HCL Informix 4GL Concepts and Use

Version 7.51

Table of Contents

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 introduces INFORMIX-4GL and provides a context for understanding the other manuals in the documentation set. It covers 4GL goals (the kinds of programming the language is meant to facilitate), concepts and nomenclature (parts of a program, database access, screen forms, and report generation), and methods (how groups of language features are used together to achieve particular effects).

Organization of This Manual

INFORMIX-4GL is a suite of tools that allow you to efficiently produce complex interactive database applications. Using the 4GL language, you can quickly write sophisticated, portable, forms-driven, full-screen applications for data entry, data lookup and display, and report generation. The 4GL development environment provides all the tools necessary to design forms, construct and manage program modules, and compile source modules.

This manual describes 4GL at the following three levels:

- **What 4GL is.** Part I covers the main features of 4GL, describes the kind of work it is meant to do, and the ways it is normally used:
  - Chapter 1, "Introducing INFORMIX-4GL," describes what 4GL is used for and where the program can be run.
  - Chapter 2, "Interfaces of INFORMIX-4GL," describes how 4GL allows interactive forms, SQL databases, sequential files, and reports to work together.
  - Chapter 3, "The INFORMIX-4GL Language," introduces the procedural and non-procedural aspects of the language.
- **How 4GL is designed.** Part II covers the fundamental ideas behind the design of 4GL so you will know its parts and how they fit together:
  - Chapter 4, "Parts of an Application," is an overview of the components of a 4GL application.
  - Chapter 5, "The Procedural Language," describes three basic 4GL language features: data definition, decisions and loops, and handling error conditions.
  - Chapter 6, "Database Access and Reports," examines the relationship between the data in a SQL database and 4GL reports.

  Some partial coding examples are used to illustrate the main points. Read Part II before you begin using 4GL.

- **How to use 4GL.** Part III explores 4GL in depth, using examples and discussion to show how its statements are used together to solve common programming tasks and build an application:
  - Chapter 7, "The User Interface," reviews the major components of an interactive 4GL application.
  - Chapter 8, "Using the Language," details the data types available in 4GL, variables, data structures and use of arrays, and other, similar topics.
  - Chapter 9, "Using Database Cursors," overviews non-procedural, row-by-row, and dynamic methods of accessing an SQL database from a 4GL application.
  - Chapter 10, "Creating Reports," shows how to design 4GL report drivers and report formatters.
  - Chapter 11, "Using the Screen and Keyboard," takes a detailed look at the methods of specifying a screen form and managing 4GL windows.
  - Chapter 12, "Handling Exceptions," looks at the problem of handling anticipated and unanticipated situations when running a 4GL application.

This manual has a companion, *INFORMIX-4GL Reference*, which shows every part of the language in great detail. *INFORMIX-4GL Concepts and Use* does not cover every feature of every statement but provides you with the vocabulary and understanding necessary to use INFORMIX-4GL and *INFORMIX-4GL Reference*.

Types of Users

This manual is written for all 4GL users. You do not need database management experience or familiarity with relational database concepts to use this manual. A knowledge of Structured Query Language (SQL) however and experience with a high-level programming language would be useful.

Software Dependencies

This publication is written with the assumption that you are using HCL Informix 14.10 as your database server.

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 Global Language Support (GLS) locale.

The examples in this manual are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for dates, times, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use non-ASCII characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules 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 locales, see the *Informix Guide to GLS Functionality*.

Accessing Databases from Within 4GL

The 4GL language approximates a superset of the Informix implementation (Version 4.1) 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. (Exceptions are the CREATE SCHEMA, DESCRIBE, INFO, SET ISOLATION, and SET LOG statements.)
- For all SQL statements that can be prepared, you can use the PREPARE feature of SQL to include SQL statements as text within prepared objects; see the description of PREPARE in *INFORMIX-4GL Reference*.
- For any SQL statement that your database server supports, you can use the SQL ... END SQL delimiters.

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.

For more information about using PREPARE and SQL ... END SQL delimiters, see *INFORMIX-4GL Reference*. For additional information on SQL statements, see the *Informix Guide to SQL: Syntax*.

Documentation Conventions

This section describes the 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

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 and code examples, variable 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 that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of product- or platform-specific information.

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.

Icon	Description
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature
IDS	Identifies information that is specific to Informix Dynamic Server and Informix Dynamic Server, Workgroup Edition
SE	Identifies information or syntax that is specific to INFORMIX-SE

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.

Documentation Included with 4GL

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

- *INFORMIX-4GL Installation Guide* describes how to install the various 4GL products.
- *INFORMIX-4GL Reference* is 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.
- *INFORMIX-4GL by Example* is a collection of 30 annotated 4GL programs. Each program 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.
- *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 to interact with your 4GL programs while they are running. It helps you learn more about the 4GL language and determine the source of errors within your programs.
- 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.

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 particular 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)

Chapter

1

Introducing INFORMIX-4GL

In This Chapter

This chapter contains a high-level overview of INFORMIX-4GL. Its aim is to orient you to the capabilities and typical uses of the product and to answer general questions such as what kind of software 4GL is and what is it used for.

What Is 4GL?

4GL is a full-featured, general-purpose, fourth-generation programming language with special facilities for producing the following features:

- Database query and database management using SQL
- Reports from a relational database or from other data sources
- Form- and menu-based multiuser applications

These special features make 4GL especially well-suited to developing large database applications.

4GL Provides a Programmer's Environment

4GL provides a *Programmer's Environment* that makes it easy to create, compile, and maintain large, multi-module programs. Within the Programmer's Environment, you can accomplish the following tasks:

- Create new program modules and modify existing modules
- Compile individual modules and entire applications
- Create and compile forms that the application uses
- Run compiled applications

- Use INFORMIX-SQL to interact with an Informix database (if you have the INFORMIX-SQL product)
- Get help at any time with the on-line help feature

You can also manage your applications by using commands at the operating-system prompt rather than by using the Programmer's Environment.

4GL Works with Databases

Although it is a complete, general-purpose programming language, 4GL is specifically designed to make certain kinds of programs especially easy to write. Programs of these kinds, collectively *interactive database applications*, face some or all of the following special challenges:

- They retrieve data from a relational database and process the data with logic that is more complicated than SQL alone permits.
- They present data using screen forms and allow users to construct queries against the database.
- They allow users to alter database records, often enforcing complex requirements for business rules, referential integrity, data validation, and security.
- They update a database with data processed from other databases or from operating system files.
- They generate multipage, multilevel reports based on data from a database or other sources, often letting the user set the parameters of the report and select the data for it.

With 4GLyou can program applications of these kinds more easily than with any other language.

In addition, 4GL has an open, readable syntax that encourages good individual or group programming style. Programs written in 4GL are easily enhanced and extended. This, with its development environment, makes it easy for programmers to become productive quickly, no matter what programming languages they know.

4GL Runs in Different Environments

4GL is the only multipurpose programming language that offers code- and display-compatibility across operating environments. Applications that you develop are portable to the different platforms subject to simple porting guidelines. You can run this version of 4GL on the following types of computers:

- On UNIX character-based terminals provided by a wide variety of hardware vendors
- On UNIX workstations

Two Implementations of 4GL

Informix provides two implementations of 4GL:

- The C Compiler, which 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 Rapid Development System, 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 4GL application. This version is sometimes abbreviated as RDS.

For details about the differences between the two versions, see *INFORMIX-4GL Reference*.

Chapter

2

Interfaces of INFORMIX-4GL

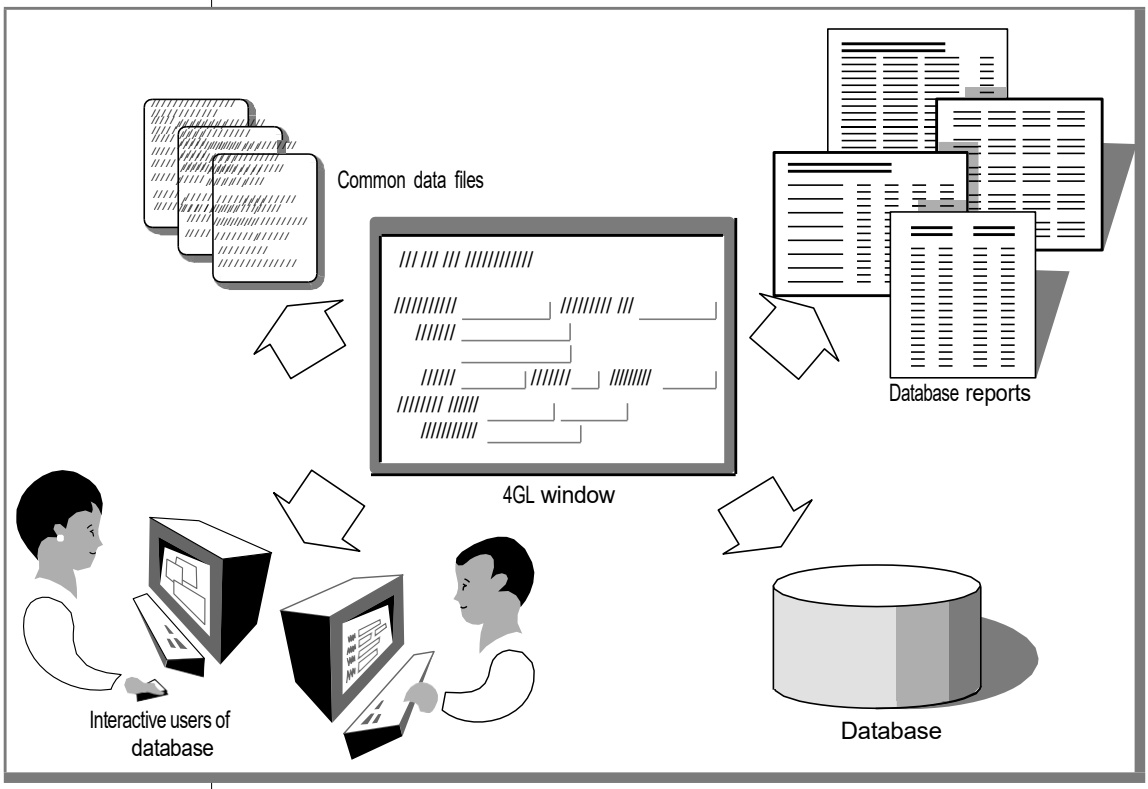
In This Chapter

Multiuser application programs that you write using INFORMIX-4GL have the following four primary interfaces to other software:

- Accessing a database through an Informix database server
- Communicating with end users through a terminal
- Accessing sequential files through the host operating system
- Generating reports that can then be sent to several destinations

This chapter describes these interfaces and their uses. Figure 2-1 is an overview of how they work.

Figure 2-1
Multiuser Application Programs



Database Access

Your 4GL programs can access relational SQL databases that appropriate Informix database servers support.

The database can be located on the same computer as the 4GL application program, but more often, the application runs on a separate *client system*.

After network connections are properly established, database access is transparent; you need not write any special code to make it happen. In fact, the same 4GL program can work with a local database server that is located on the same computer on one occasion, and over a network to a database server on another computer on another occasion.

Access to Sequential Files

Your 4GL application can use standard sequential, or flat, data files in the following ways:

- The UNLOAD statement writes selected rows of data from a database table to a specified file.
- The LOAD statement reads a file and inserts its lines as rows into a specified database table.
- The START REPORT or the REPORT statement can send output from a report to a specified sequential file or to an operating-system pipe.
- The PRINT FILE statement can incorporate the contents of a specified sequential file into the output of a 4GL report.
- The DISPLAY statement can be used to write to the screen. Using the host operating system, you can also direct these lines to a file or to another program.

Report Output

Your 4GL program can generate powerful and flexible reports, whose output is a series of print lines. Output can be directed to any of the following several destinations:

- A screen
- A printer
- A host system sequential file, specified by its pathname
- A UNIX operating-system pipe to another process

The logic for generating reports is the same in all cases. The destination can be coded into the program or selected at execution time.

Report programming is covered in more detail in Part II, with examples in Part III of this book.

User Access

To provide for portability of code across different platforms, your program interacts with the user through display areas, called *windows*, that have a fixed number of character-height rows and character-width columns.
The user of your application can be using any of the following environments:

- A terminal-emulation program on a personal computer or workstation that is connected to a UNIX network
- A character-based terminal connected to a UNIX system

No matter what the physical display device, the 4GL user interface is programmed for fixed-size characters. A 4GL window can display a fixed number of characters horizontally, and a fixed number of characters vertically. You specify these two dimensions when you declare the 4GL window, typically with dimensions that are sufficient to display a specific 4GL screen form. Each 4GL window can display no more than one 4GL screen form.

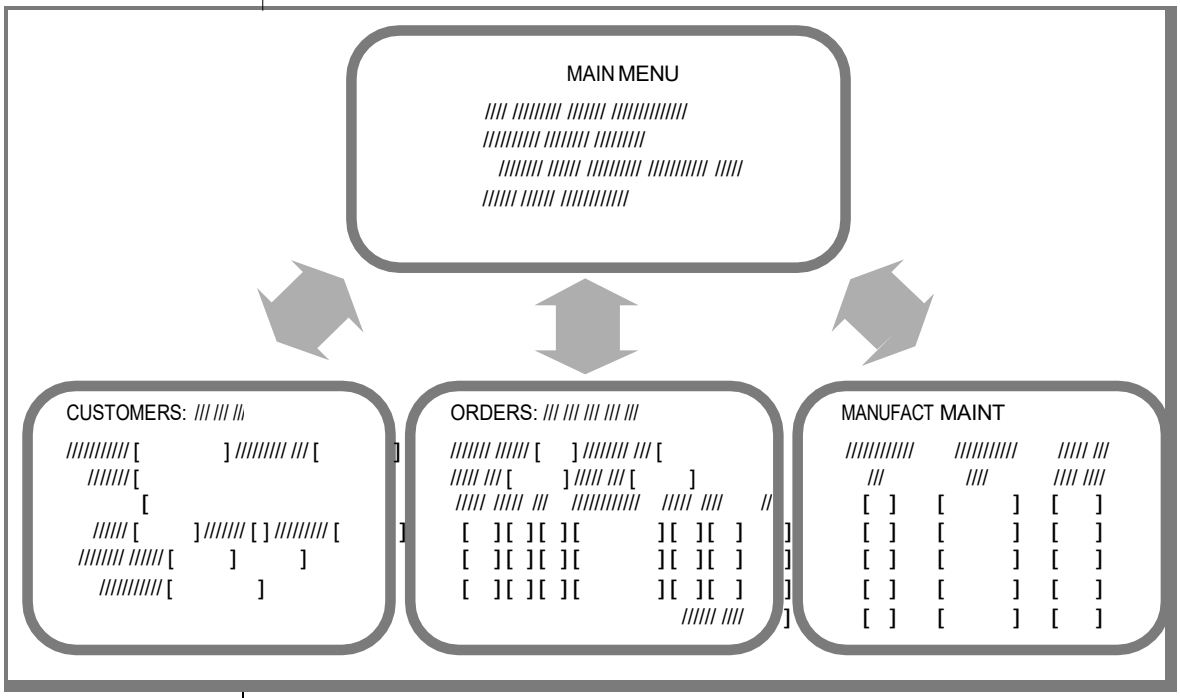
Screen forms of 4GL are also portable across 4GL applications and platforms. You design screen forms through which your program can display data. Then you compile the forms separately from the other parts of the program. The resulting compiled forms can be used with different 4GL programs.

You display a form with one program statement and fill it with data in another; you can prompt for user input to any or all fields with a third. You can easily open additional display areas for alerts and other information. Interaction through a hierarchical set of ring menus (horizontal menus) is also fully supported.

Using Forms and Menus

The typical 4GL program is designed around a hierarchy of screen forms and menus, as shown in Figure 2-2. The program presents a main menu that contains a list of major activities; for example, query, insert, or update. The user selects one activity and the program displays the form for that activity. When the activity ends, the program re-displays the main menu.

Figure 2-2
Program with Forms



The 4GL program specifies which screen elements the user interacts with at any given time. This predictability leads to an easily maintained linear program structure and simple program logic.

Summary

You use 4GL to write programs that connect key elements in the following informational structure:

- The database
- The interactive users of the database
- Common data files
- Output from database reports

While working with the database, you use the industry-standard SQL, which you can augment with your program logic.

As you will see in subsequent chapters, your programs have a simple method to access common sequential files, and nonprocedural, task-oriented methods for getting information from a database, and for defining and producing reports.

You can use 4GL to program elaborate user interactions based on forms and menus. This user interface is character-oriented; that is, output is displayed in a fixed number of evenly spaced lines, each of which contains a fixed number of monospace characters. This approach allows applications written in 4GL to run without modification on supported platforms.

Chapter

3

The INFORMIX-4GL Language

In This Chapter

This chapter provides a brief tour of INFORMIX-4GL as a language. The purpose is to give you an idea of what 4GL code looks like. Many short examples are given in the following chapters and several complete examples are supplied on-line with the product. For additional details on the 4GL programming language, see Chapter 8, "Using the Language."

As a programming language, 4GL has several important features:

- It is a procedural language, with facilities for structured programming.
- It is a nonprocedural, fourth-generation language with regard to:
  - Database access
  - Reports
  - Form-based user interaction
- It is similar to C in design, but much easier to read, write, and support.

A Structured, Procedural Language

4GL is a general-purpose programming language for creating structured programs, the way you might use Pascal, C, COBOL, or PL/1. Like these languages, 4GL offers statements that you can use to perform the following tasks:

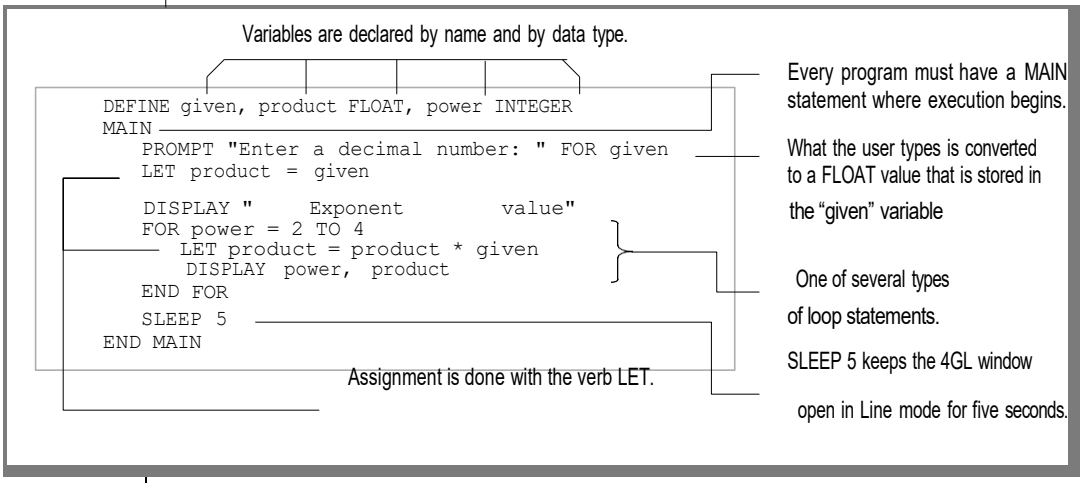
- Declare variables of different data types
- Calculate values and assign them to variables
- Declare and define functions

- Apply functions to data
- Display the contents of variables on the screen

Also like other programming languages, 4GL has control statements that you can use to define choices, loops, and statement blocks of code.

Figure 3-1 shows a short, complete program in 4GL. It prompts the user for a number and displays the square, cube, and fourth power of the number.

Figure 3-1
Short Program Example



Statements of 4GL that can contain other statements are called *compound statements*. The END FOR and END MAIN keywords respectively mark the end the FOR and MAIN statements, which are compound statements.

Like C, the 4GL language is free-form, with whitespace characters such as blank spaces, tabs, and linefeeds ignored in most contexts. You must, however, insert at least one blank space between consecutive keywords, identifiers, or literal values that appear in the same line, unless some other separator is provided. You cannot include whitespace characters within a keyword or identifier, and whitespace characters within literals are interpreted by 4GL as part of the literal.

4GL is *not* sensitive to the letter case of keywords or identifiers in a source statement. The use of only capital letters for language keywords such as DEFINE and MAIN is merely a convention used in these manuals, to help you to identify keywords. You can write keywords in lowercase, or in any combination of capitals and lowercase that you prefer. However, 4GL preserves the letter case of characters that are enclosed between quotation marks, such as in the quoted string of the PROMPT statement in the previous code example.

The 4GL language supports structured programming. Its design encourages you to build your program as a family of simple, reusable functions with well-defined interfaces. The function in Figure 3-2 returns the lesser of two integers.

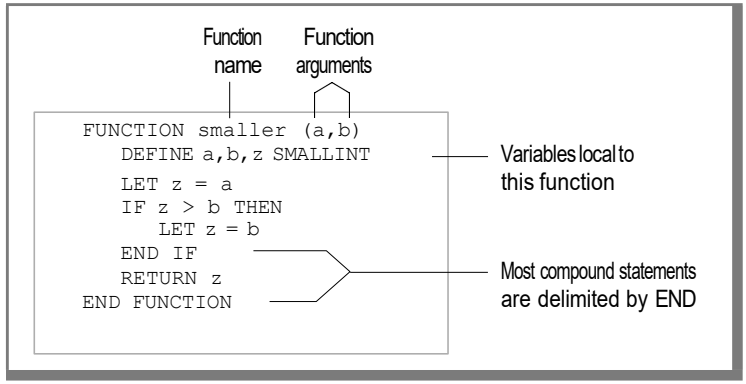


Figure 3-2
Simple 4GL Function

In these manuals, a *module* means an individual file. You cannot divide a single 4GL statement between two modules. This rule also applies to compound 4GL statements. As described in the next chapter, 4GL also has features that make it easy to assemble large programs from many small source code modules.

A Nonprocedural, Fourth-Generation Language

4GL is a nonprocedural or fourth-generation language in three important areas:

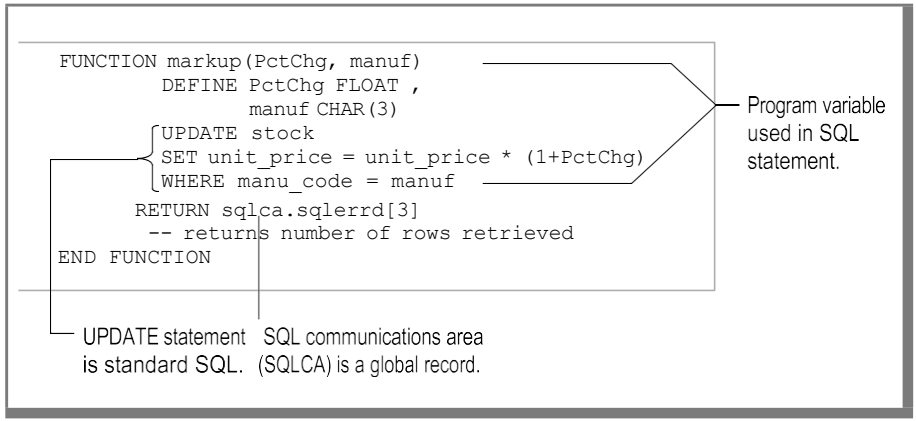
- Database access
- Report generation
- User interaction

In these areas, you specify what is to be done, while 4GL (or the database server) takes care of the actual sequence of operations.

Database Access

4GL includes all SQL statements supported by Informix as native statements of the language. The function in Figure 3-3 applies a change in price to all the items in the **stock** table that are purchased from a particular manufacturer. The function returns the number of rows changed to the calling routine.

Figure 3-3
Database Access Function



It is the nature of many SQL statements to be *nonprocedural*. That is, you use the statements to specify what is to be done, and the database server then determines how to do it.

Using 4GL, however, you can write code that applies sophisticated logic to the results of the SQL statements. For example, the function in the preceding example could contain statements to validate the argument before applying the change, or to verify the authorization of the current user to perform such a change.

Report Generation

A 4GL *report* is a specialized program block that can display data from the database. Output from reports is typically tabular in form and can be designed for printing with page headers and footers. Some reports are produced by noninteractive, batch programs, perhaps run at night. Any 4GL program can produce report output if it contains the necessary logic for extracting and formatting the data, invoking the report, and sending the output to some destination (such as to a file, to a printer, or to the screen).

4GL divides the task of producing report output into two parts. One part that might require procedural logic is the production of the rows of data that go into the report. The second part is the logic within the report itself, reflecting your decisions as to the sequence of presentation, and how to format header and footer lines, detail lines, control breaks, and display subtotals.

You write the logic of the report in a nonprocedural form, as a collection of code blocks that are called automatically as needed. For example, your code block for a group subtotal can be executed automatically on each change in the value of the group control variable. Your code block for a page footer is called automatically at the bottom of each page.

As you design and code the logic of a report, you think about each part of the report in isolation. 4GL supplies the logical glue that passes control of program execution to the appropriate report sections as required.

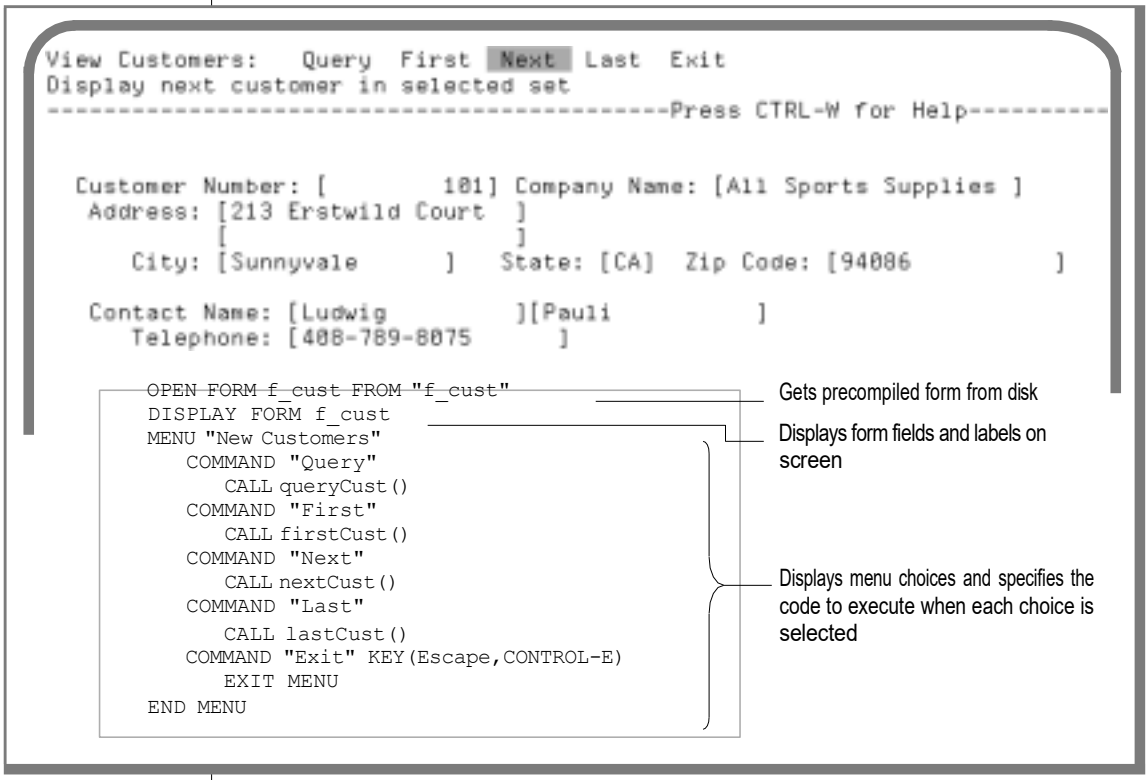
Examples of reports are shown in subsequent chapters, particularly in Chapter 10, "Creating Reports." One key point: the *report driver*, which is the part of the 4GL program that produces the *input records* (the rows of data that the report processed) does not need to arrange the order of the extracted rows. 4GL can automatically collect input records and sort them before presenting them to the report code, if that is necessary.

User Interaction

4GL contains extensive support for writing interactive programs. You can fill some or all of the fields of a form from program variables with a single statement. With another statement, you can open up form fields for user input with the entered data returned to program variables. For detailed validation, you can attach blocks of code to form fields. The code is executed when the cursor leaves or enters a field.

Figure 3-4 shows a typical screen form with a menu, together with the *p-code* that would be used to display it. (Example 19 in *INFORMIX-4GL by Example* contains the complete source code of this program.)

Figure 3-4
Screen Form with Menu



You describe a screen form in its own source file and compile it separately from program code. Because forms are independent of 4GL programs, forms are easy to use with many different 4GL applications.

You can open a 4GL *window*, optionally containing another form, above the current 4GL window, and then restore the original display. There is support for scrolling lists of data during display and editing. These features are covered in subsequent chapters.

Summary

4GL has all the features of a standard, structured programming language. It goes beyond such languages as Pascal or C in that it supports nonprocedural, task-oriented ways of programming database access, report generation, and user interaction.

Chapter

4

Parts of an Application

In This Chapter

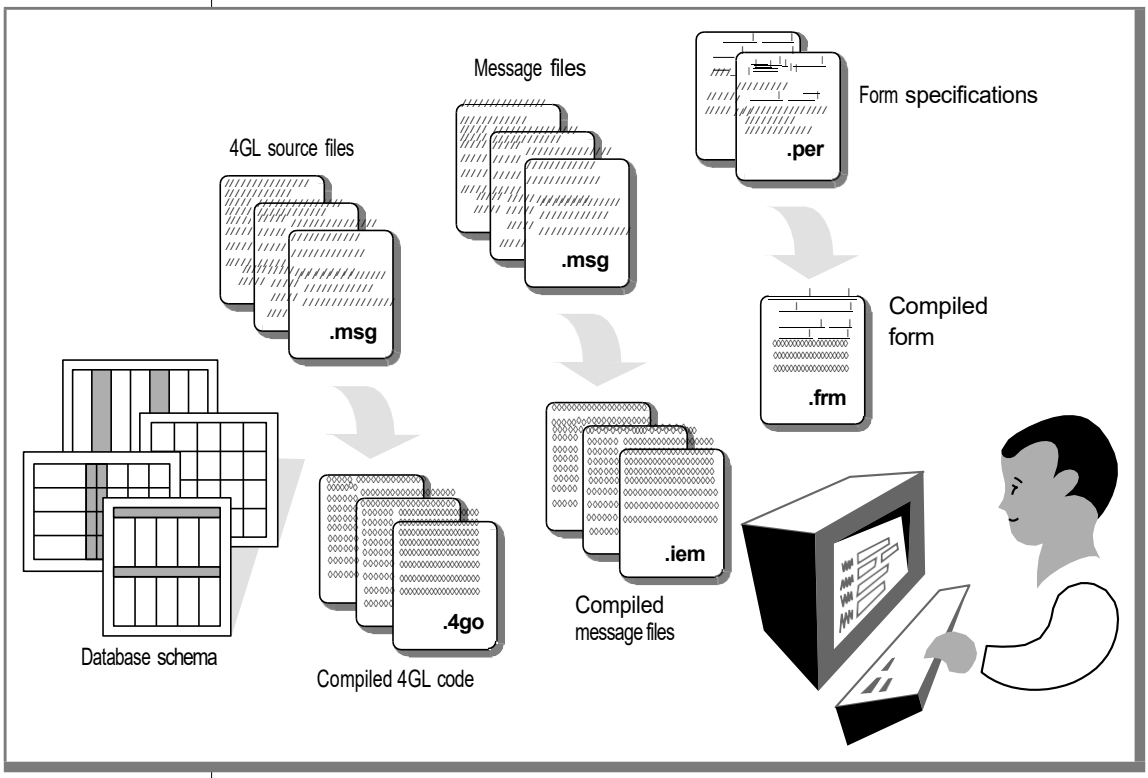
You typically use INFORMIX-4GL to build an *interactive database application*,a program that mediates interaction between a user and a database. The database schema that organizes data into relational tables gives shape to one side of the program. The needs of your user shape the other side. You write the program logic that bridges the gap between them.

Such a program has many parts that you prepare with the help of the 4GL Programmer's Environment. This chapter discusses the main parts of an application, which might include these files:

- **Form source files.** You specify the user interface to your application, using editable files that specify the visual appearance of the form and the data types and attributes of the fields in the form.
- **Form object files.** Your form specifications are compiled into a binary form by using FORM4GL, the 4GL form compiler. These are loaded by the 4GL program during execution.
- **Message source files.** You write the text of help and other messages separately from your 4GL programs. Programs can share a common set of messages, and you can change these (for instance, to support a different language) independently of the programs.
- **Message object files.** Your messages are indexed for rapid access by **mkmessage**, the 4GL message compiler. Like a form, a compiled message file can be used by many different programs.
- **Program source files.** You write your program as one or more files of 4GL source code.
- **Program object files.** Your sources are compiled into C code or p-code executable files, depending on the implementation of 4GL that you are using. For a brief description of the two implementations, see "Two Implementations of 4GL" on page 1-5.

Figure 4-1 illustrates the parts of an application.

Figure 4-1
Parts of a 4GL Application



The Database Schema

Either you or another person acting as the database administrator (DBA) must carefully design a representation of the real world, or of some important part of it, in the structure of the database.

This picture of the real world is expressed in the form of *tables* of information, each containing categories of information, called *columns*. It is not simple to make the proper choice of columns and to group them into tables so that the data can be used efficiently and reliably as your needs change. In fact, many books have been written on the subject of how best to design a database structure, usually referred to as a *schema*.

The database schema affects your 4GL program in the following ways:

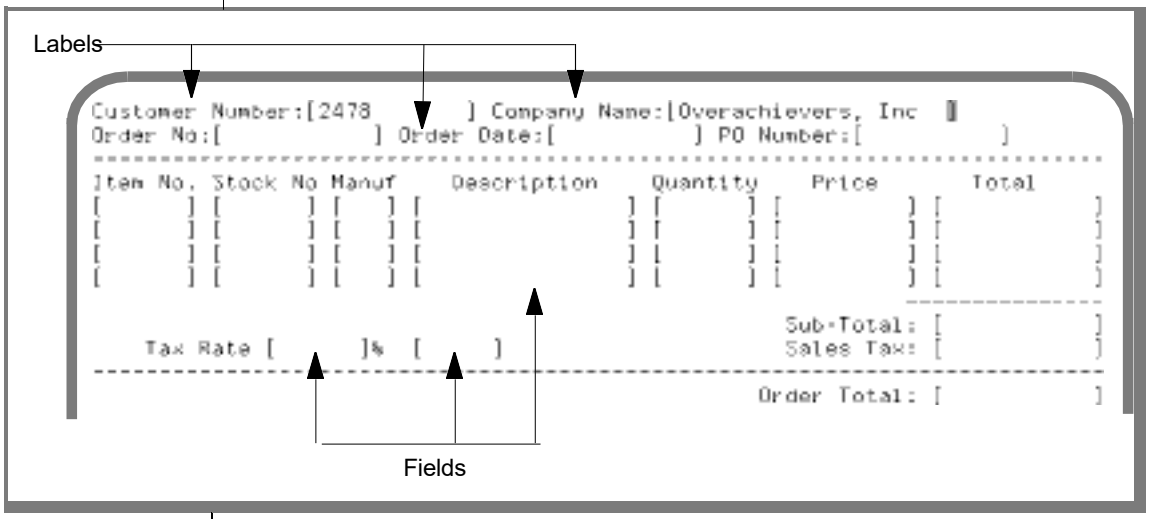
- You can have program variables that have the same data types and names as database columns. 4GL makes this easy by letting you declare a variable as being LIKE a column in a table. When the program is compiled, the compiler queries the database for the appropriate data type.
- Your program can contain SQL statements that refer to names of tables and columns. Any change in the schema might require you to examine these statements and possibly change them.
- The logic of your program can depend on the schema. For example, if your program must change the schema of several tables to perform a certain operation, then you might want to use explicit database transactions. If only a single table needs changing, you can avoid this.

In general, before you start work on a large application, you should make sure that the database schema is workable and that you and others who will be using the database understand it well.

Form Specifications and Form Files

In many applications, the user interface is defined by forms. A form is a fixed, functionally organized arrangement of *fields* (areas where you can display program data and the user can enter text) and *labels* (static, descriptive text,) as shown in Figure 4-2.

Figure 4-2
A Form That Contains Labels, Fields, Additional Text, and a Screen Array



The end user of a program does not know about the database schema or your carefully designed program logic. As the user sees it, the forms and the menus that invoke them *are* the application. The arrangement of fields and labels, and the behavior of the forms as the user presses different keys and selects different menu options, create the personality of the program.

