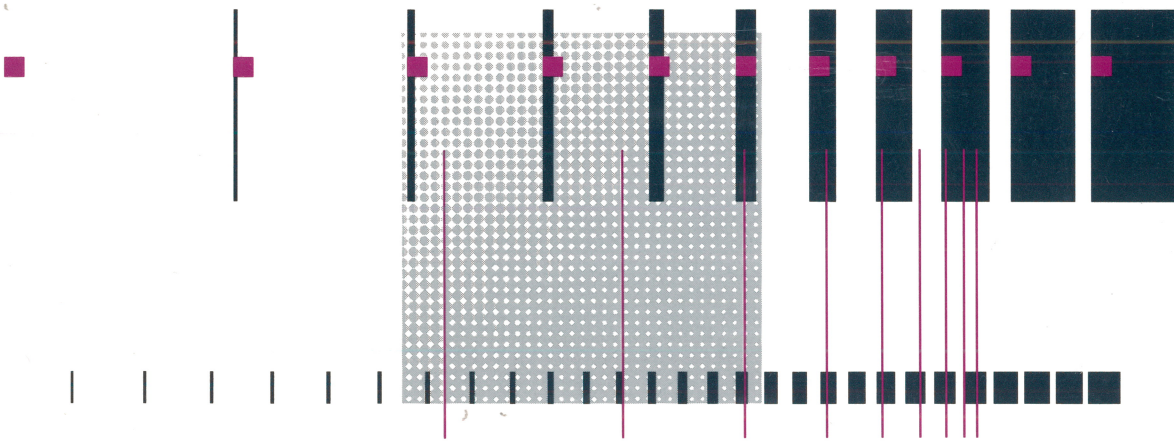
RISC/os (UMIPS) Programmer's Guide Volume I

Order Number 3207DOC



mips

The power of RISC is in the system.

RISC/os (UMIPS)
Programmer's Guide
Volume I

Order Number 3207DOC

March 1989

Your comments on our products and publications are welcome. A postage–paid form is provided for this purpose on the last page of this manual.

Mfg. Part Number 84–00062A/02–00298

MIPS Computer Systems, Inc.
930 Arques Ave.
Sunnyvale, CA 94086

Customer Service Telephone Numbers:

California:	(800)	992–MIPS
All other states:	(800)	443–MIPS
International:	(415)	330–7966

Mfg. Part Number 84–00062/02–00298

Table of Contents

Chapter 4: C Language

Chapter 5: **lint**

Chapter 6: **make**

Chapter 10: **lex**

Chapter 11: **yacc**

Chapter 12: **curses/terminfo**

Chapter 13: File and Record Locking

Purpose

This guide is designed to give you information about programming in a RISC/os (UMIPS) system environment. It does not attempt to teach you how to write programs. Rather, it is intended to supplement texts on programming languages by concentrating on the other elements that are part of getting programs into operation.

Audience and Prerequisite Knowledge

As the title suggests, we are addressing programmers, especially those who have not worked extensively with the RISC/os (UMIPS) system. No special level of programming involvement is assumed. We hope the book will be useful to people who write only an occasional program as well as those who work on or manage large application development projects.

Programmers in the expert class, or those engaged in developing system software, may find this guide lacks the depth of information they need. For them we recommend the *Programmer's Reference Manual*.

Knowledge of terminal use, of a RISC/os (UMIPS) system editor, and of the RISC/os (UMIPS) system directory/file structure is assumed. If you feel shaky about your mastery of these basic tools, you might want to look over the *User's Guide* before tackling this one.

Organization

The material is organized into nineteen chapters, as follows:

- Chapter 1 — Overview

  Identifies the special features of the RISC/os (UMIPS) system that make up the programming environment: the concept of building blocks, pipes, special files, shell programming, etc. As a framework for the material that follows, three different levels of programming in a RISC/os (UMIPS) system are defined: single-user, applications, and systems programming.

- Chapter 2 — Programming Basics

  Describes the most fundamental utilities needed to get programs running.

- Chapter 3 — Application Programming

  Enlarges on many of the topics covered in the previous chapter with particular emphasis on how things change as the project grows bigger. Describes tools for keeping programming projects organized.

- Chapters 4 through 19 — Support Tools, Descriptions, and Tutorials

  Includes detailed information about the use of many of the RISC/os (UMIPS) system tools.

At the end of the text is a glossary and an index.

The C Connection

The RISC/os (UMIPS) system supports many programming languages, and C compilers are available on many different operating systems. Nevertheless, the relationship between the RISC/os (UMIPS) operating system and C has always been and remains very close. Most of the code in the RISC/os (UMIPS) operating system is C, and over the years many organizations using the RISC/os (UMIPS) system have come to use C for an increasing portion of their application code. Thus, while this guide is intended to be useful to you no matter what language(s) you are using, you will find that, unless there is a specific language-dependent point to be made, the examples assume you are programming in C.

Notation Conventions

Whenever the text includes examples of output from the computer and/or commands entered by you, we follow the standard notation scheme that is common throughout RISC/os (UMIPS) system documentation:

- Commands that you type in from your terminal are shown in **bold** type.
- Text that is printed on your terminal by the computer is shown in `constant width` type. Constant width type is also used for code samples because it allows the most accurate representation of spacing. Spacing is often a matter of coding style, but is sometimes critical. In cases where the line on the computer screen is longer than can be shown in this document, the backslant ("\") is used at the end of the line to indicate that the next line is actually still a part of the current line.
- Comments added to a display to show that part of the display has been omitted are shown in *italic* type and are indented to separate them from the text that represents computer output or input. Comments that explain the input or output are shown in the same type font as the rest of the display.

  Italics are also used to show substitutable values, such as, *filename*, when the format of a command is shown.
- There is an implied RETURN at the end of each command and menu response you enter. Where you may be expected to enter only a RETURN (as in the case where you are accepting a menu default), the symbol **<CR>** is used.
- In cases where you are expected to enter a control character, it is shown as, for example, **CTRL-D**. This means that you press the **d** key on your keyboard while holding down the **CTRL** key.
- The dollar sign, **$**, and pound sign, **#**, symbols are the standard default prompt signs for an ordinary user and **root** respectively. **$** means you are logged in as an ordinary user. **#** means you are logged in as **root**.
- When the **#** prompt is used in an example, it means the command illustrated may be used only by **root**.

Command References

When commands are mentioned in a section of the text for the first time, a reference to the manual section where the command is formally described is included in parentheses: **command**(section). The numbered sections are located in the following manuals:

Section (1)	*User's Reference Manual*
Sections (1M), (7)	*System Administrator's Reference Manual*
Sections (2), (3), (4), (5)	*Programmer's Reference Manual*

Information in the Examples

While every effort has been made to present displays of information just as they appear on your terminal, it is possible that your system may produce slightly different output. Some displays depend on a particular machine configuration that may differ from yours. Changes between releases of the RISC/os (UMIPS) system software may cause small differences in what appears on your terminal.

Where complete code samples are shown, we have tried to make sure they compile and work as represented. Where code fragments are shown, while we can't say that they have been compiled, we have attempted to maintain the same standards of coding accuracy for them.

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

UMIPS System Tools and Where You Can Read About Them

The term "UMIPS 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 UMIPS 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 UMIPS 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 UMIPS system tools in this one volume.

Tools not covered in this text:

- the **login** procedure
- UMIPS 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 UMIPS system machine you are using the shell. The shell is the interactive command interpreter that stands between you and the UMIPS 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 UMIPS 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 UMIPS system environment is to find that an available UMIPS 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 UMIPS 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 UMIPS 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 UMIPS 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 UMIPS 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 UMIPS 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 UMIPS 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 UMIPS 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 UMIPS System Environment

This section and the section that follows, briefly describe some full scale programming languages and special purpose languages available under UMIPS. The discussion does not include all languages available under UMIPS, many of which must be purchased separately. For information on other languages available, contact your UMIPS service representative.

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. Fortunately, MIPS' Pascal provides numerous extensions to overcome some of the limitations of standard Pascal.

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.

PL/I

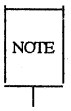
PL/I is a general-purpose, high-level programming language that combines the best features of several other languages such as FORTRAN, COBOL, and ALGOL. MIPS' PL/I conforms to ANSI standard X3.74-1981: a carefully designed subset (subset G) of the language that is both more efficient and easier to learn. Refer to the MIPS-PL/I Language Reference for details on the language.

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.



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.

Compiling and Link Editing

The command used for compiling depends on the language used;

- for C programs, **cc** both compiles and link edits
- for COBOL programs, **cobol** both compiles and link edits

- for FORTRAN programs, **f77** both compiles and link edits
- for Pascal programs, **pc** both compiles and link edits
- for PL/I programs, **PL/I** 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. For a complete list of these options, see the *Languages Programmer's Guide* or the **cc**(1) manual page in the *User's Reference Manual*.

For more information on compiling C and Fortran Programs, and programs written in other languages available under UMIPS, refer to the *Languages Programmer's Guide*.

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**.

For more information on the command line options available with FORTRAN, see the *Languages Programmer's Guide* and the **f77**(1) manual page in the *User's Reference Manual*.

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 source 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 **dbx**. 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 numerous options. They are described in the *Languages Programmer's Guide* and the **ld**(1) manual page in the *User's Reference Manual*.

The UMIPS/Language Interface

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 inquiries
fopen	**freopen**	**fdopen**		open a stream
fread	**fwrite**			binary input/output
fseek	**rewind**	**ftell**		reposition a file pointer in a stream
getc	**getchar**	**fgetc**	**getw**	get a character or word from a stream
gets	**fgets**			get a string from a stream
popen	**pclose**			begin or end a pipe to/from a process
printf	**fprintf**	**sprintf**		print formatted output
putc	**putchar**	**fputc**	**putw**	put a character or word on a stream
puts	**fputs**			put a string on a stream
scanf	**fscanf**	**sscanf**		convert formatted input
setbuf	**setvbuf**			assign buffering to a stream
%system				issue a command through the shell
tmpfile				create a temporary file
tmpnam	**tempnam**			create a name for a temporary file
ungetc				push character back into input stream
vprintf	**vfprintf**	**vsprintf**		print formatted output of a varargs argument list

For all functions: `#include <stdio.h>`
The functions are described in the *Programmer's Reference Manual,* Section 3.

Figure 2-3: C Language Standard I/O Subroutines

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.
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

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 typical 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. The new-line character is discarded and the string is terminated with a null character.

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. The string is then terminated with a null character.

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, such as trying to use these functions on a file that has not been opened for reading, 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 system call or subroutine 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 file's output  */
      /* will be buffered line by line.     */
#define _IOREAD   0001
#define _IOWRT    0002
#define _IONBF    0004
#define _IOMYBUF  0010
#define _IOEOF    0020
#define _IOERR    0040
#define _IOLBF    0100
#define _IORW     0200
      /* In addition to being flags, _IONBF,*/
      /* _IOLBF and IOFBF are possible      */
      /* values for "type" in setvbuf.      */

#ifndef NULL
#define NULL      0
#endif
#ifndef EOF
#define EOF    (-1)
#endif
#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) 2
        *(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**

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**. **libc.a** is found in the directory **/usr/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 for **/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 "**r**" 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("file", "r")) == NULL)
  (void)fprintf(stderr,"%s: Can't open input file %s\n", \
        argv[0],"file");
```

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 a file
chown		change owner and group of a file
getpid	**getpgrp getppid**	get process IDs
getuid	**geteuid getgid**	get user IDs
ioctl		control device
link	**unlink**	add or remove a directory entry
mount	**umount**	mount or unmount a file system
nice		change priority of a process
stat	**fstat**	get file status
time		get time
ulimit		get and set user limits
uname		get name of 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 a null char pointer to mark the end of the 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 process 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 or ignored) */
#define SIGBUS   10 /* bus error */
#define SIGSEGV  11 /* segmentation violation */
#define SIGSYS   12 /* bad argument to system call */
#define SIGPIPE  13 /* write on a pipe with no one to read it */
#define SIGALRM  14 /* alarm clock */
#define SIGTERM  15 /* software termination 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 and SIGPHONE only used in UNIX/PC */
/*#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 we need now. */
                    /* In the future, we can add more signal */
                    /* number 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();

   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);
   }

   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);
}
      /* 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

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    }
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

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

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 4)

```
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
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
```

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

```
SYMBOL      FILE                    FUNCTION LINE

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
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
```

Figure 2-21: **cxref** Output, Using **-c** Option (sheet 3 of 4)

```
SYMBOL      FILE                    FUNCTION LINE

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
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 4 of 4)

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.

pixie, pixstats, and prof

The **pixie**(1), **pixstats**(1) and **prof**(1) utilities can be used to examine characteristics of programs. These utilities are introduced here and described in detail in the *Language Programmer's Guide*.

The pixie Utility

pixie reads an executable program, partitions it into basic blocks, and writes an equivalent program containing additional code that counts the execution of each basic block. (A basic block is a region of the program that can be entered only at the beginning and exited only at the end). **pixie** also generates a file containing the address of each of the basic blocks. For example:

```
$ cc -o wc wc.c
$ ls wc*
wc          wc.c
```

In this example, we generate an executable wc(1) from the source code and then run **pixie** on it:

```
$ pixie wc
```

```
pixie registers: r31, r30, and r22.
old code = 15008 bytes, new code = 45036 bytes (3.0x)
$ ls wc*
wc          wc.Addrs    wc.c        wc.pixie
```

The pixstats Utility

pixstats(1) and **prof**(1) can analyze the files produced by **pixie** and produce a listing of profiling data. **pixstats** analyzes a program's execution characteristics. First use **pixie** to "translate and instrument" the executable object module for the program, as was done above to create **wc.pixie**. Next, execute the translation on an appropriate input. This produces a **.Counts** file:

```
$ wc.pixie wc.c
    126     319    2180 wc.c
```

Here we use wc.pixie as if it was the standard wc program, getting a wordcount of wc.c and also creating the .Counts file:

```
$ ls wc*
wc wc.Addrs wc.Counts    wc.c  wc.pixie
```

Now, use **pixstats** to generate a detailed report on opcode frequencies, interlocks, a mini-profile, and more:

```
 $ pixstats wc
 pixstats wc:
42450 (1.008) cycles (0.0034s @ 12.5MHz)
42094 (1.000) instructions
12399 (0.295) basic blocks
  187 (0.004) calls
 6134 (0.146) loads
 2925 (0.069) stores
 9059 (0.215) loads+stores
 9059 (0.215) data bus use
 2661 (0.063) partial word references
10885 (0.259) branches
 5094 (0.121) nops
    0 (0.000) load interlock cycles
  356 (0.008) multiply/divide interlock cycles \
                    (12/35 cycles)
    0 (0.000) flops (0 mflop/s @ 12.5MHz)
    0 (0.000) floating point data interlock cycles
    0 (0.000) floating point add unit interlock cycles
    0 (0.000) floating point multiply unit \
                    interlock cycles
    0 (0.000) floating point divide unit interlock cycles
    0 (0.000) other floating point interlock cycles
    0 (0.000) 1 cycle interlocks (2 cycle stalls -- \
                    not counted in total)
    0 (0.000) overlapped floating point cycles
   88 (0.002) interlock cycles due to basic block boundary

0.326 load nops per load
```

```
0.323 stores per memory reference
0.294 partial word references per reference

  3.4 instructions per basic block
  3.9 instructions per branch
0.273 backward branches per branch
0.277 branch nops per branch
227 cycles per call
225 instructions per call
      .
      .
      .
```

Additional information in the listing includes:

Register saves/restore
Instruction concentration
opcode distribution
Register usage

The prof Utility

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. For example:

```
$ prof wc
Profile listing generated Fri Jul 22 16:20:22 1988 with:
   prof wc

-------------------------------------------------------------
*  -p[rocedures] using basic-block counts;                  *
*  sorted in descending order by the number of              *
*  cycles executed in each                                  *
*  procedure; unexecuted procedures are excluded            *
-------------------------------------------------------------

42094 cycles

cycles %cycles  cum %  cycles bytes procedure (file)
                      /call /line

33923   80.59  80.59   33923    16 main (wc.c)
 2286    5.43  86.02      23    12 fclose (flsbuf.c)
 1365    3.24  89.26     455     5 memcpy (memcpy.s)
 1161    2.76  92.02      43    29 _flsbuf (flsbuf.c)
 1128    2.68  94.70     282    18 _doprnt (doprnt.c)
  713    1.69  96.39     713    20 _cleanup (flsbuf.c)
  347    0.82  97.22      87    24 fread (fread.c)
  240    0.57  97.79      48    19 _filbuf (filbuf.c)
  162    0.38  98.17     162    11 malloc (malloc.c)
   96    0.23  98.40      24    18 printf (printf.c)
   83    0.20  98.60      42    23 _findbuf (flsbuf.c)
   66    0.16  98.76      66    16 wcp (wc.c)
```

```
  65     0.15   98.91       65    18 _endopen (fopen.c)
  64     0.15   99.06       22    27 fflush (flsbuf.c)
  47     0.11   99.17       47    33 _xflsbuf (flsbuf.c)
  46     0.11   99.28       46    12 morecore (malloc.c)
  39     0.09   99.38       39    20 _findiop (findiop.c)
  33     0.08   99.45       11     7 sbrk (sbrk.s)
  30     0.07   99.52       30    24 _wrtchk (flsbuf.c)
  30     0.07   99.60       15    12 isatty (isatty.c)
  30     0.07   99.67        6    16 read (read.s)
  24     0.06   99.72        6    16 close (close.s)
  23     0.05   99.78       23     5 strlen (strlen.s)
  18     0.04   99.82       18    11 free (malloc.c)
  16     0.04   99.86       16     5 __start (../crt1text.s)
  13     0.03   99.89       13    26 fopen (fopen.c)
  12     0.03   99.92        6    16 ioctl (ioctl.s)
   7     0.02   99.94        7    10 exit (cuexit.c)
   6     0.01   99.95        6    16 open (open.s)
   6     0.01   99.96        6    16 getpagesize (getpagesize.s)
   6     0.01   99.98        6    16 write (write.s)
   4     0.01   99.99        ?    12 findbucket (malloc.c)
   3     0.00  100.00        ?     6 _dwmultu (dwmultu.s)
   2     0.00  100.00        2     8 _exit (exit.s)

The remainder of the listing contains:

-------------------------------------------------------------
*  -p[rocedures] using invocation counts;                   *
-------------------------------------------------------------
*  -h[eavy] using basic-block counts;                       *
```

For further information on these and other options to the prof command, refer to **prof(1).**

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 **dbx** command.

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 re-compilation 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 2-23 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 2-23: **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 can be found in the chapter **make**.

