UNISYS

System V Operating System

Programmer's Guide

Volume 1

January 1988

Printed in U S America

Priced Item

UP–13689

Table of Contents

List of Figures

Chapter 1: Programming in a UNIX System Environment: An Overview

Introduction

The 1983 Turing Award of the Association for Computing Machinery was given jointly to Ken Thompson and Dennis Ritchie, the two men who first designed and developed the UNIX operating system. The award citation said, in part:

> "The success of the UNIX system stems from its tasteful selection of a few key ideas and their elegant implementation. The model of the UNIX system has led a generation of software designers to new ways of thinking about programming. The genius of the UNIX system is its framework which enables programmers to stand on the work of others."

As programmers working in a UNIX system environment, why should we care what Thompson and Ritchie did? Does it have any relevance for us today?

It does because if we understand the thinking behind the system design and the atmosphere in which it flowered, it can help us become productive UNIX system programmers more quickly.

The Early Days

You may already have read about how Ken Thompson came across a DEC PDP-7 machine sitting unused in a hallway at AT&T Bell Laboratories, and how he and Dennis Ritchie and a few of their colleagues used that as the original machine for developing a new operating system that became UNIX.

The important thing to realize, however, is that what they were trying to do was fashion a pleasant computing environment for themselves. It was not, "Let's get together and build an operating system that will attract world-wide attention."

The sequence in which elements of the system fell into place is interesting. The first piece was the file system, followed quickly by its organization into a hierarchy of directories and files. The view of everything, data stores, programs, commands, directories, even devices, as files of one type or another was critical, as was the idea of a file as a one-dimensional array of bytes with no other structure implied. The cleanness and simplicity of this way of looking at files has been a major contributing factor to a computer environment that programmers and other users have found comfortable to work in.

The next element was the idea of processes, with one process being able to create another and communicate with it. This innovative way of looking at running programs as processes led easily to the practice (quintessentially UNIX) of reusing code by calling it from another process. With the addition of commands to manipulate files and an assembler to produce executable programs, the system was essentially able to function on its own.

The next major development was the acquisition of a DEC PDP-11 and the installation of the new system on it. This has been described by Ritchie as a stroke of good luck, in that the PDP-11 was to become a hugely successful machine, its success to some extent adding momentum to the acceptance of the system that began to be known by the name of UNIX.

By 1972 the innovative idea of pipes (connecting links between processes whereby the output of one becomes the input of the next) had been incorporated into the system, the operating system had been recoded in higher level languages (first B, then C), and had been dubbed with the name UNIX (coined by Brian Kernighan). By this point, the "pleasant computing environment" sought by Thompson and Ritchie was a reality; but some other things were going on that had a strong influence on the character of the product then and today.

It is worth pointing out that the UNIX system came out of an atmosphere that was totally different from that in which most commercially successful operating systems are produced. The more typical atmosphere is that described by Tracy Kidder in The Soul of a New Machine. In that case, dozens of talented programmers worked at white heat, in an atmosphere of extremely tight security, against murderous deadlines. By contrast, the UNIX system could be said to have had about a ten year gestation period. From the beginning it attracted the interest of a growing number of brilliant specialists, many of whom found in the UNIX system an environment that allowed them to pursue research and development interests of their own, but who in turn contributed additions to the body of tools available for succeeding ranks of UNIX programmers.

Beginning in 1971, the system began to be used for applications within AT&T Bell Laboratories, and shortly thereafter (1974) was made available at low cost and without support to colleges and universities. These versions, called research versions and identified with Arabic numbers up through 7, occasionally grew on their own and fed back to the main system additional innovative tools. The widely-used screen editor **vi**(1), for example, was added to the UNIX system by William Joy at the University of California, Berkeley. In 1979 acceding to commercial demand, AT&T began offering supported versions (called development versions) of the UNIX system. These are identified with Roman numerals and often have interim release numbers appended. The current development version, for example, is UNIX System V Release 3.0.

Versions of the UNIX system being offered now by AT&T are coming from an environment more closely related, perhaps, to the standard software factory. Features are being added to new releases in response to the expressed needs of the market place. The essential quality of the UNIX system, however, remains as the product of the innovative thinking of its originators and the collegial atmosphere in which they worked. This quality has on occasion been referred to as the UNIX philosophy, but what is meant is the way in which sophisticated programmers have come to work with the UNIX system.

UNIX System Philosophy Simply Stated

For as long as you are writing programs on a UNIX system you should keep this motto hanging on your wall:

```
* * * * * * * * * * * * * * * * * * * * *
*                                       *
*      Build on the work of others      *
*                                       *
* * * * * * * * * * * * * * * * * * * * *
```

Unlike computer environments where each new project is like starting with a blank canvas, on a UNIX system a good percentage of any programming effort is lying there in **bins**, and **lbins**, and **/usr/bins**, not to mention **etc**, waiting to be used.

The features of the UNIX system (pipes, processes, and the file system) contribute to this reusability, as does the history of sharing and contributing that extends back to 1969. You risk missing the essential nature of the UNIX system if you don't put this to work.

UNIX System Tools and Where You Can Read About Them

The term "UNIX system tools" can stand some clarification. In the narrowest sense, it means an existing piece of software used as a component in a new task. In a broader context, the term is often used to refer to elements of the UNIX system that might also be called features, utilities, programs, filters, commands, languages, functions, and so on. It gets confusing because any of the things that might be called by one or more of these names can be, and often are, used in the narrow way as part of the solution to a programming problem.

Tools Covered and Not Covered in this Guide

The *Programmer's Guide* is about tools used in the process of creating programs in a UNIX system environment, so let's take a minute to talk about which tools we mean, which ones are not going to be covered in this book, and where you might find information about those not covered here. Actually, the subject of things not covered in this guide might be even more important to you than the things that are. We couldn't possibly cover everything you ever need to know about UNIX system tools in this one volume.

Tools not covered in this text:

- the **login** procedure
- UNIX system editors and how to use them
- how the file system is organized and how you move around in it
- shell programming

Information about these subjects can be found in the *User's Guide* and a number of commercially available texts.

Tools covered here can be classified as follows:

- utilities for getting programs running
- utilities for organizing software development projects
- specialized languages
- debugging and analysis tools
- compiled language components that are not part of the language syntax, for example, standard libraries, systems calls, and functions

The Shell as a Prototyping Tool

Any time you log in to a UNIX system machine you are using the shell. The shell is the interactive command interpreter that stands between you and the UNIX system kernel, but that's only part of the story. Because of its ability to start processes, direct the flow of control, field interrupts and redirect input and output it is a full-fledged programming language. Programs that use these capabilities are known as shell procedures or shell scripts.

Much innovative use of the shell involves stringing together commands to be run under the control of a shell script. The dozens and dozens of commands that can be used in this way are documented in the *User's Reference Manual.* Time spent with the *User's Reference Manual* can be rewarding. Look through it when you are trying to find a command with just the right option to handle a knotty programming problem. The more familiar you become with the commands described in the manual pages the more you will be able to take full advantage of the UNIX system environment.

It is not our purpose here to instruct you in shell programming. What we want to stress here is the important part that shell procedures can play in developing prototypes of full-scale applications. While understanding all the nuances of shell programming can be a fairly complex task, getting a shell procedure up and running is far less time-consuming than writing, compiling and debugging compiled code.

This ability to get a program into production quickly is what makes the shell a valuable tool for program development. Shell programming allows you to "build on the work of others" to the greatest possible degree, since it allows you to piece together major components simply and efficiently. Many times even large applications can be done using shell procedures. Even if the application is initially developed as a prototype system for testing purposes rather than being put into production, many months of work can be saved.

With a prototype for testing, the range of possible user errors can be determined – something that is not always easy to plan out when an application is being designed. The method of dealing with strange user input can be worked out inexpensively, avoiding large re-coding problems.

A common occurrence in the UNIX system environment is to find that an available UNIX system tool can accomplish with a couple of lines of instructions what might take a page and a half of compiled code. Shell procedures can intermix compiled modules and regular UNIX system commands to let you take advantage of work that has gone before.

Three Programming Environments

We distinguish among three programming environments to emphasize that the information needs and the way in which UNIX system tools are used differ from one environment to another. We do not intend to imply a hierarchy of skill or experience. Highly-skilled programmers with years of experience can be found in the "single-user" category, and relative newcomers can be members of an application development or systems programming team.

Single-User Programmer

Programmers in this environment are writing programs only to ease the performance of their primary job. The resulting programs might well be added to the stock of programs available to the community in which the programmer works. This is similar to the atmosphere in which the UNIX system thrived; someone develops a useful tool and shares it with the rest of the organization. Single-user programmers may not have externally imposed requirements, or co-authors, or project management concerns. The programming task itself drives the coding very directly. One advantage of a timesharing system such as UNIX is that people with programming skills can be set free to work on their own without having to go through formal project approval channels and perhaps wait for months for a programming department to solve their problems.

Single-user programmers need to know how to:

- select an appropriate language
- compile and run programs
- use system libraries
- analyze programs
- debug programs
- keep track of program versions

Most of the information to perform these functions at the single-user level can be found in Chapter 2.

Application Programming

Programmers working in this environment are developing systems for the benefit of other, non-programming users. Most large commercial computer applications still involve a team of applications development programmers. They may be employees of the end-user organization or they may work for a software development firm. Some of the people working in this environment may be more in the project management area than working programmers.

Information needs of people in this environment include all the topics in Chapter 2, plus additional information on:

- software control systems
- file and record locking
- communication between processes
- shared memory
- advanced debugging techniques

These topics are discussed in Chapter 3.

Systems Programmers

These are programmers engaged in writing software tools that are part of, or closely related to the operating system itself. The project may involve writing a new device driver, a data base management system or an enhancement to the UNIX system kernel. In addition to knowing their way around the operating system source code and how to make changes and enhancements to it, they need to be thoroughly familiar with all the topics covered in Chapters 2 and 3.

Summary

In this overview chapter we have described the way that the UNIX system developed and the effect that has on the way programmers now work with it. We have described what is and is not to be found in the other chapters of this guide to help programmers. We have also suggested that in many cases programming problems may be easily solved by taking advantage of the UNIX system interactive command interpreter known as the shell. Finally, we identified three programming environments in the hope that it will help orient the reader to the organization of the text in the remaining chapters.

Chapter 2: Programming Basics

Introduction

The information in this chapter is for anyone just learning to write programs to run in a UNIX system environment. In Chapter 1 we identified one group of UNIX system users as single-user programmers. People in that category, particularly those who are not deeply interested in programming, may find this chapter (plus related reference manuals) tells them as much as they need to know about coding and running programs on a UNIX system computer.

Programmers whose interest does run deeper, who are part of an application development project, or who are producing programs on one UNIX system computer that are being ported to another, should view this chapter as a starter package.

Choosing a Programming Language

How do you decide which programming language to use in a given situation? One answer could be, "I always code in HAIRBOL, because that's the language I know best." Actually, in some circumstances that's a legitimate answer. But assuming more than one programming language is available to you, that different programming languages have their strengths and weaknesses, and assuming that once you've learned to use one programming language it becomes relatively easy to learn to use another, you might approach the problem of language selection by asking yourself questions like the following:

- What is the nature of the task this program is to do?

  Does the task call for the development of a complex algorithm, or is this a simple procedure that has to be done on a lot of records?

- Does the programming task have many separate parts?

  Can the program be subdivided into separately compilable functions, or is it one module?

- How soon does the program have to be available?

  Is it needed right now, or do I have enough time to work out the most efficient process possible?

- What is the scope of its use?

  Am I the only person who will use this program, or is it going to be distributed to the whole world?

- Is there a possibility the program will be ported to other systems?
- What is the life-expectancy of the program?

  Is it going to be used just a few times, or will it still be going strong five years from now?

Supported Languages in a UNIX System Environment

By "supported languages" we mean those offered by AT&T for use on an AT&T 3B2 Computer running UNIX System V Release 3.0. Since these are separately purchasable items, not all of them will necessarily be installed on your machine. On the other hand, you may have languages available on your machine that came from another source and are not mentioned in this discussion. Be that as it may, in this section and the one to follow we give brief descriptions of the nature of a) six full-scale programming languages, and b) a number of special purpose languages.

C Language

C is intimately associated with the UNIX system since it was originally developed for use in recoding the UNIX system kernel. If you need to use a lot of UNIX system function calls for low-level I/O, memory or device management, or inter-process communication, C language is a logical first choice. Most programs, however, don't require such direct interfaces with the operating system so the decision to choose C might better be based on one or more of the following characteristics:

- a variety of data types: character, integer, long integer, float, and double
- low level constructs (most of the UNIX system kernel is written in C)
- derived data types such as arrays, functions, pointers, structures and unions
- multi-dimensional arrays
- scaled pointers, and the ability to do pointer arithmetic
- bit-wise operators
- a variety of flow-of-control statements: if, if-else, switch, while, do-while, and for
- a high degree of portability

C is a language that lends itself readily to structured programming. It is natural in C to think in terms of functions. The next logical step is to view each function as a separately compilable unit. This approach (coding a program in small pieces) eases the job of making changes and/or improvements. If this begins to sound like the UNIX system philosophy of building new programs from existing tools, it's not just coincidence. As you create functions for one program you will surely find that many can be picked up, or quickly revised, for another program.

A difficulty with C is that it takes a fairly concentrated use of the language over a period of several months to reach your full potential as a C programmer. If you are a casual programmer, you might make life easier for yourself if you choose a less demanding language.

FORTRAN

The oldest of the high-level programming languages, FORTRAN is still highly prized for its variety of mathematical functions. If you are writing a program for statistical analysis or other scientific applications, FORTRAN is a good choice. An original design objective was to produce a language with good operating efficiency. This has been achieved at the expense of some flexibility in the area of type definition and data abstraction. There is, for example, only a single form of the iteration statement. FORTRAN also requires using a somewhat rigid format for input of lines of source code. This shortcoming may be overcome by using one of the UNIX system tools designed to make FORTRAN more flexible.

Pascal

Originally designed as a teaching tool for block structured programming, Pascal has gained quite a wide acceptance because of its straightforward style. Pascal is highly structured and allows system level calls (characteristics it shares with C). Since the intent of the developers, however, was to produce a language to teach people about programming it is perhaps best suited to small projects. Among its inconveniences are its lack of facilities for specifying initial values for variables and limited file processing capability.

COBOL

Probably more programmers are familiar with COBOL than with any other single programming language. It is frequently used in business applications because its strengths lie in the management of input/output and in defining record layouts.

It is somewhat cumbersome to use COBOL for complex algorithms, but it works well in cases where many records have to be passed through a simple process; a payroll withholding tax calculation, for example. It is a rather tedious language to work with because each program requires a lengthy amount of text merely to describe record layouts, processing environment and variables used in the code. The COBOL language is wordy so the compilation process is often quite complex. Once written and put into production, COBOL programs have a way of staying in use for years, and what might be thought of by some as wordiness comes to be considered self-documentation. The investment in programmer time often makes them resistant to change.

BASIC

The most commonly heard comment about BASIC is that it is easy to learn. With the spread of personal microcomputers many people have learned BASIC because it is simple to produce runnable programs in very little time. It is difficult, however, to use BASIC for large programming projects. It lacks the provision for structured flow-of-control, requires that every variable used be defined for the entire program and has no way of transferring values between functions and calling programs. Most versions of BASIC run as interpreted code rather than compiled. That makes for slower running programs. Despite its limitations, however, it is useful for getting simple procedures into operation quickly.

Assembly Language

The closest approach to machine language, assembly language is specific to the particular computer on which your program is to run. High-level languages are translated into the assembly language for a specific processor as one step of the compilation. The most common need to work in assembly language arises when you want to do some task that is not within the scope of a high-level language. Since assembly language is machine-specific, programs written in it are not portable.

Special Purpose Languages

In addition to the above formal programming languages, the UNIX system environment frequently offers one or more of the special purpose languages listed below.

<u>NOTE:</u> Since UNIX system utilities and commands are packaged in functional groupings, it is possible that not all the facilities mentioned will be available on all systems.

awk

awk (its name is an acronym constructed from the initials of its developers) scans an input file for lines that match pattern(s) described in a specification file. On finding a line that matches a pattern, **awk** performs actions also described in the specification. It is not uncommon that an **awk** program can be written in a couple of lines to do functions that would take a couple of pages to describe in a programming language like FORTRAN or C. For example, consider a case where you have a set of records that consist of a key field and a second field that represents a quantity. You have sorted the records by the key field, and you now want to add the quantities for records with duplicate keys and output a file in which no keys are duplicated. The pseudo-code for such a program might look like this:

```
Read the first record into a hold area;
Read additional records until EOF;
 {
 If the key matches the key of the record in the hold area,
  add the quantity to the quantity field of the held record;
 If the key does not match the key of the held record,
  write the held record,
  move the new record to the hold area;
 }
At EOF, write out the last record from the hold area.
```

An **awk** program to accomplish this task would look like this:

```
    { qty[$1] += $2 }
    END{ for (key in qty) print key, qty[key] }
```

This illustrates only one characteristic of **awk**; its ability to work with associative arrays. With **awk**, the input file does not have to be sorted, which is a requirement of the pseudo-program.

lex

lex is a lexical analyzer that can be added to C or FORTRAN programs. A lexical analyzer is interested in the vocabulary of a language rather than its grammar, which is a system of rules defining the structure of a language. **lex** can produce C language subroutines that recognize regular expressions specified by the user, take some action when a regular expression is recognized and pass the output stream on to the next program.

yacc

yacc (Yet Another Compiler Compiler) is a tool for describing an input language to a computer program. **yacc** produces a C language subroutine that parses an input stream according to rules laid down in a specification file. The **yacc** specification file establishes a set of grammar rules together with actions to be taken when tokens in the input match the rules. **lex** may be used with **yacc** to control the input process and pass tokens to the parser that applies the grammar rules.

M4

M4 is a macro processor that can be used as a preprocessor for assembly language, and C programs. It is described in Section (1) of the *Programmer's Reference Manual.*

bc and dc

bc enables you to use a computer terminal as you would a programmable calculator. You can edit a file of mathematical computations and call **bc** to execute them. The **bc** program uses **dc**. You can use **dc** directly, if you want, but it takes a little getting used to since it works with reverse Polish notation. That means you enter numbers into a stack followed by the operator. **bc** and **dc** are described in Section (1) of the *User's Reference*

Manual.

curses

Actually a library of C functions, **curses** is included in this list because the set of functions just about amounts to a sub-language for dealing with terminal screens. If you are writing programs that include interactive user screens, you will want to become familiar with this group of functions.

In addition to all the foregoing, don't overlook the possibility of using shell procedures.

After Your Code Is Written

The last two steps in most compilation systems in the UNIX system environment are the assembler and the link editor. The compilation system produces assembly language code. The assembler translates that code into the machine language of the computer the program is to run on. The link editor resolves all undefined references and makes the object module executable. With most languages on the UNIX system the assembler and link editor produce files in what is known as the Common Object File Format (COFF). A common format makes it easier for utilities that depend on information in the object file to work on different machines running different versions of the UNIX system.

In the Common Object File Format an object file contains:

- a file header
- optional secondary header
- a table of section headers
- data corresponding to the section header(s)
- relocation information
- line numbers
- a symbol table
- a string table

An object file is made up of sections. Usually, there are at least two: **.text**, and **.data**. Some object files contain a section called **.bss**. (**.bss** is an assembly language pseudo-op that originally stood for "block started by symbol.") **.bss**, when present, holds uninitialized data. Options of the compilers cause different items of information to be included in the Common Object File Format. For example, compiling a program with the **-g** option adds line numbers and other symbolic information that is needed for the **sdb** (Symbolic Debugger) command to be fully effective. You can spend many years programming without having to worry too much about the contents and organization of the Common Object File Format, so we are not going into any further depth of detail at this point. Detailed information is available in Chapter 11 of this guide.

Compiling and Link Editing

The command used for compiling depends on the language used;

- for C programs, **cc** both compiles and link edits
- for FORTRAN programs, **f77** both compiles and link edits

Compiling C Programs

To use the C compilation system you must have your source code in a file with a filename that ends in the characters **.c**, as in **mycode.c**. The command to invoke the compiler is:

```
cc mycode.c
```

If the compilation is successful the process proceeds through the link edit stage and the result will be an executable file by the name of **a.out**.

Several options to the **cc** command are available to control its operation. The most used options are:

-c	causes the compilation system to suppress the link edit phase. This produces an object file (**mycode.o**) that can be link edited at a later time with a **cc** command without the **-c** option.
-g	causes the compilation system to generate special information about variables and language statements used by the symbolic debugger **sdb**. If you are going through the stage of debugging your program, use this option.
-O	causes the inclusion of an additional optimization phase. This option is logically incompatible with the **-g** option. You would normally use **-O** after the program has been debugged, to reduce the size of the object file and increase execution speed.

-p	causes the compilation system to produce code that works in conjunction with the **prof**(1) command to produce a runtime profile of where the program is spending its time. Useful in identifying which routines are candidates for improved code.
-o *outfile*	tells **cc** to tell the link editor to use the specified name for the executable file, rather than the default **a.out**.

Other options can be used with **cc**. Check the *Programmer's Reference Manual.*

If you enter the **cc** command using a file name that ends in **.s**, the compilation system treats it as assembly language source code and bypasses all the steps ahead of the assembly step.

Compiling FORTRAN Programs

The **f77** command invokes the FORTRAN compilation system. The operation of the command is similar to that of the **cc** command, except the source code file(s) must have a **.f** suffix. The **f77** command compiles your source code and calls in the link editor to produce an executable file whose name is **a.out**.

The following command line options have the same meaning as they do for the **cc** command:

-c, -p, -O, -g, and **-o** *outfile*

Loading and Running BASIC Programs

BASIC programs can be invoked in two ways:

- With the command

  basic bscpgm.b

  where **bscpgm.b** is the name of the file that holds your BASIC statements. This tells the UNIX system to load and run the program. If the program includes a **run** statement naming another program, you will chain from one to the other. Variables specified in the first can be preserved for the second with the **common** statement.

- By setting up a shell script.

Compiler Diagnostic Messages

The C compiler generates error messages for statements that don't compile. The messages are generally quite understandable, but in common with most language compilers they sometimes point several statements beyond where the actual error occurred. For example, if you inadvertently put an extra ; at the end of an if statement, a subsequent else will be flagged as a syntax error. In the case where a block of several statements follows the if, the line number of the syntax error caused by the else will start you looking for the error well past where it is. Unbalanced curly braces, { }, are another common producer of syntax errors.

Link Editing

The **ld** command invokes the link editor directly. The typical user, however, seldom invokes **ld** directly. A more common practice is to use a language compilation control command (such as **cc**) that invokes **ld**. The link editor combines several object files into one, performs relocation, resolves external symbols, incorporates startup routines, and supports symbol table information used by **sdb**. You may, of course, start with a single object file rather than several. The resulting executable module is left in a file named **a.out**.

Any file named on the **ld** command line that is not an object file (typically, a name ending in **o**) is assumed to be an archive library or a file of link editor directives. The **ld** command has some 16 options. We are going to describe four of them. These options should be fed to the link editor by specifying them on the **cc** command line if you are doing both jobs with the single command, which is the usual case.

-o *outfile* provides a name to be used to replace **a.out** as the name of the output file. Obviously, the name **a.out** is of only temporary usefulness. If you know the name you want use to invoke your program, you can provide it here. Of course, it may be equally convenient to do this:

```
mv a.out progname
```

when you want to give your program a less temporary name.

-l*x*	directs the link editor to search a library **lib***x***.a**, where *x* is up to nine characters. For C programs, **libc.a** is automatically searched if the **cc** command is used. The **-l***x* option is used to bring in libraries not normally in the search path such as **libm.a**, the math library. The **-l***x* option can occur more than once on a command line, with different values for the *x*. A library is searched when its name is encountered, so the placement of the option on the command line is important. The safest place to put it is at the end of the command line. The **-l***x* option is related to the **-L** option.
-L *dir*	changes the **lib***x***.a** search sequence to search in the specified directory before looking in the default library directories, usually **/lib** or **/usr/lib**. This is useful if you have different versions of a library and you want to point the link editor to the correct one. It works on the assumption that once a library has been found no further searching for that library is necessary. Because **-L** diverts the search for the libraries specified by **-l***x* options, it must precede such options on the command line.
-u *symname*	enters *symname* as an undefined symbol in the symbol table. This is useful if you are loading entirely from an archive library, because initially the symbol table is empty and needs an unresolved reference to force the loading of the first routine.

When the link editor is called through **cc**, a startup routine (typically **/lib/crt0.o** for C programs) is linked with your program. This routine calls **exit**(2) after execution of the main program.

The link editor accepts a file containing link editor directives. The details of the link editor command language can be found in Chapter 12.

The Interface Between a Programming Language and the UNIX System

When a program is run in a computer it depends on the operating system for a variety of services. Some of the services such as bringing the program into main memory and starting the execution are completely transparent to the program. They are, in effect, arranged for in advance by the link editor when it marks an object module as executable. As a programmer you seldom need to be concerned about such matters.

Other services, however, such as input/output, file management, storage allocation do require work on the part of the programmer. These connections between a program and the UNIX operating system are what is meant by the term UNIX system/language interface. The topics included in this section are:

- How arguments are passed to a program
- System calls and subroutines
- Header files and libraries
- Input/Output
- Processes
- Error Handling, Signals, and Interrupts

Why C Is Used to Illustrate the Interface

Throughout this section C programs are used to illustrate the interface between the UNIX system and programming languages because C programs make more use of the interface mechanisms than other high-level languages. What is really being covered in this section then is the UNIX system/C Language interface. The way that other languages deal with these topics is described in the user's guides for those languages.

How Arguments Are Passed to a Program

Information or control data can be passed to a C program as arguments on the command line. When the program is run as a command, arguments on the command line are made available to the function **main** in two parameters, an argument count and an array of pointers to character strings. (Every C program is required to have an entry module by the name of **main**.) Since the argument count is always given, the program does not have to know in advance how many arguments to expect. The character strings pointed at by elements of the array of pointers contain the argument information.

The arguments are presented to the program traditionally as **argc** and **argv**, although any names you choose will work. **argc** is an integer that gives the count of the number of arguments. Since the command itself is considered to be the first argument, **argv[0]**, the count is always at least one. **argv** is an array of pointers to character strings (arrays of characters terminated by the null character \0).

If you plan to pass runtime parameters to your program, you need to include code to deal with the information. Two possible uses of runtime parameters are:

- as control data. Use the information to set internal flags that control the operation of the program.
- to provide a variable filename to the program.

Figures 2-1 and 2-2 show program fragments that illustrate these uses.

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
                  void exit();
                  int oflag = FALSE;
                  int pflag = FALSE;/* Function Flags */
                  int rflag = FALSE;
                  int ch;

                  while ((ch = getopt(argc,argv, "opr"))
                  != EOF)
                  {
                    /* For options present, set flag to
                  TRUE */
                    /* If no options present, print error
                  message */
                  switch (ch)
                  {
                  case 'o':
                  oflag = 1;
                  break;
                  case 'p':
                  pflag = 1;
                  break;
                  case 'r':
                  rflag = 1;
                  break;
                  default:
                  (void)fprintf(stderr,
                  "Usage: %s [-opr]\n", argv[0]);
                  exit(2);
                  }
                  }
                  .
                  .
                  .
}
```

Figure 2-1: Using Command Line Arguments to Set Flags

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
                FILE *fopen(), *fin;
                void perror(), exit();

                if (argc > 1)
                {
                if ((fin = fopen(argv[1], "r")) == NULL)
                {
                  /* First string (%s) is program name
                (argv[0]) */
                  /* Second string (%s) is name of file
                that could */
                  /* not be opened (argv[1]) */
                (void)fprintf(stderr,
                  "%s: cannot open %s: ",
                  argv[0], argv[1]);
                perror("");
                exit(2);
                }
                }
                .
                .
                .
}
```

Figure 2-2: Using **argv**[*n*] Pointers to Pass a Filename

The shell, which makes arguments available to your program, considers an argument to be any non-blank characters separated by blanks or tabs. Characters enclosed in double quotes ("abc def") are passed to the program as one argument even if blanks or tabs are among the characters. It goes without saying that you are responsible for error checking and otherwise making sure the argument received is what your program expects it to be.

A third argument is also present, in addition to **argc** and **argv**. The third argument, known as **envp**, is an array of pointers to environment variables. You can find more information on **envp** in the *Programmer's Reference Manual* under **exec**(2) and **environ**(5).

System Calls and Subroutines

System calls are requests from a program for an action to be performed by the UNIX system kernel. Subroutines are precoded modules used to supplement the functionality of a programming language.

Both system calls and subroutines look like functions such as those you might code for the individual parts of your program. There are, however, differences between them:

- At link edit time, the code for subroutines is copied into the object file for your program; the code invoked by a system call remains in the kernel.
- At execution time, subroutine code is executed as if it was code you had written yourself; a system function call is executed by switching from your process area to the kernel.

This means that while subroutines make your executable object file larger, runtime overhead for context switching may be less and execution may be faster.

Categories of System Calls and Subroutines

System calls divide fairly neatly into the following categories:

- file access
- file and directory manipulation
- process control
- environment control and status information

You can generally tell the category of a subroutine by the section of the *Programmer's Reference Manual* in which you find its manual page. However, the first part of Section 3 (3C and 3S) covers such a variety of subroutines it might be helpful to classify them further.

- The subroutines of sub-class 3S constitute the UNIX system/C Language standard I/O, an efficient I/O buffering scheme for C.
- The subroutines of sub-class 3C do a variety of tasks. They have in common the fact that their object code is stored in **libc.a**. They can be divided into the following categories:

  string manipulation

  character conversion

  character classification

  environment management

  memory management

Figure 2-3 lists the functions that compose the standard I/O subroutines. Frequently, one manual page describes several related functions. In Figure 2-3 the left hand column contains the name that appears at the top of the manual page; the other names in the same row are related functions described on the same manual page.

Function Name(s)	Purpose
fclose, fflush	close or flush a stream
ferror, feof, clearerr, fileno	stream status inquires
fopen, freopen, fdopen	open a stream
fread, fwrite	binary input/output
fseek, rewind, ftell	reposition on file pointer in stream
getc, getchar, fgetc, getw	get char. or word from stream
gets, fgets	get string from a stream
popen, pclose	begin/end pipe to/from process
printf, fprintf, sprintf	print formatted output

For all functions: #include <stdio.h>

The function name shown in **bold** gives the location in the *Programmer's Reference Manual*, Section 3.

Figure 2-3: C Language Standard I/O Subroutines (sheet 1 of 2)

Function Name(s)	Purpose
putc, putchar, fputc, putw	put char. or word on stream
puts, fputs	put string on a stream
scanf, fscanf, sscanf	convert formatted input
setbuf, setvbuf	assign buffering to a stream
system	issue command through shell
tmpfile	create a temporary file
tmpnam, tempnam	create name for temp. file
ungetc	push char. back into input stream
vprintf, vfprintf, vsprintf	print output of vargas arg. list

For all functions: #include <stdio.h>

The function name shown in **bold** gives the location in the *Programmer's Reference Manual*, Section 3.

Figure 2-3: C Language Standard I/O Subroutines (sheet 2 of 2)

Figure 2-4 lists string handling functions that are grouped under the heading **string**(3C) in the *Programmer's Reference Manual*.

String Operations

strcat(s1, s2)	append a copy of s2 to the end of s1.
strncat(s1, s2, n)	append n characters from s2 to the end of s1.
strcmp(s1, s2)	compare two strings. Returns an integer less than, greater than or equal to 0 to show that s1 is lexicographically less than, greater than or equal to s2.
strncmp(s1, s2, n)	compare n characters from the two strings. Results are otherwise identical to strcmp.
strcpy(s1, s2)	copy s2 to s1, stopping after the null character (\0) has been copied.
strncpy(s1, s2, n)	copy n characters from s2 to s1. s2 will be truncated if it is longer than n, or padded with null characters if it is shorter than n.
strdup(s)	returns a pointer to a new string that is a duplicate of the string pointed to by s.
strchr(s, c)	returns a pointer to the first occurrence of character c in string s, or a NULL pointer if c is not in s.
strrchr(s, c)	returns a pointer to the last occurrence of character c in string s, or a NULL pointer if c is not in s.

For all functions: #include <string.h>
string.h provides extern definitions of the string functions.

Figure 2-4: String Operations (sheet 1 of 2)

String Operations

strlen(s)	returns the number of characters in s up to the first null character.
strpbrk(s1, s2)	returns a pointer to the first occurrence in s1 of any character from s2, or a NULL pointer if no character from s2 occurs in s1.
strspn(s1, s2)	returns the length of the initial segment of s1, which consists entirely of characters from s2.
strcspn(s1, s2)	returns the length of the initial segment of s1, which consists entirely of characters not from s2.
strtok(s1, s2)	look for occurrences of s2 within s1.

For all functions: #include <string.h>
string.h provides extern definitions of the string functions.

Figure 2-4: String Operations (sheet 2 of 2)

Figure 2-5 lists macros that classify ASCII character-coded integer values. These macros are described under the heading **ctype**(3C) in Section 3 of the *Programmer's Reference Manual.*

Classify Characters

isalpha(c)	is *c* a letter
isupper(c)	is *c* an upper-case letter
islower(c)	is *c* a lower-case letter
isdigit(c)	is *c* a digit [0-9]
isxdigit(c)	is *c* a hexadecimal digit [0-9], [A-F] or [a-f]
isalnum(c)	is *c* an alphanumeric (letter or digit)
isspace(c)	is *c* a space, tab, carriage return, new-line, vertical tab or form-feed
ispunct(c)	is *c* a punctuation character (neither control nor alphanumeric)
isprint(c)	is *c* a printing character, code 040 (space) through 0176 (tilde)
isgraph(c)	same as isprint except false for 040 (space)
iscntrl(c)	is *c* a control character (less than 040) or a delete character (0177)
isascii(c)	is *c* an ASCII character (code less than 0200)

For all functions: #include <ctype.h>
Nonzero return = = true; zero return = = false

Figure 2-5: Classifying ASCII Character-Coded Integer Values

Figure 2-6 lists functions and macros that are used to convert characters, integers, or strings from one representation to another.

Function Name(s)			Purpose
a64l	**l64a**		convert between long integer and base-64 ASCII string
ecvt	**fcvt**	**gcvt**	convert floating-point number to string
l3tol	**ltol3**		convert between 3-byte integer and long integer
strtod	**atof**		convert string to double-precision number
strtol	**atol**	**atoi**	convert string to integer

conv(3C):	**Translate Characters**
toupper	lower-case to upper-case
_toupper	macro version of toupper
tolower	upper-case to lower-case
_tolower	macro version of tolower
toascii	turn off all bits that are not part of a standard ASCII character; intended for compatibility with other systems

For all **conv**(3C) macros: #include <ctype.h>

Figure 2-6: Conversion Functions and Macros

Where the Manual Pages Can Be Found

System calls are listed alphabetically in Section 2 of the *Programmer's Reference Manual*. Subroutines are listed in Section 3. We have described above what is in the first subsection of Section 3. The remaining subsections of Section 3 are:

- 3M – functions that make up the Math Library, **libm**
- 3X – various specialized functions
- 3F – the FORTRAN intrinsic function library, **libF77**
- 3N – Networking Support Utilities

How System Calls and Subroutines Are Used in C Programs

Information about the proper way to use system calls and subroutines is given on the manual page, but you have to know what you are looking for before it begins to make sense. To illustrate, a sample manual page (for **gets**(3S)) is shown in Figure 2-7.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char   *gets (s)
char   *s;

char   *fgets (s, n, stream)
char   *s;
int   n;
FILE    *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. *Fgets* reads characters from the *stream* into the array pointed to by *s*, until *n*-1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered.

SEE ALSO

ferror(3S),
fopen(3S),
fread(3S),
getc(3S),
scanf(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs a NULL pointer is returned. Otherwise *s* is returned.

Figure 2-7: Manual Page for **gets**(3S)

As you can see from the illustration, two related functions are described on this page: **gets** and **fgets**. Each function gets a string from a stream in a slightly different way. The DESCRIPTION section tells how each operates.

It is the SYNOPSIS section, however, that contains the critical information about how the function (or macro) is used in your program. Notice in Figure 2-7 that the first line in the SYNOPSIS is

#include <stdio.h>

This means that to use **gets** or **fgets** you must bring the standard I/O header file into your program (generally right at the top of the file). There is something in **stdio.h** that is needed when you use the described functions. Figure 2-9 shows a version of **stdio.h.** Check it to see if you can understand what **gets** or **fgets** uses.

The next thing shown in the SYNOPSIS section of a manual page that documents system calls or subroutines is the formal declaration of the function. The formal declaration tells you:

- **the type of object returned by the function**

  In our example, both **gets** and **fgets** return a character pointer.

- **the object or objects the function expects to receive when called**

  These are the things enclosed in the parentheses of the function. **gets** expects a character pointer. (The DESCRIPTION section sheds light on what the tokens of the formal declaration stand for.)

- **how the function is going to treat those objects**

  The declaration

  ```
  char *s;
  ```

  in **gets** means that the token s enclosed in the parentheses will be considered to be a pointer to a character string. Bear in mind that in the C language, when passed as an argument, the name of an array is converted to a pointer to the beginning of the array.

We have chosen a simple example here in **gets**. If you want to test yourself on something a little more complex, try working out the meaning of the elements of the **fgets** declaration.

While we're on the subject of **fgets**, there is another piece of C esoterica that we'll explain. Notice that the third parameter in the **fgets** declaration is referred to as **stream**. A **stream**, in this context, is a file with its associated buffering. It is declared to be a pointer to a defined type FILE. Where is FILE defined? Right! In **stdio.h**.

To finish off this discussion of the way you use functions described in the *Programmer's Reference Manual* in your own code, in Figure 2-8 we show a program fragment in which **gets** is used.

```
#include <stdio.h>

main()
{
     char sarray[80];

     for(;;)
     {
          if (gets(sarray) != NULL)
          .
          .    /* Do something with the string */
          .
     }
}
```

Figure 2-8: How **gets** Is Used in a Program

You might ask, "Where is **gets** reading from?" The answer is, "From the standard input." That generally means from something being keyed in from the terminal where the command was entered to get the program running, or output from another command that was piped to **gets**. How do we know that? The DESCRIPTION section of the **gets** manual page says, "**gets** reads characters from the standard input...." Where is the standard input defined? In **stdio.h**.

```
#ifndef _NFILE
#define _NFILE          20

#define BUFSIZ          1024
#define _SBFSIZ 8

typedef struct {
            int         _cnt;
            unsigned char           *_ptr;
            unsigned char           *_base;
            char        _flag;
            char        _file;
} FILE;

#define _IOFBF          0000  /* _IOLBF means that a */
#define _IOREAD         0001  /* file's output will be */
#define _IOWRT          0002  /* buffered line by line. */
#define _IONBF          0004  /* In addition to being */
#define _IOMYBUF        0010  /* flags, _IONBF, _IOLBF, & */
#define _IOEOF          0020  /* IOFBF are possible */
#define _IOERR          0040  /* values for "type" in */
#define _IOLBF          0100  /* setvbuf.           */
#define _IORW           0200

#ifndef NULL
#define NULL            0
#endif
#ifndef EOF
#define EOF             (-1)
#endif
```

Figure 2-9: A Version of **stdio.h** (sheet 1 of 2)

```
#define stdin                         (&_iob[0])
#define stdout                        (&_iob[1])
#define stderr                        (&_iob[2])

#define _bufend(p)       _bufendtab[(p)->_file]
#define _bufsiz(p)       (_bufend(p) - (p)->_base)

#ifndef lint
#define getc(p)          (--(p)->_cnt < 0 ? _filbuf(p) :
                         (int) *(p)->_ptr++)
#define putc(x, p)       (--(p)->_cnt < 0 ?
                         _flsbuf((unsigned char) (x), (p)) :
                         (int) (*(p)->_ptr++ = (unsigned
                         char) (x)))
#define getchar()        getc(stdin)
#define putchar(x)       putc((x), stdout)
#define clearerr(p)      ((void) ((p)->_flag &=
                          (_IOERR ¦ _IOEOF)))
#define feof(p)          ((p)->_flag & _IOEOF)
#define ferror(p)        ((p)->_flag & _IOERR)
#define fileno(p)        (p)->_file
#endif

extern FILE _iob[_NFILE];
extern FILE *fopen(), *fdopen(), *freopen(),
                         *popen(), *tmpfile();
extern long ftell();
extern void rewind(), setbuf();
extern char    *ctermid(), *cuserid(), *fgets(), *gets(),
                         *tempnam(), *tmpnam();
extern unsigned char *_bufendtab[];

#define L_ctermid        9
#define L_cuserid        9
#define P_tmpdir         "/usr/tmp/"
#define L_tmpnam         (sizeof(P_tmpdir) + 15)
#endif
```

Figure 2-9: A Version of **stdio.h** (sheet 2 of 2)

Header Files and Libraries

In the earlier parts of this chapter there have been frequent references to **stdio.h**, and a version of the file itself is shown in Figure 2-9. **stdio.h** is the most commonly used header file in the UNIX system/C environment, but there are many others.