4GL form specification files are ASCII files. You can use an ASCII text editor to design the labels and fields for your form. The following example shows a portion of the form specification file that was used to create the preceding form:

```
SCREEN
{

Customer Number:[f000        ] Company Name:[f001                  ]
Order No:[f002        ] Order Date:[f003       ] PO Number:[f004       ]
------------------------------------------------------------------------
Item No. Stock No Manuf     Description     Quantity     Price        Total
 [f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
 [f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
                                                    ----------------------
                                                Sub-Total: [f012 ]
      Tax Rate [f013  ]%  [f014 ]                Sales Tax: [f015 ]
------------------------------------------------------------------------
                                                Order Total: [f016 ]

}
TABLES
customer orders items stock state
ATTRIBUTES
f000 = orders.customer_num;
f001 = customer.company;
...
```

To view the entire text of this example, see Example 11 in *INFORMIX-4GL by Example*.

After you specify a form, you compile it with FORM4GL, the 4GL form compiler, as illustrated in Figure 4-3. The result is a portable binary file that can be opened and displayed from any 4GL program on any platform that 4GL supports.

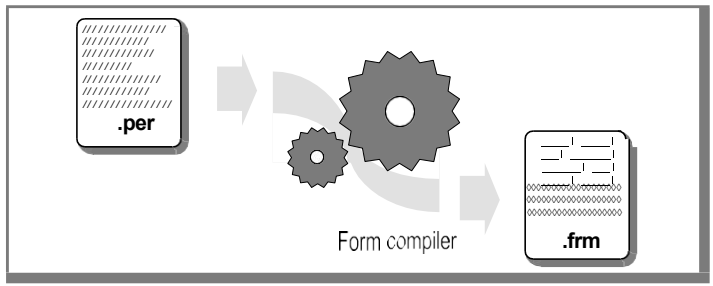


Figure 4-3
The Form Compilation Process

Compiled forms are independent of the programs that use them, so you can use the same forms in different applications for a consistent look and feel.

Because forms are so important to the users of your application, you should consider designing the main forms before any other part of the program. You can quickly prototype programs that display forms so that your users can give you their opinions.

The following text is from the program that displayed the form Figure 4-2 on page 4-6:

```
MAIN
    OPEN FORM fx FROM "f_orders"
    DISPLAY FORM fx
    DISPLAY "2478" TO orders.customer_num
    DISPLAY "Overachievers, Inc" TO customer.company
    SLEEP 60
END MAIN
```

Form Design

Designing a good user interface is integral to the success of your program. The forms you create to develop the user interface provide an interaction gateway between the end user and the database. Consider the following points when you create forms for your application:

- Is the purpose of the form clear from its title and the title of the menu command that invokes it?
- Are fields in the form arranged in the same logical order that the user typically follows in transcribing or describing the information?
- Is the same information always given the same label in every form in which it appears?
- Are form labels consistent in style and content?
- Is the relationship between various forms as clear as possible?
- Is it obvious how to complete the form and what fields are required?

Field Entry Order

With 4GL, you can constrain the user to entering fields in a preset order, or you can permit entry into fields in any order desired by the user. Because application and form requirements differ, you can control these factors on a form-by-form basis.

Program Source Files

You express the logic of your application with 4GL statements in program source files.

Organization of a Program

If you use the C Compiler version of 4GL, the files that contain executable 4GL statements require **.4gl** as the file extension; otherwise, the program compiler cannot find them. If you use the Rapid Development System (RDS) implementation of 4GL, however, you can omit the **.4gl** file extension.

Because 4GL is a structured programming language, executable statements can appear only within logical sections of the source code called *program blocks*. This can be the MAIN statement, or else a REPORT or FUNCTION statement. A *function* is a unit of executable code that can be called by name. In a small program, you can write all the functions used in the program in a single file. As programs grow larger, you will usually want to group related functions into separate files, or modules, with the declarations that they use.

Each source file usually reflects a self-contained unit of program logic. Source files are sometimes called source modules.

Execution of any program begins with a special, required program block named MAIN. The source module that contains MAIN is called the main module. The following example is a small but complete 4GL program:

```
MAIN
   CALL sayIt()
END MAIN

FUNCTION sayIt()
   DISPLAY "Hello, world!"
END FUNCTION
```

This single module contains the MAIN program block, delimited by the keywords MAIN and END MAIN, and one other function named **sayIt**().

A single function cannot be split across source modules. The preceding program example has two functions, however, so it could be split into two source modules. The first would be the MAIN program block, as follows:

```
MAIN
   CALL sayIt()
END MAIN
```

The second module could contain the three lines of function **sayIt**() just as shown above. It could also contain data or other functions related to **sayIt**(), if there were any.

Functions and reports are available globally. For example, you can reference the **sayIt()** function in any source module of your program, provided that the function is defined somewhere in the program.

The Globals Files

In 4GL programs, global variables (variables that are available to more than one source module) must be declared in *globals* files and imported through the GLOBALS '*filename*' statement by each 4GL module that uses them. For more information on local and global variables, see "Variables and Data Structures" on page 8-11.

Program Object Files

Both the C Compiler and the RDS implementations of 4GL provide their own source code compiler command. These can generate distinct, executable forms of the same 4GL program:

- For the C Compiler implementation, **c4gl**, the command to invoke a *C compiler*, generates code that can be executed directly by the hardware of the computer after an executable program is created.
- For the RDS implementation, **fglpc**, the *p-code compiler*, generates hardware-independent pseudo-machine code, which is not directly executable by the operating system or GUI, but that can be interpreted by the 4GL *p-code runner*.

Both implementations can take the same 4GL source code as input. If the C code option is selected, the output is a C language object file. When this file is linked with other 4GL libraries (and optionally with C object modules), an independent executable program is produced. This process is illustrated in Figure 4-4. Because C code is not machine-independent, a compiled C version of a 4GL program must be recompiled before it can be executed on another computer system that uses a different C compiler.

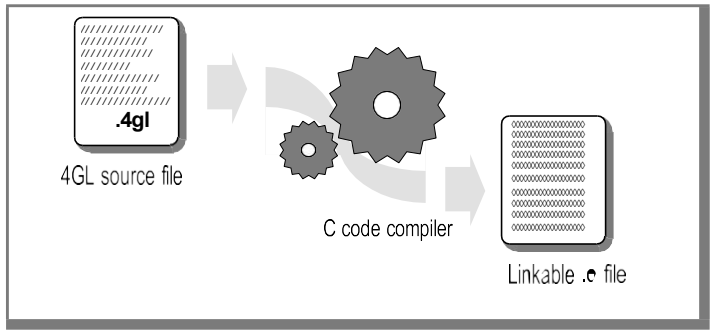


Figure 4-4
The C Object Code Generation Process

If compilation to p-code is chosen, p-code intermediate object files are created that are executable under the 4GL runner.

Both types of compiled files are binary. That is, the file is not printable or editable as text. All the modules of an application must be compiled to the same form. That is, the executable version of your program cannot mix C code and p-code units, although the p-code runner can be customized to call C object modules.

For details of the steps that are required for compiling 4GL source files of all types, as well as for using C with 4GL, see *INFORMIX-4GL Reference*.

P-Code Object Files

As illustrated in Figure 4-5, you can use **fglpc**, the command to invoke the p-code compiler, to translate a 4GL source module into p-code. The output of the p-code compiler will have the file extension **.4go**.

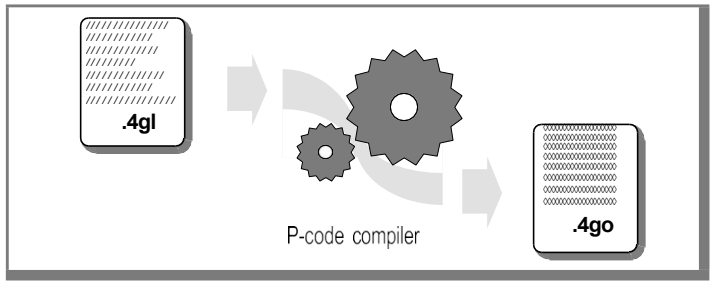
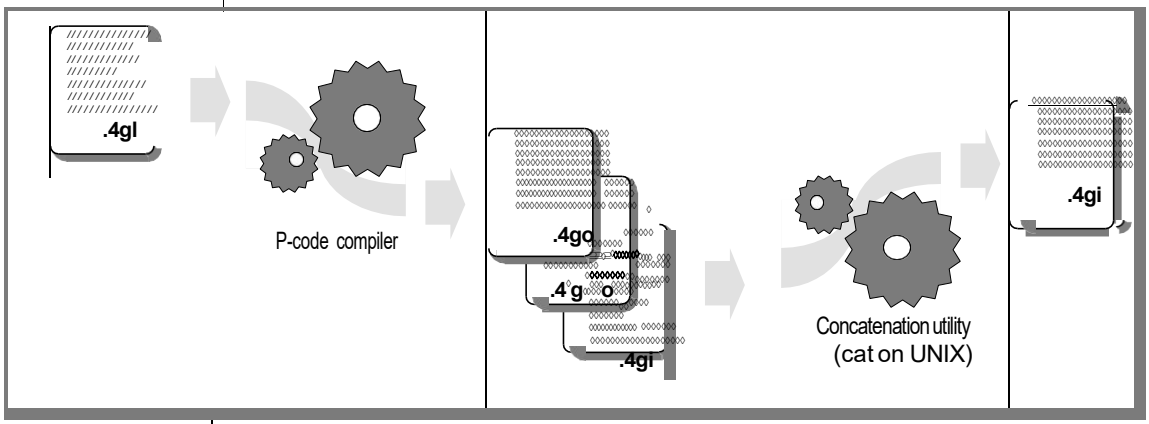


Figure 4-5
The P-Code Object Code Generation Process

When your application is in several source modules, you first compile each separate module to p-code, as shown in Figure 4-6. Then you can concatenate the individual p-code files using a utility (**cat** in UNIX environments) to make the executable **.4gi** program file.

Tip: *When you use the Programmer's Environment to build and maintain your program, module linking is done automatically for you.*

Figure 4-6
Steps to Creating an Executable Program Under the P-Code Runner



To execute a **.4gi** file, you call the 4GL *p-code runner*. Use **fglgo** to execute the p-code instructions in the file as shown in Figure 4-7, activating your program.

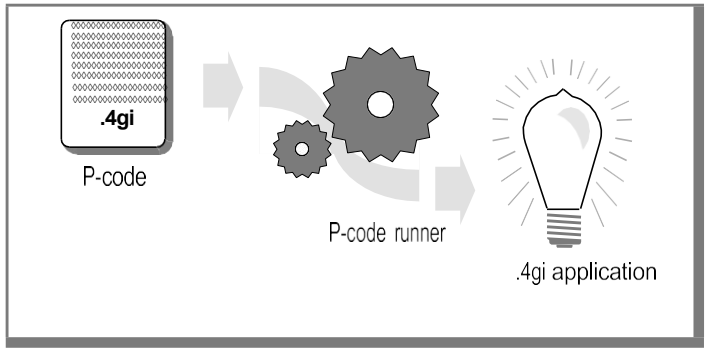


Figure 4-7
Components of a Runnable P-Code Application

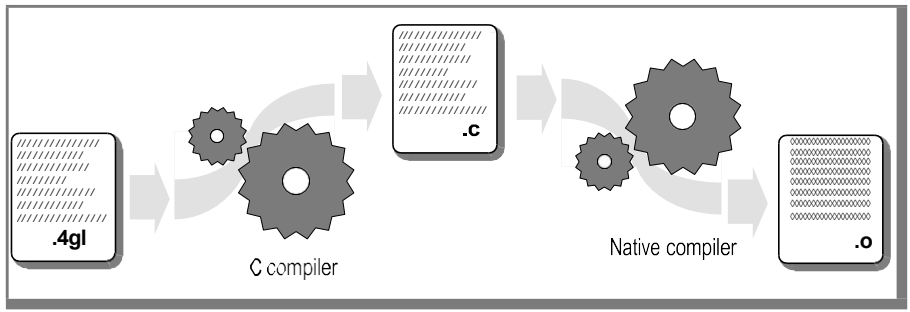
C-Code Object Files

You can use the C-code compiler, **c4gl**, to translate a source module directly into machine code, which is done in the following three primary stages:

1. The module is translated to INFORMIX-ESQL/C source code.
2. The ESQL/C processor converts that to C source code.
3. The compiler translates to C object code for your computer.

Figure 4-8 illustrates this procedure.

Figure 4-8
Steps to Creating a C Object File



From the operating-system command line, a single call to the **c4gl** command performs all the steps, or all these steps can be automatically accomplished through the Progammer's Environment.

On UNIX systems, the default extension is .**4ge**. However, it is not required. You can name your executable applications anything that you like, within the rules for valid filenames on your operating system and network.

The C file from a single source module is compiled to an **.o** file. Several **.o** files for a multi-module application can be combined into a single executable file through the Programmer's Environment or using another concatenation utility. In fact, the **c4gl** command line accepts any combination of **.4gl** files, **.ec** files, **.c** files, and **.o** files to produce a single .**4ge** executable file, as illustrated in Figure 4-9.

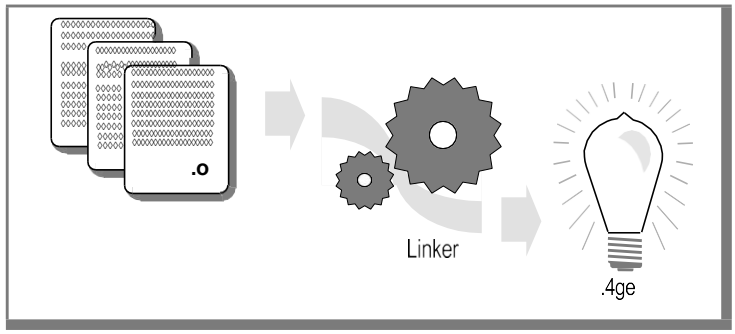


Figure 4-9
Steps to Creating an Independently Executable 4GL Program

Example Programs

Now that you know the parts of a 4GL program, you should look at a few of them to see what they are like. A number of programming examples are distributed with 4GL. *INFORMIX-4GL by Example* contains 30 complete and annotated 4GL programs.

Chapter

5

The Procedural Language

In This Chapter

INFORMIX-4GL is a fourth-generation programming language. However, it uses some concepts based on procedural languages (such as Pascal and C). This chapter describes concepts that are based on procedural programming. In particular, it describes how to accomplish the following tasks:

- Declare variables
- Organize statements for decisions and looping
- Handle exceptions

Declaration of Variables

4GL has a flexible mechanism for declaring program variables. You specify a data type for every variable, but there is automatic conversion between many data types. Data can be structured as records or arrays and can be allocated statically or dynamically. Variables can have either local or module scope.

Data Typing

4GL is a *strongly typed* language. That is, whenever you declare a program variable, you must specify its data type; for example, INTEGER or CHAR. The compiler ensures that only data of that type can be stored in that variable. The following example is a declaration of several types of variables:

```
DEFINE aFloat FLOAT
       oneInt, anotherInt INTEGER,
       aString CHAR(20)
```

Data types that 4GL supports for program variables include all the primitive data types (except SERIAL) that are valid in columns of an Informix database, as well as BYTE and TEXT, the binary large object (blob) data types.

Every data type of 4GL can be classified into one of three logical categories:

- *Simple* data types store a single value in the format of some primitive SQL data type. Every data type of 4GL except ARRAY, BYTE, TEXT, and RECORD is a simple data type.
- *Large* data types (BYTE and TEXT) can store blobs.
- *Structured* data types (ARRAY and RECORD) can store ordered sets of values.

An important point to note is that 4GL defines a specific `NULL` value for every data type. `NULL` means *unknown*, rather than `0`, which is a precise value. You can assign `NULL` to any variable, and you can test any variable for `NULL` content. This is necessary to support database operations, because `NULL` is a distinct value (or, to be more precise, a *non-value*) within a database table.

Automatic Data Type Conversion

With some strongly typed languages, it is an error to assign a value of one data type to a variable of a different data type. In contrast, 4GL allows you to assign any value to any variable, provided that there is a reasonable way of converting the value to the data type of the receiving variable.

```
LET aFloat = 2.71828
LET oneInt = aFloat-- assigns integer 2
```

The LET statement is used for assignment. The first of the preceding LET statements assigns a literal decimal number to a variable of the FLOAT data type. In this code fragment, the variables **aFloat** and **oneInt** are declared in the DEFINE statement example in "Data Typing" on page 5-3.

In the second LET statement, a FLOAT value (which is the contents of the variable **aFloat**) is assigned to the INTEGER variable **oneInt**. 4GL converts the FLOAT value to an INTEGER to match the data type of the receiving variable. To do so, the fractional part of the floating point number is truncated.

Another common data type conversion is that between a character value and almost any other data type.

```
LET aString = aFloat-- assigns "2.71828" to aString
```

This statement assigns a FLOAT value to a variable whose data type is CHAR (a character string). 4GL converts the numeric value to characters and then assigns that string value to the CHAR variable **aString**.

4GL will also attempt to convert a character string to a number or to some other data type. The second of the following LET statements assigns a string of numeric characters to a FLOAT variable:

```
LET aString = "3.141592"
LET aFloat = aString-- assigns 3.141592 into aFloat
```

If the characters in the string can be interpreted as a literal value of the receiving data type, then the conversion is done. Most data types have a printable character representation, and 4GL converts automatically between the printable form and the internal form.

Of course, there are some cases when conversion is not allowed. For example, the BYTE and TEXT data types cannot be converted into any other type. Such errors are typically detected at compile time.

Some conversions, however, can only be found to be impossible at execution time. The following example fails in its attempt to convert a large floating-point number to an INTEGER data type.

```
LET aFloat = 2E12-- about 100 times the maximum integer size
LET oneInt = aFloat-- this causes a runtime error
```

You can manage such runtime errors in any of several ways:

- Anticipating them and inserting programmed tests to avoid them
- Trapping the error at execution time
- Letting the runtime error terminate the program with an appropriate message

For a table that identifies all the pairs of data types for which 4GL supports automatic data type conversion, see *INFORMIX-4GL Reference*.

Data Structures

The structured data types enable you to organize program data into *records* and *arrays*, both of which are sometimes called *data structures*.

These data types (ARRAY and RECORD) can store ordered sets of values of other 4GL data types. You can also declare an ARRAY variable whose elements are RECORD variables, or a RECORD variable that has ARRAY or RECORD members, but you cannot declare an ARRAY of ARRAY variables.

Records

The RECORD data type stores a group of values that are treated as a unit. Each *member variable* of a record has a name. Unlike blob or simple data types, you can use a RECORD variable to represent an entire row in a database table. The following code fragment declares a record:

```
DEFINE stockRow, saveStockRow RECORD
         stock_num    INTEGER ,
         manu_code    CHAR(3) ,
         description CHAR(15) ,
         unit_price MONEY(8,2) ,
         unit         CHAR(4) ,
         unit_descr CHAR(15)
       END RECORD
```

This statement defines two program variables. Their names are **stockRow** and **saveStockRow**. Each variable is a record with six *members*. The member named **manu_code** is a three-character string. You refer to this member of the **stockRow** record as **stockRow.manu_code**. The parallel member of the other record, **saveStockRow**, would be called **saveStockRow.manu_code**. Note the *record*.*member* notation, where a period (**.**) separates the record qualifier.

The members of these records are all simple data types. A record can contain members, however, that are other records or arrays. Another interesting aspect of this record is that it contains one member for each column in the **stock** table of the **stores7** demonstration database. Because it is so common to define a record that matches one-for-one to the columns of a database table, 4GL provides an easier way of doing this:

```
DEFINE stockRow, saveStockRow RECORD LIKE stock.*
```

This statement causes the 4GL compiler to refer to the database, extract the names and types of all the columns, and insert them in the program. In this way, you can ensure that the program will always match the database schema. (A previous DATABASE statement must identify the database.)

You can also fetch a row of a database table into such a record:

```
SELECT * INTO stockRow.* FROM stock
   WHERE stock_num = 309 and manu_name = "HRO"
```

You can assign the contents of all the members of one record to another record with a single LET statement:

```
LET saveStockRow.* = stockRow.*
```

You can do this even when the two records are not defined identically. The assignment is done member-by-member. As long as the records have the same number of members, and data values from each member on the right can be converted to the data type needed by the corresponding member on the left, you can assign the contents of one record to another.

Arrays

Like RECORD, the ARRAY data type of 4GL does not correspond to any single primitive data type of SQL. The ARRAY data type stores an ordered set of values that are all the same data type, for every 4GL data type except ARRAY. You can declare one-, two-, or three-dimensional arrays. The elements of the array can be simple data types, blobs, or records, as the following example shows:

```
DEFINE stockTable ARRAY[200] OF RECORD LIKE stock.*
```

This array variable is named **stockTable**. It contains 200 elements, each of which is a record with as many members as there are columns in the **stock** table in the database. One of those columns is named **stock_num**. You would access the **stock_num** member of the 52nd element of the array by writing **stockTable[52].stock_num**.

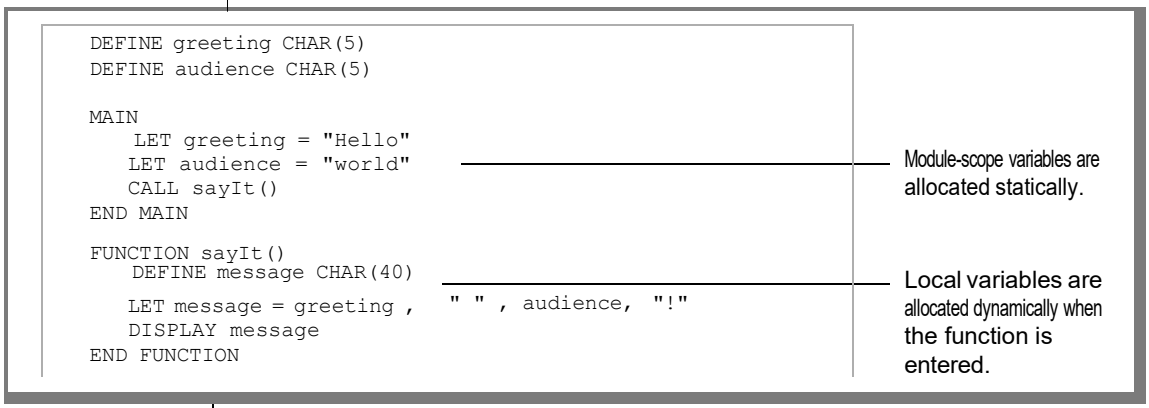
The first element of any array is indexed with subscript **1**. This differs from the C language, and some other programming languages, where the first element is always zero. The subscript value that selects an element can be given as an expression. Expressions are described in "Expressions and Values" on page 8-22.

Memory Allocation

4GL supports the allocation of memory to program variables either statically, as part of the executable program file, or dynamically, at execution time. You choose the method to use by the location of your DEFINE statement within the source module.

In the program in Figure 5-1 on page 5-8, **greeting** and **audience** are static variables, whose memory is allocated during compilation, because they are declared outside of any program block. In the same example, **message** is a variable local to the **sayIt()** function in which it is defined; memory for **message** is allocated dynamically, when its function is invoked at runtime.

Figure 5-1
Examples of Static and Dynamic Memory Allocation for Variables



The topics of data allocation, scope of reference, visibility, and program blocks are considered in detail in *INFORMIX-4GL Reference*. In summary:

- Variables that you declare outside of any MAIN, REPORT, or FUNCTION statement have as their scope of reference the same module in which they are declared. Unless their names conflict with the names of local variables, they are visible in the entire module. They can be referenced by any statement in the same source file that follows the definition. Memory for these *module variables* is allocated statically, at compile time, and become part of the *program image*.
- Variables that you declare within a function are *local* to the function New copies of these variables are created each time the function is called. They are discarded when the function exits.

Scope of Reference

The scope of reference of a variable is where its name can be used in the program. Variables that you declare can have local or module scope:

- Local variables declared within a program block are local to that program block. Their scope of reference is from the point of declaration to the end of the program block. Local variables are created each time that their FUNCTION, report, or MAIN statement is entered. They cease to exist when execution of that program block terminates.
- Module variables declared outside of any program block have a scope that extends from the point of declaration to the end of the module.

The GLOBALS ... END GLOBALS statement declares variables that are visible in any other module that includes the GLOBALS *filename* statement, where *filename* specifies the module that contains the GLOBALS ... END GLOBALS statement. (See "Global Scope: Within Several Modules" on page 8-16.)

Certain built-in features of the 4GL language are global in scope, including the named constants TRUE, FALSE, and NOTFOUND; the global variables **status**; **int_flag**, and **quit_flag**; and the members of the **SQLCA** record.

Decisions and Loops

4GL has statements for looping and decision making comparable to other computer languages. These statements are covered in "Decisions and Loops" on page 8-31 and in *INFORMIX-4GL Reference*. The following table gives a brief summary.

Statement Name	Description
IF…THEN…ELSE	Tests Boolean (TRUE/FALSE) conditions
CASE	Makes multiple-choice decisions
WHILE	Is used for general loops controlled by a Boolean condition
FOR	Is used for loops that iterate over an array

There is also FOREACH, a special loop used for database access, as described in "Row-by-Row SQL" on page 9-5. These control statements can be nested. A key point is that their syntax is simple and regular:

- Most compound 4GL statements (those that can contain other statements) are closed by specific END *statement* keywords. Thus, the IF statement is closed with END IF, CASE with END CASE, and so on.
- Every looping statement has a specific EXIT option for early termination. You leave the WHILE statement with EXIT WHILE, you leave FOR with EXIT FOR, and so on.
- No special punctuation is needed in 4GL code. You do not need to put semicolons between statements as in C or Pascal, although you can do so if you prefer. Nor are you required to use parentheses around a Boolean condition, as in C. But again, you can do so if you prefer.

4GL also supports GOTO and LABEL statements, by which control of program execution can *jump* from one statement to another within a program block.

Statement Blocks

Many 4GL statements such as LET and CALL are *atomic*; that is, they contain only themselves. Others are compound; that is, they can contain other statements. The most common compound statements include these, as the following table shows.

Statement Name	Description
IF…THEN…ELSE	THEN and ELSE each introduce a block of statements. The ELSE block is optional.
FOR	The body of a FOR loop is a block of statements.
WHILE	The body of a WHILE loop is a block of statements.
CASE	WHEN and OTHERWISE each introduce a block of statements.

Statement blocks can be nested. That is, one compound statement can contain another. Figure 5-2 is a code fragment that contains a nested IF statement block from Example 9 in *INFORMIX-4GL by Example*.

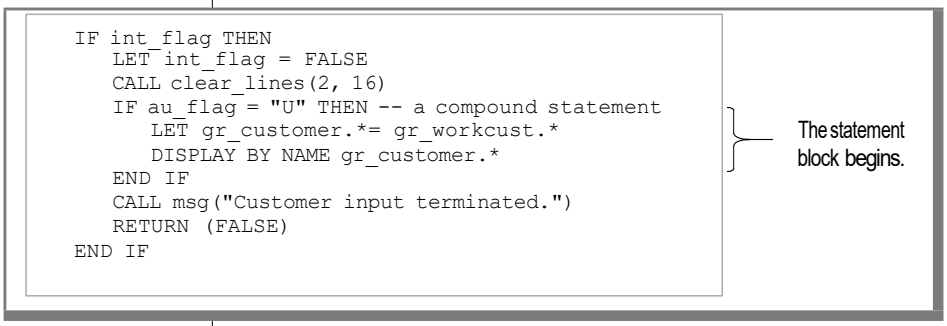


Figure 5-2 *Example of a Nested IF Statement Block*

Any statement block can be empty. That is, you do not need to supply any statements in contexts where 4GL syntax requires a statement block.

Comment Symbols

To comment your 4GL code, use double hyphens (--) or the sharp (#) sign for individual lines (as in the previous code example), or braces ({ and }) for one or more contiguous lines of code. The compiler does not process comments. Comment symbols that appear between quotation marks (single or double) are treated as literal symbols, rather than as comments. Comments cannot be nested.

Exceptions

An exception (sometimes called an *error condition* or a *runtime error*) is an unplanned event that interferes with normal execution of a program. An exception is not expected to occur in the normal course of program execution, and special action is often required when an exception does occur.

Kinds of Exceptions

There are several kinds of exceptions. Most can be classified within one of the following categories:

- **Runtime errors.** Errors in program statements detected by 4GL at runtime. These errors can be classified by the kind of statement in which they occur:
  - **SQL error.** Errors reported by the database server.
  - **File I/O.** Errors using files managed by the host operating system.
  - **Screen I/O.** Errors using the screen.
  - **Validation.** Errors using the VALIDATE statement.
  - **Expression.** Errors in evaluating 4GL expressions; for instance, a data conversion error or an invalid array subscript.
- **SQL end of data.** Warnings that you have reached the end of a set of rows being fetched through a database cursor.

- **SQL warning.** A warning condition reported by the database server, typically of lower severity than SQL errors.
- **External signals.** Events detected by the host operating system. An external signal is usually not directly related to 4GL program statements. Two common external signals that 4GL can handle are Interrupt (CONTROL-C) and Quit (CONTROL-\).

Why Exceptions Must Be Handled

Exceptions are *unplanned* in the sense that they are events of low probability. Usually your program cannot predict or control when they will occur. This does not mean that they are always *unexpected*. Consider the following examples:

- You are certain that there will be an end to any selection of rows. It is just that you do not always know which row will be the last.
- You are sure that some user will eventually try to cancel an operation, but you do not know when.

Even runtime exceptions must always be anticipated, no matter how carefully you write your code.

Because program exceptions are sure to happen, you must design the program to handle them in a rational manner. But because they are of low probability, you want to handle them:

- away from the main line of processing, so the code for the normal, expected sequence of events is clear and readable.
- with a minimum of overhead at execution time.

How Exceptions Are Handled

When a runtime error is encountered, any SQL transaction in progress is automatically rolled back. Then 4GL writes a message to the error log, which is by default the screen. You can also establish an error log file on disk.

Unless you establish some way of handling exceptions, 4GL attempts to continue program execution when any exception occurs. For some errors, however, this is not possible, and the program terminates abnormally.

For many exceptions, the default response is correct. But you can anticipate and handle other types of exceptions with 4GL statements, including the DEFER and WHENEVER statements, as follows:

- *DEFER INTERUPT* and *DEFER QUIT*

  The DEFER statement can be used to instruct your executing program to set built-in global variables, rather than terminate the program, when an Interrupt or Quit signal is generated by the user. By testing the variable, you can determine when the user is ready to end a program, and respond accordingly in an orderly manner to the request.

- *WHENEVER ERROR*

  The WHENEVER ERROR statement establishes the policy of your program for handling SQL errors, screen I/O errors, and validation errors.

- *WHENEVER ANY ERROR*

  The WHENEVER ANY ERROR statement extends the WHENEVER ERROR policy to expression errors.

- *WHENEVER WARNING*

  The WHENEVER WARNING statement establishes the policy of your program for handling SQL warnings.

- *WHENEVER NOT FOUND*

  The WHENEVER NOT FOUND statement establishes the policy of your program for handling SQL "end of data" conditions.

Not all runtime errors can be trapped by WHENEVER. For a list of fatal errors that always terminate program execution, see *INFORMIX-4GL Reference*. See also the DEFER and WHENEVER statement descriptions in *INFORMIX-4GL Reference*.

When using 4GL to develop a new program or to modify an existing program, you might want to use one set of exception-handling policies during development, when your developers can read the error messages, but use another policy after the completed program is deployed, if your users are not likely to benefit from reading the text of any runtime error messages.

Chapter

6

Database Access and Reports

In This Chapter

One main reason to use INFORMIX-4GL is the ease with which you can access and modify data in a database. SQL, the international standard for relational database access, is an integral part of the 4GL language. Another reason to use 4GL is the ease by which you can design and generate reports. This chapter describes how to use SQL to access data in a 4GL program and how to generate reports.

Using SQL in a 4GL Program

You can use SQL in a 4GL program in the following three ways:

1. Incorporate nonprocedural SQL statements as 4GL program statements.

   Any SQL statement that does not return data to the program (for example, ALTER, CREATE, UPDATE, or REVOKE, among many), can be embedded within the program for execution in sequence with other 4GL statements. In many cases, you can use data from program variables as part of the statement.

   In addition, any SELECT statement that returns a single row of data can be written into the program and used to get data from the database and assign it to variables.

2. Use a *database cursor* to retrieve a set of rows from the database, one row at a time.

   A cursor contains a SELECT statement that might return multiple rows of data. You use an OPEN statement to start the selection. You use FETCH statements to fetch one selected row at a time, assigning the column data to variables. In this way, your program can scan a selection of rows, or bring all or part of the selection into memory and store it in an array.

   The FOREACH loop is another mechanism that can be used to open a cursor and fetch rows in sequence.

3. Use dynamic SQL to prepare new statements at execution time.

   You can assemble the text of an SQL statement and then pass it to the database server for execution. In this way, you can write an application that adapts to the schema of the database or to user selection criteria at execution time.

The concepts behind all three of these methods are discussed in greater detail in Chapter 9, "Using Database Cursors," and Chapter 10, "Creating Reports." Additional discussion and examples are provided in the *Informix Guide to SQL: Tutorial*.

Tip: *In this version of 4GL, you can directly embed most 4.1 SQL statements. If you want to include SQL syntax introduced after Version 4.1, you must prepare the statement before including it in the program. (You prepare a statement by using the PREPARE statement.) For a list of supported SQL statements, see the description of the PREPARE statement in "INFORMIX-4GL Reference."*

Creating 4GL Reports

A *report* is a special program block of 4GL that can produce formatted output. typically based on values from a database. This output can be sent to the screen, to a printer, to a file, or (through a pipe) to another program.

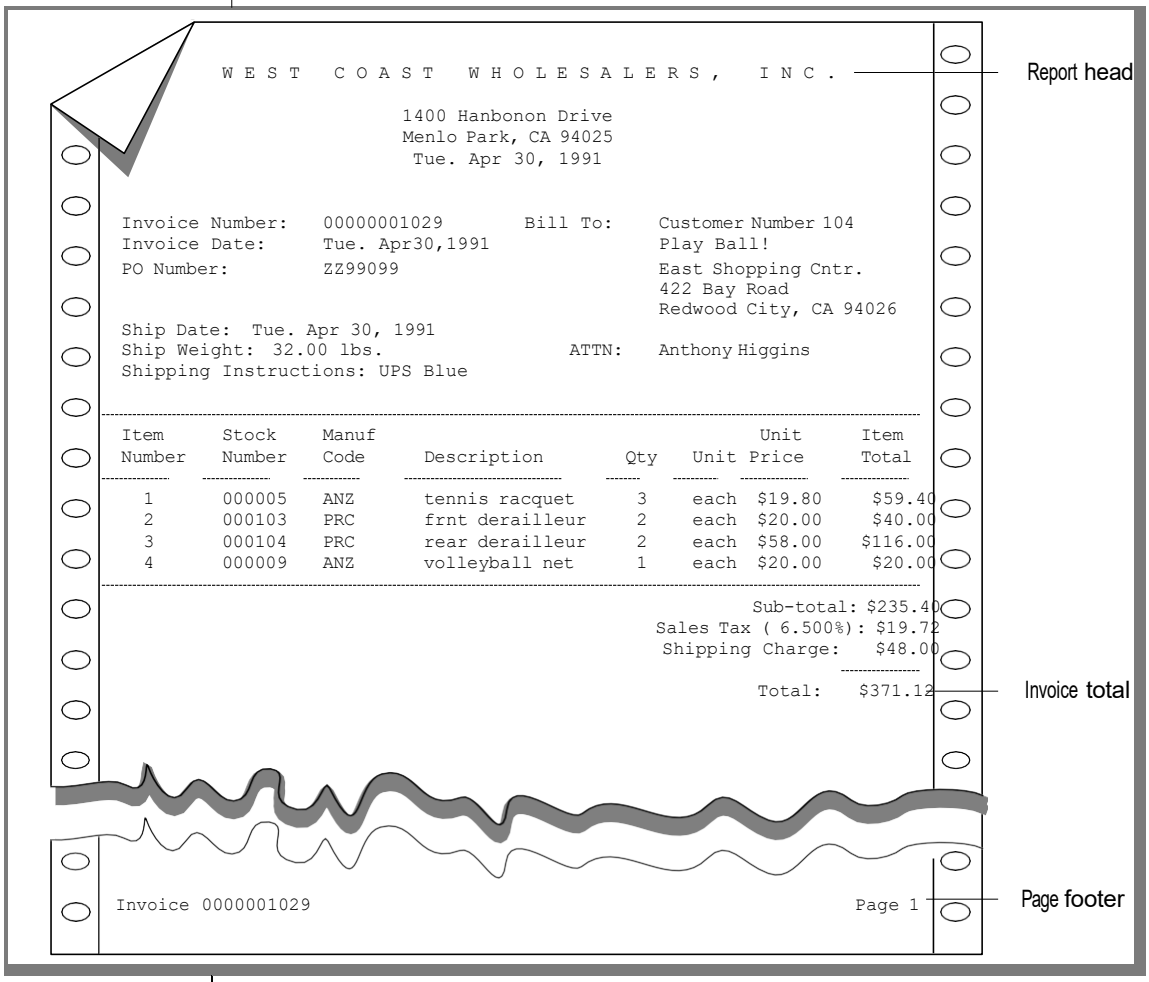
Output from a well-designed report should be arranged so that the eye of the reader can easily pick out the important facts, such as column totals or sub-totals. A report is meant to be viewed on the screen or on paper, so it needs to be arranged in pages, often with a *header* and a *footer* on each page, possibly with page totals.