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 2-24 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			
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 2-24: **nm** Output, with **-f** Option (sheet 1 of 4)

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 2-24: **nm** Output, with **-f** Option (sheet 2 of 4)

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 2-24: **nm** Output, with **-f** Option (sheet 3 of 4)

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 2-24: **nm** Output, with **-f** Option (sheet 4 of 4)

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 SCCS IDentifying 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. See the SCCS User's Guide in this document.

In addition, the UMIPS system provides RCS, which is similar to, but many would say better than, SCCS. It is an alternate source control tool described in a separte chapter of this document and the following man pages:

rcs(1)
ci(1)
co(1)
rcsmerge(1)
rcsdiff(1)
rlog(1)

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 the chapter "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*. The chapter "awk" in 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 *User's Reference Manual.* A tutorial on **lex** is contained later in 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 *User's Reference Manual.* A detailed description of **yacc** may be found in the chapter "yacc" in this guide.

Advanced Programming Tools

In "Programming Basics" 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). Programs should include the line

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

to bring in the header file shown in Figure 3-1.

```
/* 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) */
#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.* The chapter "File and Record Locking" is a detailed discussion of the subject with a number of examples.

Interprocess Communications

In Chapter 2 we described **fork**ing and **exec**ing 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 the chapter "Advanced IPC Tutorial".

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. See the chapter "curses/terminfo". 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.*

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.

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

Where to Find More Information

For more information on the Link Editor Command Language, see the *Languages Programmers Guide* and **ld**(1) in the *Users Reference Manual.*

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

See the *Assembly Language Programmer's Guide* for a detailed description of the MIPS' object file format.

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 and the symbol table see the *Assembly Language Programmer's Guide*. The object file library consists of several portions. The functions reside in **/usr/lib/libmld.a** and are loaded during the compilation of a C language program by the **-l** command line option:

cc *file*

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.
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.

Function	Reference	Brief Description
ldsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed.
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.
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

On small machines, shared libraries can both improve performance and reduce maintenance, offering both disk storage and memory savings. This may not always be true on large, high performance systems, such as MIPS machines. When the emphasis is on performance, and when disk space and memory are more plentiful, the use of shared libraries may not be worth the performance loss and increased maintenance complexities. However, shared libraries are beneficial in certain instances. See the chapter "Shared Libraries" for a detailed discussion of shared libraries.

Symbolic Debugger

This section describes the use of the symbolic debugger **dbx** within the context of an application development project.

dbx 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. **dbx** 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

See the *Language Programmer's Guide* and **dbx(1)** in the *User's Reference Manual* for details on the use of **dbx**. The manual page is in Section (1) of the *User'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

The chapter "lint" contains a description of **lint** with examples of the kinds of conditions it uncovers. The manual page is in Section (1) of the *User'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 *User's Reference Manual.* Refer to the chapter "make" in this guide for 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**, which 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 function, 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

SCCS commands are in Section (1) of the *User's Reference Manual.* The chapter "Source Code Control System (SCCS)" in this guide is an SCCS user's guide.

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];
```

```
} 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;

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.
     */
    .
    .
    .
]

/* 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 information from screen.
         */

        /*
         * Wait for write semaphore to reach 0
```

```
         * (index not being written).
         */
        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 we use the file pointers found in the index to
         * read the book file.  Then we print the information
         * on the book(s) to the screen.
         */
    } /* while */
}
/* 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)
   {
      /*
       * 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 this value to "bookpos".
       */

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

        /*
         * 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);
        }
        /*
         * 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
         *  information.
         */
#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);
            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

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: C Language

Introduction

This chapter contains a summary of the grammar and syntax rules of the C Programming Language supported by UMIPS. Refer to the *Languages Programmer's Guide* for information on the following subjects:

- How to compile C language programs
- Alignment, size, and value ranges C language variables,
- Storage mapping of C language arrays, structures, and unions
- Interface between C language programs and programs written in Pascal and FORTRAN

Lexical Conventions

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token.

Comments

The characters **/*** introduce a comment that terminates with the characters ***/**. Comments do not nest.

Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) and the dollar sign ($) count as letters. The dollar sign counts as as a letter only when the **-Wf, -X dollar** option is used. Uppercase and lowercase letters are different. There is no limit on the length of a name. Other implementations may collapse case distinctions for external names, and may reduce the number of significant characters for both external and non-external names.

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

auto	**do**	**goto**	**sizeof**	**void**
break	**double**	**if**	**static**	**volatile**
case	**else**	**int**	**struct**	**while**
char	**enum**	**long**	**switch**	
const	**extern**	**register**	**typedef**	
continue	**float**	**return**	**union**	
default	**for**	**signed**	**unsigned**	

Some implementations also reserve the word **fortran.**

Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "Storage Class and Type."

Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0** through **7** only. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant

whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant that exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant. As discussed below, on MIPS Computers integer and long values may be considered identical.

Character Constants

A character constant is a character enclosed in single quotes, as in **'x'**. The value of a character constant is the numerical value of the character in the machine's character set. Certain nongraphic characters, the single quote (**'**) and the backslash (****), may be represented according to the table of escape sequences shown in Figure 4-1:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	*ddd*

Figure 4-1: Escape Sequences for Nongraphic Characters

The escape *\ddd* consists of the backslash followed by 1, 2, or 3 octal digits that are taken to specify the value of the desired character. A special case of this construction is **\0** (not followed by a digit), which indicates the ASCII character **NUL**. If the character following a backslash is not one of those specified, the behavior is undefined. An explicit new-line character is illegal in a character constant. The type of a character constant is **int**.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant has type **double**.

Enumeration Constants

Names declared as enumerators (see "Structure, Union, and Enumeration Declarations" under "Declarations") have type **int**.

String Literals

A string literal is a sequence of characters surrounded by double quotes, as in "...". A string literal has type "array of **char**" and storage class **static** (see "Storage Class and Type") and is initialized with the given characters. The compiler places a null byte (**\0**) at the end of each string literal so that programs that scan the string literal can find its end. In a string literal, the double quote character (") must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following new-line are ignored. All string literals, even when written identically, are distinct.

Syntax Notation

Syntactic categories are indicated by *italic* type and literal words and characters by **bold** type. Alternative categories are listed on separate lines. An optional entry is indicated by the subscript "opt," so that

$$\{\ expression_{opt}\ \}$$

indicates an optional expression enclosed in braces. The syntax is summarized in "Syntax Summary" at the end of the chapter.

Storage Class and Type

The C language bases the interpretation of an identifier upon two attributes of the identifier: its storage class and its type. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

Storage Class

There are five declarable storage classes:

- automatic
- static
- external
- register
- volatile

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "Statements") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block. Volatile storage class is for variables whose contents may be modified in ways unknown to the compiler. The compiler does not optimize those variables assigned a **volatile** storage class specifier.

Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent. In particular, **char** may be signed or unsigned by default. In this implementation the default is unsigned.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. Plain integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs. The sizes for the MIPS Computer are shown in Figure 4-2.

MIPS Computer (ASCII)	
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$

Figure 4-2: MIPS Computer Hardware Characteristics

The properties of **enum** types (see "Structure, Union, and Enumeration Declarations" under "Declarations") are identical to those of some integer types. The implementation may use the range of values to determine how to allot storage.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation.

Single-precision floating point (**float**) and double precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as arithmetic types. **Char**, **int** of all sizes whether **unsigned** or not, and **enum** will collectively be called integral types. The **float** and **double** types will collectively be called floating types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- arrays of objects of most types
- functions that return objects of a given type
- pointers to objects of a given type
- structures containing a sequence of objects of various types
- unions capable of containing any one of several objects of various types

In general these methods of constructing objects can be applied recursively.

Objects and lvalues

An object is a manipulatable region of storage. An lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators that yield lvalues: for example, if **E** is an expression of pointer type, then ***E** is an lvalue expression referring to the object to which **E** points. The name "lvalue" comes from the assignment expression **E1 = E2** in which the left operand **E1** must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

Operator Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "Arithmetic Conversions." The summary will be supplemented as required by the discussion of each operator.

Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer preserves sign. On the machines from MIPS Computer Systems, sign extension of **char** variables does not occur. It is guaranteed that a member of the standard character set is non-negative.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, **'\377'** has the value **−1**.

When a longer integer is converted to a shorter integer or to a **char,** it is truncated on the left. Excess bits are simply discarded.

Float and Double

All floating arithmetic in C is carried out in double precision by default. Whenever a **float** appears in an expression it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float,** for example by an assignment, the **double** is rounded before truncation to **float** length. This result is undefined if it cannot be represented as a float. The compiler option **-float** prevents the promotion of objects from *float* to **double** in expressions, but not as actual arguments in function calls.

The use of prototypes to declare formal arguments to functions prevents promotion of the corresponding actual arguments.

Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided. The implementation conforms to ANSI/IEEE 754-1985.

Conversions of integral values to floating type behave well. Some loss of accuracy occurs if the destination lacks sufficient bits.

Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer:

(modulo $2^{wordsize}$).

In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned **short** integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus, the conversion amounts to padding with zeros on the left.

Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

1. First, any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**.
2. Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.
3. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result.
4. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.
5. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both converted to **unsigned long** and that is the type of the result.
6. Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.
7. Otherwise, both operands must be **int**, and that is the type of the result.

Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "Expression Statement" under "Statements") or as the left operand of a comma expression (see "Comma Operator" under "Expressions").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

Expressions and Operators

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of **+** (see "Additive Operators") are those expressions defined under "Primary Expressions", "Unary Operators", and "Multiplicative Operators". Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the grammar of "Syntax Summary".

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. Expressions involving a commutative and associative operator (*, **+**, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

Primary Expressions

Primary expressions involving ., **->**, subscripting, and function calls group left to right.

primary-expression:
identifier
constant
string literal
(expression)
primary-expression [expression]
primary-expression (expression-list$_{opt}$ *)*
primary-expression . identifier
primary-expression -> identifier

expression-list:
expression
expression-list , expression

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier that is declared "function returning ...", when used except in the function-name position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int** and floating constants have type **double**.

A string literal is a primary expression. Its type is originally "array of **char**", but following the same rule given above for identifiers, this is modified to "pointer to **char**" and the result is a pointer to the first character in the string literal. (There is an exception in certain initializers; see "Initialization" under "Declarations.")

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is **int**, and the type of the result is "...". The expression **E1[E2]** is identical (by definition) to ***((E1)+(E2))**. All the clues needed to understand this notation are contained in this subpart together with the discussions in "Unary Operators" and "Additive Operators" on identifiers, ***** and **+**, respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under "Types Revisited."

Function Calls

An expression that specifies a function call must have a type pointer to the function returning *void* or returning an object type other than an array.

If the function call expression has a type that specifies the type for its parameters, the number of arguments must agree with the number of parameters. The type of each argument must have a value that can be assigned to an object with the unqualified version of the type of its corresponding parameter.

A function call can be defined as a postfix expression followed by an argument list enclosed in parentheses (). A null argument list can be specified by specifying only the parentheses. Arguments to the function can be specified by a command separated list of expressions within the parentheses. The postfix expression specifies the function called.

If the expression specifying the function call consists of an undeclared identifier, the identifier is implicitly declared as if the following declaration appeared in the innermost block containing the function call:

`extern int` *identifier*()

The expression specifying the argument can be of any object type. In the function call, each parameter is assigned the value of the corresponding argument.

If the function call doesn't contain a function prototype declarator, integral promotions are performed on each argument; arguments that have type **float** are promoted to **double**. Such promotions are referred to as *default argument promotions*.

Results are undefined for the following conditions:

- When the number of arguments doesn't agree with the number of arguments.
- When the function prototype declarator is *not* specified at the function definition, and the types of arguments after promotion are not the same as those of the parameters after promotion.

- When the function prototype declarator is specified at the function definition, and the types of arguments after promotion are not the same as those of the parameters after promotion.
- When the function prototype ends with an ellipsis (, ...).

If a function prototype declarator is specified at the function called, the arguments are implicitly converted to the types of the corresponding parameters. When an ellipsis notation in a function prototype declarator is specified, the argument type conversion is stopped after the last declared parameter. Default argument promotions are performed on trailing arguments. Results are undefined when a call is executed under the following conditions:

- a parameter is declared with a type that is not the same after the default argument promotion, and
- a function prototype of the same type is not specified in the function definition.

Only those conversions just described are performed implicitly. The number and types of arguments are not compared with those of the parameters in a function definition that contains no function prototype declarator.

The order of evaluation of the function designator, the arguments, and subexpressions within the arguments is unspecified, but there is a sequence point before the actual call.

Recursive function calls are permitted, both directly and indirectly, through any sequence of other functions.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from **-** and **>**) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression **E1->MOS** is the same as **(*E1).MOS**. Structures and unions are discussed in "Structure, Union, and Enumeration Declarations" under "Declarations."

Unary Operators

Expressions with unary operators group right to left.

unary-expression:
** expression*
& lvalue
− expression
! expression
~ expression
++ lvalue
−−lvalue
lvalue ++
lvalue −−
(type-name) expression
sizeof *expression*
sizeof *(type-name)*

The unary * operator means "indirection"; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...," the type of the result is "...".

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary − operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where n is the number of bits in the corresponding signed type.

There is no unary + operator.

The result of the logical negation operator ! is one if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix **++** is incremented. The value is the new value of the operand but is not an lvalue. The expression **++x** is equivalent to **x += 1**. See the discussions "Additive Operators" and "Assignment Operators" for information on conversions.

The lvalue operand of prefix **−−** is decremented analogously to the prefix **++** operator.

When postfix **++** is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix **++** operator. The type of the result is the same as the type of the lvalue expression.

When postfix **−−** is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix **−−** operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a cast. Type names are described in "Type Names" under "Declarations."

The **sizeof** operator yields the size in bytes of its operand. (A byte is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations, a byte is the space required to hold a **char.**) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an **unsigned** constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction **sizeof(***type***)** is taken to be a unit, so the expression **sizeof(***type***)−2** is the same as **(sizeof(***type***))−2.**

Multiplicative Operators

The multiplicative operators *, /, and % group left to right. The usual arithmetic conversions are performed.

multiplicative expression:
*expression * expression*
expression / expression
expression % expression

The binary * operator indicates multiplication. The * operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary / operator indicates division.

The binary % operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b)*b + a%b** is equal to **a** (if **b** is not 0).

Additive Operators

The additive operators + and − group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + expression
expression − expression

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer that points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The **+** operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the **−** operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

Shift Operators

The shift operators **<<** and **>>** group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits.

shift-expression:
expression << *expression*
expression >> *expression*

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits. Vacated bits are 0 filled. The value of **E1>>E2** is **E1** right-shifted **E2** bit positions. The right shift is guaranteed to be logical (0 fill) if **E1** is **unsigned**; otherwise, it may be arithmetic.

Relational Operators

The relational operators group left to right.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators **<** (less than), **>** (greater than), **<=** (less than or equal to), and **>=** (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

Equality Operators

equality-expression:
expression == *expression*
expression != *expression*

The **==** (equal to) and the **!=** (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1

whenever **a<b** and **c<d** have the same truth value.)

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

Bitwise AND Operator

and-expression:
expression & expression

The & operator is associative, and expressions involving & may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

Bitwise Exclusive OR Operator

exclusive-or-expression:
expression ^ expression

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

Bitwise Inclusive OR Operator

inclusive-or-expression:
expression | expression

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

Logical AND Operator

logical-and-expression:
expression && expression

The && operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 0 otherwise. Unlike &, && guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Logical OR Operator

logical-or-expression:
expression || *expression*

The || operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand evaluates to nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Conditional Operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = *expression*
lvalue += *expression*
lvalue −= *expression*
lvalue *= *expression*
lvalue /= *expression*
lvalue %= *expression*
lvalue >>= *expression*
lvalue <<= *expression*
lvalue &= *expression*
lvalue ^= *expression*
lvalue |= *expression*

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type.

However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form **E1** *op* **= E2** may be inferred by taking it as equivalent to **E1 = E1** *op* **(E2)**; however, **E1** is evaluated only once. In **+=** and **−=**, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in "Additive Operators." All right operands and all non-pointer left operands must have arithmetic type.

Comma Operator

comma-expression:
expression , *expression*

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initializers (see "Initialization" under "Declarations"), the comma operator as described in this subpart can only appear in parentheses. For example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

Declarations

Declarations are used to specify the interpretation that C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers declarator-list$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

Storage Class Specifiers

The sc-specifiers are:

sc-specifier:
auto
static
extern
register
typedef

The **typedef** specifier does not reserve storage, but can be called a "storage class specifier" for syntactic convenience. See "**typedef**" for more information. The storage classes are as follows:

Auto An auto declaration indicates that storage is allocated at execution and exists only for the duration of that block activation.

Static The compiler allocates storage for a static declara tion at compile time. This allocation remains fixed for the duration of the program. Static variables reside in the program bss section if they are not initialized, otherwise they are placed in the data section.

Register The compiler allocates variables with the register storage class to registers. For programs compiled using the to assign all variables to registers, regardless of the storage class specified.

Extern The extern storage class indicates that the variable refers to storage defined elsewhere in an external data definition. The compiler doesn't allocate storage to extern variable declarations; it uses the following logic in defining and referencing them:

Extern is omitted If an initializer is present, a definition for the symbol is emitted. Having two or more such definitions among all the files comprising a program results in an error at link time or before. If no initializer is present, a common definition is emitted. Any number of common definitions of the same identifier may coexist.

Extern is present The compiler assumes that declaration refers to a name defined elsewhere. A declaration having an initializer is illegal. If a declared identifier is never used, the compiler doesn't issue an external reference to the linker.

Volatile The volatile storage class is specified for those variables that may be modified in ways unknown to the compiler. For example, volatile might be specified for an object corresponding to a memory mapped input/output port or an object accessed by an asynchronously interrupting function. Except for expression evaluation, no phase of the compiler optimizes any of the code dealing with volatile objects.

NOTE. If a pointer specified as volatile is assigned to another pointer without the volatile specification, the compiler treats the other pointer as non-volatile. In the following example:

```
volatile int *1;
int *j;
            .
            .
            .
(volatile*)j = i;
3108282356*10
```

the compiler treats j as a non-volatile pointer and the object it points to as non-volatile, and may optimize it.

The compiler option **volatile** causes all objects to be compiled as volatile.

Type Specifiers

The type-specifiers are

type-specifier:
- *struct-or-union-specifier*
- *typedef-name*
- *enum-specifier*

basic-type-specifier:
- *basic-type*
- *basic-type basic-type-specifiers*

basic-type:
- **char**
- **short**
- **int**
- **long**
- **unsigned**
- **signed**
- **float**
- **double**
- **void**

At most one of the words **long** or **short** may be specified in conjunction with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified in conjunction with **float**; the meaning is the same as **double**. The word **unsigned** may be specified alone, or in conjunction with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most on type-specifier may be given in a declaration. In particular, adjectival use of **long**, **short**, or **unsigned** is not permitted with **typedef** names. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "Structure, Union, and Enumeration Declarations." Declarations with **typedef** names are discussed in "**typedef**."

Type Qualifiers

A type-qualifier has the following syntax:

type-specifier:
const (reserved for future use)
volatile

Const and **volatile** must appear only once in the same specifier or qualifier list, either directly or by one or more **typedef** specifications.

The properties associated with qualified types are meaningful only for expressions that are lvalues. A **const** object that is not **volatile** can be placed in a read-only region of storage.

A volatile object can be modified in ways unknown to the compiler. Therefore, any expression referring to such an object must be evaluated according to the sequence rules described the "Program Execution Rules" section. Furthermore, at every sequence point, the value last stored agrees with that specified by the program execution rules, except as modified by factors not known by the compiler. For example, changes in the values of associated with I/O registers are unknown.

A **volatile** declaration can be used to describe an object that corresponds to a memory-mapped input/output port, or an object accessed by an asynchronously interrupting function. Actions on objects so declared are neither optimized nor reordered by the C compiler, except as permitted by the rules for evaluating expressions.

The following rules apply when the specifier or qualifier list for the declaration of an object that has aggregate or union types includes any type qualifiers:

- The type of any lvalue referring to a member or element of the object with scalar type (including, recursively, any member or element with scalar type of all contained aggregates or unions) has the equivalently qualified version of its specified type.
- The object has the specified aggregate or union type.

The following rules apply in order for two qualified scalar types to be the same:

- Both must have the identically qualified version of the same scalar type.
- The order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.

The following declaration is an example of an object that can be modified at execution time and for which the compiler cannot safely optimize references:

```
extern volatile int real_time_clock
```

Program Execution Rules

This section describes the rules that C language compiler observes when compiling programs using the **O1** command line option for optimization. This is the default, which permits the compiler to perform only minimum optimizations.

- Accessing a volatile object, modifying an object, modifying a file, or callin a function that does any of those operations are all *side effects*. Evaluation of an expression can produce side effects. At certain specified points during execution, called *sequence points*, all side effects of previous valuations are completed, and no side effects of subsequent evaluations have taken place.
- The compiler evaluates all expressions as specified by the semantics. The compiler doesn't evaluate part of an expression if (1) it determines that the value of the expression is not needed and (2) no side effects are produced by the non-evaluation (including those that result from calling a function or accessing a volatile object.
- After a machine interrupt, only the values of objects as they existed at the previous sequence point are reliable. The results of objects between the previous sequence point and the next sequence point are unknown.
- An instance of each object with automatic storage duration is associated with each entry into a block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of an interrupt signal).
- The compiler ensures that at sequence points, volatile objects are stable with respect to previous evaluations being complete and subsequent evaluations not yet having occurred.

Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer:

declarator-list:
init-declarator
init-declarator , declarator-list

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in "Initialization." The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
pointer$_{opt}$ *direct-declarator*

direct-declarator
identifier
(declarator)
direct_declarator [constant-expression$_{opt}$ *]*
direct-declarator (parameter-type-list)
direct-declarator (identifier-list$_{opt}$ *)*

pointer:
    * *type-qualifier-list*$_{opt}$
    * *type-qualifier-list*$_{opt}$ *pointer*
type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier
parameter-type-list:
    parameter-list
    parameter-list , . . .
parameter-list
    parameter-declaration
    parameter-list , *parameter-declaration*
parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator$_{opt}$
identifier-list
    identifier
    identifier-list , *identifier*

The grouping is the same as in expressions.

Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

T D1

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T** ," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in "**int x**" is just **int**). Then if **D1** has the form

***D**

the type of the contained identifier is "... pointer to **T** ."

If **D1** has the form

D()

then the contained identifier has the type "... function returning **T**."

If **D1** has the form

D[*constant-expression*]

or

D[]

then the contained identifier has type "... array of **T**." In the first case, the constant

expression is an expression whose value is determinable at compile time, whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions.") When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions that specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function, which returns an integer. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**. The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**, and then using indirection through the (pointer) result to yield an integer. In the declarator **(*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static 3-dimensional array of integers, with rank 3×5×7. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array" and the last has type **int**.

Pointer Declarations

If, in the declaration **T D1**, **D1** has the following format:

* *type-qualifier-list*^{opt} *D*

and the type specified for *ident* in the declaration **T D** is *derived-declarator-type T*, then the type specified for *ident* is *derived-declarator-type type-qualifier-list* pointer to *T*. If the type qualifier list includes **volatile**, the identifier is a *volatile-qualified* pointer.

For two ponter types to be same, both must be identically qualified and both must be pointers to the same type.

Function Declaration

If, in the declaration **T D1**, **D1** has the following format:

```
parameter-type-list
```

or

identifier-list^{opt} *D*

and the type specified for *ident* in the declaration **T D** is *derived-declarator-type T*, then the type specified for *ident* is derived-declarator-type function returning *T*.

The following rules apply to function declarations:

- A function declarator must not specify a return type that is a function type or an array type.
- An identifier list in a function declarator that is not part of a function definition must be empty.
- All declarations of a specific function in the same scope must specify the same type.
- The parameter types and optional parameter identifiers are specified in a parameter type list. If the list ends with an ellipsis (,...), information need not be specified on the number of types of parameters following the comma; **void** specified as the only item indicates a function with no parameters.
- The storage-class specifier in a declaration is recognized only only when the declared parameter is one of the members of the parameter type list for a function definition. Otherwise, the storage-class specifier is ignored.
- The identifier list declares only the identifiers of the parameters of the function. If the function declarator is part of a function definition, an empty list specifies that the function has no parameters. If the function declarator is *not* part of a function definition, an empty list specifies that no information about the number or types of the parameters is supplied.
- Two function types must specify the same return type in order for them to be the same. Two associated parameter type lists must specify the same number of parameters and agree in the use of the ellipsis terminator.

  If one function type uses a parameter typelist, and the other is is specified by a function definition that contains an identifier list, both must have the same number of parameters. The type of each identifier must be the same type as the corresponding prototype parameter, if the type resulting from default argument promotion to the type of the identifier is the same type as the type of the corresponding prototype parameters.

  For each parameter declared with a function or array type, the parameter type compared is the one that results from conversion to pointer type. For each parameter declared with qualified type, the parameter type compares is the unqualified version of its declared type.

Consider the following example:

```
int f(void), *fip(), (*pf1)();
```

This example declares the following:

The function **f** with no parameters returning an **int**
The function **fip** with no parameter specifications
returning a pointer to an **int**.
A pointer **pf1** to a function with no parameter
specification returning an **int**.

Note that the binding of ***fip()** is ***(fip())** and the same construction in an expression requires a call to **fip**; then, the use of indirection through the yields an **int**. In the declarator **(*pf1)()**, the extra parentheses are required to indicate that indirection through a pointer to a function yields a function designator. The designator is then used to call a function, which returns an **int**.

If the declaration occurs outside of a function, the identifiers have full scope and external linkage. If the declaration occurs inside a function, the identifiers of the function **f** and **fip** have block scope and external linkage; the identifier of the pointer **pf1** has block scope and no linkage.

The following example

```
int (*apf1[3])(int *x, int *y);
```

declares an array **apf1** of three pointer to functions returning **int**. Each function has two parameters that are pointers to **int**. The identifiers **x** and **y** are declared for descriptive purposes only; they go out of scope at the end of the **apf1** declaration.

The declaration

```
int (*fpf1(int (*)(long), int))(int, ...);
```

declares a function **fpf1** that returns a pointer to a function returning an **int**. The function **fpf1** has two parameters: a pointer to a function returning a and **int** (with one parameters of type **long**), and an **int**. The pointer returned by **fpf1** points to a function that has at least one parameter with the type **int**.

Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object that may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
struct-or-union { struct-decl-list }
struct-or-union identifier { struct-decl-list }
struct-or-union identifier

struct-or-union:
struct
union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a field; its length, a non-negative constant expression, is set off from the field name by a colon.

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

Within a structure, the objects declared have addresses that increase as the declarations are read left to right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field that does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. (See Figure 4-2 for sizes of basic types on machines from MIPS Computer Systems.)

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependent boundary.

The language does not restrict the types of things that are declared as fields. Moreover, even **int** fields may be considered to be unsigned. For these reasons, it is strongly recommended that fields be declared as **unsigned** where that is the intent. There are no arrays of fields, and the address-of operator, **&**, may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

struct *identifier* { *struct-decl-list* }
union *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

struct *identifier*
union *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union that contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration that gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures that contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode
{
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

struct tnode s, *sp;

declares **s** to be a structure of the given sort and **sp** to be a pointer to a structure of the given sort. With these declarations, the expression

sp->count

refers to the **count** field of the structure to which **sp** points;

s.left

refers to the left subtree pointer of the structure **s**; and

s.right->tword[0]

refers to the first character of the **tword** member of the right subtree of **s**.

Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes **color** the enumeration-tag of a type describing various colors, and then declares **cp** as a pointer to an object of that type and **col** as an object of that type. The possible values are drawn from the set {0,1,20,21}.

Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and consists of an expression or a list of values nested in braces.

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }

initializer-list:
expression
initializer-list , *initializer-list*
{ *initializer-list* }
{ *initializer-list* , }

All the expressions in an initializer for a static or external variable must be constant

expressions, which are described in "Constant Expressions," or expressions that reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that are not initialized are guaranteed to start off as garbage.

When an initializer applies to a scalar (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an aggregate (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string literal. In this case successive characters of the string literal initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes **x** as a one-dimensional array that has three members, since no size was specified and there are three initializers.

```
float y[4][3] =
{
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise, the next two lines initialize **y[1]** and **y[2]**. The initializer ends early and therefore **y[3]** is initialized with 0. Precisely, the same effect could have been achieved by

```
float y[4][3] =
{
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y** begins with a left brace but that for **y[0]** does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for **y[1]** and **y[2]**. Also,

```
float y[4][3] =
{
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of **y** (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

char msg[] = "Syntax error on line %s\n";

shows a character array whose members are initialized with a string literal. The length of the string (or size of the array) includes the terminating NUL character, \0.

Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of **sizeof**), it is desired to supply the name of a data type. This is accomplished using a "type name," which in essence is a declaration for an object of that type that omits the name of the object.

type-name:
    type-specifier abstract-declarator

abstract-declarator:
        pointer
        pointer$_{opt}$ *direct-abstract-declarator*
direct-abstract-declarator:
        (abstract-declarator)
        direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
        direct-abstract-declarator$_{opt}$ *(parameter-type-list*$_{opt}$ *)*

To avoid ambiguity, in the construction

(abstract-declarator)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
int (*[3])()
```

name respectively the types "integer," "pointer to integer," "array of three pointers to integers," "pointer to an array of three integers," "function returning pointer to integer," "pointer to function returning an integer," and "array of three pointers to functions returning an integer."

Implicit Declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is "function returning ...," it is implicitly declared to be **extern**.

In an expression, an identifier followed by (and not already declared is contextually declared to be "function returning **int**".

typedef

Declarations whose "storage class" is **typedef** do not define storage but instead define identifiers that can be used later as if they were type keywords naming fundamental or derived types.

typedef-name:
    identifier

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "Meaning of Declarators." For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int**," and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand-new types, only synonyms for types that could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

Statements

Except as indicated, statements are executed in sequence.

Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

Conditional Statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The **else** ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

while Statement

The **while** statement has the form

while (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

do Statement

The **do** statement has the form

do *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

for Statement

The **for** statement has the form:

for ($exp\text{-}1_{opt}$; $exp\text{-}2_{opt}$; $exp\text{-}3_{opt}$) *statement*

Except for the behavior of **continue**, this statement is equivalent to

```
exp-1 ;
while ( exp-2 )
{
        statement
        exp-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

switch Statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in

"Constant Expressions."

There may also be at most one statement prefix of the form

default :

which properly goes at the end of the case constants.

When the **switch** statement is executed, its expression is evaluated and compared in turn with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a **default** prefix, control passes to the statement prefixed by **default**. If no case matches and if there is no **default**, then none of the statements in the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. That is, once a case constant is matched, all **case** statements (and the **default**) from there to the end of the **switch** are executed. To exit from a switch, see "**break** Statement."

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective. A simple example of a complete **switch** statement is:

```
switch (c) {
   case 'o':
      oflag = TRUE;
      break;
   case 'p':
      pflag = TRUE;
      break;
   case 'r':
      rflag = TRUE;
      break;
   default :
      (void) fprintf(stderr, "Unknown option\n");
      exit(2);
   }
```

break Statement

The statement **break ;** causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

continue Statement

The statement **continue ;** causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...)          do                   for (...)
{                    {                    {
      ...                  ...                  ...
contin: ;            contin: ;            contin: ;
}                    } while (...);       }
```

a **continue** is equivalent to **goto contin**. (Following the **contin:** is a null statement; see "Null Statement.")

return Statement

A function returns to its caller by means of the **return** statement, which has one of the forms

return ;
return *expression* ;

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

goto Statement

Control may be transferred unconditionally by means of the statement

goto *identifier* ;

The identifier must be a label (see "Labeled Statement") located in the current function.

Labeled Statement

Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared. See "Scope Rules."

Null Statement

The null statement has the form

;

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as **while**.

External Definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see "Type Specifiers" in "Declarations") may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

External Function Definitions

A function definition has the following syntax:

function-definition:
decl-specifiers^{opt} *declarator-declaration-list*^{opt} *compound-statement*

The identifier (the name of the function) must adhere to the following rules:

- The identifier declared (the name of the function) must have a function type, as specified in the declarator portion of the function definition.
- The return type of a function shall be **void** or an object type other than an array.
- If the declarator includes a parameter type list, the declaration of each parameter must include an identifier *except* when the parameter list consists of a single parameter of type **void**; such a list must not contain an identifier. No declaration list can follow.
- If the declarator contains an identifier list, only those names in the list can be declared in the declaration list.
- An identifier declared as a **typedef** name must not be redeclared as a parameter.
- The declarations in the declaration list can contain only **register** storage-class specifiers; no initializations are permitted.

The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. If the parameter contains a parameter type list, the list also specifies the types of all the parameters. The declarator also serves as a function prototype for later calls to the same function in the same translation unit. If the declarator contains an identifier list, the parameter types can be declared in a subsequent declaration list. Any undeclared parameter has the type **int**.

Results are unknown if a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation.

On entry to a function, the value of the argument expression is converted to the type of its corresponding parameter, as if by assignment to the parameter. Array expressions and function designators as arguments are converted to pointers before the call. A declaration of a parameter as "array of *type*" is adjusted to "pointer to *type*;" a declaration of a parameter as "function of *type*" is adjusted to "pointer to function returning *type*."

Each parameter has automatic storage duration, whose identifier is an *lvalue*. The parameter must be declared at the head of the compound statement that constitutes the function body; it cannot be redeclared in the function body except within a closed block. Consider the following example:

```
extern int max(int a, int b)
{
   return a > b ? a : b; }
```

In the above example,

extern is the storage-class specifier (default)
int is the type specifier (default).
max(int a, int b) is the function declarator.
{ return a > b ? a : b; } is the function declarator.

The following example is similar to this example, except it uses the identifier-list form for the parameter declarations:

```
extern int max(a, b)
int a, b;
{
   return a> b ? a : b;
}
```

In this example, **int a, b;** is the declaration list for the parameters; it is also the default and can be omitted. The first example differs from the second in that it acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function.

The following example permits the passing of one function to another:

```
int f(void)
/*...*/
g(f);
```

Because (. does not follow **g(f)**, **f** must be declared explicitly. The definition of **g** might appear as follows:

```
g(int (*funcp)(void))
{
   /*...*/ (*funcp)() /* or funcp() ...*/
}
```

The definition could also appear as follows:

```
g(int (*funcp)(void))
{
   /*...*/ (*func)() /* or func() ...*/
}
```

A simple example of a complete function definition is

