMAT** or **DBMONEY** overrides these locale setting.

The `mmm` and `ddd` specifiers in a format string can display language-specific month-name and day-name abbreviations. This operation requires the installation of appropriate files in a subdirectory of **$INFORMIXDIR/msg** and reference to that subdirectory in the setting of the environment variable **DBLANG**. For example, in a Spanish locale, the `ddd` specifier translates the day Saturday into the day-name abbreviation `Sab`, which stands for *Sabado* (the Spanish word for *Saturday*). For more information on GLS, see Appendix E, "Developing Applications with Global Language Support." ♦

Examples of the USING Operator

Tables that follow illustrate some of the capabilities of the USING operator with number or currency operands (for the default U.S. English locale). Each table has the following format:

- The first column shows a format string (the left-hand operand).
- The second column shows a data value (the right-hand operand).
- The third column shows the resulting formatted display.
- The fourth column provides a comment (for some rows).

Here the character b in the **Formatted Result** column represents a blank space.

Format String	Data Value	Formatted Result	Comment on Result
"#####"	0	bbbbb	No zero symbol
"&&&&&"	0	00000	
"$$$$$"	0	bbbb$	No zero symbol
"*****"	0	*****	No zero symbol
"<<<<<"	0		(NULL string)
"<<<,<<<"	12345	12,345	
"<<<,<<<"	1234	1,234	
"<<<,<<<"	123	123	
"<<<,<<<"	12	12	
"##,###"	12345	12,345	
"##,###"	1234	b1,234	
"##,###"	123	bbb123	
"##,###"	12	bbbb12	
"##,###"	1	bbbbb1	
"##,###"	-1	bbbbb1	No negative sign
"##,###"	0	bbbbbb	No zero symbol
"&&,&&&"	12345	12,345	
"&&,&&&"	1234	01,234	
"&&,&&&"	123	000123	
"&&,&&&"	12	000012	
"&&,&&&"	1	000001	
"&&,&&&"	-1	000001	No negative sign
"&&,&&&"	0	000000	
"&&,&&&.&&"	12345.67	12,345.67	
"&&,&&&.&&"	1234.56	01,234.56	
"&&,&&&.&&"	123.45	000123.45	
"&&,&&&.&&"	0.01	000000.01	
"$$,$$$"	12345	******	(Overflow)
"$$,$$$"	1234	$1,234	
"$$,$$$"	123	bb$123	
"$$,$$$"	12	bbb$12	
"$$,$$$"	1	bbbb$1	
"$$,$$$"	0	bbbbb$	No zero symbol
"**,***"	12345	12,345	
"**,***"	1234	*1,234	
"**,***"	123	***123	
"**,***"	12	****12	
"**,***"	1	*****1	
"**,***"	0	******	No zero symbol

Here the character b in the **Formatted Result** column represents a blank space.

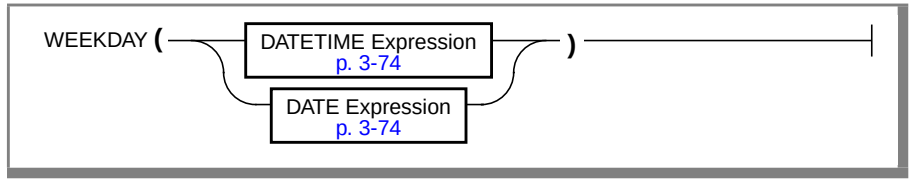
Format String	Data Value	Formatted Result	Comment on Result
"##,###.##"	12345.67	12,345.67	
"##,###.##"	1234.56	b1,234.56	
"##,###.##"	123.45	bbb123.45	
"##,###.##"	12.34	bbbb12.34	
"##,###.##"	1.23	bbbbb1.23	
"##,###.##"	0.12	bbbbb0.12	
"##,###.##"	0.01	bbbbbb.01	No leading zero
"##,###.##"	-0.01	bbbbbb.01	No negative sign
"##,###.##"	-1	bbbbb1.00	No negative sign
"$$,$$$.$$"	12345.67	*********	(*overflow*)
"$$,$$$.$$"	1234.56	$1,234.56	
"$$,$$$.##"	0.00	$.00	No leading zero
"$$,$$$.##"	1234.00	$1,234.00	
"$$,$$$.&&"	0.00	$.00	No leading zero
"$$,$$$.&&"	1234.00	$1,234.00	
"-$$$,$$$.&&"	-12345.67	-$12,345.67	
"-$$$,$$$.&&"	-1234.56	-b$1,234.56	
"-$$$,$$$.&&"	-123.45	-bbb$123.45	
"--$$,$$$.&&"	-12345.67	-$12,345.67	
"--$$,$$$.&&"	-1234.56	-$1,234.56	
"--$$,$$$.&&"	-123.45	-bb$123.45	
"--$$,$$$.&&"	-12.34	-bbb$12.34	
"--$$,$$$.&&"	-1.23	-bbbb$1.23	
"-##,###.##"	-12345.67	-12,345.67	
"-##,###.##"	-123.45	-bbb123.45	
"-##,###.##"	-12.34	-bbbb12.34	
"--#,###.##"	-12.34	-bbb12.34	
"---,###.##"	-12.34	-bb12.34	
"---,-##.##"	-12.34	-12.34	
"---,--#.##"	-1.00	-1.00	
"-##,###.##"	12345.67	12,345.67	
"-##,###.##"	1234.56	1,234.56	
"-##,###.##"	123.45	123.45	
"-##,###.##"	12.34	12.34	
"--#,###.##"	12.34	12.34	
"---,###.##"	12.34	12.34	
"---,-##.##"	12.34	12.34	
"---,---.##"	1.00	1.00	
"---,---.--"	-.01	-.01	
"---,---.&&"	-.01	-.01	

Here the character b in the **Formatted Result** column represents a blank space.

Format String	Data Value	Formatted Result	Comment on Result
"----,--$.&&"	-12345.67	-$12,345.67	
"----,--$.&&"	-1234.56	-$1,234.56	
"----,--$.&&"	-123.45	-$123.45	
"----,--$.&&"	-12.34	-$12.34	
"----,--$.&&"	-1.23	-$1.23	
"----,--$.&&"	-.12	-$.12	
"$***,***.&&"	12345.67	$*12,345.67	
"$***,***.&&"	1234.56	$**1,234.56	
"$***,***.&&"	123.45	$****123.45	
"$***,***.&&"	12.34	$*****12.34	
"$***,***.&&"	1.23	$******1.23	
"$***,***.&&"	.12	$*******.12	
"($$$,$$$.&&)"	-12345.67	($12,345.67)	Accounting parentheses
"($$$,$$$.&&)"	-1234.56	(b$1,234.56)	
"($$$,$$$.&&)"	-123.45	(bbb$123.45)	
"(($$,$$$.&&)"	-12345.67	($12,345.67)	
"(($$,$$$.&&)"	-1234.56	($1,234.56)	
"(($$,$$$.&&)"	-123.45	(bb$123.45)	
"(($$,$$$.&&)"	-12.34	(bbb$12.34)	
"(($$,$$$.&&)"	-1.23	(bbbb$1.23)	
"((((,(($.&&)"	-12345.67	($12,345.67)	
"((((,(($.&&)"	-1234.56	($1,234.56)	
"((((,(($.&&)"	-123.45	($123.45)	
"((((,(($.&&)"	-12.34	($12.34)	
"((((,(($.&&)"	-1.23	($1.23)	
"((((,(($.&&)"	-.12	($.12)	
"($$$,$$$.&&)"	12345.67	$12,345.67	
"($$$,$$$.&&)"	1234.56	$1,234.56	
"($$$,$$$.&&)"	123.45	$123.45	
"(($$,$$$.&&)"	12345.67	$12,345.67	
"(($$,$$$.&&)"	1234.56	$1,234.56	
"(($$,$$$.&&)"	123.45	$123.45	
"(($$,$$$.&&)"	12.34	$12.34	
"(($$,$$$.&&)"	1.23	$1.23	
"((((,(($.&&)"	12345.67	$12,345.67	
"((((,(($.&&)"	1234.56	$1,234.56	
"((((,(($.&&)"	123.45	$123.45	
"((((,(($.&&)"	12.34	$12.34	
"((((,(($.&&)"	1.23	$1.23	
"((((,(($.&&)"	.12	$.12	

WEEKDAY( )

The WEEKDAY( ) operator returns a positive integer, corresponding to the day of the week implied by its DATE or DATETIME operand.



Usage

This operator takes a DATETIME or DATE operand, and returns an integer in the range `0` through `6`. Here `0` represents Sunday, `1` represents Monday, and so on. The following example calls a function that uses WEEKDAY with a CASE statement to assign a three-letter day-of-the-week abbreviation to each date in an array, omitting days that fall on weekends:

```
FOR i = 1 TO 10
   CALL seize_theday(next_day)
      RETURNING day_name, next_day
   LET pa_days[i].dayo_week = day_name
   LET pa_days[i].rdate = next_day
   LET next_day = next_day + 1
END FOR
...
FUNCTION seize_theday(next_day)
   DEFINE
      week_day SMALLINT
      day_name CHAR(3)
      next_day DATE
   LET week_day = WEEKDAY(next_day)
   CASE week_day
         WHEN 1 LET day_name = "Mon"
         WHEN 2 LET day_name = "Tues"
         WHEN 3  LET day_name = "Wed"
         WHEN 4 LET day_name = "Thu"
         WHEN 5 LET day_name = "Fri"
         WHEN 6 LET day_name = "Mon"
                LET next_day = next_day + 2
         WHEN 7 LET day_name = "Mon"
                LET next_day = next_day + 1
      END CASE
   RETURN day_name, next_day
END FUNCTION -- seize_theday
```

This operator is useful for determining the day of the week from dates in recent and future centuries. It should be used with caution, however, for more remote dates, because of disagreements between the old and new calendar systems in some European countries between the invention of the Gregorian calendar in 1582 and the eventual acceptance of that calendar.

For dates thousands of years in the past (for example, the death of Socrates, or the establishment of the Middle Kingdom in ancient Egypt), it is difficult to verify that the sequential count of the seven days of the week has been accurately maintained from antiquity up to the present. Computers that use defective algorithms for calculating leap years might also have difficulties with the weekdays in modern dates after February 28 of the year 2000.

The WEEKDAY( ) operator is among a group of 4GL operators that extract a single time unit value from a DATETIME or DATE value. The following *extraction* operators of 4GL accept a DATETIME or DATE operand.

Operator	Meaning of the Returned Integer
DAY( )	The day of the month
MONTH( )	The month
YEAR( )	The year
WEEKDAY( )	The day of the week

In addition, the DATE( ) operator can extract the date portion of a DATETIME value that has YEAR TO DAY or greater precision, and the TIME operator can extract the time-of-day from a DATETIME expression that has HOUR TO FRACTION precision (or a subset thereof).

The USING operator can also return day-of-the-week information from a DATE operand (as described in "Formatting DATE Values" on page 5-127).

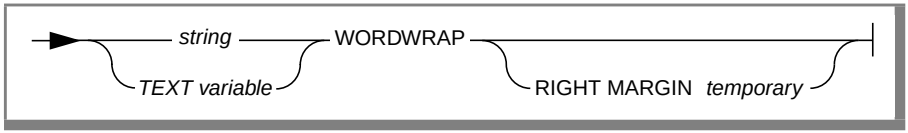
For more information, see the **GL_DATE** environment variable in Appendix D, "Environment Variables."

References

CURRENT, DATE, DAY, MONTH, TIME, TODAY, YEAR

WORDWRAP

The WORDWRAP operator divides a long text string into segments that appear in successive lines of a 4GL report. (This operator can appear only in the PRINT statement in the FORMAT section of a REPORT program block.)



Element	Description
string	is a character expression (as described in "Character Expressions" on page 3-69) to be printed in the output from the report.
temporary	is an integer expression (as described in "Integer Expressions" on page 3-63) whose returned value specifies the absolute position (in characters), counting from the left edge of the page, of a temporary right margin.
TEXT variable	is the name of a 4GL variable of the TEXT data type to be printed in the output from the report.

Usage

The WORDWRAP operator automatically *wraps* successive segments of long character strings onto successive lines of output from a 4GL report. The string value of any expression or TEXT *variable* that is too long to fit between the current character position and the specified or default right margin is divided into segments and displayed between temporary margins:

- The current character position becomes the temporary left margin.
- Unless you specify RIGHT MARGIN *temporary*, the right margin defaults to 132 or to the size from the RIGHT MARGIN clause of the OUTPUT section of the report definition.

These temporary values override the specified or default left and right margins from the OUTPUT section.

After the PRINT statement has executed, any explicit or default margins from the OUTPUT section are restored. For more information, see "PRINT" on page 7-55.

The following PRINT statement specifies a temporary left margin in column 10 and a temporary right margin in column 70 to display the character string that is stored in the 4GL variable called **mynovel**:

```
print column 10, mynovel WORDWRAP RIGHT MARGIN 70
```

Tabs, Line Breaks, and Page Breaks with WORDWRAP

The data string can include printable ASCII characters. It can also include the TAB (ASCII 9), newline character (ASCII 10), and RETURN (ASCII 13) characters that partition the string into *words*, consisting of substrings of other printable characters. Other non-printable characters might cause runtime errors. If the data string cannot fit between the margins of the current line, 4GL breaks the line at a *word* division, padding the line with blanks at the right.

From left to right, 4GL expands any TAB character to enough blank spaces to reach the next tab stop. By default, tab stops are in every eighth column, beginning at the left-hand edge of the page. If the next tab stop or a string of blank characters extends beyond the right margin, 4GL takes these actions:

- Prints blank characters only to the right margin
- Discards any remaining blank characters from the blank string or tab
- Starts a new line at the temporary left margin
- Processes the next word

4GL starts a new line when a word plus the next blank space cannot fit on the current line. If all words are separated by a single space, an even left margin results. 4GL applies the following rules (in descending order of precedence) to the portion of the data string within the right margin:

- Break at any newline character, RETURN, or newline character and RETURN pair.
- Break at the last blank (ASCII 32) or TAB character before the right margin.
- Break at the right margin if no character farther to the left is a blank, RETURN, TAB, or newline character.

4GL maintains page discipline with the WORDWRAP operator. If the character string or TEXT value operand is too long for the current page of report output, 4GL executes the statements in any PAGE TRAILER and PAGE HEADER control blocks before continuing output onto a new page.

Tip: *The WORDWRAP keyword can also specify a field attribute that supports data display and data entry in a multiple-segment field of a 4GL form. For more information, see* *"WORDWRAP" on page 6-67*.

Kinsoku Processing

GLS

In Japanese locales, a suitable break can also be made between the Japanese characters. However, certain characters must not begin a new line, and some characters must not end a line. This convention creates the need for *kinsoku processing*, whose purpose is to format the line properly, without any prohibited character at the beginning or ending of a line.

4GL reports use the wrap-down method for WORDWRAP and kinsoku processing. The wrap-down method reads the last character in the line and forces that character down to the next line if that character is prohibited from ending a line. A character that precedes another that is prohibited from beginning a line can also wrap down to the next line.

Characters that are prohibited from beginning or ending a line must be listed in the locale files. 4GL tests for prohibited characters at the beginning and ending of a line, testing the first and last visible characters.

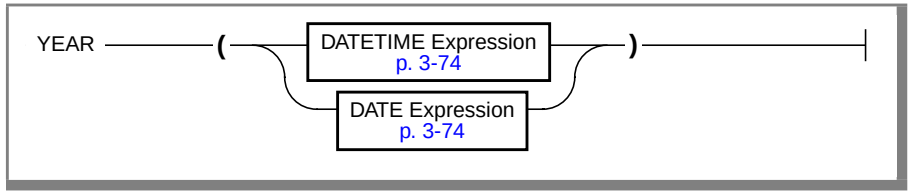
The kinsoku processing only happens once for each line. That is, no further kinsoku processing occurs even if prohibited characters are still on the same line after the first kinsoku processing takes place. ♦

References

CLIPPED, SPACES, USING

YEAR( )

The YEAR( ) operator returns an integer corresponding to the *year* portion of its DATE or DATETIME operand.



Usage

The YEAR( ) operator returns all the digits of the year value (`1999`, not `99`). (The second example that follows illustrates how to obtain a two-digit value like `99` from a four-digit year like `1999`.)

The following example extracts the current year and stores the value in an integer variable:

```
LET y_var = YEAR(TODAY)
```

You can produce a two-digit year abbreviation by using the MOD (modulus) operator:

```
LET birth_yr = (YEAR(birth_date)) MOD 100
```

In the right-hand expression, the MOD operator yields the year modulo 100, the remainder when the value representing the actual year is divided by 100.

For example, if the value of the DATE variable **birth_date** is `09-16-1953`, the YEAR( ) operator extracts the value `1953`, and the following expression returns `53`:

```
1953 MOD 100
```

That value is then assigned to the INT variable birth_yr.

References

CURRENT, DATE, DAY, MONTH, TIME, TODAY, WEEKDAY

Chapter

6

Screen Forms

In This Chapter

A *screen form* is a visual display that can support input and output tasks in an INFORMIX-4GL application. Before your 4GL program can use a screen form, you must first create a *form specification file*. This source file describes the logical format of the screen form and how to display data values in the form at runtime. It must be compiled separately from the rest of your source code. The same compiled form can be used by different 4GL programs.

The first part of this chapter describes the structure of a form specification file and the effect and syntax of its components. "Default Attributes" on page 6-80 describes the **syscolval** and **syscolatt** tables that can specify formats, validation rules, and default data values for fields. (For information on using the **upscol** utility to specify values in these tables, see Appendix B, "INFORMIX-4GL Utility Programs.") Additional sections in this chapter describe how to compile 4GL forms and how to use forms designed for PERFORM, the screen transaction processor of INFORMIX-SQL.

4GL Forms

This section describes form drivers, which control the display of a form, and form fields, including their behavior and how to navigate among them.

Form Drivers

To work with a compiled screen form, the application requires a *form driver*, a logical set of 4GL statements that control the display of the form, bind form fields to 4GL variables, and respond to actions by the user in fields.

The form driver can include 4GL screen interaction and data manipulation statements to enter, retrieve, modify, or delete data in the database. The emphasis of this chapter, however, is on how to create the form specification file, rather than on how to design and implement the form driver.

Regardless of how you define them, there is no implicit relationship between the values of program variables, form fields, and database columns. Even, for example, if you declare a 4GL variable **lname** LIKE **customer.lname**, the changes that you make to the variable do not imply any change in the column value. Functional relationships among these entities must be specified in the logic of your form driver, typically through screen interaction statements of 4GL, and through data manipulation statements of SQL. After the user presses the Accept key to terminate an INPUT ARRAY statement, for example, the form driver can use the INSERT statement to modify the database.

Similarly, a 4GL form is only a template. FORM4GL reads the system catalog at compile time to obtain the names and data types of any columns that are referenced in the form specification file. After compilation, however, the form loses its connection to the database. It can no longer distinguish the name of a table or view from the name of a screen record.

It is up to you, the programmer, to determine what data a form displays and what to do with data values that the user enters into the fields of a form. You must indicate the binding explicitly in any 4GL statement that connects 4GL variables to screen forms or to database columns. The following statements, for example, take input from a 4GL form and insert the entered value from the form into the database. (Here the @ sign in the INSERT statement tells 4GL that the first **lname** is the SQL identifier of a database column.)

```
INPUT lname FROM customer.lname
INSERT INTO customer (@lname) VALUES (lname)
```

You can use interactive 4GL statements such as OPEN FORM, OPEN WINDOW, INPUT, DISPLAY FORM, CLEAR FORM, and CONSTRUCT in the form driver to support data entry or data display through the 4GL form. Some statements support temporary binding when a program variable and a screen field have identical names. (See the individual 4GL statement descriptions in Chapter 4, "INFORMIX-4GL Statements," for the appropriate syntax.) For example, the following statement could replace the previous INPUT statement:

```
INPUT BY NAME lname
```

For more information about form drivers, refer to *INFORMIX-4GL Concepts and Use*.

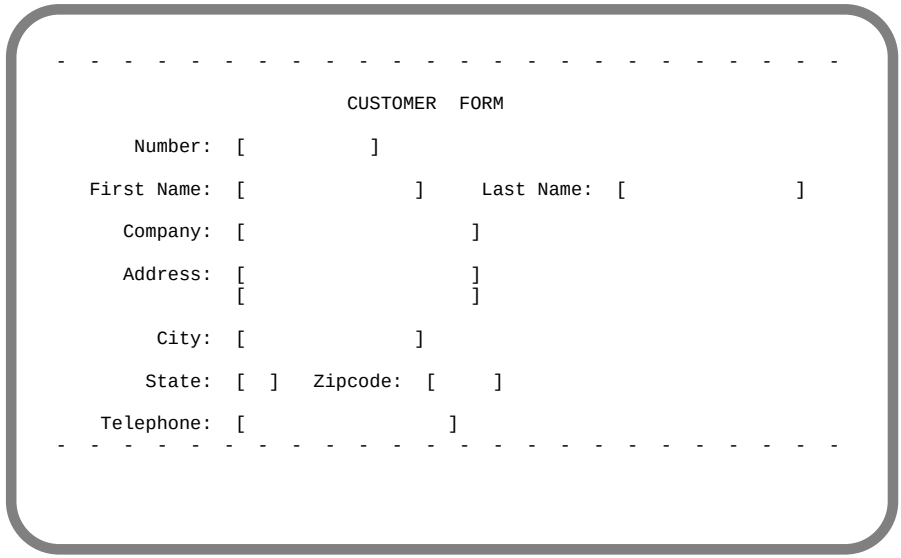
Form Fields

In a form, a *field* (sometimes called a *screen field* or *form field*) is an area where the user of the application can view, enter, and edit data, depending on its description in the form specification and the statements in the form driver. This section discusses the appearance and behavior of form fields in 4GL.

Appearance of Fields

The screen form contains *display fields* bounded by *delimiters*, such as the square brackets shown in Figure 6-1.

Figure 6-1
The customer Form



The currently active form field contains a visual cursor. This *current field* displays what the user types. For details about how to size and position fields in a form, see "Display Fields" on page 6-17. For information on assigning display and validation attributes to fields, see "Field Attribute Syntax" on page 6-33.

GLS

For nondefault locales, screen forms can include non-ASCII characters that the client locale supports. ♦

Navigation Among Form Fields

The order in which the cursor moves from field to field on a screen form is determined by the order in which you list fields in the INPUT statement. At any time before pressing RETURN in the last field, the user can use the arrow keys to move back through the fields and make corrections, or TAB to move to the next field. In CONSTRUCT, INPUT, and INPUT ARRAY statements, the NEXT FIELD clause can override the default order of field traversal.

The AUTONEXT field attribute can move the cursor automatically to the next field when the user has typed enough characters to fill a field. The user can indicate that data entry is complete by pressing the Accept key (typically ESCAPE) in any field. Data can also be entered by pressing RETURN in the last field, if INPUT WRAP was not specified in the OPTIONS statement. If INPUT WRAP was specified, RETURN moves the cursor back to the first field. Specify OPTIONS CURSOR NO WRAP to restore the default behavior.

The FIELD ORDER setting in the OPTIONS statement determines where the arrow keys move the cursor. If FIELD ORDER CONSTRAINED is specified, pressing UP ARROW moves the cursor to the previous field, and pressing DOWN ARROW moves the cursor to the next field. If FIELD ORDER UNCONSTRAINED is specified, pressing UP ARROW moves the cursor to the field above the current cursor position and pressing DOWN ARROW moves the cursor to the field below the current cursor position.

Disabled Form Fields

When a form field is not included in a screen interaction statement like INPUT or CONSTRUCT, or is specified as a NOENTRY field in a form file, it is disabled during execution of that statement. The user cannot move the cursor to the field. If the user attempts to enter the NOENTRY field by using the TAB or arrow keys, the cursor moves to the next field in traversal order, and any appropriate BEFORE and AFTER clauses are executed.

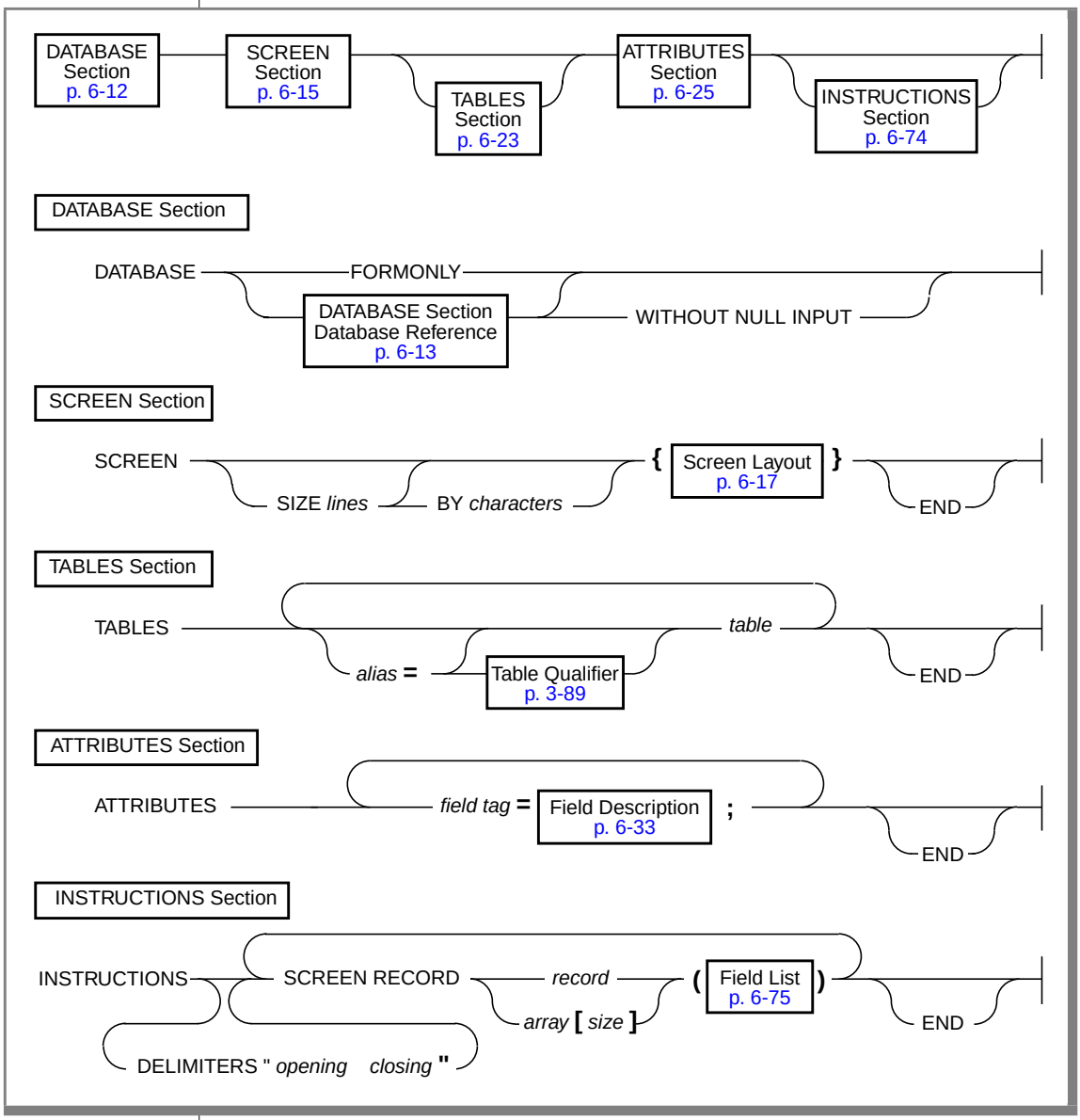
Structure of a Form Specification File

A 4GL form specification file is a source file (with file extension .**per**) that you can create with a text editor or from within the 4GL Programmer's Environment. This file consists of three required sections (DATABASE, SCREEN, and ATTRIBUTES), and it can also include two optional sections (TABLES and INSTRUCTIONS). If present, these five sections must appear in the following order:

- **DATABASE section.** Each form specification file must begin with a DATABASE section identifying the database (if any) on which the form is based. This can be any database that the current database server can access, including a remote database.
- **SCREEN section.** The SCREEN section must appear next, showing the dimensions and the exact layout of the logical elements of the form. You must specify the position of one or more *screen fields* for data entry or display as well as any additional text or ornamental characters.
- **TABLES section.** If it is present, the TABLES section must follow the SCREEN section. This section lists every table or view that is referenced in the ATTRIBUTES section. If a table requires as a qualifier the name of an *owner* or of a *database*, the TABLES section must also declare an alias for the table.
- **ATTRIBUTES section.** The ATTRIBUTES section describes each field on the form and assigns names to fields. Field descriptions can optionally include *field attribute*s to specify, for example, the appearance, acceptable input values, on-screen comments, and default values for each field.
- **INSTRUCTIONS section.** The INSTRUCTIONS section is optional. It can specify screen arrays and nondefault screen records and field delimiters.

Each section must begin with the keyword for which it is named. After you create a form specification file, you must compile it. The form driver of your 4GL application can then use 4GL variables to transfer information between the database and the fields of the screen form.

This is the syntax of a 4GL form specification.



The next five sections of this chapter identify the keywords and terms that are listed in this diagram and describe their syntax in detail.

The following example illustrates the overall structure of a typical form specification:

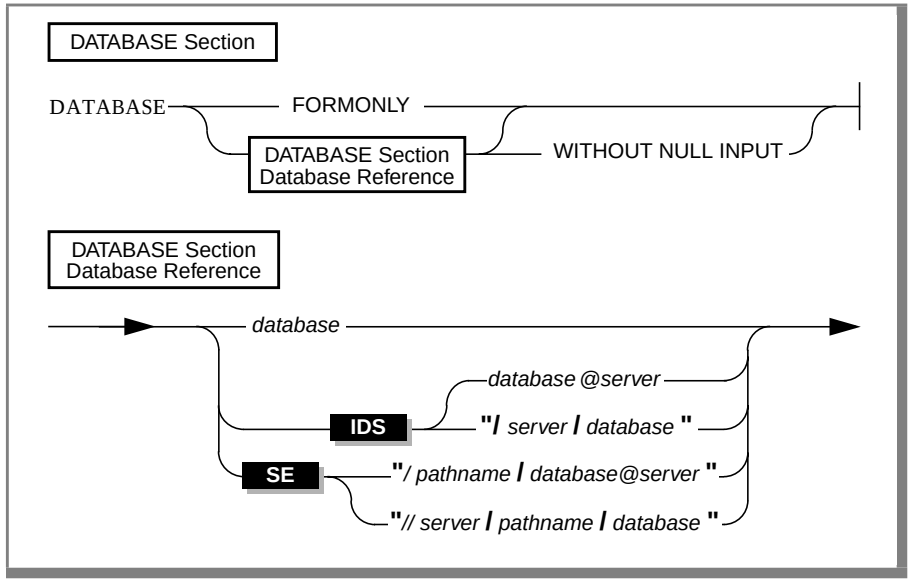
```
DATABASE stores7

SCREEN
{
--------------------------------------------------------------------
CUSTOMER INFORMATION:
Customer Number: [c1         ]             Telephone: [c10          ]
         ...

SHIPPING INFORMATION:
     Customer P.O.: [o20       ]

         Ship Date: [o21       ]          Date Paid: [o22       ]
}

TABLES
customer orders items manufact

ATTRIBUTES
c1 = customer.customer_num
   = orders.customer_num;
c10 = customer.phone, PICTURE = "###-###-####x#####";
         ...
o20 = orders.po_num;
o21 = orders.ship_date;
o22 = orders.paid_date;

INSTRUCTIONS
SCREEN RECORD sc_order[5] (orders.order_date THRU orders.paid_date)
```

In this example, the screen form has been designed to display columns from several tables in the **stores7** demonstration database and includes all five of the required and optional sections that are described in the pages that follow.

This example is incomplete because it omits portions of the SCREEN and ATTRIBUTES sections that describe some of the screen fields. The ellipsis notation (…) in those sections is a typographic device to simplify this illustration rather than a valid specification for the form.

DATABASE Section

The DATABASE section identifies the database, if any, that contains tables or views whose columns are referenced in the form specification file.



Element	Description
database	is the SQL identifier of a database.
pathname	is the path to the parent directory of the **.dbs** directory (for INFORMIX-SE databases only).
server	is the name of the host system where database resides.

Usage

The DATABASE section is required, even if the screen form does not reference any database columns or tables. You can specify only one database.

When compiling forms, 4GL uses the schema of tables from the specified database to define the data types of fields in the form and obtains default values and attributes from the **syscolval** and **syscolatt** tables in the default database.

Database References in the DATABASE Section

If the form specification includes any table or column names from a database, the DATABASE section must specify exactly one *database reference*. For Informix Dynamic Server, the following are valid *database reference* formats:

```
database
database @ server
"//server/database"
```

The last format requires quotation marks.

The next examples specify that columns or tables referenced in the TABLES, ATTRIBUTES, or INSTRUCTIONS section are in the **stores7** database; the last two DATABASE section examples specify **mammoth** as the database server:

```
DATABASE stores7
DATABASE stores7@mammoth
DATABASE "//mammoth/stores7"
```

For databases supported by the INFORMIX-SE database server, the following are all valid *database reference* formats:

```
database
"/pathname/database@server"
"//server/pathname/database"
```

Quotation marks around the last two formats are mandatory. Here *pathname* is a pathname to the directory that contains *database*, and *server* is the name of the host system where *database* resides.

For INFORMIX-SE, the following DATABASE sections illustrate these formats:

```
DATABASE newdb
DATABASE "/usr/projects/newdb@mammoth"
DATABASE "//mammoth/usr/projects/newdb"
```

The FORMONLY Option

You can create a form that is not related to any database. To do so, specify FORMONLY after the DATABASE keyword and omit the TABLES section. Also specify FORMONLY as the only table name in the ATTRIBUTES section when you declare the name of each field (as described in "ATTRIBUTES Section" on page 6-25).

The following example of a DATABASE section specifies that the screen form is not associated with any database:

```
DATABASE FORMONLY
```

Compilation errors can result if FORMONLY appears in the DATABASE section of a form that also specifies features that depend on information from the system catalog or from the **syscolval** and **syscolatt** tables of a database. You can declare fields as FORMONLY in the ATTRIBUTES section, however, even if the DATABASE section specifies a database.

The following features of 4GL forms depend on a database:

- The TABLES section
- Any field associated with a database column in the ATTRIBUTES section
- Any FORMONLY field declared LIKE a column in the ATTRIBUTES section
- DISPLAY LIKE or VALIDATE LIKE attributes in the ATTRIBUTES section

The WITHOUT NULL INPUT Option

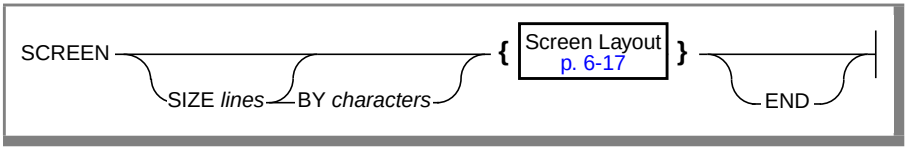
The WITHOUT NULL INPUT keywords indicate that *database-name* does not support null values. Use this option only if you have elected to create and work with a database that does not support null values.

For fields that have no other defaults, the WITHOUT NULL INPUT option causes the form to display zeros as default values for number and INTERVAL fields and blanks for character fields. DATE values default to `12/31/1899`. The default value for DATETIME fields is `1899-12-31 23:59:59.99999`.

The use of WITHOUT NULL INPUT is discouraged. This specification was useful in 1981 for Informix 1.00 databases, which did not yet support the construct of null values. Current Informix databases can prevent null input by specifying a not-null constraint on individual columns through the CREATE TABLE or ALTER TABLE statements. If your application requires non-null values in a field corresponding to a database column, you can specify in the ATTRIBUTES section that the field is FORMONLY and NOT NULL, as described in "Field Attribute Syntax" on page 6-33.

SCREEN Section

The SCREEN section of the form specification file specifies the vertical and horizontal dimensions of the physical screen and the position of one or more display fields and other information that will appear on the screen form. This section is required.



Element	Description
lines	is a literal integer that specifies how many lines of characters (measured vertically) the form can display. The default is 24.
characters	is a literal integer that specifies how many characters (measured horizontally) a line can display. The default is the maximum number of characters in any line of the screen layout.

Usage

The SCREEN keyword is required. As in other sections of a form specification, the keyword END is optional.

A single pair of braces ({ }), immediately preceded and immediately followed by NEWLINE characters, must enclose the screen layout. You cannot use comment indicators in the SCREEN section.

The SIZE Option

If you omit the SIZE keyword, *lines* defaults to 24, and *characters* defaults to the maximum number of characters in any line of your screen layout. If you create a default form from the Programmer's Environment (as described in "Default Forms" on page 6-89), the file shows the SIZE default values.

Specify *lines* as the total height of the form. Four lines are reserved for the system, so by default, no more than (`lines - 4`) lines of the form can display data. (But the OPEN WINDOW…ATTRIBUTE (FORM LINE FIRST, COMMENT LINE = OFF) statement can reduce this overhead.)

If the value of (`lines - 4`) is less than the number of lines in the screen layout, FORM4GL splits your form into a new page after every (`lines - 4`) lines. 4GL does not support multiple-page forms, so any lines beyond the first page will overlay the last line of the first page if your screen layout is too large for your screen. (To avoid this superimposition, create several form specification files if you need to display more lines than can fit on one form.)

You can override either or both of the *lines* and *characters* dimensions of the SCREEN section by specifying the following **form4gl** command, where *lines* is the height of the screen, *characters* is the width of the screen, and *form-name* is the filename (without the **.per** extension) of a form specification file.:

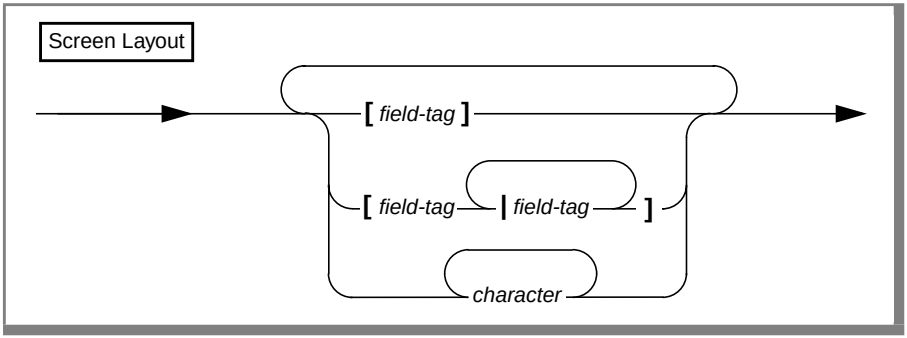
```
form4gl -l lines -c characters form-name
```

For complete information on the **form4gl** command, see "Compiling a Form at the Command Line" on page 6-87.

The portion of the SCREEN section between the braces is called the screen layout. This portion shows the geometric arrangement of the logical screen. If the SIZE clause or command line specifies dimensions that are too small for the screen layout, FORM4GL issues a compile-time warning, but it produces the compiled form that your form specification file described.

The Screen Layout

The screen layout of the SCREEN section must be enclosed between a pair of braces, each in the first character position of an otherwise empty line. The screen layout consists of display fields and (optionally) text characters.



Element	Description
field-tag	is a 4GL identifier of no more than 128 bytes within each field. The length of a field tag cannot exceed the field width.
character	is a printable character of text that will appear in the form.

Display Fields

Every 4GL form must include at least one field where data can appear. Use brackets ([ ]) to delimit fields in the screen layout. Between delimiters, each field must have an identifying field tag. For more information, see "Field Tags" on page 6-18.

Field Delimiters

Each field must be indicated by left and right delimiters to show the length of the field and its position within the screen layout. Both delimiters must appear on the same line. Usually you use left and right brackets to delimit fields. However, to make two fields appear directly next to each other, you can use the pipe symbol (|) to indicate the end of the first field and the beginning of the second field. For complete information on using a pipe symbol to delimit fields, see "Field Delimiters" on page 6-79.

Field Length

If you create a nondefault form, you normally should set the width of each display field in the SCREEN section to be equal to the width of the program variable or the database column to which it corresponds.

A field to display numeric values should be large enough to contain the largest anticipated value. When a numeric field is too small to display a data value, 4GL fills the field with asterisk (*) symbols to indicate the overflow.

Fields to display character data can be shorter than the data length. 4GL fills the field from the left and truncates from the right any string that is longer than its display field. By using multiple-segment fields, you can display portions of a long character value in successive lines of the form. For more information, see "Multiple-Segment Fields" on page 6-31.

In a default form specification file, the widths of all fields are determined by the data type of the corresponding columns in the database tables. (For more information, see "Creating and Compiling a Form" on page 6-85.) The section "Default Forms" on page 6-89 lists default field widths for each data type.

If you edit and modify the default form specification file or create a new file, you can verify that the field widths match the data length requirements of the corresponding character columns when you compile the form. See also "Compiling a Form at the Command Line" on page 6-87.

Field Tags

Field tags must follow the rules for 4GL identifiers (as described in "4GL Identifiers" on page 2-14). The first character of a field tag must be a letter or underscore (_). Other characters can be any combination of letters, digits, and underscores. Because FORM4GL is not case sensitive, both **a1** and **Al** represent the same field tag. Field tags cannot be referenced in 4GL statements. The ATTRIBUTES section declares a field name for each field tag.

Each field has only one tag, but fields with the same tag can appear at more than one position in the SCREEN section in two special cases:

- As part of a screen array

  See "Screen Arrays" on page 6-77.

- As part of a multiple-segment WORDWRAP field

  See "WORDWRAP" on page 6-67.

Otherwise, each field tag must be unique within a form.

Because a field tag must fit within the brackets that delimit its field, you can give single-character fields the tags **a** through **z**. This designation implies that a form can include no more than 27 single-character fields in the default locale.

GLS

Locales for natural languages other than English might have different limits on the number of single-character fields within a single 4GL form. ♦

Literal Characters in Forms

A screen layout can specify character strings that always appear in the form. These strings can label the form and its fields or format the display. (See also "Graphics Characters in Forms" on page 6-21.)

GLS

In U.S. English locales, literal characters in the screen layout are restricted to ASCII characters, but in other locales you can include any other printable characters that the code set of the client locale supports. ♦

Text cannot overlap display fields, but the PICTURE attribute (described in "Field Attribute Syntax" on page 6-33) can specify literal characters within CHAR or VARCHAR fields.

Data Entry of Commas in Integer Fields

All locales ignore comma (,) symbols that the user enters in INTEGER and SMALLINT fields of a 4GL screen form. Thus, the entered value "`12,345`" is interpreted as "12345." Similarly, an anomalously formatted value such as "`1,2,,3,,,4,,,,5,,,`" which uses commas in a way that no locale supports, is interpreted as "12345." No error checking on the use of commas is performed during data entry into INTEGER or SMALLINT fields.

The SCREEN section shown in Figure 6-2 appears in the **orderform.per** form specification file in the **stores7** demonstration application. This example uses default screen dimensions (24 by 80) and textual information for field labels, a screen title, and ornamental lines. ("INSTRUCTIONS Section" on page 6-74 describes how repeated field tags are used in forms that define screen arrays.)

Figure 6-2
Example of a Screen Section

```
SCREEN
{
--------------------------------------------------------------------------------
                              ORDER FORM
--------------------------------------------------------------------------------
Customer Number:[f000       ]  Contact Name:[f001           ][f002        ]
   Company Name:[f003                ]
        Address:[f004                ][f005                ]
           City:[f006           ] State:[a0] Zip Code:[f007 ]
      Telephone:[f008             ]
--------------------------------------------------------------------------------
Order No:[f009       ]  Order Date:[f010      ]  Purchase Order No:[f011  ]
    Shipping Instructions:[f012                                      ]
--------------------------------------------------------------------------------
Item No.  Stock No.  Code    Description    Quantity    Price       Total
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018    ]
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018    ]
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018    ]
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018    ]
                Running Total including Tax and Shipping Charges:[f019    ]
================================================================================
}
END
```

Important: *The backslash (\ ) is not valid as a text character; FORM4GL attempts to interpret it as the beginning of an escape sequence and does not print it. In addition, your form might not compile correctly if you attempt to use either braces ({ and }) or the field delimiter ([,], and \|) symbols as text characters in the screen layout. FORM4GL interprets any pound sign (#) or double-hyphens (--) in the screen layout as literals, not as comment indicators.*

Graphics Characters in Forms

You can include graphics characters in the SCREEN section to place boxes and other rectangular shapes in a screen form. Use the following characters to indicate the borders of one or more boxes on the form.

Symbol	Purpose
p	Use p to mark the upper-left corner.
q	Use q to mark the upper-right corner.
b	Use b to mark the lower-left corner.
d	Use d to mark the lower-right corner.
-	Use hyphens to indicate horizontal line segments.
|	Use pipe symbols to indicate vertical line segments.

The meanings of these six special characters are derived from the **gb** and **acsc** specifications in the **termcap** and **terminfo** files, respectively. 4GL substitutes the corresponding graphics characters when you display the compiled form.

Once the form has the desired configuration, use **\g** to indicate when to begin graphics mode and when to end graphics mode.

Insert **\g** before the first **p**, **q**, **d**, **b**, hyphen, or pipe symbol that represents a graphics character. To leave graphics mode, insert **\g** after the **p**, **q**, **d**, **b**, hyphen, or pipe symbol. FORM4GL exits from graphics mode automatically at the end of each screen line, regardless of whether the **\g** string terminates the line. This means that every line that requires graphics mode must start with the **\g** string, but does not necessarily need to end with it.

Do not insert **\g** into original white space of a screen layout. The backslash should displace the first graphics character in the line and push the remaining characters to the right. The process of indicating graphics distorts the appearance of a screen layout in the SCREEN section, compared to the corresponding display of the screen form.

You can include other graphics characters in a form specification file, but the meaning of a character other than the **p**, **q**, **d**, **b**, hyphen, and pipe symbol is terminal dependent.

To use graphics characters, the system **termcap** or **terminfo** file must include entries for specific variables.

The following table shows what variables need to be set in the **termcap** file.

termcap Variable	Description
gs	The escape sequence for entering graphics mode
ge	The escape sequence for leaving graphics mode
gb	The concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border

The following table shows what variables need to be set in the **terminfo** file.

terminfo Variable	Description
smacs	The escape sequence for entering graphics mode
rmacs	The escape sequence for leaving graphics mode
acsc	The concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border

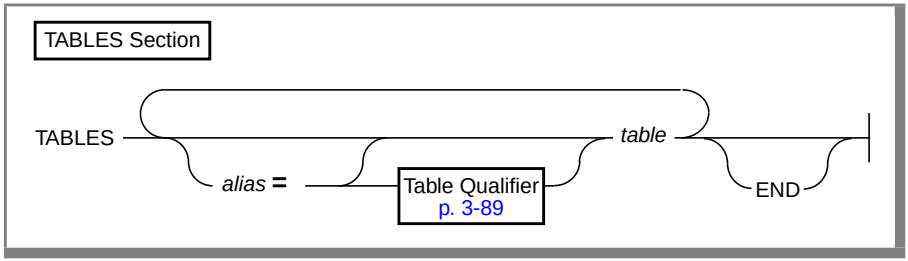
See Appendix F, "Modifying termcap and terminfo," and the manual that comes with your terminal for information about making changes to your **termcap** or **terminfo** file to support these graphics characters.

Rectangles Within Forms

You can use the built-in FGL_DRAWBOX( ) function (described in Chapter 5, "Built-In Functions and Operators") to enclose parts of the screen layout within rectangles. Rectangles that you draw with FGL_DRAWBOX( ) are part of a displayed form. Each time that you execute the corresponding DISPLAY FORM or OPEN WINDOW…WITH FORM statement, you must also redraw the rectangle. Avoid having the rectangle intersect any field or any 4GL reserved line because the rectangle will be broken at the intersection when anything is displayed in the field or in the reserved line.

TABLES Section

The TABLES section lists the database tables that are referenced elsewhere in the form specification file. You must list in this section any table, view, or synonym that includes a column whose name is referenced in the form.



Element	Description
alias	is the alias that replaces *table* in the form specification file.
table	is the identifier or synonym of a table or view in its database.

Usage

If the DATABASE section specifies FORMONLY, no TABLES section is needed unless you give a field the VALIDATE LIKE or DISPLAY LIKE attribute in the ATTRIBUTES section or specify a FORMONLY field LIKE a database column.

Every database column referenced in the ATTRIBUTES section must be part of some *table* specified in the TABLES section. The *table* identifier is the name listed in the **tabname** column of the **systables** catalog or else a synonym.

4GL allows you to specify up to 20 tables, but the actual limit on the number of tables, views, and synonyms that you can reference in a form depends on how your system is configured. The form specification file **orderform.per** in the demonstration application lists four tables:

```
TABLES customer orders items stock
```

The *table* identifier cannot be a temporary table. If the form supports entry or update of data in a view, your 4GL application should test at runtime whether the view is updatable, especially if it is based on other views.

The END keyword is optional.

Table Aliases

The TABLES section must declare an *alias* value for the identifier of any table, view, or synonym that requires a table qualifier (as described in "Table Qualifiers" on page 3-89). Table qualifiers can specify the owner of a table or a database (or *database@server* value) that is different from the database in the DATABASE section. In an ANSI-compliant database, for example, you must qualify any table name with the *owner* prefix if the form will be used by anyone other than *owner*.

You do not need to specify *alias*, unless the form will be used in an ANSI-compliant database by a user who did not create *table*, or if the form references a table, view, or synonym whose name is the same as another in the same database, so that the *owner* prefix is required for an unambiguous reference.

The alias can be the same identifier as *table*. For example, **stock** can be the alias for **stores7@naval:tom.stock**. Except to assign an alias in the TABLES section, a form specification file cannot qualify the name of a table. If a qualifier is needed, you must use an alias from the TABLES section to reference the table in other sections of the form specification file.

Table aliases cannot exceed the 128-byte limit for 4GL identifiers.

The same alias must also appear in screen interaction statements of 4GL that reference screen fields linked to columns of a table that has an alias. Statements in 4GL programs or in other sections of the form specification file can reference screen fields as *column*, as *alias*. *column*, or as *table*. *column*, but they cannot specify *owner*. *table*. *column*. You cannot specify *table*. *column* as a field name if you define a different alias for *table*.

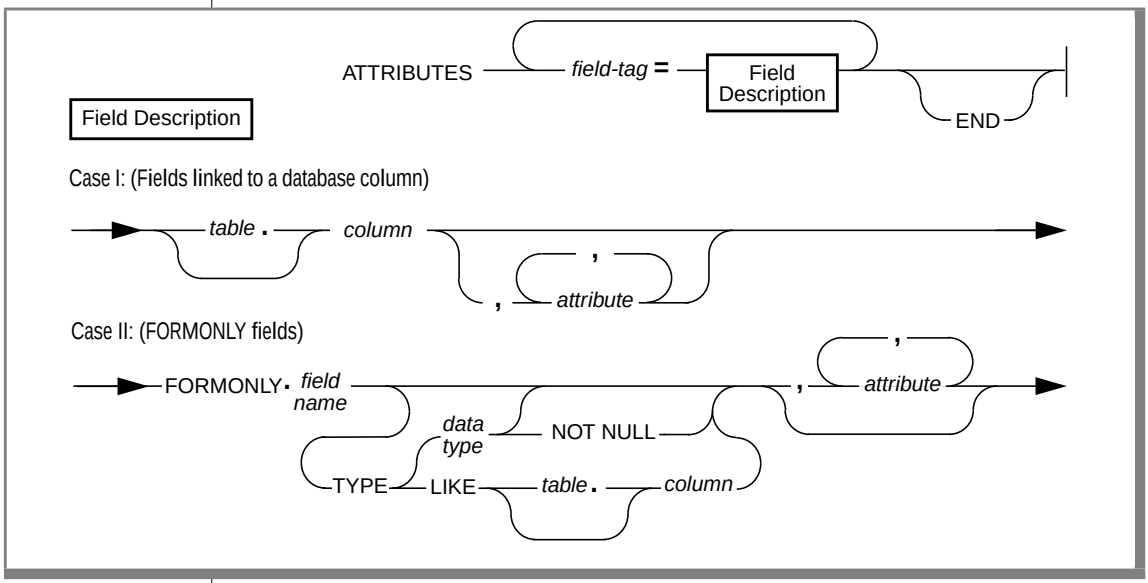
The following TABLES section specifies aliases for two tables:

```
TABLES   tab1  =  libdpt.booktab
         tab2  =  athdpt.balltab
```

ATTRIBUTES Section

The ATTRIBUTES section specifies a field description that associates an identifier and a data type with every field in the SCREEN section. You can also control the behavior and appearance of each field by using field attributes to describe how 4GL should display the field, supply a default value, limit the values that can be entered, or set other parameters. For information on field attributes, see "Field Attribute Syntax" on page 6-33.

The ATTRIBUTES section has the following syntax.



Element	Description
attribute	specifies one of the field attribute listed on page 6-32.
column	is the unqualified SQL identifier of a database column.
data type	is any 4GL data type specification except ARRAY or RECORD.
field name	is an identifier that you assign to a FORMONLY field (a field that is not associated here with any database column).
field-tag	is the field tag, as declared in the SCREEN section.
table	is the name or alias of a table, synonym, or view, as declared in the TABLES section.

Usage

The ATTRIBUTES section must describe every *field-tag* value from the SCREEN section. The order in which you list the field tags determines the order of fields in the default screen records that 4GL creates for each table. ("INSTRUCTIONS Section" on page 6-74 describes screen records.)

The tables specification is not required unless several columns in different tables have the same name or the table is an external table.

The END keyword is optional. It is supported to provide compatibility with form specification files for earlier versions of Informix products.

You can specify two kinds of field descriptions: those that associate a field tag with the data type and with the default attributes of a database column and those that link field tags to FORMONLY fields.

Fields Linked to Database Columns

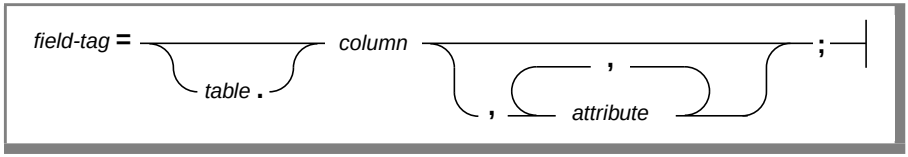
Unless a display field is FORMONLY, its field description must specify the SQL identifier of some database column as the name of the display field. Fields are associated with database columns only during the compilation of the form specification file. During the compilation process, FORM4GL examines two optional tables, **syscolval** and **syscolatt**, for default values of the attributes that you have associated with any columns of the database. (For a description of these tables, see "Default Attributes" on page 6-80.)

After FORM4GL extracts any default attributes and identifies data types from the system catalog, the association between fields and database columns is broken, and the form cannot distinguish the name or synonym of a table or view from the name of a screen record. The form driver in your 4GL program must mediate between screen fields and database columns by using 4GL program variables.

If the database server whose table, view, or synonym is referenced in the ATTRIBUTES section has the **IFX_LONGID** environment variable set to 1, then a *table* identifier or *column* identifier can require up to 128 bytes of storage.

Important: *If a form links a view to a screen field that permits data entry or data editing, it is the responsibility of the programmer to test at runtime whether the view is updatable, especially if the view is based on another view.*

This is the syntax for specifying the attributes of a screen field that is linked to a column of a database.



Element	Description
attribute	is a string of keywords, identifiers, and symbols that specify a field attribute, as listed in "Field Attributes" on page 6-32.
column	is the unqualified SQL identifier of a database column. This variable can also appear in 4GL statements that reference the field.
field-tag	is the field tag, as declared in the SCREEN section.
table	is the name or alias of a database table, synonym, or view, as declared in the TABLES section. Qualifiers are not allowed.

Usage

Although you must include an ATTRIBUTES section that assigns at least one name to every *field-tag* value from the SCREEN section, you are not required to specify any field attributes.

You are not required to specify *table* unless the name *column* is not unique within the form specification, or if table is external to the database that the DATABASE section specifies. However, Informix recommends that you always specify *table*.*column* rather than the unqualified *column* name.

If there is ambiguity, FORM4GL issues an error during compilation.

Because you can refer to field names collectively through a screen record built upon all the fields linked to the same table, your forms might be easier to work with if you specify *table* for each field. For more information about declaring screen records, see "INSTRUCTIONS Section" on page 6-74.

A screen field can display a portion of a character string if you use subscripts in the *column* specification. Subscripts are a pair of comma-separated integers in brackets ([ ]) that indicate starting and ending character positions within a string value. But if you specify in the ATTRIBUTES section that two fields are linked to the same character column in the database, you *cannot* associate each field with a different substring of the same column.

The ATTRIBUTES section in the following file lists fields linked to columns in the **customer** table. The UPSHIFT and PICTURE attributes that are assigned here are described later in this chapter.

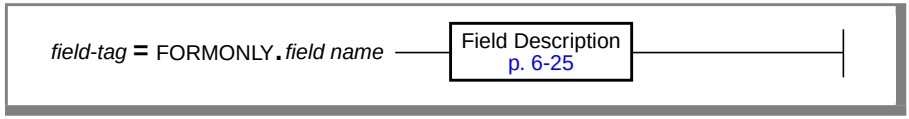
```
DATABASE stores7

SCREEN
{
   Customer Name:[f000           ][f001           ]
         Address:[f002                ][f003               ]
            City:[f004           ] State:[a0] Zip Code:[f005 ]
       Telephone:[f006              ]
}

TABLES   customer

ATTRIBUTES
f000 = customer.fname;
f001 = customer.lname;
f002 = customer.address1;
f003 = customer.address2;
f004 = customer.city;
a0 = customer.state, UPSHIFT;
f005 = customer.zipcode;
f006 = customer.phone, PICTURE = "###-###-#### XXXXX";
```

Values from a column of data type BYTE are never displayed in a form; the words <BYTE value> are shown in the corresponding display field to indicate that the user cannot see the BYTE data. The following excerpt from a form specification file shows a TEXT field **resume** and a BYTE field **photo**. In this example, the BYTE field is short because only the words <BYTE value> are displayed. Similarly, you do not need to include more than one line in a form for a TEXT field. (The PROGRAM attribute that can display TEXT or BYTE values is described in "PROGRAM" on page 6-60.)

```
resume  [f003                                                  ]
photo   [f004        ]
  . . .
attributes
f003 = employee.resume;
f004 = employee.photo;
```

FORMONLY Fields

FORMONLY fields are not associated with columns of any database table or view. They can be used to enter or display the values of program variables. If the DATABASE section specifies FORMONLY, this is the only kind of field description that you can specify in the ATTRIBUTES section.



Element	Description
field name	is an identifier that you assign to a FORMONLY field. This identifier can also appear in 4GL statements that reference the field.
field-tag	is the field tag, as declared in the SCREEN section.

Usage

Like other 4GL identifiers, *field name* cannot begin with a number. It can have up to 128 bytes, including letters, numbers, and underscore (_) symbols.

If you specify one or more FORMONLY fields, 4GL behaves as if they formed a database table named **formonly**, with the field names as column names. The following fields are examples of FORMONLY fields:

```
f021 = FORMONLY.manu_name;
f022 = FORMONLY.unit_price TYPE MONEY, COLOR = GREEN;
f023 = FORMONLY.unit_descr TYPE LIKE orders.unit_descr;
f024 = FORMONLY.order_placed
   TYPE DATETIME YEAR TO HOUR NOT NULL, DEFAULT = CURRENT;
```

The Data Type Specification

The optional 4GL data type specification uses a restricted subset of the data type declaration syntax that the DEFINE, ALTER TABLE, and CREATE TABLE statements support. The data type *cannot* be declared here as a RECORD or as an ARRAY even if 4GL uses the field to display values from a program record or a program array; screen arrays are declared in another section of the form specification file. (It also cannot be SERIAL because SERIAL is an SQL data type, and only 4GL data types are allowed here.)

If you do not specify any data type, FORM4GL treats the field as type CHAR by default. Do not assign a length to CHAR, DECIMAL, and MONEY fields because field length is determined by the display width in the SCREEN section. For example, the demonstration application uses the following FORMONLY field to store the running total price for the order as items are entered:

```
f019 = FORMONLY.t_price;
```

You are required to specify a data type only if you also specify an INCLUDE or DEFAULT attribute for this field. 4GL performs any necessary data type conversion for the corresponding program variable during input or display. 4GL evaluates the LIKE clause at compile time, not at runtime. If the database schema changes, you might need to recompile a program that uses the LIKE clause to describe a FORMONLY field in a form specification file.

Like a field linked to a database column, a FORMONLY field cannot display a BYTE value directly. The form displays the string `<BYTE value>` to indicate that the user cannot see the BYTE value. Similarly, you need not allocate more than one line on a form for a FORMONLY field of data type TEXT. You can assign the PROGRAM attribute to a FORMONLY field to display TEXT or BYTE values from 4GL variables.

The NOT NULL Keywords

The NOT NULL keywords specify that if you reference this screen field in an INPUT statement, the user must enter a non-null value in the field. (These keywords are more restrictive than the REQUIRED attribute, which permits the user to enter a null value. For more information, see "REQUIRED" on page 6-62.)

If the DATABASE section has the (deprecated) WITHOUT NULL INPUT clause, the NOT NULL keywords instruct 4GL to use zero (for number or INTERVAL data types) or blank spaces (for character data types) as the default value for this field in INPUT statements. The default DATE value is `12/31/1899`. The default value for DATETIME fields is `1899-12-31 23:59:59.99999`.

Multiple-Segment Fields

If you need the form to support entry or display of long character strings, you can specify *multiple-segment* fields that occupy several lines of the form. To create a multiple-segment field, repeat the same field tag in different fields of the layout in the SCREEN section, typically on successive lines.

WORDWRAP Fields

For a multiple-segment field to enter or display long character strings in successive lines of the form, you must also specify the WORDWRAP attribute for that field tag in the ATTRIBUTES section. During input and display, 4GL treats all segments that have that field tag as segments of a single field.

The following example shows only the SCREEN and ATTRIBUTES sections of a form specification file that specifies a multiple-segment field:

```
SCREEN SIZE 24 BY 80
{
      title: [title                                          ]
     author: [author                                ]
   synopsis: [synopsis                                                ]
             [synopsis                                                ]
             [synopsis                                                ]
             [synopsis                                                ]
             [synopsis                                                ]
}
      . . .
ATTRIBUTES
    title = booktab.title;
    author = booktab.author;
    synopsis = booktab.synopsis, WORDWRAP COMPRESS;
```

Here the screen field whose tag is **synopsis** appears in five physical segments in the screen layout and has the WORDWRAP attribute. Its value is composed of the physical segments, taken in top-to-bottom, left-to-right order. The field should ordinarily be as long or longer than the program variable or database column that it displays, so it can display all of the text. Users of your 4GL application program might expect all segments to be the same size and to be laid out in vertical alignment, as in the example, but that is not required. Your form is likely to be easier to use, however, if multiple-segment fields are compact and symmetrical.

In the description of the field in the last line of the ATTRIBUTES section of the previous example, the keyword WORDWRAP enables a multiple-line editor when the form is open and the cursor enters the field. If you omit it, words cannot flow from segment to segment of the field, and users must move the cursor from field to field with arrow keys or RETURN to edit values in the form. (For more information about the multiple-line editor and about the COMPRESS keyword, see "WORDWRAP" on page 6-67.)

Subscripted Fields

FORM4GL can create a default form specification that references a database column of a character data type whose declared length in bytes is greater than (`characters - 22`), where *characters* is the width of the form. For such columns, FORM4GL generates two or more fields in the default specification (each with a different field tag but with the same field name) whose total length is the declared length of the corresponding database column.

In the ATTRIBUTES section of the default specification, the name of each such field is immediately followed (in brackets) by an ordered pair of comma-separated literal integers that identify the first and last of the bytes in the database column that the field displays. For example:

```
c3 = engineer.code[1,56];
c4 = engineer.code[57,112];
```

These are called *subscripted fields*, and they are generated for backward compatibility with the PERFORM forms compiler. Statements of 4GL that enter or display data in screen forms (CONSTRUCT, INPUT, and INPUT ARRAY) are difficult to use with subscripted 4GL fields beyond the first field.

Use a text editor to change such fields to segmented WORDWRAP fields.

Field Attributes

FORM4GL recognizes the following field attributes.

AUTONEXT	DISPLAY LIKE	NOENTRY	UPSHIFT
CENTURY	DOWNSHIFT	PICTURE	VALIDATE LIKE
COLOR	FORMAT	PROGRAM	VERIFY
COMMENTS	INCLUDE	REQUIRED	WORDWRAP
DEFAULT	INVISIBLE	REVERSE	

The following table summarizes the effects of these field attributes, which are individually described in the sections that follow.

Attribute	Effect
AUTONEXT	Causes the cursor to advance automatically to the next field
CENTURY	Specifies expansion of 2-digit years in DATE and DATETIME fields
COLOR	Specifies the color or intensity of values displayed in a field
COMMENTS	Specifies a message to display on the Comment line
DEFAULT	Assigns a default value to a field during data entry
DISPLAY LIKE	Assigns attributes from **syscolatt** table that the **upscol** utility creates, associating attributes with specific database columns
DOWNSHIFT	Converts to lowercase any uppercase character data
FORMAT	Formats DECIMAL, SMALLFLOAT, FLOAT, or DATE output
INCLUDE	Lists a set of acceptable values during data entry
INVISIBLE	Does not echo characters on the screen during data entry
NOENTRY	Prevents the user from entering data in the field
PICTURE	Imposes a data-entry format on CHAR or VARCHAR fields
PROGRAM	Invokes an external program to display TEXT or BYTE values
REQUIRED	Requires the user to supply some value during data entry
REVERSE	Causes values in the field to be displayed in reverse video
UPSHIFT	Converts to uppercase any lowercase character data
VALIDATE LIKE	Validates data entry with the **syscolval** table that the **upscol** utility creates, associating default values with specific database columns
VERIFY	Requires that data be entered twice when the database is modified
WORDWRAP	Invokes a multiple-line editor in multiple-segment fields

Field Attribute Syntax

Syntax for assigning field attributes is described in the sections that follow. As the syntax diagram for the ATTRIBUTES section indicated on page 6-25, fields that have more than one attribute must separate successive attribute specifications with a comma (,), and must terminate the list of attributes with a semicolon (;), even if the attribute list is empty.

AUTONEXT

The AUTONEXT attribute causes the cursor to advance automatically during input to the next field when the current field is full.

Usage

You specify the order of fields in each INPUT or INPUT ARRAY statement. If the most recent OPTIONS statement specifies INPUT WRAP, the *next* field after the last field is the first field.

AUTONEXT is particularly useful with character fields in which the input data is of a standard length, such as numeric postal codes or the abbreviations in the **state** table. It is also useful if a character field has a length of `1` because only one keystroke is required to enter data and move to the next field.

If data values entered in the field do not meet requirements of other field attributes like INCLUDE or PICTURE, the cursor does *not* automatically move to the next field but remains in the current field, with an error message.

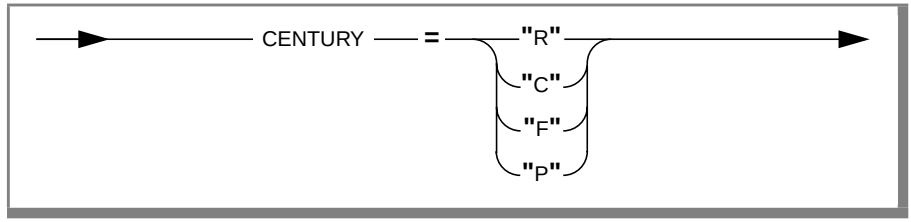
The demonstration application uses the **customer** form to enter all the names and addresses of the customers. The following excerpt from the ATTRIBUTES section of the **customer** form uses the AUTONEXT attribute:

```
a0 = customer.state, DEFAULT = "CA", AUTONEXT;
f007 = customer.zipcode, AUTONEXT;
f008 = customer.phone;
```

When two characters are entered into the **customer.state** field (thus filling the field), the cursor moves automatically to the beginning of the next screen field (the **customer.zipcode** field). When five characters are entered into the **customer.zipcode** field (filling this field), the cursor moves automatically to the beginning of the next field (the **customer.phone** field).

CENTURY

The CENTURY attribute specifies how to expand abbreviated one- and two-digit *year* specifications in a DATE and DATETIME field. Expansion is based on this setting (and on the year value from the system clock at runtime).



In most versions of 4GL earlier than 7.20, if the user enters only the two trailing digits of a year for literal DATE or DATETIME values, these digits are automatically prefixed with the digits `19`. For example, `12/31/02` is always expanded to `12/31/1902` regardless of when the program is executed.

CENTURY can specify any of four algorithms to expand abbreviated years into four-digit year values that end with the same digits (or digit) that the user entered. CENTURY supports the same settings as the **DBCENTURY** environment variable but with a scope that is restricted to a single field.

Symbol	Algorithm for Expanding Abbreviated Years
C or c	Use the past, future, or current year closest to the current date.
F or f	Use the nearest year in the future to expand the entered value.
P or p	Use the nearest year in the past to expand the entered value.
R or r	Prefix the entered value with the first two digits of the current year.

Here *past*, *closest*, *current*, and *future* are all relative to the system clock.

Unlike **DBCENTURY**, which sets a global rule for expanding abbreviated year values in DATE and DATETIME fields that do not have the CENTURY attribute, CENTURY is not case sensitive; you can substitute lowercase letters (`r`, `c`, `f`, `p`) for these uppercase letters. If you specify anything else, an error (-2018) is issued. If the CENTURY and **DBCENTURY** settings are different, CENTURY takes precedence.

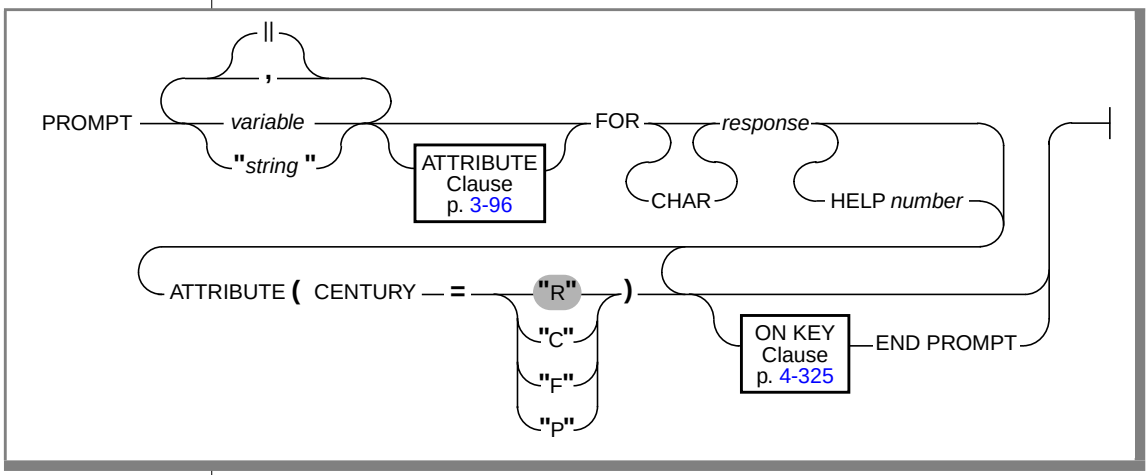
Three-digit years are not expanded. A single-digit year is first expanded to two digits by prefixing it with a zero; CENTURY then expands this value to four digits, according to the setting that you specified. Years between 1 and 99 AD (or CE) require leading zeros (to avoid expansion).

Just as with **DBCENTURY**, expansion of abbreviated years is sensitive to time of execution and to the accuracy of the system clock-calendar.

For examples of the effects of the different CENTURY settings on abbreviated year values, see "DBCENTURY" on page D-15. (Those examples are for **DBCENTURY**, but except for case-sensitivity, **DBCENTURY** and CENTURY have the same semantics.)

Important: *The CENTURY attribute has no effect on DATETIME fields that do not include YEAR as the first time unit nor on fields that are not DATE or DATETIME fields. If an abbreviated year value is entered in a character field or a number field, for example, neither CENTURY nor* ***DBCENTURY*** *has any effect.*

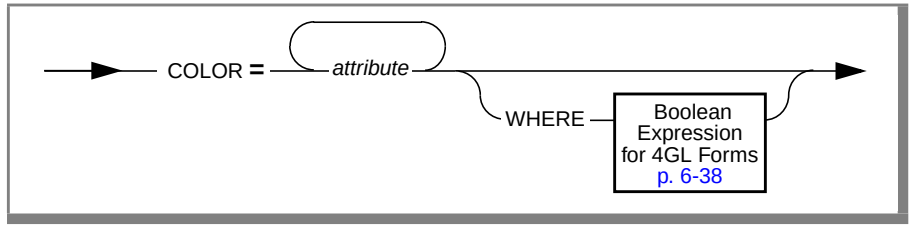
The ATTRIBUTES clause that can follow the FOR clause of the PROMPT statement can also specify CENTURY as an attribute, using this syntax.



Here *response* must be a DATE or DATETIME variable for CENTURY to be useful. This diagram is simplified, in that any FOR…ATTRIBUTE clause of PROMPT that specifies CENTURY can also specify other display attributes, as listed in "ATTRIBUTE Clause" on page 3-96. The setting is not case sensitive but must be enclosed within quotation marks. See "PROMPT" on page 4-325 for descriptions of syntax terms that appear in this diagram.

COLOR

The COLOR attribute displays field text in a color or with other video attributes, either unconditionally or only if a Boolean expression is TRUE.



Element	Description
attribute	is one of the keywords to specify a color or an intensity. You can specify zero or one color keyword and zero or more intensity keywords. The color keywords include BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, and YELLOW. The intensity keywords include REVERSE, LEFT, BLINK, and UNDERLINE.

Important: *If you are using* ***terminfo****, the only color or intensities available are REVERSE and UNDERLINE.*

Usage

If you do not use the WHERE keyword to specify a 4GL Boolean expression, the intensity or color in your display mode list applies to the field. This example specifies unconditionally that field text appears in red:

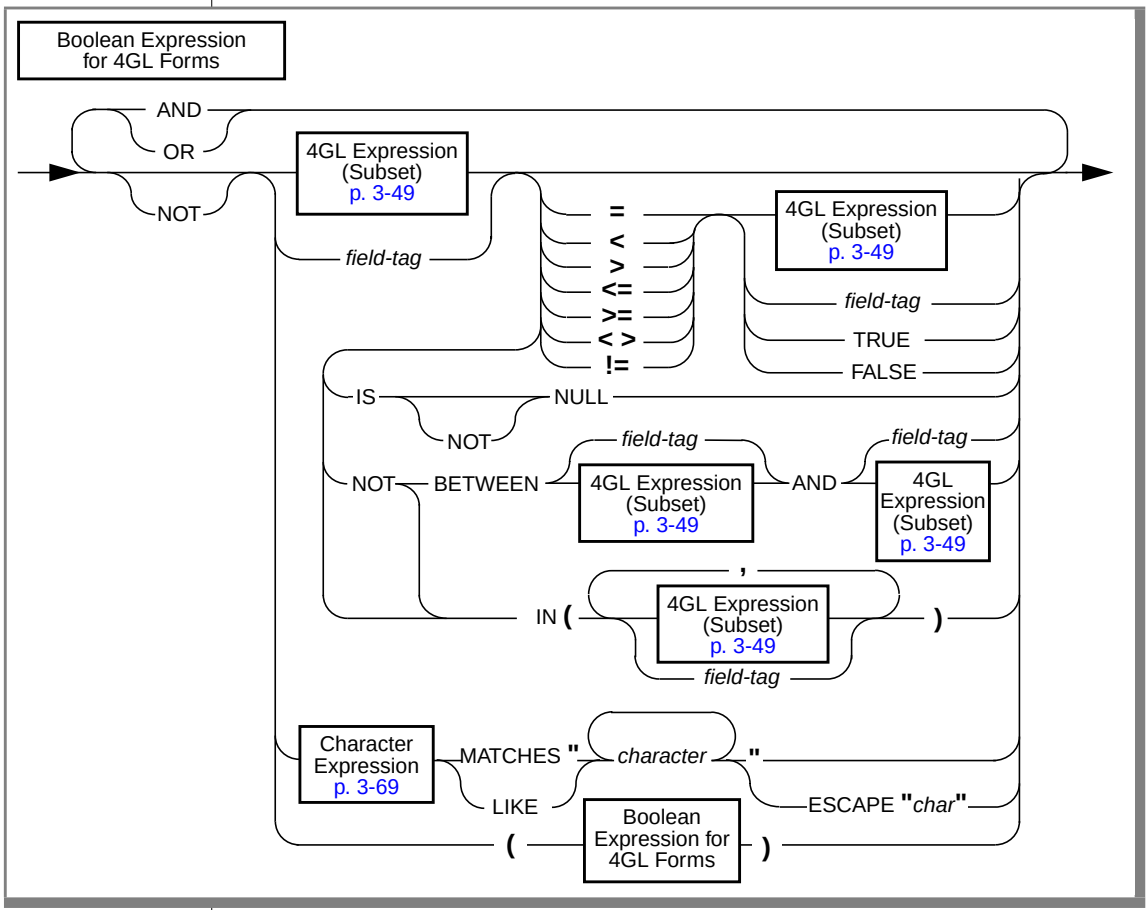
```
f000 = customer.customer_num, COLOR = RED LEFT;
```

Specifying Logical Conditions with the WHERE Option

You can also use the keywords, symbols, and operators that are allowed in 4GL Boolean expressions, including LIKE, MATCHES, TODAY, and CURRENT, in a WHERE clause to specify conditional attributes. If the Boolean expression evaluates as FALSE or NULL, the field is displayed with default characteristics, rather than with those specified by *display mode*. (For more information, see "Default Attributes" on page 6-80.)

Boolean Expressions in 4GL Form Specification Files

The syntax of 4GL Boolean expressions in the WHERE clause of a COLOR attribute specification follows.



Element	Description
char	is a single character, enclosed between a pair of single (') or double (") quotation marks.
character	is one or more literal or special characters, enclosed between two single or double quotation marks.
field-tag	is the field tag (as described in "Field Tags" on page 6-18) of the current field.

In this diagram, terms for other 4GL expressions are restricted subsets. Except for the constants TRUE and FALSE, you cannot reference the *name* of a program variable in the WHERE clause of a COLOR attribute specification. You can, however, include a *field-tag* or a *literal* value wherever the name of a variable can appear in a 4GL expression that is a component of the 4GL Boolean expression, as described in "Expressions of 4GL" on page 3-49.

If any component of a 4GL Boolean expression is null, the value of the entire 4GL Boolean expression is FALSE (rather than null), unless the IS NULL operator is also included in the expression. As in other Boolean expressions of 4GL, applying the NOT operator to a null value does not change its FALSE evaluation. The conditional attribute is displayed only if the overall condition is true. In the following example, the value of the expression is FALSE if a NULL value appears in the display field whose field tag is **f004**:

```
3.1415265 * f004 < 25000
```

If you include a *field-tag* value in a 4GL Boolean expression when you specify a conditional COLOR attribute, 4GL replaces *field-tag* at runtime with the current value in the screen field and evaluates the expression.

If *field-tag* references a field that is linked to a database column of data type TEXT or BYTE or to a FORMONLY field of either of those two data types, only the IS NOT NULL or IS NULL keywords can include that field tag in an expression. The specified color or intensity is applied to the <BYTE value> message, not to the BYTE data value because only the PROGRAM attribute can display a BYTE value. Values of the TEXT data type are displayed in the field, beginning with the first printable data character. If the TEXT value is too large to fit in the field, characters beyond what the field can hold are not displayed.

Specifying Ranges of Values and Set Membership

Conditional COLOR attributes can specify SQL Boolean operators that are not valid in ordinary 4GL Boolean expressions. You can use the BETWEEN...AND operator to specify a range of number, time, or character values. Here the first expression cannot be greater than the second (for number expressions), later than the second (for time expressions), or later in the code-set order than the second (for character expressions). Appendix A, "The ASCII Character Set," lists the numeric values of ASCII characters, which is the collating sequence for U.S. English locales.

The WHERE clause of a COLOR field description can also use the IN operator to specify a comma-separated list (enclosed between parentheses) of values with which to compare the field tag or expression.

"Set Membership and Range Tests" on page 5-40 describes the syntax of the BETWEEN...AND and IN Boolean operators in conditional COLOR specifications for 4GL forms. See also the *Informix Guide to SQL: Reference* for the complete syntax of Boolean expressions in SQL statements.

Data Type Compatibility

You might get unexpected results if you use relational operators or the BETWEEN, AND, or IN operators with expressions of dissimilar data types. In general, you can compare numbers with numbers, character strings with character strings, and time values with time values.

If a time expression component of a 4GL Boolean expression is an INTERVAL data type, any other time expression that is compared to it by a relational operator must also be an INTERVAL value. You cannot compare a span of time (an INTERVAL value) with a point in time (a DATE or DATETIME value).

Data Type Conversion in 4GL Boolean Expressions

If you specify a number, character, or time expression in a context where a 4GL Boolean expression is expected, 4GL applies the following rules after evaluating the number, character, or time expression:

- If the value is a non-zero real number (or a character string representing a non-zero number) or a non-zero INTERVAL, or any DATE or DATETIME value, the 4GL Boolean value is TRUE.
- If the value is null, and the IS NULL keywords are also included in the expression, the value of the 4GL Boolean expression is TRUE.
- Otherwise, the 4GL Boolean expression is FALSE.

The Display Modes

The display and intensity keywords of the COLOR attribute have the same effects on a field as the same keywords of the **upscol** utility (described in "Default Attributes" on page 6-80) or ATTRIBUTES clause (described in "ATTRIBUTE Clause" on page 3-96).

The following table shows the effects of the color attribute keywords on a monochrome terminal.

Attribute	Display	Attribute	Display
WHITE	Normal	CYAN	Dim
YELLOW	Bold	GREEN	Dim
MAGENTA	Bold	BLUE	Dim
RED	Bold	BLACK	Dim

The following table shows the effects of the intensity attribute keywords on a color terminal.

Attribute	Display
NORMAL	White
BOLD	Red
DIM	Blue

The LEFT attribute produces a left-aligned display in a screen field of any number data type. It has no effect on fields of other data types. (Without the `COLOR = LEFT` specification, number values are right-aligned by default.)

The next lines specify display attributes if Boolean expressions are TRUE:

```
f001  = FORMONLY.late, COLOR = RED BLINK WHERE f001 < TODAY;
f002  = manufact.manu_code, COLOR = RED WHERE f002 = "HRO";
f003  = customer.lname, COLOR = RED WHERE f003 LIKE "Quinn";
f004  = mytab.col6, COLOR = GREEN WHERE f004 < 10000;
f005  = mytab.col9, COLOR = BLUE REVERSE WHERE f005 IS NULL,
        COLOR = YELLOW WHERE f005 BETWEEN 5000 AND 10000,
         COLOR = GREEN BLINK WHERE f005 > 10000;
```

The following expression is TRUE if the field f022 does not include the underscore character:

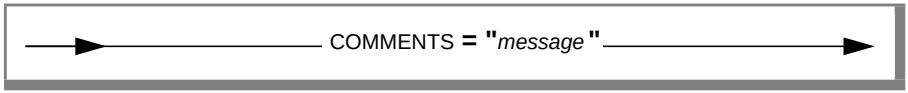
```
NOT f022 LIKE "%z_%" ESCAPE "z"
```

Related Attributes

DISPLAY LIKE, INVISIBLE, REVERSE

COMMENTS

The COMMENTS attribute displays a message on the Comment line at the bottom of the window. The message is displayed when the cursor moves to the specified field and is erased when the cursor moves to another field.



Element	Description
message	is a character string enclosed in quotation marks.

Usage

The *message* string must appear between quotation marks (") on a single line of the form specification file.

In the following example, the field description specifies a message for the Comment line to display. The message appears when the screen cursor enters the field that is linked to the **fname** column of the **customer** table. In the **stores7** database, this column contains the first name of a customer:

```
c2 = customer.fname, comments =
   "Please enter initial if available.";
```

The most common application of the COMMENTS attribute is to give information or instructions to the user. This application is particularly appropriate when the field accepts only a limited set of values. (For details of how to specify a range or a list of acceptable values for data entry, see "INCLUDE" on page 6-53.)

4GL programs can use the same screen form to support several distinct tasks (for example, data input and query by example). Do not specify the COMMENTS attribute in a field description unless *message* is appropriate to all of the tasks in which *message* can appear.

If the same field requires a different message for different tasks, specify each message using MESSAGE or DISPLAY statements rather than in the form specification file.

The Position of the Comment Line

The default position of the Comment line in the 4GL screen is line 23. You can reset this position with the OPTIONS statement.

The default position of the Comment line in a 4GL window is LAST. You can reset this position in the OPTIONS statement to show the new position in all 4GL windows. Alternatively, you can reset it in the ATTRIBUTE clause of the appropriate OPEN WINDOW statement if you want the new position in a specific 4GL window. Chapter 4 describes the OPTIONS and OPEN WINDOW statements.

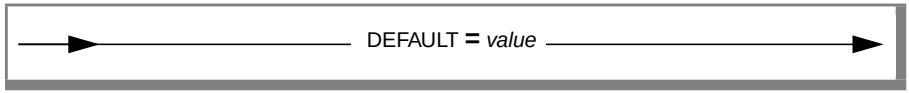
If the OPEN WINDOW statement specifies COMMENT LINE OFF, any output to the Comment line of the specified 4GL window is hidden even if the window displays a form that includes fields that include the COMMENTS attribute. (See also "Hiding the Comment Line" on page 4-290.)

Related Attribute

INCLUDE

DEFAULT

The DEFAULT attribute assigns a default value to a field during data entry.



Element	Description
value	is a default value for the field. This restricted expression cannot reference any variable or programmer-defined function.

Usage

Default values have no effect when you execute the INPUT statement by using the WITHOUT DEFAULTS option. In this case, 4GL displays the values in the program variables list on the screen. The situation is the same for the INPUT ARRAY statement except that 4GL displays the default values when the user inserts a new row.

If the field is FORMONLY, you must also specify a data type when you assign the DEFAULT attribute to a field. (See "FORMONLY Fields" on page 6-29.)

If both the DEFAULT attribute and the REQUIRED attribute are assigned to the same field, the REQUIRED attribute is ignored.

If you do not use the WITHOUT NULL INPUT option in the DATABASE section, all fields default to null values unless you use the DEFAULT attribute. If you use the WITHOUT NULL INPUT option in the DATABASE section and you do not use the DEFAULT attribute, character fields default to blanks, number and INTERVAL fields to `0`, and MONEY fields to `$0.00`. The default DATE value is `12/31/1899`. The default DATETIME value is `1899-12-31 23:59:59.99999`.

You cannot assign the DEFAULT attribute to fields of data type TEXT or BYTE.

Literal Values

The *value* can be a quoted string, a literal number, a literal DATE value, a literal DATETIME value, or a literal INTERVAL value. These values are described in Chapter 3. Or the value can be a built-in function or operator that returns a single value of a data type compatible with that of the field. For details, see "Syntax of Built-In Functions and Operators" on page 5-13.

If you include in the *value* list a character string that contains a blank space, a comma, or any special characters, or a string that does not begin with a letter, you must enclose the entire string in quotation marks.

(If you omit the quotation marks, any uppercase letters are down-shifted.)

For a DATE field, you must enclose any literal value in quotation marks ("). For a DATETIME or INTERVAL field, you can enclose *value* in quotation marks, or you can enter it as an unquoted literal, as in the following examples:

```
DATETIME (2012-12) YEAR TO MONTH
INTERVAL (10:12) HOUR TO MINUTE
- INTERVAL (28735) DAY TO DAY
```

For more information, see "DATETIME Qualifier" on page 3-76 and "INTERVAL Literal" on page 3-82.

Built-In 4GL Operators and Functions as Values

Besides these literal values, you can also specify a built-in 4GL function or operator that returns a single value of the appropriate data type. Arguments or operands must be a literal value, a built-in 4GL function or operator that returns a single value, or the named constants TRUE or FALSE. For example, a default value of data type INTERVAL can be specified in the format:

```
integer UNITS time-unit
```

Here *integer* can be a positive or negative literal integer (as described in "Literal Integers" on page 3-65) or an expression in parentheses that evaluates to an integer, and *time-unit* is a keyword from an INTERVAL qualifier, such as MONTH, DAY, HOUR. (This qualifier must be consistent with the explicit or implied data type declaration of the field; do not, for example, specify YEAR or MONTH as the *time-unit* value for a DAY TO FRACTION field.)

Use the TODAY operator as *value* to assign the current date as the default value of a DATE field. Use the CURRENT operator as *value* to assign the current date and time as the default for a DATETIME field. (4GL does not assign these values automatically as defaults, so you must specify them explicitly.) These expressions are evaluated at runtime, not at compile time.

The following field descriptions specify DEFAULT values:

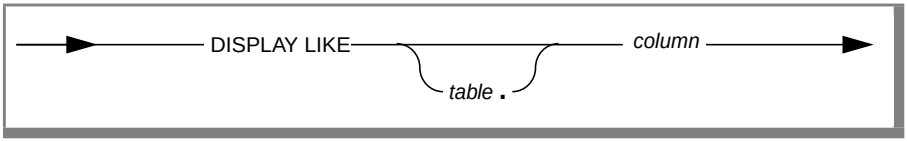
```
c8 = state, UPSHIFT, AUTONEXT,
   DEFAULT = "CA";
o12 = order_date, DEFAULT = TODAY;
f019 = FORMONLY.timestamp TYPE DATETIME YEAR TO DAY
   COLOR = RED, DEFAULT = CURRENT;
```

Related Attributes

INCLUDE, REQUIRED, VALIDATE LIKE

DISPLAY LIKE

The DISPLAY LIKE attribute takes attributes that the **upscol** utility assigned to a specified column in the **syscolatt** table and applies them to a field.



Element	Description
table	is the unqualified name or alias of a database table, synonym, or view, as declared in the TABLES section. (This value is not required unless several columns in different tables have the same name or if the table is an external table or an external, distributed table.)
column	is the name of a column in *table* or (if you omit *table*) the unique identifier of a column in one of the tables that you declared in the TABLES section. The column cannot be of data type BYTE.

Usage

Specifying this attribute is equivalent to listing all the attributes that are assigned to *table.column* in the **syscolatt** table. ("Default Attributes" on page 6-80 describes the **syscolatt** table. See also the description of the **upscol** utility in Appendix B.) You do not need to specify the DISPLAY LIKE attribute if the field is linked to *table.column* in the field name specification.

The following example instructs 4GL to apply the default display attributes of the **items.total_price** column to a FORMONLY field:

```
s12 = FORMONLY.total, DISPLAY LIKE items.total_price;
```

4GL evaluates the LIKE clause at compile time, not at runtime. If the database schema changes, you might need to recompile a program that uses the LIKE clause. Even if all of the fields in the form are FORMONLY, this attribute requires FORM4GL to access the database that contains *table*.

Related Attribute

VALIDATE LIKE

DOWNSHIFT

Assign the DOWNSHIFT attribute to a character field when you want 4GL to convert uppercase letters entered by the user to lowercase letters, both on the screen and in the corresponding program variable.

Usage

Because uppercase and lowercase letters have different values, storing character strings in one or the other format can simplify sorting and querying a database.

By specifying the DOWNSHIFT attribute, you instruct 4GL to convert character input data to lowercase letters in the program variable.

The maximum length of a character value to which you can apply the DOWNSHIFT attribute is 511 bytes.

GLS

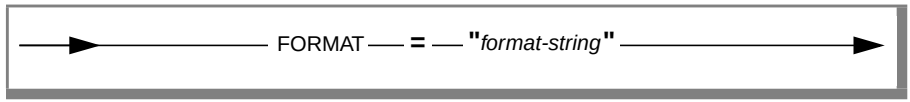
The results of conversion between uppercase and lowercase letters are based on the locale files, which specify the relationship between corresponding pairs of uppercase and lowercase letters. If the locale files do not provide this information, no case conversion occurs. ♦

Related Attribute

UPSHIFT

FORMAT

You can use the FORMAT attribute with a DECIMAL, SMALLFLOAT, FLOAT, or DATE field to control the format of output displays.



Element	Description
format-string	is a string of characters that specifies a data display format. You must enclose *format-string* within quotation marks (").

Usage

This attribute can format data that the application displays in the field. (Use the PICTURE attribute to format data entered in the field by the user.) 4GL right-aligns the data in the field. If *format-string* is smaller than the field width, FORM4GL issues a compile-time warning, but the form is usable.

Formatting Number Values

For DECIMAL, SMALLFLOAT, and FLOAT data types, *format-string* consists of pound signs (#) that represent digits and a decimal point. For example, "###.##" produces at least three places to the left of the decimal point and exactly two to the right.

If the actual number displayed requires fewer characters than *format-string* specifies, 4GL right-aligns it and pads the left with blanks.

If necessary to satisfy the *format-string* specification, 4GL rounds number values before it displays them.

GLS

The NUMERIC setting in the locale files affects how *format-string* is interpreted for numeric data. In the format string, the period symbol (.) is not a literal character but a placeholder for the decimal separator specified by environment variables or by locale file settings. Likewise, the comma (,) is a placeholder for the thousands separator specified by environment variables. Thus, the format string #,###.## formats the value 1234.56 as 1,234.56 in a U.S. English locale but as 1.234,56 in a German locale. ♦

Formatting DATE Values

For DATE data types, 4GL recognizes these symbols as special in *format-string*.

Symbol	Effect
mm	Produces the two-digit representation of the month; for example, `Jan = 01, Feb = 02`
mmm	Produces a three-letter English-language abbreviation of the month; for example, `Jan`, `Feb`
dd	Produces the two-digit representation of the day of the month
ddd	Produces a three-letter English language abbreviation of the day of the week; for example, `Mon, Tue`
yy	Produces the two-digit representation of the year, discarding the leading digits; for example, the year 2003 would appear as `03`
yyyy	Produces a four-digit representation of the year

For DATE fields, FORM4GL interprets any other characters as literals and displays them wherever you place them within *format-string*.

These *format-string* examples and their corresponding display formats for DATE fields display the twenty-third day of September 1999.

Input	Result
no `FORMAT` *attribute*	09/23/1999
`FORMAT = "mm/dd/yy"`	09/23/99
`FORMAT = "mmm dd, yyyy"`	Sep 23, 1999
`FORMAT = "yymmdd"`	990923
`FORMAT = "dd-mm-yy"`	23-09-99
`FORMAT = "(ddd.) mmm. dd, yyyy"`	(Thu.) Sep. 23, 1999

GLS

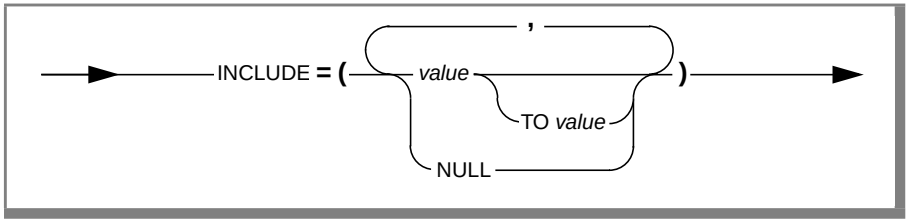
The **mmm** and **ddd** specifiers in a format string can display language-specific month name and day name abbreviations. This operation requires installing message files in a subdirectory of **$INFORMIXDIR/msg** and subsequent reference to that subdirectory by way of the environment variable **DBLANG**. For example, in a Spanish locale, the **ddd** specifier translates the day Saturday into the day name abbreviation **Sab**, which stands for *Sabado* (the Spanish word for *Saturday*). For more information on GLS, see Appendix E, "Developing Applications with Global Language Support." ♦

Related Attribute

PICTURE

INCLUDE

The INCLUDE attribute specifies acceptable values for a field and causes 4GL to check at runtime before accepting an input value.



Element	Description
value	is an element in a comma-separated list (within parentheses) of values (*value1*, *value2*, …), a range of values (*value1* TO *value2*), or any combination of individual values and ranges.

Usage

Each *value* specification is a restricted expression that cannot include the name of any 4GL variable or programmer-defined function. It can include literal values (as described in"Expressions of 4GL" on page 3-49), built-in functions, operators (as described in "Syntax of Built-In Functions and Operators" on page 5-13), and the constants TRUE, FALSE, and NOTFOUND. The same rules for DEFAULT attribute values also apply to INCLUDE values. TEXT and BYTE fields cannot have the INCLUDE attribute.

If a field has the INCLUDE attribute, the user must enter an acceptable value (from the *value* list) before 4GL accepts a new row. If the *value* list does not include the default value, the INCLUDE attribute behaves like the REQUIRED attribute, and an acceptable entry is required. Include the NULL keyword in the *value* list to specify that it is acceptable for the user to press RETURN without entering any value.

```
f006 = survey.item06, INCLUDE = (NULL, "YES", "NO");
```

In this example, the NULL keyword allows the user to enter nothing. You *cannot* accomplish the same thing by substituting a string of blanks for the NULL keyword in the INCLUDE specification because for most data types a null value is different from ASCII 32, the blank character.