The most important technique for making data clear to the eye is logical layout. The data items should almost always be arranged so that:

- the reader can quickly find an item.
- logically related items are near each other.
- groups of data with logically related values appear near one another, so that group totals and subtotals can be calculated and shown.

Figure 6-1 shows some output from a typical 4GL report.

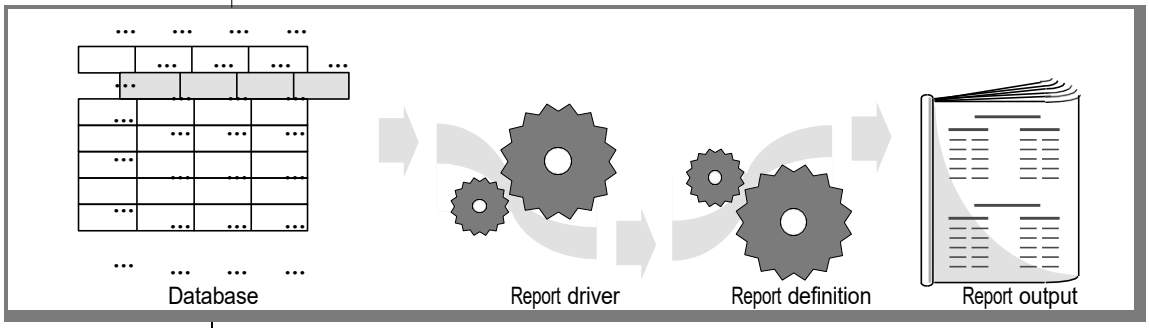
Figure 6-1
Example of a 4GL Report



In 4GL, the program logic that specifies what data to report is separate from the program logic that formats the output from the report. Any 4GL program that produces output from a report must include a part that produces the data from a database (or from some other source), and a second part that formats the data as output from a report.

In these manuals, the part of the 4GL program that sends data to the report is called the *report driver*, and the part that specifies how to format and display the output from the report is called the *report definition*, as Figure 6-2 shows.

Figure 6-2
Process for Generating Output from a 4GL Report



The Report Driver

The part of a program that supplies the report with rows of data (also know as *input records*) is called the report driver. It has the following features:

- It can interact with the database and with the user.
- It ignores all formatting issues, such as page length.
- It can supply data for multiple reports simultaneously.
- Data that it sends to a report can come from sources outside any database; for example, productivity data from user commands.

The primary concern of the row-producing logic in a report driver should be the selection of data, rather than the arrangement or formatting of data.

The report driver can use specialized 4GL statements to take the following actions:

1. Use START REPORT to initialize each report to be produced.
2. Whenever a row of report data is available, use OUTPUT TO REPORT to send it as an input record to the report definition.
3. If the report driver detects an error, use TERMINATE REPORT to bring the report generation process to an end.
4. After the last row is sent, use FINISH REPORT to end the report.

The last two actions are mutually exclusive. Unless there is the need for an abnormal termination, FINISH REPORT terminates execution of the report.

From the standpoint of the report driver, no other statements are required. The driver can concurrently produce input records for multiple reports.

Although a database is the usual source of data for a report, input records can come from any source, including the user, calculations, or sequential files. Because the report driver pays no attention to issues of formatting, grouping, or totalling, it produces input records as a by-product of other activities.

Your program is not required to produce input records in any special order. It is generally more efficient, however, to retrieve data from the database in the desired order, using the ORDER BY directive. You can, however, produce input records in any order, and leave the sorting to the report definition.

The 4GL statements that logically make up a report driver can appear within a single program block, or they can be distributed across several functions.

The Report Definition

You define how the output from the report is formatted in the REPORT statement, which in several ways resembles the FUNCTION statement of 4GL. The REPORT definition consists of the following several sections, in a fixed order:

- **REPORT prototype.** This section declares the name of the report, and list its formal arguments. This name must be unique among named program blocks.
- **DEFINE section.** This section declares variables that are local to the report, including its formal arguments and other variables (such as for calculated results).
- **OUTPUT section.** This section defines the size and margins of the report page. This section takes effect when the report is started.
- **ORDER BY section.** This section specifies the order for the input records, and whether or not they are provided to the report already ordered.
- **FORMAT section.** This section specifies what is to be done to the input records to produce and format the output from the report.

The control blocks that you write in the FORMAT section are the heart of the report definition, and contain all its intelligence. Statements in the control blocks of this section can specify actions to take in the following contexts:

- Top (header) of the first page of the report.
- Top (header) of every page after the first.
- Bottom (footer) of every page.
- Each new row as it arrives.
- The start of a group of rows. (A *group* is one or more rows having equal values in a particular column.) This statement block is often used for clearing totals and other accumulated values.
- The end of a group of rows. In this block, you typically print subtotals and other aggregate data for the group that is ending. You can call on aggregate functions like SUM and MAX for this information.
- After the last row has been processed.

You can use most 4GL statements in the FORMAT section of a report. For example, you can call functions and interact with the user. Some statements of 4GL that are not valid within a report definition include the following:

- DEFER (which can only appear in the MAIN program block)
- MAIN, FUNCTION, and REPORT (which are each program blocks; you cannot nest one 4GL program block within another)
- RETURN (which can only appear in a FUNCTION program block)

Use EXIT REPORT, rather than RETURN, to terminate processing of input records from within a REPORT definition. EXIT REPORT resembles in its effect what TERMINATE REPORT does in a report driver.

Use PRINT, rather than DISPLAY, to produce output within a REPORT definition.

4GL invokes the sections and control blocks within a REPORT definition non-procedurally, at the proper time, as determined by the data that the report is processing. You do not have to write code to calculate when a new page should start. Neither do you have to write comparisons to detect when a group of rows has started or ended. All that you have to write are the statements that are appropriate to the situation, and 4GL supplies the *glue* to make them work.

For more information about report drivers and report definitions, see Chapter 10, "Creating Reports," in this manual, as well as *INFORMIX-4GL Reference*.

Chapter

7

The User Interface

In This Chapter

Built into INFORMIX-4GL is a complete system of *character-oriented* user interaction. Because 4GL is character-oriented, the same 4GL applications can run on high-end workstations and on character-based terminals. 4GL enables you to create highly flexible, portable, interactive, multiuser applications using screen forms and menus.

Line-Mode Interaction

A 4GL program can operate with its user interface in *line mode* or *formatted mode*. Line mode is the same typewriter-like mode of interaction that UNIX shell scripts use. You put the user interface in line mode by executing a DISPLAY statement that does not specify a screen location. The following example illustrates a simple program that operates in line mode:

```
MAIN
   DEFINE centDeg, fahrDeg DECIMAL(5,2)
   DEFINE keepOn CHAR(1)
   LET keepOn = "y"
   DISPLAY "Centigrade-to-fahrenheit conversion."
   WHILE keepOn == "y"
      PROMPT "Centigrade temp: " FOR centDeg
      LET fahrDeg = (9*centDeg)/5 + 32
      DISPLAY "old-fashioned equivalent: ", fahrDeg
      PROMPT "More of this (y/n) ? " FOR CHAR keepOn
   END WHILE
END MAIN
```

Because the first DISPLAY statement does not give a screen row and column for output, the screen is put in line mode. Each line of display and each prompt displayed by PROMPT scrolls up the screen in the manner of a typewriter.

When you execute this program, the interaction on the screen resembles the following example, in which data entry by the user has been underscored:

```
Centigrade-to-fahrenheit conversion.
Centigrade temp: 16

old-fashioned equivalent: 60.80
More of this (y/n) ? y

Centigrade temp: 28

old-fashioned equivalent: 82.40
More of this (y/n) ? n
```

You can use simple interactions of this kind for quick tests of algorithms. Line mode also has the virtue that you can redirect line mode output to disk from the command line. The following program displays two columns from the **customer** table using line-mode output:

```
DATABASE stores7
MAIN
   DEFINE custno LIKE customer.customer_num,
          company LIKE customer.company
   DECLARE cust CURSOR FOR
             SELECT customer_num, company FROM customer
   FOREACH cust INTO custno, company
      DISPLAY custno, company -- Line mode display
   END FOREACH
END MAIN
```

You could execute this program from the command line, redirecting its output into a file, with a statement like the following one (assuming the program has been compiled to 4GL p-code in a file named **dump2col.4gi**):

```
fglgo dump2col | custcols.dat
```

The data could also be *piped* into another command, but there is no equivalent input statement; PROMPT only accepts input from a real keyboard.

Formatted Mode Interaction

Normally, 4GL keeps the user interface in formatted mode. That is, the output of the program is automatically formatted for screen display. The program's output is positioned by rows and columns. On character terminals, the initial 4GL window is the same size as the screen and is referred to as the *4GL screen*.

The following table shows a brief summary of the 4GL statements that you can use to manage the user interface in formatted mode.

Statement	Purpose
DISPLAY…AT	Write data at specific rows and columns.
DISPLAY FORM	Display the background of a prepared form.
DISPLAY…TO	Write data into one or more fields of a form.
PROMPT	Prompt the user for a single value or a one-character response.
INPUT	Let the user enter data into one or more fields or arrays of fields on a form.
CONSTRUCT	Let the user enter query criteria into the fields of a form.
MESSAGE	Display a short message of warning or confirmation.
ERROR	Display a short message documenting a serious error to the screen.

These statements are described in detail in "Screen and Keyboard Options" on page 11-32 and in *INFORMIX-4GL Reference*.

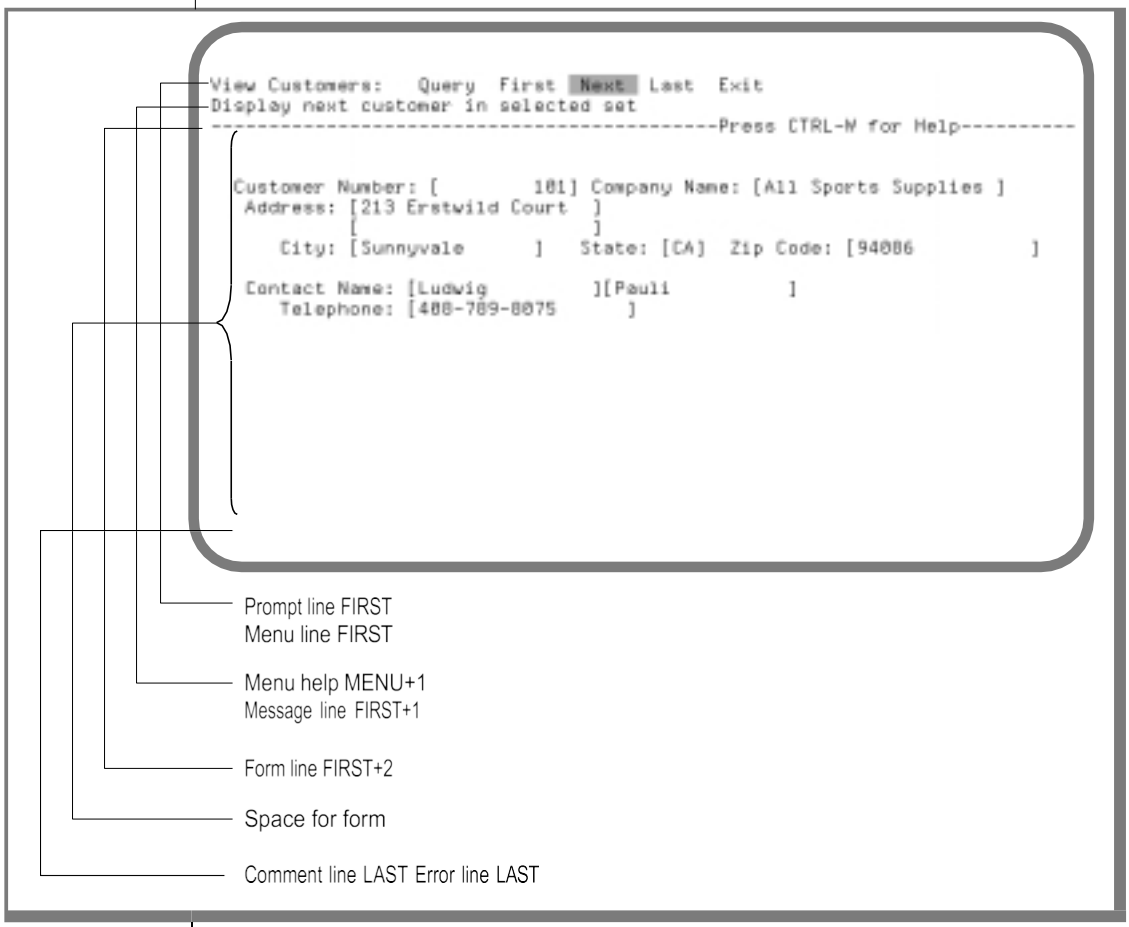
Formatted Mode Display

In formatted mode, certain lines of the two-dimensional screen are reserved for certain types of information. You can change the location of these lines, but you cannot eliminate their special uses. The following table shows the reserved screen lines with their default positions.

Line Name	Purpose	Default Position	Point of Reference
Prompt	Output and input of PROMPT statement	FIRST	4GL window
Menu	Ring-menu display occupies two lines	FIRST	4GL window
Message	Output of the MESSAGE statement	FIRST+1	4GL window
Form	Top line of any form	FIRST+2	4GL window
Comment	Explanatory text for current form field	LAST	4GL window
Error	Output of the ERROR statement	LAST	4GL screen

Figure 7-1 shows how the reserved lines are arranged on the screen when the default reserved line positions are in effect.

Figure 7-1
Default Reserved Line Positions



To create this screen, a form was displayed using DISPLAY FORM; some data values were written into form fields using DISPLAY…TO; and then a MENU statement was used to display a five-option menu (the options **Query** through **Exit**). The line of hyphens with `Press CONTROL-W for Help` is literal text in the first line of the form that was displayed.

This diagram makes the screen appear crowded with conflicting uses. In reality, the dedicated lines are used at distinct times. For example, the Prompt line is used by the PROMPT statement and the Menu line by the MENU statement. The program cannot execute both PROMPT and MENU at the same time, so no conflict is possible. By default, both lines are assigned to the first 4GL window line.

The assignment of specific rows to screen lines can be changed while the program is running (see "Screen and Keyboard Options" on page 11-32). The key point is that these lines exist and have assigned display positions.

Sample Code for Formatted Mode Display

In the following example, screen output is achieved using DISPLAY AT, so the PROMPT statement uses only the current screen position (first line of the window, by default). As a result, this dialog will not scroll, but will re-use the same two screen rows over and over.

```
MAIN
   DEFINE centDeg, fahrDeg DECIMAL(5,2)
   DEFINE keepOn CHAR(1)
   LET keepOn = "y"
   DISPLAY "Centigrade conversion" AT 12,1
   WHILE keepOn == "y"
      PROMPT "Centigrade temp: " FOR centDeg
      LET fahrDeg = (9*centDeg)/5 + 32
      DISPLAY centDeg, "C ==> ", fahrDeg, "F" AT 3,1
      PROMPT "More of this (y/n) ? " FOR CHAR keepOn
   END WHILE
END MAIN
```

Screens and Windows

In order to understand the way that forms and menus are used, you should understand the distinction between the 4GL *screen* and a 4GL *window*.

The Computer Screen and the 4GL Screen

The computer screen is the physical surface on which your program displays data. When 4GL program output is directed to the screen, it appears in the 4GL screen, also known as the logical screen.

If you are using a terminal, the entire computer screen is the 4GL screen. If you are using a workstation, the window in which you are working is considered the 4GL screen.

The 4GL Window

A 4GL window is a rectangular area within which your program can display output. Initially, your program has one 4GL window that fills the 4GL screen. Additional 4GL windows can be opened or closed as needed.

Each 4GL window is a new rectangular area on which 4GL can display output. You can use a second or additional 4GL windows in the same way that you use the first one: to display messages and to prompt for input and to display menus and forms.

All 4GL windows are contained inside the boundaries of the 4GL screen. They can be the same size or smaller than the screen, but not larger. Any 4GL window can overlap or completely obscure another 4GL window.

At any given time, only one 4GL window is *current*. This window is where DISPLAY, MENU, and other interactive statements operate. Other windows can be completely or partly visible, but only the current window is active.

How Menus Are Used

The MENU statement enables you to offer the end user a *ring menu* that contains menu options. 4GL displays the menu on the designated Menu line of the current window. As the user presses the TAB or arrow keys or SPACEBAR, 4GL moves the cursor from menu option to menu option. The user presses RETURN to select the current option. The user can also select a menu option by pressing the activating character (usually, but not always, set to be the first letter of the menu option) to select an option.

In your 4GL program, you supply the list of options and, for each option, a block of statements. When the user selects an option, 4GL executes the block of code corresponding to it. You can create as many levels of ring menus as you like.

One common use of this technique is to create nested menus, as illustrated in Figure 7-2.

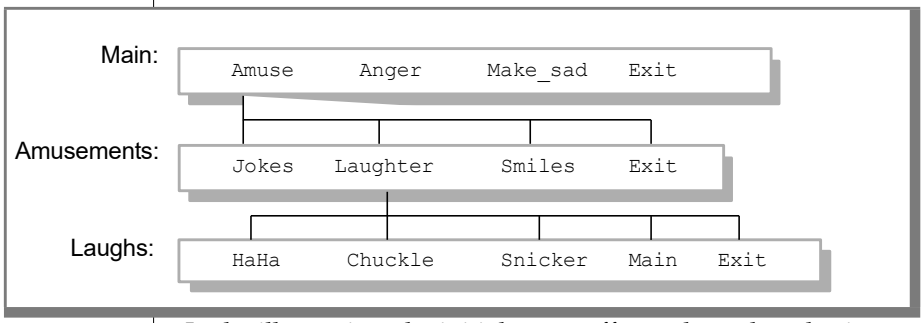


Figure 7-2
Nested Menus

In the illustration, the initial menu offers at least three basic emotions. The user can choose to enter the **Amuse**, **Anger**, or **Make_sad** ring menu or choose **Exit** to leave the program.

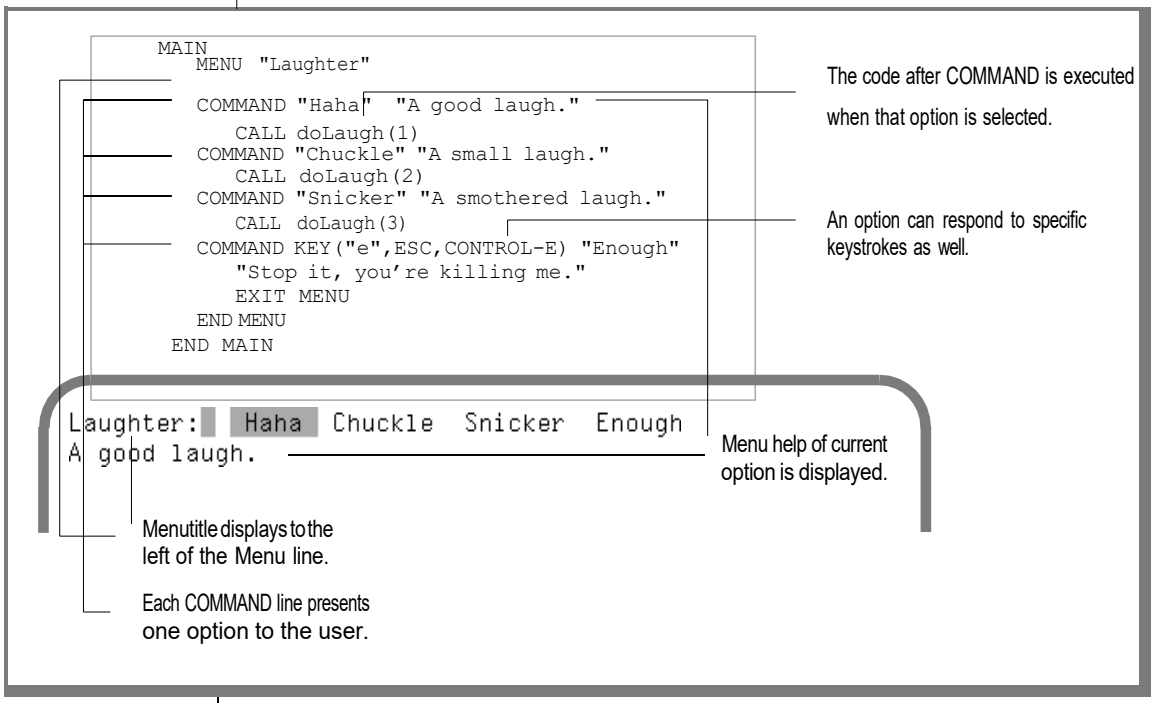
If the user chooses **Amuse**, the second-tier ring menu replaces the first.

The **Exit** menu option moves the screen cursor to the pervious menu or, if you are already at the top tier of that menu, it exits from the program. Alternatively, you could create a ring-menu option that bypasses the natural hierarchal ring-menu structure. An example is shown in the third-tier menu, which offers the ability to jump back to the top tier by choosing **Main**.

The number of ring-menu levels that you can create is limitless.

Figure 7-3 shows a simple menu-driven program. When the program runs, the menu shown in the lower portion will be seen by the end user.

Figure 7-3
Menu-Driven Program

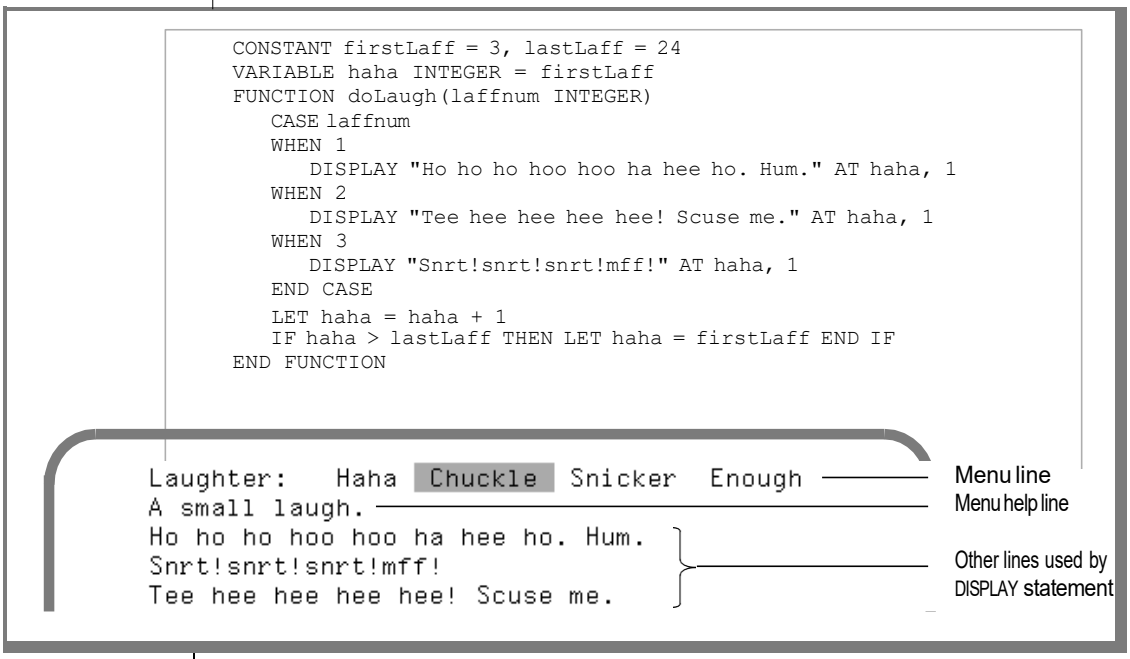


This program presents a menu that contains four options: **Haha**, **Chuckle**, **Snicker**, and **Enough**. The user selects a menu option by typing its initial letter or by moving the cursor and pressing RETURN. You can also arrange for an option to be selected using other keys; for example, the option **Enough** can be chosen by pressing either ESCAPE or the CONTROL-E.

When the user selects a menu option, 4GL executes the block of code that follows the COMMAND statement for that option. In this example, if the user selects the third option, **Snicker**, the code `CALL doLaugh(3)` will be executed. When all the code for that option has been executed (in this case when the **doLaugh( )** function returns), the MENU statement resumes execution and the user can pick another option (unless EXIT MENU is encountered, as in the case of **Enough**).

Figure 7-4 shows an implementation of **doLaugh( )** and an example of the output that it produced when several menu options were selected.

Figure 7-4
Output Produced by doLaugh() Function



After a command block completes, 4GL redraws the Menu line and Menu Help line, and waits for the user to choose another menu option. Program control remains within the MENU statement until it executes an EXIT MENU statement within some COMMAND block. The program on the previous page executes EXIT MENU when the **Enough** option is chosen.

You can write any number of lines of code in a command block. The example program shows a common style in which each command block contains a single function call, but you can use most 4GL statements in a command block. You can communicate with the user with MESSAGE, DISPLAY, or PROMPT statements, open additional 4GL windows, or even start another MENU statement.

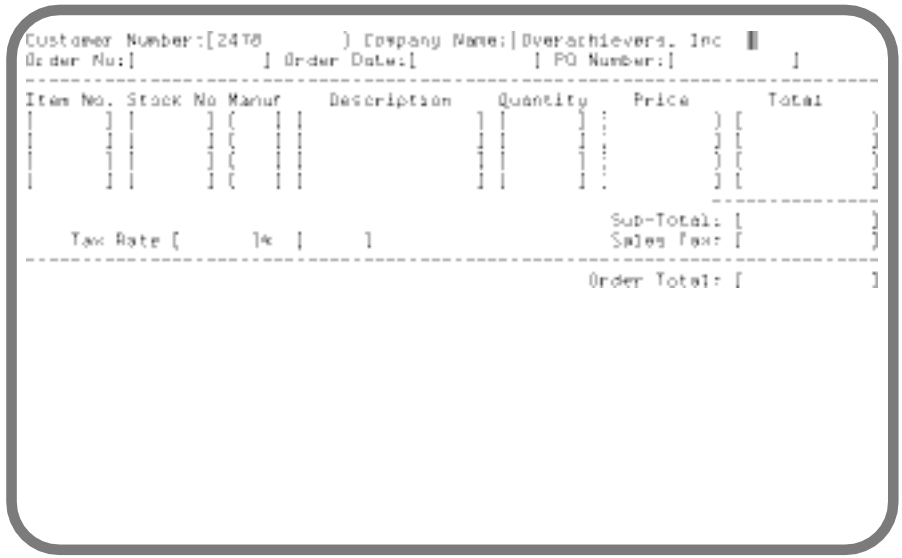
You can change the appearance of a menu while the program is executing. Within the menu, you can execute the HIDE and SHOW commands to hide or display menu options. For example, you could test the level of privilege of a user within the current database, and then either HIDE or SHOW a choice such as **Delete row**, depending on whether the user has delete privilege.

All these features are covered in detail in *INFORMIX-4GL Reference*.

How Forms Are Used

A *form* is a fixed arrangement of *fields* and *labels*. You design a form with fields to hold data items that you want to display, and labels to describe the fields to the user. Figure 7-5 shows the form used in Example 11 in *INFORMIX-4GL by Example*.

Figure 7-5
A Sample Screen Form



Defining a Form

The following two steps are involved in creating a form:

- Specify the contents of a form in a form specification file, a text file that you create with any text editor that can generate ASCII text. Form specification filenames should be given the extension **.per**.
- Compile the form specification file. Compiled forms are usually given the file extension **.frm**.

The FORM4GL utility program is used to create **.frm** files. Once compiled, a 4GL form can be used by any 4GL program.

The form specification file has several sections. The DATABASE, SCREEN, and ATTRIBUTES sections are required, while the TABLES and INSTRUCTIONS sections are optional. The order of appearance of the sections is fixed. The sections of the form specification file are described next.

After you have designed a form and compiled its specification, it is ready for use by a program.

The form specification file that produces the form in Figure 7-5 on page 7-13 is considered in further detail beginning with "Specifying a Form" on page 11-3. Special syntax and keywords of form files are discussed in *INFORMIX-4GL Reference*.

DATABASE Section

The DATABASE section names a database from which column data types can be determined when the form is compiled. Alternatively, you can use the keyword FORMONLY to indicate that the form does not rely on a database. For example, you can identify the **stores7** database as follows:

```
DATABASE stores7
```

SCREEN Section

The SCREEN section contains an ASCII version of the form, including text labels establishing the size and location of form fields. Here fields are labeled by *field tags*, internal names that are not displayed when the form appears at runtime. The following example is also the SCREEN section of the form in Figure 7-5:

```
SCREEN
{

Customer Number:[f000        ] Company Name:[f001                 ]
Order No:[f002        ] Order Date:[f003       ] PO Number:[f004       ]

Item No. Stock No Manuf     Description     Quantity     Price      Total
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
}
END
```

TABLES Section

The TABLES section lists the tables or table aliases in the default database specified in the DATABASE section, or specified here by table qualifiers, from whose columns the field data types will be taken. An alias is required if an owner or database qualifier of the table name is needed. For example, you can identify tables as follows:

```
TABLES
customer
orders
items
stock
catalog
```

ATTRIBUTES Section

In the ATTRIBUTES section, you specify the characteristics of each field: the field name, the type of data that it will display, the editing rules applied during input, and any special display attributes, such as color. For example:

```
ATTRIBUTES
f000 = customer.customer_num ;
f001 = customer.company ;
f002 = orders.order_num ;
f003 = orders.order_date ;
f004 = orders.po_num ;
f005 = items.item_num , NOENTRY ;
f006 = items.stock_num ;
f007 = items.manu_code ;
f008 = stock.description , NOENTRY ;
f009 = items.quantity ;
f010 = stock.unit_price , NOENTRY ;
f010 = items.total_price, NOENTRY ;
END
```

INSTRUCTIONS Section

In the INSTRUCTIONS section, you can group fields into *screen records* and *screen arrays*. These records and arrays can be displayed and read as logical units, as the following example shows:

```
INSTRUCTIONS
SCREEN RECORD s_items[4] ( item_num , stock_num , manu_code ,
   description , quantity, unit_price , total_price )
END
```

In the INSTRUCTIONS section, you can also change the default delimiters of form fields when they are displayed on character-based systems.

Displaying a Form

Your program can use a form in the following ways:

- With OPEN FORM or OPEN WINDOW ... WITH FORM, you load the compiled form from disk into memory and make it ready for use.

  You can open as many forms as needed, subject only to the limits of memory and maximum number of open files on your platform.

- With DISPLAY FORM, you draw the contents of a form (its labels and the outlines of its fields) in the current 4GL window. The picture of the form replaces any previous data values in that window.

  You can display a form as many times as necessary. You can display the same form in different 4GL windows. (The use of additional windows is described in "How 4GL Windows Are Used" on page 7-25.)
- With DISPLAY…TO, you can fill the fields with data from program variables.

You can also use the CLEAR FORM statement to empty the fields of data.

Reading User Input from a Form

With the INPUT statement, your program waits for the user to supply data for specific fields of the form in the current 4GL window. In the INPUT statement, you must list the following information:

- The program variables that are to receive data from the form
- The corresponding form fields in which the user will enter the data

When invoked, the INPUT statement enables the specified fields. The user moves the cursor from field to field and types new values. Each time the cursor leaves a field, the value typed into that field is deposited into the corresponding program variable. Other fields on the form are deactivated. The INPUT statement ends when the user does one of the following actions:

- Uses the **Accept** key (by default, ESCAPE) to resume execution and examine and process the values the user has entered
- Uses the **Cancel** key (by default, CONTROL-C) to resume execution and ignore any changes made to the form
- Completes entry of the last field, when field order is set to CONSTRAINED

  This is the same as **Accept**. See "Field Order Constrained and Unconstrained" on page 11-30 as well as *INFORMIX-4GL Reference*.

Screen Records

In the form file, you can specify a group of fields as a logical screen record. During input, your program can associate a program record with a screen record, automatically filling the RECORD variable in memory with data from the form.

Screen records can make your program shorter and easier to read, and can display all or part of a row from the database. You can use *asterisk* (wildcard) *notation* when referring to all the fields of the program record or the screen record. Suppose that your program defines a record variable in this way:

```
DEFINE itemRow RECORD LIKE items.*
```

This line declares a record variable with one member for each column of the **items** table. Now suppose that in the current form there are four fields that correspond to the last four columns of the **items** table. The schema of the **stores7** demonstration database, which is used in the following discussion, is described in *INFORMIX-4GL by Example*.

In the form file, these fields are grouped into a screen record with the following line in the INSTRUCTIONS section:

```
INSTRUCTIONS

SCREEN RECORD itemDetail(stock_num, manu_code, quantity,
                  total_price)
```

The program could take input from the four fields by specifying the fields and the corresponding record members individually:

```
INPUT itemRow.stock_num, itemRow.manu_code,
         item_row.quantity, itemRow.total_price
FROM  stock_num, manu_code, quantity, total_price
```

Or you can specify the last four members of the program record using THRU notation and all the fields of the screen record using an asterisk:

```
INPUT itemRow.stock_num THRU itemRow.total_price
   FROM itemDetail.*
```

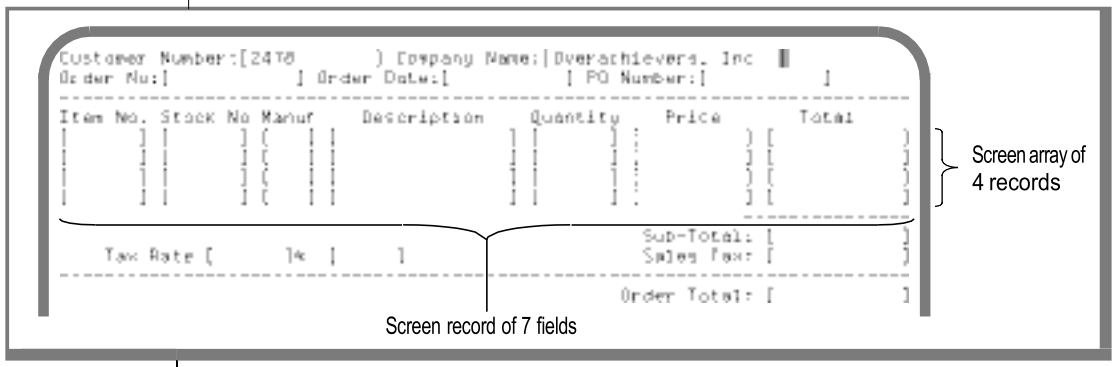
But because the names of the members in the program record are the same as the names of the form fields, this can be further shortened to:

```
INPUT BY NAME itemRow.stock_num THRU itemRow.total_price
```

Screen Arrays

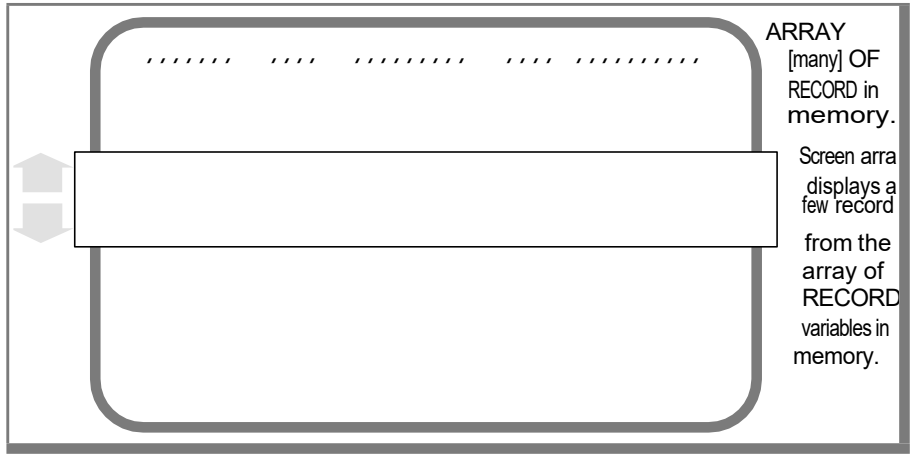
In the form specification, you can also specify a group of screen records as a screen array. During input, you can associate a *program array* of records with an array of form fields on the screen, as illustrated in Figure 7-6.