Header files carry definitions and declarations that are used by more than one function. Header filenames traditionally have the suffix **.h**, and are brought into a program at compile time by the C-preprocessor. The preprocessor does this because it interprets the **#include** statement in your program as a directive; as indeed it is. All keywords preceded by a pound sign (#) at the beginning of the line, are treated as preprocessor directives. The two most commonly used directives are **#include** and **#define**. We have already seen that the **#include** directive is used to call in (and process) the contents of the named file. The **#define** directive is used to replace a name with a token-string. For example,

```
#define _NFILE    20
```

sets to 20 the number of files a program can have open at one time. See **cpp**(1) for the complete list.

In the pages of the *Programmer's Reference Manual* there are about 45 different **.h** files named. The format of the **#include** statement for all these shows the file name enclosed in angle brackets (< >), as in

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

The angle brackets tell the C preprocessor to look in the standard places for the file. In most systems the standard place is in the **/usr/include** directory. If you have some definitions or external declarations that you want to make available in several files, you can create a **.h** file with any editor, store it in a convenient directory and make it the subject of a **#include** statement such as the following:

```
#include "../defs/rec.h"
```

It is necessary, in this case, to provide the relative pathname of the file and enclose it in quotation marks (""). Fully-qualified pathnames (those that begin with /) can create portability and organizational problems. An alternative to long or fully-qualified pathnames is to use the **-I***dir* preprocessor option when you compile the program. This option directs the preprocessor to search for **#include** files whose names are enclosed in "", first in the directory of the file being compiled, then in the directories named in the **-I** option(s), and finally in directories on the standard list. In addition, all **#include** files whose names are enclosed in angle brackets (< >) are first searched for in the list of directories named in the **-I** option and finally in the directories on the standard list.

Object File Libraries

It is common practice in UNIX system computers to keep modules of compiled code (object files) in archives; by convention, designated by a **.a** suffix. System calls from Section 2, and the subroutines in Section 3, subsections 3C and 3S, of the *Programmer's Reference Manual* that are functions (as distinct from macros) are kept in an archive file by the name of **libc.a**. In most systems, **libc.a** is found in the directory **/lib**. Many systems also have a directory **/usr/lib**. Where both **/lib** and **/usr/lib** occur, **/usr/lib** is apt to be used to hold archives that are related to specific applications.

During the link edit phase of the compilation and link edit process, copies of some of the object modules in an archive file are loaded with your executable code. By default the **cc** command that invokes the C compilation system causes the link editor to search **libc.a**. If you need to point the link editor to other libraries that are not searched by default, you do it by naming them explicitly on the command line with the **-l** option. The format of the **-l** option is **-l***x* where *x* is the library name, and can be up to nine characters. For example, if your program includes functions from the **curses** screen control package, the option

```
-lcurses
```

will cause the link editor to search for **/lib/libcurses.a** or **/usr/lib/libcurses.a** and use the first one it finds to resolve references in your program.

In cases where you want to direct the order in which archive libraries are searched, you may use the **-L** *dir* option. Assuming the **-L** option appears on the command line ahead of the **-l** option, it directs the link editor to search the named directory for **lib***x***.a** before looking in **/lib** and **/usr/lib**. This is particularly useful if you are testing out a new version of a function that already exists in an archive in a standard directory. Its success is due to the fact that once having resolved a reference the link editor stops looking. That's why the **-L** option, if used, should appear on the command line ahead of any **-l** specification.

Input/Output

We talked some about I/O earlier in this chapter in connection with system calls and subroutines. A whole set of subroutines constitutes the C language standard I/O package, and there are several system calls that deal with the same area. In this section we want to get into the subject in a little more detail and describe for you how to deal with input and output concerns in your C programs. First off, let's briefly define what the subject of I/O encompasses. It has to do with

- creating and sometimes removing files
- opening and closing files used by your program
- transferring information from a file to your program (reading)
- transferring information from your program to a file (writing)

In this section we will describe some of the subroutines you might choose for transferring information, but the heaviest emphasis will be on dealing with files.

Three Files You Always Have

Programs are permitted to have several files open simultaneously. The number may vary from system to system; the most common maximum is 20. _NFILE in **stdio.h** specifies the number of standard I/O FILEs a program is permitted to have open.

Any program automatically starts off with three files. If you will look again at Figure 2-9, about midway through you will see that **stdio.h** contains three **#define** directives that equate **stdin**, **stdout**, and **stderr** to the address of _iob[0], _iob[1], and _iob[2], respectively. The array _iob holds information dealing with the way standard I/O handles streams. It is a representation of the open file table in the control block for your program. The position in the array is a digit that is also known as the file descriptor. The default in UNIX systems is to associate all three of these files with your terminal.

The real significance is that functions and macros that deal with **stdin** or **stdout** can be used in your program with no further need to open or close files. For example, **gets**, cited above, reads a string from **stdin**; **puts** writes a null-terminated string to **stdout**. There are others that do the same (in slightly different ways: character at a time, formatted, etc.). You can specify that output be directed to **stderr** by using a function such as **fprintf**. **fprintf** works the same as **printf** except that it delivers its formatted output to a named stream, such as **stderr**. You can use the shell's redirection feature on the command line to read from or write into a named file. If you want to separate error messages from ordinary output being sent to **stdout** and thence possibly piped by the shell to a succeeding program, you can do it by using one function to handle the ordinary output and a variation of the same function that names the stream, to handle error messages.

Named Files

Any files other than **stdin**, **stdout**, and **stderr** that are to be used by your program must be explicitly connected by you before the file can be read from or written to. This can be done using the standard library routine **fopen**. **fopen** takes a pathname (which is the name by which the file is known to the UNIX file system), asks the system to keep track of the connection, and returns a pointer that you then use in functions that do the reads and writes.

A structure is defined in **stdio.h** with a type of FILE. In your program you need to have a declaration such as

```
        FILE *fin;
```

The declaration says that **fin** is a pointer to a FILE. You can then assign the name of a particular file to the pointer with a statement in your program like this:

```
        fin = fopen("filename", "r");
```

where **filename** is the pathname to open. The "f3r" means that the file is to be opened for reading. This argument is known as the **mode**. As you might suspect, there are modes for reading, writing, and both reading and writing. Actually, the file open function is often included in an if statement such as:

```
if ((fin = fopen("filename", "r")) == NULL)
  (void)fprintf(stderr,"%s: Unable to open input file
  %s\n",argv[0],"filename");
```

that takes advantage of the fact that **fopen** returns a NULL pointer if it can't open the file.

Once the file has been successfully opened, the pointer **fin** is used in functions (or macros) to refer to the file. For example:

```
        int c;
        c = getc(fin);
```

brings in a character at a time from the file into an integer variable called **c**. The variable **c** is declared as an integer even though we are reading characters because the function **getc()** returns an integer. Getting a character is often incorporated into some flow-of-control mechanism such as:

```
        while ((c = getc(fin)) != EOF)
               .
               .
               .
```

that reads through the file until EOF is returned. EOF, NULL, and the macro **getc** are all defined in **stdio.h**. **getc** and others that make up the standard I/O package keep advancing a pointer through the buffer associated with the file; the UNIX system and the standard I/O subroutines are responsible for seeing that the buffer is refilled (or written to the output file if you are producing output) when the pointer reaches the end of the buffer. All these mechanics are mercifully invisible to the program and the programmer.

The function **fclose** is used to break the connection between the pointer in your program and the pathname. The pointer may then be associated with another file by another call to **fopen**. This re-use of a file descriptor for a different stream may be necessary if your program has many files to open. For output files it is good to issue an **fclose** call because the call makes sure that all output has been sent from the output buffer before disconnecting the file. The system call **exit** closes all open files for you. It also gets you completely out of your process, however, so it is safe to use only when you are sure you are completely finished.

Low-level I/O and Why You Shouldn't Use It

The term low-level I/O is used to refer to the process of using system calls from Section 2 of the *Programmer's Reference Manual* rather than the functions and subroutines of the standard I/O package. We are going to postpone until Chapter 3 any discussion of when this might be advantageous. If you find as you go through the information in this chapter that it is a good fit with the objectives you have as a programmer, it is a safe assumption that you can work with C language programs in the UNIX system for a good many years without ever having a real need to use system calls to handle your I/O and file accessing problems. The reason low-level I/O is perilous is because it is more system-dependent. Your programs are less portable and probably no more efficient.

System Calls for Environment or Status Information

Under some circumstances you might want to be able to monitor or control the environment in your computer. There are system calls that can be used for this purpose. Some of them are shown in Figure 2-10.

Function Name(s)	Purpose
chdir	change working directory
chmod	change access permission of file
chown	change owner & group of file
getpid, getpgrp, getppid	get process IDs
getuid, geteuid, getgid	get user IDs
ioctl	control device
link, unlink	add or remove directory entry
mount, umount	mount/unmount file system
nice	change priority of a process
stat, fstat	get file status
time	get time
ulimit	get & set user limits
uname	get name current UNIX system

Figure 2-10: Environment and Status System Calls

As you can see, many of the functions shown in Figure 2-10 have equivalent UNIX system shell commands. Shell commands can easily be incorporated into shell scripts to accomplish the monitoring and control tasks you may need to do. The functions are available, however, and may be used in C programs as part of the UNIX system/C Language interface. They are documented in Section 2 of the *Programmers' Reference Manual*.

Processes

Whenever you execute a command in the UNIX system you are initiating a process that is numbered and tracked by the operating system. A flexible feature of the UNIX system is that processes can be generated by other processes. This happens more than you might ever be aware of. For example, when you log in to your system you are running a process, very probably the shell. If you then use an editor such as **vi**, take the option of invoking the shell from **vi**, and execute the **ps** command, you will see a display something like that in Figure 2-11 (which shows the results of a **ps -f** command):

```
UID   PID    PPID   C   STIME     TTY    TIME   COMMAND
abc   24210  1      0   06:13:14  tty29  0:05   -sh
abc   24631  24210  0   06:59:07  tty29  0:13   vi c2.uli
abc   28441  28358  80  09:17:22  tty29  0:01   ps -f
abc   28358  24631  2   09:15:14  tty29  0:01   sh -i
```

Figure 2-11: Process Status

As you can see, user abc (who went through the steps described above) now has four processes active. It is an interesting exercise to trace the chain that is shown in the Process ID (PID) and Parent Process ID (PPID) columns. The shell that was started when user abc logged on is Process 24210; its parent is the initialization process (Process ID 1). Process 24210 is the parent of Process 24631, and so on.

The four processes in the example above are all UNIX system shell level commands, but you can spawn new processes from your own program. (Actually, when you issue the command from your terminal to execute a program you are asking the shell to start another process, the process being your executable object module with all the functions and subroutines that were made a part of it by the link editor.)

You might think, "Well, it's one thing to switch from one program to another when I'm at my terminal working interactively with the computer; but why would a program want to run other programs, and if one does, why wouldn't I just put everything together into one big executable module?"

Overlooking the case where your program is itself an interactive application with diverse choices for the user, your program may need to run one or more other programs based on conditions it encounters in its own processing. (If it's the end of the month, go do a trial balance, for example.) The usual reasons why it might not be practical to create one monster executable are:

- The load module may get too big to fit in the maximum process size for your system.
- You may not have control over the object code of all the other modules you want to include.

Suffice it to say, there are legitimate reasons why this creation of new processes might need to be done. There are three ways to do it:

- **system**(3S) – request the shell to execute a command
- **exec**(2) – stop this process and start another
- **fork**(2) – start an additional copy of this process

system(3S)

The formal declaration of the **system** function looks like this:

```
#include <stdio.h>

int system(string)
char *string;
```

The function asks the shell to treat the string as a command line. The string can therefore be the name and arguments of any executable program or UNIX system shell command. If the exact arguments vary from one execution to the next, you may want to use **sprintf** to format the string before issuing the **system** command. When the command has finished running, **system** returns the shell exit status to your program. Execution of your program waits for the completion of the command initiated by **system** and then picks up again at the next executable statement.

exec(2)

exec is the name of a family of functions that includes **execv**, **execle**, **execve**, **execlp**, and **execvp**. They all have the function of transforming the calling process into a new process. The reason for the variety is to provide different ways of pulling together and presenting the arguments of the function. An example of one version (**execl**) might be:

```
execl("/bin/prog2", "prog", progarg1, progarg2,
   (char *)0);
```

For **execl** the argument list is

/bin/prog2	path name of the new process file
prog	the name the new process gets in its argv[0]
progarg1, progarg2	arguments to *prog2* as char *'s
(char *)0	null char pointer to mark end of arguments

Check the manual page in the *Programmer's Reference Manual* for the rest of the details. The key point of the **exec** family is that there is no return from a successful execution: the calling process is finished, the new process overlays the old. The new process also takes over the Process ID and other attributes of the old process. If the call to **exec** is unsuccessful, control is returned to your program with a return value of -1. You can check **errno** (see below) to learn why it failed.

fork(2)

The **fork** system call creates a new process that is an exact copy of the calling process. The new process is known as the child process; the caller is known as the parent process. The one major difference between the two processes is that the child gets its own unique process ID. When the **fork** process has completed successfully, it returns a 0 to the child process and the child's process ID to the parent. If the idea of having two identical processes seems a little funny, consider this:

- Because the return value is different between the child process and the parent, the program can contain the logic to determine different paths.
- The child process could say, "Okay, I'm the child. I'm supposed to issue an **exec** for an entirely different program."
- The parent process could say, "My child is going to be **exec**ing a new process. I'll issue a **wait** until I get word that that process is finished."

To take this out of the storybook world where programs talk like people and into the world of C programming (where people talk like programs), your code might include statements like this:

```
#include <errno.h>

int ch_stat, ch_pid, status;
char *progarg1;
char *progarg2;
void exit();
extern int errno;

     if ((ch_pid = fork()) < 0)
     {
          /* Could not fork...
             check errno
          */
     }
     else if (ch_pid == 0)                /* child */
     {
          (void)execl("/bin/prog2","prog",progarg1,
          progarg2,(char *)0);
          exit(2); /* execl() failed */
     }
     else           /* parent */
     {
          while ((status = wait(&ch_stat)) != ch_pid)
          {
               if (status < 0 && errno == ECHILD)
                 break;
               errno = 0;
          }
     }
```

Figure 2-12: Example of **fork**

Because the child process ID is taken over by the new **exec**'d process, the parent knows the ID. What this boils down to is a way of leaving one program to run another, returning to the point in the first program where processing left off. This is exactly what the **system**(3S) function does. As a matter of fact, **system** accomplishes it through this same procedure of **fork**ing and **exec**ing, with a **wait** in the parent.

Keep in mind that the fragment of code above includes a minimum amount of checking for error conditions. There is also potential confusion about open files and which program is writing to a file. Leaving out the possibility of named files, the new process created by the **fork** or **exec** has the three standard files that are automatically opened: **stdin**, **stdout**, and **stderr**. If the parent has buffered output that should appear before output from the child, the buffers must be flushed before the fork. Also, if the parent and the child process both read input from a stream, whatever is read by one process will be lost to the other. That is, once something has been delivered from the input buffer to a process the pointer has moved on.

Pipes

The idea of using pipes, a connection between the output of one program and the input of another, when working with commands executed by the shell is well established in the UNIX system environment. For example, to learn the number of archive files in your system you might enter a command like:

```
echo /lib/*.a /usr/lib/*.a ¦ wc -w
```

that first echoes all the files in **/lib** and **/usr/lib** that end in **.a**, then pipes the results to the **wc** command, which counts their number.

A feature of the UNIX system/C Language interface is the ability to establish pipe connections between your process and a command to be executed by the shell, or between two cooperating processes. The first uses the **popen**(3S) subroutine that is part of the standard I/O package; the second requires the system call **pipe**(2).

popen is similar in concept to the **system** subroutine in that it causes the shell to execute a command. The difference is that once having invoked **popen** from your program, you have established an open line to a concurrently running process through a stream. You can send characters or strings to this stream with standard I/O subroutines just as you would to **stdout** or to a named file. The connection remains open until your program invokes the companion **pclose** subroutine. A common application of this technique might be a pipe to a printer spooler. For example:

```
#include <stdio.h>

main()
{
     FILE *pptr;
     char *outstring;

     if ((pptr = popen("lp","w")) != NULL)
     {
          for(;;)
          {    .
               .   /* Organize output */
               .
               (void)fprintf(pptr, "%s\n", outstring);
               .
               .
               .
          }
       .
       .
       .
       pclose(pptr);
        }
        .
        .
        .
}
```

Figure 2-13: Example of a **popen** pipe

Error Handling

Within your C programs you must determine the appropriate level of checking for valid data and for acceptable return codes from functions and subroutines. If you use any of the system calls described in Section 2 of the *Programmer's Reference Manual*, you have a way in which you can find out the probable cause of a bad return value.

UNIX system calls that are not able to complete successfully almost always return a value of -1 to your program. (If you look through the system calls in Section 2, you will see that there are a few calls for which no return value is defined, but they are the exceptions.) In addition to the -1 that is returned to the program, the unsuccessful system call places an integer in an externally declared variable, **errno**. You can determine the value in **errno** if your program contains the statement

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

The value in **errno** is not cleared on successful calls, so your program should check it only if the system call returned a -1. The errors are described in **intro**(2) of the *Programmer's Reference Manual*.

The subroutine **perror**(3C) can be used to print an error message (on **stderr**) based on the value of **errno**.

Signals and Interrupts

Signals and interrupts are two words for the same thing. Both words refer to messages passed by the UNIX system to running processes. Generally, the effect is to cause the process to stop running. Some signals are generated if the process attempts to do something illegal; others can be initiated by a user against his or her own processes, or by the super-user against any process.

There is a system call, **kill**, that you can include in your program to send signals to other processes running under your user-id. The format for the **kill** call is:

```
        kill(pid, sig)
```

where **pid** is the process number against which the call is directed, and **sig** is an integer from 1 to 19 that shows the intent of the message. The name "kill" is something of an overstatement; not all the messages have a "drop dead" meaning. Some of the available signals are shown in Figure 2-14 as they are defined in < **sys/signal.h** >.

```
#define  SIGHUP     1    /* hangup */
#define  SIGINT     2    /* interrupt (rubout) */
#define  SIGQUIT    3    /* quit (ASCII FS) */
#define  SIGILL     4    /* illegal instruction (not reset
                         when caught)*/
#define  SIGTRAP    5    /* trace trap (not reset when
                         caught) */
#define  SIGIOT     6    /* IOT instruction */
#define  SIGABRT    6    /* used by abort, replace SIGIOT
                          in the future */
#define  SIGEMT     7    /* EMT instruction */
#define  SIGFPE     8    /* floating point exception */
#define  SIGKILL    9    /* kill (cannot be caught/ignored)*/
#define  SIGBUS     10   /* bus error */
#define  SIGSEGV    11   /* segmentation violation */
#define  SIGSYS     12   /* bad argument to system call */
#define  SIGPIPE    13   /* write on pipe w/ no one to read*/
#define  SIGALRM    14   /* alarm clock */
#define  SIGTERM    15   /* software term. signal from kill*/
#define  SIGUSR1    16   /* user defined signal 1 */
#define  SIGUSR2    17   /* user defined signal 2 */
#define  SIGCLD     18   /* death of a child */
#define  SIGPWR     19   /* power-fail restart */
                         /*SIGWIND/SIGPHONE in UNIX/PC only*/
/*#define SIGWIND   20   */  /* window change */
/*#define SIGPHONE  21   */  /* handset, line status change*/

#define  SIGPOLL 22 /* pollable event occurred */

#define  NSIG       23   /* The valid signal number is from
                          1 to NSIG-1 */
#define  MAXSIG     32   /* size of u_signal[], NSIG-1 <= */
                         /* MAXSIG. MAXSIG is larger than */
                         /* needed. In the future, we can */
                         /* add more signal numbers without*/
                         /* changing user.h */
```

Figure 2-14: Signal Numbers Defined in **/usr/include/sys/signal.h**

The **signal**(2) system call is designed to let you code methods of dealing with incoming signals. You have a three-way choice. You can a) accept whatever the default action is for the signal, b) have your program ignore the signal, or c) write a function of your own to deal with it.

Analysis/Debugging

The UNIX system provides several commands designed to help you discover the causes of problems in programs and to learn about potential problems.

Sample Program

To illustrate how these commands are used and the type of output they produce, we have constructed a sample program that opens and reads an input file and performs one to three subroutines according to options specified on the command line. This program does not do anything you couldn't do quite easily on your pocket calculator, but it does serve to illustrate some points. The source code is shown in Figure 2-15. The header file, **recdef.h**, is shown at the end of the source code.

The output produced by the various analysis and debugging tools illustrated in this section may vary slightly from one installation to another. The *Programmer's Reference Manual* is a good source of additional information about the contents of the reports.

```
              /* Main module -- restate.c */

#include <stdio.h>
#include "recdef.h"

#define TRUE  1
#define FALSE 0

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fopen(), *fin;
    void exit();
    int getopt();
    int oflag = FALSE;
    int pflag = FALSE;
    int rflag = FALSE;
    int ch;
    struct rec first;
    extern int opterr;
    extern float oppty(), pft(), rfe();

              /* restate.c is continued on the next page */
```

Figure 2-15: Source Code for Sample Program (sheet 1 of 4)

```
            /* restate.c continued */

    if (argc < 2)
    {
         (void) fprintf(stderr, "%s: Must specify
            option\n",argv[0]);
         (void) fprintf(stderr, "Usage: %s -rpo\n",
            argv[0]);
         exit(2);
    }

    opterr = FALSE;
    while ((ch = getopt(argc,argv,"opr")) != EOF)
    {
         switch(ch)
         {
         case 'o':
              oflag = TRUE;
              break;
         case 'p':
              pflag = TRUE;
              break;
         case 'r':
              rflag = TRUE;
              break;
         default:
              (void) fprintf(stderr, "Usage: %s -rpo\n",
                 argv[0]);
              exit(2);
         }
    }
    if ((fin = fopen("info","r")) == NULL)
    {
    (void) fprintf(stderr, "%s: cannot open input file
         %s\n",argv[0],"info");
    exit(2);
    }
```

Figure 2-15: Source Code for Sample Program (sheet 2 of 4)

```
                /* restate.c continued */

    if (fscanf(fin,"%s%f%f%f%f%f%f",first.pname,&first.ppx,
    &first.dp,&first.i,&first.c,&first.t,&first.spx) != 7)
    {
      (void) fprintf(stderr,"%s: cannot read first record
         from %s\n",argv[0],"info");
      exit(2);
    }

    printf("Property: %s\n",first.pname);

    if(oflag)
         printf(" Opportunity Cost: $%#5.2f\n",
            oppty(&first));

    if(pflag)
         printf(" Anticipated Profit(loss): $%#7.2f\n",
            pft(&first));

    if(rflag)
         printf(" Return on Funds Employed: %#3.2f%%\n",
            rfe(&first));
}
                /* End of Main Module -- restate.c */

                /* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{
    return(ps->i/12 * ps->t * ps->dp);
}
```

Figure 2-15: Source Code for Sample Program (sheet 3 of 4)

```
                /* Profit -- pft.c */

#include "recdef.h"

float
pft(ps)
struct rec *ps;
{
    return(ps->spx - ps->ppx + ps->c);
}

                /* Return on Funds Employed -- rfe.c */

#include "recdef.h"

float
rfe(ps)
struct rec *ps;
{
    return(100 * (ps->spx - ps->c) / ps->spx);
}

                /* Header File -- recdef.h */

struct rec {        /* To hold input  */
    char pname[25];
    float ppx;
    float dp;
    float i;
    float c;
    float t;
    float spx;
  } ;
```

Figure 2-15: Source Code for Sample Program (sheet 4 of 4)

cflow

cflow produces a chart of the external references in C, **yacc**, **lex**, and assembly language files. Using the modules of our sample program, the command

cflow restate.c oppty.c pft.c rfe.c

produces the output shown in Figure 2-16.

```
1    main: int(), <restate.c 11>
2         fprintf: <>
3         exit: <>
4         getopt: <>
5         fopen: <>
6         fscanf: <>
7         printf: <>
8         oppty: float(), <oppty.c 7>
9         pft: float(), <pft.c 7>
10        rfe: float(), <rfe.c 8>
```

Figure 2-16: **cflow** Output, No Options

The **-r** option looks at the caller:callee relationship from the other side. It produces the output shown in Figure 2-17.

```
1   exit: <>
2       main : <>
3   fopen: <>
4       main : 2
5   fprintf: <>
6       main : 2
7   fscanf: <>
8       main : 2
9   getopt: <>
10      main : 2
11  main: int(), <restate.c 11>
12  oppty: float(), <oppty.c 7>
13      main : 2
14  pft: float(), <pft.c 7>
15      main : 2
16  printf: <>
17      main : 2
18  rfe: float(), <rfe.c 8>
19      main : 2
```

Figure 2-17: **cflow** Output, Using **-r** Option

The **-ix** option causes external and static data symbols to be included. Our sample program has only one such symbol, **opterr**. The output is shown in Figure 2-18.

```
1    main: int(), <restate.c 11>
2        fprintf: <>
3        exit: <>
4        opterr: <>
5        getopt: <>
6        fopen: <>
7        fscanf: <>
8        printf: <>
9        oppty: float(), <oppty.c 7>
10       pft: float(), <pft.c 7>
11       rfe: float(), <rfe.c 8>
```

Figure 2-18: **cflow** Output, Using **-ix** Option

Combining the **-r** and the **-ix** options produces the output shown in Figure 2-19.

```
1    exit: <>
2         main : <>
3    fopen: <>
4         main : 2
5    fprintf: <>
6         main : 2
7    fscanf: <>
8         main : 2
9    getopt: <>
10        main : 2
11   main: int(), <restate.c 11>
12   oppty: float(), <oppty.c 7>
13        main : 2
14   opterr: <>
15        main : 2
16   pft: float(), <pft.c 7>
17        main : 2
18   printf: <>
19        main : 2
20   rfe: float(), <rfe.c 8>
21        main : 2
```

Figure 2-19: **cflow** Output, Using **-r** and **-ix** Options

ctrace

ctrace lets you follow the execution of a C program statement by statement. **ctrace** takes a **.c** file as input and inserts statements in the source code to print out variables as each program statement is executed. You must direct the output of this process to a temporary **.c** file. The temporary file is then used as input to **cc**. When the resulting **a.out** file is executed it produces output that can tell you a lot about what is going on in your program.

Options give you the ability to limit the number of times through loops. You can also include functions in your source file that turn the trace off and on so you can limit the output to portions of the program that are of particular interest.

ctrace accepts only one source code file as input. To use our sample program to illustrate, it is necessary to execute the following four commands:

```
ctrace restate.c > ct.main.c
ctrace oppty.c > ct.op.c
ctrace pft.c > ct.p.c
ctrace rfe.c > ct.r.c
```

The names of the output files are completely arbitrary. Use any names that are convenient for you. The names must end in **.c**, since the files are used as input to the C compilation system.

```
cc -o ct.run ct.main.c ct.op.c ct.p.c ct.r.c
```

Now the command

```
ct.run -opr
```

produces the output shown in Figure 2-20. The command above will cause the output to be directed to your terminal (**stdout**). It is probably a good idea to direct it to a file or to a printer so you can refer to it.

```
 8 main(argc, argv)
23  if (argc < 2)
    /* argc == 2 */
30  opterr = FALSE;
    /* FALSE == 0 */
    /* opterr == 0 */
31  while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argc == 2 */
    /* argv == 15729316 */
    /* ch == 111 or 'o' or "t" */
32  {
33      switch(ch)
        /* ch == 111 or 'o' or "t" */
35      case 'o':
36          oflag = TRUE;
            /* TRUE == 1 or "h" */
            /* oflag == 1 or "h" */
37          break;
48  }
31  while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argc == 2 */
    /* argv == 15729316 */
    /* ch == 112 or 'p' */
32  {
33      switch(ch)
        /* ch == 112 or 'p' */
38      case 'p':
39          pflag = TRUE;
            /* TRUE == 1 or "h" */
            /* pflag == 1 or "h" */
40          break;
48  }
```

Figure 2-20: **ctrace** Output (sheet 1 of 3)

```
31 while ((ch = getopt(argc,argv,"opr")) != EOF)
   /* argc == 2 */
   /* argv == 15729316 */
   /* ch == 114 or 'r' */
32 {
33      switch(ch)
        /* ch == 114 or 'r' */
41      case 'r':
42           rflag = TRUE;
             /* TRUE == 1 or "h" */
             /* rflag == 1 or "h" */
43           break;
48 }
31 while ((ch = getopt(argc,argv,"opr")) != EOF)
   /* argc == 2 */
   /* argv == 15729316 */
   /* ch == -1 */
49 if ((fin = fopen("info","r")) == NULL)
   /* fin == 140200 */
54 if (fscanf(fin,"%s%f%f%f%f%f%f",first.pname,&first.ppx,
   &first.dp,&first.i,&first.c,&first.t,&first.spx) != 7)
   /* fin == 140200 */
   /* first.pname == 15729528 */
61 printf("Property: %s0,first.pname);
   /* first.pname == 15729528 or "Linden_Place" */
      Property: Linden_Place

63 if(oflag)
   /* oflag == 1 or "h" */
64      printf(" Opportunity Cost: $%#5.2f0,oppty(&first));
 5 oppty(ps)
 8 return(ps->i/12 * ps->t * ps->dp);
   /* ps->i == 1069044203 */
   /* ps->t == 1076494336 */
   /* ps->dp == 1088765312 */ Opportunity Cost: $4476.87
```

Figure 2-20: **ctrace** Output (sheet 2 of 3)

```
 66 if(pflag)
    /* pflag == 1 or "h" */
 67       printf("Anticipated Profit(loss):
              $%#7.2f0,pft(&first));
  5 pft(ps)
  8 return(ps->spx - ps->ppx + ps->c);
    /* ps->spx == 1091649040 */
    /* ps->ppx == 1091178464 */
    /* ps->c == 1087409536 */  Anticipated Profit(loss):
                              $85950.00

 69 if(rflag)
    /* rflag == 1 or "h" */
 70       printf("Return on Funds Employed:
              %#3.2f%%0,rfe(&first));
  6 rfe(ps)
  9 return(100 * (ps->spx - ps->c) / ps->spx);
    /* ps->spx == 1091649040 */
    /* ps->c == 1087409536 */  Return on Funds
                              Employed: 94.00%

    /* return */
```

Figure 2-20: **ctrace** Output (sheet 3 of 3)

Using a program that runs successfully is not the optimal way to demonstrate **ctrace**. It would be more helpful to have an error in the operation that could be detected by **ctrace**. It would seem that this utility might be most useful in cases where the program runs to completion, but the output is not as expected.

cxref

cxref analyzes a group of C source code files and builds a cross-reference table of the automatic, static, and global symbols in each file.

The command

```
cxref -c -o cx.op restate.c oppty.c pft.c rfe.c
```

produces the output shown in Figure 2-21 in a file named, in this case, **cx.op**. The **-c** option causes the reports for the four **.c** files to be combined in one cross-reference file.

```
restate.c:

oppty.c:

pft.c:

rfe.c:

  SYMBOL        FILE                    FUNCTION  LINE

BUFSIZ     /usr/include/stdio.h    --        *9
EOF        /usr/include/stdio.h    --        49 *50
           restate.c               --        31
FALSE      restate.c               --        *6 15 16 17 30
FILE       /usr/include/stdio.h    --        *29 73 74
           restate.c               main      12
L_ctermid  /usr/include/stdio.h    --        *80
L_cuserid  /usr/include/stdio.h    --        *81
L_tmpnam   /usr/include/stdio.h    --        *83
NULL       /usr/include/stdio.h    --        46 *47
           restate.c               --        49
P_tmpdir   /usr/include/stdio.h    --        *82
TRUE       restate.c               --        *5 36 39 42
_IOEOF     /usr/include/stdio.h    --        *41
_IOERR     /usr/include/stdio.h    --        *42
_IOFBF     /usr/include/stdio.h    --        *36
_IOLBF     /usr/include/stdio.h    --        *43
_IOMYBUF   /usr/include/stdio.h    --        *40
_IONBF     /usr/include/stdio.h    --        *39
_IOREAD    /usr/include/stdio.h    --        *37
_IORW      /usr/include/stdio.h    --        *44
_IOWRT     /usr/include/stdio.h    --        *38
_NFILE     /usr/include/stdio.h    --        2 *3 73
_SBFSIZ    /usr/include/stdio.h    --        *16
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 1 of 5)

```
  SYMBOL        FILE                 FUNCTION  LINE
_base      /usr/include/stdio.h   --        *26
_bufend()
           /usr/include/stdio.h   --        *57
_bufendtab /usr/include/stdio.h   --        *78
_bufsiz()
           /usr/include/stdio.h   --        *58
_cnt       /usr/include/stdio.h   --        *20
_file      /usr/include/stdio.h   --        *28
_flag      /usr/include/stdio.h   --        *27
_iob       /usr/include/stdio.h   --        *73
           restate.c              main      25 26 45 51 57
_ptr       /usr/include/stdio.h   --        *21
argc       restate.c              --        8
           restate.c              main      *9  23  31
argv       restate.c              --        8
           restate.c              main      *10 25 26 31 45
                                               51 57
c          ./recdef.h             --        *6
           pft.c                  pft       8
           restate.c              main      55
           rfe.c                  rfe       9
ch         restate.c              main      *18  31  33
clearerr()
           /usr/include/stdio.h   --        *67
ctermid()
           /usr/include/stdio.h   --        *77
cuserid()
           /usr/include/stdio.h   --        *77
dp         ./recdef.h                       --*4
           oppty.c                oppty     8
           restate.c              main      55
exit()
           restate.c              main      *13 27 46 52 58
fdopen()
           /usr/include/stdio.h   --        *74
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 2 of 5)

```
 SYMBOL       FILE                 FUNCTION  LINE
feof()
           /usr/include/stdio.h   --        *68
ferror()
           /usr/include/stdio.h   --        *69
fgets()
           /usr/include/stdio.h   --        *77
fileno()
           /usr/include/stdio.h   --        *70
fin        restate.c              main      *12  49  54
first      restate.c              main      *19 54 55 61 64
                                               67 70
fopen()
           /usr/include/stdio.h   --        *74
           restate.c              main      12  49
fprintf    restate.c              main      25 26 45 51 57
freopen()
           /usr/include/stdio.h   --        *74
fscanf     restate.c              main      54
ftell()
           /usr/include/stdio.h   --        *75
getc()
           /usr/include/stdio.h   --        *61
getchar()
           /usr/include/stdio.h   --        *65
getopt()
           restate.c              main      *14  31
gets()
           /usr/include/stdio.h   --        *77
i          ./recdef.h             --        *5
           oppty.c                oppty     8
           restate.c              main      55
lint       /usr/include/stdio.h   --        60
main()
           restate.c              --        *8
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 3 of 5)

```
 SYMBOL        FILE                  FUNCTION  LINE
oflag      restate.c                  main      *15  36  63
oppty()
           oppty.c                    --        *5
           restate.c                  main      *21  64
opterr     restate.c                  main      *20  30
p          /usr/include/stdio.h       --        *57 *58 *61 62
           *62 63 64 67 *67 68 *68 69 *69 70 *70
pdp11      /usr/include/stdio.h       --        11
pflag      restate.c                  main      *16  39  66
pft()
           pft.c                      --        *5
           restate.c                  main      *21  67
pname      ./recdef.h                 --        *2
           restate.c                  main      54  61
popen()
           /usr/include/stdio.h       --        *74
ppx        ./recdef.h                 --        *3
           pft.c                      pft       8
           restate.c                  main      54
printf     restate.c                  main      61 64 67 70
ps         oppty.c                    --        5
           oppty.c                    oppty     *6  8
           pft.c                      --        5
           pft.c                      pft       *6  8
           rfe.c                      --        6
           rfe.c                      rfe       *7  9
putc()
           /usr/include/stdio.h       --        *62
putchar()
           /usr/include/stdio.h       --        *66
rec        ./recdef.h                 --        *1
           oppty.c                    oppty     6
           pft.c                      pft       6
           restate.c                  main      19
           rfe.c                      rfe       7
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 4 of 5)

```
 SYMBOL       FILE                  FUNCTION  LINE
rewind()
              /usr/include/stdio.h  --        *76
rfe()
              restate.c             main      *21   70
              rfe.c                 --        *6
rflag         restate.c             main      *17   42   69
setbuf()
              /usr/include/stdio.h  --        *76
spx           ./recdef.h            --        *8
              pft.c                 pft       8
              restate.c             main      55
              rfe.c                 rfe       9
stderr        /usr/include/stdio.h  --        *55
              restate.c             --        25 26 45 51 57
stdin         /usr/include/stdio.h  --        *53
stdout        /usr/include/stdio.h  --        *54
t             ./recdef.h            --        *7
              oppty.c               oppty     8
              restate.c             main      55
tempnam()
              /usr/include/stdio.h  --        *77
tmpfile()
              /usr/include/stdio.h  --        *74
tmpnam()
              /usr/include/stdio.h  --        *77
u370          /usr/include/stdio.h  --        5
u3b           /usr/include/stdio.h  --        8   19
u3b5          /usr/include/stdio.h  --        8   19
vax           /usr/include/stdio.h  --        8   19
x             /usr/include/stdio.h  --        *62 63 64 66 *66
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 5 of 5)

lint

lint looks for features in a C program that are apt to cause execution errors, that are wasteful of resources, or that create problems of portability.

The command

lint restate.c oppty.c pft.c rfe.c

produces the output shown in Figure 2-22.

```
restate.c:

restate.c
==============
(71)  warning: main() returns random value to
              invocation environment
oppty.c:
pft.c:
rfe.c:


==============
function returns value which is always ignored
    printf
```

Figure 2-22: **lint** Output

lint has options that will produce additional information. Check the *User's Reference Manual*. The error messages give you the line numbers of some items you may want to review.

prof

prof produces a report on the amount of execution time spent in various portions of your program and the number of times each function is called. The program must be compiled with the **-p** option. When a program that was compiled with that option is run, a file called **mon.out** is produced. **mon.out** and **a.out** (or whatever name identifies your executable file) are input to the **prof** command.

The sequence of steps needed to produce a profile report for our sample program is as follows:

Step 1: Compile the programs with the **-p** option:

```
cc -p restate.c oppty.c pft.c rfe.c
```

Step 2: Run the program to produce a file **mon.out**.

```
a.out -opr
```

Step 3: Execute the **prof** command:

```
prof a.out
```

The example of the output of this last step is shown in Figure 2-23. The figures may vary from one run to another. You will also notice that programs of very small size, like that used in the example, produce statistics that are not overly helpful.

```
%Time    Seconds      Cumsecs     #Calls    msec/call     Name
  50.0       0.03         0.03         3           8.    fcvt
  20.0       0.01         0.04         6           2.    atof
  20.0       0.01         0.05         5           2.    write
  10.0       0.00         0.05         1           5.    fwrite
   0.0       0.00         0.05         1           0.    monitor
   0.0       0.00         0.05         1           0.    creat
   0.0       0.00         0.05         4           0.    printf
   0.0       0.00         0.05         2           0.    profil
   0.0       0.00         0.05         1           0.    fscanf
   0.0       0.00         0.05         1           0.    _doscan
   0.0       0.00         0.05         1           0.    oppty
   0.0       0.00         0.05         1           0.    _filbuf
   0.0       0.00         0.05         3           0.    strchr
   0.0       0.00         0.05         1           0.    strcmp
   0.0       0.00         0.05         1           0.    ldexp
   0.0       0.00         0.05         1           0.    getenv
   0.0       0.00         0.05         1           0.    fopen
   0.0       0.00         0.05         1           0.    _findiop
   0.0       0.00         0.05         1           0.    open
   0.0       0.00         0.05         1           0.    main
   0.0       0.00         0.05         1           0.    read
   0.0       0.00         0.05         1           0.    strcpy
   0.0       0.00         0.05        14            0    ungetc
   0.0       0.00         0.05         4           0.    _doprnt
   0.0       0.00         0.05         1           0.    pft
   0.0       0.00         0.05         1           0.    rfe
   0.0       0.00         0.05         4           0.    _xflsbuf
   0.0       0.00         0.05         1           0.    _wrtchk
   0.0       0.00         0.05         2           0.    _findbuf
   0.0       0.00         0.05         2           0.    isatty
   0.0       0.00         0.05         2           0.    ioctl
   0.0       0.00         0.05         1           0.    malloc
   0.0       0.00         0.05         1           0.    memchr
   0.0       0.00         0.05         1           0.    memcpy
   0.0       0.00         0.05         2           0.    sbrk
   0.0       0.00         0.05         4           0.    getopt
```

Figure 2-23: **prof** Output

size

size produces information on the number of bytes occupied by the three sections (text, data, and bss) of a common object file when the program is brought into main memory to be run. Here are the results of one invocation of the **size** command with our object file as an argument.