Including a COMMENTS attribute for the same field to describe acceptable values makes data entry easier because you can display a message to advise the user of whatever restrictions you have imposed on data entry:

```
i18 = items.quantity, INCLUDE = (1 TO 50),
   COMMENTS = "Acceptable values are 1 through 50";
```

If you include in the *value* list a character string that contains a blank space, a comma, or any special characters or a string that does not begin with a letter, you must enclose the entire string in quotation marks ("). (If you omit the quotation marks, any uppercase letters are down-shifted.)

Ranges of Values

You can use the TO keyword to specify an inclusive range of acceptable values. For example, ranges in the following field description include the postal abbreviations for the names of the contiguous states of the United States:

```
i20 = customer.state,
INCLUDE = (NULL, "AL" TO "GA", "IA" TO "WY"),
COMMENTS = "No Alaska (AK) or Hawaii (HI) addresses here.";
```

When you specify a range of values, the *lower* value must appear first. The meaning of *lower* depends on the data type of the field:

- For number or INTERVAL fields, it is the larger (or only) negative value, or (if neither value is negative) the value closer to zero.
- For other time fields, it is the earlier DATE or DATETIME value.
- For character fields, the lower value is the string that starts with a character closer to the beginning of the collating sequence. (See Appendix A for a listing of the ASCII collating sequence, which U.S. English locales use.)

  For nondefault locales, the code-set order is used as the collating sequence. ♦

GLS

In a number field, for example, the range `"5 TO 10"` is valid. In a character field, however, it produces a compile-time error. (In the default locale, for example, the character string `"10"` is less than `"5"` because 1 comes before 5 in the ASCII collating sequence.)

FORMONLY Fields

You must specify a data type when you assign the INCLUDE attribute to a FORMONLY field (as described in "FORMONLY Fields" on page 6-29). The TYPE clause is required in the following example:

```
f006 = FORMONLY.item07 TYPE CHAR(*),
    INCLUDE = (NULL, "PERHAPS", "MAYBE");
```

Related Attributes

COMMENTS, DEFAULT, REQUIRED

INVISIBLE

The INVISIBLE attribute prevents user-entered data from being echoed on the screen during a CONSTRUCT, INPUT, INPUT ARRAY, or PROMPT statement.

Usage

Characters that the user enters in a field with this attribute are not displayed during data entry, but the cursor moves through the field as the user types. No other aspects of data entry are affected by the INVISIBLE attribute.

The following example illustrates the use of the INVISIBLE attribute:

```
i001 = FORMONLY.secret_password TYPE LIKE state.sname,
   INVISIBLE,
   COMMENTS = "Enter your secret password.";
```

If you specify INVISIBLE and any other display attribute for a field, 4GL ignores the INVISIBLE attribute.

By default, the ERROR statement displays messages in REVERSE. If you specify the INVISIBLE attribute, ERROR displays its message in NORMAL.

The INVISIBLE attribute has no effect on editing BYTE or TEXT fields.

This attribute does *not* prevent a DISPLAY, DISPLAY ARRAY, DISPLAY FORM, MESSAGE, or OPEN WINDOW statement from displaying data in the field.

Specify the INVISIBLE attribute, rather than `COLOR = BLACK`, if you do not want the field to display what the user types during data entry. (The BLACK color attribute displays black characters on a color terminal and displays as DIM on a monochrome terminal.)

Related Attribute

COLOR

NOENTRY

The NOENTRY attribute prevents data entry in the field during an INPUT or INPUT ARRAY statement.

Usage

The following example illustrates the use of the NOENTRY attribute:

```
i13 = stock.stock_num, NOENTRY;
```

When the user enters data in the **stock** table, the **stock_num** column is not available because this SERIAL column gets its value from the database server during the INSERT statement.

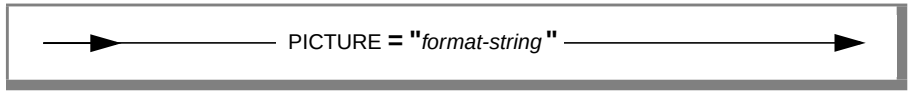
The NOENTRY attribute does *not* prevent data entry into a field during a CONSTRUCT statement (for a query by example).

Related Attribute

INVISIBLE

PICTURE

The PICTURE attribute specifies a character pattern for data entry in a text field and prevents entry of values that conflict with the specified pattern.



Element	Description
format-string	is a string of characters to specify a character pattern for data entry. This string must be enclosed within quotation marks (").

Usage

A *format-string* value can include literals and these three special symbols.

Symbol	Meaning
A	Any letter
#	Any digit
X	Any character

4GL treats any other character in *format-string* as a literal. The cursor skips over any literals during data entry. 4GL displays the literal characters in the display field and leaves blanks elsewhere.

For example, note the following field specification:

```
c10 = customer.phone,
     picture = "###-###-####x#####";
```

It displays these symbols in the **customer.phone** field before data entry:

```
[   -   -    x     ]
```

If the user attempts to enter a character that conflicts with *format-string*, the terminal beeps, and 4GL does not echo the character on the screen.

The *format-string* value must fill the entire width of the display field. The PICTURE attribute, however, does not require data entry into the entire field. It only requires that whatever characters are entered conform to *format-string*.

When PICTURE specifies input formats for DATETIME or INTERVAL fields, FORM4GL does not check the syntax of *format-string*, but your form will work if the syntax is correct. Any error in *format-string*, however, such as an incorrect field separator, produces a runtime error.

As another example, suppose you specify a field for part numbers like this:

```
f1 = part_no, picture = "AA#####-AA(X)";
```

4GL accepts any of the following inputs:

```
LF49367-BB(*)
TG38524-AS(3)
YG67489-ZZ(D)
```

The user does not enter the hyphen or the parentheses, but 4GL includes them in the string that it passes to the program variable.

GLS

The PICTURE attribute is not affected by GLS environment variables because it only formats character information. ♦

Editing Keys During Data Entry

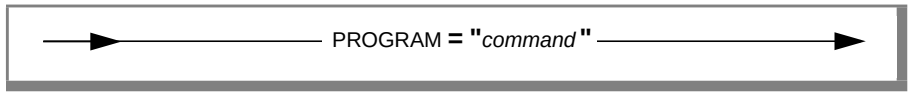
4GL supports pressing CONTROL-X in fields that specify the PICTURE attribute. Pressing CONTROL-X deletes the current character.

Related Attribute

FORMAT

PROGRAM

The PROGRAM attribute can specify an external application program to work with screen fields of data type TEXT or BYTE.



Element	Description
command	is a command string (or the name of a shell script) that invokes an editing program, enclosed within quotation marks.

Usage

You can assign the PROGRAM attribute to a BYTE or TEXT field to call an external program to work with the BYTE or TEXT values. Users can invoke the external program by pressing the exclamation point (!) key while the screen cursor is in a BYTE or TEXT field. The external program then takes over control of the screen. When the user exits from the external program, the form is redisplayed with any display attributes besides PROGRAM in effect. For example, this field description designates vi as the external editor of a multiple-segment TEXT field that also has the WORDWRAP attribute:

```
f010 = personnel.resume, WORDWRAP, PROGRAM = "vi";
```

Here the WORDWRAP attribute (described in "WORDWRAP" on page 6-67) specifies that as much of the TEXT value as possible be displayed in successive segments of the multiple-segment field when the form displays a value in the field, but the WORDWRAP editor cannot edit a TEXT value.

If the cursor enters the field whose tag is **f010** in the same example and presses the ! key, the form is cleared from the screen. Now the user can run the vi utility to view or edit the TEXT value. When the editing session ends, the form is restored on the screen, and control returns to the 4GL application.

When a display field is of data type TEXT, but the screen cursor is not in that field, the 4GL application can display as many of the leading characters of a TEXT data value as can fit in the field. (For BYTE fields, 4GL displays <BYTE value> in the field.) This behavior is independent of the PROGRAM attribute.

Default Editors

If a user moves the cursor to a TEXT field and presses the exclamation point key in the first character position of the field, 4GL attempts to invoke an external program. The program invoked for a TEXT field is chosen from among the following programs, in *descending* order of priority:

- The program (if any) identified by the `PROGRAM ="command"` attribute specification for the field
- The program (if any) named in the **DBEDIT** environment variable
- The default editor, which depends on the host operating system

Specify the editor to use with the **DBEDIT** environment variable; this variable should contain the name of a UNIX application such as **vi** or **emacs**. When the user exits from the editor, control returns to the 4GL screen.

4GL applications that display or modify a value in a BYTE field must use the PROGRAM attribute explicitly to assign an editor. For BYTE fields, the default editor is not called, and the **DBEDIT** variable is not examined.

The Command String

Before invoking the program, your application copies the BYTE or TEXT field to a temporary disk file. It then issues a system command composed of the *command* string that you specify after the PROGRAM keyword, followed by the name of the temporary file.

The *command* string can be longer than a single word. You can add additional command parameters. The *command* string can also be the name of a shell script, so that you can initiate a whole series of actions.

Your 4GL program needs to execute an INSERT or UPDATE statement by using the appropriate program variables after input is terminated. For example, you would use a statement similar to those that follow:

```
INSERT INTO mytable (textcol, bytecol)
   VALUES (p_texdata, p_bytdata)

UPDATE mytable SET (textcol, bytecol) = (p_texdata, p_bytdata)
```

REQUIRED

The REQUIRED attribute forces the user to enter data in the field during an INPUT or INPUT ARRAY statement.

Usage

The REQUIRED keyword is effective only when the field name appears in the list of screen fields of an INPUT or INPUT ARRAY statement. For example, suppose the ATTRIBUTES section includes the following field description:

```
o20 = orders.po_num, REQUIRED;
```

Because of the REQUIRED specification, 4GL requires the entry of a purchase order value when the form is used to collect information for a new order.

You cannot specify a default value for a REQUIRED field. If both the REQUIRED and DEFAULT attributes are assigned to the same field, 4GL assumes that the DEFAULT value satisfies the REQUIRED attribute.

This attribute requires only that the user enter a printable character in the field. If the user subsequently erases the entry during the same input, 4GL considers the REQUIRED attribute satisfied. To insist on a non-null entry, specify that the field is FORMONLY and NOT NULL.

Related Attribute

NOENTRY

REVERSE

The REVERSE attribute displays any value in the field in reverse video (dark characters in a bright field).

Usage

The following example specifies that a field linked to the **customer_num** column displays data in reverse (sometimes called *inverse*) video:

```
f000 = customer.customer_num, REVERSE;
```

On terminals that do not support reverse video, fields that have the REVERSE attribute are enclosed between angle brackets (< >).

The REVERSE attribute disables any other COLOR attribute for the same field.

Related Attribute

COLOR

UPSHIFT

During data entry in a character field, the UPSHIFT attribute converts lowercase letters to uppercase letters, both on the screen display and in the 4GL program variable that stores the contents of that field.

Usage

Because uppercase and lowercase letters have different code-set values, storing all character strings in one or the other format can simplify sorting and querying a database.

The following example includes UPSHIFT in the attribute list of a field:

```
c8  = state, UPSHIFT, AUTONEXT,
      INCLUDE = ("CA", "OR", "NV", "WA"),
      DEFAULT = "CA" ;
```

Because of the UPSHIFT attribute, 4GL enters uppercase characters in the **state** field regardless of the case used to enter them.

The AUTONEXT attribute tells 4GL to move automatically to the next field once you type the total number of characters allowed for the field (in this instance, two characters). The INCLUDE attribute restricts entry in this field to the characters CA, OR, NV, or WA only. The DEFAULT value for the field is CA.

GLS

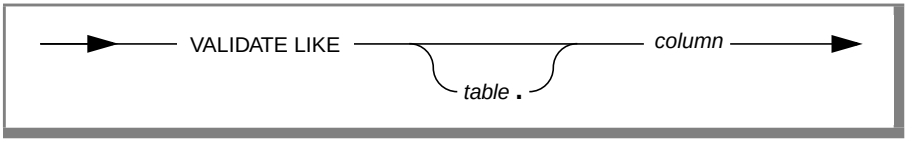
The results of conversion between uppercase and lowercase letters are based on the locale files, which specify the relationship between corresponding pairs of uppercase and lowercase letters. If the locale files do not provide this information, no case conversion occurs. ♦

Related Attribute

DOWNSHIFT

VALIDATE LIKE

The VALIDATE LIKE attribute instructs 4GL to validate the data entered in the field by using the validation rules that the **upscol** utility assigned to the specified database column in the **syscolval** table.



Element	Description
column	is the name of a column in *table*, or (if you omit *table*) the unique identifier of a column in one of the tables that you declared in the TABLES section.
table	is the unqualified name or alias of a database table, synonym, or view, as declared in the TABLES section. (This value is not required unless several columns in different tables have the same name or if the table is an external table or an external, distributed table.)

Usage

This attribute is equivalent to listing all the attributes that you have assigned to *table.column* in the **syscolval** table. "Default Attributes" on page 6-80 describes the **syscolval** table and the effects of this table in an ANSI-compliant database. The following example assigns the default attributes of the customer.state column to a FORMONLY field:

```
s13 = FORMONLY.state, VALIDATE LIKE customer.state;
```

The restrictions on the DISPLAY LIKE attribute also apply to this attribute. You do not need the VALIDATE LIKE attribute if *table.column* is the same as *field name*. You cannot specify a column of data type BYTE as *table.column*. Even if all of the fields in the form are FORMONLY, this attribute requires FORM4GL to access the database that contains *table*.

Related Attribute

DISPLAY LIKE

VERIFY

The VERIFY attribute requires users to enter data in the field twice to reduce the probability of erroneous data entry.

Usage

Because some data is critical, this attribute supplies an additional step in data entry to ensure the integrity of your data. After the user enters a value into a VERIFY field and presses RETURN, 4GL erases the field and requests reentry of the value. The user must enter exactly the same data each time, character for character: `15000` is not exactly the same as `15000.00`.

For example, if you specify a field for salary information in the following way, 4GL requires entry of exactly the same data twice:

```
s10 = quantity, VERIFY;
```

An error message appears if the user does not enter the same keystrokes.

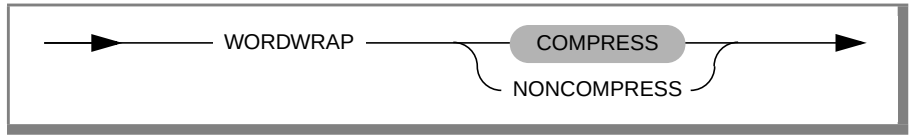
The VERIFY attribute takes effect while INPUT, INPUT ARRAY, or UPDATE statements of 4GL are executing. It has no effect on CONSTRUCT statements.

Related Attributes

INCLUDE, REQUIRED, VALIDATE LIKE

WORDWRAP

In a multiple-segment field, the WORDWRAP attribute enables a multiple-line editor. This editor can *wrap* long character strings to the next line of a multiple-segment field for data entry, data editing, and data display.



Usage

If the same field tag is repeated in two or more locations in the screen layout, this attribute instructs 4GL to treat all the instances of that field tag as successive segments of a *multiple-segment field* (described in "Multiple-Segment Fields" on page 6-31). These fields can display data strings that are too long to fit on a single line of the screen form. For example, the following excerpt from a form specification file shows a VARCHAR field linked to the **history** column in the **employee** table.

```
history     [f002                                                ]
            [f002                                                ]
            [f002                                                ]

attributes
f002 = employee.history, WORDWRAP COMPRESS;
```

4GL replaces each set of multiple-segment fields with a single WORDWRAP field of a rectangular shape. The COMPRESS keyword option is applied to this field, and the delimiters are replaced with blank spaces.

When a variable is bound to the WORDWRAP field during INPUT, only the number of characters allowed by the bound variable can be entered. If necessary, text in the field scrolls to allow the full number of characters to be entered. Data compression takes place before storage in the bound variable.

If the lines of the multiple-segment field are not contiguous or if the field has an irregular shape, the WORDWRAP field that results is based on the maximum height and width of the multiple-segment field as a unit.

A multiple-segment field with an irregular shape is shown in Figure 6-3.

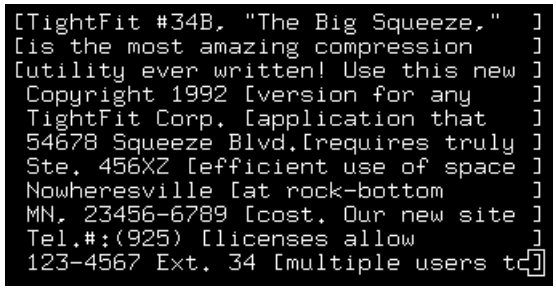


Figure 6-3
Example of an Irregularly Shaped Field

The resulting WORDWRAP field can overlap or be overlapped by labels or individual form fields. To prevent such unpredictable effects, consolidate the segments of multiple-segment fields into rectangular shapes.

Data Entry and Editing with WORDWRAP

When text is entered into a multiple-segment WORDWRAP field, 4GL breaks character strings into segments at blanks (if it can) and pads field segments with blanks to the right. Where possible, contiguous nonblank substrings (*words*) within a string are not broken at field segment boundaries.

When keyboard input reaches the end of a line, the multiple-line editor brings the current word down to the next field segment and moves text down to subsequent lines as necessary. (The next field segment is determined by the left-to-right, top-to-bottom order of field segments within the screen layout.) When the user deletes text, the editor pulls words up from lower field segments whenever it can.

GLS

In nondefault locales, WORDWRAP fields can process non-ASCII characters that the locale supports. For the special case of Japanese locales, WORDWRAP fields support Kinsoku processing, with the same features that are described in "Kinsoku Processing with WORDWRAP" on page 7-66. ♦

Data Display with WORDWRAP

If a CHAR, VARCHAR, or TEXT value is displayed in a WORDWRAP field, 4GL puts the first data character in the first character position of the first segment and displays consecutive data characters in successive positions to the right.

If the entire data string is too long to fit in the first field segment, 4GL continues the display in the next field segment, dividing the data string at blank characters. This process continues until all the field segments are filled or until the end of the data string is reached.

The WORDWRAP attribute displays a TEXT field so that it fits into the form without any field segments that begin with a blank. For a TEXT field, the WORDWRAP attribute only affects how the value is displayed; WORDWRAP does not enable the multiple-line editor. To let users edit a TEXT field, you must use the PROGRAM attribute to indicate the name of an external editor.

Displaying Program Variables with WORDWRAP

Text in WORDWRAP fields can include printable ASCII characters, the TAB (ASCII 9) character, and the newline (ASCII 10) character. These characters are retained in the program variable. Other nonprintable characters might result in runtime errors. The TAB character aligns the display at the next tab stop, and the newline character continues the display at the start of the next line. By default, tab stops are in every eighth column, beginning at the left edge of the field. In non-English locales, WORDWRAP fields can include printable characters that the locale supports.

Ordinarily, the length of the variable should not be greater than the total length of all the field segments. If the data string is longer than the field (or if too much padding is required for WORDWRAP), 4GL fills the field and discards the excess data. This process displays a long variable in summary form. If a truncated value is used to update the database, however, characters are lost.

The editor distinguishes between *intentional* blanks (from the database or typed by the user) and *editor* blanks (inserted at the ends of lines for word-wrap or to align after a newline character). Intentional blanks are retained as part of the data. Editor blanks are inserted and deleted automatically as required. When designing a multiple-segment field, allow room for editor blanks, over and above the data length. The expected number of editor blanks is half the length of an average word per segment. Text that requires more space than you expect might be truncated after the final field segment.

The COMPRESS and UNCOMPRESS Options

The COMPRESS keyword prevents blanks produced by the editor from being included in the program variable.

COMPRESS is applied by default and can cause truncation to occur if the sum of intentional characters exceeds the field or column size. Because of editing blanks in the WORDWRAP field, the stored value might not correspond exactly to its multiple-line display, so a 4GL report generally cannot display the data in identical form.

To suppress COMPRESS, specify UNCOMPRESS after the WORDWRAP keyword. This option causes any editor blanks to be saved when the WORDWRAP string is saved in a database column, in a variable, or in a file.

In the following fragment of a form specification file, a CHAR value in the column **charcolm** is displayed in the multiple-segment field whose tag is **mlf**:

```
SCREEN SIZE 24 by 80
{
Enter text:
   [mlf                                              ]
   [mlf                                              ]
         . . .
   [mlf                                              ]
   [mlf                                              ]
}

TABLES  tablet . . .

ATTRIBUTES
   mlf = tablet.charcolm, WORDWRAP COMPRESS;
```

If the data string is too long to fit in the first line, successive segments are displayed in successive lines, until all of the lines are filled or until the last text character is displayed (whichever happens first).

If the form is used to insert data into **tablet.charcolm**, the keyword COMPRESS specifies that 4GL will not store editor blanks.

WORDWRAP Editing Keys

When data is entered or updated in a WORDWRAP field, the user can use keys to move the screen cursor over the data and to insert, delete, and type over the data. The cursor never pauses on editor blanks.

The editor has two modes, *insert* (to add data at the cursor) and *typeover* (to replace existing data with entered data). You cannot overwrite a newline character. If the cursor in typeover mode encounters a newline character, the cursor mode automatically changes to insert, *pushing* the newline character to the right. Some keystrokes behave differently in the two modes.

When the cursor first enters a multiple-segment field, it is positioned on the first character of the first field segment, and the editing mode is set to *typeover*. The cursor movement keys are as follows:

- RETURN leaves the entire multiple-segment field and goes to the first character of the next field.
- BACKSPACE or LEFT ARROW moves the cursor left one character, unless at the left edge of a field segment. From the left edge of the first segment, these keys either move the cursor to the first character of the preceding field or only beep, depending on whether INPUT WRAP is in effect. (Input wrap mode is controlled by the OPTIONS statement.) From the left edge of a lower field segment, these keys move the cursor to the right-most intentional character of the previous field segment.
- RIGHT ARROW moves the cursor right one character, unless at the right-most intentional character in a segment. From the right-most intentional character of the last segment, this key either moves the cursor to the first character of the next field or only beeps, depending on INPUT WRAP mode. From the right-most intentional character of a higher segment, this key moves the cursor to the first intentional character in a lower segment.

- UP ARROW moves the cursor from the top-most segment to the first character of the preceding field. From a lower segment, this key moves the cursor to the character in the same column of the next higher segment, jogging left, if required, to avoid editor blanks, or if it encounters a tab.
- DOWN ARROW moves the cursor from the lowest segment to the first character of the next field. From a higher segment, this key moves the cursor to the character in the same column in the next lower segment, jogging left if required to avoid editor blanks, or if it encounters a tab.
- TAB enters a TAB character in insert mode and moves the cursor to the next tab stop. This action can cause following text to jump right to align at a tab stop. In typeover mode, this key moves the cursor to the next tab stop that falls on an intentional character, going to the next field segment if required.

The character keys enter data. Any following data shifts right, and words can move down to subsequent segments. This action can result in characters being discarded from the final field segment. These keystrokes can also alter data:

- CONTROL-A switches between typeover and insert mode.
- CONTROL-X deletes the character under the cursor, possibly causing words to be pulled up from subsequent segments.
- CONTROL-D deletes all text from the cursor to the end of the multiple-line field (not merely to the end of the current field segment).
- CONTROL-N inserts a newline character, causing subsequent text to align at the first column of the next segment of the field and possibly moving words down to subsequent segments. This action can result in characters being discarded from the final segment of the field.

Non-WORDWRAP Displays

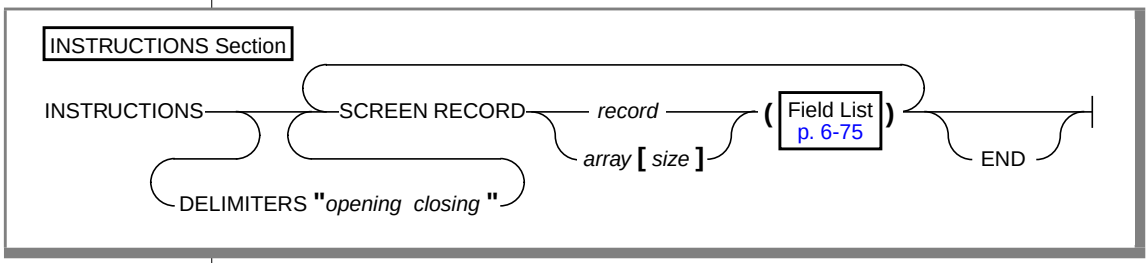
The appearance of a character value on the screen can vary, depending on whether or not it is displayed in a multiple-segment WORDWRAP field. For instance, if a value that was entered by using WORDWRAP is displayed without this attribute, words will generally be broken, not wrapped, and TAB and NEWLINE characters will be displayed as question marks. These differences do not represent any loss of data but only a different mode of display. *(*You can view this effect, for example, if you also have INFORMIX-SQL installed on your system, and you use the **Query Language** menu to display character data values that were entered using WORDWRAP.*)*

If a value prepared under the multiple-line editor is again edited without WORDWRAP, however, some formatting might be lost. For example, a user might type over a TAB or NEWLINE character, not realizing what it was. Similarly, a user might remove a blank from the first column of a line and thus join a word to the last word on the previous line. These mistakes will be visible when the value is next displayed in a WORDWRAP field or in a 4GL report that uses the WORDWRAP operator.

INSTRUCTIONS Section

The INSTRUCTIONS section is the optional final section of a form specification file. This section can declare nondefault screen records and screen arrays.

The INSTRUCTIONS section appears after the last field description (or after the optional END keyword) of the ATTRIBUTES section.



Element	Description
array	is the 4GL identifier that you declare here for the screen array. (It is also the name of the screen record that comprises each line of the array.)
closing	is the closing field delimiter.
opening	is the opening field delimiter.
record	is the 4GL identifier that you declare here for the screen record.
size	is a literal integer, enclosed in brackets ([ ]), to specify the number of screen records in the screen array.

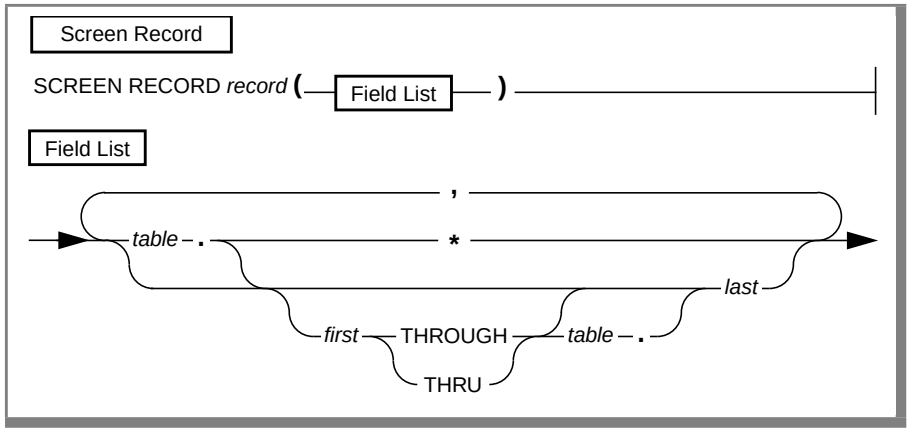
The END keyword is optional and provides compatibility with earlier Informix products.

Screen Records

A *screen record* is a group of fields that screen interaction statements of the 4GL program can reference as a single object. By establishing a correspondence between a set of screen fields (the screen record) and a set of 4GL variables (typically a program record), you can pass values between the program and the fields of the screen record. In many applications, it is convenient to define a screen record that corresponds to a *row* of a database table.

Nondefault Screen Records

The INSTRUCTIONS section of a form specification file can declare nondefault screen records. You use the SCREEN RECORD keywords of the INSTRUCTIONS section to declare a name for the screen record and to specify a list of fields that are members of the screen record. A record declaration has this syntax.



Element	Description
first	is a field name that you declared in the ATTRIBUTES section.
last	is a field name that you declared later than *first*.
record	is the 4GL identifier that you declare for the screen record.
table	is the name, alias, or synonym of a table (or FORMONLY keyword).

The field name is the SQL identifier of a database column linked to the field, unless you specify FORMONLY as the table reference. If the database server has the **IFX_LONGID** environment variable set to 1, then table identifiers can require up to 128 bytes of storage.

The *record* name of a nondefault screen record can require up to 128 bytes of storage, and must comply with the rules for 4GL identifiers (as described in "4GL Identifiers" on page 2-14).

Default Screen Records

4GL recognizes *default screen record*s that consist of all the screen fields linked to the same database table within a given form. FORM4GL automatically creates a *default record* for each table that is used to reference a field in the ATTRIBUTES section. The components of the default record correspond to the set of display fields that are linked to columns in that table.

If the database server has the **IFX_LONGID** environment variable set to 1, then a table identifier (for the name of a default screen record) or column identifiers (for member fields within a default screen record) can require up to 128 bytes of storage.

The name of the default screen record is the table name (or the alias, if you declared an alias for that table in the TABLES section). For example, all the fields linked to columns of the **customer** table constitute a default screen record whose name is **customer**. If a form includes one or more FORMONLY fields, those fields constitute a default screen record called **formonly**.

Like the name of a screen field, the identifier of a screen record must be unique within the form, and it has a scope that is restricted to when its form is open. Statements can reference *record* only when the screen form that includes it is being displayed. FORM4GL returns an error if *record* is the same as the name or alias of a table in the TABLES section.

The List of Member Fields

The fields within a screen record are called members of the record. The list of member fields must be enclosed within a pair of parentheses. Use commas to separate elements of the list of field names.

You must specify the *table* qualifier if the field name is not unique among the fields in the ATTRIBUTES section or if table is a required alias (as described in "TABLES Section" on page 6-23). Otherwise, *table* is optional but including it might make the form specification file easier to read.

A screen record can include screen fields whose identifiers have different *table* specifications (including the FORMONLY keyword). You can use the notation *table*.* to include default screen records in the list of fields:

```
SCREEN RECORD worlds_record
      (items.*, customer.*, state.code, FORMONLY.total)
```

Here the asterisks (*) represent all of the fields in the form that the ATTRIBUTES section associated with columns in the **items** and **state** tables. These fields do not necessarily correspond to all of the columns in these tables unless the form includes fields that are linked to all of the columns.

You can use the keyword THRU to specify consecutive fields, in the order of their listing in the ATTRIBUTES section from *field name1* to *field name2*, inclusive. (The keyword THROUGH is a synonym for THRU.) For example, the following instruction creates a screen record called **address** from fields linked to some columns of the **customer** table. This record can simplify 4GL statements to update customer address and telephone data.

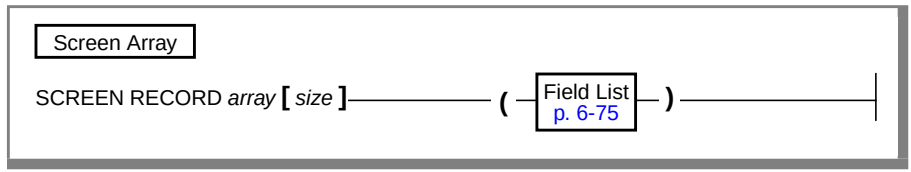
```
SCREEN RECORD address
      (customer.address1 THRU customer.phone)
```

The order of fields in the portion of a screen record specified by the *table*.* or THRU notation is the order of the field names within the ATTRIBUTES section.

Screen Arrays

A *screen array* is usually a repetitive array of fields in the screen layout, each containing identical groups of screen fields. Each *row* of a screen array is a screen record. Each *column* of a screen array consists of fields with the same field tag in the SCREEN section of the form specification file.

You must declare screen arrays in the INSTRUCTIONS section and use syntax like the syntax described for a screen record in the previous section, but with an additional parameter to specify the *number of screen records* in the array.



Element	Description
array	is the 4GL identifier that you declare here for the screen array. (It is also the identifier of the screen record that comprises each line of the array.)
size	is a literal integer, enclosed in square brackets ([ ]), to specify how many screen records are in the screen array.

The *size* value should be the number of lines in the logical form where the set of fields that comprise each screen record is repeated within the screen array. For example, a SCREEN section might represent a screen array like this:

```
SCREEN
{
    CARRIER    FLIGHT    ARRIVES   DEPARTS
   [ f00001]  [f00002]  [ f0003]  [ f0004]
   [ f00001]  [f00002]  [ f0003]  [ f0004]
   [ f00001]  [f00002]  [ f0003]  [ f0004]
}
```

This example requires a *size* of **[3]**. Except for the *size* parameter, syntax for specifying the identifier and the field names of a screen array is the same as for a simple screen record (as described in "Nondefault Screen Records" on page 6-75). Unlike 4GL program arrays, which can have up to three dimensions, every 4GL screen array has exactly one dimension.

The next example declares an array of six records, each of which includes two default screen records, namely the **manufact.*** and **state.*** screen records:

```
SCREEN RECORD mant_array [6]
     (manufact.*, state.*, cust_calls.user_id, FORMONLY.delta)
```

To illustrate the declaration of a typical screen array in more detail, consider the following fragment of a form specification file:

```
SCREEN
{
...
Item 1 [p     ][q     ][u     ][t       ]
Item 2 [p     ][q     ][u     ][t       ]
Item 3 [p     ][q     ][u     ][t       ]
Item 4 [p     ][q     ][u     ][t       ]
Item 5 [p     ][q     ][u     ][t       ]
}
TABLES orders items stock
ATTRIBUTES
...
p = stock.stock_num;
q = items.quantity;
u = stock.unit_price;
t = items.total_price;
...
INSTRUCTIONS
SCREEN RECORD sc_items[5] (stock.stock_num,
          items.quantity, stock.unit_price,
          items.total_price)
```

The **sc_items** screen array has five rows and four columns and includes fields linked to columns from two database tables. Rows are numbered from 1 to 5. The screen record that follows the display label Item 3 in the screen layout, for example, can be referenced as **sc_items[3]** in a 4GL statement.

If there are no other columns of the **items** table in the form, the default screen record **items** contains two fields, which correspond to the **items.quantity** and **items.total_price** fields that are linked to columns of the **items** table.

If a screen array contains a default screen record, you can reference its fields in specific lines of the screen array (such as **items[5]** for the **q** and **t** fields in the last line), as if you had declared an array of records linked to that table.

You can reference *array-name* in the DISPLAY, DISPLAY ARRAY, INPUT, INPUT ARRAY, and SCROLL statements of 4GL but only when the screen form that includes the screen array is the current form.

Screen records and screen arrays can display program records. If the fields in the screen record have the same sequence of data types as the columns in a database table, you can use the screen record to simplify 4GL operations that pass values between program variables and rows of the database.

Field Delimiters

You can change the delimiters that 4GL uses for fields from brackets ([ ]) to any other printable character, including blank spaces. The DELIMITERS instruction tells 4GL what symbols to use as field delimiters when it displays the form on the screen. The opening and closing delimiter marks must be enclosed within quotation marks (").

The following specifications display < and > as opening and closing delimiters of screen fields:

```
INSTRUCTIONS
    DELIMITERS "<>"
END
```

Each delimiter occupies a space, so two fields on the same line are ordinarily separated by at least two spaces. To specify only one space between consecutive screen fields, use the following procedure.

To use only one space between fields

1. In the SCREEN section, substitute a pipe symbol (|) for paired back-to-back ( ][ ) brackets that separate adjacent fields.
2. In the INSTRUCTIONS section, define some symbol as both the beginning and the ending delimiter.

   For example, you could specify "| |" or "/ /" or ": :" or " " (blanks).

The following specifications substitute | for][between adjacent fields in the same line of the screen layout and display a colon as both the opening and closing delimiter:

```
SCREEN
   {
          . . .
   Full Name-[f011       |f012       ]
          . . .
   }
          . . .
INSTRUCTIONS
   DELIMITERS "::"
```

Here the fields whose tags are **f011** and **f012** will be displayed as:

```
Full Name-:           |           :
```

If you substitute blanks for colons as DELIMITERS symbols, field boundaries are not marked (or are only marked if they have attributes that contrast with the surrounding background).

Important: *FORM4GL requires brackets ([ ]) in the SCREEN section of a form specification file, regardless of any DELIMITERS instruction.*

Default Attributes

Field attributes can also be specified in two special tables in the database, **syscolval** and **syscolatt**. These tables are maintained by the **upscol** utility, as described in Appendix B. FORM4GL searches these tables for default validation and display attribute specifications. It applies these specifications to form fields whose names match the names of the specified database columns or that reference these columns in the DISPLAY LIKE or VALIDATE LIKE attribute specifications.

The schema of the **syscolval** table is as follows.

Column	Data Type	Column	Data Type
tabname	CHAR(18)	**attrname**	CHAR(10)
colname	CHAR(18)	**attrval**	CHAR(64)

The schema of the **syscolatt** table is as follows.

Column	Data Type	Column	Data Type
tabname	CHAR(18)	**underline**	CHAR(1)
colname	CHAR(18)	**blink**	CHAR(1)
seqno	SERIAL	**left**	CHAR(1)
color	SMALLINT	**def_format**	CHAR(64)
inverse	CHAR(1)	**condition**	CHAR(64)

Here **tabname** and **colname** are the names of the table and column to which the attributes apply. Here **colname** cannot be a BYTE nor TEXT column. Valid values for the **attrname** and **attrval** columns in **syscolval** are as follows.

attrname	attrval
AUTONEXT	YES, NO (the default)
COMMENTS	As in this chapter
DEFAULT	As in this chapter
INCLUDE	As in this chapter
PICTURE	As in this chapter
SHIFT	UP, DOWN, NO (the default)
VERIFY	YES, NO (the default)

FORM4GL adds the attributes from these tables to any attributes that are listed in the form specification file. In case of conflict, attributes from the form specification file take priority. 4GL applies the resulting set of field attributes during execution of INPUT and INPUT ARRAY statements (by using **syscolval**) and during execution of DISPLAY and DISPLAY ARRAY statements (by using **syscolatt**).

The **color** column in **syscolatt** stores an integer that describes color (for color terminals) or intensities (for monochrome terminals).

The next table shows the displays specified by each value of **color** and the correspondence between default color names, number codes, and intensities.

Number	Color Terminal	Monochrome Terminal
0	WHITE	NORMAL †
1	YELLOW	BOLD
2	MAGENTA	BOLD
3	RED	BOLD †
4	CYAN	DIM
5	GREEN	DIM
6	BLUE	DIM †
7	BLACK	DIM

† If BOLD is specified as the attribute, the field is displayed as RED on a color screen. If the keyword DIM is specified as the attribute, the field is displayed as BLUE on a color screen. Color terminals display NORMAL as WHITE.

The background for colors is BLACK in all cases. The same keywords are also supported by the COLOR attribute for color and monochrome terminals.

The valid values for **inverse**, **underline**, **blink** and **left** are `Y` (yes) and `N` (no). The default for each of these columns is `N`; that is, normal display (bright characters in a dark field), no underline, steady font, and right-aligned numbers. Which of these attributes can be displayed simultaneously with the color combinations or with each other is terminal dependent.

The **def_format** column takes the same string that you would enter for the FORMAT attribute in a screen form. Do not use quotation marks.

The **condition** column takes string values that are a restricted set of the WHERE clauses of a SELECT statement, except that the WHERE keyword and the column name are omitted. 4GL assumes that the value in the column identified by **tabname** and **colname** is the subject of all comparisons.

Examples of valid entries for the **condition** column follow:

```
<= 100 MATCHES "[A-M]*"        BETWEEN 101 AND 1000
IN ("CA", "OR", "WA")          >= 1001 NOT LIKE "%analyst%"
```

The VALIDATE statement (described in Chapter 3) compares the members of a program record or variable list to the validation rules in **syscolval**. The INITIALIZE statement (described in Chapter 4) can read the default values in **syscolval** for a list of columns and assign these values to a corresponding list of 4GL variables.

Some statements (including CONSTRUCT, DISPLAY, DISPLAY ARRAY, ERROR, INPUT, INPUT ARRAY, MESSAGE, PROMPT, OPEN WINDOW, and OPTIONS) support an ATTRIBUTE clause (described in "ATTRIBUTE Clause" on page 3-96) that can specify color and intensity attributes.

You can override default attributes in **syscolatt** by assigning other attributes in the form specification file or in the ATTRIBUTE clause of the CONSTRUCT, DISPLAY, DISPLAY ARRAY, INPUT, or INPUT ARRAY statement. If the current 4GL statement is one of these and includes an ATTRIBUTE clause, the field displays only the attributes that are specified in that clause. For example, if a column is designated as RED and BLINK in **syscolatt**, or in the form specification file, and your 4GL program executes the following statement, the field has only the BLUE attribute, not blinking BLUE:

```
DISPLAY . . . ATTRIBUTE BLUE
```

If an ATTRIBUTE clause is present in the currently executing statement, there is no implicit carry-over of display attributes from the compiled form (except FORMAT).

Precedence of Field Attribute Specifications

4GL uses these rules of precedence (highest to lowest) to resolve any conflicts among multiply defined display attribute specifications:

1. The ATTRIBUTE clause of the current 4GL statement
2. The field descriptions in the ATTRIBUTES section of the current form
3. The default attributes specified in the **syscolatt** table of any fields linked to database columns

   To modify the **syscolatt** table, use the **upscol** utility. For information on using this utility, see Appendix B.
4. The ATTRIBUTE clause of the most recent OPTIONS statement

5. The ATTRIBUTE clause of the current form in the most recent DISPLAY FORM statement
6. The ATTRIBUTE clause of the current 4GL window in the most recent OPEN WINDOW statement

ANSI

Default Attributes in an ANSI-Compliant Database

In a database that is not ANSI compliant, the default screen attributes and validation criteria that you specify with the **upscol** utility are stored in two tables, **syscolval** and **syscolatt**. These defaults are available to every user of a form that references the specified column of the database.

In an ANSI-compliant database, however, the separate *owner***.syscolval** and *owner***.syscolatt** tables are created for each user of the **upscol** utility. These tables store the default specifications of that individual user. Which set of tables is used by FORM4GL depends on the nature of the request.

If the TABLES section specifies a table alias for *owner.table*, FORM4GL uses the **upscol** tables of the owner of *table*. If that user owns no **upscol** tables, no defaults are assigned to fields associated with that table alias. If the TABLES section of the form does not specify a table alias that includes the owner of a database table, the **upscol** tables owned by the user running FORM4GL are applied to fields associated with that database table unless the user owns no **upscol** tables. In the ATTRIBUTES section, field descriptions of the following forms use **upscol** tables (if they exist) owned by whoever runs FORM4GL, unless *table* is an alias that specifies a different owner:

```
field-tag =  . . .  DISPLAY  LIKE  table.column
field-tag =  . . .  VALIDATE  LIKE  table.column
```

If *table* is an alias for *owner.table*, FORM4GL uses the **upscol** tables of the owner specified by *table*, if they exist. If no **upscol** tables exist, the DISPLAY LIKE and VALIDATE LIKE attributes have no effect. If *owner* is not the correct owner, the compilation fails and an error message is issued. See also the INITIALIZE and VALIDATE statements in Chapter 4.

Creating and Compiling a Form

For your 4GL program to work with a screen form, you must create a form specification file that conforms to the syntax described earlier in this chapter, and then compile the form. You can compile the form in one of two ways: from within the Programmer's Environment or at the command line. Both methods require that the database and any tables referenced in the form already exist, and that the database server be running and able to access the database. These methods of compiling a form are described in this section. Also, a section on using default forms is included.

Compiling a Form Through the Programmer's Environment

The Programmer's Environment is a system of menus that supports the steps in the process of developing 4GL application programs. The Programmer's Environment is described in Chapter 1.

To create a screen form in the Programmer's Environment

1. At the system prompt, enter one of the following commands:
   - If you have the C Compiler version, enter `i4gl`.
   - If you have the Rapid Development System, enter `r4gl`.
2. Press `F` at the **INFORMIX-4GL** menu to select the **Form** option.
3. Press `G` to select the **Generate** from the **Form** menu.

   (Alternatively, you can select the **New** option. 4GL then prompts you for a form name, prompts you for an editor if you have not already selected one, and invokes that editor with an empty form specification file. Now you must enter form specifications. The **Generate** option is usually a more efficient way to create a customized form.)
4. Enter the name of the database and the name that you want to assign to the form (for example, **myform**).

   4GL asks you for the names of the tables whose columns you want in your form.

5. Enter the names of the tables you want to use.

   After you select the tables, FORM4GL creates a default form specification file, as well as a compiled default form, and then displays the **FORM** design menu.

   The default form specification file formats the screen as a list of all the columns in the tables that you entered in step 4. It does not provide any special instructions to 4GL about how to display the data.

6. Press M to select the **Modify** option, and 4GL presents the **MODIFY FORM** menu:

   a. Select the default form specification (given as **myform** earlier), and 4GL calls a system editor to display the file.

   b. Edit the default form specification file to produce your customized screen form and associated instructions.

      (You can specify an editor by using the **DBEDIT** environment variable. This strategy is fully explained in Appendix D, "Environment Variables.")

   c. Save your file and quit the editor, and 4GL returns you to the **MODIFY FORM** menu.

7. Press C to select **Compile**.

   If your form specification file compiles successfully, FORM4GL creates a form file with the extension **.frm** (for example, **myform.frm**). In this case, skip to step 9. If your form specification file does not compile successfully, go on to step 8.

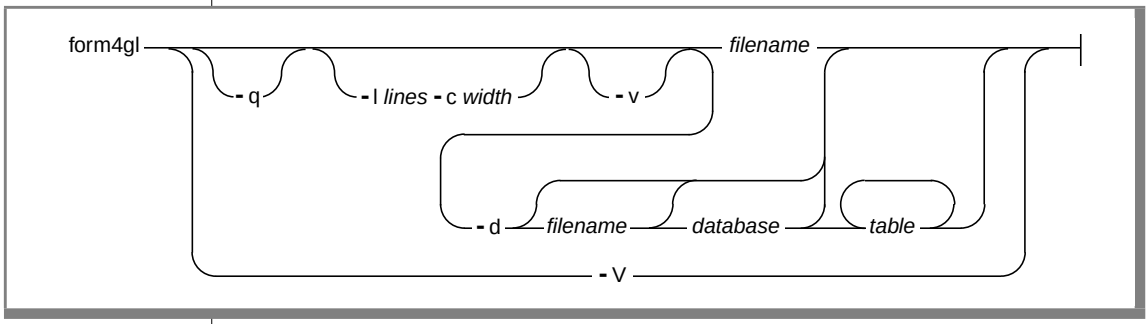
8. Press RETURN to select the **Correct** option from the **COMPILE FORM** menu.

   4GL again calls your editor to display the form specification file, with the compilation errors marked. When correcting your errors, you need not delete the error messages. 4GL does that for you. Save the file and go back to step 7.

9. Save your form specification file with the **Save-and-exit** option.

Compiling a Form at the Command Line

The FORM4GL command line has the following syntax.



Element	Description
database	is the SQL identifier of a database.
filename	is the name of a form specification file (with no **.per** extension).
lines	is an integer that specifies the height of the form in lines of characters that the terminal can display. (The default is 24.)
table	is the name of a database table, as declared in the TABLES section.
width	is an integer that specifies the width of the form in characters. (The default is the number of characters in the longest line of the screen layout, as specified in the SCREEN section.)

The **-v** option instructs the form compiler to verify that all fields are as wide as any corresponding character fields that the ATTRIBUTES section specifies.

The **-V** option instructs the form compiler to display the version number and then exit, without compiling anything.

Use the **-d** option to generate a default form specification file. If you use this option, the compiler prompts you for the names of your form file, database, and tables. For more information, see the next section, "Default Forms."

To create a customized screen form directly from the operating system

1. Create a default form specification file by entering the following command at the operating system prompt:

   ```
   form4gl -d
   ```

   FORM4GL asks for the name of your form specification file, the name of your database, and the name of a table whose columns you want in your form. It continues to ask for another table name until you press RETURN for the name of a table. FORM4GL then creates a default form specification file and appends the extension **.per** to its name. It also creates a compiled default form with the extension **.frm**.

2. Use the system editor to modify the default form specification file to meet your specifications.

   If, as an alternative, you create a new form specification file and skip step 1, be sure to give the filename the extension **.per**.

3. Enter a command of the form:

   ```
   form4gl myform
   ```

   Here **myform** is the name of your form specification file (without the **.per** extension).

   If the compilation is successful, FORM4GL creates a compiled form file called **myform.frm** and you are finished creating your customized screen form. If not, FORM4GL instead creates a file named **myform.err**, and you need to go on to step 4.

4. Review the file **myform.err** to discover the compilation errors.

5. Make corrections in the file **myform.per**, and then go back to step 3.

Default Forms

For many applications, it is convenient to create a default form and then edit this form to satisfy your specific application requirements. When you create a default form, you must specify its filename, a database name, and the name of at least one table whose columns are to be linked to fields in the form.

The width of a display field is the number of characters that can be placed between the delimiters. In a default form specification, FORM4GL assigns lengths to fields according to the declared data type of the column.

Data Type	Default Field Width (in Characters)
BYTE	12
CHAR , NCHAR	MIN (57, *n*), where *n* is the length from the data-type declaration
DATE	10
DATETIME	From 2 to 25, as implied in data-type declaration (Each *unit of time* = 2 (except YEAR and FRACTION); every separator = 1.)
DECIMAL	(2 + *m*), where *m* is the precision from the data type declaration
FLOAT	14
INTEGER	11
INTERVAL	From 3 to 25 (as implied in data type declaration, plus one)
MONEY	(3 + *m*), where *m* is the precision from the data type declaration
SMALLINT	6
SMALLFLOAT	14
TEXT	12
VARCHAR, NVARCHAR	MIN (57, *n*), where *n* is the maximum declared length

SERIAL columns are linked to INTEGER fields. Field length is not directly related to the data display in BYTE and TEXT fields, both of which require the PROGRAM attribute to invoke an external program or editor. The 4GL form can display as many TEXT characters as fit in the field and displays the string `<BYTE value>` in a BYTE field. (For details about how to display BYTE and TEXT values, see "PROGRAM" on page 6-60.)

If you edit a default form, make sure that the fields are wide enough to accommodate the widest value that might be entered or displayed. To prevent 4GL from truncating displayed data, follow these rules:

- Make character fields as wide as the corresponding database column. You can use multiple-segment fields to display long strings (as described in "Multiple-Segment Fields" on page 6-31).
- Make number, DATETIME, and INTERVAL fields wide enough to accommodate the largest displayed value.

Default field tags like **f000** are assigned to the first display field, **f001** to the second, and so on, by FORM4GL. It assigns a field tag like **a0** to any two- or three-character field that cannot accommodate a four-character default field tag. Up to 26 single-character fields can be assigned the single characters **a**, **b**, **c**, and so forth, as default field tags.

The default screen layout has as many lines as the number of columns in the tables. Each line of the screen layout contains a single field, beginning in the 20th character position. FORM4GL uses column names as default field labels, which appear at the left of each field. The next example shows a default form that is based only on the **customer** table of the **stores7** database:

```
database stores7
screen size 24 by 80
{
customer_num       [f000       ]
fname              [f001           ]
lname              [f002           ]
company            [f003               ]
address1           [f004               ]
address2           [f005               ]
city               [f006           ]
state              [a0]
zipcode            [f007 ]
phone              [f008             ]
}
end
tables
customer
attributes
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
```

```
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state;
f007 = customer.zipcode;
f008 = customer.phone;
end
```

If the number of fields is greater than (`lines - 4`), you must edit the default file, either to increase the *lines* value after the SIZE keyword (if the screen size permits this) or else to reduce the number of lines in the screen layout.

Using PERFORM Forms in 4GL

The syntax of FORM4GL forms is different in several significant ways from the syntax of PERFORM, the screen form generation utility of INFORMIX-SQL. You can use PERFORM forms with 4GL, but you must first recompile them using FORM4GL. In addition, not all PERFORM features are operative. You must also use 4GL user-interaction statements like OPEN FORM, OPEN WINDOW, INPUT, DISPLAY FORM, CLEAR FORM, and CONSTRUCT to write a *form driver* to support data entry and data display through the 4GL form.

If you have designed forms for the PERFORM screen transaction program of INFORMIX-SQL, you need to know how those forms behave when used with 4GL. The following features differ from PERFORM to 4GL:

- Only the DELIMITERS keyword in the INSTRUCTIONS section of a PERFORM form is supported by 4GL. Other keywords in that section are ignored. To support other INSTRUCTIONS features of PERFORM requires coding in your 4GL program. (See the BEFORE and AFTER clauses of the INPUT statement.)
- 4GL does not support multiple-page forms (those with more than one screen layout); these forms produce undesirable overlays. (Use multiple 4GL forms to produce the effects of forms that have several pages.)
- There is no concept of *current table* in 4GL. An INPUT or INPUT ARRAY statement allows the user to enter data into fields that correspond to columns in different tables or even in different databases.

- Joins defined in the PERFORM form are ignored in 4GL. You can associate two field names with the same field tag by using the same notation as in a PERFORM join, but no join is effected. However, you can create more complex joins and lookups in 4GL with the full power of SQL.
- The PERFORM attributes LOOKUP, NOUPDATE, QUERYCLEAR, RIGHT, and ZEROFILL are inoperative in 4GL. The DISPLAY, DISPLAY ARRAY, DISPLAY FORM, MESSAGE, and OPEN WINDOW statements of 4GL all ignore the INVISIBLE attribute.
- The *conditions* of a COLOR attribute cannot reference other field tags or aggregate functions.
- Default attributes listed in **syscolval** and **syscolatt** do not apply to PERFORM forms unless you recompile the forms with FORM4GL.

Chapter

7

INFORMIX-4GL Reports

In This Chapter

Creating reports is the method of producing output from 4GL programs that offers the greatest formatting flexibility. This chapter describes how to define reports to format data sets.

The following list summarizes the features besides reports that INFORMIX-4GL offers for outputting values from a relational database or values from 4GL program variables:

- Output of unformatted database rows to an ASCII file by using the UNLOAD statement (as described in "UNLOAD" on page 4-367)
- Direct screen output by using the DISPLAY statement to display values that the SELECT statement has retrieved from the database and stored in 4GL program variables (as described in "DISPLAY" on page 4-90)

  (The SELECT statement is described in the *Informix Guide to SQL: Syntax*.)
- Output to a 4GL form (as described in Chapter 6, "Screen Forms") through the DISPLAY or DISPLAY ARRAY statements
- Output to a reserved line of 4GL through the ERROR, PROMPT, MENU, or MESSAGE statement, or the COMMENTS attribute (as described in "Reserved Lines" on page 4-114)
- Output of TEXT or BYTE values to an external editor that you specify through the PROGRAM field attribute of a 4GL form (as described in "PROGRAM" on page 6-60)
- Output to the screen or to a file (or to another program, such as a text editor) from a 4GL report (as described in "Sending Report Output to the Screen" on page 7-19)

Features of 4GL Reports

For relational database management applications, 4GL includes a general-purpose *report writer* that supports the following features:

- The option to display report output to the screen for editing
- Full control over page layout for your 4GL report, including first-page and generic page headers, page trailers, columnar presentation, and special formatting before and after groups sorted by value
- Facilities for creating the report either from the rows returned by a cursor or from input records assembled from any other source, such as output from several different SELECT statements
- Control blocks to manipulate data from a database cursor on a row-by-row basis, either before or after the row is formatted by the report
- Aggregate functions that can calculate and display frequencies, percentages, sums, averages, maxima, and minima
- The USING operator and other built-in 4GL functions and operators for formatting and displaying information in output from the report
- The WORDWRAP operator to format long character strings that occupy multiple lines of output from the report
- The option to update the database or execute any sequence of SQL and other 4GL statements while writing a report, if the intermediate values calculated by the report meet specified criteria; for example, to write an alert message containing a second report

Producing 4GL Reports

Many relational database management applications are designed to produce a report that contains information from the database. A 4GL report can arrange and format the data according to your instructions and display the output on the screen, send it to a printer, or store it as a file for future use.

To write a report, a 4GL program must include two distinct components:

- The *report driver* specifies what data the report includes.
- The REPORT routine (also called the *report definition*) formats the data.

The *report driver* retrieves the specified rows from a database, stores their values in program variables, and sends these, one input record at a time, to the report definition. After the last input record is received and formatted, 4GL calculates any aggregate values that are based on all the data and then sends the entire report to some output device.

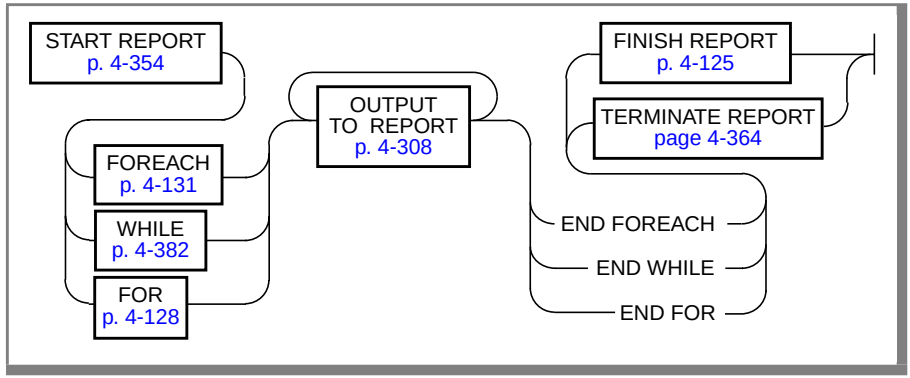
By separating the two tasks of *data retrieval* and *data formatting* in this way, 4GL simplifies the production of recurrent reports and makes it easy to apply the same report format to different data sets.

The Report Driver

The report driver invokes the report, retrieves data, and sends the data (as *input records*) to be formatted by the REPORT program block. A report driver can be part of the MAIN program block, or it can be in one or more 4GL functions. It requires special-purpose statements to interface with the report definition:

- START REPORT
- OUTPUT TO REPORT
- FINISH REPORT (or TERMINATE REPORT)

The following diagram (simplified to omit most of the control logic) shows the elements of a report driver. These elements can appear in different program blocks, but they are typically embedded within a FOR, FOREACH, or WHILE loop.



The report driver takes the following actions:

- Uses START REPORT to initialize the report
- Begins a FOR, FOREACH, or WHILE loop to control the repeated fetching of rows, and to store the retrieved data as input records
- Uses OUTPUT TO REPORT to pass data to the report
- Terminates the loop (with the END FOR, END FOREACH, or END WHILE keywords) after all the values have been passed to the report
- Uses FINISH REPORT to execute any ON LAST ROW control block and to activate two-pass report processing (as described in "The EXTERNAL Keyword" on page 7-27) or else uses TERMINATE REPORT to exit from the report before processing is completed (typically because of an error)

For information about the statements in the preceding list, see Chapter 4, "INFORMIX-4GL Statements."

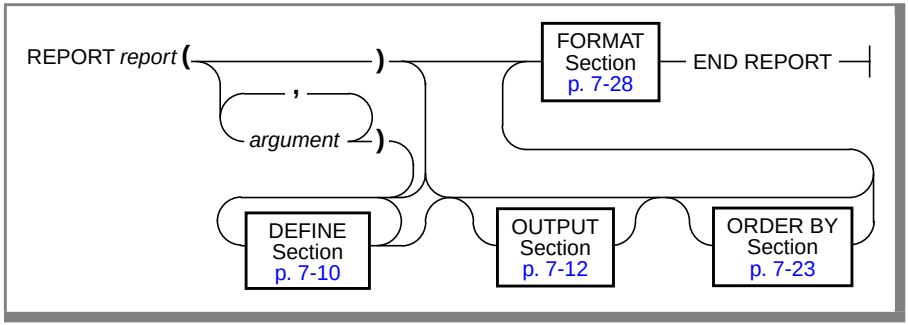
The Report Definition

The report definition formats input records. Like the FUNCTION or MAIN statement, it is a program block that can be the scope of local variables. It is not, however, a function: it is not reentrant, and CALL cannot invoke it.

The report definition receives data from its driver in sets called *input records*. These records can include program records, but other data types (including built-in and programmer-defined classes) are also supported. Each input record is formatted and printed as specified by control blocks and statements within the report definition. Most 4GL statements and functions can be included in a report definition, and certain specialized statements and operators for formatting output can appear only in a report definition.

The section "Statements Prohibited in FORMAT Section Control Blocks" on page 7-33 identifies some 4GL statements that produce compilation errors when they appear within a REPORT control block.

The REPORT program block has the following syntax.



Element	Description
argument	is the name of a formal argument in each input record. The list can include arguments of the RECORD data type, but the *record*.* notation and ARRAY data type are not valid here.)
report	is the 4GL identifier that you declare here for the report.

To format input records, a typical report definition includes these sections:

- A REPORT *prototype* to declare the name of the report, and the names of the formal arguments of input records that the report formats
- A DEFINE section to declare local variables and formal arguments
- Optional OUTPUT and ORDER BY sections to specify (respectively) the page layout of output from the report, and sorting instructions
- Control blocks within the FORMAT section to produce headers, footers, and formatted output from the data in the input records
- The END REPORT keywords that terminate the report definition

In a typical 4GL application, the input records that the report formats contain values retrieved from a database, but a 4GL report can also process input records that were not derived from any database.

The Report Prototype

The *report* name value and the *argument list* value (enclosed in parentheses) are called the *report prototype*. In its syntax, it resembles a function prototype (as described in "The Prototype of the Function" on page 4-141).

You must declare the *name* of the report and the names of all the arguments that contain the data that the driver passes to the report:

```
REPORT mcbeth_report (sound,fury)
```

A report name has global scope and must not conflict with names of other reports, functions, or global variables or with its own formal arguments.

A formal argument cannot be an ARRAY variable or a RECORD variable that contains an ARRAY member. Unless the argument list is empty, its arguments must be declared in the DEFINE section as local variables. You must specify an *argument list* value whenever any condition is true among those listed in "DEFINE Section" on page 7-10 for declaring report arguments.

If you do not specify an argument list, output from the report can include text from the control blocks, but the only data that the report can include must be contained in variables of global or module scope.

Components of the Report Definition

The report definition is composed of up to four sections. If any of the first three are included, they must appear in the following order:

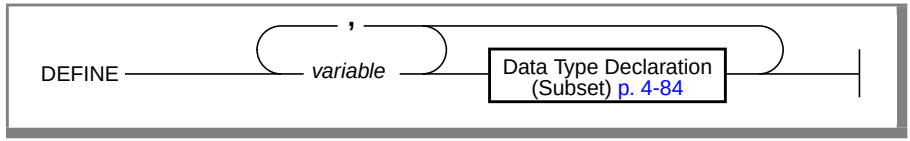
- **DEFINE section.** This section declares the data types of local variables used within the report, and of any variables (the input records) that are passed as arguments to the report by the calling statement. Reports without arguments or local variables do *not* require a DEFINE section.
- **OUTPUT section.** Output from the report consists of successive *pages*, each containing a fixed number of *lines* whose margins and maximum number of characters is fixed. This section can set margin and page size values, and can also specify where to send the formatted output.
- **ORDER BY section.** This section specifies how the variables on records are to be sorted. It is required if the report driver does not send sorted data to the report. The specified sort order determines the order in which 4GL processes any GROUP OF control blocks in the FORMAT section.
- **FORMAT section.** This section is required. It specifies the appearance of the report, including page headers, page trailers, and aggregate functions of the data. It can also contain control blocks that specify actions to take before or after specific groups of rows are processed. (Alternatively, it can produce a default report by only specifying FORMAT EVERY ROW.)

Each of these four sections begins with the keyword for which it is named. These elements of a report definition are described in sections that follow.

Like MAIN or FUNCTION, the report definition must appear outside any other program block. It must begin with the REPORT statement and must end with the END REPORT keywords. The FORMAT section is always required (and DEFINE is usually required). You can include other sections as needed.

DEFINE Section

This section declares a data type for each formal argument in the REPORT prototype and for any additional *local variables* that can be referenced only within the REPORT program block. The DEFINE section is required if you pass arguments to the report or if you reference local variables in the report.



Element	Description
variable	is the name of a local variable or formal argument of the report.

Usage

For declaring local variables, the same rules apply to the DEFINE section as to the DEFINE statement (described in "DEFINE" on page 4-81) in MAIN and FUNCTION program blocks. Two exceptions, however, restrict the data types of formal arguments:

- Report arguments cannot be of type ARRAY.
- Report arguments cannot be records that include ARRAY members.

Data types of local variables that are not formal arguments are unrestricted.

You must include arguments in the report prototype and declare them in the DEFINE section, if any of the following conditions is true:

- If you specify FORMAT EVERY ROW to create a default report, you must pass all the values for each record of the report.
- If an ORDER BY section is included, you must pass all the values that ORDER BY references for each input record of the report.
- If you use the AFTER GROUP OF control block, you must pass at least the arguments that are named in that control block.
- If an aggregate that depends on all records of the report appears anywhere except in the ON LAST ROW control block, you must pass each of the records of the report through the argument list.

(The statements mentioned in the preceding list are described later in this chapter.)

Aggregates dependent on all records include:

- GROUP PERCENT(*) (anywhere in a report)
- any aggregate without the GROUP keyword (anywhere outside the ON LAST ROW control block)

For more information, see "The COUNT (*) and PERCENT (*) Aggregates" on page 7-61.

If your report calls an aggregate function, an error might result if any argument of an aggregate function is not also a format argument of the report. You can, however, use global or module variables as arguments of aggregates if the value of the variable does not change while the report is executing.

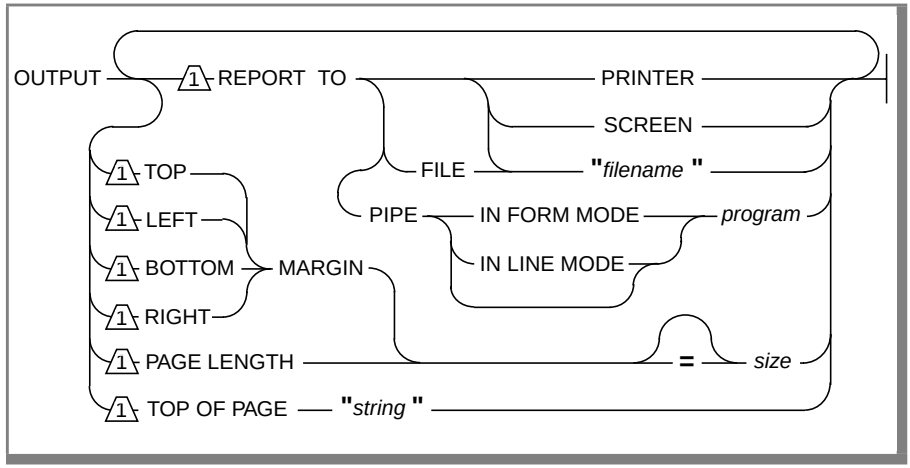
If you use the LIKE keyword to specify data types indirectly, the DATABASE statement must appear before the first program block of the same module that includes the report definition, as described in "The Default Database at Compile Time" on page 4-73. For more information, see also "Indirect Typing" on page 4-83.

If the DEFINE section declares a variable or argument with the same identifier as a global or module variable, the global or module variable is not visible in the report. See also "DEFINE" on page 4-81.

A report can reference variables of global or module scope that are not declared in the DEFINE section. Their values can be printed, but they can cause problems in aggregates and in BEFORE GROUP OF and AFTER GROUP OF clauses. Any references to nonlocal variables can produce unexpected results, however, if their values change while a two-pass report is executing.

OUTPUT Section

The OUTPUT section can specify the destination and dimensions for output from the report and the page-eject sequence for the printer. If you omit the OUTPUT section, the report uses default values to format each page. (This section is superseded by any corresponding START REPORT specifications.)



Element	Description
filename	is a quoted string that specifies the name of a file to receive the report output. The *filename* can also include a pathname.
program	is a quoted string (or CHAR or VARCHAR variable) that specifies a program, shell script, or command line to receive the output.
size	is a literal integer that specifies the height (in lines) or width (in characters) of a page of output from the report or of its margins.
string	is a quoted string that specifies the page-eject character sequence.

Usage

The OUTPUT section can direct the output from the report to a printer, file, or pipe, and can initialize the page dimensions and margins of report output. The START REPORT statement of the report driver can override all of these specifications by assigning another destination in its TO clause or by assigning other dimensions, margins, or another page-eject sequence in the WITH clause. For more information, see "START REPORT" on page 4-354.

Because the *size* specifications for the dimensions and margins of a page of report output that the OUTPUT section can specify must be literal integers, you might prefer to reset these values in the START REPORT statement, where you can use variables to assign these values dynamically at runtime.

Important: *Versions of 4GL earlier than 7.3 were not able to assign the destination or the page dimensions of output dynamically through the START REPORT statement. These START REPORT features are described in "The TO Clause" on page 4-355 and "The WITH Clause" on page 4-360.*

The OUTPUT section consists of the OUTPUT keyword, followed by one or more specifications. The OUTPUT section has the following structure:

- The REPORT TO clause specifies a default destination for output. If you omit this clause, the default is to the screen.
- If REPORT TO specifies PRINTER, the TOP OF PAGE clause can specify a 1- or 2-character page-eject sequence that causes the printer to begin a new page of report output rather than padding each page with blank lines.

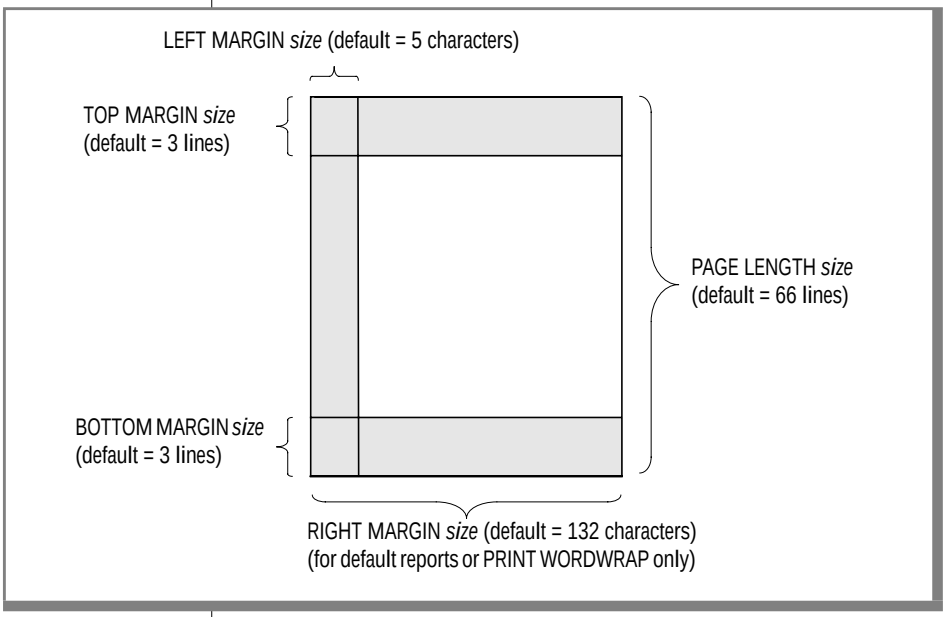


Figure 7-1
Physical Dimensions of a Page of Report Output

The five clauses that are shown in Figure 7-1 specify the physical dimensions of a 4GL report page:

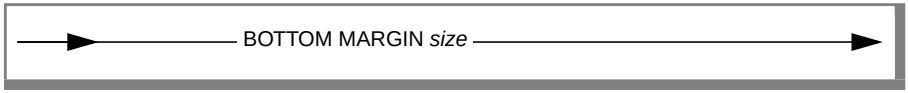
- The LEFT MARGIN clause specifies how many blank spaces to include at the beginning of each new line of output. The default is 5 blank spaces.
- The RIGHT MARGIN clause specifies the maximum number of characters in each line of output, including the left margin. If you omit this clause but specify FORMAT EVERY ROW, the default is 132 characters wide.
- The TOP MARGIN clause specifies how many blank lines appear above the first line of text on each page of output. The default is 3 blank lines.
- The BOTTOM MARGIN clause specifies how many blank lines follow the last line of output on each page. The default is 3 blank lines.
- The PAGE LENGTH clause specifies the total number of lines on each page, including data, the margins, and any page headers or page trailers from the FORMAT section. The default page length is 66 lines.

These values cannot be negative and cannot be larger than 32,766.

Sections that follow describe these OUTPUT section specifications in alphabetical order.

The BOTTOM MARGIN Clause

This clause sets a bottom margin for each page of output from the report.



Element	Description
size	is a literal integer (as described in "Literal Integers" on page 3-65) that specifies the non-negative vertical height (in lines) of the bottom margin of each page.

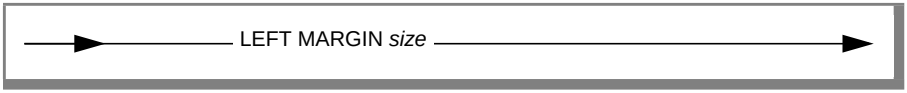
The bottom margin appears as *size* blank lines below any output specified by the PAGE TRAILER control block of the FORMAT section. If you do not include a BOTTOM MARGIN specification, the default bottom margin is three lines, meaning that at least three lines are left blank at the end of each page. The restriction $1 \leq size \leq 32{,}766$ applies (as for all OUTPUT section specifications).

The following BOTTOM MARGIN specification instructs 4GL to continue printing to the bottom of each page, with no blank lines as a bottom margin:

```
OUTPUT
   REPORT TO "sendthis.out"
   TOP MARGIN 0
   BOTTOM MARGIN 0
   PAGE LENGTH 6
```

The LEFT MARGIN Clause

This clause sets the width of a left margin for each line of report output.



Element	Description
size	is a literal integer (described in "Literal Integers" on page 3-65) to specify the non-negative width (in characters) of the left margin of each page.

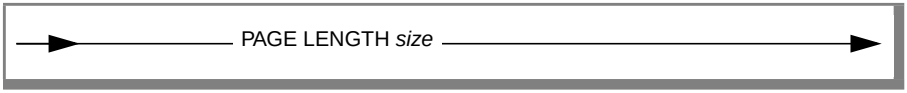
Output begins in the (*size* + 1) character position. Measurements indicated by arguments to the COLUMN function are always relative to the margin set by LEFT MARGIN. If you do not include a LEFT MARGIN clause, the default value for the left margin is five character positions. When this default is in effect, any output of data begins in the sixth character position.

The following LEFT MARGIN specification instructs 4GL to begin printing each line of output as far to the left as possible.

```
OUTPUT
   REPORT TO "about.out"
   LEFT MARGIN 0
   PAGE LENGTH 6
```

The PAGE LENGTH Clause

This clause specifies the number of lines on each page of report output.



Element	Description
size	is a literal integer (as described in "Literal Integers" on page 3-65) that specifies the non-negative height (in lines) of each page, including top and bottom margins.

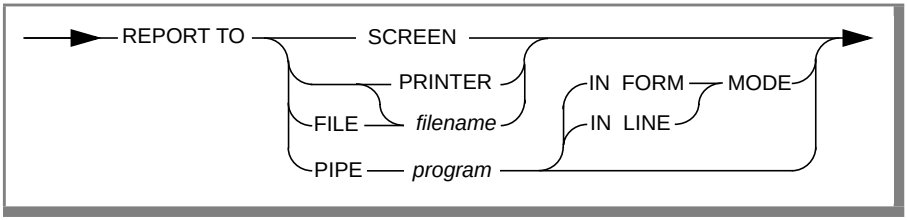
Any top or bottom margin is included within the size that you specify here. If you omit the PAGE LENGTH specification, the default page length is 66 lines. The next example specifies a PAGE LENGTH value of 22 lines:

```
OUTPUT
   PAGE LENGTH 22
   BOTTOM MARGIN 0
```

Depending on font size, 22 lines is the maximum length that some systems can use with the PAUSE statement without causing undesirable scrolling.

The REPORT TO Clause

This clause specifies a default destination for output from the report. This destination can be a file, an operating system pipe, or the system printer.



Element	Description
filename	is a quoted string that contains the name of a file to receive the report output. This filename can also include a pathname.
program	is a quoted string that specifies a program, shell script, or command line to receive the output from the report.

If the START REPORT statement includes a valid TO clause that directs output of the report to some destination, the START REPORT destination takes precedence, and any REPORT TO clause in the OUTPUT section has no effect.

Sending Report Output to a Pipe

REPORT TO PIPE sends the output to an operating system pipe. You can also specify IN LINE MODE or IN FORM MODE after the PIPE keyword to specify the screen mode of the display that *program* produces.

If no screen mode is specified, IN FORM MODE is the default unless a previous OPTIONS statement has set IN LINE MODE as the default. For more information about line mode and formatted mode in 4GL operations that produce screen output, see "Screen Display Modes" on page 4-341.

The quoted string that follows the PIPE (and optional screen mode) keywords must contain the name of a program, shell script, or command line that is to receive the report output. This string can also include command-line arguments.

The following OUTPUT section directs the report output to the **more** utility:

```
OUTPUT
   REPORT TO PIPE "more"
```

Sending Report Output to a Printer

If you specify REPORT TO PRINTER, 4GL sends the output to the program named in the **DBPRINT** environment variable. If **DBPRINT** is not set, output from the report is sent to the default printer. For example, the following code segment sends report output to the printer:

```
OUTPUT REPORT TO PRINTER
```

Sending Report Output to a File

To send the output to a printer other than the system printer, use the REPORT TO FILE "*filename*" option to send output to a file, and then send the file to a printer of your choice. Here the FILE keyword is optional.

The next example of an OUTPUT section directs the report output to the **label.out** file:

```
OUTPUT
   REPORT TO "label.out"
   LEFT MARGIN 0
   TOP MARGIN 0
   BOTTOM MARGIN 0
   PAGE LENGTH 6
```

Sending Report Output to the Screen

You can use the SCREEN keyword explicitly to specify this destination. Output is directed to the screen by default if both the REPORT TO clause and the TO clause of the START REPORT statement are omitted. To pause the display of the report after each screenful of output, you can include the PAUSE statement in the PAGE HEADER or PAGE TRAILER block of the report. The PAUSE statements waits for the user to press RETURN before displaying more output. For more information, see "PAUSE" on page 7-54.

The RIGHT MARGIN Clause

This clause sets the right margin for each line of a *default report* (one that specifies EVERY ROW in the FORMAT section) or of a PRINT WORDWRAP statement.

The syntax of the RIGHT MARGIN clause follows.

Element	Description
size	is a literal integer that specifies the maximum number of characters on each line, including the left margin.

This clause sets the right margin by specifying a line width, in characters. The *size* value is not dependent on the LEFT MARGIN but starts its count from the left edge of the page, so the width of the LEFT MARGIN is included in the size of RIGHT MARGIN. The 132-character default size is effective only when both of the following conditions are true:

- The RIGHT MARGIN clause is omitted from the OUTPUT section.
- The FORMAT section contains the EVERY ROW specification for a default report format, or else a PRINT statement with WORDWRAP is executing.

A default EVERY ROW report lists the variable names across the top of the page and presents the data in columns beneath these headings. If there is not sufficient room between left and right margins to do this, 4GL produces a two-column output format that lists the *variable name* and the *data value* of each output record on each line of output.

The following example illustrates a RIGHT MARGIN clause. After processing the OUTPUT section, 4GL sets a maximum line width of 70 and does not allow text to be printed to the right of the 70th character position:

```
REPORT simple(customer)
DEFINE customer LIKE customer.*
OUTPUT
   RIGHT MARGIN 70
FORMAT
   EVERY ROW
END REPORT
```

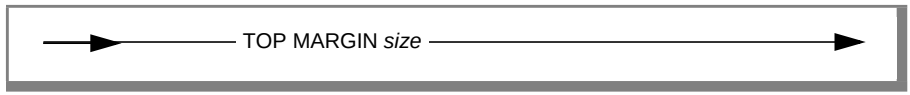
Setting a Temporary Line Width with WORDWRAP

The PRINT statement in the FORMAT section can also include a WORDWRAP RIGHT MARGIN clause. This clause sets a temporary right margin that cannot be larger than the explicit or default right margin of the OUTPUT section.

While its PRINT statement is executing, this temporary line width overrides the explicit or default right margin from the OUTPUT section. After the PRINT statement completes execution, the explicit or default RIGHT MARGIN size from the OUTPUT section is restored as the maximum line width.

The TOP MARGIN Clause

This clause sets a top margin for each page of the report.



Element	Description
size	is a literal integer (as described in "Literal Integers" on page 3-65) that specifies the vertical height (in lines) of the top margin of each page.

If you omit the TOP MARGIN specification, the default top margin is three lines, and any page header begins in the fourth line.

The following TOP MARGIN clause begins printing at the top of each page:

```
OUTPUT
   TOP MARGIN 0
   PAGE LENGTH 65
```

The number of blank lines specified as the top margin *size* value appears in report output above any page header that you specify in a PAGE HEADER or FIRST PAGE HEADER control block of the FORMAT section.

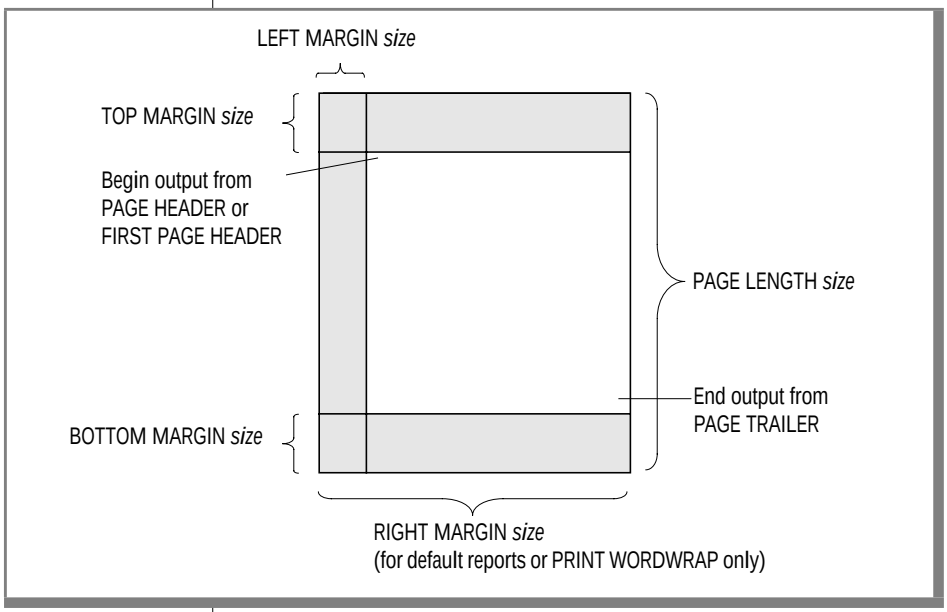
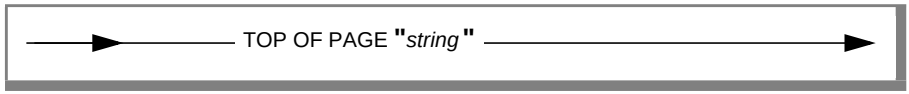


Figure 7-2
Margins of a Page of Report Output

The sum of the *size* values that you specify as your top and bottom margins, plus the number of lines (if any) for the page header and trailer, represents the portion of each page that is *not* available for displaying data. Unless the page length is greater than this total, your report cannot display any records.

The TOP OF PAGE Clause

This optional clause identifies the page-eject character sequence for a printer.



Element	Description
string	is a quoted string that begins with the page-eject character.

If you include the TOP OF PAGE clause, 4GL uses the specified page-eject character to set up new pages.

For example, the TOP OF PAGE clause in the following example specifies CONTROL-L as the *page-eject* character:

```
REPORT labels_report (rl)
   DEFINE rl RECORD LIKE customer.*
OUTPUT
   TOP OF PAGE "^L"
   REPORT TO "r_out"
```

On many printers, this string is "^L", the ASCII form-feed character. 4GL uses the first character of the string as the TOP OF PAGE character unless it is the caret (^). In this case, 4GL interprets the second character as a control character. (If you are not sure of what character string to specify for a given printer, refer to the documentation for that printer.)

If the report definition includes the TOP OF PAGE clause, all page breaks in the output are initiated by using the specified page-eject character rather than by padding with blank lines. If no TOP OF PAGE clause is included, LINEFEED characters are used (before the page trailer) to pad each page to the proper length before each page break.

New Pages of Report Output

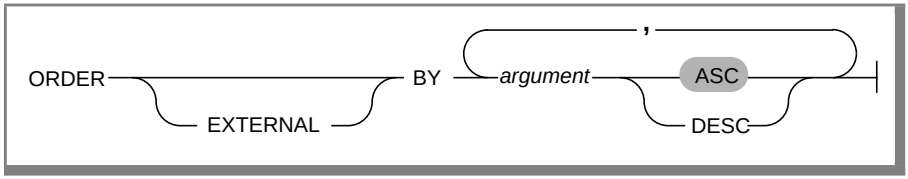
In the output from the report, 4GL includes blank line padding (or else the page-eject character, if you specify this in the *string* value) to advance to the next page whenever the program causes a new page of output to be set up. New pages can be initiated by any of the following conditions:

- PRINT attempts to print on a page that is already full.
- SKIP TO TOP OF PAGE is executed.
- SKIP *n* LINES specifies more lines than are available on the current page.
- NEED specifies more lines than are available on the current page.

If you omit the TOP OF PAGE clause, 4GL fills the remaining lines of the current page with LINEFEED characters when a new page is set up.

ORDER BY Section

The ORDER BY section specifies how to sort input records, and determines the sequence of execution of GROUP OF control blocks in the FORMAT section.



Element	Description
argument	is the name of an argument from the report prototype (as described in "The Report Prototype" on page 7-8). The list of variables that you specify here is called the sort list.

Usage

The ORDER BY section specifies a sort list for the input records. Include this section if values that the report definition receives from the report driver are significant in determining how BEFORE GROUP OF or AFTER GROUP OF control blocks will process the data in the formatted report output.

If you omit the ORDER BY section, 4GL processes input records in the order received from the report driver and processes any GROUP OF control blocks in their order of appearance in the FORMAT section. If records are not sorted in the report driver, the GROUP OF control blocks might be executed at random intervals (that is, after any input record) because unsorted values tend to change from record to record. For more information, see "AFTER GROUP OF" on page 7-34 and "BEFORE GROUP OF" on page 7-37.

If you specify only one variable in the GROUP OF control blocks, and the input records are already sorted in sequence on that variable by the SELECT statement, you do not need to include an ORDER BY section in the report.

The Sort List

The list of variables in the ORDER BY section specifies the order in which 4GL sorts the input records. You can only sort on the variables that appear in the argument list of the REPORT statement. The following program fragment, for example, sorts output in ascending order of **stock_tot** values:

```
REPORT r_invoice (c, stock_tot)
   DEFINE  c RECORD LIKE customer.*,
         stock_tot SMALLINT

   ORDER BY stock_tot
```

If you include more than one variable in the sort list, 4GL uses the left-to-right sequence of variables as the order of decreasing precedence. Unless the DESC keyword is specified, records are sorted in *ascending* (lowest-to-highest) order by values of the first (highest-priority) variable. Records having the same value for the first variable are ordered by values of the second variable and so on. Records with the same values on all but the last (lowest-priority) variable in the sort list are ordered by that variable.

GLS

Sorting that is based on character values uses the code-set order, unless you set the **DBNLS** environment variable to `1` and define a nondefault collation sequence in the COLLATION category of the locale files. ♦

If you specify the DESC keyword, the report sorts records in *descending* (highest-to-lowest) order of values for the specified variables; precedence of variables within the sort list, however, is the same as with the ASC keyword, based on the left-to-right order of variables within the sort list.

The next program fragment sorts records first by **zipcode**, and then within the same **zipcode** by **comp_name**, and within **comp_name** by **address1**:

```
REPORT labels_rpt(c)
   DEFINE c RECORD LIKE custome.*
   ORDER BY c.zipcode, c.comp_name, c.address1
```

You can also sort the records by specifying a sort list in the ORDER BY clause of the SELECT statement (in the report driver). If you specify sort lists in both the report driver and the report definition, the sort list in the ORDER BY section of the REPORT takes precedence.

If all the input records come from database rows that are returned by a single cursor, the report executes more quickly if you use the ORDER BY clause of the SELECT statement instead of the ORDER BY section of the report.

Even if the report definition receives records sorted by the report driver, you might want to specify ORDER EXTERNAL BY to specify the exact order in which GROUP OF control blocks are processed. The EXTERNAL keyword can prevent the input records from being sorted again. For more information, see "The EXTERNAL Keyword" on page 7-27.

The Sequence of Execution of GROUP OF Control Blocks

The ORDER BY section determines the order in which 4GL processes BEFORE GROUP OF and AFTER GROUP OF control blocks. If you omit the ORDER BY section, 4GL processes any GROUP OF control blocks in the lexical order of their appearance within the FORMAT section.

If you include an ORDER BY section, and the FORMAT section contains more than one BEFORE GROUP OF or AFTER GROUP OF control block, the order in which these control blocks are executed is determined by the sort list in the ORDER BY section. In this case, their order within the FORMAT section is not significant because the sort list overrides their lexical order.

4GL processes all the statements in a BEFORE GROUP OF or AFTER GROUP OF control block on these occasions:

- Each time the value of the current group variable changes
- Each time the value of a higher-priority variable changes

How often the value of the group variable changes depends in part on whether the input records have been sorted:

- If the records are sorted, AFTER GROUP OF executes after 4GL processes the last record of the group of records; BEFORE GROUP OF executes before 4GL processes the first records with the same value for the group variable.
- If the records are not sorted, the BEFORE GROUP OF and AFTER GROUP OF control blocks might be executed before and after each record because the value of the group variable might change with each record. All the AFTER GROUP OF and BEFORE GROUP OF control blocks are executed in the same lexical order in which they appear in the FORMAT section.

The following program illustrates how the ORDER BY section and the GROUP OF control blocks interact:

```
MAIN
   START REPORT sample_rpt TO "sample.out"
   OUTPUT TO REPORT sample_rpt (1,1,1)
   OUTPUT TO REPORT sample_rpt (2,2,2)
   FINISH REPORT sample_rtp
END MAIN

REPORT sample_rpt (a,b,c)
   DEFINE a,b,c, col INTEGER
   ORDER EXTERNAL BY a,b,c
   FORMAT
      FIRST PAGE HEADER
         LET col = 0
   ON EVERY ROW
         PRINT COLUMN col, "**rec**", a,b,c
   AFTER GROUP OF c
      PRINT COLUMN col, "after c"
      LET col = col - 4
   AFTER GROUP OF a
      PRINT COLUMN col, "after a"
      LET col = col - 4
   AFTER GROUP OF b
      PRINT COLUMN col, "after b"
      LET col = col -4
   BEFORE GROUP OF b
      LET col = col + 4
      PRINT COLUMN col, "before b"
   BEFORE GROUP OF a
      LET col = col + 4
      PRINT COLUMN col, "before a"
   BEFORE GROUP OF c
      LET col = col + 4
      PRINT COLUMN col, "before c"
END REPORT
```

The **sample_rpt** report in the previous example produces output in **a**, **b**, **c** order (for the BEFORE GROUP OF control blocks) and **c**, **b**, **a** order (for the AFTER GROUP OF control blocks), based on the **a**, **b**, **c** order that the ORDER BY section specifies:

```
before a
   before b
      before c
      **rec**          1          1          1
      after c
      after b
after a
before a
```

```
      before b
         before c
         **rec**          2          2          2
         after c
      after b
   after a
```

If you delete or comment out the ORDER BY section, however, the resulting code would produce the following output, based on the physical sequence of variables in GROUP OF control blocks (here c, a, b) in the FORMAT section:

```
before c
   before a
      before b
      **rec**          1          1          1
      after b
   after a
after c
before c
   before a
      before b
      **rec**          2          2          2
      after b
   after a
after c
```

The EXTERNAL Keyword

Specify ORDER EXTERNAL BY if the input records have already been sorted by the SELECT statement. The list of variables after the keywords ORDER EXTERNAL BY control the execution order of GROUP BY control blocks.

Without the EXTERNAL keyword, the report is a *two-pass* report, meaning that 4GL processes the set of input records twice. During the first pass, it sorts the data and stores the sorted values in a temporary file in the database. During the second pass, 4GL calculates any aggregate values and produces output from data in the temporary files.

With the EXTERNAL keyword, 4GL only needs to make a single pass through the data: it does not need to build the temporary table in the database for sorting the data. Specifying EXTERNAL to instruct 4GL not to sort the records again might result in an improvement in performance.

In the previous code example, the EXTERNAL keyword in the ORDER BY section instructs the report to accept the input records without sorting them. Without this keyword, the report needs access to a database to create its temporary table for sorting. (If no database is open and you run a two-pass report, a runtime error occurs when 4GL cannot create the temporary table.)

If the input records for your report come sequenced in the desired order (for example, from the rows returned by only one cursor), or to sequence values in descending order, use the ORDER BY clause in the SELECT statement that is associated with the cursor. Then use the EXTERNAL keyword in the ORDER BY section of your report.

FORMAT Section

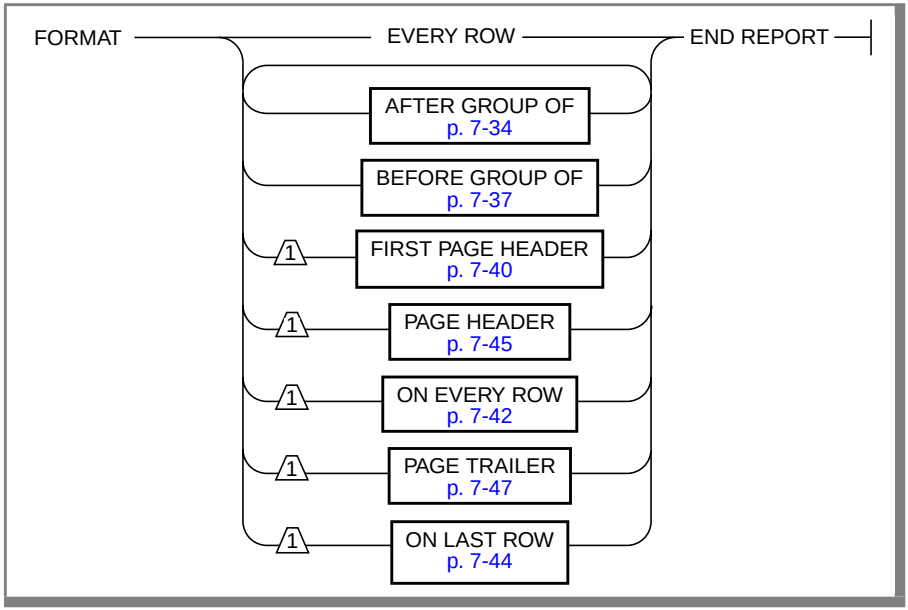
A report definition must contain a FORMAT section. The FORMAT section determines how the output from the report will look. It works with the values that are passed to the REPORT program block through the argument list or with global or module variables in each record of the report. In a source file, the FORMAT section begins with the FORMAT keyword and ends with the END REPORT keywords.

4GL supports two types of FORMAT sections. The simplest (a default report) contains only the EVERY ROW keywords between the FORMAT and END REPORT keywords.

More complex FORMAT sections can contain *control blocks* like ON EVERY ROW or BEFORE GROUP OF, which contain statements to execute while the report is being processed. Control blocks can contain report execution statements and other executable 4GL statements that are not SQL statements. For more information, see "Statements in REPORT Control Blocks" on page 48.

Sections that follow describe the syntax of default FORMAT sections, of the seven types of FORMAT section control blocks, and of the report execution statements that can appear only within a control block.

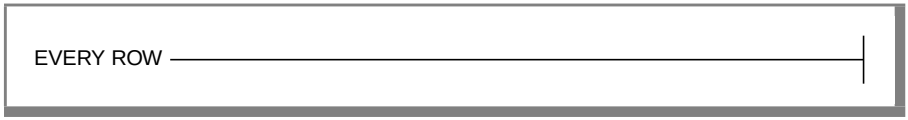
The FORMAT section has the following structure.



If you do not use the EVERY ROW keywords to specify a default report, you can combine one or more control blocks in any order within the FORMAT section. Except for BEFORE GROUP OF and AFTER GROUP OF control blocks, each type of control block must be unique within the report.

EVERY ROW

The EVERY ROW keywords specify a default output format, including every input record that is passed to the report. If you use the EVERY ROW option, no other statements or control blocks are valid.



Usage

This option formats the report in a simple default format, containing only the values that are passed to the REPORT program block through its arguments, and the names of the arguments.

You cannot modify the EVERY ROW statement with any of the statements listed in "Statements in REPORT Control Blocks" on page 7-48, nor can you include any control blocks in the FORMAT section. To display every record in a format other than the default format, use the ON EVERY ROW control block (as described in the "FORMAT Section Control Blocks" on page 7-32).