Figure 7-6
Screen Record Associated with a Screen Array



As Figure 7-7 shows, the program array typically has many more rows of data than will fit on the screen.

Figure 7
Typical Screen Array Displaying Part of a Much Larger Program Arr



4GL lets the user *scroll* the array on the screen through the rows of the program array. The user can change the display using the 4GL **PageUp** and **PageDown** logical keys or scroll through the array one line at a time by using the arrow keys.

To add a record, the user can press the logical **Inser**t key, and 4GL will open the display to create an empty screen record. When the user has filled this record, 4GL inserts the data into the program array.

To delete data, the user can press the logical **Delete** key, and 4GL will delete the current record from the display and from the program array, and redraw the screen array so that deleted records will no longer be shown. Depending on how your program is written, you can also programmatically remove the record from the database. For a complete list of logical key assignments, see the description of the OPTIONS statement in *INFORMIX-4GL Reference*.

How the Input Process Is Controlled

Your program stops in the INPUT or INPUT ARRAY statement and waits while the user enters data. You can write blocks of code, however, that are automatically executed by 4GL during input, to monitor and control the actions of the user during data entry. The control blocks with ROW, INSERT, or DELETE keywords are valid only in INPUT ARRAY statements, and not with INPUT:

- BEFORE INPUT

  Just as the INPUT operation is starting, this block of code can display initial or default values, clear totals, and generally prepare the screen and program variables.

- BEFORE FIELD

  As the cursor is entering the specified field, this block can take some action, such as initialize the contents of the field, based on values in other fields.

- AFTER FIELD

  When input in the specified field is complete, this block can validate what the user entered, or can initialize other fields, based on the value just entered.

- ON KEY

  When the user presses any of a list of keys that you specify, this block can give the user special assistance; for example, displaying a list of common values for the current field.

- BEFORE ROW

  When the cursor is about to enter a new row of a screen array, this block can take some action, such as update other fields on the screen to reflect the row being entered.

- AFTER ROW

  When the cursor is leaving a row of a screen array, this block can take some action, such as update the screen or the database to account for changes made in the row.

- BEFORE INSERT

  When the user has requested creation of a new row in a screen array, this block can take some action, such as initialize the new row with default values.

- BEFORE DELETE

  When the user presses the **Delete** key to remove a row from a screen array, this block can take some specified action before the row is deleted.

- AFTER DELETE

  When the user presses the **Delete** key to remove a row from a screen array, this block can take some specified action after the row is deleted.

- AFTER INSERT

  When the cursor is about to leave a newly inserted row of a screen array, this block can update totals based on the new data, and can insert the new row into the database.

- AFTER INPUT

  When the input operation is ending, this block can validate the entered data, check for required fields that might be missing, and erase any special usage messages.

You write these blocks as part of the INPUT or INPUT ARRAY statement. When the INPUT or INPUT ARRAY statement is executed, 4GL enables the screen for input and awaits user keystrokes. When the user presses a key that creates any of the situations described in the preceding list, 4GL automatically calls your block of code.

You can include most executable 4GL statements in these blocks, as well as two special keyword clauses that can reposition the screen cursor:

- NEXT FIELD

  Use this clause in an AFTER FIELD or ON KEY block to direct the cursor to a specified next field, or back to the same field to correct an error in data entry.

- NEXT ROW

  Use this clause in an AFTER ROW or ON KEY block to move the cursor to a specified row of a screen array. 4GL scrolls the array as necessary to show the specified row.

NEXT FIELD and NEXT ROW should only be used in situations where the program controls the order in which fields are visited by the user. For additional information on programming this type of user interaction, see "Field Order Constrained and Unconstrained" on page 11-30.

How Query by Example Is Done

4GL lets you take input from a form in another way: instead of entering literal values for the program to process, your user can enter logical criteria for a query. The process is known as *query by example*. The user enters a value or a range of values into one or several form fields. Then your program retrieves the database rows that satisfy the requirements that the user entered.

The 4GL statement that makes query by example possible is CONSTRUCT, as illustrated in Figure 7-8.

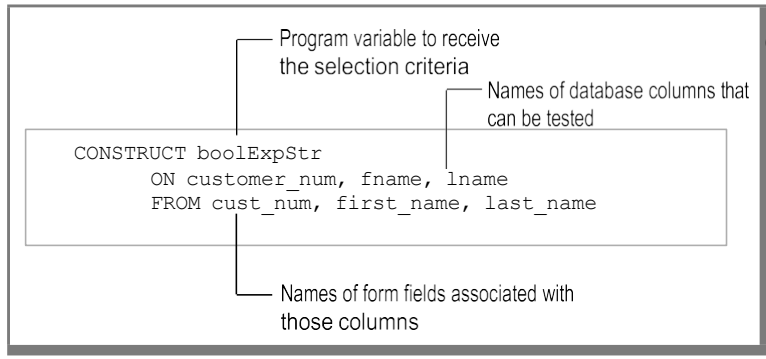


Figure 7-8 CONSTRUCT Statement

The CONSTRUCT statement operates much like INPUT. The CONSTRUCT statement lists names of database columns and names of fields in the current form that correspond to those database columns. It also supports control blocks (including ON KEY, BEFORE FIELD, AFTER FIELD, BEFORE CONSTRUCT, and AFTER CONSTRUCT) that resemble in their names and functionality several of the INPUT statement control blocks.

You provide a single program variable to hold the criteria for the query. When the CONSTRUCT statement executes, 4GL enables the form fields listed in this CONSTRUCT statement for input. The user can enter either specific values, or requirements such as `>5` (meaning any value greater than 5) or `="Sm[iy]th*"` (meaning any value beginning with "Smith" or "Smyth").

When the user presses **Accept**, 4GL converts the input into a Boolean expression suitable for use in the WHERE clause of a SELECT statement. This character string is returned to the program variable. When the CONSTRUCT statement shown in the previous illustration has completed, the following character string might be stored in the program variable **boolExpStr**:

```
customer_num > 5 AND lname MATCHES "Sm[iy]th*"
```

Now it is up to your program to use the Boolean expression to fetch the database row, or rows, that the user wants to see. This fetch includes the following steps:

1. Combine the Boolean expression string with other text to form a complete SELECT statement:

   ```
   LET fullStmt = "SELECT * FROM customer WHERE " ,
   boolExpStr
   ```

2. Prepare the SELECT statement for execution:

   ```
   PREPARE qbeSel FROM fullStmt
   ```

3. Associate the prepared statement with a database cursor:

   ```
   DECLARE qbeCur CURSOR FOR qbeSel
   ```

4. Open the cursor and fetch the rows it has selected.

What you do with the rows depends on the specific application. Often the reason for the CONSTRUCT is to select rows to be viewed by the user. In such a case, the program could display each row individually in a form, or grouped in a screen array. Or you might choose a set of rows for processing in a report or specify a set of rows to be deleted or updated, and so forth.

How 4GL Windows Are Used

Each 4GL program begins with a 4GL window that fills and covers the entire 4GL screen. This screen area might be any of the following display devices:

- A terminal-emulation window on a workstation
- The physical screen of a serial terminal

Additional 4GL windows can be created and further manipulated with the following statements.

Statement	Purpose
OPEN WINDOW	Create a new window and make it the current window. You specify its size and location in relation to the upper-left corner of the screen.
CLOSE WINDOW	Close and discard a window by name.
CLEAR WINDOW	Empty the contents of a window, erasing anything displayed on it.
CURRENT WINDOW	Bring a window to the front if necessary, and make it current.

Only the current 4GL window has keyboard focus. This means that any interaction initiated by the user through the keyboard goes to the current 4GL window.

The statements listed in the table in "Formatted Mode Interaction" on page 7-5 all operate upon the current 4GL window.

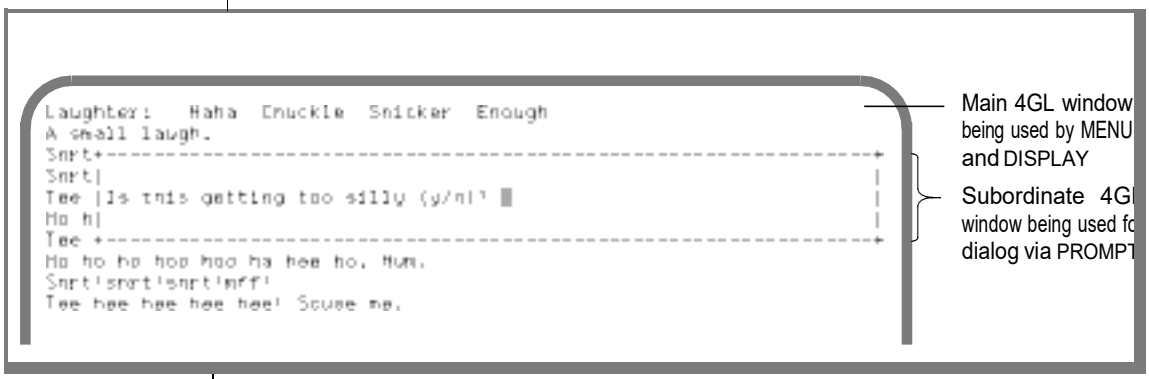
Alerts and Modal Dialog Boxes

One frequent use of a 4GL window is to display an error message or a dialog box to which the user must respond. Using only a few statements, you can perform the following tasks:

- Open a 4GL window.
- Conduct a dialog with the user.
- Close the 4GL window.

Figure 7-9 shows what an alert window might look like.

Figure 7-9
Alert Window



The code that presented this window follows:

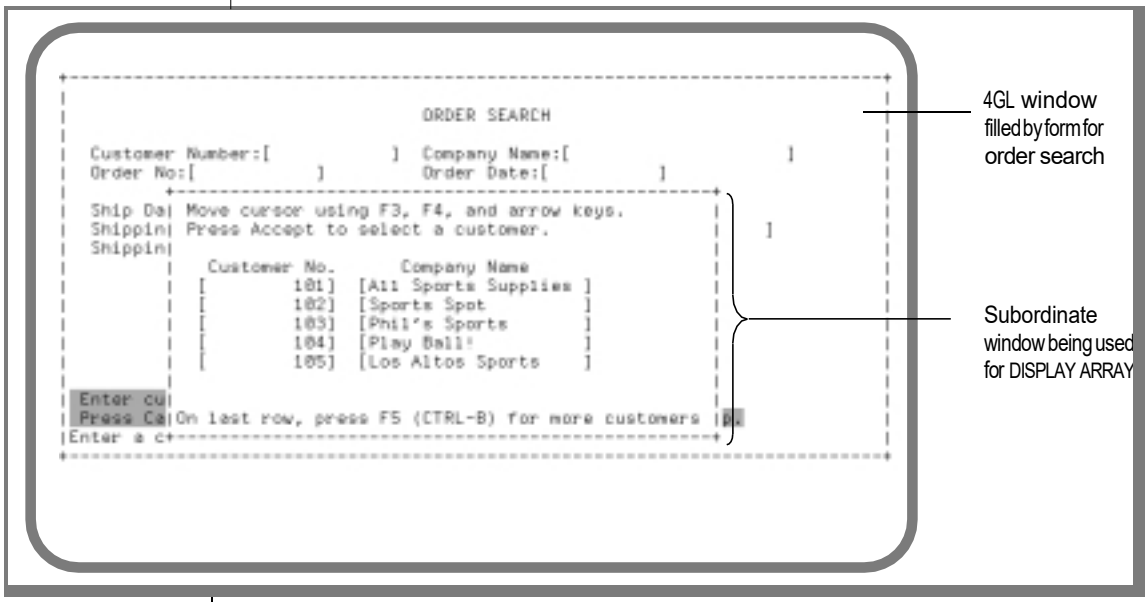
```
FUNCTION domodal()
   DEFINE ansr CHAR(1)
   OPEN WINDOW modal AT 4,6
      WITH 3 ROWS, 64 COLUMNS
      ATTRIBUTE(BORDER, PROMPT LINE 2)
   PROMPT "Is this getting too silly (y/n)? " FOR CHAR ansr
   CLOSE WINDOW modal
   RETURN ("Y" = UPSHIFT(ansr) )
END FUNCTION
```

The **domodal( )** function opens a subordinate 4GL window, prompts the use for a single-letter response, closes the window, and returns either TRUE o FALSE, depending on whether the response was the letter `Y` or not. A functio like this can be used at almost any point within a program.

Information Displays

Another common use for a 4GL window is to display helpful information to the user during input. In Figure 7-10, you see a larger window that contains a form, partly covered by another 4GL window.

Figure 7-10
Subordinate Window Displayed During Input



The user is to enter a customer number in a field of the form. Assume that user presses a designated key to ask for help. Within the INPUT statement, in the ON KEY block for that key, the program calls a function that does the following:

- Opens a 4GL window
- Displays a form in that 4GL window
- Uses DISPLAY ARRAY to show a scrolling list of rows from the **customer** table, using the form in the current 4GL window

When the user presses **Accept**, the function will note the customer number ir the last-current row. This value will be returned as the result of the functior Before the function returns, it will close the 4GL window it opened. That makes the larger 4GL window current again.

The ON KEY block will display the returned customer number in the form field so the user will not have to enter it.

How the Help System Works

4GL supports a simple and effective system of providing help to users in the form of help messages associated with interactive statements in your program. The purpose of a help message is to guide the user in contexts where it might not be clear how the program is intended to be used.

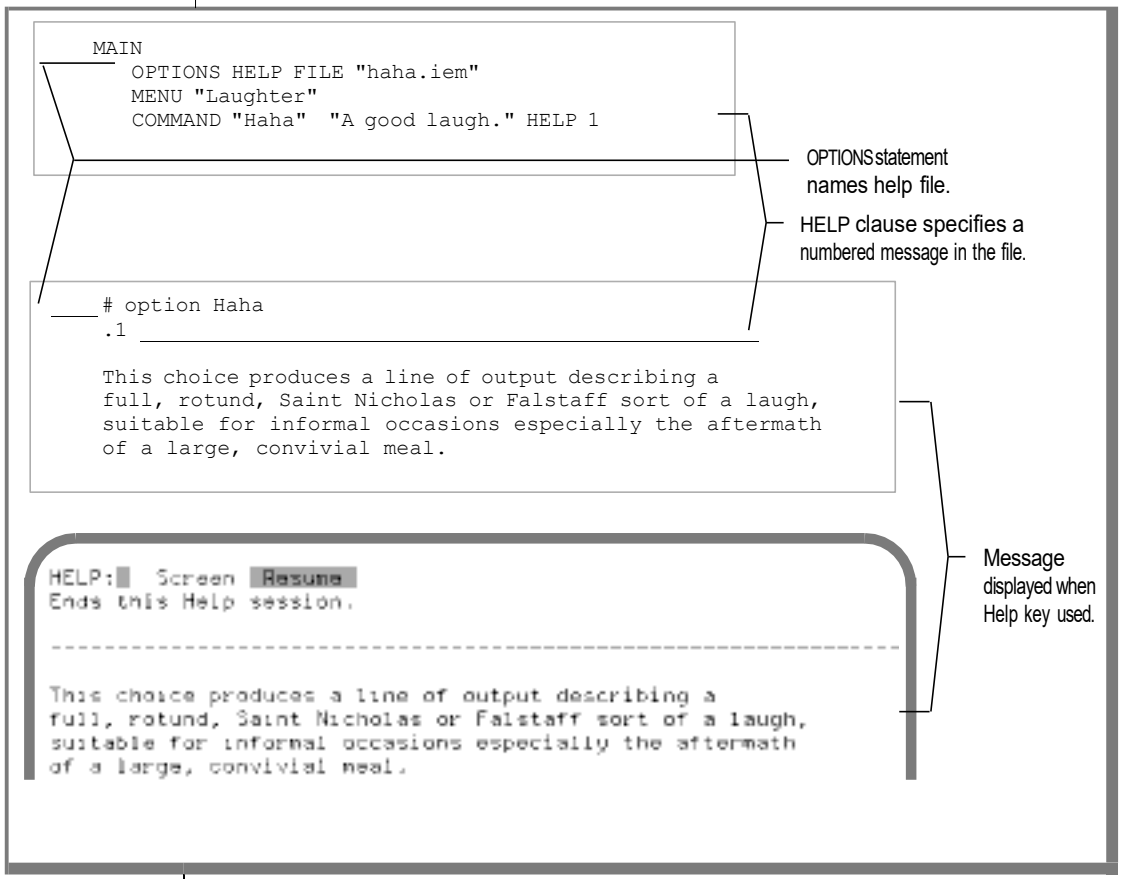
You might take the following typical steps in creating a useful help facility for an application:

1. Think of each situation where the user might need detailed guidance and write a message that explains what to do at that point.

   The text can be as long as you want. Only printable characters, blank spaces, and tab characters are allowed.
2. Assign to each message a unique positive integer number between 1 and 32700.
3. Put the message texts with their numbers and optional comments into a text file.
4. Compile the text file using the 4GL message compiler utility **mkmessage**.
5. In your program source file, specify the name of the file of compiled help messages in an OPTIONS HELP FILE statement.
6. In each interactive statement, specify the HELP *number* clause to name the help message number that you designed for that statement.

Figure 7-11 gives an overview of this process.

Figure 7-11
A Source Module, an ASCII Help File, and a Help Window



If the user presses the **Help** key (CONTROL-W by default) while the interactive statement is executing, 4GL displays the message whose number you specified.

The display takes up the full current window, hiding the main window and any subordinate windows.

Like a form file, a message file can be used with more than one program. Yo can use more than one message file within a program; executing the OPTION HELP FILE statement changes the active file at any time.

You can write a HELP *number* clause in any statement that requests input from the user: PROMPT, INPUT, or CONSTRUCT. You can specify a different help message number for each option of a MENU statement. In addition, you can start the help display from anywhere in the program by calling the library function **showhelp(*n*)**, where *n* specifies the message number in the current help file. Using this function, you can start a help display from, for example, an ON KEY block of a DISPLAY statement.

Chapter

8

Using the Language

In This Chapter

INFORMIX-4GL has the features of a structured language such as Pascal or C as well as advanced features of its own. This chapter surveys the basic features of 4GL as a programming language.

Data Types of 4GL

As Chapter 5, "The Procedural Language," indicates, 4GL is a *strongly-typed* language. Program variables, the formal arguments of functions and reports, and the values that are returned by functions and expressions must be represented as being of some data type. The data type determines the internal format in which a data value is stored, and also implies what operations on that value are valid.

Sections that follow describe the simple, large, and structured data types of 4GL and identify their role in various programming contexts, including declarations of variables, expressions, and function calls.

Simple Data Types

4GL supports a number of simple data types. They are called simple because each of these data types describes a single item of information (as opposed to a collection or an array of items). The simple data types of 4GL are a robust subset of the data types for database columns of Informix database servers.

Each of the simple data types is discussed in detail in *INFORMIX-4GL Reference*. There you will find information on their minimum and maximum capacities, the proper format for literal values of each data type, and other details. The data types are summarized below as background material for the rest of the chapter.

Number Data Types

4GL supports seven representations of numeric values (some of which have more than one keyword or have *scale* or *precision* options).

Data Type Keywords	Kind of Data Represented
DEC(*p*, *s*) DECIMAL(*p*, *s*) NUMERIC(*p*, *s*)	Decimal fixed-point numbers, of specified precision (*p*) and scale (*s*)
MONEY(*p*, *s*), MONEY(*p*), MONEY	Currency amounts, of specified precision (*p*) and scale (*s*) (defaulting to 16, 2)
DEC(p) DECIMAL(*p*) NUMERIC(*p*)	Decimal floating-point numbers, of specified precision (*p*) (defaulting to 16)
FLOAT DOUBLE PRECISION	Binary floating-point numbers, with precision of C **double**
REAL SMALLFLOAT	Binary floating-point numbers, with precision of C float
INT INTEGER	Whole numbers, from -2,147,483,647 to +2,147,483,647
SMALLINT	Whole numbers, from -32,767 to +32,767

Synonyms for the names of some data types, such as REAL for SMALLFLOAT, are supported to conform to the ANSI/ISO standard for SQL. The *arithmetic operators* of 4GL can manipulate and return number values, as the following table shows.

Operator Symbols	Names of Operators
+, -	Unary plus and unary minus operators
+, -	Binary addition and binary subtraction operators
*, /, **	Multiplication, division, and exponentiation operators
MOD	Modulus operator

Differences Between DECIMAL and MONEY Data Types

The internal representations of fixed-point DECIMAL and MONEY data types are identical. The only difference between these two data types is that when 4GL displays a MONEY value to the screen or a report, it formats the number as currency.

Numeric Precision

Some 4GL number data types are implemented as standard C data types.

Data Type	Description
FLOAT	Same as C **double**
SMALLFLOAT	Same as C **float**
INTEGER	Same as C long
SMALLINT	Same as C short

A library of C functions for working with DECIMAL and MONEY data types is included with both 4GL and INFORMIX-ESQL/C.

Time Data Types

4GL supports the following three data types for keeping track of time.

Data Type	Kind of Data Represented
DATE	Points in time, specified as calendar dates
DATETIME	Points in time stored with a specified precision, as calendar dates, times of day, or both
INTERVAL	Spans of time stored with a specified precision, either in years and months, or else in days or smaller units of time

A DATE value is stored internally as a count of days before or after midnight, 31 December 1899; that is, January 1, 1900, would be stored as 1. When it displays a DATE value to the screen or a report, 4GL formats it according to directions in the **DBDATE** environment variable, so your user can tailor the display of date values to match local conventions. You can also employ the USING operator to format DATE values.

GLS

By using the GLS features of 4GL, DATE values (and DATETIME values) can be displayed according to the cultural conventions of the current locale. You can also use the **DBCENTURY** environment variable to choose the present, previous, or next century. For details, see *INFORMIX-4GL Reference* and the *Informix Guide to GLS Functionality*. ♦

You can mix DATE values with integers when doing arithmetic (for example, subtracting 7 to get a date for the same day of the previous week). The difference between two DATE values, however, is an INTEGER value. You must use the UNITS operator to convert the difference to an INTERVAL value.

A DATETIME value can have more or less precision than a DATE value: it can specify a date, a time, or both, and can be exact to a fraction of a second.

An INTERVAL value represents a span of time, not a particular moment in time. For example, "three hours" is an interval; "three o'clock" is a point in time. You can do arithmetic that mixes DATE, DATETIME, and INTERVAL values, yielding new DATETIME or INTERVAL values.

4GL does not support automatic conversion between INTERVAL values with YEAR or MONTH time units and INTERVAL values of smaller precision.

Some built-in operators accept or return DATE values. Built-in functions and operators are described in *INFORMIX-4GL Reference*.

Built-In Operator	Purpose
CURRENT	Returns the current date and time as a DATETIME value
DATE(*expr*)	Converts an integer or string value to DATE or DATETIME
DAY(*date-expr*)	Returns the day of the month from a DATE or DATETIME value
EXTEND(*date*, *qual*)	Changes the precision of a DATE or DATETIME value, returning a DATETIME value
MDY(*m*, *d*, *y*)	Composes a DATE value from integer values for *month*, *day*, *year*
MONTH(*date-expr*)	Returns the month from a DATE or DATETIME value
TIME	Returns the current time of day as a character string
TODAY	Returns the current date as a DATE value
int-expr UNITS *qual*	Converts an integer to an INTERVAL value of specified precision
WEEKDAY(*date-expr*)	Returns the day of the week from a DATE or DATETIME value
YEAR(*date-expr*)	Returns the year from a DATE or DATETIME value

These operators are identical in name and use to functions available in SQL statements. Used in SQL, they apply to values in the database. You can also use them in other 4GL statements, applying them to program variables.

DATE, DATETIME, and INTERVAL data types are discussed in detailed in *INFORMIX-4GL Reference*.

Character Data Types

4GL supports several data types to represent strings of bytes in memory, as the following table shows.

Data Type	Kind of Data Represented
CHAR(*length*) CHARACTER(*length*)	Character strings of fixed length, up to 32,767 bytes
NCHAR(*size*)	Character strings of fixed length, up to 32,767 bytes
VARCHAR(*length*)	Character strings of varying length, up to 255 bytes
NVARCHAR(*size*)	Character strings of varying length, up to 255 bytes
TEXT	Character strings up to 2^{31} bytes

The most important of these is CHAR or its synonym CHARACTER, which is the default 4GL data type. TEXT can store strings, but it is classified as *large* (rather than *character*) data type, because 4GL manipulates TEXT values in a different way from CHAR or VARCHAR values.

NCHAR and NVARCHAR are both locale-sensitive character data types,and are interpreted and converted to CHAR or VARCHAR data types. When NCHAR and NVARCHAR data types are sent from the server to the client, they are converted to CHAR or VARCHAR data types, respectively. When CHAR or VARCHAR data types are sent from the client to the server, they are converted to NCHAR and NVARCHAR data types, respectively.

The following operators are among built-in operators of 4GL that can accept or return CHAR or VARCHAR values.

Operator	Purpose
variable [*start*, *end*]	Specifies a substring from a CHAR or VARCHAR value
ASCII *int-expr*	Returns a specific ASCII character as a CHAR(1) value

(1 of 2)

Operator	Purpose
char-expr CLIPPED	Returns a character value, without trailing blank spaces
LENGTH(*char-expr*)	Returns the length bytes, disregarding trailing blank spaces
value USING "*mask*"	Returns the string representation of a value, formatted to fit a specified pattern

(2 of 2)

Within a DISPLAY statement (that displays values to the screen) or a PRINT statement (that sends values to a report), you can use the COLUMN operator to set the beginning position of the value of the current line of output.

CHAR and VARCHAR Compared

In the database, the important difference between CHAR and VARCHAR columns is that only the actual length of a VARCHAR value is stored to disk, while a CHAR value always occupies its full declared size.

This difference is not important for program variables because 4GL always allocates enough memory to hold the declared size of a VARCHAR value. For example, VARCHAR(25) always occupies 26 bytes of memory, even if you store a 1-byte value in it. (The 26th byte stores the end-of-data symbol.) You cannot economize on program memory by using VARCHAR in place of CHAR.

In a program, the difference between the two data types is that when you reference a CHAR variable, you always get its full declared size in bytes, filled out with trailing blank spaces if necessary. The CLIPPED operator can be used to drop any trailing whitespace. When you refer to a VARCHAR variable, you get only its current data contents, without any padding. For this reason, the CLIPPED operator is less often needed with VARCHAR variables.

Large Data Types

IDS

The data types BYTE and TEXT are collectively known as *large* or *blob* (binary large object) data types. They represent strings of data that can be of any length. The only difference between a BYTE and a TEXT data type is that a BYTE value can contain any combination of binary values, while TEXT data items contain any number of printable characters. Informix Dynamic Server supports blob data types. ♦

The LOCATE statement of 4GL must specify whether the contents of a blob variable are to be held in memory or in a file. You can change this location dynamically, as the program runs. Typical uses for blob variables are as follows:

- To fetch blob values from the database and store them in program variables of the same data type

  When the receiving variable is located in a file, this is effectively a disk-to-disk copy from the database to the file.
- To store blob values from program variables into the database

  When the source variable is located in a file, this is effectively a disk-to-disk copy from the file to the database.
- To view or modify a blob value, using some external program that understands the contents of the blob

The following example uses a blob variable with an external editor:

1. Retrieve a blob consisting of a graphic from an Informix database and store it as a blob variable located in a disk file.
2. Call a paint-type editing program, make changes in the graphic, save the changes, and exit from the editor.
3. Reload the modified graphic in the database.

Editing graphics generally requires a GUI.

Variables and Data Structures

A program variable is a named location in memory where a value can be stored. There are five things to know about any variable:

- **Data type.** What kind of data value can it hold?

  A given variable can store data in a specific format (or range of formats) that is determined by the declared data type of the variable.

- **Structure.** Does it contain only a single value or is it a collection of several values? If it is an aggregate, how can the individual simple values be accessed?

  For example, an array is a vector of values of the same data type. You access a single value by writing a subscript, as in **custNumList[15]**.

- **Scope of reference.** In what parts of the program can 4GL or the database server recognize its name?

  There are three possibilities, which are discussed in greater detail later in this chapter; they are local, module, and global.

- **Visibility.** Within its scope of reference, is there any conflict with other program variables that have the same name?

  INFORMIX-4GL Reference discusses the visibility rules for 4GL variables whose names are not unique within their scope.

- **Time of allocation.** When during the execution of the program is the variable created and initialized?

  There are two possible times: memory for the variable can be allocated at compile time, as part of the executable program file, or else at runtime, dynamically, while the program is executing.

Declaring the Data Type

The characteristics of a variable are decided when you declare a variable. To declare a variable is to write a statement that tells 4GL about the variable. Use the keyword DEFINE for this. The following DEFINE statement declares four simple variables:

```
DEFINE
   j, custNum INTEGER ,
   callDate DATETIME YEAR TO SECOND ,
   sorryMsg CHAR(40)
```

State the data type of a variable after its name. In the preceding declaration, variables **j** and **custNum** are INTEGER; the data type of **callDate** is DATETIME YEAR TO SECOND, and the data type of **sorryMsg** is CHAR(40). Any of the simple data types that were listed earlier in this chapter can be used.

You can also use the LIKE keyword to specify that the data type of a variable is the same as the data type of a specified column in an Informix database:

```
DEFINE custFname LIKE customer.fname
```

The advantage of using LIKE in this way for *indirect typing* is that if the database schema changes, you need only recompile your program to make sure that the data types of your variables match those in the database.

Creating Structured Data Types

Until now, only simple and blob data types have been considered. 4GL also supports data types that contain many individual values. Such data types are considered to have a structure. You specify the structure of a variable by stating that it is an ARRAY or a RECORD in the DEFINE statement.

Declaring an Array

An *array* is a set of elements that are all of the same data type, ordered along one or more dimensions. The following two examples are array declarations:

```
DEFINE custNumTab ARRAY [2000] OF LIKE customer.customer_num

DEFINE custByProd ARRAY [100, 25] OF MONEY(12)
```

The number of elements is specified in brackets. The example shows a single dimension array 2,000 elements long and a 100 by 25 (100*x*25) two-dimensional array. Three-dimensional arrays can also be created.

All elements of an array have the same data type. This can be one of the simple or large data types, or it can be a record. You specify the data type of the array in the OF clause that follows the name of the array.

You access an element of an array by naming the array with a subscript expression in brackets, as in **custNumTab[175]** or **custByProd[j,prodNum]**.

Declaring a Record

A record is a collection of variables, each with its own data type and name. You put them in a record so you can treat them as a group, as follows:

```
DEFINE person RECORD
   honorific VARCHAR(40) , -- e.g. "Excellency"
   initial CHAR(1) ,
   famName CHAR(30)
END RECORD
```

You access a member of a record by writing the name of the record, a dot (known as *dot notation*), and the name of the member. For example, **person.initial** is the second member of the **person** record.

You can declare a record that has one member for each column in a database table. The names of the members and their data types are derived from the database. The only exception is that SERIAL data types are converted to INTEGER data types. In the simplest form, you write:

```
RECORD LIKE tablename.*
```

As in the following:

```
DEFINE custRec RECORD LIKE customer.*
```

The statement creates a record named **custRec** having one member for each column of the **customer** table. Each record member has the name and the data type of the corresponding column in the table.

You can augment table columns with other members. The following clause retrieves the names of columns and their types:

```
LIKE tablename.*
```

This, in effect, defines a RECORD:

```
DEFINE custPlus RECORD
      row INTEGER ,
      customer RECORD LIKE customer.* ,
      balanceDue DECIMAL(8,2)
   END RECORD
```

The preceding statement creates a record named **custPlus** having a member for each column of the **customer** table and two additional members. The member **custPlus.row** is an integer. The member **custPlus.balanceDue** is a decimal number. In this case, where the LIKE clause only generates some of the members of the record, you must use an END RECORD clause to finish the record definition.

Because **custPlus.customer** is a record within the record, a reference to the **lname** member of the record is specified as:

```
custplus.customer.lname
```

Declaring the Scope of a Variable

You specify the scope of a variable within its source module by where in the source module you write the DEFINE statement:

- If it is within a function, the scope is local to that function. The variable can only be referenced while the function is executing. (Functions are described in "Functions and Calls" on page 8-34.)
- If it is at the top of the source module and outside any MAIN, REPORT, or FUNCTION statement, the variable is considered a module variable. Its name can be used anywhere from that point to the end of the source module.
- If it is in a GLOBALS statement in a module separate from any other statements, the variable is available in that module (and in any other module that includes the GLOBALS "***filename***".**4gl** statement that defines that variable).

The context of the DEFINE statement also determines the following characteristics of a variable:

- **Scope of reference**. This portion of the source code is where the 4GL compiler recognizes the variable name.
- **Time of allocation.** 4GL allocates memory for the variable at time of allocation.

4GL also supports recursion (a function calling itself). Each separate call to a function allocates its own copy of local variables.

Scope of Reference

The context of a DEFINE statement determines the scope of reference of the name of a variable, sometimes more briefly referred to as its scope. During compile time, the scope of reference is that portion of the source code in which the 4GL compiler can recognize the name of the variable. Outside its scope, the name is unknown or might reference a different variable.

Local Scope: Within a Program Block

Within the definition of a function or report, or within a MAIN program block, DEFINE declares local variables. The scope of reference of a local variable is restricted to the same program block in which it is declared. The DEFINE statements that declare local variables must precede any executable statement within the same program block.

Modular Scope: Within a Source Module

Outside any FUNCTION, REPORT, or MAIN program block, DEFINE declares module variables. The scope of reference of a module variable is the entire source module. Module variable definitions must appear at the beginning of the source module, before the first program block.

Global Scope: Within Several Modules

The GLOBALS "***filename***".**4gl** statement can extend the scope of module variables that you declare outside any FUNCTION, REPORT, or MAIN program block and within a GLOBALS ... END GLOBALS statement. (This can contain DATABASE and DEFINE statements.) The following scope of reference is for a variable that you declare in this way:

- The module where the GLOBALS ... END GLOBALS statement appears
- Any other modules that include a GLOBALS "***filename***".**4gl** statement, where "***filename***".**4gl** is the module that contains the GLOBALS ... END GLOBALS statement that declares the variable.

Both the GLOBALS ... END GLOBALS and the GLOBALS "***filename***".**4gl** statements must appear at the beginning of a source module, before the first program block, and before any executable statements.

Time of Memory Allocation

The context of the DEFINE statement also determines when the memory for the variable is allocated. How and when 4GL handles memory allocation for a variable depends on the scope of reference of the variable.

Allocation of Local Variables

Storage for local variables is allocated *dynamically*. 4GL allocates this storage when control of execution passes to the program block (a FUNCTION, MAIN, or REPORT statement) that contains the declaration of the local variable.

Local variables are initialized in the same order in which their names appear in their program block.

Allocation of Module Variables

Storage for module variables is allocated statically, in the executable image of the program. 4GL allocates this storage when the program begins execution and deallocates it when the program terminates.

Module variables are initialized in the same order in which their names (and the names of any global variables) appear in the source module.

Allocation of Global Variables

Storage for global variables is also allocated statically. 4GL allocates this storage when the program begins execution and deallocates it when the program terminates. To be able to handle references to a global variable across several source modules, however, 4GL makes a distinction between variable declaration and variable definition:

- *Variable declaration* tells the 4GL compiler the name and the data type of the variables so that the compiler can verify references to this variable in a given source code module.
- *Variable definition* allocates the memory for the global variable. For global variables, memory is allocated statically, as part of the program image.

The GLOBALS ... END GLOBALS statement defines the global variables. It also declares them so the compiler can verify references to a global variable in the same module where it is defined.

To make a global variable visible in other modules of the program, you only need to declare these variables. You do not need to define them because memory only needs to be allocated once and is done so with the GLOBALS ... END GLOBALS statement.

To declare global variables in other modules, you must:

- put the GLOBALS ... END GLOBALS statement in a separate source file.
- at the top of each source file that references a global variable, put the GLOBALS "***filename***"**.4gl** statement, where "***filename***"**.4gl** is the name of a file containing the GLOBALS ... END GLOBALS statement.