```
11832 + 3872 + 2240 = 17944
```

Don't confuse this number with the number of characters in the object file that appears when you do an **ls -l** command. That figure includes the symbol table and other header information that is not used at run time.

strip

strip removes the symbol and line number information from a common object file. When you issue this command the number of characters shown by the **ls -l** command approaches the figure shown by the **size** command, but still includes some header information that is not counted as part of the .text, .data, or .bss section. After the **strip** command has been executed, it is no longer possible to use the file with the **sdb** command.

sdb

sdb stands for Symbolic Debugger, which means you can use the symbolic names in your program to pinpoint where a problem has occurred. You can use **sdb** to debug C, FORTRAN 77, or PASCAL programs. There are two basic ways to use **sdb**: by running your program under control of **sdb**, or by using **sdb** to rummage through a core image file left by a program that failed. The first way lets you see what the program is doing up to the point at which it fails (or to skip around the failure point and proceed with the run). The second method lets you check the status at the moment of failure, which may or may not disclose the reason the program failed.

Chapter 15 contains a tutorial on **sdb** that describes the interactive commands you can use to work your way through your program. For the time being we want to tell you just a couple of key things you need to do when using it.

1. Compile your program(s) with the **-g** option, which causes additional information to be generated for use by **sdb**.
2. Run your program under **sdb** with the command:

   ```
   sdb myprog - srcdir
   ```

   where **myprog** is the name of your executable file (**a.out** is the default), and **srcdir** is an optional list of the directories where source code for your modules may be found. The dash between the two arguments keeps **sdb** from looking for a core image file.

Program Organizing Utilities

The following three utilities are helpful in keeping your programming work organized effectively.

The make Command

When you have a program that is made up of more than one module of code you begin to run into problems of keeping track of which modules are up to date and which need to be recompiled when changes are made in another module. The **make** command is used to ensure that dependencies between modules are recorded so that changes in one module results in the recompilation of dependent programs. Even control of a program as simple as the one shown in Figure 2-15 is made easier through the use of **make**.

The **make** utility requires a description file that you create with an editor. The description file (also referred to by its default name: **makefile**) contains the information used by **make** to keep a target file current. The target file is typically an executable program. A description file contains three types of information:

dependency information	tells the **make** utility the relationship between the modules that comprise the target program.
executable commands	needed to generate the target program. **make** uses the dependency information to determine which executable commands should be passed to the shell for execution.
macro definitions	provide a shorthand notation within the description file to make maintenance easier. Macro definitions can be overridden by information from the command line when the **make** command is entered.

The **make** command works by checking the "last changed" time of the modules named in the description file. When **make** finds a component that has been changed more recently than modules that depend on it, the specified commands (usually compilations) are passed to the shell for execution.

The **make** command takes three kinds of arguments: options, macro definitions, and target filenames. If no description filename is given as an option on the command line, **make** searches the current directory for a file named **makefile** or **Makefile**. Figure 2-24 shows a **makefile** for our sample program.

```
OBJECTS = restate.o oppty.o pft.o rfe.o
all: restate
restate: $(OBJECTS)
    $(CC) $(CFLAGS) $(LDFLAGS) $(OBJECTS) -o restate

$(OBJECTS): ./recdef.h

clean:
    rm -f $(OBJECTS)

clobber: clean
    rm -f restate
```

Figure 2-24: **make** Description File

The following things are worth noticing in this description file:

- It identifies the target, **restate**, as being dependent on the four object modules. Each of the object modules in turn is defined as being dependent on the header file, **recdef.h**, and by default, on its corresponding source file.

- A macro, OBJECTS, is defined as a convenient shorthand for referring to all of the component modules.

Whenever testing or debugging results in a change to one of the components of **restate**, for example, a command such as the following should be entered:

```
make CFLAGS=-g restate
```

This has been a very brief overview of the **make** utility. There is more on **make** in Chapter 3, and a detailed description of **make** can be found in Chapter 13.

The Archive

The most common use of an archive file, although not the only one, is to hold object modules that make up a library. The library can be named on the link editor command line (or with a link editor option on the **cc** command line). This causes the link editor to search the symbol table of the archive file when attempting to resolve references.

The **ar** command is used to create an archive file, to manipulate its contents and to maintain its symbol table. The structure of the **ar** command is a little different from the normal UNIX system arrangement of command line options. When you enter the **ar** command you include a one-character key from the set **drqtpmx** that defines the type of action you intend. The key may be combined with one or more additional characters from the set **vuaibcls** that modify the way the requested operation is performed. The makeup of the command line is

ar -key [*posname*] *afile* [*name*]...

where *posname* is the name of a member of the archive and may be used with some optional key characters to make sure that the files in your archive are in a particular order. The *afile* argument is the name of your archive file. By convention, the suffix **.a** is used to indicate the named file is an archive file. (**libc.a**, for example, is the archive file that contains many of the object files of the standard C subroutines.) One or more *names* may be furnished. These identify files that are subjected to the action specified in the *key*.

We can make an archive file to contain the modules used in our sample program, **restate**. The command to do this is

```
ar -rv rste.a restate.o oppty.o pft.o rfe.o
```

If these are the only **.o** files in the current directory, you can use shell metacharacters as follows:

```
ar -rv rste.a *.o
```

Either command will produce this feedback:

```
a - restate.o
a - oppty.o
a - pft.o
a - rfe.o
ar: creating rste.a
```

The **nm** command is used to get a variety of information from the symbol table of common object files. The object files can be, but don't have to be, in an archive file. Figure 2-25 shows the output of this command when executed with the **-f** (for full) option on the archive we just created. The object files were compiled with the **-g** option.

```
Symbols from rste.a[restate.o]

Name        Value   Class             Type      Size   Line   Section

.0fake     |      |        |          strtag | struct |  16  |
restate.c  |      | file   |                 |        |      |
_cnt       |    0 | strmem |             int |        |      |
_ptr       |    4 | strmem |          *Uchar |        |      |
_base      |    8 | strmem |          *Uchar |        |      |
_flag      |   12 | strmem |            char |        |      |
_file      |   13 | strmem |            char |        |      |
.eos       |      | endstr |                 |     16 |      |
rec        |      | strtag |          struct |     52 |      |
pname      |    0 | strmem |        char[25] |     25 |      |
ppx        |   28 | strmem |           float |        |      |
dp         |   32 | strmem |           float |        |      |
i          |   36 | strmem |           float |        |      |
c          |   40 | strmem |           float |        |      |
t          |   44 | strmem |           float |        |      |
spx        |   48 | strmem |           float |        |      |
.eos       |      | endstr |                 |     52 |      |
main       |    0 | extern |           int( ) |   520 |      | .text
.bf        |   10 | fcn    |                 |        |  11  | .text
argc       |    0 | argm't |             int |        |      |
argv       |    4 | argm't |          **char |        |      |
fin        |    0 | auto   | *struct-.0fake  |     16 |      |
oflag      |    4 | auto   |             int |        |      |
pflag      |    8 | auto   |             int |        |      |
rflag      |   12 | auto   |             int |        |      |
ch         |   16 | auto   |             int |        |      |
```

Figure 2-25: **nm** Output, with **-f** Option (sheet 1 of 5)

```
Symbols from rste.a[restate.o]

Name      Value   Class          Type          Size   Line   Section

first    |   20 | auto   |       struct-rec |   52 |      |
.ef      |  518 | fcn    |                  |      |   61 | .text
FILE     |      | typdef | struct-.0fake    |   16 |      |
.text    |    0 | static |                  |   31 |   39 | .text
.data    |  520 | static |                  |      |    4 | .data
.bss     |  824 | static |                  |      |      | .bss
_iob     |    0 | extern |                  |      |      |
fprintf  |    0 | extern |                  |      |      |
exit     |    0 | extern |                  |      |      |
opterr   |    0 | extern |                  |      |      |
getopt   |    0 | extern |                  |      |      |
fopen    |    0 | extern |                  |      |      |
fscanf   |    0 | extern |                  |      |      |
printf   |    0 | extern |                  |      |      |
oppty    |    0 | extern |                  |      |      |
pft      |    0 | extern |                  |      |      |
rfe      |    0 | extern |                  |      |      |
```

Figure 2-25: **nm** Output, with **-f** Option (sheet 2 of 5)

```
Symbols from rste.a[oppty.o]

Name      Value   Class          Type      Size   Line   Section

oppty.c  |       | file     |             |      |      |
rec      |       | strtag   |      struct |  52  |      |
pname    |     0 | strmem   |   char[25]  |  25  |      |
ppx      |    28 | strmem   |       float |      |      |
dp       |    32 | strmem   |       float |      |      |
i        |    36 | strmem   |       float |      |      |
c        |    40 | strmem   |       float |      |      |
t        |    44 | strmem   |       float |      |      |
spx      |    48 | strmem   |       float |      |      |
.eos     |       | endstr   |             |  52  |      |
oppty    |     0 | extern   |     float() |  64  |      | .text
.bf      |    10 | fcn      |             |      |    7 | .text
ps       |     0 | argm't   | *struct-rec |  52  |      |
.ef      |    62 | fcn      |             |      |    3 | .text
.text    |     0 | static   |             |    4 |    1 | .text
.data    |    64 | static   |             |      |      | .data
.bss     |    72 | static   |             |      |      | .bss
```

Figure 2-25: **nm** Output, with **-f** Option (sheet 3 of 5)

```
Symbols from rste.a[pft.o]

Name      Value    Class          Type      Size   Line   Section

pft.c   |       | file     |              |      |      |
rec     |       | strtag   |       struct |  52  |      |
pname   |     0 | strmem   |    char[25]  |  25  |      |
ppx     |    28 | strmem   |        float |      |      |
dp      |    32 | strmem   |        float |      |      |
i       |    36 | strmem   |        float |      |      |
c       |    40 | strmem   |        float |      |      |
t       |    44 | strmem   |        float |      |      |
spx     |    48 | strmem   |        float |      |      |
..eos   |       | endstr   |              |  52  |      |
pft     |     0 | extern   |      float() |  60  |      | .text
..bf    |    10 | fcn      |              |      |    7 | .text
ps      |     0 | argm't   |  *struct-rec |  52  |      |
..ef    |    58 | fcn      |              |      |    3 | .text
..text  |     0 | static   |              |   4  |      | .text
..data  |    60 | static   |              |      |      | .data
..bss   |    60 | static   |              |      |      | .bss
```

Figure 2-25: **nm** Output, with **-f** Option (sheet 4 of 5)

Symbols from rste.a[rfe.o]

Name	Value	Class	Type	Size	Line	Section
rfe.c		file				
rec		strtag	struct	52		
pname	0	strmem	char[25]	25		
ppx	28	strmem	float			
dp	32	strmem	float			
i	36	strmem	float			
c	40	strmem	float			
t	44	strmem	float			
spx	48	strmem	float			
.eos		endstr		52		
rfe	0	extern	float()	68		.text
.bf	10	fcn			8	.text
ps	0	argm't	*struct-rec	52		
.ef	64	fcn			3	.text
.text	0	static		4	1	.text
.data	68	static				.data
.bss	76	static				.bss

Figure 2-25: **nm** Output, with **-f** Option (sheet 5 of 5)

For nm to work on an archive file all of the contents of the archive have to be object modules. If you have stored other things in the archive, you will get the message:

```
nm: rste.a  bad magic
```

when you try to execute the command.

Use of SCCS by Single-User Programmers

The UNIX system Source Code Control System (SCCS) is a set of programs designed to keep track of different versions of programs. When a program has been placed under control of SCCS, only a single copy of any one version of the code can be retrieved for editing at a given time. When program code is changed and the program returned to SCCS, only the changes are recorded. Each version of the code is identified by its SID, or **S**CCS **ID**entifying number. By specifying the SID when the code is extracted from the SCCS file, it is possible to return to an earlier version. If an early version is extracted with the intent of editing it and returning it to SCCS, a new branch of the development tree is started. The set of programs that make up SCCS appear as UNIX system commands. The commands are:

admin
get
delta
prs
rmdel
cdc
what
sccsdiff
comb
val

It is most common to think of SCCS as a tool for project control of large programming projects. It is, however, entirely possible for any individual user of the UNIX system to set up a private SCCS system. Chapter 14 is an SCCS user's guide.

Chapter 3: Application Programming

Introduction

This chapter deals with programming where the objective is to produce sets of programs (applications) that will run on a UNIX system computer.

The chapter begins with a discussion of how the ground rules change as you move up the scale from writing programs that are essentially for your own private use (we have called this single-user programming), to working as a member of a programming team developing an application that is to be turned over to others to use.

There is a section on how the criteria for selecting appropriate programming languages may be influenced by the requirements of the application.

The next three sections of the chapter deal with a number of loosely-related topics that are of importance to programmers working in the application development environment. Most of these mirror topics that were discussed in Chapter 2, Programming Basics, but here we try to point out aspects of the subject that are particularly pertinent to application programming. They are covered under the following headings:

Advanced Programming	deals with such topics as File and Record Locking, Interprocess Communication, and programming terminal screens.
Support Tools	covers the Common Object File Format, link editor directives, shared libraries, SDB, and **lint**.
Project Control Tools	includes some discussion of **make** and SCCS.

The chapter concludes with a description of a sample application called **liber** that uses several of the components described in earlier portions of the chapter.

Application Programming

The characteristics of the application programming environment that make it different from single-user programming have at their base the need for interaction and for sharing of information.

Numbers

Perhaps the most obvious difference between application programming and single-user programming is in the quantities of the components. Not only are applications generally developed by teams of programmers, but the number of separate modules of code can grow into the hundreds on even a fairly simple application.

When more than one programmer works on a project, there is a need to share such information as:

- the operation of each function
- the number, identity and type of arguments expected by a function
- if pointers are passed to a function, are the objects being pointed to modified by the called function, and what is the lifetime of the pointed-to object
- the data type returned by a function

In an application, there is an odds-on possibility that the same function can be used in many different programs, by many different programmers. The object code needs to be kept in a library accessible to anyone on the project who needs it.

Portability

When you are working on a program to be used on a single model of a computer, your concerns about portability are minimal. In application development, on the other hand, a desirable objective often is to produce code that will run on many different UNIX system computers. Some of the things that affect portability will be touched on later in this chapter.

Documentation

A single-user program has modest needs for documentation. There should be enough to remind the program's creator how to use it, and what the intent was in portions of the code.

On an application development project there is a significant need for two types of internal documentation:

- comments throughout the source code that enable successor programmers to understand easily what is happening in the code. Applications can be expected to have a useful life of 5 or more years, and frequently need to be modified during that time. It is not realistic to expect that the same person who wrote the program will always be available to make modifications. Even if that does happen the comments will make the maintenance job a lot easier.
- hard-copy descriptions of functions should be available to all members of an application development team. Without them it is difficult to keep track of available modules, which can result in the same function being written over again.

Unless end-users have clear, readily-available instructions in how to install and use an application they either will not do it at all (if that is an option), or do it improperly.

The microcomputer software industry has become ever more keenly aware of the importance of good end-user documentation. There are cases on record where the success of a software package has been attributed in large part to the fact that it had exceptionally good documentation. There are also cases where a pretty good piece of software was not widely used due to the inaccessibility of its manuals. There appears to be no truth to the rumor that in one or two cases, end-users have thrown the software away and just read the manual.

Project Management

Without effective project management, an application development project is in trouble. This subject will not be dealt with in this guide, except to mention the following three things that are vital functions of project management:

- tracking dependencies between modules of code
- dealing with change requests in a controlled way
- seeing that milestone dates are met

Language Selection

In this section we talk about some of the considerations that influence the selection of programming languages, and describe two of the special purpose languages that are part of the UNIX system environment.

Influences

In single-user programming the choice of language is often a matter of personal preference; a language is chosen because it is the one the programmer feels most comfortable with.

An additional set of considerations comes into play when making the same decision for an application development project.

> Is there an existing standard within the organization that should be observed?
>
> > A firm may decide to emphasize one language because a good supply of programmers is available who are familiar with it.
>
> Does one language have better facilities for handling the particular algorithm?
>
> > One would like to see all language selection based on such objective criteria, but it is often necessary to balance this against the skills of the organization.
>
> Is there an inherent compatibility between the language and the UNIX operating system?
>
> > This is sometimes the impetus behind selecting C for programs destined for a UNIX system machine.
>
> Are there existing tools that can be used?
>
> > If parsing of input lines is an important phase of the application, perhaps a parser generator such as **yacc** should be employed to develop what the application

needs.

Does the application integrate other software into the whole package?

> If, for example, a package is to be built around an existing data base management system, there may be constraints on the variety of languages the data base management system can accommodate.

Special Purpose Languages

The UNIX system contains a number of tools that can be included in the category of special purpose languages. Three that are especially interesting are **awk**, **lex**, and **yacc**.

What awk Is Like

The **awk** utility scans an ASCII input file record by record, looking for matches to specific patterns. When a match is found, an action is taken. Patterns and their accompanying actions are contained in a specification file referred to as the program. The program can be made up of a number of statements. However, since each statement has the potential for causing a complex action, most **awk** programs consist of only a few. The set of statements may include definitions of the pattern that separates one record from another (a newline character, for example), and what separates one field of a record from the next (white space, for example). It may also include actions to be performed before the first record of the input file is read, and other actions to be performed after the final record has been read. All statements in between are evaluated in order for each record in the input file. To paraphrase the action of a simple **awk** program, it would go something like this:

> Look through the input file.
> Every time you see this specific pattern, do this action.

A more complex **awk** program might be paraphrased like this:

First do some initialization.
Then, look through the input file.
Every time you see this specific pattern, do this action.
Every time you see this other pattern, do another action.
After all the records have been read, do these final things.

The directions for finding the patterns and for describing the actions can get pretty complicated, but the essential idea is as simple as the two sets of statements above.

One of the strong points of **awk** is that once you are familiar with the language syntax, programs can be written very quickly. They don't always run very fast, however, so they are seldom appropriate if you want to run the same program repeatedly on a large quantities of records. In such a case, it is likely to be better to translate the program to a compiled language.

How awk Is Used

One typical use of **awk** would be to extract information from a file and print it out in a report. Another might be to pull fields from records in an input file, arrange them in a different order and pass the resulting rearranged data to a function that adds records to your data base. There is an example of a use of **awk** in the sample application at the end of this chapter.

Where to Find More Information

The manual page for **awk** is in Section (1) of the *User's Reference Manual*. Chapter 4 in Part 2 of this guide contains a description of the **awk** syntax and a number of examples showing ways in which **awk** may be used.

What lex and yacc Are Like

lex and **yacc** are often mentioned in the same breath because they perform complementary parts of what can be viewed as a single task: making sense out of input. The two utilities also share the common characteristic of producing source code for C language subroutines from specifications that appear on the surface to be quite similar.

Recognizing input is a recurring problem in programming. Input can be from various sources. In a language compiler, for example, the input is normally contained in a file of source language statements. The UNIX system shell language most often receives its input from a person keying in commands from a terminal. Frequently, information coming out of one program is fed into another where it must be evaluated.

The process of input recognition can be subdivided into two tasks: lexical analysis and parsing, and that's where **lex** and **yacc** come in. In both utilities, the specifications cause the generation of C language subroutines that deal with streams of characters; **lex** generates subroutines that do lexical analysis while **yacc** generates subroutines that do parsing.

To describe those two tasks in dictionary terms:

> Lexical analysis has to do with identifying the words or vocabulary of a language as distinguished from its grammar or structure.
>
> Parsing is the act of describing units of the language grammatically. Students in elementary school are often taught to do this with sentence diagrams.

Of course, the important thing to remember here is that in each case the rules for our lexical analysis or parsing are those we set down ourselves in the **lex** or **yacc** specifications. Because of this, the dividing line between lexical analysis and parsing sometimes becomes fuzzy.

The fact that **lex** and **yacc** produce C language source code means that these parts of what may be a large programming project can be separately maintained. The generated source code is processed by the C compiler to produce an object file. The object file can be link edited with others to produce programs that then perform whatever process follows from the recognition of the input.

How lex Is Used

A **lex** subroutine scans a stream of input characters and waves a flag each time it identifies something that matches one or another of its rules. The waved flag is referred to as a token. The rules are stated in a format that closely resembles the one used by the UNIX system text editor for regular expressions. For example,

```
[ \t]+
```

describes a rule that recognizes a string of one or more blanks or tabs (without mentioning any action to be taken). A more complete statement of that rule might have this notation:

```
[ \t]+  ;
```

which, in effect, says to ignore white space. It carries this meaning because no action is specified when a string of one or more blanks or tabs is recognized. The semicolon marks the end of the statement. Another rule, one that does take some action, could be stated like this:

```
[0-9]+  {
        i = atoi(yytext);
        return(NBR);
        }
```

This rule depends on several things:

NBR must have been defined as a token in an earlier part of the **lex** source code called the declaration section. (It may be in a header file which is **#include**'d in the declaration section.)

i is declared as an **extern int** in the declaration section.

It is a characteristic of **lex** that things it finds are made available in a character string called **yytext**.

Actions can make use of standard C syntax. Here, the standard C subroutine, **atoi**, is used to convert the string to an integer.

What this rule boils down to is **lex** saying, "Hey, I found the kind of token we call NBR, and its value is now in **i**."

To review the steps of the process:

1. The **lex** specification statements are processed by the **lex** utility to produce a file called **lex.yy.c**. (This is the standard name for a file generated by **lex**, just as **a.out** is the standard name for the executable file generated by the link editor.)
2. **lex.yy.c** is transformed by the C compiler (with a **-c** option) into an object file called **lex.yy.o** that contains a subroutine called **yylex()**.
3. **lex.yy.o** is link edited with other subroutines. Presumably one of those subroutines will call **yylex()** with a statement such as:

   ```
   while((token = yylex()) != 0)
   ```

   and other subroutines (or even **main**) will deal with what comes back.

Where to Find More Information

The manual page for **lex** is in Section (1) of the *Programmer's Reference Manual.* A tutorial on **lex** is contained in Chapter 5 in Part 2 of this guide.

How yacc Is Used

yacc subroutines are produced by pretty much the same series of steps as **lex**:

1. The **yacc** specification is processed by the **yacc** utility to produce a file called **y.tab.c**.
2. **y.tab.c** is compiled by the C compiler producing an object file, **y.tab.o**, that contains the subroutine **yyparse()**. A significant difference is that **yyparse()** calls a subroutine called **yylex()** to perform lexical analysis.

3. The object file **y.tab.o** may be link edited with other subroutines, one of which will be called **yylex()**.

There are two things worth noting about this sequence:

1. The parser generated by the **yacc** specifications calls a lexical analyzer to scan the input stream and return tokens.
2. While the lexical analyzer is called by the same name as one produced by **lex**, it does not have to be the product of a **lex** specification. It can be any subroutine that does the lexical analysis.

What really differentiates these two utilities is the format for their rules. As noted above, **lex** rules are regular expressions like those used by UNIX system editors. **yacc** rules are chains of definitions and alternative definitions, written in Backus-Naur form, accompanied by actions. The rules may refer to other rules defined further down the specification. Actions are sequences of C language statements enclosed in braces. They frequently contain numbered variables that enable you to reference values associated with parts of the rules. An example might make that easier to understand:

```
%token  NUMBER
%%
expr    : numb                  { $$ = $1; }
        | expr '+' expr                 { $$ = $1 + $3; }
        | expr '-' expr                 { $$ = $1 - $3; }
        | expr '*' expr                 { $$ = $1 * $3; }
        | expr '/' expr                 { $$ = $1 / $3; }
        | '(' expr ')'          { $$ = $2; }
        ;
numb    : NUMBER                { $$ = $1; }
        ;
```

This fragment of a **yacc** specification shows

- NUMBER identified as a token in the declaration section
- the start of the rules section indicated by the pair of percent signs
- a number of alternate definitions for *expr* separated by the ¦ sign and terminated by the semicolon
- actions to be taken when a rule is matched
- within actions, numbered variables used to represent components of the rule:

  $$ means the value to be returned as the value of the whole rule

  $*n* means the value associated with the *n*th component of the rule, counting from the left
- *numb* defined as meaning the token NUMBER. This is a trivial example that illustrates that one rule can be referenced within another, as well as within itself.

As with **lex**, the compiled **yacc** object file will generally be link edited with other subroutines that handle processing that takes place after the parsing – or even ahead of it.

Where to Find More Information

The manual page for **yacc** is in Section (1) of the *Programmer's Reference Manual.* A detailed description of **yacc** may be found in Chapter 6 of this guide.

Advanced Programming Tools

In Chapter 2 we described the use of such basic elements of programming in the UNIX system environment as the standard I/O library, header files, system calls and subroutines. In this section we introduce tools that are more apt to be used by members of an application development team than by a single-user programmer. The section contains material on the following topics:

- memory management
- file and record locking
- interprocess communication
- programming terminal screens

Memory Management

There are situations where a program needs to ask the operating system for blocks of memory. It may be, for example, that a number of records have been extracted from a data base and need to be held for some further processing. Rather than writing them out to a file on secondary storage and then reading them back in again, it is likely to be a great deal more efficient to hold them in memory for the duration of the process. (This is not to ignore the possibility that portions of memory may be paged out before the program is finished; but such an occurrence is not pertinent to this discussion.) There are two C language subroutines available for acquiring blocks of memory and they are both called **malloc**. One of them is **malloc**(3C), the other is **malloc**(3X). Each has several related commands that do specialized tasks in the same area. They are:

- **free** – to inform the system that space is being relinquished
- **realloc** – to change the size and possibly move the block
- **calloc** – to allocate space for an array and initialize it to zeros

In addition, **malloc**(3X) has a function, **mallopt**, that provides for control over the space allocation algorithm, and a structure, **mallinfo**, from which the program can get information about the usage of the allocated space.

malloc(3X) runs faster than the other version. It is loaded by specifying

-lmalloc

on the **cc**(1) or **ld**(1) command line to direct the link editor to the proper library. When you use **malloc**(3X) your program should contain the statement

```
#include <malloc.h>
```

where the values for **mallopt** options are defined.

See the *Programmer's Reference Manual* for the formal definitions of the two **mallocs**.

File and Record Locking

The provision for locking files, or portions of files, is primarily used to prevent the sort of error that can occur when two or more users of a file try to update information at the same time. The classic example is the airlines reservation system where two ticket agents each assign a passenger to Seat A, Row 5 on the 5 o'clock flight to Detroit. A locking mechanism is designed to prevent such mishaps by blocking Agent B from even seeing the seat assignment file until Agent A's transaction is complete.

File locking and record locking are really the same thing, except that file locking implies the whole file is affected; record locking means that only a specified portion of the file is locked. (Remember, in the UNIX system, file structure is undefined; a record is a concept of the programs that use the file.)

Two types of locks are available: read locks and write locks. If a process places a read lock on a file, other processes can also read the file but all are prevented from writing to it, that is, changing any of the data. If a process places a write lock on a file, no other processes can read or write in the file until the lock is removed. Write locks are also known as exclusive locks. The term shared lock is sometimes applied to read locks.

Another distinction needs to be made between mandatory and advisory locking. Mandatory locking means that the discipline is enforced automatically for the system calls that read, write or create files. This is done through a permission flag established by the file's owner (or the super-user). Advisory locking means that the processes that use the file take the responsibility for setting and removing locks as needed. Thus mandatory may sound like a simpler and better deal, but it isn't so. The mandatory locking capability is included in the system to comply with an agreement with */usr/group*, an organization that represents the interests of UNIX system users. The principal weakness in the mandatory method is that the lock is in place only while the single system call is being made. It is extremely common for a single transaction to require a series of reads and writes before it can be considered complete. In cases like this, the term atomic is used to describe a transaction that must be viewed as an indivisible unit. The preferred way to manage locking in such a circumstance is to make certain the lock is in place before any I/O starts, and that it is not removed until the transaction is done. That calls for locking of the advisory variety.

How File and Record Locking Works

The system call for file and record locking is **fcntl**(2). To bring in the header file shown in Figure 3-1, include the line:

```
#include <fcntl.h>
```

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define  O_RDONLY  0
#define  O_WRONLY  1
#define  O_RDWR    2
#define  O_NDELAY  04      /* Non-blocking I/O */
#define  O_APPEND  010     /* append (writes guaranteed
                              at the end) */
```

```
- CONTINUED -

#define  O_SYNC           020/* synchronous write option */
/* Flag values accessible only to open(2) */
#define  O_CREAT   00400  /* open with file create (uses
                             third open arg)*/
#define  O_TRUNC   01000  /* open with truncation */
#define  O_EXCL    02000  /* exclusive open */
/* fcntl(2) requests */
#define  F_DUPFD   0      /* Duplicate fildes */
#define  F_GETFD   1      /* Get fildes flags */
#define  F_SETFD   2      /* Set fildes flags */
#define  F_GETFL   3      /* Get file flags */
#define  F_SETFL   4      /* Set file flags */
#define  F_GETLK   5      /* Get file lock */
#define  F_SETLK   6      /* Set file lock */
#define  F_SETLKW  7      /* Set file lock and wait */
#define  F_CHKFL   8      /* Check legality of file
                             flag changes */
/* file segment locking set data type - information
                             passed to system by user */
struct flock {
         short     l_type;
         short     l_whence;
         long      l_start;
         long      l_len;  /* len = 0 means until
                             end of file */
         short     l_sysid;
         short     l_pid;
};
/* file segment locking types */
         /* Read lock */
#define  F_RDLCK   01
         /* Write lock */
#define  F_WRLCK   02
         /* Remove lock(s) */
#define  F_UNLCK   03
```

Figure 3-1: The **fcntl.h** Header File

The format of the **fcntl**(2) system call is

```
int fcntl(fildes, cmd, arg)
int fildes, cmd, arg;
```

fildes is the file descriptor returned by the **open** system call. In addition to defining tags that are used as the commands on **fcntl** system calls, **fcntl.h** includes the declaration for a *struct flock* that is used to pass values that control where locks are to be placed.

lockf

A subroutine, **lockf**(3), can also be used to lock sections of a file or an entire file. The format of **lockf** is:

```
#include <unistd.h>

int lockf (fildes, function, size)
int fildes, function;
long size;
```

fildes is the file descriptor; *function* is one of four control values defined in **unistd.h** that let you lock, unlock, test and lock, or simply test to see if a lock is already in place. *size* is the number of contiguous bytes to be locked or unlocked. The section of contiguous bytes can be either forward or backward from the current offset in the file. (You can arrange to be somewhere in the middle of the file by using the **lseek**(2) system call.)

Where to Find More Information

There is an example of file and record locking in the sample application at the end of this chapter. The manual pages that apply to this facility are **fcntl**(2), **fcntl**(5), **lockf**(3), and **chmod**(2) in the *Programmer's Reference Manual.* Chapter 7 in Part 2 of this guide is a detailed discussion of the subject with a number of examples.

Interprocess Communications

In Chapter 2 we described **forking** and **execing** as methods of communicating between processes. Business applications running on a UNIX system computer often need more sophisticated methods. In applications, for example, where fast response is critical, a number of processes may be brought up at the start of a business day to be constantly available to handle transactions on demand. This cuts out initialization time that can add seconds to the time required to deal with the transaction. To go back to the ticket reservation example again for a moment, if a customer calls to reserve a seat on the 5 o'clock flight to Detroit, you don't want to have to say, "Yes, sir. Just hang on a minute while I start up the reservations program." In transaction driven systems, the normal mode of processing is to have all the components of the application standing by waiting for some sort of an indication that there is work to do.

To meet requirements of this type the UNIX system offers a set of nine system calls and their accompanying header files, all under the umbrella name of Interprocess Communications (IPC).

The IPC system calls come in sets of three; one set each for messages, semaphores, and shared memory. These three terms define three different styles of communication between processes:

messages	communication is in the form of data stored in a buffer. The buffer can be either sent or received.
semaphores	communication is in the form of positive integers with a value between 0 and 32,767. Semaphores may be contained in an array the size of which is determined by the system administrator. The default maximum size for the array is 25.
shared memory	communication takes place through a common area of main memory. One or more processes can attach a segment of memory and as a consequence can share whatever data is placed there.

The sets of IPC system calls are:

msgget	**semget**	**shmget**
msgctl	**semctl**	**shmctl**
msgop	**semop**	**shmop**

IPC get Calls

The **get** calls each return to the calling program an identifier for the type of IPC facility that is being requested.

IPC ctl Calls

The **ctl** calls provide a variety of control operations that include obtaining (IPC_STAT), setting (IPC_SET) and removing (IPC_RMID), the values in data structures associated with the identifiers picked up by the **get** calls.

IPC op Calls

The **op** manual pages describe calls that are used to perform the particular operations characteristic of the type of IPC facility being used. **msgop** has calls that send or receive messages. **semop** (the only one of the three that is actually the name of a system call) is used to increment or decrement the value of a semaphore, among other functions. **shmop** has calls that attach or detach shared memory segments.

Where to Find More Information

An example of the use of some IPC features is included in the sample application at the end of this chapter. The system calls are all located in Section (2) of the **Programmer's Reference Manual**. Don't overlook **intro**(2). It includes descriptions of the data structures that are used by IPC facilities. A detailed description of IPC, with many code examples that use the IPC system calls, is contained in Chapter 9 in Part 2 of this guide.

Programming Terminal Screens

The facility for setting up terminal screens to meet the needs of your application is provided by two parts of the UNIX system. The first of these, **terminfo**, is a data base of compiled entries that describe the capabilities of terminals and the way they perform various operations.

The **terminfo** data base normally begins at the directory **/usr/lib/terminfo**. The members of this directory are themselves directories, generally with single-character names that are the first character in the name of the terminal. The compiled files of operating characteristics are at the next level down the hierarchy. For example, the entry for a Teletype 5425 is located in both the file **/usr/lib/terminfo/5/5425** and the file **/usr/lib/terminfo/t/tty5425**.

Describing the capabilities of a terminal can be a painstaking task. Quite a good selection of terminal entries is included in the **terminfo** data base that comes with your 3B2 Computer. However, if you have a type of terminal that is not already described in the data base, the best way to proceed is to find a description of one that comes close to having the same capabilities as yours and building on that one. There is a routine (**setupterm**) in **curses**(3X) that can be used to print out descriptions from the data base. Once you have worked out the code that describes the capabilities of your terminal, the **tic**(1M) command is used to compile the entry and add it to the data base.

curses

After you have made sure that the operating capabilities of your terminal are a part of the **terminfo** data base, you can then proceed to use the routines that make up the **curses**(3X) package to create and manage screens for your application.

The **curses** library includes functions to:

- define portions of your terminal screen as windows
- define pads that extend beyond the borders of your physical terminal screen and let you see portions of the pad on your terminal

- read input from a terminal screen into a program
- write output from a program to your terminal screen
- manipulate the information in a window in a virtual screen area and then send it to your physical screen

Where to Find More Information

In the sample application at the end of this chapter, we show how you might use **curses** routines. Chapter 10 in Part 2 of this guide contains a tutorial on the subject. The manual pages for **curses** are in Section (3X), and those for **terminfo** are in Section (4) of the *Programmer's Reference Manual.*

Programming Support Tools

This section covers UNIX system components that are part of the programming environment, but that have a highly specialized use. We refer to such things as:

- link edit command language
- Common Object File Format
- libraries
- Symbolic Debugger
- **lint** as a portability tool

Link Edit Command Language

The link editor command language is for use when the default arrangement of the **ld** output will not do the job. The default locations for the standard Common Object File Format sections are described in **a.out**(4) in the *Programmer's Reference Manual*. On a 3B2 Computer, when an **a.out** file is loaded into memory for execution, the text segment starts at location 0x80800000, and the data section starts at the next segment boundary after the end of the text. The stack begins at 0xC0020000 and grows to higher memory addresses.

The link editor command language provides directives for describing different arrangements. The two major types of link editor directives are MEMORY and SECTIONS. MEMORY directives can be used to define the boundaries of configured and unconfigured sections of memory within a machine, to name sections, and to assign specific attributes (read, write, execute, and initialize) to portions of memory. SECTIONS directives, among a lot of other functions, can be used to bind sections of the object file to specific addresses within the configured portions of memory.

Why would you want to be able to do those things? Well, the truth is that in the majority of cases you don't have to worry about it. The need to control the link editor output becomes more urgent under two, possibly related, sets of circumstances.

1. Your application is large and consists of a lot of object files.
2. The hardware your application is to run on is tight for space.

Where to Find More Information

Chapter 12 in Part 2 of this guide gives a detailed description of the subject.

Common Object File Format

The details of the Common Object File Format have never been looked on as stimulating reading. In fact, they have been recommended to hard-core insomniacs as preferred bedtime fare. However, if you're going to break into the ranks of really sophisticated UNIX system programmers, you're going to have to get a good grasp of COFF. A knowledge of COFF is fundamental to using the link editor command language. It is also good background knowledge for tasks such as:

- setting up archive libraries or shared libraries
- using the Symbolic Debugger

The following system header files contain definitions of data structures of parts of the Common Object File Format:

<syms.h>	symbol table format
<linenum.h>	line number entries
<ldfcn.h>	COFF access routines
<filehdr.h>	file header for a common object file
<a.out.h>	common assembler and link editor output
<scnhdr.h>	section header for a common object file
<reloc.h>	relocation information for a common object file
<storclass.h>	storage classes for common object files

The object file access routines are described below under the heading "The Object File Library."

Where to Find More Information

Chapter 11 in Part 2 of this guide gives a detailed description of COFF.

Libraries

A library is a collection of related object files and/or declarations that simplify programming effort. Programming groups involved in the development of applications often find it convenient to establish private libraries. For example, an application with a number of programs using a common data base can keep the I/O routines in a library that is searched at link edit time.

Prior to Release 3.0 of the UNIX System V the libraries, whether system supplied or application developed, were collections of common object format files stored in an archive (*filename*.**a**) file that was searched by the link editor to resolve references. Files in the archive that were needed to satisfy unresolved references became a part of the resulting executable.

Beginning with Release 3.0, shared libraries are supported. Shared libraries are similar to archive libraries in that they are collections of object files that are acted upon by the link editor. The difference, however, is that shared libraries perform a static linking between the file in the library and the executable that is the output of **ld**. The result is a saving of space, because all executables that need a file from the library share a single copy. We go into shared libraries later in this section.

In Chapter 2 we described many of the functions that are found in the standard C library, **libc.a**. The next two sections describe two other libraries, the object file library and the math library.

The Object File Library

The object file library provides functions for the access and manipulation of object files. Some functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. The need to work at this level of detail with object files occurs most often in the development of new tools that manipulate object files. For a description of the format of an object file, see "The Common Object File Format" in Chapter 11. This library consists of several portions. The functions reside in **/lib/libld.a** and are loaded during the compilation of a C language program by the **-l** command line option:

cc *file* **-lld**

which causes the link editor to search the object file library. The argument **-lld** must appear after all files that reference functions in **libld.a**.

The following header files must be included in the source code.

```
#include <stdio.h>
#include <a.out.h>
#include <ldfcn.h>
```

Function	Reference	Brief Description
ldaclose	**ldclose**(3X)	Close object file being processed.
ldahread	**ldahread**(3X)	Read archive header.
ldaopen	**ldopen**(3X)	Open object file for reading.
ldclose	**ldclose**(3X)	Close object file being processed.
ldfhread	**ldfhread**(3X)	Read file header of object file being processed.
ldgetname	**ldgetname**(3X)	Retrieve the name of an object file symbol table entry.
ldlinit	**ldlread**(3X)	Prepare object file for reading line number entries via **ldlitem**.
ldlitem	**ldlread**(3X)	Read line number entry from object file after **ldlinit**.
ldlread	**ldlread**(3X)	Read line number entry from object file.
ldlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed given the name of a section.

Function	Reference	Brief Description
ldnrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed given the name of a section.
ldnshread	**ldshread**(3X)	Read section header of the named section of the object file being processed.
ldnsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed given the name of a section.
ldohseek	**ldohseek**(3X)	Seeks to the optional file header of the object file being processed.
ldopen	**ldopen**(3X)	Open object file for reading.
ldrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed.
ldshread	**ldshread**(3X)	Read section header of an object file being processed.
ldsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed.

Function	Reference	Brief Description
ldtbindex	**ldtbindex**(3X)	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	**ldtbread**(3X)	Reads a specific symbol table entry of the object file being processed.
ldtbseek	**ldtbseek**(3X)	Seeks to the symbol table of the object file being processed.
sgetl	**sputl**(3X)	Access long integer data in a machine independent format.
sputl	**sputl**(3X)	Translate a long integer into a machine independent format.