```
int max(a, b, c)
        int a, b, c;
{
        int m;

        m = (a > b) ? a : b;
        return((m > c) ? m : c);
}
```

Here **int** is the type-specifier; **max(a, b, c)** is the function-declarator; **int a, b, c;** is the declaration-list for the formal parameters; **{ ... }** is the block giving the code for the statement.

In the absence of prototypes, the C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. All **char** and **short** formal parameter declarations are similarly adjusted to read **int**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of . . . " are adjusted to read "pointer to . . . ".

External Data Definitions

An external data definition has the form

data-definition:
declaration

The storage class of such data may be **extern** (which is the default) or **static**, but not **auto** or **register**.

Scope Rules

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers that are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see "Structure, Union, and Enumeration Declarations" in "Declarations") that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
        int distance;
        ...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program that refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function that references the data.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of **extern** does not change the meaning of an external declaration.

In restricted environments, the use of the **extern** storage class takes on an additional meaning. In these environments, the explicit appearance of the **extern** keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without **extern**) in the set of files and libraries comprising a mult-file program.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

Compiler Control Lines

The C compilation system contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the # and the directive, but no additional material (such as comments) is permitted. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect that lasts (independent of scope) until the end of the source program file.

Token Replacement

A control line of the form

#define *identifier token-string*$_{opt}$

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

#define *identifier(identifier, ...) token-string*$_{opt}$

where there is no space between the first identifier and the (, is a macro definition with arguments. There may be zero or more formal parameters. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replace.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued. This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

#undef *identifier*

causes the identifier's preprocessor definition (if any) to be forgotten.

If a **#defined** identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

File Inclusion

A control line of the form

#include "*filename*"

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the file containing the **#include**, and then in a sequence of specified or standard places. Alternatively, a control line of the form

#include <*filename*>

searches only the specified or standard places and not the directory of the **#include**. (How the places are specified is not part of the language. See **cpp**(1) for a description of how to specify additional libraries.)

#includes may be nested.

Conditional Compilation

A compiler control line of the form

#if *restricted-constant-expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in "Constant Expressions"; the following additional restrictions apply here: the constant expression may not contain **sizeof**, casts, or an enumeration constant.)

A restricted-constant expression may also contain the additional unary expression

defined *identifier*

or

defined (*identifier*)

which evaluates to one if the identifier is currently defined in the preprocessor and zero if it is not.

All currently defined identifiers in restricted-constant-expressions are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted-constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#if defined** (*identifier*).

A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to **#if !defined** (*identifier*).

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#else

and then by a control line

#endif

If the checked condition is true, then any lines between **#else** and **#endif** are ignored. If the checked condition is false, then any lines between the test and a **#else** or, lacking a **#else**, the **#endif** are ignored.

Another control directive is

#elif *restricted-constant-expression*

An arbitrary number of **#elif** directives can be included between **#if, #ifdef**, or **#ifndef** and **#else**, or **#endif** directives. These constructions may be nested.

Line Control

For the benefit of other preprocessors that generate C programs, a line of the form

#line *constant "filename"*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by *"filename"*. If *"filename"* is absent, the remembered file name does not change.

Version Control

This capability, known as *S-lists*, helps administer version control information. A line of the form

#ident *"version"*

puts any arbitrary string in the **.comment** section of the **a.out** file. It is usually used for version control. It is worth remembering that **.comment** sections are not loaded into memory when the **a.out** file is executed.

Types Revisited

This part summarizes the operations that can be performed on objects of certain types.

Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the **->** or the **.** must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union
{
        struct
        {
                int         type;
        } n;
        struct
        {
                int         type;
                int         intnode;
        } ni;
        struct
        {
                int         type;
                float       floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

Functions

There are only two things that can be done with a function: call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of **g** might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by **(**.

Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [] is interpreted in such a way that **E1[E2]** is identical to ***((E1)+(E2))**. Because of the conversion rules that apply to **+**, if **E1** is an array and **E2** an integer, then **E1[E2]** refers to the **E2 -th** member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If **E** is an *n*-dimensional array of rank i×j×...×k, then **E** appearing in an expression is converted to a pointer to an (n−1)-dimensional array with rank j×...×k. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (n−1)-dimensional array, which itself is immediately converted into a pointer.

For example, consider **int x[3][5];** Here **x** is a 3×5 array of integers. When **x** appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression **x[i]**, which is equivalent to ***(x+i)**, **x** is first converted to a pointer as described; then **i** is converted to the type of **x**, which involves multiplying **i** by the length the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see "Unary Operators" under "Expressions" and "Type Names" under "Declarations."

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the use of the function is portable.

Constant Expressions

In several places C requires expressions that evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, and **sizeof** expressions, possibly connected by the binary operators

+ - * / % & | ^ << >> == != < > <= >= && ||

or by the unary operators

- ~

or by the ternary operator

?:

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary & operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

Portability Considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

Expressions

The basic expressions are:

expression:
primary
** expression*
& lvalue
– expression
! expression
~ expression
++ lvalue
–– lvalue
lvalue ++
lvalue ––
sizeof *expression*
sizeof (*type-name*)
(type-name) expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

primary:
identifier
constant
string literal
(expression)
primary (expression-list$_{opt}$ *)*
primary [expression]
primary . identifier
primary –> identifier

lvalue:
identifier
primary [expression]
lvalue . identifier
primary –> identifier
** expression*
(lvalue)

The primary-expression operators

() [] . ->

have highest priority and group left to right. The unary operators

* & – ! ~ ++ –– **sizeof** *(type-name)*

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below.

binop:

```
*   /   %
+   -
>>   <<
<   >   <=   >=
==   !=
&
^
|
&&
||
```

The conditional operator groups right to left.

Assignment operators all have the same priority and all group right to left.

asgnop:

```
= +=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=
```

The comma operator has the lowest priority and groups left to right.

Declarations

declaration:

decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:

type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$
type-qualifier decl-specifiers$_{opt}$

sc-specifier:

auto
static
extern
register
typedef

type-specifier:
    struct-or-union-specifier
    typedef-name
    enum-specifier

basic-type-specifier:
    basic-type
    basic-type basic-type-specifiers

basic-type:
    char
    short
    int
    long
    unsigned
    float
    double
    void
    signed

enum-specifier:
    enum *{ enum-list }*
    enum *identifier { enum-list }*
    enum *identifier*

enum-list:
    enumerator
    enum-list , enumerator

enumerator:
    identifier
    identifier = constant-expression

init-declarator-list:
    init-declarator
    init-declarator , init-declarator-list

init-declarator:
    declarator initializer$_{opt}$

type-qualifier
    volatile
declarator:
    pointer$_{opt}$ *direct-declarator*
direct-declarator:
    identifier
    (declarator)
    direct-declarator [constant-expression$_{opt}$ *]*
    direct-declarator (parameter-type-list)
    direct-declarator (identifier-list$_{opt}$ *)*
pointer:
    **type-specifier-list*$_{opt}$
    **type-specifier-list*$_{opt}$ *pointer*
parameter-type-list:
    parameter-list
    parameter-list , ...
parameter-list
    parameter-declaration
    parameter-list , parameter-declaration
parameter-declarations:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator$_{opt}$
identifier-list
    identifier
    identifier-list , identifier

struct-or-union-specifier:
    struct *{ struct-decl-list }*
    struct *identifier { struct-decl-list }*
    struct *identifier*
    union *{ struct-decl-list }*
    union *identifier { struct-decl-list }*
    union *identifier*

struct-decl-list:
    struct-declaration
    struct-declaration struct-decl-list

struct-declaration:
    type-specifier struct-declarator-list ;

struct-declarator-list:
    struct-declarator
    struct-declarator , struct-declarator-list

struct-declarator:
    declarator
    declarator : constant-expression
    : constant-expression

initializer:
    = expression
    = { initializer-list }
    = { initializer-list , }

initializer-list:
    expression
    initializer-list , initializer-list
    { initializer-list }
    { initializer-list , }

type-name:
    type-specifier abstract-declarator

abstract-declarator;
    pointer
    pointer$_{opt}$ *direct-abstract-declarator*
direct-abstract-declarator:
    (abstract-declarator)
    direct-abstract-declarator$_{opt}$ *[constant expression*$_{opt}$ *]*
    direct-abstract-declarator$_{opt}$ *[parameter-type-list*$_{opt}$ *)*

typedef-name:
    identifier

Statements

compound-statement:
    { declaration-list$_{opt}$ *statement-list*$_{opt}$ *}*

declaration-list:
    declaration
    declaration declaration-list

statement-list:
    statement
    statement statement-list

statement:
compound-statement
expression ;
if *(expression) statement*
if *(expression) statement* **else** *statement*
while *(expression) statement*
do *statement* **while** *(expression) ;*
for *(exp_{opt};exp_{opt};exp_{opt}) statement*
switch *(expression) statement*
case *constant-expression : statement*
default *: statement*
break ;
continue ;
return ;
return *expression ;*
goto *identifier ;*
identifier : statement
;

External Definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

Preprocessor

#define *identifier token-string*$_{opt}$
#define *identifier*(*identifier*,...)*token-string*$_{opt}$
#undef *identifier*
#include "*filename*"
#include <*filename***>**
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#elif *restricted-constant-expression*
#else
#endif
#line *constant* "*filename*"
#ident "*version*"

Chapter 5: lint

Introduction

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error prone constructions, which nevertheless are legal. **lint** accepts multiple input files and library specifications and checks them for consistency.

Usage

The **lint** command has the form:

lint [*options*] *files ... library-descriptors ...*

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

−a Suppress messages about assignments of long values to variables that are not long.

−b Suppress messages about break statements that cannot be reached.

−c Only check for intra-file bugs; leave external information in files suffixed with **.ln**.

−h Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).

−n Do not check for compatibility with either the standard or the portable **lint** library.

−o *name* Create a lint library from input files named **llib−l***name***.ln**.

−p Attempt to check portability.

−u Suppress messages about function and external variables used and not defined or defined and not used.

−v Suppress messages about unused arguments in functions.

−x Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as **−ab** or **−xha**.

The names of files that contain C language programs should end with the suffix **.c**, which is mandatory for **lint** and the C compiler.

lint accepts certain arguments, such as:

−lm

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions. The next section, "**lint** Message Types," describes how it is done.

lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined in a library file but are not used in a source file do not result in messages. **lint** does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file that contains descriptions of the programs that are normally loaded when a C language program is run. When the **−p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **−n** option can be used to suppress all library checking.

lint Message Types

The following paragraphs describe the major categories of messages printed by **lint.**

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

lint prints messages about variables and functions which are defined but not otherwise mentioned, unless the message is suppressed by means of the **−u** or **−x** option.

Certain styles of programming may permit a function to be written with an interface where some of the function's arguments are optional. Such a function can be designed to accomplish a variety of tasks depending on which arguments are used. Normally **lint** prints messages about unused arguments; however, the **−v** option is available to suppress the printing of these messages. When **−v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
/* ARGSUSED */
```

to the source code before the function. This has the effect of the **−v** option for only one function. Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked. For example:

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

When **lint** is applied to some but not all files out of a collection that are to be loaded together, it issues complaints about unused or undefined variables. This information is, of course, more distracting than helpful. Functions and variables that are defined may not be used; conversely, functions and variables defined elsewhere may be used. The **−u** option suppresses the spurious messages.

Set/Used Information

lint attempts to detect cases where a variable is used before it is set. **lint** detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use" since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print error messages about program fragments that are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables that are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

lint attempts to detect unreachable portions of a program. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It attempts to detect loops that cannot be left at the bottom and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. **lint** also prints messages about loops that cannot be entered at the top. Valid programs may have such loops, but they are considered to be bad style. If you do not want messages about unreached portions of the program, use the **−b** option.

lint has no way of detecting functions that are called and never return. Thus, a call to **exit** may cause unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program is thought to be unreachable in a way that is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added to the source code at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached, and **lint** will not print a message about the unreachable portion.

Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements, but messages about them are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. The recommendation is to invoke **lint** with the **−b** option when dealing with such input.

Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function values that have never been returned. **lint** addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; **lint** will give the message

```
function name has return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) [
            if ( a ) return ( 3 );
            g ();
]
```

Notice that, if **a** tests false, **f** will call **g** and then return with no defined return value; this will trigger a message from **lint**. If **g**, like **exit**, never returns, the message will still be produced when in fact nothing is wrong. A comment

```
/*NOTREACHED*/
```

in the source code will cause the message to be suppressed. In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition that can be overcome by specifying the function as being of type (void). For example:

```
(void) fprintf(stderr,"File busy. Try again later!\n");
```

When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The opposite problem, using a function value when the function does not return one, is also detected. This is a serious problem.

Type Checking

lint enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- across certain binary operators and implied assignments
- at the structure selection operators
- between the definition and uses of functions

- in the use of enumerations

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char, short, int, long, unsigned, float,** and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*s can, of course, be intermixed with pointers to *x*s.

The type checking rules also require that, in structure references, the left operand of the **->** be a pointer to structure, the left operand of the **.** be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char, short, int,** and **unsigned.** Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment

```
/* NOSTRICT */
```

should be added to the source code immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where **p** is a character pointer. **lint** will print a message as a result of detecting this. Consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled his intentions. Nevertheless, **lint** will continue to print messages about this.

Nonportable Character Use

On some systems, characters are signed quantities with a range from −128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
        . . .
if( (c = getchar()) < 0 ) . . .
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually

returning integer values. In any case, **lint** will print the message

```
nonportable character comparison
```

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, which are truncated. The **–a** option can be used to suppress messages about the assignment of **longs** to **ints**.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **–h** option is used to suppress these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message

```
null effect
```

from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) . . .
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) . . .
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. **lint** will print the message

```
degenerate unsigned comparison
```

in these cases. If a program contains something similar to

```
if( 1 != 0 ) . . .
```

lint will print the message

```
constant in conditional context
```

since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) . . .
```

and

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (e.g., **=+**, **=-**, ...) could cause ambiguous expressions, such as:

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., **+=**, **-=**, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . . .
```

and the compiler must read past *x* in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. **lint** tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message

```
possible pointer alignment problem
```

results from this situation.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause **lint** to print the message

```
warning: i evaluation order undefined
```

in order to call attention to this condition.

Chapter 6: make

Introduction

The trend toward increased modularity of programs means that a project may have to cope with a large assortment of individual files. There may also be a wide range of generation procedures needed to turn the assortment of individual files into the final executable product.

make(1) provides a method for maintaining up-to-date versions of programs that consist of a number of files that may be generated in a variety of ways.

An individual programmer can easily forget

- file-to-file dependencies
- files that were modified and the impact that has on other files
- the exact sequence of operations needed to generate a new version of the program

In a description file, **make** keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that make up a program, the **make** command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

The basic operation of **make** is to

- find the target in the description file
- ensure that all the files on which the target depends, the files needed to generate the target, exist and are up to date
- create the target file if any of the generators have been modified more recently than the target

The description file that holds the information on interfile dependencies and command sequences is conventionally called **makefile**, **Makefile**, or **s.[mM]akefile**. If this naming convention is followed, the simple command **make** is usually sufficient to regenerate the target regardless of the number files edited since the last **make**. In most cases, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, rather than typing all the commands to regenerate the target, typing the **make** command ensures the regeneration is done in the prescribed way.

Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up to date. The target file is regenerated if it has not been modified since the dependents were modified. The **make** program searches the graph of dependencies. The operation of **make** depends on its ability to find the date and time that a file was last modified.

The **make** program operates using three sources of information:

- a user-supplied description file
- filenames and last-modified times from the file system
- built-in rules to bridge some of the gaps

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files **x.c**, **y.c**, and **z.c** with the **math** library. By convention, the output of the C language compilations will be found in files named **x.o**, **y.o**, and **z.o**. Assume that the files **x.c** and **y.c** share some declarations in a file named **defs.h**, but that **z.c** does not. That is, **x.c** and **y.c** have the line

```
#include "defs.h"
```

The following specification describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o   -lm  -o  prog

x.o  y.o :   defs.h
```

If this information were stored in a file named **makefile**, the command

make

would perform the operations needed to regenerate **prog** after any changes had been made to any of the four source files **x.c**, **y.c**, **z.c**, or **defs.h**. In the example above, the first line states that **prog** depends on three **.o** files. Once these object files are current, the second line describes how to load them to create **prog**. The third line states that **x.o** and **y.o** depend on the file **defs.h**. From the file system, **make** discovers that there are three **.c** files corresponding to the needed **.o** files and uses built-in rules on how to generate an object from a C source file (i.e., issue a **cc −c** command).

If **make** did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -o  prog
x.o :  x.c  defs.h
        cc  -c  x.c
y.o :  y.c  defs.h
        cc  -c  y.c
z.o :  z.c
        cc  -c  z.c
```

If none of the source or object files have changed since the last time **prog** was made, and all of the files are current, the command **make** announces this fact and stops. If, however, the **defs.h** file has been edited, **x.c** and **y.c** (but not **z.c**) are recompiled; and then **prog** is created from the new **x.o** and **y.o** files, and the existing **z.o** file. If only the file **y.c** had changed, only it is recompiled; but it is still necessary to reload **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

make x.o

would regenerate **x.o** if **x.c** or **defs.h** had changed.

A method often useful to programmers is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros (for information about macros, see "Description Files and Substitutions" below.) Thus, an entry "save" might be included to copy a certain set of files, or an entry "clean" might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

A simple macro mechanism for substitution in dependency lines and command strings is used by **make**. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by an equals sign followed by what the macro stands for. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be parenthesized. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two are equivalent.

$***, **$@, **$?**, and **$<** are four special macros that change values during the execution of the command. (These four macros are described later in this chapter under "Description Files and Substitutions.") The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lm
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
  . . .
```

The command

make LIBES="-ll -lm"

loads the three objects with both the **lex** (**-ll**) and the **math** (**-lm**) libraries, because macro definitions on the command line override definitions in the description file. (In

UNIX system commands, arguments with embedded blanks must be quoted.)

As an example of the use of **make**, a description file that might be used to maintain the **make** command itself is given. The code for **make** is spread over a number of C language source files and has a **yacc** grammar. The description file contains the following:

```
# Description file for the make command

FILES = Makefile defs.h main.c doname.c misc.c
        files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lld
LINT = lint -p
CFLAGS = -O
LP = /usr/bin/lp

make:  $(OBJECTS)
        $(CC) $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        @size make

$(OBJECTS):  defs.h

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make && rm make

lint :  dosys.c doname.c files.c main.c misc.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c \
        gram.c

            # print files that are out-of-date
            # with respect to "print" file.

print:      $(FILES)
      pr $? | $(LP)
      touch print
```

The **make** program prints out each command before issuing it.

The following output results from typing the command **make** in a directory containing only the source and description files:

```
cc  -O -c main.c
cc  -O -c doname.c
cc  -O -c misc.c
cc  -O -c files.c
cc  -O -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -O -c gram.c
cc  main.o doname.o misc.o files.o dosys.o
     gram.o  -lld -o make
13188 + 3348 + 3044 = 19580
```

The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an at sign, **@**, in the description file.

Description Files and Substitutions

The following section will explain the customary elements of the description file.

Comments

The comment convention is that a sharp, #, and all characters on the same line after a sharp are ignored. Blank lines and lines beginning with a sharp are totally ignored.

Continuation Lines

If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

Macro Definitions

A macro definition is an identifier followed by an equal sign. The identifier must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **make**'s own rules. (See Figure 6-2 at the end of the chapter.)

General Form

The general form of an entry in a description file is

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands]
      [# ...] [ \t commands] [# ...]
  . . .
```

Items inside brackets may be omitted and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as * and ? are expanded when the line is evaluated. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a sharp, #, except when the sharp is in quotes.

Dependency Information

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the more common single-colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule may be invoked. In the double-colon case, a command sequence may be associated with more than one dependency line. If the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double colon form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in the "Archive Libraries" section later in this chapter.)

Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (**-s** option of the **make** command) or if the command line in the description file begins with an **@** sign. **make** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the **-i** flag has been specified on the **make** command line, if the fake target name..IGNORE appears in the description file, or if the command string in the description file begins with a hyphen. If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The **$@** macro is set to the full target name of the current target. The **$@** macro is evaluated only for explicitly named dependencies. The $? macro is set to the string of names that were found to be younger than the target. The $? macro is evaluated when explicit rules from the **makefile** are evaluated. If the command was generated by an implicit rule, the **$<** macro is the name of the related file that caused the action; and the **$*** macro is the prefix shared by the current and the dependent filenames. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name DEFAULT are used. If there is no such name, **make** prints a message and stops.

In addition, a description file may also use the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)** (see below).

Extensions of $*, $@, and $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The **D** refers to the directory part of the single character macro. The **F** refers to the filename part of the single character macro. These additions are useful when building hierarchical

makefiles. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a command can be

cd $(<D); $(MAKE) $(<F)

Output Translations

Macros in shell commands are translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of **string1** in the evaluated macro, **string2** is substituted. The meaning of finding **string1** in **$(macro)** is that the evaluated **$(macro)** is considered as a series of strings each delimited by white space (blanks or tabs). Thus, the occurrence of **string1** in **$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script, which can handle all the C language programs (i.e., those files ending in **.c**). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the wealth of information that **make** generates.