The following example of a report definition uses the EVERY ROW option:

```
REPORT minimal(customer)
DEFINE customer RECORD LIKE customer.*
FORMAT
   EVERY ROW
END REPORT
```

Here is a portion of the output from the preceding default specification:

```
customer.customer_num  101
customer.fname         Ludwig
customer.lname         Pauli
customer.company       All Sports Supplies
customer.address1      213 Erstwild Court
customer.address2
customer.city          Sunnyvale
customer.state         CA
customer.zipcode       94086
customer.phone         408-789-8075

customer.customer_num  102
customer.fname         Carole
customer.lname         Sadler
   customer.company       Sports Spot
customer.address1      785 Geary St
customer.address2
customer.city          San Francisco
customer.state         CA
customer.zipcode       94117
customer.phone         415-822-1289
```

Reports generated with the EVERY ROW option use as column headings the names of the variables that the report driver passes as arguments at runtime. If all fields of each input record can fit horizontally on a single line, the default report prints the names across the top of each page and the values beneath. Otherwise, it formats the report with the names down the left side of the page and the values to the right, as in the previous example. When a variable contains a null value, the default report prints only the name of the variable, with nothing for the value.

The following example is a brief report specification that uses the EVERY ROW statement. (Assume here that the cursor that retrieved the input records for this report was declared with an ORDER BY clause, so that no ORDER BY section is needed in this report definition.)

```
DATABASE stores7

REPORT simple(order_num, customer_num, order_date)
DEFINE order_num LIKE orders.order_num,
       customer_num LIKE orders.customer_num,
       order_date LIKE orders.order_date
FORMAT
   EVERY ROW
END REPORT
```

The following example is part of the output from the preceding report definition:

```
order_num customer_num order_date

     1001         104  01/20/1993
     1002         101  06/01/1993
     1003         104  10/12/1993
     1004         106  04/12/1993
     1005         116  12/04/1993
     1006         112  09/19/1993
     1007         117  03/25/1993
     1008         110  11/17/1993
     1009         111  02/14/1993
     1010         115  05/29/1993
     1011         104  03/23/1993
     1012         117  06/05/1993
```

You can use the RIGHT MARGIN keywords in the OUTPUT section to control the width of a report that uses the EVERY ROW statement.

FORMAT Section Control Blocks

Control blocks define the structure of a report by specifying one or more statements to be executed when specific parts of the report are processed. If no data records are output to the report, *none* of the statements in these blocks are executed. (See "Statements in REPORT Control Blocks" on page 7-48.) Each of the seven types of control blocks is optional, but if you do not use the EVERY ROW keywords, you must include at least one control block in the FORMAT section.

Control Block	When Statements in Block Are Executed
FIRST PAGE HEADER	Before processing of the first page begins
PAGE HEADER	Before processing of the each subsequent page begins
BEFORE GROUP OF	Before processing a group of sorted records
ON EVERY ROW	As each record is passed to the report
AFTER GROUP OF	After processing a group of sorted records
PAGE TRAILER	After processing of each page ends
ON LAST ROW	After the last record is passed to the report

A report can include BEFORE GROUP OF, AFTER GROUP OF, and ON EVERY ROW control blocks where the GROUP OF blocks reference the same variable. In this case, when the value of the variable changes, the report processes all BEFORE GROUP OF blocks before the ON EVERY ROW block and the ON EVERY ROW block before all AFTER GROUP OF blocks.

If a report driver includes START REPORT and FINISH REPORT statements, but no data records are passed to the report, no control blocks are executed. That is, unless the report executes an OUTPUT TO REPORT statement that passes at least one input record to the report; then neither the FIRST PAGE HEADER control block nor any other control block is executed.

The sequence in which the BEFORE GROUP OF and AFTER GROUP OF control blocks are executed depends on the sort list in the ORDER BY section. For example, assume that the ORDER BY section specifies a sort list of variables **a**, **b**, and **c** in that order (as in the example in "The Sequence of Execution of GROUP OF Control Blocks" on page 7-25). 4GL processes the control blocks in the following order, regardless of the physical sequence in which these control blocks appear within the FORMAT section.

```
BEFORE GROUP OF a              {1}
   BEFORE GROUP OF b           {2}
      BEFORE GROUP OF c        {3}
         ON EVERY ROW          {4}
      AFTER GROUP OF c         {3}
   AFTER GROUP OF b            {2}
AFTER GROUP OF a               {1}
```

Figure 7-3
The Order of Group Processing, if "a,b,c" Is the Sort List in the ORDER BY Section

In this example, a control block can be executed multiple times relative to any other block that is marked with a lower number in the right column. Without an ORDER BY section, the default is the physical order of first mention of the variables in either BEFORE or AFTER GROUP OF control blocks.

Important: *New values assigned to variables in the PAGE HEADER control block are not available until after the first PRINT, SKIP, or NEED statement is executed in an ON EVERY ROW control block. This situation guarantees that any group values printed in the PAGE HEADER control block have the same values as in the ON EVERY ROW control block. Versions of 4GL earlier than 4.1 did not support this feature.*

Statements Prohibited in FORMAT Section Control Blocks

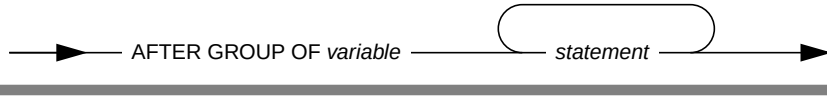
The following statements of 4GL are not valid within any control block of the FORMAT section of a REPORT program block.

CONSTRUCT	FUNCTION	MENU
DEFER	INPUT	PROMPT
DEFINE	INPUT ARRAY	REPORT
DISPLAY ARRAY	MAIN	RETURN

In this version of 4GL, a compile-time error is issued if you attempt to include any of these statements in a control block of a report. You can, however, call a function that includes any of these statements (except MAIN and DEFER).

AFTER GROUP OF

The AFTER GROUP OF control block specifies the action that 4GL takes after it processes a group of sorted records. Grouping is determined by the ORDER BY specification in the SELECT statement or in the report definition.



Element	Description
statement	is a report execution statement (as described in "Statements in REPORT Control Blocks" on page 7-48) or another 4GL statement.
variable	is the name of a formal argument to the report definition. You must pass at least the value of variable through the arguments.

Usage

A group of records is all of the input records that contain the same value for the variable whose name follows the AFTER GROUP OF keywords. This group variable must be passed through the report arguments. A report can include no more than one AFTER GROUP OF control block for any group variable.

When 4GL executes the statements in a AFTER GROUP OF control block, local variables have the values from the last record of the current group. From this perspective, the AFTER GROUP OF control block could be thought of as the *on last record of group* control block.

The Order of Processing AFTER GROUP OF Control Blocks

4GL executes the AFTER GROUP OF control block on these occasions:

- Whenever the value of the group variable changes
- Whenever the value of a higher-priority variable in the sort list changes
- At the end of the report (after processing the last input record but before 4GL executes any ON LAST ROW or PAGE TRAILER control blocks)

  In this case, each AFTER GROUP OF of control block is executed in ascending priority.

"The Sort List" on page 7-24 describes how input records are sorted according to a group variable (or a list of group variables) and the order of precedence among several variables in the sort list. How often the value of the group variable changes depends in part on whether the input records have been sorted by the SELECT statement:

- If records are already sorted when the report driver passes them to the report, the AFTER GROUP OF block is executed after 4GL has processed the last record of the group.
- If records are not sorted, the AFTER GROUP OF control blocks might be executed after any record because the value of the group variable might change with each record. If the report includes no ORDER BY section, all AFTER GROUP OF control blocks are executed in the same order in which they appear in the FORMAT section.

The AFTER GROUP OF control block is designed to work with sorted data. You can sort the records by specifying a sort list in either of the following ways:

- An ORDER BY section in the report definition
- The ORDER BY clause of the SELECT statement in the report driver

To sort data in the ORDER BY clause of the SELECT statement, perform the following tasks:

- Use the *column* value in the ORDER BY clause of the SELECT statement as the group variable in the AFTER GROUP OF control block.
- If the report contains BEFORE or AFTER GROUP OF control blocks, make sure to include an ORDER EXTERNAL BY section in the report definition to specify the precedence of variables in the sort list.

To sort data in the report definition (with an ORDER BY section), include the name of a formal argument as the group variable in both the ORDER BY section and in the AFTER GROUP OF control block. If you specify sort lists in both the report driver and in the report definition, the sort list in the ORDER BY section of the report definition takes precedence.

If the sort list includes more than one variable, 4GL sorts the records by values in the first variable (highest priority). Records that have the same value for the first variable are then ordered by the second variable and so on until records that have the same values for all other variables are ordered by the last variable (lowest priority) in the sort list.

The GROUP Keyword in Aggregate Functions

In the AFTER GROUP OF control block, you can include the GROUP keyword to qualify aggregate report functions like AVG( ), SUM( ), MIN( ), or MAX( ):

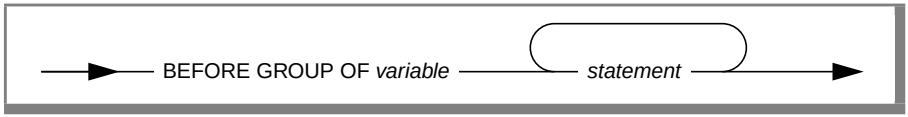
```
AFTER GROUP OF r.order_num
   PRINT "", r.order_date, 7 SPACES,
      r.order_num USING "###&",
      8 SPACES, r.ship_date, " ",
      GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"

AFTER GROUP OF r.customer_num
   PRINT 42 SPACES, "--------------------"
   PRINT 42 SPACES,
      GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
```

Using the GROUP keyword to qualify an aggregate function is only valid within the AFTER GROUP OF control block. It is not valid, for example, in the BEFORE GROUP OF control block. The aggregate report functions of 4GL are described in Chapter 5 and in "Aggregate Report Functions" on page 7-60.

After the last input record is processed, 4GL executes the AFTER GROUP OF control blocks before it executes the ON LAST ROW control block.

BEFORE GROUP OF

The BEFORE GROUP OF control block specifies what action 4GL takes before it processes a group of input records. Group hierarchy is determined by the ORDER BY specification in the SELECT statement or in the report definition.



Element	Description
statement	is a report execution statement (described in "Statements in REPORT Control Blocks" on page 7-48) or another 4GL statement.
variable	is the name of a variable from the list of formal arguments to the report definition. You must pass at least the value of variable through the arguments of the report definition.

Usage

A group of records is all of the input records that contain the same value for the variable specified after the BEFORE GROUP OF keywords. You can include no more than one BEFORE GROUP OF control block for each group variable.

When 4GL executes the statements in a BEFORE GROUP OF control block, the report variables have the values from the first record of the new group. From this perspective, the BEFORE GROUP OF control block could be thought of as the *on first record of group* control block.

The Order of Processing BEFORE GROUP OF Control Blocks

Each BEFORE GROUP OF block is executed in order, from highest to lowest priority, at the start of a report (after any FIRST PAGE HEADER or PAGE HEADER control blocks, but before processing the first record) and on these occasions:

- Whenever the value of the group variable changes (after any AFTER GROUP OF block for the old value completes execution)
- Whenever the value of a higher-priority variable in the sort list changes (after any AFTER GROUP OF block for the old value completes execution)

How often the value of the group variable changes depends in part on whether the input records have been sorted by the SELECT statement:

- If records are already sorted, the BEFORE GROUP OF block executes before 4GL processes the first record of the group.
- If records are not sorted, the BEFORE GROUP OF block might be executed after any record because the value of the group variable can change with each record. If no ORDER BY section is specified, all BEFORE GROUP OF control blocks are executed in the same order in which they appear in the FORMAT section.

The BEFORE GROUP OF control block is designed to work with sorted data. You can sort the records by specifying a sort list in either of the following areas:

- An ORDER BY section in the report definition
- The ORDER BY clause of the SELECT statement in the report driver

To sort data in the report definition (with an ORDER BY section), make sure that the name of the group variable appears in both the ORDER BY section and in the BEFORE GROUP OF control block.

To sort data in the ORDER BY clause of a SELECT statement, perform the following tasks:

- Use the column name in the ORDER BY clause of the SELECT statement as the group variable in the BEFORE GROUP OF control block.
- If the report contains BEFORE or AFTER GROUP OF control blocks, make sure that you include an ORDER EXTERNAL BY section in the report to specify the precedence of variables in the sort list.

If you specify sort lists in both the report driver and the report definition, the sort list in the ORDER BY section of the REPORT takes precedence.

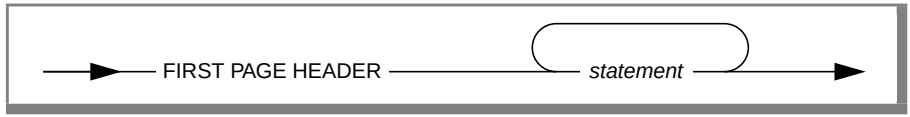
When 4GL starts to generate a report, it first executes the BEFORE GROUP OF control blocks in descending order of priority before it executes the ON EVERY ROW control block. (See also Figure 7-3 on page 7-33.)

If the report is not already at the top of the page, the SKIP TO TOP OF PAGE statement in a BEFORE GROUP OF control block causes the output for each group to start at the top of a page.

```
BEFORE GROUP OF r.customer_num
   SKIP TO TOP OF PAGE
```

FIRST PAGE HEADER

This control block specifies the action that 4GL takes before it begins processing the first input record. You can use it, for example, to specify what appears near the top of the first page of output from the report.



Element	Description
statement	is a report execution statement (described in "Statements in REPORT Control Blocks" on page 7-48) or another 4GL statement.

Usage

Because 4GL executes the FIRST PAGE HEADER control block before generating any output, you can use this control block to initialize variables that you use in the FORMAT section.

In the following example, from a report that produces multiple labels across the page, the FIRST PAGE HEADER does not display any information:

```
FIRST PAGE HEADER
   {Nothing is displayed in this control block.  It just
    initializes variables that are used in the ON EVERY ROW
    control block.}

   {Initialize label counter.}
   LET i = 1

   {Determine label width; allow 8 spaces between labels).}
   LET l_size = 72/count1

   {Divide 8 spaces among the labels across the page.}
   LET white = 8/count1
```

If a report driver includes START REPORT and FINISH REPORT statements, but no data records are passed to the report, this control block is not executed. That is, unless the report executes an OUTPUT TO REPORT statement that passes at least one input record to the report, neither the FIRST PAGE HEADER control block nor any other control block is executed.

Displaying Titles and Headings

As its name implies, you can also use a FIRST PAGE HEADER control block to produce a title page as well as column headings. On the first page of a report, this control block overrides any PAGE HEADER control block. That is, if both a FIRST PAGE HEADER and a PAGE HEADER control block exist, output from the first appears at the beginning of the first page, and output from the second begins all subsequent pages.

The TOP MARGIN (set in the OUTPUT section) determines how close the header appears to the top of the page.

Restrictions on the List of Statements

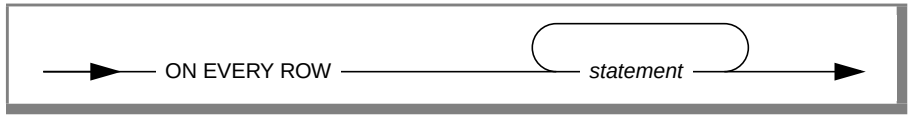
The following restrictions apply to FIRST PAGE HEADER control blocks:

- You cannot include a SKIP *integer* LINES statement inside a loop within this control block.
- The NEED statement is not valid within this control block.
- If you use an IF…THEN…ELSE statement within this control block, the number of lines displayed by any PRINT statements following the THEN keyword must be equal to the number of lines displayed by any PRINT statements following the ELSE keyword.
- If you use a CASE, FOR, or WHILE statement that contains a PRINT statement within this control block, you must terminate the PRINT statement with a semicolon (;). The semicolon suppresses any LINEFEED characters in the loop, keeping the number of lines in the header constant from page to page.
- You cannot use a PRINT *filename* statement to read and display text from a file within this control block.

Corresponding restrictions also apply to CASE, FOR, IF, NEED, SKIP, PRINT, and WHILE statements in PAGE HEADER and PAGE TRAILER control blocks.

ON EVERY ROW

The ON EVERY ROW control block specifies the action to be taken by 4GL for every input record that is passed to the report definition.



Element	Description
statement	is a report execution statement (described in "Statements in REPORT Control Blocks" on page 7-48) or another 4GL statement.

Usage

4GL executes the statements within the ON EVERY ROW control block for each new input record that is passed to the report. The following example is from a report that lists all the customers, their addresses, and their telephone numbers across the page:

```
ON EVERY ROW
  PRINT customer_num USING "###&",
     COLUMN 12, fname CLIPPED, 1 SPACE,
     lname CLIPPED, COLUMN 35, city CLIPPED,
     ", " , state,  COLUMN 57, zipcode,
     COLUMN 65, phone
```

The next example displays information about items and their prices:

```
ON EVERY ROW
  PRINT snum USING "##&", COLUMN 10, manu_code, COLUMN 18,
    description CLIPPED, COLUMN 38, quantity USING "##&",
    COLUMN 43, unit_price USING "$$$$.&&",
    COLUMN 55, total_price USING "$$,$$$,$$$.&&"
```

The next example is from a mailing label report:

```
ON EVERY ROW
  IF (city IS NOT NULL) AND
         (state IS NOT NULL) THEN
      PRINT fname CLIPPED, 1 SPACE, lname
      PRINT company
      PRINT address1
      IF (address2 IS NOT NULL) THEN PRINT address2
      PRINT city CLIPPED ", " , state, 2 SPACES, zipcode
      SKIP TO TOP OF PAGE
  END IF
```

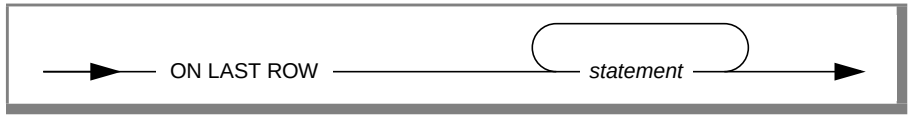
4GL delays processing the PAGE HEADER control block (or the FIRST PAGE HEADER control block, if it exists) until it encounters the first PRINT, SKIP, or NEED statement in the ON EVERY ROW control block.

Group Control Blocks

If a BEFORE GROUP OF control block is triggered by a change in the value of a variable, 4GL executes all appropriate BEFORE GROUP OF control blocks (in the order of their priority) before it executes the ON EVERY ROW control block. Similarly, if execution of an AFTER GROUP OF control block is triggered by a change in the value of a variable, 4GL executes all appropriate AFTER GROUP OF control blocks (in the reverse order of their priority) before it executes the ON EVERY ROW control block.

ON LAST ROW

The ON LAST ROW control block specifies the action that 4GL is to take after it processes the last input record that was passed to the report definition and encounters the FINISH REPORT statement.



Element	Description
statement	is a report execution statement (described in "Statements in REPORT Control Blocks" on page 7-48) or another 4GL statement.

Usage

The statements in the ON LAST ROW control block are executed after the statements in the ON EVERY ROW and AFTER GROUP OF control blocks if these blocks are present.

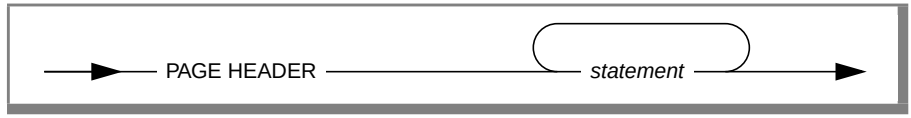
When 4GL processes the statements in an ON LAST ROW control block, the variables that the report is processing still have the values from the final record that the report processed. The ON LAST ROW control block can use aggregate functions to display report totals, as in this example:

```
ON LAST ROW
  SKIP 1 LINE
  PRINT COLUMN 12, "TOTAL NUMBER OF CUSTOMERS:",
        COLUMN 57, COUNT(*) USING "#&"
```

If the report driver executes the TERMINATE REPORT statement (rather than FINISH REPORT), the ON LAST ROW control block is not executed.

PAGE HEADER

The PAGE HEADER control block specifies the action that 4GL takes before it begins processing each page of the report. It can specify what information, if any, appears at the top of each new page of output from the report.



Element	Description
statement	is a report execution statement (described in "Statements in REPORT Control Blocks" on page 7-48) or another 4GL statement.

Usage

You can use a PAGE HEADER control block to display column headings. The following example produces column headings for printing data across the page:

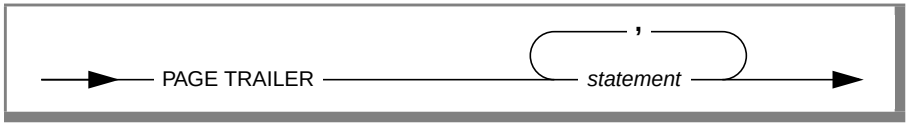
```
PAGE HEADER
  PRINT "NUMBER",
  COLUMN 12, "NAME",
  COLUMN 35, "LOCATION",
  COLUMN 57, "ZIP",
  COLUMN 65, "PHONE"
  SKIP 1 LINE
```

This control block is executed whenever a new page is added to the report. The TOP MARGIN specification (in the OUTPUT section) affects how many blank lines appear above the output produced by statements in the PAGE HEADER control block. You can use the PAGENO operator in a PRINT statement within a PAGE HEADER control block to display the current page number automatically at the top of every page. The FIRST PAGE HEADER control block overrides this control block on the first page of a report.

New group values can appear in the PAGE HEADER control block when this control block is executed after a simultaneous end-of-group and end-of-page situation. 4GL delays the processing of the PAGE HEADER control block until it encounters the first PRINT, SKIP, or NEED statement in the ON EVERY ROW, BEFORE GROUP OF, or AFTER GROUP OF control block. This order guarantees that any group columns printed in the PAGE HEADER control block have the same values as the columns printed in the ON EVERY ROW control block. The same restrictions apply to CASE, FOR, IF, NEED, SKIP, PRINT, and WHILE statements in the PAGE HEADER control block as apply to the FIRST PAGE HEADER and PAGE TRAILER control blocks.

PAGE TRAILER

The PAGE TRAILER control block specifies what information, if any, appears at the bottom of each page of output from the report.



Element	Description
statement	is a report execution statement or another 4GL statement.

Usage

4GL executes the statements in the PAGE TRAILER control block before the PAGE HEADER control block when a new page is needed. New pages can be initiated by any of the following conditions:

- PRINT attempts to print on a page that is already full.
- SKIP TO TOP OF PAGE is executed.
- SKIP *n* LINES specifies more lines than are available on the current page.
- NEED specifies more lines than are available on the current page.

You can use the PAGENO operator in a PRINT statement within a PAGE TRAILER control block to display the page number automatically at the bottom of every page, as in the following example:

```
PAGE TRAILER
   PRINT COLUMN 28,
      PAGENO USING "page <<<<"
```

The BOTTOM MARGIN specification (in the OUTPUT section) affects how close to the bottom of the page the output displays the page trailer.

Restrictions on the List of Statements

The number of lines produced by the PAGE TRAILER control block cannot vary from page to page and must be unambiguously expressed. See the list of specific restrictions that apply to CASE, FOR, IF, NEED, SKIP, PRINT, and WHILE statements in the FIRST PAGE HEADER control block.

Statements in REPORT Control Blocks

Control blocks determine *when* 4GL takes an action in a report; within each block, the statements determine *what action* 4GL takes. The list of statements in a control block terminates when another control block begins or when EXIT REPORT or END REPORT is encountered.

You can include most SQL statements and most 4GL statements in report control blocks, as well as several statements that can be used only in the FORMAT section of a report definition. "Statements Prohibited in FORMAT Section Control Blocks" on page 7-33 lists 4GL statements that are not valid in this context.

The 4GL statements most frequently used in the control blocks of reports are CASE, FOR, IF, LET, and WHILE. They have the same syntax as elsewhere in 4GL applications, as Chapter 4 describes. (Local variables referenced in such statements must be declared in the DEFINE section of the report definition; you cannot declare variables within these control blocks.)

Statements Valid Only in the FORMAT Section

The following statements, sometimes called *report execution statements*, can appear only in control blocks of the FORMAT section of a report definition.

Statement	Effect
EXIT REPORT	Terminates processing of the report
NEED	Forces a page break unless some specified number of lines is available on the current page of the report
PAUSE	Allows the user to control scrolling of screen output (This statement has no effect if output is sent to any destination except the screen.)
PRINT	Appends a specified item to the output of the report
SKIP	Inserts blank lines into a report or forces a page break

Descriptions of these report execution statements follow. None of them are valid within a MAIN or FUNCTION program block.

EXIT REPORT

This statement causes processing of the report to terminate and returns control of execution to the next statement following the most recently executed OUTPUT TO REPORT statement of the report driver.

Usage

Executing the EXIT REPORT statement has the following effects:

- Terminates the processing of the current report
- Deletes any intermediate files or temporary tables that were created in processing the REPORT statement

The EXIT REPORT statement has the same effect as TERMINATE REPORT except that EXIT REPORT must appear within the definition of the report that it terminates. This statement is useful after the program (or the user) becomes aware that a problem prevents the report from producing part of its intended output. For details of output that is not produced when a report terminates before it has finished processing, see "TERMINATE REPORT" on page 4-364.

The distinction between EXIT REPORT and TERMINATE REPORT is contextual:

- EXIT REPORT is valid only within the REPORT definition.
- TERMINATE REPORT is valid only within the report driver.

You can include the TERMINATE REPORT statement in a REPORT definition, but there it can only reference a different report. In this case, it is logically part of the driver of the other report.

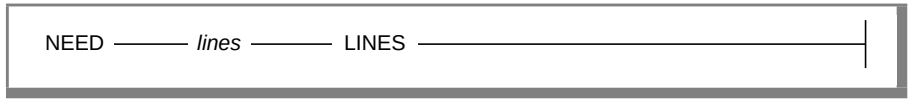
The RETURN statement cannot be used as a substitute for EXIT REPORT. An error is issued if RETURN is encountered within the definition of a 4GL report.

References

NEED, PAUSE, PRINT, REPORT, RETURN, SKIP, TERMINATE REPORT

NEED

This statement causes any subsequent display to start on the next page if fewer than the specified number of lines remain between the current line and the bottom margin of the current page of report output.



Element	Description
lines	is an integer expression that returns a positive whole number that is less than the page length.

Usage

This statement has the effect of a conditional SKIP TO TOP OF PAGE statement, the *condition* being that the number to which the integer expression evaluates is greater than the number of lines that remain on the current page.

The NEED statement can prevent the report from dividing parts of the output that you want to keep together on a single page. In the following example, the NEED statement causes the PRINT statement to send output to the next page unless at least six lines remain on the current page:

```
AFTER GROUP OF r.order_num
   NEED 6 LINES
   PRINT " ",r.order_date, 7 SPACES,
      GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
```

The *lines* value specifies how many lines must remain between the line above the current character position and the bottom margin for the next PRINT statement to produce output on the current page. If fewer than *lines* remain on the page, 4GL prints both the PAGE TRAILER and the PAGE HEADER.

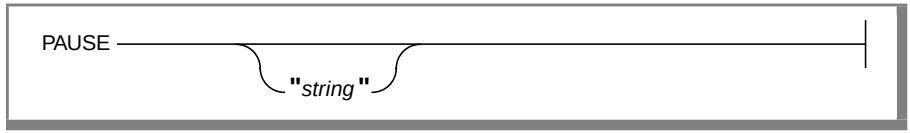
The NEED statement does not include the BOTTOM MARGIN value when it compares *lines* to the number of lines remaining on the current page. NEED is not valid in FIRST PAGE HEADER, PAGE HEADER, or PAGE TRAILER blocks.

References

PAUSE, PRINT, REPORT, SKIP

PAUSE

The PAUSE statement temporarily suspends output to the screen until the user presses RETURN.



Element	Description
string	is a quoted string that PAUSE displays. If you do not supply a message, PAUSE displays no message.

Usage

Output is sent by default to the screen unless the START REPORT statement or the OUTPUT section specifies a destination for report output. The PAUSE statement can be executed only if the report sends its output to the screen. It has no effect if you include a TO clause in either of these contexts:

- In the OUTPUT section of the report definition (as described in "The REPORT TO Clause" on page 7-17)
- In the START REPORT statement of the report driver (as described in "The TO Clause" on page 4-355)

Include the PAUSE statement in the PAGE HEADER or PAGE TRAILER block of the report. For example, the following code causes 4GL to skip a line and pause at the end of each page of report output displayed on the screen:

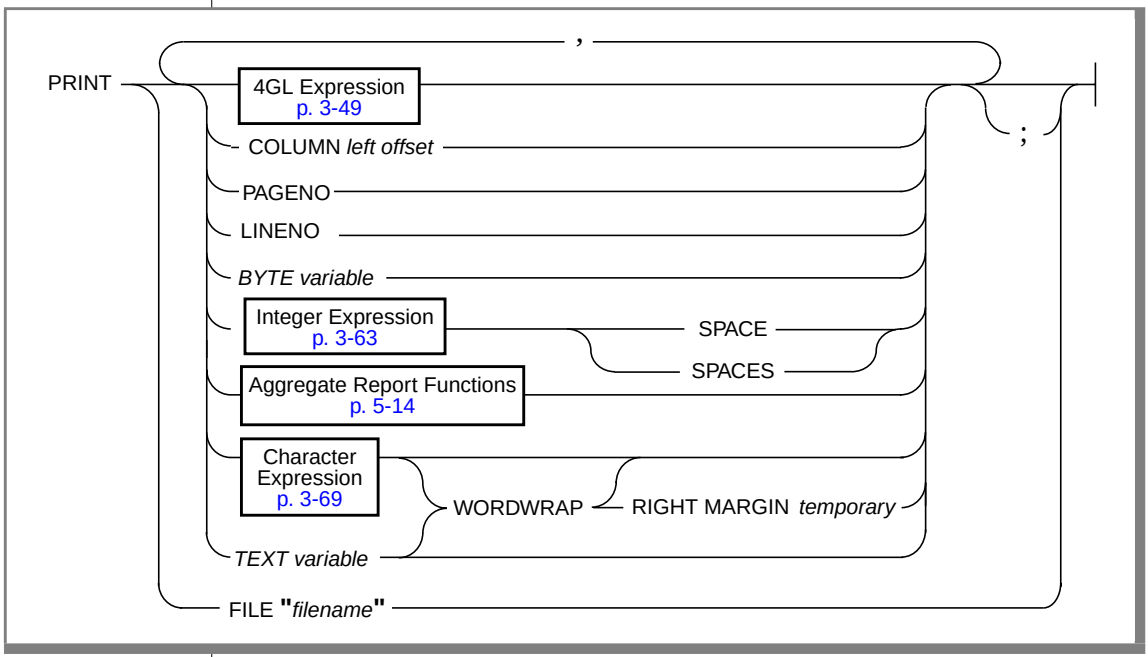
```
PAGE TRAILER
   SKIP 1 LINE
   PAUSE "Press RETURN to display next screen."
```

References

NEED, PRINT, REPORT, SKIP

PRINT

The PRINT statement produces output from a report definition.



Element	Description
BYTE variable	is the identifier of a 4GL variable of data type BYTE.
filename	is a quoted string that specifies the name of a text file to include in the output from the report. This string can include a pathname.
left offset	is an expression that returns a positive whole number. It specifies a character position offset (from the left margin) no greater than the difference (*right margin - left margin)*.
temporary	is an expression that evaluates to a positive whole number. It specifies the absolute position of a temporary right margin.
TEXT variable	is the identifier of an 4GL variable of the TEXT data type.

For the syntax of 4GL expressions like *left offset*, *relative offset*, and *temporary* that return integer values, see "Integer Expressions" on page 3-63.

Usage

This statement can include character data in the form of an ASCII file, a TEXT variable, or a comma-separated expression list of character expressions in the output of the report. (For *TEXT variable* or *filename*, you cannot specify additional output in the same PRINT statement.) You cannot display a BYTE value. Unless its scope of reference is global or the current module, any program variable in expression list must be declared in the DEFINE section.

Output is sent to the destination specified in the REPORT TO clause of the OUTPUT section or in the TO clause of the START REPORT statement of the report driver. Otherwise, the screen is the destination. (For more information, see "Sending Report Output to the Screen" on page 7-19.)

The following example is from the FORMAT section of a report definition that displays both quoted strings and values from rows of the **customer** table:

```
FIRST PAGE HEADER
PRINT COLUMN 30, "CUSTOMER LIST"
SKIP 2 LINES
PRINT "Listings for the State of ", thisstate
SKIP 2 LINES
PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
  COLUMN 57, "ZIP", COLUMN 65, "PHONE"
SKIP 1 LINE
PAGE HEADER
PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
   COLUMN 57, "ZIP", COLUMN 65, "PHONE"
SKIP 1 LINE
ON EVERY ROW
PRINT customer_num USING "###&", COLUMN 12, fname CLIPPED,
   1 SPACE, lname CLIPPED, COLUMN 35, city CLIPPED, ", " ,
   state, COLUMN 57, zipcode, COLUMN 65, phone
```

You cannot use PRINT to display a BYTE value. The string "<byte value>" is the only output from PRINT of any object that is not of the TEXT data type.

The FILE Option

The PRINT FILE statement reads the contents of the specified filename into the report, beginning at the current character position. This statement permits you to insert a multiple-line character string into the output of a report.

The following example uses the PRINT FILE statement to include the body of a form letter from file **occupant.let** in the output of a report that generates letters:

```
PRINT "Dear", 1 SPACES, "fname",","
PRINT FILE "/usr/claire/occupant.let"
```

If *filename* stores the value of a TEXT variable, the PRINT FILE *filename* statement has the same effect as specifying PRINT *variable*. (But only PRINT *variable* can include the WORDWRAP operator, as described in "The WORDWRAP Operator" on page 7-65.)

The Character Position

PRINT statement output begins at the current character position, sometimes called simply the *current position*. On each page of a report, the initial default character position is the first character position in the first line. This position can be offset horizontally and vertically by *margin* and *header* specifications and by executing any of the following statements:

- The SKIP statement moves it down to the left margin of a new line.
- The NEED statement can conditionally move it to a new page.
- The PRINT statement moves it horizontally (and sometimes down).

Unless you use the keyword CLIPPED or USING, values are displayed with widths (including any sign) that depend on their declared data types.

Data Type	Default Display Width (in characters)
BYTE	12 (4GL displays the string <byte value> as the only output.)
CHAR	The length from the data type declaration
DATE	10
DATETIME	From 2 to 25, as implied in the data type declaration
DECIMAL	(2 + *m*), where *m* is the precision from the data type declaration

(1 of 2)

Data Type	Default Display Width (in characters)
FLOAT	14
INTEGER	11
INTERVAL	From 3 to 25, as implied in the data type declaration
MONEY	(3 + *m*), where *m* is the precision from the data type declaration
NCHAR	The length from the data type declaration
NVARCHAR	The data length of the character string
SMALLFLOAT	14
SMALLINT	6
TEXT	The data length of the character string
VARCHAR	The data length of the character string

(2 of 2)

Unless you specify the FILE or WORDWRAP option, each PRINT statement displays output on a single line. This fragment displays output on two lines:

```
PRINT fname, lname
PRINT city, ", " , state, 2 SPACES, zipcode
```

If you terminate a PRINT statement with a semicolon, however, you suppress the implicit LINEFEED character at the end of the line. The next example has the same effect as the PRINT statements in the previous example:

```
PRINT fname;
PRINT lname
PRINT city, ", ";
PRINT state, 2 SPACES, zipcode
```

The Expression List

The expression list of a PRINT statement returns one or more values that can be displayed as printable characters. The following built-in functions and operators can appear in the expression list of PRINT. Some of these can appear *only* in a REPORT program block. Letter superscripts indicate restrictions on the context where some of the following functions can appear within an 4GL program. (All of these built-in functions and operators are described in Chapter 5.)

ASCII	DATE( )	MIN( )[s]	SUM( )[s]
AVG( )[s]	DAY( )	MDY( )	TIME
CLIPPED	EXTEND( )	MONTH( )	TODAY
COLUMN	GROUP[r]	ORD( )	UNITS
COUNT(*)[s]	LENGTH( )	PAGENO[r]	USING
CURRENT	LINENO[r]	PERCENT(*)[r]	WEEKDAY( )
DATE	MAX( )[s]	SPACES[r]	YEAR( )

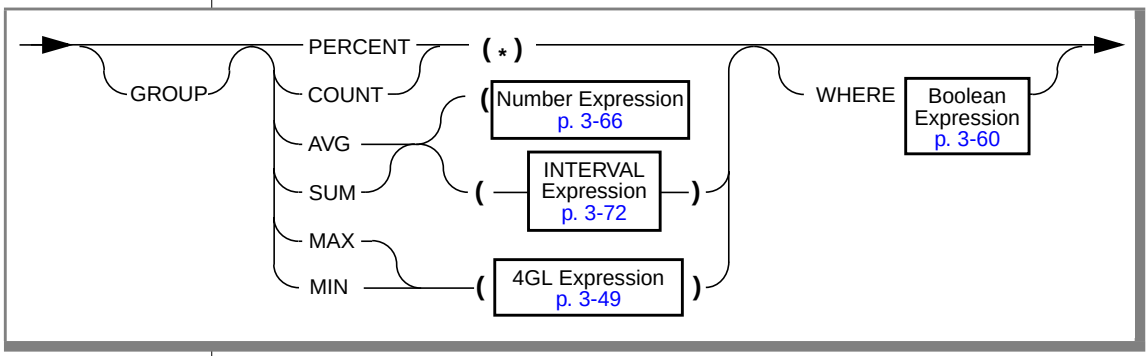
[r] You can use these expressions only within the FORMAT section of a REPORT definition. (A description appears later in this section.)

[s] You can use these aggregate functions only in the FORMAT section of a REPORT or in statements of SQL. (The PERCENT(*) aggregate cannot appear in these SQL statements.)

If the expression list applies the USING operator to format a DATE or MONEY value, the format string of the USING operator takes precedence over the **DBDATE**, **DBMONEY**, and **DBFORMAT** environment variables. For more information, see "USING" on page 5-123.

Aggregate Report Functions

Aggregate report functions summarize data from several records in a report. The syntax and effects of aggregates in a report resemble those of SQL aggregate functions but are *not* identical. (See the *Informix Guide to SQL: Syntax* for the syntax of SQL aggregate functions in SQL statements.)



The *expression* (in parentheses) that SUM( ), AVG( ), MIN( ), or MAX( ) takes as an argument is typically of a number or INTERVAL data type; ARRAY, BYTE, RECORD, and TEXT are not valid. The AVG( ), SUM( ), MIN( ), and MAX( ) aggregates ignore input records for which their arguments have null values, but each returns NULL if every record has a null value for the argument.

The GROUP Keyword

This optional keyword causes the aggregate function to include data only for a *group* of records that have the same value for a variable that you specify in an AFTER GROUP OF control block. An aggregate function can only include the GROUP keyword within an AFTER GROUP OF control block.

The WHERE Clause

The optional WHERE clause allows you to select among records passed to the report, so that only records for which the Boolean expression is TRUE are included. (See also "Boolean Expressions" on page 3-60.)

The AVG( ) and SUM( ) Aggregates

These aggregates evaluate as the average (that is, the arithmetic mean value) and the total (respectively) of *expression* among all records or among records qualified by the optional WHERE clause and any GROUP specification.

The COUNT (*) and PERCENT (*) Aggregates

These aggregates return, respectively, the total number of records qualified by the optional WHERE clause and the percentage of the total number of records in the report. You must include the (*) symbol.

The following fragment of a report definition uses the AFTER GROUP OF control block and GROUP keyword to form sets of records according to how many items are in each order. The last PRINT statement calculates the total price of each order, adds a shipping charge, and prints the result.

```
AFTER GROUP OF number
  SKIP 1 LINE
  PRINT 4 SPACES, "Shipping charges for the order:  ",
    ship_charge USING "$$$$.&&"
  PRINT 4 SPACES,  "Count of small orders:  ",
    count(*) WHERE total_price < 200.00 USING "##,###"
  SKIP 1 LINE
  PRINT 5 SPACES, "Total amount for the order:  ",
    ship_charge + GROUP SUM(total_price) USING "$$,$$$,$$$.&&"
```

Because no WHERE clause is specified here, GROUP SUM( ) combines the **total_price** of every item in the group included in the order.

The MIN( ) and MAX( ) Aggregates

For number, currency, and INTERVAL values, MIN( ) returns the minimum and MAX( ) returns the maximum values for *expression* among all records or among records qualified by the WHERE clause and any GROUP specification.

For DATETIME or DATE data values, *greater than* means *later* and *less than* means *earlier* in time.

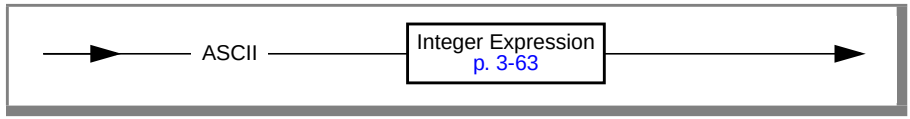
Character strings are sorted according to their first character. If your 4GL program is executed in the default (U.S. English) locale, for character data types, *greater than* means *after* in the ASCII collating sequence, where a> A> 1, and *less than* means *before* in the ASCII sequence, where 1< A< a. Appendix A lists the ASCII characters in their default collating sequence.

GLS

Character values are sorted in code-set order, unless the COLLATION category in the locale files specifies a nondefault collation sequence (and the **DBNLS** environment variable is set to `1`). ♦

The ASCII Operator

The ASCII operator has the following syntax.



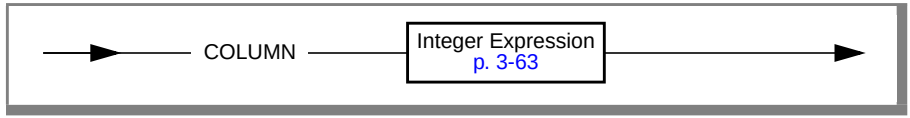
This returns the character whose numeric code you specify, just as described in Chapter 5, with one exception. To print a NULL character in a report, call the ASCII operator with `0` in a PRINT statement. For example, the following statement prints the NULL character:

```
PRINT ASCII 0
```

ASCII `0` only displays a NULL character in the PRINT statement. In other contexts, ASCII `0` returns a blank space. (See also "ASCII" on page 5-31.)

The COLUMN Operator

The COLUMN operator can appear in PRINT statements to move the character position forward within the current line. It has the following syntax.



The operand must be a non-negative integer that specifies a character position offset (from the left margin) no greater than the line width (that is, no greater than the difference (`right margin - left margin`).

This designation moves the character position to a left-offset, where `1` is the first position after the left margin. If *current position* is greater than the operand, the COLUMN specification is ignored. (See also "COLUMN" on page 5-47.)

The LINENO Operator

This operator takes no operand but returns the value of the line number of the report line that is currently printing. 4GL calculates the line number by calculating the number of lines from the top of the current page, including the TOP MARGIN. In the following example, a PRINT statement instructs 4GL to calculate and display the current line number, beginning in the tenth character position after the left margin:

```
IF (LINENO > 9) THEN
   PRINT COLUMN 10, LINENO USING "Line <<<"
END IF
```

The PAGENO Operator

This operator takes no operand but returns the number of the page that 4GL is currently printing. The next example conditionally prints the value of PAGENO, using the USING operator to format it, if its value is less than 10,000.

```
IF (PAGENO < 10000) THEN
   PRINT COLUMN 28, PAGENO USING "page <<<<"
END IF
```

You can use PAGENO in the PAGE HEADER or PAGE TRAILER block, or in other control blocks to number sequentially the pages of a report.

If you use the SQL aggregate COUNT(*) in the SELECT statement to find how many records are returned by the query, and if the number of records that appear on each page of output is both fixed and known, you can calculate the total number of pages, as in the following example:

```
SELECT COUNT(*) num  FROM customer INTO TEMP cnt
SELECT * FROM customer, cnt   --Note temp table in FROM clause
      . . .                                          --and no join is
necessary
FORMAT
   FIRST PAGE HEADER
   LET y = cnt/50--assumes 50 records per page; you must
                --round up if there is a remainder.}
   PAGE TRAILER
   PRINT "Page ", PAGENO USING "<<" " of ", y USING "<<"
```

If the calculated number of pages was 20, the first page trailer would be:

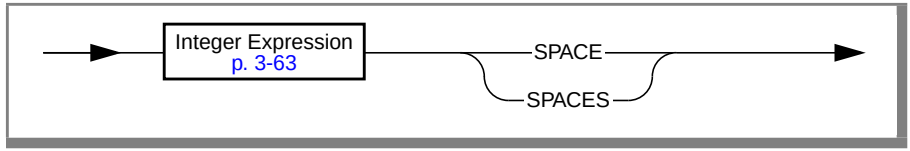
```
Page 1 of 20
```

PAGENO is incremented with each page, so the last page trailer would be:

```
Page 20 of 20
```

The SPACE or SPACES Operator

This operator has the following syntax.



This operator returns a string of blanks, equivalent to a quoted string containing the specified number of blanks. In a PRINT statement, these blanks are inserted at the current character position.

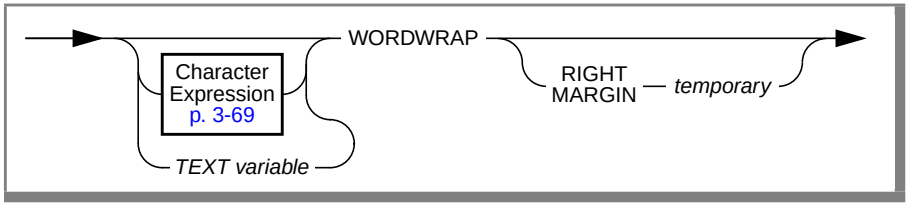
Its operand must be an integer expression (as described in "Integer Expressions" on page 3-63) that returns a positive number, specifying an offset (from the current character position) no greater than the difference (`right margin - current position`). After PRINT SPACES has executed, the new current character position has moved to the right by the specified number of characters.

Outside PRINT statements, SPACE or SPACES) and its operand must appear within parentheses.

The following statements use this operator to separate values in PRINT statements, to concatenate six blank spaces to the string `"=ZIP"` and to print the result after the variable **zipcode**:

```
FORMAT
  ON EVERY ROW
     LET mystring = (6 SPACES), "=ZIP"
     PRINT fname, 2 SPACES, lname
     PRINT company
     PRINT address1
     PRINT city, ", " , state, 2 SPACES, zipcode, mystring
```

The WORDWRAP Operator



Element	Description
temporary	is a literal integer (as described in "Literal Integers" on page 3-65) that specifies a character position (from the left edge of the page) of a temporary right margin.
TEXT variable	is the name of a 4GL variable of the TEXT data type.

The WORDWRAP operator automatically wraps successive segments of long character strings onto successive lines of report output. Any string value that is too long to fit between the current position and the right margin is divided into segments and displayed between temporary margins:

- The current character position becomes the temporary left margin.
- Unless you specify RIGHT MARGIN, the right margin defaults to 132, or to the *size* value from the RIGHT MARGIN clause of the OUTPUT section.

Specify WORDWRAP RIGHT MARGIN *temporary* to set a temporary right margin, counting from the left edge of the page. This value cannot be smaller than the current character position or greater than 132 (or the *size* value from the RIGHT MARGIN clause of the OUTPUT section). The current character position becomes the temporary left margin. These temporary values override the specified or default left and right margins from the OUTPUT section.

After the PRINT statement has executed, any explicit or default margins from the OUTPUT section are restored.

The following PRINT statement specifies a temporary left margin in column 10 and a temporary right margin in column 70 to display the character string that is stored in the 4GL variable called **mynovel**:

```
PRINT COLUMN 10, mynovel WORDWRAP RIGHT MARGIN 70
```

Tabs, Line Breaks, and Page Breaks with WORDWRAP

The data string can include printable ASCII characters. It can also include the TAB (ASCII 9), LINEFEED (ASCII 10), and ENTER (ASCII 13) characters to partition the string into *words* that consist of substrings of other printable characters. Other nonprintable characters might cause runtime errors. If the data string cannot fit between the margins of the current line, 4GL breaks the line at a word division, and pads the line with blanks at the right.

From left to right, 4GL expands any TAB character to enough blank spaces to reach the next tab stop. By default, tab stops are in every eighth column, beginning at the left-hand edge of the page. If the next tab stop or a string of blank characters extends beyond the right margin, 4GL takes these actions:

1. Prints blank characters only to the right margin
2. Discards any remaining blanks from the blank string or tab
3. Starts a new line at the temporary left margin
4. Processes the next word

4GL starts a new line when a word plus the next blank space cannot fit on the current line. If all words are separated by a single space, this action creates an even left margin. 4GL applies the following rules (in descending order of precedence) to the portion of the data string within the right margin:

- Break at any LINEFEED, or ENTER, or LINEFEED, ENTER pair.
- Break at the last blank (ASCII 32) or TAB character before the right margin.
- Break at the right margin, if no character farther to the left is a blank, ENTER, TAB, or LINEFEED character.

4GL maintains page discipline under the WORDWRAP option. If the string is too long for the current page, 4GL executes the statements in any page trailer and header control blocks before continuing output onto a new page.

GLS

Kinsoku Processing with WORDWRAP

For Japanese locales, a suitable break can also be made between the Japanese characters. However, certain characters must not begin a new line, and some characters must not end a line. This convention creates the need for *kinsoku processing*, whose purpose is to format the line properly, without any prohibited word at the beginning or ending of a line.

4GL reports use the wrap-down method for WORDWRAP and kinsoku processing. The wrap-down method forces down to the next line characters that are prohibited from ending a line. A character that precedes another that is prohibited from beginning a line can also wrap down to the next line.

Characters that are prohibited from beginning or ending a line must be listed in the locale. 4GL tests for prohibited characters at the beginning and ending of a line, testing the first and last visible characters.

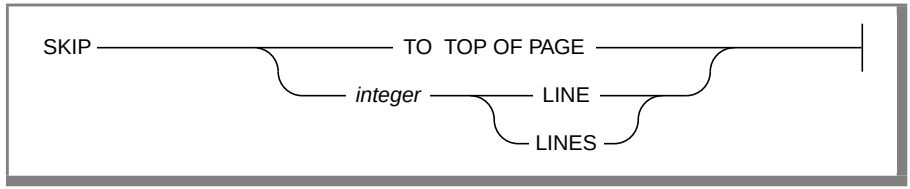
The kinsoku processing only happens once for each line. That is, no further kinsoku processing occurs, even if prohibited characters are still on the same line after the first kinsoku processing. ♦

References

DISPLAY, NEED, REPORT, SKIP

SKIP

The SKIP statement can insert blank lines into the output of a report or advance the character position to the top of the next page of report output.



Element	Description
integer	is a literal integer (as described in "Literal Integers" on page 3-65) that specifies the number of lines.

Usage

The SKIP statement allows you to insert blank lines into report output or to skip to the top of the next page as if you had included an equivalent number of PRINT statements without specifying any expression list. The LINE and LINES keywords are synonyms in the SKIP statement.

Output from any PAGE HEADER or PAGE TRAILER control block appears in its usual location. This program fragment prints names and addresses:

```
FIRST PAGE HEADER
   PRINT COLUMN 30, "CUSTOMER LIST"
   SKIP 2 LINES
   PRINT "Listings for the State of ", thisstate
   SKIP 2 LINES
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
      COLUMN 57, "ZIP", COLUMN 65, "PHONE"
   SKIP 1 LINE
PAGE HEADER
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
      COLUMN 57, "ZIP", COLUMN 65, "PHONE"
   SKIP 1 LINE
ON EVERY ROW
   PRINT customer_num USING "####",
   COLUMN 12, fname CLIPPED, 1 SPACE,
      lname CLIPPED, COLUMN 35, city CLIPPED, ", " , state,
      COLUMN 57, zipcode, COLUMN 65, phone
```

Restrictions on SKIP Statements

The SKIP LINES statement cannot appear within a CASE statement, a FOR loop, or a WHILE loop. The SKIP TO TOP OF PAGE statement cannot appear in a FIRST PAGE HEADER, PAGE HEADER, or PAGE TRAILER control block.

List of Appendixes

Appendix

A

The ASCII Character Set

This appendix lists the ASCII (American Standard Code for Information Interchange) character set, in ascending order of numeric codes 0 through 127. In the default (U.S. English) locale, this ASCII collating sequence is the basis for relational comparisons of strings in INFORMIX-4GL and SQL Boolean expressions.

GLS

Nondefault locales always include the ASCII characters but can include additional non-ASCII characters and can specify other collation sequences. When character strings are sorted by 4GL, however, the code-set order is always the basis for collation; any other collation sequence specified in the locale files is ignored. See the *Informix Guide to GLS Functionality* for details of how Informix database servers perform collation in nondefault locales. In some situations (involving NCHAR or NVARCHAR values and nondefault locales), the database server and 4GL do not follow the same rules for collating character strings.

See also "Collation Order" on page E-4. ♦

In the table that follows, a caret (^) prefix in the first **Character** column represents the CONTROL key.

Code	Character	Code	Character	Code	Character
0	^@	43	+	86	V
1	^A	44	,	87	W
2	^B	45	-	88	X
3	^C	46	.	89	Y
4	^D	47	/	90	Z
5	^E	48	0	91	[
6	^F	49	1	92	\
7	^G	50	2	93	]

Code	Character	Code	Character	Code	Character
8	^H	51	3	94	^
9	^I	52	4	95	_
10	^J	53	5	96	`
11	^K	54	6	97	a
12	^L	55	7	98	b
13	^M	56	8	99	c
14	^N	57	9	100	d
15	^O	58	:	101	e
16	^P	59	;	102	f
17	^Q	60	<	103	g
18	^R	61	=	104	h
19	^S	62	>	105	i
20	^T	63	?	106	j
21	^U	64	@	107	k
22	^V	65	A	108	l
23	^W	66	B	109	m
24	^X	67	C	110	n
25	^Y	68	D	111	o
26	^Z	69	E	112	p
27	esc	70	F	113	q
28	^\	71	G	114	r
29	^]	72	H	115	s
30	^^	73	I	116	t
31	^_	74	J	117	u
32		75	K	118	v
33	!	76	L	119	w
34	"	77	M	120	x
35	#	78	N	121	y
36	$	79	O	122	z
37	%	80	P	123	{
38	&	81	Q	124	|
39	'	82	R	125	}
40	(	83	S	126	~
41	)	84	T	127	del
42	*	85	U		

(2 of 2)

Appendix

B

INFORMIX-4GL Utility Programs

This appendix describes the utility programs that are included with INFORMIX-4GL. You can invoke these utilities at the system prompt to perform the following tasks:

- The **mkmessage** utility compiles programmer-defined help messages for 4GL applications.
- The **upscol** utility enables you to establish default attributes for display fields that are linked to database columns in your screen forms. It can also establish initial default values for program variables and screen fields that you associate with columns of tables in your database.

The mkmessage Utility

The **mkmessage** utility converts ASCII source files that contain user messages to a format that 4GL programs can use in on-line displays. This section describes how to use **mkmessage** with help files and with customized runtime error messages.

Programmer-Defined Help Messages

When executing a 4GL program, the user can request help whenever the program is waiting for user input, such as making a menu selection, while inputting data to a form, or while responding to a prompt. You can supply help messages that are displayed whenever the user presses the Help key (specified in the OPTIONS statement). These messages can be specific to the menu option currently highlighted or to a CONSTRUCT, INPUT, INPUT ARRAY, or PROMPT statement.

Message Source Files

4GL looks for the appropriate help message in the help file that you specify in an OPTIONS statement by using the HELP FILE option. You can have several help files, but only one can be in effect at a time. The structure of the message source file is as follows:

```
.num
message-text
```

In the example, *.num* is a period followed by an integer, and *message-text* is one or more lines of characters. (Characters can include blanks and must be from the code set of the locale.) The file can contain as many messages as you like.

Each help message should be preceded by a line with nothing on it but a period (in the first column) and a unique integer *num*. The *message-text* line starts on the next line and continues until the next numbered line. Each line must end in a RETURN. All blank lines between two numbered lines are considered part of the message that belongs to the first of the two numbers. Lines that begin with **#** are interpreted as comment lines (and ignored).

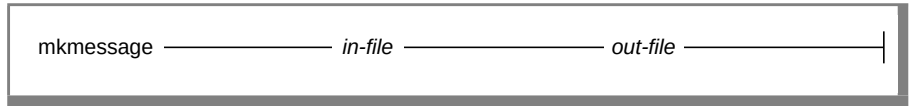
The integer *num* can specify the help message in your 4GL programs. (For 4GL statements that support a HELP clause for displaying contextual help messages, see the CONSTRUCT, INPUT, INPUT ARRAY, MENU, and PROMPT statement descriptions in Chapter 4, "INFORMIX-4GL Statements.")

If the message text occupies more than 20 lines, 4GL automatically breaks the message into *pages* of 20 lines. You can change these default page breaks by placing the CONTROL-L character in the first column of a line in your message file to start a new page. 4GL handles clearing and redisplaying the screen.

For an example of a message file, see "helpdemo.src" on page H-30.

Creating Executable Message Files

Once you have created your message source file, you can process it for use by 4GL with this syntax.



Element	Description
in-file	is an ASCII source file of help messages.
out-file	s the pathname of the executable output file.

After creating an output file with the **mkmessage** utility, specify *out-file* in the OPTIONS statement to identify it as the current help file.

To use help messages from the help file on a field-by-field basis in an INPUT or INPUT ARRAY statement, you must use the INFIELD( ) and SHOWHELP( ) library functions that are supplied with 4GL. For example, you can use these functions as the following code segment demonstrates:

```
OPTIONS
    HELP FILE "stores7.hlp",
    HELP KEY F1
...
INPUT pr_fname, pr_lname, pr_phone
    FROM fname, lname, phone HELP 101
    ON KEY (F1)
        CASE
            WHEN INFIELD(lname)
                CALL showhelp(111)
            WHEN INFIELD(fname)
```

```
                CALL showhelp(112)
            WHEN INFIELD(phone)
                CALL showhelp(113)
            OTHERWISE
                CALL showhelp(101)
        END CASE
END INPUT
```

Customized Error Messages

You can also use the **mkmessage** utility to customize runtime error messages. 4GL is distributed with a file called **4glusr.msg**. This ASCII file contains some common error messages, including the messages for runtime errors that cannot be trapped by the WHENEVER ERROR statement and messages that support the 4GL **Help** menu. The **4glusr.iem** file contains the executable version of this file.

You can edit the messages in **4glusr.msg** with a text editor (for example, to make them specific to a 4GL application or to translate them into another language). Be sure to preserve the required numeric codes, prefixed by a period (.) to identify each message.

If you choose to modify the contents of the **4glusr.msg** message file, you must specify **4glusr.iem** in your **mkmessage** command line as the object filename:

```
mkmessage in-file 4glusr.iem
```

The executable file **4glusr.iem** is initially installed in the directory **$INFORMIX/msg**. 4GL looks for message files in one of two directories, namely **/$INFORMIXDIR/$DBLANG** or else **/$INFORMIXDIR/msg**. If **DBLANG** is defined, 4GL looks only in **/$INFORMIXDIR/$DBLANG**. If this directory is not defined, 4GL looks only in **/$INFORMIXDIR/msg**. You must place the newly modified file **4glusr.iem** in the appropriate **/$INFORMIXDIR/msg** or **/$INFORMIXDIR/$DBLANG** directory.

The upscol Utility

The **upscol** utility program allows you to create and modify the **syscolval** and **syscolatt** tables, which contain default information for fields in screen forms that correspond to database columns. Chapter 6, "Screen Forms," describes these tables and their use by 4GL.

You invoke the **upscol** utility by entering the command `upscol` at the system prompt. After you select a database at the CHOOSE DATABASE screen, the following menu appears.

```
UPDATE SYSCOL:  Validate  Attributes  Exit
Update information in the data validation table.

-------------------- db-name ------------ Press CTRL-W for Help ------
```

The options in the UPDATE SYSCOL menu are:

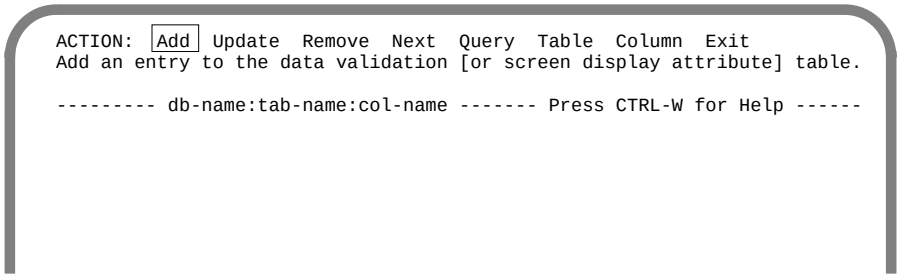
- **Validate.** Updates the information in **syscolval**.
- **Attributes.** Updates the information in **syscolatt**.
- **Exit.** Returns to the operating system.

If you select either **Validate** or **Attributes**, **upscol** checks whether the corresponding table exists and, if not, **upscol** asks whether you want to create the table. In the text that follows, the corresponding table is called **syscol**. If you choose not to create it, enter `n`, and you return to the **UPDATE SYSCOL** menu.

If the data validation table already exists, or if you enter `y` to create it, **upscol** displays the CHOOSE TABLE screen and prompts you for the name of a table in the database. After you select a table, the CHOOSE COLUMN screen prompts you to select the name of a column whose default values you want to modify in **syscol**.

In the illustrations that follow, **tab-name** represents the selected *table* (from the specified **db-name** database), and **col-name** represents the selected *column* within the **tab-name** table. An actual **upscol** session would display the SQL identifiers of whatever *database*, *table*, and *column* you specified.

The selected table and column names appear, with the database name, on the dividing line beneath the next menu, which is called the ACTION menu.



Now **upscol** displays the first row of **syscol** that relates to the table and column in the work area beneath this menu. If no such entries exist, a message stating this appears on the Error line.

The options in the ACTION menu are these:

- **Add.** Adds new rows to the **syscol** table.
- **Update.** Updates the currently displayed row.
- **Remove.** Removes the currently displayed row (after a prompt for verification).
- **Next.** Displays the next row of **syscol**.
- **Query.** Restarts the display at the first row of **syscol** for the table and column.
- **Table.** Selects a new database table and column.
- **Column.** Selects a new column within the chosen table.
- **Exit.** Return to the UPDATE SYSCOL menu.

Adding or Updating Under the Validate Option

When you select **Add** in the **ACTION** menu after choosing the **Validate** option in the **UPDATE SYSCOL** menu, the **VALIDATE** menu appears.

```
VALIDATE: Autonext  Comment  Default  Include  Picture  Shift  Verify  Exit
Automatically proceed to next field when at end of current field.

--------- db-name:tab-name:col-name ----------- Press CTRL-W for Help ------
```

The options are attribute names and their selection has the following effects:

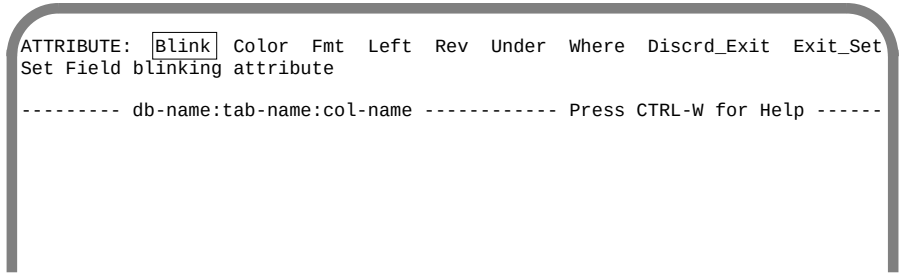
- **Autonext.** Produces a menu with three options, **Yes**, **No**, and **Exit**. **Exit** returns you to the **VALIDATE** menu. The default is **No**.
- **Comment.** Produces a prompt to enter a Comment line message. No quotation marks are required around the message, but it must fit on a single screen line.
- **Default.** Produces a prompt to enter the DEFAULT attribute, formatted as described in Chapter 6. Quotation marks are required where necessary to avoid ambiguity.
- **Include.** Produces a prompt to enter the INCLUDE attribute, formatted as described in Chapter 6. Quotation marks are required where necessary to avoid ambiguity.
- **Picture.** Produces a prompt to enter the PICTURE attribute, formatted as described in Chapter 6. No quotation marks are required.
- **Shift.** Produces a menu with four options, **Up**, **Down**, **None**, and **Exit**. **Up** corresponds to the UPSHIFT attribute and **Down** to the DOWNSHIFT attribute. **Exit** returns you to the **VALIDATE** menu. The default is **None**.
- **Verify.** Produces a menu with three options, **Yes**, **No**, and **Exit**. **Exit** returns you to the **VALIDATE** menu. The default is **No**.
- **Exit.** Returns you to the **ACTION** menu.

The **upscol** utility adds or modifies a row of **syscolval** after you complete each of these options except **Exit**.

The **Update** option on the **ACTION** menu takes you immediately to the **ATTRIBUTE** menu or prompt that corresponds to the current attribute for the current column. You can look at another attribute for the current column by using the **Next** option, start through the list again by using the **Query** option, remove the current attribute with the **Remove** option, and select a new column or table with the **Column** or **Table** option.

Adding or Updating Under the Attribute Option

When you select **Add** or **Update** in the **ACTION** menu after choosing **Attribute** in the **UPDATE SYSCOL** menu, the **ATTRIBUTE** menu appears.



If you are adding a new row to **syscolatt**, a default row is displayed in the work area below the menu. If you are updating an existing row of **syscolatt**, the current row appears. No entry is made in **syscolatt** until you choose **Exit_Set**, so you can alter all the attributes before you decide whether to modify **syscolatt** (with **Exit_Set**) or to cancel the changes (with **Discrd_Exit**).

The options of the **ATTRIBUTE** menu include screen attribute names, and their selection has the following effects:

- **Blink.** Produces a menu with three options, **Yes**, **No**, and **Exit**. The default is **No**.
- **Color.** Produces a menu with the available color options (for color terminals) or intensities (for monochrome terminals) for display of data values in **tab-name.col-name**. You can toggle back and forth among the colors or intensities by pressing CONTROL-N.
- **Fmt.** Prompts you for the format string to be used when **tab-name.col-name** is displayed.

- **Left.** Produces a menu with three options, **Yes**, **No**, and **Exit**. **Yes** causes numeric data to be left-aligned within the screen field. The default is **No**.
- **Rev.** Produces a menu with three options, **Yes**, **No**, and **Exit**. **Yes** causes the field to be displayed in reverse video. The default is **No**.
- **Under.** Produces a menu with three options, **Yes**, **No**, and **Exit**. **Yes** causes the field to be displayed with underlining. The default is **No**.
- **Where.** Prompts for the values and value ranges under which these attributes will apply. See Chapter 6 for allowable syntax.
- **Discrd_Exit.** Discards the indicated changes and returns you to the **ACTION** menu.
- **Exit_Set.** Enters the indicated changes into the **syscolatt** table and returns you to the **ACTION** menu.

After you complete each of these options except **Discrd_Exit**, **upscol** adds or modifies a row of **syscolatt**.

Whoever runs the **upscol** utility produces a pair of tables, **syscolval** and **syscolatt**, which provide default values for all the users of a database that is not ANSI compliant.

If the current database is ANSI-compliant, however, the user who runs **upscol** becomes the owner of the **syscolatt** and **syscolval** tables specified at the **upscol** menus, but other users can produce their own **user.syscolval** and **user.syscolatt** tables. The default specifications in an **upscol** table are applied by 4GL only to columns of tables that have the same owner as the **upscol** table. (For details, see "Default Attributes in an ANSI-Compliant Database" on page 6-84, and the INITIALIZE and VALIDATE statements in Chapter 6.)

Appendix

C

Using C with INFORMIX-4GL

Some programming tasks might be more easily or more efficiently coded with a combination of INFORMIX-4GL code and C code. In these cases, you have two options:

- Write a 4GL program that calls C functions.
- Write a C program that calls 4GL functions.

To call either a C function or a 4GL function, you must know about the *argument stack* mechanism (discussed in "Using the Argument Stack" on page C-3) that 4GL uses to pass arguments between the functions and the calling code. This appendix discusses issues that relate to the application programming interface (API) between the 4GL language and the C language:

- Calling a C function from a 4GL program

  The CALL statement of 4GL can invoke C functions that observe the calling conventions of the 4GL argument stack (as described in "Calling a C Function from a 4GL Program" on page C-12).

- Calling a 4GL function from a C program

  For a C program to call a 4GL function, it must include a special header file. 4GL provides macros to initialize the argument stack and to support other requirements of the C language (as described in "Macros for Calling 4GL Functions" on page C-21).

- Decimal functions for C

  4GL provides a library of functions that facilitate the conversion of its own DECIMAL data type values to and from every data type of the C language (as described in "Decimal Functions for C" on page C-26).

The following topics are described in this appendix.

The sections that describe these functions also contain code examples.

Using the Argument Stack

Within a 4GL program, 4GL uses a *pushdown stack* to pass arguments and results between 4GL functions. The caller of a function pushes its arguments onto the stack; the called function pops them off the stack to use the values. The called function pushes its return values onto the stack, and the caller pops them off to retrieve the values.The argument stack is also used when a 4GL program calls a C function that you have written or when a C program calls a 4GL function. This section describes the following operations:

- Passing values between 4GL functions
- Receiving values from 4GL
- Passing values to 4GL

C code that calls any of the library functions that push, pop, or return values in the pushdown stack should include the file **fglsys.h** at the top of the file:

```
#include <fglsys.h>
```

Passing Values Between 4GL Functions

Consider the following 4GL program:

```
MAIN
   DEFINE j,k,sum,dif SMALLINT
   LET j=5
   LET k=7
   CALL sumdiff(j,k) RETURNING sum, dif
END MAIN

FUNCTION sumdiff(a,b)
   DEFINE a,b,s,d DECIMAL(16)
   LET s = a+b
   LET d = a-b
   RETURN s,d
END FUNCTION
```

When the program executes the CALL statement, 4GL first notes the current argument stack depth. Then it pushes the function arguments onto the top of the stack in sequence from left to right (first **j** and then **k** in the example). The stack receives not only the value of an argument but its data type as well (SMALLINT in the example).

The called function pops the arguments off the stack into local variables (**a** and **b**, respectively). In the example, the types of these variables (DECIMAL) are different from the types of the arguments that were pushed by the caller (SMALLINT). However, because the stack stores the type of each passed value, 4GL is able to convert the arguments to the proper type. In this instance, 4GL easily converts SMALLINT values to DECIMAL. Any type conversion that is supported by the LET statement is supported when popping stacked values.

The RETURN statement pushes the returned values onto the stack in sequence from left to right (first **s** and then **d** in the example). The RETURNING clause in the originating CALL statement then pops these values off the stack and into the specified variables (**sum** and **diff**, respectively) and converts the data types DECIMAL back to SMALLINT.

When 4GL calls a function within an expression, the use of the stack is the same as in the CALL statement. The function is expected to return a single value on the stack, and 4GL attempts to convert this value as required to evaluate the expression.

Important: *Releases of 4GL earlier than Version 7.31 provided function libraries that used different function names. For details, see the* ***fglsys.h*** *file, included in 4GL. To consistently maintain differences on 32-bit and 64-bit platforms, 4GL 7.31 introduces new mappings for C data types. These new data types should be used when calling any 4GL library. For more details of the mapping between the new data types and C-defined variables, see the* ***fgltypes.h*** *file, included in 4GL.*

Function prototypes in the sections that follow use the function names that were provided in releases of 4GL earlier than Version 7.31, but show the data types of the current functions. The corresponding new function names are shown immediately after each list of pre-Version 7.31 function prototypes.

Important: *If you are compiling C programs with your 4GL program based on the pre-Version 7.31 functions for manipulating the 4GL function stack, in order to get the correct mappings from the old functions names to the new functions names, you must include* ***fglsys.h*** *as the header file in the C program.*

Receiving Values from 4GL

C functions or programs receive arguments from 4GL by using the argument stack. Within the C function or program, you pop values off the stack by using *pop external functions* that are included with 4GL. If you try to pop a value when none is on the stack, a core dump or other fatal behavior can occur.

This section describes the pop external functions, according to the data type of the value that each pops from the argument stack.

Library Functions for Popping Numbers

You can call the following 4GL library functions from a C function or program to pop number values from the argument stack:

```
extern void popint(mint *iv)
extern void popshort(int2 *siv)
extern void poplong(int4 *liv)
extern void popflo(float *fv)
extern void popdub(double *dfv)
extern void popdec(dec_t *decv)
```

The correspondence between pre-Version 7.31 function names and new function names follows.

Old Function Name	New Function Name
popint	ibm_lib4gl_popMInt
popshort	ibm_lib4gl_popInt2
poplong	ibm_lib4gl_popInt4
popflo	ibm_lib4gl_popFloat
popdub	ibm_lib4gl_popDouble
popdec	ibm_lib4gl_popDecimal

Each of these functions, like all library functions for popping values, performs the following actions:

1. Removes one value from the argument stack
2. Converts its data type if necessary
3. Copies it to the designated variable

If the value on the stack cannot be converted to the specified type, the result is undefined.

If the **ibm_lib4gl_popInt2( )** function returns a value outside the range of -32767 to 32767, a conversion error has occurred.

The **dec_t** structure referred to by **ibm_lib4gl_popDecimal( )** is used to hold a DECIMAL value. It is discussed later in this appendix.

Library Functions for Popping Character Strings

You can call the following 4GL library functions to pop character values:

```
extern void popquote(int1 *qv, mint len)
extern void popstring(int1 *qv, mint len)
extern void popvchar(int1 *qv, mint len)
```

The correspondence between pre-Version 7.31 function names and new function names follows.

Old Function Name	New Function Name
popquote	ibm_lib4gl_popQuotedStr
popstring	ibm_lib4gl_popString
popvchar	ibm_lib4gl_popVarChar

Both **ibm_lib4gl_popQuotedStr( )** and **ibm_lib4gl_popVarChar( )** copy exactly **len** bytes into the string buffer ***qv**. Here **ibm_lib4gl_popQuotedStr( )** pads with spaces as necessary, but **ibm_lib4gl_popVarChar( )** does not pad to the full length. The final byte copied to the buffer is a null byte to terminate the string, so the maximum string data length is **len**-1. If the stacked argument is longer than **len**-1, its trailing bytes are lost.

The **len** argument sets the maximum size of the receiving string buffer. Using **ibm_lib4gl_popQuotedStr( )**, you receive exactly **len** bytes (including trailing blank spaces and the null), even if the value on the stack is an empty string. To find the true data length of a string retrieved by **ibm_lib4gl_popQuotedStr( )**, you must trim trailing spaces from the popped value. (The functions **ibm_lib4gl_popString( )** and **ibm_lib4gl_popQuotedStr( )** are identical, except that **ibm_lib4gl_popString( )** automatically trims any trailing blanks.)

Because 4GL can convert values to CHAR from any data type except TEXT or BYTE, you can use these functions to pop almost any argument.

Library Functions for Popping Time Values

You can call the following 4GL library functions to pop DATE, DATETIME, and INTERVAL values:

```
extern void popdate(int4 *datv)
extern void popdtime(dtime_t *dtv, mint qual)
extern void popinv(intrvl_t *iv, mint qual)
```

The correspondence between pre-Version 7.31 function names and new function names follows.

Old Function Name	New Function Name
popdate	ibm_lib4gl_popDate
popdtime	ibm_lib4gl_popDateTime
popinv	ibm_lib4gl_popInterval

The structure types **dtime_t** and **intrvl_t** are used to represent DATETIME and INTERVAL data in a C program. They are discussed in the *INFORMIX-ESQL/C Programmer's Manual*. The **qual** argument receives the binary representation of the DATETIME or INTERVAL qualifier. The *INFORMIX-ESQL/C Programmer's Manual* also discusses library functions for manipulating and printing DATE, DATETIME, and INTERVAL variables.

Library Functions for Popping BYTE or TEXT Values

You can call the following function to pop a BYTE or TEXT argument:

```
extern void poplocator(loc_t **blob)
```

The correspondence between pre-Version 7.31 function names and new function names follows.

Old Function Name	New Function Name
poplocator	ibm_lib4gl_popBlobLocator

The structure type **loc_t** defines a BYTE or TEXT value. Its use is discussed in your *Administrator's Guide*. The **ibm_lib4gl_popBlobLocator( )** function is unusual in that it does not copy the passed value. The function copies only the address of the passed value (as is indicated by the double asterisk in the function prototype). The following C fragment illustrates this:

```
mint get_a_text(mint nargs)
{
   loc_t *theText;
   ibm_lib4gl_popBlobLocator(&theText)
   ...
}
```

What **ibm_lib4gl_popBlobLocator( )** stores at the address specified by its parameter is the address of the locator structure owned by the calling function. A change to the locator or the value that it describes is visible to the calling program, which is not the case with other data types.

Any BYTE or TEXT argument must be popped as BYTE or TEXT because 4GL provides no automatic data type conversion.

Passing Values to 4GL

C functions or programs can pass one or more arguments to 4GL by putting the arguments on the stack:

- When returning values to a 4GL program from a C function, you use the *external return functions* that are provided with 4GL to put the arguments on the stack.
- When passing values to a 4GL function from a C program, you use the *external push functions* that are provided with 4GL to put the arguments on the stack.

The external return functions copy their arguments to storage allocated outside the calling function. This storage is released when the returned value is popped. This situation makes it possible to return values from local variables of the function.

The external push functions do not make a copy of the pushed data value in allocated memory. They require a pushed value to be a variable that will be valid for the duration of the call. It is up to the calling code to dispose of the values, as necessary, after the call. Sections that follow describe the external return and push functions.

The Return Library Functions

The following 4GL library functions are available to return values:

```
extern void retint(mint iv)
extern void retshort(int2 siv)
extern void retlong(int4 lv)
extern void retflo(float *fv)
extern void retdub(double *dfv)
extern void retdec(dec_t *decv)

extern void retquote(int1 *str0)
extern void retstring(int1 *str0)
extern void retvchar(int1 *vc)

extern void retdate(int4 date)
extern void retdtime(dtime_t *dtv)

extern void retinv(intrvl_t *inv)
```

The correspondence between pre-Version 7.31 function names and new function names follows.

Old Function Name	New Function Name
retint	ibm_lib4gl_returnMInt
retshort	ibm_lib4gl_returnInt2
retlong	ibm_lib4gl_returnInt4
retflo	ibm_lib4gl_returnFloat
retdub	ibm_lib4gl_returnDouble
retdec	ibm_lib4gl_returnDecimal
retquote	ibm_lib4gl_returnQuotedStr
retstring	ibm_lib4gl_returnString
retvchar	ibm_lib4gl_returnVarChar
retdate	ibm_lib4gl_returnDate
retdtime	ibm_lib4gl_returnDateTime
retinv	ibm_lib4gl_returnInterval

The argument of **ibm_lib4gl_returnQuotedStr( )** is a null-terminated string. The **ibm_lib4gl_returnString( )** function is included only for symmetry; it internally calls **ibm_lib4gl_returnQuotedStr( )**.

No library function is available for returning BYTE or TEXT values, which are passed by reference.

The C function can return data in whatever form is convenient. If conversion is possible, 4GL converts the data type as required when popping the value. If data type conversion is not possible, an error occurs.

C functions called from 4GL must always exit with the statement **return(*n*)**, where ***n*** is the number of return values pushed onto the stack. A function that returns nothing must exit with **return(0)**.

The Push Library Functions

You can use the following 4GL library functions to push number values:

```
extern void pushint(mint iv)
extern void pushshort(int2 siv)
extern void pushlong(int4 liv)
extern void pushflo(float *fv)
extern void pushdub(double *dfv)
extern void pushdec(dec_t *decv, unsigned decp)
```

The correspondence between pre-Version 7.31 function names and new function names follows.

Old Function Name	New Function Name
pushint	ibm_lib4gl_pushMInt
pushshort	ibm_lib4gl_pushInt2
pushlong	ibm_lib4gl_pushInt4
pushflo	ibm_lib4gl_pushFloat
pushdub	ibm_lib4gl_pushDouble
pushdec	ibm_lib4gl_pushDecimal

The **dec_t** structure type and the C functions for manipulating decimal data are discussed later in this appendix. The second argument of **ibm_lib4gl_pushDecimal( )**, namely **decp**, specifies the decimal precision and scale.

For example, to give a decimal variable named **dec_var** the precision of 15 and the scale of 2, you could specify the following values:

```
ibm_lib4gl_pushDecimal(dec_var, PRECMAKE(15,2));
```

Here PRECMAKE is a macro defined in the **decimal.h** file.

You can use the following library functions to push character values:

```
extern void pushquote(int1 *cv, mint len)
extern void pushvchar(int1 *vcv, mint len)
```

The correspondence between pre-Version 7.31 function names and new function names follows.

Old Function Name	New Function Name
pushquote	ibm_lib4gl_pushQuotedStr
pushvchar	ibm_lib4gl_pushVarChar

The arguments to **ibm_lib4gl_pushQuotedStr( )** and **ibm_lib4gl_pushVarChar( )** are an unterminated character string and the count of characters that it contains (not including any null terminator).

You can use the following library functions to push DATE, DATETIME, and INTERVAL values:

```
extern void pushdate(int4 datv)
extern void pushdtime(dtime_t *dtv)
extern void pushinv(intrvl_t *inv)
```

The correspondence between pre-Version 7.31 function names and new function names follows.

Old Function Name	New Function Name
pushdate	ibm_lib4gl_pushDate
pushdtime	ibm_lib4gl_pushDateTime
pushinv	ibm_lib4gl_pushInterval

This library function pushes the location of a TEXT or BYTE argument:

```
extern void pushlocator(loc_t *blob)
```

The correspondence between pre-Version 7.31 function names and new function names follows.

Old Function Name	New Function Name
pushlocator	ibm_lib4gl_pushBlobLocator

Calling a C Function from a 4GL Program

To call a C function from a 4GL program, use the CALL statement and specify the following information:

- The name of the C function
- Any arguments to pass to the C function
- Any variables to return to the 4GL program

Important: *To run or debug a 4GL Rapid Development System program that calls C functions, you must first create a customized runner. For a complete description of this process, refer to* *"Creating a Customized Runner" on page 1-87*.

For example, the following CALL statement calls the C function **sendmsg( )**. It passes two arguments (**chartype** and **4**, respectively) to the function and expects two arguments to be passed back (**msg_status** and **return_code**, respectively):

```
CALL sendmsg(chartype, 4) RETURNING msg_status, return_code
```

The C function receives an integer argument that specifies how many values were pushed on the argument stack (in this case, two arguments). This is the number of values to be popped off the stack in the C function. The function also needs to return values for the **msg_status** and **return_code** arguments before passing control back to the 4GL program.

The C function should not assume it has been passed the correct number of stacked values. The C function should test its integer argument to see how many 4GL arguments were stacked for it. (If a function is called from two or more statements in the same source module, 4GL verifies that the same number of arguments is used in each call. A function could be called, however, from different source modules, with a different number of arguments from each module. This error, if it is an error, is not caught by 4GL.)

This example shows a C function that requires exactly one argument:

```
#include <fglsys.h>
#include <fgltypes.h>
...
...
mint nxt_bus_day(mint nargs);
{
   int4 theDate;
   if (nargs != 1)
   {
      fprintf(stderr,
         "nxt_bus_day: wrong number of parms (%d)\n",
         nargs );
      ibm_lib4gl_returnDate(0L);
      return(1);
   }
   ibm_lib4gl_popDate(&theDate);
   switch(rdayofweek(theDate))
   {
   case 5: /* change friday -> monday */
         ++theDate;
   case 6: /* saturday -> monday*/
         ++theDate;
   default: /* (sun..thur) go to next day */
         ++theDate;
   }
   ibm_lib4gl_returnDate(theDate); /* stack result */
   return(1) /* return count of stacked */
}
```

The function returns the date of the next business day after a given date. Because the function must receive exactly one argument, the function checks for the number of arguments passed. If the function receives a different number of arguments, it terminates the program (with an identifying message).

The C function in the next example can operate with one, two, or three arguments. The purpose of the function is to return the index of the next occurrence of a given character in a string. The string is the first argument and is required. The second argument is the character to search for; if it is omitted, a space character is used. The third argument is an offset at which to start the search; if it is omitted, zero is used.

```
#include <fglsys.h>
#include <fgltypes.h>
...
...
#define STSIZE 512+1
mint fglindex(mint nargs);
{
   int1 theString[STSIZE], theChar[2];
   mint offset, pos;
   *theChar = ''; /* initialize defaults */
   offset = 0;
   switch(nargs)
   {
   case 3: /* fglindex(s,c,n) */
      ibm_lib4gl_pushMInt(&offset);
   case 2: /* fglindex(s,c) */
      ibm_lib4gl_pushQuotedStr(theChar,2);
   case 1: /* fglindex(s) */
      ibm_lib4gl_pushQuotedStr(theString,STSIZE);
      break;
   default: /* zero or >3 parms, ret 0 */
      for(;nargs;nargs)
      ibm_lib4gl_pushQuotedStr(theString,STSIZE);
      ibm_lib4gl_returnMInt(999);
      return(1);
   }
   if (pos = index(theString+offset,*theChar) )
      ibm_lib4gl_returnMInt(offset+pos-1);
   else
      ibm_lib4gl_returnMInt(0);
   return(1);
}
```

The **switch** statement is useful in popping the correct number of arguments from the stack. By arranging the valid cases in descending order, the correct number of arguments can be popped in the correct sequence with minimal coding. In this example, the C function does not terminate the 4GL program when given an incorrect number of arguments. Instead, it disposes of all stacked arguments by popping them as character strings. Then it returns an impossible value.

Important: *A 4GL Rapid Development System program that calls C functions cannot specify as an argument to the C function a 4GL program variable whose scope of reference is global.*

Compiling and Executing the Program

The version of 4GL you are using determines how you compile and run a 4GL program that calls C functions. If you are using the 4GL Rapid Development System, you need to create a customized runner to handle the C functions. If you are using the C Compiler version, you do not need a customized runner. For complete information on compiling and executing 4GL programs, see Chapter 1, "Compiling INFORMIX-4GL Source Files." For information on creating a customized runner, see "RDS Programs That Call C Functions" on page 1-83.

Calling a 4GL Function from a C Program

4GL provides an application programming interface (API) with the C language that allows you to call 4GL functions from a C program. You can call either 4GL Rapid Development System functions or C Compiler functions.

To write a C program that calls 4GL functions

1. Include the **fglapi.h** header file.
2. Execute the **fgl_start( )** macro to perform initialization tasks.
3. Execute the **fgl_call( )** macro to call each 4GL function.
4. If the 4GL function displays a form, execute the **fgl_exitfm( )** macro to reset your terminal for character mode.
5. At the end of the program, execute the **fgl_end( )** macro to free resources.

To pass values between the C program and 4GL function, use the push and pop functions described in "Using the Argument Stack" on page C-3.

This section first explains how to use these features of the API with C:

- Including the **fglapi.h** file
- Initializing values
- Calling 4GL functions
- Handling Interrupt signals
- Compiling and executing the C program

Including the fglapi.h File

You must include the **fglapi.h** header file in any C program that calls 4GL functions. This header file defines the **fgl_start( )**, **fgl_call( )**, **fgl_exitfm( )**, and **fgl_end( )** macros and is located in the **$INFORMIXDIR/incl** directory. (See Appendix D, "Environment Variables,"for information on how to set the **INFORMIXDIR** environment variable.)

You can include **fglapi.h**, as demonstrated in the following example:

```
#include <fglapi.h>
o4Main()
{
   ...
}
```

Initializing the Argument Stack

Before you can call a 4GL function in a C program, you must execute the **fgl_start( )** macro. This macro performs the following actions:

- It initializes the argument stack so that you can pass arguments between the C program and the 4GL functions.
- If you are using the p-code compiler, it specifies the filename (and path) of the file that contains the 4GL functions.

You can execute this macro *once* per C program.

The following example demonstrates how to call the **fgl_start( )** macro. It specifies a file named **test** as the file that contains the 4GL functions:

```
#include <fglapi.h>

o4Main()
{
fgl_start("test");
...
}
```

If you compile the 4GL function to C code, the filename argument is optional and is ignored if you specify it. In this case, you can call **fgl_start( )** as follows:

```
#include <fglapi.h>

o4Main()
{
fgl_start();
...
}
```

The **fgl_start( )** macro is described in detail in "fgl_start( )" on page 21.

Invoking the 4GL Function

The C program must perform the following actions to call a 4GL function:

1. Push the argument values that the function expects onto the argument stack
2. Use the **fgl_call( )** macro to identify the name of the 4GL function and to tell it how many arguments to expect

The 4GL function must perform the following actions to receive arguments from and to pass values back to the C program:

1. Include the appropriate arguments in the FUNCTION statement
2. Use the DEFINE statement to define variables for all the arguments passed to the function
3. Use the RETURN statement in the 4GL function to return control to the C program and to list any values to pass to the calling C program

The C program can then pop the values passed from the function off the argument stack.

For example, the C program listed on the next page calls a 4GL function named **get_customer( )**.

The program passes one argument to the **get_customer( )** function. Then **get_customer( )** passes one argument back to the C program. The argument passed to the function is the filename and path of the demonstration database. The C program prompts the user for this filename.

The **get_customer( )** function displays a menu of the first 10 customers in the **customer** table of the specified database. The user then chooses a customer name from the menu, and the function passes the chosen name back to the C program. Finally, the C program displays the name of the customer.

```
#include <fglapi.h>
#include <fglsys.h>
#include <fgltypes.h>
#include <stdio.h>

o4Main()
{
   int1 str[80];

   fgl_start("example");
   printf("enter the full path name of a STORES database: ");
   fflush(stdout);
   scanf("%s", str);
   ibm_lib4gl_pushQuotedStr(str, strlen(str));
   fgl_call(get_customer, 1);
   ibm_lib4gl_popQuotedStr(str, 80);
   printf("name entered: %s\n", str);
   fgl_end();
}
```

The logic of the 4GL function **get_customer( )** is as follows:

```
FUNCTION get_customer(dbname)
   DEFINE dbname CHAR(30),
          cust_array ARRAY[50] of CHAR(15),
          i INT
   DATABASE dbname
   DECLARE c1 CURSOR FOR SELECT lname
      FROM customer ORDER BY lname

   LET i = 1
   FOREACH c1 INTO cust_array[i]
      LET i = i + 1
   END FOREACH

   MENU "enter name=>"
      COMMAND cust_array[1] RETURN cust_array[1]
      COMMAND cust_array[2] RETURN cust_array[2]
```

```
        COMMAND cust_array[3] RETURN cust_array[3]
        COMMAND cust_array[4] RETURN cust_array[4]
        COMMAND cust_array[5] RETURN cust_array[5]
        COMMAND cust_array[6] RETURN cust_array[6]
        COMMAND cust_array[7] RETURN cust_array[7]
        COMMAND cust_array[8] RETURN cust_array[8]
        COMMAND cust_array[9] RETURN cust_array[9]
        COMMAND cust_array[10] RETURN cust_array[10]
    END MENU
END FUNCTION
```

Using Interrupt Signals

An 4GL program can trap Interrupt signals by using the DEFER INTERRUPT and DEFER QUIT statements. When executing a C program that calls 4GL functions, you must be careful how you handle interrupts in the C program, so that you do not confuse the 4GL signal handling with any signal handling that occurs in the C program.

The **fgl_start( )** macro defines functions to call when interrupts occur. When one of these interrupts occurs, the appropriate function clears the screen and terminates the program.

By using DEFER INTERRUPT and DEFER QUIT within a 4GL function, you can control the processing that occurs when the interrupt is detected.

Compiling and Executing the C Program

The method by which you compile and execute a C program that calls a 4GL function is similar to the method you use to compile and execute a 4GL program. The following table shows the commands to compile a C program that calls 4GL functions based on the version of 4GL you are using.

Version of 4GL	Compilation Commands
C Compiler	**c4gl** command
RDS	**fglpc** and **cfglgo** commands

When compiling a C program that calls a 4GL function, you must specify the **-api** option of the compilation command. Do not specify the **fgiusr.c** file on the command line unless you are calling external C functions from 4GL.

The following examples illustrate the compilation and execution, using two source code files and one executable:

- The file **mymain.c** contains the C program.
- The file **my4gl.4gl** contains the 4GL function.
- The file **myprog.exe** is the resulting executable.

Compiling a C Program That Calls C Compiler Functions

To compile a C program that calls a C Compiler function, use the **c4gl** command as shown in the following example:

```
c4gl mymain.c my4gl.4gl -o mymain.exe
./mymain.exe
```

For complete information on the **c4gl** command, see "Compiling a 4GL Module" on page 1-35.

Compiling a C Program That Calls 4GL RDS Functions

To compile a C program that calls a compiled 4GL Rapid Development System function, use the **fglpc** and **cfglgo** commands as shown in the following example:

```
fglpc my4gl
cfglgo -api mymain.c -o mymain.exe
./mymain.exe my4gl
```

For complete information on the **fglpc** command, see "Compiling an RDS Source File" on page 1-77. For complete information on the **cfglgo** command, see "Creating a Customized Runner" on page 1-87.

Macros for Calling 4GL Functions

Four macros are provided with 4GL for you to use in C programs that call 4GL functions:

- **fgl_start( )**
- **fgl_call( )**
- **fgl_exitfm( )**
- **fgl_end( )**

These macros are described in the sections that follow.

fgl_start( )

The **fgl_start( )** macro initializes the 4GL argument stack, prepares for signal handling, and, if you are using the 4GL Rapid Development System, specifies the path of the file that contains the 4GL functions.

```
fgl_start(filename)
   int1  *filename;
```

filename	is the filename (and the directory path) of the file that contains the 4GL functions to call.

You can specify *filename* by using either a quoted string or a character variable. The file extension, **.4go** or **.4gi**, is optional.

The following list describes the return codes of **fgl_start( )** and the conditions that evoke them.

`0` The macro executed successfully.

`< 0` The macro failed.

You must specify the **fgl_start( )** macro before using any of the following items:

- The **fgl_call( )** macro
- The 4GL pushing or popping functions

Important: *To avoid confusion, you might want to make* ***fgl_start( )*** *the first function call in a C program that calls 4GL functions.*

If you are using the 4GL Rapid Development System, you must specify *filename*. If you are using the C Compiler, *filename* is optional. For compatibility, however, you might want to specify an empty string, such as " ", as a placeholder for *filename*. Specifying an empty string makes it easier to convert a C Compiler program to an RDS program.

This code example specifies **test** as the file that contains the 4GL functions:

```
#include <fglapi.h>

o4Main()
{
   ...
   fgl_start("test");
   ...
}
```

Once a 4GL function begins execution through use of the **fgl_call( )** macro, the function has access to the arguments passed to it by **fgl_call( )** and the command line arguments passed to the calling C function itself. The arguments passed by **fgl_call( )** are accessed by the 4GL function in the normal manner—through its argument list. The command-line arguments passed to the calling C function, however, are accessed by the 4GL function through use of the 4GL functions **ARG_VAL( )** and **NUM_ARGS( )**. These latter two functions operate in the normal way, as though the command-line arguments passed to the C function had been instead used as command-line arguments to execute the 4GL MAIN function block.

fgl_call( )

The **fgl_call( )** macro calls the 4GL function to execute. This macro passes the following arguments to the 4GL function:

- The name of the function
- The number of arguments being passed

The **fgl_call( )** macro returns the number of arguments being passed back to the program from the function.

```
fgl_call(funcname, nparams)
   int1* funcname;
   mint nparams;
```

funcname	is the name of the function to call.
nparams	is the number of arguments you are passing to the function.

You must push onto the argument stack any values to be passed to the 4GL function before executing the **fgl_call( )** macro. For more information on using the push functions, see "The Push Library Functions" on page C-10.

To read any arguments passed back to the C program from the 4GL function, use the pop functions. For more information on using the pop functions, see "Receiving Values from 4GL" on page C-5.

The following C source code pushes three arguments onto the argument stack, and then calls the **out_rep1( )** function:

```
#include <fglapi.h>
#include <fglsys.h>
...

o4Main()
{
   ...
      {
      fgl_start()
      ...
      ibm_lib4gl_pushQuotedStr(p->pw_name, strlen(p->pw_name));
      ibm_lib4gl_pushQuotedStr(p->pw_dir, strlen(p->pw_dir));
      ibm_lib4gl_pushMInt(p->pw_uid);
      fgl_call(out_rep1, 3);
      ...
      }
   ...
}
```

fgl_exitfm( )

The **fgl_exitfm( )** macro resets the terminal for character mode. Use this macro after calling a 4GL function that displays a form.

```
fgl_exitfm( )
```

Place this function after any **fgl_call( )** macro that causes 4GL to display one or more forms. This macro resets the terminal for character mode. If you do not execute this macro, the terminal might behave unusually, and the end user might be unable to enter any input.