Common Object File Interface Macros (ldfcn.h)

The interface between the calling program and the object file access routines is based on the defined type LDFILE, which is in the header file **ldfcn.h** (see **ldfcn**(4)). The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen**(3X) allocates and initializes the LDFILE structure and returns a pointer to the structure. The fields of the LDFILE structure may be accessed individually through the following macros:

- The TYPE macro returns the magic number of the file, which is used to distinguish between archive files and object files that are not part of an archive.

- The IOPTR macro returns the file pointer, which was opened by **ldopen**(3X) and is used by the input/output functions of the C library.
- The OFFSET macro returns the file address of the beginning of the object file. This value is non-zero only if the object file is a member of the archive file.
- The HEADER macro accesses the file header structure of the object file.

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an LDFILE structure into a reference to its file descriptor field. The available macros are described in **ldfcn**(4) in the *Programmer's Reference Manual.*

The Math Library

The math library package consists of functions and a header file. The functions are located and loaded during the compilation of a C language program by the **-l** option on a command line, as follows:

```
cc file -lm
```

This option causes the link editor to search the math library, **libm.a**. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line:

```
#include <math.h>
```

near the beginning of each file that uses the routines.

The functions are grouped into the following categories:

- trigonometric functions
- Bessel functions
- hyperbolic functions
- miscellaneous functions

Trigonometric Functions

These functions are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double-precision.

Function	Reference	Brief Description
acos	**trig**(3M)	Return arc cosine.
asin	**trig**(3M)	Return arc sine.
atan	**trig**(3M)	Return arc tangent.
atan2	**trig**(3M)	Return arc tangent of a ratio.
cos	**trig**(3M)	Return cosine.
sin	**trig**(3M)	Return sine.
tan	**trig**(3M)	Return tangent.

Bessel Functions

These functions calculate Bessel functions of the first and second kinds of several orders for real values. The Bessel functions are **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. The functions are located in section **bessel**(3M).

Hyperbolic Functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

Function	Reference	Brief Description
cosh	**sinh**(3M)	Return hyperbolic cosine.
sinh	**sinh**(3M)	Return hyperbolic sine.
tanh	**sinh**(3M)	Return hyperbolic tangent.

Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double-precision numbers.

Function	Reference	Brief Description
ceil	**floor**(3M)	Returns the smallest integer not less than a given value.
exp	**exp**(3M)	Returns the exponential function of a given value.
fabs	**floor**(3M)	Returns the absolute value of a given value.
floor	**floor**(3M)	Returns the largest integer not greater than a given value.
fmod	**floor**(3M)	Returns the remainder produced by the division of two given values.
gamma	**gamma**(3M)	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.
hypot	**hypot**(3M)	Return the square root of the sum of the squares of two numbers.

Function	Reference	Brief Description
log	**exp**(3M)	Returns the natural logarithm of a given value.
log10	**exp**(3M)	Returns the logarithm base ten of a given value.
matherr	**matherr**(3M)	Error-handling function.
pow	**exp**(3M)	Returns the result of a given value raised to another given value.
sqrt	**exp**(3M)	Returns the square root of a given value.

Shared Libraries

As noted above, beginning with UNIX System V Release 3.0, shared libraries are supported. Not only are some system libraries (**libc** and the networking library) available in both archive and shared library form, but also applications have the option of creating private application shared libraries.

The reason why shared libraries are desirable is that they save space, both on disk and in memory. With an archive library, when the link editor goes to the archive to resolve a reference it takes a copy of the object file that it needs for the resolution and binds it into the **a.out** file. From that point on the copied file is a part of the executable, whether it is in memory to be run or sitting in secondary storage. If you have a lot of executables that use, say, **printf** (which just happens to require much of the standard I/O library) you can be talking about a sizeable amount of space.

With a shared library, the link editor does not copy code into the executable files. When the operating system starts a process that uses a shared library it maps the shared library contents into the address space of the process. Only one copy of the shared code exists, and many processes can use it at the same time.

This fundamental difference between archives and shared libraries has another significant aspect. When code in an archive library is modified, all existing executables are uneffected. They continue using the older version until they are re-link edited. When code in a shared library is modified, all programs that share that code use the new version the next time they are executed.

All this may sound like a really terrific deal, but as with most things in life there are complications. To begin with, in the paragraphs above we didn't give you quite all the facts. For example, each process that uses shared library code gets its own copy of the entire data region of the library. It is actually only the text region that is really shared. So the truth is that shared libraries can add space to executing **a.out**'s even though the chances are good that they will cause more shrinkage than expansion. What this means is that when there is a choice between using a shared library and an archive, you shouldn't use the shared library unless it saves space. If you were using a shared **libc** to access only **strcmp**, for example, you would pick up more in shared library data than you would save by sharing the text.

The answer to this problem, and to others that are somewhat more complex, is to assign the responsibility for shared libraries to a central person or group within the application. The shared library developer should be the one to resolve questions of when to use shared and when to use archive system libraries. If a private library is to be built for your application, one person or organization should be responsible for its development and maintenance.

Where to Find More Information

The sample application at the end of this chapter includes an example of the use of a shared library. Chapter 8 in Part 2 of this guide describes how shared libraries are built and maintained.

Symbolic Debugger

The use of **sdb** was mentioned briefly in Chapter 2. In this section we want to say a few words about **sdb** within the context of an application development project.

sdb works on a process, and enables a programmer to find errors in the code. It is a tool a programmer might use while coding and unit testing a program, to make sure it runs according to its design. **sdb** would normally be used prior to the time the program is turned over, along with the rest of the application, to testers. During this phase of the application development cycle programs are compiled with the **-g** option of **cc** to facilitate the use of the debugger. The symbol table should not be stripped from the object file. Once the programmer is satisfied that the program is error-free, **strip**(1) can be used to reduce the file storage overhead taken by the file.

If the application uses a private shared library, the possibility arises that a program bug may be located in a file that resides in the shared library. Dealing with a problem of this sort calls for coordination by the administrator of the shared library. Any change to an object file that is part of a shared library means the change effects all processes that use that file. One program's bug may be another program's feature.

Where to Find More Information

Chapter 15 in Part 2 of this guide contains information on how to use **sdb**. The manual page is in Section (1) of the *Programmer's Reference Manual.*

lint as a Portability Tool

It is a characteristic of the UNIX system that language compilation systems are somewhat permissive. Generally speaking it is a design objective that a compiler should run fast. Most C compilers, therefore, let some things go unflagged as long as the language syntax is observed statement by statement. This sometimes means that while your program may run, the output will have some surprises. It also sometimes means that while the program may run on the machine on which the compilation system runs, there may be real difficulties in running it on some other machine.

That's where **lint** comes in. **lint** produces comments about inconsistencies in the code. The types of anomalies flagged by **lint** are:

- cases of disagreement between the type of value expected from a called function and what the function actually returns
- disagreement between the types and number of arguments expected by functions and what the function receives
- inconsistencies that might prove to be bugs
- things that might cause portability problems

Here is an example of a portability problem that would be caught by **lint**.

Code such as this:

```
        int i = lseek(fdes, offset, whence)
```

would get by most compilers. However, **lseek** returns a long integer representing the address of a location in the file. On a machine with a 16-bit integer and a bigger **long int**, it would produce incorrect results, because **i** would contain only the last 16 bits of the value returned.

Since it is reasonable to expect that an application written for a UNIX system machine will be able to run on a variety of computers, it is important that the use of **lint** be a regular part of the application development.

Where to Find More Information

Chapter 16 in Part 2 of this guide contains a description of **lint** with examples of the kinds of conditions it uncovers. The manual page is in Section (1) of the *Programmer's Reference Manual.*

Project Control Tools

Volumes have been written on the subject of project control. It is an item of top priority for the managers of any application development team. Two UNIX system tools that can play a role in this area are described in this section.

make

make is extremely useful in an application development project for keeping track of what object files need to be recompiled as changes are made to source code files. One of the characteristics of programs in a UNIX system environment is that they are made up of many small pieces, each in its own object file, that are link edited together to form the executable file. Quite a few of the UNIX system tools are devoted to supporting that style of program architecture. For example, archive libraries, shared libraries and even the fact that the **cc** command accepts **.o** files as well as **.c** files, and that it can stop short of the **ld** step and produce **.o** files instead of an **a.out**, are all important elements of modular architecture. The two main advantages of this type of programming are that

- A file that performs one function can be re-used in any program that needs it.
- When one function is changed, the whole program does not have to be recompiled.

On the flip side, however, a consequence of the proliferation of object files is an increased difficulty in keeping track of what does need to be recompiled, and what doesn't. **make** is designed to help deal with this problem. You use **make** by describing in a specification file, called **makefile**, the relationship (that is, the dependencies) between the different files of your program. Once having done that, you conclude a session in which possibly a number of your source code files have been changed by running the **make** command. **make** takes care of generating a new **a.out** by comparing the time-last-changed of your source code files with the dependency rules you have given it.

make has the ability to work with files in archive libraries or under control of the Source Code Control System (SCCS).

Where to Find More Information

The **make**(1) manual page is contained in the *Programmer's Reference Manual.* Chapter 13 in Part 2 of this guide gives a complete description of how to use **make.**

SCCS

SCCS is an acronym for Source Code Control System. It consists of a set of 14 commands used to track evolving versions of files. Its use is not limited to source code; any text files can be handled, so an application's documentation can also be put under control of SCCS. SCCS can:

- store and retrieve files under its control
- allow no more than a single copy of a file to be edited at one time
- provide an audit trail of changes to files
- reconstruct any earlier version of a file that may be wanted

SCCS files are stored in a special coded format. Only through commands that are part of the SCCS package can files be made available in a user's directory for editing, compiling, etc. From the point at which a file is first placed under SCCS control, only changes to the original version are stored. For example, let's say that the program, **restate**, that was used in several examples in Chapter 2, was controlled by SCCS. One of the original pieces of that program is a file called **oppty.c** that looks like this:

```
                    /* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{
                    return(ps->i/12 * ps->t * ps->dp);
}
```

If you decide to add a message to this funtion, you might change the file like this:

```
                    /* Opportunity Cost -- oppty.c */
#include "recdef.h"
#include <stdio.h>

float
oppty(ps)
struct rec *ps;
{
                    (void) fprintf(stderr, "Opportunity
                       calling\n");
                    return(ps->i/12 * ps->t * ps->dp);
}
```

SCCS saves only the two new lines from the second version, with a coded notation that shows where in the text the two lines belong. It also includes a note of the version number, lines deleted, lines inserted, total lines in the file, the date and time of the change and the login id of the person making the change.

Where to Find More Information

Chapter 14 in Part 2 of this guide is an SCCS user's guide. SCCS commands are in Section (1) of the *Programmer's Reference Manual.*

liber, A Library System

To illustrate the use of UNIX system programming tools in the development of an application, we are going to pretend we are engaged in the development of a computer system for a library. The system is known as **liber**. The early stages of system development, we assume, have already been completed; feasibility studies have been done, the preliminary design is described in the coming paragraphs. We are going to stop short of producing a complete detailed design and module specifications for our system. You will have to accept that these exist. In using portions of the system for examples of the topics covered in this chapter, we will work from these virtual specifications.

We make no claim as to the efficacy of this design. It is the way it is only in order to provide some passably realistic examples of UNIX system programming tools in use.

liber is a system for keeping track of the books in a library. The hardware consists of a single computer with terminals throughout the library. One terminal is used for adding new books to the data base. Others are used for checking out books and as electronic card catalogs.

The design of the system calls for it to be brought up at the beginning of the day and remain running while the library is in operation. The system has one master index that contains the unique identifier of each title in the library. When the system is running the index resides in memory. Semaphores are used to control access to the index. In the pages that follow fragments of some of the system's programs are shown to illustrate the way they work together. The startup program performs the system initialization; opening the semaphores and shared memory; reading the index into the shared memory; and kicking off the other programs. The id numbers for the shared memory and semaphores (**shmid**, **wrtsem**, and **rdsem**) are read from a file during initialization. The programs all share the in-memory index. They attach it with the following code:

```
/* attach shared memory for index */
if ((int)(index = (INDEX *) shmat(shmid, NULL, 0)) == -1)
{
        (void) fprintf(stderr, "shmat failed: %d\n", errno);
        exit(1);
}
```

Of the programs shown, **add-books** is the only one that alters the index. The semaphores are used to ensure that no other programs will try to read the index while **add-books** is altering it. The checkout program locks the file record for the book, so that each copy being checked out is recorded separately and the book cannot be checked out at two different checkout stations at the same time.

The program fragments do not provide any details on the structure of the index or the book records in the data base.

```
                    /* liber.h - header file for the
                     *          library system.
                     */
typedef ... INDEX;/* data structure for book file index */
typedef struct {  /* type of records in book file */
    char title[30];
    char author[30];
    .
    .
    .
```

- CONTINUED -

```
} BOOK;
int shmid;
int wrtsem;
int rdsem;
INDEX *index;

int book_file;
BOOK book_buf;

/*  startup program*/


/*
 * 1. Open shared memory for file index and read it in.
 * 2. Open two semaphores for providing exclusive write
 *    access to index.
 * 3. Stash id's for shared memory segment and semaphores
 *    in a file
 *    where they can be accessed by the programs.
 * 4. Start programs:  add-books, card-catalog, and checkout
 *    running on the various terminals throughout the
 *    library.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "liber.h"

void exit();
extern int errno;

key_t key;
int shmid;
int wrtsem;
int rdsem;
FILE *ipc_file;
```

- CONTINUED -

```
main()
{
     .
     .
     .
     if ((shmid = shmget(key, sizeof(INDEX), IPC_CREAT |
        0666)) == -1)
     {
          (void) fprintf(stderr, "startup: shmget failed:
             errno=%d\n", errno);
          exit(1);
     }
     if ((wrtsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
     {
          (void) fprintf(stderr, "startup: semget failed:
             errno=%d\n", errno);
          exit(1);
     }
     if ((rdsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
     {
          (void) fprintf(stderr, "startup: semget failed:
             errno=%d\n", errno);
          exit(1);
     }
     (void) fprintf(ipc_file, "%d\n%d\n%d\n", shmid,
             wrtsem, rdsem);

     /*
      * Start the add-books program running on the terminal
      * in the basement. Start the checkout and card-catalog
      * programs running on the various other terminals
      * throughout the library.
      */
     .
     .
     .
}
```

- CONTINUED -

```
/*  card-catalog program*/

/*
 * 1. Read screen for author and title.
 * 2. Use semaphores to prevent reading index while
    it is being written.
 * 3. Use index to get position of book record in book
    file.
 * 4. Print book record on screen or indicate book was
    not found.
 * 5. Go to 1.
 */

#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>
#include <fcntl.h>
#include "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];

main() {
     .
     .
     .

while (1)
     {
          /*
           * Read author/title/subject info. from screen.
           */

          /*
           * Wait for write semaphore to reach 0 (index
           * not being written).
           */
```

- CONTINUED -

```
        sop[0].sem_op = 1;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed:
                   %d\n", errno);
                exit(1);
        }
        /*
         * Increment read semaphore so potential writer
         * will wait for us to finish reading the index.
         */
        sop[0].sem_op = 0;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed:
                   %d\n", errno);
                exit(1);
        }

        /* Use index to find file pointer(s) for book(s) */

        /* Decrement read semaphore */
        sop[0].sem_op = -1;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed:
                   %d\n", errno);
                exit(1);
        }

        /*
         * Now use the file pointers found in the index to
         * read the book file. Then print the information
         * on the book(s) to the screen.
         */
    } /* while */
}
```

```
                          - CONTINUED -

/*   checkout program*/

/*
 * 1. Read screen for Dewey Decimal number of book to be
 *    checked out.
 * 2. Use semaphores to prevent reading index while it is
 *    being written.
 * 3. Use index to get position of book record in book file.
 * 4. If book not found print message on screen, otherwise
 *    lock book record and read.
 * 5. If book already checked out print message on screen,
 *    otherwise mark record "checked out" and write back
 *    to book file.
 * 6. Unlock book record.
 * 7. Go to 1.
 */

#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>
#include <fcntl.h>
#include "liber.h"

void exit();
long lseek();
extern int errno;
struct flock flk;
struct sembuf sop[1];
long bookpos;
main()
{
     .
     .
     .
     while (1)
```

- CONTINUED -

```
{
        /*
         * Read Dewey Decimal number from screen.
         */
        /*
         * Wait for write semaphore to reach 0 (index
         * not being written).
         */
        sop[0].sem_flg = 0;
        sop[0].sem_op = 0;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed:
                   %d\n", errno);
                exit(1);
        }
        /*
         * Increment read semaphore so potential writer
         * will wait for us to finish reading the index.
         */
        sop[0].sem_op = 1;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed:
                   %d\n", errno);
                exit(1);
        }

        /*
         * Now we can use the index to find the book's
         * record position. Assign value to "bookpos".
         */

        /* Decrement read semaphore */
        sop[0].sem_op = -1;
        if (semop(rdsem, sop, 1) == -1)
```

- CONTINUED -

```
{
        (void) fprintf(stderr, "semop failed:
           %d\n", errno);
        exit(1);
}

/* Lock the book's record in book file, read the
        record. */
flk.l_type = F_WRLCK;
flk.l_whence = 0;
flk.l_start = bookpos;
flk.l_len = sizeof(BOOK);
if (fcntl(book_file, F_SETLKW, &flk) == -1)
{
        (void) fprintf(stderr, "trouble locking:
           %d\n", errno);
        exit(1);
}
if (lseek(book_file, bookpos, 0) == -1)
{
        Error processing for lseek;
}
if (read(book_file, &book_buf, sizeof(BOOK)) == -1)
{
        Error processing for read;
}

/*
 * If the book is checked out inform the client,
 * otherwise mark the book's record as checked out
 * and write it back into the book file.
 */

/* Unlock the book's record in book file. */
flk.l_type = F_UNLCK;
if (fcntl(book_file, F_SETLK, &flk) == -1)
```

- CONTINUED -

```
        {
                (void) fprintf(stderr, "trouble unlocking:
                   %d\n", errno);
                exit(1);
        }
    } /* while */
}
/*  add-books program*/

/*
 * 1. Read a new book entry from screen.
 * 2. Insert book in book file.
 * 3. Use semaphore "wrtsem" to block new readers.
 * 4. Wait for semaphore "rdsem" to reach 0.
 * 5. Insert book into index.
 * 6. Decrement wrtsem.
 * 7. Go to 1.
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "liber.h"

void exit();
extern int errno;
struct sembuf sop[1];
BOOK bookbuf;

main()
{
    .
    .
    .
    for (;;)
    {
```

- CONTINUED -

```
/*
 * Read information on new book from screen.
 */

addscr(&bookbuf);

/* write new record at the end of the bookfile.
 * Code not shown, but
 * addscr() returns a 1 if title information has
 * been entered, 0 if not.
 */
/*
 * Increment write semaphore, blocking new readers
 * from accessing the index.
 */
sop[0].sem_flg = 0;
sop[0].sem_op = 1;
if (semop(wrtsem, sop, 1) == -1)
{
         (void) fprintf(stderr, "semop failed:
            %d\n", errno);
         exit(1);
}
/*
 * Wait for read semaphore to reach 0 (all readers
 * to finish using the index).
 */
sop[0].sem_op = 0;
if (semop(rdsem, sop, 1) == -1)
{
         (void) fprintf(stderr, "semop failed:
            %d\n", errno);
         exit(1);
}
```

```
                         - CONTINUED -

        /*
         * Now that we have exclusive access to the index
         * we insert our new book with its file pointer.
         */

        /* Decrement write semaphore, permitting readers
         * to read index.
         */
        sop[0].sem_op = -1;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed:
                   %d\n", errno);
                exit(1);
        }
    } /* for */
    .
    .
    .
}
```

The example following, **addscr()**, illustrates two significant points about **curses** screens:

1. Information read in from a **curses** window can be stored in fields that are part of a structure defined in the header file for the application.
2. The address of the structure can be passed from another function where the record is processed.

```
                 /*  addscr is called from add-books. The
                  *  user is prompted for title info.
                  */
#include <curses.h>

WINDOW *cmdwin;

addscr(bb)
struct BOOK *bb;
{
     int c;

     initscr();
     nonl();
     noecho();
     cbreak();

     cmdwin = newwin(6, 40, 3, 20);
     mvprintw(0, 0, "This screen is for adding titles to
        the data base");
     mvprintw(1, 0, "Enter  a  to add;  q  to quit: ");
     refresh();
     for (;;)
     {
          refresh();
          c = getch();
          switch (c) {
            case 'a':
                   werase(cmdwin);
                   box(cmdwin, '|', '-');
                   mvwprintw(cmdwin, 1, 1, "Enter title: ");
                   wmove(cmdwin, 2, 1);
                   echo();
                   wrefresh(cmdwin);
                   wgetstr(cmdwin, bb->title);
```

- CONTINUED -

```
                noecho();
                werase(cmdwin);
                box(cmdwin, '|', '-');
                mvwprintw(cmdwin, 1, 1, "Enter author: ");
                wmove(cmdwin, 2, 1);
                echo();
                wrefresh(cmdwin);
                wgetstr(cmdwin, bb->author);
                noecho();
                werase(cmdwin);
                wrefresh(cmdwin);
                endwin();
                return(1);

            case 'q':
                erase();
                endwin();
                return(0);
            }
    }
}

#
# Makefile for liber library system
#

CC = cc
CFLAGS = -O
all: startup add-books checkout card-catalog

startup: liber.h startup.c
	$(CC) $(CFLAGS) -o startup startup.c
```

```
                      - CONTINUED -

add-books: add-books.o addscr.o
	$(CC) $(CFLAGS) -o add-books add-books.o addscr.o

add-books.o: liber.h

checkout: liber.h checkout.c
	$(CC) $(CFLAGS) -o checkout checkout.c

card-catalog: liber.h card-catalog.c
	$(CC) $(CFLAGS) -o card-catalog card-catalog.c
```

Chapter 4: awk

Introduction

awk is a file-processing programming language designed to make many common information and retrieval text manipulation tasks easy to state and perform. **awk**:

- generates reports
- matches patterns
- validates data
- filters data for transmission

In the first part of this chapter, we give a general statement of the **awk** syntax. Then, under the heading "Using **awk**," we provide a number of examples that show the syntax rules in use.

Program Structure

An **awk** program is a sequence of statements of the form

```
pattern {action}
pattern {action}
  ...
```

awk runs on a set of input files. The basic operation of **awk** is to scan a set of input lines, in order, one at a time. In each line, **awk** searches for the pattern described in the **awk** program. If that pattern is found in the input line, a corresponding action is performed. In this way, each statement of the **awk** program is executed for a given input line. When all the patterns are tested, the next input line is fetched; and the **awk** program is once again executed from the beginning.

In the **awk** command, either the pattern or the action may be omitted, but not both. If there is no action for a pattern, the matching line is simply printed. If there is no pattern for an action, then the action is performed for every input line. The null **awk** program does nothing. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

For example, this **awk** program

```
/x/     {print}
```

prints every input line that has an **x** in it.

An **awk** program has the following structure:

- a **BEGIN** section
- a **record** or main section
- an **END** section

The **BEGIN** section is run before any input lines are read, and the **END** section is run after all the data files are processed. The **record** section is run over and over for each separate line of input. The words **BEGIN** and **END** are actually special patterns recognized by **awk**.

Values are assigned to variables from the **awk** command line. The **BEGIN** section is run before these assignments are made.

Lexical Units

All **awk** programs are made up of lexical units called tokens. In **awk** there are eight token types:

1. numeric constants
2. string constants
3. keywords
4. identifiers
5. operators
6. record and field tokens
7. comments
8. tokens used for grouping

Numeric Constants

A numeric constant is either a decimal constant or a floating constant. A decimal constant is a nonnull sequence of digits containing at most one decimal point as in **12**, **12.**, **1.2**, and **.12**. A floating constant is a decimal constant followed by **e** or **E** followed by an optional + or − sign followed by a nonnull sequence of digits as in **12e3**, **1.2e3**, **1.2e−3**, and **1.2E+3**. The maximum size and precision of a numeric constant are machine dependent.

String Constants

A string constant is a sequence of zero or more characters surrounded by double quotes as in **","**, **"a"**, **"ab"**, and **"12"**. A double quote is put in a string by preceding it with a backslash, \, as in "He said, \" Sit! \"". A newline is put in a string by using **n** in its place. No other characters need to be escaped. Strings can be (almost) any length.

Keywords

Strings used as keywords are shown in Figure 4-1.

Keywords

BEGIN	**break**	**log**
END	**close**	**next**
FILENAME	**continue**	**number**
FS	**exit**	**print**
NF	**exp**	**printf**
NR	**for**	**split**
OFS	**getline**	**sprintf**
ORS	**if**	**sqrt**
OFMT	**in**	**string**
RS	**index**	**substr**
	int	**while**
	length	

Figure 4-1: **awk** Keywords

Identifiers

Identifiers in **awk** serve to denote variables and arrays. An identifier is a sequence of letters, digits, and underscores, beginning with a letter or an underscore. Uppercase and lowercase letters are different.

Operators

awk has assignment, arithmetic, relational, and logical operators similar to those in the C programming language and regular expression pattern matching operators similar to those in **egrep**(1) and **lex**(1).

Assignment operators are shown in Figure 4-2.

Symbol	Usage	Description
=	assignment	
+=	**plus-equals**	**X += Y is similar to X = X+Y**
-=	minus-equals	**X-=Y is similar to X = X-Y**
*=	times-equals	**X *= Y is similar to X = X*Y**
/=	divide-equals	**X /= Y is similar to X = X/Y**
%=	mod-equals	**X %= Y is similar to X = X%Y**
++	**prefix and postfix increments**	++X and X++ are similar to X=X+1
--	**prefix and postfix decrements**	--X and X-- are similar to X = X -1

Figure 4-2: **awk** Assignment Operators

Arithmetic operators are shown in Figure 4-3.

Symbol	Description
+	unary and binary plus
−	unary and binary minus
*	multiplication
/	division
%	modulus
(...)	grouping

Figure 4-3: **awk** Arithmetic Operators

Relational operators are shown in Figure 4-4.

Symbol	Description
<	less than
< =	less than or equal to
= =	equal to
!=	not equal to
> =	greater than or equal to
>	greater than

Figure 4-4: **awk** Relational Operators

Logical operators are shown in Figure 4-5.

Symbol	Description
&&	and
||	or
!	not

Figure 4-5: **awk** Logical Operators

Regular expression matching operators are shown in the Figure 4-6.

Symbol	Description
~	matches
!~	does not match

Figure 4-6: Operators for Matching Regular Expressions in **awk**

Record and Field Tokens

$0 is a special variable whose value is that of the current input record. **$1**, **$2**, and so forth, are special variables whose values are those of the first field, the second field, and so forth, of the current input record. The keyword **NF** (Number of Fields) is a special variable whose value is the number of fields in the current input record. Thus **$NF** has, as its value, the value of the last field of the current input record. Notice that the first field of each record is numbered 1 and that the number of fields can vary from record to record. None of these variables is defined in the action associated with a **BEGIN** or **END** pattern, where there is no current input record.

The keyword **NR** (Number of Records) is a variable whose value is the number of input records read so far. The first input record read is 1.

Record Separators

The keyword **RS** (Record Separator) is a variable whose value is the current record separator. The value of **RS** is initially set to newline, indicating that adjacent input records are separated by a newline. Keyword **RS** may be changed to any character, *c*, by executing the assignment statement **RS** = "*c*" in an action.

Field Separator

The keyword **FS** (Field Separator) is a variable indicating the current field separator. Initially, the value of **FS** is a blank, indicating that fields are separated by white space, i.e., any nonnull sequence of blanks and tabs. Keyword **FS** is changed to any single character, *c*, by executing the assignment statement **F** = "*c*" in an action or by using the optional command line argument −F*c*. Two values of *c* have special meaning, **space** and **t**. The assignment statement **FS** =" " makes white space (a tab or blank) the field separator; and on the command line, **–F\t** makes a tab the field separator.

If the field separator is not a blank, then there is a field in the record on each side of the separator. For instance, if the field separator is **1**, the record **1XXX1** has three fields. The first and last are null. If the field separator is blank, then fields are separated by white space, and none of the **NF** fields are null.

Multiline Records

The assignment **RS** = " " makes an empty line the record separator and makes a nonnull sequence (consisting of blanks, tabs, and possibly a newline) the field separator. With this setting, none of the first **NF** fields of any record are null.

Output Record and Field Separators

The value of **OFS** (Output Field Separator) is the output field separator. It is put between fields by **print**. The value of **ORS** (Output Record Separators) is put after each record by **print**. Initially, **ORS** is set to a newline and **OFS** to a space. These values may change to any string by assignments such as **ORS** = **"abc"** and **OFS** = **"xyz"**.

Comments

A comment is introduced by a # and terminated by a newline. For example:

```
#    this line is a comment
```

A comment can be appended to the end of any line of an **awk** program.

Tokens Used for Grouping

Tokens in **awk** are usually separated by nonnull sequences of blanks, tabs, and newlines, or by other punctuation symbols such as commas and semicolons. Braces, {...}, surround actions, slashes, /.../, surround regular expression patterns, and double quotes, "...", surround string constants.

Primary Expressions

In **awk**, patterns and actions are made up of expressions. The basic building blocks of expressions are the primary expressions:

numeric constants
string constants
variables
functions

Each expression has both a numeric and a string value, one of which is usually preferred. The rules for determining the preferred value of an expression are explained below.

Numeric Constants

The format of a numeric constant was defined previously in "Lexical Units." Numeric values are stored as floating point numbers. The string value of a numeric constant is computed from the numeric value. The preferred value is the numeric value. Numeric values for string constants are in Figure 4-7.

Numeric Constant	Numeric Value	String Value
0	0	0
1	1	1
.5	0.5	.5
.5e2	50	50

Figure 4-7: Numeric Values for String Constants

String Constants

The format of a string constant was defined previously in "Lexical Units." The numeric value of a string constant is 0 unless the string is a numeric constant enclosed in double quotes. In this case, the numeric value is the number represented. The preferred value of a string constant is its string value. The string value of a string constant is always the string itself. String values for string constants are in Figure 4-8.

String Constant	Numeric Value	String Value
""	0	empty space
"a"	0	a
"XYZ"	0	XYZ
"o"	0	0
"1"	1	1
".5"	0.5	.5
".5e2"	0.5	.5e2

Figure 4-8: String Values for String Constants

Variables

A variable is one of the following:

identifier
identifier [*expression*]
$term

The numeric value of any uninitialized variable is 0, and the string value is the empty string.

An identifier by itself is a simple variable. A variable of the form *identifier* [*expression*] represents an element of an associative array named by *identifier*. The string value of *expression* is used as the index into the array. The preferred value of *identifier* or *identifier* [*expression*] is determined by context.

The variable **$0** refers to the current input record. Its string and numeric values are those of the current input record. If the current input record represents a number, then the numeric value of **$0** is the number and the string value is the literal string. The preferred value of **$0** is string unless the current input record is a number. **$0** cannot be changed by assignment.

The variables **$1**, **$2**, **...** refer to fields 1, 2, and so forth, of the current input record. The string and numeric value of **$i** for **1< =i< =NF** are those of the **i**th field of the current input record. As with **$0**, if the **i**th field represents a number, then the numeric value of **$i** is the number and the string value is the literal string. The preferred value of **$i** is string unless the **i**th field is a number. **$i** may be changed by assignment; the value of **$0** is changed accordingly.

In general, *$term* refers to the input record if *term* has the numeric value 0 and to field **i** if the greatest integer in the numeric value of *term* is **i**. If **i**<0 or if **i**> =100, then accessing **$i** causes **awk** to produce an error diagnostic. If **NF**<**i**< =100, then **$i** behaves like an uninitialized variable. Accessing **$i** for **i** > **NF** does not change the value of **NF**.

Functions

awk has a number of built-in functions that perform common arithmetic and string operations. The arithmetic functions are in Figure 4-9.

Functions
exp (*expression*)
int (*expression*)
log (*expression*)
sqrt (*expression*)

Figure 4-9: Built-in Functions for Arithmetic and String Operations

These functions (**exp**, **int**, **log**, and **sqrt**) compute the exponential, integer part, natural logarithm, and square root, respectively, of the numeric value of *expression*. The (*expression*) may be omitted; then the function is applied to **$0**. The preferred value of an arithmetic function is numeric. String functions are shown in Figure 4-10.

String Functions

getline
index(*expression1*, *expression2*)
length(*expression*)
split(*expression*, *identifier*, *expression2*)
split(*expression*, *identifier*)
sprintf(*format*, *expression1*, *expression2...*)
substr(*expression1*, *expression2*)
substr(*expression1*, *expression2*, *expression3*)

Figure 4-10: **awk** String Functions

The function **getline** causes the next input record to replace the current record. It returns 1 if there is a next input record or a 0 if there is no next input record. The value of **NR** is updated.

The function **index**(*e1,e2*) takes the string value of expressions *e1* and *e2* and returns the first position of where *e2* occurs as a substring in *e1*. If *e2* does not occur in *e1*, index returns 0. For example:

```
index ("abc", "bc")=2
index ("abc", "ac")=0.
```

The function **length** without an argument returns the number of characters in the current input record. With an expression argument, **length**(e) returns the number of characters in the string value of *e*. For example:

```
length ("abc")=3
length (17)=2.
```

The function **split** splits the string value of expression *e* into fields that are then stored in *array*[1], *array*[2], ..., *array*[*n*] using the string value of *sep* as the field separator. Split returns the number of fields found in The function **split** uses the current value of **FS** to indicate the field separator. For example, after invoking

```
n = split ($0, a), a[1],
```

a[2], ..., **a**[*n*] is the same sequence of values as **$1**, **$2 ...**, **$NF**.

The function **sprintf**(*f, e1, e2, ...*) produces the value of expressions *e1, e2, ...* in the format specified by the string value of the expression *f*. The format control conventions are those of the **printf**(3S) statement in the C programming language (except that the use of the asterisk, *, for field width or precision is not allowed).

The function **substr** returns the suffix of *string* starting at position The function **substr** returns the substring of *string* that begins at position *pos* and is *length* characters long. If *pos* + *length* is greater than the length of *string* then **substr** is equivalent to **substr** For example:

```
substr("abc", 2, 1) = "b"
substr("abc", 2, 2) = "bc"
substr("abc", 2, 3) = "bc"
```

Positions less than 1 are taken as 1. A negative or zero length produces a null result. The preferred value of **sprintf** and **substr** is string. The preferred value of the remaining string functions is numeric.

Terms

Various arithmetic operators are applied to primary expressions to produce larger syntactic units called terms. All arithmetic is done in floating point. A term has one of the following forms:

primary expression
term binop term
unop term
incremented variable
(*term*)

Binary Terms

In a term of the form

term1 binop term2

binop can be one of the five binary arithmetic operators +, −, * (multiplication), /(division), % (modulus). The binary operator is applied to the numeric value of the operands *term1* and *term2,* and the result is the usual numeric value. This numeric value is the preferred value, but it can be interpreted as a string value (see **Numeric Constants**). The operators *, /, and % have higher precedence than + and −. All operators are left associative.

Unary Term

In a term of the form

unop term

unop can be unary + or −. The unary operator is applied to the numeric value of *term,* and the result is the usual numeric value which is preferred. However, it can be interpreted as a string value. Unary + and − have higher precedence than *, /, and %.

Incremented Vars

An incremented variable has one of the forms

++ *var*
− − *var*
var ++
var − −

The + + *var* has the value *var* + 1 and has the effect of *var* = *var* + 1. Similarly, − − *var* has the value *var* − 1 and has the effect of *var* = *var* − 1. Therefore, *var* + + has the same value as *var* and has the effect of *var* = *var* + 1. Similarly, *var* − − has the same value as *var* and has the effect of *var* = *var* − 1. The preferred value of an incremented variable is numeric.

Parenthesized Terms

Parentheses are used to group terms in the usual manner.

Expressions

An **awk** expression is one of the following:

term
term term ...
var asgnop expression

Concatenation of Terms

In an expression of the form *term1 term2 ...,* the string value of the terms are concatenated. The preferred value of the resulting expression is a string value. Concatenation of terms has lower precedence than binary + and -. For example,

```
1+2 3+4
```

has the string (and numeric) value 37.

Assignment Expressions

An assignment expression is one of the forms

var asgnop expression

where *asgnop* is one of the six assignment operators:

```
=
+=
-=
*=
/=
%=
```

The preferred value of *var* is the same as that of *expression.*

In an expression of the form

var = expression

the numeric and string values of *var* become those of *expression.*

var op = expression

is equivalent to

var = var op expression

where *op* is one of: +, −, *, /, %. The *asgnops* are right associative and have the lowest precedence of any operator. Thus, a += b *= c−2 is equivalent to the sequence of assignments

```
b = b * (c-2)
a = a + b
```

Using awk

The remainder of this chapter undertakes to show the syntax rules of awk in action. The material is organized under the following topics:

- input and output
- patterns
- actions
- special features

Input and Output

Presenting Your Program for Processing

There are two ways to present your program of pattern/action statements to **awk** for processing:

1. If the program is short (a line or two), it is often easiest to make the program the first argument on the command line:

   awk ' *program* ' [*filename...*]

   where *program* is your **awk** program, and *filename...* is an optional input file(s). Note that there are single quotes around the program name in order for the shell to accept the entire string (program) as the first argument to **awk**. For example, write to the shell

   awk ' /x/ {print} ' file1

   to run the **awk** program /x/ {print} on the input file **file1**. If no input file is specified, **awk** expects input from the standard input, **stdin**. You can also specify that input comes from **stdin** by using the hyphen, -, as one of the files. The pattern-action statement

   awk ' *program* ' **file1 -**

   looks for input from **file1** and from **stdin**. It processes first from **file1** and then from **stdin**.

2. Alternately, if your **awk** program is long or is one you want to preserve for re-use in the future, it is convenient to put the program in a separate file, **awkprog**, for example, and tell **awk** to fetch it from there. This is done by using the **–f** option on the command line, as follows:

   awk –f awkprog *filename...* where *filename...* is an optional list of input files that may include **stdin** as is shown above.

These alternative ways of presenting your **awk** program for processing are illustrated by the following:

```
awk ' BEGIN {print "hello, world" exit} '
```

prints

```
hello, world
```

on the standard output when given to the shell.

This **awk** program could be run by putting

```
BEGIN {print "hello, world" exit}
```

in a file named **awkprog**, and then the command

```
awk -f awkprog
```

given to the shell would have the same effect as the first procedure.

Input: Records and Fields

awk reads its input one record at a time. Unless changed by you, a record is a sequence of characters from the input ending with a newline character or with an end of file. **awk** reads in characters until it encounters a newline or end of file. The string of characters, thus read, is assigned to the variable **$0**.

Once **awk** has read in a record, it then views the record as being made up of fields. Unless changed by you, a field is a string of characters separated by blanks or tabs.

Sample Input File, countries

For use as an example, we have created the file, **countries**. **countries** contains the area in thousands of square miles, the population in millions, and the continent for the ten largest countries in the world. (Figures are from 1978; Russia is placed in Asia.)

```
Russia       8650    262    Asia
Canada       3852    24     North America
China        3692    866    Asia
USA          3615    219    North America
Brazil       3286    116    South America
Australia    2968    14     Australia
India        1269    637    Asia
Argentina    1072    26     South America
Sudan        968     19     Africa
Algeria      920     18     Africa
```

Figure 4-11: Sample Input File, **countries**

The wide spaces are tabs in the original input and a single blank separates North and South from America. We use this data as the input for many of the **awk** programs in this chapter since it is typical of the type of material that **awk** is best at processing (a mixture of words and numbers arranged in fields or columns separated by blanks and tabs).

Each of these lines has either four or five fields if blanks and/or tabs separate the fields. This is what **awk** assumes unless told otherwise. In the above example, the first record is

Russia 8650 262 Asia

When this record is read by **awk**, it is assigned to the variable **$0**. If you want to refer to this entire record, it is done through the variable, **$0**. For example, the following action:

```
{print $0}
```

prints the entire record.

Fields within a record are assigned to the variables **$1**, **$2**, **$3**, and so forth; that is, the first field of the present record is referred to as **$1** by the **awk** program. The second field of the present record is referred to as **$2** by the **awk** program. The **i**th field of the present record is referred to as **$i** by the **awk** program. Thus, in the above example of the file **countries**, in the first record:

$1 is equal to the string "Russia"
$2 is equal to the integer 8650
$3 is equal to the integer 262
$4 is equal to the string "Asia"
$5 is equal to the null string

... and so forth.