Recursive makefiles

Another feature of **make** concerns the environment and recursive invocations. If the sequence $(MAKE) appears anywhere in a shell command line, the line is executed even if the **−n** flag is set. Since the **−n** flag is exported across invocations of **make** (through the MAKEFLAGS variable), the only thing that is executed is the **make** command itself. This feature is useful when a hierarchy of **makefile**(s) describes a set of software subsystems. For testing purposes, **make −n ...** can be executed and everything that would have been done will be printed including output from lower level invocations of **make**.

Suffixes and Transformation Rules

make uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the **−r** flag is used on the **make** command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name .SUFFIXES. **make** searches for a file with any of the suffixes on the list. If it finds one, **make** transforms it into a file with another suffix. The transformation rule names are the concatenation of the before and after suffixes. The name of the rule to transform a **.r** file to a **.o** file is thus **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro **$*** is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro **$<** is the full name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for .SUFFIXES in the description file. The dependents are added to the usual list. A .SUFFIXES line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

Implicit Rules

make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o Object file

.c C source file

.c~ SCCS C source file

.f FORTRAN source file

.f~ SCCS FORTRAN source file

.s Assembler source file

.s~ SCCS Assembler source file

.y **yacc** source grammar

.y~ SCCS **yacc** source grammar

.l **lex** source grammar

.l~ SCCS **ex** source grammar

.h Header file

.h~ SCCS header file

.sh Shell file

.sh~ SCCS shell file

Figure 6-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

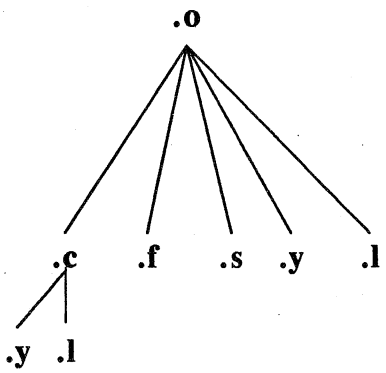


Figure 6-1: Summary of Default Transformation Path

If the file **x.o** is needed and an **x.c** is found in the description or directory, the **x.o** file would be compiled. If there is also an **x.l**, that source file would be run through **lex** before compiling the result. However, if there is no **x.c** but there is an **x.l**, **make** would discard the intermediate C language file and use the direct link as shown in Figure 6-1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, F77, YACC, and LEX. The command

make CC=newcc

will cause the **newcc** command to be used instead of the usual C language compiler. The macros ASFLAGS, CFLAGS, F77FLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus

make "CFLAGS=-g"

causes the **cc** command to include debugging information.

Archive Libraries

The **make** program has an interface to archive libraries. A user may name a member of a library in the following manner:

```
projlib(object.o)
   or
projlib((entrypt))
```

where the second method actually refers to an entry point of an object file within the library. (**make** looks through the library, locates the entry point, and translates it to the correct object filename.)

To use this procedure to maintain an archive library, the following type of **makefile** is required:

```
projlib::   projlib(pfile1.o)
        $(CC) -c -O pfile1.c
        $(AR) $(ARFLAGS) projlib pfile1.o
        rm pfile1.o
projlib::   projlib(pfile2.o)
        $(CC) -c -O pfile2.c
        $(AR) $(ARFLAGS) projlib pfile2.o
        rm pfile2.o
```

... and so on for each object ...

This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the filename being the only difference each time. (This is true in most cases.)

The **make** command also gives the user access to a rule for building libraries. The handle for the rule is the **.a** suffix. Thus, a **.c.a** rule is the rule for compiling a C language source file, adding it to the library, and removing the **.o** cadaver. Similarly, the **.y.a**, the **.s.a**, and the **.l.a** rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are **.c.a**, **.c~.a**, **.f.a**, **.f~.a**, and **.s~.a**. (The tilde, ~, syntax will be described shortly.) The user may define other needed rules in the description file.

The above two-member library is then maintained with the following shorter **makefile**:

```
projlib:        projlib(pfile1.o) projlib(pfile2.o)
            @echo projlib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rule is as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
```

Thus, the **$@** macro is the **.a** target (**projlib**); the **$<** and **$*** macros are set to the out-of-date C language file; and the filename minus the suffix, respectively (**pfile1.c** and **pfile1**). The **$<** macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees the construction

```
projlib:    projlib(pfile1.o)
        @echo projlib up-to-date
```

Assume the object in the library is out of date with respect to **pfile1.c**. Also, there is no **pfile1.o** file.

1. **make projlib**.
2. Before **make**ing **projlib**, check each dependent of **projlib**.
3. **projlib(pfile1.o)** is a dependent of **projlib** and needs to be generated.
4. Before generating **projlib(pfile1.o)**, check each dependent of **projlib(pfile1.o)**. (There are none.)
5. Use internal rules to try to create **projlib(pfile1.o)**. (There is no explicit rule.) Note that **projlib(pfile1.o)** has a parenthesis in the name to identify the target suffix as **.a**. This is the key. There is no explicit **.a** at the end of the **projlib** library name. The parenthesis implies the **.a** suffix. In this sense, the **.a** is hard-wired into **make**.
6. Break the name **projlib(pfile1.o)** up into **projlib** and **pfile1.o**. Define two macros, **$@** (=**projlib**) and **$*** (=**pfile1**).
7. Look for a rule *.X*.**a** and a file **$***.*X*. The first *.X* (in the .SUFFIXES list) which fulfills these conditions is **.c** so the rule is **.c.a**, and the file is **pfile1.c**. Set **$<** to be **pfile1.c** and execute the rule. In fact, **make** must then compile **pfile1.c**.
8. The library has been updated. Execute the command associated with the **projlib:** dependency; namely

   ```
   @echo projlib up-to-date
   ```

It should be noted that to let **pfile1.o** have dependencies, the following syntax is required:

```
projlib(pfile1.o):      $(INCDIR)/stdio.h  pfile1.c
```

There is also a macro for referencing the archive member name when this form is used. The **$%** macro is evaluated each time **$@** is evaluated. If there is no current archive member, **$%** is null. If an archive member exists, then **$%** evaluates to the expression between the parenthesis.

Source Code Control System Filenames: the Tilde

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX operating system machines, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, **s.** precedes the filename part of the complete path name.

To allow **make** easy access to the prefix **s.** the tilde, ~, is used as an identifier of SCCS files. Hence, **.c~.o** refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the internal rule is

```
.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS filename search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

.c~
.f~
.y~
.l~
.s~
.sh~
.h~

The following rules involving SCCS transformations are internally defined:

.c~:
.f~:
.sh~:
.c~.a:
.c~.c:
.c~.o:
.f~.a:
.f~.f:
.f~.o:
.s~.a:
.s~.s:
.s~.o:
.y~.c:
.y~.o:
.l~.l:
.l~.o:
.h~.h:

Obviously, the user can define other rules and suffixes, which may prove useful. The tilde provides a handle on the SCCS filename format so that this is possible.

The Null Suffix

There are many programs that consist of a single source file. **make** handles this case by the null suffix rule. Thus, to maintain the UNIX system program **cat**, a rule in the **makefile** of the following form is needed:

```
.c:
        $(CC) $(CFLAGS) $< -o $@
```

In fact, this **.c:** rule is internally defined so no **makefile** is necessary at all. The user only needs to type

make cat dd echo date

(these are all UNIX system single-file programs) and all four C language source files are passed through the above shell command line associated with the **.c:** rule. The internally defined single suffix rules are

.c:
.c~:
.f:
.f~:
.sh:
.sh~:

Others may be added in the **makefile** by the user.

include Files

The **make** program has a capability similar to the **#include** directive of the C preprocessor. If the string **include** appears as the first seven letters of a line in a **makefile** and is followed by a blank or a tab, the rest of the line is assumed to be a filename, which the current invocation of **make** will read. Macros may be used in filenames. The file descriptors are stacked for reading **include** files so that no more than 16 levels of nested **includes** are supported.

SCCS Makefiles

Makefiles under SCCS control are accessible to **make**. That is, if **make** is typed and only a file named **s.makefile** or **s.Makefile** exists, **make** will do a **get** on the file, then read and remove the file.

Dynamic Dependency Parameters

The parameter has meaning only on the dependency line in a makefile. The **$$@** refers to the current "thing" to the left of the colon (which is **$@**). Also the form **$$(@F)** exists, which allows access to the file part of **$@**. Thus, in the following:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string **cat.c**. This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX software command directory could have a **makefile** like:

```
CMDS = cat dd echo date cmp comm chown

$(CMDS):        $$@.c
        $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate **makefile** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the **makefile.**

The second useful form of the dependency parameter is **$$(@F)**. It represents the filename part of **$$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the **/usr/include** directory from a makefile in the **/usr/src/head** directory. Thus, the **/usr/src/head/makefile** would look like

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the **/usr/include** directory whenever one of the above files in **/usr/src/head** was updated.

Command Usage

The **make** command description is found under **make**(1) in the *User's Reference Manual.*

The make Command

The **make** command takes macro definitions, options, description filenames, and target filenames as arguments in the form:

make [*options*] [*macro definitions*] [*targets*]

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The permissible options are as follows:

−i Ignore error codes returned by invoked commands. This mode is entered if the fake target name .IGNORE appears in the description file.

−s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name .SILENT appears in the description file.

−r Do not use the built-in rules.

−n No execute mode. Print commands, but do not execute them. Even lines beginning with an **@** sign are printed.

−t Touch the target files (causing them to be up to date) rather than issue the usual commands.

−q Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up to date.

−p Print out the complete set of macro definitions and target descriptions.

−k Abandon work on the current entry if something goes wrong, but continue on other branches that do not depend on the current entry.

−e Environment variables override assignments within **makefiles**.

−f Description filename. The next argument is assumed to be the name of a description file. A filename of **−** denotes the standard input. If there are no **−f** arguments, the file named **makefile** or **Makefile** or **s.[mM]akefile** in the current directory is read. The contents of the description files override the built-in rules if they are present.

The following two arguments are evaluated in the same manner as flags:

.DEFAULT If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists.

.PRECIOUS Dependents on this target are not removed when quit or interrupt is pressed.

Finally, the remaining arguments are assumed to be the names of targets to be made and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with a period is made.

Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment. A macro, MAKEFLAGS, is maintained by **make**. The macro is defined as the collection of all input flag arguments into a string (without minus signs). The macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the **makefile** update MAKEFLAGS. Thus, to describe how the environment interacts with **make**, the MAKEFLAGS macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the MAKEFLAGS environment variable. If it is not present or null, the internal **make** variable MAKEFLAGS is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the **−f**, **−p**, and **−r** flags.)
2. Read the internal list of macro definitions.
3. Read the environment. The environment variables are treated as macro definitions and marked as **exported** (in the shell sense).
4. Read the **makefile**(s). The assignments in the **makefile**(s) overrides the environment. This order is chosen so that when a **makefile** is read and executed, you know what to expect. That is, you get what is seen unless the **−e** flag is used. The **−e** is the line flag, which tells **make** to have the environment override the **makefile** assignments. Thus, if **make −e ...** is typed, the variables in the environment override the definitions in the **makefile**. Also MAKEFLAGS override the environment if assigned. This is useful for further invocations of **make** from the current **makefile**.

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions
2. environment
3. **makefile**(s)
4. command line

The **−e** flag has the effect of rearranging the order to:

1. internal definitions
2. **makefile**(s)
3. environment
4. command line

This order is general enough to allow a programmer to define a **makefile** or set of **makefiles** whose parameters are dynamically definable.

Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of dependency. If file **x.c** has a

```
#include "defs.h"
```

line, then the object file **x.o** depends on **defs.h**; the source file **x.c** does not. If **defs.h** is changed, nothing is done to the file **x.c** while file **x.o** must be recreated.

To discover what **make** would do, the **−n** option is very useful. The command

make −n

orders **make** to print out the commands that **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a comment to an **include** file), the **−t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

make −ts

(touch silently) causes the relevant files to appear up to date. Obvious care is necessary because this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

Internal Rules

The standard set of internal rules used by **make** are reproduced below.

```
#
#       SUFFIXES RECOGNIZED BY MAKE
#

.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .h .h~ .sh .sh~ .f .f~

#
#       PREDEFINED MACROS
#

MAKE=make
AR=ar
ARFLAGS=-rv
AS=as
ASFLAGS=
CC=cc
CFLAGS=-O
F77=f77
F77FLAGS=
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
YACC=yacc
YFLAGS=
```

Figure 6-2: **make** Internal Rules (Sheet 1 of 4)

```
#
#       SINGLE SUFFIX RULES
#

.c:
        $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
.c~:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) $(LDFLAGS) $*.c -o $*
        -rm -f $*.c

.f:
        $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $@
.f~:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $*
        -rm -f $*.f

.sh:
        cp $< $@; chmod 0777 $@
.sh~:
        $(GET) $(GFLAGS) $<
        cp $*.sh $*; chmod 0777 $@
        -rm -f $*.sh
```

Figure 6-2: **make** Internal Rules (Sheet 2 of 4)

```
#
#       DOUBLE SUFFIX RULES
#
.c~.c  .f~.f  .s~.s  .sh~.sh  .y~.y  .l~.l  .h~.h:
        $(GET) $(GFLAGS) $<

.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o

.c~.a:
        $(GET) $(GFLAGS) $<
        $(CC) -c $(CFLAGS) $*.c
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.[co]

.c.o:
        $(CC) $(CFLAGS) -c $<

.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c

.f.a:
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.o

.f~.a:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.[fo]

.f.o:
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f

.f~.o:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        -rm -f $*.f

.s~.a:
        $(GET) $(GFLAGS) $<
        $(AS) $(ASFLAGS) -o $*.o $*.s
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.[so]
```

Figure 6-2: **make** Internal Rules (Sheet 3 of 4)

```
.s.o:
	$(AS) $(ASFLAGS) -o $@ $<
.s~.o:
	$(GET) $(GFLAGS) $<
	$(AS) $(ASFLAGS) -o $*.o $*.s
	-rm -f $*.s
.l.c :
	$(LEX) $(LFLAGS) $<
	mv lex.yy.c $@
.l~.c:
	$(GET) $(GFLAGS) $<
	$(LEX) $(LFLAGS) $*.l
	mv lex.yy.c $@
.l.o:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	rm lex.yy.c
	mv lex.yy.o $@
	-rm -f $*.l
.l~.o:
	$(GET) $(GFLAGS) $<
	$(LEX) $(LFLAGS) $*.l
	$(CC) $(CFLAGS) -c lex.yy.c
	rm -f lex.yy.c $*.l
	mv lex.yy.o $*.o
.y.c :
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@
.y~.c :
	$(GET) $(GFLAGS) $<
	$(YACC) $(YFLAGS) $*.y
	mv y.tab.c $*.c
	-rm -f $*.y
.y.o:
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) -c y.tab.c
	rm y.tab.c
	mv y.tab.o $@
.y~.o:
	$(GET) $(GFLAGS) $<
	$(YACC) $(YFLAGS) $*.y
	$(CC) $(CFLAGS) -c y.tab.c
	rm -f y.tab.c $*.y
	mv y.tab.o $*.o
```

Figure 6-2: **make** Internal Rules (Sheet 4 of 4)

Chapter 7: Source Code Control System (SCCS)

Introduction

The Source Code Control System (SCCS) is a file maintenance and enhancement tracking tool that runs under the UNIX system. SCCS takes custody of a file and, when changes are made, identifies and stores them in the file with the original source code and/or documentation. As other changes are made, they too are identified and retained in the file.

Retrieval of the original or any set of changes is possible. Any version of the file as it develops can be reconstructed for inspection or additional modification. History data can be stored with each version: why the changes were made, who made them, when they were made.

This guide covers the following:

- SCCS for Beginners: how to make, retrieve, and update an SCCS file
- Delta Numbering: how versions of an SCCS file are named
- SCCS Command Conventions: what rules apply to SCCS commands
- SCCS Commands: the fourteen SCCS commands and their more useful arguments
- SCCS Files: protection, format, and auditing of SCCS files

Neither the implementation of SCCS nor the installation procedure for SCCS is described in this guide.

SCCS For Beginners

Several terminal session fragments are presented in this section. Try them all. The best way to learn SCCS is to use it.

Terminology

A delta is a set of changes made to a file under SCCS custody. To identify and keep track of a delta, it is assigned an SID (SCCS IDentification) number. The SID for any original file turned over to SCCS is composed of release number 1 and level number 1, stated as 1.1. The SID for the first set of changes made to that file, that is, its first delta is release 1 version 2, or 1.2. The next delta would be 1.3, the next 1.4, and so on. More on delta numbering later. At this point, it is enough to know that by default SCCS assigns SIDs automatically.

Creating an SCCS File via admin

Suppose, for example, you have a file called **lang** that is simply a list of five programming language names. Use a text editor to create file **lang** containing the following list.

```
C
PL/1
FORTRAN
COBOL
ALGOL
```

Custody of your **lang** file can be given to SCCS using the **admin** command (i.e., administer SCCS file). The following creates an SCCS file from the **lang** file:

```
admin -ilang s.lang
```

All SCCS files must have names that begin with **s.**, hence **s.lang**. The **-i** keyletter, together with its value **lang**, means **admin** is to create an SCCS file and initialize it with the contents of the file **lang**.

The **admin** command replies

```
No id keywords (cm7)
```

This is a warning message that may also be issued by other SCCS commands. Ignore it for now. Its significance is described later with the **get** command under "SCCS Commands." In the following examples, this warning message is not shown although it may be issued.

Remove the **lang** file. It is no longer needed because it exists now under SCCS as **s.lang**.

```
rm lang
```

Retrieving a File via get

Use the **get** command as follows:

get s.lang

This retrieves **s.lang** and prints

```
1.1
5 lines
```

This tells you that **get** retrieved version 1.1 of the file, which is made up of five lines of text.

The retrieved text has been placed in a new file known as a "g.file." SCCS forms the g.file name by deleting the prefix **s.** from the name of the SCCS file. Thus, the original **lang** file has been recreated.

If you list, **ls**(1), the contents of your directory, you will see both **lang** and **s.lang**. SCCS retains **s.lang** for use by other users.

The **get s.lang** command creates **lang** as read-only and keeps no information regarding its creation. Because you are going to make changes to it, **get** must be informed of your intention to do so. This is done as follows:

get −e s.lang

get −e causes SCCS to create **lang** for both reading and writing (editing). It also places certain information about **lang** in another new file, called the "p.file" (**p.lang** in this case), which is needed later by the **delta** command.

get −e prints the same messages as **get**, except that now the SID for the first delta you will create is issued:

```
1.1
new delta  1.2
5 lines
```

Change **lang** by adding two more programming languages:

SNOBOL
ADA

Recording Changes via delta

Next, use the **delta** command as follows:

delta s.lang

delta then prompts with

```
comments?
```

Your response should be an explanation of why the changes were made. For example,

added more languages

delta now reads the p.file, **p.lang**, and determines what changes you made to **lang**. It does this by doing its own **get** to retrieve the original version and applying the **diff**(1) command to the original version and the edited version. Next, **delta** stores the changes in **s.lang** and destroys the no longer needed **p.lang** and **lang** files.