The following example pushes a value onto the argument stack, calls the 4GL function, pops the returned value, and then executes the **fgl_exitfm( )** macro to reset the terminal to character mode:

```
#include <fglapi.h>
#include <fglsys.h>
#include <stdio.h>

o4Main()
{
   fgl_start()
   ...
   ibm_lib4gl_pushQuotedStr(str, strlen(str));
   fgl_call(get_customer, 1);
   ibm_lib4gl_popQuotedStr(str, 80);
   fgl_exitfm();
   ...
}
```

fgl_end( )

The **fgl_end( )** macro frees resources resulting from the execution of a C program that calls a 4GL function.

```
fgl_end( )
```

The **fgl_end( )** macro performs the following actions:

- Deletes any **temp** files created by TEXT or BYTE objects
- Closes any files opened by the 4GL function
- Frees the allocated memory

Call this macro at the end of a C program that calls a 4GL function.

The following example demonstrates popping the value returned from 4GL, printing this value, and then freeing resources:

```
#include <fglapi.h>
#include <fglsys.h>
#include <stdio.h>

o4Main()
{
   fgl_start()
   ...
   ibm_lib4gl_popQuotedStr(str, 80);
   printf("name entered: %s\n", str);
   fgl_end()
}
```

Decimal Functions for C

The data type DECIMAL is a machine-independent method for representing numbers of up to 32 significant digits, with or without a decimal point, and with exponents in the range -128 to +126. 4GL provides routines that facilitate the conversion of DECIMAL-type numbers to and from every data type allowed in the C language.

DECIMAL-type numbers consist of an exponent and a mantissa (or fractional part) in base 100. In normalized form, the first digit of the mantissa must be greater than zero.

When used within a C program, DECIMAL-type numbers are stored in a C structure of the following type:

```
#define DECSIZE 16

struct decimal
   {
   int2 dec_exp;
   int2 dec_pos;
   int2 dec_ndgts;
   int1  dec_dgts[DECSIZE];
   };

typedef struct decimal dec_t;
```

The **decimal** structure and the type definition **dec_t** can be found in the header file **decimal.h**. Include this file in all C source files that use any of the 4GL decimal functions.

The **decimal** structure has four parts:

- **dec_exp**. Holds the exponent of the normalized DECIMAL-type number. This exponent represents a power of 100.
- **dec_pos**. Holds the sign of the DECIMAL-type number (1 when the number is zero or greater; 0 when less than zero).
- **dec_ndgts**. Contains the number of base-100 significant digits of the DECIMAL-type number.
- **dec_dgts**. A character array that holds the significant digits of the normalized DECIMAL-type number (`dec_dgts[0] != 0`). Each character in the array is a one-byte binary number in base 100. The number of significant digits in **dec_dgts** is stored in **dec_ndgts**.

All operations on DECIMAL-type numbers should take place through the functions provided in the 4GL library, as described in the following pages. Any other operations, modifications, or analysis of DECIMAL-type numbers can produce unpredictable results.

The following C function calls are available in 4GL to process DECIMAL-type numbers.

Function	Effect	Page
deccvasc( )	Convert C **int1** type to DECIMAL type	C-28
dectoasc( )	Convert DECIMAL type to C **int1** type	C-30
deccvint( )	Convert C **int** type to DECIMAL type	C-32
dectoint( )	Convert DECIMAL type to C **int** type	C-33
deccvlong( )	Convert C **int4** type to DECIMAL type	C-34
dectolong( )	Convert DECIMAL type to C **int4** type	C-35
deccvflt( )	Convert C **float** type to DECIMAL type	C-36
dectoflt( )	Convert DECIMAL type to C **float** type	C-37
deccvdbl( )	Convert C **double** type to DECIMAL type	C-38
dectodbl( )	Convert DECIMAL type to C **double** type	C-39
decadd( )	Add two DECIMAL numbers	C-40
decsub( )	Subtract two DECIMAL numbers	C-40
decmul( )	Multiply two DECIMAL numbers	C-40
decdiv( )	Divide two DECIMAL numbers	C-40
deccmp( )	Compare two DECIMAL numbers	C-41
deccopy( )	Copy a DECIMAL number	C-42
dececvt( )	Convert DECIMAL value to ASCII string	C-43
decfcvt( )	Convert DECIMAL value to ASCII string	C-43

deccvasc( )

Use **deccvasc( )** to convert a value stored as a printable character in a C **int1** type into a DECIMAL-type number.

```
deccvasc(cp, len, np)
int1  *cp;
mint  len;
dec_t  *np;
```

cp	points to a string that holds the value to be converted. Leading blank spaces in the character string are ignored. The character string can have a leading plus (+) or minus (-) sign, a decimal point (.), and numbers to the right of the decimal point. The character string can contain an exponent preceded by either e or *E*. The exponent value can also be preceded by a plus or minus sign.
len	is the length of the string.
np	is a pointer to a **dec_t** structure that receives the result of the conversion.

These are the return codes of **deccvasc( )** and the conditions that evoke them:

`0`	Function was successful.
`-1200`	Number is too large to fit into a DECIMAL-type (overflow).
`-1201`	Number is too small to fit into a DECIMAL-type (underflow).
`-1213`	String has non-numeric characters.
`-1216`	String has bad exponent.

Example

The following segment of code gets the character string **input** from the terminal, and converts it to **number**, a DECIMAL-type number:

```
#include <decimal.h>

int1 input[80];
dec_t number;
      . . .
/* get input from terminal */
getline(input);

/* convert input into decimal number */
deccvasc(input, 32, &number);
```

dectoasc( )

Use **dectoasc( )** to convert a DECIMAL-type number to an ASCII string.

```
dectoasc(np, cp, len, right)
dec_t  *np;
int1  *cp;
mint  len;
mint  right;
```

np	is a pointer to the decimal structure whose associated decimal value is to be converted to an ASCII string.
cp	is a pointer to the beginning of the character buffer to hold the ASCII string.
len	is the maximum length (in bytes) of the string buffer. If the number does not fit into a character string of length len, **dectoasc( )** converts the number to exponential notation. If the number still does not fit, **dectoasc( )** fills the string with asterisks. If the number is shorter than the string, it is left-aligned and padded on the right with blank characters.
right	is an integer that indicates the number of decimal places to the right of the decimal point. If right equals -1, the number of decimal places is determined by the decimal value of **np*.

Because the ASCII string returned by **dectoasc( )** is not null-terminated, your program must add a null character to the string before printing it.

The following list describes the return codes of **dectoasc( )** and the conditions that evoke them:

`0`	Conversion was successful.
`-1`	Conversion was not successful.

Example

The following segment of code accepts the character string **input** from the terminal and converts it to **number**, a DECIMAL-type number. The **number** value is then converted to the character string **output**, a null character is appended, and the string is printed.

```
#include <decimal.h>

int1 input[80];
int1 output[16];
dec_t number;
   . . .
/* get input from terminal */
getline(input);

/* convert input into decimal number */
deccvasc(input, 32, &number);

/* convert number to ASCII string */
dectoasc(&number, output, 15, 1);

/* add null character to end of string prior to printing */
output[15] = ' ';

/* print the value just entered */
printf("You just entered %s", output);
```

deccvint( )

Use **deccvint( )** to convert a C type **int** to a DECIMAL-type number.

```
deccvint(integer, np)
mint  integer;
dec_t  *np;
```

integer	is the integer that is to be converted.
np	is a pointer to a **dec_t** structure that receives the result of the conversion.

The following list describes the return codes of **deccvint( )** and the conditions that evoke them:

`0`	Conversion was successful.
`-1`	Conversion was not successful.

Example

```
#include <decimal.h>

dec_t decnum;

/* convert the integer value -999
 * into a DECIMAL-type number
 */
deccvint(-999, &decnum);
```

dectoint( )

Use **dectoint( )** to convert a DECIMAL-type number to a C type **int**.

```
dectoint(np, ip)
dec_t  *np;
mint  *ip;
```

np	is a pointer to a decimal structure whose value is converted to an integer.
ip	is a pointer to the integer that receives the result of the conversion.

The following list describes the return codes of **dectoint( )** and the conditions that evoke them:

`0` Conversion was successful.

`-1200` The magnitude of the DECIMAL-type number is greater than 32,767.

Example

```
#include <decimal.h>

dec_t mydecimal;
mint myinteger;

/* convert the value in
 * mydecimal into an integer
 * and place the results in
 * the variable myinteger.
 */
dectoint(&mydecimal, &myinteger);
```

deccvlong( )

Use **deccvlong( )** to convert a C type **int4** value to a DECIMAL-type number.

```
deccvlong(lng, np)
int4  lng;
dec_t  *np;
```

lng	is a pointer to a **long** integer.
np	is a pointer to a **dec_t** structure that receives the result of the conversion.

Example

```
#include <decimal.h>

dec_t mydecimal;
int4 mylong;

/* Set the decimal structure
 * mydecimal to 37.
 */
deccvlong(37L, &mydecimal);
   . . .
mylong = 123456L;
/* Convert the variable mylong into
 * a DECIMAL-type number held in
 * mydecimal.
 */
deccvlong(mylong, &mydecimal);
```

dectolong( )

Use **dectolong( )** to convert a DECIMAL-type number to a C type **int4**.

```
dectolong(np, lngp)
   dec_t *np;
   int4  *lngp;
```

np	is a pointer to a decimal structure.
lngp	is a pointer to a **long** type that receives the result of the conversion.

These are the return codes of **dectolong( )** and the conditions that evoke them:

`0`	Conversion was successful.
`-1200`	Magnitude of the DECIMAL-type number exceeds 2,147,483,647.

Example

```
#include <decimal.h>

dec_t mydecimal;
int4 mylong;

/* convert the DECIMAL-type value
 * held in the decimal structure
 * mydecimal to a long pointed to
 * by mylong.
 */
dectolong(&mydecimal, &mylong);
```

deccvflt( )

Use **deccvflt( )** to convert a C type **float** to a DECIMAL-type number.

```
deccvflt(flt, np)
   float flt;
   dec_t *np;
```

flt	is a floating-point number.
np	is a pointer to a **dec_t** structure that receives the result of the conversion.

Example

```
#include <decimal.h>

dec_t mydecimal;
float myfloat;

/* Set the decimal structure
 * myfloat to 3.14159.
 */
deccvflt(3.14159, &mydecimal);

myfloat = 123456.78;

/* Convert the variable myfloat into
 * a DECIMAL-type number held in
 * mydecimal.
 */
deccvflt(myfloat, &mydecimal);
```

dectoflt( )

Use **dectoflt( )** to convert a DECIMAL-type number to a C type **float**.

```
dectoflt(np, fltp)
   dec_t *np;
   float *fltp;
```

np	is a pointer to a decimal structure.
fltp	is a pointer to a floating-point number that receives the result of the conversion.

On most implementations of C, the resulting floating-point number has eight significant digits.

Example

```
#include <decimal.h>

dec_t mydecimal;
float myfloat;

/* convert the DECIMAL-type value
 * held in the decimal structure
 * mydecimal to a floating point number pointed to
 * by myfloat.
 */
dectoflt(&mydecimal, &myfloat);
```

deccvdbl( )

Use **deccvdbl( )** to convert a C type **double** to a DECIMAL-type number.

```
deccvdbl(dbl, np)
   double dbl;
   dec_t *np;
```

dbl	is a double-precision floating-point number.
np	is a pointer to a **dec_t** structure that receives the result of the conversion.

Example

```
#include <decimal.h>

dec_t mydecimal;
double mydouble;

/* Set the decimal structure
 * mydecimal to 3.14159.
 */
deccvdbl(3.14159, &mydecimal);

mydouble = 123456.78;

/* Convert the variable mydouble into
 * a DECIMAL-type number held in
 * mydecimal.
 */
deccvdbl(mydouble, &mydecimal);
```

dectodbl( )

Use **dectodbl( )** to convert a DECIMAL-type number to a C type **double**.

```
dectodbl(np, dblp)
   dec_t *np;
   double *dblp;
```

np	is a pointer to a decimal structure.
dblp	is a pointer to a double-precision, floating-point number that receives the result of the conversion.

The resulting double-precision value receives a total of 16 significant digits on most implementations of the C language.

Example

```
#include <decimal.h>

dec_t mydecimal;
double mydouble;

/* convert the DECIMAL-type value
 * held in the decimal structure
 * mydecimal to a double pointed to
 * by mydouble.
 */
dectodbl(&mydecimal, &mydouble);
```

decadd( ), decsub( ), decmul( ), and decdiv( )

The decimal arithmetic routines take pointers to three decimal structures as parameters. The first two decimal structures hold the operands of the arithmetic function. The third decimal structure holds the result.

```
decadd(n1, n2, result)
   dec_t  *n1;
   dec_t  *n2;
   dec_t  *result;  /* result = n1 + n2 */

decsub(n1, n2, result)
   dec_t  *n1;
   dec_t  *n2;
   dec_t  *result;  /* result = n1 - n2 */

decmul(n1, n2, result)
   dec_t  *n1;
   dec_t  *n2;
   dec_t  *result;  /* result = n1 * n2 */

decdiv(n1, n2, result)
   dec_t  *n1;
   dec_t  *n2;
   dec_t  *result;  /* result = n1 / n2 */
```

n1	is a pointer to the decimal structure of the first operand.
n2	is a pointer to the decimal structure of the second operand.
result	is a pointer to the decimal structure of the result of the operation. The result value can use the same pointer as either *n1* or *n2*.

The following list describes the return codes of the decimal arithmetic routines and the conditions that evoke them:

`0` Operation was successful.

`-1200` Operation resulted in overflow.

`-1201` Operation resulted in underflow.

`-1202` Operation attempts to divide by zero.

deccmp( )

Use **deccmp( )** to compare two DECIMAL-type numbers.

```
mint deccmp(n1, n2)
   dec_t *n1;
   dec_t *n2;
```

n1	is a pointer to the decimal structure of the first number.
n2	is a pointer to the decimal structure of the second number.

The following list describes the return codes of **deccmp( )** and the conditions that evoke them:

`0` The two values are the same.

`-1` The first value is less than the second.

`+1` The first value is greater than the second.

deccopy( )

Use **deccopy( )** to copy the value of one **dec_t** structure to another.

```
deccopy(n1, n2)
   dec_t *n1;
   dec_t *n2;
```

n1	is a pointer to the source **dec_t** structure.
n2	is a pointer to the destination **dec_t** structure.

dececvt( ) and decfcvt( )

These functions convert a DECIMAL value to an ASCII string.

```
int1 *dececvt(np, ndigit, decpt, sign)
   dec_t *np;
   mint ndigit;
   mint *decpt;
   mint *sign;

int1 *decfcvt(np, ndigit, decpt, sign)
   dec_t *np;
   mint ndigit;
   mint *decpt;
   mint *sign;
```

np	is a pointer to a **dec_t** structure that contains the value of the number that is to be converted to a string.
ndigit	is, for **dececvt( )**, the length of the ASCII string; for **decfcvt( )**, it is the number of digits to the right of the decimal point.
decpt	points to an integer that is the position of the decimal point relative to the beginning of the string. A negative value for *decpt means to the left of the returned digits.
sign	is a pointer to the sign of the result. If the sign of the result is negative, *sign is nonzero; otherwise, the value is zero.

The **dececvt( )** function converts the decimal value pointed to by *np* into a null-terminated string of *ndigit* ASCII digits and returns a pointer to the string.

The low-order digit of the DECIMAL number is rounded.

The **decfcvt( )** function is identical to **dececvt( )** except that *ndigit* specifies the number of digits to the right of the decimal point instead of the total number of digits.

Examples

In the following example, **np** points to a **dec_t** structure that contains `12345.67`, and ***decpt** points to an integer that contains a `5`:

```
ptr = dececvt (np,4,&decpt,&sign);  = 1235
ptr = dececvt (np,10,&decpt,&sign); = 1234567000
ptr = decfcvt (np,1,&decpt,&sign);  = 123457
ptr = decfcvt (np,3,&decpt,&sign);  = 12345670
```

In this example, **np** points to a **dec_t** structure that contains a `0.001234` value, and ***decpt** points to an integer that contains a `-2` value:

```
ptr = dececvt (np,4,&decpt,&sign);  = 1234
ptr = dececvt (np,10,&decpt,&sign); = 1234000000
ptr = decfcvt (np,1,&decpt,&sign);  =
ptr = decfcvt (np,3,&decpt,&sign);  = 1
```

Appendix

D

Environment Variables

Various *environment variables* can affect the functionality of your INFORMIX-4GL program. These environment variables can describe your terminal, specify search paths for files, or define other parameters.

This appendix describes environment variables that affect 4GL and shows how to set them. It is divided into three main sections:

- Informix environment variables

  This section describes some Informix-defined environment variables that are used with 4GL. Many of these variables are not for frequent use but are included in case they are necessary for correct operation of 4GL.

GLS

- GLS environment variables

  You must set some or all of these variables to benefit from GLS (global language support) if you are developing or running a 4GL application for a locale other than U.S. English. GLS is described in Appendix E, "Developing Applications with Global Language Support." ♦

- UNIX environment variables that your Informix database server recognizes

  This section describes some standard UNIX environment variables that are recognized by Informix products.

Some environment variables are required; others are optional. For example, you must set several UNIX environment variables (or else accept their default settings). This appendix also identifies some environment variables that this version of 4GL ignores although earlier 4GL versions recognized them. (See also the *Informix Guide to SQL: Reference* for information about other environment variables that can affect Informix database servers.)

Where to Set Environment Variables

You can set Informix, GLS, and UNIX environment variables in three ways:

- At the system prompt on the command line

  If you set an environment variable at the system prompt, you must reassign it the next time that you log in to the system.

- In a special shell file, depending on your UNIX shell as follows:
  - For the C shell, **.login** or **.cshrc**
  - For the Bourne shell or the Korn shell, .profile

  An environment variable that is set in your .login, **.cshrc**, or **.profile** file is assigned automatically whenever you log in to the system.

Warning: *Make sure that you do not inadvertently set an environment variable differently in your* ***.login*** *and* ***.cshrc*** *C shell files.*

- In an environment-configuration file

  You can define all the environment variables that are used by Informix products in this common or private file. Using a configuration file reduces the number of environment variables that you must set at the command line or in a shell file.

An environment-configuration file can contain comment lines (preceded by #) and variable lines and their values (separated by blanks and tabs), as in the following example:

```
# This is an example of an environment-configuration file
#
# These are Informix-defined variable settings
#
DBDATE DMY4-
DBFORMAT *:.:,:DM
DBLANG german
```

Use the **ENVIGNORE** environment variable to later override one or more entries in this file. Use the following Informix **chkenv** utility to check the contents of an environment-configuration file and return an error message if there is a bad environment variable entry in the file or if the file is too large:

```
chkenv filename
```

The **chkenv** utility is described in the *Informix Guide to SQL: Reference*, in the chapter about SQL utilities.

The common (shared) environment-configuration file resides in **$INFORMIXDIR/etc/informix.rc**. The permission for this shared file must be set to 644. A private environment-configuration file must be stored in the user's home directory as **.informix** and must be readable by the user.

Important: *The first time that you set an environment variable in a shell or configuration file, before you begin work with 4GL, first log out and log back in, and then "**source**" the file (with C shell), or use "." to execute an environment-configuration file (with a Bourne or Korn shell). This action allows the process to read your new setting. Resetting environment variables during a 4GL session generally has no effect on any 4GL program that is running because 4GL uses the settings that were in effect when you logged in rather than when program execution commenced.*

How to Set Environment Variables

You can change default settings and add new ones by setting one or more of the environment variables recognized by your Informix product. If you are already using an Informix product, some or all the appropriate environment variables might already be set.

After one or more Informix products have been installed, enter the following command at the system prompt to view your current environment settings.

System	Command
BSD UNIX	`env`
UNIX System V	`printenv`

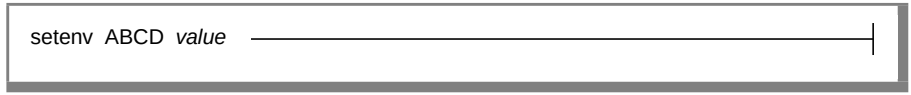
Use standard UNIX commands to set environment variables. Depending on the type of shell that you use, the following table shows how you set the (fictional) ABCD environment variable to *value*.

Shell	Command
C	`setenv ABCD` *value*
Bourne or Korn	`ABCD=`*value* `export ABCD`
Korn	`export ABCD=`*value*

When Bourne shell settings are shown in this appendix, the Korn shell (a superset of the Bourne shell) is always implied as well. Korn shell syntax allows for a shortcut, as shown in the preceding table.

Important: *The environment variables are case sensitive. For example, if you specify a lowercase letter where uppercase is required, your setting might result in the default value being used, rather than what you set, or other unexpected behavior.*

The following diagram shows how the syntax for setting an environment variable is represented in this appendix. These diagrams always indicate the setting for the C shell; for the Bourne or Korn shell, follow the syntax in the preceding table.



For more information on how to read syntax diagrams, see "How to Read a Syntax Diagram" on page 15 in the Introduction to this manual.

To unset most of the environment variables shown in this appendix, enter the following command.

Shell	Command
C	`unsetenv ABCD`
Bourne or Korn	`unset ABCD`

Default Environment Variable Settings

The following are default assumptions that Informix products make about your environment. Environment variables that can change specific default values are shown in parentheses. Other product-specific default values are described where appropriate in this appendix:

- The program, preprocessor, and any associated files and libraries of your product have been installed in the **/usr/informix** directory.
- The default database server for explicit or implicit connections is indicated by an entry in the **$INFORMIXDIR/etc/sqlhosts** file. (**INFORMIXSERVER**)
- The default directory for message files is **$INFORMIXDIR/msg**. (**DBLANG** unset; the **LANG** variable is no longer used by 4GL)
- If you are using INFORMIX-SE, the default or current database is in the current directory. (**DBPATH**)

- Temporary files for INFORMIX-SE are stored in the **/tmp** directory. (**DBTEMP**)
- The default terminal-dependent keyboard and screen capabilities are defined in the **termcap** file in the **$INFORMIXDIR/etc** directory. (**INFORMIXTERM**)
- For products that use an editor, the default editor is the predominant editor for the operating system, usually **vi**. (**DBEDIT**)
- The UNIX utility that sends files to a printer:
  - For UNIX System V, use **lp**.
  - For BSD and other UNIX systems (**DBPRINT**), use **lpr**.
- In the default locale (U.S. English), the default format for money values is `$000.00`. (**DBMONEY** set to `$.`)
- In the default locale (U.S. English), the default format for date values is `MM/DD/YYYY`. (**DBDATE** set to `MDY4/`)
- The default field separator for data files of LOAD and UNLOAD statements is the pipe symbol (`|=ASCII 124`). (**DBDELIMITER** set to `|`)

List of Environment Variables

The following tables list Informix, GLS, and UNIX environment variables that can be set (or should *not* be set) for an Informix database server or for 4GL.

Informix Environment Variable	Restrictions	Page
C4GLFLAGS		D-10
C4GLNOPARAMCHK		D-11
CC	For code compiled to C only	D-12
COLUMNS		D-12
DBANSIWARN		D-14
DBCENTURY		D-15
DBDATE		D-17

(1 of 3)

Informix Environment Variable	Restrictions	Page
DBDELIMITER		D-19
DBEDIT		D-20
DBESCWT		D-21
DBFORM		D-23
DBFORMAT		D-25
DBLANG		D-28
DBMONEY		D-30
DBPATH		D-32
DBPRINT		D-35
DBREMOTECMD	Dynamic Server only	D-36
DBSPACETEMP	Dynamic Server only	D-37
DBSRC	Interactive Debugger only	D-38
DBTEMP	INFORMIX-SE only	D-39
DBTIME	Not used by 4GL	D-39
DBUPSPACE		D-40
ENVIGNORE		D-41
FET_BUF_SIZE		D-42
FGLPCFLAGS		D-43
FGLSKIPNXTPG		D-43
INFORMIXC	For code compiled to C only	D-44
INFORMIXCONRETRY		D-44
INFORMIXCONTIME		D-45
INFORMIXDIR		D-47
INFORMIXSERVER		D-48

(2 of 3)

Informix Environment Variable	Restrictions	Page
INFORMIXSHMBASE	Dynamic Server only	D-49
INFORMIXSTACKSIZE	Dynamic Server only	D-50
INFORMIXTERM		D-51
IXOLDFLDSCOPE		D-53
LINES		D-55
ONCONFIG	Dynamic Server only	D-56
PDQPRIORITY		D-57
PROGRAM_DESIGN_DBS		D-58
PSORT_DBTEMP	Dynamic Server only	D-60
PSORT_NPROCS	Dynamic Server only	D-61
SQLEXEC	No longer used	D-62
SQLRM	Must be unset	D-64
SQLRMDIR	Must be unset	D-65
SUPOUTPIPEMSG		D-63

(3 of 3)

GLS

The following table lists the GLS environment variables.

GLS Environment Variable	Restrictions
CLIENT_LOCALE	
COLLCHAR	No longer used
DBAPICODE	
DB_LOCALE	
DBNLS	
GL_DATE	

(1 of 2)

GLS Environment Variable	Restrictions
GL_DATETIME	
LANG	No longer used
SERVER_LOCALE	

(2 of 2)

For information about these GLS environment variables, see the *Informix Guide to GLS Functionality* and the *Informix GLS Programmer's Manual*. ♦

The following table lists the UNIX environment variables.

UNIX Environment Variable	Page
PATH	D-67
TERM	D-69
TERMCAP	D-70
TERMINFO	D-71

For additional information, see the *Informix Guide to SQL: Reference*.

Informix Environment Variables

This section lists alphabetically the environment variables that you can set when you use 4GL and identifies some that should not be set.

C4GLFLAGS

The environment variables **C4GLFLAGS** allows you to set certain compiler options as defaults so that they need not appear in the command line, simplifying many compilations. Typical values follow:

-a	Checks array bounds at runtime.
-ansi	Issues warnings for Informix extensions to SQL syntax.
-anyerr	Resets **status** when 4GL expressions are evaluated.
-keep	Retains intermediate files during compilation.
-shared	Uses shared 4GL program libraries.
-z	Supports functions with a variable number of arguments.

Informix recommends compiling all 4GL programs with **-anyerr**. If you use compiled 4GL and your port supports shared libraries, you can improve system performance and reduce program size by using the **-shared** option.

Do not specify **-phase**, **-e**, or **-c** (or, for the C compiler, **-P**, **-E**, or **-S**) in this environment variable because these options prevent compilation from completing. For example, if you specify **-e** in the **C4GLFLAGS** variable, the system always stops compiling after producing a **.c** file, without producing object files or executable files. (See also **FGLPCFLAGS** later in this appendix.)

C4GLNOPARAMCHK

C4GLNOPARAMCHK is an environment variable that, if set, turns off the error checking on the number of arguments passed into a function and on the number of values returned from it. By default, these items are now checked as stringently in 4GL that is compiled to C as they are in the p-code system (RDS).

Compiled 4GL has always been different from RDS in how it checks the number of arguments passed to a function and the number of values returned from a function. RDS requires the number of arguments passed and the number of values returned to be correct. By default, 4GL now checks the number of arguments and number of return values, and generates a fatal error if the numbers are incorrect. Code that worked under RDS also works with the new compiled code; only code that did not work in p-code fails.

You can disable this checking mechanism by setting **C4GLNOPARAMCHK** *at compile time*. The variable **C4GLNOPARAMCHK** must be set and exported in the user environment at compile time. It can have any value or no value.

The ability to turn off parameter-count checking via **C4GLNOPARAMCHK** is provided as a migration aid only; it helps 4GL developers locate and correct parameter mismatches over time.

The following behavior regarding parameter-count checking depends on **C4GLNOPARAMCHK** and on the error scope:

- If the **C4GLNOPARAMCHK** environment variable is not set, all function calls and returned values are checked; an error is issued if there is a mismatch in the number of parameters passed or returned.
- If **C4GLNOPARAMCHK** is set and AnyError error scope is *not* in effect, parameter count checking code is not generated.
- If **C4GLNOPARAMCHK** is set and AnyError handling *is* in effect, if the WHENEVER ERROR action is CONTINUE, no parameter-count checking is generated; for any other error actions (such as STOP, GOTO, CALL), code for parameter-count checking is generated.

CC

The **c4gl** command uses the **INFORMIXC** and **CC** environment variables (defaulting to **cc** on most computers) in the final stage of compilation. Setting one of these environment variables lets you substitute any C compiler. Because **CC** is recognized by many versions of **make**, this environment variable is also compatible with other UNIX programs. You can use the following Bourne shell code to determine the compiler:

```
${INFORMIXC:=${CC:-cc}}
```

The C compiler used is the value of the **INFORMIXC** environment variable if it is not empty. If **INFORMIXC** is empty, the (non-empty) value of the **CC** environment variable is used. If both **CC** and **INFORMIXC** are empty, the default is **cc**.

Important: *If you use **gcc**, be aware that 4GL assumes that strings are writable, so you need to compile using the **-fwritable-strings** flag. Failure to specify this option can cause unpredictable results, possibly including core dumps.*

COLUMNS

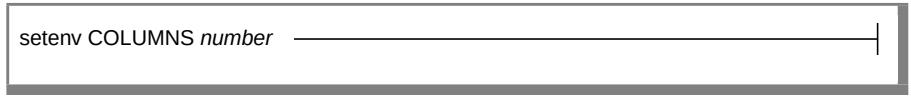
UNIX platforms support various ways to control the sizes of screens and windows in terms of lines (or rows) and columns. Depending on the method that your platform uses, two environment variables, **COLUMNS** and **LINES**, might be useful in controlling the character dimensions of your screen.

One common way to control the dimensions of screens is the use of input/output control (**ioctl()**) calls. To see if your platform uses this method, enter the command `stty -a`. If the response includes explicit values for rows and columns, **ioctl()** control is in effect, as in the following example:

```
% stty -a
speed 9600 baud;
rows = 24; columns = 80;
intr = ^c; quit = ^|; erase = ^h; kill = ^u;
```

If your platform uses ioctl() calls, the operating system or windowing facility probably provides a way to resize the screen using the mouse or trackball.

If your platform does not use **ioctl()** calls to control screen dimensions, you can use the **LINES** and **COLUMNS** environment variables to specify the screen dimensions. Use the following syntax for setting **COLUMNS**.



Element	Description
number	is a literal integer, specifying the horizontal width of the screen in columns (sometimes called character positions) in the screen display.

On such platforms, if **LINES** or **COLUMNS** is not set, the corresponding value is taken from the **rows** or **columns** field in the **terminfo** or **termcap** entry in use, as indicated by the **TERM** environment variable.

The following example sets **COLUMNS** to `80` and **LINES** to `24`.

C shell	Bourne or Korn shell

`setenv COLUMNS 80`
`setenv LINES 24`
	`COLUMNS=80`
	`LINES=24`
	`export COLUMNS LINES`

If either **LINES** or **COLUMNS** is set to an invalid value (that is, to a value that is not a positive integer), the invalid value is ignored, and the required value is read from the **termcap** or **terminfo** entry as applicable.

DBANSIWARN

The **DBANSIWARN** environment variable indicates that you want to check for Informix extensions to the ANSI standard for SQL syntax. Unlike most environment variables, you do not need to set **DBANSIWARN** to a value. Setting it to any value or to no value, as follows, is sufficient.

```
setenv DBANSIWARN
```

Setting **DBANSIWARN** before you compile a 4GL program is functionally equivalent to including the **-ansi** flag in the command line. When the 4GL preprocessor recognizes Informix extensions to the ANSI standard for SQL syntax in your source code during compilation, warning messages are written to the screen. (Only SQL statements can cause such warnings; 4GL statements that are not SQL statements are not affected.)

Similarly, setting the **DBANSIWARN** environment variable before executing a 4GL program is functionally equivalent to including the **-ansi** flag at the command line. If you set **DBANSIWARN**, a warning is displayed on the screen at runtime if 4GL detects an SQL statement that includes an Informix syntax extension to the ANSI standard for SQL.

At runtime, **DBANSIWARN** causes the SQL Communication Area (**SQLCA**) variable **SQLCA.SQLWARN.SQLAWARN[6]** to be set to `W` when 4GL detects that an SQL statement that is not ANSI compliant is executed. (For more information on the **SQLCA** global record, see "Error Handling with SQLCA" on page 2-45 or refer to the *Informix Guide to SQL: Reference*.)

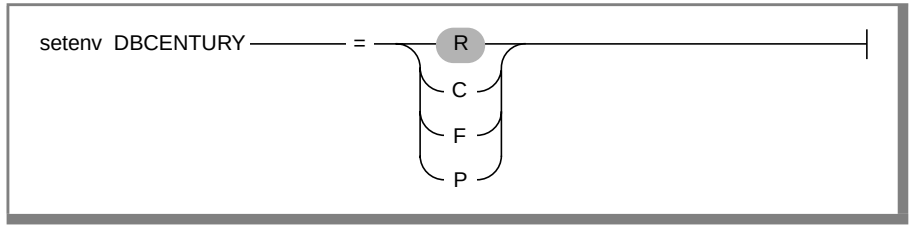
Once you set **DBANSIWARN**, Informix extension checking is automatic until you log out or unset **DBANSIWARN**. To turn off checking for Informix syntax extensions, you can disable **DBANSIWARN** with the following command.

Shell	Command
C	`unsetenv DBANSIWARN`
Bourne	`unset DBANSIWARN`

DBCENTURY

The **DBCENTURY** environment variable specifies how to expand abbreviated one- and two-digit *year* specifications within DATE and DATETIME values. Expansion is based on this setting (and on the system clock at runtime).

In most versions prior to 4GL 7.2, if only the two trailing digits of a year were entered for literal DATE or DATETIME values, these digits were prefixed with *19*; thus, `12/31/01` was always expanded to `12/31/1901`. **DBCENTURY** supports four new algorithms to expand abbreviated years into four-digit year values that end with the same digits (or digit) that the user entered.



Element	Description
C	Use the past, future, or current year closest to the current date.
F	Use the nearest year in the future to expand the entered value.
P	Use the nearest year in the past to expand the entered value.
R	Prefix the entered value with the first two digits of the current year.

Here *past*, *current*, and *future* are all relative to the system clock.

Values are case sensitive; only these four uppercase letters are valid. If you specify anything else (for example, a lowercase letter), or if **DBCENTURY** is not set, `R` is used as the default.

Three-digit years are not expanded. If a year is entered as a single digit, it is first expanded to two digits by prefixing it with a zero; **DBCENTURY** then expands this value to four digits. Years before 99 AD (or CE) require leading zeros (to avoid expansion).

Important: *If the database server and the client system have different settings for* ***DBCENTURY****, the client system setting takes precedence for abbreviations of years in dates entered through the 4GL application. Expansion is sensitive to the time of execution and to the accuracy of the system clock-calendar. You can avoid the need to rely on* ***DBCENTURY*** *by requiring the user to enter four-digit years or by setting the CENTURY attribute in the form specification of DATE and DATETIME fields.*

The following examples illustrate the effect of various **DBCENTURY** settings on expanding DATE and DATETIME values that include abbreviated years.

Runtime Date	11/11/1999			
Abbreviated Value	`1/1/10`			
DBCENTURY Setting	`R`	`C`	`F`	`P`
Resulting Value	01/01/1910	01/01/2010	01/01/2010	01/01/1910
Runtime Date	11/11/1999			
Abbreviated Value	`1/1/0`			
DBCENTURY Setting	`R`	`C`	`F`	`P`
Resulting Value	01/01/1900	01/01/2000	01/01/2000	01/01/1900
Runtime Date	04/06/2010			
Abbreviated Value	`1/1/50`			
DBCENTURY Setting	`R`	`C`	`F`	`P`
Resulting Value	01/01/2050	01/01/2050	01/01/2050	01/01/1950
Runtime Date	4/6/1999			
Abbreviated Value	`DATETIME (1-1) YEAR TO MONTH`			
DBCENTURY Setting	`R`	`C`	`F`	`P`
Resulting Value	1901-1	2001-1	2001-1	1901-1
Runtime Date	11/11/1999			
Abbreviated Value	`DATETIME (1-12) YEAR TO MONTH`			
DBCENTURY Setting	`R`	`C`	`F`	`P`
Resulting Value	1901-12	2001-12	2001-12	1901-12

(1 of 2)

Runtime Date	04/06/2010			
Abbreviated Value	DATETIME (50-1) YEAR TO MONTH			
DBCENTURY Setting	R	C	F	P
Resulting Value	2050-1	2050-1	2050-1	1950-1

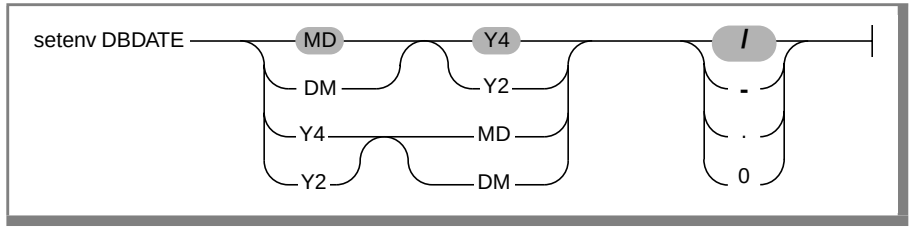
(2 of 2)

A limitation of **DBCENTURY** is that it specifies a single global algorithm for all abbreviated years and applies that throughout the 4GL program. (For a more flexible feature, see the CENTURY field attribute in Chapter 6, "Screen Forms." CENTURY can apply different expansion algorithms to abbreviated years at the individual field level.)

DBDATE

DBDATE specifies the default display format for DATE values by indicating the following items:

- The order of the *month*, *day*, and *year* time units within a date
- Whether the *year* is printed with two digits (Y2) or four digits (Y4)
- The time-unit separator between the *month*, *day*, and *year*



Element	Description
-, .	are characters that are valid as time-unit separators.
/	is the default time-unit separator for the default locale.
0	is a zero that indicates no time-unit separator.
D	is a character representing the day of the month.
M	is a character representing the month.
Y2	are characters that abbreviate the year as its last two digits.
Y4	are characters that represent the year as four digits.

Blank spaces are not valid between elements of the **DBDATE** setting. In the default locale, the default setting is MDY4/, where M represents the month, D represents the day, Y4 represents a four-digit year, and the slash (/) is a time-unit separator (for example, 9/22/1999 for September 22, 1999).

Any printable character of your locale is valid as the time-unit separator except the digits one through nine. The zero (0) specifies that there is no separator.

The slash (/) appears as the default separator if you attempt to specify any numeric character other than zero as the separator, or if you do not include any separator character in the **DBDATE** specification. The hyphen appears as the default separator in some non-English locales.

You must always specify the separator character last. The number of digits that you specify for the year must always follow the Y.

Date formatting specified in a USING clause or FORMAT attribute overrides the formatting specified in **DBDATE**.

To make the date appear as *mmddyy*, set **DBDATE** as follows.

Shell	Command
C	`setenv DBDATE MDY20`
Bourne	`DBDATE=MDY20` `export DBDATE`

MDY represents the order of month, day, and year; 2 indicates two digits for the year; and 0 specifies no separator. As a result, the date is displayed as 122599.

To make the date appear in European format (*dd-mm-yyyy*), set the **DBDATE** environment variable as follows.

Shell	Command
C	`setenv DBDATE DMY4-`
Bourne	`DBDATE=DMY4-` `export DBDATE`

DMY represents the order of day, month, and year; 4 indicates four digits for the year; and - specifies a hyphen separator. As a result, the date is displayed as 25-12-1999.

DBDELIMITER

The **DBDELIMITER** environment variable specifies the field delimiter used by the **dbexport** utility in unloaded data files or with the LOAD and UNLOAD statements in 4GL.

setenv DBDELIMITER '*delimiter*'

Element	Description
delimiter	is the default field delimiter for unloaded data files.

Any single character that is valid in the codeset of your locale is allowed, but do not specify any of the following characters:

- Hexadecimal numbers (*0* through *9*, *a* through *f*, *A* through *F*)
- Newline character *or* CONTROL-J
- The backslash (\ ) symbol
- The minus or hyphen (-) symbol (ASCII 45)
- The period (.) symbol (ASCII 46)
- Any character that your locale (or the **DBFORMAT** or **DBMONEY** environment variable) identifies as the decimal separator.

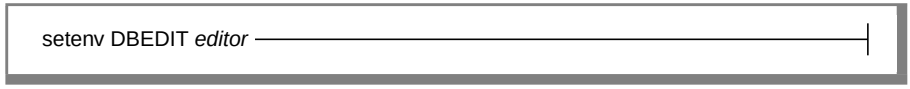
Warning: *No error message is issued if you specify any of these "forbidden" characters, but doing so can result in files of unloaded data that cannot be reloaded. The utilities that unload and load data do not automatically insert escape characters before the sign or before the decimal separator of number values.*

The pipe (|) symbol (ASCII 124) is the default. To change the field delimiter to a plus sign, set the **DBDELIMITER** environment variable as follows.

Shell	Command
C	`setenv DBDELIMITER '+'`
Bourne	`DBDELIMITER='+'` `export DBDELIMITER`

DBEDIT

The **DBEDIT** environment variable specifies the default text editor. If **DBEDIT** is set, the specified editor is called directly. If **DBEDIT** is not set, you are prompted to specify an editor as the default for the rest of the 4GL session.



Element	Description
editor	is the name of the text editor that you want as the default.

For most systems, the default editor is **vi**. If you use another editor, be sure that it creates ASCII files. Some word processors in *document mode* introduce printer control characters that can interfere with operation of 4GL.

To specify the **emacs** text editor, set the **DBEDIT** environment variable as follows.

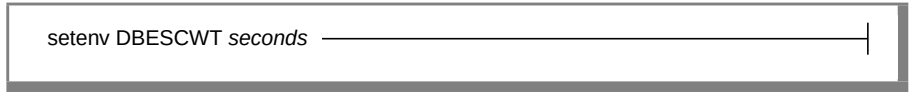
Shell	Command
C	`setenv DBEDIT emacs`
Bourne	`DBEDIT=emacs` `export DBEDIT`

DBESCWT

The **DBESCWT** environment variable controls the way that 4GL programs interpret the sequence of characters that the function keys and arrow keys send on some types of terminals. When a user presses one of these keys, many terminal types send a sequence of characters that starts with the ESC character. 4GL uses the ESC character as the default ACCEPT key, so that after reading an ESC character, the software must read the next character to see whether it is one of those that make up a function key or arrow key sequence.

If the computer or network is slow, the runtime software might interpret a function key keystroke as a distinct ESC character, later followed by one or more characters echoed to the screen. **DBESCWT** enables the programmer to distinguish such sequences by specifying the maximum delay between an ESC character and any subsequent character of the same escape sequence.

The syntax for setting the **DBESCWT** environment variable follows.



Element	Description
seconds	*is a value for the delay, in seconds, where 1 ≤ seconds ≤ 60.*

You can set **DBESCWT** to a value between 1 and 60, indicating the number of seconds that the software waits after it receives the ESC character from the keyboard before it decides that you have hit ESC rather than a function or arrow key. If **DBESCWT** is not set, the default wait time is one second.

The following examples instruct 4GL to wait up to 2 seconds at runtime before interpreting an ESC character received from the keyboard as a distinct ESC character (rather than as the beginning of the escape sequence for an arrow key or function key).

C shell	Bourne or Korn shell
`setenv DBESCWT 2`	`DBESCWT=2` `export DBESCWT`

Function and arrow keys normally generate escape sequences faster than a typist can type, so the 4GL application usually can make the distinction between the escape sequences of special keys and users typing the ESCAPE key. Only use **DBESCWT** on systems with poor response times or where the software is misinterpreting arrow and function key sequences. Setting **DBESCWT** slows the performance of your 4GL application.

DBFORM

The **DBFORM** environment variable specifies the subdirectory of **$INFORMIXDIR** (or full pathname) in which the menu form files for the currently active language reside. (**$INFORMIXDIR** stands for the name of the directory referenced by the environment variable **INFORMIXDIR**.)

Menu form files provide a set of language-translated menus to replace the standard 4GL menus. Menu form files have the suffix **.frm**. Menu form files are included in language supplements, which contain instructions specifying where the files should be installed and what **DBFORM** settings to specify .

setenv DBFORM *pathname*

Element	Description
pathname	specifies the subdirectory of **$INFORMIXDIR** or the full pathname of the directory that contains the message files.

If **DBFORM** is not set, the default directory for menu form files is **$INFORMIXDIR**/**forms/english**. The files should be installed in a subdirectory under the **forms** subdirectory under **$INFORMIXDIR**. For example, French menu files could be installed in **$INFORMIXDIR/forms/french** or in **$INFORMIXDIR/forms/fr.88591**. The English language version will normally be installed in **$INFORMIXDIR/forms** or **$INFORMIXDIR/forms/english**. Non-English menu form files should not be installed in either of the locations where English files are normally found.

Figure D-1 illustrates the search method employed for locating message files for a specific language (where the value set in the **DBFORM** environment variable is indicated as **$DBFORM**).

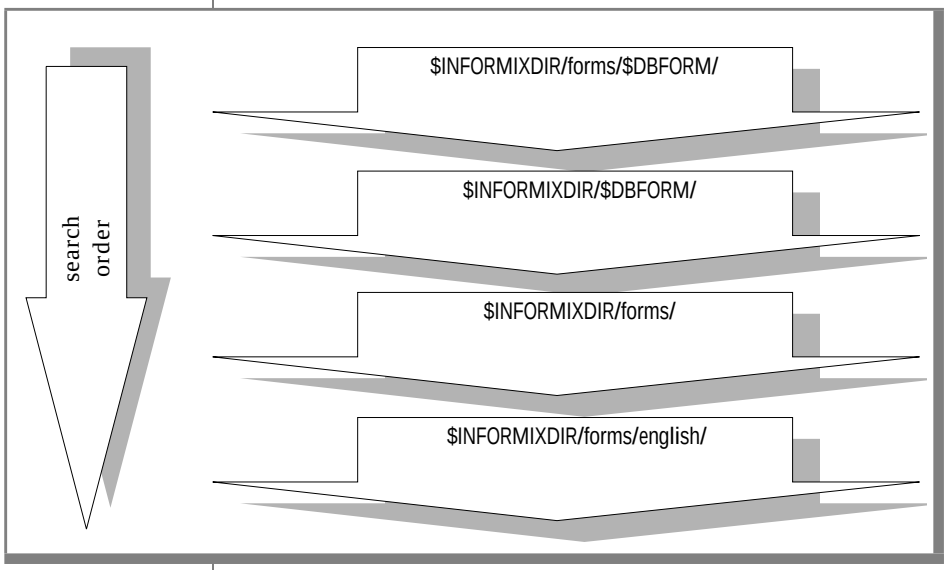


Figure D-1
Directory Search Order, Depending on $DBFORM

To specify a menu form directory

1. Use the **mkdir** command to create the appropriate subdirectory in **$INFORMIXDIR/forms**.
2. Set the owner and group of the subdirectory to **informix** and the access permission for this directory to 755.
3. Set the **DBFORM** environment variable to the new subdirectory, specifying only the subdirectory name and not the full pathname.
4. Copy the **.frm** files to the new menu form directory specified by **$INFORMIXDIR/forms/$DBFORM**.

   All files in the menu form directory should have the owner and group **informix** and access permission 644.
5. Run your program, or otherwise continue using 4GL.

For example, you can store the set of menu form files for the French language in **$INFORMIXDIR/forms/french** as follows:

```
setenv DBFORM french
```

DBFORMAT

The **DBFORMAT** environment variable specifies the format in which values are entered, displayed, or passed to the database for number data types:

DECIMAL	INTEGER	SMALLFLOAT
FLOAT	MONEY	SMALLINT

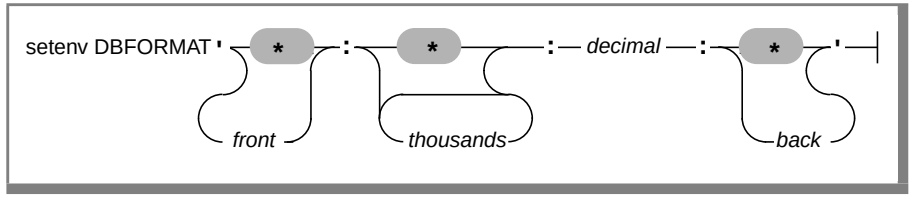
The default format specified in **DBFORMAT** affects numeric and monetary values in display, input, and output operations.

DBFORMAT can specify the leading and trailing currency symbols (but not their default positions within a monetary value) and the decimal and thousands separators. The decimal and thousands separators defined by **DBFORMAT** apply to both monetary and other numeric data.

Features of 4GL affected by the setting in **DBFORMAT** include (but are not restricted to) the following items:

- USING operator or FORMAT attribute
- DISPLAY or PRINT statement
- LET statement, where a CHAR or VARCHAR variable is assigned a monetary or number value
- LOAD and UNLOAD statements that use ASCII files (or whatever the locale regards as a *flat* file) to pass data to or from the database
- PREPARE statements that process number values

The syntax for setting **DBFORMAT** is as follows



Element	Description
back	is the trailing currency symbol.
decimal	is a characters that you specify as a valid decimal separator.
front	is the leading currency symbol.
thousands	is a characters that you specify as a valid thousands separator.

The asterisk (*) specifies that a symbol or separator is not applicable; it is the default for any *front*, *thousands*, or *back* term that you do not define.

If you specify more than one character for *decimal* or *thousands*, the values in the *decimal* or *thousands* list cannot be separated by spaces (nor by any other symbols). 4GL uses the first value specified as the thousands or decimal separator when displaying the number or currency value in output. The user can include any of the decimal or thousands separators when entering values.

Any printable character that your locale supports is valid for the thousands separator or for the decimal separator, except:

- Digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- <, >, |, ?, !, =, [,]

The same character cannot be both the thousands and decimal separator. A blank space (ASCII 32) can be the thousands separator (and is conventionally used for this purpose in some locales). The asterisk (*) symbol is valid as the decimal separator, but is not valid as the thousands separator.

In versions of 4GL prior to 6.0, the colon (**:**) was not allowed as the *thousands* separator. The colon symbol is supported in this version, but must be preceded by a backslash (\ ), as in the specification `:\:.:DM`.

The colon (**:**) symbol is supportd as a decimal separator but must be preceded by a backslash (\ ) symbol in the **DBFORMAT** specification.

You must include all three colons. Enclosing the **DBFORMAT** specification in a pair of single quotation marks is recommended to prevent the shell from attempting to interpret (or execute) any of the **DBFORMAT** characters.

The setting in **DBFORMAT** also affects how formatting masks of the FORMAT attribute and of the USING operator are interpreted. In formatting masks of FORMAT and USING, the following symbols are not literal characters but are placeholders for what **DBFORMAT** specifies:

- The dollar ($) sign is a placeholder for the *front* currency symbol.
- The comma is a placeholder for the thousands separator.
- The period is a placeholder for the decimal separator.

In formatting masks of the FORMAT attribute, the at (**@**) sign is a placeholder for the *back* currency symbol. (The @ symbol has no special significance in formatting masks for the USING operator.)

The following table illustrates the results of different combinations of **DBFORMAT** setting and format string on the same value.

Value	Format String	DBFORMAT Setting	Displayed Result
1234.56	$$#,###.##	$:,:.:	$1,234.56
1234.56	$$#,###.##	:.:,:DM	1.234,56
1234.56	#,###.##@	$:,:.:	1,234.56
1234.56	#,###.##@	:.:,:DM	1.234,56DM

When the user enters number or currency values, 4GL behaves as follows:

- It disregards any *front* (leading) or *back* (trailing) currency symbol and any thousands separators that the user enters.
- If a symbol is entered that was defined as a decimal separator in **DBFORMAT**, it is interpreted as the decimal separator.

When 4GL displays or prints values:

- The **DBFORMAT**-defined leading or trailing currency symbol is displayed for MONEY values.
- If a leading or trailing currency symbol is specified by the FORMAT attribute for non-MONEY data types, the symbol is displayed.
- The thousands separator is not displayed unless it is included in a formatting mask of the FORMAT attribute or of the USING operator.

When MONEY values are converted to character strings by the LET statement, both automatic data type conversion and explicit conversion with a USING clause insert the **DBFORMAT**-defined separators and currency symbol into the converted strings.

For example, suppose **DBFORMAT** is set as follows:

```
*:.:,:DM
```

The value `1234.56` will print or display as follows:

```
1234,56DM
```

Here DM stands for deutsche marks. Values input by the user into a screen form are expected to contain commas, not periods, as their decimal separator because **DBFORMAT** has *:.:,:DM as its setting in this example.

Restrictions on the Decimal Separator in SQL Operations

When number values appear in an SQL statement, they must use the format of the C language, with a period (.) as the decimal separator. Use of a comma as the decimal separator, which is the default for some locales, can produce errors or unexpected results.

For example, suppose that you specify ":.:,:" as the **DBFORMAT** setting, and the name of any non-integer numeric 4GL variable appears in an expression within the SELECT clause of a prepared SELECT statement.

When the query is executed, the comma (,) used as a decimal separator is treated by the database server as a separator within the list of columns (as if the digits to the left and the right of the decimal separator were the names or aliases of two different columns). Any subsequent terms of the expression after the variable will seem to have been ignored, but they are applied only to the digits to the right of the decimal point. Similarly, if the variable appears in the WHERE clause, a syntax error will result.

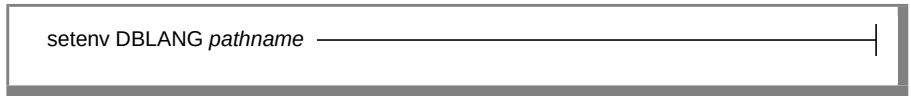
DBLANG

The **DBLANG** variable specifies the subdirectory of **$INFORMIXDIR** (or the full pathname) in which the message files for the currently active language reside. (Here **$INFORMIXDIR** stands for the name of the directory referenced by the environment variable **INFORMIXDIR**.) Message files provide a set of error messages for the database server and tools that have been translated into a national language. Compiled message files have the suffix **.iem**.

A language supplement contains the following items:

- Message files
- Instructions that specify where the files should be installed and what **DBLANG** settings to specify

The syntax for setting **DBLANG** is as follows.



Element	Description
pathname	specifies the subdirectory of **$INFORMIXDIR** or the full pathname of the directory that contains the message files.

If **DBLANG** is not set, the default directory in which message files are stored is **$INFORMIXDIR/release/en_us/0333**. For nondefault locales, some other file system must replace **/en_us**. For the directory structure of Informix GLS products, see Figure E-1 on page E-7.

Figure D-2 illustrates the search method employed for locating message files for a particular language (where value of the variable **DBLANG** is designated as **$DBLANG**).

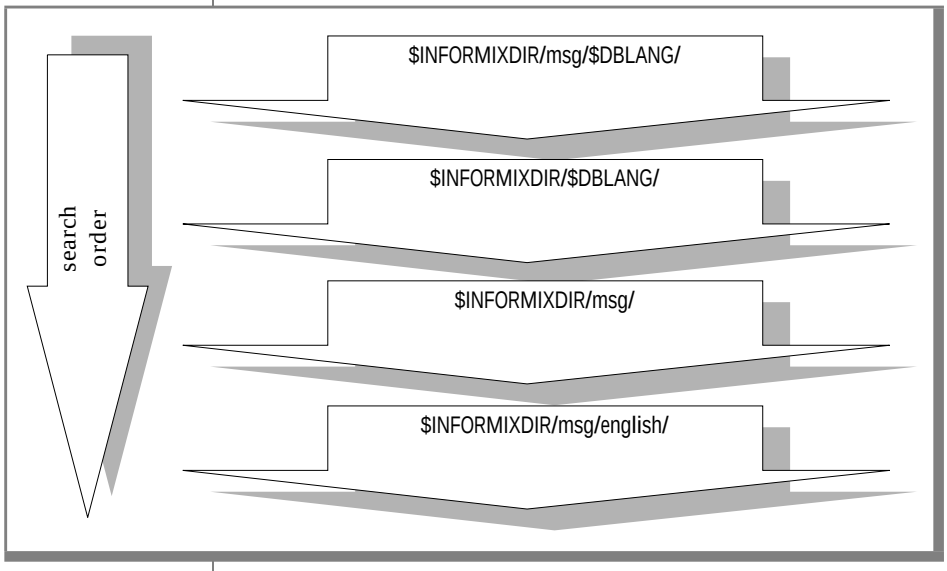


Figure D-2
Directory Search Order, Depending on $DBLANG

To specify a message directory

1. Use the **mkdir** command to create the appropriate subdirectory in **$INFORMIXDIR/msg**.
2. Set the owner and group of the subdirectory to **informix** and the access permission for this directory to 755.
3. Set the **DBLANG** environment variable to the new subdirectory, specifying only the subdirectory name and not the full pathname.
4. Copy the **.iem** files to the directory **$INFORMIXDIR/msg/$DBLANG**. All files in the new message directory should have the owner and group **informix** and should have UNIX access permission 644.

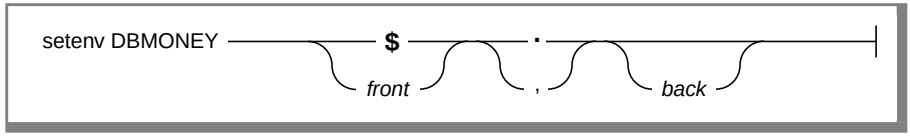
Now you can run your 4GL program or otherwise continue your work.

For example, you can store the set of message files for the French language in **$INFORMIXDIR/msg/french** as follows:

```
setenv DBLANG french
```

DBMONEY

The **DBMONEY** environment variable specifies the default display format for MONEY values and the default thousands separator and decimal separator for displaying DECIMAL values. The syntax for setting **DBMONEY** is as follows.



Element	Description
front	is a character string representing a leading currency symbol that precedes the MONEY value. This string can be up to seven characters long and can contain any character except a comma or a period.
back	is a character string representing a trailing currency symbol that follows the MONEY value. This string can be up to seven characters long and can contain any character except a comma or a period.

The term that follows *front* is a literal decimal separator symbol that separates the integral part from the fractional part of the MONEY value.

Because only its position within a **DBMONEY** setting indicates whether a symbol is the *front* or *back* currency symbol, the first two terms are required. If you use **DBMONEY** to specify a *back* symbol, for example, you must also supply a *front* currency symbol and a decimal separator (a comma or period). Similarly, if you use **DBMONEY** to change the decimal separator from a period to a comma, you must also supply a *front* symbol.

To avoid ambiguity in displayed numbers and currency values, do not use the thousands separator of **DBFORMAT** as the decimal separator of **DBMONEY**. For example, specifying comma as the **DBFORMAT** thousands separator dictates using the period as the **DBMONEY** decimal separator.

GLS

In the default (U.S. English) locale, the default setting for **DBMONEY** is:

```
$.
```

Here a dollar sign ($) is the default *front* currency symbol that precedes the MONEY value, a period (.) separates the integral part from any fractional part of the MONEY value, and no trailing currency symbol is used.

For example, in the default locale, `100.50` is formatted as `$100.50`.

In some other locales, however, comma is the default decimal separator, and the conventions of some locales require a *back* trailing currency symbol. In some East Asian locales, currency symbols can be multibyte characters.

Suppose that you want to represent MONEY values in deutsche marks, which conventionally use `DM` as the currency symbol and a comma as the decimal separator. You can set the **DBMONEY** environment variable as follows.

Shell	Command
C	`setenv DBMONEY DM,`
Bourne	`DBMONEY=DM,` `export DBMONEY`

Here, `DM` is the currency symbol preceding the MONEY value, and a comma separates the integral part from the fractional part of the MONEY value. As a result, the MONEY data value `100.50` is displayed as `DM100,50`. ♦

DBPATH

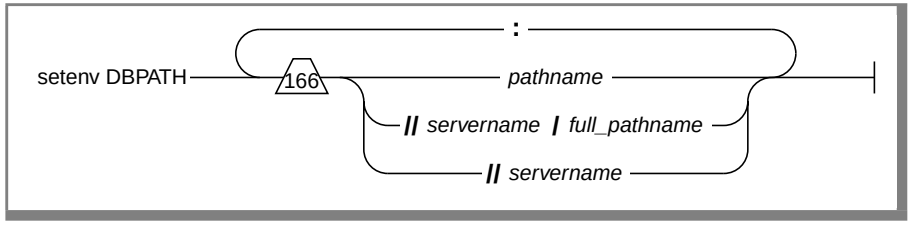
Use **DBPATH** to identify the database servers that contain databases (if you are using Dynamic Server) or the directories and database servers that contain databases (if you are using INFORMIX-SE). The **DBPATH** environment variable also specifies a list of directories (in addition to the current directory) in which 4GL looks for command scripts (**.sql** files).

The CONNECT, DATABASE, START DATABASE, and DROP DATABASE statements use **DBPATH** to locate the database under two conditions:

- If the location of a database is not explicitly stated and if the database cannot be located in the default database server
- For INFORMIX-SE, to find the default directory

The CREATE DATABASE statement does not use **DBPATH**.

You can add a new **DBPATH** entry to existing entries. To do so, use the $ format described for the UNIX environment variable **PATH**.



Element	Description
full_pathname	is a full pathname, from **root**, for a directory in which **.sql** files are stored or in which INFORMIX-SE databases are stored.
pathname	is a valid relative pathname for a directory in which **.sql** files are stored or in which INFORMIX-SE databases are stored.
servername	is the name of a Dynamic Server or INFORMIX-SE database server on which databases are stored. You cannot, however, reference database files by using a *servername* qualifier.

DBPATH can contain up to 16 entries. Each entry (*pathname*, or *servername*, or *servername* and *full_pathname*) must be less than 128 characters long. In addition, the maximum length of **DBPATH** also depends on the hardware and operating system platform on which you are setting **DBPATH**.

When you access a database using the CONNECT, DATABASE, START DATABASE, or DROP DATABASE statement, the search for the database is done first in the directory or database server specified in the statement. If no database server is specified, the default database server as set in the **INFORMIXSERVER** environment variable is used.

For INFORMIX-SE, if no directory is specified in the statement, the default directory is searched for the database. (The default directory is the current working directory if the database server is on the local computer, or your login directory if the database server is on a remote computer.) If a directory is specified but is not a full path, the directory is considered to be relative to the default directory.

If the database is not located during the initial search, and if **DBPATH** is set, the database servers or directories in **DBPATH** are searched for the indicated database. The entries to **DBPATH** are considered in order.

Searching Local Directories

Use a pathname without a database server name to have the database server search for databases or **.sql** scripts on your local computer. If you are using 4GL with INFORMIX-SE, you can search for a database and **.sql** scripts; with Dynamic Server, you can look only for **.sql** scripts.

For example, the following **DBPATH** setting causes 4GL to search for the database files in your current directory and then in Joachim's and Sonja's directories on the local computer:

```
setenv DBPATH /usr/joachim:/usr/sonja
```

As shown in the previous example, if the pathname specifies a directory name but not a database server name, the directory is sought on the computer running the default database server as specified by the **INFORMIXSERVER** environment variable. For instance, with this example, if **INFORMIXSERVER** is set to **quality**, the **DBPATH** value is interpreted as follows, where the double slash precedes the database server name:

```
setenv DBPATH //quality/usr/joachim://quality/usr/sonja
```

Searching Networked Computers for Databases

If you are using more than one database server, you can set **DBPATH** to explicitly contain the database server and directory names that you want to be searched for databases. For example, if **INFORMIXSERVER** is set to **quality** but you also want to search the **marketing** database server for **/usr/joachim**, set **DBPATH** as follows:

```
setenv DBPATH //marketing/usr/joachim:/usr/sonja
```

Specifying a Server Name

You can set **DBPATH** to contain only database server names. This action allows you to locate only databases and not locate command files.

The Dynamic Server administrator must include each database server mentioned by **DBPATH** in the **$INFORMIXDIR/etc/sqlhosts** file. For information on communication-configuration files and database server names, see your *Administrator's Guide*.

For example, if **INFORMIXSERVER** is set to **quality**, you can search for a Dynamic Server database first on the **quality** database server and then on the **marketing** database server by setting **DBPATH** as follows:

```
setenv DBPATH //marketing
```

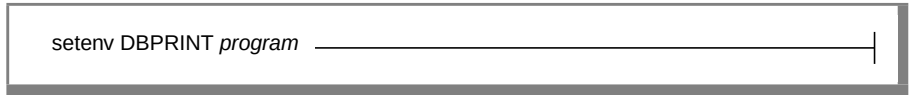
For Dynamic Server, keep the following considerations in mind:

- If you specify a local database server, the current working directory is searched for databases.
- If you specify a remote database server, the search for databases is done in the **login** directory of the user on the computer where the database server is running.

For INFORMIX-SE, you can set **DBPATH** to contain just the database server names (and no directory names) to locate only databases and not command scripts.

DBPRINT

The **DBPRINT** environment variable specifies the print program that you want to use in producing output from your 4GL program to a list device.



Element	Description
program	is the name of a command, shell script, or UNIX utility that supports standard ASCII input.

The default program is as follows:

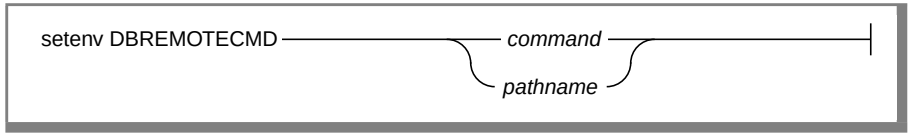
- For most BSD UNIX systems, the default program is usually **lpr**.
- For UNIX System V, the default program is usually **lp**.

Set the **DBPRINT** environment variable as follows to specify the **myprint** print program.

Shell	Command
C	`setenv DBPRINT myprint`
Bourne or Korn	`DBPRINT=myprint` `export DBPRINT`

DBREMOTECMD

You can set the **DBREMOTECMD** environment variable to override the default remote shell used when you perform remote tape operations with Dynamic Server. Set it by using either a simple command or the full pathname. If you use the full pathname, the database server searches your **PATH** for the specified command.



Element	Description
command	is the command to override the default remote shell.
pathname	is the pathname to override the default remote shell.

Informix highly recommends using the full pathname syntax on the interactive UNIX platform to avoid problems with like-named programs in other directories and possible confusion with the *restricted shell* (**/usr/bin/rsh**).

Set the **DBREMOTECMD** environment variable as follows for a simple command name.

Shell	Command
C	`setenv DBREMOTECMD rcmd`
Bourne	`DBREMOTECMD=rcmd` `export DBREMOTECMD`

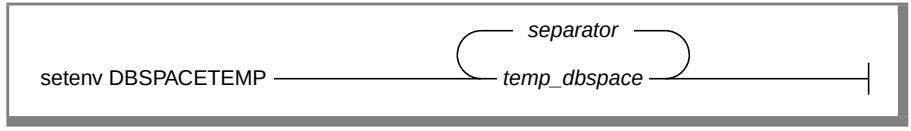
Set the **DBREMOTECMD** environment variable as follows to specify the full pathname.

Shell	Command
C	`setenv DBREMOTECMD /usr/bin/remsh`
Bourne	`DBREMOTECMD=/usr/bin/remsh` `export DBREMOTECMD`

For more information on **DBREMOTECMD**, see the *Archive and Backup Guide*. It discusses using remote tape devices for archives, restores, and logical log backups.

DBSPACETEMP

You can set the **DBSPACETEMP** environment variable to specify the dbspace to be used for building all temporary tables and for holding temporary files used for sorting. This arrangement spreads temporary space across any number of disk drives.



Element	Description
separator	can be either colon or a comma.
temp-dbspace	is a list of valid existing temporary dbspaces.

You can set the **DBSPACETEMP** environment variable to override the default dbspaces used for temporary tables and sorting space specified in the **DBSPACETEMP** configuration parameter in the configuration file. For example, you might set **DBSPACETEMP** as follows.

Shell	Command
C	`setenv DBSPACETEMP sorttmp1: sorttmp2: sorttmp3`
Bourne	`DBSPACETEMP=sorttmp1:sorttmp2:sorttmp3` `export DBSPACETEMP`

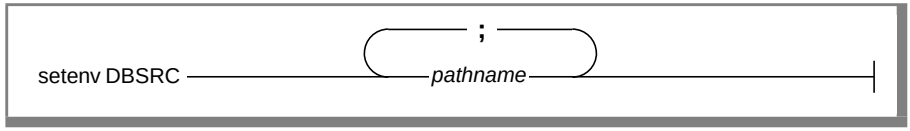
Separate the dbspace entries with either colons or commas. The number of dbspaces is limited by the maximum size of the environment variable, as defined by the UNIX shell. The default, if left unspecified, is the root dbspace. The database server does not create the specified dbspace if it does not exist.

For the creation of temporary tables, if neither **DBSPACETEMP** nor the **DBSPACETEMP** parameter in the **onconfig** file is set, the database server creates the temporary tables in the dbspace where the database was created. For sorting, the database server uses the following disk space for writing temporary information, in the following order of precedence:

1. The operating system directory (or directories) that are specified by **PSORT_DBTEMP**, if this environment variable is set
2. The dbspace (or dbspaces) specified by **DBSPACETEMP**, if it is set
3. The dbspace (or dbspaces) specified by the **onconfig** parameter **DBSPACETEMP**
4. The operating system file space in **/tmp**

DBSRC

The **DBSRC** environment variable specifies directories for **.4gl** source files.



Element	Description
pathname	is the full pathname of a directory for **.4gl** source files.

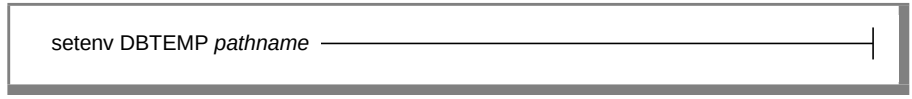
The directories specified by **DBSRC** are part of the search path only during debugging session. (The Interactive Debugger must be purchased separately from 4GL.) You are not required to specify **DBSRC.**

For example, the Debugger will search for files in the **programs** and **krystl** directories (outside its default search path) if you set **DBSRC** as follows.

Shell	Command
C	`setenv DBSRC /b/junet/programs:/b/krystl`
Bourne	`DBSRC=/b/junet/programs:/b/krystl` `export DBSRC`

DBTEMP

The **DBTEMP** environment variable specifies the full pathname of the directory into which you want INFORMIX-SE to place its temporary files. You need not set **DBTEMP** if the default value, **/tmp**, is satisfactory.



Element	Description
pathname	is the full pathname of the directory for temporary files.

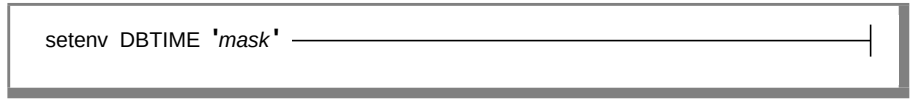
Set the **DBTEMP** environment variable as follows to specify the pathname **usr/magda/mytemp**.

Shell	Command
C	`setenv DBTEMP usr/magda/mytemp`
Bourne	`DBTEMP=usr/magda/mytemp` `export DBTEMP`

For the creation of temporary tables, if **DBTEMP** is not set, the temporary tables are created in the directory of the database (that is, the **.dbs** directory).

DBTIME

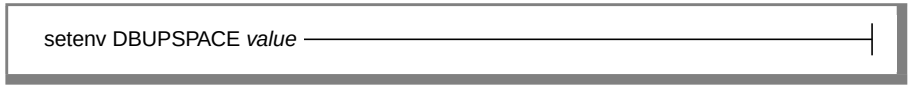
The **DBTIME** environment variable has no effect on 4GL programs. (For some Informix products, **DBTIME** can set a default format for DATETIME values.)



Element	Description
mask	is a quoted string, specifying a formatting mask.

DBUPSPACE

The **DBUPSPACE** environment variable lets you specify and thus constrain the amount of system disk space that the UPDATE STATISTICS statement can use when trying to simultaneously construct multiple column distributions.



Element	Description
value	represents a disk space amount in kilobytes.

For example, if **DBUPSPACE** is set to 2500 (kilobytes) by the following command, no more than 2.5 megabytes of disk space is to be used to accomplish sorting during the execution of an UPDATE STATISTICS statement.

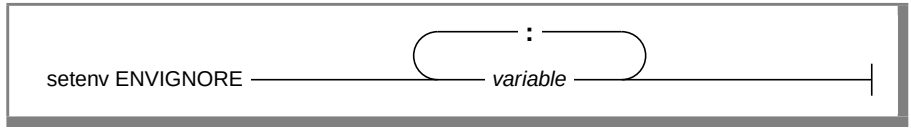
Shell	Command
C	`setenv DBUPSPACE 2500`
Bourne	`DBUPSPACE=2500` `export DBUPSPACE`

If a table requires 5 megabytes of disk space for sorting, UPDATE STATISTICS accomplishes the task in two passes; the distributions for one half of the columns are constructed with each pass.

If you try to set **DBUPSPACE** to any value less than 1024 kilobytes, it is automatically set to 1024 kilobytes, but no error message is returned. If this value is not large enough to allow more than one distribution to be constructed at a time, at least one distribution is done, even if the amount of disk space required for the one is greater than specified in **DBUPSPACE**.

ENVIGNORE

Use the **ENVIGNORE** environment variable to deactivate specified environment variable entries in the common (shared) and private environment-configuration files.



Element	Description
variable	is the list of environment variables that you want to deactivate.

For example, to ignore the **DBPATH** and **DBMONEY** entries in the environment-configuration files, specify the following command.

Shell	Command
C	`setenv ENVIGNORE DBPATH:DBMONEY`
Bourne	`ENVIGNORE=DBPATH:DBMONEY` `export ENVIGNORE`

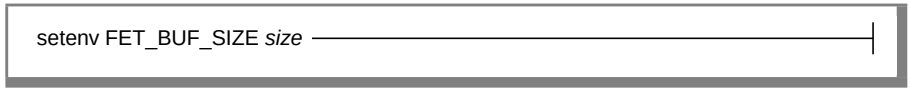
Directory **$INFORMIXDIR/etc/informix.rc** stores the common environment-configuration file. The private environment-configuration file is stored in the user's home directory as **.informix**. See "Where to Set Environment Variables" on page D-2 for more information on creating or modifying an environment-configuration file.

ENVIGNORE itself cannot be set in an environment-configuration file.

FET_BUF_SIZE

The *fetch buffer* is the buffer that the database server uses to send data (except for BYTE or TEXT data) to client applications. The buffer resides in the client process.

You can set the size of the fetch buffer by setting the **FET_BUF_SIZE** environment variable to the desired value prior to runtime. This action sets the fetch buffer size for the duration of running the 4GL application.



Element	Description
size	is a literal integer that specifies the size of the fetch buffer (in bytes).

The bigger the buffer, the more data the database server can send to the application before returning control to the application. The greater the size of this buffer, the more rows can be returned, and the less often the application must wait while the database server retrieves rows. A large buffer can improve performance by reducing the overhead of refilling the client-side buffer.

For example, the following command sets this environment variable to 20,000 bytes (20 kilobytes).

Shell	Command
C	`setenv FET_BUF_SIZE 20000`
Bourne	`FET_BUF_SIZE=20000` `export FET_BUF_SIZE`

Specify the *size* value in bytes, up to a maximum value for a SMALLINT (or of a C language **short**) on your system. For most 32-bit systems, this value is 32,767 bytes. If the parameter is not set externally, a default is used. (As in most environment settings, a thousands separator is not valid in *size*.)

FGLPCFLAGS

The environment variable **FGLPCFLAGS** allows you to set certain compiler options as defaults for 4GL programs that are compiled to p-code. These options simplify many compilations. Typical values are these:

-ansi Issues warnings for Informix extensions to SQL syntax.

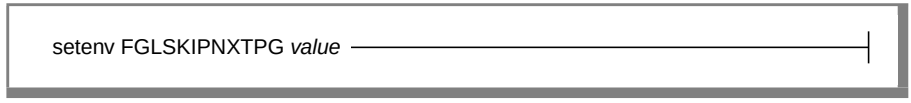
-anyerr Resets **status** when 4GL expressions are evaluated.

-z Supports functions with a variable number of arguments.

Informix recommends compiling all 4GL programs with **-anyerr** specified.

FGLSKIPNXTPG

The SKIP TO TOP OF PAGE command has no effect when a report is already at the top of the page (that is, when no data value has yet been sent to a new page). Programs that rely on the older behavior of SKIP TO TOP OF PAGE can override this effect if the environment variable **FGLSKIPNXTPG** is set at runtime to any non-blank value, using the following syntax.



Element	Description
value	is any nonblank character.

If you set **FGLSKIPNXTPG** to any value at report execution time, SKIP TO TOP OF PAGE forces a page change even if no data value has yet been printed in the body of the current page. The following examples set it to `1`.

C shell	Bourne or Korn shell
`setenv FGLSKIPNXTPG 1`	`FGLSKIPNXTPG=1` `export FGLSKIPNXTPG`

INFORMIXC

The **c4gl** command uses the **INFORMIXC** and **CC** environment variables (defaulting to **cc** on most computers) in the final stage of compilation. Setting one of these environment variables lets you substitute any C compiler. Because **CC** is acknowledged by many versions of **make**, this environment variable is compatible with other programs also. Use the following Bourne shell code to determine the compiler:

```
${INFORMIXC:=${CC:-cc}}
```

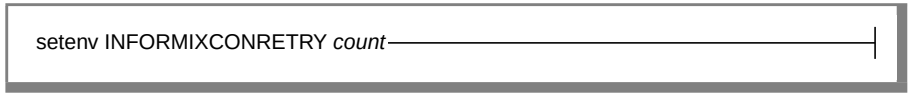
For the compiler, 4GL uses an **INFORMIXC** value that is not empty. If no non-empty **INFORMIXC** value is specified, 4GL uses the (non-empty) value of the **CC** environment variable. If neither of those exist, the compiler defaults to **cc**.

Important: *For users of **gcc**, Informix assumes that strings are writable, so you need to compile using the **-fwritable-strings** option. Failure to do so can produce unpredictable results, possibly including core dumps.*

INFORMIXCONRETRY

The **INFORMIXCONRETRY** environment variable enables you to specify the maximum number of *additional* connection attempts that should be made to each database server by the client. For example, if you set this environment variable to 4, no more than 5 attempts to connect will be made during the time limit specified by the **INFORMIXCONTIME** environment variable.

Set the **INFORMIXCONRETRY** environment variable as follows.



Element	Description
count	is the number of additional attempts to connect to each database server.

For example, you can set **INFORMIXCONRETRY** to three additional connection attempts (after the initial attempt) as follows.

Shell	Command
Bourne	`setenv INFORMIXCONRETRY 3`
C	`INFORMIXCONRETRY=3` `export INFORMIXCONRETRY`

The default value for **INFORMIXCONRETRY** is one retry after the initial connection attempt.

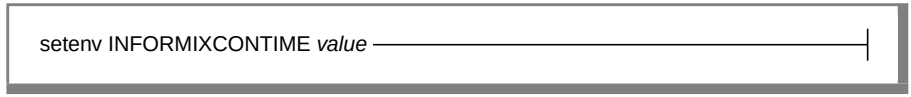
The **INFORMIXCONTIME** setting, described in the next section, takes precedence over the **INFORMIXCONRETRY** setting. That is, if time expires after the third retry, no further attempts are made, even if **INFORMIXCONRETRY** was set to `10`.

INFORMIXCONTIME

The **INFORMIXCONTIME** environment variable enables you to specify the minimum time limit, in seconds, for the SQL statement CONNECT to attempt to connect to a database server before it returns an error.

You might encounter connection difficulties related to system or network load problems. For instance, if the database server is busy establishing new SQL client threads, some of the clients might fail because the database server cannot issue a network function call fast enough. The **INFORMIXCONTIME** and **INFORMIXCONRETRY** environment variables let you configure your client-side connection capability to retry to connect instead of returning an error.

Set the **INFORMIXCONTIME** environment variable as follows.



Element	Description
value	represents the minimum number of seconds spent in attempts to connect to each database server.

For example, set **INFORMIXCONTIME** to 60 seconds as follows.

Shell	Command
C	`setenv INFORMIXCONTIME 60`
Bourne	`INFORMIXCONTIME=60` `export INFORMIXCONTIME`

If **INFORMIXCONTIME** is set to `60` and **INFORMIXCONRETRY** is set to `3`, as shown in this appendix, attempts to connect to the database server (after the initial attempt at 0 seconds) will be made at 20, 40, and 60 seconds, if necessary, before aborting. This 20-second interval is the result of **INFORMIXCONTIME** divided by **INFORMIXCONRETRY**.

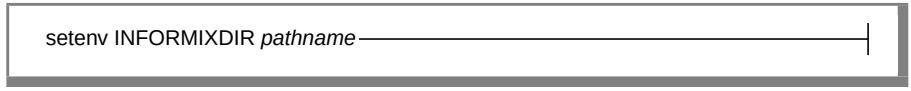
If execution of the CONNECT statement involves searching **DBPATH**, the following rules apply:

- All appropriate servers in the **DBPATH** setting are accessed at least once even though the **INFORMIXCONTIME** value might be exceeded. Thus, the CONNECT statement might take longer than the **INFORMIXCONTIME** time limit to return an error indicating connection failure or that the database was not found.
- **INFORMIXCONRETRY** specifies how many additional attempted connections should be made for each database server entry in **DBPATH**.
- The **INFORMIXCONTIME** value is initially divided among the number of database server entries specified in **DBPATH**. Thus, if **DBPATH** contains numerous database servers, increase the **INFORMIXCONTIME** value accordingly to allow for multiple connection attempts.

The default value for **INFORMIXCONTIME** is 15 seconds after the initial connection attempt. The **INFORMIXCONTIME** setting takes precedence over the **INFORMIXCONRETRY** setting; retry efforts could end after the **INFORMIXCONTIME** value has been exceeded, but before the **INFORMIXCONRETRY** value has been reached.

INFORMIXDIR

The **INFORMIXDIR** environment variable specifies the directory that contains the subdirectories in which your product files are installed. **INFORMIXDIR** must be set. If you have multiple versions of a database server, set **INFORMIXDIR** to the appropriate directory name for the version that you want to access. For more information about when to set the **INFORMIXDIR** environment variable, see the *Installation Guide* for your database server.



Element	Description
pathname	is the directory path where the product files are installed.

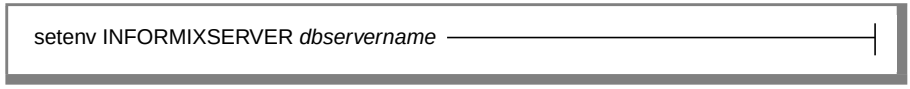
Do not set *pathname* to a path whose character-string value requires more than 64 bytes of storage.

The following examples set the **INFORMIXDIR** environment variable to the recommended installation directory.

Shell	Command
C	`setenv INFORMIXDIR /usr/informix`
Bourne	`INFORMIXDIR=/usr/informix` `export INFORMIXDIR`

INFORMIXSERVER

The **INFORMIXSERVER** environment variable specifies the default database server to which an explicit or implicit connection is made by 4GL. The database server can be either local or remote.



Element	Description
dbservername	is the name of the default database server.

The value of **INFORMIXSERVER** must correspond to a valid *dbservername* entry in the **$INFORMIXDIR/etc/sqlhosts** file on the computer running the application. It must be specified using lowercase characters and cannot exceed 18 characters for Dynamic Server and cannot exceed 10 characters for INFORMIX-SE. For example, to specify the **coral** database server as the default for connection, enter the following command.

Shell	Command
C	`setenv INFORMIXSERVER coral`
Bourne	`INFORMIXSERVER=coral` `export INFORMIXSERVER`

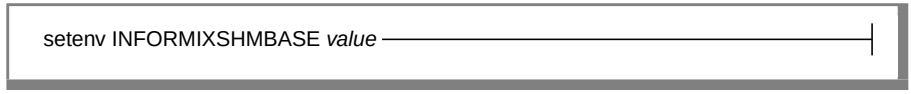
INFORMIXSERVER specifies the database server to which an application connects if the CONNECT DEFAULT statement is executed. It also defines the database server to which an initial implicit connection is established if the first statement in an application is not a CONNECT statement.

Important: ***INFORMIXSERVER*** *must be set even if the application or 4GL does not use implicit or explicit default connections.*

INFORMIXSHMBASE

The **INFORMIXSHMBASE** environment variable affects only client applications connected to the database server using the IPC shared-memory (**ipcshm**) communication protocol.

Use **INFORMIXSHMBASE** to specify where shared-memory communication segments are attached to the client process so that client applications can avoid collisions with other memory segments used by the application. If you do not set **INFORMIXSHMBASE**, the memory address of the communication segments defaults to an implementation-specific value such as `0x800000`.



Element	Description
value	is used to calculate the memory address.

The database server calculates the memory address where segments are attached by multiplying the value of **INFORMIXSHMBASE** by 1024. For example, to set the memory address to the value `0x800000`, set the environment variable **INFORMIXSHMBASE** as follows.

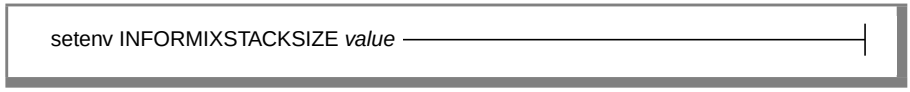
Shell	Command
C	`setenv INFORMIXSHMBASE 8192`
Bourne	`INFORMIXSHMBASE=8192` `export INFORMIXSHMBASE`

Resetting **INFORMIXSHMBASE** requires a thorough understanding of how the application uses memory. Usually you do not reset **INFORMIXSHMBASE**. For more information, see your *Administrator's Guide*.

INFORMIXSTACKSIZE

The **INFORMIXSTACKSIZE** environment variable affects only client applications connected to Dynamic Server.

The database administrator can set **INFORMIXSTACKSIZE** to specify the stack size (in kilobytes) that the database server will use for a particular client session. Use **INFORMIXSTACKSIZE** to override the value of the **onconfig** configuration parameter **STACKSIZE** for a particular application or user.



Element	Description
value	is the stack size for SQL client threads in kilobytes.

For example, to decrease the **INFORMIXSTACKSIZE** to 20 kilobytes, enter the following command.

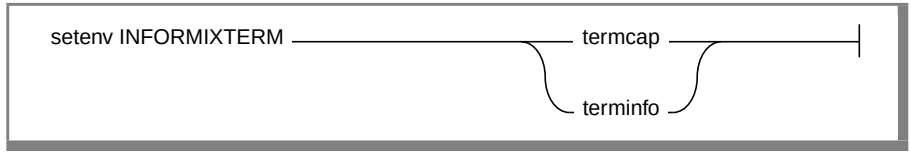
Shell	Command
C	`setenv INFORMIXSTACKSIZE 20`
Bourne	`INFORMIXSTACKSIZE=20` `exportINFORMIXSTACKSIZE`

Important: *If **INFORMIXSTACKSIZE** is not set, the stack size is taken from the database server configuration parameter **STACKSIZE**, or it defaults to an implementation-specific value. The default stack size value for the primary thread for an SQL client is 32 kilobytes for nonrecursive database activity.*

Warning: *For specific instructions for setting this value, see the "Administrator's Guide" for your database server. If you incorrectly set the value of **INFORMIXSTACKSIZE**, it can cause the database server to fail.*

INFORMIXTERM

On platforms that support **terminfo** capability, the **INFORMIXTERM** variable specifies whether 4GL should use the information in the **termcap** file or the **terminfo** directory. **INFORMIXTERM** determines terminal-dependent keyboard and screen capabilities such as the operation of function keys, color and intensity attributes in screen displays, and the definition of window border and graphics characters.



If **INFORMIXTERM** is not set, the default setting is **termcap**. When 4GL is installed on your system, a **termcap** file is placed in the **etc** subdirectory of **$INFORMIXDIR**. This file is a superset of an operating system **termcap** file.

To use **terminfo**, your system must fully support the UNIX System V **terminfo** library interface. Not all UNIX platforms provide proper **terminfo** support. To determine whether your platform supports **terminfo** to Informix requirements (and therefore, whether setting **INFORMIXDIR** will have an effect), use the following command:

```
nm $INFORMIXDIR/bin/upscol | grep tigetstr
```

If the **nm** command yields no output, your platform does not have **terminfo** support, and setting **INFORMIXTERM** has no effect.

If the command produces a reference line for the **tigetstr** function, your platform supports **terminfo** and you can use the **INFORMIXTERM** environment variable to select **terminfo** functionality. Here is an example of a reference line:

```
[3810]  |   1281040|      80|FUNC |GLOB |0    |9      |tigetstr
```

You can use the **termcap** file supplied by Informix, the system **termcap** file, or a **termcap** file that you created yourself. You must set the **TERMCAP** environment variable if you do not use the default **termcap** file.

The **terminfo** directory contains a file for each terminal name that has been defined. It is supported only on computers that provide full support for the UNIX System V **terminfo** library. For details, see the machine notes for your computer. (See also Appendix F, “Modifying termcap and terminfo.”)

You can set the **INFORMIXTERM** environment variable to **termcap** as follows.

Shell	Command
C	`setenv INFORMIXTERM termcap`
Bourne	`INFORMIXTERM=termcap` `export INFORMIXTERM`

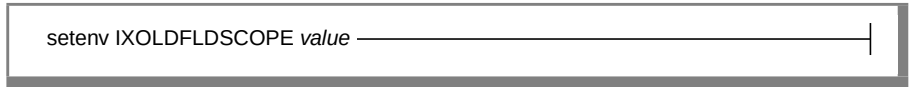
Alternatively, you can set **INFORMIXTERM** to **terminfo** as follows.

Shell	Command
C	`setenv INFORMIXTERM terminfo`
Bourne	`INFORMIXTERM=terminfo` `export INFORMIXTERM`

If **INFORMIXTERM** is set to **termcap**, you must set the UNIX environment variable **TERMCAP**. Similarly, if **INFORMIXTERM** is set to **terminfo**, you must set the UNIX environment variable **TERMINFO**.

IXOLDFLDSCOPE

The **IXOLDFLDSCOPE** environment variable is recognized by Version 6.04 and later of 4GL. It allows programs that were written for Version 6.00 and earlier that have incorrect field qualifiers to avoid the field qualifier checking mechanism that was introduced with Version 6.01. To accomplish this task, you can use the following syntax for setting **IXOLDFLDSCOPE**.



Element	Description
value	is any non-blank character.

Prior to Version 6.01, the following syntax ran successfully even if the *qualifier* name did not match the name of the field qualifier in the form (*table name, screen record name*) or the literal FORMONLY:

```
{BEFORE|AFTER} FIELD qualifier.fieldname
```

Starting with Version 6.01, the qualifiers for field names and the qualifiers on the form must match. As a result, some 4GL programs that ran prior to Version 6.01 now terminate with error -1129 at runtime because the qualifier names, although stated explicitly, do not match.

For example, the following fragment of a form specification declares a screen record called **f_scrn_rec**:

```
ATTRIBUTES
a = FORMONLY.myfield1;
...
INSTRUCTIONS
SCREEN RECORD f_scrn_rec (FORMONLY.myfield1 THRU
FORMONLY.myfieldn)
```

Suppose that a 4GL program referenced the same **f_scrn_rec** screen record in the following source code:

```
DEFINE progrec RECORD ...    END RECORD
...
INPUT progrec.* WITHOUT DEFAULTS FROM f_scrn_rec.*
...
   AFTER FIELD FORMONLY.myfield1
      LET mykeyval = fgl_lastkey()
END INPUT
```

In the 4GL example, the INPUT statement uses the **f_scrn_rec** screen record name as the qualifier, but the AFTER FIELD clause references the inherent qualifier, FORMONLY. (If the fields were based on database columns, a *table* identifier would replace FORMONLY.) The field-matching code for CONSTRUCT and INPUT considers the proper qualifier to be the one given in the FROM clause. A mismatch results, and a runtime error is issued:

```
-1129, Field in BEFORE/AFTER clause not found in form,
```

If you encounter this error, correct these references. If doing so presents a significant barrier, however, to upgrading a 4GL program earlier than Version 6.01, you can use **IXOLDFLDSCOPE** as a backward-compatible mechanism to disable the qualifier test.

To activate this backward-compatible (no checking) mode, you must set and export the variable **IXOLDFLDSCOPE** in your environment at runtime (*not* at compile time). You can set **IXOLDFLDSCOPE** to any nonblank value, as in the following example, where the value is set to Y.

C shell	Bourne or Korn shell
`setenv IXOLDFLDSCOPE Y`	`IXOLDFLDSCOPE= Y` `export IXOLDFLDSCOPE`

Including **IXOLDFLDSCOPE** in the client environment disables checking field qualifiers. When it is set, however, you lose the benefits of (for example) being able to use fields from multiple records in an INPUT statement even if a given field name occurs in more than one participating record.

LINES

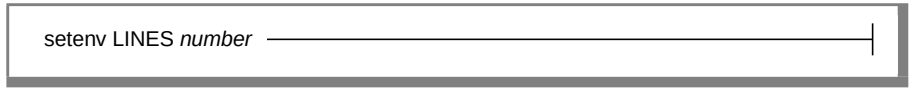
UNIX platforms support various ways to control the sizes of screens and windows in terms of lines (or rows) and columns. Depending on the method that your platform uses, two environment variables, **LINES** and **COLUMNS**, might be useful in controlling the character dimensions of your screen.

One common way to control the character dimensions of screens is the use of input/output control (**ioctl()**) calls. To see if your platform uses this method, enter the command `stty -a`. If the system response includes explicit values for rows and columns, **ioctl()** control is in effect. The following example illustrates this command:

```
% stty -a
speed 9600 baud;
rows = 24; columns = 80;
intr = ^c; quit = ^|; erase = ^h; kill = ^u;
```

If your platform uses **ioctl()** calls to control screen dimensions, the operating system or windowing facility probably provides a way to resize the screen using a pointing device, such as the mouse or trackball.

If your platform does not use **ioctl()** calls to control screen dimensions, you can use the **LINES** and **COLUMNS** environment variables to specify the screen dimensions. Use the following syntax for setting **LINES**.



Element	Description
number	is a literal integer, specifying the vertical height of the screen as a number of lines (sometimes called rows) in the screen display.

On such platforms, if **LINES** or **COLUMNS** is not set, the corresponding value is taken from the rows or columns field in the **terminfo** or **termcap** entry in use, as indicated by the **TERM** environment variable.

The following example sets **LINES** to 24 and **COLUMNS** to 80.

C shell	Bourne or Korn shell

`setenv COLUMNS 80`
`setenv LINES 24`
	`COLUMNS=80`
	`LINES=24`
	`export COLUMNS LINES`

If either **LINES** or **COLUMNS** is set to an invalid value (that is, to a value that is not a positive integer), the invalid value is ignored, and the required value is read from the **termcap** or **terminfo** entry as applicable.

ONCONFIG

The database administrator can set the **ONCONFIG** environment variable (previously known as **TBCONFIG**), which contains the name of the UNIX file that holds the configuration parameters for the database server. This file is read as input to either the disk-space or shared-memory initialization procedure.

Element	Description
filename	is the name of the file in **$INFORMIXDIR/etc** that contains the database server configuration parameters.

If you are not the database administrator, you need to set **ONCONFIG** only if more than one database server is initialized in your **$INFORMIXDIR** directory, and you want to maintain multiple configuration files with different values. If **ONCONFIG** is not set, the default is **onconfig**.

Each database server has its own **onconfig** file that must be stored in the **$INFORMIXDIR/etc** directory. You might prefer to name **onconfig** so that it can easily be related to a specific database server. For example, when the desired filename is **onconfig3**, you can set the **ONCONFIG** environment variable as follows.

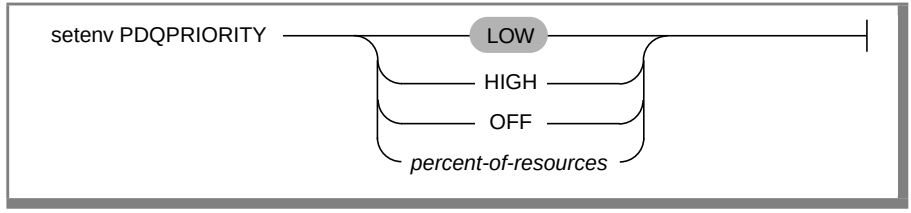
Shell	Command
C	`setenv ONCONFIG onconfig3`
Bourne	`ONCONFIG=onconfig3` `export ONCONFIG`

For more information, see the *Administrator's Guide* for your database server.

PDQPRIORITY

The **PDQPRIORITY** environment variable determines the amount of resources that the database server allocates to process a query in parallel.

The following diagram shows the syntax for setting this variable.



Element	Description
percentage-of-resources	is an integer between 0 and 100, indicating a query priority level.

The keyword settings of **PDQPRIORITY** have the following meanings:

- **HIGH**: The database server determines the operators and respective degree of parallelism.
- **LOW**: Data values are fetched from fragmented tables in parallel.
- **OFF**: Parallel data query is not enabled.

For example, if the desired setting is 25 percent, you can set the **PDQPRIORITY** environment variable as follows.

Shell	Command
C	`setenv PDQPRIORITY 25`
Bourne or Korn	`PDQPRIORITY=25` `export PDQPRIORITY`

For more information, see the *Informix Guide to SQL: Reference*.