To print the continent, followed by the name of the country, followed by its population, use the following command:

```
awk '{print $4, $1, $3}' countries
```

You'll notice that this does not produce exactly the output you may have wanted because the field separator defaults to white space (tabs or blanks). **North America** and **South America** inconveniently contain a blank. Try it again with the following command line:

```
awk -F\t '{print $4, $1, $3}' countries
```

Input: From the Command Line

We have seen above, under "Presenting Your Program for Processing," that you can give your program to **awk** for processing by either including it on the command line enclosed by single quotes, or by putting it in a file and naming the file on the command line (preceded by the **-f** flag). It is also possible to set variables from the command line.

In **awk**, values may be assigned to variables from within an **awk** program. Because you do not declare types of variables, a variable is created simply by referring to it. An example of assigning a value to a variable is:

```
x=5
```

This statement in an **awk** program assigns the value **5** to the variable **x**. This type of assignment can be done from the command line. This provides another way to supply input values to **awk** programs. For example:

```
awk ' {print x }' x=5 -
```

will print the value **5** on the standard output. The minus sign at the end of this command is necessary to indicate that input is coming from **stdin** instead of a file called **x=5**. After entering the command, the user must proceed to enter input. The input is terminated with a CTRL-d.

If the input comes from a file, named **file1** in the example, the command is

```
awk '{print x}' file1
```

It is not possible to assign values to variables used in the **BEGIN** section in this way.

If it is necessary to change the record separator and the field separator, it is useful to do so from the command line as in the following example:

```
awk -f awkprog RS=":" file1
```

Here, the record separator is changed to the character :. This causes your program in the file **awkprog** to run with records separated by the colon instead of the newline character and with input coming from **file1**. It is similarly useful to change the field separator from the command line.

There is a separate option, −F*x*, that is placed directly after the command **awk**. This changes the field separator from white space to the character *x*. For example:

```
awk -F: -f awkprog file1
```

changes the field separator, **FS**, to the character :. Note that if the field separator is specifically set to a tab (that is, with the −**F** option or by making a direct assignment to **FS**), then blanks are not recognized by **awk** as separating fields. However, the reverse is not true. Even if the field separator is specifically set to a blank, tabs are still recognized by **awk** as separating fields.

Output: Printing

An action may have no pattern; in this case, the action is executed for all lines as in the simple printing program

```
{print}
```

This is one of the simplest actions performed by **awk**. It prints each line of the input to the output. More useful is to print one or more fields from each line. For instance, using the file **countries** that was used earlier,

awk '{ print $1, $3 }' countries

prints the name of the country and the population:

```
Russia 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 14
Sudan 19
Algeria 18
```

A semicolon at the end of statements is optional. **awk** accepts

```
{print $1}
```

and

```
{print $1;}
```

equally and takes them to mean the same thing. If you want to put two **awk** statements on the same line of an **awk** script, the semicolon is necessary, for example, if you want the number **5** printed:

```
{x=5; print x}
```

Parentheses are also optional with the print statement.

```
{print $3, $2}
```

is the same as

```
{print ($3, $2)}
```

Items separated by a comma in a **print** statement are separated by the current output field separator (normally spaces, even though the input is separated by tabs) when printed. The **OFS** is another special variable that can be changed by you. (These special variables are summarized below.) **print** also prints strings directly from your programs, as with the **awk** script

```
{print "hello, world"}
```

As we have already seen, **awk** makes available a number of special variables with useful values, for example, **FS** and **RS**. We introduce two other special variables in the next example. **NR** and **NF** are both integers that contain the number of the present record and the number of fields in the present record, respectively. Thus,

```
{print NR, NF, $0}
```

prints each record number and the number of fields in each record followed by the record itself. Using this program on the file **countries** yields:

```
1 4 Russia     8650   262   Asia
2 5 Canada     3852   24    North America
3 4 China      3692   866   Asia
4 5 USA        3615   219   North America
5 5 Brazil     3286   116   South America
6 4 Australia  2968   14    Australia
7 4 India      1269   637   Asia
8 5 Argentina  1072   26    South America
9 4 Sudan      968    19    Africa
10 4 Algeria   920    18    Africa
```

and the program

```
{print NR, $1}
```

prints

```
1 Russia
2 Canada
3 China
4 USA
5 Brazil
6 Australia
7 India
8 Argentina
9 Sudan
10 Algeria
```

This is an easy way to supply sequence numbers to a list. **print**, by itself, prints the input record. Use

```
{print ""}
```

to print an empty line.

awk also provides the statement **printf** so that you can format output as desired. **print** uses the default format **%.6g** for each numeric variable printed.

printf "*format*", *expr*, *expr*, ...

formats the expressions in the list according to the specification in the string *format*, and prints them. The *format* statement is almost identical to that of **printf**(3S) in the C library. For example:

```
{ printf "%10s %6d %6d\n", $1, $2, $3 }
```

prints **$1** as a string of 10 characters (right justified). The second and third fields (6-digit numbers) make a neatly columned table.

```
   Russia   8650   262
   Canada   3852   244
    China   3692   866
      USA   3615   219
   Brazil   3286   116
Australia   2968   14
    India   1269   637
Argentina   1072   26
    Sudan   968    19
  Algeria   920    18
```

With **printf**, no output separators or newlines are produced automatically. You must add them as in this example. The escape characters **\n**, **\t**, **\b** (backspace), and **\r** (carriage return) may be specified.

There is a third way that printing can occur on standard output when a pattern without an action is specified. In this case, the entire record, **$0**, is printed. For example, the program

```
/x/
```

prints any record that contains the character **x**.

There are two special variables that go with printing, **OFS** and **ORS**. By default, these are set to blank and the newline character, respectively. The variable **OFS** is printed on the standard output when a comma occurs in a **print** statement such as

```
{ x="hello"; y="world"
print x,y
}
```

which prints

```
hello world
```

However, without the comma in the print statement as

```
{ x="hello"; y="world"
print x y
}
```

you get

```
helloworld
```

To get a comma on the output, you can either insert it in the print statement as in this case

```
{ x="hello"; y="world"
print x"," y
}
```

or you can change **OFS** in a **BEGIN** section as in

```
BEGIN {OFS=", "}
{ x="hello"; y="world"
print x, y
}
```

Both of these last two scripts yield

```
hello, world
```

Note that the output field separator is not used when **$0** is printed.

Output: to Different Files

The UNIX operating system shell allows you to redirect standard output to a file. **awk** also lets you direct output to many different files from within your **awk** program. For example, with our input file **countries**, we want to print all the data from countries of Asia in a file called **ASIA**, all the data from countries in Africa in a file called **AFRICA**, and so forth. This is done with the following **awk** program:

```
{ if ($4 == "Asia") print > "ASIA"
  if ($4 == "Europe") print > "EUROPE"
  if ($4 == "North") print > "NORTH_AMERICA"
  if ($4 == "South") print > "SOUTH_AMERICA"
  if ($4 == "Australia") print > "AUSTRALIA"
  if ($4 == "Africa") print > "AFRICA"
}
```

Flow of control statements is discussed later.

In general, you may direct output into a file after a **print** or a **printf** statement by using a statement of the form

print > "*filename*"

where *filename* is the name of the file receiving the data. The **print** statement may have any legal arguments to it.

Notice that the filename is quoted. Without quotes, filenames are treated as uninitialized variables and all output then goes to **stdout**, unless redirected on the command line.

If > is replaced by > >, output is appended to the file rather than overwriting it. Notice that there is an upper limit to the number of files that are written in this way. At present it is ten.

Output: to Pipes

It is also possible to direct printing into a pipe instead of a file. For example:

```
{
if ($2 == "XX") print | "mailx mary"
}
```

where **mary** is a person's login name. Any record with the second field equal to **XX** is sent to the user, **mary**, as mail. **awk** waits until the entire program is run before it executes the command that was piped to; in this case, the **mailx**(1) command. For example:

```
{
print $1 | "sort"
}
```

takes the first field of each input record, sorts these fields, and then prints them.

Another example of using a pipe for output is the following idiom, which guarantees that its output always goes to your terminal:

```
{
print ... | "cat -v > /dev/tty"
}
```

Only one output statement to a pipe is permitted in an **awk** program. In all output statements involving redirection of output, the files or pipes are identified by their names, but they are created and opened only once in the entire run.

Patterns

A pattern in front of an action acts as a selector that determines if the action is to be executed. A variety of expressions are used as patterns:

- certain keywords
- arithmetic relational expressions
- regular expressions
- combinations of these

BEGIN and END

The keyword, **BEGIN**, is a special pattern that matches the beginning of the input before the first record is read. The keyword, **END**, is a special pattern that matches the end of the input after the last line is processed. **BEGIN** and **END** thus provide a way to gain control before and after processing for initialization and wrapping up.

As you have seen, you can use **BEGIN** to put column headings on the output

```
BEGIN {print "Country", "Area", "Population", "Continent"}
        {print}
```

which produces

```
Country Area Population Continent

Russia   8650   262    Asia
Canada   3852   24     North America
China    3692   866    Asia
USA      3615   219    North America
Brazil   3286   116    South America
Australia2968   14     Australia
India    1269   637    Asia
Argentina1072   26     South America
Sudan    968    19     Africa
Algeria  920    18     Africa
```

Formatting is not very good here; **printf** would do a better job and is generally used when appearance is important.

Recall also, that the **BEGIN** section is a good place to change special variables such as **FS** or **RS**. For example:

```
BEGIN { FS= "\t"
        printf "Country\t\t
           Area\tPopulation\tContinent\n\n"}
        {printf "%-10s\t%6d\t%6d\t\t% -14s\n",
           $1, $2, $3, $4}
END     {print "The number of records is", NR}
```

In this program, **FS** is set to a tab in the **BEGIN** section and as a result all records in the file **countries** have exactly four fields. Note that if **BEGIN** is present it is the first pattern; **END** is the last if it is used.

Relational Expressions

An **awk** pattern is any expression involving comparisons between strings of characters or numbers. For example, if you want to print only countries with more than 100 million population, use

```
$3 > 100
```

This tiny **awk** program is a pattern without an action so it prints each line whose third field is greater than 100 as follows:

```
Russia    8650    262    Asia
China     3692    866    Asia
USA       3615    219    North America
Brazil    3286    116    South America
India     1269    637    Asia
```

To print the names of the countries that are in Asia, type

```
$4 == "Asia" {print $1}
```

which produces

```
Russia
China
India
```

The conditions tested are <, <=, ==, !=, >=, and >. In such relational tests if both operands are numeric, a numerical comparison is made. Otherwise, the operands are compared as strings. Thus,

```
$1 >= "S"
```

selects lines that begin with **S**, **T**, **U**, and greater, which in this case are

```
USA     3615    219     North America
Sudan   968     19      Africa
```

In the absence of other information, fields are treated as strings, so the program

```
$1 == $4
```

compares the first and fourth fields as strings of characters and prints the single line

```
Australia       2968       14 Australia
```

Regular Expressions

awk provides more powerful capabilities for searching for strings of characters than were illustrated in the previous section. These are regular expressions. The simplest regular expression is a literal string of characters enclosed in slashes.

```
/Asia/
```

This is a complete **awk** program that prints all lines that contain any occurrence of the name **Asia**. If a line contains **Asia** as part of a larger word like **Asiatic**, it is also printed (but there are no such words in the **countries** file.)

awk regular expressions include regular expression forms found in the text editor, **ed**(1), and the pattern finder, **grep**(1), in which certain characters have special meanings.

For example, we could print all lines that begin with **A** with

```
/^A/
```

or all lines that begin with **A**, **B**, or **C** with

```
/^[ABC]/
```

or all lines that end with **ia** with

```
/ia$/
```

In general, the circumflex, ^, indicates the beginning of a line. The dollar sign, **$**, indicates the end of the line and characters enclosed in brackets, **[ ]**, match any one of the characters enclosed. In addition, **awk** allows parentheses for grouping, the pipe, ¦, for alternatives, **+** for one or more occurrences, and **?** for zero or one occurrences. For example:

```
/x¦y/ {print}
```

prints all records that contain either an **x** or a **y**.

```
/ax+b/        {print}
```

prints all records that contain an **a** followed by one or more **x**'s followed by a **b**. For example, axb, Paxxxxxxxb, QaxxbR.

```
/ax?b/        {print}
```

prints all records that contain an **a** followed by zero or one **x** followed by a **b**. For example: ab, axb, yaxbPPP, CabD.

The two characters, **.** and *, have the same meaning as they have in **ed**(1) namely, **.** can stand for any character and * means zero or more occurrences of the character preceding it. For example:

/a.b/

matches any record that contains an **a** followed by any character followed by a **b**. That is, the record must contain an **a** and a **b** separated by exactly one character. For example, /**a.b**/ matches axb, aPb and xxxxaXbxx, but not ab, axxb.

```
/ax*c/
```

matches a record that contains an **a** followed by zero or more **x**'s followed by a **c**. For example, it matches

```
ac
axc
pqraxxxxxxxxxxc901
```

Just as in **ed**(1), it is possible to turn off the special meaning of metacharacters such as ^ and * by preceding these characters with a backslash. An example of this is the pattern

```
/\/*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) by using the operators ~ or !~. For example, with the input file **countries** as before, the program

```
$1 ~ /ia$/        {print $1}
```

prints all countries whose name ends in **ia**:

```
Russia
Australia
India
Algeria
```

which is indeed different from lines that end in **ia**.

Combinations of Patterns

A pattern can be made up of similar patterns combined with the operators || (OR), && (AND), ! (NOT), and parentheses. For example:

```
$2 >= 3000 && $3 >= 100
```

selects lines where both area and population are large. For example:

```
Russia   8650   262   Asia
China    3692   866   Asia
USA      3615   219   North America
Brazil   3286   116   South America
```

while

```
$4 == "Asia" || $4 == "Africa"
```

selects lines with **Asia** or **Africa** as the fourth field. An alternate way to write this last expression is with a regular expression:

```
$4 ~ /^ Asia|Africa)$/
```

which says to select records where the 4th field matches **Africa** or begins with **Asia**.

&& and || guarantee that their operands are evaluated from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

The pattern that selects an action may also consist of two patterns separated by a comma as in

pattern1, *pattern2*   { *action* }

In this case, the *action* is performed for each line between an occurrence of *pattern1* and the next occurrence of *pattern2* (inclusive). As an example with no action

```
/Canada/,/Brazil/
```

prints all lines between the one containing **Canada** and the line containing **Brazil**. For example:

```
Canada    3852    24     North America
China     3692    866    Asia
USA       3615    219    North America
Brazil    3286    116    South America
```

while

```
NR == 2, NR == 5 { ... }
```

does the action for lines 2 through 5 of the input. Different types of patterns may be mixed as in

```
/Canada/, $4 == "Africa"
```

which prints all lines from the first line containing **Canada** up to and including the next record whose fourth field is **Africa**.

NOTE: The foregoing discussion of pattern matching pertains to the pattern portion of the pattern/action **awk** statement. Pattern matching can also take place inside an **if** or **while** statement in the action portion. See the section "Flow of Control."

Actions

An **awk** action is a sequence of action statements separated by newlines or semicolons. These action statements do a variety of bookkeeping and string manipulating tasks.

Variables, Expressions, and Assignments

awk provides the ability to do arithmetic and to store the results in variables for later use in the program. As an example, consider printing the population density for each country in the file **countries**.

```
{print $1, (1000000 * $3) / ($2 * 1000) }
```

(Recall that in this file the population is in millions and the area in thousands.) The result is population density in people per square mile.

```
Russia 30.289
Canada 6.23053
China 234.561
USA 60.5809
Brazil 35.3013
Australia 4.71698
India 501.97
Argentina 24.2537
Sudan 19.6281
Algeria 19.5652
```

The formatting is not good; using **printf** instead gives the program

```
{printf "%10s %6.1f\n", $1, (1000000 * $3) / ($2 * 1000)}
```

and the output

```
Russia      30.3
Canada       6.2
China      234.6
USA         60.6
Brazil      35.3
Australia    4.7
India      502.0
Argentina   24.3
Sudan       19.6
Algeria     19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, and % (modulus).

To compute the total population and number of countries from Asia, we could write

```
/Asia/   { pop += $3; ++n }
END      {print "total population of", n, "Asian countries is",
            pop }
```

which produces

```
        total population of 3 Asian countries is 1765.
```

The operators, ++, −−, −=, /=, *=, +=, and %= are available in **awk** as they are in C. The same is true of the ++ operator; it adds one to the value of a variable. The increment operators ++ and −− (as in C) are used as prefix or as postfix operators. These operators are also used in expressions.

Initialization of Variables

In the previous example, we did not initialize **pop** nor **n**; yet everything worked properly. This is because (by default) variables are initialized to the null string, which has a numerical value of 0. This eliminates the need for most initialization of variables in BEGIN sections. We can use default initialization to advantage in this program, which finds the country with the largest population.

```
maxpop < $3 {
        maxpop = $3
        country = $1
        }
END     {print country, maxpop}
```

which produces

```
China 866
```

Field Variables

Fields in **awk** share essentially all of the properties of variables. They are used in arithmetic and string operations, may be initialized to the null string, or have other values assigned to them. Thus, divide the second field by 1000 to convert the area to millions of square miles by

```
{ $2 /= 1000; print }
```

or process two fields into a third with

```
BEGIN   { FS = "\t" }
        { $4 = 1000 * $3 / $2; print }
```

or assign strings to a field as in

```
/USA/    { $1 = "United States" ; print }
```

which replaces **USA** by **United States** and prints the affected line:

```
United States 3615 219 North America
```

Fields are accessed by expressions; thus, **$NF** is the last field and **$(NF – 1)** is the second to the last. Note that the parentheses are needed since **$NF – 1** is 1 less than the value in the last field.

String Concatenation

Strings are concatenated by writing them one after the other as in the following example:

```
{ x = "hello"
  x = x ", world"
  print x
}
```

which prints the usual

```
hello, world
```

With input from the file **countries**, the following program:

```
/A/         { s = s " " $1 }
END         { print s }
```

prints

```
Australia Argentina Algeria
```

Variables, string expressions, and numeric expressions may appear in concatenations; the numeric expressions are treated as strings in this case.

Special Variables

Some variables in **awk** have special meanings. These are detailed here and the complete list given.

NR	Number of the current record.
NF	Number of fields in the current record.
FS	Input field separator, by default it is set to a blank or tab.
RS	Input record separator, by default it is set to the newline character.
$i	The ith input field of the current record.

$0	The entire current input record.
OFS	Output field separator, by default it is set to a blank.
ORS	Output record separator, by default it is set to the newline character.
OFMT	The format for printing numbers, with the print statement, by default is **%.6g**
FILENAME	The name of the input file currently being read. This is useful because **awk** commands are typically of the form **awk -f** *program* **file1 file2 file3 ...**

Type

Variables (and fields) take on numeric or string values according to context. For example, in

```
pop += $3
```

pop is presumably a number, while in

```
country = $1
```

country is a string. In

```
maxpop < $3
```

the type of **maxpop** depends on the data found in **$3**. It is determined when the program is run.

In general, each variable and field is potentially a string or a number, or both at any time. When a variable is set by the assignment

```
v = expr
```

its type is set to that of *expr*. (Assignment also includes + =, + +, − =, and so forth.) An arithmetic expression is of the type **number**; a concatenation of strings is of type **string**. If the assignment is a simple copy as in

```
        v1 = v2
```

then the type of **v1** becomes that of **v2**.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to strings if necessary and the comparison is made on strings.

The type of any expression may be coerced to numeric by a subterfuge such as

```
        expr + 0
```

and to string by

```
        expr ""
```

This last expression is **string** concatenated with the null string.

Arrays

As well as ordinary variables, **awk** provides 1-dimensional arrays. Array elements are not declared; they spring into existence by being mentioned. Subscripts may have any non-null value including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
        x[NR] = $0
```

assigns the current input line to the **NR**th element of the array **x**. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the following **awk** program:

```
                { x[NR] = $0 }
        END     { ... program ... }
```

The first line of this program records each input line into the array **x**. In particular, the following program

```
        { x[NR] = $1}
```

(when run on the file **countries**) produces an array of elements with

```
x[1] = "Russia"
x[2] = "Canada"
x[3] = "China"
```

... and so forth.

Arrays are also indexed by non-numeric values that give **awk** a capability rather like the associative memory of Snobol tables. For example, we can write

```
/Asia/{pop["Asia"] += $3}
/Africa/{pop[Africa] += $3}
END     {print "Asia=" pop["Asia"],
              "Africa="pop["Africa"] }
```

which produces

```
Asia=1765 Africa=37
```

Notice the concatenation. Also, any expression can be used as a subscript in an array reference. Thus,

```
area[$1] = $2
```

uses the first field of a line (as a string) to index the array **area.**

Special Features

In this final section we describe the use of some special **awk** features.

Built-In Functions

The function **length** is provided by **awk** to compute the length of a string of characters. The following program prints each record preceded by its length:

```
{print length, $0 }
```

In this case the variable **length** means **length($0)**, the length of the present record. In general, **length**(*x*) will return the length of *x* as a string.

With input from the file **countries**, the following **awk** program will print the longest country name:

```
length($1) > max  {max = length($1); name = $1 }
END              {print name}
```

The function **split**

```
split(s, array)
```

assigns the fields of the string **s** to successive elements of the array, **array**.

For example;

```
split("Now is the time", w)
```

assigns the value **Now** to w[1], **is** to w[2], **the** to w[3], and **time** to w[4]. All other elements of the array w[ ], if any, are set to the null string. It is possible to have a character other than a blank as the separator for the elements of w. For this, use **split** with three elements.

```
n = split(s, array, sep)
```

This splits the string **s** into **array**[1], ..., **array**[n]. The number of elements found is returned as the value of **split**. If the *sep* argument is present, its first character is used as the field separator; otherwise, **FS** is used. This is useful if in the middle of an **awk** script, it is necessary to change the record separator for one record. Also provided by **awk** are the math functions

sqrt
log
exp
int

They provide the square root function, the base **e** logarithm function, exponential and integral part functions. This last function returns the greatest integer less than or equal to its argument. These functions are the same as those of the C math library (**int** corresponds to the **libm floor** function) and so they have the same return on error as those in **libm**. (See the *Programmer's Reference Manual.*)

The function **substr**

```
substr(s,m,n)
```

produces the substring of **s** that begins at position **m** and is at most *n* characters long. If the third argument (**n** in this case) is omitted, the substring goes to the end of **s**. For example, we could abbreviate the country names in the file **countries** by

```
{ $1 = substr($1, 1, 3); print }
```

which produces

```
Rus    8650   262   Asia
Can    3852    24   North America
Chi    3692   866   Asia
USA    3615   219   North America
Bra    3286   116   South America
Aus    2968    14   Australia
Ind    1269   637   Asia
Arg    1072    26   South America
Sud     968    19   Africa
Alg     920    18   Africa
```

If **s** is a number, **substr** uses its printed image:

substr(123456789,3,4) = 3456.

The function **index**

```
index (s1,s2)
```

returns the leftmost position where the string **s2** occurs in **s1** or zero if **s2** does not occur in **s1**.

The function **sprintf** formats expressions as the **printf** statement does but assigns the resulting expression to a variable instead of sending the results to **stdout**. For example:

```
x = sprintf("%10s %6d", $1, $2)
```

sets **x** to the string produced by formatting the values of **$1** and **$2**. The **x** may then be used in subsequent computations.

The function **getline** immediately reads the next input record. Fields **NR** and **$0** are set but control is left at exactly the same spot in the **awk** program. **getline** returns 0 for the end of file and a 1 for a normal record.

Flow of Control

awk provides the basic flow of control statements within actions

- **if-else**
- **while**
- **for**

with statement grouping as in C language.

The **if** statement is used as follows:

if (*condition*) *statement1* **else** *statement2*

The *condition* is evaluated; and if it is true, *statement1* is executed; otherwise, *statement2* is executed. The **else** part is optional. Several statements enclosed in braces, { }, are treated as a single statement. Rewriting the maximum population computation from the pattern section with an **if** statement results in

```
{       if (maxpop < $3) {
                maxpop = $3
                country = $1
        }
}
END     { print country, maxpop }
```

There is also a **while** statement in **awk**.

while (*condition*) *statement*

The *condition* is evaluated; if it is true, the *statement* is executed. The *condition* is evaluated again, and if true, the *statement* is executed. The cycle repeats as long as the condition is true. For example, the following prints all input fields, one per line:

```
{       i = 1
        while (i <= NF) {
                print $i
                ++i
    }
}
```

Another example is the Euclidean algorithm for finding the greatest common divisor of **$1** and **$2**:

```
{printf "the greatest common divisor of " $1 "and ",
   $2, "is"
while ($1 != $2) {
   if ($1 > $2) $1 -= $2
   else     $2 -= $1
}
printf $1 "\n"
}
```

The **for** statement is like that of C, which is:

for (*expression1* ; *condition* ; *expression2*) *statement*

So

```
        {       for (i = 1 ; i <= NF; i++)
                        print $i
        }
```

is another **awk** program that prints all input fields, one per line.

There is an alternate form of the **for** statement that is useful for accessing the elements of an associative array in **awk**.

for (*i in array*) *statement*

executes *statement* with the variable **i** set in turn to each subscript of *array*. The subscripts are each accessed once but in undefined order. Chaos will ensue if the variable **i** is altered or if any new elements are created within the loop. For example, you could use the **for** statement to print the record number followed by the record of all input records after the main program is executed.

```
                { x[NR] = $0 }
        END     { for(i in x) print i, x[i] }
```

A more practical example is the following use of strings to index arrays to add the populations of countries by continents:

```
        BEGIN   {FS="\t"}
                {population[$4] += $3}
        END     {for(i in population)
                        print i, population[i]
            }
```

In this program, the body of the **for** loop is executed for **i** equal to the string **Asia**, then for **i** equal to the string **North America**, and so forth until all the possible values of **i** are exhausted; that is, until all the strings of names of countries are used. Note, however, the order the loops are executed is not specified. If the loop associated with **Canada** is executed before the loop associated with the string **Russia**, such a program produces

```
        South America 26
        Africa 16
        Asia 637
        Australia 14
        North America 219
```

Note that the expression in the condition part of an **if, while**, or, **for** statement can include

- relational operators like <, < =, >, > =, = =, and ! =
- regular expressions that are used with the matching operators ~ and !~
- the logical operators ||, &&, and !
- parentheses for grouping

The **break** statement (when it occurs within a **while** or **for** loop) causes an immediate exit from the **while** or **for** loop.

The **continue** statement (when it occurs within a **while** or **for** loop) causes the next iteration of the loop to begin.

The **next** statement in an **awk** program causes **awk** to skip immediately to the next record and begin scanning patterns from the top of the program. (Note the difference between **getline** and **next**. **getline** does not skip to the top of the **awk** program.)

If an **exit** statement occurs in the **BEGIN** section of an **awk** program, the program stops executing and the **END** section is not executed (if there is one).

An **exit** that occurs in the main body of the **awk** program causes execution of the main body of the **awk** program to stop. No more records are read, and the **END** section is executed.

An **exit** in the **END** section causes execution to terminate at that point.

Report Generation

The flow of control statements in the last section are especially useful when **awk** is used as a report generator. **awk** is useful for tabulating, summarizing, and formatting information. We have seen an example of **awk** tabulating populations in the last section. Here is another example of this. Suppose you have a file **prog.usage** that contains lines of three fields: **name**, **program**, and **usage**:

```
Smith    draw    3
Brown    eqn     1
Jones    nroff   4
Smith    nroff   1
Jones    spell   5
Brown    spell   9
Smith    draw    6
```

The first line indicates that Smith used the **draw** program three times. If you want to create a program that has the total usage of each program along with the names in alphabetical order and the total usage, use the following program, called **list1**:

```
            {use[$1 "" $2] += $3}
    END     {for (np in use)
                    print np "   " use[np]  | "sort +0 +2nr"
          }
```

This program produces the following output when used on the input file, **prog.usage**.

```
Brown    eqn     1
Brown    spell   9
Jones    nroff   4
Jones    spell   5
Smith    draw    9
Smith    nroff   1
```

If you would like to format the previous output so that each name is printed only once, pipe the output of the previous **awk** program into the following program, called **format1**:

```
    {       if ($1 != prev) {
                    print $1 ":"
                    prev = $1
            }
            print "  " $2 "  " $3
    }
```

The variable **prev** is used to ensure each unique value of **$1** prints only once. The command

```
awk -f list1 prog.usage ¦ awk -f format1
```

gives the output

```
Brown:
          eqn     1
          spell   9
Jones:
          nroff   4
          spell   5
Smith:
          draw    9
          nroff   1
```

It is often useful to combine different **awk** scripts and other shell commands such as **sort**(1), as was done in the **list1** script.

Cooperation with the Shell

Normally, an **awk** program is either contained in a file or enclosed within single quotes as in

```
awk '{print $1}' ...
```

Since **awk** uses many of the same characters the shell does (such as **$** and the double quote) surrounding the program by single quotes ensures that the shell passes the program to **awk** intact.

Consider writing an **awk** program to print the *n*th field, where *n* is a parameter determined when the program is run. That is, we want a program called **field** such that

```
field n
```

runs the **awk** program

```
awk '{print $n}'
```

How does the value of *n* get into the **awk** program?

There are several ways to do this. One is to define **field** as follows:

```
awk '{print $'$1'}'
```

Spaces are critical here: as written there is only one argument, even though there are two sets of quotes. The **$1** is outside the quotes, visible to the shell, and therefore substituted properly when **field** is invoked.

Another way to do this job relies on the fact that the shell substitutes for **$** parameters within double quotes.

```
awk "{print \$ $1}"
```

Here the trick is to protect the first **$** with a \; the **$1** is again replaced by the number when **field** is invoked.

Multidimensional Arrays

You can simulate the effect of multidimensional arrays by creating your own subscripts. For example:

```
for (i = 1; i <= 10; i++)
        for (j = 1; j <= 10; j++)
                mult[i "," j] = . . .
```

creates an array whose subscripts have the form **i,j**; that is, 1,1; 1,2 and so forth; and thus simulate a 2-dimensional array.

Chapter 5: lex

An Overview of lex Programming

lex is a software tool that lets you solve a wide class of problems drawn from text processing, code enciphering, compiler writing, and other areas. In text processing, you may check the spelling of words for errors; in code enciphering, you may translate certain patterns of characters into others; and in compiler writing, you may determine what the tokens (smallest meaningful sequences of characters) are in the program to be compiled. The problem common to all of these tasks is recognizing different strings of characters that satisfy certain characteristics. In the compiler writing case, creating the ability to solve the problem requires implementing the compiler's lexical analyzer. Hence the name **lex**.

It is not essential to use **lex** to handle problems of this kind. You could write programs in a standard language like C to handle them, too. In fact, what **lex** does is produce such C programs. (**lex** is therefore called a program generator.) What **lex** offers you, once you acquire a facility with it, is typically a faster, easier way to create programs that perform these tasks. Its weakness is that it often produces C programs that are longer than necessary for the task at hand and that execute more slowly than they otherwise might. In many applications this is a minor consideration, and the advantages of using **lex** considerably outweigh it.

To understand what **lex** does, see the diagram in Figure 5-1. We begin with the **lex** source (often called the **lex** specification) that you, the programmer, write to solve the problem at hand. This **lex** source consists of a list of rules specifying sequences of characters (expressions) to be searched for in an input text, and the actions to take when an expression is found. The source is read by the **lex** program generator. The output of the program generator is a C program that, in turn, must be compiled by a host language C compiler to generate the executable object program that does the lexical analysis. Note that this procedure is not typically automatic – user intervention is required. Finally, the lexical analyzer program produced by this process takes as input any source file and produces the desired output, such as altered text or a list of tokens.

lex can also be used to collect statistical data on features of the input, such as character count, word length, number of occurrences of a word, and so forth. In later sections of this chapter, we will see

- how to write **lex** source to do some of these tasks
- how to translate **lex** source
- how to compile, link, and execute the lexical analyzer in C
- how to run the lexical analyzer program

We will then be on our way to appreciating the power that **lex** provides.

Figure 5-1: Creation and Use of a Lexical Analyzer with **lex**

Writing lex Programs

A **lex** specification consists of at most three sections: definitions, rules, and user subroutines. The rules section is mandatory. Sections for definitions and user subroutines are optional, but if present, must appear in the indicated order.

The Fundamentals of lex Rules

The mandatory rules section opens with the delimiter %%. If a subroutines section follows, another %% delimiter ends the rules section. If there is no second delimiter, the rules section is presumed to continue to the end of the program.

Each rule consists of a specification of the pattern sought and the action(s) to take on finding it. (Note the dual meaning of the term specification – it may mean either the entire **lex** source itself or, within it, a representation of a particular pattern to be recognized.) Whenever the input consists of patterns not sought, **lex** writes out the input exactly as it finds it. So, the simplest **lex** program is just the beginning rules delimiter, %%. It writes out the entire input to the output with no changes at all. Typically, the rules are more elaborate than that.

Specifications

You specify the patterns you are interested in with a notation called regular expressions. A regular expression is formed by stringing together characters with or without operators. The simplest regular expressions are strings of text characters with no operators at all. For example,

apple
orange
pluto

These three regular expressions match any occurrences of those character strings in an input text. If you want to have your lexical analyzer **a.out** remove every occurrence of **orange**, from the input text, you could specify the rule

```
orange;
```

Because you did not specify an action on the right (before the semi-colon), **lex** does nothing but print out the original input text with every occurrence of this regular expression removed, that is, without any occurrence of the string **orange** at all.

Unlike **orange** above, most of the expressions that we want to search for cannot be specified so easily. The expression itself might simply be too long. More commonly, the class of desired expressions is too large; it may, in fact, be infinite. Thanks to the use of operators, we can form regular expressions signifying any expression of a certain class. The **+** operator, for instance, means one or more occurrences of the preceding expression, the **?** means 0 or 1 occurrence(s) of the preceding expression (this is equivalent, of course, to saying that the preceding expression is optional), and ***** means 0 or more occurrences of the preceding expression. (It may at first seem odd to speak of 0 occurrences of an expression and to need an operator to capture the idea, but it is often quite helpful. We will see an example in a moment.) So **m+** is a regular expression matching any string of **m**s such as each of the following:

```
mmm
m
mmmmm
mm
```

and **7*** is a regular expression matching any string of zero or more 7s:

```
77
77777

777
```

The string of blanks on the third line matches simply because it has no 7s in it at all.

Brackets, [ ], indicate any one character from the string of characters specified between the brackets. Thus, **[dgka]** matches a single **d**, **g**, **k**, or **a**. Note that commas are not included within the brackets. Any comma here would be taken as a character to be recognized in the input text. Ranges within a standard alphabetic or numeric order are indicated with a hyphen, **-**. The sequence **[a-z]**, for instance, indicates any lowercase letter.

Somewhat more interestingly,

```
[A-Za-z0-9*&#]
```

is a regular expression that matches any letter (whether upper- or lowercase), any digit, an asterisk, an ampersand, or a sharp character. Given the input text

```
$$$$?? ????!!!*$$ $$$$$$&+====r~~# ((
```

the lexical analyzer with the previous specification in one of its rules will recognize the *, &, r, and #, perform on each recognition whatever action the rule specifies (we have not indicated an action here), and print out the rest of the text as it stands.

The operators become especially powerful in combination. For example, the regular expression to recognize an identifier in many programming languages is

```
[a-zA-Z][0-9a-zA-Z]*
```

An identifier in these languages is defined to be a letter followed by zero or more letters or digits, and that is just what the regular expression says. The first pair of brackets matches any letter. The second, if it were not followed by a *, would match any digit or letter. The two pairs of brackets with their enclosed characters would then match any letter followed by a digit or a letter. But with the asterisk, *, the example matches any letter followed by any number of letters or digits. In particular, it would recognize the following as identifiers:

e
pay
distance
pH
EngineNo99
R2D2

Note that it would not recognize the following as identifiers:

not_idenTIFER
5times
$hello

because **not_idenTIFER** has an embedded underscore; **5times** starts with a digit, not a letter; and **$hello** starts with a special character. Of course, you may want to write the specifications for these three examples as an exercise.

A potential problem with operator characters is how we can refer to them as characters to look for in our search pattern. The last example, for instance, will not recognize text with an * in it. **lex** solves the problem in one of two ways: a character enclosed in quotation marks or a character preceded by a \ is taken literally, that is, as part of the text to be searched for. To use the backslash method to recognize, say, an * followed by any number of digits, we can use the pattern

```
\*[1-9]*
```

To recognize a \ itself, we need two backslashes: \\.

Actions

Once **lex** recognizes a string matching the regular expression at the start of a rule, it looks to the right of the rule for the action to be performed. Kinds of actions include recording the token type found and its value, if any; replacing one token with another; and counting the number of instances of a token or token type. What you want to do is write these actions as program fragments in the host language C. An action may consist of as many statements as are needed for the job at hand. You may want to print out a message noting that the text has been found or a message transforming the text in some way. Thus, to recognize the expression Amelia Earhart and to note such recognition, the rule

```
"Amelia Earhart"   printf("found Amelia");
```

would do. And to replace in a text lengthy medical terms with their equivalent acronyms, a rule such as

```
Electroencephalogram    printf("EEG");
```

would be called for. To count the lines in a text, we need to recognize end-of-lines and increment a linecounter. **lex** uses the standard escape sequences from C like **n** for end-of-line. To count lines we might have

```
\n    lineno++;
```

where **lineno**, like other C variables, is declared in the definitions section that we discuss later.

lex stores every character string that it recognizes in a character array called **yytext[]**. You can print or manipulate the contents of this array as you want. Sometimes your action may consist of two or more C statements and you must (or for style and clarity, you choose to) write it on several lines. To inform **lex** that the action is for one rule only, simply enclose the C code in braces. For example, to count the total number of all digit strings in an input text, print the running total of the number of digit strings (not their sum, here) and print out each one as soon as it is found, your **lex** code might be

```
+?[1-9]+            { digstrngcount++;
                      printf("%d",digstrngcount);
                      printf("%s", yytext);   }
```

This specification matches digit strings whether they are preceded by a plus sign or not, because the **?** indicates that the preceding plus sign is optional. In addition, it will catch negative digit strings because that portion following the minus sign, **-**, will match the specification. The next section explains how to distinguish negative from positive integers.

Advanced lex Usage

lex provides a suite of features that lets you process input text riddled with quite complicated patterns. These include rules that decide what specification is relevant, when more than one seems so at first; functions that transform one matching pattern into another; and the use of definitions and subroutines. Before considering these features, you may want to affirm your understanding thus far by examining an example drawing together several of the points already covered.