When this process is complete, **delta** outputs

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number 1.2 is the SID of the delta you just created, and the next three lines summarize what was done to **s.lang**.

Additional Information about get

The command,

get s.lang

retrieves the latest version of the file **s.lang**, now 1.2. SCCS does this by starting with the original version of the file and applying the delta you made. If you use the **get** command now, any of the following will retrieve version 1.2.

get s.lang
get −r1 s.lang
get −r1.2 s.lang

The numbers following **−r** are SIDs. When you omit the level number of the SID (as in **get −r1 s.lang**), the default is the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case also 1.2.

Whenever a major change is made to a file, you may want to signify it by changing the release number, the first number of the SID. This, too, is done with the **get** command.

get −e −r2 s.lang

Because release 2 does not exist, **get** retrieves the latest version before release 2. **get** also interprets this as a request to change the release number of the new delta to 2, thereby naming it 2.1 rather than 1.3. The output is

```
1.2
new delta 2.1
7 lines
```

which means version 1.2 has been retrieved, and 2.1 is the version **delta** will create. If the file is now edited, for example, by deleting COBOL from the list of languages, and **delta** is executed

delta s.lang
comments? **deleted cobol from list of languages**

you will see by **delta**'s output that version 2.1 is indeed created.

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas can now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release can be created in a similar manner.

The help Command

If the command

get lang

is now executed, the following message will be output:

```
ERROR [lang]: not an SCCS file (co1)
```

The code **co1** can be used with **help** to print a fuller explanation of the message.

help co1

This gives the following explanation of why **get lang** produced an error message:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

help is useful whenever there is doubt about the meaning of almost any SCCS message.

Delta Numbering

Think of deltas as the nodes of a tree in which the root node is the original version of the file. The root is normally named 1.1 and deltas (nodes) are named 1.2, 1.3, etc. The components of these SIDs are called release and level numbers, respectively. Thus, normal naming of new deltas proceeds by incrementing the level number. This is done automatically by SCCS whenever a delta is made.

Because the user may change the release number to indicate a major change, the release number then applies to all new deltas unless specifically changed again. Thus, the evolution of a particular file could be represented by Figure 7-1.

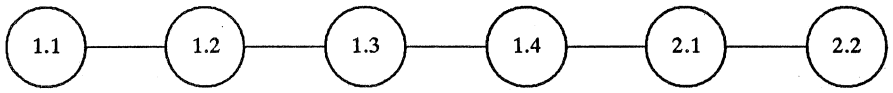


Figure 7-1: Evolution of an SCCS File

This is the normal sequential development of an SCCS file, with each delta dependent on the preceding deltas. Such a structure is called the trunk of an SCCS tree.

There are situations that require branching an SCCS tree. That is, changes are planned to a given delta that will not be dependent on all previous deltas. For example, consider a program in production use at version 1.3 and for which development work on release 2 is already in progress. Release 2 may already have a delta in progress as shown in Figure 7-1. Assume that a production user reports a problem in version 1.3 that cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.). This new delta is the first node of a new branch of the tree.

Branch delta names always have four SID components: the same release number and level number as the trunk delta, plus a branch number and sequence number. The format is as follows:

release.level.branch.sequence

The branch number of the first delta branching off any trunk delta is always 1, and its sequence number is also 1. For example, the full SID for a delta branching off trunk delta 1.3 will be 1.3.1.1. As other deltas on that same branch are created, only the sequence number changes: 1.3.1.2, 1.3.1.3, etc. This is shown in Figure 7-2.

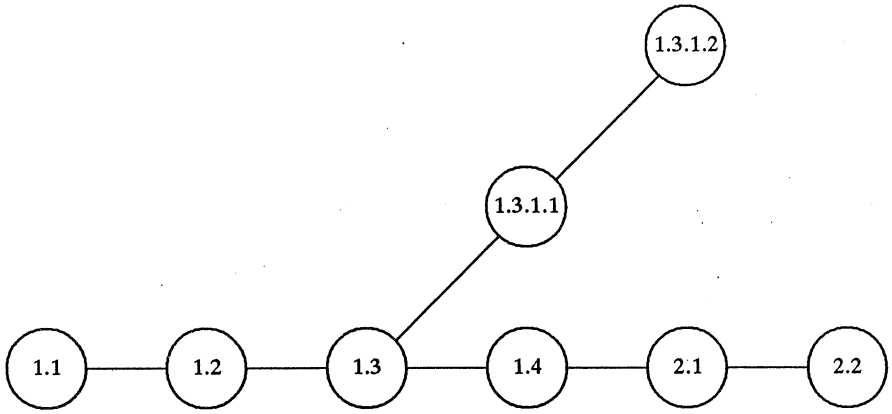


Figure 7-2: Tree Structure with Branch Deltas

The branch number is incremented only when a delta is created that starts a new branch off an existing branch, as shown in Figure 7-3. As this secondary branch develops, the sequence numbers of its deltas are incremented (1.3.2.1, 1.3.2.2, etc.), but the secondary branch number remains the same.

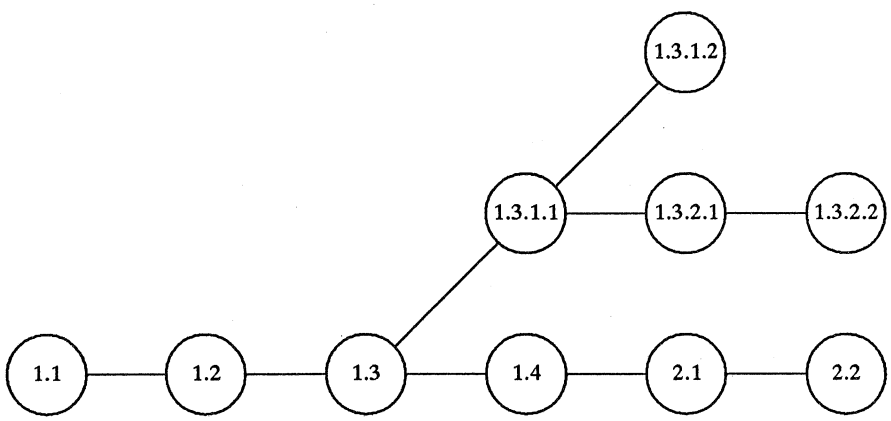


Figure 7-3: Extended Branching Concept

The concept of branching may be extended to any delta in the tree, and the numbering of the resulting deltas proceeds as shown above. SCCS allows the generation of complex tree structures. Although this capability has been provided for certain specialized uses, the SCCS tree should be kept as simple as possible. Comprehension of its structure becomes difficult as the tree becomes complex.

SCCS Command Conventions

SCCS commands accept two types of arguments:

- keyletters
- filenames

Keyletters are options that begin with a minus sign, **−**, followed by a lowercase letter and, in some cases, a value.

File and/or directory names specify the file(s) the command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files (because of permission modes via **chmod**(1)) in the named directories are silently ignored.

In general, filename arguments may not begin with a minus sign. If a filename of **−** (a lone minus sign) is specified, the command will read the standard input (usually your terminal) for lines and take each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the commands **find**(1) or **ls**(1).

Keyletters are processed before filenames. Therefore, the placement of keyletters is arbitrary—that is, they may be interspersed with filenames. Filenames, however, are processed left to right. Somewhat different conventions apply to **help**(1), **what**(1), **sccsdiff**(1), and **val**(1), detailed later under "SCCS Commands."

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags will be discussed, but for a complete description see **admin**(1) in the *User's Reference Manual.*

The distinction between real user (see **passwd**(1)) and effective user will be of concern in discussing various actions of SCCS commands. For now, assume that the real and effective users are the same—the person logged into the UNIX system.

x.files and z.files

All SCCS commands that modify an SCCS file do so by writing a copy called the "x.file." This is done to ensure that the SCCS file is not damaged if processing terminates abnormally. SCCS names the x.file by replacing the **s.** of the SCCS filename with **x.**. The x.file is created in the same directory as the SCCS file, given the same mode (see **chmod**(1)), and is owned by the effective user. When processing is complete, the old SCCS file is destroyed and the modified x.file is renamed (**x.** is relaced by **s.**) and becomes the new SCCS file.

To prevent simultaneous updates to an SCCS file, the same modifying commands also create a lock-file called the "z.file." SCCS forms its name by replacing the **s.** of the SCCS filename with a **z.** prefix. The z.file contains the process number of the command that creates it, and its existence prevents other commands from processing the SCCS file. The z.file is created with access permission mode 444 (read only) in the same directory as the SCCS file and is owned by the effective user. It exists only for the duration of the execution of the command that creates it.

In general, users can ignore x.files and z.files. They are useful only in the event of system crashes or similar situations.

Error Messages

SCCS commands produce error messages on the diagnostic output in this format:

```
ERROR [name-of-file-being-processed]: message text (code)
```

The code in parentheses can be used as an argument to the **help** command to obtain a further explanation of the message. Detection of a fatal error during the processing of a file causes the SCCS command to stop processing that file and proceed with the next file specified.

SCCS Commands

This section describes the major features of the fourteen SCCS commands and their most common arguments. Full descriptions with details of all arguments are in the *User's Reference Manual*.

Here is a quick-reference overview of the commands:

get	retrieves versions of SCCS files
unget	undoes the effect of a **get −e** prior to the file being **deltaed**
delta	applies deltas (changes) to SCCS files and creates new versions
admin	initializes SCCS files, manipulates their descriptive text, and controls delta creation rights
prs	prints portions of an SCCS file in user specified format
sact	prints information about files that are currently out for edit
help	gives explanations of error messages
rmdel	removes a delta from an SCCS file allows removal of deltas created by mistake
cdc	changes the commentary associated with a delta
what	searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it useful in finding identifying information inserted by the **get** command
sccsdiff	shows differences between any two versions of an SCCS file
comb	combines consecutive deltas into one to reduce the size of an SCCS file
val	validates an SCCS file
vc	a filter that may be used for version control

The get Command

The **get**(1) command creates a file that contains a specified version of an SCCS file. The version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The resulting file is called the "g.file." It is created in the current directory and is owned by the real user. The mode assigned to the g.file depends on how the **get** command is used.

The most common use of **get** is

```
get s.abc
```

which normally retrieves the latest version of file **abc** from the SCCS file tree trunk and produces (for example) on the standard output

```
1.3
67 lines
No id keywords (cm7)
```

meaning version 1.3 of file **s.abc** was retrieved (assuming 1.3 is the latest trunk delta), it has 67 lines of text, and no ID keywords were substituted in the file.

The generated g.file (file **abc**) is given access permission mode 444 (read only). This particular way of using **get** is intended to produce g.files only for inspection, compilation, etc. It is not intended for editing (making deltas).

When several files are specified, the same information is output for each one. For example,

```
get s.abc s.xyz
```

produces

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.xyz:
1.7
85 lines
No id keywords (cm7)
```

ID Keywords

In generating a g.file for compilation, it is useful to record the date and time of creation, the version retrieved, the module's name, etc. within the g.file. This information appears in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of those ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs, %. For example,

```
%I%
```

is the ID keyword replaced by the SID of the retrieved version of a file. Similarly, **%H%** and **%M%** are the names of the g.file. Thus, executing **get** on an SCCS file that contains the PL/I declaration,

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100 VAR INIT('MODNAME 2.3 07/18/85');
```

When no ID keywords are substituted by **get**, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by **get** although the presence of the **i** flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately twenty ID keywords provided, see **get**(1) in the *User's Reference Manual*.

Retrieval of Different Versions

The version of an SCCS file **get** retrieves is the most recently created delta of the highest numbered trunk release. However, any other version can be retrieved with **get −r** by specifying the version's SID. Thus,

```
get −r1.3 s.abc
```

retrieves version 1.3 of file **s.abc** and produces (for example) on the standard output

```
1.3
64 lines
```

A branch delta may be retrieved similarly,

get −r1.5.2.3 s.abc

which produces (for example) on the standard output

```
1.5.2.3
234 lines
```

When a SID is specified and the particular version does not exist in the SCCS file, an error message results.

Omitting the level number, as in

get −r3 s.abc

causes retrieval of the trunk delta with the highest level number within the given release. Thus, the above command might output,

```
3.7
213 lines
```

If the given release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assume release 9 does not exist in file **s.abc** and release 7 is the highest-numbered release below 9. Executing

get −r9 s.abc

might produce

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file **s.abc** below release 9. Similarly, omitting the sequence number, as in

get −r4.3.2 s.abc

results in the retrieval of the branch delta with the highest sequence number on the given branch. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

get −t will retrieve the latest (top) version of a particular release when no **−r** is used or when its value is simply a release number. The latest version is the delta produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

get −r3 −t s.abc

might produce

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce

```
3.2.1.5
46 lines
```

Retrieval With Intent to Make a Delta

get −e indicates an intent to make a delta. First, **get** checks the following.

1. The user list to determine if the login name or group ID of the person executing **get** is present. The login name or group ID must be present for the user to be allowed to make deltas. (See "The **admin** Command" for a discussion of making user lists.)
2. The release number (R) of the version being retrieved satisfies the relation

   floor is less than or equal to R, which is
   less than or equal to ceiling

   to determine if the release being accessed is a protected release. The floor and ceiling are flags in the SCCS file representing start and end of range.
3. The R is not locked against editing. The lock is a flag in the SCCS file.
4. Whether multiple concurrent edits are allowed for the SCCS file by the **j** flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, **get −e** causes the creation of a g.file in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable g.file already exists, **get** terminates with an error. This is to prevent inadvertent destruction of a g.file being edited for the purpose of making a delta.

Any ID keywords appearing in the g.file are not substituted by **get −e** because the generated g.file is subsequently used to create another delta. Replacement of ID keywords causes them to be permanently changed in the SCCS file. Because of this, **get** does not need to check for their presence in the g.file. Thus, the message

```
No id keywords (cm7)
```

is never output when **get −e** is used.

In addition, **get −e** causes the creation (or updating) of a p.file that is used to pass information to the **delta** command.

The following

```
get −e s.abc
```

produces (for example) on the standard output

```
1.3
new delta 1.4
67 lines
```

Undoing a get −e

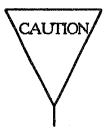
There may be times when a file is retrieved for editing in error; there is really no editing that needs to be done at this time. In such cases, the **unget** command can be used to cancel the delta reservation that was set up.

Additional get Options

If **get −r** and/or **−t** are used together with **−e**, the version retrieved for editing is the one specified with **−r** and/or **−t**.

get −i and **−x** are used to specify a list (see **get**(1) in the *User's Reference Manual* for the syntax of such a list) of deltas to be included and excluded, respectively. Including a delta means forcing its changes to be included in the retrieved version. This is useful in applying the same changes to more than one version of the SCCS file. Excluding a delta means forcing it not to be applied. This may be used to undo the effects of a previous delta in the version to be created.

Whenever deltas are included or excluded, **get** checks for possible interference with other deltas. Two deltas can interfere, for example, when each one changes the same line of the retrieved g.file. A warning shows the range of lines within the retrieved g.file where the problem may exist. The user should examine the g.file to determine what the problem is and take appropriate corrective steps (e.g., edit the file).



get −i and **get −x** should be used with extreme care.

get −k is used either to regenerate a g.file that may have been accidentally removed or ruined after **get −e**, or simply to generate a g.file in which the replacement of ID keywords has been suppressed. A g.file generated by **get −k** is identical to one produced by **get −e**, but no processing related to the p.file takes place.

Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows several deltas to be in progress at any given time. This means that several **get −e** commands may be executed on the same file as long as no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The p.file created by **get −e** is named by automatic replacement of the SCCS filename's prefix **s.** with **p.**. It is created in the same directory as the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The p.file contains the following information for each delta that is still in progress:

- the SID of the retrieved version
- the SID given to the new delta when it is created
- the login name of the real user executing **get**

The first execution of **get −e** causes the creation of a p.file for the corresponding SCCS file. Subsequent executions only update the p.file with a line containing the above information. Before updating, however, **get** checks to assure that no entry already in the p.file specifies that the SID of the version to be retrieved is already retrieved (unless multiple concurrent edits are allowed). If the check succeeds, the user is informed that other deltas are in progress and processing continues. If the check fails, an error message results.

It should be noted that concurrent executions of **get** must be carried out from different directories. Subsequent executions from the same directory will attempt to overwrite the g.file, which is an SCCS error condition. In practice, this problem does not arise since each user normally has a different working directory. See "Protection" under "SCCS Files" for a discussion of how different users are permitted to use SCCS commands on the same files.

Figure 7-4 shows the possible SID components a user can specify with **get** (left-most column), the version that will then be retrieved by **get**, and the resulting SID for the delta, which **delta** will create (right-most column).

SID Specified in get*	–b Key-Letter Used†	Other Conditions	SID Retrieved by get	SID of Delta To be Created by delta
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1)
R	no	R > mR	mR.mL	R.1§
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R< mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk successor number in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunk successor	R.L	R.L.(mB+1).1

Figure 7-4: Determination of New SID (sheet 1 of 2)

SID Specified in get*	-b Key-Letter used†	Other Condition	SID Retrieved by get	SID of Delta to be Created by delta
R.L	–	Trunk successor in release ≥ R	R.L	R.L.(mS+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch successor	R.L.B.S	R.L.(mB+1).1

Figure 7-4: Determination of New SID (sheet 2 of 2)

Footnotes to Figure 7-4:

* R, L, B, and S mean release, level, branch, and sequence numbers in the SID, and m means maximum. Thus, for example, R.mL means the maximum level number within release R. R.L.(mB+1).1 means the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R. Note that if the SID specified is R.L, R.L.B, or R.L.B.S, each of these specified SID numbers must exist.

† The **–b** keyletter is effective only if the **b** flag (see **admin(1)**) is present in the file. An entry of – means irrelevant.

‡ This case applies if the **d** (default SID) flag is not present. If the **d** flag is present in the file, the SID is interpreted as if specified on the command line. Thus, one of the other cases in this figure applies.

§ This is used to force the creation of the first delta in a new release.

** hR is the highest existing release that is lower than the specified, nonexistent release R.

Concurrent Edits of Same SID

Under normal conditions, more than one **get –e** for the same SID is not permitted. That is, **delta** must be executed before a subsequent **get –e** is executed on the same SID.