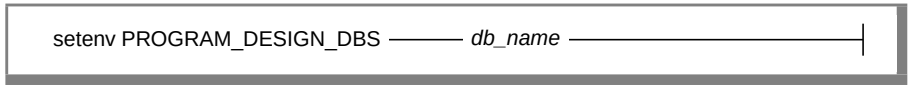
PROGRAM_DESIGN_DBS

The 4GL and RDS Programmer's Environments (**i4gl** and **r4gl**) use a relational database to store the names of the objects that are used to create programs and their build dependencies. This database is an Informix database.

The default name for this database is syspgm4gl.

In Versions 6.01 and later, both Programmer's Environments let the user declare another name for the Programmer's Environment database. This capability is particularly useful when you are using Dynamic Server because only one database can be named **syspgm4gl** at any given time.

To specify a different name for the Programmer's Environment database, set the **PROGRAM_DESIGN_DBS** environment variable to the desired name.



Element	Description
db_name	is the nondefault name that you declare here for the Programmer's Environment database.

Important: *The dbname value must obey the rules for database names for your database server. That is, the name can contain only digits, letters, and underscores, and it can be no more than 10 bytes long (or 18 bytes for Dynamic Server).*

For example, to use the name **jane_syspg** for your Programmer's Environment database, set **PROGRAM_DESIGN_DBS** to **jane_syspg** in your client environment before you run **i4gl** or **r4gl**.

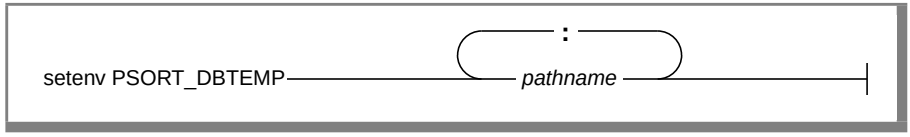
Shell	Command
C	`setenv PROGRAM_DESIGN_DBS jane_syspg`
Bourne or Korn	`PROGRAM_DESIGN_DBS=jane_syspg` `export PROGRAM_DESIGN_DBS`

PSORT_DBTEMP

The **PSORT_DBTEMP** environment variable affects only client applications connected to the database server.

PSORT_DBTEMP specifies a directory or directories where the Informix database server writes the temporary files that it uses when sorting data. For more information on other places where the database server can write information during a sort, see "DBSPACETEMP" on page D-37.

This variable is used even if the variable **PSORT_NPROCS** is not set.



Element	Description
pathname	is the name of the UNIX directory used for intermediate writes during a sort.

Set the **PSORT_DBTEMP** environment variable as follows to specify the directory; for example, **/usr/leif/tempsort**.

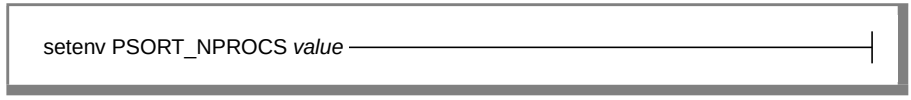
Shell	Command
C	`setenv PSORT_DBTEMP /usr/leif/tempsort`
Bourne	`PSORT_DBTEMP=/usr/leif/tempsort` `export PSORT_DBTEMP`

For maximum performance, specify directories that reside in file systems on different disks.

PSORT_NPROCS

The **PSORT_NPROCS** environment variable affects only client applications connected to the database server.

PSORT_NPROCS enables the **Psort** parallel-process sorting package to improve performance. The setting defines the upper limit for the number of threads used to sort a query.



Element	Description
value	specifies the maximum number of threads to be used to sort a query.

Set the **PSORT_NPROCS** environment variable as follows to specify the maximum value.

Shell	Command
C	`setenv PSORT_NPROCS 4`
Bourne	`PSORT_NPROCS=4` `export PSORT_NPROCS`

To maximize the effectiveness of **Psort**, set **PSORT_NPROCS** to the number of available processors in the hardware. If **PSORT_NPROCS** is set to zero, **Psort** uses three (3) as the default number of threads.

Use the following command to disable **Psort**.

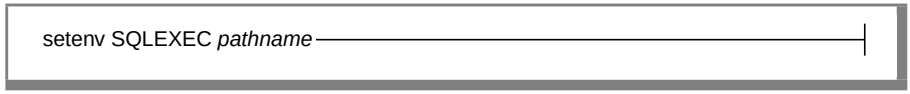
Shell	Command
C	`unsetenv PSORT_NPROCS`
Bourne	`unset PSORT_NPROCS`

For additional information about the **PSORT_NPROCS** environment variable, see your *Administrator's Guide*.

SQLEXEC

This variable is not used in versions of 4GL later than 6.x.

The **SQLEXEC** environment variable specifies the location of the Version 6.0 relay module executable that allows a Version 4.1 or earlier client to communicate indirectly with a local Version 6.0 Informix database server. You must, therefore, set **SQLEXEC** only to establish communications between a Version 4.1 or earlier client and a Version 6.0 database server.



Element	Description
pathname	specifies the pathname for the relay module.

Set **SQLEXEC** as follows to specify the full pathname of the relay module, which is in the **lib** subdirectory of your **$INFORMIXDIR** directory.

Shell	Command
C	`setenv SQLEXEC $INFORMIXDIR/lib/sqlrm`
Bourne	`SQLEXEC=$INFORMIXDIR/lib/sqlrm` `export SQLEXEC`

If you set the **SQLEXEC** environment variable on the C shell command line instead of in your **.login** or **.cshrc** file, you must include braces ({ })symbols around the existing **INFORMIXDIR**, as follows.

Shell	Command
C	`setenv SQLEXEC ${INFORMIXDIR}/lib/sqlrm`

For information on the relay module, see the 6.0 version for the *Administrator's Guide* for your database server.

If you were previously using a 6.0 or later database server with 4GL 4.1, you were using either INFORMIX-NET or the INFORMIX Relay Module (**sqlrm**) to provide a compatible interface to the newer database servers.

The **SQLEXEC** environment variable was set to a full or relative pathname ending in **sqlexec** (if using INFORMIX-NET) or **sqlrm** (if using the relay module), and the **INFORMIXSERVER** variable was set to the desired database server name. Additionally, if you were using the database server locally, the **ONCONFIG** environment variable might have been set to the name of the configuration file that belongs to the desired database server.

After you install the new version of 4GL with your database servers, you need to change your environment variable settings as follows:

- The **SQLEXEC** environment variable no longer applies. Unset it in your environments (including any login and shell configuration scripts).
- The **INFORMIXSERVER** (and **ONCONFIG**, if applicable) environment variable does not change as a direct result of the 4GL upgrade. For more about **INFORMIXSERVER** and **ONCONFIG**, see the *Informix Guide to SQL: Reference*.

Important: *The specified database server must be on-line and accepting connections for 4GL to create or access a databases.*

SUPOUTPIPEMSG

In a 4GL report, you can direct the output to a pipe either by appending TO PIPE to the START REPORT statement or by including the REPORT TO PIPE clause in the OUTPUT section of the report definition. A program that further processes the output, such as a shell script or printer routine, usually receives the output of the report through the pipe.

In 4GL Versions 6.02 and earlier, if the program at the end of the pipe terminated while receiving data from the pipe, the 4GL application would exit silently without any indication of a problem.

In 4GL Versions 6.03 and higher, when the receiving program at the reading end of the pipe dies, the next PRINT or SKIP statement will, by default, result in error -1324, `A report output file cannot be written to`. This situation cannot be handled by the 4GL program by using the WHENEVER ERROR statement.

If you prefer the old behavior (where the 4GL program is silently terminated), you can retain that behavior using the **SUPOUTPIPEMSG** environment variable. Setting it to any value or to no value, as follows, is sufficient.

If **SUPOUTPIPEMSG** is set in the user environment at program execution time and a report output pipe dies prematurely, the 4GL program terminates without any error message.

SQLRM

This variable is not used in versions of 4GL later than 6.x.

If the system administrator is configuring a client/server environment in which a Version 4.1 4GL client accesses a local Version 6.0 database server, the **SQLRM** environment variable must be *unset* before **SQLEXEC** can be used to spawn a Version 6.0 relay module.

Unset **SQLRM** as follows.

Shell	Command
C	`unsetenv SQLRM`
Bourne	`unset SQLRM`

For information on the relay module, see the 6.0 version of your *Administrator's Guide.*

SQLRMDIR

This variable is not used in versions of 4GL later than 6.x.

If the database administrator is configuring a client/server environment in which a Version 4.1 4GL client accesses a local Informix 6.0 database server, the **SQLRM** environment variable must be *unset*.

Unset **SQLRMDIR** as follows.

Shell	Command
C	`unsetenv SQLRMDIR`
Bourne	`unset SQLRMDIR`

GLS

GLS Environment Variables

These variables affect global language support (GLS) or else are no longer recognized by 4GL; some of these variables are described in Appendix E.

GLS Environment Variable	Restrictions
CLIENT_LOCALE	
COLLCHAR	Not used by 4GL
DBAPICODE	
DB_LOCALE	
DBNLS	
GL_DATE	
GL_DATETIME	
LANG	Not used by 4GL
SERVER_LOCALE	

For details, see the descriptions of these environment variables in the *Informix Guide to GLS Functionality*. These environment variables typically do not need to be set for applications that run only in the default (U.S. English) locale.

If **DBNLS** is set to `1`, 4GL looks in the locale files for a nondefault collation sequence. If a collation order is defined, 4GL uses it, rather than the code-set order, in all sort operations that it performs on character values. If **DBNLS** is not set, or if the locale files define no collation sequence, 4GL sorts string values in code-set order.

Default Values of CLIENT_LOCALE and DB_LOCALE

The following table shows the values assumed by INFORMIX-4GL and INFORMIX-SQL when you define only some of the required values of locales.

(A value of `ja-jp.ujis` is assumed in the following table, where **CL** stands for **CLIENT_LOCALE** and **DL** stands for **DB_LOCALE**.)

User Defined				Values in 4GL	
CL Defined	**CL Value**	**DL Defined**	**DL Value**	**CL Value**	**DL Value**
No	--	No	--	en_us.8859	en_us.8859
Yes	ja_jp.ujis	No	--	ja_jp.ujis	ja_jp.ujis
Yes	ja_jp.ujis	Yes	ja_jp.ujis	ja_jp.ujis	ja_jp.ujis
No	--	Yes	ja_jp.ujis	en_us.8859	ja_jp.ujis

Default Value of DBLANG

If you do not set the **DBLANG** environment variable, by default it is set to the value of **CLIENT_LOCALE**.

Environment Configuration Files for Asian Locales

Earlier 4GL versions supported an equal sign (=), blank space (ASCII 32), or tab separator between the variable and its value in the **$HOME/.informix** file; or the **$INFORMIXDIR/etc/informix.rc** file; for example:

```
variable1 = value2
```

In versions of 4GL later than 6.x, however, only tabs or blank spaces are valid in these files as the separator:

```
variable1 value2
```

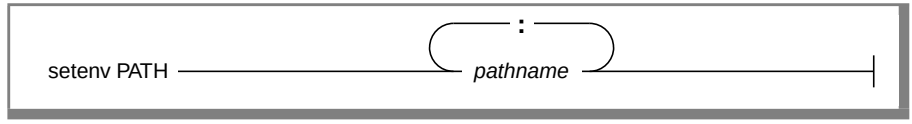
If you are upgrading to this version from a 6.x version of 4GL for deployment in East Asian locales, you might need to edit these configuration files to replace any equal sign that was used as a separator in this context.

UNIX Environment Variables

Informix products also rely on the correct setting of certain standard UNIX system environment variables. The **LD_LIBRARY_PATH, PATH,** and **TERM** environment variables must always be set. (Some platforms use a different name, such as **LPATH** or **SHLIB_PATH**, for **LD_LIBRARY_PATH**; see the section "Runtime Requirements" on page 1-45.) You also might need to set the **TERMCAP** or **TERMINFO** environment variable to use 4GL effectively.

PATH

The **PATH** environment variable specifies the order in which the UNIX shell searches directories for executable programs. You must include the directory that contains 4GL in your **PATH** setting before you can use 4GL.



Element	Description
pathname	specifies the search path for executables.

You can specify the correct search path in various ways. Be sure to include a colon (:) as a separator between UNIX directories. (On Windows NT systems, however, a semicolon (;) is the pathname separator.)

The following example uses the explicit path **/usr/informix**. This path must correspond to the **INFORMIXDIR** setting. The C shell example is valid for **.login** or **.cshrc** files.

Shell	Command
C	`setenv PATH $PATH:/usr/informix/bin`
Bourne	`PATH=$PATH:/usr/informix/bin` `export PATH`

The next example specifies **$INFORMIXDIR** instead of **/usr/informix**. It tells the shell to search the directories that were specified when **INFORMIXDIR** was set. The C shell example is valid for **.login** or **.cshrc** files.

Shell	Command
C	`setenv PATH $PATH:$INFORMIXDIR/bin`
Bourne or Korn	`PATH=$PATH:$INFORMIXDIR/bin` `export PATH`

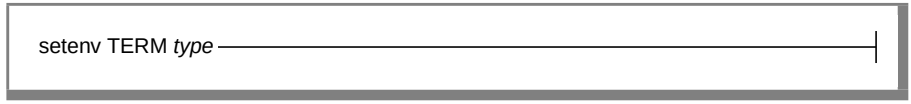
You might prefer to use this version to ensure that your **PATH** entry does not conflict with the path that was set in **INFORMIXDIR** and so that you do not need to reset **PATH** whenever you reset **INFORMIXDIR**.

If you set **PATH** on the C shell command line instead of in your **.login** or **.cshrc** file, you must enclose the **INFORMIXDIR** and **PATH** settings within braces ({ }), like this.

Shell	Command
C	`setenv PATH ${INFORMIXDIR}/bin:${PATH }`

TERM

The **TERM** environment variable is used for terminal handling. It enables 4GL to recognize and communicate with the terminal that you are using.



Element	Description
type	specifies the terminal type.

The terminal type specified in the **TERM** setting must correspond to an entry in the **termcap** file or **terminfo** directory. Before you can set the **TERM** environment variable, you must obtain the code for your terminal from the system administrator or database administrator.

For example, to specify the terminal whose code is `vt100`, set the **TERM** environment variable as follows.

Shell	Command
C	`setenv TERM vt100`
Bourne	`TERM=vt100` `export TERM`

TERMCAP

The **TERMCAP** environment variable is used for terminal handling. It tells 4GL to communicate with the **termcap** file instead of the **terminfo** directory.

Element	Description
pathname	specifies the location of the **termcap** file.

The **termcap** file contains a list of various types of terminals and their characteristics. Set **TERMCAP** as follows.

Shell	Command
C	`setenv TERMCAP /usr/informix/etc/termcap`
Bourne or Korn	`TERMCAP=/usr/informix/etc/termcap` `export TERMCAP`

If you set the **TERMCAP** environment variable, be sure that the **INFORMIXTERM** environment variable is set to the default, **termcap**.

TERMINFO

The **TERMINFO** environment variable is used for terminal handling. It is supported only on computers that provide full support for the UNIX System V **terminfo** library.

```
setenv TERMINFO /usr/lib/terminfo
```

TERMINFO tells 4GL to communicate with the **terminfo** directory instead of the **termcap** file. The **terminfo** directory has subdirectories that contain files pertaining to terminals and their characteristics.

Set **TERMINFO** as follows.

Shell	Command
C	`setenv TERMINFO /usr/lib/terminfo`
Bourne	`TERMINFO=/usr/lib/terminfo` `export TERMINFO`

If you set the **TERMINFO** environment variable, you also must set the **INFORMIXTERM** environment variable to **terminfo**.

See also Appendix F, "Modifying termcap and terminfo."

Appendix

E

Developing Applications with Global Language Support

The emerging global economy creates both opportunities and challenges for software application developers. Increasingly, applications are targeted toward multinational corporations or for customers in different countries. These applications typically require such international features as translatable user messages, menus, and reports, as well as date, time, and currency formats that can be easily changed to fit local cultural standards.

This appendix identifies fundamental terms and concepts that are involved in the localization of 4GL applications. Most of these terms are discussed in greater detail in the chapter on GLS fundamentals in the *Informix Guide to GLS Functionality*, primarily in relation to the database server.

Internationalization and Localization

The terms *internationalization* and *localization* are near antonyms, but they both describe activities that are critical for applications that will be deployed in more than one locale. The first term, *internationalization*, refers to the work of analysts and developers who must design and write code that is generalized for different cultural contexts. The second term, *localization*, refers to the work of developers and translators, who must adapt an internationalized application to the specific needs of a given linguistic or cultural setting.

Internationalization is the process of making software applications easily adaptable to different cultural and language environments.

Internationalization features support non-ASCII characters in character string values, and adaptable number, time, and currency formats. Internationalization also implies the ability to switch runtime environments from one language to another. Internationalization removes the need to recompile source code for a specific natural language or cultural environment.

A fully internationalized application can run in different cultural environments with minimal adjustments, in some instances by simply exchanging language-specific files and setting up the operating environment.

An internationalized application must support the use of extended ASCII code sets. The default environment for 4GL is based on the ASCII code set of 128 characters, as listed in Appendix A, "The ASCII Character Set." Each of these encoded values (or *code points*) requires seven bits of a byte to store each of the values `0` through `127`, representing the letters, digits, punctuation, and other logical characters of ASCII. Because each ASCII character can be stored within a single byte, ASCII is called a *single-byte* character set. All other character sets that 4GL can support must include ASCII as a subset.

An internationalized application should, at a minimum, be *8-bit clean*. A program, GUI, or operating system is referred to as 8-bit clean if it allows the high-order bit of a character code to take on a value of `1`. 4GL applications are 8-bit clean, and therefore support the use of extended ASCII character sets.

Localization is the process of translating and adapting an internationalized product to specific language and cultural environments.

Localization usually involves setting the appropriate number, time, and currency formats for the intended country, as well as creating a translation of the runtime user interface (including help and error messages, prompts, menus, and reports).

You can reduce the cost and effort of localization if the application is designed with international requirements in mind. This version of 4GL supports localization in several areas:

- Entry, display, and editing of non-English characters
- References to SQL identifiers containing non-English characters
- Collation of strings containing non-English symbols
- Non-English formats for number, currency, and time values

For basic GLS concepts and for details of how Informix database servers and the Client SDK products implement GLS, see the *Guide to GLS Functionality*.

Localization efforts are significantly easier when internationalization is built into the application from the start. "General Guidelines" on page E-18 provides guidelines for writing 4GL programs that can be easily localized.

Global Language Support Terms

Global language support (GLS) refers to the set of features that makes it possible to develop user interfaces and other parts of an application so that they can use non-Roman alphabets, diacritical marks, and so on. To understand the requirements of GLS, you will need to become familiar with the terms described in this section.

Code Sets and Logical Characters

For a given language, the *code set* specifies a one-to-one correspondence between each logical element (called a *logical character*, or a *code point*) of the character set, and the bit patterns that uniquely encode that character. In U.S. English, for example, the ASCII characters constitute a code set.

Code sets are based on logical characters, independent of the font that a display device uses to represent a given character. The size or font in which 4GL displays a given character is determined by factors independent of the code set. (For example, if you select a font that includes no representation of the Chinese character for *star*, only white space is displayed for that character until you specify a font that supports it.)

Collation Order

Collation order is the sequence in which character strings are sorted. Database servers can support collation in either *code-set order* (the sequence of code points) or *localized order* (some other predefined sequence). See the chapter on GLS fundamentals in the *Guide to GLS Functionality* for details of localized collation.

4GL sorts strings in code-set order, unless the COLLATION category of the locale files specifies a localized order (and **DBNLS** is set to 1). Informix databases can use localized collation for NCHAR or NVARCHAR columns of the database but sort CHAR and VARCHAR values in code-set order.

Single-Byte and Multibyte Characters

Most alphabet-based languages, such as English, Greek, and Tagalog, require no more than the 256 different code points that a single byte can represent. This simplifies aspects of processing character data in those languages; for example, the number of bytes of storage that an ASCII character string requires has a linear relationship to the number of characters in the string.

In non-alphabetic languages, however, the number of different characters can be much greater than 256. Languages like Chinese, Japanese, and Korean include thousands of different characters, and typically require more than one byte to store a given logical character. Characters that occupy two or more bytes of storage are called *multibyte characters*.

Locales

For 4GL (and for Informix database servers and connectivity products), a *locale* is a set of files that specify the linguistic and cultural conventions that the user expects to see when the application runs. A locale can specify these:

- The name of the code set
- The collation order for character-string data
- Culture-specific display formats for other data types
- The correspondence between uppercase and lowercase letters
- Determination of which characters are printable and which are nonprinting

The chapter on GLS fundamentals in the *Guide to GLS Functionality* provides details of formats for number, currency, and time values. If no locale is specified, default values are for United States English, which is the **en_us.8859-1** locale on UNIX systems or Windows code page 1252. For deployment, 4GL is also delivered with the locale **en_us.1252@dict**, which corresponds to that Windows code page.

The locale **en_us.1252@dict** allows you to compile and run programs that contain non-English characters from any single-byte language, but the default data formats are those of U.S. English. Alternatively, you can use the **Setnet32** utility to specify some nondefault locale, such as one of those listed in "Locales Supported by 4GL" on page E-9.

Global Language Support

GLS is a set of features that enable you to create localized applications for languages other than U.S. English and for country-specific cultural issues, including the localized representation of dates, currency values, and numbers. 4GL supports the entry, retrieval, and display of multibyte characters in some East Asian languages, such as Japanese and Chinese.

The following built-in functions or operators have been modified since version 6.0 of 4GL to provide support for non-English locales. Some can accept multibyte characters as arguments or operands, or can return values that include multibyte characters.

GLS-Enabled Built-In Function or Operator
CLIPPED operator
DOWNSHIFT( )
FGL_GETENV( )
FGL_KEYVAL( )
LENGTH( )
Substring ([ ]) operator
UPSHIFT( )
WORDWRAP operator

See Chapter 5, "Built-In Functions and Operators," for the syntax and semantics of these built-in functions and operators. (In addition, certain other built-in functions and operators of 4GL can also process or return multibyte values.)

Installation in Non-English Locales

This section identifies the general requirements for installation of 4GL in non-English locales. Because *non-English* refers to all locales other than **en_us.8859-1** (for UNIX) or **en-us.1252@dict** (for Windows), most locales of the English-speaking world are *non-English* in this context, as are the locales of most of the rest of the world.

The directory structure of Informix GLS products is shown in Figure E-1.

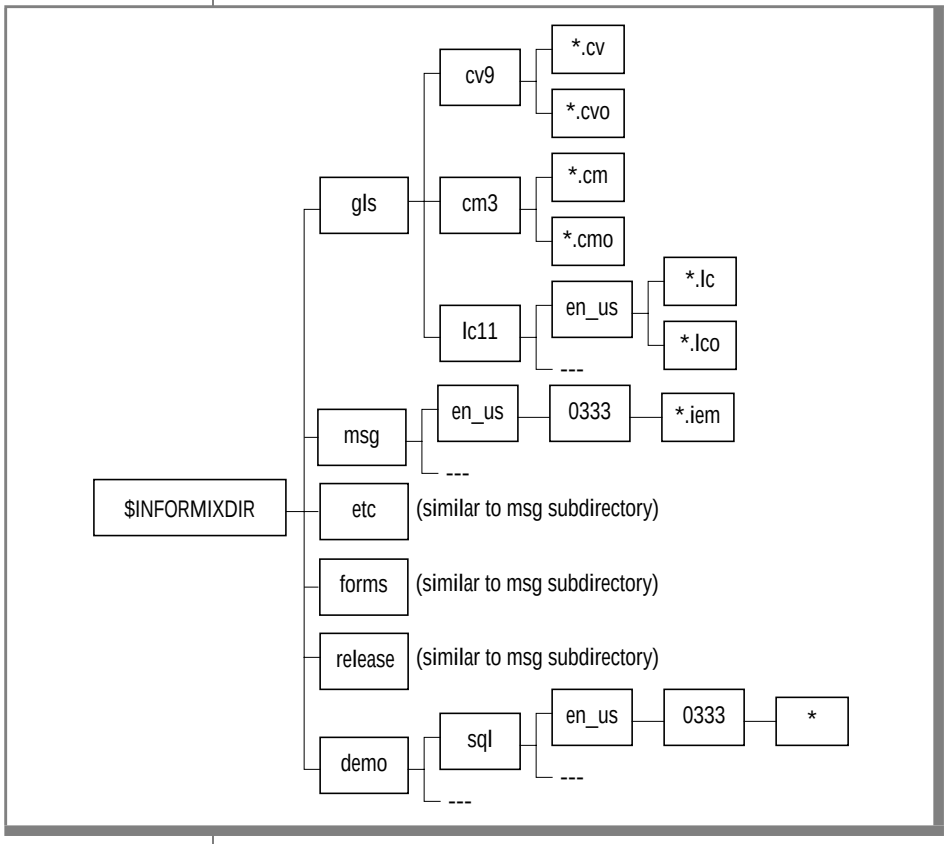


Figure E-1
Directory Structure of GLS Products

Requirements for International Application Development

The following requirements must be met to develop a 4GL application that is fully adapted to a language or to a country:

- The targeted hardware platform and operating system need to support the desired language and country combination.

  The operating system environment on both the client platform and the database server platform might require special versions to support the entry, manipulation, and display of non-English data.
- The Informix products need to support the language. Informix products are 8-bit clean and allow entry, manipulation, and display of most European and Asian language data.
- Error messages generated by 4GL and the database server should be available in a localized version, so that only local languages appear in the runtime environment.
- All parts of the user interface created by the application developer (such as menus, prompts, error messages, and help) should be translated into the target language.

In many cases, the last three of these four requirements can be met by using an Informix language supplement. Your Informix sales representative can advise you regarding the availability of language supplements, of localized versions of Windows, and of database servers that are compatible with 4GL.

Language Supplements

Use of 4GL with some non-English languages might require an Informix *language supplement* specific to the conventions of the country or language. Language supplements are currently required, for example, for Informix database servers to support each of the following East Asian languages.

Country or Language	Informix Language Supplement
People's Republic of China	Language Supplement ZHCN 7.20
Taiwanese	Language Supplement ZHTW 7.20
Japanese	Language Supplement JA 7.20
Korean	Language Supplement KO 7.20
Thai (simplified)	Language Supplement TH 7.20

Language supplements for these East Asian languages include locale files, translated message files, and translated menu files. Localized versions of 4GL for East Asian locales (for example, Japanese 4GL) will include the relevant files. See the release notes for additional information.

A corresponding International Language Supplement includes locale files and code-set conversion files for most European languages. Because most of these files are included with the ESQL/C software that is provided with 4GL, this supplement need not be purchased by 4GL customers unless the required locale is not included with 4GL.

When the Informix database server is installed in locales based on non-English European languages, both the default (English) database server and the International Language Supplement must be installed.

When 4GL is installed, the locale files must also be installed. Contact your Informix sales office for information regarding current support for specific locales.

Locales Supported by 4GL

A *locale* is the part of the processing environment that defines conventions for a given language or culture, such as formatting time and money values, and classifying, converting, and collating characters. The Informix GLS locale definition is similar to the X/Open CAE Specification.

Code sets that 4GL supports include those listed in the following table.

Country or Language
People's Republic of China
Taiwanese
Japanese
Korean
Eastern European (Latin)
Eastern European (Cyrillic)
Western European (Latin)
Greek
Turkish

4GL provides limited support for the Thai language through code set **th_th.thai620**, with Language Supplement TH 7.20, for non-composite Thai characters. (4GL does not support composite Thai characters.)

Client Locales and Database Server Locales

The locale of the system on which the 4GL application is running is called the *client locale*. The locale of the database server is called the *server locale*. "Handling Code-Set Conversion" on page E-34 describes special procedures that might be required if the client locale and the server locale are not identical.

Setting Environment Variables for Specific Locales

4GL requires that environment variables be set correctly on UNIX systems that support the database server or 4GL applications that support application server and display server partitions. For details about setting environment variables on UNIX systems for global language support, see the *Informix Guide to GLS Functionality*. See also "Configuring the Language Environment" on page E-24 for additional information about setting environment variables.

See the information on managing GLS files in the *Informix Guide to GLS Functionality* for an example of non-English locale files.

Requirements for All Locales

This section outlines the steps that are needed to create localized 4GL applications:

1. Set up the development environment.

   The system administration tools that you use must belong to the database server. You can use a UNIX terminal (if it supports the local code set).

2. Write the code.

   Filenames (source and compiled) must contain only English characters.

3. Compile and debug the code.

   The 4GL compiler can compile and link the components of the application.

   The Message Compiler can compile non-English text, so that runtime messages can be displayed in the local language. The user interface of the Message Compiler is in English.

   The INFORMIX-4GL Interactive Debugger is not GLS enabled. (The Dynamic 4GL Debugger has sufficient GLS capability to display non-English characters from the client locale.)

4. Deploy the code.

   Deployment is relatively unrestricted. Applications that can be created through the steps outlined here are *localized* applications for a specific locale and therefore are not *internationalized*. (That is, they should not be used in another locale that requires, for example, a different code set from that of the message files.)

The 4GL Compilers

The compilers have GLS capability, as sections that follow describe.

The 4GL Character Set

4GL keywords, identifiers, delimiters, and special symbols in source code are restricted to the same ASCII characters described in Chapter 2, "The INFORMIX-4GL Language." Additional printable characters from the client locale, however, are valid within source code files in the following contexts:

- Within comments
- Within 4GL identifiers
- Within certain SQL identifiers (as listed in the table in "SQL and 4GL Identifiers" on page E-13)
- Within expressions where character-string literals are valid

In non-English locales, 4GL identifiers can include non-ASCII characters in identifiers if those characters are defined in the code set of the locale that **CLIENT_LOCALE** specifies. In multibyte East Asian locales that support languages whose written form is not alphabet-based, a 4GL identifier need not begin with a letter, but the storage length cannot exceed 128 bytes. (A Chinese identifier, for example, that contains 128 logical characters would exceed this limit if any logical character in the identifier required more than one byte of storage.)

Non-English characters in other contexts, or characters that the client locale does not support, will generally cause compilation errors.

At runtime, the user can enter, edit, and display valid characters from the code set of the client locale. Whether a given character from a non-English code set is *printable* or *nonprintable* depends on the client locale.

Values that include non-English characters can be passed between a 4GL application and the database server, if the client and server systems have the same locale. If the locales are different, data can still be transferred between the 4GL client and the database server, provided that the client locale includes appropriate code-set conversion tables. See "Configuring the Language Environment" on page E-24, or GLS fundamentals in the *Informix Guide to GLS Functionality*, for information about establishing a locale and about code-set conversion between locales. See also "Handling Code-Set Conversion" on page E-34.

Non-English Characters

The following features of the 4GL compiler are GLS-enabled to support non-English characters that are valid in the client locale:

- Names of identifiers
- Values of CHAR and VARCHAR variables and formal arguments
- Characters within TEXT data
- Message text, quoted strings, and values returned by functions
- Text within comments, forms, menus, and output from reports

Named 4GL program entities include variables, functions, cursors, formal arguments, labels, reports, and prepared objects. 4GL has a limit of 128 bytes on the lengths of these names, but C compiler or linker restrictions might impose lower limits.

SQL and 4GL Identifiers

SQL identifiers are the names of database entities, such as table and column names, indexes, and constraints. The first character must be an alphabetic character, as defined by the locale, or an underscore (= ASCII 95) symbol. You can use alphanumeric characters and underscores (_) for the rest of the SQL identifier. Most SQL identifiers can be up to 18 bytes in length, if the **IFX_LONGID** environment variable is not set, or if it is set to `0` on the database server. (If **IFX_LONGID** is set to `1`, however, then SQL identifiers can be up to 128 bytes in length.)

What characters are valid in SQL identifiers depends on the locale of the database server; see "Client Locales and Database Server Locales" on page E-10. Neither single-byte nor multibyte white space characters can appear in SQL identifiers.

SE

For INFORMIX-SE database servers, whether non-English characters are permitted in the names of databases, tables, or log files depends on whether the operating system permits such characters in filenames. ♦

The user interface of the 4GL compiler is in English. If edit fields contain multibyte characters, there is no checking, and the results might be unpredictable. Embedded SQL statements can include valid non-English identifiers for some database entities. The following tables summarize the instances where non-English characters are valid as identifiers within 4GL source code modules. The first table lists SQL identifiers.

SQL Identifier	Allow Non-English Characters?
Column name	Yes
Constraint name	Yes
Database name	Yes (Operating system limitations on INFORMIX-SE)
Index name	Yes
Log filename	Yes (Operating system limitations on INFORMIX-SE)
Stored procedure name	Yes
Synonym	Yes
Table name	Yes (Operating system limitations on INFORMIX-SE)
View name	Yes

The following 4GL identifiers allow non-English characters.

4GL Identifier	Allow Non-English Characters?
Variable name	Yes
Cursor name	Yes
Filename or pathname	No
Formal argument name	Yes
Function or report name	Yes
Prepared statement name	Yes
Statement label	Yes

Input and output filenames for the 4GL compiler cannot be localized. Only ASCII characters are valid in input and output pathnames or filenames. (If support for uppercase ASCII letters is required, specify **en_us.1252@dict** as the locale at compile time. Uppercase letters are *not* defined in **en_us.1252**.)

Collation Sequence

The default collation (sorting) sequence in 4GL statements is implied by the *code-set order* in the files that define the client locale. If the **DBNLS** environment variable is set to `1`, 4GL looks in the locale files for the COLLATION category. If this category defines a nondefault collation order, 4GL uses this order, rather than the code-set order, for all sort operations that it performs on character values. (Here the COLLATION category functionally replaces the **LC_COLLATE** environment variable, which specified the nondefault sort order in earlier versions.)

If the locale files define a nondefault collation, this order is applied to all strings that the 4GL application sorts. In contrast, the database server applies localized collation (if any is defined) only to values from NCHAR and NVARCHAR database columns, but not to values from CHAR or VARCHAR columns (which it sorts in code-set order).

The 4GL application and the database server can use a different collation sequence, or a 4GL application can connect to two or more database servers that use different collation sequences in SQL operations. The collation sequence can affect the value of Boolean expressions that use relational operators as well as the sorted order of rows in queries and in reports.

Locale Restrictions

The compiler requires the **en_us.0333** locale. It accepts as input any source file containing data values in the format of the client locale. The compiler can generate binaries or p-code files with client-locale text strings. The runtime locale of a 4GL program must be the same as its compile-time locale.

The Forms Compiler

The **fglform** forms compiler can process form specifications that include non-English characters that are valid in the client locale. It can also produce compiled forms that can display characters from the client locale and that can accept such characters in input from the user.

The Message Compiler

The **mkmessage** message compiler has a user interface in English but can compile non-English text into runtime messages in the local language.

East Asian Language Support

4GL can create applications for Asian languages that use multibyte code sets. The following features are supported by 4GL in multibyte locales:

- Menu items, identifiers, and text labels in the native language
- Features to avoid the creation of partial characters
- Non-English data within 4GL applications
- Cultural conventions, including the representation of date, time, currency, and numeric values, and localized collation
- Kinsoku processing for Japanese language text with WORDWRAP
- Icon modification without changing the 4GL application binary
- Text geometry that adjusts automatically to meet localization needs
- Application comparisons that adopt the comparison rules and collating sequence that the locale defines implicitly (SQL comparison and collation depend on the database server.)

This version of 4GL does not support composite characters, such as are required in some code sets that support the Thai language.

4GL comments and character string values can include multibyte characters that are supported by the client locale in contexts like these:

- Character expressions and multiple-value character expressions
- Literal values within quoted strings
- Variables, formal arguments, and returned values of CHAR, VARCHAR, and TEXT data types

Multibyte characters can also appear in 4GL source code (or in user-defined query criteria) that specifies the SQL identifier of any of the database objects listed in the table on "SQL and 4GL Identifiers" on page E-13. 4GL does not, however, support multibyte characters as currency symbols or as separators in display formats that **DBDATE** or **DBFORMAT** specifies.

Logical Characters

Within a single-byte locale, every character of data within character-string values requires only a single byte of memory storage, and a single character position for display by a character-mode device.

This simple one-to-one relationship in character-string operations between data characters, display width, and storage requirements does not exist in East Asian locales that support multibyte characters. In such locales, a single logical character might correspond to a single byte or to two or more bytes. In such locales, it becomes necessary to distinguish among the *logical characters* within a string, the *display width* that the corresponding glyph occupies in a display or in report output, and the number of *bytes* of memory storage that must be allocated to hold the string.

In locales that support multibyte characters, some built-in functions and operators that process string values operate on logical characters, rather than on bytes. For code sets that use multibyte characters, this modifies the byte-based behavior of several features in 4GL 6.x (and earlier versions). A single logical character can occupy one or more character positions in a screen display or in output of a report, and requires at least one byte of storage, and possibly more than one.

Declaring the CHAR or VARCHAR data types of variables, formal arguments, and returned values is *byte* based. Runtime processing of some character strings, however, is done on a *logical character* basis in multibyte locales.

Partial Characters

The most important motivation for distinguishing between logical characters and their component bytes is the need to avoid partial characters. These are fragments of multibyte characters. Entering partial characters into the database implies corruption of the database, and risks malfunction of the database server.

Partial characters are created when a multibyte character is truncated or split up in such a manner that the original sequence of bytes is not retained. Partial characters can be created during operations like the following ones:

- Substring operations
- INSERT and UPDATE operations of SQL
- Word wrapping in reports and screen displays
- Buffer to buffer copy

4GL does not allow partial characters and handles them as follows:

- Replaces truncated multibyte characters by single-byte white spaces
- Wraps words in a way that ensures that no partial characters are created in reports and screen displays
- Performs code-set conversion in a way that ensures that no partial characters are created

For example, suppose that the following SELECT statement of SQL:

```
SELECT col1[3,5] FROM tab1
```

retrieved three data values from **col1** (where **col1** is a CHAR, NCHAR, NVARCHAR, or VARCHAR column); here the first line is not a data value but indicates the alignment of bytes within the substrings:

AA^2BB^2AA	*becomes*	"s^1Bs^1"
ABA^2C^2AA	*becomes*	"A^2s^1"
A^2B^2CABC	*becomes*	"B^2C"

Here the notation s^1 denotes a single-byte white space. Any uppercase letter followed by a superscript (^{2}) means an East Asian character with multibyte storage width; for simplicity, this example assumes a 2-byte storage requirement for the multibyte characters. In the first example, the A^2 would become a partial character in the substring, so it is replaced by a single-byte white space. In the same substring, the B^2 would lose its trailing byte, so a similar replacement takes place.

General Guidelines

This section lists the issues that you need to consider when writing and translating applications.

Internationalization Guidelines

To make a 4GL application world-ready, keep the following guidelines in mind:

- Do not assume that application users are English speaking or will accept any pre-set business rules or formats.
- Use code libraries wherever possible. This centralizes common code and makes changes and maintenance easier when developing for international markets.

  Specific programming areas that might require special attention (and that are treated in detail in the chapter on GLS fundamentals in the *Informix Guide to GLS Functionality*) include:

  - Character-string display, entry, storage, retrieval, and processing
  - Formats for literal date, time, currency, and numeric values
  - Code-set conversion between client and server
- In all windows that will appear in more than one language, consider differences in word length among languages when you are designing the window and its graphical objects.
- Allow space for the expansion of user message strings. Brief English strings such as *Popup* can double in size as a result of translation. On average, you can expect a 30% increase in the size of messages.

- When designing windows, remember that names, addresses, dates, times, and telephone numbers have different formats in different countries.
- When possible, use picture buttons instead of buttons with titles.
- Consider that measurement systems can also differ. Most countries outside the U.S. express quantities using the metric system. For example, liters, centimeters, and kilometers instead of quarts, inches, and miles.
- Make sure that all screens, menus, user messages, reports, help facilities, and application parameters (such as holidays, bank years, formulas) that were developed with Informix tools for the application are either table-driven or are controlled by text files or environment variables that are easy to modify. This issue is discussed later in this appendix.
- Avoid embedding any messages, prompts, or elements of the user interface into the source code of the program. Ideally, all user interface elements can be switched dynamically by referencing a different set of translated files.
- Consider different keyboard layouts. A character (such as "/") that is easily accessible on an ASCII keyboard might require several keystrokes in the standard keyboard of some other country.
- Consider creating a configuration utility to deal with different font types. Some applications that will be deployed in several different countries might need to load different fonts to accommodate specific national characters.

  Because these fonts are often supplied by third parties, you might not be able to predict the font names when you develop the application. In this case, you can use the default font names and provide a configuration utility that allows the user to specify the font name before running the application.
- Consider differences in paper size when designing reports. Most countries outside the U.S. use the ISO Standard A4 paper size, which is 21 by 29.7 centimeters, slightly longer and narrower than the American standard 8.5 by 11 inches.

- Avoid fragmentation of messages or potentially ambiguous key or command words. Avoid determining variable portions of a message at runtime; for example, the differing syntax of other languages can make the order in which your functions return parameters an obstacle to correct translation.
- Wherever possible, avoid abbreviations, acronyms, contractions, and slang.
- Place comments around any string pertaining to the user interface to facilitate localization.
- Use localized error messages and help files. The message compiler utility that is provided with 4GL enables you to create customized help files as well as a localized version of the 4GL runtime message file (the **4gluser.msg** file in the **msg** directory). Internationalizing messages is further discussed in "Localizing Prompts and Messages" on page E-32.
- You can handle reports (which are 4GL programs) in the same way that you internationalize the rest of your 4GL source code.

Localization Guidelines

As noted previously, localization refers to the actual process of adapting the application to the cultural environment of end users. This process often involves translation of the user interface and user documentation and can be time consuming and costly. Here are some guidelines to follow:

- Consult the native operating system internationalization guide.

  Most platforms provide documentation on internationalization. This material might help you to determine which date, time, and money formats are appropriate for the target language and culture.

  For more information about internationalizing Informix products, see the *Informix Guide to SQL: Reference*.

  For information about the terms and constructs of Informix global language support (GLS) technology, see the *Informix Guide to GLS Functionality*.
- Make sure the targeted hardware, operating system environments, and Informix product versions of your applications can support the desired language and culture.

- Find out if the runtime environment of 4GL and of the database server is currently available in the target language.

  For example, the 4GL runtime environment (and the Informix Dynamic Server administrator's environment) is usually translated into several languages, including French, German, Spanish, Russian, and Japanese.

- Keep a glossary of all strings and keywords in a database or text file.

  This glossary will make it easier to see which messages are duplicated throughout the source code. The glossary will also increase the consistency of terms and language in the user interface throughout the application. Once the glossary is created for one language, it can be used for product updates and additional localizations.

- Create a mechanism that allows a glossary to drive the definition of the user interface.

  This can be particularly useful if you expect to localize the application often. A translator can edit the glossary without having to understand the source code of the application. Your tool can then create the user interface from the translated glossary, and the translator can focus on making cosmetic enhancements to the translation (such as positioning the messages appropriately) and correcting minor errors.

- Consider creating a checklist of those user interface elements in your application that should be externalized into text files from the source code, and therefore from the compiled portion of the program. These text files can then be modified even after the program is compiled. Externalize the following elements:

  - Menus
  - Forms
  - Messages
  - Labels
  - Help (**.msg**) text
  - Numeric, date, time, and money formats
  - Report names

- Consider retaining a professional translator for some or all of this process.

  A faulty translation is very costly. You can spend a great deal of time and money correcting errors in your localized product. And if you do not correct the problems, your users will be dissatisfied with your application.

Localization Methodology Overview

This section lists the elements of an application and indicates some ways in which each can be localized. This overview, while not comprehensive, illustrates how to approach a project of this nature. The rest of this appendix expands on the approaches listed here.

For many of the application elements discussed in this section, the two methods of localization are the *table-based* approach and the *file-based* approach. The table-based approach involves obtaining translation information from a database using SQL queries. The file-based approach involves retrieving the values of the variables from a text file.

Application Help and Error Messages

Two methods are available for localizing application help and error messages.

Table-Based Localization of Messages

To use this method, you need to verify the availability of tables. It often also requires the hard coding of defaults in case the database cannot be accessed.

File-Based Localization of Messages

This method uses the **mkmessage** message compiler utility to create help and error message files. For more information, see "Localizing Prompts and Messages" on page E-32.

Date, Time, and Currency Formats

To localize formats for date, time, and money values, set the Informix environment variables **DBFORMAT**, **DBMONEY**, and **DBDATE**. Formatting conventions of some East Asian locales require that the **GL_DATE** or **GL_DATETIME** environment variable be set. For more information about these and other environment variables, see Appendix D, "Environment Variables."

Informix System Error Messages

The following methods are available for localizing Informix system messages and error messages.

Informix Translation

Informix provides error message translation for a variety of languages. You can use the **DBLANG** environment variable to point to a message directory containing translated messages. Contact your local Informix sales office for a list of available language translations.

Customized System Error Message Files

If no Informix translation of the error messages is available, and if the source code of error message files is delivered with the product, you can localize the message source files using the **mkmessage** utility. For more information, see "Localizing Prompts and Messages" on page E-32.

Code-Set Conversion

For details, see "Handling Code-Set Conversion" on page E-34.

Configuring the Language Environment

Environment settings that affect the language environment exist both in your 4GL environment and in your system environment. Using the GLS features of 4GL with Informix database servers involves several compatibility issues:

- The English database servers create English databases with ASCII data. For these, the 4GL program must access the database servers with **DB_LOCALE** set to **en_us.8859-1**.
- The 5.x ALS versions of Informix database servers can use variables such as **DBCODESET** and **DBCSOVERRIDE** as substitutes for **DB_LOCALE** and **DBCONNECT**, respectively. These environment variables need to be set by using **Setnet32**.
- The 5.x ALS versions use **DBASCIIBC** to emulate the 4.x ASCII database servers. This environment variable should be set in the registry if such behavior is desired.
- The **SERVER_LOCALE** environment variable is set on the database server, not on the 4GL client. This variable specifies the locale that the database server uses to read or write operating system files. If it is not set, the default is U.S. English (**en_us.8859-1**).

If no setting is specified, the 4GL application uses an English locale. But the registry sets everything to the local language, code set, or locale, so the practical default is for applications to use the local locale.

The non-internationalized portions of the product are initialized with the default (U.S. English) locale. That is, both **CLIENT_LOCALE** (**en_us.1252**) and **DB_LOCALE** (**en_us.8859-1**) are set to English. This initialization is necessary because many common functions are shared between the internationalized and non-internationalized components.

Important: *Except for **DBFORMAT**, all of the environment variables that are described in the sections that follow apply to Informix database servers.*

Please note also the following considerations:

- The application cannot support connections to different databases with different locales concurrently; for example, in extended joins.
- The environment variables discussed here deal with the environment **DB_LOCALE** that is passed to the database server.
- **CLIENT_LOCALE** cannot be changed dynamically during execution.
- The previous point has one exception: the **CLIENT_LOCALE** can always be set to English (because English is a subset of all locales).

The **DB_LOCALE** code set should match the **DB_LOCALE** code set of the database. Otherwise, data corruption can occur because no validation of code-set compatibility is performed by the database server.

Environment Variables That Support GLS

This section examines the environment variables that support the GLS capabilities of 4GL, including the following 4GL environment variables:

- **DBDATE** defines date display formats.
- **DBMONEY** defines monetary display formats.
- **DBFORMAT** defines numeric and monetary display formats and has more options than **DBMONEY**.

INFORMIX-4GL also supports the following GLS environment variables:

- **DB_APICODE** specifies a code set that has a mapping file.
- **DB_LOCALE** is the locale of the database to which the application is connected.
- **CLIENT_LOCALE** is the locale of the system that is executing the 4GL application.
- **DBLANG** points to the directory for Informix error messages.
- **DBNLS** supports localized collation and other GLS features.
- **GL_DATE** defines date displays, including East Asian formats.
- **GL_DATETIME** defines date and time displays, including East Asian formats.
- **SERVER_LOCALE** is the locale of the database server for file I/O.

4GL does not use **DB_LOCALE** directly; this variable, as well as **DBLANG**, is used by ESQL/C. See the *Informix Guide to GLS Functionality* for details of **DBLANG, DB_LOCALE, GL_DATE**, **GL_DATETIME**, and other GLS environment variables. See also Appendix D, "Environment Variables."

DBAPICODE

This environment variable specifies the name of a mapping file for peripheral devices (for example, a keyboard, a display terminal, or a printer) whose character set is different from that of the database server.

DB_LOCALE

This environment variable specifies the locale of the database to which the 4GL component or application is connected. The only Informix databases that currently support non-English languages exist in UNIX. Therefore, when the locales are non-English, the localized 4GL application can only connect to these databases. The format for setting **DB_LOCALE** is DB_LOCALE=<*locale*>.

The following points should be noted regarding **DB_LOCALE**:

- If the application uses this value to access a database, the locale of that database must match the value specified in **DB_LOCALE**. If it does not match, the database connection might be refused (unless **DBCSOVERRIDE** is set to 1), depending on the database server version.
- If a database is created, this new database has the value specified by **DB_LOCALE**.
- If **DB_LOCALE** is invalid, either because of wrong formatting or specifying a locale that does not exist, an error is issued.
- If the code set implied by **DB_LOCALE** cannot be converted to what **CLIENT_LOCALE** implies, or vice versa, an error is issued.
- If **DB_LOCALE** is not specified, there is no default value; in this case, ESQL/C behaves as if code-set conversion were not needed.

Default Values of GLS Environment Settings

The following table shows the values assumed by 4GL when you define only some of the required values of locales.

(A value of `ja-jp.ujis` is assumed in the following example, **CL** stands for **CLIENT_LOCALE**, and **DL** stands for **DB_LOCALE**.)

User Defined				Values in Product	
CL Defined	**CL Value**	**DL Defined**	**DL Value**	**CL Value**	**DL Value**
No	--	No	--	en_us.8859	en_us.8859
Yes	ja_jp.ujis	No	--	ja_jp.ujis	ja_jp.ujis
Yes	ja_jp.ujis	Yes	ja_jp.ujis	ja_jp.ujis	ja_jp.ujis
No	--	Yes	ja_jp.ujis	en_us.8859	ja_jp.ujis

If you do not set the **DBLANG** environment variable, it is set to the value of **CLIENT_LOCALE**.

CLIENT_LOCALE

This environment variable specifies the locale of the (input) source code and the compiled code (to be generated). This is also the locale of the error files (if any) and the intermediate files. The format of **CLIENT_LOCALE** is the same as that of **DB_LOCALE**:

- The characters that reach the user interface (the non-ASCII characters) must be in the **CLIENT_LOCALE**.
- If **DB_LOCALE** is invalid, either because of wrong formatting or specifying a locale that does not exist, an error is issued.
- The **DB_LOCALE** and **CLIENT_LOCALE** settings need to be compatible, meaning there should be proper code-set conversion tables between them. Otherwise, an error is issued.
- If **CLIENT_LOCALE** is not set, Windows code page 1252 is the default.

- The **CLIENT_LOCALE** must match the environment of the user interface (meaning that it should be compatible with the local version of Windows). Otherwise, an error is issued.
- By default, collation by the 4GL application follows the code-set order of **CLIENT_LOCALE**, except in SQL statements (where the database server uses its own collation sequence). Any **LC_COLLATE** specification is ignored. If **DBNLS** is set to 1, however, and the COLLATION category of the locale file defines a nondefault order for sorting strings, 4GL uses the order that COLLATION specifies.

DBLANG

The value of **DBLANG** is used to complete the pathname to the directories that contain the required message, help, and demo files. The format of **DBLANG** is the same as that of **DB_LOCALE**:

- If **DBLANG** is not set, the value defaults to that of **CLIENT_LOCALE**.
- If **DBLANG** is invalid, **en_us.1252** is the default value. This case occurs if **DBLANG** is improperly formatted, or if it points to a locale that does not exist, or points to a locale that is incompatible with the version of Windows on which the 4GL application is running.

See also the description of **DBLANG** in the *Informix Guide to GLS Functionality*.

DBDATE

The **DBDATE** environment variable has been modified to support era-based dates (Japanese and Taiwanese). The days of the week and months of the year (in local form) are stored in the locale files. If this environment variable is set, it might override other means of specifying date formats.

DBNLS

The **DBNLS** environment variable must be set to `1` if the 4GL application needs to pass values between CHAR or VARCHAR program variables and database columns of the NCHAR or NVARCHAR data type. The same setting is also required if you want character strings to be sorted in a nondefault collation sequence rather than in code-set order. (In this case, your locale files must also include a COLLATION category to define the nondefault sorting.)

DBMONEY

This environment variable has been modified to accept multibyte currency symbols. 4GL components that read the value of **DBMONEY** (or **DBFORMAT**) must be able to correctly process multibyte characters as currency symbols. If **DBMONEY** is set, its value might override other means of specifying currency formats.

DBFORMAT

This environment variable has been modified to accept multibyte currency symbols. Unlike the version of **DBFORMAT** for English products, display of the decimal point is optional, rather than mandatory, in 4GL. (Use of a comma as the **DBFORMAT** decimal separator can produce errors or unpredictable results in SQL statements in which 4GL variables are expanded to number values that are formatted with comma as the decimal separator.)

If **DBFORMAT** is set, its value can override other means of specifying number or monetary formats.

See also the descriptions of **DBDATE**, **DBFORMAT**, and **DBMONEY** in Appendix D.

The **glfiles** utility is described in the *Informix Guide to GLS Functionality* and is packaged with INFORMIX-4GL and INFORMIX-SQL products. This utility allows you to generate lists of the following files:

- GLS locales available in the system
- Informix code-set conversion files available
- Informix code-set files available

Storing Localization Information

This section describes the process involved in creating an application so that it can read translation information either from a file or from a database table at runtime.

File-Based Localization

You can store the translations of localized information in disk files and access them at runtime as needed.

You can use subdirectories to store language-sensitive files, so they can easily be switched to create a new runtime environment. In the following example, the filename is composed by reading the value of an environment variable (created by the programmer) that specifies a language subdirectory:

```
LET file001 = FGL_GETENV("LANGUAGE"), "/", "trans.4gl"
# Evaluates to "eng/trans.4gl" if LANGUAGE is "eng"
# Program reads the eng directory for copy of translation
#
# Evaluates to "ger/trans.4gl" if LANGUAGE is "ger"
# Program reads the ger directory for copy of translation
#
LET tranfile = file001
```

Table-Based Localization

As noted earlier, localization information can also be stored in database tables. This information can be used when you initialize or run the application to change the value of variables that define titles, menus, and other language or culturally sensitive elements of the user interface. An advantage of the table-based approach is that it is highly portable between systems.

Setting Up a Table

The following example shows one way that you might set up a table to store menu options:

```
CREATE TABLE menu_elements(
   option_language    CHAR(3),  #language ID code
   option_number      SMALLINT, # identifying number
   option_text        CHAR(80), # text
   option_maxlen      SMALLINT  # maximum length of string
   )

CREATE UNIQUE INDEX ix_menustr
   ON menu_elements(option_language, option_number)
```

Example data:

```
ENG150Cold Beer
FRE150Bière froide
GER150Kaltes Bier
SPA150Cerveza fría
ENG151Iced Tea
...
```

Querying the Table

A global variable that contains the language code of the application, which corresponds to the value in the **option_language** column, can be set in the program at startup. Each time a character string is needed, a function could be called that uses the language and identifying number to query the table for the appropriate string:

```
LET lang = getLanguage()# returns 3 letter code
   # from option_language column
```

Localizing Prompts and Messages

You can use the 4GL message compiler utility to create translated message files for your application messages. These files, which usually have the extension **.iem**, run very quickly.

Creating Message Files

For any natural language, follow these steps to create new language versions of the messages and prompts that your application displays.

To create new message files

1. With a text editor that can create flat files, create a source (**.msg**) file with the following format:

   ```
   .message-number
   message-text
   .message-number
   message-text
   ```

   For example:

   ```
   .1000
   Part not found.
   .1001
   Price must be a positive number.
   .1002
   Invalid format for phone number.
   ```

   To translate the messages into another language, simply provide translated versions for the message text, using the same format.

2. At the system prompt, invoke the message compiler utility (**mkmesssage**) by using a command of the following form:

   ```
   mkmsg filename
   ```

   The message compiler processes *filename***.msg** and produces a compiled message file that has the name *filename***.iem**.

   If you want the compiled message file to have a different name from the source file, specify that filename as a final argument:

   ```
   mkmsg source output
   ```

   The syntax of **mkmesssage** is described in Appendix B, "INFORMIX-4GL Utility Programs."

Accessing Message Files

To access the compiled message file from your application, you can write a function that reads the messages from the compiled (**.iem**) file. For example, the calling program includes logic to display a `Part not found` message in the following pseudo-code:

```
DEFINE OK, noPart INT, msg CHAR(79)
LET noPart = 1000
   ...
IF (status == NOTFOUND) THEN
   CALL getMsg(noPart)
END IF
```

Function **getMsg( )** reads the compiled message file and finds the message that corresponds to the integer value of the variable **noPart**.

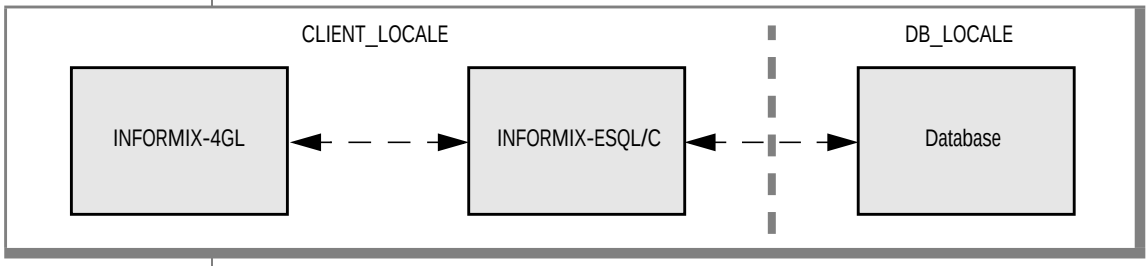
To supply new versions of the messages, you need only provide a new source file and compile it with the message compiler. The function calls in your application remain the same.

Handling Code-Set Conversion

The process of converting characters at the locale of the 4GL application to characters at the locale of the database server (or vice versa) is called *code-set conversion*. If your application needs to run on computers that encode different character sets, it might be necessary to enable code-set conversion. This section provides some background and details.

Code-set conversion is performed by ESQL/C; no explicit code-set conversion is done by 4GL. Figure E-2 shows the relationship between 4GL, ESQL/C, and the database.

Figure E-2
Processes and Their Locales



The code sets in the **CLIENT_LOCALE** can differ from those in **DB_LOCALE**. In the **CLIENT_LOCALE**, the code sets (which are specified in locales) use code points that are pre-defined by Microsoft standards. Code sets that are used in the **DB_LOCALE** tend to use characters that are based on UNIX conventions, if the application is designed to access legacy data.

Code-set conversion is done by way of a code-set conversion file. Files for code-set conversion between **CLIENT_LOCALE** and **DB_LOCALE** need to be present on the client. For conversion to take place, conversion files need to be present in the **%informixdir%\gls\cv** directory.

For details of converting between client and server code sets, see the sections that follow. See also GLS fundamentals in the *Informix Guide to GLS Functionality*.

What Is Code-Set Conversion?

Different operating systems sometimes encode the same characters in different ways. For example, the character *a-circumflex* is encoded:

- in Windows code page 1252 as hexadecimal 0xE2.
- in IBM CCSID 437 as hexadecimal 0x83.

If the encoding for *a-circumflex* on the Windows system is sent unchanged to the IBM system, it will be printed as the Greek character gamma. This happens because, on the IBM system, gamma is encoded as 0xE2.

This means character data strings that are passed between two computers using different character set encodings must be converted between the two different encodings. Otherwise, character data originating from one computer will not be correctly displayed or processed on the other computer.

This appendix uses the term *code set* in the same way that the Windows documentation uses the terms *character set* and *code page*.

Converting character data from one encoding schema to another is called *code-set conversion*. If a code-set conversion is required from computer A to computer B, it is also required from computer B to computer A. You must explicitly enable code-set conversion; no conversion is done by default.

What Code-Set Conversion Is Not

Code-set conversion is not a semantic translation; that is, it does not convert words between different languages. For example, it does not convert between English *yes* and French *oui*. It only ensures that each character is processed and printed the same, regardless of how the characters are encoded.

Code-set conversion does not create a character in the target code set if the character exists only in the source code set. For example, if the character *a-circumflex* is being passed to a computer whose code set does not contain an *a-circumflex* character, the target computer will never be able to exactly process or print the *a-circumflex* character. This situation is described in more detail in “Mismatch Processing” on page E-37.

When You Do Not Need Code-Set Conversion

You do not need code-set conversion in any of the following situations:

- The client and the server are on the same computer.
- The code set of your client and of all the databases to which you are connecting are the same.
- The subset of characters that you will be sending between the client and the server are encoded identically. For example, if you are sending only English characters between a client and a server, and each English character has the same encoding on both computers, no code-set conversion is required. In this case, the non-English characters can have different encodings.
- The character-string data values are passed from the client to the server for storage only and are neither processed nor printed by the server. For example, no code-set conversion is required if a client:
  - Passes character-string data to the server
  - Does not process or print the data on the server computer
  - Retrieves the same data for processing or printing on computers that use the same code set as the client that populated the database

Sorting data by using the ORDER BY statement or retrieving data by using a LIKE or MATCHES clause, however, will probably produce erroneous results if the data strings are not converted before they are stored.

What Data Values Are Converted

If you enable code-set conversion, data values are converted by ESQL/C from the 4GL client to the database server and from the database server to the client. The CHAR, VARCHAR, and TEXT data types are converted, as are column names, table names, database names, and SQL command text.

Mismatch Processing

If both code sets encode exactly the same characters, mismatch handling is unnecessary. If the source code set contains any characters that are not contained in the target code set, however, the conversion must define how the mismatched characters are to be mapped to the target code set.

Four ways that code-set conversions handle mismatch processing are as follows:

- **Round-trip conversion.** This method maps each mismatched character in the source code set to a unique character in the target code set. On the return, the original character is mapped back to itself. This guarantees that a two-way conversion will result in no loss of information; however, data converted in only one direction might confuse the processing or printing on the target computer.
- **Substitution conversion.** This method maps all mismatched characters in the source code set to a single specific character in the target code set that serves to highlight mismatched characters. This guarantees that a one-way conversion will clearly show the mismatched characters; however, a two-way conversion will result in information loss if mismatched characters are transferred.
- **Graphical replacement conversion.** This method maps each mismatched character in the source code set to a character in the target code set that resembles the source character (this includes mapping one-character ligatures to their two-character equivalents). This might confuse printing on the target computer. Round-trip conversions should contain as many graphical replacement conversions as possible.
- **Substitution plus graphical replacement.** This method maps as many mismatched characters as possible to their graphical replacements, and maps the remaining mismatched characters to the substitution character.

Informix-supplied code-set conversion source files have header comments that indicate which method was used.

Enabling Code-Set Conversion

Code-set conversion is handled by UNIX environment variables.

To establish code-set conversion

1. Determine the code set used by the client.
2. Determine the code set used by all the databases to which this client will be connecting in a single connection.
3. Specify the conversion filenames.
4. Start the application.

Determining the Code Sets Used by the Client and Database

Because each operating system has its own way of declaring the code set it is using, consult your UNIX operating system documentation or your system administrator to determine the code set used by the client computer.

Your system administrator should also know which code set is being used by the database.

Specifying the Conversion Filenames

Set the **DBAPICODE** environment variable to specify a code set that has a mapping file in the message directory **$INFORMIXDIR/msg** (or a directory pointed to by the **DBLANG** value). The Informix **crtcmap** utility helps you to create mapping files.

For detailed information about **DBAPICODE** and the **crtcmap** utility, see the *Informix Guide to SQL: Reference*.

Appendix

F

Modifying termcap and terminfo

INFORMIX-4GL programs can use function keys and can display color or intensity attributes in screen displays. These and other keyboard and screen options are terminal-dependent. To determine terminal-dependent characteristics, 4GL uses the information in the **termcap** file or in the **terminfo** directory. 4GL uses the **INFORMIXTERM** environment variable to determine whether to use **termcap** or **terminfo**. For more information about **INFORMIXTERM**, see Appendix D, "Environment Variables."

With 4GL, Informix distributes **termcap** files that contain additional capabilities for many common terminals (such as the Wyse 50 and the Televideo 950). These capabilities include intensity-change or color-change descriptions or both. This appendix describes these capabilities, as well as the general format of **termcap** and **terminfo** entries.

Because **terminfo** does not support color, you can only use 4GL color functionality with **termcap**. To use color in 4GL, you must set the **INFORMIXTERM** environment variable to **termcap**.

You can use the information in this appendix, combined with the information in your terminal manual, to modify the contents of your **termcap** file or **terminfo** file. This appendix is divided into two main sections, **termcap** and **terminfo**. Depending on which you are using, read the appropriate section.

termcap

When 4GL is installed on your system, a **termcap** file is placed in **$INFORMIXDIR/etc**. This file is a superset of an operating system **termcap** file. The Informix **termcap** file contains additional capabilities for many terminals. You might want to modify this file further in the following instances:

- To include color-change and intensity-change capabilities
- To extend function key definitions
- To specify or alter the graphics characters used for window borders
- To customize your terminal entry in other ways

Some terminals cannot support color or graphics characters. Read this appendix and the user guide for your terminal to determine whether or not the changes described in this appendix are applicable to your terminal.

Format of a termcap Definition

This section describes the general format of **termcap** entries. For a complete description of **termcap**, refer to your operating system documentation.

A **termcap** entry contains a list of names for the terminal, followed by a list of the terminal's capabilities. There are three types of capabilities:

- Boolean capabilities
- Numeric capabilities
- String capabilities

All **termcap** entries have the following format:

- The ESCAPE character is specified as a backslash (\ ) followed by the letter *E*, and CONTROL is specified as a caret (^) symbol. Do not attempt to use the ESCAPE or CONTROL keys to indicate escape sequences or control characters in a **termcap** entry.
- Each capability, including the last one in the entry, is followed by a colon (:).
- Entries must be defined on a single logical line; a backslash appears at the end of each line that wraps to the next line.

The following example shows a basic **termcap** entry for the Wyse 50 terminal:

```
# Entry for Wyse 50:

w5|wy50|wyse50:
:if=/usr/lib/tabset/std:\
:al=\EE:am:bs:ce=\Et:cm=\E=%+ %+ :cl=\E*:co#80:\
:dc=\EW:dl=\ER:ho=^^:ei=:kh=^^:im=:ic=\EQ:in:li#24:\
:nd=^L:pt:se=\EG0:so=\EG4:sg#1:ug#1:\
:up=^K:ku=^K:kd=^J:kl=^H:kr=^L:kb=:\
:k0=^A@^M:k1=^AA^M:k2=^AB^M:k3=^AC^M:k4=^AD^M:\
:k5=^AE^M:k6=^AF^M:k7=^AG^M:\
:HI=^|:Po=^R:Pe=^T:
```

Comment lines begin with a pound sign (#).

Terminal Names

A **termcap** entry starts with one or more names for the terminal, each separated by a pipe symbol (|). For example, the **termcap** entry for Wyse 50 terminals starts with the following line:

```
w5|wy50|wyse50:\
```

The **termcap** entry can be accessed by using any one of these names.

Boolean Capabilities

A Boolean capability is a two-character code that indicates whether or not a terminal has a specific feature. If the Boolean capability is present in the **termcap** entry, the terminal has that particular feature. The following example shows some of the Boolean capabilities for the Wyse 50 terminal:

```
        :bs:am:

#   bs   backspace with CTRL-H
#   am   automatic margins
```

Numeric Capabilities

A numeric capability is a two-character code followed by a pound sign (#) and a value. The following example shows the numeric capabilities for the number of columns and the number of lines on a Wyse 50 terminal:

```
        :co#80:li#24:

#   co   number of columns in a line
#   li   number of lines on the screen
```

Similarly, **sg** is a numeric capability that indicates the number of character positions required on the screen for reverse video. The entry `:sg#1:` indicates that a terminal requires one additional character position when reverse video is turned ON or OFF. If you do not include a given numeric capability, 4GL assumes that the value is zero.

String Capabilities

A string capability specifies a sequence that can be used to perform a terminal operation. A string capability is a two-character code followed by an equal sign (=) and a string ending at the next colon (:) delimiter symbol.

Most **termcap** entries include string capabilities for clearing the screen, moving the cursor, and using arrow keys, underscore, function keys, and so on. The following example shows many of the string capabilities for the Wyse 50 terminal:

```
:ce=\Et:cl=\E*:\
:nd=^L:up=^K:\
:so=\EG4:se=\EG0:\
:ku=^K:kd=^J:kr=^L:kl=^H:\
:k0=^A@^M:k1=^AA^M:k2=^AB^M:k3=^AC^M:


#   ce=\Et          clear to end of line
#   cl=\E*          clear the screen
#   nd=^L           non-destructive cursor right
#   up=^K           up one line
#
#   so=\EG4         start stand-out
#   se=\EG0         end stand-out
#
#   ku=^K           up arrow key
#   kd=^J           down arrow key
#   kr=^L           right arrow key
#   kl=^H           left arrow key
#
#   k0=^A@^M        function key F1
#   k1=^AA^M        function key F2
#   k2=^AB^M        function key F3
#   k3=^AC^M        function key F4
```

Extending Function Key Definitions

4GL recognizes function keys F1 through F36. These keys correspond to the **termcap** capabilities **k0** through **k9**, followed by **kA** through **kZ**. The **termcap** entry for these capabilities is the sequence of ASCII characters your terminal sends when you press the function keys (or any other keys you choose to use as function keys). For the Wyse 50 and Televideo 950 terminals, the first eight function keys send the characters shown in the following table.

Function Key	termcap Entry
F1	`k0=^A@^M`
F2	`k1=^AA^M`
F3	`k2=^AB^M`
F4	`k3=^AC^M`
F5	`k4=^AD^M`
F6	`k5=^AE^M`
F7	`k6=^AF^M`
F8	`k7=^AG^M`

You can also define keys that correspond to the following capabilities:

- Insert line (**ki**)
- Delete line (**kj**)
- Next page (**kf**)
- Previous page (**kg**)

If these keys are defined in your **termcap** file, 4GL uses them. Otherwise, 4GL uses CONTROL-J, CONTROL-K, CONTROL-M, and CONTROL-N, respectively.

You can also use the OPTIONS statement to name other function keys or CONTROL keys for these operations.

Specifying Characters for Window Borders

4GL uses characters defined in the **termcap** file to draw the border of a window. If no characters are defined in this file, 4GL uses the hyphen (-) for horizontal lines, the pipe symbol (|) for vertical lines, and the plus sign (+) for corners.

The **termcap** file provided with 4GL contains border character definitions for many common terminals. You can look at the **termcap** file to see if the entry for your terminal has been modified to include these definitions. If your terminal entry does not contain border character definitions, or to specify alternative border characters, you or your system administrator can modify the **termcap** file.

To modify the definition for your terminal type in the termcap file

1. Determine the escape sequences for turning graphics mode on and off.

   This information is located in the manual that comes with your terminal. For example, on Wyse 50 terminals, the escape sequence for entering graphics mode is ESC H^B and the escape sequence for leaving graphics mode is ESC H^C.

   Terminals without a graphics mode do not have this escape sequence. The procedure for specifying alternative border characters on a non-graphics terminal is discussed at the end of this section.

2. Identify the ASCII equivalents for the six graphics characters that 4GL requires to draw the border.

   The ASCII equivalent of a graphics character is the key that you press in graphics mode to obtain the indicated character.

   The following table shows the graphics characters and the ASCII equivalents for a Wyse 50 terminal.

Window Border Position	Graphics Character	ASCII Equivalent
Upper-left corner	⌈	2
Lower-left corner	Î	1
Upper-right corner	˘	3
Lower-right corner	°	5
Horizontal	-	z
Vertical	|	6

   This information should be located in the manual that comes with your terminal.

3. Edit the **termcap** entry for your terminal.

You might want to make a copy of your **termcap** file before you edit it. You can use the **TERMCAP** environment variable to point to whichever copy of the **termcap** file you want to access.

Use the following format to enter values for the following **termcap** capabilities:

```
termcap-capability=value
```

The following table describes the codes for **termcap** capabilities.

Code	Description
gs	The escape sequence for entering graphics mode. In the termcap file, the ESCAPE key is represented as a backslash (\ ) followed by the letter E. The CONTROL key is represented as a caret (^). For example, the Wyse 50 escape sequence ESC-H CONTROL-B is represented as \EH^B.
ge	The escape sequence for leaving graphics mode. For example, the Wyse 50 escape sequence ESC-H CONTROL-C is represented as \EH^C.
gb	The concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border. Use the following order: Upper-left corner Lower-left corner Upper-right corner Lower-right corner Horizontal lines Vertical lines

Follow these guidelines when you insert information in the **termcap** entry:

1. Delimit entries with a colon (:).
2. End each continuing line with a backslash (\ ).
3. End the last line in the entry with a colon.

For example, if you are using a Wyse 50 terminal, add the following information in the **termcap** entry for the Wyse 50:

```
:gs=\EH^B:\      # sets gs to ESC H CTRL B
:ge=\EH^C:\      # sets ge to ESC H CTRL C
:gb=2135z6:\     # sets gb to the ASCII equivalents
                 # of graphics characters for upper
                 # left, lower left, upper right,
                 # lower right, horizontal,
                   # and vertical
```

If you prefer, you can enter this information in a linear sequence.

```
:gs=\EH^B:ge=\EH^C:gb=2135z6:\
```

If Your termcap File Contains sg#1 Capabilities

The **termcap** file for some terminals contains **sg#1** capabilities. If **sg#1** is included, 4GL reserves an additional column to the left and right of the window. If you specify a border around the 4GL window, these two columns are in addition to the two additional columns required for the border.

Terminals Without Graphics Capabilities

For terminals without graphics capabilities, you must enter a blank value for the **gs** and **ge** capabilities. For **gb**, enter the characters you want 4GL to use for the window border.

The following example shows possible values for **gs**, **ge**, and **gb** in an entry for a terminal without graphics capabilities. In this example, window borders are drawn using underscores (_) for horizontal lines, pipe symbols (|) for vertical lines, periods (.) for the top corners, and plus symbols (+) for the lower corners.

```
:gs=:ge=:gb=.|.|_|+:
```

4GL uses the graphics characters in the **termcap** file when you specify a window border in an OPEN WINDOW statement.

Adding Color and Intensity

Many of the terminal entries in the Informix **termcap** file have been modified to include color or intensity capabilities or both. (The **termcap** file is located in the **$INFORMIXDIR/etc** directory.) You can view the **termcap** file to determine if the entry for your terminal type includes these capabilities. If your terminal entry includes the **ZA** capability, your terminal is set up for color or intensity or both. If it does not, you can add color and intensity capabilities by using the information in this section. The following topics are outlined in this section:

- Color and intensity
- The **ZA** capability

- Stack operations
- Examples

Understand these topics before you modify your terminal entry.

Color and Intensity Attributes

You can write your 4GL program either for a monochrome terminal or for a color terminal and then run the program on either type of terminal. If you set up the **termcap** files as described here, the color attributes and the intensity attributes are related, as shown in the following table.

Number	Color Terminal	Monochrome Terminal
0	White	Normal
1	Yellow	Bold
2	Magenta	Bold
3	Red	Bold†
4	Cyan	Dim
5	Green	Dim
6	Blue	Dim†
7	Black	Dim

† Signifies that if the keyword BOLD is indicated as the attribute, the field will be RED on a color terminal. If the keyword DIM is indicated as the attribute, the field will be BLUE on a color terminal.

The background for colors is BLACK in all cases.

In either color or monochrome mode, you can add the REVERSE, BLINK, or UNDERLINE attribute if your terminal supports them.

The ZA String Capability

4GL uses a parameterized string capability **ZA** in the **termcap** file to determine color assignments. Unlike other **termcap** string capabilities that you set equal to a literal sequence of ASCII characters, **ZA** is a function string that depends upon four parameters.

Parameter	Value
Parameter 2 (**p2**)	Color number between `0` and `7`
Parameter 2 (**p2**)	`0` = Normal; `1` = Reverse
Parameter 3 (**p3**)	`0` = No-Blink; `1` = Blink
Parameter 4 (**p4**)	`0` = No-Underscore; `1` = Underscore

ZA uses the values of these four parameters and a stack machine to determine which characters to send to the terminal. The **ZA** function is called and these parameters are evaluated when a color or intensity attribute is encountered in a 4GL program. You can use the information in your terminal manual to set the **ZA** parameters to the correct values for your terminal.

To define the **ZA** string for your terminal, you use *stack operators* to push and pop values onto and off the *stack*. The next section describes several stack operators. Use these descriptions and the subsequent examples to understand how to define the string for your terminal.

Stack Operations

The **ZA** string uses stack operations to either push values onto the stack or pop values off the stack. Typically, the instructions in the **ZA** string push a parameter onto the stack, compare it to one or more constants, and then send an appropriate sequence of characters to the terminal. More complex operations are often necessary, and by storing the display attributes in static stack machine registers (named **a** through **z**), you can achieve terminal-specific optimizations.

A summary follows of the different stack operators you can use to write the descriptions. For a complete discussion of stack operators, consult your operating system documentation.

Operators That Send Characters to the Terminal

Use the following operators to send characters to the terminal:

`%d` pops a numeric value from the stack and sends a maximum of three digits to the terminal. For example, if the value `145` is at the top of the stack, `%d` pops the value off the stack and sends the ASCII representations of 1, 4, and 5 to the terminal. If the value `2005` is at the top of the stack, `%d` pops the value off the stack and sends the ASCII representation of 5 to the terminal.

`%2d` pops a numeric value from the stack and sends a maximum of two digits to the terminal, padding to two places. For example, if the value `145` is at the top of the stack, `%2d` pops the value off the stack and sends the ASCII representations of 4 and 5 to the terminal. If the value `5` is at the top of the stack, `%2d` pops the value off the stack and sends the ASCII representations of 0 and 5 to the terminal.

`%3d` pops a numeric value from the stack and sends a maximum of three digits to the terminal, padding to three places. For example, if the value `7` is at the top of the stack, `%3d` pops the value off the stack and sends the ASCII representations of 0, 0, and 7 to the terminal.

`%c` pops a single character from the stack and sends it to the terminal.

Operators That Manipulate the Stack

Use the following operators to manipulate the stack:

`%p[1-9]` pushes the value of the specified parameter on the stack. The notation for parameters is **p1**, **p2**, …**p9**. For example, if the value of **p1** is `3`, `%p1` pushes `3` on the stack.

`%P[a-z]` pops a value from the stack and stores it in the specified variable. The notation for variables is **Pa**, **Pb**, …**Pz**. For example, if the value `45` is on the top of the stack, `%Pb` pops `45` from the stack and stores it in the variable **Pb**.

`%g[a-z]` gets the value stored in the corresponding variable (`P[a-z]`) and pushes it on the stack. For example, if the value `45` is stored in the variable **Pb**, `%gb` gets 45 from **Pb** and pushes it on the stack.

%´c´	pushes a single character on the stack. For example, %'k' pushes k on the stack.
%{n}	pushes an integer constant on the stack. The integer can be any length and can be either positive or negative. For example, %{0} pushes the value 0 on the stack.
%S[a-z]	pops a value from the stack and stores it in the specified static variable. (Static storage is nonvolatile since the stored value remains from one attribute evaluation to the next.) The notation for static variables is **Sa**, **Sb**, …**Sz**. For example, if the value 45 is on the top of the stack, %Sb pops 45 from the stack and stores it in the static variable **Sb**. This value is accessible for the duration of the 4GL program.
%G[a-z]	gets the value stored in the corresponding static variable (S[a-z]) and pushes it on the stack. For example, if the value 45 is stored in the variable **Sb**, %Gb gets 45 from **Sb** and pushes it on the stack.

Arithmetic Operators

Each of these arithmetic operators pops the top two values from the stack, performs an operation, and pushes the result on the stack:

%+	Addition. For example, %{2}%{3}%+ is equivalent to 2+3.
%-	Subtraction. For example, %{7}%{3}%- is equivalent to 7-3.
%*	Multiplication. For example, %{6}%{3}%* is equivalent to 6*3.
%/	Integer division. For example, %{7}%{3}%/ is equivalent to 7/3 and produces a result of 2.
%m	Modulus (or remainder). For example, %{7}%{3}%m is equivalent to (7 mod 3) and produces a result of 1.

Bit Operators

The following bit operators pop the top two values from the stack, perform an operation, and push the result on the stack:

`%&` Bit-and. For example, `%{12}%{21}%&` is equivalent to (12 and 21) and produces a result of 4.

```
          Binary          Decimal

0   1   1   0   0     =    12

1   0   1   0   1     =    21

-------------------------   and

0   0   1   0   0       =     4
```

`%|` Bit-or. For example, `%{12}%{21}%|` is equivalent to (12 or 21) and produces a result of 29.

```
          Binary          Decimal

0   1   1   0   0     =    12

1   0   1   0   1     =    21

-------------------------   or

1   1   1   0   1       =    29
```

`%^` Exclusive-or. For example, `%{12}%{21}%^` is equivalent to (12 exclusive-or 21) and produces a result of 25.

```
          Binary          Decimal

0   1   1   0   0     =    12

1   0   1   0   1     =    21

-------------------------  exclusive or

1   1   0   0   1       =    25
```

The following unary operator pops the top value from the stack, performs an operation, and pushes the result on the stack:

%~ Bitwise complement. For example, %{25}%~ results in a value of -26, as shown in the following display.

```
          Binary                    Decimal

0  0  0  1   1   0   0   1         =  25

---------------------------        Complement

1  1  1  0   0   1   1   0         =  -26
```

Logical Operators

The following logical operators pop the top two values from the stack, perform an operation, and push the logical result (either `0` for FALSE or `1` for TRUE) on the stack:

%= Equal to. For example, if the parameter **p1** has the value `3`, the expression `%p1%{2}%=` is equivalent to 3=2 and produces a result of `0` (FALSE).

%> Greater than. For example, if the parameter **p1** has the value `3`, the expression `%p1%{0}%>` is equivalent to 3>0 and produces a result of `1` (TRUE).

%< Less than. For example, if the parameter p1 has the value `3`, the expression `%p1%{4}%<` is equivalent to 3<4 and produces a result of `1` (TRUE).

The following unary operator pops the top value from the stack, performs an operation, and pushes the logical result (either `0` or `1`) on the stack.

%! Logical negation. This operator produces a value of zero for all nonzero numbers and a value of `1` for zero. For example, `%{2}%!` results in a value of `0`, and `%{0}%!` results in a value of `1`.

Conditional Statements

The condition statement IF-THEN-ELSE has the following format:

```
%? expr %t thenpart %e elsepart %;
```

The %e *elsepart* is optional. You can nest conditional statements in the *thenpart* or the *elsepart*.

When 4GL evaluates a conditional statement, it pops the top value from the stack and evaluates it as either `true` or `false`. If the value is `true`, 4GL performs the operations after the %t; otherwise it performs the operations after the %e (if any).

For example, the expression:

```
%?%p1%{3}%=%t;31%;
```

is equivalent to:

```
if p1 = 3 then print ";31"
```

Assuming that p1 has the value 3, 4GL performs the following actions:

- %? does not perform an operation but is included to make the conditional statement easier to read.
- %p1 pushes the value of **p1** on the stack.
- %{3} pushes the value 3 on the stack.
- %= pops the value of **p1** and the value 3 from the stack, evaluates the Boolean expression p1=3, and pushes the resulting value 1 (TRUE) on the stack.
- %t pops the value from the stack, evaluates 1 as TRUE, and executes the operations after %t. (Because ";31" is not a stack machine operation, 4GL prints ";31" to the terminal.)
- %; terminates the conditional statement.

Summary of Operators

The following table summarizes the allowed operations.

Operation	Description
%d	Writes **pop()** in decimal format
%2d	Writes **pop()** in two-place decimal format
%3d	Writes **pop()** in three-place decimal format
%c	Writes **pop()** as a single character
%p[1-9]	Pushes *i*th parameter
%P[a-z]	Pops and stores a variable
%g[a-z]	Gets a variable and pushes it on the stack
%'c'	Pushes a CHAR constant
%{n}	Pushes an integer constant
%S[a-z]	Pops and stores a static variable
%G[a-z]	Gets a static variable and pushes
%+	Addition. Pops two values off the stack, adds them together, and pushes the result onto the stack
%-	Subtraction. Pops two values off the stack, subtracts one from the other, and pushes the result onto the stack
%*	Multiplication. Pops two values off the stack, multiplies them by each other, and pushes the result onto the stack
%/	Integer division. Pops two values off the stack, divides one by the other, and pushes the result onto the stack
%m	Modulus. Pops two values off the stack and pushes the remainder when one number is divided by the other
%&	Bitwise AND
%|	Bitwise OR
%^	Bitwise exclusive OR

(1 of 2)

Operation	Description
%~	Bitwise complement
%=	Equal to. Pops two values off the stack, finds out whether the values are equal, and pushes `1` onto the stack if the values are equal or `0` if they are not equal
%>	Greater than. Pushes two values off the stack, finds out whether the first value is greater than the second, and pushes the result onto the stack
%<	Less than. Pushes two values off the stack, finds out whether the first value is less than the second, and pushes the result onto the stack
%!	Logical negation. Pops one value off the stack, logically inverts or negates it (converts zero to 1 and nonzero to 0), and pushes the negated value back onto the stack
%?	*expr* %t *thenpart* %e *elsepart* %; IF *expr* THEN *thenpart* ELSE *elsepart*; the %e *elsepart* is optional. ELSE-IF's are possible (c's are conditions): %? c1 %t…%e c2 %t…%e c3 %t…%e…%; Nested IF's allowed
All other characters are written to the terminal; use '%%' to write '%'.	

(2 of 2)

Examples

To illustrate, consider the monochrome Wyse terminal. The following table shows the escape sequences for some display characteristics.

Escape Sequence	Results
ESC G 0	Normal
ESC G 1	Blank (invisible)
ESC G 2	Blink

(1 of 2)

Escape Sequence	Results
ESC G 4	Reverse
ESC G 5	Reverse and blank
ESC G 6	Reverse and blink
ESC G 8	Underscore
ESC G 9	Underscore and blank
ESC G :	Underscore and blink
ESC G <	Underscore and reverse
ESC G =	Underscore, reverse, and blank
ESC G >	Underscore, reverse, and blink

(2 of 2)

The characters after G form an ASCII sequence from the character 0 (zero) through ?. You can generate the character by starting with 0 and adding 1 for blank, 2 for blink, 4 for reverse, and 8 for underline.

You can construct the **termcap** entry in stages, as outlined in the following display. %pi refers to pushing the i^{th} parameter on the stack. The designation for is \E. The **termcap** entry for the Wyse terminal must contain the following **ZA** entry for 4GL monochrome attributes such as REVERSE and BOLD to work correctly:

```
ZA =
EG                      #print EG
%'0'                    #push '0' (normal) on the stack
%?%p1%{7}%=%t%{1}%|     #if p1 = 7 (invisible), set
                        #the 1 bit (blank);
%e%p1%{3}%>             #if p1 > 3 and < 7, set the 64 flag (dim);
  %p1%{7}%<%&%t%{64}%|  #
     %;%;               #
%?%p2%t%{4}%|%;         #if p2 is set, set the 4 bit (reverse)
%?%p3%t%{2}%|%;         #if p3 is set, set the 2 bit (blink)
%?%p4%t%{8}%|%;         #if p4 is set, set the 8 bit (underline)
%c:                     #print whatever character
                        #is on top of the stack
```

You then concatenate these lines as a single string that ends with a colon and has no embedded NEWLINE characters. The actual **ZA** entry for the Wyse 50 terminal follows:

```
ZA = \EG%'0'%?%p1%{7}%=%t%{1}%|%e%p1%{3}%>%p1%{7}%<%&%t%{64}
%|%;%;%?%p2%t%{4}%|%;%?%p3%t%{2}%|%;%?%p4%t%{8}%|%;%c:
```

The next example is for the ID Systems Corporation ID231, a color terminal. On this terminal, to set color and other characteristics, you must enclose a character sequence between a lead-in sequence (`ESC [ 0`) and a terminating character (`m`). The first in the sequence is a two-digit number that determines whether the assigned color is in the background (30) or in the foreground (40). The next is another two-digit number that is the other of 30 or 40, incremented by the color number. These characters are followed by 5 if there is blinking and by 4 for underlining.

The code in the following example sets up the entire escape sequence:

```
ZA =
\E[0;                    #print lead-in
%?%p1%{0}%=%t%{7}        #encode color number (translate
%e%p1%{1}%=%t%{3}        #   from  to the number
%e%p1%{2}%=%t%{5}        #   for the ID231)
%e%p1%{3}%=%t%{1}        #
%e%p1%{4}%=%t%{6}        #
%e%p1%{5}%=%t%{2}        #
%e%p1%{6}%=%t%{4}        #
%e%p1%{7}%=%t%{0}%;      #
%?%p2%t30;%{40}%+%2d     #if p2 is set, print '30' and
                         #  '40' + color number (reverse)
%e40;%{30}%+%2d%;        #  else print '40' and
                         #  '30' + color number (normal)
%?%p3%t;5%;              #if p3 is set, print 5 (blink)
%?%p4%t;4%;              #if p4 is set, print 4 (underline)
m                        #print 'm' to end character
                         #  sequence
```

When you concatenate these strings, the **termcap** entry is as follows:

```
ZA =\E[0;%?%p1%{0}%=%t%{7}%e%p1%{1}%=%t%{3}%e%p1%{2}%=
%t%{5}%e%p1%{3}%=%t%{1}%e%p1%{4}%=%t%{6}%e%p1%{5}%=%t%
{2}%e%p1%{6}%=%t%{4}%e%p1%{7}%=%t%{0}%;%?%p2%t30;%{40}
%+%2d%e40;%{30}%+%2d%;%?%p3%t;5%;%?%p4%t;4%;m
```

In addition to the **ZA** capability, you can use other **termcap** capabilities. **ZG** is the number of character positions on the screen occupied by the attributes of **ZA**. Like the **sg** numeric capability, **ZG** is not required if no extra character positions are needed for display attributes. The value for the **ZG** entry is usually the same value as for the **sg** entry.

terminfo

If you have set the **INFORMIXTERM** environment variable to **terminfo**, 4GL uses the **terminfo** directory indicated by the **TERMINFO** environment variable (or **/usr/lib/terminfo** if **TERMINFO** is not set). 4GL uses the information in **terminfo** to draw window borders, define function keys, and display certain intensity attributes.

You might want to modify a file in the **terminfo** directory in the following instances:

- To extend function key definitions
- To specify or change the graphics characters used for window borders
- To customize your terminal entry in other ways

If you use **terminfo** (instead of **termcap**), you cannot use color attributes with 4GL. To use color attributes with 4GL, you must use **termcap**.

Some terminals cannot support graphics characters. Read this appendix and the user guide that comes with your terminal to determine whether or not the changes described in this appendix are applicable to your terminal.

To modify a **terminfo** file, you need to be familiar with the following information:

- The format of **terminfo** entries
- The **infocmp** program
- The **tic** program

This information is summarized in this appendix; however, refer to your operating system documentation for a complete discussion.

Format of a terminfo Entry

The **terminfo** directory contains a file for each terminal name that is defined. Each file contains a compiled **terminfo** entry for that terminal. This section describes the general format of **terminfo** entries. For a complete description of **terminfo**, refer to your operating system documentation.

A **terminfo** entry contains a list of names for the terminal, followed by a list of the terminal capabilities. There are three types of capabilities:

- Boolean capabilities
- Numeric capabilities
- String capabilities

All **terminfo** entries have the following format:

- ESCAPE is specified as a backslash (\ ) followed by the letter E, and CONTROL is specified as a caret (^). Do not attempt to use the ESCAPE or CONTROL key to indicate escape sequences or control characters in a **terminfo** entry.
- Each capability, including the last one in the entry, is followed by a comma as a delimiter.

The following example shows a basic **terminfo** entry for the Wyse 50 terminal:

```
. Entry for Wyse 50:

w5|wy50|wyse50,
  am, cols#80, lines#24, cuul=^K, clear=^Z,
  home=^^, cuf1=^L, cup=\E=%p1%'\s'%+%c%p2%'\s'%+%c,
  bw, ul, bel=^G, cr=\r, cud1=\n, cub1=\b, kpb=\b, kcudl=\n,
  kdub1=\b, nel=\r\n, ind=\n,
  xmc#1, cbt=\EI,
```

Comment lines begin with a period (.).

Terminal Names

A **terminfo** entry starts with one or more names for the terminal, each separated by a pipe symbol (|). For example, the **terminfo** entry for the Wyse 50 terminal starts with the following line:

```
w5|wy50|wyse50,
```

The **terminfo** entry can be accessed by using any one of these names.

Boolean Capabilities

A Boolean capability is a two- to five-character code that indicates whether or not a terminal has a specific feature. If the Boolean capability is present in the **terminfo** entry, the terminal has that particular feature.

The next example shows some of the Boolean capabilities for the Wyse 50:

```
        bw,am,

.   bw   backward wrap
.   am   automatic margins
```

Numeric Capabilities

A numeric capability is a 2- to 5-character code followed by a pound sign (#) and a value. The following example shows the numeric capabilities for the number of columns and the number of lines on a Wyse 50 terminal:

```
        cols#80,lines#24,

.   cols    number of columns in a line
.   lines   number of lines on the screen
```

String Capabilities

A string capability specifies a sequence that can be used to perform a terminal operation. A string capability is a two- to five-character code followed by an equal (=) sign and a string ending at the next comma (,) delimiter.

Most **terminfo** entries include string capabilities for clearing the screen, cursor movement, arrow keys, underscore, function keys, and so on. The following example shows many of the string capabilities for the Wyse 50 terminal:

```
        el=\ET,clear=E*,
        cuf1=^L,cuu1=^K,
        smso=\EG4,rmso=\EG0,
        kcuu1=^K,kcud1=^J,kcuf1=^L,kcub1=^H,
        kf0=^A@^M,kf1=^AA^M,kf2=^AB^M,kf3=^AC^M,

.   el=\Et          clear to end of line
.   clear=\E*       clear the screen
.   cufl=^L         non-destructive cursor right
.   cuul=^K         up one line
.
.   smso=\EG4       start stand-out
.   rmso=\EG0       end stand-out
.
.   kcuul=^K        up arrow key
.   kcudl=^J        down arrow key
.   kcufl=^L        right arrow key
.   kcubl=^H        left arrow key
.
.   kf0=^A@^M       function key F1
.   kf1=^AA^M       function key F2
.   kf2=^AB^M       function key F3
.   kf3=^AC^M       function key F4
```

Extending Function Key Definitions

4GL recognizes function keys F1 through F36. These keys correspond to the **terminfo** capabilities **kf0** through **kf36**. The **terminfo** entry for these capabilities is the sequence of ASCII characters that your terminal sends when you press the function keys (or any other keys you choose to use as function keys). For the Wyse 50 and Televideo 950 terminals, the first eight function keys send the characters shown in the following table.

Function Key	terminfo Entry
F1	`kf0=^A@^M`
F2	`kf1=^AA^M`
F3	`kf2=^AB^M`
F4	`kf3=^AC^M`
F5	`kf4=^AD^M`
F6	`kf5=^AE^M`
F7	`kf6=^AF^M`
F8	`kf7=^AG^M`

You can also define keys that correspond to the following capabilities:

- Insert line (**kill**)
- Delete line (**kdll**)
- Next page (**knp**)
- Previous page (**kpp**)

If these keys are defined in your **terminfo** file, 4GL uses them. Otherwise, 4GL uses CONTROL-J, CONTROL-K, CONTROL-M, and CONTROL-N, respectively.

You can also use the OPTIONS statement to assign other function keys or CONTROL keys for these operations.

Specifying Characters for Window Borders

4GL uses characters defined in the **terminfo** files to draw the border of a window. If no characters are defined in this file, INFORMIX-4GL uses the hyphen (-) for horizontal lines, the pipe symbol (|) for vertical lines, and the plus sign (+) for corners.

You can look at the **terminfo** source file (using **infocmp**) to see if the entry for your terminal includes these definitions. (Look for the **acsc** capability, described later in this section.) If the file for your terminal does not contain border character definitions, or to specify alternative border characters, you or your system administrator can modify the **terminfo** source file. You can refer to your operating system documentation for a complete description of how to decompile **terminfo** entries by using the **infocmp** program.

To specify border characters in the terminfo source file

1. Determine the escape sequences for turning graphics mode on and off.

   This information is located in the manual that comes with your terminal. For example, on Wyse 50 terminals, the escape sequence for entering graphics mode is ESC H^B and the escape sequence for leaving graphics mode is ESC H^C.

   Terminals without a graphics mode do not have this escape sequence. The procedure for specifying alternative border characters on a non-graphics terminal is discussed at the end of this section.

2. Identify the ASCII equivalents for the six graphics characters that 4GL requires to draw the border.

   The ASCII equivalent of a graphics character is the key that the user presses in graphics mode to obtain the indicated character.

   The following table shows the graphics characters and the ASCII equivalents for a Wyse 50 terminal.