```
%%
-[0-9]+         printf("negative integer");
+?[0-9]+        printf("positive integer");
-0.[0-9]+       printf("negative fraction, no whole
                  number part");
rail[ ]+road    printf("railroad is one word");
crook           printf("Here's a crook");
function        subprogcount++;
G[a-zA-Z]*     { printf("may have a G word here:
                  ", yytext);
                  Gstringcount++; }
```

The first three rules recognize negative integers, positive integers, and negative fractions between 0 and -1. The use of the terminating + in each specification ensures that one or more digits compose the number in question. Each of the next three rules recognizes a specific pattern. The specification for **railroad** matches cases where one or more blanks intervene between the two syllables of the word. In the cases of **railroad** and **crook**, you may have simply printed a synonym rather than the messages stated. The rule recognizing a **function** simply increments a counter. The last rule illustrates several points:

- The braces specify an action sequence extending over several lines.
- Its action uses the **lex** array **yytext**[], which stores the recognized character string.
- Its specification uses the * to indicate that zero or more letters may follow the **G**.

Some Special Features

Besides storing the recognized character string in **yytext**[], **lex** automatically counts the number of characters in a match and stores it in the variable **yyleng**. You may use this variable to refer to any specific character just placed in the array **yytext**[]. Remember that C numbers locations in an array starting with 0, so to print out the third digit (if there is one) in a just recognized integer, you might write

```
        [1-9]+          {if  (yyleng > 2)
                         printf("%c", yytext[2]); }
```

lex follows a number of high-level rules to resolve ambiguities that may arise from the set of rules that you write. *Prima facie*, any reserved word, for instance, could match two rules. In the lexical analyzer example developed later in the section on **lex** and **yacc**, the reserved word **end** could match the second rule as well as the seventh, the one for identifiers.

NOTE: **lex** follows the rule that where there is a match with two or more rules in a specification, the first rule is the one whose action will be executed.

By placing the rule for **end** and the other reserved words before the rule for identifiers, we ensure that our reserved words will be duly recognized.

Another potential problem arises from cases where one pattern you are searching for is the prefix of another. For instance, the last two rules in the lexical analyzer example above are designed to recognize > and > = . If the text has the string > = at one point, you might worry that the lexical analyzer would stop as soon as it recognized the > character to execute the rule for > rather than read the next character and execute the rule for > =.

NOTE: **lex** follows the rule that it matches the longest character string possible and executes the rule for that.

Here it would recognize the > = and act accordingly. As a further example, the rule would enable you to distinguish + from + + in a program in C.

Still another potential problem exists when the analyzer must read characters beyond the string you are seeking because you cannot be sure you've in fact found it until you've read the additional characters. These cases reveal the importance of trailing context. The classic example here is the DO statement in

FORTRAN. In the statement

```
        DO 50 k = 1 , 20, 1
```

we cannot be sure that the first 1 is the initial value of the index **k** until we read the first comma. Until then, we might have the assignment statement

```
        DO50k = 1
```

(Remember that FORTRAN ignores all blanks.) The way to handle this is to use the forward-looking slash, / (not the backslash, \), which signifies that what follows is trailing context, something not to be stored in **yytext**[], because it is not part of the token itself. So the rule to recognize the FORTRAN DO statement could be

```
    30/[ ]*[0-9][ ]*[a-z A-Z0-9]+=[a-z A-Z0-9]+,   printf("found
    DO");
```

Different versions of FORTRAN have limits on the size of identifiers, here the index name. To simplify the example, the rule accepts an index name of any length.

lex uses the **$** as an operator to mark a special trailing context – the end of line. (It is therefore equivalent to **n**.) An example would be a rule to ignore all blanks and tabs at the end of a line:

```
        [  \t]+$       ;
```

On the other hand, if you want to match a pattern only when it starts a line, **lex** offers you the circumflex, ^, as the operator. The formatter **nroff**, for example, demands that you never start a line with a blank, so you might want to check input to **nroff** with some such rule as:

```
^[ ]      printf("error: remove leading blank");
```

Finally, some of your action statements themselves may require your reading another character, putting one back to be read again a moment later, or writing a character on an output device. **lex** supplies three functions to handle these tasks – **input**(), **unput**(**c**), and **output**(**c**), respectively. One way to ignore all characters between two special characters, say between a pair of double quotation marks, would be to use **input**(), thus:

```
\"          while (input() != '"');
```

Upon finding the first double quotation mark, the generated **a.out** will simply continue reading all subsequent characters so long as none is a quotation mark, and not again look for a match until it finds a second double quotation mark.

To handle special I/O needs, such as writing to several files, you may use standard I/O routines in C to rewrite the functions **input**(), **unput**(**c**), and **output**. These and other programmer-defined functions should be placed in your subroutine section. Your new routines will then replace the standard ones. The standard **input**(), in fact, is equivalent to **getchar**(), and the standard **output**(**c**) is equivalent to **putchar**(**c**).

There are a number of **lex** routines that let you handle sequences of characters to be processed in more than one way. These include **yymore**(), **yyless**(n), and **REJECT**. Recall that the text matching a given specification is stored in the array **yytext**[]. In general, once the action is performed for the specification, the characters in **yytext**[] are overwritten with succeeding characters in the input stream to form the next match. The function **yymore**(), by contrast, ensures that the succeeding characters recognized are appended to those already in **yytext**[]. This lets you do one thing and then another, when one string of characters is significant and a longer one including the first is significant as well. Consider a character string bound by **B**s and interspersed with one at an arbitrary location.

```
B...B...B
```

In a simple code deciphering situation, you may want to count the number of characters between the first and second **B**'s and add it to the number of characters between the second and third **B**. (Only the last **B** is not to be counted.) The code to do this is

```
B[^B]*        { if (flag = 0)
                         save = yyleng;
                         flag = 1;
                         yymore();
                      else     {
                         importantno = save + yyleng;
                         flag = 0; }
                     }
```

where **flag**, **save**, and **importantno** are declared (and at least **flag** initialized to 0) in the definitions section. The **flag** distinguishes the character sequence terminating just before the second **B** from that terminating just before the third.

The function **yyless**(*n*) lets you reset the end point of the string to be considered to the *n*th character in the original **yytext**[]. Suppose you are again in the code deciphering business and the gimmick here is to work with only half the characters in a sequence ending with a certain one, say upper- or lowercase **Z**. The code you want might be

```
[a-yA-Y]+[Zz]       {  yyless(yyleng/2);
                    ... process first half of string... }
```

Finally, the function REJECT lets you more easily process strings of characters even when they overlap or contain one another as parts. REJECT does this by immediately jumping to the next rule and its specification without changing the contents of **yytext[]**. If you want to count the number of occurrences both of the regular expression **snapdragon** and of its subexpression **dragon** in an input text, the following will do:

```
snapdragon     {countflowers++; REJECT;}
dragon         countmonsters++;
```

As an example of one pattern overlapping another, the following counts the number of occurrences of the expressions **comedian** and **diana**, even where the input text has sequences such as **comediana..**:

```
comedian        {comiccount++; REJECT;}
diana           princesscount++;
```

Note that the actions here may be considerably more complicated than simply incrementing a counter. In all cases, the counters and other necessary variables are declared in the definitions section commencing the **lex** specification.

Definitions

The **lex** definitions section may contain any of several classes of items. The most critical are external definitions, **#include** statements, and abbreviations. Recall that for legal **lex** source this section is optional, but in most cases some of these items are necessary. External definitions have the form and function that they do in C. They declare that variables globally defined elsewhere (perhaps in another source file) will be accessed in your **lex**-generated **a.out**. Consider a declaration from an example to be developed later.

```
extern int tokval;
```

When you store an integer value in a variable declared in this way, it will be accessible in the routine, say a parser, that calls it. If, on the other hand, you want to define a local variable for use within the action sequence of one rule (as you might for the index variable for a loop), you can declare the variable at the start of the action itself right after the left brace, { .

The purpose of the **#include** statement is the same as in C: to include files of importance for your program. Some variable declarations and **lex** definitions might be needed in more than one **lex** source file. It is then advantageous to place them all in one file to be included in every file that needs them. One example occurs in using **lex** with **yacc**, which generates parsers that call a lexical analyzer. In this context, you should include the file **y.tab.h**, which may contain **#defines** for token names. Like the declarations, **#include** statements should come between %{ and }%, thus:

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

In the definitions section, after the %} that ends your **#include**'s and declarations, you place your abbreviations for regular expressions to be used in the rules section. The abbreviation appears on the left of the line and, separated by one or more spaces, its definition or translation appears on the right. When you later use abbreviations in your rules, be sure to enclose them within braces.

NOTE: The purpose of abbreviations is to avoid needless repetition in writing your specifications and to provide clarity in reading them.

As an example, reconsider the **lex** source reviewed at the beginning of this section on advanced **lex** usage. The use of definitions simplifies our later reference to digits, letters, and blanks. This is especially true if the specifications appear several times:

```
D               [0-9]
L               [a-zA-Z]
B               [ ]
%%
-{D}+            printf("negative integer");
+?{D}+           printf("positive integer");
-0.{D}+          printf("negative fraction");
G{L}*            printf("may have a G word here");
rail{B}+road      printf("railroad is one word");
crook            printf("criminal");
 \"\./{B}+       printf(".\"");
  .                .
  .                .
```

The last rule, newly added to the example and somewhat more complex than the others, is used in the WRITER'S WORKBENCH Software, an AT&T software product for promoting good writing. (See the *UNIX System WRITER'S WORKBENCH Software Release 3.0 User's Guide* for information on this product.) The rule ensures that a period always precedes a quotation mark at the end of a sentence. It would change example". to example."

Subroutines

You may want to use subroutines in **lex** for much the same reason that you do so in other programming languages. Action code that is to be used for several rules can be written once and called when needed. As with definitions, this can simplify the writing and reading of programs. The function **put_in_tabl**(), to be discussed in the next section on **lex** and **yacc**, is a good candidate for a subroutine.

Another reason to place a routine in this section is to highlight some code of interest or to simplify the rules section, even if the code is to be used for one rule only. As an example, consider the following routine to ignore comments in a language like C where comments occur between /* and */ :

```
"/*"                    skipcmnts();
.
.               /* rest of rules */
%%
skipcmnts()
{
        for(;;)
        {
           while (input() != '*');
           if (input() != '/') {
                 unput(yytext[yyleng-1]);
           else return;
        }
 }
```

There are three points of interest in this example. First, the **unput(c)** function (putting back the last character read) is necessary to avoid missing the final / if the comment ends unusually with a **/ . In this case, eventually having read an *, the analyzer finds that the next character is not the terminal / and must read some more. Second, the expression **yytext[yyleng-1]** picks out that last character read. Third, this routine assumes that the comments are not nested. (This is indeed the case with the C language.) If, unlike C, they are nested in the source text, after **input**()ing the first */ ending the inner group of comments, the **a.out** will read the rest of the comments as if they were part of the input to be searched for patterns.

Other examples of subroutines would be programmer-defined versions of the I/O routines **input**(), **unput(c)**, and **output**(), discussed above. Subroutines such as these that may be exploited by many different programs would probably do best to be stored in their own individual file or library to be called as needed. The appropriate **#include** statements would then be necessary in the definitions section.

Using lex with yacc

If you work on a compiler project or develop a program to check the validity of an input language, you may want to use the UNIX system program tool **yacc**. **yacc** generates parsers, programs that analyze input to ensure that it is syntactically correct. (**yacc** is discussed in detail in Chapter 6 of this guide.) **lex** often forms a fruitful union with **yacc** in the compiler development context. Whether or not you plan to use **lex** with **yacc**, be sure to read this section because it covers information of interest to all **lex** programmers.

The lexical analyzer that **lex** generates (not the file that stores it) takes the name **yylex**(). This name is convenient because **yacc** calls its lexical analyzer by this very name. To use **lex** to create the lexical analyzer for the parser of a compiler, you want to end each **lex** action with the statement **return** *token*, where *token* is a defined term whose value is an integer. The integer value of the token returned indicates to the parser what the lexical analyzer has found. The parser, whose file is called **y.tab.c** by **yacc**, then resumes control and makes another call to the lexical analyzer when it needs another token.

In a compiler, the different values of the token indicate what, if any, reserved word of the language has been found or whether an identifier, constant, arithmetic operand, or relational operator has been found. In the latter cases, the analyzer must also specify the exact value of the token: what the identifier is, whether the constant, say, is 9 or 888, whether the operand is + or * (multiply), and whether the relational operator is = or >. Consider the following portion of **lex** source for a lexical analyzer for some programming language perhaps slightly reminiscent of Ada:

```
beginreturn(BEGIN);
end                           return(END);
while                         return(WHILE);
if                            return(IF);
package                       return(PACKAGE);
reverse                       return(REVERSE);
loop                          return(LOOP);
[a-zA-Z][a-zA-Z0-9]*        { tokval = put_in_tabl();
                                return(IDENTIFIER); }
[0-9]+                      { tokval = put_in_tabl();
                                return(INTEGER); }
\+                         { tokval = PLUS;
                                return(ARITHOP); }
\-                        { tokval = MINUS;
                                return(ARITHOP); }
>                           { tokval = GREATER;
                                 return(RELOP); }
>=                          { tokval = GREATEREQL;
                                return(RELOP); }
```

Despite appearances, the tokens returned, and the values assigned to **tokval**, are indeed integers. Good programming style dictates that we use informative terms such as **BEGIN**, **END**, **WHILE**, and so forth to signify the integers the parser understands, rather than use the integers themselves. You establish the association by using **#define** statements in your parser calling routine in C. For example,

```
        #define BEGIN  1
        #define END  2
           .
        #define PLUS 7
           .
```

If the need arises to change the integer for some token type, you then change the **#define** statement in the parser rather than hunt through the entire program, changing every occurrence of the particular integer. In using **yacc** to generate your parser, it is helpful to insert the statement

```
            #include y.tab.h
```

into the definitions section of your **lex** source. The file **y.tab.h** provides **#define** statements that associate token names such as **BEGIN**, **END**, and so on with the integers of significance to the generated parser.

To indicate the reserved words in the example, the returned integer values suffice. For the other token types, the integer value of the token type is stored in the programmer-defined variable **tokval**. This variable, whose definition was an example in the definitions section, is globally defined so that the parser as well as the lexical analyzer can access it. **yacc** provides the variable **yylval** for the same purpose.

Note that the example shows two ways to assign a value to **tokval**. First, a function **put_in_tabl**() places the name and type of the identifier or constant in a symbol table so that the compiler can refer to it in this or a later stage of the compilation process. More to the present point, **put_in_tabl**() assigns a type value to **tokval** so that the parser can use the information immediately to determine the syntactic correctness of the input text. The function **put_in_tabl**() would be a routine that the compiler writer might place in the subroutines section discussed later. Second, in the last few actions of the example, **tokval** is assigned a specific integer indicating which operand or relational operator the analyzer recognized. If the variable PLUS, for instance, is associated with the integer 7 by means of the **#define** statement above, then when a + sign is recognized, the action assigns to **tokval** the value 7, which indicates the +. The analyzer indicates the general class of operator by the value it returns to the parser (in the example, the integer signified by ARITHOP or RELOP).

Running lex under the UNIX System

As you review the following few steps, you might recall Figure 5-1 at the start of the chapter. To produce the lexical analyzer in C, run

```
lex lex.l
```

where **lex.l** is the file containing your **lex** specification. The name **lex.l** is conventionally the favorite, but you may use whatever name you want. The output file that **lex** produces is automatically called **lex.yy.c**; this is the lexical analyzer program that you created with lex. You then compile and link this as you would any C program, making sure that you invoke the **lex** library with the **-ll** option:

```
cc lex.yy.c -ll
```

The **lex** library provides a default **main()** program that calls the lexical analyzer under the name **yylex()**, so you need not supply your own **main()**.

If you have the **lex** specification spread across several files, you can run **lex** with each of them individually, but be sure to rename or move each **lex.yy.c** file (with **mv**) before you run **lex** on the next one. Otherwise, each will overwrite the previous one. Once you have all the generated **.c** files, you can compile all of them, of course, in one command line.

With the executable **a.out** produced, you are ready to analyze any desired input text. Suppose that the text is stored under the filename **textin** (this name is also arbitrary). The lexical analyzer **a.out** by default takes input from your terminal. To have it take the file **textin** as input, simply use redirection, thus:

```
a.out < textin
```

By default, output will appear on your terminal, but you can redirect this as well:

```
a.out < textin > textout
```

In running **lex** with **yacc**, either may be run first.

```
yacc -d grammar.y
lex lex.l
```

spawns a parser in the file **y.tab.c**. (The **-d** option creates the file **y.tab.h**, which contains the **#define** statements that associate the **yacc** assigned integer token values with the user-defined token names.) To compile and link the output files produced, run

```
cc lex.yy.c y.tab.c -ly -ll
```

Note that the **yacc** library is loaded (with the **-ly** option) before the **lex** library (with the **-ll** option) to ensure that the **main()** program supplied will call the **yacc** parser.

There are several options available with the **lex** command. If you use one or more of them, place them between the command name **lex** and the filename argument. If you care to see the C program, **lex.yy.c**, that **lex** generates on your terminal (the default output device), use the **-t** option.

```
lex -t lex.l
```

The **-v** option prints out for you a small set of statistics describing the so-called finite automata that **lex** produces with the C program **lex.yy.c**. (For a detailed account of finite automata and their importance for **lex**, see the Aho, Sethi, and Ullman text, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.)

lex uses a table (a two-dimensional array in C) to represent its finite automaton. The maximum number of states that the finite automaton requires is set by default to 500. If your **lex** source has a large number of rules or the rules are very complex, this default value may be too small. You can enlarge the value by placing another entry in the definitions section of your **lex** source, as follows:

```
%n 700
```

This entry tells **lex** to make the table large enough to handle as many as 700 states. (The **-v** option will indicate how large a number you should choose.) If you have need to increase the maximum number of state transitions beyond 2000, the designated parameter is **a**, thus:

```
%a 2800
```

Finally, check the *Programmer's Reference Manual* page on **lex** for a list of all the options available with the **lex** command. In addition, review the paper by Lesk (the originator of **lex**) and Schmidt, "Lex – A Lexical Analyzer Generator," in volume 5 of the *UNIX Programmer's Manual*, Holt, Rinehart, and Winston, 1986. It is somewhat dated, but offers several interesting examples.

This tutorial has introduced you to **lex** programming. As with any programming language, the way to master it is to write programs and then write some more.

Chapter 6: yacc

Introduction

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification that includes:

- a set of rules to describe the elements of the input
- code to be invoked when a rule is recognized
- either a definition or declaration of a low-level routine to examine the input

yacc then turns the specification into a C language function that examines the input stream. This function, called a parser, works by calling the low-level input scanner. The low-level input scanner, called a lexical analyzer, picks up items from the input stream. The selected items are known as tokens. Tokens are compared to the input construct rules, called grammar rules. When one of the rules is recognized, the user code supplied for this rule, (an action) is invoked. Actions are fragments of C language code. They can return values and make use of values returned by other actions.

The heart of the **yacc** specification is the collection of grammar rules. Each rule describes a construct and gives it a name. For example, one grammar rule might be

```
date  :  month_name  day  ´,´  year   ;
```

where **date, month_name, day**, and **year** represent constructs of interest; presumably, **month_name, day**, and **year** are defined in greater detail elsewhere. In the example, the comma is enclosed in single quotes. This means that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule and have no significance in evaluating the input. With proper definitions, the input

```
July  4, 1776
```

might be matched by the rule.

The lexical analyzer is an important part of the parsing function. This user-supplied routine reads the input stream, recognizes the lower-level constructs, and communicates these as tokens to the parser. The lexical analyzer recognizes constructs of the input stream as terminal symbols; the parser recognizes constructs as nonterminal symbols. To avoid confusion, we will refer to terminal symbols as tokens.

There is considerable leeway in deciding whether to recognize constructs using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n'  ;
month_name : 'F' 'e' 'b'  ;

             ...

month_name : 'D' 'e' 'c'  ;
```

might be used in the above example. While the lexical analyzer only needs to recognize individual letters, such low-level rules tend to waste time and space, and may complicate the specification beyond the ability of **yacc** to deal with it. Usually, the lexical analyzer recognizes the month names and returns an indication that a **month_name** is seen. In this case, **month_name** is a token and the detailed rules are not needed.

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7/4/1776
```

as a synonym for

```
        July 4, 1776
```

on input. In most cases, this new rule could be slipped into a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. With a left-to-right scan input errors are detected as early as is theoretically possible. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data usually can be found quickly. Error handling, provided as part of the input specifications, permits the reentry of bad data or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases often can be corrected by making the lexical analyzer more powerful or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems. Moreover, the constructs that are difficult for **yacc** to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The remainder of this chapter describes the following subjects:

- basic process of preparing a **yacc** specification
- parser operation
- handling ambiguities
- handling operator precedences in arithmetic expressions
- error detection and recovery
- the operating environment and special features of the parsers **yacc** produces

- suggestions to improve the style and efficiency of the specifications
- advanced topics

In addition, there are two examples and a summary of the **yacc** input syntax.

Basic Specifications

Names refer to either tokens or nonterminal symbols. **yacc** requires token names to be declared as such. While the lexical analyzer may be included as part of the specification file, it is perhaps more in keeping with modular design to keep it as a separate file. Like the lexical analyzer, other subroutines may be included as well. Thus, every specification file theoretically consists of three sections: the declarations, (grammar) rules, and subroutines. The sections are separated by double percent signs, %% (the percent sign is generally used in **yacc** specifications as an escape character).

A full specification file looks like:

> *declarations*
> %%
> *rules*
> %%
> *subroutines*

when all sections are used. The *declarations* and *subroutines* sections are optional. The smallest legal **yacc** specification is

> %%
> **rules**

Blanks, tabs, and newlines are ignored, but they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in /* ... */, as in the C language.

The rules section is made up of one or more grammar rules. A grammar rule has the form

```
A   :   BODY   ;
```

where **A** represents a nonterminal symbol, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of any length and may be made up of letters, dots, underscores, and digits although a digit may not be the first character of a name. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes, '. As in the C language, the backslash, \, is an escape character within literals, and all the C language escapes are recognized. Thus:

'\n'	newline
'\r'	return
'\''	single quote (')
'\\'	backslash (\ )
'\t'	tab
'\b'	backspace
'\f'	form feed
'\xxx'	*xxx* in octal notation

are understood by **yacc**. For a number of technical reasons, the NULL character (\0 or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar, ¦, can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule is dropped before a vertical bar. Thus the grammar rules

```
A   :  B  C  D   ;
A   :  E  F   ;
A   :  G   ;
```

can be given to **yacc** as

```
A   :  B  C  D
    ¦  E  F
    ¦  G
    ;
```

by using the vertical bar. It is not necessary that all grammar rules with the same left side appear together in the grammar rules section although it makes the input more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by

```
epsilon :   ;
```

The blank space following the colon is understood by **yacc** to be a nonterminal symbol named **epsilon**.

Names representing tokens must be declared. This is most simply done by writing

```
%token   name1  name2 ...
```

in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the start symbol has particular importance. By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible and desirable to declare the start symbol explicitly in the declarations section using the **%start** keyword.

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the end-marker. The end-marker is represented by either a zero or a negative number. If the tokens up to but not including the end-marker form a construct that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually the end-marker represents some reasonably obvious I/O status, such as end of file or end of record.

Actions

With each grammar rule, the user may associate actions to be performed when the rule is recognized. Actions may return values and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens if desired.

An action is an arbitrary C language statement and as such can do input and output, call subroutines, and alter arrays and variables. An action is specified by one or more statements enclosed in curly braces, {, and }. For example:

```
A   :  '('  B  ')'
    {
       hello( 1, "abc" );
    }
```

and

```
XXX    :  YYY  ZZZ
       {
          (void) printf("a message\n");
          flag = 25;
       }
```

are grammar rules with actions.

The dollar sign symbol, **$**, is used to facilitate communication between the actions and the parser, The pseudo-variable **$$** represents the value returned by the complete action. For example, the action

```
{  $$ = 1;  }
```

returns the value of one; in fact, that's all it does.

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables **$1, $2, ... $***n*. These refer to the values returned by components 1 through *n* of the right side of a rule, with the components being numbered from left to right. If the rule is

```
A   :  B  C  D   ;
```

then **$2** has the value returned by **C**, and **$3** the value returned by **D**.

The rule

```
expr   :   '('  expr  ')'   ;
```

provides a common example. One would expect the value returned by this rule to be the value of the *expr* within the parentheses. Since the first component of the action is the literal left parenthesis, the desired logical result can be indicated by

```
expr    :     '('  expr  ')'
        {
              $$ = $2 ;
        }
```

By default, the value of a rule is the value of the first element in it (**$1**). Thus, grammar rules of the form

```
A   :   B   ;
```

frequently need not have an explicit action. In previous examples, all the actions came at the end of rules. Sometimes, it is desirable to get control before a rule is fully parsed. **yacc** permits an action to be written in the middle of a rule as well as at the end. This action is assumed to return a value accessible through the usual **$** mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule below the effect is to set **x** to 1 and **y** to the value returned by **C**.

```
A    :    B
             {
                 $$ = 1;
             }
             C
     {
             x = $2;
             y = $3;
     }
     ;
```

Actions that do not terminate a rule are handled by **yacc** by manufacturing a new nonterminal symbol name and a new rule matching this name to the empty string. The interior action is the action triggered by recognizing this added rule. **yacc** treats the above example as if it had been written

```
$ACT    :    /* empty */
        {
             $$ = 1;
        }
        ;

A       :    B  $ACT  C
        {
             x = $2;
             y = $3;
        }
        ;
```

where **$ACT** is an empty action.

In many applications, output is not done directly by the actions. A data structure, such as a parse tree, is constructed in memory and transformations are applied to it before output is generated. Parse trees are particularly easy to construct given routines to build and maintain the tree structure desired. For example, suppose there is a C function node written so that the call

```
node( L, n1, n2 )
```

creates a node with label **L** and descendants **n1** and **n2** and returns the index of the newly created node. Then a parse tree can be built by supplying actions such as

```
expr    :    expr  '+'  expr
        {
             $$ = node( '+', $1, $3 );
        }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and can be made known to the lexical analyzer. For example:

```
        %{   int variable = 0;   %}
```

could be placed in the declarations section making **variable** accessible to all of the actions. Users should avoid names beginning with **yy** because the **yacc** parser uses only such names. In the examples shown thus far all the values are integers. A discussion of values of other types is found in the section "Advanced Topics."

Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called **yylex**. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable **yylval**.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by **yacc** or the user. In either case, the **#define** mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like

```
int yylex()
{
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch (c)
        {
           ...
           case '0':
           case '1':
           ...
           case '9':
           yylval = c - '0';
           return (DIGIT);
           ...
        }
        ...
}
```

to return the appropriate token.

The intent is to return a token number of DIGIT and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the subroutines section of the specification file, the identifier DIGIT is defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling and should not be used naively.

In the default situation, token numbers are chosen by **yacc**. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257. If the **yacc** command is invoked with the **-d** option a file called **y.tab.h** is generated. **y.tab.h** contains **#define** statements for the tokens.

If the user prefers to assign the token numbers, the first appearance of the token name or literal in the declarations section must be followed immediately by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined this way are assigned default definitions by **yacc**. The potential for duplication exists here. Care must be taken to make sure that all token numbers are distinct.

For historical reasons, the end-marker must have token number 0 or negative. This token number cannot be redefined by the user. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the **lex** utility. Lexical analyzers produced by **lex** are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. **lex** can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN), which do not fit any theoretical framework and whose lexical analyzers must be crafted by hand.

Parser Operation

yacc turns the specification file into a C language procedure, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex and will not be discussed here. The parser itself, though, is relatively simple and understanding its usage will make treatment of error recovery and ambiguities easier.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the look-ahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0 (the stack contains only state 0) and no look-ahead token has been read.

The machine has only four actions available – **shift**, **reduce**, **accept**, and **error**. A step of the parser is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.
2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off of the stack and in the look-ahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a look-ahead token. For example, in state 56 there may be an action

```
IF   shift 34
```

which says, in state 56, if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

The **reduce** action keeps the stack from growing without bounds. **reduce** actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right-hand side by the left-hand side. It may be necessary to consult the look-ahead token to decide whether or not to **reduce** (usually it is not necessary). In fact, the default action (represented by a dot) is often a **reduce** action.

reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. The action

```
.    reduce 18
```

refers to grammar rule 18, while the action

```
IF    shift 34
```

refers to state 34.

Suppose the rule

```
A   :   x  y  z    ;
```

is being reduced. The **reduce** action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing x, y, and z and no longer serve any useful purpose. After popping these states, a state is uncovered, which was the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of **A**. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a **goto** action. In particular, the look-ahead token is cleared by a shift but is not affected by a **goto**. In any case, the uncovered state contains an entry such as

```
A    goto 20
```

causing state 20 to be pushed onto the stack and become the current state.

In effect, the **reduce** action turns back the clock in the parse popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stacks. The uncovered state is in fact the current state.

The **reduce** action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions. When a **shift** takes place, the external variable **yylval** is copied onto the value stack. After the return from the user code, the reduction is carried out. When the **goto** action is done, the external variable **yyval** is copied onto the value stack. The pseudo-variables **$1**, **$2**, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The **accept** action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker and indicates that the parser has successfully done its job. The **error** action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) will be discussed later.

Consider:

```
%token  DING  DONG  DELL
%%
rhyme    :   sound  place
         ;
sound    :   DING  DONG
         ;
place    :   DELL
         ;
```

as a **yacc** specification.

When **yacc** is invoked with the **-v** option, a file called **y.output** is produced with a human-readable description of the parser. The **y.output** file corresponding to the above grammar (with some statistics stripped off the end) follows.

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place
```

```
                              - CONTINUED -

        DELL  shift 5
        .  error

        place   goto 4

state 3
        sound   :   DING_DONG

        DONG  shift 6
        .  error

state 4
        rhyme   :   sound  place_    (1)

        .   reduce  1

state 5
        place   :   DELL_    (3)

        .   reduce  3

state 6
        sound   :   DING  DONG_    (2)

        .   reduce  2
```

The actions for each state are specified and there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen and what is yet to come in each rule. The following input

```
       DING  DONG  DELL
```

can be used to track the operations of the parser. Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, DING, is read and becomes the look-ahead token. The action in state 0 on DING is **shift 3**, state 3 is pushed onto the stack, and the look-ahead token is cleared. State 3 becomes the current state. The next token, DONG, is read and becomes the look-ahead token. The action in state 3 on the token DONG is **shift 6**, state 6 is pushed onto the stack, and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, the parser reduces by

```
        sound  :   DING  DONG
```

which is rule 2. Two states, 6 and 3, are popped off of the stack uncovering state 0. Consulting the description of state 0 (looking for a **goto** on **sound**),

```
        sound   goto 2
```

is obtained. State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token, DELL, must be read. The action is **shift 5**, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The **goto** in state 2 on **place** (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a **goto** on **rhyme** causing the parser to enter state 1. In state 1, the input is read and the end-marker is obtained indicated by **$end** in the **y.output** file. The action in state 1 (when the end-marker is seen) successfully ends the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, etc. A few minutes spent with this and other simple examples is repaid when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
        expr    :    expr  '-'  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
        expr  -  expr  -  expr
```

the rule allows this input to be structured as either

```
        (  expr  -  expr  )  -  expr
```

or as

```
        expr  -  (  expr  -  expr  )
```

(The first is called left association, the second right association.)

yacc detects such ambiguities when it is attempting to build the parser. Given the input

```
        expr  -  expr  -  expr
```

consider the problem that confronts the parser. When the parser has read the second *expr*, the input seen

```
        expr  -  expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to **expr** (the left side of the rule). The parser would then read the final part of the input

```
        -  expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, if the parser sees

```
expr  -  expr
```

it could defer the immediate application of the rule and continue reading the input until

```
expr  -  expr  -  expr
```

is seen. It could then apply the rule to the rightmost three symbols reducing them to *expr*, which results in

```
expr  -  expr
```

being left. Now the rule can be reduced once more. The effect is to take the right associative interpretation. Thus, having read

```
expr  -  expr
```

the parser can do one of two legal things, a shift or a reduction. It has no way of deciding between them. This is called a **shift-reduce** conflict. It may also happen that the parser has a choice of two legal reductions. This is called a **reduce-reduce** conflict. Note that there are never any **shift-shift** conflicts.

When there are **shift-reduce** or **reduce-reduce** conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing the choice to make in a given situation is called a disambiguating rule.

yacc invokes two default disambiguating rules:

1. In a **shift-reduce** conflict, the default is to do the shift.
2. In a **reduce-reduce** conflict, the default is to reduce by the earlier grammar rule (in the **yacc** specification).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but **reduce-reduce** conflicts should be avoided when possible.

Conflicts may arise because of mistakes in input or logic or because the grammar rules (while consistent) require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, **yacc** always reports the number of **shift-reduce** and **reduce-reduce** conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural and produces slower parsers. Thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider

```
stat    :   IF  '('  cond  ')'  stat
        |   IF  '('  cond  ')'  stat  ELSE  stat
        ;
```

which is a fragment from a programming language involving an **if-then-else** statement. In these rules, IF and ELSE are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the simple **if** rule and the second the **if-else** rule.

These two rules form an ambiguous construction because input of the form

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways

```
IF  ( C1 )
{
        IF  ( C2 )
                S1
}
ELSE
        S2
```

or

```
IF  ( C1 )
{
        IF  ( C2 )
                S1
        ELSE
                S2
}
```

where the second interpretation is the one given in most programming languages having this construct; each ELSE is associated with the last preceding un-ELSE'd IF. In this example, consider the situation where the parser has seen

```
IF  (  C1  )  IF  (  C2  )  S1
```

and is looking at the ELSE. It can immediately reduce by the simple **if** rule to get

```
IF  (  C1  )  stat
```

and then read the remaining input

```
ELSE  S2
```

and reduce

```
IF  (  C1  )  stat  ELSE  S2
```

by the **if-else** rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, S2 read, and then the right-hand portion of

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be reduced by the if-else rule to get

```
IF  (  C1  )  stat
```

which can be reduced by the simple **if** rule. This leads to the second of the above groupings of the input which is usually desired.

Once again, the parser can do two valid things – there is a **shift-reduce** conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This **shift-reduce** conflict arises only when there is a particular current input symbol, ELSE, and particular inputs, such as

```
IF  (  C1  )  IF  (  C2  )  S1
```

have already been seen. In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (**-v**) option output file. For example, the output corresponding to the above conflict state might be

```
23: shift-reduce conflict (shift 45, reduce 18) on ELSE

state 23

   stat  :  IF  (  cond  )  stat_         (18)
   stat  :  IF  (  cond  )  stat_ELSE  stat

   ELSE     shift 45
   .        reduce 18
```

where the first line describes the conflict – giving the state and the input symbol. The ordinary state description gives the grammar rules active in the state and the parser actions. Recall that the underline marks the portion of the grammar rules, which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF  (  cond  )  stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is ELSE, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
        stat   :   IF   (   cond   )   stat   ELSE_stat
```

because the ELSE will have been shifted in this state. In state 23, the alternative action (describing a dot, .), is to be done if the input symbol is not mentioned explicitly in the actions. In this case, if the input symbol is not ELSE, the parser reduces to

```
        stat   :   IF   '('   cond   ')'   stat
```

by grammar rule 18.

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the **y.output** file, the rule numbers are printed in parentheses after those rules, which can be reduced. In most states, there is a reduce action possible in the state and this is the default command. The user who encounters unexpected **shift-reduce** conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
        expr  :  expr  OP  expr
```

and

```
        expr  :  UNARY  expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts. As disambiguating rules, the user specifies the precedence or binding strength of all the operators and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus:

```
        %left  '+'  '-'
        %left  '*'  '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative and have lower precedence than star and slash, which are also left associative. The keyword **%right** is used to describe right associative operators, and the keyword **%nonassoc** is used to describe operators, like the operator **.LT.** in FORTRAN, that may not associate with themselves. Thus:

```
A  .LT.  B  .LT.  C
```

is illegal in FORTRAN and such an operator would be described with the keyword **%nonassoc** in **yacc**. As an example of the behavior of these declarations, the description

```
%right  '='
%left  '+'  '-'
%left  '*'  '/'

%%

expr    :   expr  '='  expr
        |   expr  '+'  expr
        |   expr  '-'  expr
        |   expr  '*'  expr
        |   expr  '/'  expr
        |   NAME
        ;
```

might be used to structure the input

```
a  =  b  =  c*d  -  e  -  f*g
```

as follows

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

in order to perform the correct precedence of operators. When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary minus, −.

Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, **%prec**, changes the precedence level associated with a particular grammar rule. The keyword **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules

```
%left  '+'  '-'
%left  '*'  '/'

%%

expr    :    expr  '+'  expr
        |    expr  '-'  expr
        |    expr  '*'  expr
        |    expr  '/'  expr
        |    '-'  expr      %prec  '*'
        |    NAME
        ;
```

might be used to give unary minus the same precedence as multiplication.

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

Precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. Precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a **reduce-reduce** conflict or there is a **shift-reduce** conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a **shift-reduce** conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action – **shift** or **reduce** – associated with the higher precedence. If precedences are equal, then associativity is used. Left associative implies **reduce**; right associative implies **shift**; nonassociating implies **error**.

Conflicts resolved by precedence are not counted in the number of **shift-reduce** and **reduce-reduce** conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar. It is a good idea to be sparing with precedences and use them in a cookbook fashion until some experience has been gained. The **y.output** file is very useful in deciding whether the parser is actually doing what was intended.

Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and/or, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides the token name **error**. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place. The parser pops its stack until it enters a state where the token **error** is legal. It then behaves as if the token **error** were the current look-ahead token and performs the action encountered. The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
	stat	:	error
```

means that on a syntax error the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Rules such as

```
        stat    :    error   ';'
```

are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule will be reduced and any cleanup action associated with it performed.

Another form of **error** rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example

```
input   :    error   '\n'
                     {
                         (void) printf( "Reenter last line: " );
                     }
                     input
        {
           $$ = $4;
        }
        ;
```

is one way to do this. There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable. For this reason, there is a mechanism that can force the parser to believe that error recovery has been accomplished. The statement

```
        yyerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as

```
input   :   error  '\n'
                 {
                    yyerrok;
                    (void) printf( "Reenter last line: " );
                 }
                 input
        {
            $$ = $4;
        }
        ;
```

which is somewhat better.

As previously mentioned, the token seen immediately after the **error** symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous look-ahead token must be cleared. The statement

```
        yyclearin ;
```

in an action will have this effect. For example, suppose the action after **error** were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by **yylex** is presumably the first token in a legal statement. The old illegal token must be discarded and the **error** state reset. A rule similar to

```
stat    :   error
        {
           resynch();
           yyerrok  ;
           yyclearin;
        }
        ;
```

could perform this.

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C language subroutines, called **y.tab.c**. The function produced by **yacc** is called **yyparse**(); it is an integer valued function. When it is called, it in turn repeatedly calls **yylex**(), the lexical analyzer supplied by the user (see "Lexical Analysis"), to obtain input tokens. Eventually, an error is detected, **yyparse**() returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, **yyparse**() returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C language program, a routine called **main**() must be defined that eventually calls **yparse**(). In addition, a routine called **yyerror**() is needed to print a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of **main**() and **yerror**(). The library is accessed by a **-ly** argument to the **cc**(1) command or to the loader. The source codes

```
main()
{
     return (yyparse());
}
```

and

```
# include <stdio.h>

yyerror(s)
        char *s;
{
        (void) fprintf(stderr, "%s\n", s);
}
```

show the triviality of these default programs. The argument to **yerror**() is a string containing an error message, usually the string **syntax error**. The average application wants to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics. Since the **main()** routine is probably supplied by the user (to read arguments, etc.), the **yacc** library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable **yydebug** is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions including a discussion of the input symbols read and what the parser actions are. It is possible to set this variable by using **sdb**.

Hints for Preparing Specifications

This part contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following are a few style hints.

1. Use all uppercase letters for token names and all lowercase letters for nonterminal names. This is useful in debugging.
2. Put grammar rules and actions on separate lines. It makes editing easier.
3. Put all rules with the same left-hand side together. Put the left-hand side in only once and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by one tab stop and action bodies by two tab stops.
6. Put complicated actions into subroutines defined in separate files.

Example 1 is written following this style, as are the examples in this section (where space permits). The user must decide about these stylistic questions. The central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the **yacc** parser encourages so called left recursive grammar rules. Rules of the form

```
name    :    name   rest_of_rule   ;
```

match this algorithm. These rules such as

```
list    :    item
        |    list   ','   item
        ;
```

and

```
seq     :    item
        |    seq   item
        ;
```

frequently arise when writing specifications of sequences and lists. In each of these cases, the first rule will be reduced for the first item only; and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :    item
        |    item   seq
        ;
```

the parser is a bit bigger; and the items are seen and reduced from right to left. More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, the user should use left recursion wherever reasonable.

It is worth considering if a sequence with zero elements has any meaning, and if so, consider writing the sequence specification as

```
seq     :    /* empty */
        |    seq   item
        ;
```

using an empty rule. Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read.

Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen when it hasn't seen enough to know!

Lexical Tie-Ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings, or names might be entered into a symbol table in declarations but not in expressions. One way of handling these situations is to create a global flag that is examined by the lexical analyzer and set by actions. For example,

```
%{
   int dflag;
%}
  ...  other declarations ...

%%

prog    :   decls  stats
        ;

decls   :   /* empty */
        {
               dflag = 1;
        }
        |   decls  declaration
        ;

stats   :   /* empty */
        {
               dflag = 0;
        }
        |   stats  statement
        ;

    ...  other rules ...
```

specifies a program that consists of zero or more declarations followed by zero or more statements. The flag **dflag** is now 0 when reading statements and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of back-door approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit you to use words like **if**, which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**. It is difficult to pass information to the lexical analyzer telling it this instance of **if** is a keyword and that instance is a variable. The user can make a stab at it using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be reserved, i.e., forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

Advanced Topics

This part discusses a number of advanced features of **yacc**.

Simulating error and accept in Actions

The parsing actions of **error** and **accept** can be simulated in an action by use of macros YYACCEPT and YYERROR. The YYACCEPT macro causes **yyparse()** to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; **yyerror()** is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple end-markers or context sensitive syntax checking.

Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit.

```
sent    :   adj  noun  verb  adj  noun
        {
            look at the sentence ...
        }
        ;
adj     :   THE
        {
                $$ = THE;
        }
        |   YOUNG
        {
                $$ = YOUNG;
        }
        ...
        ;
noun    :   DOG
        {
            $$ = DOG;
        }
        |   CRONE
        {
            if( $0 == YOUNG )
            {
                (void) printf( "what?\n" );
            }
            $$ = CRONE;
        }
        ;
        ...
```

In this case, the digit may be 0 or negative. In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol **noun** in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism prevents a great deal of trouble especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. **yacc** can also support values of other types including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type checked. **yacc** value stack is declared to be a **union** of the various types of values desired. The user declares the union and associates union member names with each token and nonterminal symbol having a value. When the value is referenced through a **$$** or **$n** construction, **yacc** will automatically insert the appropriate union name so that no unwanted conversions take place. In addition, type checking commands such as **lint** are far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union. This must be done by the user since other subroutines, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes

```
%union
{
   body of union ...
}
```

in the declaration section. This declares the **yacc** value stack and the external variables **yylval** and **yyval** to have type equal to this union. If **yacc** was invoked with the **-d** option, the union declaration is copied onto the **y.tab.h** file as YYSTYPE.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
<name>
```

is used to indicate a union member name. If this follows one of the keywords **%token**, **%left**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name **optype**. Another keyword, **%type**, is used to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

to associate the union member **nodetype** with the nonterminal symbols **expr** and **stat**.