Multiple concurrent edits are allowed if the **j** flag is set in the SCCS file. Thus:

```
get -e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by

```
get -e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening **delta**. In this case, a **delta** after the first **get** will produce delta 1.2 (assuming 1.1 is the most recent trunk delta), and a **delta** after the second **get** will produce delta 1.1.1.1.

Keyletters That Affect Output

get -p causes the retrieved text to be written to the standard output rather than to a g.file. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. **get -p** is used, for example, to create a g.file with an arbitrary name, as in

get -p s.abc > *arbitrary-file-name*

get -s suppresses output normally directed to the standard output, such as the SID of the retrieved version and the number of lines retrieved, but it does not affect messages normally directed to the diagnostic output. **get -s** is used to prevent nondiagnostic messages from appearing on the user's terminal and is often used with **-p** to pipe the output, as in

get -p -s s.abc | pg

get -g suppresses the retrieval of the text of an SCCS file. This is useful in several ways. For example, to verify a particular SID in an SCCS file

get -g -r4.3 s.abc

outputs the SID 4.3 if it exists in the SCCS file **s.abc** or an error message if it does not. Another use of **get -g** is in regenerating a p.file that may have been accidentally destroyed, as in

get -e -g s.abc

get -l causes SCCS to create an "l.file." It is named by replacing the **s.** of the SCCS filename with **l.**, created in the current directory with mode 444 (read only) and owned by the real user. The l.file contains a table (whose format is described under **get**(1) in the *User's Reference Manual*) showing the deltas used in constructing a particular version of the SCCS file. For example

get -r2.3 -l s.abc

generates an l.file showing the deltas applied to retrieve version 2.3 of file **s.abc**. Specifying **p** with **-l**, as in

get -lp -r2.3 s.abc

causes the output to be written to the standard output rather than to the l.file. **get -g** can be used with **-l** to suppress the retrieval of the text.

get −m identifies the changes applied to an SCCS file. Each line of the g.file is preceded by the SID of the delta that caused the line to be inserted. The SID is separated from the text of the line by a tab character.

get −n causes each line of a g.file to be preceded by the value of the ID keyword and a tab character. This is most often used in a pipeline with **grep**(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

get −p −n −s *directory* | **grep** *pattern*

If both **−m** and **−n** are specified, each line of the generated g.file is preceded by the value of the **chap3.13** ID keyword and a tab (this is the effect of **−n**) and is followed by the line in the format produced by **−m**. Because use of **−m** and/or **−n** causes the contents of the g.file to be modified, such a g.file must not be used for creating a delta. Therefore, neither **−m** nor **−n** may be specified together with **get −e**.

See **get**(1) in the *User's Reference Manual* for a full description of additional keyletters.

The delta Command

The **delta**(1) command is used to incorporate changes made to a g.file into the corresponding SCCS file—that is, to create a delta and, therefore, a new version of the file.

The **delta** command requires the existence of a p.file (created via **get −e**). It examines the p.file to verify the presence of an entry containing the user's login name. If none is found, an error message results.

get −e performs. If all checks are successful, **delta** determines what has been changed in the g.file by comparing it via **diff**(1) with its own temporary copy of the g.file as it was before editing. This temporary copy of the g.file is called the d.file and is obtained by performing an internal **get** on the SID specified in the p.file entry.

The required p.file entry is the one containing the login name of the user executing **delta**, because the user who retrieved the g.file must be the one who creates the delta. However, if the login name of the user appears in more than one entry, the same user has executed **get −e** more than once on the same SCCS file. Then, **delta −r** must be used to specify the SID that uniquely identifies the p.file entry. This entry is then the one used to obtain the SID of the delta to be created.

In practice, the most common use of **delta** is

delta s.abc

which prompts

```
comments?
```

to which the user replies with a description of why the delta is being made, ending the reply with a newline character. The user's response may be up to 512 characters long with newlines (not intended to terminate the response) escaped by backslashes, \.

If the SCCS file has a **v** flag, **delta** first prompts with

```
MRs?
```

(Modification Requests), on the standard output. The standard input is then read for MR numbers, separated by blanks and/or tabs, ended with a newline character. A Modification Request is a formal way of asking for a correction or enhancement to the file. In some controlled environments where changes to source files are tracked, deltas are permitted only when initiated by a trouble report, change request, trouble ticket, etc., collectively called MRs. Recording MR numbers within deltas is a way of enforcing the rules of the change management process.

delta −y and/or **−m** can be used to enter comments and MR numbers on the command line rather than through the standard input, as in

delta −y"*descriptive comment*" **−m**"*mrnum1 mrnum2*" **s.abc**

In this case, the prompts for comments and MRs are not printed, and the standard input is not read. These two keyletters are useful when **delta** is executed from within a shell procedure (see **sh**(1) in the *User's Reference Manual*).

delta −m is allowed only if the SCCS file has a **v** flag.

No matter how comments and MR numbers are entered with **delta**, they are recorded as part of the entry for the delta being created. Also, they apply to all SCCS files specified with the **delta**.

If **delta** is used with more than one file argument and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have the flag, none of the files named may have it.

When **delta** processing is complete, the standard output displays the SID of the new delta (from the p.file) and the number of lines inserted, deleted, and left unchanged. For example:

```
1.4
14 inserted
7 deleted
345 unchanged
```

If line counts do not agree with the user's perception of the changes made to a g.file, it may be because there are various ways to describe a set of changes, especially if lines are moved around in the g.file. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should always agree with the number of lines in the edited g.file.

If you are in the process of making a delta, the **delta** command finds no ID keywords in the edited g.file, the message

```
No id keywords (cm7)
```

is issued after the prompts for commentary but before any other output. This means that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by making a delta from a g.file that was created by a **get** without **−e** (ID keywords are replaced by **get** in such a case). It could also be caused by accidentally deleting or changing ID keywords while editing the g.file. Or, it is possible that the file had no ID keywords. In any case, the delta will be created unless there is an **i** flag in the SCCS file

(meaning the error should be treated as fatal), in which case the delta will not be created.

After the processing of an SCCS file is complete, the corresponding p.file entry is removed from the p.file. All updates to the p.file are made to a temporary copy, the "q.file," whose use is similar to the use of the x.file described earlier under "SCCS Command Conventions." If there is only one entry in the p.file, then the p.file itself is removed.

In addition, **delta** removes the edited g.file unless **−n** is specified. For example

```
delta −n s.abc
```

will keep the g.file after processing.

delta −s suppresses all output normally directed to the standard output, other than `comments?` and `MRs?`. Thus, use of **−s** with **−y** (and/or **−m**) causes **delta** to neither read the standard input nor write the standard output.

The differences between the g.file and the d.file constitute the delta and may be printed on the standard output by using **delta −p**. The format of this output is similar to that produced by **diff**(1).

The admin Command

The **admin**(1) command is used to administer SCCS files—that is, to create new SCCS files and change the parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters with **admin** or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing SCCS files.

Two keyletters are used in detecting and correcting corrupted SCCS files (see "Auditing" under "SCCS Files").

Newly created SCCS files are given access permission mode 444 (read only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command on that file.

Creation of SCCS Files

An SCCS file can be created by executing the command

```
admin −ifirst s.abc
```

in which the value **first** with **−i** is the name of a file from which the text of the initial delta of the SCCS file **s.abc** is to be taken. Omission of a value with **−i** means **admin** is to read the standard input for the text of the initial delta.

The command

```
admin −i s.abc < first
```

is equivalent to the previous example.

If the text of the initial delta does not contain ID keywords, the message

```
No id keywords (cm7)
```

is issued by **admin** as a warning. However, if the command also sets the **i** flag (not to be confused with the **−i** keyletter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using **admin −i**.

admin −r is used to specify a release number for the first delta. Thus:

admin −ifirst −r3 s.abc

means the first delta should be named 3.1 rather than the normal 1.1. Because **−r** has meaning only when creating the first delta, its use is permitted only with **−i**.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may want to record why this was done. Comments (**admin −y**) and/or MR numbers (**−m**) can be entered in exactly the same way as a **delta**.

If **−y** is omitted, a comment line of the form :

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (**admin −m**), the **v** flag must be set via **−f**. The **v** flag simply determines whether MR numbers must be supplied when using any SCCS command that modifies a delta commentary (see **sccsfile**(4) in the *Programmer's Reference Manual*) in the SCCS file. Thus:

admin −ifirst −m*mrnum1* **−fv s.abc**

Note that **−y** and **−m** are effective only if a new SCCS file is being created.

Initialization and Modification of SCCS File Parameters

Part of an SCCS file is reserved for descriptive text, usually a summary of the file's contents and purpose. It can be initialized or changed by using **admin −t**.

When an SCCS file is first being created and **−t** is used, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

admin −ifirst −tdesc s.abc

specifies that the descriptive text is to be taken from file **desc**.

When processing an existing SCCS file, **−t** specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

admin −tdesc s.abc

specifies that the descriptive text of the SCCS file is to be replaced by the contents of **desc**. Omission of the filename after the **−t** keyletter as in

admin −t s.abc

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized or changed by **admin −f**, or deleted via **−d**.

SCCS file flags are used to direct certain actions of the various commands. (See **admin**(1) in the *User's Reference Manual* for a description of all the flags.) For example, the **i** flag specifies that a warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. The **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command.

admin −f is used to set flags and, if desired, their values. For example

admin −ifirst −fi −fm*modname* **s.abc**

sets the **i** and **m** (module name) flags. The value *modname* specified for the **m** flag is the value that the **get** command will use to replace the %M% ID keyword. (In the absence of the **m** flag, the name of the g.file is used as the replacement for the %M% ID keyword.) Several **−f** keyletters may be supplied on a single **admin**, and they may be used whether the command is creating a new SCCS file or processing an existing one.

admin −d is used to delete a flag from an existing SCCS file. As an example, the command

admin −dm s.abc

removes the **m** flag from the SCCS file. Several **−d** keyletters may be used with one **admin** and may be intermixed with **−f**.

SCCS files contain a list of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default, allowing anyone to create deltas. To create a user list (or add to an existing one), **admin −a** is used. For example,

admin −axyz −awql −a1234 s.abc

adds the login names **xyz** and **wql** and the group ID **1234** to the list. **admin −a** may be used whether creating a new SCCS file or processing an existing one.

admin −e (erase) is used to remove login names or group IDs from the list.

The prs Command

The **prs**(1) command is used to print all or part of an SCCS file on the standard output. If **prs −d** is used, the output will be in a format called data specification. Data specification is a string of SCCS file data keywords (not to be confused with **get** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example,

:I:

is defined as the data keyword replaced by the SID of a specified delta. Similarly, **:F:** is the data keyword for the SCCS filename currently being processed, and **:C:** is the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list, see **prs**(1) in the *User's Reference Manual.*

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example,

prs −d":I: this is the top delta for :F: :I:" s.abc

may produce on the standard output

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying its SID using **prs −r**. For example,

```
prs −d":F:: :I: comment line is: :C:" −r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If **−r** is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained with **−l** or **−e**. The use of **prs −e** substitutes data keywords for the SID designated via **−r** and all deltas created earlier, while **prs −l** substitutes data keywords for the SID designated via **−r** and all deltas created later. Thus, the command

```
prs −d:I: −r1.4 −e s.abc
```

may output

```
1.4
1.3
1.2.1.1
1.2
1.1
```

and the command

```
prs −d:I: −r1.4 −l s.abc
```

may produce

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both **−e** and **−l**.

The sact Command

sact(1) is like a special form of the **prs** command that produces a report about files that are out for edit. The command takes only one type of argument: a list of file or directory names. The report shows the SID of any file in the list that is out for edit, the SID of the impending delta, the login of the user who executed the **get −e** command, and the date and time the **get −e** was executed. It is a useful command for an administrator.

The help Command

The **help**(1) command prints the syntax of SCCS commands and of messages that may appear on the user's terminal. Arguments to **help** are simply SCCS commands or the code numbers that appear in parentheses after SCCS messages. (If no argument is given, **help** prompts for one.) Explanatory information is printed on the standard output. If no information is found, an error message is printed. When more than one argument is used, each is processed independently, and an error resulting from one will not stop the processing of the others.

NOTE

There is no conflict between the **help**(1) command of SCCS and the UNIX system **help**(1) utilities. The installation procedure for each package checks for the prior existence of the other.

Explanatory information related to a command is a synopsis of the command. For example,

help ge5 rmdel

produces

```
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
  rmdel  -rSID  name  ...
```

The rmdel Command

The **rmdel**(1) command allows removal of a delta from an SCCS file. Its use should be reserved for deltas in which incorrect global changes were made. The delta to be removed must be a leaf delta. That is, it must be the most recently created delta on its branch or on the trunk of the SCCS file tree. In Figure 7-3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed. Only after they are removed can deltas 1.3.2.1 and 2.1 be removed.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must be either the one who created the delta being removed or the owner of the SCCS file and its directory.

The **–r** keyletter is mandatory with **rmdel**. It is used to specify the complete SID of the delta to be removed. Thus,

rmdel –r2.3 s.abc

specifies the removal of trunk delta 2.3.

Before removing the delta, **rmdel** checks that the release number (R) of the given SID satisfies the relation:

floor less than or equal to R less than or equal to ceiling

The **rmdel** command also checks the SID to make sure it is not for a version on which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's user list (or the user list must be empty). Also, the release specified cannot be locked against editing. That is, if the **l** flag is set (see **admin**(1) in the *User's Reference Manual*), the release must not be contained in the list. If these conditions are not satisfied, processing is terminated, and the delta is not removed.

Once a specified delta has been removed, its type indicator in the delta table of the SCCS file is changed from D (delta) to R (removed).

The cdc Command

The **cdc**(1) command is used to change the commentary made when the delta was created. It is similar to the **rmdel** command (e.g., **−r** and full SID are necessary), although the delta need not be a leaf delta. For example,

cdc −r3.4 s.abc

specifies that the commentary of delta 3.4 is to be changed. New commentary is then prompted for as with **delta**.

The old commentary is kept, but it is preceded by a comment line indicating that it has been superseded, and the new commentary is entered ahead of the comment line. The inserted comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the insertion of new and deletion of old ("!" prefix) MR numbers. Thus,

```
cdc −r1.4 s.abc
MRs? mrnum3 !mrnum1          (The MRs? prompt appears only
                              if the v flag has been set.)
comments? deleted wrong MR number and inserted correct MR number
```

inserts **mrnum3** and deletes **mrnum1** for delta 1.4.

An MR (Modification Request) is described above under the **delta** command.

The what Command

The **what**(1) command is used to find identifying information within any UNIX file whose name is given as an argument. No keyletters are accepted. The **what** command searches the given file(s) for all occurrences of the string **@(#)**, which is the replacement for the **%Z%** ID keyword (see **get**(1)). It prints on the standard output whatever follows the string until the first double quote, ", greater than, **>**, backslash, \, newline, or nonprinting NUL character.

For example, if an SCCS file called **s.prog.c** (a C language program) contains the following line:

```
char  id[]= "%W%";
```

and the command

get –r3.4 s.prog.c

is used, the resulting g.file is compiled to produce **prog.o** and **a.out**. Then, the command

what prog.c prog.o a.out

produces

```
prog.c:
  prog.c:  3.4
prog.o:
  prog.c:  3.4
a.out:
  prog.c:  3.4
```

The string searched for by **what** need not be inserted via an ID keyword of **get**; it may be inserted in any convenient manner.

The sccsdiff Command

The **sccsdiff**(1) command determines (and prints on the standard output) the differences between any two versions of an SCCS file. The versions to be compared are specified with **sccsdiff –r** in the same way as with **get –r**. SID numbers must be specified as the first two arguments. Any following keyletters are interpreted as arguments to the **pr**(1) command (which prints the differences) and must appear before any filenames. The SCCS file(s) to be processed are named last. Directory names and a name of **–** (a lone minus sign) are not acceptable to **sccsdiff**.

The following is an example of the format of **sccsdiff**:

sccsdiff –r3.4 –r5.6 s.abc

The differences are printed the same way as by **diff**(1).

The comb Command

The **comb**(1) command lets the user try to reduce the size of an SCCS file. It generates a shell procedure (see **sh**(1) in the *User's Reference Manual*) on the standard output, which reconstructs the file by discarding unwanted deltas and combining other specified deltas. (It is not recommended that **comb** be used as a matter of routine.)

In the absence of any keyletters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the shape of an SCCS tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 7-3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

Some of the keyletters used with this command are:

comb –s This option generates a shell procedure that produces a report of the percentage space (if any) the user will save. This is often useful as an advance step.

comb −p This option is used to specify the oldest delta the user wants preserved.

comb −c This option is used to specify a list (see **get**(1) in the *User's Reference Manual* for its syntax) of deltas the user wants preserved. All other deltas will be discarded.

The shell procedure generated by **comb** is not guaranteed to save space. A reconstructed file may even be larger than the original. Note, too, that the shape of an SCCS file tree may be altered by the reconstruction process.

The val Command

The **val**(1) command is used to determine whether a file is an SCCS file meeting the characteristics specified by certain keyletters. It checks for the existence of a particular delta when the SID for that delta is specified with **−r**.

The string following **−y** or **−m** is used to check the value set by the **t** or **m** flag, respectively. See **admin**(1) in the *User's Reference Manual* for descriptions of these flags.

The **val** command treats the special argument **−** differently from other SCCS commands. It allows **val** to read the argument list from the standard input instead of from the command line, and the standard input is read until an end-of-file (CTRL-D) is entered. This permits one **val** command with different values for keyletters and file arguments. For example,