The following example declares global variables in the **globs.4gl** source file:

```
GLOBALS
   DEFINE a,b,c INT ,
   x,y,z CHAR(10)
END GLOBALS
```

To reference these variables in other source modules, you would put the following statement at the top of each source module using a global variable:

```
GLOBALS "globs.4gl"
```

The following section discusses global variable declarations in detail.

Using Global Variables

The GLOBALS statement defines and declares a global variable. All references to a global variable refer to the same location in memory. This statement has two forms:

- If you use the LIKE keyword in any DEFINE statement, you must identify the database that contains the referenced database columns. You can do this in either of the following two ways:
  - Precede the GLOBALS ... END GLOBALS statement with a DATABASE statement specifying the database containing the referenced columns.
  - Qualify each referenced column with its table name.
- If a global variable defined within the GLOBALS ... END GLOBALS block has the same name as a local variable, then the local identifier takes precedence within its scope of reference. A module variable in the same source module *cannot* have the same name as a global variable.

The following program segment defines a variable like the **customer** table of the **stores7** demonstration database:

```
DATABASE stores7
GLOBALS
   DEFINE p_customer RECORD LIKE customer.*
END GLOBALS
```

Global Variable Declaration

The globals file contains global variable definitions and must have a **.4gl** file extension; for example, this file can be named **globals.4gl**. You compile this file as part of your multiple-module program. This globals file should contain *only* GLOBALS ... END GLOBALS statements. It cannot contain 4GL executable statements. (DATABASE and DEFINE statements are valid, however, because these are not executable.) Different source files can reference different globals files.

Using GLOBALS Within a Single Module

The following program fragment defines a global record, a global array, and a simple global variable that are referenced by code in the same source module:

```
DATABASE stores7
GLOBALS
DEFINE p_customer RECORD LIKE lbraz.customer.* ,
   p_state ARRAY[50] OF RECORD LIKE state.* ,
   state_cnt SMALLINT ,
   arraysize SMALLINT
END GLOBALS

MAIN

   LET arraysize = 50
 ...
END MAIN

FUNCTION get_states()
 ...
FOREACH c_state INTO p_state[state_cnt].*
  LET state_cnt = state_cnt + 1
  IF state_cnt > arraysize THEN
    EXIT FOREACH
  END IF
END FOREACH
 ...
END FUNCTION

FUNCTION state_help()
DEFINE idx SMALLINT ...
CALL SET_COUNT(state_cnt)
DISPLAY ARRAY p_state TO s_state.*
LET idx = ARR_CURR()
LET p_customer.state = p_state[idx].code
DISPLAY BY NAME p_customer.state
 ...
END FUNCTION
```

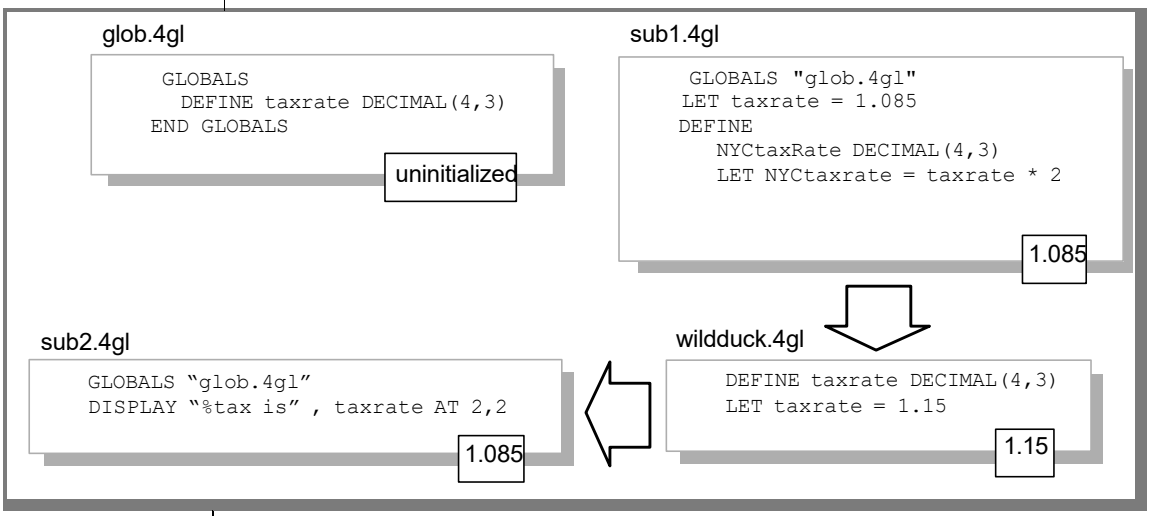
The 4GL compiler will generate an error if you defined a module variable with a name of **arraysize**, **p_customer**, **p_state**, or **state_cnt** in the same module containing the GLOBALS statement.

However, used in this context, the GLOBALS statement simply defines variables with module scope. For other modules to be able to access the variables, they must use the GLOBALS "***filename***"**.4gl** statement, and to do so, the GLOBALS ... END GLOBALS statement must appear in a separate source file.

Global Versus Module Scope

When a variable is declared in a GLOBALS statement, the variable is still a module variable as far as the current source module is concerned, but you can use it anywhere in your program. When you do this, the source modules share the single copy of the variable. If the value is changed in one module, it is changed throughout the program. Figure 8-1 shows that the numbers in the boxes indicate the value of **taxRate** at the specified point in the program.

Figure 8-1
The Difference Between Global and Local Variable



In Figure 8-1, the source module **glob.4gl** declares a variable **taxrate** as a global variable. The current value of the global variable is available to any function in any module that references the global source module.

Another source module, **sub1.4gl**, references the GLOBALS file **glob.4gl**, and assigns a value of `1.085` to **taxrate**.

A third module, **wildduck.4gl**, defines a module variable named **taxrate** and assigns it a value. Because there is no reference to the globals file **glob.4gl**, this module-level assignment is permitted. No change in the value of **taxRate** in **wildDuck.4gl** will affect the current value of the global variable **taxRate**.

Finally, a routine in a fourth module, **sub2.4gl**, displays the current value of the global variable **taxrate**, `1.085`.

Initializing Variables

You initialize variables using the LET statement:

```
DEFINE Pi, TwoPi SMALLFLOAT
   LET Pi = 3.1415926
   LET TwoPi = 2*Pi
```

The declaration of a variable must precede any executable statement in the same program block.

4GL programmers often initialize global variables at a single point in the program, such as immediately after the last DEFINE or DATABASE statement in the MAIN program block.

Expressions and Values

A *value* is a specific item of information. An *expression* specifies how a value is to be calculated. You can use expressions at many points in a 4GL program, including the following contexts:

- Function calls, to specify the arguments to functions
- Decision and looping statements, to control program flow
- Reports, to specify the values to print in the report
- SQL statements, to specify values to be inserted into the database
- LET statements, to specify values to be assigned to variables

Every value has a data type, one of the simple or large data types that were listed earlier in this chapter. The 4GL compiler knows the data type of every component of an expression, so it can tell the data type of the returned value that the expression specifies.

For a detailed discussion of 4GL and SQL expressions, see *INFORMIX-4GL Reference*.

Literal Values

The simplest kind of expression is a *literal* value, that is, a literal representation of a number, a date, a character string, or some other kind of data.

Some literal numbers are: -7, 3.1415926, 1e-8. 4GL assumes any literal number with a fractional component has the data type DECIMAL(32), the most precise number data type available.

Enclose literal character values in quotes: `"Record inserted."` You can use a single quote (`'`) or double quotes (`"`) to delimit a character literal, but both delimiters of the same value must be the same. In single-byte locales, 4GL assumes that any literal character value has the data type CHAR(*n*), where *n* is the number of characters between the quotes.

You can also write literal values of DATE, DATETIME, and INTERVAL data types. For details of these data types, see *INFORMIX-4GL Reference*.

Values from Variables

The next simplest kind of expression is the name of a simple variable or the name of one element of a structured variable. Suppose that the following names are declared and a value assigned to **pi**:

```
DEFINE
   j, prodNum INTEGER
   pi  SMALLFLOAT
   LET pi = 3.1415926
```

Following these statements, you can use the name **pi** as an expression meaning 3.1415926 and having the data type SMALLFLOAT. The names **j** and **prodNum** are expressions meaning, "the value that was last stored in this variable," and have a data type of INTEGER.

References to a component of a structured variable are also expressions. The ARRAY element `custByProd[j,prod_num]` is an expression meaning, "the value that was last stored in the subscripted element of this array." For the expression to be valid, both **j** and **prod_num** must have values that are positive integers within the declared dimensions of the **custByProd** ARRAY variable.

Values from Function Calls

A function can return one or more values. In fact, this is the most common reason to write a function: to calculate and return a value. (Functions are discussed in more detail in "Functions and Calls" on page 8-34.) Any function that returns a single value can be used as an expression.

You write a call to a function by writing the name of the function followed by its arguments enclosed in parentheses. See also "Data Type Conversion" on page 8-29.

The use of functions returning more than one value is discussed in "Working with Multiple Values" on page 8-37.

Numeric Expressions

You can write expressions that calculate numeric values. You can use the arithmetic operators to combine numeric literals and the names of numeric constants, variables, and functions to calculate new values. The following statement from an example in the preceding chapter contains several numeric expressions:

```
LET fahrDeg = (9*centDeg)/5 + 32
```

In this statement, 9, 5, and 32 are literals, and **centDeg** is the name of a variable. The expression `9*centDeg` tells 4GL to produce a new value by multiplying two values, and `(9*centDeg)/5+32` tells it how to produce another new value by dividing and then adding.

As previously noted, 4GL carries intermediate results of calculations on numbers with fractional components in its most precise data type, DECIMAL(32). This helps prevent many errors due to rounding and truncation, and reduces the chance that an intermediate result will overflow. However, it is still possible for a badly planned calculation to cause overflow, round-off, or truncation errors. As with any computer language, you should think through the precision requirements of any critical calculation.

Integer and numeric expressions are described in *INFORMIX-4GL Reference*.

Boolean Expressions

In 4GL, the result of comparing two values by using any Boolean or relational operator is an integer value of `1` (if the result of the comparison is TRUE) or `0` (if the result is FALSE). Suppose the following variables have been declared and initialized:

```
DEFINE maxRow, rowNum INTEGER
```

The expression `rowNum==maxRow` is a Boolean comparison; that is, it is a Boolean expression with a value of either `1` or `0`.

Tip: *You can write an equality comparison using either a single equals sign or a double one. If you have used the C language, you might prefer to write ==; if your experience has been with other languages you might prefer to use just one.*

Other Boolean operators include <> or != for "not-equals," and LIKE or MATCHES for character pattern matching. Two especially important relational operators are IS NULL and IS NOT NULL. You can use these to test for the presence of a null value in a variable. (For more information on the use of null values, see "Null Values" on page 8-27 and *INFORMIX-4GL Reference*.) Most other Boolean operators of 4GL return FALSE if any operand is NULL.

All these relational tests return `1` to mean "true" or `0` to mean "false." The names TRUE and FALSE are predefined 4GL constants with those values.

You can combine numeric values with the Boolean operators AND, OR, and NOT. Often you use these to combine the results of relational expressions, writing expressions such as `keepOn="Y" AND rowNum < maxRows`. That expression means:

- Take the value of the comparison `keepOn="Y"` (which is `1` or `0`).
- Take the value of the comparison `rowNum<maxRows` (`1` or `0`).
- Combine those two values using AND to produce a new value.

Usually you write Boolean expressions as part of IF statements and other decision-making statements (for some examples, see "Decisions and Loops" on page 8-31). However, a comparison is simply a numeric expression. You can store its value in a variable, pass it as a function argument, or use it any other way that an expression can be used.

For more on relational and Boolean expressions, see *INFORMIX-4GL Reference*.

Character Expressions

You can express a character value literally (as a quoted string) or as an expression that returns a character string. In many contexts, 4GL can convert a value of any simple data type automatically to a string that a CHAR or VARCHAR variable of sufficient declared length can store.

The following expressions return a character value:

- The name of an initialized CHAR, VARCHAR, NCHAR, or NVARCHAR variable
- A call to a function that returns a CHAR, VARCHAR, NCHAR, or NVARCHAR value
- A *substring* of a literal, a variable, or the returned value from a function returning a CHAR, VARCHAR, NCHAR, or NVARCHAR value

A substring is written as one or two numbers in square brackets. The first is the position of the first character to extract, and the second is the position of the last. Suppose the following variable exists:

```
DEFINE delConfirm CHAR(11)
LET delConfirm = "Row deleted"
```

Now you can write `delConfirm[1,3]` as an expression with the value `Row` and the expression `delConfirm[5,8]` is the value `dele`. The expression `delConfirm[4]` or `delConfirm[4,4]` is a single-space character.

It should be noted that the substring expression uses the same notation as the subscript to an array. The following example shows an array of character values:

```
DEFINE companies ARRAY[75] OF CHAR(15)
```

The expression `companies[61]` produces the character value from the 61st element of the array. The expression `companies[1,7]` would cause an error at compile time because the array **companies** does not have two dimensions. However, the expression `companies[61][1,7]` accesses the 61st element and then extracts the first through the seventh letters of that value.

Null Values

For every SQL data type, the Informix database servers define a null value. A value of NULL in a database column means *do not know*, or *not applicable*, or *unknown*. Because null values can be read from the database into program variables, 4GL also supports a null value for every data type. The keyword NULL stands for this *unknown* value. A variable of any data type can contain NULL, and a function can return a variable with a value of NULL. If a null value appears in a character expression, 4GL substitutes blank characters for the value.

Null Values in Arithmetic

If you do arithmetic that combines a number with NULL, the result is NULL. Adding 1 to *unknown* must result in *unknown*. If any value in an arithmetic expression is NULL, then the expression returns the value NULL.

Null Values in Comparisons

If you compare a null value to anything else, the result is NULL. Is "A" equal to *unknown*? The only answer can be *unknown*. Thus every comparison expression has not two but three possible results: TRUE, FALSE, and *unknown*, or NULL.

It is worth noting that the decision-making statements such as IF and WHILE do not distinguish this third result. They treat NULL the same as FALSE. This can cause problems when you test values that might be NULL. That is why the operators IS NULL and IS NOT NULL are provided; they allow you to detect null values and respond according to the requirements of your application.

Null Values in Boolean Expressions

The Boolean operators AND and OR give special treatment to NULL. In the case of OR, when one of its arguments is TRUE, its result is TRUE no matter what the other argument might be. But if one of its arguments is FALSE and the other is NULL, then OR must return NULL, because 4GL does not know whether its result should be TRUE or FALSE.

Assignment and Data Conversion

In creating a useful 4GL program, you write expressions to describe values and then you do something with the values. Sometimes you display them or pass them as arguments to functions. Most often, you *assign* a value for a variable; that is, you tell 4GL to store the value in the memory reserved for the variable.

```
DEFINE fahrDeg, centDeg DECIMAL(4,2)
LET centDeg = 10.28
LET fahrDeg = (9*centDeg)/5 + 32
```

The expression is `(9*centDeg)/5+32`. The LET statement causes 4GL to calculate the value of the expression and to store that value in the memory reserved for the variable **fahrDeg**.

You can assign a value to a variable in the following ways:

- Use the LET statement. This assignment is the most common.
- Use the CALL … RETURNING statement to call a function and store the value it returns:

  ```
  CALL tempConvert(32) RETURNING fahrDeg
  ```

- Use the INTO clause of an SQL statement to get a value from the database and assign it to a variable:

  ```
  SELECT COUNT(*) INTO maxRow FROM stock
  ```

  Other SQL statements that support INTO include SELECT, FETCH, and FOREACH.
- Use the INITIALIZE statement to set a variable, or all members of a record, to NULL or to other values:

  ```
  INITIALIZE custRow.* TO NULL
  ```

- Use PROMPT to accept values that the user entered from the keyboard:

  ```
  PROMPT "Enter temp to convert: " FOR centDeg
  ```

- Use INPUT or INPUT ARRAY to get values from fields of a form and put them in variables:

  ```
  INPUT custRec.* FROM customer.*
  ```

- Use CONSTRUCT to get a query by example expression from a form and put it in a variable:

  ```
  CONSTRUCT BY NAME whereClause ON customer.*
  ```

Data Type Conversion

All the preceding methods of assignment perform the same action: they store values in variables. Each performs automatic *data conversion* when is necessary and possible. Data conversion is necessary when the data type of the passed value is different from the data type of the variable (or the formal argument of a 4GL function or report) that receives it.

As previously noted, 4GL has liberal rules for data conversion. It will attempt to convert a value of almost any data type to match the data type of the receiving variable. For a table that summarizes the rules and shows data type incompatibilities, see *INFORMIX-4GL Reference*. The pairs of data types that are identified in that table as supporting automatic data type conversion are called *compatible* 4GL data types.

```
DEFINE num DECIMAL(8,6)
DEFINE  chr  CHAR(8)
LET num = 2.18781
LET chr = num
```

The second assignment statement asks 4GL to initialize **chr**, a character variable, from the value of **num**, a numeric variable. In other words, this statement asks 4GL to convert the value in **num** to a character. It does that using the same rules it would use when displaying the number, in this case producing the string `2.187810` (with all six declared decimal places of the fractional part filled in).

```
LET num = chr[1,3]
```

Given the initialization of **chr** to the string `2.187810`, the expression `chr[1,3]` returns the characters `2.1`. Because the receiving variable is of data type `DECIMAL(8,6)`, 4GL converts the characters into the number value `2.100000` and assigns that value to **num**.

For systems or locales that define some symbol other than period (.) to represent the decimal separator, 4GL would use that symbol to replace the period in the converted string value.

Conversion Errors

Some data type conversions cannot be done. When 4GL can recognize at compile time that a particular conversion is illegal, it returns a compiler error.

4GL can convert BYTE to TEXT, and TEXT to BYTE, but it does not attempt to convert these to other data types because 4GL does not know enough about the internal structure of the values of these large data types to convert them.

Some conversions might prove to be impossible only at execution time. Then the error will be detected while the program is running. For example, the following program tries to assign a CHAR value to a SMALLINT variable:

```
DATABASE stores7
DEFINE a, b SMALLINT, c,d CHAR(10)

MAIN
LET c="apple"
DISPLAY "This is c ", c AT 3,3
SLEEP 2

LET a=c
DISPLAY "This is a ", a AT 5,5
SLEEP 4

END MAIN
```

For more information, see "Runtime Errors" on page 12-4.

Decisions and Loops

The statements you use to control the sequence of execution are similar to those in other languages you might have used. You can find details in *INFORMIX-4GL Reference* for each of the following statements.

The IF…THEN…ELSE statement tests for Boolean (yes/no) conditions. You write the test as a conditional statement, usually a relational comparison or a Boolean combination of relational comparison. If the value of the expression is 1 (TRUE), the THEN statements are executed. When the expression evaluates to 0 (FALSE) or is NULL, the ELSE statements are executed. For example:

```
IF promptAnswer MATCHES "[yY]" THEN
   DELETE WHERE CURRENT OF custCursor
ELSE
   DISPLAY "Row not deleted at your request"
END IF
```

The CASE(*expr*) statement implements multiple branches. This statement has two forms. The first form is a simple form that compares one expression for equality against a list of possible values. For example:

```
CASE (WEEKDAY(ship_date))
WHEN 0-- Sunday
   DISPLAY "Will ship by noon Monday"
WHEN 5-- Friday
   DISPLAY "Will ship by noon Saturday"
WHEN 6-- Saturday already
   DISPLAY "Will ship by noon Monday"
OTHERWISE
   IF DATETIME (12) HOUR TO HOUR <
         EXTEND(CURRENT,HOUR TO HOUR) THENDISPLAY "Will ship by 5
today"ELSE -- past noon
      DISPLAY "Will ship by noon tomorrow"
   END IF
END CASE
```

The second form of CASE is effectively a list of *else-if* tests. No expression follows the keyword CASE, but a complete Boolean expression (instead of a comparison value) follows each WHEN keyword. For example:

```
MAIN
DEFINE promptAnswer CHAR(10)
PROMPT "Delete current row? " FOR promptAnswer
CASE
   WHEN promptAnswer MATCHES "[Yy]"
      DISPLAY "Row will be deleted." AT 2,2
```

```
      WHEN promptAnswer MATCHES "[Nn]"
         DISPLAY "Row not deleted." AT 2,2
      WHEN promptAnswer MATCHES ("Maybe")
         DISPLAY "Please make a decision." AT 2,2
      OTHERWISE
         DISPLAY "Please read the instructions again." AT 2,2
   END CASE
   SLEEP 5
   END MAIN
```

The WHILE statement provides for generalized looping. You can use the EXIT statement to break out of a loop early. For example:

```
LET j=1
WHILE manyObj[j] IS NOT NULL
   LET j = j + 1
   IF j > maxArraySize THEN -- off the end of the array
      LET j = maxArraySize
      EXIT WHILE
   END IF
END WHILE
DISPLAY "Array contains ",j," elements."
```

The FOR statement provides counting loops. For example:

```
FOR j = 1 TO maxArraySize
   IF manyObj[j] IS NOT NULL THEN
      LET j = j-1
      EXIT FOR
   END IF
END FOR
DISPLAY "Array contains ",j," elements."
```

An additional loop, the FOREACH loop, is discussed in "Row-by-Row SQL" on page 9-5.

Decisions Based on NULL

If a Boolean comparison evaluates to NULL (see "Null Values" on page 8-27), it will have the same effect as FALSE:

- IF NULL … always executes the ELSE statements (if any).
- CASE (NULL) … always executes the OTHERWISE statements (if any).
- WHILE NULL … does not execute its loop statements at all.

Using a NULL value as either the starting or the ending number in a FOR loop results in an endless loop. The FOR loop ends when the control variable equals the upper limit, but a NULL value cannot equal anything; hence the loop never ends.

Functions and Calls

A function is a named block of executable code. The function is your primary tool for achieving a readable, modular program.

Function Definition

You *define* a function when you specify the executable statements it contains. Figure 8-2 shows a definition for a simple function.

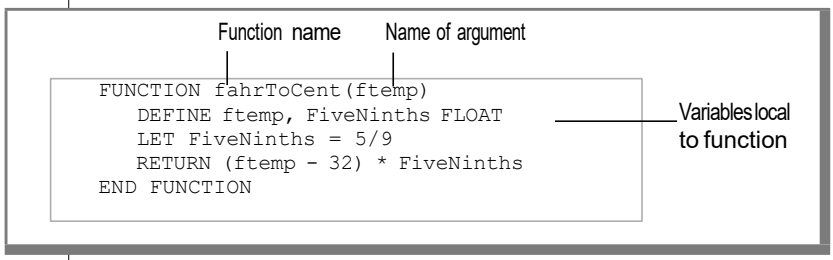


Figure 8-2
Simple Function Definition

This example shows the important parts of a function definition. It contains the following statements:

- A FUNCTION statement that defines the following aspects:
  - The name of the function (**fahrToCent** in the example)
  - How many arguments it takes (just one in the example)
- A function *program block*, statements between the FUNCTION and END FUNCTION:
  - DEFINE statements must appear before other kinds of statements.
  - Executable statements do the work of the function. In the example there is only one, a RETURN statement.

Variables declared in the program block are local to the function. The variable named **FiveNinths** is local to this function; it is not available outside the function, although other **FiveNinths** variables can be declared at the module level, locally in other functions, or globally.

When the 4GL compiler processes a function definition, it generates the executable code of the function. Once defined, a function is available to any 4GL module in your program. That is, the scope of reference of the name of a function (or of a report) is global. The name of a function or report must not be the same as the name of one of its formal arguments, however, or else the local name (the argument) will occlude the visibility of the global name.

Invoking Functions

You cause a function to be executed by *calling* it. The two ways to call a function are:

- In an expression
- Through the CALL statement

When a function returns a single value, you can call it as part of an expression. The **fahrToCent( )** function described previously returns a single value, so it can be called in an expression that expects the data type of the returned value:

```
LET tmp_range = fahrToCent(maxTemp) - fahrToCent(minTemp)
```

This statement contains two calls to the function **fahrToCent( )**. The statement subtracts one of these values from the other, and assigns the result to the variable **tmp_range** of data type FLOAT.

When a function returns no values or multiple values, you must use the CALL statement. Functions that return one value can be called in this way also.

```
CALL mergeFiles()
CALL fahrToCent(currTemp) RETURNING cTemp
```

The useful ability to return more than one value from a function is considered further in "Working with Multiple Values" on page 8-37.

Arguments and Local Variables

The arguments you provide when a function is called are, in effect, local variables of the function. That is, these names (such as the name **ftemp** in the **fahrToCent( )** function) represent values that are passed to the function when it is called. They are local to the function.

The following events happen when a function is called:

- Variables local to the function are allocated in memory, including the those that represent the arguments. In the function **fahrToCent( )**, local variables are **fiveNinths** and **ftemp**, its argument.
- Each expression in the argument of the function call is evaluated. In the following call, the expression `targetTemp + 20` is evaluated.

```
LET limitTemp = fahrToCent(targetTemp + 20)
```

- Each argument value is assigned to its argument variable, as described earlier ("Assignment and Data Conversion" on page 8-28). When an argument value has a different data type from the argument variable, 4GL attempts to convert it, as in any assignment.
- The statements of the function definition are executed.
- The local variables, including the argument variables, are discarded and their memory is reclaimed.

The key point is that the expressions you write in the call to a function are assignments to local variables of the function. Knowing this, you can answer some common questions:

- Does a value passed to a function require a specific data type?

  No, because the value is assigned to the argument variable, and 4GL will attempt to convert it to the specific type. It is sufficient that the expression in the call and the function argument be of compatible data types.
- Can a function assign new values to its arguments?

  Yes, because they are simply local variables.
- Does this change the contents of variables that appear in the call to the function?

  No, because the argument variables are local to the function.

This method of passing arguments to functions is known as *call by value*. An alternative technique, *call by reference*, is used in some other programming languages, but generally not by 4GL. The only calls by reference in 4GL are references to BYTE and TEXT data types. These are called by reference because passing blobs by value is not practical.

The use of call by value has an effect on performance. Each argument value is copied into the function's variable. When the arguments are bulky character strings, the time such copying takes can be significant. A common way of avoiding this time penalty is to use global variables.

Working with Multiple Values

4GL lets you work with record variables in a consistent and flexible way. The basic rules for records are:

- The name of a record followed by a dot and an asterisk, *record.**, also means a list of all the members of the record.
- You can select a range of members using *record.first* THRU *last*, where *first* and *last* are names of members of *record*.

These examples illustrate the use of these rules:

```
DEFINE
   rSSS1, rSSS2 RECORD s1, s2, s3 SMALLINT END RECORD
   rFFC RECORD f1, f2 FLOAT , c3 CHAR(7) END RECORD

FUNCTION takes3(a,b,c)
DEFINE a,b,c SMALLINT
...
END FUNCTION
```

These statements define three record variables and declare a function that takes three arguments. The function **takes3( )** is used in examples in subsequent sections.

Assigning One Record to Another

To assign a value to a single member of a record, you use LET.

```
LET rSSS1.s1 = 101
LET rSSS1.s2 = rSSS1.s1 + 1
LET rSSS1.s3 = 103
```

You can assign one record to another using LET when they have the same number of members.

```
LET rSSS2.* = rSSS1.*
```

This statement assigns the three members of **rSSS1** to the corresponding members of **rSSS2**.

In other words, 4GL assigns members one at a time, with automatic data type conversion performed as required. The members must all have simple data types, and the data types must be the same, or else must be compatible. (As defined earlier, *compatible* means that data type conversion is possible.)

Passing Records to Functions

The name of a record is a list of values; and a function takes a list of arguments. Thus you can use a record as a list of arguments.

```
CALL takes3(rFFC.*)
```

The previous statement is equivalent to listing the members:

```
CALL takes3(rFFC.f1,rFFC.f2,rFFC.c3)
```

When calling a function, you can also mix record members and single expressions as arguments:

```
CALL takes3(17, rSSS1.f2 THRU rSSS1.f3)
```

Returning Records from Functions

A function can return more than one value. To make this occur, write a RETURN statement in the FUNCTION definition that contains a list of expressions. Figure 8-3 illustrates this statement. The function **agedBalances( )** returns the amounts that a specified customer owes as three numbers: amounts owed for 30 days or less, 31 to 60 days, and more than 60 days.

Figure 8-3
agedBalances() Function

```
DATABASE stores7
FUNCTION agedBalances(cn)
DEFINE cn LIKE customer.customer_num ,
       bal30, bal60, bal90 DEC(8,2) ,
       ordDate LIKE orders.order_date ,
       ordAmt DEC(8,2)
   LET bal30 = 0.00
   LET bal60 = 0
   LET bal90 = 0

   DECLARE balCurs CURSOR FOR
      SELECT order_date, SUM(items.total_price)
      FROM orders, items
      WHERE orders.customer_num = cn
      AND orders.order_num = items.order_num
      GROUP BY order_date

   FOREACH balCurs INTO ordDate, ordAmt
      IF ordDate <= TODAY - 90 THEN
         LET bal90 = bal90 + ordAmt
      ELSE IF ordDate <= TODAY - 60 THEN
         LET bal60 = bal60 + ordAmt
      ELSE
         LET bal30 = bal30 + ordAmt
      END IF
      END FOREACH
   RETURN bal30, bal60, bal90

END FUNCTION
```

The RETURN statement must match in number of values in the calling function.

A function like this one can be used in several ways. It can be used in a CALL … RETURNING statement. You list variables to receive the values.

```
DEFINE balShort, balMed, balLong MONEY(10)
...
CALL agedBalances(custNumber)
   RETURNING balShort, balMed, balLong
```

If you have a record with appropriate members of the appropriate number of data types, you can refer to it in the RETURNING clause of the CALL statement, as follows:

```
DEFINE balRec RECORD b1, b2, b3 MONEY(10) END RECORD
...
CALL agedBalances(custNumber) RETURNING balRec.*
```

Chapter

9

Using Database Cursors

In This Chapter

In many ways, the SQL language can be considered a subset of the 4GL language because you can *embed* many SQL statements in a 4GL program. This chapter describes how to work with database cursors using SQL in a 4GL program.

The SQL Language

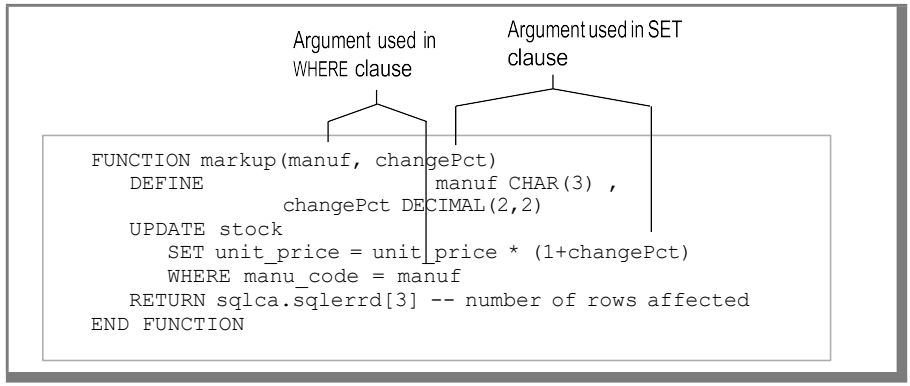
SQL can be used both procedurally and non-procedurally, depending on the needs of your application. Similarly, SQL statements can be static—that is, created at compile time—or dynamic. Dynamic statements are composed during runtime, based in whole or in part on information supplied or selected by the user of the application.

For additional information about the use of SQL in general and embedded in a 4GL program in particular, see the *Informix Guide to SQL: Tutorial*, and the *Informix Guide to SQL: Syntax*. For information on preparing SQL statements, see *INFORMIX-4GL Reference*. If you want to use an SQL statement that was not part of 4.10 SQL (such as CREATE TRIGGER) or a statement using features that were not in 4.10 SQL (such as CREATE TABLE with a fragmentation specification), then the statement must be prepared.

Nonprocedural SQL

Figure 9-1 provides an example of SQL use shown earlier in this book. It defines a function named **markup( )** whose purpose is to alter the prices of stock received from a specified manufacturer.

Figure 9-1
markup() Function



The function takes two arguments. The first, **manuf**, is the code for the supplier whose prices are to be changed. The second, **changePct**, is the fraction by which prices should be changed.

The following example is a call to **markup( )**:

```
LET rowCount = markup("ANZ",0.05)
DISPLAY rowCount, " stock items changed."
```

The SQL statement UPDATE in the definition of the **markup( )** function causes a change in the unit prices of certain stock items in the database. The function argument values are used in this UPDATE statement, one in the SET clause and one in the WHERE clause.

This function is an example of the nonprocedural use of SQL. The UPDATE statement will examine many rows of the **stock** table. It might update all, some, or none of them. The 4GL program does not loop, updating rows one at a time; instead it specifies the set of rows using a WHERE clause and leaves the sequence of events to the database server.

Nonprocedural SELECT

All 4.1-level SQL statements except the SELECT statement can be used by writing them in the body of a function. SELECT also can be used this way as long as only a single row is returned. The following function returns the count of unpaid orders for a single customer, given the customer name:

```
FUNCTION unpaidCount(cust)
   DEFINE cust LIKE customer.company ,
          theAnswer INTEGER
   SELECT COUNT(*) INTO theAnswer
      FROM customer, orders
      WHERE customer.company = cust
      AND customer.customer_num = orders.customer_num
      AND orders.paid_date IS NULL
   RETURN theAnswer
END FUNCTION
```

Because the SELECT statement returns only an aggregate result (a count), it can return only a single value. The argument variable **cust** is used in the WHERE clause. The result of the SELECT operation is assigned to the local variable **theAnswer** by the INTO clause.

SQL statements do not allow program variables in all contexts. You can refer to the syntax diagrams in *Informix Guide to SQL: Syntax* to find out which ones do not.

Row-by-Row SQL

When a SELECT statement can return more than one row of data, you must write procedural logic to deal with each row as it is retrieved. You do this in four or five steps, as shown:

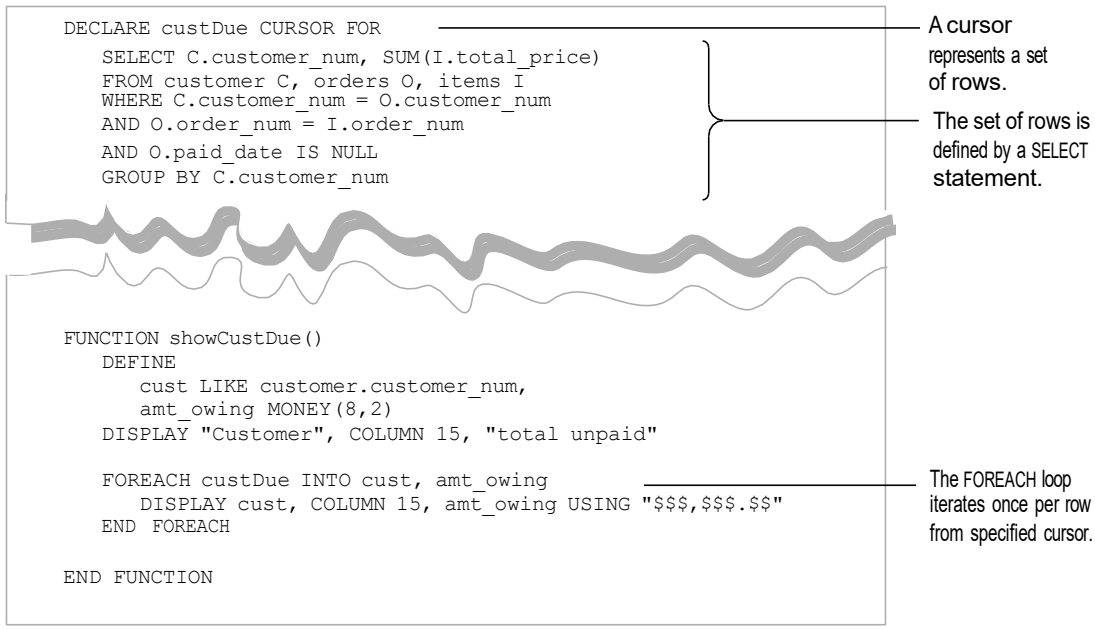
1. If you want to generate your SQL statement dynamically—for example, using the 4GL CONSTRUCT statement to generate dynamic search criteria—place your statement text in a CHAR variable and use the SQL PREPARE statement. For more information, see "Dynamic SQL" on page 9-10 and the *Informix Guide to SQL: Syntax*.
2. You declare a database *cursor*, which is a name that stands for a selection of rows.