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as **$0**) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between < and > immediately after the first $. The example

```
rule    :    aaa
                  {
                     $<intval>$ = 3;
                  }
                  bbb
        {
             fun( $<intval>2, $<other>0 );
        }
        ;
```

shows this usage. This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Example 2. The facilities in this subsection are not triggered until they are used. In particular, the use of **%type** will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of **$n** or **$$** to refer to something with no defined type is diagnosed.

If these facilities are not triggered, the **yacc** value stack is used to hold **ints**.

yacc Input Syntax

This section has a description of the **yacc** input syntax as a **yacc** specification. Context dependencies, etc. are not considered. Ironically, although **yacc** accepts an LALR(1) grammar, the **yacc** input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, and comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERs but never as part of C_IDENTIFIERs.

```
    /* grammar for the input to yacc */

    /* basic entries */
%token      IDENTIFIER    /* incld. identifiers & literals */
%token      C_IDENTIFIER  /* identifier followed by a : */
%token      NUMBER        /* [0-9]+ */

     /*     reserved words: %type=>TYPE %left=>LEFT,etc. */

%token      LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token      MARK     /* the %% mark */
%token      LCURL    /* the %{ mark */
%token      RCURL    /* the %} mark */

    /*  ASCII character literals stand for themselves */
```

```
                         - CONTINUED -
%token    spec

%%

spec    :    defs MARK rules tail
        ;
tail    :    MARK
        {
                 In this action, eat up the rest of the file
        }
        |    /* empty: the second MARK is optional */
        ;

defs    :    /* empty */
        |    defs def
        ;
def     :    START IDENTIFIER
        |    UNION
        {
               Copy union definition to output
        }
        |   LCURL
        {
                    Copy C code to output file
        }
            RCURL
        |   rword tag nlist
        ;

rword   :   TOKEN
        |   LEFT
        |   RIGHT
        |   NONASSOC
        |   TYPE
        ;

tag     :   /* empty: union tag is optional */
        |   '<' IDENTIFIER '>'
        ;
```

```
                              - CONTINUED -
nlist   :  nmno
        |  nlist nmno
        |  nlist ',' nmno
        ;
nmno    :  IDENTIFIER          /* Note: literal illegal
                          with % type */
        |  IDENTIFIER NUMBER   /* Note: illegal with %
                          type */
        ;

   /* rule section */

rules   :  C_IDENTIFIER rbody prec
        |  rules rule
        ;
rule    :  C_IDENTIFIER rbody prec
        |  '|' rbody prec
        ;


rbody   :  /* empty */
        |  rbody IDENTIFIER
        |  rbody act
        ;

act     :  '{'
           {
                 Copy action translate $$ etc.
           }
           '}'
        ;

prec    :  /* empty */
        |  PREC IDENTIFIER
        |  PREC IDENTIFIER act
        |  prec ';'
        ;
```

Examples

1. A Simple Example

This example gives the complete **yacc** applications for a small desk calculator; the calculator has 26 registers labeled **a** through **z** and accepts arithmetic expressions made up of the operators

```
+, -, *, /, % (mod operator), & (bitwise and), |
(bitwise or) and assignments.
```

If an expression at the top level is an assignment, only the assignment is done; otherwise, the expression is printed. As in the C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer is much simpler than for most applications, and the output is produced immediately line by line. Note the way that decimal and octal integers are read in by grammar rules. This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

 int regs[26];
 int base;

%}
```

```
                         - CONTINUED -
%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /* supplies precedence for unary minus */

%%          /* beginning of rules section */

list        :  /* empty */
            |  list stat '\n'
            |  list error '\n'
            {
               yyerrok;
            }
            ;

stat        :  expr
            {
               (void) printf( "%d\n", $1 );
            }
            |  LETTER '=' expr
            {
               regs[$1] = $3;
            }
            ;

expr        :  '(' expr ')'
            {
                  $$ = $2;
            }
            |  expr '+' expr
            {
                  $$ = $1 + $3;
```

- CONTINUED -

```
          }
          ¦  expr '-' expr
          {
                $$ = $1 - $3;
          }
          ¦  expr '*' expr
          {
                $$ = $1 * $3;
          }
          ¦  expr '/' expr
          {
                $$ = $1 / $3;
          }
          ¦   exp '%' expr
          {
                 $$ = $1 % $3;
          }
          ¦   expr '&' expr
          {
                $$ = $1 & $3;
          }
          ¦   expr '¦' expr
          {
                $$ = $1 ¦ $3;
          }
          ¦  '-' expr  %prec UMINUS
          {
                $$ = -$2;
          }
          ¦  LETTER
          {
                $$ = reg[$1];
          }
          ¦  number
          ;
```

```
                          - CONTINUED -

number    :   DIGIT
          {
                 $$ = $1; base = ($1==0) ? 8 ; 10;
          }
          |   number DIGIT
          {
                 $$ = base * $1 + $2;
          }
          ;

%%              /* beginning of subroutines section */

int yylex( )    /* lexical analysis routine */
{               /* return LETTER for lowercase letter, */
                /* yylval = 0 through 25 */
                /* returns DIGIT for digit, yylval = 0 - 9 */
                /* all other char. are returned immediately*/

          int c;
                      /*skip blanks*/
          while ((c = getchar()) == ' ')
                 ;

                      /* c is now nonblank */

          if (islower(c))
          {
                      yylval = c - 'a';
                      return (LETTER);
          }
          if (isdigit(c))
          {
                      yylval = c - '0';
                      return (DIGIT);

          }
          return (c);
}
```

2. An Advanced Example

This section gives an example of a grammar using some of the advanced features. The desk calculator example in Example 1 is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants; the arithmetic operations +, - *, /, unary - **a** through **z**. Moreover, it also understands intervals written

```
(X,Y)
```

where **X** is less than or equal to **Y**. There are 26 interval valued variables **A** through **Z** that may also be used. The usage is similar to that in Example 1; assignments return no value and print nothing while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C. Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, INTERVAL, by using **typedef**. **yacc** value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that the entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions – division by an interval containing 0 and an interval presented in the wrong order. The error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**: 18 **shift-reduce** and 26 **reduce-reduce**. The problem can be seen by looking at the two input lines.

```
2.5 + (3.5 - 4.)
```

and

```
        2.5 + (3.5, 4)
```

Notice that the 2.5 is to be used in an interval value expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator-one when the left operand is a scalar and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflict will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C language library routine **atof()** is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar provoking a syntax error in the parser and thence error recovery.

```
%{

#include <stdio.h>
#include <ctype.h>

typedef struct interval
{
     double lo, hi;
}  INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[26];
INTERVAL vreg[26];

%}

%start line

%union
{
  int ival;
  double dval;
  INTERVAL vval;
}

%token <ival> DREG VREG  /* indices into dreg, vreg arrays*/


%token <dval> CONST       /* floating point constant */

%type <dval> dexp         /* expression */

%type <vval> vexp         /* interval expression */
```

```
                          - CONTINUED -

  /* precedence information about the operators */

%left   '+' '-'
%left   '*' '/'
%left   UMINUS   /* precedence for unary minus */

%%             /* beginning of rules section */

lines    :  /* empty */
         |  lines line
         ;
line     :  dexp '\n'
         {
                (void) printf("%15.8f\n",$1);
         }
         |  vexp '\n'
         {
                (void) printf("(%15.8f, %15.8f)\n",
                   $1.lo, $1.hi);

         }
         |  DREG '=' dexp '\n'
         {
              dreg[$1] = $3;
         }
         |  VREG '=' vexp '\n'
         {
              vreg[$1] = $3;
         }
         |  error '\n'
         {
                yyerrok;
         }
         ;

dexp     :  CONST
         |  DREG
         {
               $$ = dreg[$1];
```

- CONTINUED -

```
                }
              | dexp '+' dexp
                {
                        $$ = $1 + $3;
                }
              | dexp '-' dexp
                {
                        $$ = $1 - $3;
                }
              | dexp '*' dexp
                {
                        $$ = $1 * $3;
                }
              | dexp '/' dexp
                {
                        $$ = $1 / $3;
                }
              | '-' dexp    %prec UMINUS
                {
                       $$ = -$2;
                }
              | '(' dexp')'
                {
                        $$ = $2;
                }
              ;

vexp          : dexp
                {
                     $$.hi = $$.lo = $1;
                }
              | '(' dexp ',' dexp ')'
                {
                     $$.lo = $2;
                     $$.hi = $4;
                     if( $$.lo > $$.hi )
```

```
                         - CONTINUED -
                  {
                         (void) printf("interval out of
                           order \n");
                         YYERROR;
                  }
          }
          |  VREG
          {
                      $$ = vreg[$1];
          }
          |  vexp '+' vexp
          {
                  $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo;
          }
          |  dexp '+' vexp
          {
                  $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo;
          }
          |  vexp '-' vexp
          {
                  $$.hi = $1.hi - $3.lo;
                  $$.lo = $1.lo - $3.hi;
          }
          |  dvep '-' vdep
          {
                  $$.hi = $1 - $3.lo;
                  $$.lo = $1 - $3.hi
          }
          |  vexp '*' vexp
          {
                  $$ = vmul( $1.lo,$.hi,$3 )
          }
          |  dexp '*' vexp
```

- CONTINUED -

```
        {
                $$ = vmul( $1, $1, $3 )
        }
        |  vexp '/' vexp
        {
                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 )
        }
        |  dexp '/' vexp
        {
                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 )
        }
        |  '-' vexp    %prec UMINUS
        {
                $$.hi = -$2.lo;$$.lo = -$2.hi
        }
        |  '(' vexp ')'
        {
                $$ = $2
        }
        ;

%%                        /* beginning of subroutines section*/

# define BSZ 50   /* buffer size for floating point number*/

        /* lexical analysis */

int yylex( )
{
        register int c;

                                /* skip over blanks */
        while ((c = getchar()) == ' ')
                ;
        if (isupper(c))
```

- CONTINUED -

```
      {
          yylval.ival = c - 'A'
          return (VREG);
      }
      if (islower(c))
      {
          yylval.ival = c - 'a',
          return( DREG );
      }

          /* gobble up digits. points, exponents */

      if (isdigit(c) || c == '.')
      {
          char buf[BSZ+1], *cp = buf;
          int dot = 0, exp = 0;

          for(; (cp - buf) < BSZ ; ++cp, c = getchar())
              {
                  *cp = c;
                   if (isdigit(c))
                     continue;
                   if (c == '.')
                  {
                   if (dot++ || exp)
                     return ('.');   /* will cause syntax
                                     error */
                     continue;
                  }
                  if( c == 'e')
                  {
                   if (exp++)
                     return ('e');   /* will cause syntax
                                     error */
                   continue;
                  }
                        /* end of number */
                  break;
              }
```

```
                         - CONTINUED -
                    *cp = ' ';
                    if (cp - buf >= BSZ)
                       (void) printf("constant too long -
                           truncated\n");
                    else
                       ungetc(c, stdin);   /* push back last
                           char read */
                    yylval.dval = atof(buf);
                    return (CONST);
           }
           return (c);
}
INTERVAL
hilo(a, b, c, d)
           double a, b, c, d;
{
             /* returns the smallest interval containing a,
                  b, c, and d */

             /* used by *,/ routine */
           INTERVAL v;

           if (a > b)
           {
                   v.hi = a;
                   v.lo = b;
           }
           else
           {
                   v.hi = b;
                   v.lo = a;
           }
           if (c > d)
           {
                   if (c > v.hi)
                       v.hi = c;
                   if (d < v.lo)
                       v.lo = d;
           }
```

- CONTINUED -

```
        else
        }
                if (d > v.hi)
                    v.hi = d;
                if (c < v.lo)
                    v.lo = c;
        }
        return (v);
}
INTERVAL
vmul(a, b, v)
        double a, b;
        INTERVAL v;
{
        return (hilo(a * v.hi, a * v,lo, b * v.hi,
               b * v.lo));
}
dcheck(v)
        INTERVAL v;
{
        if (v.hi >= 0. && v.lo <= 0.)
        {
             (void) printf("divisor interval contains
               0.\n");
             return (1);
        }
        return (0);
{

INTERVAL
vdiv(a, b, v)
        double a, b;
        INTERVAL v;
{
   return (hilo(a / v.hi, a / v,lo, b / v.hi, b / v.lo));
}
```

Chapter 7: File and Record Locking

Introduction

Mandatory and advisory file and record locking both are available on current releases of the UNIX system. The intent of this capability to is provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multi-user applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like */usr/group*, an organization of UNIX system users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self-synchronizing processes. In mandatory locking, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double checks the programs against accessing the data out of sequence.

The remainder of this chapter describes how file and record locking capabilities can be used. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The manual pages for the **fcntl**(2) system call, the **lockf**(3) library function, and **fcntl**(5) data structures and commands are referred to throughout this section. You should read them before continuing.

Terminology

Before discussing how record locking should be used, let us first define a few terms.

Record
: A contiguous set of bytes in a file. The UNIX operating system does not impose any record structure on files. This may be done by the programs that use the files.

Cooperating Processes
: Processes that work together in some well defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non-cooperating processes from accessing those files. The term process will be used interchangeably with cooperating process to refer to a task obeying such protocols.

Read (Share) Locks
: These are used to gain limited access to sections of files. When a read lock is in place on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. If a process holds a read lock it may assume that no other process will be writing or updating that record at the same time. This access method also permits many processes to read the given record. This might be necessary when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Locks
: These are used to gain complete control over sections of files. When a write lock is in place on a record, no other process may read or write lock that record, in whole or in part. If a process holds a write lock it may assume that no other process will be reading or writing that record at the same time.

Advisory Locking

A form of record locking that does not interact with the I/O subsystem (i.e. **creat**(2), **open**(2), **read**(2), and **write**(2)). The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file will be controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking

A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the **creat**(2), **open**(2), **read**(2), and **write**(2) system calls. If a record is locked, then access of that record by any other process is restricted according to the type of lock on the record. The control over records should still be performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

File Protection

There are access permissions for UNIX system files to control who may read, write, or execute such a file. These access permissions may only be set by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the ultimate disposition of a file. Note that if the directory permissions allow anyone to write in it, then files within the directory may be removed, even if those files do not have read, write or execute permission for that user. Any information that is worth protecting, is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, will only protect the portions of the files that are locked. Other parts of these files might be corrupted if proper precautions are not taken.

Only a known set of programs and/or administrators should be able to read or write a data base. This can be done easily by setting the set-group-ID bit (see **chmod**(1)) of the data base accessing programs. The files can then be accessed by a known set of programs that obey the record locking protocol. An example of such file protection, although record locking is not used, is the **mail**(1) command. In that command only the particular user and the **mail** command can read and write in the unread mail files.

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be done, then the file must be opened with at least read accessibility and likewise for write locks and write accessibility. For our example we will open our file for both read and write access:

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


int fd; /* file descriptor */
char *filename;

main(argc, argv)
int argc;
char *argv[];
{
       extern void exit(), perror();

       /* get data base file name from command line and
        * open the file for read and write access.
        */
       if (argc < 2) {
       (void) fprintf(stderr, "usage: %s filename\n",
          argv[0]);
       exit(2);
       }
       filename = argv[1];
       fd = open(filename, O_RDWR);
       if (fd < 0) {
       perror(filename);
       exit(2);
       }
       .
       .
       .
```

The file is now open for us to perform both locking and I/O functions. We then proceed with the task of setting a lock.

Setting a File Lock

There are several ways for us to set a lock on a file. In part, these methods depend upon how the lock interacts with the rest of the program. There are also questions of performance as well as portability. Two methods will be given here, one using the **fcntl**(2) system call, the other using the */usr/group* standards compatible **lockf**(3) library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero (0) until the end of the maximum file size. This point extends beyond any real end of the file so that no lock can be placed on this file beyond this point. To do this the value of the size of the lock is set to zero. The code using the **fcntl**(2) system call is as follows:

```
#include <fcntl.h>
#define MAX_TRY10
int try;
struct flock lck;

try = 0;

/* set up the record locking structure, the address
 * of which is passed to the fcntl system call.
 */
lck.l_type = F_WRLCK;/* setting a write lock */
lck.l_whence = 0;/* offset l_start from beginning of
   file */
lck.l_start = 0L;
lck.l_len = 0L;/*until end of file address space*/

/* Attempt locking MAX_TRY times before giving up.
 */
```

- CONTINUED -

```
while (fcntl(fd, F_SETLK, &lck) < 0) {
if (errno == EAGAIN || errno == EACCES) {
/* there might be other errors cases in which
 * you might try again.
 */
if (++try < MAX_TRY) {
(void) sleep(2);
continue;
}
(void) fprintf(stderr,"File busy try again
   later!\n");
return;
}
perror("fcntl");
exit(2);
}
.
.
.
```

This portion of code tries to lock a file. This is attempted several times until one of the following things happens:

- the file is locked
- an error occurs
- it gives up trying because MAX_TRY has been exceeded

To perform the same task using the **lockf**(3) function, the code is as follows:

```
#include <unistd.h>
#define MAX_TRY10
int try;
try = 0;

/* make sure the file pointer
 * is at the beginning of the file.
 */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up.
 */
while (lockf(fd, F_TLOCK, 0L) < 0) {
if (errno == EAGAIN || errno == EACCES) {
/* there might be other errors cases in which
 * you might try again.
 */
if (++try < MAX_TRY) {
sleep(2);
continue;
}
(void) fprintf(stderr,"File busy try again
   later!\n");
return;
}
perror("lockf");
exit(2);
}
.
.
.
```

It should be noted that the **lockf**(3) example appears to be simpler, but the **fcntl**(2) example exhibits additional flexibility. Using the **fcntl**(2) method, it is possible to set the type and start of the lock request simply by setting a few structure variables. **lockf**(3) merely sets write (exclusive) locks; an additional system call (**lseek**(2)) is required to specify the start of the lock.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We will now try to solve an interesting and real problem. There are two records (these records may be in the same or different file) that must be updated simultaneously so that other processes get a consistent view of this information. (This type of problem comes up, for example, when updating the interrecord pointers in a doubly linked list.) To do this you must decide the following questions:

- What do you want to lock?
- For multiple locks, what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy if one cannot obtain all the required locks. It is because of contention for these records that we have decided to use record locking in the first place. Different programs might:

- wait a certain amount of time, and try again
- abort the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of the above

Let us now look at our example of inserting an entry into a doubly linked list. For the example, we will assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If there are processes with pending write locks that are sleeping on the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is merely reset with the new lock type. Because the */usr/group* **lockf** function does not have read locks, lock promotion is not applicable to that call. An example of record locking with lock promotion follows:

```
struct record {
        .
        ./* data portion of record */
        .
        long prev;/* index to previous record in the list */
        long next;/* index to next record in the list */
};

/* Lock promotion using fcntl(2)
 * When this routine is entered it is assumed that there
 * are read locks on "here" and "next".
 * If write locks on "here" and "next" are obtained:
 *    Set a write lock on "this".
 *    Return index to "this" record.
 * If any write lock is not obtained:
 *    Restore read locks on "here" and "next".
 *    Remove all other locks.
 *    Return a -1.
 */
long
set3lock (this, here, next)
long this, here, next;
{
```

```
                        - CONTINUED -
      struct flock lck;

      lck.l_type = F_WRLCK;/* setting a write lock */
      lck.l_whence = 0;/* offset l_start from beginning
         of file */
      lck.l_start = here;
      lck.l_len = sizeof(struct record);

      /* promote lock on "here" to write lock */
      if (fcntl(fd, F_SETLKW, &lck) < 0) {
      return (-1);
      }
      /* lock "this" with write lock */
      lck.l_start = this;
      if (fcntl(fd, F_SETLKW, &lck) < 0) {
      /* Lock on "this" failed;
       * demote lock on "here" to read lock.
       */
      lck.l_type = F_RDLCK;
      lck.l_start = here;
      (void) fcntl(fd, F_SETLKW, &lck);
      return (-1);
      }
      /* promote lock on "next" to write lock */
      lck.l_start = next;
      if (fcntl(fd, F_SETLKW, &lck) < 0) {
      /* Lock on "next" failed;
       * demote lock on "here" to read lock,
       */
      lck.l_type = F_RDLCK;
      lck.l_start = here;
      (void) fcntl(fd, F_SETLK, &lck);
      /* and remove lock on "this".
       */
      lck.l_type = F_UNLCK;
      lck.l_start = this;
      (void) fcntl(fd, F_SETLK, &lck);
      return (-1);/* cannot set lock, try again or quit */
      }
      return (this);
}
```

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the F_SETLKW command. If the F_SETLK command was used instead, the **fcntl** system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

Let us now look at a similar example using the **lockf** function. Since there are no read locks, all (write) locks will be referenced generically as locks.

```
/* Lock promotion using lockf(3)
 * When this routine is entered it is assumed that there
 * are no locks on "here" and "next".
 * If locks are obtained:
 *    Set a lock on "this".
 *    Return index to "this" record.
 * If any lock is not obtained:
 *    Remove all other locks.
 *    Return a -1.
 */

#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;

{

        /* lock "here" */
        (void) lseek(fd, here, 0);
        if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        return (-1);
        }
        /* lock "this" */
        (void) lseek(fd, this, 0);
        if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
```

- CONTINUED -

```
        /* Lock on "this" failed.
         * Clear lock on "here".
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);
        }
        /* lock "next" */
        (void) lseek(fd, next, 0);
        if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {

        /* Lock on "next" failed.
         * Clear lock on "here",
         */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));

        /* and remove lock on "this".
         */
        (void) lseek(fd, this, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);/* cannot set lock, try again
           or quit */

        }

        return (this);
}
```

Locks are removed in the same manner as they are set, only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process and will only affect locks that were placed by this process. The unlock only affects the section of the file defined in the previous example by **lck**. It is possible to unlock or change the type of lock on a subsection of a previously set lock. This may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Getting Lock Information

One can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. A lock is set up as in the previous examples and the F_GETLK command is used in the **fcntl** call. If the lock passed to **fcntl** would be blocked, the first blocking lock is returned to the process through the structure passed to **fcntl**. That is, the lock data passed to **fcntl** is overwritten by blocking lock information. This information includes two pieces of data that have not been discussed yet, **l_pid** and **l_sysid**, that are only used by F_GETLK. (For systems that do not support a distributed architecture the value in **l_sysid** should be ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to **fcntl** using the F_GETLK command would not be blocked by another process' lock, then the **l_type** field is changed to F_UNLCK and the remaining fields in the structure are unaffected. Let us use this capability to print all the segments locked by other processes. Note that if there are several read locks over the same segment only one of these will be found.

```
struct flock lck;

/* Find and print "write lock" blocked segments of
       this file. */
       (void) printf("sysid pid type start length\n");
       lck.l_whence = 0;
       lck.l_start = 0L;
       lck.l_len = 0L;
       do {
       lck.l_type = F_WRLCK;
       (void) fcntl(fd, F_GETLK, &lck);
       if (lck.l_type != F_UNLCK) {
       (void) printf("%5d %5d  %c  %8d %8d\n",
       lck.l_sysid,
       lck.l_pid,
       (lck.l_type == F_WRLCK) ? 'W' : 'R',
       lck.l_start,
       lck.l_len);
       /* if this lock goes to the end of the address
        * space, no need to look further, so break out.
        */
       if (lck.l_len == 0)
       break;
       /* otherwise, look for new lock after the one
        * just found.
        */
       lck.l_start += lck.l_len;
       }
       } while (lck.l_type != F_UNLCK);
```

fcntl with the F_GETLK command will always return correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The **lockf** function with the F_TEST command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. A routine using **lockf** to test for a lock on a file follows:

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0)
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 0) {
switch (errno) {
case EACCES:
case EAGAIN:
(void) printf("file is locked by another process\n");
break;
case EBADF:
/* bad argument passed to lockf */
perror("lockf");
break;
default:
(void) printf("lockf: unknown error <%d>\n", errno);
break;
}
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by **l_start**, when using a **l_whence** value of 1. If both the parent and child process set locks on the same file, there is a possibility that a lock will be set using a file pointer that was reset by the other process. This problem appears in the **lockf**(3) function call as well and is a result of the */usr/group* requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This will result in the creation of a new and separate file pointer that can be manipulated without this problem occurring.

Another solution is to use the **fcntl** system call with a **l_whence** value of 0 or 2. This makes the locking function atomic, so that even processes sharing file pointers can be locked without difficulty.

Deadlock Handling

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf** call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can only potentially occur when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call will fail and set **errno** to the deadlock error number. If a process wishes to avoid the use of the systems deadlock detection it should set its locks using F_GETLK instead of F_GETLKW.

Selecting Advisory or Mandatory Locking

The use of mandatory locking is not recommended for reasons that will be made clear in a subsequent section. Whether or not locks are enforced by the I/O system calls is determined at the time the calls are made and the state of the permissions on the file (see **chmod**(2)). For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Mandatory enforcement can be assured by the following code:

```
#include <sys/types.h>
#include <sys/stat.h>

int mode;
struct stat buf;
        .
        .
        .
        if (stat(filename, &buf) < 0) {
        perror("program");
        exit (2);
        }
        /* get currently set mode */
        mode = buf.st_mode;
        /* remove group execute permission from mode */
        mode &= ~(S_IEXEC>>3);
        /* set 'set group id bit' in mode */
        mode |= S_ISGID;
        if (chmod(filename, mode) < 0) {
        perror("program");
        exit(2);
        }
        .
        .
        .
```

Files that are to be record locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

The **chmod**(1) command can also be easily used to set a file to have mandatory locking. This can be done with the command:

chmod +l *filename*

The **ls**(1) command was also changed to show this setting when you ask for the long listing format:

ls -l *filename*

causes the following to be printed:

```
-rw---l---  1 abc other 1048576 Dec 3 11:44 filename
```

Caveat Emptor – Mandatory Locking

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all such pieces before any I/O begins. Thus advisory enforcement is sufficient for all programs that perform in this way.
- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking and Future Releases of the UNIX System

Provisions have been made for file and record locking in a UNIX system environment. In such an environment the system on which the locking process resides may be remote from the system on which the file and record locks reside. In this way multiple processes on different systems may put locks upon a single file that resides on one of these or yet another system. The record locks for a file reside on the system that maintains the file. It is also important to note that deadlock detection/avoidance is only determined by the record locks being held by and for a single system. Therefore, it is necessary that a process only hold record locks on a single system at any given time for the deadlock mechanism to be effective. If a process needs to maintain locks over several systems, it is suggested that the process avoid the **sleep-when-blocked** features of **fcntl** or **lockf** and that the process maintain its own deadlock detection. If the process uses the **sleep-when-blocked** feature, then a timeout mechanism should be provided by the process so that it does not hang waiting for a lock to be cleared.

Chapter 8: Shared Libraries

Introduction

With the UNIX system running on smaller machines, such as the AT&T 3B2 Computer, efficient use of disk storage space, memory, and computer power is becoming increasingly important. A shared library can offer savings in all three areas. For example, if constructed properly, a shared library can make **a.out** files (executable object files) smaller on disk storage and processes (**a.out** files that are executing) smaller in memory.

The first part of this chapter, "Using a Shared Library," is designed to help you use UNIX System V shared libraries. It describes what a shared library is and how to use one to build **a.out** files. It also offers advice about when and when not to use a shared library and how to determine whether an **a.out** uses a shared library.

The second part in this chapter, "Building a Shared Library," describes how to build a shared library. You do not need to read this part to use shared libraries. It addresses library developers, advanced programmers who are expected to build their own shared libraries. Specifically, this part describes how to use the UNIX system tool **mkshlib**(1) (documented in the *Programmer's Reference Manual*) and how to write C code for shared libraries on a UNIX system. An example is included. Read this part of the chapter only if you have to build a shared library.

NOTE: Shared libraries are a new feature of UNIX System V Release 3.0. An executable object file that needs shared libraries will not run on previous releases of UNIX System V.

Using a Shared Library

If you are accustomed to using libraries to build your applications programs, shared libraries should blend into your work easily. This part of the chapter explains what shared libraries are and how and when to use them on the UNIX system.

What is a Shared Library?

A shared library is a file containing object code that several **a.out** files may use simultaneously while executing. A shared library, like a library that is not shared, is an archive file. For simplicity, however, we refer to an archive file with shared library members as a shared library and one without as an archive library.

When a program is compiled or link edited with a shared library, the library code that defines the program's external references is not copied into the program's object file. Instead, a special section called **.lib** that identifies the library code is created in the object file. When the UNIX system executes the resulting **a.out** file, it uses the information in this section to bring the required shared library code into the address space of the process.

A shared library offers several benefits by not copying code into **a.out** files. It can

- save disk storage space

  Because shared library code is not copied into all the **a.out** files that use the code, these files are smaller and use less disk space.

- save memory

  By sharing library code at run time, the dynamic memory needs of processes are reduced.

- make executable files using library code easier to maintain

  As mentioned above, shared library code is brought into a process' address space at run time. Updating a shared library effectively updates all executable files that use the library, because the operating system brings the updated version into new processes. If an error in shared library code is fixed, all processes automatically use the corrected code.

Archive libraries cannot, of course, offer this benefit: changes to archive libraries do not affect executable files, because code from the libraries is copied to the files during link editing, not during execution.

"Deciding Whether to Use a Shared Library" in this chapter describes shared libraries in more detail.

The UNIX System Shared Libraries

AT&T provides the C shared library with UNIX System V Release 3.0, and later releases; the networking library included with the Networking Support Utilities is also a shared library. Other shared libraries may be available now from software vendors and in the future from AT&T.

These libraries, like all shared libraries, are made up of two files called the host library and the target library. The host library is the file that the link editor searches when linking programs to create the **.lib** sections in **a.out** files; the target library is the file that the UNIX system uses when running those files. Naturally, the target library must be present for the **a.out** file to run.

Shared Library	Host Library Command Line Option	Target Library Path Name
C Library	**–lc_s**	**/shlib/libc_s**
Networking Library	**–lnsl_s**	**/shlib/libnsl_s**

Notice the **_s** suffix on the library names; we use it to identify both host and target shared libraries. For example, it distinguishes the standard relocatable C library **libc** from the shared C library **libc_s**. The **_s** also indicates that the libraries are statically linked.

The relocatable C library is still available on the UNIX system; this library is searched by default during the compilation or link editing of C programs. All other archive libraries from previous releases of the system are also available. Just as you use the archive libraries' names, you must use a shared library's name when you want to use it to build your **a.out** files. You tell the link editor its name with the **-l** option, as shown below.

Building an a.out File

You direct the link editor to search a shared library the same way you direct a search of an archive library on the UNIX system:

```
cc   file.c   -o file   ...   -llibrary_file   ...
```

To direct a search of the networking library, for example, you use the following command line.

```
cc   file.c   -o file   ...   -lnsl_s  ...
```

And to link all the files in your current directory together with the shared C library you'd use the following command line:

```
cc   *.c   -lc_s
```

Normally, you should include the **-lc_s** argument after all other **-l** arguments on a command line. The shared C library will then be treated like the relocatable C library, which is searched by default after all other libraries specified on a command line are searched.

Coding an Application

Application source code in C or assembly language is compatible with both archive libraries and shared libraries. As a result, you should not have to change the code in any applications you already have when you use a shared library with them. When coding a new application for use with a shared library, you should just observe your standard coding conventions.

However, do keep the following two points in mind, which apply when using either an archive or a shared library:

- Don't define symbols in your application with the same names as those in a library.

  Although there are exceptions, you should avoid redefining standard library routines, such as **printf**(3S) and

strcmp(3C). Replacements that are incompatibly defined can cause any library, shared or unshared, to behave incorrectly.

- Don't use undocumented archive routines.

  Use only the functions and data mentioned on the manual pages describing the routines in Section 3 of the *Programmer's Reference Manual*. For example, don't try to outsmart the **ctype** design by manipulating the underlying implementation.

Deciding Whether to Use a Shared Library

You should base your decision to use a shared library on whether it saves space in disk storage and memory for your program. A well-designed shared library almost always saves space. So, as a general rule, use a shared library when it is available.

To determine what savings are gained from using a shared library, you might build the same application with both an archive and a shared library, assuming both kinds of library are available. Remember, that you may do this because source code is compatible between shared libraries and archive libraries. (See the above section "Coding an Application.") Then compare the two versions of the application for size and performance. For example,

```
$ cat hello.c
main()
{
   printf("Hello\n");
}
$ cc -o unshared hello.c
$ cc -o shared hello.c -lc_s
$ size unshared shared
unshared: 8680 + 1388 + 2248 = 12316
shared: 300 + 680 + 2248 = 3228
```

If the application calls only a few library members, it is possible that using a shared library could take more disk storage or memory. The following section gives a more detailed discussion about when a shared library does and does not save space.

When making your decision about using shared libraries, also remember that they are not available on UNIX System V releases prior to Release 3.0. If your program must run on previous releases, you will need to use archive libraries.

More About Saving Space

This section is designed to help you better understand why your programs will usually benefit from using a shared library. It explains

- how shared libraries save space that archive libraries cannot
- how shared libraries are implemented on the UNIX system
- how shared libraries might increase space usage

How Shared Libraries Save Space

To better understand how a shared library saves space, we need to compare it to an archive library.

A host shared library resembles an archive library in three ways. First, as noted earlier, both are archive files. Second, the object code in the library typically defines commonly used text symbols and data symbols. The symbols defined inside and made available outside the library are called exported symbols. Note that the library may also have imported symbols, symbols that it uses but usually does not define. Third, the link editor searches the library for these symbols when linking a program to resolve its external references. By resolving the references, the link editor produces an executable version of the program, the **a.out** file.

NOTE: Note that the link editor on the UNIX system is a static linking tool; static linking requires that all symbolic references in a program be resolved before the program may be executed. The link editor uses static linking with both an archive library and a shared library.

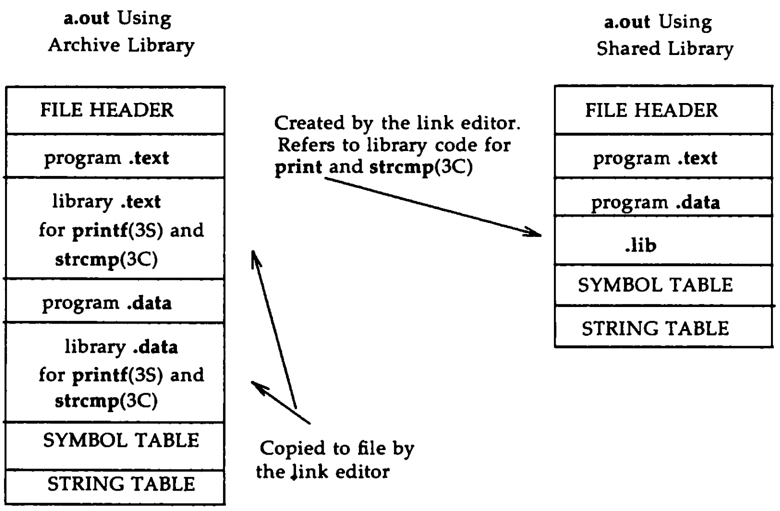
Although these similarities exist, a shared library differs significantly from an archive library. The major differences relate to how the libraries are handled to resolve symbolic references, a topic already discussed briefly.

Consider how the UNIX system handles both types of libraries during link editing. To produce an **a.out** file using an archive library, the link editor copies the library code that defines a program's unresolved external reference from the library into appropriate **.text** and **.data** sections in the program's object file. In contrast, to produce an **a.out** file using a shared library, the link editor does not copy any code from the library into the program's object file. Instead, it creates a special section called **.lib** in the file that identifies the library code needed at run time and resolves the external references to shared library symbols with their correct values. When the UNIX system executes the resulting **a.out** file, it uses the information in the **.lib** section to bring the required shared library code into the address space of the process.

Figure 8-1 depicts the **a.out** files produced using a regular archive version and a shared version of the standard C library to compile the following program:

```
main()
{
   ...
   printf( "How do you like this manual?\n" );
   ...
   result = strcmp( "I do.", answer );
   ...
}
```

Notice that the shared version is smaller. Figure 8-2 depicts the process images in memory of these two files when they are executed.



a.out Using Archive Library
FILE HEADER
program **.text**
library **.text** for **printf**(3S) and **strcmp**(3C)
program **.data**
library **.data** for **printf**(3S) and **strcmp**(3C)
SYMBOL TABLE
STRING TABLE
Created by the link editor. Refers to library code for **print** and **strcmp**(3C)
Copied to file by the link editor
a.out Using Shared Library
FILE HEADER
program **.text**
program **.data**
.lib
SYMBOL TABLE
STRING TABLE

Figure 8-1: **a.out** Files Created Using an Archive Library and a Shared Library

Now consider what happens when several a.out files need the same code from a library. When using an archive library, each file gets its own copy of the code. This results in duplication of the same code on the disk and in memory when the **a.out** files are run as processes. In contrast, when a shared library is used, the library code remains separate from the code in the **a.out** files, as indicated in Figure 8-2. This separation enables all processes using the same shared library to reference a single copy of the code.

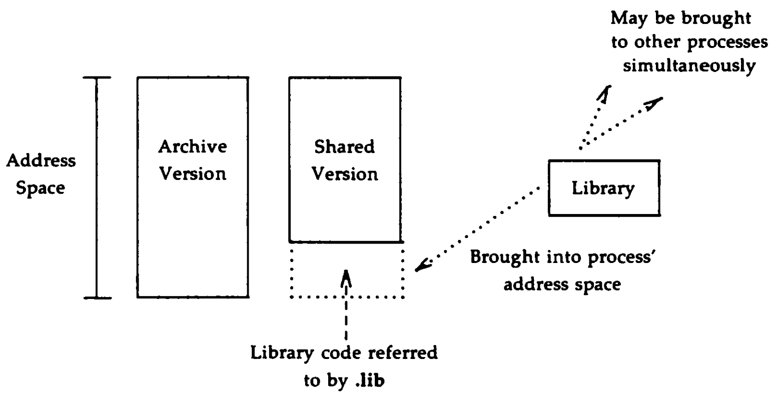

May be brought to other processes simultaneously
Address Space
Archive Version
Shared Version
Library
Brought into process' address space
Library code referred to by .lib

Figure 8-2: Processes Using an Archive and a Shared Library

How Shared Libraries Are Implemented

Now that you have a better understanding of how shared libraries save space, you need to consider their implementation on the UNIX system to understand how they might increase space usage (this happens seldomly).

The Host Library and Target Library

As previously mentioned, every shared library has two parts: the host library used for linking that resides on the host machine and the target library used for execution that resides on the target machine. The host machine is the machine on which you build an **a.out** file; the target machine is the machine on which you run the file. Of course, the host and target may be the same machine, but they don't have to be.

The host library is just like an archive library. Each of its members (typically a complete object file) defines some text and data symbols in its symbol table. The link editor searches this file when a shared library is used during the compilation or link editing of a program.

The search is for definitions of symbols referenced in the program but not defined there. However, as mentioned earlier, the link editor does not copy the library code defining the symbols into the program's object file. Instead, it uses the library members to locate the definitions and then places symbols in the file that tell where the library code is. The result is the special section in the **a.out** file mentioned earlier (see the section "What is a Shared Library?") and shown in Figure 8-1 as **.lib**.

The target library used for execution resembles an **a.out** file. The UNIX operating system reads this file during execution if a process needs a shared library. The special **.lib** section in the **a.out** file tells which shared libraries are needed. When the UNIX system executes the **a.out** file, it uses this section to bring the appropriate library code into the address space of the process. In this way, before the process starts to run, all required library code has been made available.

Shared libraries enable the sharing of **.text** sections in the target library, which is where text symbols are defined. Although processes that use the shared library have their own virtual address spaces, they share a single physical copy of the library's text among them. That is, the UNIX system uses the same physical code for each process that attaches a shared library's text.

The target library cannot share its **.data** sections. Each process using data from the library has its own private data region (contiguous area of virtual address space that mirrors the **.data** section of the target library). Processes that share text do not share data and stack area so that they do not interfere with one another.

As suggested above, the target library is a lot like an **a.out** file, which can also share its text, but not its data. Also, a process must have execute permission for a target library to execute an **a.out** file that uses the library.

The Branch Table

When the link editor resolves an external reference in a program, it gets the address of the referenced symbol from the host library. This is because a static linking loader like **ld** binds symbols to addresses during link editing. In this way, the **a.out** file for the program has an address for each referenced symbol.