```
val − −yc −mabc s.abc −mxyz −ypl1 s.xyz
```

first checks if file **s.abc** has a value **c** for its type flag and value **abc** for the module name flag. Once this is done, **val** processes the remaining file, in this case **s.xyz**.

The **val** command returns an 8-bit code. Each bit set shows a specific error (see **val**(1) for a description of errors and codes). In addition, an appropriate diagnostic is printed unless suppressed by **−s**. A return code of 0 means all files met the characteristics specified.

The vc Command

The **vc**(1) command is an **awk**-like tool used for version control of sets of files. While it is distributed as part of the SCCS package, it does not require the files it operates on to be under SCCS control. A complete description of **vc** may be found in the *User's Reference Manual*.

SCCS Files

This section covers protection mechanisms used by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

Protection

SCCS relies on the capabilities of the UNIX system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files—that is, changes by non-SCCS commands. Protection features provided directly by SCCS are the release lock flag, the release floor and ceiling flags, and the user list.

Files created by the **admin** command are given access permission mode 444 (read only). This mode should remain unchanged because it prevents modification of SCCS files by non-SCCS commands. Directories containing SCCS files should be given mode 755, which allows only the owner of the directory to modify it.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies their protection and auditing. The contents of directories should be logical groupings—subsystems of the same large project, for example.

SCCS files should have only one link (name) because commands that modify them do so by creating a copy of the file (the x.file; see "SCCS Command Conventions"). When processing is done, the old file is automatically removed and the x.file renamed (**s.** prefix). If the old file had additional links, this breaks them. Then, rather than process such files, SCCS commands will produce an error message.

When only one person uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

When several users with unique user IDs are assigned SCCS responsibilities (e.g., on large development projects), one user—that is, one user ID—must be chosen as the owner of the SCCS files. This person will administer the files (e.g. use the **admin** command) and will be SCCS administrator for the project. Because other users do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and, if desired, **rmdel** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the set user ID on execution bit on (see **chmod**(1) in the *User's Reference Manual*). This assures that the effective user ID is the user ID of the SCCS administrator. With the privileges of the interface program during command execution, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the user list for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. Thus, they may modify SCCS only with **delta** and, possibly, **rmdel** and **cdc**.

A project-dependent interface program, as its name implies, can be custom built for each project. Its creation is discussed later under "An SCCS Interface Program."

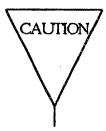
Formatting

SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	a line containing the logical sum of all the characters of the file (not including the checksum itself)
Delta Table	information about each delta, such as type, SID, date and time of creation, and commentary
User Names	list of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas
Flags	indicators that control certain actions of SCCS commands
Descriptive Text	usually a summary of the contents and purpose of the file
Body	the text administered by SCCS, intermixed with internal SCCS control lines

Details on these file sections may be found in **sccsfile**(4). The checksum is discussed below under "Auditing."

Since SCCS files are ASCII files they can be processed by non-SCCS commands like **ed**(1), **grep**(1), and **cat**(1). This is convenient when an SCCS file must be modified manually (e.g., a delta's time and date were recorded incorrectly because the system clock was set incorrectly), or when a user wants simply to look at the file.



Extreme care should be exercised when modifying SCCS files with non-SCCS commands.

Auditing

When a system or hardware malfunction destroys an SCCS file, any command will issue an error message. Commands also use the checksum stored in an SCCS file to determine whether the file has been corrupted since it was last accessed (possibly by having lost one or more blocks or by having been modified with **ed**(1)). No SCCS command will process a corrupted SCCS file except the **admin** command with **−h** or **−z**, as described below.

SCCS files should be audited for possible corruptions on a regular basis. The simplest and fastest way to do an audit is to use **admin −h** and specify all SCCS files:

admin −h s.*file1* s.*file2* ...
or
admin −h *directory1 directory2* ...

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

```
corrupted file (co6)
```

is produced for that file. The process continues until all specified files have been examined. When examining directories (as in the second example above), the checksum process will not detect missing files. A simple way to learn whether files are missing from a directory is to execute the **ls**(1) command periodically, and compare

the outputs. Any file whose name appeared in a previous output but not in the current one no longer exists.

When a file has been corrupted, the way to restore it depends on the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX system operations group and request that the file be restored from a backup copy. If the damage is minor, repair through editing may be possible. After such a repair, the **admin** command must be executed:

admin −z s.*file*

The purpose of this is to recompute the checksum and bring it into agreement with the contents of the file. After this command is executed, any corruption that existed in the file will no longer be detectable.

Chapter 8: An Introduction to RCS

Abstract

This document was preapred by:

Walter F. Tichy
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

The Revision Control System (RCS) manages software libraries. It greatly increases programmer productivity by centralizing and cataloging changes to a software project. This document describes the benefits of using a source code control system. It then gives a tutorial introduction to the use of RCS.

Functions of RCS

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc. It greatly increases programmer productivity by providing the following functions.

- RCS stores and retrieves multiple revisions of program and other text. Thus, one can maintain one or more releases while developing the next release, with a minimum of space overhead. Changes no longer destroy the original -- previous revisions remain accessible.
  - Maintains each module as a tree of revisions.
  - Project libraries can be organized centrally, decentralized, or any way you like.
  - RCS works for any type of text: programs, documentation, memos, papers, graphics, VLSI layouts, form letters, etc.
- RCS maintains a complete history of changes. Thus, one can find out what happened to a module easily and quickly, without having to compare source listings or having to track down colleagues.
  - RCS performs automatic record keeping.
  - RCS logs all changes automatically.
  - RCS guarantees project continuity.
- RCS manages multiple lines of development.
- RCS can merge multiple lines of development. Thus, when several parallel lines of development must be consolidated into one line, the merging of changes is automatic.
- RCS flags coding conflicts. If two or more lines of development modify the same section of code, RCS can alert programmers about overlapping changes.
- RCS resolves access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and makes sure that one change will not wipe out the other one.
- RCS provides high-level retrieval functions. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
- RCS provides release and configuration control. Revisions can be marked as released, stable, experimental, etc. Configurations of modules can be described simply and directly.
- RCS performs automatic identification of modules with name, revision number, creation time, author, etc. Thus, it is always possible to determine which revisions of which modules make up a given configuration.
- Provides high-level management visibility. Thus, it is easy to track the status of a software project.
  - RCS provides a complete change history.

- RCS records who did what when to which revision of which module.

- RCS is fully compatible with existing software development tools. RCS is unobtrusive -- its interface to the file system is such that all your existing software tools can be used as before.
- RCS' basic user interface is extremely simple. The novice only needs to learn two commands. Its more sophisticated features have been tuned towards advanced software development environments and the experienced software professional.
- RCS simplifies software distribution if customers also maintain sources with RCS. This technique assures proper identification of versions and configurations, and tracking of customer changes. Customer changes can be merged into distributed versions locally or by the development group.
- RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding differences are compressed into the shortest possible form.

Getting Started with RCS

Suppose you have a file f.c that you wish to put under control of RCS. Invoke the checkin command:

```
ci  f.c
```

This command creates f.c,v, stores f.c into it as revision 1.1, and deletes f.c. It also asks you for a description. The description should be a synopsis of the contents of the file. All later checkin commands will ask you for a log entry, which should summarize the changes that you made.

Files ending in ,v are called RCS files ("v" stands for "versions"), the others are called working files. To get back the working file f.c in the previous example, use the checkout command:

```
co  f.c
```

This command extracts the latest revision from f.c,v and writes it into f.c. You can now edit f.c and check it in back in by invoking:

```
ci  f.c
```

Ci increments the revision number properly. If *ci* complains with the message

```
ci error: no lock set by <your login>
```

then your system administrator has decided to create all RCS files with the locking attribute set to "strict". With strict locking, you you must lock the revision during the previous checkout. Thus, your last checkout should have been

```
co  -l  f.c
```

Locking assures that you, and only you, can check in the next update, and avoids nasty problems if several people work on the same file. Of course, it is too late now to do the checkout with locking, because you probably modified f.c already, and a second checkout would overwrite your changes. Instead, invoke

```
rcs  -l  f.c
```

This command will lock the latest revision for you, unless somebody else got ahead of you already. If someone else has the lock you will have to negotiate your changes with them.

If your RCS file is private, i.e., if you are the only person who is going to deposit revisions into it, strict locking is not needed and you can turn it off. If strict locking is turned off, the owner off the RCS file need not have a lock for checkin; all others still do. Turning strict locking off and on is done with the commands:

```
rcs  -U  f.c        and        rcs  -L  f.c
```

You can set the locking to strict or non-strict on every RCS file.

If you do not want to clutter your working directory with RCS files, create a subdirectory called RCS in your working directory, and move all your RCS files there. RCS commands will look first into that directory to find needed files. All the commands discussed above will still work, without any change.

NOTE

Pairs of RCS and working files can really be specified in 3 ways: a) both are given, b) only the working file is given, c) only the RCS file is given. Both files may have arbitrary path prefixes; RCS commands pair them up intelligently.

To avoid the deletion of the working file during checkin (should you want to continue editing), invoke

```
ci -l f.c
```

This command checks in f.c as usual, but performs an additional checkout with locking. Thus, it saves you one checkout operation. There is also an option **-u** for *ci* that does a checkin followed by a checkout without locking. This is useful if you want to compile the file after the checkin. Both options also update the identification markers in your file (see below).

You can give *ci* the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2. The command

```
ci -r2 f.c      or      ci -r2.1 f.c
```

assigns the number 2.1 to the new revision. From then on, *ci* will number the subsequent revisions with 2.2, 2.3, etc. The corresponding *co* commands

```
co -r2 f.c      and     co -r2.1 f.c
```

retrieve the latest revision numbered 2.x and the revision 2.1, respectively. *Co* without a revision number selects the latest revision on the "trunk", i.e., the highest revision with a number consisting of 2 fields. Numbers with more than 2 fields are needed for branches. For example, to start a branch at revision 1.3, invoke

```
ci -r1.3.1 f.c
```

This command starts a branch numbered 1 at revision 1.3, and assigns the number 1.3.1.1 to the new revision. For more information about branches, see **rcsfile**(4).

Automatic Identification

RCS can put special strings for identification into your source and object code. To obtain such identification, place the marker

```
$Header:$
```

into your text, for instance inside a comment. RCS will replace this marker with a string of the form

```
$Header: filename 1.2 88/10/06 16:20:05 owner Locked $
```

You never need to touch this string, because RCS keeps it up to date automatically. To propagate the marker into your object code, simply put it into a literal character string. In C, this is done as follows:

```
static char rcsid[] = "$Header:$"
```

The command *ident* extracts such markers from any file, even object code. Thus, *ident* helps you to find out which revisions of which modules were used in a given program.

You may also find it useful to put the marker

```
$Log:$
```

into your text, inside a comment. This marker accumulates the log messages that are requested during checkin. Thus, you can maintain the complete history of your file directly inside it. There are several additional identification markers; see **co**(1) for details.

How to combine MAKE and RCS

If your RCS files are in the same directory as your working files, you can put a default rule into your makefile. Do not use a rule of the form .c,v.c, because such a rule keeps a copy of every working file checked out, even those you are not working on. Instead, use this:

```
.SUFFIXES: .c,v

.c,v.o:
          co -q $*.c
          cc $(CFLAGS) -c $*.c
          rm -f $*.c

prog:   f1.o f2.o .....
          cc f1.o f2.o ..... -o prog
```

This rule has the following effect. If a file f.c does not exist, and f.o is older than f.c,v, MAKE checks out f.c, compiles f.c into f.o, and then deletes f.c. From then on, MAKE will use f.o until you change f.c,v.

If f.c exists (presumably because you are working on it), the default rule .c.o takes precedence, and f.c is compiled into f.o, but not deleted.

If you keep your RCS file in the directory ./RCS, all this will not work and you have to write explicit checkout rules for every file, like

```
f1.c:   RCS/f1.c,v; co -q f1.c
```

Unfortunately, these rules do not have the property of removing unneeded .c-files.

Additional Information on RCS

If you want to know more about RCS (for example how to work with a tree of revisions and how to use symbolic revision numbers) read the following paper:

> Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

Taking a look at the manual page **rcsfile**(4) should also help to understand the revision tree permitted by RCS.

Chapter 9: 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**. Multiple **BEGIN** and **END** sections are permitted, but are executed in the order found in the program. It is customary to place **BEGIN** at the beginning of the program, and **END** at the end, but it is not required.

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 9-1.

Keywords

ARGC	**NR**	**atan2**	**exit**	**index**	**printf**	**string**
ARGV	**OFMT**	**break**	**exp**	**int**	**rand**	**sub**
BEGIN	**OFS**	**close**	**for**	**length**	**return**	**substr**
END	**ORS**	**continue**	**function**	**log**	**sin**	**system**
FILENAME	**RLENGTH cos**	**getline**	**match**	**split**	**while**	
FNR	**RS**	**delete**	**gsub**	**next**	**sprintf**	
FS	**RSTART**	**do**	**if**	**number**	**sqrt**	
NF	**SUBSEP**	**else**	**in**	**print**	**srand**	

Figure 9-1: **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 9-2.

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
=	exponentiate-equals	X ^= Y is 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 9-2: **awk** Assignment Operators

Arithmetic operators are shown in Figure 9-3. All arithmetic is done in floating-point.

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

Figure 9-3: **awk** Arithmetic Operators

Relational operators are shown in Figure 9-4.

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

Figure 9-4: **awk** Relational Operators

Logical operators are shown in Figure 9-5.

Symbol	Description
&&	and
||	or
!	not

Figure 9-5: **awk** Logical Operators

Regular expression matching operators are shown in the Figure 9-6.

Symbol	Description
~	matches
!~	does not match

Figure 9-6: Operators for Matching Regular Expressions in **awk**

Conditional operators are shown in Figure 9-7.

Symbol	Description
?:	expr1 ? expr2 : expr3
is same as	if (expr1) expr2 else expr3

Figure 9-7: **awk** Conditional Operators

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 9-7.

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

Figure 9-8: 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 9-8.

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 9-9: 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 9-9.

Functions

atan2 (*expr1, expr2*)
cos (*expression*)
exp (*expression*)
int (*expression*)
log (*expression*)
rand ()
sqrt (*expression*)
sin (*expr*)
srand (*expr*)

Figure 9-10: Built-in Functions for Arithmetic 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*. **atan2** computes the acrtangent of *expr1/expr2* in the range of $-\pi$ to $+\pi$. **cos** computes the cosine of *expression*, **rand** gives a random number, **sin** gives the sin of *expression* in radians, and **srand** is supplied the new seed for **rand**. 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 9-10.

String Functions

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

Figure 9-11: **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 **gsub**(*e1,e2*) substitutes *e1* for *e2* in $0, and **gsub**(*e1,e2,e3*) substitutes *e1* for *e2* in the string *e3*. The substititutions performed by **gsub** are global, that is, the substitution is performed for each occurrence in the record.

The functions **sub**(*e1,e2*) and **sub**(*e1,e2,e3*) are similar to their **gsub** counterparts with the exception that the substitution is only made for the lefmost occurrence of the string.

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 **match**(*e1,e2*) tests for the substring *e2* in the expression *e1* and returns the leftmost index character if it is found, or else returns a 0.

The function **split**(*e, array, sep*) 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 *e*. The function **split**(*e, array*) 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**(*string, pos*) returns the suffix of *string* starting at position *pos*. The function **substr**(*string, pos, length*) 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**(*string, pos, length*) is equivalent to **substr**(*string, pos*). 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 9-12: 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 **$2** by the **awk** program. The ith field of the present record is referred to as **$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
Australia   2968  14          Australia
India       1269  637         Asia
Argentina   1072  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
```

Conditional Expressions

Conditional expressions perform an "if-else" conditional statement. The syntax is:

expr1 ? expr2 : expr3

where *expr1* is evaluated and, if true, the value of the conditional expression is *expr2*; if false, the value is *expr3*.

For example:

```
{ print ($5 == "America" ? $0 : "not applicable:" $1) }
```

prints the records of countries in North and South America from our sample file, and prints a line stating that the record for the country is not applicable for any other countries.

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
pqraxxxxxxxxxxc9Q1
```

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 **4**th 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**.

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 math functions:

Math Function	**Description**
atan2	inverse trigonometric function
cos	trigonometric function
rand	random number generator
sin	trigonometric function
sqrt	square root
srand	seed for random number
log	natural logarithm
exp	exponential
int	integer no greater than

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.

Substitutions and Matches

Substitutions can be performed with the gsub and sub functions, and match is used to return the number of matches to a particular string.

Substitutions

The **gsub** function performs substitution globally for all matches in the record, while **sub** substitutes only the leftmost match. For example, take a record:

```
bananas in Havana
```

performing a **sub** operation such as:

```
{ sub ("ana", "anda") ; print }
```

produces this output:

```
bandanas in Havana
```

While a **gsub** operation:

{ gsub ("ana", "anda") ; print }

on the same record produces:

```
bandanas in Havanda
```

Matches

With match, the built-in variables RSTART and RLENGTH are automatically set to the index of the matched string and its length, respectively. For example, the **match** operation:

{ match ("Havana", "an") ; print }

peformed on the record:

```
bananas in Havana
```

simply outputs the record, while this **match** operation:

{ match ("Havana", "an") ; print RSTART, RLENGTH }

outputs:

```
4 2
```

where "4" is the starting, or index, character of the matching string and "2" is its length.

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

Array subscripts are strings, thus 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 10: 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 10-1. 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.

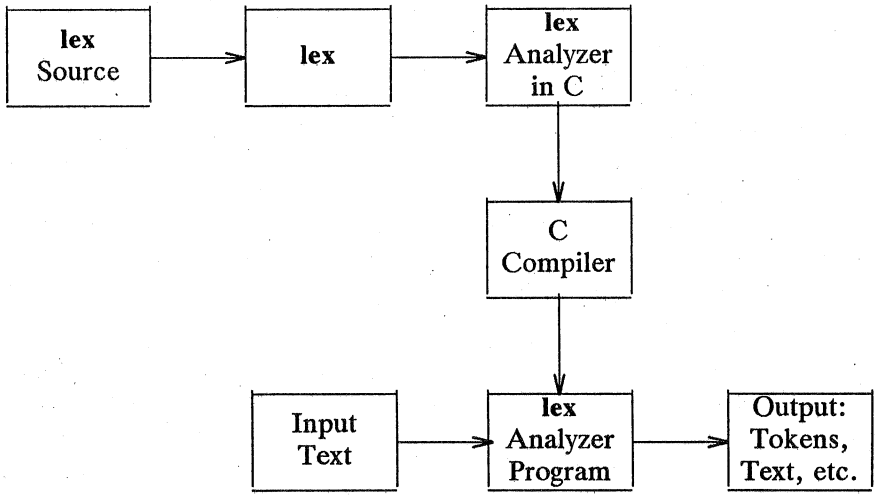


Figure 10-1: Creation and Use of a Lexical Analyzer with **lex**

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 2
                       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.

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 >=.

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.

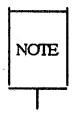
```
extern int tokval;
```

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 **#define**s 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.



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, 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 later in 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:

```
begin                   return(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 10-1 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
```

This 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 11: 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
%%
2rules
%%
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

        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 **dbx**.

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    /* includes identifiers and literals */
%token      C_IDENTIFIER  /* identifier (but not literal)
                          /*    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 */

%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 '>'
        ;

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;

%}

%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;
            }
            |  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
            ;

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 through 9 */
                /* all other characters 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 */

  /* 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
        {
                $$ = dreg[$1];
        }
        |  dexp '+' dexp
        {
                $$ = $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 )
        {
            (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
{
        $$ = vmul( $1, $1, $3 )
}
|   vexp '/' vexp
{
        if( dcheck( $3 ) ) YYERROR;
        $$ = 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))
   {
      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;
        }

        *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;
    }
    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));
}
```