Window Border Position	Graphics Character	ASCII Equivalent
Lower-left corner	└	1
Upper-right corner	ˇ	3
Lower-right corner	°	5
Horizontal	-	z
Vertical	|	6

   This information should be located in the manual that comes with your terminal.

3. Edit the **terminfo** source file for your terminal.

a. You can decompile it by using **infocmp** redirected to a file.

b. You might want to make a copy of your **terminfo** directory before you edit files.

You can use the **TERMINFO** environment variable to point to whichever copy of the **terminfo** directory you want to access.

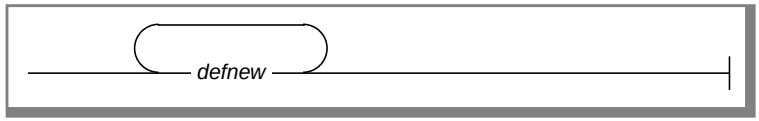
c. Use the following format to enter values for **terminfo** capabilities:

```
terminfo-capability=value
```

Enter values for the following **terminfo** capabilities.

Code	Description
smacs	The escape sequence for entering graphics mode. In a **terminfo** file, ESCAPE is represented as a backslash (\ ) followed by the letter *E*; CONTROL is represented as a caret (^). For example, the Wyse 50 escape sequence ESC-H CTRL-B is represented as \EH^B.
rmacs	The escape sequence for leaving graphics mode. For example, the Wyse 50 escape sequence ESC-H CTRL-C is represented as \EH^C.
acsc	The concatenated, paired list of ASCII equivalents for the six graphics characters used to draw the border. You can specify the characters in any order, but you must pair the ASCII equivalents for your terminal with the following system default characters: `l` for the upper-left corner `m` for the lower-left corner `k` for the upper-right corner `j` for the lower-right corner `q` for horizontal lines `x` for vertical lines

Use the following format to specify the **acsc** value.



Element	Description
def	is the default character for a particular border character and
new	is the equivalent on that terminal for the same border character.

For example, on the Wyse 50 terminal, given the ASCII equivalents for window border position and the system default characters for border position, the **acsc** capability is set as shown:

```
acsc=l2m1k3j5qzx6
```

4. Use **tic** to recompile the modified **terminfo** file.

   See your operating system documentation for a description of the **tic** program.

   The following example shows the full setting for specifying alternative border characters on the Wyse 50:

```
smacs=\EH^B,          . sets smacs to ESC H CTRL B
rmacs=\EH^C,          . sets rmacs to ESC H CTRL C
acsc=l2m1k3j5qzx6,    . sets acsc to the ASCII equivalents
                      . of graphics characters for upper
                      . left (l), lower left (m), upper rig
ht (k),
                      . lower right (j), horizontal (q),
                         . and vertical (x)
```

   If you prefer, you can enter this information in a linear sequence.

```
smacs=\EH^B,rmacs=\EH^C,acsc=l2m1k3j5qzx6,
```

If Your terminfo File Contains xmc#1 Capabilities

The **terminfo** file for some terminals contains **xmc#1** capabilities. If **xmc#1** is included, 4GL reserves an additional column to the left and right of the window. If you specify a border around the 4GL window, these two columns are in addition to the two additional columns required for the border.

Terminals Without Graphics Capabilities

For terminals without graphics capabilities, you must enter a blank value for the **smacs** and **rmacs** capabilities. For **acsc**, enter the characters that you want 4GL to use for the window border.

The following example shows possible values for **smacs**, **rmacs**, and **acsc** in an entry for a terminal without graphics capabilities. In this example, window borders are drawn using underscores (_) for horizontal lines, pipe symbols (|) for vertical lines, periods (.) for the top corners, and pipe symbols (|) for the lower corners.

```
smacs=,rmacs=,acsc=l.m|k.j|q_x|,
```

4GL uses the graphics characters in the **terminfo** file when you specify a window border in an OPEN WINDOW statement.

Color and Intensity

If you use **terminfo**, you cannot use color or the following intensity attributes in your 4GL programs:

```
BOLD
DIM
INVISIBLE
BLINK
```

If you specify these attributes in your 4GL code, they are ignored.

If the **terminfo** entry for your terminal contains the **ul** and **so** attributes, you can use the UNDERLINE and REVERSE intensity attributes. You can see if your **terminfo** entry includes these capabilities by using the **infocmp** program. Refer to your operating system documentation for information about **infocmp**.

To use color and intensity in your 4GL programs, you must use **termcap** (by setting the **INFORMIXTERM** environment variable to **termcap**, and by setting the **TERMCAP** environment variable to **$INFORMIXDIR/etc/termcap**). For more information, see Appendix D, “Environment Variables.”

Appendix

G

Reserved Words

INFORMIX-4GL has no reserved words in the sense of a string that obeys the rules for identifiers but that always produces a compilation error.

This appendix, however, lists keywords that you should not use as programmer-defined identifiers in a 4GL application. If you do, the program might fail with a compilation or runtime error, or produce unexpected results. (If you receive an error message that seems to be unrelated to the statement that produced the error, review this appendix to see if the error was caused by a reserved word used as an identifier.)

In general, you cannot use as an identifier the name of a built-in constant or variable or the name of an operator that can begin an expression. Chapter 5, "Built-In Functions and Operators," describes restricted functionality that results if you declare a 4GL identifier with the same name as a built-in function or operator.

You are not prevented from declaring most other keywords of 4GL as identifiers, but you might not be able to reference such identifiers in contexts where the same keyword makes sense. For example, if you open a 4GL window named **screen**, you will not be able to reference it in statements like CURRENT WINDOW, where the SCREEN keyword specifies the 4GL screen. Similarly, if ASCII is declared as the name of a variable, you cannot use it in contexts where an expression is valid, because the ASCII operator has different semantics but takes precedence over a variable.

Do not declare function names that occur in ESQL/C or in the standard C or POSIX libraries, in the **fglusr.h** or **fglsys.h** header files, or that start with `fgl`. If you avoid these names, you will not run into problems.

Reserved Words of 4GL

Words in the following list are reserved in the sense that they are not meaningful as the names of variables. Do not declare any of these words as 4GL identifiers.

AND	FIELD_TOUCHED	NULL
ASCII	GET_FLDBUF	ORD
AVERAGE	INTERVAL	OR
AVG	INT_FLAG	PAGENO
CHAR_LENGTH	LENGTH	PERCENT
COLUMN	LINENO	QUIT_FLAG
CONSTANT	MAX	SQLCA
COUNT	MDY	STATUS
COPY	MIN	SUM
CURRENT	MOD	TIME
DATE	MONTH	TODAY
DATETIME	NEW	TRUE
DAY	NOT	WEEKDAY
EXTEND	NOTFOUND	WORDWRAP
FALSE	NOW	YEAR

In addition to these words, do not declare the names of operating system calls or C or C++ language keywords as identifiers in 4GL programs. For a list of these words, see the documentation for your C or C++ compiler and the documentation for your implementation of UNIX or LINUX.

Apart from the risk of unexpected behavior or errors, your 4GL code is likely to be difficult to read and to maintain if you use keywords as identifiers.

ANSI

Reserved Words of ANSI SQL

As in the case of the 4GL reserved words, declaring any of the ANSI reserved words of SQL as an identifier can sometimes result in runtime errors if your 4GL application accesses a database that is ANSI-compliant.

If the default database when you compile is ANSI compliant, 4GL issues a warning if you use an ANSI reserved word as an identifier in an embedded SQL statement or other 4GL statement, and either of the following is true:

- The **DBANSIWARN** environment variable is set.
- You specify the **-ansi** command-line flag.

The ANSI SQL-92 reserved words are listed in the following table.

AGGREGATE	DELETE	ITEM	RETAIN
ALL	DESC	JOIN	ROLLBACK
ALL_ROWS	DISTINCT	LANGUAGE	SCHEMA
AND	DOUBLE	LEFT	SECTION
ANY	END	LIKE	SELCONST
AS	ESCAPE	LOCKS	SELECT
ASC	EXEC	MAX	SET
AVG	EXISTS	MEMORY_RESIDENT	SMALLINT
ALL_ROWS	FETCH	MIN	SOME
AND	FIRST_ROWS	MODULE	SQL
ANY	FLOAT	NON_RESIDENT	SQLCODE
CACHE	FOR	NOT	SQLERROR
CASE	FORTRAN	NULL	SUBSTR
CHAR	FOUND	NUMERIC	SUBSTRING
CHARACTER	FROM	NVL	SUM
CHECK	DELETE	OF	TABLE
CLOSE	DESC	ON	TO
COBOL	GO	OPEN	UNION
COMMIT	GOTO	OPTION	UNIQUE
CONTINUE	GRANT	OR	UPDATE
CRCOLS	GROUP	ORDER	USER
COUNT	HAVING	OUT	VALUES
CRCOLS	IN	PASCAL	VIEW
CREATE	INDICATOR	PLI	WHENEVER
CURRENT	INNER	PRECISION	WHERE
CURSOR	INSERT	PRIVILEGES	WITH
DEC	INT	PROCEDURE	WORK
DECIMAL	INTEGER	PUBLIC	
DECLARE	INTO	REAL	
DECODE	IS	REPLICATION	

If you compare this list with the shorter one for 4GL on the previous page, you can see that some words (for example, AND, AVG, COUNT, and CURRENT) are reserved both by 4GL and by the ANSI/ISO Entry Level standard for SQL.

The draft ANSI SQL3 standard lists additional reserved words, such as TYPE, which Informix database servers later than Version 7.3 recognize as reserved words. See the *Informix Guide to SQL: Tutorial* for more information about using the keywords of SQL as identifiers in SQL statements.

Appendix

H

The Demonstration Application

This appendix contains the form specifications, INFORMIX-4GL source code modules, and help message source code for the **demo4.4ge** demonstration application.

The **demo4.4ge** application documented here is not meant to be a complete application as some of the functions called by the menus are merely placeholders and have not been implemented.

File Name	Description
custform.per	Form for displaying customer information
orderform.per	Form for entering an order
state_list.per	Form for displaying a list of states
stock_sel.per	Form for displaying a list of stock items
d4_globals.4gl	Module containing global definitions
d4_main.4gl	Module that contains MAIN routine
d4_cust.4gl	Module that handles the **Customer** option
d4_orders.4gl	Module that handles the **Orders** option
d4_stock.4gl	Module that handles the **Stock** option
d4_report.4gl	Module that handles the **Report** option
d4_demo.4gl	Module that handles a hidden source code option
helpdemo.src	Source file for help messages

custform.per

```
DATABASE stores7

SCREEN
{


                            Customer Form

        Number      :[f000       ]
        Owner Name  :[f001           ][f002           ]
        Company     :[f003                ]
        Address     :[f04                ]
                     [f005                ]
        City        :[f006           ] State:[a0] Zipcode:[f007 ]
        Telephone   :[f008              ]

}

TABLES
customer

ATTRIBUTES
f000 = customer.customer_num, NOENTRY;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state, UPSHIFT;
f007 = customer.zipcode;
f008 = customer.phone, PICTURE = "###-###-#### XXXXX";
```

orderform.per

```
DATABASE stores7

SCREEN
{
 ------------------------------------------------------------------------------
                              ORDER FORM
 ------------------------------------------------------------------------------
 Customer Number:[f000       ]  Contact Name:[f001           ][f002       ]
    Company Name:[f003                ]
         Address:[f004                ][f005               ]
            City:[f006           ] State:[a0] Zip Code:[f007 ]
       Telephone:[f008              ]
 ------------------------------------------------------------------------------
 Order No:[f009       ]    Order Date:[f010      ]    PO Number:[f011
        Shipping Instructions:[f012                                        ]
 ------------------------------------------------------------------------------
 Item No.  Stock No.  Code    Description    Quantity    Price       Total
 [f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018   ]
 [f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018   ]
 [f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018   ]
 [f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018   ]
                Running Total including Tax and Shipping Charges:[f019   ]
 ==============================================================================
}

TABLES
customer orders items stock

ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state, UPSHIFT;
f007 = customer.zipcode;
f008 = customer.phone, PICTURE = "###-###-#### XXXXX";

f009 = orders.order_num;
f010 = orders.order_date, DEFAULT = TODAY;
f011 = orders.po_num;
f012 = orders.ship_instruct;

f013 = items.item_num, NOENTRY;
f014 = items.stock_num;
a1 = items.manu_code, UPSHIFT;
f015 = stock.description, NOENTRY;
f016 = items.quantity;
f017 = stock.unit_price, NOENTRY;
f018 = items.total_price, NOENTRY;
f019 = formonly.t_price TYPE MONEY;

INSTRUCTIONS
SCREEN RECORD s_items[4](items.item_num, items.stock_num, items.manu_code,
    stock.description, items.quantity, stock.unit_price, items.total_price)
```

state_list.per

```
DATABASE stores7

SCREEN
{
   State Selection

[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
}

TABLES
state

ATTRIBUTES
a0 = state.code;
f000 = state.sname;

INSTRUCTIONS
DELIMITERS "  "
SCREEN RECORD s_state[7](state.*)
```

stock_sel.per

```
DATABASE stores7

SCREEN
{
  [f018][f019][f020           ][f021           ][f022     ][f023          ]

  [f018][f019][f020           ][f021           ][f022     ][f023          ]
  [f018][f019][f020           ][f021           ][f022     ][f023          ]
}

TABLES
stock

ATTRIBUTES
f018 = FORMONLY.stock_num;
f019 = FORMONLY.manu_code;
f020 = FORMONLY.manu_name;
f021 = FORMONLY.description;
f022 = FORMONLY.unit_price;
f023 = FORMONLY.unit_descr;

INSTRUCTIONS
DELIMITERS "  "
SCREEN RECORD s_stock[3] (FORMONLY.stock_num THRU FORMONLY.unit_descr)
```

d4_globals.4gl

```
DATABASE stores7

GLOBALS
   DEFINE
      p_customer RECORD LIKE customer.*,
      p_orders RECORD
            order_num LIKE orders.order_num,
            order_date LIKE orders.order_date,
            po_num LIKE orders.po_num,
            ship_instruct LIKE orders.ship_instruct
         END RECORD,
      p_items ARRAY[10] OF RECORD
            item_num LIKE items.item_num,
            stock_num LIKE items.stock_num,
            manu_code LIKE items.manu_code,
            description LIKE stock.description,
            quantity LIKE items.quantity,
            unit_price LIKE stock.unit_price,
            total_price LIKE items.total_price
         END RECORD,
      p_stock ARRAY[30] OF RECORD
            stock_num LIKE stock.stock_num,
            manu_code LIKE manufact.manu_code,
            manu_name LIKE manufact.manu_name,
            description LIKE stock.description,
            unit_price LIKE stock.unit_price,
            unit_descr LIKE stock.unit_descr
         END RECORD,
      p_state ARRAY[50] OF RECORD LIKE state.*,
      state_cnt, stock_cnt INTEGER,
      print_option CHAR(1)
END GLOBALS
```

d4_main.4gl

```
GLOBALS
   "d4_globals.4gl"


MAIN

   DEFER INTERRUPT
   OPTIONS
   HELP FILE "helpdemo"
   LET print_option = "s"
   CALL get_states()
   CALL get_stocks()

   CALL ring_menu()
   MENU "MAIN"
      COMMAND "Customer" "Enter and maintain customer data" HELP 101
         CALL customer()
         CALL ring_menu()
      COMMAND "Orders" "Enter and maintain orders" HELP 102
         CALL orders()
         CALL ring_menu()
      COMMAND "Stock" "Enter and maintain stock list" HELP 103
         CALL stock()
         CALL ring_menu()
      COMMAND "Reports" "Print reports and mailing labels" HELP 104
         CALL reports()
         CALL ring_menu()
      COMMAND key("!")
         CALL bang()
         CALL ring_menu()
         NEXT OPTION "Customer"
      COMMAND key("X")
         CALL demo()
         CALL ring_menu()
         NEXT OPTION "Customer"
      COMMAND "Exit" "Exit program and return to operating system" HELP 105
         CLEAR SCREEN
         EXIT PROGRAM
   END MENU
END MAIN


FUNCTION bang()
   DEFINE cmd CHAR(80),
      x CHAR(1)

   CALL clear_menu()
   LET x = "!"
   WHILE x = "!"
      PROMPT "!" FOR cmd
      RUN cmd
      PROMPT "Type RETURN to continue." FOR CHAR x
   END WHILE
END FUNCTION
```

```
FUNCTION mess(str, mrow)
   DEFINE str CHAR(80),
      mrow SMALLINT

   DISPLAY " ", str CLIPPED AT mrow,1
   SLEEP 3
   DISPLAY "" AT mrow,1
END FUNCTION


FUNCTION ring_menu()

   DISPLAY "---------------------------------------- ",
      "Type Control-W for MENU HELP -------" AT 4,2 ATTRIBUTE(MAGENTA)
END FUNCTION


FUNCTION clear_menu()

   DISPLAY "" AT 1,1
   DISPLAY "" AT 2,1
END FUNCTION


FUNCTION get_states()

   DECLARE c_state CURSOR FOR
      SELECT * FROM state
      ORDER BY sname
   LET state_cnt = 1
   FOREACH c_state INTO p_state[state_cnt].*
      LET state_cnt = state_cnt + 1
      IF state_cnt > 50 THEN
         EXIT FOREACH
      END IF
   END FOREACH
   LET state_cnt = state_cnt - 1
END FUNCTION


FUNCTION get_stocks()

   DECLARE stock_list CURSOR FOR
      SELECT stock_num, manufact.manu_code,
         manu_name, description, unit_price, unit_descr
      FROM stock, manufact
      WHERE stock.manu_code = manufact.manu_code
      ORDER BY stock_num
   LET stock_cnt = 1
   FOREACH stock_list INTO p_stock[stock_cnt].*
      LET stock_cnt = stock_cnt + 1
      IF stock_cnt > 30 THEN
         EXIT FOREACH
      END IF
   END FOREACH
   LET stock_cnt = stock_cnt - 1
END FUNCTION
```

d4_cust.4gl

```
GLOBALS
    "d4_globals.4gl"

FUNCTION customer()

   OPTIONS
      FORM LINE 7
   OPEN FORM customer FROM "custform"
   DISPLAY FORM customer
      ATTRIBUTE(MAGENTA)
   CALL ring_menu()
   CALL fgl_drawbox(3,30,3,43)
   CALL fgl_drawbox(3,61,8,7)
   CALL fgl_drawbox(11,61,8,7)
   LET p_customer.customer_num = NULL
   MENU "CUSTOMER"
      COMMAND "One-add" "Add a new customer to the database" HELP 201
         CALL add_customer(FALSE)
      COMMAND "Many-add" "Add several new customer to database" HELP 202
         CALL add_customer(TRUE)
      COMMAND "Find-cust" "Look up specific customer" HELP 203
         CALL query_customer(23)
         IF p_customer.customer_num IS NOT NULL THEN
            NEXT OPTION "Update-cust"
         END IF
      COMMAND "Update-cust" "Modify current customer information" HELP 204
         CALL update_customer()
         NEXT OPTION "Find-cust"
      COMMAND "Delete-cust" "Remove a customer from database" HELP 205
         CALL delete_customer()
         NEXT OPTION "Find-cust"
      COMMAND "Exit" "Return to MAIN Menu" HELP 206
         CLEAR SCREEN
         EXIT MENU
   END MENU
   OPTIONS
      FORM LINE 3
END FUNCTION


FUNCTION add_customer(repeat)
   DEFINE repeat INTEGER

   CALL clear_menu()
   MESSAGE "Press F1 or CTRL-F for field help; ",
      "F2 or CTRL-Z to return to menu"
   IF repeat THEN
      WHILE input_cust()
         ERROR "Customer data entered" ATTRIBUTE (GREEN)
      END WHILE
      CALL mess("Multiple insert completed -
 current screen values ignored", 23)
   ELSE
      IF input_cust() THEN
         ERROR "Customer data entered" ATTRIBUTE (GREEN)
      ELSE
         CLEAR FORM
         LET p_customer.customer_num = NULL
```

```
            ERROR "Customer addition aborted" ATTRIBUTE (RED, REVERSE)
         END IF
      END IF
   END FUNCTION
   FUNCTION input_cust()

      DISPLAY "Press ESC to enter new customer data" AT 1,1
      INPUT BY NAME p_customer.*
         AFTER FIELD state
            CALL statehelp()
            DISPLAY "Press ESC to enter new customer data", "" AT 1,1
         ON KEY (F1, CONTROL-F)
            CALL customer_help()
         ON KEY (F2, CONTROL-Z)
            LET int_flag = TRUE
            EXIT INPUT
      END INPUT
      IF int_flag THEN
         LET int_flag = FALSE
         RETURN(FALSE)
      END IF
      LET p_customer.customer_num = 0
      INSERT INTO customer VALUES (p_customer.*)
      LET p_customer.customer_num = SQLCA.SQLERRD[2]
      DISPLAY BY NAME p_customer.customer_num ATTRIBUTE(MAGENTA)
      RETURN(TRUE)
   END FUNCTION


   FUNCTION query_customer(mrow)
      DEFINE where_part CHAR(200),
         query_text CHAR(250),
         answer CHAR(1),
         mrow, chosen, exist SMALLINT

      CLEAR FORM
      CALL clear_menu()

      MESSAGE "Enter criteria for selection"
      CONSTRUCT where_part ON customer.* FROM customer.*
      MESSAGE ""
      IF int_flag THEN
         LET int_flag = FALSE
         CLEAR FORM
         ERROR "Customer query aborted" ATTRIBUTE(RED, REVERSE)
         LET p_customer.customer_num = NULL
         RETURN (p_customer.customer_num)
      END IF
      LET query_text = "select * from customer where ", where_part CLIPPED,
         " order by lname"
      PREPARE statement_1 FROM query_text
      DECLARE customer_set SCROLL CURSOR FOR statement_1
      OPEN customer_set
      FETCH FIRST customer_set INTO p_customer.*
      IF status = NOTFOUND THEN
         LET exist = FALSE
      ELSE
         LET exist = TRUE
         DISPLAY BY NAME p_customer.*
         MENU "BROWSE"
            COMMAND "Next" "View the next customer in the list"
```

```
            FETCH NEXT customer_set INTO p_customer.*
            IF status = NOTFOUND THEN
               ERROR "No more customers in this direction"
                  ATTRIBUTE(RED, REVERSE)
               FETCH LAST customer_set INTO p_customer.*
            END IF
            DISPLAY BY NAME p_customer.*  ATTRIBUTE(CYAN)
         COMMAND "Previous" "View the previous customer in the list"
            FETCH PREVIOUS customer_set INTO p_customer.*
            IF status = NOTFOUND THEN
               ERROR "No more customers in this direction"
                  ATTRIBUTE(RED, REVERSE)
               FETCH FIRST customer_set INTO p_customer.*
            END IF
            DISPLAY BY NAME p_customer.*  ATTRIBUTE(CYAN)
         COMMAND "First" "View the first customer in the list"
            FETCH FIRST customer_set INTO p_customer.*
            DISPLAY BY NAME p_customer.*  ATTRIBUTE(CYAN)
         COMMAND "Last" "View the last customer in the list"
            FETCH LAST customer_set INTO p_customer.*
            DISPLAY BY NAME p_customer.*  ATTRIBUTE(CYAN)
         COMMAND "Select" "Exit BROWSE selecting the current customer"
            LET chosen = TRUE
            EXIT MENU
         COMMAND "Quit" "Quit BROWSE without selecting a customer"
            LET chosen = FALSE
            EXIT MENU
      END MENU
   END IF
   CLOSE customer_set

   IF NOT exist THEN
      CLEAR FORM
      CALL mess("No customer satisfies query", mrow)
      LET p_customer.customer_num = NULL
      RETURN (FALSE)
   END IF
   IF NOT chosen THEN
      CLEAR FORM
      LET p_customer.customer_num = NULL
      CALL mess("No selection made", mrow)
      RETURN (FALSE)
   END IF
   RETURN (TRUE)
END FUNCTION
FUNCTION update_customer()

   CALL clear_menu()
   IF p_customer.customer_num IS NULL THEN
      CALL mess("No customer has been selected; use the
         Find-cust option",23)
      RETURN
   END IF
   MESSAGE "Press F1 or CTRL-F for field-level help"
   DISPLAY "Press ESC to update customer data; DEL to abort" AT 1,1
   INPUT BY NAME p_customer.* WITHOUT DEFAULTS
      AFTER FIELD state
         CALL statehelp()
         DISPLAY "Press ESC to update customer data; DEL to abort",
            "" AT 1,1
      ON KEY (F1, CONTROL-F)
```

```
            CALL customer_help()
      END INPUT
      IF NOT int_flag THEN
         UPDATE customer SET customer.* = p_customer.*
            WHERE customer_num = p_customer.customer_num
         CALL mess("Customer data modified", 23)
      ELSE
         LET int_flag = FALSE
         SELECT * INTO p_customer.* FROM customer
            WHERE customer_num = p_customer.customer_num
         DISPLAY BY NAME p_customer.*
         ERROR "Customer update aborted" ATTRIBUTE (RED, REVERSE)
      END IF
   END FUNCTION


   FUNCTION delete_customer()
      DEFINE answer CHAR(1),
         num_orders INTEGER

      CALL clear_menu()
      IF p_customer.customer_num IS NULL THEN
         ERROR "No customer has been selected; use the Find-customer option"
            ATTRIBUTE (RED, REVERSE)
         RETURN
      END IF

      SELECT COUNT(*) INTO num_orders
         FROM orders
         WHERE customer_num = p_customer.customer_num
      IF num_orders THEN
         ERROR "This customer has active orders and can not be removed"
            ATTRIBUTE (RED, REVERSE)
         RETURN
      END IF

      PROMPT " Are you sure you want to delete this customer row? "
         FOR CHAR answer
      IF answer MATCHES "[yY]" THEN
         DELETE FROM customer
            WHERE customer_num = p_customer.customer_num
         CLEAR FORM
         CALL mess("Customer entry deleted", 23)
         LET p_customer.customer_num = NULL
      ELSE
         ERROR "Deletion aborted" ATTRIBUTE (RED, REVERSE)
      END IF
   END FUNCTION



   FUNCTION customer_help()
      CASE
         WHEN infield(customer_num) CALL showhelp(1001)
         WHEN infield(fname) CALL showhelp(1002)
         WHEN infield(lname) CALL showhelp(1003)
         WHEN infield(company) CALL showhelp(1004)
         WHEN infield(address1) CALL showhelp(1005)
         WHEN infield(address2) CALL showhelp(1006)
         WHEN infield(city) CALL showhelp(1007)
         WHEN infield(state) CALL showhelp(1008)
```

```
      WHEN infield(zipcode) CALL showhelp(1009)
      WHEN infield(phone) CALL showhelp(1010)
   END CASE
END FUNCTION


FUNCTION statehelp()
   DEFINE idx INTEGER

   SELECT COUNT(*) INTO idx
      FROM state
      WHERE code = p_customer.state
   IF idx = 1 THEN
      RETURN
   END IF

   DISPLAY "Move cursor using F3, F4, and arrow keys; press ESC to select
      state"
      AT 1,1
   OPEN WINDOW w_state AT 8,37
      WITH FORM "state_list"
      ATTRIBUTE (BORDER, RED, FORM LINE 2)

   CALL set_count(state_cnt)
   DISPLAY ARRAY p_state TO s_state.*
   LET idx = arr_curr()

   CLOSE WINDOW w_state
   LET p_customer.state = p_state[idx].code
   DISPLAY BY NAME p_customer.state ATTRIBUTE (MAGENTA)
   RETURN
END FUNCTION
```

d4_orders.4gl

```
GLOBALS
   "d4_globals.4gl"

FUNCTION orders()

   OPEN FORM order_form FROM "orderform"
   DISPLAY FORM order_form
      ATTRIBUTE(MAGENTA)
   MENU "ORDERS"
      COMMAND "Add-order" "Enter new order to database and print invoice"
            HELP 301
         CALL add_order()
      COMMAND "Update-order" "Enter shipping or payment data" HELP 302
         CALL update_order()
      COMMAND "Find-order" "Look up and display orders" HELP 303
         CALL get_order()
      COMMAND "Delete-order" "Remove an order from the database" HELP 304
         CALL delete_order()
      COMMAND "Exit" "Return to MAIN Menu" HELP 305
         CLEAR SCREEN
         EXIT MENU
   END MENU
END FUNCTION

FUNCTION add_order()
   DEFINE pa_curr, s_curr, num_stocks INTEGER,
      file_name CHAR(20),
      query_stat INTEGER

   CALL clear_menu()
   LET query_stat = query_customer(2)
   IF query_stat IS NULL THEN
      RETURN
   END IF
   IF NOT query_stat THEN
      OPEN WINDOW cust_w AT 3,5
         WITH 19 ROWS, 72 COLUMNS
         ATTRIBUTE(BORDER, YELLOW)
      OPEN FORM o_cust FROM "custform"
      DISPLAY FORM o_cust
         ATTRIBUTE(MAGENTA)
      CALL fgl_drawbox(3,61,4,7)
      CALL fgl_drawbox(11,61,4,7)
      CALL add_customer(FALSE)
      CLOSE FORM o_cust
      CLOSE WINDOW cust_w
      IF p_customer.customer_num IS NULL THEN
         RETURN
      ELSE
         DISPLAY by name p_customer.*
      END IF
   END IF

   MESSAGE "Enter the order date, PO number and shipping instructions."
   INPUT BY NAME p_orders.order_date, p_orders.po_num,
      p_orders.ship_instruct
   IF int_flag THEN
      LET int_flag = FALSE
```

```
      CLEAR FORM
      ERROR "Order input aborted" ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF
   INPUT ARRAY p_items FROM s_items.* HELP 311
      BEFORE FIELD stock_num
         MESSAGE "Press ESC to write order"
         DISPLAY "Enter a stock number or press CTRL-B to scan stock list"
            AT 1,1
      BEFORE FIELD manu_code
         MESSAGE "Enter the code for a manufacturer"
      BEFORE FIELD quantity
         DISPLAY "" AT 1,1
         MESSAGE "Enter the item quantity"
      ON KEY (CONTROL-B)
         IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
            LET pa_curr = arr_curr()
            LET s_curr = scr_line()
            CALL get_stock() RETURNING
               p_items[pa_curr].stock_num, p_items[pa_curr].manu_code,
               p_items[pa_curr].description, p_items[pa_curr].unit_price
            DISPLAY p_items[pa_curr].stock_num TO s_items[s_curr].stock_num
            DISPLAY p_items[pa_curr].manu_code TO s_items[s_curr].manu_code
            DISPLAY p_items[pa_curr].description TO s_items[s_curr].
               description
            DISPLAY p_items[pa_curr].unit_price TO s_items[s_curr].
               unit_price
            NEXT FIELD quantity
         END IF
      AFTER FIELD stock_num, manu_code
         LET pa_curr = arr_curr()
         IF p_items[pa_curr].stock_num IS NOT NULL
            AND p_items[pa_curr].manu_code IS NOT NULL
         THEN
            CALL get_item()
         END IF
      AFTER FIELD quantity
         MESSAGE ""
         LET pa_curr = arr_curr()
         IF p_items[pa_curr].unit_price IS NOT NULL
            AND p_items[pa_curr].quantity IS NOT NULL
         THEN
            CALL item_total()
         ELSE
            ERROR
            "A valid stock code, manufacturer, and quantity must all be
               entered"
               ATTRIBUTE (RED, REVERSE)
            NEXT FIELD stock_num
         END IF
      AFTER INSERT, DELETE
         CALL renum_items()
         CALL order_total()
      AFTER ROW
         CALL order_total()
   END INPUT
```

```
   IF int_flag THEN
      LET int_flag = FALSE
      CLEAR FORM
      ERROR "Order input aborted" ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF
   WHENEVER ERROR CONTINUE
   BEGIN WORK
   INSERT INTO orders (order_num, order_date, customer_num,
         ship_instruct, po_num)
      VALUES (0, p_orders.order_date, p_customer.customer_num,
         p_orders.ship_instruct, p_orders.po_num)
   IF status < 0 THEN
      ROLLBACK WORK
      ERROR "Unable to complete update of orders table"
         ATTRIBUTE(RED, REVERSE, BLINK)
      RETURN
   END IF
   LET p_orders.order_num = SQLCA.SQLERRD[2]
   DISPLAY BY NAME p_orders.order_num
   IF NOT insert_items() THEN
      ROLLBACK WORK
      ERROR "Unable to insert items" ATTRIBUTE(RED, REVERSE, BLINK)
      RETURN
   END IF

   COMMIT WORK
   WHENEVER ERROR STOP
   CALL mess("Order added", 23)
   LET file_name = "inv", p_orders.order_num USING "<<<<&",".out"
   CALL invoice(file_name)
   CLEAR FORM
END FUNCTION


FUNCTION update_order()

   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION delete_order()

   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION order_total()
   DEFINE order_total MONEY(8),
      i INTEGER

   LET order_total = 0.00
   FOR i = 1 TO ARR_COUNT()
      IF p_items[i].total_price IS NOT NULL THEN
         LET order_total = order_total + p_items[i].total_price
```

```
      END IF
   END FOR
   LET order_total = 1.1 * order_total
   DISPLAY order_total TO t_price
      ATTRIBUTE(GREEN)
END FUNCTION
FUNCTION item_total()
   DEFINE pa_curr, sc_curr INTEGER

   LET pa_curr = arr_curr()
   LET sc_curr = scr_line()
   LET p_items[pa_curr].total_price =
       p_items[pa_curr].quantity * p_items[pa_curr].unit_price
   DISPLAY p_items[pa_curr].total_price TO s_items[sc_curr].total_price
END FUNCTION


FUNCTION renum_items()
   DEFINE pa_curr, pa_total, sc_curr, sc_total, k INTEGER

   LET pa_curr = arr_curr()
   LET pa_total = arr_count()
   LET sc_curr = scr_line()
   LET sc_total = 4
   FOR k = pa_curr TO pa_total
      LET p_items[k].item_num = k
      IF sc_curr <= sc_total THEN
         DISPLAY k TO s_items[sc_curr].item_num
         LET sc_curr = sc_curr + 1
      END IF
   END FOR
END FUNCTION


FUNCTION insert_items()
   DEFINE idx INTEGER

   FOR idx = 1 TO arr_count()
      IF p_items[idx].quantity != 0 THEN
         INSERT INTO items
            VALUES (p_items[idx].item_num, p_orders.order_num,
               p_items[idx].stock_num, p_items[idx].manu_code,
               p_items[idx].quantity, p_items[idx].total_price)
         IF status < 0 THEN
            RETURN (FALSE)
         END IF
      END IF
   END FOR
   RETURN (TRUE)
END FUNCTION


FUNCTION get_stock()
   DEFINE idx integer

   OPEN WINDOW stock_w AT 7, 3
      WITH FORM "stock_sel"
      ATTRIBUTE(BORDER, YELLOW)
   CALL set_count(stock_cnt)
   DISPLAY
```

```
    "Use cursor using F3, F4, and arrow keys; press ESC to select a stock
       item"
      AT 1,1
   DISPLAY ARRAY p_stock TO s_stock.*
   LET idx = arr_curr()
   CLOSE WINDOW stock_w
   RETURN p_stock[idx].stock_num, p_stock[idx].manu_code,
      p_stock[idx].description, p_stock[idx].unit_price
END FUNCTION



FUNCTION get_order()
   DEFINE idx, exist, chosen INTEGER,
      answer CHAR(1)

   CALL clear_menu()
   CLEAR FORM
   IF NOT query_customer(2) THEN
      RETURN
   END IF
   DECLARE order_list CURSOR FOR
      SELECT order_num, order_date, po_num, ship_instruct
         FROM orders
         WHERE customer_num = p_customer.customer_num
   LET exist = FALSE
   LET chosen = FALSE
   FOREACH order_list INTO p_orders.*
      LET exist = TRUE
      CLEAR orders.*
      FOR idx = 1 TO 4
         CLEAR s_items[idx].*
      END FOR
      DISPLAY p_orders.* TO orders.*
      DECLARE item_list CURSOR FOR
         SELECT item_num, items.stock_num, items.manu_code,
               description, quantity, unit_price, total_price
            FROM items, stock
            WHERE order_num = p_orders.order_num
               AND items.stock_num = stock.stock_num
               AND items.manu_code = stock.manu_code
            ORDER BY item_num
      LET idx = 1

      FOREACH item_list INTO p_items[idx].*
         LET idx = idx + 1
         IF idx > 10 THEN
            ERROR "More than 10 items; only 10 items displayed"
               ATTRIBUTE (RED, REVERSE)
            EXIT FOREACH
         END IF
      END FOREACH
      CALL set_count(idx - 1)
      CALL order_total()
      MESSAGE "Press ESC when you finish viewing the items"
      DISPLAY ARRAY p_items TO s_items.*
         ATTRIBUTE(CYAN)
      MESSAGE ""
      IF int_flag THEN
         LET int_flag = FALSE
         EXIT FOREACH
```

```
      END IF
      PROMPT " Enter 'y' to select this order ",
         "or RETURN to view next order: " FOR CHAR answer
      IF answer MATCHES "[yY]" THEN
         LET chosen = TRUE
         EXIT FOREACH
      END IF
   END FOREACH

   IF NOT exist THEN
      ERROR "No orders found for this customer" ATTRIBUTE (RED)
   ELSE
      IF NOT chosen THEN
         CLEAR FORM
         ERROR "No order selected for this customer" ATTRIBUTE (RED)
      END IF
   END IF
END FUNCTION


FUNCTION get_item()
   DEFINE pa_curr, sc_curr INTEGER

   LET pa_curr = arr_curr()
   LET sc_curr = scr_line()
   SELECT description, unit_price
      INTO p_items[pa_curr].description,
         p_items[pa_curr].unit_price
      FROM stock
      WHERE stock.stock_num = p_items[pa_curr].stock_num
         AND stock.manu_code = p_items[pa_curr].manu_code
   IF status THEN
      LET p_items[pa_curr].description = NULL
      LET p_items[pa_curr].unit_price = NULL
   END IF
   DISPLAY p_items[pa_curr].description, p_items[pa_curr].unit_price
      TO s_items[sc_curr].description, s_items[sc_curr].unit_price
   IF p_items[pa_curr].quantity IS NOT NULL THEN
      CALL item_total()
   END IF
END FUNCTION


FUNCTION invoice(file_name)
   DEFINE x_invoice RECORD
         order_num              LIKE orders.order_num,
         order_date             LIKE orders.order_date,
         ship_instruct          LIKE orders.ship_instruct,
         backlog                LIKE orders.backlog,
         po_num                 LIKE orders.po_num,
         ship_date              LIKE orders.ship_date,
         ship_weight            LIKE orders.ship_weight,
         ship_charge            LIKE orders.ship_charge,
         item_num               LIKE items.item_num,
         stock_num              LIKE items.stock_num,
         manu_code              LIKE items.manu_code,
         quantity               LIKE items.quantity,
         total_price            LIKE items.total_price,
         description            LIKE stock.description,
         unit_price             LIKE stock.unit_price,
         unit                   LIKE stock.unit,
```

```
            unit_descr             LIKE stock.unit_descr,
            manu_name              LIKE manufact.manu_name
         END RECORD,
         file_name CHAR(20),
         msg CHAR(40)

      DECLARE invoice_data CURSOR FOR
         SELECT o.order_num,order_date,ship_instruct,backlog,po_num,ship_date,
            ship_weight,ship_charge,
            item_num,i.stock_num,i.manu_code,quantity,total_price,
            s.description,unit_price,unit,unit_descr,
            manu_name
         FROM orders o,items i,stock s,manufact m
         WHERE
            ((o.order_num=p_orders.order_num) AND
            (i.order_num=p_orders.order_num) AND
            (i.stock_num=s.stock_num AND
            i.manu_code=s.manu_code) AND
            (i.manu_code=m.manu_code))
         ORDER BY 9
      CASE (print_option)
         WHEN "f"
            START REPORT r_invoice TO file_name
            CALL clear_menu()
            MESSAGE "Writing invoice -- please wait"
         WHEN "p"
            START REPORT r_invoice TO PRINTER
            CALL clear_menu()
            MESSAGE "Writing invoice -- please wait"
         WHEN "s"
            START REPORT r_invoice
      END CASE
      FOREACH invoice_data INTO x_invoice.*
         OUTPUT TO REPORT r_invoice (p_customer.*, x_invoice.*)
      END FOREACH
      FINISH REPORT r_invoice
      IF print_option = "f" THEN
         LET msg = "Invoice written to file ", file_name CLIPPED
         CALL mess(msg, 23)
      END IF
   END FUNCTION


   REPORT r_invoice (c, x)
      DEFINE c RECORD LIKE customer.*,
         x RECORD
            order_num      LIKE orders.order_num,
            order_date     LIKE orders.order_date,
            ship_instruct  LIKE orders.ship_instruct,
            backlog        LIKE orders.backlog,
            po_num         LIKE orders.po_num,
            ship_date      LIKE orders.ship_date,
            ship_weight    LIKE orders.ship_weight,
            ship_charge    LIKE orders.ship_charge,
            item_num       LIKE items.item_num,
            stock_num      LIKE items.stock_num,
            manu_code      LIKE items.manu_code,
            quantity       LIKE items.quantity,
            total_price    LIKE items.total_price,
            description    LIKE stock.description,
            unit_price     LIKE stock.unit_price,
```

```
            unit          LIKE stock.unit,
            unit_descr    LIKE stock.unit_descr,
            manu_name     LIKE manufact.manu_name
         END RECORD,
         sales_tax, calc_total MONEY(8,2)

   OUTPUT
      LEFT MARGIN 0
      RIGHT MARGIN 0
      TOP MARGIN 1
      BOTTOM MARGIN 1
      PAGE LENGTH 48

   FORMAT
      BEFORE GROUP OF x.order_num
         SKIP TO TOP OF PAGE
         SKIP 1 LINE
         PRINT 10 SPACES,
            "   W E S T   C O A S T   W H O L E S A L E R S ,   I N C ."
         PRINT 30 SPACES," 1400 Hanbonon Drive"
         PRINT 30 SPACES,"Menlo Park, CA  94025"
         SKIP 1 LINES
         PRINT "Bill To:", COLUMN 10,c.fname CLIPPED, " ", c.lname CLIPPED;
         PRINT COLUMN 56,"Invoice No.        ",x.order_num USING "&&&&&"
         PRINT COLUMN 10,c.company
         PRINT COLUMN 10,c.address1 CLIPPED;
         PRINT COLUMN 56,"Invoice Date: ", x.order_date
         PRINT COLUMN 10,c.address2 CLIPPED;
         PRINT COLUMN 56,"Customer No.       ", c.customer_num USING "####&"
         PRINT COLUMN 10,c.city CLIPPED,", ",c.state CLIPPED,"   ",
            c.zipcode CLIPPED;
         PRINT COLUMN 56,"PO No. ",x.po_num
         PRINT COLUMN 10,c.phone CLIPPED;
         PRINT COLUMN 56,"Backlog Status: ",x.backlog
         SKIP 1 LINES
         PRINT COLUMN 10,"Shipping Instructions: ", x.ship_instruct
         PRINT COLUMN 10,"Ship Date: ",x.ship_date USING "ddd. mmm dd, yyyy";
         PRINT "   Weight: ", x.ship_weight USING "######&.&&"
         SKIP 1 LINES
         PRINT "----------------------------------------";
         PRINT "----------------------------------------"
         PRINT "   Stock                              Unit        ";
         PRINT "                        Item "
         PRINT " #  Num Man    Description      Qty   Cost   Unit ";
         PRINT " Unit Description      Total"
         SKIP 1 LINES
         LET calc_total = 0.00

      ON EVERY ROW
         PRINT x.item_num USING "#&","  ",
            x.stock_num USING "&&", "  ",x.manu_code;
         PRINT "  ",x.description,"  ",x.quantity USING "###&", "  ";
         PRINT x.unit_price USING "$$$&.&&","  ",x.unit, "  ",
            x.unit_descr,"  ";
         PRINT x.total_price USING "$$$$$$$&.&&"
         LET calc_total = calc_total + x.total_price

      AFTER GROUP OF x.order_num
         SKIP 1 LINES
         PRINT "----------------------------------------";
         PRINT "----------------------------------------"
```

```
      PRINT COLUMN 50, "        Sub-total: ",calc_total USING "$$$$$$$&.&&"
      LET sales_tax = 0.065 * calc_total
      LET x.ship_charge = 0.035 * calc_total
      PRINT COLUMN 45, "Shipping Charge (3.5%): ",
         x.ship_charge USING "$$$$$$$&.&&"
      PRINT COLUMN 50, " Sales Tax (6.5%): ",sales_tax USING "$$$$$$$&.&&"
      PRINT COLUMN 50, "                   -----------"
      LET calc_total = calc_total + x.ship_charge + sales_tax
      PRINT COLUMN 50, "            Total: ",calc_total USING "$$$$$$$&.&&"
      IF print_option = "s" THEN
         PAUSE "Type RETURN to continue"
      END IF
END REPORT
```

d4_stock.4gl

```
GLOBALS
   "d4_globals.4gl"


FUNCTION stock()
   MENU "STOCK"
      COMMAND "Add-stock" "Add new stock items to database" HELP 401
         CALL add_stock()
      COMMAND "Find-stock" "Look up specific stock item" HELP 402
         CALL query_stock()
      COMMAND "Update-stock" "Modify current stock information" HELP 403
         CALL update_stock()
      COMMAND "Delete-stock" "Remove a stock item from database" HELP 404
         CALL delete_stock()
      COMMAND "Exit" "Return to MAIN Menu" HELP 405
         CLEAR SCREEN
         EXIT MENU
   END MENU
END FUNCTION


FUNCTION add_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION query_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION update_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION delete_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION
```

d4_report.4gl

```
GLOBALS
   "d4_globals.4gl"


FUNCTION reports()
   CALL ring_menu()
   MENU "REPORTS"
      COMMAND "Labels" "Print mailing labels from customer list"
            HELP 501
         CALL print_labels()
         CLEAR SCREEN
         CALL ring_menu()
      COMMAND "Accounts-receivable" "Print current unpaid orders" HELP 502
         CALL print_ar()
         CLEAR SCREEN
         CALL ring_menu()
      COMMAND "Backlog" "Print backlogged orders" HELP 503
         CALL print_backlog()
         CLEAR SCREEN
         CALL ring_menu()
      COMMAND "Stock-list" "Print stock available" HELP 504
         CALL print_stock()
         CLEAR SCREEN
         CALL ring_menu()
      COMMAND "Options" "Change the report output options" HELP 505
         CALL update_options()
         CALL ring_menu()
      COMMAND "Exit" "Return to MAIN Menu" HELP 506
         CLEAR SCREEN
         EXIT MENU
   END MENU
END FUNCTION
FUNCTION print_labels()
   DEFINE where_part CHAR(200),
      query_text CHAR(250),
      msg CHAR(50),
      file_name CHAR(20)

   OPTIONS
      FORM LINE 7
   OPEN FORM customer FROM "custform"
   DISPLAY FORM customer
      ATTRIBUTE(MAGENTA)
   CALL fgl_drawbox(3,30,3,43)
   CALL fgl_drawbox(3,61,8,7)
   CALL fgl_drawbox(11,61,8,7)
   CALL clear_menu()
   DISPLAY "CUSTOMER LABELS:" AT 1,1
   MESSAGE "Use query-by-example to select customer list"
   CONSTRUCT BY NAME where_part ON customer.*
   IF int_flag THEN
      LET int_flag = FALSE
      ERROR "Label print request aborted"
      RETURN
   END IF
   MESSAGE ""
   LET query_text = "select * from customer where ", where_part CLIPPED,
      " order by zipcode"
```

```
   PREPARE label_st FROM query_text
   DECLARE label_list CURSOR FOR label_st
   CASE (print_option)
      WHEN "f"
         PROMPT " Enter file name for labels >" FOR file_name
         IF file_name IS NULL THEN
            LET file_name = "labels.out"
         END IF
         MESSAGE "Printing mailing labels to ", file_name CLIPPED,
            " -- Please wait"
         START REPORT labels_report TO file_name
      WHEN "p"
         MESSAGE "Printing mailing labels -- Please wait"
         START REPORT labels_report TO PRINTER
      WHEN "s"
         START REPORT labels_report
         CLEAR SCREEN
   END CASE
   FOREACH label_list INTO p_customer.*
      OUTPUT TO REPORT labels_report (p_customer.*)
      IF int_flag THEN
         LET int_flag = FALSE
         EXIT FOREACH
      END IF
   END FOREACH
   FINISH REPORT labels_report
   IF int_flag THEN
      LET int_flag = FALSE
      ERROR "Label printing aborted" ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF
   IF print_option = "f" THEN
      LET msg = "Labels printed to ", file_name CLIPPED
      CALL mess(msg, 23)
   END IF
   CLOSE FORM customer
   OPTIONS
      FORM LINE 3
END FUNCTION


REPORT labels_report (rl)
   DEFINE rl RECORD LIKE customer.*

OUTPUT
   TOP MARGIN 0
   BOTTOM MARGIN 0
   PAGE LENGTH 6

FORMAT
   ON EVERY ROW
   SKIP TO TOP OF PAGE
   PRINT rl.fname CLIPPED, 1 SPACE, rl.lname
   PRINT rl.company
   PRINT rl.address1
   IF rl.address2 IS NOT NULL THEN
      PRINT rl.address2
   END IF
   PRINT rl.city CLIPPED, ", ", rl.state, 2 SPACES, rl.zipcode
   IF print_option = "s" THEN
      PAUSE "Type RETURN to continue"
```

```
   END IF
END REPORT



FUNCTION print_ar()
   DEFINE r RECORD
          customer_num     LIKE customer.customer_num,
          fname            LIKE customer.fname,
          lname            LIKE customer.lname,
          company          LIKE customer.company,
          order_num        LIKE orders.order_num,
          order_date       LIKE orders.order_date,
          ship_date        LIKE orders.ship_date,
          paid_date        LIKE orders.paid_date,
          total_price      LIKE items.total_price
      END RECORD,
   file_name CHAR(20),
   msg CHAR(50)
   DECLARE ar_list CURSOR FOR
   SELECT customer.customer_num,fname,lname,company,
          orders.order_num,order_date,ship_date,paid_date,
          total_price
      FROM customer,orders,items
      WHERE customer.customer_num=orders.customer_num AND
         paid_date IS NULL AND
         orders.order_num=items.order_num
      ORDER BY 1,5

   CALL clear_menu()
   CASE (print_option)
      WHEN "f"
         PROMPT " Enter file name for AR Report >" FOR file_name
         IF file_name IS NULL THEN
            LET file_name = "ar.out"
         END IF
         MESSAGE "Printing AR REPORT to ", file_name CLIPPED,
            " -- Please wait"
         START REPORT ar_report TO file_name
      WHEN "p"
         MESSAGE "Printing AR REPORT -- Please wait"
         START REPORT ar_report TO PRINTER
      WHEN "s"
         START REPORT ar_report
         CLEAR SCREEN
         MESSAGE "Printing AR REPORT -- Please wait"
   END CASE
   FOREACH ar_list INTO r.*
      OUTPUT TO REPORT ar_report (r.*)
      IF int_flag THEN
         LET int_flag = FALSE
         EXIT FOREACH
      END IF
   END FOREACH
   FINISH REPORT ar_report
   IF int_flag THEN
      LET int_flag = FALSE
      ERROR "AR REPORT printing aborted" ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF
   IF print_option = "f" THEN
```

```
      LET msg = "AR REPORT printed to ", file_name CLIPPED
      CALL mess(msg, 23)
   END IF
END FUNCTION
REPORT ar_report (r)
   DEFINE r RECORD
         customer_num     LIKE customer.customer_num,
         fname            LIKE customer.fname,
         lname            LIKE customer.lname,
         company          LIKE customer.company,
         order_num        LIKE orders.order_num,
         order_date       LIKE orders.order_date,
         ship_date        LIKE orders.ship_date,
         paid_date        LIKE orders.paid_date,
         total_price      LIKE items.total_price
      END RECORD,
   name_text CHAR(80)

OUTPUT
   PAGE LENGTH 22
   LEFT MARGIN 0

FORMAT
   PAGE HEADER
      PRINT 15 SPACES,"West Coast Wholesalers, Inc."
      PRINT 6 SPACES,
         "Statement of ACCOUNTS RECEIVABLE - ",
         TODAY USING "mmm dd, yyyy"
      SKIP 1 LINES
      LET name_text = r.fname CLIPPED," ",r.lname CLIPPED,"/",
         r.company CLIPPED
      PRINT 29 - length(name_text)/2 SPACES, name_text
      SKIP 1 LINES
      PRINT " Order Date   Order Number    Ship Date            Amount"
      PRINT "-------------------------------------------------------------"

   BEFORE GROUP OF r.customer_num
      SKIP TO TOP OF PAGE

   AFTER GROUP OF r.order_num
      NEED 3 LINES
      PRINT " ",r.order_date,7 SPACES,r.order_num USING "###&",8 SPACES,
         r.ship_date,"  ",
         GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
   AFTER GROUP OF r.customer_num
      PRINT 42 SPACES,"----------------"
      PRINT 42 SPACES,GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
   PAGE TRAILER
      IF print_option = "s" THEN
         PAUSE "Type RETURN to continue"
      END IF
END REPORT




FUNCTION update_options()
   DEFINE po CHAR(2)

   DISPLAY "Current print option:" AT 8,25
```

```
   LET po = " ", print_option
   DISPLAY po AT 8,46 ATTRIBUTE(CYAN)
   MENU "REPORT OPTIONS"
      COMMAND "File" "Send all reports to a file"
         LET print_option = "f"
         EXIT MENU
      COMMAND "Printer" "Send all reports to the printer"
         LET print_option = "p"
         EXIT MENU
      COMMAND "Screen" "Send all reports to the terminal screen"
         LET print_option = "s"
         EXIT MENU
      COMMAND "Exit"
         EXIT MENU
   END MENU
   DISPLAY "" AT 8,1
END FUNCTION


FUNCTION print_backlog()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION print_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION
```

d4_demo.4gl

```
FUNCTION demo()

   CALL ring_menu()
   MENU "DEMO"
      COMMAND "Menus" "Source code for MAIN Menu"
         CALL showhelp(2001)
      COMMAND "Windows" "Source code for STATE CODE Window"
         CALL showhelp(2007)
      COMMAND "Forms" "Source code for new CUSTOMER data entry"
         CALL showhelp(2006)
      COMMAND "Detail-Scrolling"
        "Source code for scrolling of new ORDER line-items"
         CALL showhelp(2003)
      COMMAND "Scroll-
Cursor" "Source code for customer record BROWSE/SCROLL"
         CALL showhelp(2008)
      COMMAND "Query_language" "Source code for new order insertion using
         SQL"
         CALL showhelp(2004)
      COMMAND "Construct_query"
         "Source code for QUERY-BY-EXAMPLE selection and reporting"
         CALL showhelp(2002)
      COMMAND "Reports" "Source code for MAILING LABEL report"
         CALL showhelp(2005)
      COMMAND "Exit" "Return to MAIN MENU"
         CLEAR SCREEN
         EXIT MENU
   END MENU
END FUNCTION
```

helpdemo.src

```
.101
The Customer option presents you with a menu that allows you to:

      o   Add new customers to the database
      o   Locate customers in the database
      o   Update customer files
      o   Remove customers from the database

.102
The Orders option presents you with a menu that allows you to:

      o   Enter a new order and print an invoice
      o   Update an existing order
      o   Look up and display orders
      o   Remove orders from the database

.103
The Stock option presents you with a menu that allows you to:

      o   Add new items to the list of stock
      o   Look up and display stock items
      o   Modify current stock descriptions and values
      o   Remove items from the list of stock

.104
The Reports option presents you with a menu that allows you to:

      o   Select and print mailing labels sorted by zip code
      o   Print a report of current accounts receivable
      o   Print a report of backloged orders
      o   Print a list of current stock available
      o   Change the report output options

.105
The Exit option leaves the program and returns you to the operating system.

.201
The One-add option enables you to enter data on new customers to
the database. You may get assistance on what input is appropriate
for each field by pressing the function key F1 when the cursor
is in the field.   When you  have  entered all the data you want
for a given customer, press ESC to enter the data in the database.
If you want to abort a given entry and not write it to the database,
press the INTERRUPT key (usually DEL or CTRL-C).
```

```
.202
The  Many-add  option  enables  you  to  enter  data  on  new customers
to the database.  You may get assistance on what input is appropriate
for each  field by  pressing  the  function key F1 when the cursor
is in the field.   When you have  entered  all the data you
want for a given customer, press ESC to enter the data in the database. If
you want to abort a given entry and not write it to  the
database, press the INTERRUPT key (usually DEL or CTRL-C). After each
entry, the cursor will move to the beginning of the form
and await  the  entry of the next customer.  If you have no more
customers to add, press  CTRL-Z  to return to the CUSTOMER Menu.

.203
The Find- cust option allows you to select one or more customers and to
display their  data  on the screen by using query-by-example input.
Use the RETURN or arrow  keys to move through the form.
Enter the criteria you want the program to use in searching
for customers.  Your options include:

    o  Literal values
    o  A range of values (separated by ":")
    o  A list of values (separated by "|")
    o  Relational operators (for example ">105")
    o  Wildcards like ? and * to match single or any number of characters

.204
The Update-cust option enables you to alter data on old customers in the
database. You must first select a current customer row to deal with
by using the Find-cust option. You may get assistance on what input is
appropriate for each field by pressing the function key F1 when the cursor
is in the field. When you have altered all the data you want for
a given customer, press ESC to enter the data in the database.
If you want to abort the changes and not write them to the database,
press the INTERRUPT key (usually DEL or CTRL-C).

.205
The  Delete-cust option enables you to remove customers from the database.
You must  first  select  a current customer row to deal with by using the
Find-cust option.  For your protection, you will be asked  to  confirm
that  the  record should  be  deleted.   Once  deleted,  it  cannot  be
restored.  Customers with active orders can not be deleted.

.206
The Exit option of the CUSTOMER Menu takes you back to the MAIN Menu.

.301
The  Add-order  option enables you to add a new order for an existing
customer. You must first select the desired  customer  using  query-by-
example  selection criteria.  You will  then  enter  the  order  date,
PO  number,  and  shipping instructions.  The detail line items are then
entered into a scrolling  display array.  Up to ten items may be entered
using the four line screen array.  After the  new  order is
entered, an invoice is automatically generated and displayed on the screen.

.302
The Update-order option is currently not implemented.
```

```
.303
The Find-order option enables you to browse through and select an existing
order. You must first select the desired customer (or customers) who's
orders you  wish  to  scan.  For each customer selected, each corresponding
order will be  displayed on the screen for examination. You may
either select an invoice, skip to the next invoice, or cancel processing.

.304
The Delete-order option is currently not implemented.

.305
The Exit option of the ORDER Menu returns you to the MAIN Menu.

.311
You  may enter up to ten line items into the scrolling screen array.  A num
ber of  standard  functions  are available for manipulating the cursor in
a screen array.

    o  F1Insert new line in the screen array
    o  F2  Remove the current line from the screen array
    o  F3Page down one page in the screen array
    o  F4Page up one page in the screen array
    o  ESCExit input array
    o  CTRL-BWhen in the Stock Number or Manufacturer Code fields,
       a window will open in the middle of the screen and
       display a scrolled list of all items in stock, identified
       by the stock number and manufacturer.  Using F3, F4, and
       the up and down arrow keys, move the cursor to the line
       that identifies the desired item and hit ESC.  The
       window will disappear and the selected information will
       automatically appear in the proper line.
    o  etc...The arrow-keys, and the standard field editing keys
       are available

The item_total field will be displayed in reverse-video green for total
amounts over $500.

.401
The Add-stock option is currently not implemented.

.402
The Find-stock option is currently not implemented.

.403
The Update-stock option is currently not implemented.

.404
The Delete-stock option is currently not implemented.

.405
The Exit option of the STOCK Menu returns you to the MAIN Menu.

.501
The Labels option enables you to create a  list  of  mailing  labels  gener
ated using a query-by-example specification.  You will be prompted for
the  output file name.

.502
The  Accounts-receivable  option enables you to create a report
summarizing all unpaid  orders in the database.  You will be prompted for
the output file name.
```

```
.503
The Backlog option is currently not implemented.

.504
The Stock-list option is currently not implemented.

.505
The  Options  option  enables  you  to  change  the  destination  of any re
port generated during the current session.  The default option  is  to
display  all reports  on the terminal screen.  The other options are
to print all reports to either the printer or an operating system file.

.506
The Exit option of the REPORT Menu returns you to the MAIN Menu.



.1001
The Number field on the Customer Form contains the serial integer  assigned
to the  customer  row  when  the  data  for the customer is first
entered into the database.  It is a unique number for each customer.
The lowest value  of  this field is 101.

.1002
The first section following the Name label should contain the first name of
the contact person at the customer's company.

.1003
The second section following the Name label should contain the last name of
the contact person at the customer's company.

.1004
This field should contain the name of the customer's company.

.1005
The  first  line  of the Address section of the form should contain the
mailing address of the company.

.1006
The  second  line  of  the Address section of the form should be used only
when there  is  not  sufficient room in the first line to contain the
entire mailing address.

.1007
The City field should contain the city name portion of the mailing  address
of the customer.

.1008
Enter  the two-character code for the desired state.  If no code is
entered, or the entered code is not in the program's list of valid entries,
a  window  will appear  on the screen with a scrolling list of all
states and codes. Using the F3,  F4,  up  and  down  arrow keys, move the
cursor to the line containing the desired state.  After typing ESC,
the window will disappear and  the  selected state code will appear in
the customer entry screen.

.1009
Enter the five digit Zip Code in this field.
```

```
.1010
Enter the telephone number of the contact person at the customer's company.
Include the Area Code and extension using the format "###-###-#### #####".

.2001
The following is the INFORMIX-4GL source for the main menu.  Note that only
the text is specified by the MENU statement; the structure and runtime menu
functions are built-in.

   OPTIONS
      HELP FILE "helpdemo"
   OPEN FORM menu_form FROM "ring_menu"
   DISPLAY FORM menu_form
   MENU "MAIN"
      COMMAND "Customer" "Enter and maintain customer data" HELP 101
         CALL customer()
         DISPLAY FORM menu_form
      COMMAND "Orders" "Enter and maintain orders" HELP 102
         CALL orders()
         DISPLAY FORM menu_form
      COMMAND "Stock" "Enter and maintain stock list" HELP 103
         CALL stock()
         DISPLAY FORM menu_form

      COMMAND "Reports" "Print reports and mailing labels" HELP 104
         CALL reports()
         DISPLAY FORM menu_form
      COMMAND "Exit" "Exit program and return to operating system" HELP 105
         CLEAR SCREEN
         EXIT PROGRAM
   END MENU

.2002
The following is the INFORMIX-4GL source code for mailing-
label selection and printing. The CONSTRUCT statement manages the query-by-
example input and builds the corresponding SQL where-clause.

   CONSTRUCT BY NAME where_part ON customer.*
   LET query_text = "select * from customer where ", where_part CLIPPED,
      " order by zipcode"
   PREPARE mail_query FROM query_text
   DECLARE label_list CURSOR FOR mail_query
   PROMPT "Enter file name for labels >" FOR file_name
   MESSAGE "Printing mailing labels to ", file_name CLIPPED," --
 Please wait"
   START REPORT labels_report TO file_name
   FOREACH label_list INTO p_customer.*
      OUTPUT TO REPORT labels_report (p_customer.*)
   END FOREACH
   FINISH REPORT labels_report

See the source code option REPORT for the corresponding report routine.
```

```
.2003
The following is the INFORMIX-4GL source code for order entry using
scrolled input fields.  Only the INPUT ARRAY statement in needed
to utilize the full scrolling features.  Some additional code has been
added merely to customize the array processing to this application.

   DISPLAY "Press ESC to write order" AT 1,1
   INPUT ARRAY p_items FROM s_items.* HELP 311
      BEFORE FIELD stock_num
         MESSAGE "Enter a stock number."
      BEFORE FIELD manu_code
         MESSAGE "Enter the code for a manufacturer."
      AFTER FIELD stock_num, manu_code
         LET pa = arr_curr()
         LET sc = scr_line()
         SELECT description, unit_price
            INTO p_items[pa].description,
               p_items[pa].unit_price
            FROM stock
            WHERE stock_num = p_items[pa].stock_num AND
               stock_manu = p_items[pa].menu_code
         DISPLAY p_items[pa].description, p_items[pa].unit_price
            TO stock[sc].*
         CALL item_total()
      AFTER FIELD quantity
         CALL item_total()
      AFTER INSERT, DELETE, ROW
         CALL order_total()
   END INPUT

See the source code option QUERY-LANGUAGE for the SQL statements that
insert the order information into the database.

.2004
The following is the INFORMIX-4GL source code that uses SQL to insert the
entered order information into the database.  Note that the use of
transactions ensures that database integrity is maintained, even if an
intermediate operation fails.

   BEGIN WORK
   LET p_orders.order_num = 0
   INSERT INTO orders VALUES (p_orders.*)
   IF status < 0 THEN
      ROLLBACK WORK
      MESSAGE "Unable to complete update of orders table"
      RETURN
   END IF
   LET p_orders.order_num = SQLCA.SQLERRD[2]
   DISPLAY BY NAME p_orders.order_num
   FOR i = 1 to arr_count()
      INSERT INTO items
         VALUES (p_items[counter].item_num, p_orders.order_num,
            p_items[counter].stock_num,    p_items[counter].manu_code,
            p_items[counter].quantity,     p_items[counter].total_price)
      IF status < 0 THEN
         ROLLBACK WORK
         Message "Unable to insert items"
         RETURN FALSE
      END IF
   END FOR
   COMMIT WORK
```

```
.2005
The following is the INFORMIX-4GL source code that generates the mailing-
label report.  See the source code option CONSTRUCT for the report
calling sequence.

   REPORT labels_report (rl)
      DEFINE rl RECORD LIKE customer.*
      OUTPUT
         TOP MARGIN 0
         PAGE LENGTH 6
      FORMAT
         ON EVERY ROW
            SKIP TO TOP OF PAGE
            PRINT rl.fname CLIPPED, 1 SPACE, rl.lname
            PRINT rl.company
            PRINT rl.address1
            IF rl.address2 IS NOT NULL THEN
               PRINT rl.address2
            END IF
            PRINT rl.city CLIPPED, ", ", rl.state, 2 SPACES, rl.zipcode
   END REPORT

.2006
The following is the INFORMIX-4GL source code that manages a simple form
for data entry.  Note the use of special key definitions during data entry.

   OPEN FORM cust_form FROM "customer"
   DISPLAY FORM cust_form
   MESSAGE "Press F1 or CTRL-F for field help;",
      "F2 or CTRL-Z to return to CUSTOMER Menu"
   DISPLAY "Press ESC to enter new customer data or DEL to abort entry"
   INPUT BY NAME p_customer.*
      AFTER FIELD state
         CALL statehelp()
      ON KEY (F1, CONTROL-F)
         CALL customer_help()
      ON KEY (F2, CONTROL-Z)
         CLEAR FORM
         RETURN
   END INPUT

.2007
The following is the INFORMIX-4GL source code that opens a window in the
customer entry screen, displays the list of valid state names and codes, sa
ves  the index into the p_state array for the selected state, closes the
window, and returns the index to the calling routine.

   OPEN WINDOW w_state AT 8,40
      WITH FORM "state_list"
      ATTRIBUTE (BORDER, RED, FORM LINE 2)

   CALL set_count(state_cnt)
   DISPLAY ARRAY p_state TO s_state.*
   LET idx = arr_curr()

   CLOSE WINDOW w_state
   RETURN (idx)
```

```
.2008
The following is the INFORMIX-4GL source code that allows the user
to browse through the rows returned by a "scroll" cursor.

   DECLARE customer_set SCROLL CURSOR FOR
      SELECT * FROM customer
         ORDER BY lname
   OPEN customer_set
   FETCH FIRST customer_set INTO p_customer.*
   IF status = NOTFOUND THEN
      LET exist = FALSE
   ELSE
      LET exist = TRUE
      DISPLAY BY NAME p_customer.*
      MENU "BROWSE"
         COMMAND "Next" "View the next customer in the list"
            FETCH NEXT customer_set INTO p_customer.*
            IF status = NOTFOUND THEN
               ERROR "No more customers in this direction"
               FETCH LAST customer_set INTO p_customer.*
            END IF
            DISPLAY BY NAME p_customer.*
         COMMAND "Previous" "View the previous customer in the list"
            FETCH PREVIOUS customer_set INTO p_customer.*
            IF status = NOTFOUND THEN
               ERROR "No more customers in this direction"
               FETCH FIRST customer_set INTO p_customer.*
            END IF
            DISPLAY BY NAME p_customer.*
         COMMAND "First" "View the first customer in the list"
            FETCH FIRST customer_set INTO p_customer.*
            DISPLAY BY NAME p_customer.*
         COMMAND "Last" "View the last customer in the list"
            FETCH LAST customer_set INTO p_customer.*
            DISPLAY BY NAME p_customer.*
         COMMAND "Select" "Exit BROWSE selecting the current customer"
            LET chosen = TRUE
            EXIT MENU
         COMMAND "Quit" "Quit BROWSE without selecting a customer"
            LET chosen = FALSE
            EXIT MENU
      END MENU
   END IF
   CLOSE customer_set
```

SQL Statements That Can Be Embedded in 4GL Code

This appendix lists the syntax of SQL statements, as implemented for Informix database servers, that can be directly embedded in INFORMIX-4GL code. Most of these statements are from the Version 4.10 release of Informix servers, but a few are from subsequent releases. All of these statements are supported by the 4GL compiler.

Statements That Cannot Be Embedded

Any SQL statements or SQL syntax not listed in this appendix cannot be embedded within 4GL modules. SQL syntax that is not listed in this appendix can appear in 4GL source code only if all of the following are true of the SQL statement:

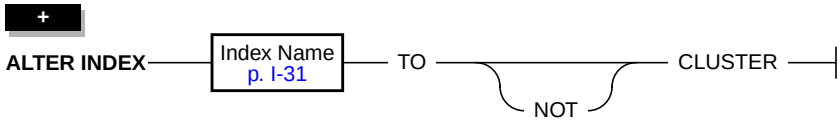
- It is preparable.
- It is supported by your Informix database server.
- It appears between SQL ... END SQL delimiters, as described in "SQL" on page 4-349.

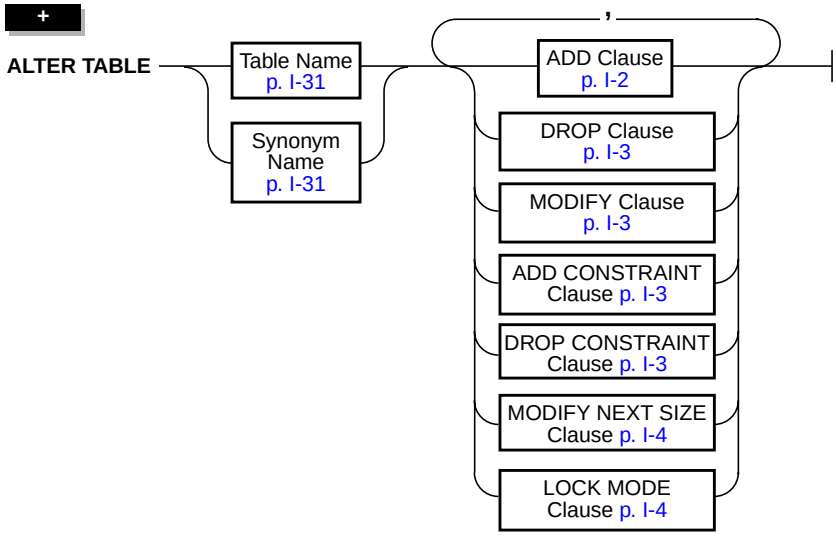
Alternatively, if the first two conditions are true, you can also use the PREPARE statement of SQL to do the following:

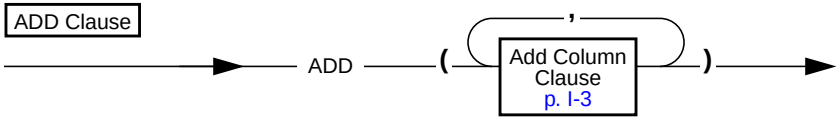
1. Store the SQL statement as a character string.
2. Set up the statement for execution by means of the PREPARE statement (see page I-18).
3. Process the statement by means of the EXECUTE or EXECUTE IMMEDIATE statement (see page I-12).
4. If the statement does not need to be reused, release the resources that it occupies with the FREE statement.

Embedded SQL Statements

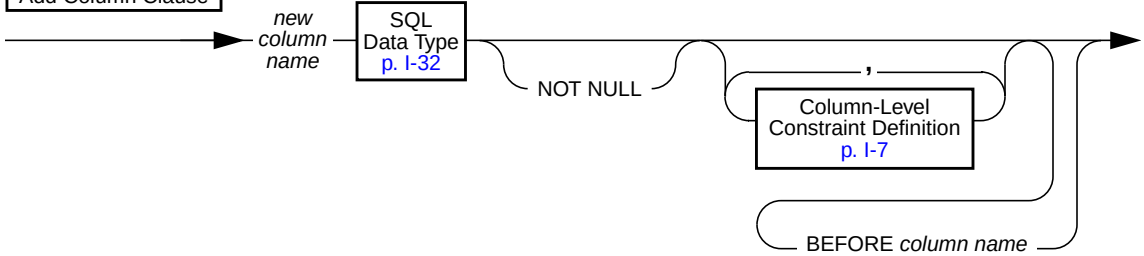
The following SQL statements can be directly embedded in INFORMIX-4GL code.



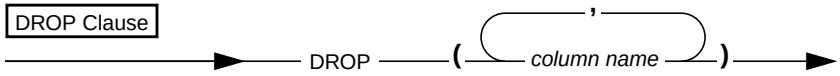




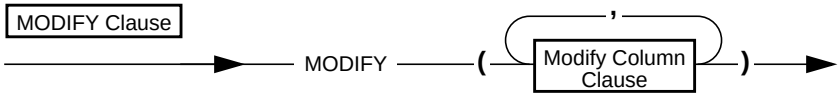
Add Column Clause



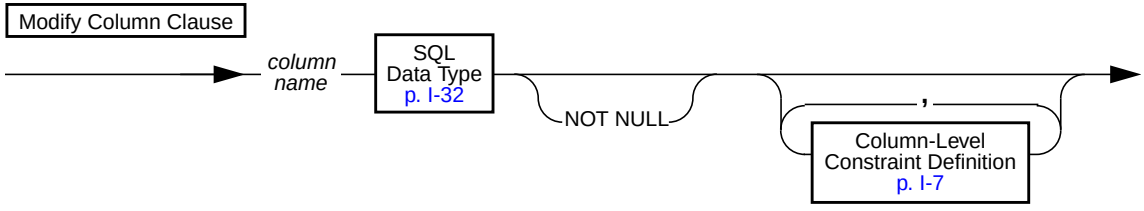
DROP Clause



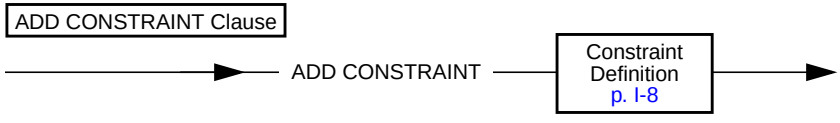
MODIFY Clause



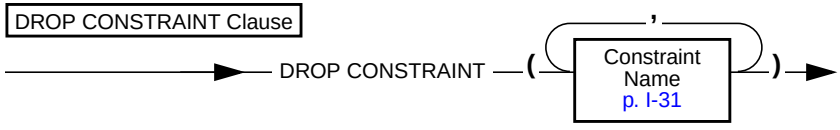
Modify Column Clause



ADD CONSTRAINT Clause

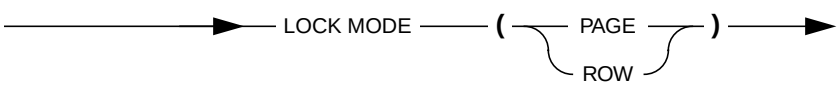


DROP CONSTRAINT Clause



MODIFY NEXT SIZE Clause

LOCK MODE Clause



+

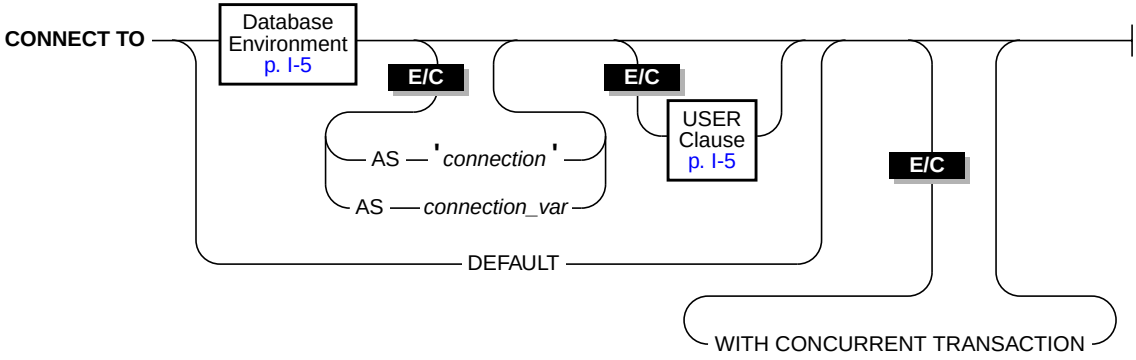
BEGIN WORK

CLOSE Cursor Name (p. I-31)

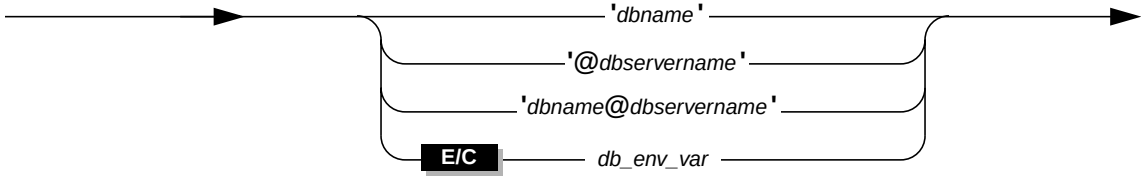
+

CLOSE DATABASE

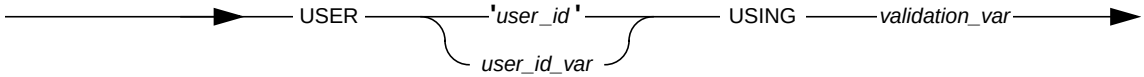
COMMIT WORK



Database Environment



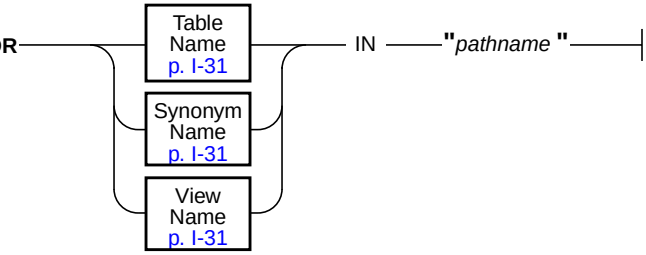
USER Clause



SE

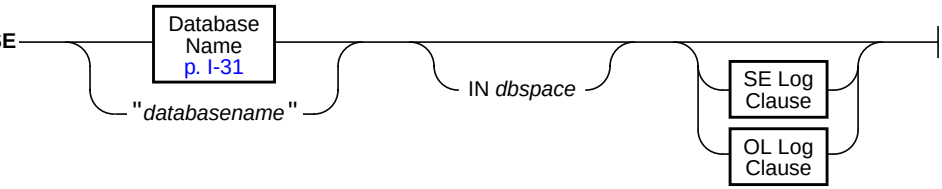
+

CREATE AUDIT FOR

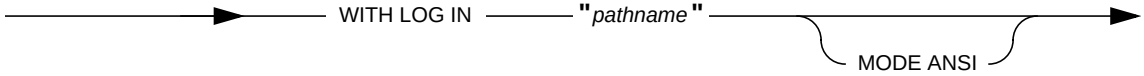


+

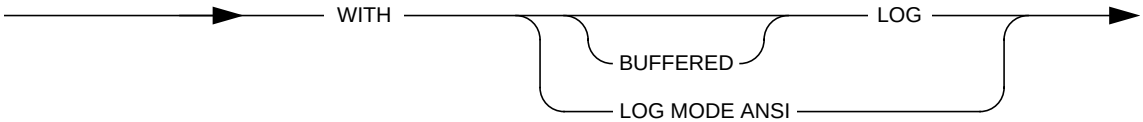
CREATE DATABASE



SE Log Clause

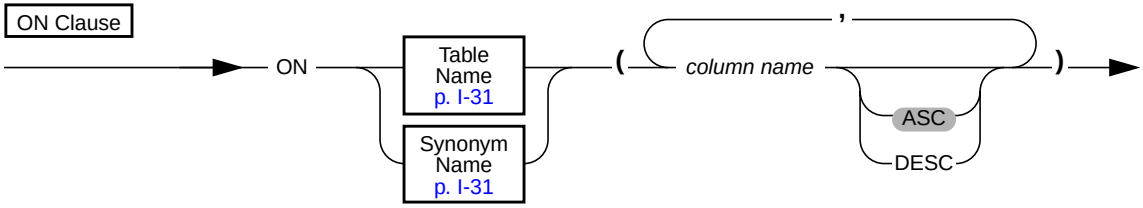


OL Log Clause

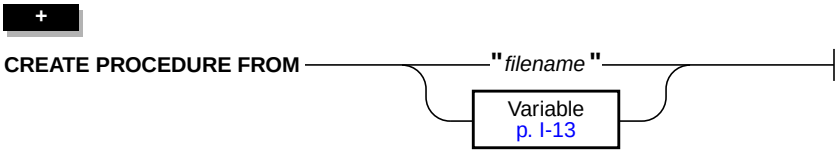


+

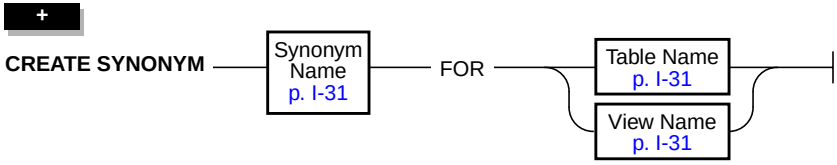
ON Clause

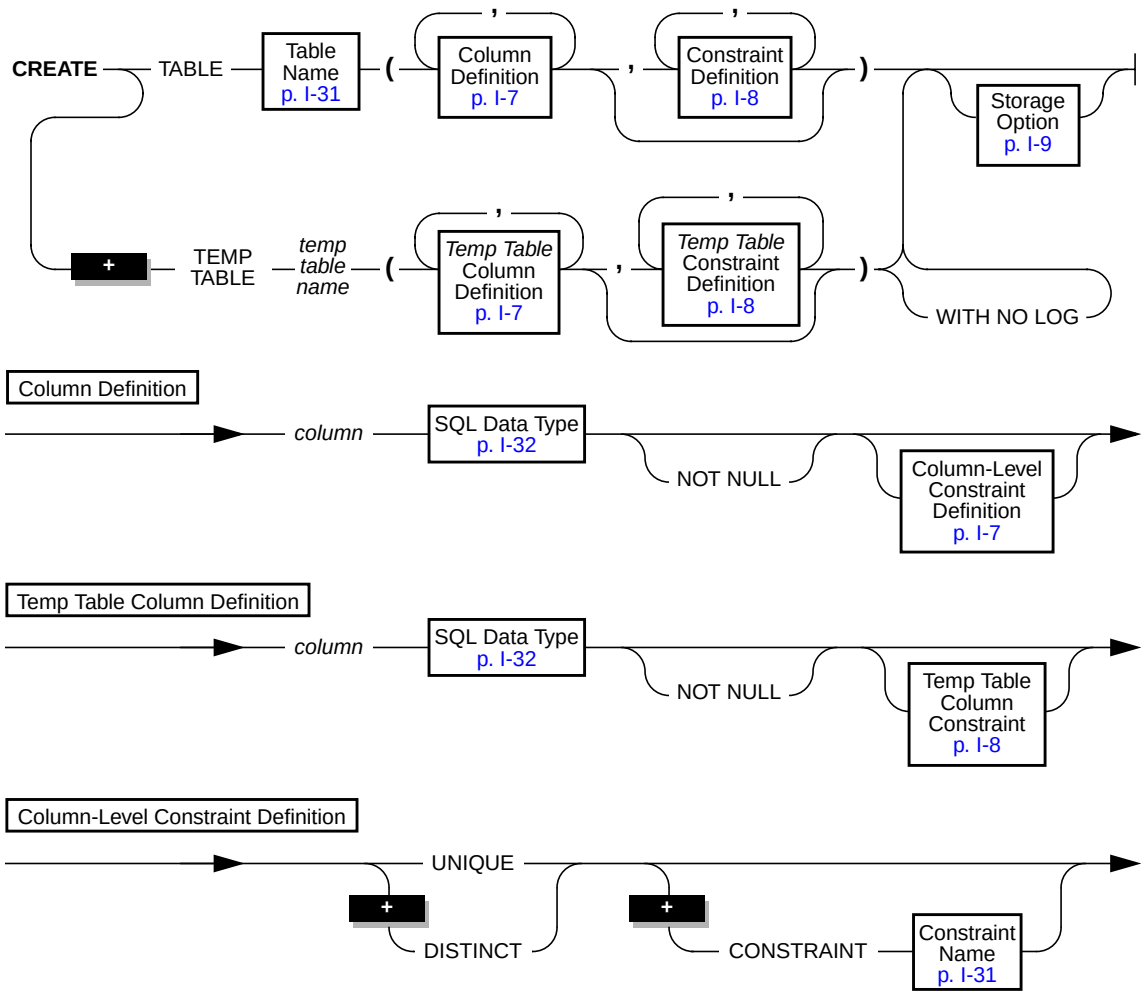


+

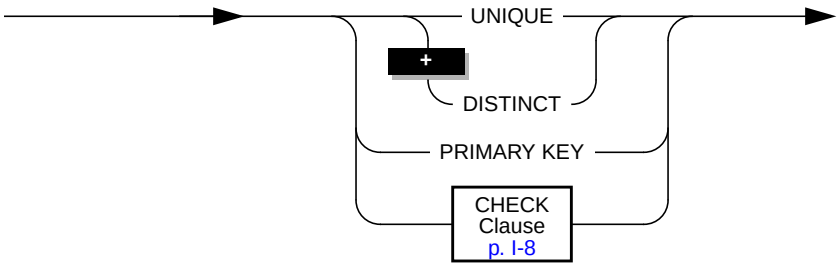


+

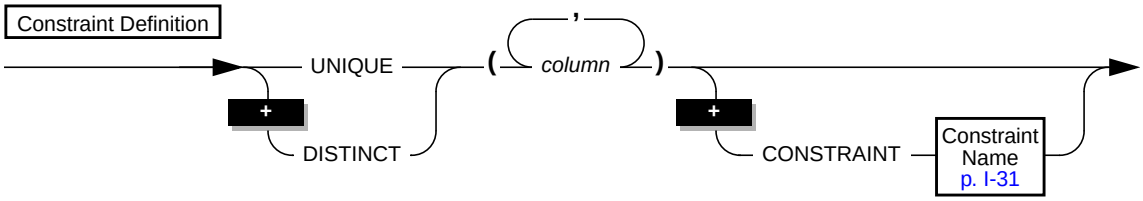



CREATE
TABLE
Table Name p. I-31
(
Column Definition p. I-7
,
Constraint Definition p. I-8
)
Storage Option p. I-9
+
TEMP TABLE
temp table name
Temp Table Column Definition p. I-7
Temp Table Constraint Definition p. I-8
WITH NO LOG
Column Definition
column
SQL Data Type p. I-32
NOT NULL
Column-Level Constraint Definition p. I-7
Temp Table Column Definition
Temp Table Column Constraint p. I-8
Column-Level Constraint Definition
UNIQUE
DISTINCT
CONSTRAINT
Constraint Name p. I-31

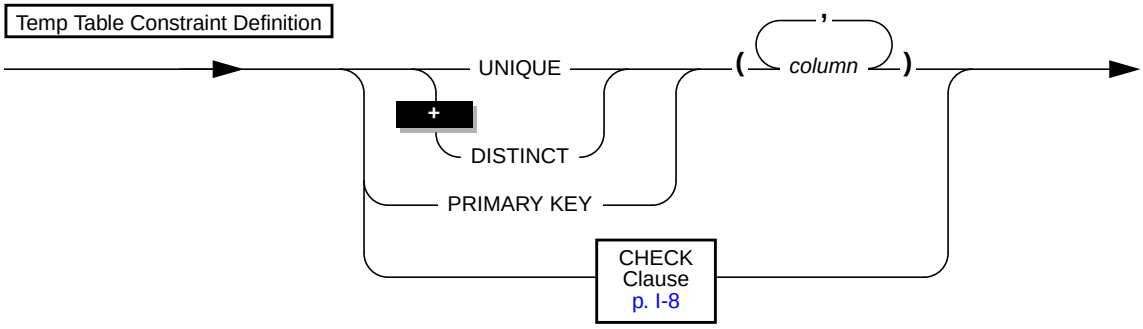
Temp Table Column Constraint



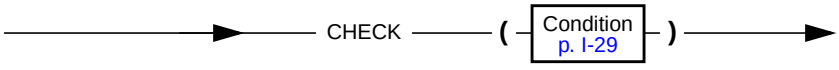
Constraint Definition



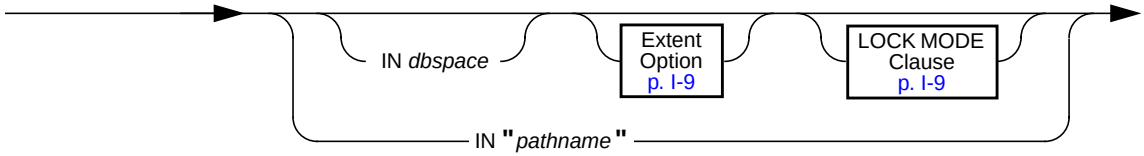
Temp Table Constraint Definition



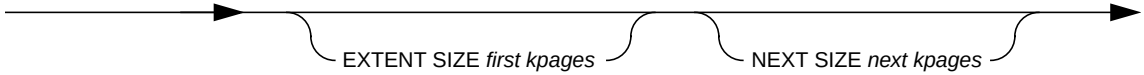
CHECK Clause



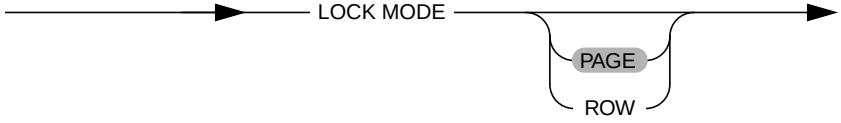
Storage Option

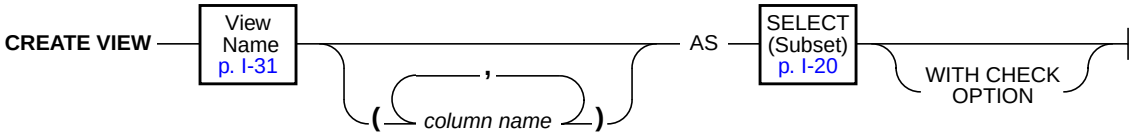


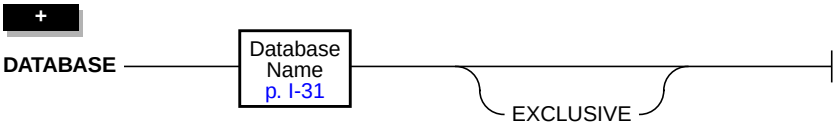
Extent Option

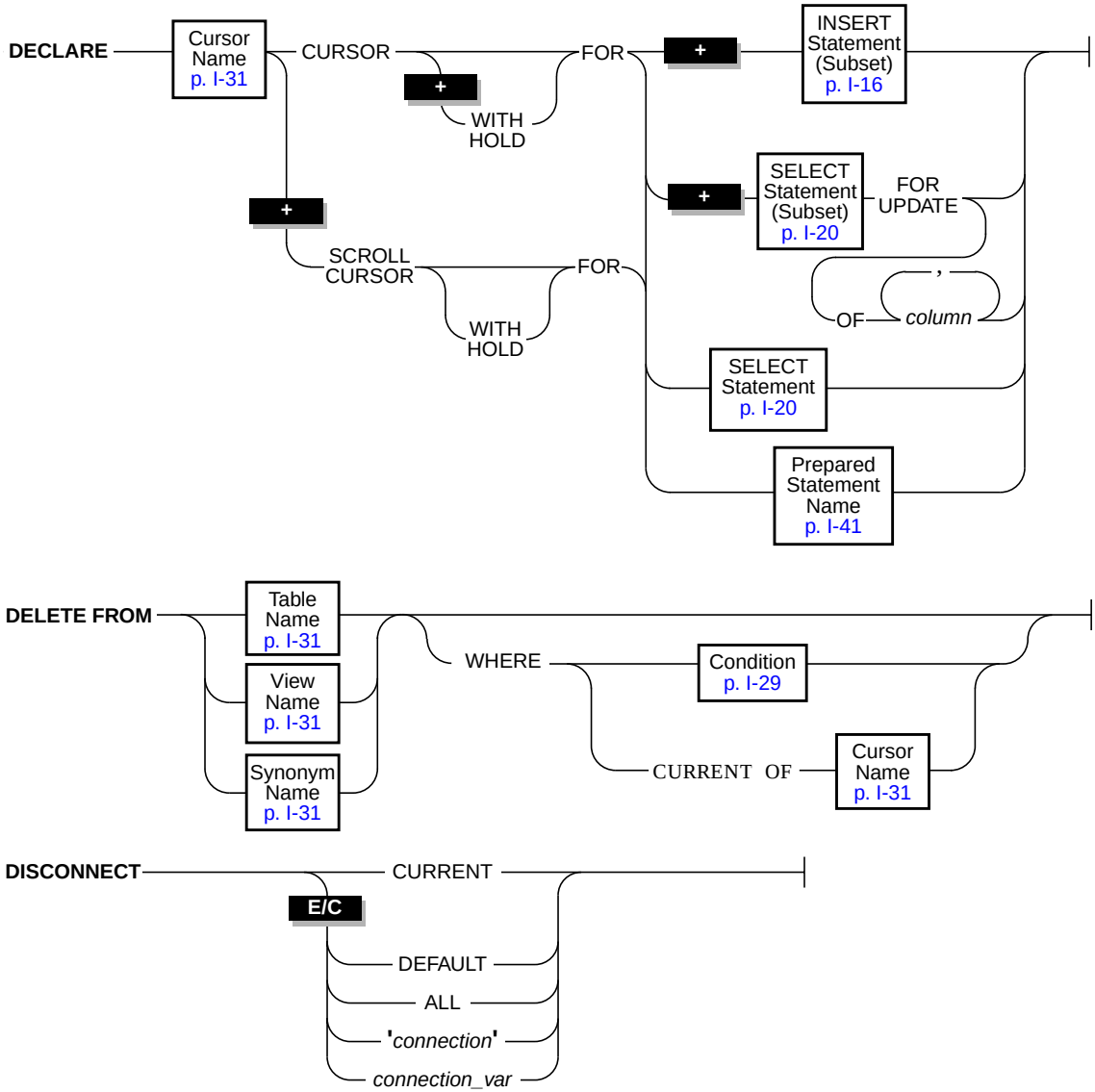


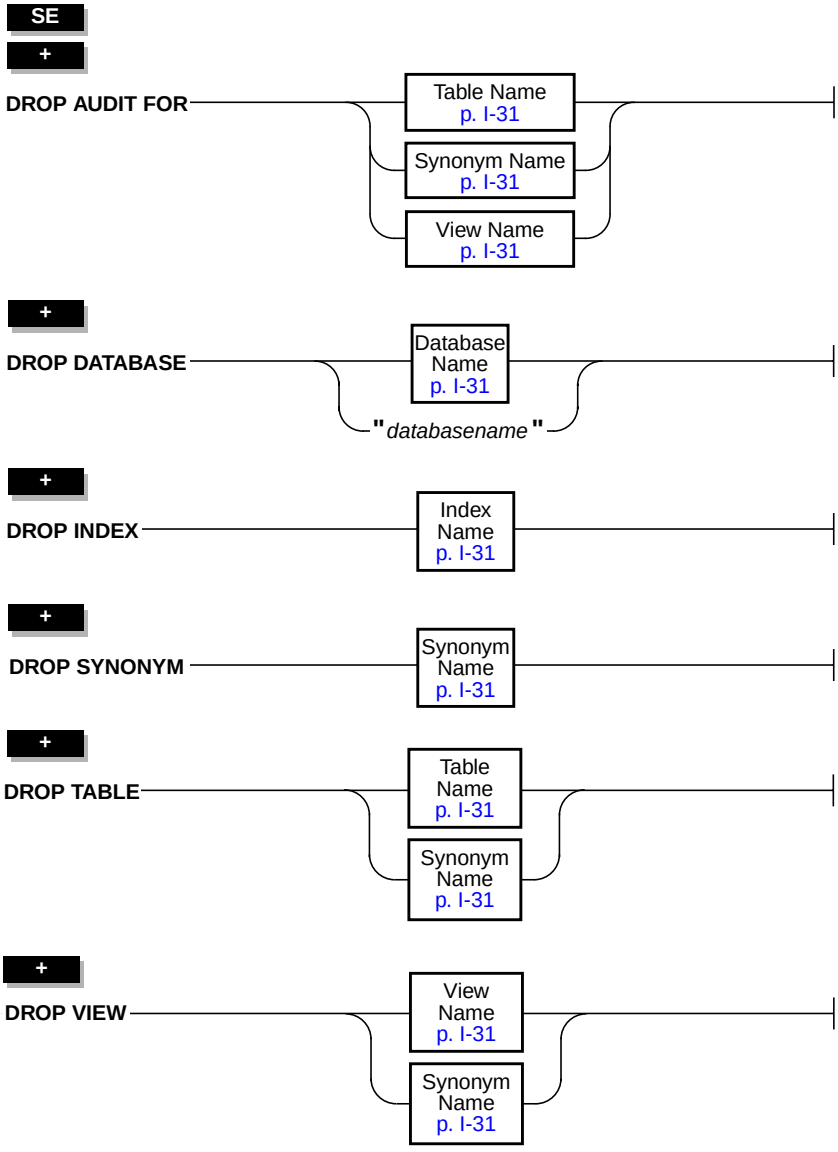
LOCK MODE Clause

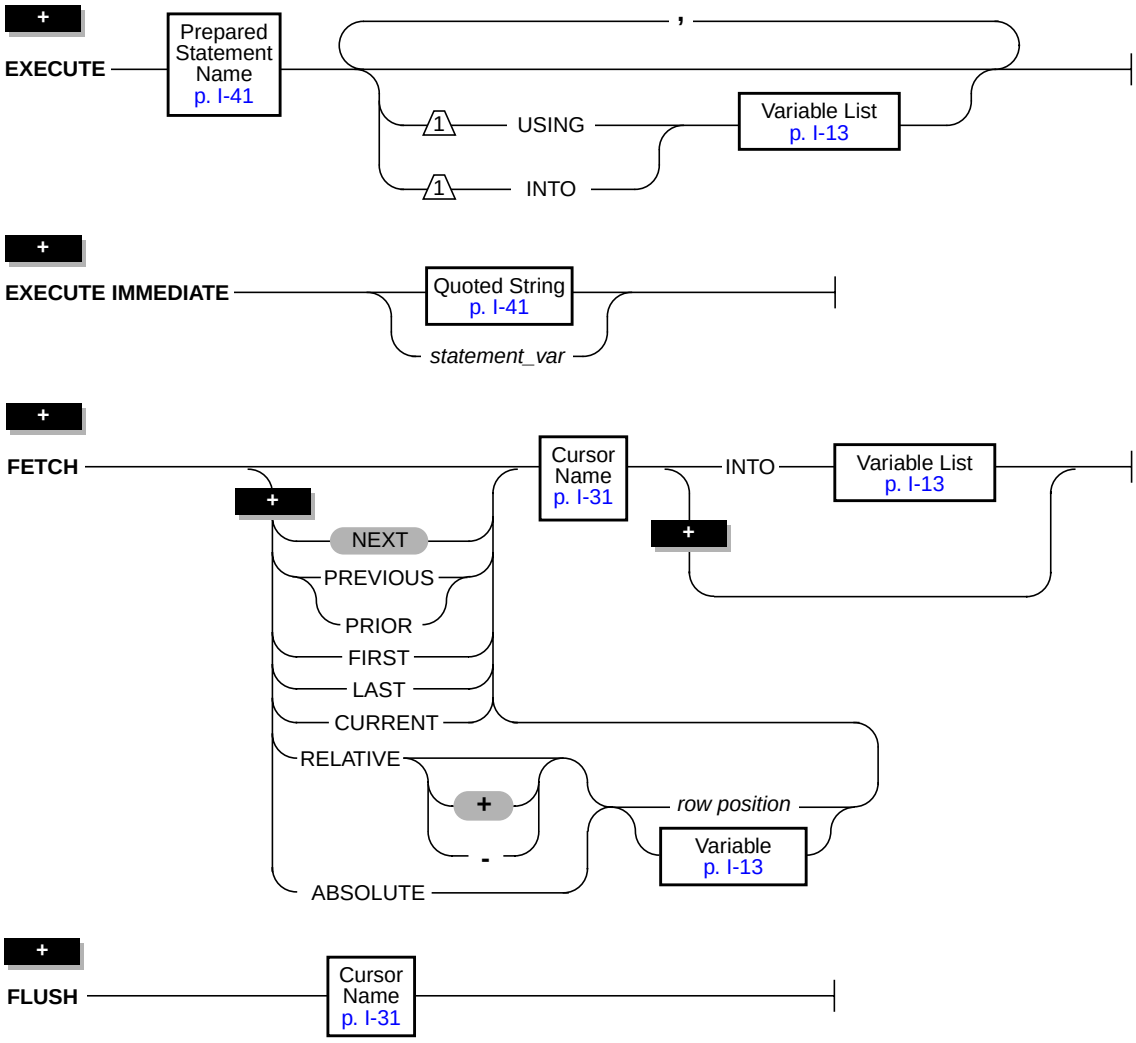


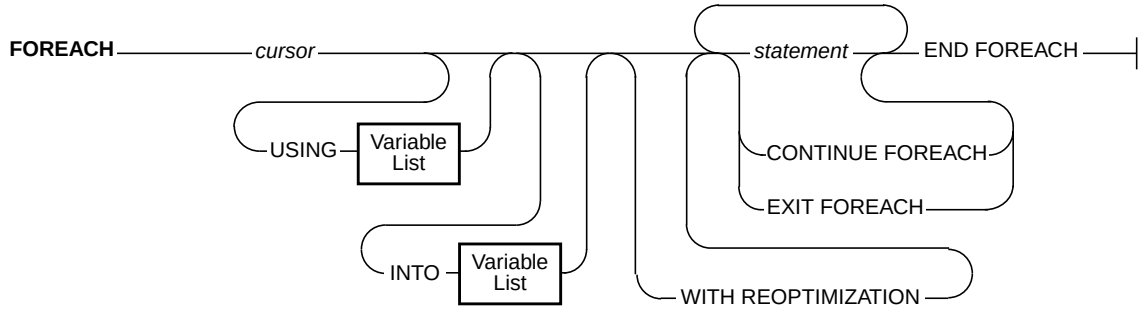


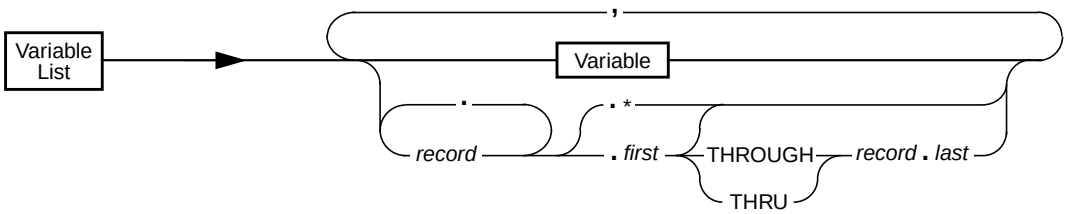


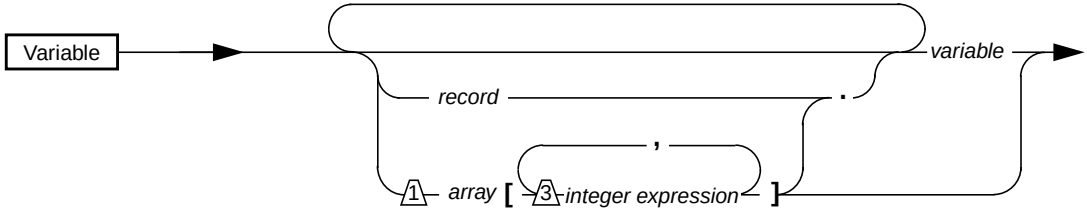

DECLARE
Cursor Name p. I-31
CURSOR
+
WITH HOLD
FOR
+
INSERT Statement (Subset) p. I-16
+
SELECT Statement (Subset) p. I-20
FOR UPDATE
OF
,
column
+
SCROLL CURSOR
WITH HOLD
FOR
SELECT Statement p. I-20
Prepared Statement Name p. I-41
DELETE FROM
Table Name p. I-31
View Name p. I-31
Synonym Name p. I-31
WHERE
Condition p. I-29
CURRENT OF
Cursor Name p. I-31
DISCONNECT
CURRENT
E/C
DEFAULT
ALL
'connection'
connection_var

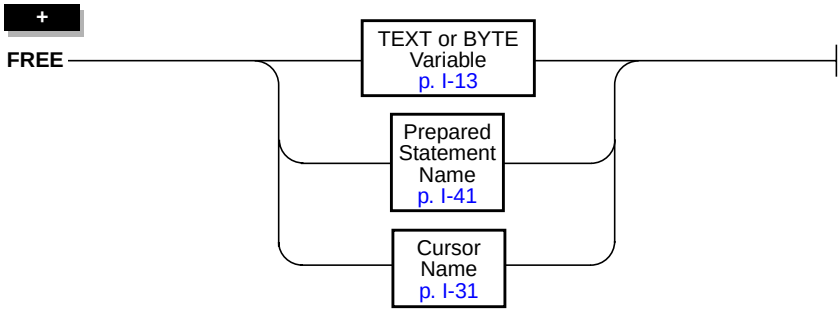

SE
+
DROP AUDIT FOR
Table Name
p. I-31
Synonym Name
p. I-31
View Name
p. I-31
+
DROP DATABASE
Database Name
p. I-31
"databasename"
+
DROP INDEX
Index Name
p. I-31
+
DROP SYNONYM
Synonym Name
p. I-31
+
DROP TABLE
Table Name
p. I-31
Synonym Name
p. I-31
+
DROP VIEW
View Name
p. I-31
Synonym Name
p. I-31


+
EXECUTE
Prepared Statement Name
p. I-41
,
1
USING
Variable List
p. I-13
1
INTO
+
EXECUTE IMMEDIATE
Quoted String
p. I-41
statement_var
+
FETCH
+
NEXT
PREVIOUS
PRIOR
FIRST
LAST
CURRENT
RELATIVE
ABSOLUTE
Cursor Name
p. I-31
INTO
Variable List
p. I-13
+
+
-
row position
Variable
p. I-13
+
FLUSH
Cursor Name
p. I-31


FOREACH
cursor
USING
Variable List
INTO
Variable List
statement
CONTINUE FOREACH
EXIT FOREACH
WITH REOPTIMIZATION
END FOREACH


Variable List
,
Variable
.
record
.*
.first
THROUGH
THRU
record.last


Variable
record
.
variable
,
1
array
[
3
integer expression
]



The 4GL compiler treats the name of the object to be freed in the order shown in the diagram. In other words, the compiler looks first for a TEXT or BYTE variable having the given name; if one exists, that is the object that is freed. If no TEXT or BYTE variable having that name exists, the compiler then looks for a prepared statement or a cursor having that name and frees that.