What happens if library code is updated and the address of a symbol changes? Nothing happens to an **a.out** file built with an archive library, because that file already has a copy of the code defining the symbol. (Even though it isn't the updated copy, the **a.out** file will still run.) However, the change can adversely affect an **a.out** file built with a shared library. This file has only a symbol telling where the required library code is. If the library code were updated, the location of that code might change. Therefore, if the **a.out** file ran after the change took place, the operating system could bring in the wrong code. To keep the **a.out** file current, you might have to recompile a program that uses a shared library after each library update.

To prevent the need to recompile, a shared library is implemented with a branch table on the UNIX system. A branch table associates text symbols with an absolute address that does not change even when library code is changed. Each address labels a jump instruction to the address of the code that defines a symbol. Instead of being directly associated with the addresses of code, text symbols have addresses in the branch table.

Figure 8-3 shows two **a.out** files executing that make a call to **printf**(3S). The process on the left was built using an archive library. It already has a copy of the library code defining the **printf**(3S) symbol. The process on the right was built using a shared library. This file references an absolute address (10) in the branch table of the shared library at run time; at this address, a jump instruction references the needed code.

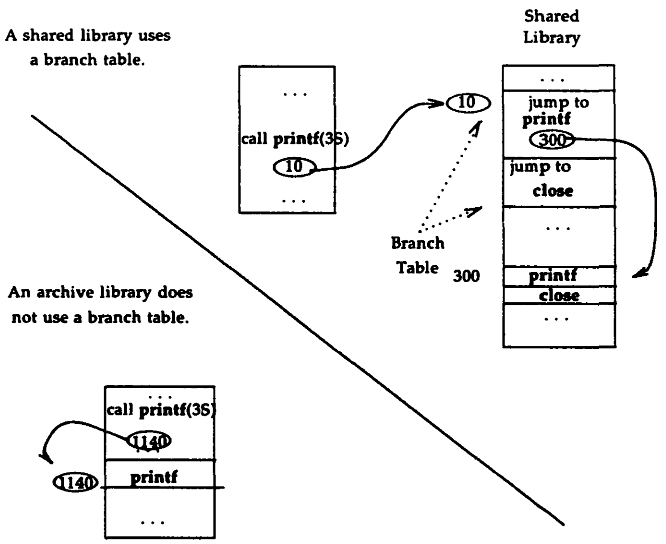


Figure 8-3: A Branch Table in a Shared Library

How Shared Libraries Might Increase Space Usage

A host library might add space to an **a.out** file. Recall that UNIX System V Release 3.0 uses static linking, which requires that all external references in a program be resolved before it is executed. Also recall that a shared library may have imported symbols, which are used but not defined by the library. These symbols might introduce unresolved references during the linking process. To resolve these references, the link editor has to add the **.text** and **.data** sections defining the referenced imported symbols to the **a.out** file. These sections increase the size of the **a.out** file.

A target library might also add space to a process. Again recall from "How Shared Libraries are Implemented" in this chapter that a shared library's target file may have both text and data regions connected to a process. While the text region is shared by all processes that use the library, the data region is not. Every process that uses the library gets its own private copy of the entire library data region. Naturally, this region adds to the process's memory requirements. As a result, if an application uses only a small part of a shared library's text and data, executing the application might require more memory with a shared library than without one. For example, it would be unwise to use the shared C library to access only **strcmp**(3C). Although sharing **strcmp**(3C) saves disk storage and memory, the memory cost for sharing all the shared C library's private data region outweighs the savings. The archive version of the library would be more appropriate.

Identifying a.out Files that Use Shared Libraries

Suppose you have an executable file and you want to know whether it uses a shared library. You can use the **dump**(1) command (documented in the *Programmer's Reference Manual*) to look at the section headers for the file:

```
dump  -hv  a.out
```

If the file has a **.lib** section, a shared library is needed. If the **a.out** does not have a **.lib** section, it does not use shared libraries.

With a little more work, you can even tell what libraries a file uses by looking at the **.lib** section contents.

```
dump  -L  a.out
```

Debugging a.out Files that Use Shared Libraries

Debugging support for shared libraries is currently limited. Shared library data are not dumped to core files, and **sdb**(1) (documented in the *Programmer's Reference Manual*) does not read shared libraries' symbol tables. If you encounter an error that appears not to be in your application code, you may find debugging easier if you rebuild the application with the archive version of the library used.

Building a Shared Library

This part of the chapter explains how to build a shared library. It covers the major steps to the building process, the use of the UNIX system tool **mkshlib**(1) which builds the host and target libraries, and some guidelines for writing shared library code.

This part assumes that you are an advanced C programmer faced with the task of building a shared library. It also assumes you are familiar with the archive library building process. You do not need to read this part of the chapter if you only plan to use the UNIX system shared libraries or other shared libraries that have already been built.

The Building Process

To build a shared library on the UNIX system, you have to complete six major tasks:

- Choose region addresses.
- Choose the path name for the shared library target file.
- Select the library contents.
- Rewrite existing library code to be included in the shared library.
- Write the library specification file.
- Use the **mkshlib** tool to build the host and target libraries.

Here each of these tasks is discussed.

Step 1: Choosing Region Addresses

The first thing you need to do is choose region addresses for your shared library.

Shared library regions on the AT&T 3B2 Computer correspond to memory management unit (MMU) segment size, each of which is 128 KB. The following table gives a list of the segment assignments on the 3B2 Computer (as of the copyright date for this guide) and shows what virtual addresses are available for libraries you might build.

Start Address	Contents	Target Path Name
0x80000000 ... 0x803E0000	Reserved for AT&T UNIX Shared C Library AT&T Networking Library	 **/shlib/libc_s** **/shlib/libnsl_s**
0x80400000 0x80420000	Generic Database Library	Unassigned
0x80440000 0x80460000	Generic Statistical Library	Unassigned
0x80480000 0x804A0000	Generic User Interface Library	Unassigned
0x804C0000 0x804E0000	Generic Screen Handling Library	Unassigned
0x80500000 0x80520000	Generic Graphics Library	Unassigned
0x80540000 0x80560000	Generic Networking Library	Unassigned
0x80580000 ... 0x80660000	Generic - to be defined	Unassigned
0x80680000 ... 0x807E0000	For private use	Unassigned

What does this table tell you? First, the AT&T shared C library and the networking library reside at the path names given above and use addresses in the range reserved for AT&T. If you build a shared library that uses reserved addresses you run the risk of conflicting with future AT&T products.

Second, a number of segments are allocated for shared libraries that provide various services such as graphics, database access, and so on. These categories are intended to reduce the chance of address conflicts among commercially available libraries. Although two libraries of the same type may conflict, that doesn't matter. A single process should not usually need to use two shared libraries of the same type. If the need arises, a program can use one shared library and one archive library.

NOTE: Any number of libraries can use the same virtual addresses, even on the same machine. Conflicts occur only within a single process, not among separate processes. Thus two shared libraries can have the same region addresses without causing problems, as long as a single **a.out** file doesn't need to use both libraries.

Third, several segments are reserved for private use. If you are building a large system with many **a.out** files and processes, shared libraries might improve its performance. As long as you don't intend to release the shared libraries as separate products, you should use the private region addresses. You can put your shared libraries into these segments and avoid conflicting with commercial shared libraries. You should also use the private areas when you will own all the **a.out** files that access your shared library. Don't risk address conflicts.

NOTE: If you plan to build a commercial shared library, you are strongly encouraged to provide a compatible, relocatable archive as well. Some of your customers might not find the shared library appropriate for their applications. Others might want their applications to run on versions of the UNIX system without shared library support.

Step 2: Choosing the Target Library Path Name

After you choose the region addresses for your shared library, you should choose the path name for the target library. We chose **/shlib/libc_s** for the shared C library and **/shlib/libnsl_s** for the networking library. (As mentioned earlier, we use the **_s** suffix in the path names of all statically linked shared libraries.) To choose a path name for your shared library, consult the established list of names for your computer or see your system administrator. Also keep in mind that shared libraries needed to boot a UNIX system should normally be located in **/shlib**; other application libraries normally reside in **/usr/lib** or in private application directories. Of course, if your shared library is for personal use, you can choose any convenient path name for the target library.

Step 3: Selecting Library Contents

Selecting the contents for your shared library is the most important task in the building process. Some routines are prime candidates for sharing; others are not. For example, it's a good idea to include large, frequently used routines in a shared library but to exclude smaller routines that aren't used as much. What you include will depend on the individual needs of the programmers and other users for whom you are building the library. There are some general guidelines you should follow, however. They are discussed in the section "Choosing Library Members" in this chapter. Also see the guidelines in the following sections "Importing Symbols" and "Tuning the Shared Library Code."

Step 4: Rewriting Existing Library Code

If you choose to include some existing code from an archive library in a shared library, changing some of the code will make the shared code easier to maintain. See the section "Changing Existing Code for the Shared Library" in this chapter.

Step 5: Writing the Library Specification File

After you select and edit all the code for your shared library, you have to build the shared library specification file. The library specification file contains all the information that **mkshlib** needs to build both the host and target libraries. An example specification file is shown in the next section, "An Example." The contents and format of the specification file are given by the following directives (see also the **mkshlib**(1) manual page):

#address *sectname address*
: Specifies the start address, *address*, of section *sectname* for the target file. This directive is typically used to specify the start addresses of the **.text** and **.data** sections.

#target *pathname*
: Specifies the path name, *pathname*, of the target shared library on the target machine. This is the location where the operating system looks for the shared library during execution. Normally, *pathname* will be an absolute path

name, but it does not have to be.

This directive can be specified only once per shared library specification file.

#branch Starts the branch table specifications. The lines following this directive are taken to be branch table specification lines.

Branch table specification lines have the following format:

funcname <white space> *position*

funcname is the name of the symbol given a branch table entry and *position* specifies the position of *funcname*'s branch table entry. *position* may be a single integer or a range of integers of the form *position1-position2*. Each *position* must be greater than or equal to one. The same position cannot be specified more than once, and every position from one to the highest given position must be accounted for.

If a symbol is given more than one branch table entry by associating a range of positions with the symbol or by specifying the same symbol on more than one branch table specification line, then the symbol is defined to have the address of the highest associated branch table entry. All other branch table entries for the symbol can be thought of as empty slots and can be replaced by new entries in future versions of the shared library.

Finally, only functions should be given branch table entries, and those functions must be external.

This directive can be specified only once per shared library specification file.

#objects Specifies the names of the object files constituting the target shared library. The lines following this directive are taken to be the list of input object files in the order they are to be loaded into the target. The list simply consists of each filename followed by white space. This list of objects will be used to build the shared library.

This directive can be specified only once per shared library specification file.

#init *object* Specifies that the object file, *object*, requires initialization code. The lines following this directive are taken to be initialization specification lines.

Initialization specification lines have the following format:

pimport <white space> *import*

pimport is a pointer to the associated imported symbol, *import*, and must be defined in the current specified object file, *object*. The initialization code generated for each line resembles the C assignment statement:

pimport = &*import*;

The assignments set the pointers to default values. All initializations for a particular object file must be given at once and multiple specifications of the same object file are not allowed.

#ident "*string*" *Specifies a string, string*, to be included in the **.comment** section of the target shared library and the **.comment** sections of every member of the host shared library. Only one **#ident** directive is permitted per shared library specification file.

##	Specifies a comment. The rest of the line is ignored.

All directives that are followed by multi-line specifications are valid until the next directive or the end of file.

Step 6: Using mkshlib to Build the Host and Target

The UNIX system command **mkshlib**(1) builds both the host and target libraries. **mkshlib** invokes other tools such as the assembler, **as**(1), and link editor, **ld**(1). Tools are invoked through the use of **execvp** (see **exec**(2)) which searches directories in a user's $PATH environment variable. Also, prefixes to **mkshlib** are parsed in much the same manner as prefixes to the **cc**(1) command and invoked tools are given the prefix, where appropriate. For example, **3b2mkshlib** invokes **3b2ld**. These commands all are documented in the *Programmer's Reference Manual.*

The user input to **mkshlib** consists of the library specification file and command line options. We just discussed the specification file; let's take a look at the options now. The shared library build tool has the following syntax:

mkshlib -s *specfil* **-t** *target* [**-h** *host*] [**-n**] [**-q**] .DE

-s *specfil*	Specifies the shared library specification file, *specfil*. This file contains all the information necessary to build a shared library, as described in Step 5. Its contents include the branch table specifications for the target, the path name in which the target should be installed, the start addresses of text and data for the target, the initialization specifications for the host, and the list of object files to be included in the shared library.
-t *target*	Specifies the name, *target*, of the target shared library produced on the host machine. When *target* is moved to the target machine, it should be installed at the location given in the specification file (see the **#target** directive in the section "Writing the Library Specification File"). If the **-n** option is given, then a new target shared library will not be generated.

-h *host*	Specifies the name of the host shared library, *host*. If this option is not given, then the host shared library will not be produced.
-n	Prevents a new target shared library from being generated. This option is useful when producing only a new host shared library. The **-t** option must still be supplied since a version of the target shared library is needed to build the host shared library.
-q	Suppresses the printing of certain warning messages.

An Example

Follow each of the steps in the library building process to build a small example shared library. While building this library, appropriate guidelines will be displayed amidst text. Note that the example code is contrived to show samples of problem areas, not to do anything useful.

The name of our library will be **libexam**. Assume the original code was a single source file, as shown below.

```
/* Original exam.c */
#include <stdio.h>

extern int      strlen();
extern char     *malloc(), *strcpy();

int     count   = 0;
char    *Error;

char *
excopy(e)
        char    *e;
{
        char    *new;

        ++count;
        if ( (new = malloc(strlen(e)+1)) == 0 )
        {
                Error = "no memory";
                return 0;
        }
        return strcpy(new, e);
}

excount()
{
        fprintf(stderr, "excount %d\n", count);
        return count;
}
```

To begin, let's choose the region address spaces for the library's **.text** and **.data** sections from the segments reserved for private use on the 3B2 Computer; note that the region addresses must be on a segment boundary (128K):

```
.text   0x80680000
.data   0x806a0000
```

Also choose the path name for our target library:

```
/my/directory/libexam_s
```

Now you need to identify the imported symbols in the library code. (See the guidelines in the section about "Importing Symbols": **malloc**, **strcpy**, **strlen**, **fprintf**, and **_iob**.) A header file defines C preprocessor macros for these symbols; note that you don't use **_iob** directly except through the macro **stderr** from <**stdio.h**>. Also notice the **_libexam_** prefixes for the symbols. The pointers for imported symbols are exported and, therefore, might conflict with other symbols. Using the library name as a prefix reduces the chance of a conflict occurring.

```
/* New file import.h */

#define malloc     (*_libexam_malloc)
#define strcpy     (*_libexam_strcpy)
#define strlen     (*_libexam_strlen)
#define fprintf    (*_libexam_fprintf)
#define _iob       (*_libexam__iob)

extern char        *malloc();
extern char        *strcpy();
extern int         strlen();
extern int         fprintf();
```

NOTE: The file **import.h** does not declare _**iob** as **extern**; it relies on the header file <**stdio.h**> for this information.

You will also need a new source file to hold definitions of the imported symbol pointers. Remember that all global data need to be initialized:

```
/* New file import.c */
#include <stdio.h>

char   *(*_libexam_malloc)()    0;
char   *(*_libexam_strcpy)()    0;
int    (*_libexam_strlen)()     0;
int    (*_libexam_fprintf)()    0;
FILE   (*_libexam__iob)[]       0;
```

Next, look at the library's global data to see what needs to be visible externally. (See the guideline "Minimize Global Data.") The variable **count** does not need to be external, because it is accessed through **excount()**. Make it static. (This should have been done for the relocatable version.)

Now the library's global data need to be moved into separate source files. (See the guideline "Define Text and Global Data in Separate Source Files.") The only global datum left is **Error**, and it needs to be initialized. (See the guideline "Initialize Global Data.") **Error** must remain global, because it passes information back to the calling routine:

```
/* New file global.c */

char   *Error    0;
```

Integrating these changes into the original source file, we get the following (notice that the symbol names must be declared as **externs**):

```
/* Modified exam.c */
#include "import.h"

#include <stdio.h>

extern int      strlen();
extern char     *malloc(), *strcpy();

static int      count = 0;
extern char     *Error;

char *
excopy(e)
        char    *e;
{
        char    *new;

        ++count;
        if ( (new = malloc(strlen(e)+1)) == 0 )
        {
               Error = "no memory";
               return 0;
        }
        return strcpy(new, e);
}

excount()
{
        fprintf(stderr, "excount %d\n", count);
        return count;
}
```

NOTE: The new header file **import.h** must be included before **<stdio.h>**.

Next, we must write the shared library specification file for **mkshlib**:

```
        /* New file libexam.sl */
 1        #target /my/directory/libexam_s
 2        #address .text 0x80680000
 3        #address .data 0x806a0000

 4        #branch
 5                excopy          1
 6                excount         2

 7        #objects
 8                import.o
 9                global.o
10                exam.o

11        #init import.o
12                _libexam_malloc  malloc
13                _libexam_strcpy  strcpy
14                _libexam_strlen  strlen
15                _libexam_fprintf fprintf
16                _libexam__iob    _iob
```

Briefly, here is what the specification file does. Line 1 gives the path name of the shared library on the target machine. The target shared library must be installed there for **a.out** files that use it to work correctly. Lines 2 and 3 give the virtual addresses for the shared library text and data regions, respectively. Line 4 through 6 specify the branch table. Lines 5 and 6 assign the functions **excopy()** and **excount()** to branch table entries 1 and 2. Only external text symbols, such as C functions, should be placed in the branch table.

Lines 7 through 10 give the list of object files that will be used to construct the host and target shared libraries. When building the host shared library archive, each file listed here will reside in its own archive member. When building the target library, the order of object files will be preserved. The data files must be first. Otherwise, a change in the size of static data in **exam.o** would move external data symbols and break compatibility.

Lines 11 through 16 give imported symbol information for the object file **import.o**. You can imagine assignments of the symbol values on the right to the symbols on the left. Thus **_libexam_malloc** will hold a pointer to **malloc**, and so on.

Now, we have to compile the **.o** files as we would for any other library:

```
cc -c import.c global.c exam.c
```

Finally, we need to invoke **mkshlib** to build our host and target libraries:

```
mkshlib -s libexam.sl -t libexam_s -h libexam_s.a
```

Presuming all of the source files have been compiled appropriately, the **mkshlib** command line shown above will create both the host library, **libexam_s.a**, and the target library, **libexam_s**.

Guidelines for Writing Shared Library Code

Because the main advantage of a shared library over an archive library is sharing and the space it saves, these guidelines stress ways to increase sharing while avoiding the disadvantages of a shared library. The guidelines also stress upward compatibility. When appropriate, we describe our experience with building the shared C library to illustrate how following a particular guideline helped us.

We recommend that you read these guidelines once from beginning to end to get a perspective of the things you need to consider when building a shared library. Then use it as a checklist to guide your planning and decision-making.

Before we consider these guidelines, let's consider the restrictions to building a shared library common to all the guidelines. These restrictions involve static linking. Here's a summary of them, some of which are discussed in more detail later. Keep them in mind when reading the guidelines in this section:

- Exported symbols have fixed addresses.

  If an exported symbol moves, you have to re-link all **a.out** files that use the library. This restriction applies both to text and data symbols.

- If the library's text changes for one process at run time, it changes for all processes.

  Therefore, any library changes that apply only to a single process must occur in data, not in text, because only the data region is private. (Besides, the text region is read-only.)

- If the library uses a symbol directly, that symbol's run time value (address) must be known when the library is built.

- Imported symbols cannot be referenced directly.

  Their addresses are not known when you build the library, and they can be different for different processes. You can use imported symbols by adding an indirection through a pointer in the library's data.

Choosing Library Members

Include Large, Frequently Used Routines

These routines are prime candidates for sharing. Placing them in a shared library saves code space for individual **a.out** files and saves memory, too, when several concurrent processes need the same code. **printf**(3S) and related C library routines (which are documented in the *Programmer's Reference Manual*) are good examples.

When we built the shared C library...

The **printf**(3S) family of routines is used frequently. Consequently, we included **printf**(3S) and related routines in the shared C library.

Exclude Infrequently Used Routines

Putting these routines in a shared library can degrade performance, particularly on paging systems. Traditional **a.out** files contain all code they need at run time. By definition, the code in an **a.out** file is (at least distantly) related to the process. Therefore, if a process calls a function, it may already be in memory because of its proximity to other text in the process.

If the function is in the shared library, a page fault may be more likely to occur, because the surrounding library code may be unrelated to the calling process. Only rarely will any single **a.out** file use everything in the shared C library. If a shared library has unrelated functions, and unrelated processes make random calls to those functions, the locality of reference may be decreased. The decreased locality may cause more paging activity and, thereby, decrease performance. See also "Organize to Improve Locality."

When we built the shared C library...

Our original shared C library had about 44 KB of text. After profiling the code in the library, we removed small routines that were not often used. The current library has under 29 KB of text. The point is that functions used only by a few **a.out** files do not save much disk space by being in a shared library, and their inclusion can cause more paging and decrease performance.

Exclude Routines that Use Much Static Data

These modules increase the size of processes. As "How Shared Libraries are Implemented" and "Deciding Whether to Use a Shared Library" explain, every process that uses a shared library gets its own private copy of the library's data, regardless of how much of the data is needed. Library data is static: it is not shared and cannot be loaded selectively with the provision that unreferenced pages may be removed from the working set.

For example, **getgrent**(3C), which is documented in the *Programmer's Reference Manual*, is not used by many standard UNIX commands. Some versions of the module define over 1400 bytes of unshared, static data. It probably should not be included in a shared library. You can import global data, if necessary, but not local, static data.

Exclude Routines that Complicate Maintenance

All exported symbols must remain at constant addresses. The branch table makes this easy for text symbols, but data symbols don't have an equivalent mechanism. The more data a library has, the more likely some of them will have to change size. Any change in the size of exported data may affect symbol addresses and break compatibility.

Include Routines the Library Itself Needs

It usually pays to make the library self-contained. For example, **printf**(3S) requires much of the standard I/O library. A shared library containing **printf**(3S) should contain the rest of the standard I/O routines, too.

NOTE: This guideline should not take priority over the others in this section. If you exclude some routine that the library itself needs based on a previous guideline, consider leaving the symbol out of the library and importing it.

Changing Existing Code for the Shared Library

All C code that works in a shared library will also work in an archive library. However, the reverse is not true because a shared library must explicitly handle imported symbols. The following guidelines are meant to help you produce shared library code that is still valid for archive libraries (although it may be slightly bigger and slower). The guidelines mostly explain how to structure data for ease of maintenance, since most compatibility problems involve restructuring data from a shared library to an archive library.

Minimize Global Data

In the current shared library implementation, all external data symbols are global; they are visible to applications. This can make maintenance difficult. You should try to reduce global data, as described below.

First, try to use automatic (stack) variables. Don't use permanent storage if automatic variables work. Using automatic variables saves static data space and reduces the number of symbols visible to application processes.

Second, see whether variables really must be external. Static symbols are not visible outside the library, so they may change addresses between library versions. Only external variables must remain constant.

Third, allocate buffers at run time instead of defining them at compile time. This does two important things. It reduces the size of the library's data region for all processes and, therefore, saves memory; only the processes that actually need the buffers get them. It also allows the size of the buffer to change from one release to the next without affecting compatibility. Statically allocated buffers cannot change size without affecting the addresses of other symbols and, perhaps, breaking compatibility.

Define Text and Global Data in Separate Source Files

Separating text from global data makes it easier to prevent data symbols from moving. If new exported variables are needed, they can be added at the end of the old definitions to preserve the old symbols' addresses.

Archive libraries let the link editor extract individual members. This sometimes encourages programmers to define related variables and text in the same source file. This works fine for relocatable files, but shared libraries have a different set of restrictions. Suppose exported variables were scattered throughout the library modules. Then visible and hidden data would be intermixed. Changing hidden data, such as a string, like **hello** in the following example, moves subsequent data symbols, even the exported symbols:

```
Before                       Broken Successor

int head = 0;                int head = 0;
func()                       func()
{                            {
   ...                          ...
   p = "hello";                 p = "hello, world";
   ...                          ...
}                            }
int tail = 0;                int tail = 0;
```

Assume the relative virtual address of **head** is 0 for both examples. The string literals will have the same address too, but they have different lengths. The old and new addresses of **tail** thus will be 12 and 20, respectively. If **tail** is supposed to be visible outside the library, the two versions will not be compatible.

Adding new exported variables to a shared library may change the addresses of static symbols, but this doesn't affect compatibility. An **a.out** file has no way to reference static library symbols directly, so it cannot depend on their values. Thus it pays to group all exported data symbols and place them at lower addresses than the static (hidden) data. You can write the specification file to control this. In the list of object files, make the global data files first.

```
#objects
   data1.o
   ...
   lastdata.o
   text1.o
   text2.o
   ...
```

If the data modules are not first, a seemingly harmless change (such as a new string literal) can break existing **a.out** files.

Shared library users get all library data at run time, regardless of the source file organization. Consequently, you can put all exported variables' definitions in a single source file without a penalty. You can also use several source files for data definitions.

Initialize Global Data

Initialize exported variables, including the pointers for imported symbols. Although this uses more disk space in the target shared library, the expansion is limited to a single file. Using initialized variables is another way to prevent address changes.

The C compilation system on UNIX System V puts uninitialized variables in a common area, and the link editor assigns addresses to them in an unpredictable way. In other words, the order of uninitialized symbols may change from one link editor run to the next. However, the link editor will not change the order of initialized variables, thus allowing a library developer to preserve compatibility.

Preserve Branch Table Order

You should add new functions only at the end of the branch table. After you have a specification file for the library, try to maintain compatibility with previous versions. You may add new functions without breaking old **a.out** files as long as previous assignments are not changed. This lets you distribute a new library without having to re-link all of the **a.out** files that used a previous version of the library.

Importing Symbols

Shared library code cannot directly use symbols defined outside a library, but an escape hatch exists. You can define pointers in the data area and arrange for those pointers to be initialized to the addresses of imported symbols. Library code then accesses imported symbols indirectly, delaying symbol binding until run time. Libraries can import both text and data symbols. Moreover, imported symbols can come from the user's code, another library, or even the library itself. In Figure 8-4, the symbols _**libc.ptr1** and _**libc.ptr2** are imported from user's code and the symbol _**libc_malloc** from the library itself.

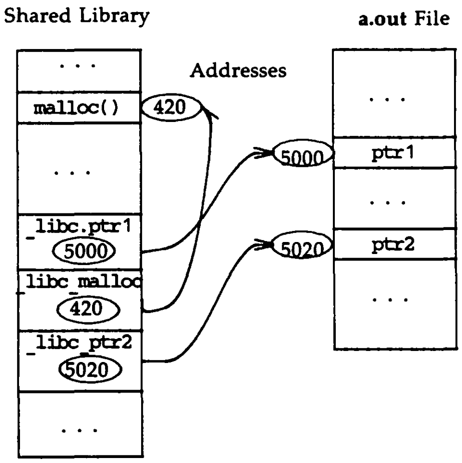


Figure 8-4: Imported Symbols in a Shared Library

The following guidelines describe when and how to use imported symbols.

Imported Symbols that the Library Does Not Define

Archive libraries typically contain relocatable files, which allow undefined references. Although the host shared library is an archive, too, that archive is constructed to mirror the target library, which more closely resembles an **a.out** file. Neither target shared libraries nor **a.out** files can have unresolved symbols.

Consequently, shared libraries must import any symbols they use but do not define. Some shared libraries will derive from existing archive libraries. For the reasons stated above, it may not be appropriate to include all the archive's modules in the target shared library. If you leave something out that the library calls, you have to make an imported symbol pointer for it.

Imported Symbols that Users Must Be Able to Redefine

Optionally, shared libraries can import their own symbols. At first this might appear to be an unnecessary complication, but consider the following. Two standard libraries, **libc** and **libmalloc**, provide a **malloc** family. Even though most UNIX commands use the **malloc** from the C library, they can choose either library or define their own.

When we built the shared C library...

Three possible strategies existed for the shared C library. First, we could have excluded **malloc**(3X). Other library members would have needed it, and so it would have been an imported symbol. This would have worked, but it would have meant less savings.

Second, we could have included the **malloc**(3X) family and not imported it. This would have given us more savings for typical commands, but it had a price. Other library routines call **malloc**(3X) directly, and those calls could not have been overridden. If an application tried to redefine **malloc**(3X), the library calls would not have used the alternate version. Furthermore, the link editor would have found multiple definitions of **malloc**(3X) while building the application. To resolve this the library developer would have to change source code to remove the custom **malloc**(3X), or the developer would have to refrain from using the shared library.

Finally, we could have included **malloc**(3X) in the shared library, treating it as an imported symbol. This is what we did. Even though **malloc**(3X) is in the library, nothing else there refers to it directly. If the application does not redefine **malloc**(3X), both application and library calls are routed to the library version. All calls are mapped to the alternate, if present.

You might want to permit redefinition of all library symbols in some libraries. You can do this by importing all symbols the library defines, in addition to those it uses but does not define. Although this adds a little space and time overhead to the library, the technique allows a shared library to be one hundred percent compatible with an existing archive at link time and run time.

Mechanics of Importing Symbols

Let's assume a shared library wants to import the symbol **malloc**. The original archive code and the shared library code appear below.

Archive Code	Shared Library Code
	`/* See pointers.c on next page */`
`extern char *malloc();`	`extern char *(*_libc_malloc)();`

Archive Code:

```
extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Shared Library Code:

```
/* See pointers.c on next
                  page */

extern char
   *(*_libc_malloc)();

export()
{
    ...
    p = (*_libc_malloc)(n);
    ...
}
```

Making this transformation is straightforward, but two sets of source code would be necessary to support both an archive and a shared library. Some simple macro definitions can hide the transformations and allow source code compatibility. A header file defines the macros, and a different version of this header file would exist for each type of library. The **-I** flag to **cpp**(1) would direct the C preprocessor to look in the appropriate directory to find the desired file.

Archive **import.h**

```
/* empty */
```

Shared **import.h**

```
/*
 *  Macros for importing
 *  symbols.  One #define
 *  per symbol.
 */

 ...
 #define malloc
      (*_libc_malloc)
 ...
 extern char *malloc();
 ...
```

These header files allow one source both to serve the original archive source and to serve a shared library, too, because they supply the indirections for imported symbols. The declaration of **malloc** in **import.h** actually declares the pointer **_libc_malloc**.

Common Source

```
#include "import.h"

extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Alternatively, one can hide the `#include` with `#ifdef`:

Common Source

```
#ifdef SHLIB
#      include "import.h"
#endif

extern char *malloc();

export()
{
    ...
    p = malloc(n);
    ...
}
```

Of course the transformation is not complete. You must define the pointer **_libc_malloc**.

File **pointers.c**

```
char *(*_libc_malloc)() = 0;
```

Note that **_libc_malloc** is initialized to zero, because it is an exported data symbol.

Special initialization code sets the pointers. Shared library code should not use the pointer before it contains the correct value. In the example the address of **malloc** must be assigned to **_libc_malloc**. Tools that build the shared library generate the initialization code according to the library specification file.

Pointer Initialization Fragments

A host shared library archive member can define one or many imported symbol pointers. Regardless of the number, every imported symbol pointer should have initialization code.

This code goes into the **a.out** file and does two things. First, it creates an unresolved reference to make sure the symbol being imported gets resolved. Second, initialization fragments set the imported symbol pointers to their values before the process reaches **main**. If the imported symbol pointer can be used at run time, the imported symbol will be present, and the imported symbol pointer will be set properly.

NOTE: Initialization fragments reside in the host, not the target, shared library. The link editor copies initialization code into **a.out** files to set imported pointers to their correct values.

Library specification files describe how to initialize the imported symbol pointers. For example, the following specification line would set **_libc_malloc** to the address of **malloc**:

```
#init pmalloc.o
_libc_malloc    malloc
```

When **mkshlib** builds the host library, it modifies the file **pmalloc.o**, adding relocatable code to perform the following assignment statement:

```
_libc_malloc = &malloc;
```

When the link editor extracts **pmalloc.o** from the host library, the relocatable code goes into the **a.out** file. As the link editor builds the final **a.out** file, it resolves the unresolved references and collects all initialization fragments. When the **a.out** file is executed, the run time startup (**crt1**) executes the initialization fragments to set the library pointers.

Selectively Loading Imported Symbols

Defining fewer pointers in each archive member increases the granularity of symbol selection and can prevent unnecessary objects from being linked into the **a.out** file. For example, if an archive member defines three pointers to imported symbols, the link editor will resolve all three, even though only one might be needed.

You can reduce unnecessary loading by writing C source files that define imported symbol pointers singly or in related groups. If an imported symbol must be individually selectable, put its pointer in its own source file (and archive member). This will give the link editor a finer granularity to use when it resolves the symbols.

Let's look at some examples. In the coarse method, a single source file might define all pointers to imported symbols:

Old **pointers.c**

```
int (*_libc_ptr1)() = 0;
char *(*_libc_malloc)() = 0;
int (*_libc_ptr2)() = 0;
```

Being able to use them individually requires multiple source files and archive members. Each of the new files defines a single pointer or a small group of related pointers:

File	Contents
ptr1.c	`int (*_libc_ptr1)() = 0;`
pmalloc.c	`char *(*_libc_malloc)() = 0;`
ptr2.c	`int (*_libc_ptr2)() = 0;`

Originally, a single object file, **pointers.o**, defines all pointers. Extracting it requires definitions for **ptr1, malloc,** and **ptr2**. The modified example lets one extract each pointer individually, thus avoiding the unresolved reference for unnecessary symbols.

Providing Archive Library Compatibility

Having compatible libraries makes it easy to substitute one for the other. In almost all cases, this can be done without makefile or source file changes. Perhaps the best way to explain this guideline is by example:

When we built the shared C library...

We had an existing archive library to use as the base. This obviously gave us code for individual routines, and the archive library also gave us a model to use for the shared library itself.

We wanted the host library archive file to be compatible with the relocatable archive C library. However, we did not want the shared library target file to include all routines from the archive: including them all would have hurt performance.

Reaching these goals was, perhaps, easier than you might think. We did it by building the host library in two steps. First, we used the available shared library tools to create the host library to match exactly the target. The resulting archive file was not compatible with the archive C library at this point. Second, we added to the host library the set of relocatable objects residing in the archive C library that were missing from the host library. Although this set is not in the shared library target, its inclusion in the host library makes the relocatable and shared C libraries compatible.

Tuning the Shared Library Code

Some suggestions for how to organize shared library code to improve performance are presented here. They apply to paging systems, such as UNIX System V Release 3.0. The suggestions come from the experience of building the shared C library.

The archive C library contains several diverse groups of functions. Many processes use different combinations of these groups, making the paging behavior of any shared C library difficult to predict. A shared library should offer greater benefits for more homogeneous collections of code. For example, a data base library probably could be organized to reduce system paging substantially, if its static and dynamic calling dependencies were more predictable.

Profile the Code

To begin, profile the code that might go into the shared library.

Choose Library Contents

Based on profiling information, make some decisions about what to include in the shared library. **a.out** file size is a static property, and paging is a dynamic property. These static and dynamic characteristics may conflict, so you have to decide whether the performance lost is worth the disk space gained. See "Choosing Library Members" in this chapter for more information.

Organize to Improve Locality

When a function is in **a.out** files, it probably resides in a page with other code that is used more often (see "Exclude Infrequently Used Routines"). Try to improve locality of reference by grouping dynamically related functions. If every call of **funcA** generates calls to **funcB** and **funcC**, try to put them in the same page. **cflow**(1) (documented in the *Programmer's Reference Manual*) generates this static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.

Align for Paging

The key is to arrange the shared library target's object files so that frequently used functions do not unnecessarily cross page boundaries. When arranging object files within the target library, be sure to keep the text and data files separate. You can reorder text object files without breaking compatibility; the same is not true for object files that define global data. Once again, an example might best explain this guideline:

When we built the shared C library...

We used a 3B2 Computer to build the library; the architecture of the 3B2 Computer uses 2 KB pages. Using name lists and disassemblies of the shared library target file, we determined where the page boundaries fell.

After grouping related functions, we broke them into page-sized chunks. Although some object files and functions are larger than a single page, most of them are smaller. Then we used the infrequently called functions as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault decreased.

After determining the branch table, we rearranged the library's object files without breaking compatibility. We put frequently used, unrelated functions together, because we figured they would be called randomly enough to keep the pages in memory. System calls went into another page as a group, and so on. The following example shows how to change the order of the library's object files:

```
Before                  After

 #objects                #objects
     ...                     ...
     printf.o                strcmp.o
     fopen.o                 malloc.o
     malloc.o                printf.o
     strcmp.o                fopen.o
     ...                     ...
```

Avoid Hardware Thrashing

Finally, you may have to consider the hardware you're using to obtain better performance. Using the 3B2 Computer, for example, you need to consider its memory management. Part of the memory management hardware is an 8-entry cache for translating virtual to physical addresses. Each segment (128 KB) is mapped to one of the eight entries. Consequently, segments 0, 8, 16, ... use entry 0; segments 1, 9, 17, ... use entry 1; and so on.

You get better performance by arranging the typical process to avoid cache entry conflicts. If a heavily used library had both its text and its data segment mapped to the same cache entry, the performance penalty would be particularly severe. Every library instruction would bring the text segment information into the cache. Instructions that referenced data would flush the entry to load the data segment. Of course, the next instruction would reference text and flush the cache entry, again.

When we built the shared C library...

We avoided the cache entry conflicts. At least with the 3B2 Computer architecture, a library's text and data segment numbers should differ by something other than eight.

Making A Shared Library Upward Compatible

The following guidelines explain how to build upward-compatible shared libraries. Note, however, that upward compatibility may not always be an issue. Consider the case in which a shared library is one piece of a larger system and is not delivered as a separate product. In this restricted case, you can identify all **a.out** files that use a particular library. As long as you rebuild all the **a.out** files every time the library changes, versions of the library may be incompatible with each other. This may complicate development, but it is possible.

Comparing Previous Versions of the Library

Shared library developers normally want newer versions of a library to be compatible with previous ones. As mentioned before, **a.out** files will not execute properly otherwise.

The following procedures let you check libraries for compatibility. In these tests, two libraries are said to be compatible if their exported symbols have the same addresses. Although this criterion usually works, it is not foolproof. For example, if a library developer changes the number of arguments a function requires, the new function may not be compatible with the old. This kind of change may not alter symbol addresses, but it will break old **a.out** files.

Let's assume we want to compare two target shared libraries: **new.libx_s** and **old.libx_s**. We use the **nm**(1) command to look at their symbols and **sed**(1) to delete everything except external symbols. A small **sed** program simplifies the job.

New file **cmplib.sed**

```
sed     '/¦extern¦.*/!d
        s///
        /^.bt/d
        /^etext /d
        /^edata /d
        /^end /d'
```

The first line of the **sed** script deletes all lines except those for external symbols. The second line leaves only symbol names and values in the output. The last four lines delete special symbols that have no bearing on library compatibility; they are not visible to application programs. You will have to create your own file to hold the **sed** script.

Now we are ready to create lists of symbol names and values for the new and old libraries:

```
nm old.libx_s ¦ sed -f cmplib.sed >old.nm
nm new.libx_s ¦ sed -f cmplib.sed >new.nm
```

Next, we compare the symbol values to identify differences:

```
diff old.nm new.nm
```

If all symbols in the two libraries have the same values, the **diff**(1) command will produce no output, and the libraries are compatible. Otherwise, some symbols are different and the two libraries may be incompatible. **diff**(1), **nm**(1), and **sed**(1) are documented in the *User's Reference Manual*.

Dealing with Incompatible Libraries

When you determine that two libraries are incompatible, you have to deal with the incompatibility. You can deal with it in one of two ways. First, you can rebuild all the **a.out** files that use your library. If feasible, this is probably the best choice. Unfortunately, you might not be able to find those **a.out** files, let alone force their owners to rebuild them with your new library.

So your second choice is to give a different target path name to the new version of the library. The host and target path names are independent; so you don't have to change the host library path name. New **a.out** files will use your new target library, but old **a.out** files will continue to access the old library.

As the library developer, it is your responsibility to check for compatibility and, probably, to provide a new target library path name for a new version of a library that is incompatible with older versions. If you fail to resolve compatibility problems, **a.out** files that use your library will not work properly.

NOTE: You should try to avoid multiple library versions. If too many copies of the same shared library exist, they might actually use more disk space and more memory than the equivalent relocatable version would have.

Summary

This chapter described the UNIX system shared libraries and explained how to use them. It also explained how to build your own shared libraries. Using any shared library almost always saves disk storage space, memory, and computer power; and running the UNIX system on smaller machines makes the efficient use of these resources increasingly important. Therefore, you should normally use a shared library whenever it's available.