3. You specify the rows using a SELECT statement. While you often specify a selection from a single table, you are free to specify rows formed by unions or joins over many tables, and including calculated values, literal values, aggregates, and counts.
4. You use a FOREACH statement to automatically open the cursor, FETCH one row for each traversal of the FOREACH loop, and then close the cursor after you have processed the last row of the selection set.

Alternatively, you can open the cursor, causing the database server to retrieve the first of the specified set of rows, retrieve rows one at a time through the cursor using the FETCH statement (and process each one as it is produced), and then close the cursor, releasing the set of rows.

Figure 9-2 on page 9-7 contains a SELECT statement that retrieves one row for each customer that has an unpaid order in the demonstration database. The selected data consists of the customer number and the total value of that customer's unpaid orders. The DECLARE statement must be executed before FOREACH in **showCustDue( )**.

Figure 2
Row-by-Row SQL Example



Unless used in conjunction with a PREPARE statement, the SELECT statement is written within a DECLARE statement, which creates a database cursor. The cursor, when opened, represents the set of all selected rows. (For more on database cursors and active sets, see the *Informix Guide to SQL: Tutorial*.)

The FOREACH statement in 4GL has three effects:

- It opens the database cursor.
- For every row in the selected set, FOREACH:
  - fetches the column values for that row. (In the example, it assigns the fetched values to local variables **cust** and **amt_owing**.)
  - executes the statements in the body of the loop (a single DISPLAY in the preceding example).
- It closes the cursor.

This pattern is common for many programs: open a cursor, fetch the rows and process each row, and close the cursor. The step *process each row*, of course, can be elaborate, especially when you *process* a row by displaying it in a screen form for the user to read or change.

Updating the Cursor's Current Row

When you have fetched a row of a single table (not a row produced by joining tables) through a cursor, you can delete or update that particular row, by performing the following steps:

- Make the cursor an *update cursor*, which locks the selected row.
- Indicate in your UPDATE statement that you want to update the current cursor row.

To make the cursor an update cursor, add the keywords FOR UPDATE to the cursor declaration; you can also limit the update to certain columns by specifying those column names in the FOR UPDATE clause. When using the cursor, you can update or delete the current row by writing an UPDATE or DELETE statement as usual and adding the clause WHERE CURRENT OF *cursor*, supplying the name of the cursor from which you fetched the row.

The following function uses a cursor to scan the **orders** table and deletes any row for which the paid date is at least a month old. Note that the same task could more easily be accomplished by a nonprocedural UPDATE.

```
DECLARE oldOrder CURSOR FOR
   SELECT order_num, paid_date FROM orders
   — — no WHERE clause, all rows scanned
   FOR UPDATE
FOREACH oldOrder INTO o_num, p_date
   IF 30 < (TODAY - o_num) THEN
      DELETE FROM orders WHERE CURRENT OF oldOrder
   END IF
END FOREACH
```

The clause FOR UPDATE tells the database server that you can update or delete fetched rows. It is not required in an ANSI-compliant database.

Updating Through a Primary Key

Often you will find reasons why you cannot or should not update the current row through the same cursor. For example, if the cursor produces rows based on a join of multiple tables, you can use the nonprocedural UPDATE or DELETE statement instead.

Be sure that the cursor produces a row that contains the primary key of the row to be updated, so that you can isolate the exact row you want to modify. When you fetch a row, you fetch the values of its primary key into program variables.

If you decide to change the row, you execute the UPDATE or DELETE statement that contains a WHERE clause that selects the specific row based on its primary key values.

Updating with a Second Cursor

The following situation arises frequently. You want to select a set of rows using a cursor. You will display each row on the screen and wait for the user to react. The user might then tell you to delete or update the displayed row.

This situation is not a problem when your program is the only one using the database; you can use UPDATE … WHERE CURRENT OF or you can update using the primary key, whichever is appropriate.

However, the situation does become difficult when multiple users might be working in the same table at the same time, using multiple copies of your program or using different programs. You do not want to lock the row while your user examines it; your user might answer the telephone or go to lunch, blocking other users out. Hence you do not want to select rows using an update cursor, which locks rows.

The answer is to use two cursors. The first, primary cursor selects the rows of interest. You include in each row the primary key columns. The second cursor selects only one row based on its primary key and is declared FOR UPDATE. When the user chooses to update the current row, proceed as follows:

1. Open the second cursor.
2. Fetch the one matching row into a temporary record.

   If the row with this ROWID value cannot be found, you know that another user must have deleted it while your user was looking at the screen display.
3. Compare the second set of column values to the ones you displayed to the user.

   If any important ones have changed, you know that some other user has altered this row while your user was looking at the display. Notify your user and do not proceed.
4. Update the row through the second cursor using WHERE CURRENT OF.
5. Close the second cursor.

You can find examples of this kind of programming in *INFORMIX-4GL by Example*.

Dynamic SQL

In the preceding examples, the SQL statements are *static*. That is, they were written into the program source, and hence are static in that their clauses are fixed at the time the source module is compiled. Only the values supplied from program variables can be changed at execution time.

You will need to generate the contents of the SQL statement itself many times while the program is running. For instance, you probably want users of your program to be able to retrieve records based on queries they devise during the day-to-day operation of their business. In other words, in *real time*. When you do this, you are using *dynamic* SQL.

The following function uses dynamic SQL. It assembles the text of a GRANT statement and executes it. It takes the following three arguments:

- The name of the user to receive the privilege.
- The name of a table on which the privilege is to be granted.
- The name of a table-level privilege to be granted (for example, INSERT), as follows:

```
FUNCTION tableGrant( whom, tab, priv )
   DEFINE whom, tab, priv CHAR(20), granText CHAR(100)
   LET granText ="GRANT " , priv, " ON " , tab,
                 " TO " , whom
   PREPARE granite FROM granText
   EXECUTE granite
END FUNCTION
```

This function does nothing about handling the many possible errors that could arise in preparing and executing this statement. You can find a version of the same program that does handle errors in "Using WHENEVER in a Program" on page 12-13.

Chapter

10

Creating Reports

In This Chapter

As explained in "Creating 4GL Reports" on page 6-5, a 4GL report program has the following parts:

- A *report driver* that produces rows of data
- A *report definition* that sorts the rows (if necessary), creates subtotals and other summary information, and formats the rows for output

When you design a 4GL report program, you can design these two parts independently. A report driver can produce rows for any number of reports.

This chapter explains how to design the report driver and the report formatter to generate reports. The 4GL statements that you use for generating reports are covered in detail in *INFORMIX-4GL Reference*. You can also find several examples of programs that produce reports distributed with 4GL.

Designing the Report Driver

This section describes what a report driver does and provides an example of row-producing code. The report driver executes the following steps:

1. Initialize the report using the START REPORT statement.

   This statement initializes the report definition. It can also specify the destination of output from the report, such as the screen, the printer, a file, or another program.

2. Generate rows of data, sending each row using OUTPUT TO REPORT.

   This statement, which is similar to a function call, passes one row of data to the report. Although called a "row," each group of data values need not come from a row of a database table. The values can come from any source, including calculations made by your program. It is equally valid to look at a *row* as an *input record*.

3. Conclude row processing.

4. Terminate the report using FINISH REPORT.

Totals and other aggregates are calculated, output from the report is sent to some destination, and any intermediate files are closed.

These steps assume that no problems are encountered in processing the rows and in producing output from the report. If the report driver detects an error at any point, you can then use the TERMINATE REPORT statement (instead of FINISH REPORT) to abort the report.

An Example of a Report Driver

Row production can be a natural part of a 4GL application. Figure 10-1 shows a brief example of row-producing code. (The report definition appears in Figure 10-3 on page 10-8.)

Figure 10-1
Report Driver Code Example

```
FUNCTION minRows(destfile)
   DEFINE    mn LIKE manufact.manu_Name ,
             sn LIKE stock.stock_num ,
             sd LIKE stock.description ,
             sp LIKE stock.unit_price ,
             su LIKE stock.unit ,
             destfile CHAR(120)
   DECLARE minStock CURSOR FOR
      SELECT manu_Name , stock_num , description ,
             unit_price, unit
      FROM manufact M, stock S
      WHERE M.manu_code = S.manu_code
      ORDER BY 1,2

    START REPORT minStockRep TO destFile

   FOREACH minStock INTO mn,sn,sd,sp,su
    OUTPUT TO REPORT minStockRep(mn, sn, sd, sp, su)
   END FOREACH
   FINISH REPORT minStockRep
END FUNCTION
```

Rows are generated in sorted order.

Report is initialized; file destination can be a variable.

Values for one row are passed like function arguments.

Report output completed and file closed.

This function takes a filename (it might be a complete pathname) as its argument and produces a report with that destination.

The values in each row describe one row of merchandise from the **stores7** demonstration database. The values are produced by a database cursor defined on a join of the **stock** and **manufact** tables. They are produced in sorted order using the row-ordering capability of the database server.

The name of the report, **minStockRep()**, appears in the START REPORT, OUTPUT TO REPORT, FINISH REPORT, and TERMINATE REPORT statements.

Designing the Report Definition

This section describes the REPORT statement and its sections, control blocks, and functions that you use in a report definition. Although both the FUNCTION and REPORT statements define 4GL program blocks, a report is not a function. Some of the differences are as follows:

- The CALL statement cannot invoke a report; attempting to do so produces a fatal error.
- Reports are not recursive. The results will be unpredictable if the START REPORT statement invokes a report that is already running.
- The Interactive Debugger cannot analyze a report definition.
- A report does not return anything to its driver. It is an error to include the RETURN statement within a report definition.
- The 4GL statements NEED, PAUSE, PRINT, and SKIP are valid in report definitions, but produce errors if they appear in a function:

Although a report has the general form of a function, its contents are quite different. The body of a function contains whatever statements you specify, while the body of a report contains several independent statement blocks that must appear in a fixed sequence, if they are present, and are executed as needed. The fixed sequence for the statement blocks within the REPORT statement is:

- DEFINE
- OUTPUT
- ORDER BY
- FORMAT

However, within the FORMAT statement block, statements can appear in any order. Figure 10-2 on page 10-7 shows the **minStockRep()** report definition that completes the preceding example. This code is examined in detail in the topics that follow.

Figure 10-2
minStockRep() Report Definition

```
REPORT minStockRep(manName, stNum, stDes, stPrice, stUnit)

DEFINE
   misc, showManName SMALLINT ,
DEFINE
   manName, thisMan LIKE manufact.manu_name ,
   stNum LIKE stock.stock_num ,
   stDes LIKE stock.description ,
   stPrice LIKE stock.unit_price ,
   stUnit LIKE stock.unit
OUTPUT
   LEFT MARGIN 8
   PAGE LENGTH 20 -- short page for testing purposes
ORDER
   EXTERNAL BY manName, stNum

FORMAT
   FIRST PAGE HEADER
       PRINT "Stock report",COLUMN 62, PAGENO USING "###"
             SKIP 2 LINES
   PAGE HEADER
      PRINT "Stock report",COLUMN 62, PAGENO USING "###"
      SKIP 2 LINES
   PAGE TRAILER
      SKIP 2 LINES
      LET misc = 65 - LENGTH(thisMan)
      PRINT COLUMN misc, thisMan
      LET showManName = TRUE
   BEFORE GROUP OF manName
      LET thisMan = manName
      LET showManName = TRUE
   AFTER GROUP OF manName
      SKIP 1 LINE
   ON EVERY ROW
      IF showManName THEN -- start of new group so...
              PRINT thisMan; -- with no newline
           LET showManName = FALSE
      END IF
      PRINT COLUMN 20, stNum USING "###" ,
         COLUMN 25, stDes CLIPPED ,
         COLUMN 45, stPrice USING "$,$$$.&&" ,
         COLUMN 55, stUnit CLIPPED
   ON LAST ROW
      SKIP TO TOP OF PAGE
      PRINT COUNT(*), " total rows processed."
END REPORT
```

Values for one row passed as arguments.

Local variables created at START REPORT time; kept until FINISH REPORT.

States that rows are produced in sorted sequence.

Statement blocks are called when necessary as rows are processed.

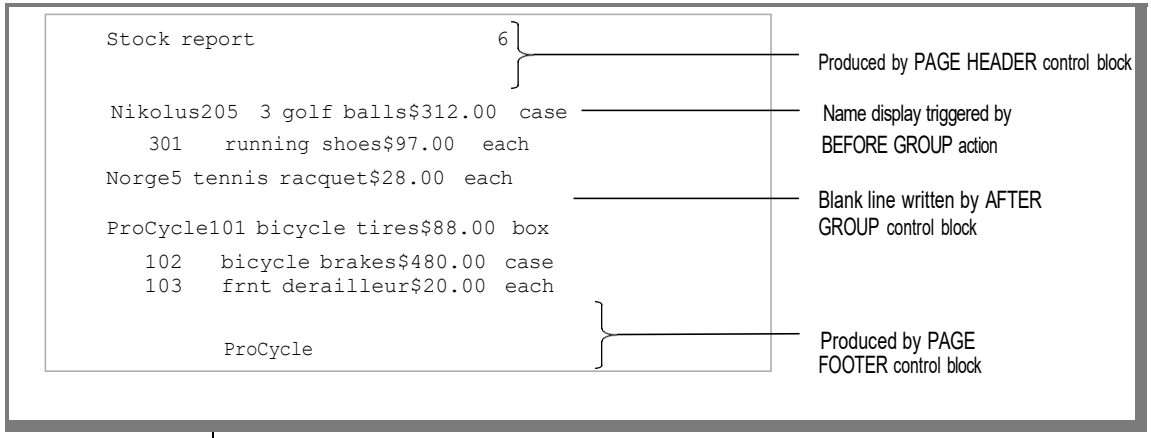
Calculated COLUMN value is used to right-justify name.

Logic to display manufacturer in first line of group or at top of page.

Semicolon suppresses new line after printing.

Figure 10-3 shows an excerpt from the output of this report. The page length was set to 20 for testing; it would normally be longer.

Figure 1
A Sample 4GL Re t



The REPORT Statement

All reports are defined by a REPORT statement. This statement is similar i some ways to a FUNCTION statement. It begins with a prototype that declare the name of the report, and a list of formal arguments. The following examp is the beginning of the **minStockRep()** report definition:

```
REPORT minStockRep(manName, stNum, stDes, stPrice, stUnit)

DEFINE
   misc, showManName SMALLINT ,
   manName, thisMan LIKE manufact.manu_name ,
   stNum LIKE stock.num ,
   stDes LIKE stock.description ,
   stPrice LIKE stock.unit_price ,
   stUnit LIKE stock.unit
```

The report name is used to identify this report in the START REPORT, OUTPUT TO REPORT, FINISH REPORT, and TERMINATE REPORT statements within the report driver. This name cannot be the same as the identifier of another report, or function, or global variable, that is defined in the same 4GL program.

The END REPORT keywords (described in a later section) terminate the report definition.

The formal arguments to the report indicate the number of actual values that the report driver can pass to the report as one input record. This report takes five arguments. For the purposes of this report, one set of these values makes a row. Within the body of the report, you must define a local variable of the REPORT program block that corresponds to each argument. You can then refer to these variables to find values of the current report row.

When the report driver code executes OUTPUT TO REPORT, it sends another set of values (that is, another row), to the report for processing. The following statement is from "An Example of a Report Driver" on page 10-5:

```
FOREACH minStock INTO mn, sn, sd, sp, su
   OUTPUT TO REPORT minStockRep( mn, sn, sd, sp, su )
END FOREACH
```

The Report Declaration Section

A report declares local variables, much like a function. Their definition must immediately follow the REPORT statement. You must declare a local variable that matches the name and data type of each argument. You can also declare additional local variables to store other values that the report uses.

The **minStockRep( )** report defines several local variables using the LIKE keyword. Just as you can define the local variables of a function using LIKE, you can use the LIKE keyword to assign a data type from the current database to a report variable.

The local variables of a report are created and initialized when the START REPORT statement is executed. They remain in existence until the report is ended by FINISH REPORT (or by TERMINATE REPORT). These variables are not reinitialized each time OUTPUT TO REPORT is executed. (This is one of the ways that a report differs from a function. The local variables of a function are created anew each time that the function is called.)

The OUTPUT Section

The OUTPUT section of a report is executed and takes effect at START REPORT time. The values that it sets can be changed only if START REPORT overrides them by specifying different values. The report assumes that output is produced in a monospace font, so that every character has the same width.

This section contains statements that set the basic format of the report. The five statements that establish page layout are summarized in the following table.

Statement	Usage
LEFT MARGIN	Number of spaces inserted to the left of every print line
RIGHT MARGIN	Total number of printed characters in any line, including left margin spaces (This statement is ignored unless the FORMAT EVERY ROW default report formatting option, or default WORDWRAP is used.)
TOP MARGIN	Number of blank lines to print above the page header.
BOTTOM MARGIN	Number of blank lines to print after the page trailer (also known as a *footer*)
PAGE LENGTH	Total number of lines per page, including margins and page header and trailer sections

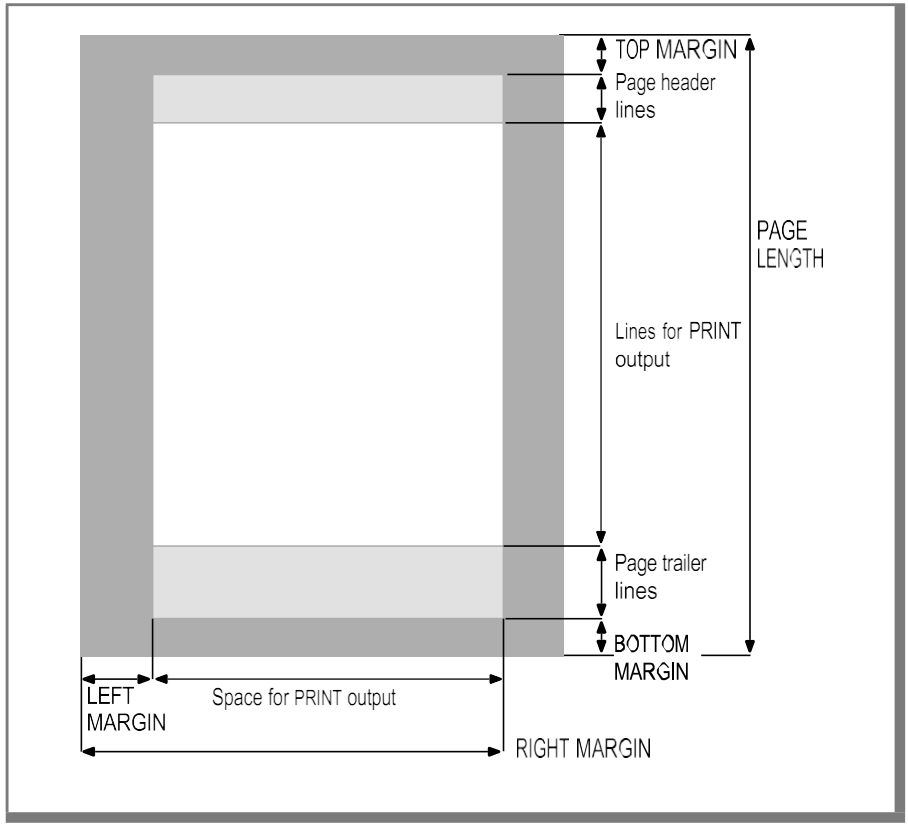
These values must be expressed as literal integers. (Use the START REPORT statement, if you want to use variables to specify page dimensions.)

The TOP OF PAGE specification in the OUTPUT section can specify a character value (as a quoted string) that 4GL can use to cause a page-eject. If you omit it, 4GL starts a new page by printing empty lines.

The REPORT TO keywords can be followed by a default destination for output from the report. (But any destination in the START REPORT statement takes precedence over what REPORT TO specifies.) The destination must be specified as a quoted string (except that a named pipe destination can be a character variable).

Figure 10-4 illustrates the page layout statements.

Figure 10-4
Statements That Establish Page Layout Specifications



The vertical arrows represent page dimensions that you specify in units of *lines;* the horizontal arrows represent page dimensions that you specify in units of monospace *characters*.

The ORDER BY Section

The ORDER BY section of a report specifies whether the rows of data are to be sorted, and if so, whether or not they will be produced in sorted sequence. You must decide between the following three cases:

1. The order of the rows is not important; that is, the report is simply a list of rows in the order in which they happen to be generated. To choose this, omit the ORDER BY section entirely.

   When ORDER BY is not used, you should not process rows in groups or take aggregate values over them.

2. The rows need to be generated in the order specified by the report driver. To choose this, specify ORDER EXTERNAL BY and list the field on which the sort takes place.

   If the report driver code fails to generate the rows in their proper sequence, the report output will be incorrect.

3. The rows need to be sorted, but the report driver does not produce them in the correct order. To choose this, you write ORDER BY and specify the fields to sort on. You do not use EXTERNAL in this case.

When rows are to be sorted, it is best if the report driver code can produce them in correct order. When the rows come from the database, you can use the ORDER BY clause of the SELECT statement. The database server has the most efficient ways of producing sorted rows.

When it is necessary that the report itself order the rows, 4GL uses *two-pass report logic*, as discussed in "One-Pass and Two-Pass Reports" on page 10-13.

Sort Keys

The ORDER BY statement can specify the *sort keys* of the report. Here is the ORDER BY specification in **minStockRep()**. The EXTERNAL keyword specifies that rows should be sorted, and that they are produced by the report driver in sorted order, rather than sorted by **minStockRep()**.

```
ORDER
   EXTERNAL BY manName,stNum
```

The priority of the sort keys decreases from first to last; that is, the first one named is the major sort key. In the example, rows are sorted on **manName**. Within groups that contain matching **manName** values, rows are sorted on the value of **stNum**.

The sort keys are used to define groups of rows. You can use the BEFORE GROUP and AFTER GROUP sections to take special actions on these groups.

One-Pass and Two-Pass Reports

Report data is processed in one of the following two ways:

- **One-pass.** Rows are processed as they are produced. Each time a row is produced by an OUTPUT TO REPORT statement, it is processed and the resulting output is written to the report destination.
- **Two-pass.** Rows are collected, saved, sorted, and then processed. As rows are produced, they are saved in a temporary table in the current database. When FINISH REPORT is executed, all the saved rows are retrieved in sorted order and processed.

4GL chooses between these methods based on two things: how the report rows are ordered and how the report uses aggregate functions.

Two-Pass Logic for Row Order

Sorting of rows is essential to most reports. If the rows are not sorted, they cannot be divided into groups with similar values. These groups are the basis for subtotals and other summary information.

Sometimes it is convenient for the report driver to produce the rows in the sequence you need. Other times this is not possible; the rows are produced in random order. In such cases, 4GL uses a two-pass report to sort rows before they are formatted.

Two-Pass Logic for Aggregate Values

Aggregate functions are used to get totals and other computed values (see "Using Aggregate Functions" on page 10-21). You are allowed to refer to aggregate values anywhere in the FORMAT section of a report. When you use aggregate functions, you are asking for values based on the contents of all rows within the defined REPORT (a total aggregate), or within a specified GROUP (a GROUP aggregate).

When you do not use aggregate functions, or when you use them only in the ON LAST ROW block, 4GL can employ one-pass logic. But if you refer to aggregate functions outside the LAST ROW block, 4GL must use a two-pass report. A two-pass report works as follows:

- When the report driver executes OUTPUT TO REPORT, the row value is saved in a temporary table, and the aggregate function values are accumulated in memory.
- When the report driver executes FINISH REPORT to indicate that no more rows will be produced, 4GL retrieves all the rows from the temporary table in their proper sequence and sends them to the report for formatting.

The values of the aggregate functions are now available while the rows are processed because they have been pre-computed.

Further Implications of Two-Pass Logic

When a report uses one-pass logic, the execution time of report output is distributed over all the OUTPUT TO REPORT statements because each row is formatted as it is received. The only action of FINISH REPORT is to print final totals.

When a report uses two-pass logic, the OUTPUT TO REPORT statement runs quickly because it merely inserts a row in a temporary table. The actions of FINISH REPORT, however, can be quite lengthy, because that is when all rows are retrieved and formatted.

A two-pass report builds a temporary table in the database that is current at the time START REPORT is executed. The same database must be open when OUTPUT and FINISH statements are executed. This places a restriction on the report driver: it cannot change databases (that is, execute the DATABASE statement or a CONNECT statement embedded in an ESQL/C function) during a two-pass report.

An error also occurs if the report definition requires two-pass logic but no database is open when the report attempts to read or write a temporary table. Even if your report obtains all of its rows from some source other than a database, two-pass logic requires a database for the temporary tables.

You must use caution, however, when writing code that refers to global variables or that interacts with the user. In a two-pass report, the formatting code is not called until all rows have been produced and FINISH REPORT has been executed. Global variables might not have the same values, and the screen might not display the same data as when the rows were produced.

The FORMAT Section

Within the FORMAT section of a report, you place blocks of code that produce output lines of data in the report. The following table summarizes the control blocks.

Statement	Usage
PAGE HEADER	Prints the heading of any page
FIRST PAGE HEADER	Prints a special heading or cover page
PAGE TRAILER	Prints a footer at the end of any page
BEFORE GROUP	Initializes counters and totals at the start of a group of rows with similar contents; prints group headings

(1 of 2)

AFTER GROUP	Prints totals and other summary information following a group of rows with similar contents
ON EVERY ROW	Formats and displays detail lines; accumulates totals and calculated values for use by AFTER GROUP blocks
ON LAST ROW	Displays final totals and aggregate values over all rows

(2 of 2)

4GL executes these control blocks automatically at appropriate times as rows are processed. For example, 4GL calls the PAGE TRAILER code block when it is time to print the page trailer. When that block completes, 4GL prints the blank lines corresponding to the BOTTOM MARGIN and prints the page-eject string, if any.

If additional information is written to the output from the report, 4GL prints the TOP MARGIN blank lines and calls the PAGE HEADER block.

Contents of a Control Block

Within a formatting control block, you can write any executable 4GL statements that you like, except for the following statements:

- CONSTRUCT
- DISPLAY ARRAY
- INPUT
- INPUT ARRAY
- MENU
- RETURN

You can, however, call functions that use these statements to interact with the user. You can even start other reports and send output to them.

Formatting Reports

Usually a block contains code to test and set the values of local variables, and code to format and print the input records (whose values are stored in the local variables corresponding to the REPORT statement argument list).

The following report execution statements are available to display data.

Statement	Usage
SKIP	Inserts blank lines
NEED	Forces a set of lines to appear on the same page
PRINT FILE	Embeds the contents of a file in the output
PRINT	Writes lines of output
PAUSE	Waits for the user to press RETURN, but only if the output is going to the screen; otherwise, it is not an option

It is with PRINT that you send report data to the output destination. Like the DISPLAY statement, PRINT accepts a list of values to display. Within a PRINT statement, you can use various features of 4GL to format the output, including the following operators.

Keyword	Usage
ASCII	Prints specified character values
CLIPPED	Eliminates trailing spaces from CHAR or NCHAR values
COLUMN	Positions the data in the specified column
PAGENO	Returns the current page number on report output
SPACES	Prints a specified number of spaces
USING	Formats dates, numbers, and currency amounts
WORDWRAP	Displays long character strings in a multiple-segment field

PAGE HEADER and TRAILER Control Blocks

Within the PAGE HEADER and PAGE TRAILER control blocks, you write code that formats the pages of the report with fixed headings and pagination. The **minStockRep()** report from Figure 10-1 on page 10-5 contains this page heading code:

```
PAGE HEADER
   PRINT "Stock report",COLUMN 62,PAGENO USING "###"
   SKIP 2 LINES
```

It prints a fixed heading and a page number, and keeps count of the pages. A total of three lines is written, one line of heading and two blank lines. These lines are in addition to the TOP MARGIN lines specified in the OUTPUT section.

If a FIRST PAGE HEADER block is present, then the PAGE HEADER block does not take control until the second page of output is started.

The **minStockRep()** report contains this page trailer code:

```
PAGE TRAILER
   SKIP 2 LINES
   LET misc = 65 - LENGTH(thisMan)
   PRINT COLUMN misc,thisMan
   LET showManName = TRUE
```

The report prints the manufacturer name from the last-processed group of rows, right-justified, on the last line of the page. The person reading the report can find a manufacturer quickly by scanning the bottom-right corner of each page. The code also sets a flag that tells the ON EVERY ROW block to display the manufacturer name in the next detail line (because that will be the first detail line of the next page).

The FIRST PAGE HEADER section is similar to the PAGE HEADER except that 4GL calls it only once, before any other block. You can use it to display a cover or a special heading on the first page. In a two-pass report, you could put code in this section to notify the user that report output is finally beginning.

ON EVERY ROW Control Block

In the ON EVERY ROW control block, you write the code to display each detail row of the report. The following block is from the sample report specification in Figure 10-1 on page 10-5:

```
ON EVERY ROW
   IF showManName THEN -- start of new group so...
      PRINT thisMan; -- with no newline
      LET showManName = FALSE
   END IF
   PRINT COLUMN 20, stNum USING "###" ,
      COLUMN 25, stDes CLIPPED ,
      COLUMN 45, stPrice USING "$,$$$.&&" ,
      COLUMN 55, stUnit CLIPPED
```

It displays a line like this:

```
Nikolus             205  3 golf balls $312.00  case
```

The leading spaces are produced by the LEFT MARGIN statement in the OUTPUT section. This code suppresses the manufacturer name except in the first row of a group or the first row on a new page.

ON LAST ROW Control Block

After the final row and any closing AFTER GROUP OF statements have been processed, and the FINISH REPORT statement has been encountered, 4GL calls the ON LAST ROW control block. In this block, you can write code to summarize the entire report. You can use SKIP TO TOP OF PAGE in this block to ensure that the final totals appear on a new page.

The ON LAST ROW block from **minStockRep()** follows:

```
ON LAST ROW
   SKIP TO TOP OF PAGE
   PRINT COUNT(*), " total rows processed."
```

For the use of COUNT(*) and other aggregate functions, see "Using Aggregate Functions" on page 10-21.

BEFORE GROUP and AFTER GROUP Control Blocks

Whenever the value of a sort key changes between one row and the next, it marks the end of one or more groups of rows and the start of another. In the same **minStockRep()** example, whenever there is a change of **manName**, a group of rows ends and another begins. The same is true of a change in **stNum**, but because stock numbers are unique in this database, these "groups" never have more than one member.

A BEFORE GROUP control block is called when the first row of the new group has been received, but before it is processed by the ON EVERY ROW control block. In it, you can put statements that:

- initialize counts, totals, and other values calculated group-by-group.
- print group headings, use the NEED STATEMENT to ensure sufficient space on the page, or force a skip to a new page for the group.
- set flags and local variables used in the ON EVERY ROW block.

The BEFORE GROUP statement used in **minStockRep()** from Figure 10-1 on page 10-5 follows:

```
BEFORE GROUP OF manName
   LET thisMan = manName
   LET showManName = TRUE
```

The first statement saves the manufacturer name from the first row of the new group so it can be used in the page trailer, as described earlier. The second statement sets a flag that tells the ON EVERY ROW block to display the manufacturer name in the next detail line because that will be the first detail line of this group.

An AFTER GROUP block is called when the last row of its group has been processed by the ON EVERY ROW block. In it, you can put statements that calculate and print subtotals, summaries, and counts for the group.

Nested Groups

Each of the sort keys that you list in the ORDER section defines a group. In the report example there are two keys and therefore two groups:

```
ORDER
   EXTERNAL BY manName,stNum
```

These groups are "nested" in the sense that the rows in a major group can contain multiple groups of the minor group.

In general, the BEFORE and AFTER blocks for minor groups will be called more times than those for major groups. The group for the last sort key you specify will change the most often. Only the ON EVERY ROW block will be executed more frequently.

Default Reports

The FORMAT section is required in every report definition, but you are not required to specify any control blocks. If it is useful for the report output to be simply a listing of every input record that was passed by the report driver, then you can specify EVERY ROW as the only FORMAT section specification. No other control blocks are valid when you specify EVERY ROW. Reports that use this simplified format for output are called *default reports*, and are seldom used in production code.

Using Aggregate Functions

Often, reports that deal with sorted data need to collect aggregate values over the rows: counts, subtotals, averages, and extreme high or low values. You can produce such statistics yourself by writing code in different blocks. For example, you could collect an average value over a group the following way:

1. In the BEFORE GROUP block, initialize variables for the sum and the count to zero.
2. In the ON EVERY ROW block, increment the count variable and add the current row's value to the sum variable.
3. In the AFTER GROUP block, calculate the average and display it.

4GL contains built-in aggregate-value functions for most common needs. You can, however, write your own statistical functions to calculate other values, or to avoid the need to do two-pass reporting, or both. For example, if you want to keep running totals or page totals, these must be hard-coded.

Aggregate Calculations

For sums, averages (means), and extremes, 4GL supplies these functions.

Function	Usage Over Many Rows
SUM(*expression*)	Accumulates a sum
AVG(*expression*)	Calculates an average
MIN(*expression*)	Finds a minimal value
MAX(*expression*)	Finds a maximal value

For MIN( ) and MAX( ), the expression can be any 4GL expression. For SUM( ) and AVG( ), the expression must be a number or INTERVAL expression. Typically, the argument of an aggregate function is one of the arguments to the REPORT, but it can also be a literal value, a local variable, and even a function call.

When using function calls or global variables, keep in mind that rows might all be processed at FINISH REPORT time. Also keep in mind that aggregates are accumulated separately from the code that uses them. Therefore, doing a calculation on a local variable and then aggregating it will not produce the answer you expect. Unless the argument is an aggregate function in a report parameter (which is not modified before any attempted aggregation), the results of the aggregate are not readily predictable.

These aggregate functions can be qualified with a WHERE clause to select only a subset of rows. The WHERE clause typically applies criteria to the row values themselves, but you can employ any Boolean expression that is valid in an IF statement. For example, the following lines could be added to the LAST ROW block in Figure 10-1 on page 10-5:

```
PRINT "Lowest item price", MIN(stPrice)
PRINT "Average low-cost item" ,
     AVG(stPrice) WHERE stPrice < 100
```

These lines would display the minimum overall **stPrice** values, and the average of all **stPrice** values that were less than $100.

Aggregate Counts

For counts, 4GL supplies the COUNT(*) and PERCENT(*) functions. The value of COUNT(*) is the number of records processed by the report. You can see it in use in the ON LAST ROW control block in Figure 10-1 on page 10-5.

COUNT(*) can also be qualified with a WHERE clause, however, so as to count only a subset of rows. The following lines could be added to the ON LAST ROW control block of **minStockRep()**:

```
PRINT "number of boxed items" ,
    COUNT(*) WHERE stUnit = "box"
```

The PERCENT(*) function returns the value of one count as a percentage of the total number of rows processed:

```
PRINT "percent of case lots" ,
    PERCENT(*) WHERE stUnit = "case"
```

Aggregates Over a Group of Rows

You most often need aggregate values collected over the rows of one group; for example, a sum over a *group* to produce a *group* subtotal. You can use any of the six aggregate functions within an AFTER GROUP block for this purpose. You must use the keyword GROUP to specify that you want the aggregate value over the rows of the current group.

For example, the following lines could be added to the AFTER GROUP block in the **minStockRep()** report on Figure 10-1 on page 10-5:

```
PRINT "Count of stock items: " , GROUP COUNT(*) USING "<<<"
PRINT "Avg price of 'each' items: " ,
      GROUP AVG(stPrice) WHERE stUnit = "each"
```

This code displays a group count, which is simply a count of rows in that group, and an average. The average is taken over a subset of the rows of the group. When the subset is empty (when no rows have `stUnit="each"`), the value of the aggregate function is NULL. Otherwise, any null values are disregarded in calculating aggregate values.

The value of GROUP PERCENT(*) WHERE… might not be what you expect. It returns the number of rows in the group that met the condition, as a percentage of the total number of rows in the entire report, not as a percentage of the rows in the group. Because it requires the total number of rows, GROUP PERCENT forces a report to use two-pass logic. To calculate a percentage within a group, you can use explicit code such as:

```
PRINT "Pct 'each' items in group " ,
( (100 * GROUP COUNT(*) WHERE stUnit = "each")
   / (GROUP COUNT(*) )
USING "<<.&&"
```

END REPORT and EXIT REPORT

Just as END MAIN marks the end of the MAIN statement, and END FUNCTION marks the end of a function definition, a REPORT program block must be terminated by the END REPORT keywords. If control of execution reaches the END REPORT keywords, it returns to the report driver.

You can exit from a report definition before the END REPORT keywords by executing the EXIT REPORT statement. This statement has the same effect within a REPORT statement that TERMINATE REPORT has in the report driver. When EXIT REPORT is executed, 4GL closes the report, and its local variables are deallocated.