When a TEXT or BYTE variable has the same name as a prepared statement or cursor, you cannot free resources allocated to the prepared statement or to the cursor.

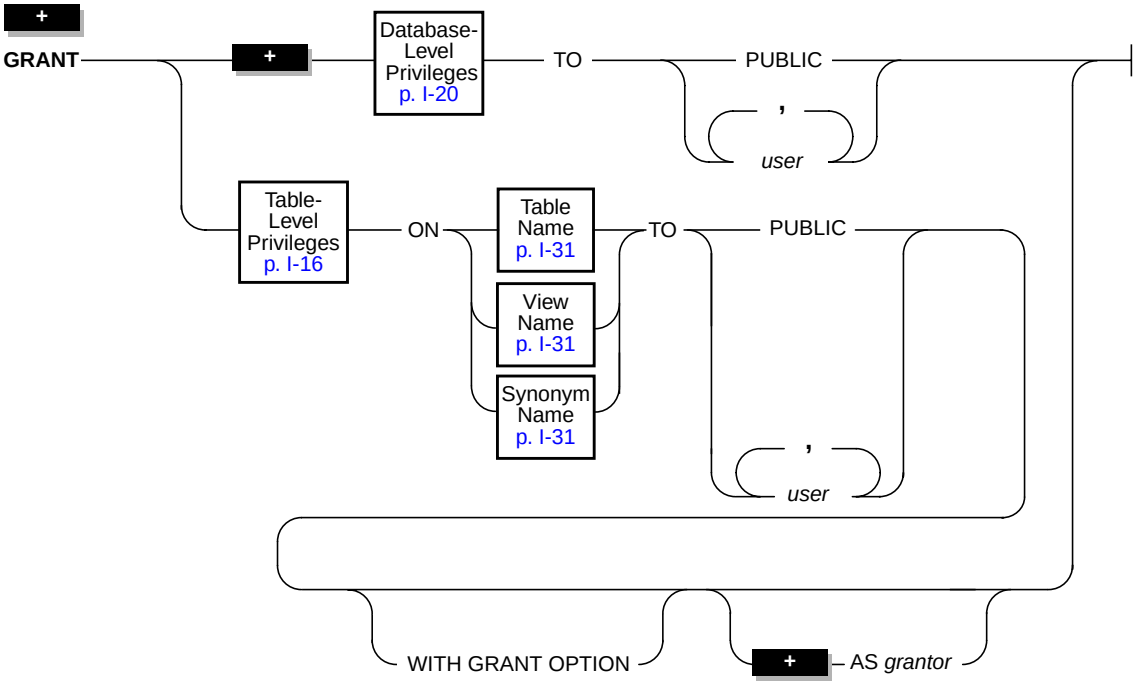
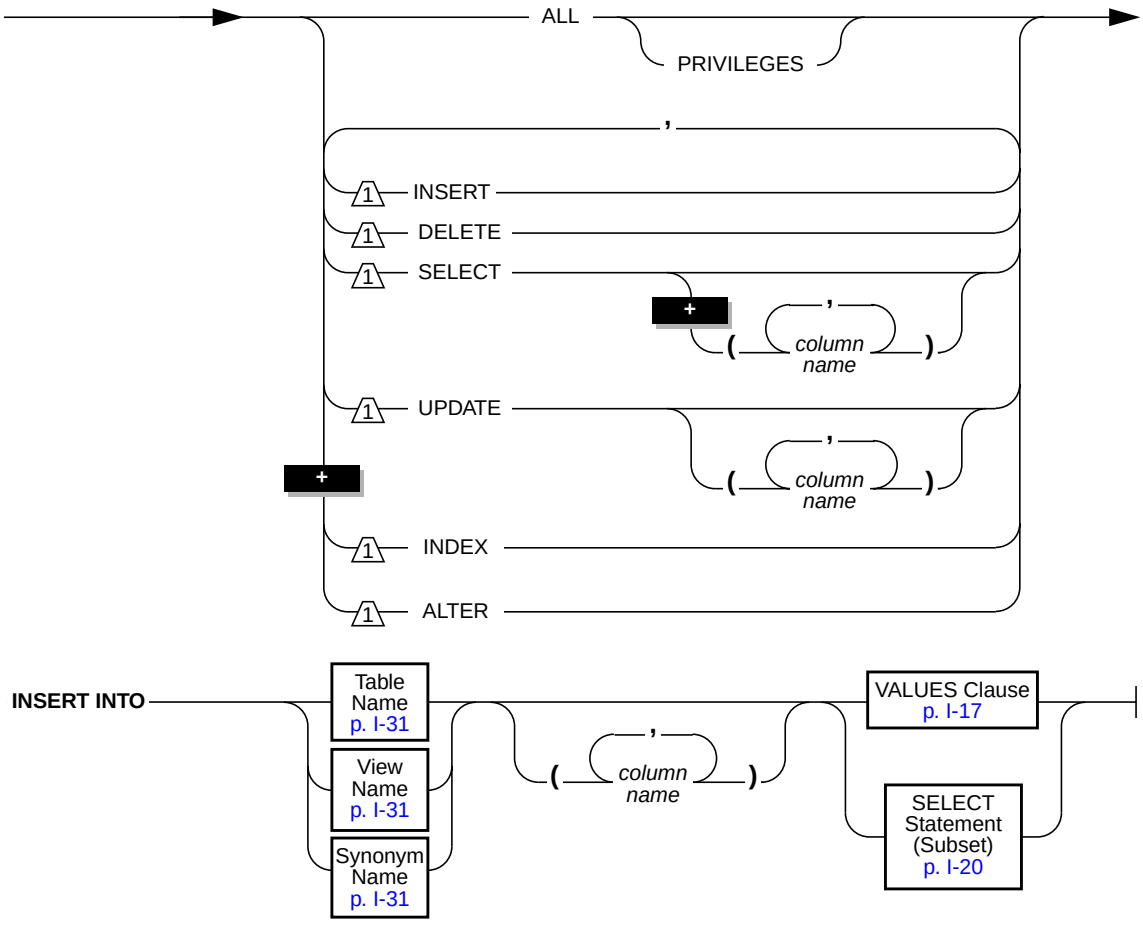
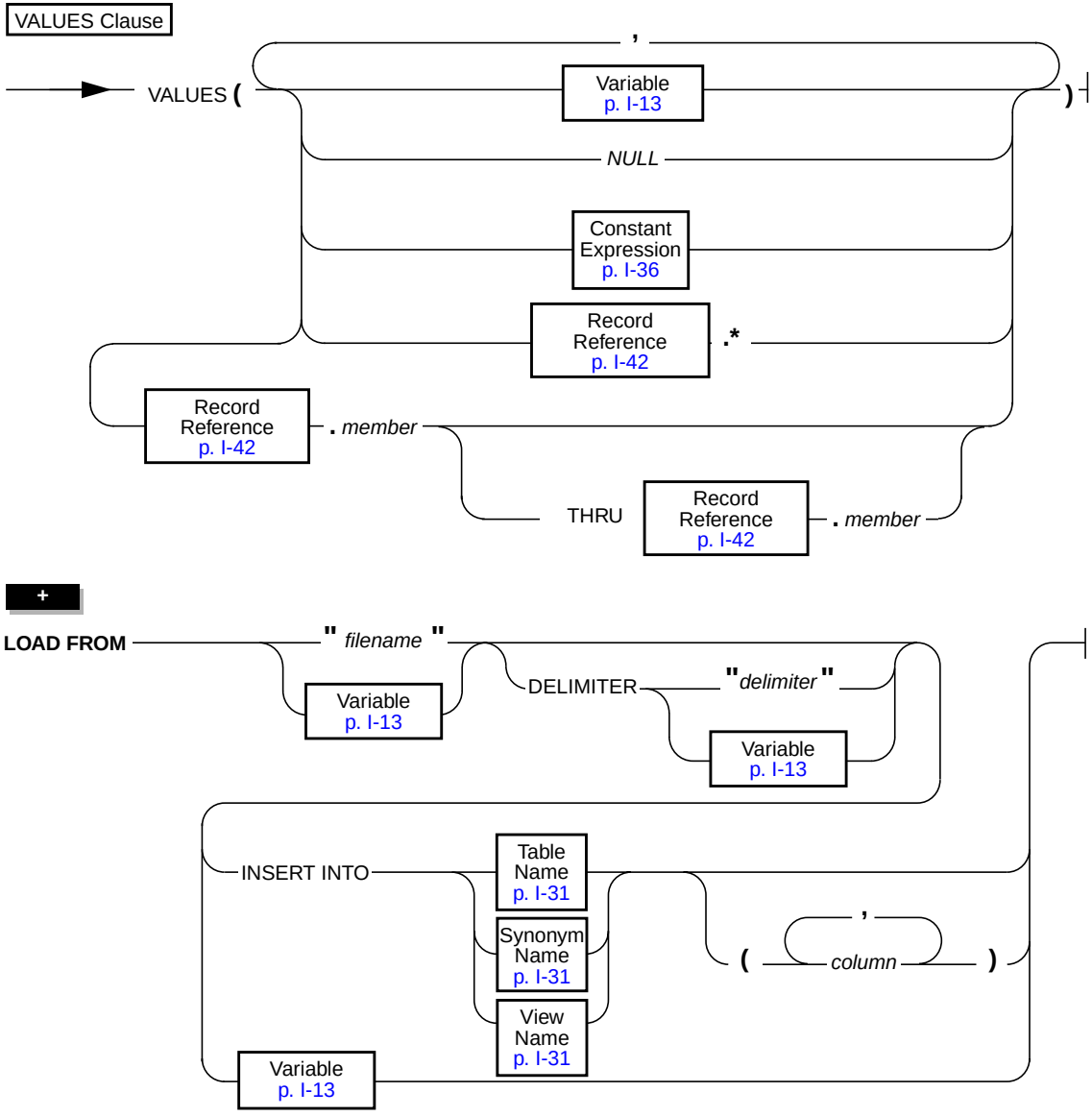
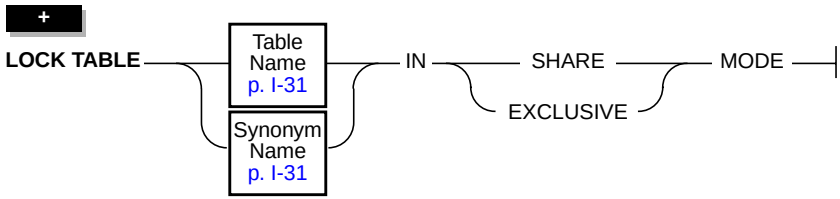
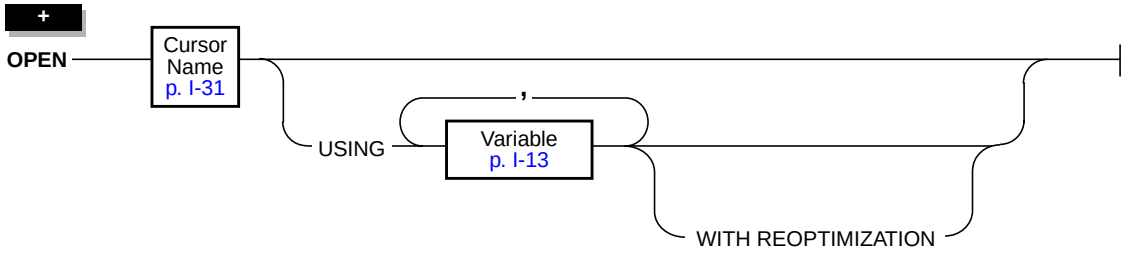

+
GRANT
+
Database-Level Privileges p. I-20
TO
PUBLIC
,
user
Table-Level Privileges p. I-16
ON
Table Name p. I-31
View Name p. I-31
Synonym Name p. I-31
TO
PUBLIC
,
user
WITH GRANT OPTION
+
AS grantor

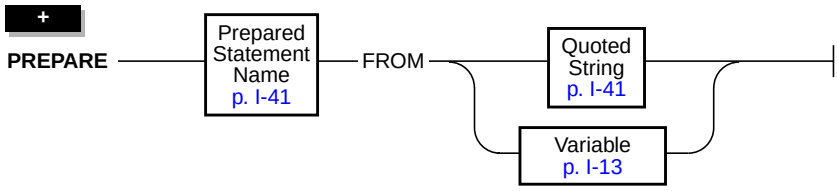
Table-Level Privileges

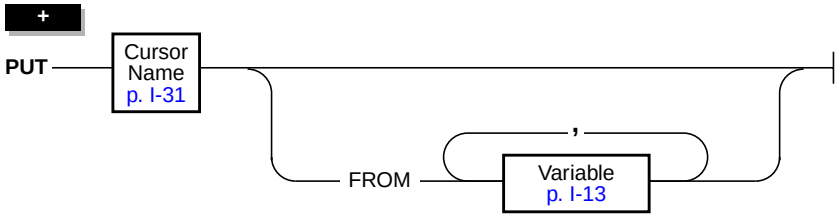


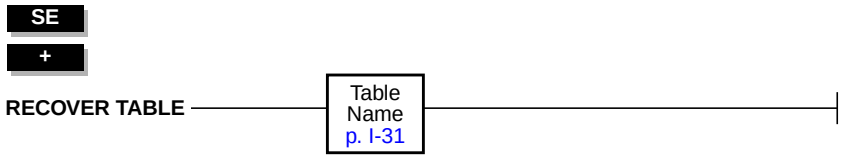

VALUES Clause
VALUES (
,
Variable
p. I-13
)
NULL
Constant
Expression
p. I-36
Record
Reference
p. I-42
.*
Record
Reference
p. I-42
. member
THRU
Record
Reference
p. I-42
. member
+
LOAD FROM
" filename "
Variable
p. I-13
DELIMITER
"delimiter"
Variable
p. I-13
INSERT INTO
Table
Name
p. I-31
Synonym
Name
p. I-31
View
Name
p. I-31
Variable
p. I-13
(
,
column
)

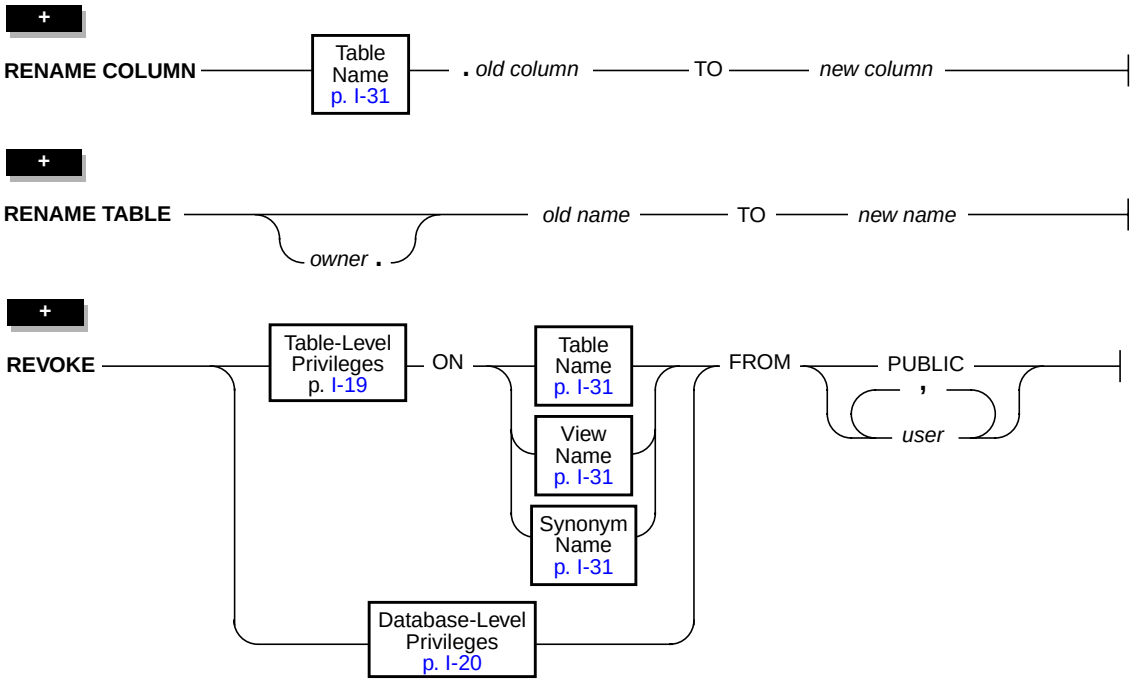

+
LOCK TABLE
Table Name p. I-31
Synonym Name p. I-31
IN
SHARE
EXCLUSIVE
MODE


+
OPEN
Cursor Name p. I-31
USING
,
Variable p. I-13
WITH REOPTIMIZATION


+
PREPARE
Prepared Statement Name p. I-41
FROM
Quoted String p. I-41
Variable p. I-13


+
PUT
Cursor Name p. I-31
FROM
,
Variable p. I-13


SE
+
RECOVER TABLE
Table Name p. I-31


+
RENAME COLUMN
Table Name p. I-31
. old column
TO
new column
+
RENAME TABLE
owner .
old name
TO
new name
+
REVOKE
Table-Level Privileges p. I-19
ON
Table Name p. I-31
View Name p. I-31
Synonym Name p. I-31
Database-Level Privileges p. I-20
FROM
PUBLIC
,
user

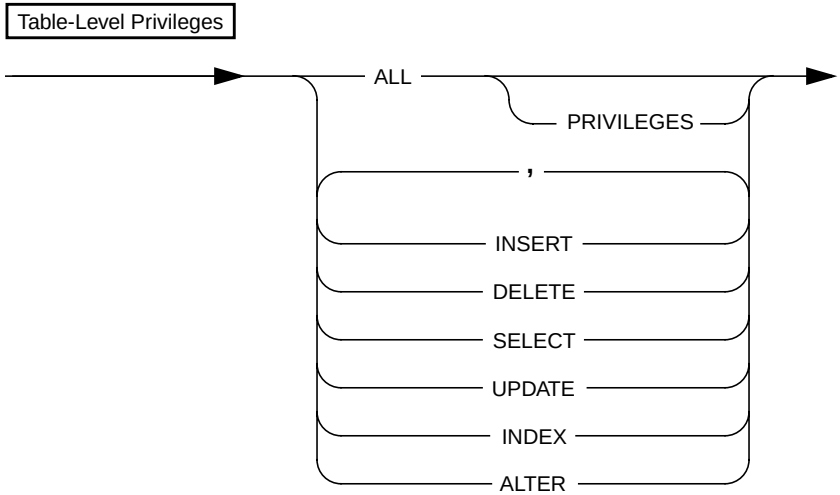
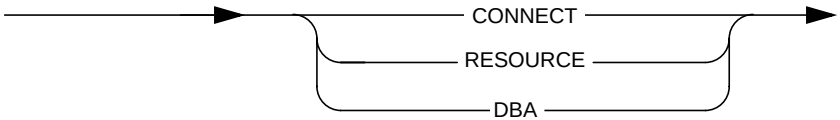
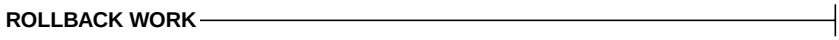


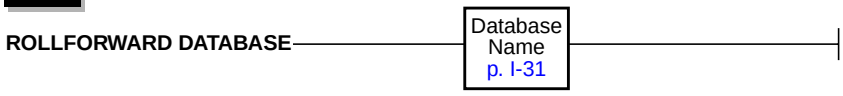
Table-Level Privileges
ALL
PRIVILEGES
,
INSERT
DELETE
SELECT
UPDATE
INDEX
ALTER

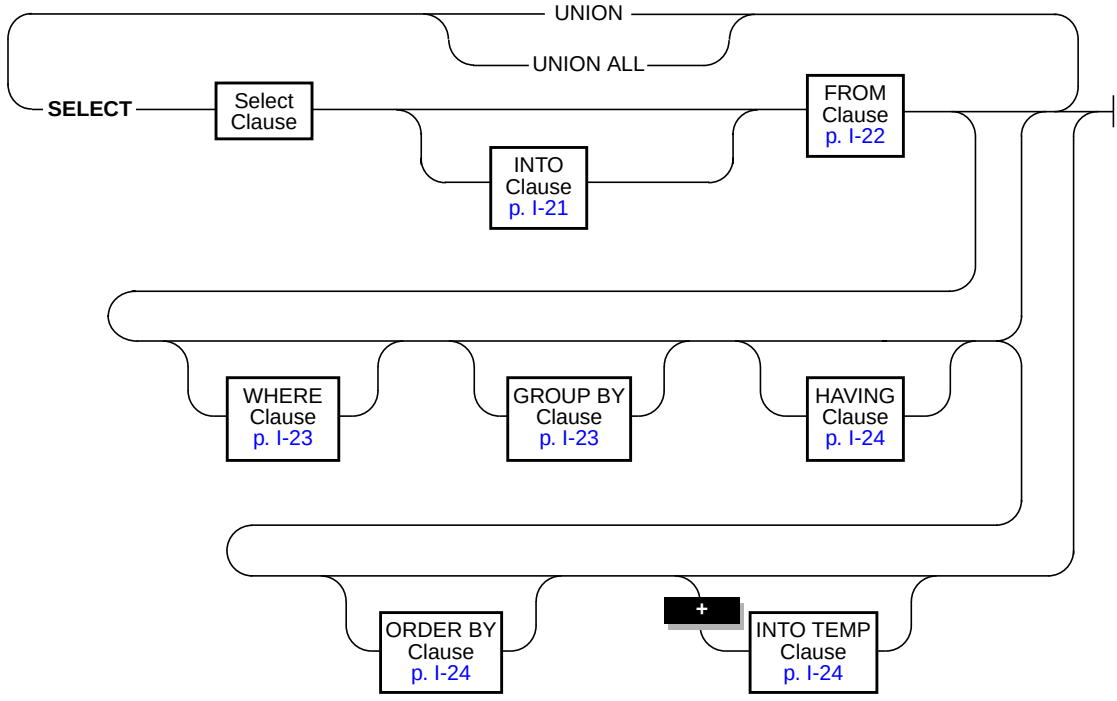
Database-Level Privileges

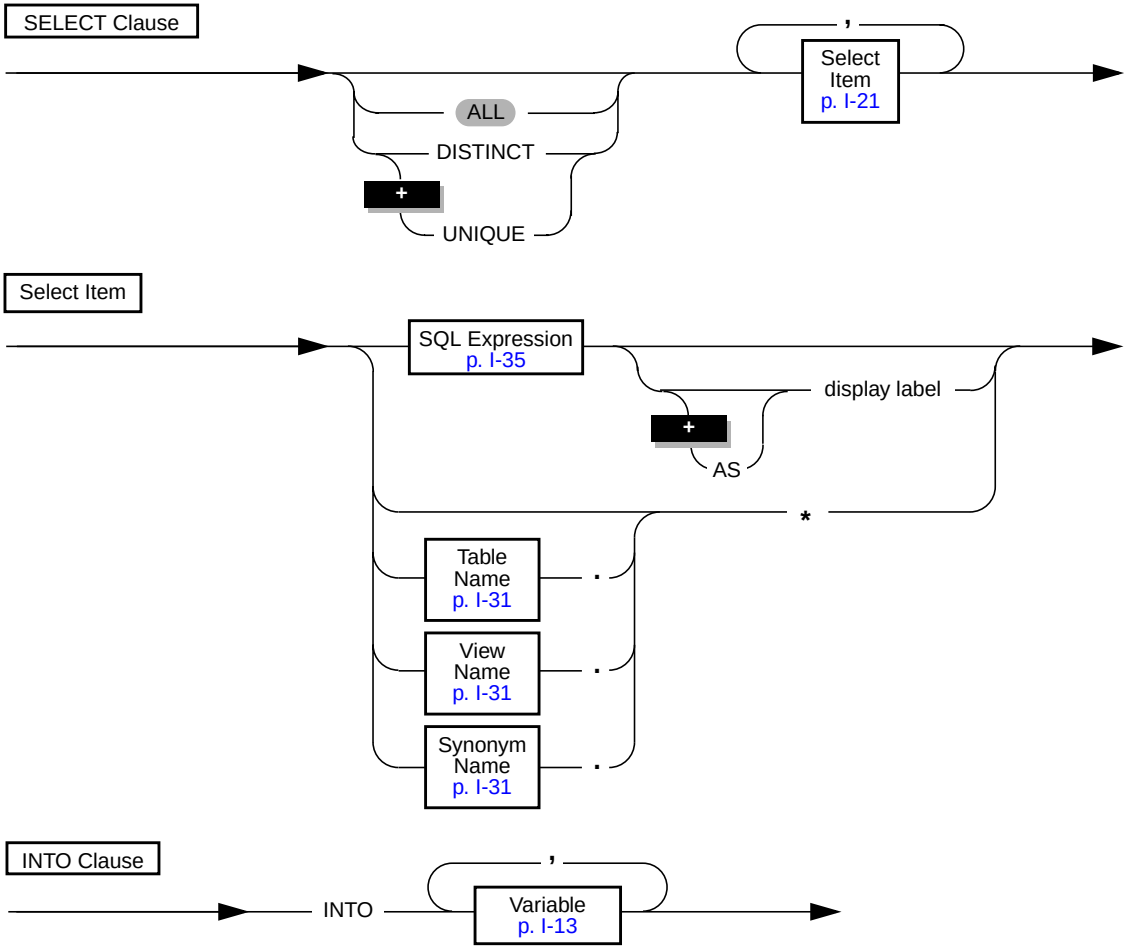




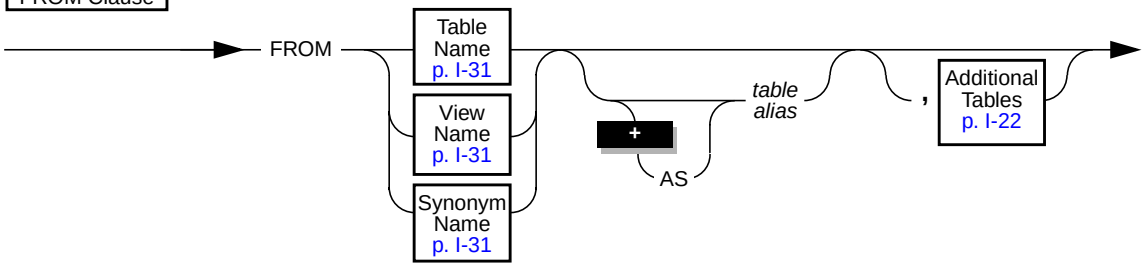
SE
+



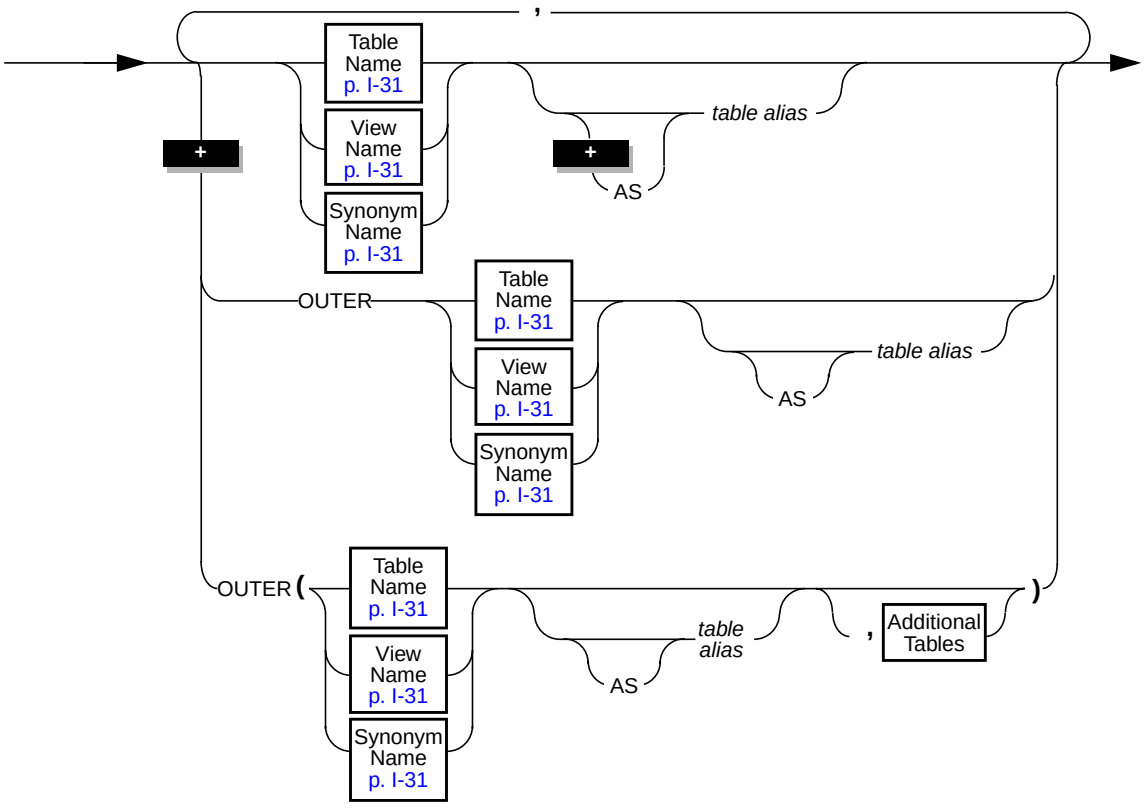


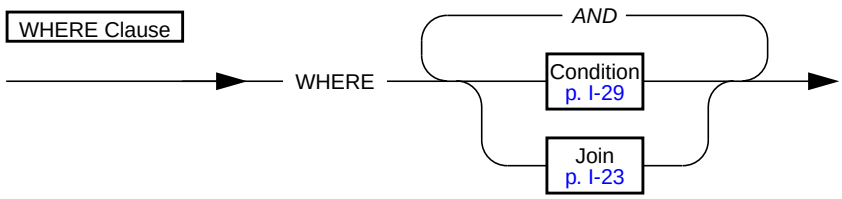

SELECT Clause
,
Select Item
p. I-21
ALL
DISTINCT
+
UNIQUE
Select Item
SQL Expression
p. I-35
display label
+
AS
*
Table Name
p. I-31
.
View Name
p. I-31
.
Synonym Name
p. I-31
.
INTO Clause
,
INTO
Variable
p. I-13

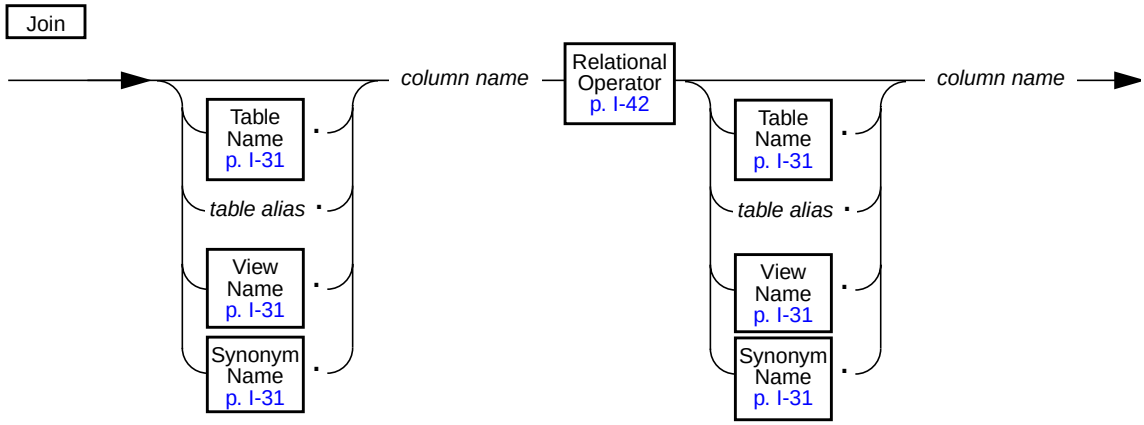
FROM Clause

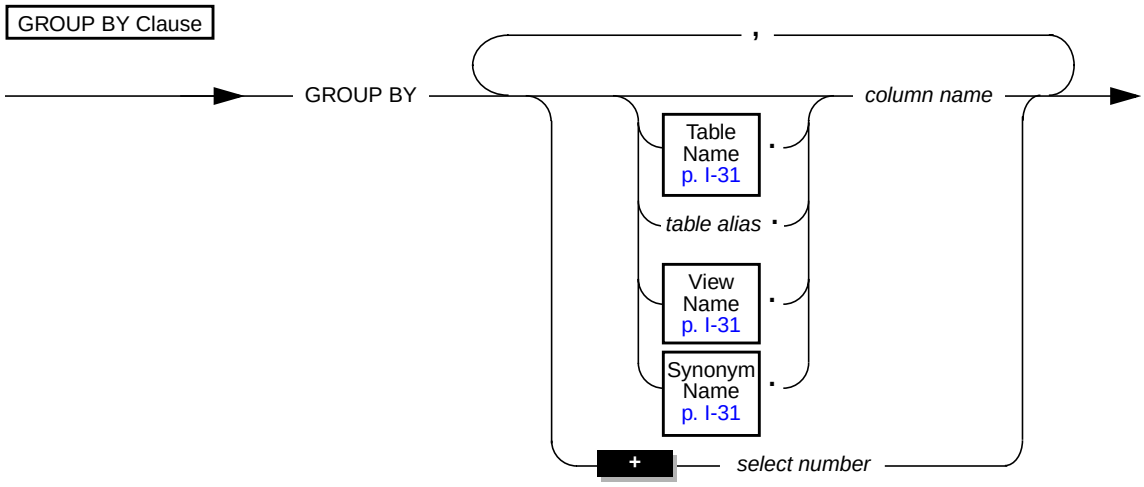


Additional Tables

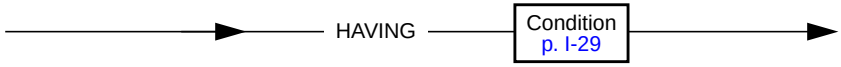



WHERE Clause
WHERE
AND
Condition
p. I-29
Join
p. I-23

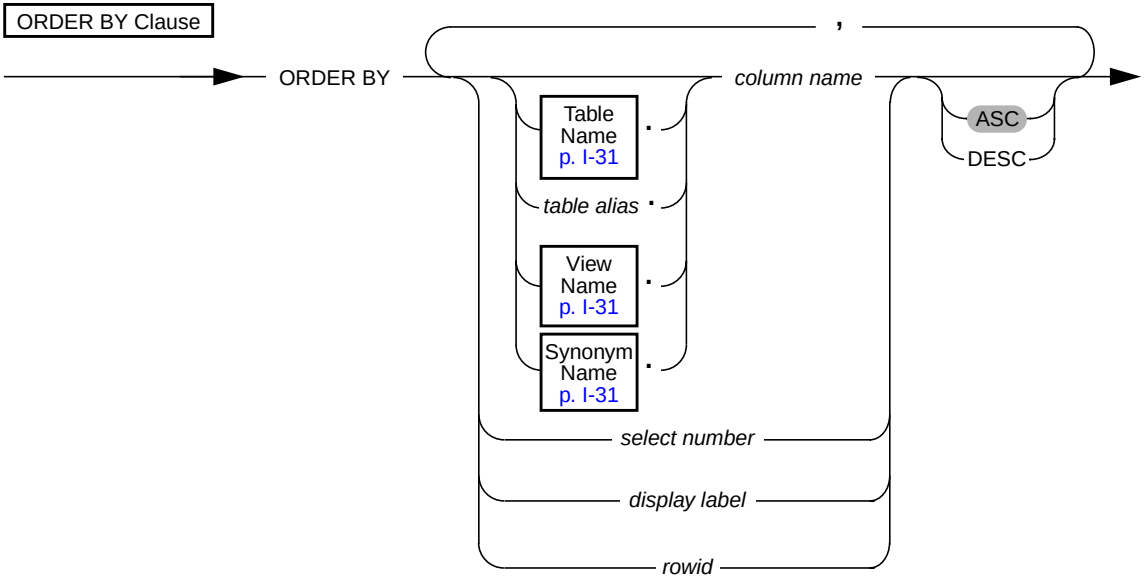

Join
column name
Relational Operator
p. I-42
column name
Table Name
p. I-31
table alias
View Name
p. I-31
Synonym Name
p. I-31
Table Name
p. I-31
table alias
View Name
p. I-31
Synonym Name
p. I-31


GROUP BY Clause
GROUP BY
,
column name
Table Name
p. I-31
table alias
View Name
p. I-31
Synonym Name
p. I-31
+
select number

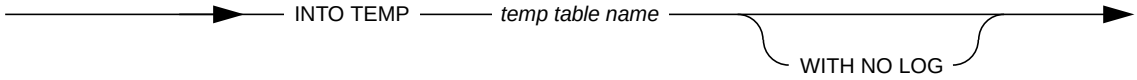
HAVING Clause

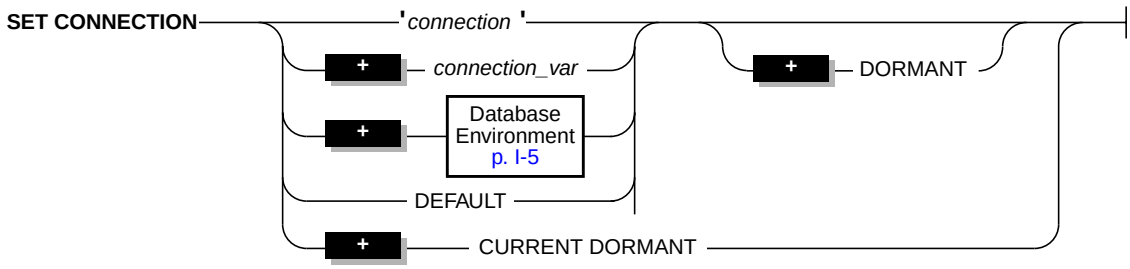


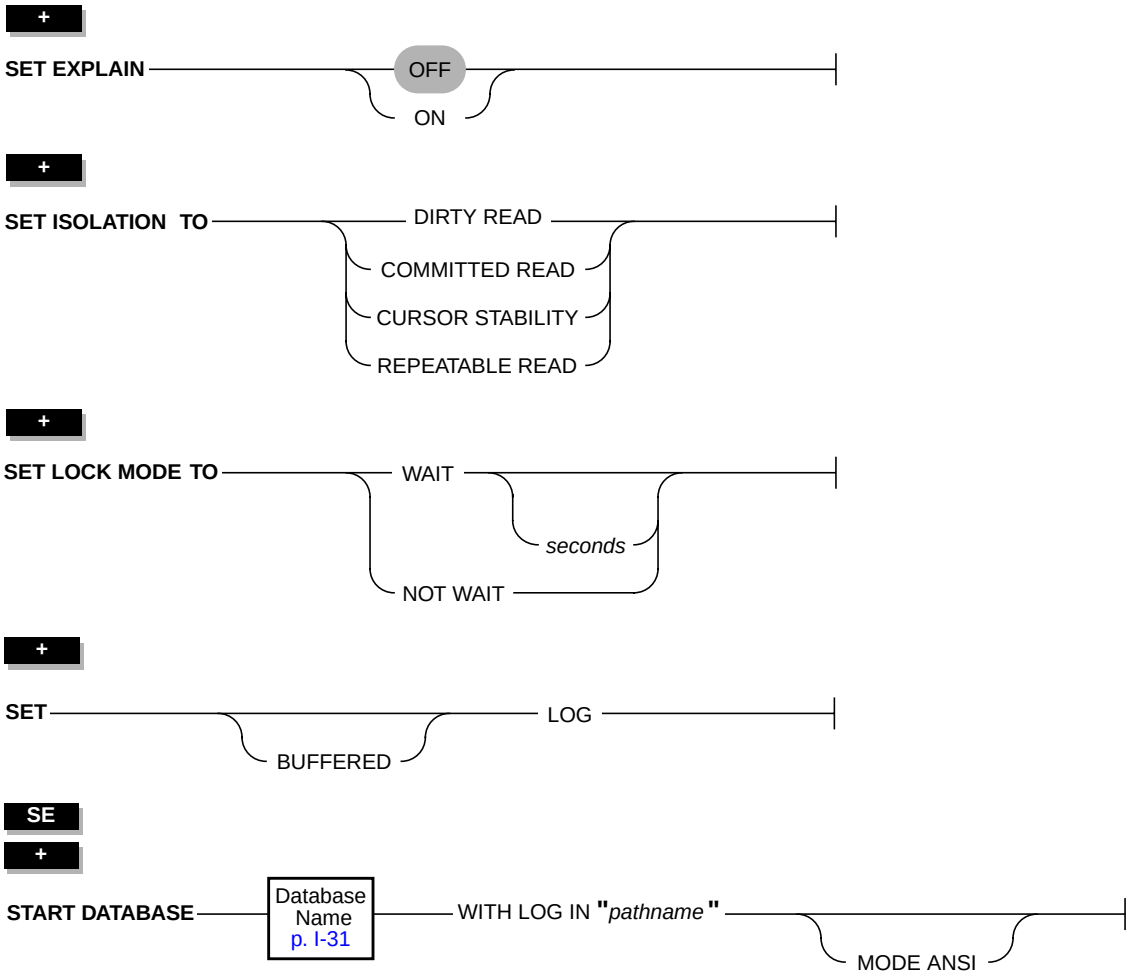
ORDER BY Clause

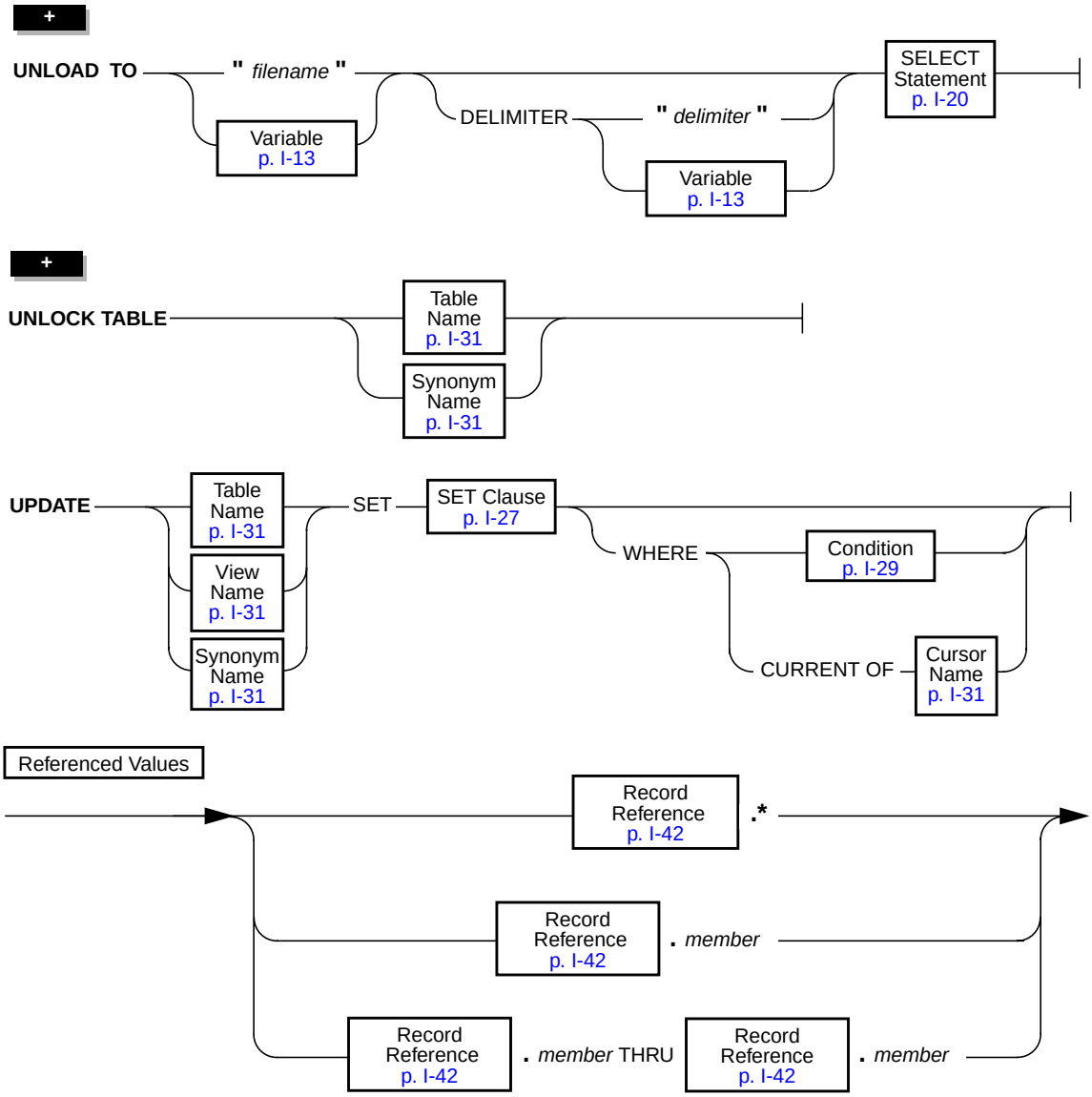


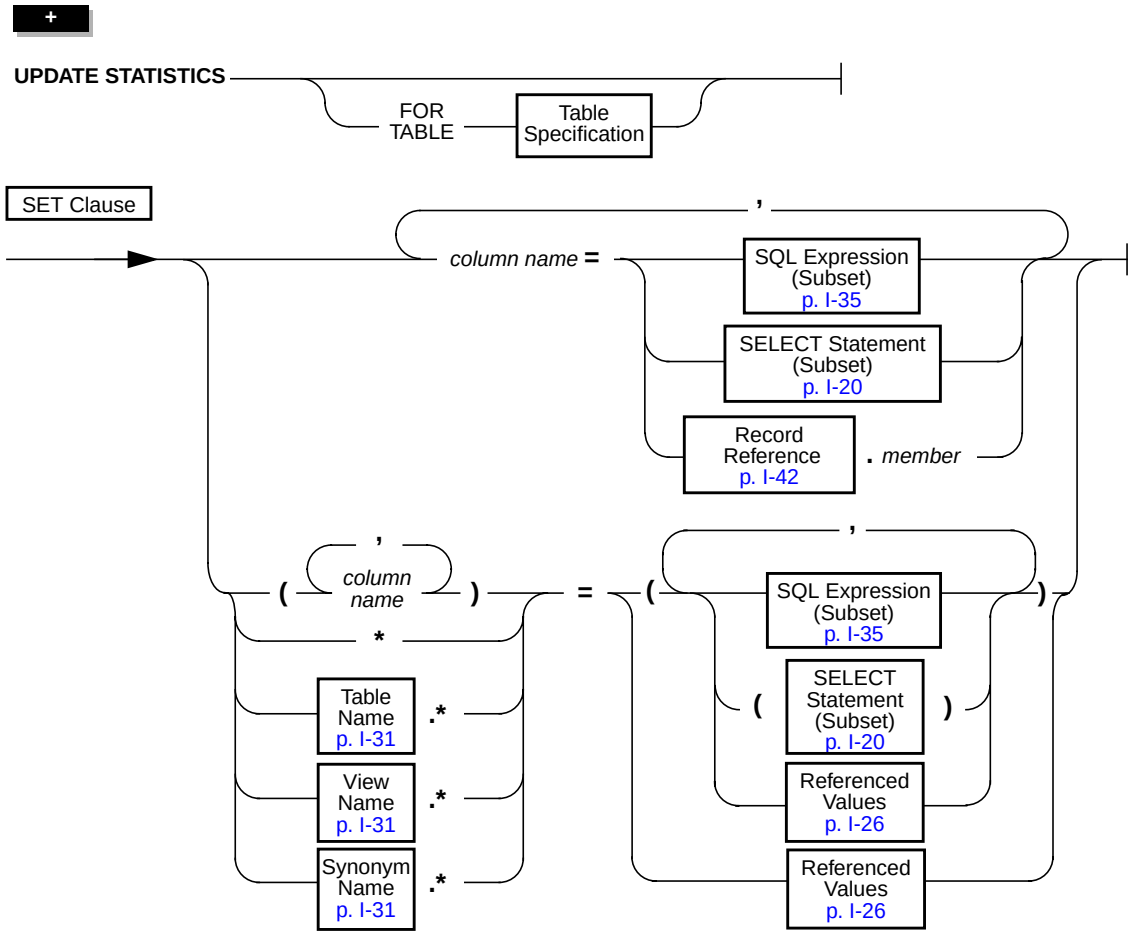
INTO TEMP Clause






+
SET EXPLAIN
OFF
ON
+
SET ISOLATION TO
DIRTY READ
COMMITTED READ
CURSOR STABILITY
REPEATABLE READ
+
SET LOCK MODE TO
WAIT
seconds
NOT WAIT
+
SET
BUFFERED
LOG
SE
+
START DATABASE
Database Name
p. I-31
WITH LOG IN "pathname"
MODE ANSI


+
UNLOAD TO
" filename "
Variable
p. I-13
DELIMITER
" delimiter "
Variable
p. I-13
SELECT
Statement
p. I-20
+
UNLOCK TABLE
Table
Name
p. I-31
Synonym
Name
p. I-31
UPDATE
Table
Name
p. I-31
View
Name
p. I-31
Synonym
Name
p. I-31
SET
SET Clause
p. I-27
WHERE
Condition
p. I-29
CURRENT OF
Cursor
Name
p. I-31
Referenced Values
Record
Reference
p. I-42
.*
Record
Reference
p. I-42
. member
Record
Reference
p. I-42
. member THRU
Record
Reference
p. I-42
. member


+
UPDATE STATISTICS
FOR TABLE
Table Specification
SET Clause
column name =
SQL Expression (Subset) p. I-35
SELECT Statement (Subset) p. I-20
Record Reference p. I-42
. member
column name
*
Table Name p. I-31 .*
View Name p. I-31 .*
Synonym Name p. I-31 .*
SQL Expression (Subset) p. I-35
SELECT Statement (Subset) p. I-20
Referenced Values p. I-26
Referenced Values p. I-26

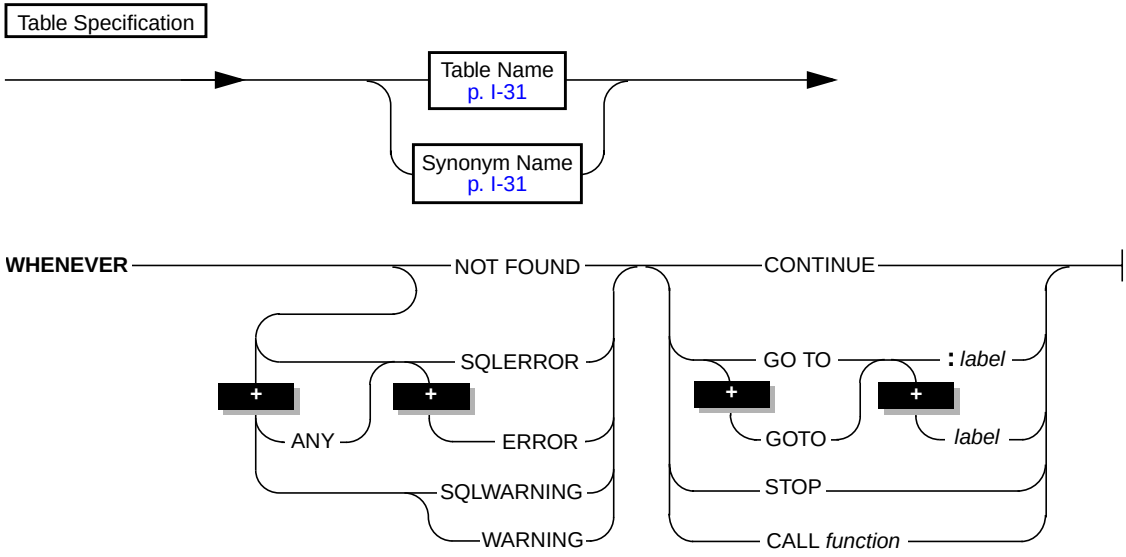
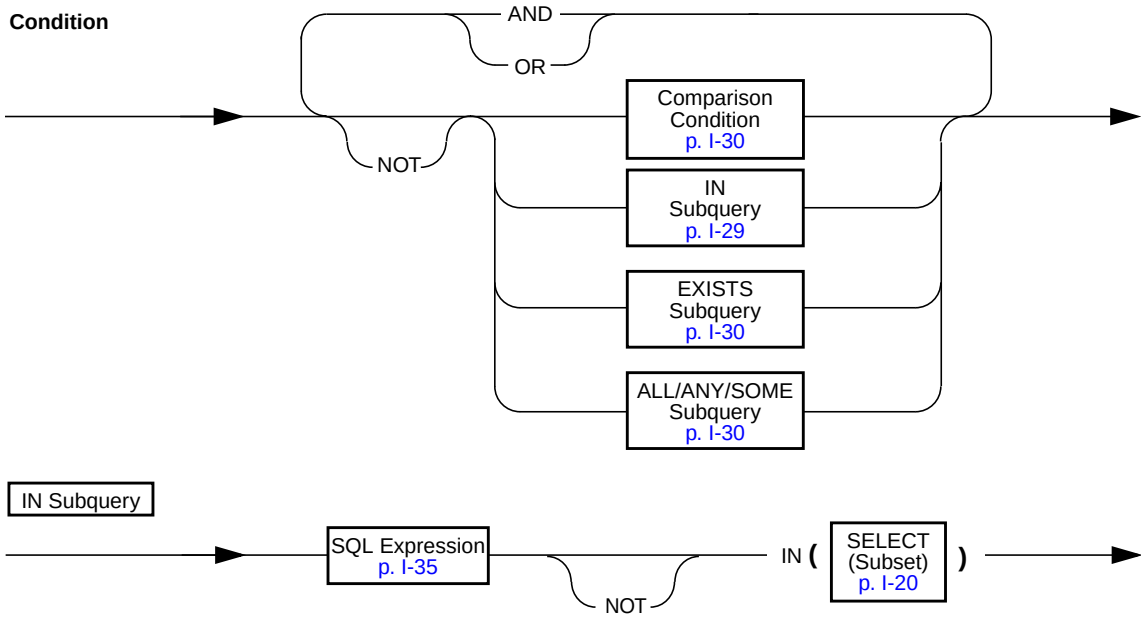


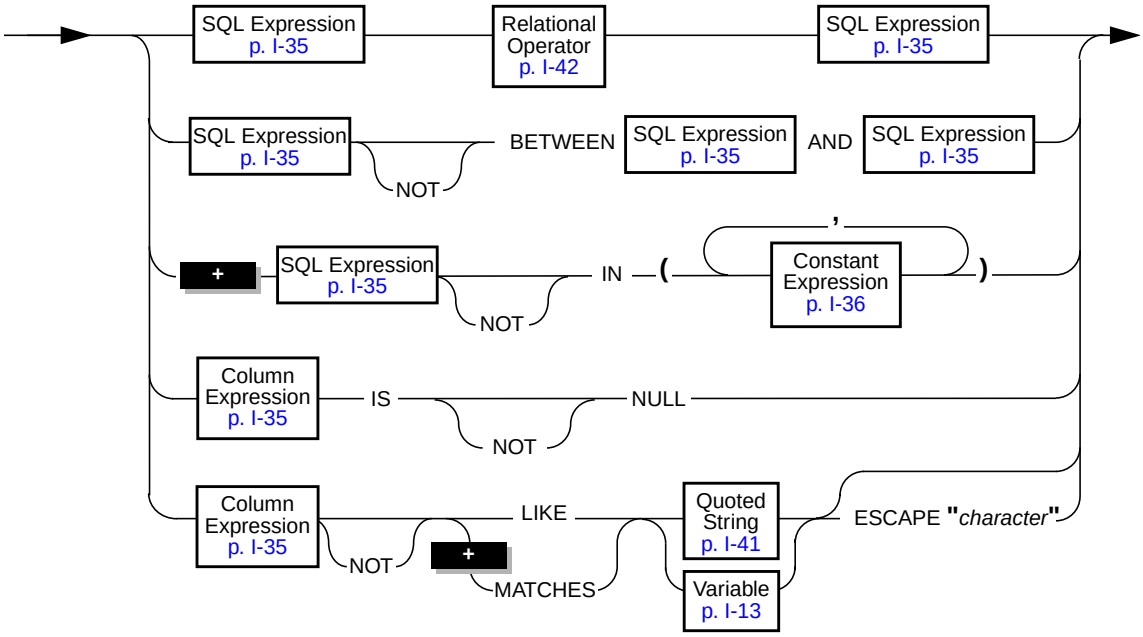
Table Specification
Table Name
p. I-31
Synonym Name
p. I-31
WHENEVER
NOT FOUND
CONTINUE
SQLERROR
GO TO
: label
ANY
ERROR
GOTO
label
SQLWARNING
STOP
WARNING
CALL function

Embedded SQL SEGMENTS

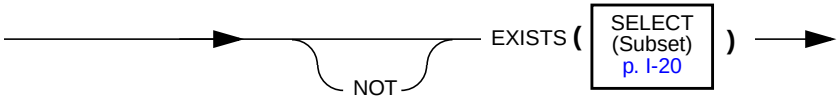
The following SQL segments can be embedded directly in INFORMIX-4GL code.



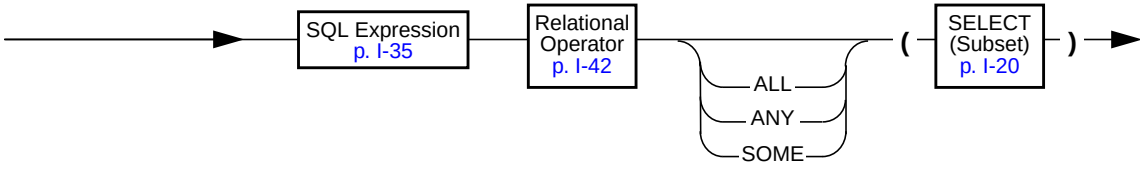
Comparison Condition



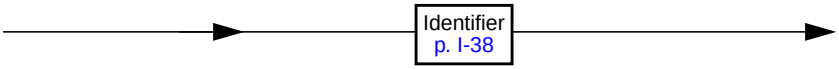
EXISTS Subquery



ALL/ANY/SOME Subquery



Cursor Name



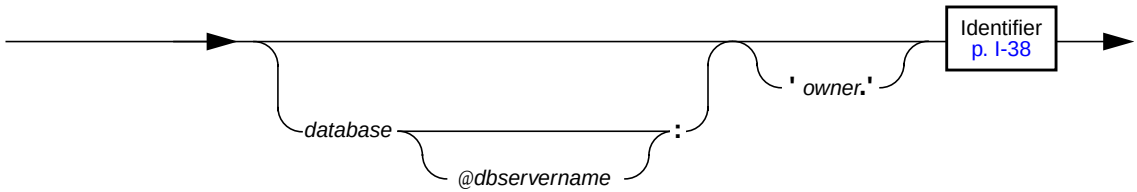
Constraint Name

Index Name

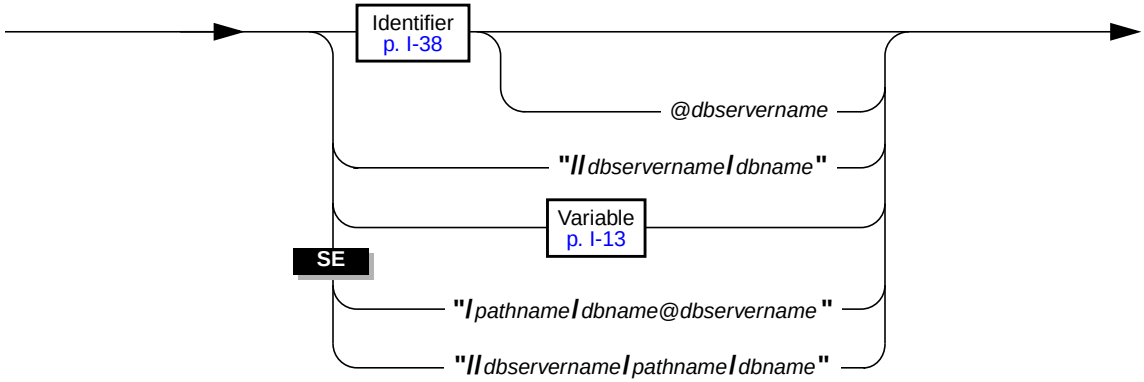
Synonym Name

Table Name

View Name



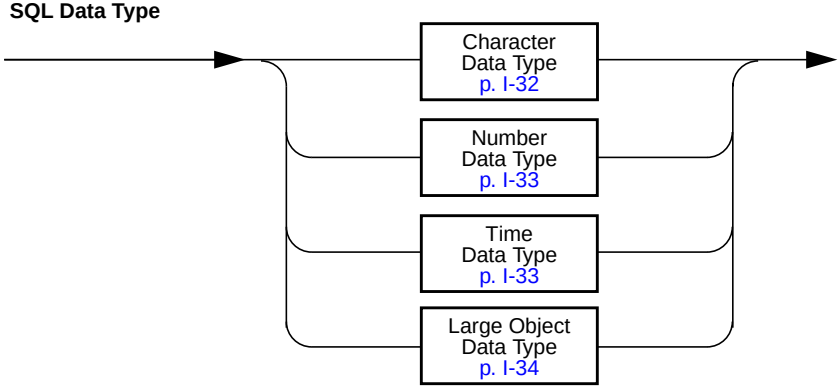
Database Name



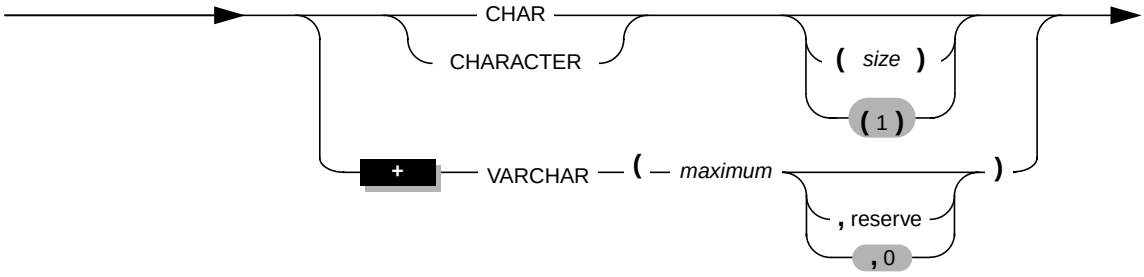
For SE engines, database identifiers can have up to ten characters in UNIX.

When the identifier for a database is also the name of a 4GL variable, the compiler uses the variable. To override this compiler action, quote the database identifier.

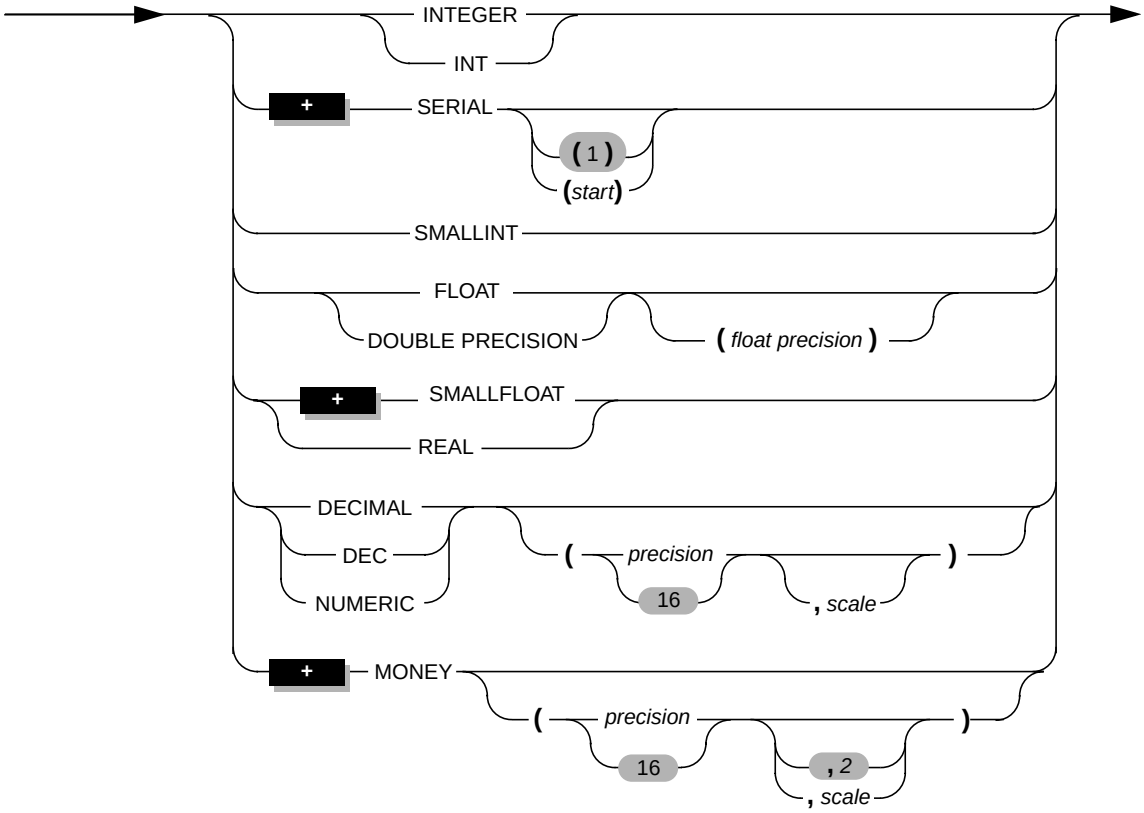
SQL Data Type



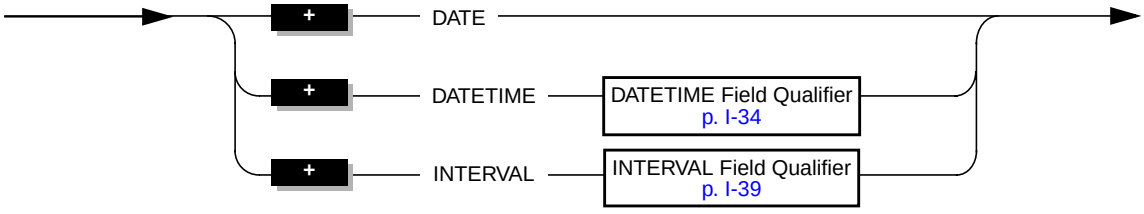
Character Data Type



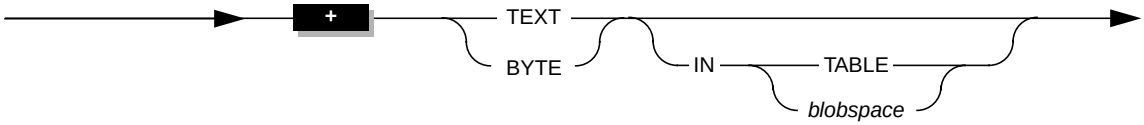
Number Data Type



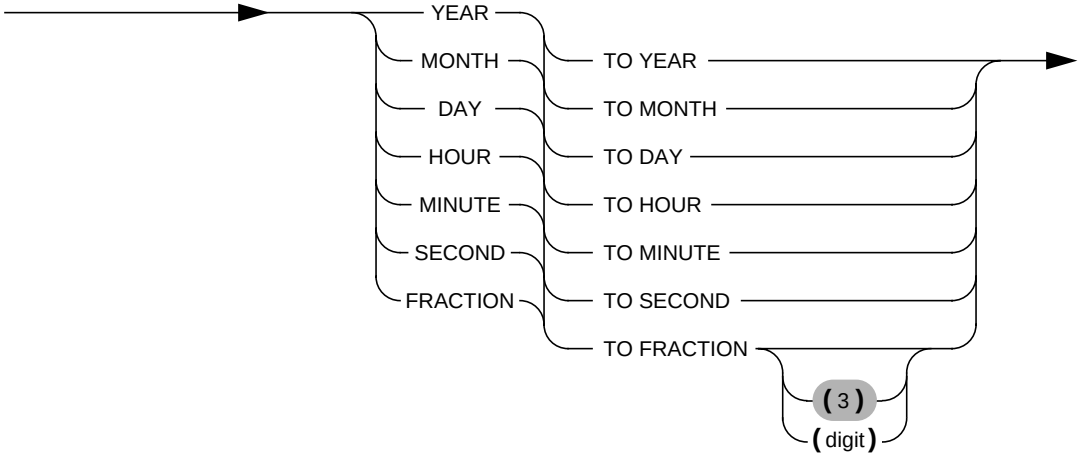
Time Data Type

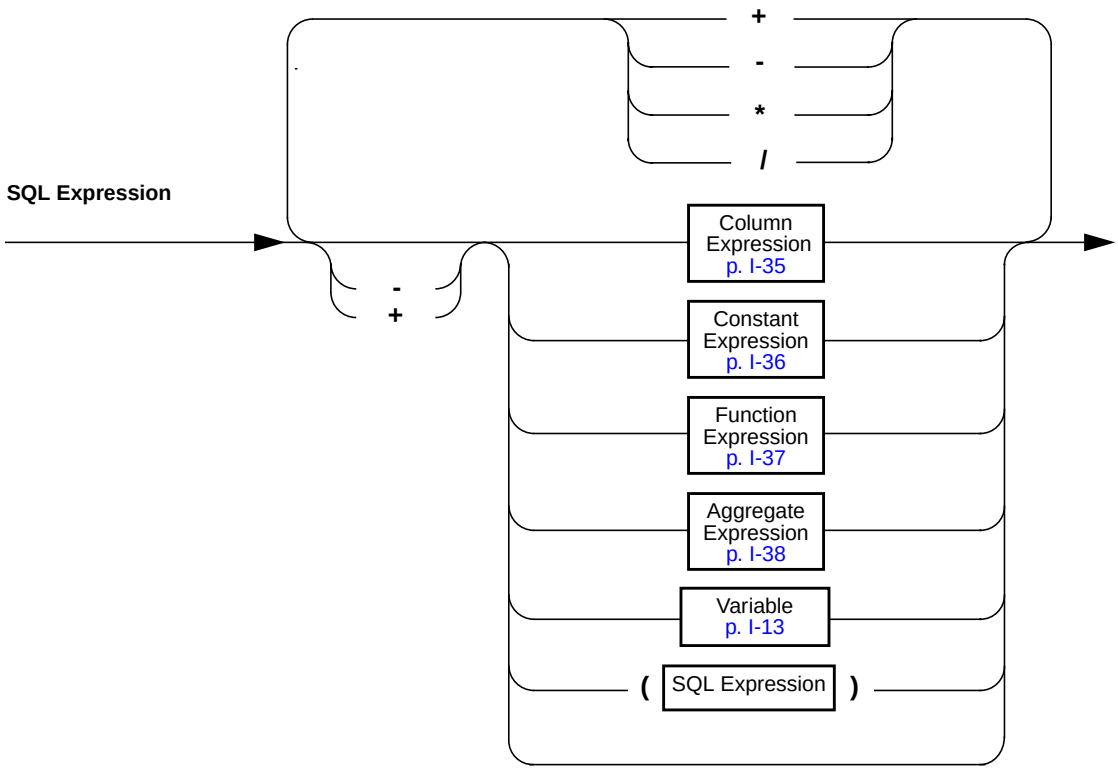


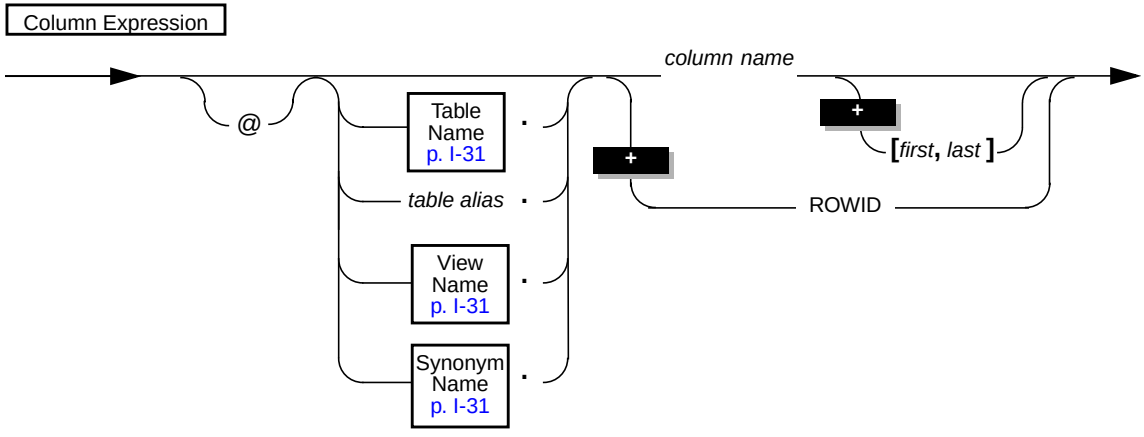
Large Object Data Type



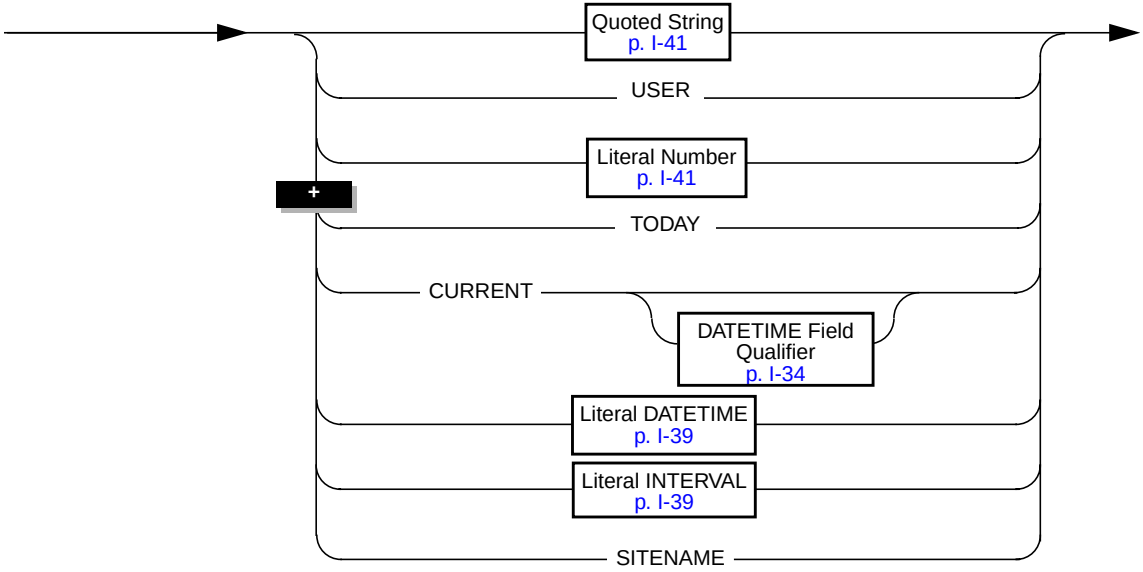
DATETIME Field Qualifier



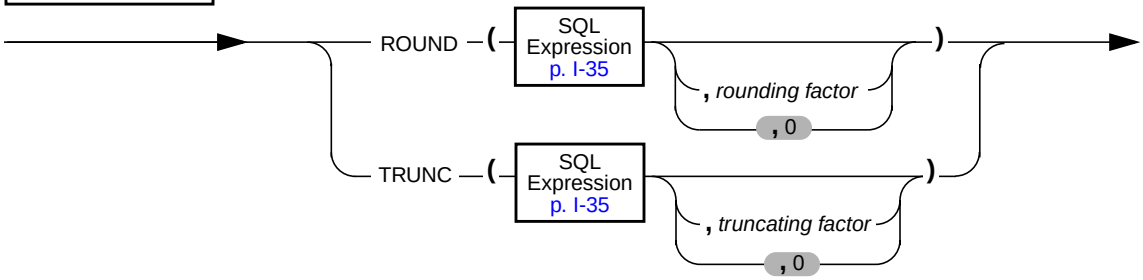

SQL Expression
+
-
*
/
-
+
Column Expression
p. I-35
Constant Expression
p. I-36
Function Expression
p. I-37
Aggregate Expression
p. I-38
Variable
p. I-13
(SQL Expression)


Column Expression
@
Table Name
p. I-31
table alias
View Name
p. I-31
Synonym Name
p. I-31
.
column name
+
ROWID
+
[first, last]

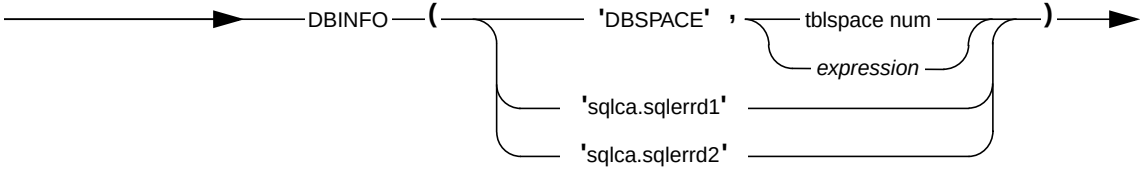
Constant Expression



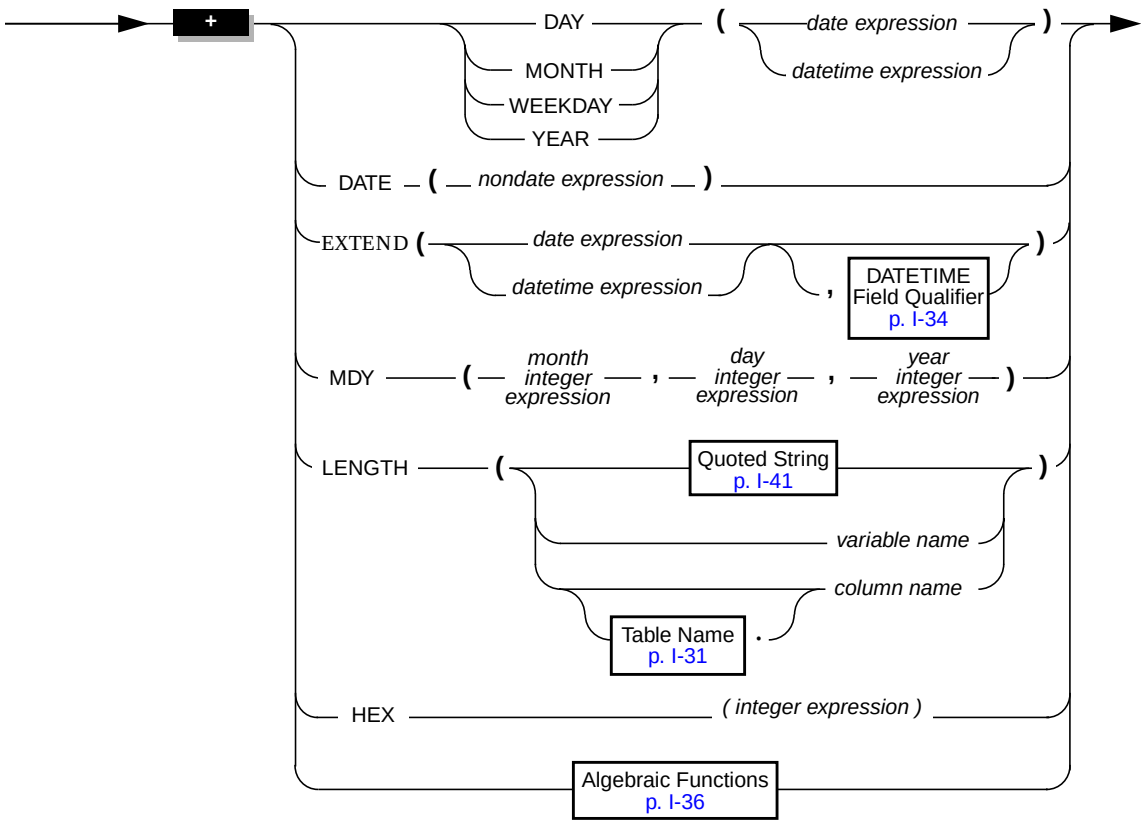
Algebraic Functions



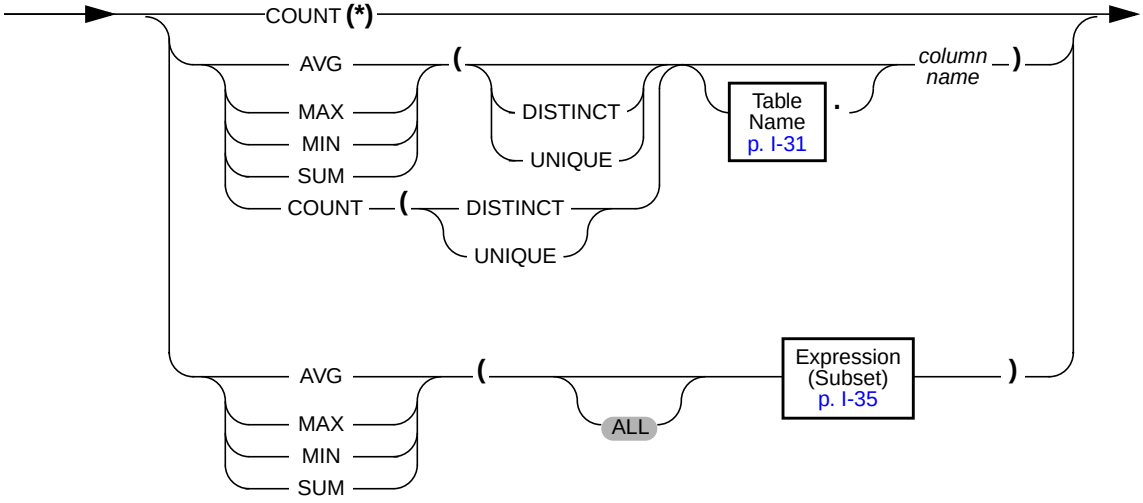
DBINFO Function



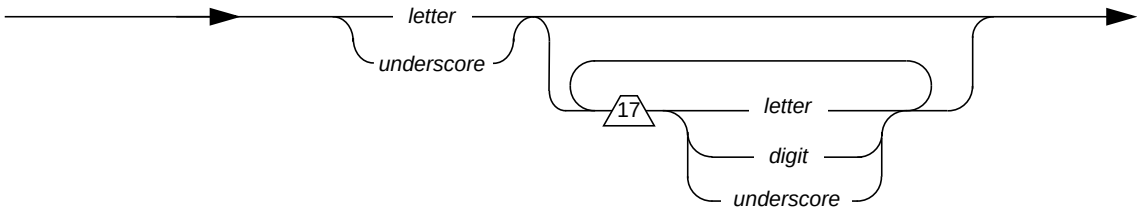
Function Expression



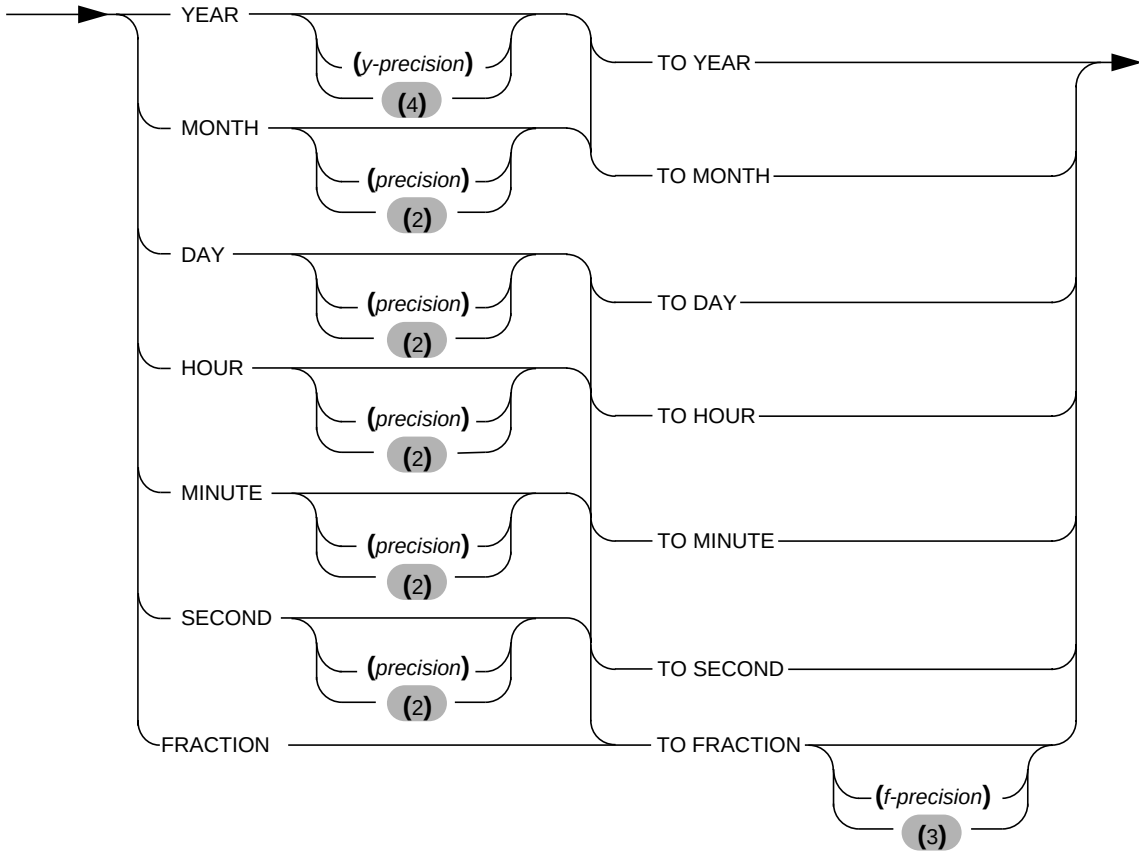
Aggregate Expression



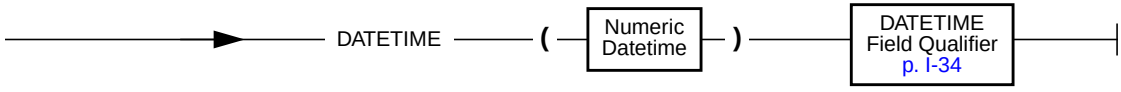
Identifier



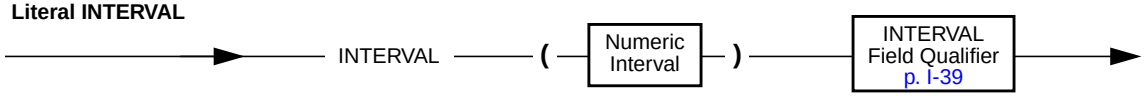
INTERVAL Field Qualifier



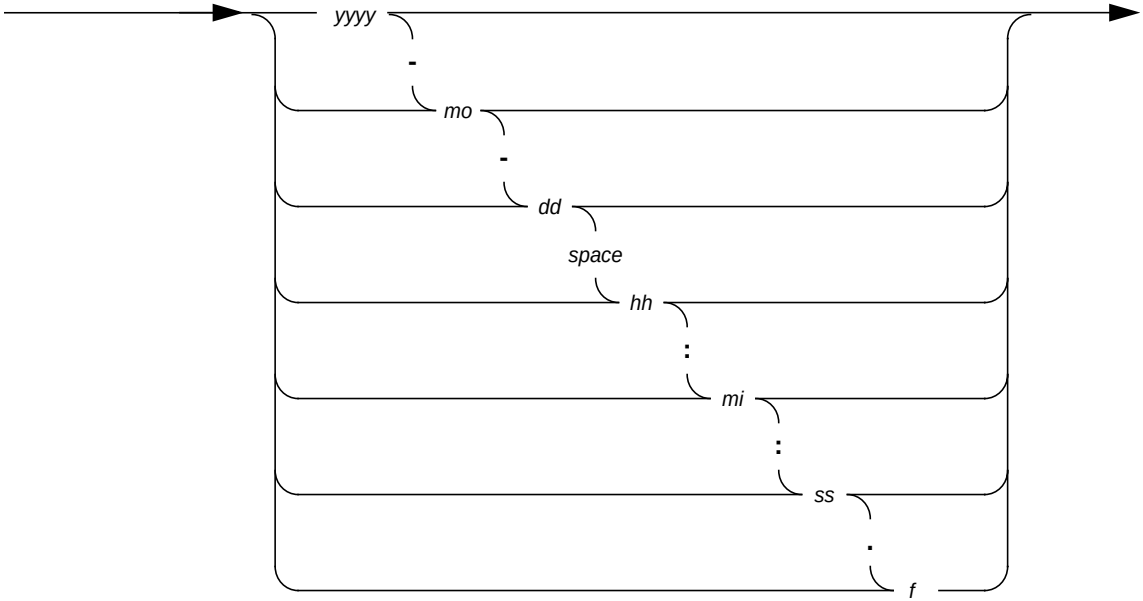
Literal DATETIME



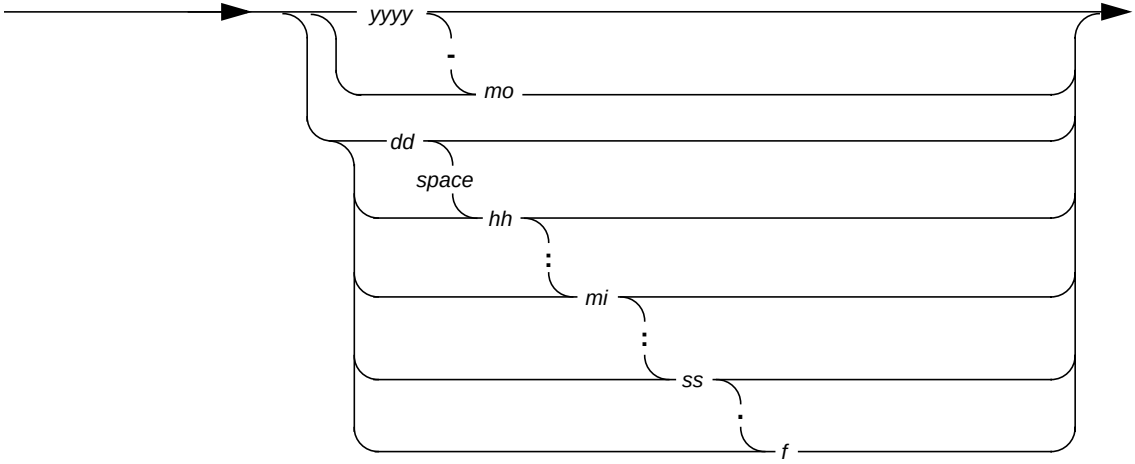
Literal INTERVAL



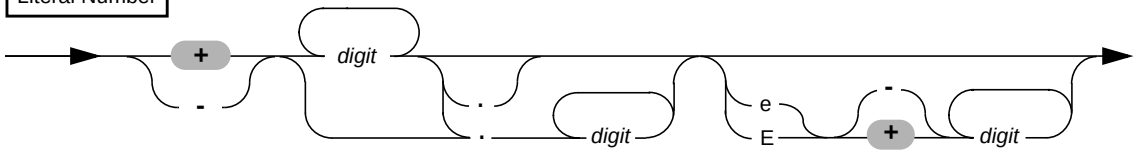
Numeric Datetime



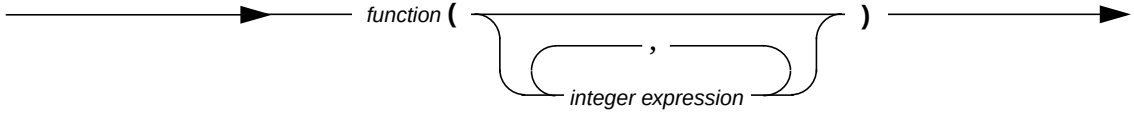
Numeric Interval



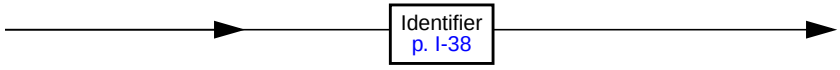
Literal Number



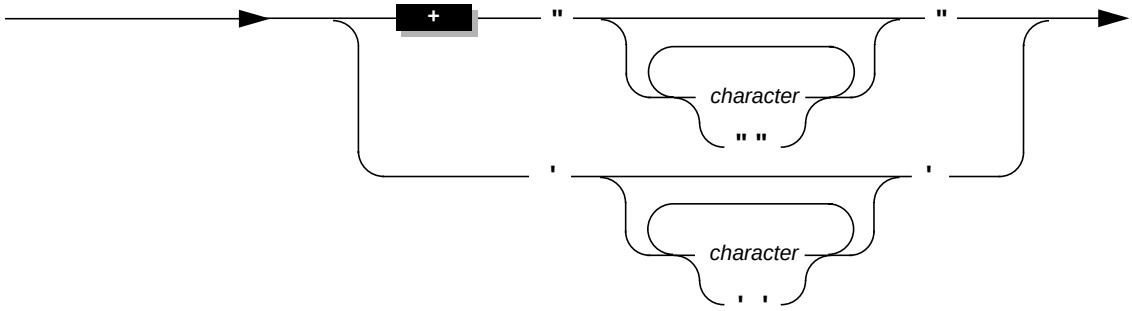
Function Call



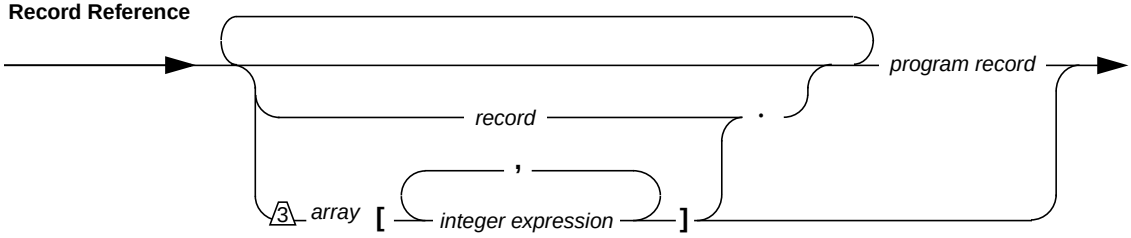
Prepared Statement Name



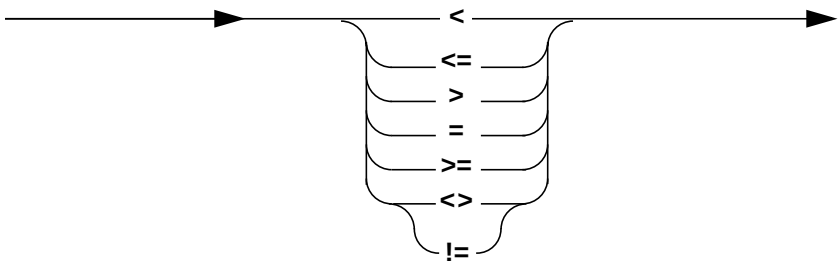
Quoted String



Record Reference



Relational Operator



Appendix

J

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and / or changes in the product(s) and / or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J74 / G4
555 Bailey Ave
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Glossary

4GL function

A 4GL program block defined with the FUNCTION statement. The function header follows the FUNCTION keyword and defines the name and formal argument list for the function. The function body (all statements between the function header and the END FUNCTION keywords) defines the actions of the function. The function header and the function body together are often called the function definition. To return values, use the RETURN statement within the function body. Frequently, a 4GL function is simply referred to as a function. See also *argument, function, programmer-defined function, program block, return value.*

4GL screen

1) In the INFORMIX-4GL Interactive Debugger, the 4GL screen is where the Debugger displays the 4GL application.