Typically, the EXIT REPORT statement is not executed unless some condition is detected that indicates that further processing of input records should not continue; for example, when the printer is out of paper. You can use EXIT REPORT within a report in situations where RETURN would be appropriate in a function. But as noted earlier in this chapter, RETURN is not valid in a report.

Chapter

11

Using the Screen and Keyboard

In This Chapter

The architecture of the INFORMIX-4GL user interface is described in Chapter 3, "The INFORMIX-4GL Language." The concepts behind the statements described in this chapter are covered in "The User Interface" on page 7-3.

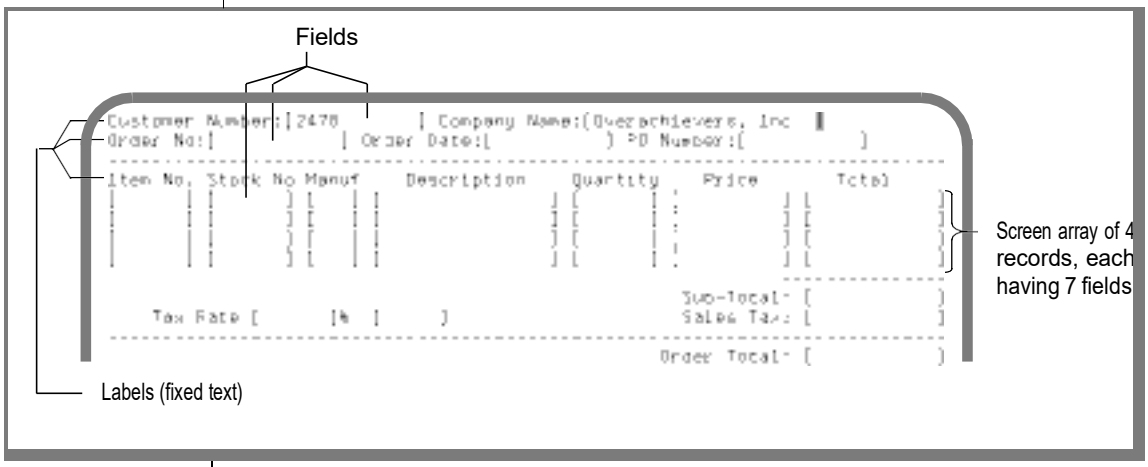
This chapter details the way you program the user interface for your 4GL applications. Because screen forms are important to this, the first topic explains how you specify a form. Then statements you use to open 4GL windows and fill them with forms and menus are discussed. At the end of the chapter, keyboard and screen line customization are discussed.

Specifying a Form

This section describes the contents of a form specification file in detail. The idea of managing screen forms in separate files was introduced in "Form Specifications and Form Files" on page 4-6. An overview of how your programs use forms is given in "How Forms Are Used" on page 7-13.

Figure 11-1 shows a screen image of a sample form.

Figure 1
A Screen Image of a Sample For



4GL forms are traditionally designed in a WYSIWYG (what-you-see-is-what-you-get) environment using an ASCII text editor. Each ASCII form specification contains several sections as summarized in the following table. Some are mandatory, others are optional. The order of the sections is significant.

Form Section	Usage
DATABASE	Specifies the database containing the tables and views whose columns are associated with fields in your form
SCREEN	Specifies the arrangement of fields and text that will appear in your form after it is compiled and displayed
TABLES	Specifies which tables have columns associated with the fields in the form, and can declare aliases for tables that require qualifiers
ATTRIBUTES	Declares a name for each field, and assigns attributes to it
INSTRUCTIONS	Specifies non-default delimiters and screen records

For more information on form specifications and statements, see *INFORMIX-4GL Reference*.

The DATABASE Section

Typically a form field holds data from a particular column in the database. For example, the previous form begins with a field labeled **Customer Number**. This field displays data from the **customer_num** column in the **customer** table of the **stores** demonstration database.

You can associate a form field with a database column. In such cases, FORM4GL, the form compiler, will look in the specified database to determine the data type for that column and will use that as the data type of the form field.

The first line in the form specification file is a DATABASE specification. It tells the form compiler which database to use when looking for columns.

```
DATABASE stores7
```

Some forms use no database at all; that is, they have no fields that correspond to a database column. When this is the case, you can write:

```
DATABASE formonly
```

The SCREEN Section

In the SCREEN section, you specify the appearance of the form on the screen. With a text editor, you *paint* a picture of the form as you want it to appear. The SCREEN section of the form on Figure 11-1 on page 11-4 appears below. The lines between the curly braces ({}) are a picture of the form as it is to appear on the screen. (The lines that contain the braces are *not* part of the form image.)

```
SCREEN
{

Customer Number:[f000        ] Company Name:[f001                 ]
Order No:[f002        ] Order Date:[f003       ] PO Number:[f004       ]

Item No. Stock No Manuf     Description     Quantity     Price      Total
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]
[f005  ] [f006  ] [f07] [f008             ] [f009  ] [f010      ] [f011 ]

                                                  Sub-Total: [f012 ]
     Tax Rate [f013  ]%  [f014 ]                  Sales Tax: [f015 ]

                                                 Order Total: [f016 ]
}
```

The square brackets ([ ]) delimit fields into which the program can later display data and take input. Text outside brackets is label text. It will be displayed exactly as typed. A field consists of the spaces between brackets. If you need to make two fields adjacent, you can use a single pipe symbol (|) to separate the fields.

Each form field must contain a field tag, a name used to link the field in the SCREEN section to its attributes in the ATTRIBUTES section of the form specification. The first field in the SCREEN section of the previous example has the field tag **f000**.

When you use the same field tag in more than one field (the tag **f009** appears four times in the preceding example), you indicate that the field is really one part of a larger whole, either a multiple-segment field or a screen array.

Specifying Screen Dimensions

By default, the forms compiler assumes that your form will be displayed in a format of 24 lines of 80 columns each. If you intend to use the form in a larger window, you must specify larger dimensions in the SCREEN statement.

Screen Records and Screen Arrays

A screen record is a collection of fields that are treated as a group, just as a program record is a collection of members you want to treat as a related group. 4GL makes it easy to display all the elements of a program record into the corresponding fields of a screen record, or to take input from a screen record into a program record.

A screen array is an array of similar fields that your program will treat as a unit, like a program array. 4GL enables you to associate an array of data in the program with an array of fields on the screen, so that the user can view or edit the rows of data on the screen.

The form example in Figure 11-1 on page 11-4 has a screen record composed of the fields tagged **f005**, **f006**, **f07**, **f008**, **f009**, **f010**, **f011**. These fields make up an order item detail record. They are repeated four times vertically to make an array of four records. A statement in the INSTRUCTIONS section of the form is used to declare the array of records. See "The INSTRUCTIONS Section" on page 11-13.

The process of displaying an array of data is discussed in "Taking Input Through an Array" on page 11-30.

Multiple-Segment Fields

To display a string that is too long to fit on a single line of the form, you can create multiple-segment fields that occupy rectangular areas on successive lines of the form. The SCREEN section must repeat the same field tag in each segment of the multiple-segment field, and the ATTRIBUTES section must assign the WORDWRAP attribute to the field.

Your 4GL program treats a multiple-segment field as a single field. When it displays text, any string longer than the first segment is broken at the last tab or blank character before the end of the segment. Thus, word boundaries are respected. The rest of the string is continued in the first character position of the next segment. This process is repeated until the end of the last segment, or until the last character is displayed, whichever happens first.

The TABLES Section

The TABLES section is closely related to the DATABASE section. In it you specify the tables that supply data for this form. The TABLES section of the form in Figure 11-1 on page 11-4 is:

```
TABLES
    customer orders items stock state
```

Any tables that contain columns named in the ATTRIBUTES section of the form (described in the next section of this chapter), must be listed in the TABLES section. The forms compiler will look for these tables in the database named in the DATABASE section.

You can also assign aliases to table names in the TABLES section. For example, in the TABLES section you can write:

```
TABLES postings = financial.postings
```

This gives the short table name **postings** to the table known in full as **financial.postings**. You must use the short name in the ATTRIBUTES section of the form.

Aliases are needed in the following cases:

- You are using an ANSI-compliant database. In such cases, you must declare table aliases—even if you are only drawing data from one table—unless the end user of your application is also the owner of every table in the TABLES section. This is because table owners must be specified when using ANSI-compliant databases.
- You are drawing data from several tables with the identical column names. The alias is necessary because you cannot qualify a form field name with a table identifier or owner name.
- You are accessing data from tables in remote databases. The alias is necessary because you are restricted in the attributes section to the following format:

  ```
  table_identifier.column_name
  ```

- You want a condensed way to refer to a table. The full name of a table can be quite lengthy because it can contain an owner name, a database name, and a site name. Aliases make referencing such tables more convenient and your form specification more readable.

Like the DATABASE section, the TABLES section is used only to get information needed by FORM4GL. It has no effect when the form is used by a program. The data you display in a form can come from any source, including any table in an Informix database.

The ATTRIBUTES Section

In the ATTRIBUTES section, you give detailed specifications for each of the fields you have defined in the SCREEN section. The two most important attributes for any field are its field name and its data type.

The field name is the name that you use, in your program, to put data into a field and get data out of it.

You assign these and other attributes in a series of specifications. The ATTRIBUTES section for the form in Figure 11-1 on page 11-4 follows. It illustrates some of the many attributes that can be assigned to a field. (For a complete list of attributes, see *INFORMIX-4GL Reference*.)

```
ATTRIBUTES
 f000 = orders.customer_num;
 f001 = customer.company;
 f002 = orders.order_num;
 f003 = orders.order_date, DEFAULT = TODAY;
 f004 = orders.po_num;
 f005 = items.item_num, NOENTRY;
 f006 = items.stock_num;
 f007 = items.manu_code, UPSHIFT;
 f008 = stock.description, NOENTRY;
 f009 = items.quantity;
 f010 = stock.unit_price, NOENTRY;
 f011 = items.total_price, NOENTRY;
 f012 = formonly.order_amount,TYPE MONEY(7,2);
 f013 = formonly.tax_rate;
 f014 = state.code, NOENTRY;
 f015 = formonly.sales_tax TYPE MONEY;
 f016 = formonly.order_total;
```

The Field Name

A *field name* is the name your program uses to refer to a field for data display or input. The field name is often the same as the name of a database column. For example, the first specification from the sample ATTRIBUTES section follows:

```
f000 = orders.customer_num;
```

This says that the field shown in the SCREEN section with a field tag of **f000** is associated with the **customer_num** column of the **orders** table. As a result, the name of this field is **customer_num**. To display data in this field, you would write a statement such as:

```
DISPLAY max_cust_num+1 TO customer_num
```

The Field Data Type

The data type affects how data is displayed in the field; for example, numeric data is right-justified and character data is left-justified. The data type also affects how the field behaves while the user is entering data during input; for example, the user cannot enter non-numeric characters in a field with a numeric type. You can specify the type of data that can be stored in a field directly or indirectly.

Fields Related to Database Columns

You will often want fields to display data taken directly from a database column. To simplify the definition of such fields, you can name the database column and table. The field then receives both its name and its data type from the database:

```
f009 = items.quantity ;
```

In this example, all the fields with tag **f009** (there are four in "The SCREEN Section" on page 11-5) take their name and data type from the **quantity** column of the **items** table. The **items** table must have been listed in the TABLES section of the form; also, a table of that name must exist in the database named in the DATABASE section at the time the form is compiled.

FORM4GL opens the named database and verifies that it contains an **items** table with a **quantity** column. It takes the data type of that column as the data type for the field. The advantage is that, if the database schema changes the data type of a particular column, you can keep your forms consistent with the new schema simply by recompiling them.

Form Only Fields

A field that is not related to a particular database column is called a *form only* field. You specify its name and its data type in the attributes statement:

```
f012 = formonly.order_amount TYPE MONEY(7,2)
```

The field with tag **f012** is given the name **order_amount**. In a program, you display data in the field by using this name. FORM4GL cannot consult a database to see what the type should be, so you must tell it with a TYPE attribute clause. If you omit type information, the default is a character string equal in size to the number of spaces between brackets in the SCREEN section.

Editing Rules

You can specify editing rules for fields in the ATTRIBUTES section. Some of the rules you can apply to a field are shown in the following table. (For details on form attributes, see *INFORMIX-4GL Reference*.)

Attribute	Usage
DOWNSHIFT UPSHIFT	Changes the lettercase as the user types. (An example of the UPSHIFT attribute can be seen in "The ATTRIBUTES Section" on page 11-9.)
INCLUDE	Specifies a list of values permitted on input. (For example, you could limit entry to the letters M or F, or to a certain range of numbers.)

(1 of 2)

Attribute	Usage
PICTURE	Specifies an edit pattern or *format string*, such as (###) ###-#### for a USA telephone number. (The user is allowed to type only letters or digits as the pattern dictates. The punctuation you supply, such as parentheses or hyphens, is filled in automatically as the user types.)
REQUIRED	Specifies that user is not allowed to press **Accept** without having entered some value to the field.
VERIFY	Makes the user enter data in the field twice, checking that it is identical the second time.

(2 of 2)

You can also program specific input-editing rules by writing AFTER FIELD blocks in the INPUT statement.

Default Values

In the ATTRIBUTES section, you can specify a default value for a field. The value will be filled in automatically when an INPUT operation begins, and the user can replace it. Alternatively, you can DISPLAY data in fields prior to input, and use the displayed data as default values.

The INSTRUCTIONS Section

You use the INSTRUCTIONS section of the form for special features: field delimiters, screen records, and screen arrays.

Field Delimiters

Normally when a form is displayed, fields are displayed with the same square brackets that you used in the SCREEN section to define them. In the INSTRUCTIONS section, you can specify different delimiters. Most often this feature is used to change the delimiters to spaces, thus causing the fields to be drawn without any visible markers.

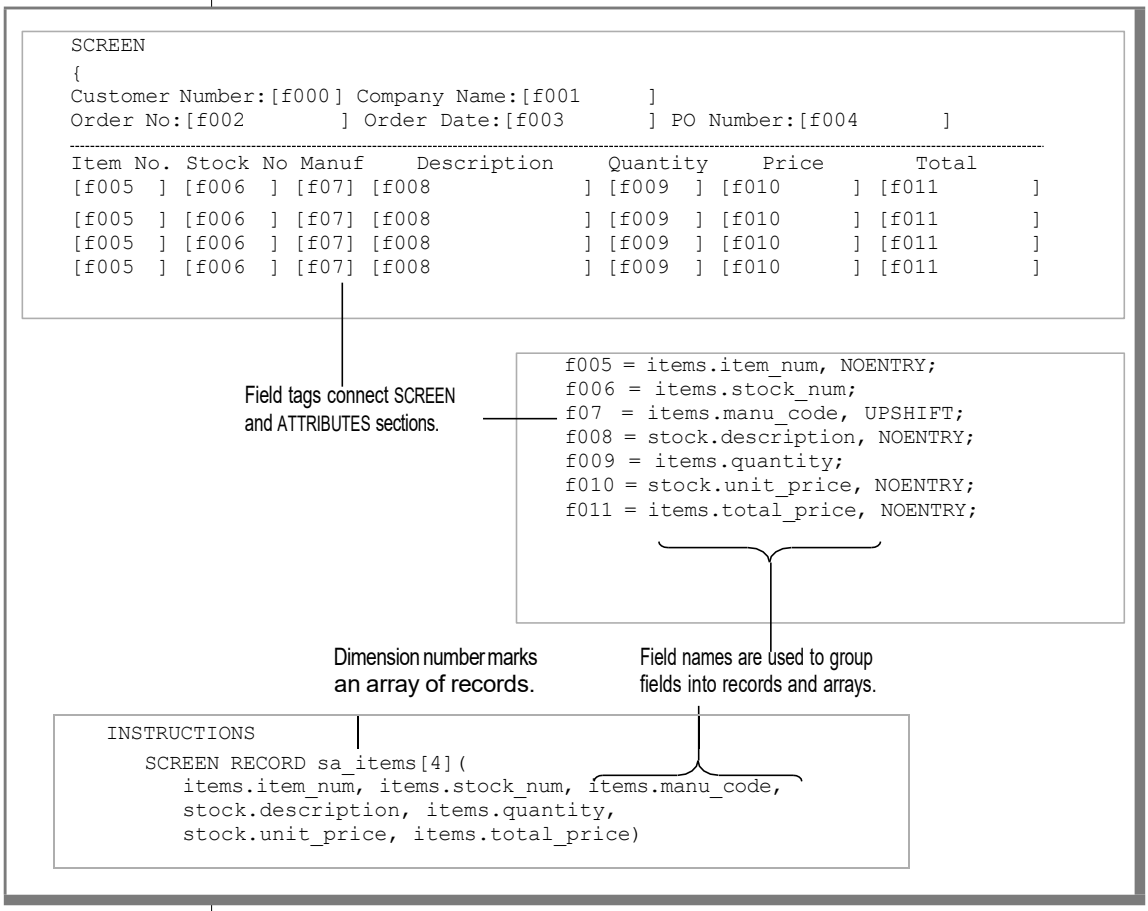
Screen Records

A screen record is a group of fields that you want your program to treat as a unit. A screen record, like a program variable that is a record, is a collection of named members. You can display all the members of a record variable in the matching fields of a screen record with a single DISPLAY statement. You can request input from all the fields of a screen record, and have the input values deposited in the matching members of a program variable, with one INPUT statement.

Screen Arrays

A screen array is a group of screen records that you want to treat as a scrolling table. The form shown on Figure 11-1 on page 11-4 has a screen array defined this way. Figure 11-2 on page 11-14 shows the relationship between array fields as they appear in the SCREEN section, the ATTRIBUTES section, and the INSTRUCTIONS section of a form.

Figure 11-2
Array Fields in SCREEN, ATTRIBUTES, and INSTRUCTIONS Sections of a Form

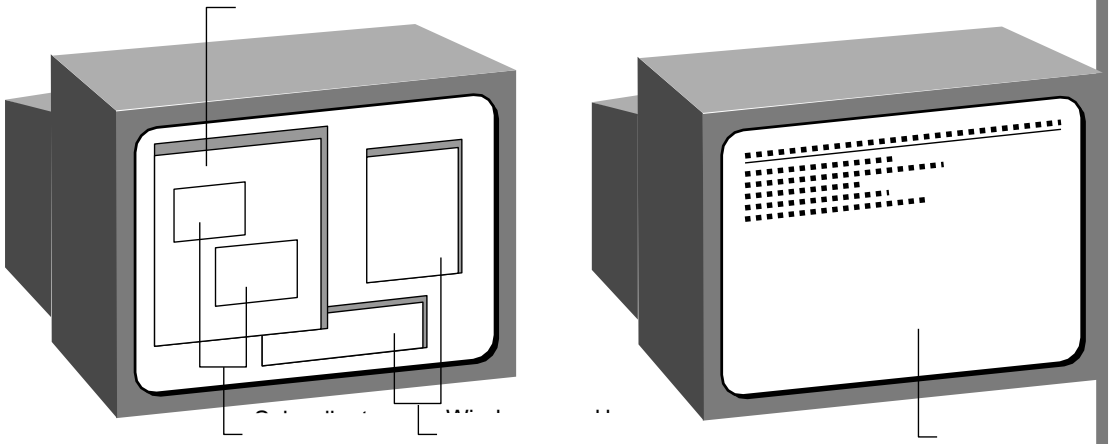


To use the array, your program must contain an array of records with similar members. It can use the DISPLAY ARRAY statement to display all the records in the array in the rows of the screen array.

Using Windows and Forms

As Figure 11-3 ilustrates, for character-based terminals and the workstation that emulate them, all of your screen output takes place in a single windo . If the output device is a real terminal, its screen is the window. If the outp device is a graphical screen emulating a terminal in a graphical window, th the screen is *that* window. This section discusses the ways your program c make the best use of 4GL windows.

Figure 1 3
The 4GL Screen on a Workstation and a Term l



The program is displayed on the 4GL screen. However, you can further control program output by creating additional rectangular areas within the confines of the screen. These can also be considered 4GL *windows*. The size and contents of 4GL windows within it, are under the control of your program.

An application can have many 4GL windows. They can be the same size as or smaller than the 4GL screen. They can overlap. You can bring a particular window to the top, so that it is fully visible, or completely cover a particular window. And 4GL windows can be closed and opened, depending on the requirements of the application.

Opening and Displaying a 4GL Window

The initial 4GL window is created automatically when 4GL first encounters an I/O statement. This first 4GL window fills the screen. In the statements you use for controlling windows, you can refer to this initial 4GL window as SCREEN.

The program cannot find out what that size is. Most programs assume that it allows 24 lines of 80 columns because that is the size of most character-based terminals. If your program contains forms or reports that require a larger window, it might not run on some terminals.

Opening Additional 4GL Windows

You can open additional 4GL windows with the OPEN WINDOW statement. This statement is covered in detail in *INFORMIX-4GL Reference*. An example of the statement follows:

```
OPEN WINDOW stockPopup AT 7, 4 WITH 12 ROWS, 73 COLUMNS
   ATTRIBUTE(BORDER, FORM LINE 4)
```

You can specify the following things when you open a new 4GL window:

- **Location.** With the AT clause, you specify the location of the upper-left corner (1,1) of the new window in relation to the screen. The units are character positions.
- **Size.** In the WITH clause, you specify the size of the window in one of two ways: with a specific number of rows and columns, or by specifying a form. If you specify FORM or WINDOW WITH FORM, the screen dimensions of that form establish the size of the new 4GL window (see "Specifying Screen Dimensions" on page 11-6).
- **Border.** In the ATTRIBUTE clause, you can specify a border for the window. The statement in the preceding example opens a bordered window.
- **Color.** In the ATTRIBUTE clause, you can also specify a color for all text displayed in the window. You cannot specify a background color, only the foreground, or text, color.
- **Line numbers.** You can set the locations of the standard lines for the menu, messages, prompts, and comments. These lines are discussed in detail in "Changing Screen Line Assignments" on page 11-33.

4GL Window Names

The first argument of OPEN WINDOW is a name for the window. You can use this argument to assign a global name to the window.

The following line specifies **errorAlert** as a global name for a window:

```
OPEN WINDOW errorAlert AT 10, 20 WITH 4 ROWS, 40 COLUMNS
```

Anywhere else in the program (even in another source module), you can write a statement that refers to the **errorAlert** window. To make a window current, you would enter:

```
CURRENT WINDOW errorAlert
```

Controlling the Current 4GL Window

Only one 4GL window can be current at a time. When you open a new 4GL window, it becomes current. It is *on top* visually, covering any other 4GL windows that it overlaps.

The current window receives all output of the DISPLAY, PROMPT, MENU, and MESSAGE statements. It is used by all INPUT statements. An error occurs if the program tries to use a form field when that field is not part of the form in the current window.

Clearing the 4GL Window

You can clear all displayed text from a window with the CLEAR WINDOW statement. This removes all output, including menus, form fields, and labels. The window being cleared need not be the current window.

If you clear the current window from within a MENU statement, the menu will be redisplayed. This is not true of forms; you must redisplay a form explicitly.

Closing the 4GL Window

To remove a window, use the CLOSE WINDOW statement. It makes the window invisible and unusable until you re-create it with OPEN WINDOW. When you close a window, the next window below it becomes the current window.

This means that if you are using a form or menu (and hence using the current window), and you call a subroutine that opens a window, uses it, and closes it again, the original window will again be current when the subroutine returns. This is the behavior you would expect.

However, if you are using a window and call a subroutine that makes another window current and does not close it, the wrong window will be current when the subroutine returns, and an error might follow.

Displaying a Menu

The key concepts of menus are included in "How Menus Are Used" on page 7-10. That topic also includes an example of the code you use to create a menu. The details of the MENU statement are in *INFORMIX-4GL Reference*.

When you execute the MENU statement, a ring menu is displayed on the assigned Menu line of the current window. Normally, you will display a menu across the top line of a window, usually above the display of a form. Other ways to use menus are as follows:

```
FUNCTION alertMenu(msg , op1 , op2 , op3)
   DEFINE   windowWidth, indent SMALLINT ,
            ret , msg , op1 , op2 , op3 CHAR(20)
   LET windowWidth = 40
   LET indent = (windowWidth-LENGTH(msg))/2
   OPEN WINDOW alert2 AT 10,20 WITH 5 ROWS, windowWidth COLUMNS
      ATTRIBUTE (BORDER, MENU LINE LAST)
   DISPLAY msg CLIPPED AT 3,1
   MENU "Respond"
   COMMAND op1
      LET ret = op1EXIT MENU
   COMMAND op2
      LET ret = op2EXIT MENU
   COMMAND op3
      LET ret = op3 EXIT MENU
   END MENU
   CLOSE WINDOW alert2
   RETURN ret
END FUNCTION
```

The **alertmenu()** function is given a short message and three choices, as the example shows.The choices would normally be keywords such as OK, NO, CANCEL, or SAVE. The function opens a small 4GL window. In the window, it displays the message line below a menu composed of the three choices.

The choices in the menu are not constants as is usually the case, but variables, specifically, the function arguments. An example of how the function could be called follows:

```
MAIN
DEFINE ret CHAR(20)
CALL alertMenu("alert message here" , "FIRST" , "SECOND" , "THIRD")
   RETURNING ret
END MAIN
```

The function would display a window that looks like Figure 11-4.

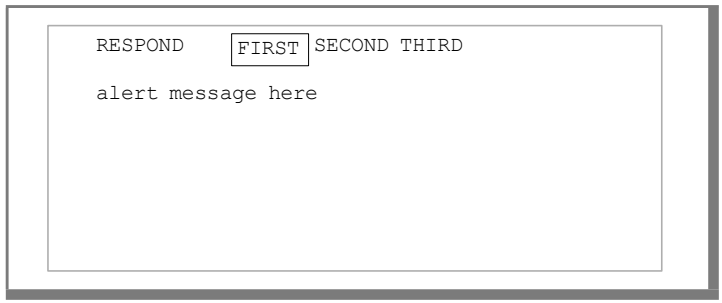


Figure 11-4
An Alternative Way of Using the MENU Statement

Whichever menu option the user chooses is returned as the function's result.

Note that although the function opens its window with the attribute MENU LINE LAST, the menu begins on the next-to-last line. See "Changing Screen Line Assignments" on page 11-33.

Opening and Displaying a Form

It takes only two program statements to open and display a form. The OPEN FORM statement brings the compiled form into memory by using a command such as:

```
OPEN FORM orderFrm FROM "forders"
```

This statement causes 4GL to search for a compiled form file named **forders.frm**. The file suffix **.frm** is supplied automatically and must not be used. 4GL looks first in the current directory and then in directories named in the **DBPATH** environment variable.

Form Names and Form References

The first argument of OPEN FORM is a name for the form. You use that name to manage the form later in the program.

When you specify a form name in conjunction with the OPEN FORM statement, it becomes the global name of the form:

```
OPEN FORM orderForm FROM "forders"
```

Anywhere else in the program (even in another source module), you can write a statement that refers to this form.

Displaying the Form

After opening a form, you can use DISPLAY FORM to display it in the current window:

```
DISPLAY FORM orderForm
```

The current 4GL window is cleared from the Form line to the bottom. The fields and labels of the form are drawn on the current window, starting at the Form line. The fields are initially empty.

The form must fit within the current 4GL window, allowing for reserved lines. For more information, see "Specifying Screen Dimensions" on page 11-6 and "Opening Additional 4GL Windows" on page 11-16.

By default, the Form line is the third line of the 4GL window, but you can change it. For more information, see "Changing Screen Line Assignments" on page 11-33.

You can display a form repeatedly in one window or in different windows. However, you can only display one form per 4GL window.

Displaying Data in a Form

When the form has been displayed, your program can display data in its fields. The DISPLAY TO statement is used for this. It takes a list of expressions and a list of field names in which to display them. You can display data in one field at a time.

```
DISPLAY "Salaried" TO emp_status
```

More often, you display a list of expressions, in the form of variables, in a list of fields:

```
DISPLAYtheCustNum, theCustName, 0, 0
   TO customer_num, company, order_amt, order_total
```

You can display all the elements of a record in the fields of a screen record with one statement.

A common technique is to use the names of database columns as the names of both the fields in a form and the members of a record. The following example shows how a record variable is defined to contain one member for each column in a table:

```
DEFINE custRow RECORD LIKE customer.*
```

If the form in the current window has a screen record with corresponding fields (see "Screen Records" on page 11-13), you can display all the members of this record variable in the fields of the screen record with one statement:

```
DISPLAY custRow.* TO custFields.*
```

Alternatively, when the current form has fields whose names are the same as the members of the record, you can display all the members this way:

```
DISPLAY BY NAME custRow.*
```

The BY NAME clause can be used whenever you want to display program variables in fields that have the same names.

Changing Display Attributes

4GL supports visual attributes such as REVERSE and BOLD, and a range of colors when the output device supports them. These display attributes can be assigned to a 4GL window, to an entire form, or to one or more individual fields. The following techniques will help you make the best use of attributes:

- To set display attributes for the entire 4GL application, use the OPTIONS statement before opening any windows.
- To set display attributes for all text in one window, use the ATTRIBUTE clause of the OPEN WINDOW statement.
- To set display attributes for all lines of a form without changing the attributes of other lines of the window, use the ATTRIBUTE clause of the DISPLAY FORM statement.
- To set specific display attributes for one field of a form, use the REVERSE or the COLOR clause for that field in the ATTRIBUTES section of the form specification file. The COLOR clause takes a WHERE keyword, so you can make the color of the field dependent on its contents or on the contents of other fields.
- To override the display attributes of specific fields as you display data in them, use the ATTRIBUTE clause of the DISPLAY TO statement.

The preceding methods of setting visual attributes are the most important. There are others, and the precedence among methods is more complicated than this list shows. For more information on visual attributes, see *INFORMIX-4GL Reference*.

Combining a Menu and a Form

It is common to have both a menu and a form in the same 4GL window. The form fields provide the structure for displaying information. Your user can choose menu options to tell the program what information to display.

A common example is a program that lets the user browse through a series of rows from the database, one row at a time. Several of the programming examples in *INFORMIX-4GL by Example* are devoted to just this problem. The basic technique is as follows:

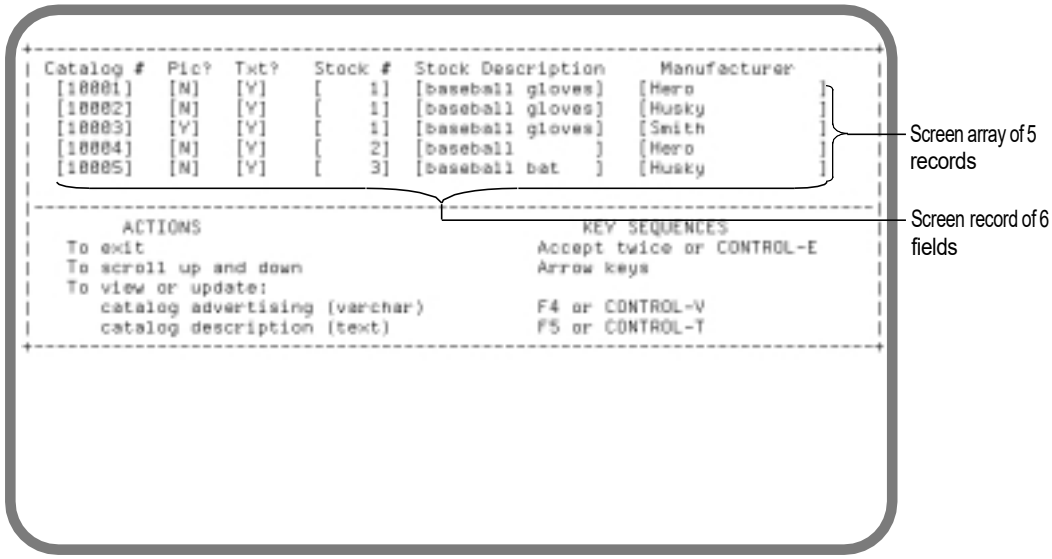
1. Set up a database cursor to represent the selected set of rows.
2. Display a form that has fields for the columns of one row.
3. Execute a MENU statement that includes an option such as **Next** to cause the display of the next row.
4. In the COMMAND block for the **Next** menu option, you use:
   - FETCH to get the next row from the cursor.
   - DISPLAY to show the fetched values in the form.

Thus each time the user selects menu option **Next**, the program displays a new row. By using a SCROLL cursor (which supports backward as well as forward movement through the rows) you can easily provide menu choices for **Prior**, **First**, and **Last** row displays.

Displaying a Scrolling Array

Your 4GL program might frequently need to display a scrolling list of rows. The screen array is used for this. In the form in Figure 11-5, the rows of the **catalog** table from the 4GL demonstration database are being scrolled through a screen array. The form specification file for this form is from Example 18 of *INFORMIX-4GL by Example*.

Figure 11-5
A Scrolling Array from the catalog Table

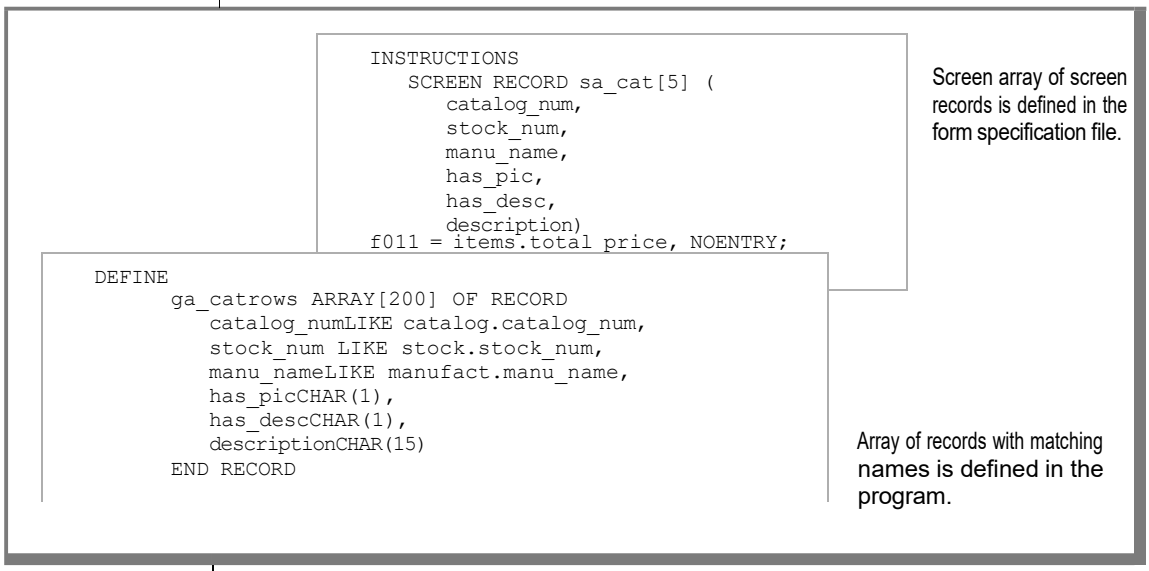


To implement a scrolling display like this, your program must do two things:

- Display a form containing a screen array of screen records.
- Define an array of records, each record having members that correspond to the fields of the screen records.

In the example, these two things are accomplished in the manner shown in Figure 11-6.

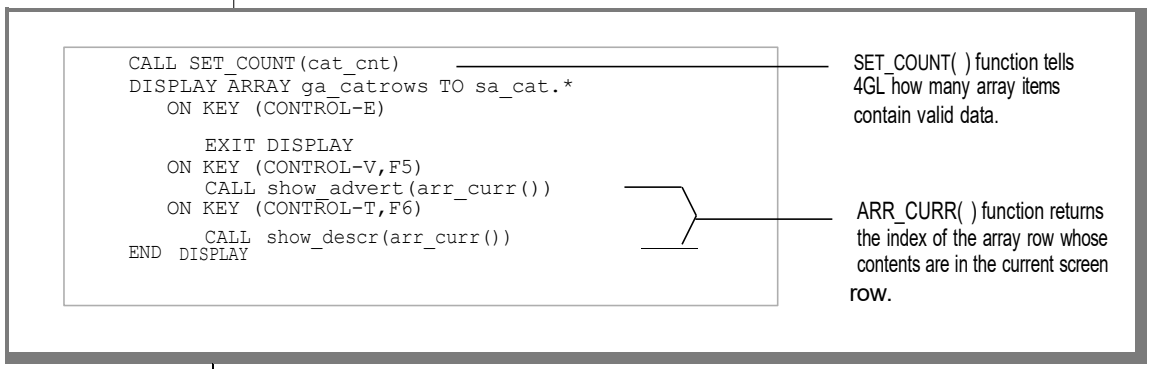
Figure 11-6
Definitions of a Screen Array and a Record Array



The screen array has 5 records; the array in the program has 200. The members of the records have the same names and appear in the same order.

The program uses a FOREACH loop to fill the array with rows fetched from the database (refer to "Row-by-Row SQL" on page 9-5). Once the program array has been loaded with data, the scrolling display can be started with the DISPLAY ARRAY statement. But first the program must call the built-in function SET_COUNT( ) to tell 4GL how many records in the array have useful data, as illustrated in Figure 11-7. Only these rows will be shown in the display.

Figure 11-7
The SET_COUNT( ) Function and the ARR_CURR( ) Function



During a DISPLAY ARRAY statement, 4GL interprets the keystrokes used fo scrolling (up and down arrows and others). It responds to them by scrollin the rows from the array in memory through the lines of the screen array.

As shown in the previous example, you can write ON KEY blocks within DISPLAY ARRAY to act on specific keystrokes. In the example, if the user presses either CONTROL-V or F5, the program calls a function named **show_advert( )**. If you read Example 18 in *INFORMIX-4GL by Example*, you will see that this function opens a new window to display an expanded view of one column.

Taking Input Through a Form

Once your program displays a form, it can take input from the user through the form fields. In its simplest form, the INPUT statement, like the DISPLAY statement, takes a list of program variables and a list of field names.

```
INPUT stockNum, quantity FROM stock_num, item_qty
```

In this example, two fields—**stock_num** and **item_qty**—are enabled for input. Fields by these names must, of course, exist in the form displayed into the current window.

The program waits while the user types data into the field and presses the **Accept** key. (The **Accept** key is ESCAPE by default, but the actual keyboard assignment can be changed; see "Runtime Key Assignments" on page 11-36). The data is assigned into the program variables **stockNum** and **quantity** and the program proceeds.

INPUT supports the same shortcuts for naming records as DISPLAY does. You can ask for input to all members of a record, from all fields of a screen record, and you can ask for input BY NAME from fields that have the same names as the program variables.

Help and Comments

In the ATTRIBUTES section of a form, you can specify an explanatory comment for any form field. These comments are displayed during input. When the cursor enters a field, the comment for that field is displayed on a specified line of the screen, the Comments line. This line is by default the last line of the window, but can be changed; see "Changing Screen Line Assignments" on page 11-33.

You can also associate a help message with any INPUT operation. See "How the Help System Works" on page 7-28.

Keystroke-Level Controls

Some programs require precise control over user actions during input. You can do this, too, by writing one or more ON KEY blocks as part of the INPUT statement. 4GL executes the statements in your ON KEY code block whenever the user presses one of the specified keys during input.

A typical use for an ON KEY block is to display special assistance. You can tell your user something like `Press CONTROL-P for a list of price codes.` In an ON KEY block for the CONTROL-P key, you can open a 4GL window and display in it the promised list. After getting the necessary information, your user can finish entering data and terminate the INPUT statement by pressing the **Accept** or **Cancel** key.

Field-Level Control

Sometimes you want to make a form even more responsive to the user, or you might require more detailed control over the user's actions:

- To make a form seem lively and "intelligent," you want to cause a visible response to the user's last action, if possible anticipating the user's likely next action.
- To catch errors early, saving the user time, you want to verify each value as soon as it is entered, with respect to values in other fields.

To achieve this level of control, write BEFORE FIELD and AFTER FIELD blocks of code as part of the INPUT statement. These are groups of 4GL statements that are called automatically as the user moves the cursor through the fields on the form.

Using a BEFORE FIELD Block

A BEFORE FIELD block is executed as the cursor is just entering a field. In the following example, a message is displayed when the cursor enters a field, and is removed when the cursor leaves the field:

```
INPUT...
   ...
   BEFORE FIELD customer_num
      MESSAGE "Enter customer number or press F5 for a list."
   AFTER FIELD customer_num
      MESSAGE ""
```

You could get the same effect by writing the message as a COMMENT attribute for this field in the form specification. But if you did that, the message would be displayed whenever the form is used. In this case, the popup list of customers is a service offered only within this particular INPUT statement. The same form might be used in other contexts where you do not mean to do anything special for an F5 key.

A typical use of BEFORE FIELD is to prepare likely default values. As the cursor enters a shipping-charge field, the program calculates, stores, and displays an estimated charge. This is done only if no value has previously been entered to the field.

```
BEFORE FIELD shipCharge
   IF shipRec.shipCharge IS NULL THEN
      LET shipRec.shipCharge =
             shipEstCalc(shipRec.shipWeight,custRec.state)
      DISPLAY BY NAME shipRec.shipCharge
   END IF
```

Using an AFTER FIELD Block

An AFTER FIELD block is called as the cursor is just leaving a field. In it, you can write statements that perform the following tasks:

- Check the value of the field for validity with respect to other form fields and with respect to the database.
- Display values in other fields as a result of the value just entered into this one.

The following simple example shows validation:

```
AFTER FIELD customer_num
-- Prevent user from leaving an empty customer_num field
   IF gr_customer.customer_num IS NULL THEN
      ERROR "You must enter a customer number. Please do so."
      NEXT FIELD customer_num
   END IF
```

The NEXT FIELD statement sends the cursor to the specified field (in this case, back to the field it had just left). It terminates execution of the AFTER FIELD block and starts execution of the BEFORE FIELD block of the destination field.

The block from which the preceding example is taken (Example 15 in *INFORMIX-4GL by Example*) does more. When a customer number is entered, it performs the following tasks:

- Uses SELECT to read that customer's row from a database table
- If no row exists, displays the fact and uses NEXT FIELD to repeat the input
- Initializes other fields of the form with data from the database row

Field Order Constrained and Unconstrained

In a BEFORE or AFTER FIELD block or an ON KEY block, you can also write a NEXT FIELD statement, forcing the cursor to move to a particular field. Many existing 4GL programs control the cursor in this way. It is done to direct the user's attention to important data, or to require the user to enter certain data.

This tight control over actions of your user is a natural way to manage computer interaction on a character-based terminal. Programmers can assume that the cursor never enters the **Discount** field without first having passed through the **Customer Number** field, for example. The BEFORE FIELD block for **Discount** could therefore refer to the value of **Customer Number** with certainty that it was present.

Through the 4GL OPTIONS statement you can adjust the amount of freedom your user will have in moving through a form. If you set FIELD ORDER CONSTRAINED, you can accurately predict the path your user will follow when moving through a form.

If, however, FIELD ORDER UNCONSTRAINED is set, the user will be able to move through the form in any particular order using the arrow key.

Taking Input Through an Array

The DISPLAY ARRAY statement lets the user view the contents of an array of records, but the user cannot change them. You can use INPUT ARRAY to allow the user to alter the contents of records in the array, to delete records, and to insert new records.

The preparation for array input is similar to that for DISPLAY ARRAY:

1. You design a form containing a screen array of screen records.
2. You define a program array of records. The members of each record match the fields of the screen record.

3. If the user can alter existing data, you pre-load data into the program array and use the built-in SET_COUNT() function to specify how many array items contain data.
4. You execute an INPUT ARRAY statement, naming the program array as a receiving variable and the screen array as its corresponding field.

The same INPUT statement can also name ordinary program variables corresponding to ordinary form fields. The user directs the cursor through the fields as usual.

When the cursor enters the screen array on the form, 4GL handles the scrolling of the array, allowing the user to edit and change the contents of the fields by using arrow keys to navigate through the cells of the screen array. You can control and monitor these changes with BEFORE and AFTER FIELD blocks as usual. The built-in ARR_CURR() function is available to tell you which array element is being changed. Besides the NEXT FIELD statement, you can execute the NEXT ROW statement to reposition the cursor to a different row.

4GL also supports an **Insert** key to open a new, empty row in the array, and a **Delete** key to delete a row. You can monitor and control these actions with BEFORE and AFTER INSERT and DELETE blocks. You can redefine the **Insert** or **Delete** key using the OPTIONS statement.

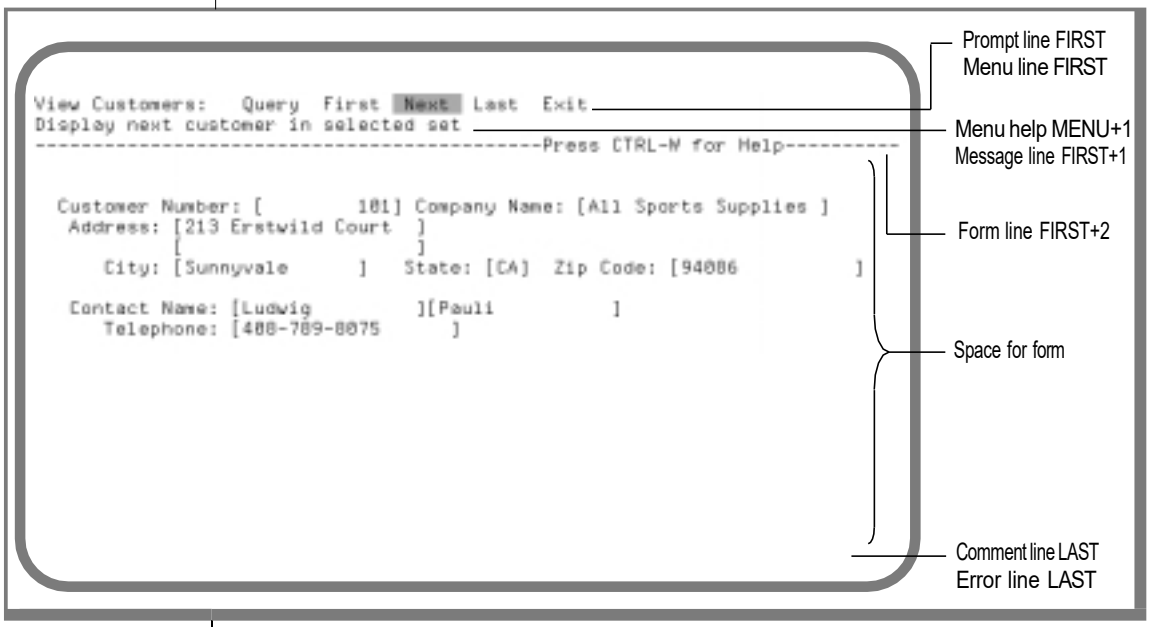
Screen and Keyboard Options

Now that you understand how 4GL uses the screen, a review of options for customizing the user interface follows.

Reserved Screen Lines

Figure 11-8 shows a summary of the *reserved*, or *dedicated*, screen lines.

Figure 11-8
Reserved Screen Lines



- **Prompt line**. The PROMPT statement prompts the user for input to one or more variables. It displays its prompt string on the specified line and the user types a response on the same line.
- **Menu line.** The MENU statement displays a "ring" (horizontal) menu on this line. The user moves the cursor from choice to choice with the TAB and arrow keys.

  MENU and PROMPT can share the same line. Each clears the line and rewrites it as necessary.

- **Menu Help line.** In the MENU statement, you can also write an explanatory phrase for each menu choice. As the user moves the cursor across menu options, the explanations are displayed on this line. The Menu line is always the line immediately below the menu.
- **Message line.** The MESSAGE statement is used to show a short message to the user. The Message line can share the line after the Menu line because the MENU statement rewrites its help line when it starts or resumes. The Message line could also be the same as the Comment line.
- **Form line.** The *top line* of a screen form is displayed on this line by the DISPLAY FORM statement.
- **Comment line.** When designing a screen form, you can specify an explanatory comment for any field. As the user moves the cursor through the fields of the form, these explanations are displayed on the Comment line.
- **Error line.** The ERROR statement is used for providing the user with a message serious enough to warrant an audible warning. The Error line and the Comment line should not be assigned the same position. Otherwise, if an error message appears in a field that has the COMMENTS attribute, comment text overwrites the error message.

The Error line is special in another way. All other reserved lines represent positions within the current 4GL window where text can be written, replacing previous text on that line. The Error line specifies a position where a one-line error message appears on the physical screen, without regard to the position or size of the current 4GL window.

Changing Screen Line Assignments

The default layout of reserved screen lines is usually satisfactory. If you change it, you should make sure that your new layout is consistent across your application and with other applications that your users will see. You can change the assignment of logical lines to line numbers in two ways, depending on your needs:

- With the OPTIONS statement, you can change the assignment of one or more lines for all windows.
- With the ATTRIBUTES clause of OPEN WINDOW, you can assign the logical lines for one window when you create it.

There are, however, some line assignments you cannot change. For example, the Menu Help line always follows the Menu line, so the Menu line can never be last. If you specify MENU LINE LAST, 4GL treats it as if you had assigned LAST-1.

Getting the Most on the Screen

The most common reason for changing screen line assignments is to increase the maximum number of lines available for other purposes. To do this, you make multiple screen lines use the same row. The following table shows which lines can share the same screen row.

	Menu	Menu Help	Message	Form	Comment	Error
Prompt	Yes	Yes	Yes	Note 1	Yes	Yes
Menu		No	Note 2	Note 1	Note 3	Yes
Menu help			Note 2	Note 1	Yes	Yes
Message				Note 1	Yes	Yes
Form					Note 1	Yes
Comment						Yes

In this table, the word *Yes* means that the two intersecting types of screen lines can share the same screen row because the lines appear at different times, and each clears the row before using it. Potential problems are discussed in the following notes:

1. The screen form is not automatically redrawn after it has been overwritten. If you display a prompt, menu, message, or comment in a line used by a form, the only way to restore the complete form is to redisplay the form and then redisplay the data in its fields.
2. If you make the Message line the same as Menu or Menu Help, you must be careful when using the MESSAGE statement from within MENU. You must program a delay before resuming the menu operation. Otherwise the menu will replace the message text too quickly for the user to read it. If messages and menus are used at different times, there is no difficulty having them using the same row.
3. When you use both a menu and a form, you should probably not make the Comment and Menu lines the same. You can make the Comment line the same as the Menu Help line. Then both types of explanations appear on the same screen row.

The error text is always displayed on the designated Error line of the physical screen. When you design a 4GL window as described in "How 4GL Windows Are Used" on page 7-25, you do not need to allow for an Error line.

Runtime Key Assignments

The 4GL runtime environment uses several logical function keys and provides default keyboard assignments. These can easily be reassigned. The abstract function keys are summarized in the following table.

Key Name	Purpose of Key	Default Keystrokes
Accept	Selects the current menu option in a statement; terminates input during CONSTRUCT, INPUT, and INPUT ARRAY; terminates DISPLAY ARRAY	ESCAPE
Interrupt	Represents the external interrupt signal; available when interrupts are deferred with the DEFER statement	stty interrupt key (usually CONTROL-C)
Insert	Requests insertion of a new line during INPUT ARRAY, starting execution of a BEFORE INSERT block	F1
Delete	Requests deletion of the current line during ARRAY, starting execution of a BEFORE DELETE block	F2
Next	Causes scrolling to the next page (group of lines) during DISPLAY ARRAY and INPUT ARRAY	F3
Previous	Causes scrolling to the previous page (group of lines) during DISPLAY ARRAY and INPUT ARRAY	F4
Help	Starts the display of the specified help message from the current help file	CONTROL-W

You can change the assignment of the logical keys to physical keystrokes with the OPTIONS statement. The two common problems that require you to change them are:

- The ESCAPE key is often a prefix value for function keys.

  The operating system might wait a fraction of a second after ESCAPE is pressed, in order to make sure it is not the start of an escape sequence, before passing it to the program. On some systems, this can cause a delay in the response of your program to the **Accept** key.

- The numbering of function keys is not consistent from one version of UNIX to another.

  Some terminals might have different physical keys than those defined in the **termcap** file.

Dedicated Keystrokes

The following physical keys have dedicated uses during some 4GL statements.

Key Name	Use in INPUT, INPUT ARRAY, and CONSTRUCT	Use in MENU
CONTROL-A*	Switches between overtype and insert modes	None
CONTROL-D*	Deletes from the cursor to the end of the field	None
CONTROL-H* (backspace)	During text entry, moves the cursor left one position (nondestructive backspace)	Moves highlight to next option to the left
CONTROL-I* or TAB*	Moves the cursor to the next field, except in a WORDWRAP field, where it inserts a tab or skips to a tab, depending on mode	None
CONTROL-J (Linefeed)	Moves the cursor to the next field, except in a WORDWRAP field, where it inserts a new line or moves down one line, depending on mode	Moves the highlight to the next option to the right

An asterisk indicates that the key cannot be used in an ON KEY clause.

(1 of 2)

Key Name	Use in INPUT, INPUT ARRAY, and CONSTRUCT	Use in MENU
CONTROL-L*	During text entry, moves the cursor right one position	Moves the highlight to the next option to the right
CONTROL-M or RETURN	Completes entry of the current field; cursor moves to next field (if any; otherwise, same as **Accept**)	Accepts the option that is currently highlighted
CONTROL-N	Same as CONTROL-J	None
CONTROL-R*	Causes the screen to be redrawn	Causes the screen to be redrawn
CONTROL-X*	Deletes the character under the cursor	None
LEFT ARROW	Same as BACKSPACE	Same as BACKSPACE
Right Arrow	Same as CONTROL-L	Same as CONTROL-L
Up Arrow	Usually moves to the previous field, except in a WORDWRAP field, where it moves up one line in the field, and in an INPUT ARRAY, it moves to the corresponding field in the previous row	Moves the highlight to the next option to the left
DOWN ARROW	Usually moves to the next field, except in a WORDWRAP field, where it moves down one line in the field, and in an INPUT ARRAY, it moves to the corresponding field in the next row	Moves the highlight to the next option to the right
INSERT	Same as the logical **Insert** key	None
DELETE	Same as the logical **Delete** key	None
PGUP	Same as the logical **Next** key	None
PGDN	Same as the logical **Previous** key	None

An asterisk indicates that the key cannot be used in an ON KEY clause.

(2 of 2)

Intercepting Keys with ON KEY

The names in the first column of the preceding table are accepted in an ON KEY clause, with the noted exceptions. If you redefine these keys using ON KEY, they lose their dedicated abilities. For example, if you redefine the UP key in a DISPLAY ARRAY statement, the user will have no way to move the cursor upward. Similarly, if you specify RIGHT in a KEY clause of a MENU COMMAND, the user will not be able to move through the menu using the RIGHT ARROW key.

Chapter

12

Handling Exceptions

In This Chapter

Exceptions, usually referred to as *errors*, are unusual occurrences that you might sometimes wish would never happen to your program. Of course, you know they will happen, and you know you need to write your programs so they behave in reasonable ways when errors and other unplanned for events occur.

This chapter reviews the categories of exceptional conditions and how 4GL reacts to them when you do not specify what to do. It then details the mechanisms that 4GL gives you for handling them, as follows:

- The DEFER statement allows you to convert asynchronous signals into synchronous flags that you can poll.
- The WHENEVER statement lets you change how 4GL responds to specific error conditions.

The following table summarizes the types of exceptions 4GL can recognize and the flags it can test and set in each case.

Error Type	Flag Checked
Interrupt signal	**int_flag**
Quit signal	**quit_flag**
Runtime error, expression error, file error, display error, initialization error	**status**
SQL error	**status**, **SQLCA.SQLCODE**
SQL warnings	**SQLCA.SQLAWARN**
SQL end of data error	**status=NOTFOUND** or **SQLCA.SQLCODE=NOTFOUND**

Exceptions

You can write your 4GL program to recognize and respond to the following types of exceptions:

- Runtime errors (sometimes called execution errors)
- SQL end-of-data conditions
- SQL warnings
- Asynchronous signals, meaning signals from the keyboard or elsewhere, occurring at an unplanned for time

Runtime Errors

Runtime, or execution, errors are serious conditions that are detected by the database server or by 4GL at runtime. Although they are divided into several categories, these errors all have one thing in common: if they occur and the program does not explicitly handle them, the program will terminate immediately. To handle these types of errors, you must use the WHENEVER ERROR statement. For more information, see "Using the WHENEVER Mechanism" on page 12-10.

A negative error code number is associated with every type of execution error. For every number, there is an error message. Error numbers and their associated messages are available on-line. In addition, your 4GL application can call a function, ERR_GET( ), to retrieve the message text for any error number. For more information, see *INFORMIX-4GL Reference*.

Execution errors are divided into five groups based on the kinds of program actions that can cause them:

- **Expression errors.** These arise when 4GL attempts to evaluate an expression that violates the rules of the language. For example, error –1348 occurs when an attempt is made to divide by zero. Error -1350 occurs when 4GL cannot convert between data types.
- **File errors.** These arise when 4GL tries to access a file and the operating system returns an error code. For example, error -1324 means that a report output file could not be written.

- **SQL errors.** These arise when the database server detects an error in an SQL statement. For example, error -201 results from a syntax error in an SQL statement, while error -346 shows that an attempt to update a row in a table failed.
- **Screen errors.** These arise when something goes wrong with a screen interaction. For example, error -1135 means that the row or column in a DISPLAY AT statement falls outside the current 4GL window.
- **Initialization and validation errors.** The INITIALIZE and VALIDATE statements are used to initialize or test program variables against a special database table, **syscolval**. Errors in the operation of these statements are in a separate category; you handle them apart from other errors.

SQL End of Data

When the database server is unable to retrieve a specified row, it reaches an end-of-data condition and sets the **SQLCODE** member of the **SQLCA** record to `100`. 4GL includes a built-in constant called NOTFOUND that has a value of `100`.

By default, 4GL continues execution of your program when an end-of-data condition is encountered. However, you can test for this condition after the following statements:

- FETCH
- FOREACH
- SELECT

If the value of **SQLCA.SQLCODE** is greater than zero, there was no row available.

Alternatively, your program can treat an end-of-data condition as an error condition so that when it occurs, your program is diverted to code that handles it.

To change the default behavior of 4GL, use the WHENEVER NOT FOUND statement. For more information, see "Using the WHENEVER Mechanism" on page 12-10.

SQL Warnings

Some SQL statements can detect conditions that are not errors, but that can provide important information for your program. These conditions are called *warnings* and are signalled by setting warning flags in the **SQLAWARN** member of the **SQLCA** record.

As with end of data, you have a choice in how to treat an SQL warning. By default, 4GL sets **SQLCA.SQLAWARN** and continues execution of the program. You can insert code after any SQL statement to test the **SQLAWARN** flag values.

To change this default behavior, you can use the WHENEVER WARNING statement, described in "Using the WHENEVER Mechanism" on page 12-10.

Asynchronous Signals: Interrupt and Quit

External signals are *asynchronous* signals that—unless specifically *deferred*—are delivered by the operating system to a running program. An asynchronous signal is one that is not related to an action of the program.

Common external signals such as Interrupt and Quit would typically be generated by the user. The keystroke that generates an Interrupt signal is known as the **Interrupt** key; the keystroke that generates a Quit signal is known as the **Quit** key. On some systems, physical keys associated with external signals can be reassigned.

Generally speaking, unless intercepted, an external signal that reaches an application causes the application to terminate immediately. 4GL provides mechanisms for testing for and handling Interrupt and Quit signals.

At first glance, it is difficult to see why an external signal of any kind should be allowed to terminate a running program. But in fact, it is often quite useful to allow user-induced exceptions to end programs during certain stages of program development or debugging.

For example, if you create a FOREACH or WHILE routine without a termination point, you cannot stop the routine without killing the process or rebooting the system. However, if interrupts are not trapped, pressing the **Interrupt** key (CONTROL-C by default) ends the program immediately. Later, once you have perfected the routine, any external signals can be deferred using the 4GL DEFER statement (described in the next section) and more polite mechanisms put in place to handle unanticipated user requests.

Using the DEFER Statement

Because it is generally not convenient for the user to immediately terminate an application from the keyboard, 4GL provides a DEFER statement that:

- captures an Interrupt or Quit signal.
- sets the appropriate 4GL flag, allowing you to deal with the external signal programmatically.

The DEFER mechanism allows you to choose how to handle Interrupt or Quit signals. To enable this mechanism, you include the DEFER statement once, at the beginning of your 4GL program. This statement has two forms: DEFER INTERRUP and DEFER QUIT.

With the DEFER INTERRUPT statement, the **Interrupt** key combination is trapped and TRUE is assigned to the built-in global variable **int_flag**. After it has been deferred, the Interrupt signal has no effect on most program statements, but it does terminate some interactive statements, as described in the next section.

With the DEFER QUIT statement, the **Quit** key combination, CONTROL-\, is caught, and TRUE is assigned to the built-in global variable **quit_flag**.

Some systems can deliver other external signals, but these are not handled by the DEFER mechanism of 4GL. There are also synchronous signals that represent runtime errors that are trapped by the operating system, usually for major program faults such as indexing past the end of an array.

The DEFER statement can only appear in the MAIN statement. Its effect cannot later be undone. After its signal has been deferred, the physical **Interrupt** key or **Quit** key can be assigned to a logical key in an ON KEY clause.

Interrupt with Interactive Statements

The following table summarizes the effect of a user-generated Interrupt signal when DEFER INTERRUPT is in effect.

Statement	Description
INPUT	When INTERRUPT is named in an ON KEY clause, the signal is treated as just another keystroke. That is, DEFER can trap the signal. Otherwise, the signal terminates the INPUT statement. In this case, any AFTER INPUT block is executed, and control of program execution then passes to the next statement.
CONSTRUCT	When INTERRUPT is named in an ON KEY clause, the signal is treated as the activation key for that control block. Otherwise, the signal causes CONSTRUCT to end. Any AFTER CONSTRUCT block is executed, and control passes to the next statement.
DISPLAY ARRAY	When INTERRUPT is named in an ON KEY clause, the signal is treated as just another keystroke. Otherwise, the operation ends, and control passes to the next statement.
MENU	When INTERRUPT is named in a MENU clause, the signal is treated as just another keystroke. The menu operation continues, but INTERRUPT can be named in a COMMAND KEY list and can be used to initiate an action.
PROMPT	When INTERRUPT is named in an ON KEY clause, the signal is treated as just another keystroke. That is, DEFER can trap the signal. If an Interrupt signal is generated, the PROMPT statement terminates, and NULL is assigned to the receiving variable.

INTERRUPT with INPUT and CONSTRUCT

You will often use program logic similar to the following program fragment with INPUT or CONSTRUCT statements:

```
DEFER INTERRUPT
LET int_flag = FALSE
LET cancelled = FALSE
-- the Interrupt flag is specifically set FALSE
INPUT ...
   BEFORE INPUT
      MESSAGE "Use CTRL-E to cancel input."
   ON KEY (CONTROL-E)-- logical cancel
```

```
      LET cancelled = TRUE
      EXIT INPUT
   ...other clauses of INPUT...
   AFTER INPUT
-- tests to see if an Interrupt has been received (is TRUE)
      IF int_flag THEN
         LET cancelled = TRUE
      ELSE
         ...post-process based on Accept key...
      END IF
END INPUT
IF cancelled THEN
   MESSAGE "Input cancelled at your request!"
END IF
```

The code sample establishes three ways for the user to terminate the INPUT operation:

- **Accept key.** Normal completion of INPUT is signaled by pressing the **Accept** key. The statements in the AFTER INPUT block are used to perform any final validation of the entered data.
- **Interrupt key.** When the user generates an Interrupt signal, 4GL ends the INPUT operation, but in doing so it executes the AFTER INPUT block. The program checks the setting of **int_flag**. If TRUE, the INPUT operation is terminated. It exits the AFTER INPUT block early if it was entered due to an Interrupt signal.
- **CONTROL-E**. The program establishes a logical **Cancel** key based on using an unassigned control key trapped by an ON KEY block.

The example code converts both cancellations, the built-in cancellation due to the Interrupt signal and the programmed signal based on CONTROL-E, into a TRUE value in a variable named **cancelled**. This is value cleared before the INPUT operation begins, and is tested afterward.

When INTERRUPT is used in an ON KEY clause, INTERRUPT no longer terminates the operation, so the preceding example could be written in this way:

```
-- DEFER INTERRUPT run earlier
   LET cancelled = FALSE
   INPUT ...
      BEFORE INPUT
         MESSAGE "Use ctrl-E to cancel entry"
      ON KEY (control-E, Interrupt) -- logical cancel
         LET cancelled = TRUE
```

```
            EXIT INPUT
         ...other clauses of INPUT...
      END INPUT
      IF cancelled THEN
         MESSAGE "Input cancelled at your request!"
      END IF
```

This explicit handling of INTERRUPT with an ON KEY statement prevents execution of AFTER INPUT following INTERRUPT. Logic of either type can also be used with CONSTRUCT.

Deferred INTERRUPT with the MENU Statement

Normally the arrival of the Interrupt signal has no effect on a MENU operation. Typically a program should treat a user-generated **Interrupt** key as a sign that the user is at least impatient. You can write a MENU program block so that it treats INTERRUPT as a keystroke and uses it to exit the menu:

```
MENU "Scrolling menu"
COMMAND "First" "View first row of set" HELP 301
   ...
COMMAND "Next" "View next row of set" HELP 302
...etc etc
COMMAND KEY(ESCAPE, INTERRUPT) "Exit" "Exit this menu"
   EXIT MENU
END MENU
```

Using the WHENEVER Mechanism

The WHENEVER compiler directive handles various exceptional conditions:

- Runtime errors from the database server or from the 4GL program
- Warnings from the database server or from the 4GL program
- SQL end-of-data conditions

The WHENEVER directive is based on the ANSI/ISO standard for embedded SQL, which defines the keywords and basic operation of that statement. Informix has extended the standard syntax of WHENEVER to support additional keywords, conditions, and actions.

Like the declarations of variables or DEFER, the WHENEVER directive is not an executable statement.

What WHENEVER Does

Every WHENEVER compiler directive has the following logical format:

```
WHENEVER    condition    action
```

Here *condition* specifies some exceptional condition, and *action* specifies an action for the 4GL program to take when that condition is detected. One possible action is CONTINUE, which instructs 4GL to take no action.

After you issue the WHENEVER directive, you can write your program logic as if the specified condition would never happen. For details on how to use WHENEVER, see *INFORMIX-4GL Reference*.

Actions of WHENEVER

You can specify the following four possible types of actions using variations of the WHENEVER statement:

- WHENEVER … CONTINUE specifies that if the exceptional condition happens, it is to be ignored. 4GL sets the **status** variable and attempts to continue executing the program.
- WHENEVER … STOP specifies that if the exceptional condition happens, the program terminates. Any uncommitted database transaction is rolled back and an error message is displayed.
- WHENEVER … CALL *function_name* specifies that if the exceptional condition occurs, then the specified function should be called. The function that you specify cannot have arguments.
- WHENEVER … GOTO *label_name* specifies that if the exceptional condition occurs, then control of program execution should jump to a specified label within the same program block

Including *WHENEVER* in a source module changes the way that the compiler produces code, starting with that line, and continuing to the end of the module, or to the next *WHENEVER* directive for the same condition. If you specify an action other than to *do nothing* (*CONTINUE*), then the compiler automatically generates code to test for the exceptional condition after each statement that might cause it, and to carry out the specified action if the specified exceptional condition occurs.

Errors Handled by WHENEVER

The *condition* keyword of WHENEVER specifies what errors are to be handled. The keywords and the errors trapped are shown in this table.

Keyword	End of Data	SQL Warning	SQL Errors	Screen Errors	Initialize and	Expression Errors	File Errors
NOT FOUND	▬						
WARNING or SQLWARNING		▬					
ERROR or SQLERROR			▬	▬	▬		
ANY ERROR			▬	▬	▬	▬	▬

By default, 4GL executes a program that has no WHENEVER directives as if it contained the following WHENEVER directives:

```
WHENEVER NOT FOUND CONTINUE
WHENEVER WARNING CONTINUE
WHENEVER ANY ERROR CONTINUE
```

That is, all exceptions are ignored. Some errors, however, called *fatal errors*, cannot be trapped by WHENEVER; these errors are listed in *INFORMIX-4GL Reference*. A fatal error causes the program to terminate, regardless of what any WHENEVER directive specifies.

Using WHENEVER in a Program

In the next code example, a function contains two SQL statements, either of which could produce an error if the function arguments are not correct. For the purpose of this function, see "Dynamic SQL" on page 9-10.

To prevent runtime errors in these two statements from terminating the program, the WHENEVER compiler directive has been inserted with code to test the SQL error value after each statement.

```
FUNCTION tableGrant( )
   DEFINE whom , tab , priv , granText VARCHAR(255)
   LET granText = "GRANT " , priv,
      " ON " , tab ,
      " TO " , whom
WHENEVER SQLERROR CONTINUE
   PREPARE granite FROM granText
   IF SQLCA.SQLCODE == 0 THEN
      EXECUTE granite
      IF SQLCA.SQLCODE == 0 THEN
         RETURN -- all done in good style
      END IF
   END IF
WHENEVER SQLERROR STOP
   -- Some SQL error if we get here
   DISPLAY "The following prepared statement is in error:"
   DISPLAY granText
   DISPLAY "Error is ",SQLCA.SQLCODE USING "-<<<<<<",
      ", meaning:"
   DISPLAY ERR_GET(SQLCA.SQLCODE)
END FUNCTION
```

The WHENEVER directives are placed around the statements of interest. Here WHENEVER SQLERROR CONTINUE causes errors in the PREPARE or EXECUTE statements to be ignored. The only effect of an error will be to set a negative code in the SQL communications area and the **status** variable.

The WHENEVER SQLERROR STOP directive terminates the program if an error occurs. If it were omitted, then the effect of the first WHENEVER directive would continue to the end of the source module.

Using WHENEVER ERROR for Non-Fatal Errors

Most execution errors can be prevented from terminating the program by the WHENEVER ERROR directive, which establishes the policy of your program for handling runtime errors. For example, when an error is encountered at a given point in your program, you could take any of the following actions:

- Ignore errors.
- Call a function.
- Jump to a label.
- Display a PROMPT statement to get more guidance from your user.

The WHENEVER ERROR directive is a condensed way of putting an IF statement after every SQL statement in order to establish what action should be taken when errors occur.

Using WHENEVER ANY ERROR for Expression Errors

You can cause a variable to be checked after an error occurs in evaluating an expression by using the ANY keyword with a WHENEVER ERROR directive.

Using WHENEVER WARNING for SQL Warnings

By default, 4GL sets to `W` one or more characters in **SQLCA.SQLAWARN** and continues execution when it encounters a warning. Although a WHENEVER WARNING directive can be used to make the program call a function or go to a label when an SQL warning flag is set, normally you do not want the SQL warning flags to divert the program.

Using WHENEVER NOT FOUND for SQL End of Data

By default, 4GL sets **SQLCA.SQLCODE** (and **status**) to `100` (NOTFOUND) and continues execution when it encounters an end-of-data condition. Although the statement WHENEVER NOT FOUND can be used to make the program call a function or go to a label when SQL input finds an end-of-data condition, you normally do not want this condition to cause any diversion of the program.

Notifying the User

The 4GL Error line is a one-line display dedicated to the display of messages. (See "Screen and Keyboard Options" on page 11-32.) The Error line becomes visible in the following three cases:

- When you execute the ERROR statement to display a message that you composed
- When you execute the ERR_PRINT( ) function to display a message based on a 4GL error number
- When you execute the ERR_QUIT( ) function to display a message and terminate the program

In fact, the Error line is a one-line 4GL window that is automatically opened on the screen (see "Opening and Displaying a 4GL Window" on page 11-16). After an ERROR statement or call to ERR_PRINT( ), this window remains open and visible until a keystroke event occurs. Then it is closed, revealing any form or menu that might be hidden beneath it.

Logging Runtime Errors

You can maintain a file that lists sequentially all the error messages that are issued while the 4GL application is running. One easy way to do this is through the STARTLOG( ) function and involves the following steps:

1. Call the built-in STARTLOG(" *filename*") function from the MAIN statement, where *filename* specifies the file (and can also include a pathname) in which error messages will be logged.
2. After an error is detected, use the following assignment statement:

   ```
   LET variable = ERR_GET(status)
   ```

   where *variable* is a variable declared as data type CHAR and is of sufficient length to hold the text of the error message.
3. Call ERRORLOG(*variable*) to make a new entry in *filename*.

The last two steps are not needed if you are satisfied with the default error records that are logged automatically after STARTLOG( ) has been invoked. (If you do include the last two steps, then the first step is not required.)

In step 2, if *filename* matches no existing file specification, 4GL creates that file. If *filename* already exists, 4GL opens that file and positions the file pointer so that any subsequent error or warning message will be appended to this file. For your 4GL program to be portable, you should store *filename* in a variable, rather than calling STARTLOG( ) with a literal string as its argument.

For details of the ERR_GET( ), ERRORLOG( ), and STARTLOG( ) functions and for the default format of STARTLOG( ) error records, see *INFORMIX-4GL Reference*.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X