2) When running a 4GL application, the 4GL screen is the display area of the screen. This area displays the application forms, 4GL windows, and text. See also *4GL window, screen*.

4GL window

A rectangular region in the 4GL screen, possibly one of many, managed by a 4GL application. The default 4GL window is the 4GL screen. The OPEN WINDOW statement creates a new 4GL window. 4GL manages its windows with a stack. Each window is pushed onto this stack when it is opened. A 4GL program performs its input and output in the current window. See also *4GL screen, current, popup window, reserved lines, screen, stack.*

abnormal termination

The termination of the 4GL application through any mechanism other than terminating the MAIN program block at the END MAIN keywords or with a RETURN statement. Whenever a runtime error occurs, or the user presses an Interrupt or Quit key, or the EXIT PROGRAM statement is executed, an abnormal termination occurs. In the INFORMIX-4GL Interactive Debugger, you can inspect the application state after an abnormal termination. You cannot, however, resume execution. See also *debug*, *exception handling*, *MAIN program block*, *normal termination*, *program execution*.

Accept key

The logical key that the user can press within a 4GL application to indicate acceptance of the entered data or query criteria. Pressing it requests normal completion of a user interaction statement. By default, the physical key for Accept is ESCAPE. See also *data entry*, *Interrupt key*, *logical key*, *query criteria*, *Quit key*, *user interaction statement*.

access mode

The status of an open file that determines read and write access to that file.

activation key

A logical key that the developer defines to provide the user with some programmer-defined feature. The developer can define an activation key in the ON KEY clause of the CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or PROMPT statement, or in the KEY clause of the MENU statement. When the user presses the activation key, 4GL executes the control block associated with that key. See also *control block*, *key*, *logical key*, *menu option*.

active form

A screen form for which the current user interaction statement is executing. An application can have several active forms if it has several 4GL windows, each continuing a screen form with a user interaction statement executing. With multiple active forms, only one form is current. See also *4GL window*, *current*, *screen form*, *user interaction statement*.

active function

A 4GL program block (MAIN, FUNCTION, or REPORT) that has started execution but not completed execution. The active functions consist of all functions on the call stack, which are the current function and all functions that are waiting for a function call to return. The MAIN block is always active for the 4GL session. The INFORMIX-4GL Interactive Debugger can inspect the active functions that are not reports. See also *active variable*, *abnormal termination*, *call stack*, *debug*, *normal termination*, *program block*, *program execution*.

active set

The collection of database rows that satisfies a query associated with a database cursor. An active set is stored in memory at runtime. It contains only a copy of the rows that match the query criteria. See also *cursor*, *query*, *row*.

active variable A 4GL variable for which storage exists. The active variables consist of the local variables of all active functions, the module variables of all modules that contain the active functions, and all global variables. You can evaluate or assign values to active variables within the INFORMIX-4GL Interactive Debugger. See also *active function*, *abnormal termination*, *global variable*, *local variable*, *module variable*, *normal termination*, *program execution*, *scope*.

active window The window that contains the current keyboard focus. See also *keyboard focus*.

actual argument In a function call, the value that is passed by the calling routine as an argument to the programmer-defined function. This value must be of a compatible data type with the corresponding formal argument in the function definition. There are two methods for an actual argument to be passed to a function: pass-by-reference and pass-by-value. See also *argument*, *calling routine*, *data type conversion*, *formal argument*, *function call*, *pass-by-reference*, *pass-by-value*, *programmer-defined function*.

aggregate function 1) A function built into the SQL language that returns a single value based on the values of a column in several rows of a table. These functions are called SQL aggregate functions. Examples of SQL aggregate functions are SUM( ), COUNT( ), MIN( ), and MAX( ). These functions are valid only within SQL statements. See also *column*, *row*, *SQL*, *table*.

2) A function built into the 4GL language that returns a single value based on the values of input records. These functions are called report aggregate functions. Examples of report aggregate functions are SUM( ), GROUP SUM( ), PERCENT(*), MIN( ), and MAX( ). Report aggregate functions are valid only within a REPORT program block. See also *built-in function*, *input record*, *program block*, *report*.

alias In the database, an alias is a synonym for a table name. It immediately follows the name of the table in an SQL statement. It can be used wherever the name of a table can be used. Aliases are often used to create short, readable names for long or external table names. See also *table*.

ANSI compliant A database that conforms to certain ANSI (American National Standards Institute) performance standards. Informix databases can be created either as ANS compliant or as not ANSI compliant. An ANSI-compliant database enforces ANSI requirements such as implicit transactions, required owner-naming, and no buffered logging. The term MODE ANSI is sometimes used to refer to an ANSI-compliant database. See also *database*, *implicit transaction*.

application development tool

Software, such as INFORMIX-SQL, INFORMIX-4GL, and INFORMIX-ESQL/C, which a developer can use to create and maintain a database. Such software allows a user to send instructions and data to and receive information from a database server. See also *database server*.

application program

1) A computer program developed and implemented for dealing with some business activity. (A computer program is a group of instructions that cause a computer to perform a sequence of operations.) An application program is synthesized from some combination of application development tools. See also *application development tool*, *database*, *developer*, *user*.

2) In 4GL, an application program is the 4GL program, with one MAIN program block, its supporting 4GL source modules, its form specification files, and its help message file. See also *form specification file*, *help file*, *MAIN program block*, *source module*.

application program interface (API)

A rigorous definition of the method by which a program can access the services provided by another program. Developers of an API often provide libraries of callable functions that implement the API. For example, the 4GL API enables a C program to call a 4GL routine; Motif enables a C program to call X Windows; INFORMIX-ESQL/C, which is an SQL API, enables a C program to access a database. There can be many different APIs that provide access to the same set of services, though possibly at different points of entry. In some cases, the same API can be used to access different services. (For example, NetBIOS is a network API that is protocol independent and is often used to access a variety of different protocols such as OSI, AND TCP/.) See also *application development tool*.

argument

A value passed from a calling routine to a function. In the calling routine, the value passed is called an actual argument. Within the function definition, the name of the argument is called a formal argument. When the function is called, the value of the actual argument is assigned to the corresponding formal argument variable. See also *actual argument*, *calling routine*, *formal argument*, *pass-by-reference*, *pass-by-value*, *programmer-defined function*, *report*.

arithmetic operators

Operators that perform arithmetic operations on operands of number (and some time) data types. The following are 4GL binary arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**), and modulus (MOD). 4GL unary arithmetic operators are unary minus (-) and unary plus (+). Arithmetic operators return a numeric value (or else NULL) . See also *associativity*, *binary operator*, *operand*, *operator*, *precedence*, *unary operator*.

array

1) A data structure that has a fixed number of components. Each component is called an element. All elements in an array have the same data type. See also *array element*, *screen array.*

2) In uppercase, ARRAY is the keyword for defining a program array in 4GL. The ARRAY data type is a structured data type of up to three dimensions. It cannot have another array as an element. You can reference an array element by listing the array name followed by one or more comma-separated subscripts within brackets. See also *program array, structured data type, subscript*.

array element

A component of a program array, of any 4GL data type except ARRAY. To reference any element within an array, use one or more integer subscripts (sometimes called an array index). See also *array, program array, subscript*.

ASCII

1) Acronym for American Standards Code for Information Interchange. In the default locale, the ordered set of internal codes that a computer uses to represent characters. This set includes both printable and non-printable characters. Appendix A, "The ASCII Character Set," lists the ASCII characters and their codes. See also *control character, escape character, printable character.*

2) As opposed to binary information, an ASCII file is readable in a text editor. 4GL source modules and form specification files are examples of ASCII files. This manual sometimes uses the term *ASCII file* in reference to files that contain only printable characters (which might include non-ASCII characters) of the client locale. See also *form specification file, source module, text editor.*

assign

To store a value in a variable. In 4GL, the LET statement performs assignment. 4GL evaluates the expression on the right-hand side of the equal sign (=) and assigns it to the variable on the left-hand side. You can also assign values to a variable with the CALL...RETURNING, CONSTRUCT, FOREACH...INTO, INITIALIZE, INPUT, INPUT ARRAY, and PROMPT statements, and with the INTO clause of SELECT and EXECUTE statements of SQL. See also *expression, operator, precedence, variable*.

associativity

The principle that determines the order in which operands at the same level of precedence in an expression are evaluated. For example, to evaluate the expression `a - b + c`, 4GL first evaluates `a - b` and then adds `c` to the result because binary arithmetic operators associate to the left. You can use parentheses (( )) symbols, as in algebra, to override the default associativity of 4GL operators. See also *binary operator, expression, precedence*.

asterisk notation The symbols **.*** appended to the name of a database table (*table*.*****) or of a 4GL record variable (*record*.*****), which expands to the names of all the columns in the *table* or of all members in the *record*. In other contexts, asterisk can be a wildcard for matching strings, or can indicate overflow in a display or other features of 4GL. See also *program record*, *table*.

attribute 1) A characteristic or aspect of some entity that the developer can set. 4GL provides attributes for form fields (field attributes), screen forms (form attributes), database columns (column attributes), and for output text (display attributes). Set field attributes on a field-by-field basis in the form specification file. Set form attributes with the ATTRIBUTE clause of the DISPLAY FORM statement and of the 4GL user interaction statements. You can also set display attributes with the ATTRIBUTE clause. You can set default column attributes on a column-by-column basis with the **upscol** utility. See also *column*, *field*, *form*, *form specification file*, *user interaction statement*.

2) In some database terminologies, attribute is a term used for a column. See also *column*.

audit trail A history of all changes to a table in an INFORMIX-SE database.

B+ tree A method of organizing an index for efficient retrieval of records. All Informix database servers use this access method.

background process In a multiprocessing environment, a process that is not performing input or output. It can continue to run without needing access to a window or the screen. See also *foreground process*, *process*, *screen*.

batch A mode of execution in which a program runs without input from a user. 4GL programs that do not use user interactive statements are batch programs. If a batch program produces output, it should direct the output to a file or the printer, not the screen. Often reports are run in a batch mode. See also *interactive*, *program execution*, *report*, *user interaction statement*.

binary operator An operator that requires two operands. The binary operator appears between the two operands. In 4GL, examples of binary operators include addition (+), multiplication (*), and logical AND. 4GL associates most binary operators from left-to-right. See also *arithmetic operators*, *associativity*, *Boolean operators*, *operand*, *operator*, *precedence*, *relational operators*, *unary operator*.

binding A one-to-one correspondence between entities in two domains. The association between an identifier and its resource (a location in memory) is called a binding. In 4GL, the correspondence between form fields and program variables during data entry is also called a binding. Several 4GL screen interaction statements include a binding clause that lists the program variables and their corresponding form fields (or database columns). These statements include CONSTRUCT, INPUT, INPUT ARRAY, and PROMPT. See also *data entry*, *identifier*, *program array*, *program record*, *screen array*, *screen record*, *user interaction statement*.

blank space The character with the value of ASCII 32. A string of blank spaces is not the same as a null string (which has nothing in it). 4GL pads string values with blank spaces up to the declared size of the CHAR variable (but not VARCHAR variables). Blank spaces are also used to separate elements of a menu or form. More generally, characters in any locale that can separate terms within 4GL statements, but that display as blank spaces, are often referred to as *white space* characters. See also *ASCII*, *clipped*, *null value*, *printable character*, *string*.

blob A legacy acronym for binary large object that is now known as and includes BYTE and TEXT data types. Blobs are data objects that effectively have no maximum size (theoretically as large as 2^{31} bytes). See also *byte*, *data type*, *text*.

blobpage The unit of disk allocation within a blobspace in the database. The System Administrator determines the size of a blobpage; the size can vary.

blobspace A logical collection of blobpages to store TEXT and BYTE data in the database.

Boolean 4GL includes two Boolean constants: FALSE (= 0) and TRUE (=1). If an operand evaluates to null, Boolean operators can return an unknown result that 4GL treats as FALSE in some contexts. Because 4GL does not have a Boolean data type, Boolean values are typically stored as integer data types. See also *Boolean operators*, *constant*, *integer*, *relational operators*.

Boolean operator An operator (for example, AND, OR, NOT, or any of the relational operators) that returns a Boolean value. In some contexts, 4GL interprets null values as FALSE and any nonzero value as TRUE. Boolean expressions can also include relational operators. See also *associativity*, *binary operator*, *Boolean*, *operand*, *operator*, *precedence*, *relational operators*, *unary operator*.

built-in function A function that is part of the 4GL language and can therefore be called without needing to be defined by the developer. Calls to built-in functions have the same syntax as those for programmer-defined functions. Examples of built-in functions are ARG_VAL( ), ARR_CURR( ), FGL_KEYVAL( ), and SCR_LINE( ). See also *4GL function*, *aggregate function*, *built-in operator*, *calling routine*, *function call*, *programmer-defined function*.

built-in operator An operator that is part of the 4GL language. Built-in operators are keywords or symbols that perform special tasks. They differ from built-in functions in that they cannot be invoked with the CALL statement and they cannot be called from a C function. Examples of built-in operators are ASCII, CURRENT, DATE, and TODAY. (Here built-in is redundant; the programmer cannot define new 4GL operators.) See also *built-in function*, *keyword*, *operator*.

byte 1) A unit of storage that corresponds to 8 binary bits. A kilobyte is 1,024 bytes. A megabyte is 1,048,576 bytes. See also *character*.

2) In uppercase letters, BYTE is the 4GL and SQL data type that can store up to 2^{31} bytes of binary data. See also *blob*.

C Compiler The version of 4GL that precompiles 4GL code into Informix ESQL/C code and then translates the ESQL/C into object code that is executable directly from the command line. See also *compile*, *Rapid Development System*.

call stack A stack (also called the *function stack*) used by 4GL at runtime to keep track of active functions. An *active functio*n is one that has been called but that has not yet returned. Each time the program calls a function, the state of the function is pushed onto the call stack. When a function terminates, the state is popped off the call stack. The MAIN program block is always at the bottom of this stack. You can examine the call stack with the INFORMIX-4GL Interactive Debugger. See also *active function*, *MAIN program block*, *stack*.

call-by-reference See *pass-by-reference*.

call-by-value See *pass-by-value*.

calling routine The program block that invokes a function (or report). In 4GL, the calling routine can be either a MAIN, FUNCTION, or REPORT program block. A call can be explicit, by using the CALL statement, or implicit, by embedding the function name (and any arguments) within an expression. The calling routine can pass values to the function and can receive returned values. See also *argument*, *expression*, *function call*, *program block*, *return value*.

case sensitivity The ability to distinguish between uppercase and lowercase letters. 4GL is not case sensitive, except within quoted strings; thus, variables **a** and **A** refer to the same address in memory. Certain command-line syntax elements (such as command names and options) are case sensitive. Unless the locale files define the correspondence between uppercase and lowercase characters, case sensitivity is not supported. See also *identifier*, *keyword*, *naming conventions*.

character 1) Any letter, digit, symbol, or control sequence that can be represented by the character set of the client locale. In some East Asian locales, one logical character can require more than one byte of storage. See also *ASCII*, *blank space*, *control character*, *escape character*, *printable character*.

2) The character data types are CHAR and VARCHAR (and in some contexts, a TEXT variable). See also *blob*, *data type*, *string*, *string operators*, *subscript*.

3) What a single keystroke, control character, or escape sequence produces that the program, operating system, or output device treats as a single unit. See also *activation key*, *logical key*, *operating system*.

character set In the default locale, the ASCII characters, each corresponding to an integer value from 0 to 255 (8-bit) or 0 to 127 (7-bit). In other locales, the character set is specified by a code set in the locale files. In some East Asian locales, a single logical character can require more than one byte of storage, and the number of logical characters in the code set can be a 4-digit number. Every locale must support the ASCII characters. See also *ASCII, Global Language Support*.

clipped The CLIPPED operator of 4GL removes trailing white space from string values. It is often used in DISPLAY and PRINT statements. (SQL statements use TRIM.) See also *blank space*, *built-in operator*, *string operators*.

close To cease to use an open entity. In programming, closing something releases control of it and deallocates any resources that it used. For example, when you close a database cursor, you release any memory or disk space that was used to hold the active set for that cursor. When you close a file, you tell the system that you no longer require the file and others can use it. When you close a form, you release any memory or disk used to store it. When you close a 4GL window, you deallocate memory for the image of the window and pop it from the window stack. Typically things cannot be closed until they have been opened. See also *4GL window*, *close*, *cursor*, *file*, *open*, *screen form*.

close a cursor To drop the association between the active set of rows that result from a query and a cursor.

column

1) In a database, a column is a data category that contains a specific type of information common to every row of the table. In other database terminologies, a column is sometimes called a field or an attribute. See also *attribute, database, row, table*.

2) In a screen or in page of output from a report, a *column* is the x-coordinate of a given position (on the horizontal axis). The y-coordinate (on the vertical axis) is called a *row* (or a *line*). Several 4GL statements use *columns* (and *rows* or *lines*) in this sense to identify locations in displays. See also *row, screen*.

command line

A line of text typed by the user at the operating system prompt to run a program. In a character-based environment, all programs are invoked by a command line with optional command-line arguments and options. See also *operating system*.

comment

Text in a source file that the compiler ignores. These comments can explain the contents of the file or disable a statement. In 4GL source modules, comments can be introduced by a left brace ({) in the first position of the line and terminated by a right brace (}) anywhere on a line to close the comment. Comments in the same line as code can begin with a double-hyphen (- -) or pound (#) sign. In 4GL form specifications, left braces or the double-hyphen can begin comments; the # symbol is not valid. *Conditional comments* have comment indicators that Dynamic 4GL (for - -#) or INFORMIX-4GL (for - -@) interpret as white space. See also *form specification file, header, source module*.

commit

To successfully end a transaction by accepting all changes to the database since the beginning of the transaction. When the transaction is committed, all open database cursors (except hold cursors) are closed and all locks are released. The COMMIT WORK statement commits the current transaction. See also *cursor, log, roll back, transaction*.

compile

1) To translate a program from source code written by the developer to a form executable by the computer (machine code). This translation is done by a system program called a compiler. Results of the translation are called object code, or a compilation. See also *debug, execute, interpret, link, source module*.

2) In 4GL, you can compile 4GL source code into either p-code or C language code. For the p-code, the compiler translates the 4GL code into an intermediate form (p-code) that must be executed by the P-Code Runner. For C code, the compiler first calls a preprocessor to translate 4GL code into INFORMIX-ESQL/C code. It then translates the ESQL/C into object code, executable directly from a command line. See also *C Compiler, command line, compiler directive, p-code, preprocessor, Rapid Development System*.

compiler directive

An instruction within a source module to a compiler, as opposed to an executable statement. In the C language, directives can address the preprocessor portion of the compilation, requesting, for example, conditional compilation or inclusion of a named file. In 4GL, the WHENEVER and DEFER statements are compiler directive. The effect of WHENEVER lasts until the end of the source file (or until overridden by another WHENEVER). See also *compile*, *exception handling*, *preprocessor*, *source module*, *statement*.

composite index

An index constructed on two or more columns of a table. The ordering imposed by the composite index varies least frequently on the first named column and most frequently on the last named column.

composite join

A join between two tables based on the relationship among two or more columns in each table.

concatenate

To form a character string by appending a second character string to the end of the first character string. In 4GL and SQL, the concatenation operator is a double pipe (| |) symbol. See also *character*, *string*, *string operators*.

consistency checking

The process of verifying that the character set, collation order, and other client settings for a user session match the settings in the database locale.

constant

A named value that, unlike a variable, cannot change during execution of a program. Constants of 4GL include NOTFOUND, FALSE, and TRUE. These values cannot be redefined by the developer. The documentation of Informix database servers sometimes applies this term to *literals*. See also *Boolean*, *literal*, *variable*.

constraint

See *UNIQUE CONSTRAINT*.

control block

A statement block that executes when a certain condition (the activation clause) becomes true. In 4GL, control blocks like BEFORE FIELD, AFTER INPUT, AFTER CONSTRUCT, and ON KEY occur in the user interaction statements; in this context, they are often called form management blocks. Control blocks also occur in the FORMAT section of a report, including PAGE HEADER, AFTER GROUP OF, and ON EVERY ROW. See also *activation key*, *report*, *statement block*, *user interaction statements*.

control character A character whose occurrence in a specific context initiates, modifies, or stops a control function (an operation to control a device, for example, in moving a cursor or in reading data). Control characters have values below ASCII 32 in the ASCII character set. In a 4GL program, some control characters have predefined functions (pressing CONTROL-W obtains on-line help). The developer can also define actions that use CONTROL keys with another key to execute some programming action. See also *activation key, ASCII, character, logical key, modifier key.*

current The one item, among many similar items, that is about to be or was most recently used. The current directory is the host system directory that was selected most recently (it is where files are first looked for). The current row is the row that was last fetched through a database cursor (it can be deleted or updated using the cursor). The current window is the window most recently activated (it receives the user's keystrokes). The current statement is the program statement being executed. (This statement is displayed in an error message if the program terminates.) See also *cursor, directory, row, statement*.

cursor 1) A focal point at which action can be applied. A text cursor is a mark showing the focal point for keyboard input. See also *keyboard focus, text cursor.*

2) A database cursor is an identifier associated with an active set. It points to the current row in the active set. This row can be fetched, deleted, or updated. 4GL supports the following types of database cursors: sequential, scroll, hold, update, and insert. See also *active set, close, current, identifier, open, prepared statement, query, row, scrolling*.

data conversion See *data type conversion*.

data entry 1) The action of providing, usually at the keyboard, data values to a computer program. Data entry is performed by the user of an application at runtime. In a database application, these values are usually stored in a database. See also *application program, key, user interaction statements.*

2) A set of data values to be stored in program variables and, often, in a database table. The INPUT, INPUT ARRAY, and PROMPT statements accept data entry. See also *query criteria, table, user interaction statement*.

data file A file that contains the input used by a program. Data files are not executable; they contain data that is to be read or acted on by other programs. See also *file*.

data type

An interpretation to use on a stored value. In 4GL, database columns, program variables, form fields, and formal arguments of a function (or report) all have data types associated with them. The 4GL data types include: integer (SMALLINT, INTEGER); fixed-point (DECIMAL(*p,s*), MONEY); floating-point (DECIMAL(*p*), FLOAT, SMALLFLOAT); character (CHAR, VARCHAR); large (BYTE, TEXT); time (DATE, DATETIME, INTERVAL); and structured (ARRAY, RECORD). Informix databases can support additional data types. See also *blob, character, data type conversion, declare, fixed-point number, floating-point number, integer, interval, operator, simple data type, structured data type, variable.*

data type conversion

The process of interpreting and storing a value of one data type as some other data type; sometimes called simply data conversion. The pairs of data types for which data conversion is possible without error are called compatible data types. This process can be automatic or explicit. 4GL performs some automatic data conversion in expressions and assignment. It also provides facilities to perform some explicit data conversion (for example the EXTEND( ) function). In addition, the LIKE keyword, when used in a variable definition, provides indirect data typing. See also *column, data type, define, indirect typing.*

database

A collection of related data organized in a way to optimize access to this data. A relational database organizes data into tables, rows, and columns. Informix databases are relational databases. At runtime, a separate database server process is the portion of the database management system that actually manipulates the data in the database files. To access a database, a 4GL application must specify it with the DATABASE statement and must use SQL statements. See also *column, database server, process, row, SQL, system catalog, table.*

database cursor

See *cursor.*

database locale

Locale of the user at the time of database creation, permanently saved in database system tables and consulted when the database is accessed. See also *Global Language Support, user locale.*

database server

The part of a database management system that manipulates data files. This process receives SQL statements from the database application, parses them, optimizes the approach to the data, retrieves the data from the database, and returns the data to the application. The database server is also called the back end. See also *application development tool, database, process.*

debug

1) To find and remove runtime errors in a computer program. This analysis is often done by special software products called debuggers. You can analyze a program to detect and locate errors in its logic, change the source code appropriately, and then compile and run the program again. See also *compile*, *execute*, *link*.

2) If you are using the Rapid Development System, you can use the INFORMIX-4GL Interactive Debugger to debug 4GL programs. The Debugger helps you to control and monitor program execution and inspect the application state. The Debugger is purchased separately from 4GL. See also *program execution*, *Rapid Development System*.

debugger

A software product to analyze programs and to detect and locate errors in program logic. The INFORMIX-4GL Interactive Debugger is a 4GL source language debugger that supports a wide variety of programming tools, such as tracing program logic and stopping execution at preset points. See *breakpoint* and *tracepoint*.

declare

To make the name and data type of a variable known to a compiler. In 4GL, the DEFINE statement declares variables so the 4GL compiler can verify references to the variables in the succeeding code. The GLOBALS statement declares global variables. See also *compile*, *data type*, *define*, *global variable*, *variable*.

default

The value that will be used, or the action that will be taken, unless you specify otherwise. In many SQL and 4GL statements, there is a default action that will be used if you do not specify another; for example, the FETCH statement retrieves the NEXT row unless you specify a different keyword such as PRIOR. Screen forms can specify a default value for input from each field, in case the user fails to enter one. See also *screen form*, *variable*.

define

To allocate memory for storage of a variable. At runtime, the DEFINE statement indicates how much storage should be allocated for a variable. The GLOBALS statement defines global variables. To define a function (or report) is to specify the actions performed by the function. See also *data type*, *declare*, *execute*, *function*, *global variable*, *report*, *variable*.

Delete key

The logical key that the user can press within a 4GL application to delete the current line of a screen array (the current screen record) during the INPUT ARRAY statement. 4GL automatically deletes the associated line of the program array. By default, the physical key for Delete is F2. See also *Insert key*, *logical key*, *program array*, *screen array*, *screen record*.

delimiter A character that separates one unit of text from another. The eye can easily see boundaries based on context, but programs need unambiguous marker characters to detect the end of one item and the start of the next. In data produced by the UNLOAD statement, the data from each column ends with a delimiter (| by default) so that the LOAD command can recognize the end. In the form specification file, brackets ([ ]) mark or delimit the fields of the form. See also *form specification file*.

developer An individual who develops computer programs, taking them from design, coding, and debugging to general release. 4GL provides the developer with a means of developing database applications. Also referred to as the programmer. See also *application program*, *user*.

development environment The special set of tools that a developer can use to develop computer programs. This environment might include text editors, language compilers, function libraries, program linkers, program debuggers, and other program utilities. Some of these tools might be accessed by wrapper programs, which decide which tools need to be run. The Programmer's Environment is an integrated development environment that combines many of these tools into a single, cooperative environment. See also *compile*, *debug*, *execute*, *link*, *Programmer's Environment*, *text editor*.

directory A directory is a file folder; it contains other files. Directories can also contain other directories; directory structures are used to organize related files into categories. The user can construct a hierarchy of subdirectories and files that resembles an inverted tree in structure. Each user has a home directory that represents the top level of the user's personal hierarchy of other directories and files. The current directory is a single directory (selectable by the user) that enables the user to be in one directory at a time and to refer to its files unambiguously. To refer to files in other directories, the user must supply a path, that is, a list of the subdirectories that describe the location of those files. See also *current*, *file*, *operating system*.

DIRTY READ A level of process isolation that does not account for locks and does allow viewing of any existing rows, even rows that might be currently altered from inside an uncommitted transaction. DIRTY READ is the lowest level of isolation (no isolation at all). It is the level at which INFORMIX-SE operates, and it is an optional level under Informix Dynamic Server. See *process isolation*.

display field Field used in a screen form to indicate where data is to be displayed on the screen. A display field is usually associated with a column in a table.

element See *array element*.

environment variable

A special variable with a value that is maintained by the operating system and made available to all programs. Environment variables are usually stored in a special system area and contain system specifics such as the path (the directories in which the operating system looks for user-invoked commands). See also *operating system*, *shell*.

error

An exception that indicates failure of a requested action or an illegal specification. Errors can occur during compilation—during preprocessing of program statements or during the linking stage—or during execution of the program. At runtime, some errors are fatal in that the program cannot continue execution (runtime errors); others are recoverable in that the program can take corrective action and continue execution. Rounding errors can occur during truncation in rounding; overflow errors can occur during arithmetic operations in which the size of the result is larger than the size of the space in which the result is to be stored. See also *compile*, *error handling*, *exception*, *execute*, *link*, *status variable*, *truncation*.

error handling

Program code that checks for a runtime error. By default, 4GL terminates a program when it encounters a runtime error. The 4GL WHENEVER ERROR statement can change this default error-handling behavior. With the WHENEVER ERROR CONTINUE statement, 4GL sets the built-in **status** variable to a negative value and continues execution when it encounters a runtime error. The program must explicitly check the value of **status** and determine appropriate action. See also *error*, *error log*, *error text*, *exception handling*, *program execution*, *status variable*.

error log

A file that receives error information for a program at runtime. 4GL contains some built-in functions that allow you to make entries in the error log: ERR_GET( ), ERR_PRINT( ), ERR_QUIT( ), ERRORLOG( ), and STARTLOG( ). See also *built-in function*, *error*, *error handling*, *status variable*.

error message

Text that describes a 4GL error. Each error is identified by an integer, usually negative, called an error code. Each code corresponds to a specific error message. Such messages can be retrieved by running the **finderr** utility or within a program by making a call to the ERR_GET( ), ERR_PRINT( ), or ERR_QUIT( ) built-in functions. By default, 4GL automatically displays some runtime error messages on the screen. See also *error handling*, *error log*, *status variable*.

error trapping

Code within a program that anticipates and reacts to runtime errors.

escape character A character that indicates that the following character, normally interpreted by a program, is to be printed as a literal character instead. Usually programs that handle user input (such as text editors and shells) have some characters that have special interpreted meanings. The escape character is used with the interpreted character to escape or ignore the interpreted meaning. The ASCII escape character is a nonprinting character with value of ASCII 27. In 4GL, the backlash (\ ) can be used as an escape character: the string "\\" would indicate that the backslash character is to be sent. See also *ASCII, character, printable character, shell, text editor.*

Escape key The physical key usually marked ESC on the keyboard. It sends the ASCII code for the escape character. This key is the default Accept key in user interaction statements like CONSTRUCT, DISPLAY ARRAY, INPUT, or INPUT ARRAY. See also *Accept key, escape character, key, user interaction statement*.

exception A runtime event for which the program might want to take some action. Exceptions in 4GL include: runtime errors (an error returned by a database server, a state initiated by a stored procedure statement, or an error detected by the application program), a database query that returns no rows (**status** variable is set to NOTFOUND), warnings (SQL conditions), and the pressing of the Interrupt or Quit key by the user. See also *error, exception handling, Interrupt key, Quit key, **status** variable, warning.*

exception handling Program code that checks for exceptions and performs recovery actions in the event they occur. By default, 4GL performs the following exception handling: runtime errors—terminate the program; SQL NOTFOUND—set **status** to NOTFOUND and continue execution; warnings—continue execution; and Interrupt (or Quit) key—terminate program. The developer can change the default exception handling for these first three types of exceptions with the WHENEVER statement: runtime errors (WHENEVER ERROR, WHENEVER ANY ERROR); NOTFOUND (WHENEVER NOT FOUND); and warnings (WHENEVER WARNING). The DEFER statement changes handling of the Interrupt and Quit keys. See also *error handling, Interrupt key, program execution, Quit key, status, warning.*

executable file

1) A file that contains machine code (in binary form) that has been linked and is ready to be run on a computer. Can also refer to a collection of instructions that can be executed by a command interpreter or processor, for example a batch file. See also *compile*, *execute*, *file*, *interpret*, *link*.

2) In 4GL, an executable file can be either interpretive p-code (with a **.4gi** or **.4go** extension) or compiled C code (with an **.exe** extension), depending on the version of 4GL that you are using. See also *command line*, *p-code*.

executable statement

A statement that requires processing action at runtime. Executable statements are distinguished from declarative statements (that provide information about the nature of the data without themselves causing any processing) and compiler directives that are instructions to the compiler. All 4GL statements are executable except MAIN, DEFINE, DEFER, FUNCTION, GLOBALS, REPORT, and WHENEVER. See also *compiler directive*, *declare*, *define*, *statement*.

execute

1) To run a compiled or interpreted program by carrying out the instructions in an executable file. To execute or run a program, the operating system must create a process for the program and then allocate the CPU (and any other resources needed by the program) to this process. The state of the program in execution is often called runtime. See also *compile*, *debug*, *executable file*, *interpret*, *link*, *operating system*, *process*, *resources*.

2) A 4GL executable file contains either interpreted p-code or compiled C code, depending on the version of 4GL that you are using. See also *C Compiler, command line*, *Rapid Development System*.

execution stack

See *call stack*.

expansion page

An additional page filled with data from a single row. Informix Dynamic Server uses expansion pages when the data for a row cannot fit in the initial page. Expansion pages are added and filled as needed. The original page entry contains pointers to the expansion pages.

explicit transaction

A transaction that the developer must explicitly begin and end. The BEGIN WORK statement indicates the beginning of the transaction. The developer must explicitly indicate the end of the transaction with the COMMIT WORK and ROLLBACK WORK statement. A database that is not ANSI compliant but that has a transaction log uses explicit transactions. See also *ANSI compliant*, *commit*, *roll back*, *transaction*.

exponent

1) In the representation of a FLOAT or SMALLFLOAT value, a signed integer that indicates the power to which the mantissa is to be raised. See also *floating-point number*, *mantissa*.

2) The right-hand unsigned integer operand of the exponentiation (**) operator in 4GL expressions. See also *arithmetic operators*.

expression

A sequence of operators, operands, and parentheses that can be evaluated to a single value, usually at runtime. In 4GL, an expression should evaluate to a simple 4GL data type: number (Boolean, integer, floating-point, and fixed-point), character, or time (DATE, DATETIME, and INTERVAL). See also *Boolean*, *character*, *data type*, *fixed-point*, *floating-point*, *integer*, *interval*, *operand*, *operator*, *precedence*, *regular expression*.

extent

A continuous segment of disk space allocated to a tblspace in the database. The programmer can specify both the initial extent size for a table and the size of all subsequent extents assigned to the table.

external signal

The notification of an operating system event that is delivered to a process. 4GL programs can handle two external signals: Interrupt and Quit. See also *Interrupt key*, *operating system*, *process*, *Quit key*.

field

1) An area for holding a data value. Usually refers to a delimited, unprotected area on a screen form used for entry and display of a data value. Such fields can have field attributes associated with them that control, for example, the case of letters, default values, and so on. When a form is active, the location of the cursor designates what is called the current field, the field in which the user can enter data. A field on a screen form corresponds to a column in a database, unless it has been defined as a form-only field. See also *attribute*, *current*, *form*, *form specification file*, *form-only field*, *multiple-segment field*, *screen array*, *screen form*.

2) In some database terminologies, a column. See also *column*.

field buffer

A portion of computer memory associated with and holding the current contents of a screen field. In 4GL, the GET_FLDBUF( ) built-in function allows the developer to examine the field buffer. See also *built-in function*, *field*.

field tag

A unique name that identifies a field in the SCREEN section of a form specification file. It is also used in the ATTRIBUTES section to assign a set of field attributes to the field. Unlike a label, a field tag does not appear when the form is displayed. See also *attribute*, *field*, *form specification file*, *label*.

file

1) A named collection of information stored together, usually on disk. A file can contain the words in a letter or report, a computer program, or a listing of data. Files are usually stored in directories and are managed by the operating system. See also *data file*, *directory*, *executable file*, *form specification file*, *help file*, *log*, *operating system*, *output file*, *source module*.

2) In some database terminologies, the term used for a table. See also *table*.

filename extension

The part of the filename following the period (.). It usually identifies the purpose of the file. For example, form specification files have a **.per** extension while compiled form files have a **.frm** extension. 4GL source modules have a **.4gl** extension. INFORMIX-ESQL/C files have a **.ec** extension. See also *executable file*, *file*, *form specification file*, *help file*, *source file*.

fill character

Specific ASCII characters that are used to provide formatting instructions in a format string. In 4GL reports, the ampersand (&) instructs the PRINT statement to insert leading zeros when a number does not completely fill the format string. Fill characters are used in reports and forms. See also *ASCII*, *character*, *form specification file*, *format string*, *report*.

fixed-point number

A real number with a fixed scale. In 4GL, the fixed-point number data types are DECIMAL(*p*,*s*) and MONEY. These data types store values that include a fractional part as fixed-point numbers. See also *data type*, *floating-point number*, *scale*, *simple data type*.

flag

1) A value used to indicate or flag some condition. You can define a program variable as a flag (usually assigning it the values TRUE and FALSE). See also *Boolean*, *variable*.

2) A command-line option to an operating system program is sometimes called a flag as well. See also *command line*, *operating system*.

floating-point number

A real number with a fixed precision and undefined scale. The decimal point floats as needed to represent an assigned value. In 4GL, the floating-point number data types are FLOAT, SMALLFLOAT, and DECIMAL(*p*). See also *data type*, *exponent*, *fixed-point number*, *mantissa*, *precision*, *scale*, *simple data type*.

forced residency

An option that forces UNIX to keep Informix Dynamic Server shared memory segments always resident in memory, preventing UNIX from swapping out these segments to disk on a busy system. (This option is not available in all UNIX systems.)

foreground process

A process that currently has access to a window or the screen. It requires this access because it needs to perform input or output. See also *background process*, *process*, *screen*.

form See *screen form*.

form field A field on a screen form. See *field*.

form specification file An ASCII source file that describes the logical format of the screen form and provides instructions about how to display data values in the form at runtime. You define a screen form in its source file (with a **.per** extension) and create a compiled version (with a **.frm** extension) for use at runtime. **form4gl** creates the compiled version. Sometimes this file is referred to as simply the form specification. See also *field, field tag, file, screen array, screen form*.

form-only field A field on a screen form that is not associated with any database column. Usually, a form-only field is used for display purposes only. See also *column, field, form specification file, screen form*.

formal argument In a function definition, the variable in the argument list that serves as a placeholder for an actual argument. The argument list determines the number and data types of the function's arguments. In the function call, the actual argument sends the value to the function. See also *actual argument, calling routine, formal argument, function call*.

format string A quoted string whose characters specify how to display data values. The USING operator, the FORMAT and PICTURE field attributes, and certain environment variables can use format strings. See also *attribute, environment variable, fill character, quoted string, string*.

formatted mode An output mode of a 4GL program in which screen addressing is used. 4GL enters this mode when it executes any 4GL user interaction or output statement (ERROR, MESSAGE, DISPLAY AT, and so on) except a simple DISPLAY statement (one without an AT, BY NAME, or TO clause). Output in formatted mode displays in the 4GL screen. It should not be mixed with line mode. See also *line mode*.

fourth-generation language

A programming language, approximating a natural language, designed and developed for a given class of applications. Because they focus on a specific type of application, such languages can anticipate the actions programs need to perform. As a result, many typical operations can be encapsulated into a generalized but powerful statement. Sometimes abbreviated as 4GL.

4GL is a fourth-generation language for the creation of database applications. It includes the ability to embed SQL statements in a program as well as providing additional statements, operators, and functions to assist in the creation of database applications. See also *application program*, *built-in function*, *built-in operator*, *database*, *operator*, *SQL*, *statement*.

function

1) A named collection of statements defined to perform an application task, often one that needs to be repeated. Functions can be defined to accept arguments and to return values. See also *argument*, *built-in function*, *programmer-defined function*, *return value*, *SQL*, *statement*.

2) A 4GL function (a program block defined with the FUNCTION statement) is often referred to simply as a function. See also *4GL function*.

function call

The invocation, by a calling routine, of a programmer-defined or built-in function. This syntax includes the function name followed by the actual argument values, in parentheses. Calls can be explicit (with the CALL statement) or implicit (embedding the call in a 4GL expression). See also *actual argument*, *calling routine*, *programmer-defined function*, *pass-by-reference*, *pass-by-value*.

function definition

See *4GL function*.

function key

Most keyboards have functions keys F1 through F12. In 4GL, you can define actions to perform when the user presses a certain function key. To define these actions, use the ON KEY clause of the CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or PROMPT statement, or in the KEY clause of the MENU statement. Here and in the OPTIONS statement, you can use the notation F1 through F64 to denote the individual function keys (but not all keyboards support that many function keys). See also *control block*, *key*, *logical key*.

global variable A variable defined outside all program blocks and accessible within all program blocks of the program. The scope of a global variable is all statements that follow the global variable declaration and all source modules which reference the global variable declaration. In 4GL, global variables are declared in the GLOBALS…END GLOBALS statement, either at the top of a source module (outside all program blocks), or in a separate source file. See also *declare, define, program block, scope of reference, source module, variable*.

GLS Acronym for *Global Language Support*. The ability to support the character sets and the cultural conventions for data display of most European and Asian languages. (This version of 4GL does not support right-to-left or bidirectional Middle Eastern languages.)

header A comment block at the top of an ASCII file that identifies the file contents. Contents can include the purpose, author, modification information, and other relevant information. Headers are often used in 4GL source modules and form specification files. See also *comment, form specification file, source module*.

help file A file that contains help messages for a 4GL application. Text associated with each message must be uniquely identified within a given help file. You define a help message in its source file (by default having a **.msg** extension) and create a compiled version (default **.iem** extension) for use at runtime. The **mkmessage** utility creates the compiled version. Sometimes this file is referred to as a help message file. See also *file, help message*.

Help key The logical key that the user can press in a 4GL application to display a help message for the current form, field, or menu option. By default, the physical key for help is CONTROL-W. See also *logical key, help file, help message*.

help message Text that provides information and guidelines on a specific topic or for a given context. In 4GL, each message is identified by a positive, nonzero integer, called a help number. Each number corresponds to one and only one help message text resident in the currently designated help file. Help messages can be displayed automatically or at the request of the user (the Help key). See also *Help key, help file*.

hexadecimal number A number represented in base 16. The right-most digit is multiplied by 16 to the power of zero. The digit immediately to the left is multiplied by 16 to the first power. The digit immediately to the left of that is multiplied by 16 to the second power, and so on. The characters that represent a hexadecimal number are 0-9 and A-F (for 10-15).

highlight

An inverse-video rectangular area that marks the user's place on the screen. A highlight often indicates the current option on a menu or the current character in an editing session. If a terminal cannot display highlighting, the current option often appears in angle brackets while the current character is underlined.

identifier

A sequence of letters, digits, and symbols that the compiler recognizes as a programmer-defined name of some entity. In 4GL, an identifier can include letters, digits, and underscores (_). It must have either a letter or an underscore as the first character. It can be up to 50 characters in length but the first 7 must be unique among similar program entities that have the same scope of reference. 4GL does not distinguish between uppercase and lowercase letters. Types of identifiers include variable names, cursor names, function names, report names, table names, window names, form names, prepared statement names, and report names. See also *4GL function*, *4GL window*, *case sensitivity*, *cursor*, *keyword*, *name space*, *prepared statement*, *report*, *reserved word*, *scope of reference*, *screen form*, *table*, *variable*.

implicit transaction

A transaction that automatically begins when an SQL statement that alters the database executes. The developer must explicitly indicate the end of the transaction with the COMMIT WORK and ROLLBACK WORK statements. An ANSI-compliant database uses implicit transactions. See also *ANSI compliant*, *commit*, *roll back*, *transaction*.

index

1) A database file that contains a list of unique data values, with pointers to the database rows that contain those values. Indexes are used to reduce the time required to order rows and to optimize the performance of database queries. See also *database*, *query*, *row*.

2) A subscript value into an array. See also *array*, *subscript*.

indirect typing

The process of assigning a data type to a variable by referencing a database table or column. In 4GL, indirect typing is carried out by the keyword LIKE in a variable definition. See also *column*, *data type conversion*, *define*, *table*, *variable*.

input record

A group of related values that are passed to a 4GL report for formatting. The report formats data one input record at a time. The OUTPUT TO REPORT statement sends an input record to a report. See also *report*.

Insert key The logical key that the user can press in a 4GL application to insert a new line at the current position of the screen array during the INPUT ARRAY statement. 4GL automatically inserts a line at the associated position of the program array. By default, the physical key for Insert is F1. See also *Delete key, logical key, program array, screen array.*

int_flag variable See *Interrupt key.*

integer 1) A real number with no fractional part. In 4GL, the INTEGER and SMALLINT data types can store integer values within the limits of their ranges. Boolean values are also stored as integers. See also *Boolean*, *data type*, *simple data type.*

2) In uppercase, INTEGER is a data type for storing integers whose absolute value is no greater than 2,147,483,647.

interactive A mode of execution in which a program accepts input from a user and processes that input, or a program that sends output to the screen, or a program that does both. 4GL programs that use user interactive statements are interactive. See also *batch*, *program execution*, *user interaction statement*.

interpret 1) To run a program that has been compiled to intermediate code. The executable file contains instructions in intermediate code. This translation is done by a system program called an interpreter, sometimes called a runner. See also *compile, debug, execute, link.*

2) The Rapid Development System can interpret 4GL source code by executing the intermediate code (the p-code) produced by the 4GL compiler. See also *p-code*, *Rapid Development System*.

Interrupt key The logical key that the user can press within a 4GL application to indicate cancellation of the user interaction statement. If the 4GL program does not include the DEFER INTERRUPT statement, pressing this key terminates program execution. With DEFER INTERRUPT, pressing the Interrupt key sets the built-in **int_flag** variable to TRUE and cancels the current interaction statement, resuming execution at the next 4GL statement. Further response to the Interrupt signal can be deferred, however, until the next pause for user input. The physical key for Interrupt is CONTROL-C. See also *Accept key, exception handling, interrupt signal, logical key, Quit key, user interaction statement*.

Interrupt signal A high-priority kind of signal sent to a running program either by the operating system directly or by the user. An Interrupt signal is usually a command to interrupt a running program. In a 4GL application, the user can invoke an Interrupt signal by pressing the Interrupt key. See also *Interrupt key, program execution*.

interval 1) A span of time. In 4GL, there are two types of intervals, namely *year-month* (those measured in years and months), and *day-time* (those measured in smaller time units: days, hours, minutes, seconds, and fraction of seconds). See also *data type*, *simple data type*.

2) In uppercase, INTERVAL is a data type for intervals of time.

ISAM Acronym for Indexed Sequential Access Method. An access method is a way of retrieving pieces of information (rows) from a larger set of information (table). An ISAM allows you to find information in a specific order or to find specific pieces of information quickly through an index.

join The process of combining information from two or more tables based on some common domain of information. Rows from one table are paired with rows from another table when information in the corresponding rows match on the joining criterion. For example, if a **customer_number** column exists in both the **customer** table and the **orders** table, you can construct a query that pairs each **customer** row with all the associated **orders** rows based on the common **customer_number**.

key 1) In database terminology, a *key* value is that part of a row that makes the row unique from all other rows; for example, a SERIAL number. At least one such value must exist in any row; the most important is designated as the primary key. See also *column*, *rowid*, *table*.

2) In application terminology, when speaking of the keyboard, a key is what the user presses to enter text or commands in the application. The actual keys of the keyboard (A, ESCAPE, RETURN) are often called physical keys to distinguish them from logical keys (which might be made up of a sequence of keys). See also *accelerator key*, *activation key*, *control character*, *function key*, *keyboard*, *logical key*, *mnemonic key*, *modifier key*, RETURN *key*.

keyboard focus The area on the screen that currently gets input from the keyboard; for example, a text field. The keyboard focus is usually indicated by highlighting or the presence of a text cursor. See also *active window*, *key*, *screen*, *text cursor*.

keyword A sequence of letters recognized by a compiler as having a reserved meaning within a language. In 4GL, examples of keywords are INPUT, INSERT, TO, and LIKE. In 4GL documentation, keywords are usually shown in uppercase to assist readability, but 4GL keywords are not case sensitive. See also *case sensitivity*, *identifier*, *reserved word*.

label

A character string used as a point of reference. For example, a label on a screen form helps to identify a form field. In a LABEL statement, the label is the identifier that indicates the position in a 4GL program to which GOTO statements can transfer control. See also *screen form*, *string*.

language supplement

A product obtained from an Informix sales office that provides settings for an additional national language. See also *locale file*, *Global Language Support*.

line mode

An output mode of a 4GL program in which screen addressing is not used. 4GL enters this mode and displays the line mode overlay when it executes a simple DISPLAY statement (one without an AT, BY NAME, or TO clause).4GL remains in line mode as long as it encounters additional simple DISPLAY statements or the PROMPT statement. When it encounters any other output statement (ERROR, DISPLAY AT, and so on) or a user interaction statement, 4GL returns to formatted mode. Because this mode results in a simple stream of characters to standard output, it should not be mixed with formatted mode. The OPTIONS, RUN, START REPORT, and REPORT statements can explicitly specify IN FORM MODE or IN LINE MODE to set the screen mode to formatted mode or line mode for PIPE output or for processes that RUN executes. See also *formatted mode*, *line mode overlay*.

line mode overlay

A window that overlays the entire 4GL screen when 4GL enters line mode. The line mode overlay remains as long as the program remains in line mode. It is only updated, however, when 4GL executes either a PROMPT or a SLEEP statement. See also *4GL window*, *formatted mode*, *line mode*.

link

1) To combine one or more compiled source modules into a single executable file or program. This merging is done by a system program called a linker or a link editor. The linker verifies it can locate all functions called and all variables used. See also *compile*, *debug*, *execute*, *executable file*, *source module*.

2) The 4GL Programmer's Environment can link compiled 4GL source modules into a single executable file. The Programmer's Environment can handle the entire process of compilation, linking, and running of a multimodule 4GL program. If you do not use the Programmer's Environment, you must explicitly link compiled 4GL source modules into a single executable file. See also *p-code*, *Programmer's Environment*.

literal Characters specifying some fixed value, such as a pathname, number, or date. In the format string of a PICTURE field attribute, for example, any characters except A, #, and X are literals, because they are displayed unchanged in the formatted display. In 4GL statements, literal character values must appear between single (' ') or double (" ") quotation marks. The documentation of Informix database servers sometimes uses the term *constant* in references to literals. See also *attribute, constant, form specification file, quoted string, string*.

local variable A variable declared in a program block. The scope of a local variable is limited to statements within the same program block. In 4GL, the DEFINE statement declares local variables within MAIN, FUNCTION, or REPORT program blocks. See also *declare, define, program block, scope of reference, variable*.

locale file A file installed on the system and that defines conventions for some specific language or culture, such as formatting time or money, and classifying, converting, and collating (sorting) characters. See also *language supplement*.

local variable A variable that has meaning only in the program block in which it is defined. See *variable* and *scope of reference*.

log 1) With an Informix Dynamic Server database server, a physical log contains images of entire pages before they were changed. Physical logs are used during fast recovery when Dynamic Server is coming up. See also *error log, file*.

2) A logical log, sometimes called a transaction log, records changes performed on a database during the period the log was active. A logical log includes, as needed, images of the row before it was changed and images of the row after it was changed. Logical logs are used to roll back transactions, recover from system failures, and restore databases from archives. See also *commit, roll back, transaction*.

3) The **STARTLOG( )** function of 4GL can specify an error log file in which to record runtime errors.

logical key A key that the user can press to perform certain tasks predefined by 4GL or the operating system. These include the following keys: Accept key, Delete key, Help key, Insert key, Interrupt key, Quit key. Each key is associated with some default physical key. The OPTIONS statement can assign most logical keys to different physical keys. Certain 4GL statements can reference logical keys with keywords like ACCEPT, DELETE, INSERT, and so forth. See also *Accept key, Delete key, Help key, Insert key, Interrupt key, key, Quit key*.

login The procedure that identifies a user to a computer. If the login is successful, the user is granted access to the system. See also *user name*.

main menu The top menu in a hierarchy of nested menus. See also *menu*, *ring menu*.

MAIN program block The program block that 4GL begins executing when it starts a 4GL program. This program block is defined with the MAIN statement and includes all statements between the MAIN and the END MAIN keywords. When it reaches the END MAIN keywords, 4GL ends the program. See also *4GL function*, *function*, *normal execution*, *program block*, *report*.

mantissa 1) In the representation of a FLOAT or SMALLFLOAT value, a signed integer that indicates the number that is to be raised to the power indicated by the exponent. See also *exponent*, *floating-point number*.

2) The left-hand unsigned integer operand of the exponentiation (**) operator in 4GL expressions. See also *arithmetic operators*.

member See *record member*.

menu A graphical object from which the user can choose one of several options, called menu items. Menus often control a program by providing menu items for actions that can be performed. See also *main menu*, *ring menu*.

menu option A choice the user can make from a ring menu. A menu option can be visible, in which case it appears in the ring menu. It can also be invisible, in which case the user must know the correct activation key for choosing the option. A hidden menu option cannot be activated by the user unless the program uses SHOW MENU within the MENU statement. See also *activation key, ring menu*.

mnemonic key A shorthand name for a key, menu option, or command. See also *activation key, alias*, *key*.

MODE ANSI See *ANSI compliant*.

modifier key A key that is held while pressing another key to modify its meaning. See also *control character*, *key*.

module A group of related functions. If these related functions share variables, these variables can be defined as module variables. During program execution, the current module is the source file that contains the program block currently being executed. See also *current*, *module variable, source module*.

module variable A variable defined outside all program blocks. The scope of reference of a module variable is all statements that follow its definition. In 4GL, module variables are defined with the DEFINE statement at the top of a source module, outside all program blocks. See also *declare*, *define, program block, scope of reference, source module, variable*.

multiple-segment field

In a screen form, a field consisting of several, separately delimited parts, each sharing a common field tag. Such a field allows long character strings to be displayed or entered on successive lines of the form. A multiple-segment field requires the WORDWRAP field attribute in the form specification file. See also *field*, *field tag*, *form specification file*, *screen form*.

name space

The set of identifiers of different types whose names must be unique within the same scope. For example, the following types of identifiers have the same name space: cursor names, window names, form names, function names, global variable names, and report names. Because they share the same name space, none of them can have the same name. For example, a cursor cannot have the same name as a window or a global variable. A cursor can have the same name as a local variable, however, if the cursor does not have the same scope as the variable. See also *identifier*, *naming conventions*, *scope of reference*, *variable*.

naming conventions

Guidelines for the creation of identifier names that assist the programmer in recognizing the purpose of the identifier from its name. For example, prefixes can be used to identify names of cursors (**c_**), windows (**w_**), program records (**p_**), program arrays (**pa_**), global variables (**g_**), screen arrays (**sa_**). These guidelines are distinct from *naming rules*, which the 4GL compiler and runtime enforce with error messages (and failure of your program logic) if a violation occurs. See also *identifier*, *name space*, *scope of reference*, *variable*.

navigation

Traversing fields, menus, or other controls within a 4GL window. The user can navigate by using the TAB, arrow, and other keys. See also *key*.

normal termination

The termination of the 4GL application by exiting from the MAIN program block at the END MAIN keywords. In the INFORMIX-4GL Interactive Debugger, you can no longer inspect the application state after normal termination because there are no active functions. (Some earlier versions of 4GL treated the RETURN statement in the MAIN program block as a means of normal termination, but this is now treated as an error.) See also *abnormal termination*, *active function*, *debug*, *MAIN program block*, *program execution*.

null value

1) A value meaning *not known* or *not applicable*. Every data type can represent a null value, which is distinct from a string of blanks or from a value of zero. Database columns and program variables can have null values. In 4GL, this is represented by the keyword NULL. To test for a null value, use the IS NULL and IS NOT NULL operators. See also *Boolean operators*.

2) In some contexts, null is casually used to mean *empty*; for example, a character string with zero length is sometimes called the null string. This can lead to confusion, because an empty string (the string " ") has a specific non-null value, distinct from a null string. An empty string has a definite length (zero) while a NULL character value has an unknown length and value. In several contexts, however, 4GL represents a NULL character value as the empty string or as a single blank. See also *blank space*, *string*.

open

To prepare something for use. In programming, opening something often entails allocating memory and other resources to it, and sometimes means getting exclusive access. Typically, things cannot be used until they have been opened; they remain usable until they are closed. To open a cursor means to have the database server process the query up to the point of locating the first selected row; this can entail significant processing and space in memory and on disk. To open a file is to locate the file and bring it into memory. To open a form is to find the compiled form file, bring it into memory, and prepare to display it. To open a 4GL window is to allocate memory for the image of the window, push it onto the window stack, and to display it on the screen. See also *4GL window*, *close*, *cursor*, *file*, *query*, *resources*, *screen form*.

operand

A value on which an operation is performed by an operator. An operand can be a variable, a constant, a literal value, a function that returns a single value, or another expression. See also *constant*, *expression*, *operator*, *variable*.

operating system

The software that provides an interface between application programs and hardware. It is the part of a computer system that makes it possible for the user to interact with the computer. It manages processes by allocating the resources they need. See also *command line*, *execute*, *process*, *resources*.

operator

A symbol or keyword built into a language that returns a value from the values of its operands. Operators can generate a value from a single value (unary operators) or from two values (binary operators). See also *arithmetic operators*, *assign*, *associativity*, *binary operator*, *Boolean operators*, *built-in operator*, *operand*, *precedence*, *relational operators*, *string operators*, *unary operators*.

output file

A file in which the results of a query or a report are stored. See also *file*, *query*, *report*.

owner A designation that associates an individual with a file or set of files. Informix databases can use ownership to restrict access to certain columns or tables. On UNIX systems, ownership also applies to files and directories for the purpose of limiting access to their contents and location within the file system. See also *database*, *operating system*.

p-code Abbreviation for pseudo-code. P-code is an intermediate form of code generated by the Rapid Development System. Although p-code takes more memory to run, it is machine independent. See also *compile*, *debug*, *execute*, *interpret*, *link*.

page A unit of data analogous to the page of a book. One page of a program array is the number of rows that can be displayed in the screen array at one time. The database server stores data in pages. See also *program array*, *screen array*.

page header The top part of a page in a report. A running header appears at the top of each page of a report. Information (for example, the title and date) printed at the top of each page of a report is formatted in the PAGE HEADER and FIRST PAGE HEADER control blocks of a report. See also *control block*, *page trailer*, *report*.

page trailer The bottom part of a page in a report. Information (for example, the page number) printed at the bottom of each page of a report is formatted in the PAGE TRAILER control block. A page trailer is also referred to as a footer. See also *control block*, *page header*, *report*.

pass-by-reference A method used in a function call that determines how an argument is passed to the programmer-defined function. With pass-by-reference, the address in memory of the actual argument is passed to the function. This method means that changes made to the value of the formal argument within the body of the function will be visible from the calling routine when the function terminates. 4GL uses pass-by-reference only for blob (BYTE and TEXT) variables. See also *argument*, *blob*, *function call*, *pass-by-value*.

pass-by-value A method used in a function call that determines how an argument is passed to the programmer-defined function. With pass-by-value, the actual argument is evaluated and the resulting value is passed to the function. This method means that changes made to the value of the formal argument within the body of the function will not be visible from the calling routine when the function terminates. 4GL uses pass-by-value for variables of all data types except blob (BYTE and TEXT). See also *argument*, *blob*, *data type*, *function call*, *pass-by-reference*.

pathname The list of directories needed to identify a file within a directory hierarchy. In UNIX, directories of a pathname are separated by the slash (/). A file can be referred in two ways: by its absolute pathname—all directories starting from the root (top) of the directory hierarchy—or by its relative pathname—the directories relative to the current directory. See also *current*, *directory*, *file*.

permission On some operating systems, the right to have access to files and directories. Compare with *privileges*.

phantom row A row of a table that is initially modified or inserted during a transaction but subsequently rolled back. Another process might see a phantom row if the isolation level is DIRTY READ. No other isolation level allows a changed but uncommitted row to be seen.

pipe A connection of one process to another process such that the output of the first process is sent directly as input to the second process. It is one of several ways in which processes can communicate. It is common to speak of a process piping some data to another process. See also *process*.

pop To remove a value from a stack in memory. See *stack* and *push*.

popup window A 4GL window that automatically appears when a predefined condition or event occurs. In a 4GL application, a popup window often contains a list of values for a given field. The user can choose from this list rather than needing to type in the value directly. See also *4GL window*.

precedence of operators The hierarchy of operators. It determines the order in which 4GL evaluates operators within an expression. 4GL evaluates higher-precedence operators before those of lower precedence. The order in which operators at the same precedence level are evaluated is left to right. Precedence order can be changed by surrounding expressions with parentheses. See also *associativity*, *expression*, *operator*.

precision The total number of significant digits in the representation of a numeric value or in a data type specification. The number 3.14 has a precision of 3. See also *floating-point number*, *scale*.

prepared statement The executable form of an SQL statement. SQL statements can be executed dynamically by creating character strings with the text of the statement. This character string must then be prepared with the PREPARE statement. The result of the PREPARE is a prepared statement. The prepared statement can then be executed with the EXECUTE or DECLARE statements. See also *query by example*, *SQL*, *statement*, *statement identifier*.

preprocessor A program that translates macro code into statements that conforms to the host language. The results of preprocessing can then be passed to a standard language compiler, such as C or COBOL. When using the C Compiler of 4GL, the compiler first sends the 4GL source module through a preprocessor to translate SQL statements into INFORMIX-ESQL/C calls before passing the file to a C compiler. See also *compile*, *source module*.

primary key The information from a column or set of columns that uniquely identifies each row in a table. The primary key is sometimes called a unique key.

print position The logical location of the print head. A print position can be compared to a screen cursor in that both refer to a specific x,y coordinate on the page or screen. See also *column*, *report*, *row*.

printable character A character that can be displayed on a terminal or printer. The locale files specify what characters are printable. These characters might include the ASCII codes 32 through 126: *A-Z*, *a-z*, 0-9, symbols (!, #, $, ;, *, and so on), TAB (CONTROL-I), NEWLINE (CONTROL-J), FORMFEED (CONTROL-L) and the blank space character. See also *ASCII*, *blank space*, *character*.

privileges The right to use or change the contents of a database.

process An independent unit of operating system execution. It keeps track of the state of execution for a program. The operating system creates a process for each program being executed. It allocates resources needed by a program (memory, disk, CPU) to its process. The current process is the one that has been allocated use of the CPU. A 4GL application usually runs with two processes: the 4GL application (the front end) and a database server (the back end). See also *application development tool*, *background process*, *database server*, *foreground process*, *operating system*, *pipe*, *resources*, *shell*.

program array A 4GL variable defined with the ARRAY keyword. A common use for a program array is as an array of records to store information to be displayed in a screen array. The DISPLAY ARRAY and INPUT ARRAY statements can manipulate program array values or records within the screen array. See also *array*, *program record*, *screen array*, *structured data type*, *variable*.

program block A programmer-defined group of 4GL statements that has its own scope during execution. The scope includes definitions and values of variables and can include arguments and return values. 4GL has the following program blocks: a MAIN program block, 4GL functions, and reports. Every executable statement must appear within some program block. Program blocks can neither overlap nor be nested. Any variable defined within a program block is local to that block. See also *4GL function, call stack, executable statement, MAIN program block, programmer-defined function, report, scope of reference, statement block, variable*.

program execution The process of running (executing) a program. A program can be in the following states: running, suspended (by the system or the program itself), or terminated (abnormally or normally). See also *abnormal termination*, *debug*, *execute*, *normal termination*.

program record A 4GL variable defined with the RECORD keyword. A common use for a program record is for storing information in a screen record, a row of a table, or in a line of a screen array. See also *program array*, *record*, *row*, *screen record*, *structured data type*, *variable*.

program design database A database that describes the resources needed to create some executable programs. It is called **syspg4gl** (by default), and is accessed by the Programmer's Environment. Regardless of the version of 4GL that you are using, (RDS Version or C Compiler), this database tracks for each 4GL program such resources as source files and compiler options. See also *executable file*, *Programmer's Environment*.

programmer-defined function A function written in 4GL that can be called in a 4GL program. The developer can write 4GL functions (defined with the FUNCTION statement), a MAIN program block (defined with the MAIN statement), and reports (defined with the REPORT statement). Collectively these types of functions are often called 4GL program blocks. See also *4GL function*, *function*, *built-in function*, *built-in operator*, *MAIN program block*, *program block*, *report*.

Programmer's Environment The interface to the 4GL application development package. The Programmer's Environment is an integrated development environment that allows you to create, compile, link, run, and debug a 4GL program. See also *C Compiler*, *compile*, *debug*, *development environment*, *execute*, *link*, *program design database*, *Rapid Development System*, *target*.

push To place a value onto a stack in memory. See *stack* and *pop*.

query A request to the database to retrieve data that meets certain criteria. The SELECT statement performs database queries. In 4GL, the CONSTRUCT statement allows you to implement a query by example. See *database*, *exception*, *output file*, *query by example*.

query by example A formalized way of implementing a query. The CONSTRUCT statement allows the user to enter query criteria on a screen form, and then it creates a Boolean expression based on these criteria. This Boolean expression can then be appended to an SQL statement (usually a SELECT) to retrieve the desired rows from the database. The SQL statement must then be prepared and executed. See also *Boolean expression*, *cursor*, *query*, *prepared statement*, *query criteria*, *screen form*.

query criteria A set of data values that specify qualifications to apply when looking for data to be returned in a query. The CONSTRUCT statement accepts query criteria on a screen form. See also *data entry*, *query by example*, *screen form*, *user interaction statement*.

Quit key The logical key that the user can press within a 4GL application to indicate cancellation of the entered data or query criteria. Pressing it requests abnormal completion of the INPUT, CONSTRUCT, PROMPT, INPUT ARRAY, or DISPLAY ARRAY statements. The physical key for Quit is CONTROL-\. If the 4GL program does not include the DEFER QUIT statement, pressing this key terminates program execution. With DEFER QUIT, pressing the Quit key sets the built-in **quit_flag** variable to TRUE but does not cancel the current interaction statement. See also *Accept key*, *exception handling*, *Interrupt key*, *logical key*.

quit_flag variable See *Quit key*.

quoted string A string enclosed in double quotation (" ") marks. With the exception of fill characters, the contents of quoted strings are literals. See also *character*, *fill character*, *literal*, *string*.

Rapid Development System (RDS) One of two implementations of the 4GL application development language for UNIX systems. The RDS compiler produces p-code that can then be executed by a runner. The other implementation of 4GL for UNIX systems is the C Compiler; it uses preprocessors to generate C code, which is then compiled and linked to make a stand-alone, executable file. See also *C Compiler*, *compile*, *execute*, *interpret*, *link*, *p-code*, *preprocessor*, *Programmer's Environment*, *program design database*.

raw device A UNIX disk partition that is defined as a character device and that is not mounted. The UNIX file system is unaware of a raw device.

raw I/O

The process of transferring data between memory and a raw device. Informix Dynamic Server can bypass the UNIX file system and address raw devices directly. The advantage of raw I/O is that data on a disk can be organized for more efficient access. Raw I/O is sometimes referred to as direct I/O.

record

1) A data structure that has a fixed number of components. Each component is called a member. Members can have the same or different data types. See also *input record*, *record member*, *screen record*.

2) In uppercase, RECORD is the keyword for defining a program record in 4GL. The RECORD data type is a structured data type. In 4GL, all members of the record are accessed by listing the record name followed by the member name, with a period (.) separating them. See also *asterisk notation*, *program record*, *structured data type*.

3) In some database terminologies, a term used for a row. See also *row*.

record member

A named component of a program record. A member can be of any 4GL data type, including RECORD or ARRAY. Some 4GL statements support the asterisk notation (*record*.*) to specify all the members of a record. See also *asterisk notation*, *program record*, *record*.

regular expression

A pattern used to match variable text. The elements of a regular expression include literal characters that must match exactly; the wildcard symbols—asterisk (*) to mean one or more characters here and question mark (?) to mean any one character, and the class—a list of characters (within brackets) that are acceptable. The MATCHES and LIKE operators of SQL allow you to search for character strings that match to regular expression patterns. See also *expression*, *literal*, *wildcard*.

relation

See *table*.

relational database

See *database*.

relational operators

Operators that perform comparison operations. These operators return the values TRUE (=1), FALSE (=0), and in some cases UNKNOWN. (If an operand evaluates to NULL, Boolean operators can yield a third unknown result that 4GL treats as FALSE.) 4GL relational operators are: equal (=), not equal (!= or <>), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). All relational operators have the same precedence level. See also *associativity*, *binary operator*, *Boolean operators*, *operator*, *precedence*.

REPEATABLE READ

A level of process isolation available through Informix Dynamic Server that ensures all data read during a transaction is not modified by another process. Transactions under REPEATABLE READ are also known as serializable transactions. It is the default level of isolation under Informix Dynamic Server for MODE ANSI databases.

report

A 4GL program block defined with the REPORT statement. A report formats data sent as input records. The report header follows the REPORT keyword and defines the name and formal argument list (the input record) for the report. The report body (all statements between the report header and the END REPORT keywords) defines the actions of the report. See also *argument*, *control block*, *function*, *input record*, *output file*, *page header*, *page trailer*, *program block*, *programmer-defined function*.

reserved lines

Areas in a 4GL window that are set aside for use by the form or window. These areas include the Error line, where the output of the ERROR statement appears (the default is the last line on the screen); the Comment line, where the text in the COMMENTS field attribute appears (the default is the next to last line on the screen and the last line on all other 4GL windows); the Form line, where the first line of the screen form appears (the default is the third line of the current 4GL window); the Menu line, where the ring menu appears (the default is the first line of the current 4GL window); the Message line, where the output of the MESSAGE statement appears (the default is the second line of the current 4GL window); and the Prompt line, where the output of the PROMPT statement appears (the default is the 1st line of the current 4GL window). These default positions can be changed with the OPTIONS statement or with the ATTRIBUTES clause of the OPEN WINDOW statement. See also *4GL window*, *ring menu*, *screen*, *screen form*.

reserved word

A string that you cannot use in any other context of the language or program. In 4GL versions before 4.1, all keywords were reserved words. In Version 4.1 and higher, you can use keywords as identifiers, if they do not create an ambiguity for the 4GL compiler. If you compile a 4GL program for an ANSI-compliant database, many keywords are still reserved words, and therefore should not be declared as identifiers. See also *ANSI compliant*, *identifier*, *keyword*.

resources

1) The hardware and software needs of an executing program. Examples include CPU, memory, disk, printer, terminal (or workstation). These are allocated to the program's process by the operating system. See also *operating system*, *process*, *terminal*.

2) Visual and other attributes that can be chosen at runtime. Resources can be chosen by using command-line options or by using other platform-specific methods. On X/Motif systems, resources can be specified using pre-defined resource names in some files. 4GL statements and ATTRIBUTE settings can override resource choices made through the native resource management systems of a platform. See also *attribute*.

RETURN key

The key to indicate the end-of-line. The RETURN key is the default Accept key in 4GL. See also *key*.

returned value

The value returned by a 4GL function to the calling routine. To return a value, the FUNCTION program block must include the RETURN statement. The calling routine must have some way of handling returned values for a function. Reports cannot return a value to a calling function. See also *4GL function*, *calling routine*, *function definition*, *programmer-defined function*.

ring menu

A menu in which the items appear in a single horizontal line. Each menu item is called a menu option. The user can press the SPACEBAR or RIGHT ARROW to make the menu cursor traverse the menu like a ring, as if the first option followed the last. In 4GL, the MENU statement creates a ring menu. The highlighted option is the current option. If the menu includes more options than the current 4GL window can display on a single line, the menu continues onto successive pages, with the first page following the last. The line below the Menu line can display text that describes the current option. See also *activation key*, *menu*, *menu option*, *page*, *user interaction statement*.

roll back

To terminate a transaction by undoing any changes to the database since the beginning of the transaction. The database is restored to the state that existed before the transaction began. When the transaction is rolled back, all open database cursors (except hold cursors) are closed and all locks are released. The ROLLBACK WORK statement rolls back the current transaction. See also *commit*, *cursor*, *log*, *transaction*.

root dbspace

The initial dbspace for an Informix Dynamic Server system. In addition to any data, the root dbspace contains all system management tables, the physical log, and at least the initial logical log.

row

1) In a database, a row is a set of related values, called columns, stored together in a table. A table holds a collection of rows, each one distinct from the others in the contents of its key. In other database terminologies, a row is sometimes called a record or a tuple. See also *column, current, cursor, key, rowid, table.*

2) In a screen form, a row is the visible display of the values from one database row. The row (of data fields on the screen) might or might not be identical to a row (of values in a table in the database). A single line of a screen array is sometimes called a row. See also screen array.

3) In a report, a row is the information sent by the report driver function. A 4GL program generates a report by sending rows of data to a report function. These rows might or might not correspond to database rows. These rows are also called input records. See also *input record.*

4) On a screen or in output from a report, a *row* (or *line*) is the y-coordinate of a given position on the vertical axis. The x-coordinate (of the horizontal axis) is called a *column*. Several 4GL statements use *rows* (or *lines*) and *columns* in this sense to identify locations within displays. See also *column, screen*.

rowid

A hidden, automatically generated column in each table of some Informix databases. It uniquely identifies a row, based on its position within the table. A **rowid** number is assigned when each row is added to a table and released when a row is deleted. Once assigned, the **rowid** for a specific row cannot be changed, and the **rowid** number cannot be reused for that table. See also *column, row, table.*

run

See *execute, interpret*.

runtime errors

Errors that occur during program execution. See also *compile-time errors*.

scale

The number of digits to the right of the decimal point in the representation of a number or in a data type specification. The number 3.14 has a scale of two. See also *fixed-point number, floating-point number, precision*.

scope of reference

The portion of the 4GL source code in which the compiler can recognize an identifier name. The scope of reference (often referred to simply as scope) refers to the program blocks in which an identifier can be referenced. Outside its scope, an identifier might not be defined or might even be defined differently. In 4GL, there are three levels of scope: local (a single program block), module (all program blocks in a single source module), and global (all program blocks within a program). See also *define, global variable, identifier, local variable, module variable, name space, program block, scope, source module, variable*.

screen

1) On a character terminal, the rectangular area on a CRT in which text is displayed.The screen takes up the entire terminal display and it can display the output of only one program at a time. See also *terminal*.

2) On a workstation, the entire display in which text and, possibly, graphical objects are visible. Under a window manager in a graphical environment, a physical screen might contain multiple graphical windows.

3) In a 4GL application, the default 4GL window that is displayed in the 4GL screen. This logical screen is a data structure kept in memory that is a representation of a 4GL screen. The logical screen is not directly affected by window manager operations, though its graphical image on the physical screen might change. See also *4GL screen*, *4GL window*, *column*, *row*.

screen array

In a 4GL form, a screen array consists of consecutive lines that contain identical fields and field tags. Each line of the screen array is a screen record. The screen array defines the region of the form that will display program array values. The DISPLAY ARRAY and INPUT ARRAY statements can manipulate program array values or records within a screen array. See also *field*, *field tag*, *program array*, *row*, *screen record*, *scrolling*.

screen field

See *field*.

screen form

A data-entry form displayed in a 4GL window (or the 4GL screen) and used to support input or output tasks in a 4GL application. A screen form is defined in a form specification file. Before a 4GL program can use a screen form, this file must first be compiled. The form in the current 4GL window is called the current form. Most user interaction statements use a screen form for their input and output. See also *4GL window*, *4GL screen*, *active form*, *attribute*, *current*, *form specification file*, *reserved lines*, *user interaction statement*.

screen record

A named group of fields on a screen form. Screen forms have one default screen record for each table referred to in the TABLES section, including FORMONLY. The name of a default screen record is the same as the name of the table. See also *program record*, *record*, *screen array*, *table*.

scrolling

To move forward and back (or up and down) through a series of items. Referring to a screen array, scrolling is the action of bringing invisible lines into view. Displayed data can be scrolled either vertically (to bring different rows into view) or horizontally (to show different columns). Referring to database cursors, a sequential cursor can return only the current row and cannot return to it, but a scroll cursor can fetch any row in the active set. Thus a scrolling cursor can be used to implement a scrolling screen display. See also *cursor*, *screen array*.

search path The list of directories in which the operating system or a program will look for needed files. This path can be set by the user. Often, the user can specify several different paths to be searched; if one path does not lead to the file, one of the others might. For executable files, the setting of an environment variable called **PATH** is used. For Informix database files, the setting of the **DBPATH** environment variable is used. See also *database*, *environment variable*, *operating system*.

self-join A join between a table and itself. A self-join occurs when a table is used two or more times in a SELECT statement (under different aliases) and joined to itself.

semaphore A UNIX communication device that controls the use of system resources.

shell A process that handles the user interaction with the operating system. From the shell, the user can execute operating system commands. A shell is usually provided to contain activity in a given part of the computer system. In UNIX, for example, the shell handles command-line input, and standard output and error reporting. UNIX shells have their own special commands that are not usable within applications. They even have their own special variables and scripting facilities that make the user interface customizable. See also *command line*, *environment variable*, *operating system*, *process*.

simple data type Any 4GL or SQL data type that has no component values. Simple data types include integer (SMALLINT, INTEGER); floating-point (FLOAT, SMALLFLOAT, DECIMAL(*p*)); fixed-point (DECIMAL(*p*,*s*), MONEY); time (DATE, DATETIME, INTERVAL); and character (CHAR, VARCHAR). Although individual characters in a string can be accessed, as in a C language character array, the data types CHAR and VARCHAR are considered simple data types, rather than structured data types. See also *blob*, *character*, *data type*, *fixed-point number*, *floating-point number*, *integer*, *interval*, *structured data type*.

singleton transaction A transaction that is made up of a single SQL statement. The transaction automatically begins before each SQL statement that alters the database executes and ends when this statement completes. If the single SQL statement fails, the transaction is rolled back; otherwise it is committed. A database that is not ANSI compliant and that does not use transaction logging uses singleton transactions. See also *ANSI compliant*, *commit*, *roll back*, *transaction*.

source file A file that contains source code for a language; it is used as input to a compiler or interpreter. See also *compile*, *file*, *interpret*, *source module*.

source module A module that contains one or more related 4GL program blocks. A source module is a single ASCII file with the **.4gl** extension. Several source modules can be compiled and linked to produce a single executable file. See also *compile, executable file, execute, file, file extension, link, module, program block.*

SQL Acronym for structured query language. A database query language developed by IBM and standardized by an ANSI standards committee. Informix relational database management products are based on an extended implementation of ANSI-standard SQL. See also *cursor, database, prepared statement, statement identifier.*

SQLCA record Acronym for SQL Communications Area. It is a built-in record that stores information about the most recently executed SQL statement. The **SQLCODE** member stores the result code returned by the database server; it is used for error handling by 4GL and the Informix embedded-language products. The **SQLAWARN** member is a string of eight characters whose individual characters signal warning conditions. **SQLERRD** is an array of six integers that returns information about the results of an SQL statement. See also *database server, error handling, status variable*.

stack A data structure that stores information linearly with all operations performed at one end (the top). Such types of data structures are often called LIFO (last-in, first-out) structures. Stack operations include push, which adds a new piece of data to the top of the stack, and pop, which removes the piece of information at the top of the stack. 4GL uses one stack to transfer arguments to C functions and another to keep track of open 4GL windows. See also *4GL window, call stack.*

statement An instruction that 4GL executes. This instruction is a single executable unit of program code but might cover several lines within the source module. For example, the FOR statement might have several lines between the line introduced with the FOR keyword and the line introduced with END FOR keywords. The FOR statement, however, is a single statement because it performs a single action. During program execution, the statement currently being executed is often called the current statement. A statement is distinct from a command: the LET or PRINT command is executed by the INFORMIX-4GL Interactive Debugger; the LET or PRINT statement can be compiled and executed by 4GL. See also *current, source module, statement block.*

statement block A group of statements executed together. For example, all statements between the WHILE keyword and the END WHILE keywords constitute a statement block. All the statements within the AFTER INPUT block of the INPUT (or INPUT ARRAY statement) are also considered a statement block. See also *control block*, *program block*, *statement*.

statement identifier The name that represents a prepared statement created by a PREPARE statement. It is used in the management of dynamic SQL statements by 4GL and the Informix embedded language products. See also *identifier*, *prepared statement*.

status variable The built-in variable that 4GL sets after executing each SQL and form-related statement. If the statement is successful, **status** is set to zero. If the value of **status** is negative, 4GL terminates program execution unless the program contains the appropriate error handling. After execution of SQL statements, 4GL copies the value of **SQLCA.SQLCODE** into **status**. See also *error handling*, *SQLCA record*.

string A value that consists of one or more characters. You can store strings in CHAR, VARCHAR, and TEXT variables. Strings can include printable or nonprinting characters, but 4GL does not provide facilities to display nonprintable characters. Literal string values in 4GL statements generally must be enclosed within quotation (" ") marks. See also *character*, *literal*, *printable character*, *quoted string*, *subscript*, *substring*.

string operators Operators that perform operations on character strings. 4GL string operators include concatenation (| |) and substring ([ ]) operators, the CLIPPED operator that truncates trailing white space, and the USING operator that can apply formatting masks to character strings. See also *associativity*, *built-in operator*, *clipped*, *concatenate*, *operator*, *precedence*, *subscript*.

structured data type Any 4GL data type that contains component values. Structured data types include ARRAY and RECORD. Although individual characters in a string can be accessed, the data types CHAR and VARCHAR are considered simple data types, not structured data types. See also *array*, *data type*, *record*, *simple data type*.

subquery A query that is embedded as part of another SQL statement.

subscript An integer value to access a single part or element of certain data structures like strings and arrays. In 4GL, the subscript operator is an integer value, surrounded by brackets ([ ]). For example, the syntax "**strng[3]**" accesses the third character of the CHAR string variable by specifying a subscript (or index) of 3; the syntax "**pa_customer[5]**" accesses the fifth element of the **pa_customer** program array. Two subscripts allow you to specify the starting and ending characters. For example, "**strng[3,10]**" accesses the third through tenth characters of **strng**. See also *array, character, program array, string, string operators, substring*.

substring Consecutive characters within a string. To access a substring in a character expression, put square brackets around a pair of comma-separated unsigned integers to specify the location of the substring within a character string. For example, "**strng[3,10]**" accesses the third through tenth characters of **strng**. See also *character, subscript, string*.

system catalog Database tables that contain information about the database itself, such as the names of tables or columns in the database, the number of rows in a table, information about indexes and database privileges, and so forth. See also *column, database, index, table*.

system log The UNIX file that the database keeps to record significant events like checkpoints, filling of log files, recovery of data, and errors.

table A collection of related database rows. It can be thought of as a rectangular array of data in which each row describes a set of related information and each column contains one piece of the information. A table sometimes is referred to as a file or a relation. See also *column, cursor, database, key, row, rowid*.

target The intended result of a build. The Programmer's Environment can create executable files from multiple source modules. This process is called a build. See also *executable file, Programmer's Environment, source module*.

termcap An ASCII file in UNIX systems that contains the names and capabilities of all terminals known to the system. See also *terminal, terminfo*.

terminal A peripheral device usually centered around a raster screen. Terminals usually come with keyboards, and are used by the user of a computer system to communicate with the computer by typing commands in and looking at the output on the screen. Terminals are often character-based and are thereby distinguishable from displays that are usually graphical. Terminals support monochrome or color output, depending on their designed capabilities and computer system configuration. See also *character*, *key*, *screen*.

terminfo A database in UNIX systems that contains compiled files of terminal capabilities for all terminals known to the system. See also *terminal*, *termcap*.

text 1) In the SCREEN section a 4GL form, any characters outside the fields, such as labels, titles, and ornamental lines. See also *label*.

2) In uppercase letters, TEXT is the 4GL and SQL data type that can store up to 2^{31} bytes of character data. See also *blob*.

text cursor Pointer within a text field that shows the position where typed text will be entered. Often referred to simply as the cursor. See also *cursor*, *text*.

text editor System software used to create and to modify ASCII files. Usually source code is entered into a source file in a text editor. See also *ASCII*, *source file*.

text field A graphical object for displaying, entering, and modifying text, a single line of character data. For example, form fields are text fields for use in screen forms. Text fields are used more generally, for example, to accept text in PROMPT statements. See also *field*, *screen form*, *text*.

tic A UNIX program that compiles **terminfo** source files or **terminfo** files that have been decompiled using **infocmp**.

tracepoint A named object, specified by a debugger, that the programmer can associate with a statement, program block, or variable. When the tracepoint is reached, the debugger displays information about the associated statement, program block, or variable and executes any optional commands that are specified by the programmer. A tracepoint must be enabled to take effect. See *debugger*.

transaction A collection of one or more SQL statements that must be treated as a single unit of work. The SQL statements within the transaction must all be successful for the transaction to succeed. If one of the statements in a transaction fails, the entire transaction can be rolled back (cancelled). If the transaction is successful, the work is committed and all changes to the database from the transaction are accepted. The transaction log contains the changes made to the database during a transaction. If a database is not ANSI compliant, it uses singleton transactions if it does not use a transaction log and it uses explicit transactions otherwise. If a database is ANSI compliant, it uses implicit transactions. See also *ANSI compliant*, *commit*, *explicit transaction*, *implicit transaction*, *log*, *roll back*, *singleton transaction*.

truncation The process of discarding trailing characters from a string value, or discarding trailing digits from a number. Truncation can produce a warning or error in data type conversion, if the receiving data type has a smaller length or scale than the source data type. It can also cause rounding errors. See also *data type conversion*, *error*, *scale*.

tuple See *row*.

unary operator An operator that requires only one operand. The unary operator appears before the operand. In an expression, unary operators always have higher precedence than binary operators. In 4GL, examples include logical NOT, unary plus (+), and unary minus (-). 4GL associates most unary operators from right to left. See also *arithmetic operators*, *associativity*, *binary operator*, *Boolean operators*, *operand*, *operator*, *precedence*.

UNIQUE CONSTRAINT A specification that a database column (or composite list of columns) cannot contain two rows with identical values. You can assign a name to a constraint.

user An individual who interacts with a 4GL program. The user is a person or process that uses an application program for its intended purpose. Also referred to as the end user. The documentation for Informix database servers sometimes applies this term to the programmer. See also *application program*, *developer*.

user interaction statement A 4GL statement that allows a user to interact with a screen form or a field. These statements include CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, MENU, and PROMPT. They suspend execution of the 4GL application for user input. See also *attribute*, *control block*, *data entry*, *query by example*, *ring menu*, *statement*.

user interface The portion of a software or hardware system where communication to or from a human can occur. In 4GL applications, it is that part of a program that waits for input from the user of that program and displays messages or other output based on that user's input. Typical user interfaces include menus, prompts, screen forms, and on-line help messages. See also *shell*.

user name A string that identifies a specific user to a computer. It is common for it to be based on an actual name, but this is not required. Once created, the user name is used thereafter during the login process, in establishing ownership, and in permissions and privileges. See also *login*, *owner*.

variable A named storage location that holds a value (which can be modified) of a specified data type. A program can access and change this value by specifying its name. A 4GL variable (sometimes called a program variable) can transfer information between a 4GL form, report, and program. To use a 4GL variable, you must first declare it (with the DEFINE statement) to specify its name and data type. Names of variables must follow the naming rules for 4GL identifiers. See also *assign*, *constant*, *declare*, *define*, *global variable*, *identifier*, *local variable*, *module variable*, *name space*, *screen form*.

virtual column A derived column of information that is not stored in the database. For example, you can create virtual columns in a SELECT statement by arithmetically manipulating a single column, such as multiplying its values by some quantity, or by combining columns, such as adding the values from two columns.

warning An exception that indicates an unexpected or abnormal condition that could lead to an error in processing or data storage. Warnings can be generated because of language syntax during compilation or because of processing or data exceptions. At runtime, warnings can be generated by the 4GL program or by the database server. By default, 4GL continues execution when it encounters a warning. The developer can change this default behavior with the WHENEVER WARNING directive. See also *database server*, *exception*.

wildcard In a pattern-matching string, a character that means any characters at this point. A For example, in the pattern `"v*.4gl"` the asterisk means any number of characters after the `v` and preceding the period. See also *regular expression*.

window In 4GL, a rectangular area on the screen in which you can take actions without leaving the context of the background program.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z