
SCO
THE SANTA CRUZ OPERATION

SCO
THE SANTA CRUZ OPERATION

For the Apple® Lisa® 2

Product
XENIX™ Development System
Introductory Softcare™ Support

XENIX is a fully licensed version of the UNIX™ Operating System as authorized by AT&T for single and multiuser operation.

WARNING: the XENIX Development System is licensed for use only with the XENIX Operating System. Any other use of this product is in violation of applicable license agreements.

Version
XENIX 3.0, Release 1.0

Requirements

System:	XENIX Operating System **Apple Lisa 2**
Memory:	512 KBytes or greater
Hard Disk:	Apple hard disk(s) totalling 10Mb or greater

Please Note
Additional requirements information may be obtained by calling SCO Softcare™ Center 9:00 a.m. to 5:00 p.m. Pacific Time, Monday through Friday.

Read carefully the enclosed License Agreement before opening the diskette package. Opening the diskette package indicates your acceptance of this License Agreement. If you do not agree with it, you have ten (10) days from date of purchase to return this package, (with the diskette package UNOPENED) and a copy of your purchase receipt, and your money will be refunded.

The XENIX™ Development System

Programmer's Guide for the Apple Lisa 2™

The Santa Cruz Operation, Inc.

The Santa Cruz Operation, Inc.
500 Chestnut Street
P.O. Box 1900
Santa Cruz, California 95061
(408) 425-7222 · TWX: 910-598-4510 SCO SACZ

Release: 68-5-24-84-1.0/1.0

Contents

Chapter 1
Introduction

1.1 Overview

This guide explains how to use the XENIX Software Development system to create and maintain C and assembly language programs. The system provides a broad spectrum of programs and commands to help you design and develop applications and system software. These programs and commands let you create C and assembly language programs for execution on the XENIX system. They also let you debug these programs, automate their creation, and maintain versions of the programs you develop.

The following sections introduce the programs and commands of the XENIX Software Development System and explain the steps you can take to develop programs for the XENIX system. Most of the programs and commands in these introductory sections are fully explained later in this guide. Some commands mentioned here are part of the XENIX Timesharing System and are explained in the XENIX *User's Guide* and XENIX *Operations Guide*.

1.2 Creating C Language Programs

All C language programs start as a collection of C program statements on files. The XENIX system provides a number of text editors that let you create source files easily and efficiently. The most convenient editor is the screen-oriented editor *vi*. *Vi* provides many editing commands that let you easily insert, replace, move, and search for text. All commands can be invoked from command keys or from a command line. The program has also has a variety of options that let you modify its operation.

Once a C language program has been written to a source file, you can create an executable program using the cc command. The cc command invokes the XENIX C compiler which compiles the source file. This command also invokes other XENIX programs to prepare the compiled program for execution.

You can debug an executable C program with the XENIX debugger *adb*. *Adb* provides a direct interface to the machine instructions that make up an executable program.

If you wish to check a program before compilation, you can use *lint*, the XENIX C program checker. *Lint* checks the content and construction of C language programs for syntactical and logical errors. It also enforces a strict set of guidelines for proper C programming style. *Lint* is normally used in the early stages of program development to check for illegal and improper usage of the C language.

1.3 Creating Other Programs

The C programming language can meet the needs of most programming projects. In cases where finer control of execution is required, you may create

assembly language programs using the XENIX assembler *as*. *As* assembles source files and produces relocatable object files that can be linked to your C language programs with *ld*. The *ld* program is the XENIX linker. It links relocatable object files created by the C compiler or assembler and produces executable programs. Note that the cc command automatically invokes the linker and the assembler so use of either is optional.

You can create source files for lexical analyzers and parsers using the program generators *lex* and *yacc*. The *lex* program is the XENIX lexical analyzer generator. It generates lexical analyzers, written in C program statements, from given specification files. Lexical analyzers are used in programs to pick patterns out of complex input and convert these patterns into meaningful values or tokens. The *yacc* program is the XENIX parser generator. It generates parsers, written in C program statements, from given specification files. Parsers are used in programs to convert meaningful sequences of tokens and values into actions. *Lex* and *yacc* are often used together to make complete programs.

You can preprocess C and assembly language source files, or even *lex* and *yacc* source files using the *m4* macro processor. The *m4* program performs several preprocessing functions, such as converting macros to their defined values and including the contents of files into a source file.

1.4 Creating and Maintaining Libraries

You can create libraries of useful C and assembly language functions and programs using the *ar* and *ranlib* programs. *Ar*, the XENIX archiver, can be used to create libraries of relocatable object files. *Ranlib*, the XENIX random library generator, converts archive libraries to random libraries and places a table of contents at the front of each library.

The **lorder** command finds the ordering relation in an object library. The **tsort** command topologically sorts name lists so that forward dependencies are apparent.

1.5 Maintaining Program Source Files

You can automate the creation of executable programs from C and assembly language source files and maintain your source files using the *make* program and the SCCS commands.

The *make* program is the XENIX program maintainer. It automates the steps required to create executable programs and provides a mechanism for ensuring up to date programs. It is used with small, large, and medium-scale programming projects.

The Source Code Control (SCCS) commands let you maintain different versions of a single program. The commands compress all versions of a source file into a

single file containing a list of differences. These commands also restore compressed files to their original size and content.

Many XENIX commands let you carefully examine a program's source files. The **ctags** command creates a tags file so that C functions can be quickly found in a set of related C source files. The **mkstr** command creates an error message file by examining a C source file.

Other commands let you examine object and executable binary files. The **nm** command prints the list of symbol names in a program. The **hd** command performs a hexadecimal dump of given files, printing files in a variety of formats, one of which is hexadecimal. The **od** command performs an octal dump of given files. **adb** (see chapter 6), allows disassembly of your program. The **size** command reports the size of an object file. The **strings** command finds and prints readable text (strings) in an object or other binary file. The **strip** command removes symbols and relocation bits from executable files. The **sum** command computes check sum for a file and counts blocks. It is used in looking for bad spots in a file and for verifying transmission of data between systems. The **xstr** command extracts strings from C programs to implement shared strings.

1.6 Creating Programs With Shell Commands

In some cases, it is easier to write a program as a series of XENIX shell commands than it is to create a C language program. Shell commands provide much of the same control capability as the C language and give direct access to all the commands and programs normally available to the XENIX user.

The **csh** command invokes the C-shell, a XENIX command interpreter. The C-shell interprets and executes commands taken from the keyboard or from a command file. It has a C-like syntax which makes programming in this command language easy. It also has an aliasing facility, and a command history mechanism.

1.7 Using This Guide

This guide is intended for programmers who are familiar with the C programming language and with the XENIX system.

C language programmers should read Chapters 2, 3, and 6 for an explanation of how to compile and debug C language programs.

Assembly language programmers should read Chapter 7 for an explanation of the XENIX assembler and Chapter 6 for an explanation of how to debug programs.

Programmers who wish to automate the compilation process of their programs should read Chapter 4 for an explanation of the *make* program. Programmers

who wish to organize and maintain multiple versions of their programs should read Chapter 5 for an explanation of the Source Code Control System (SCCS) commands.

Special project programmers who need a convenient way to produce lexical analyzers and parsers should read Chapters 8 and 9 for explanations of the *lex* and *yacc* program generators.

Chapter 1 introduces the XENIX software development programs provided with this package.

Chapter 2 explains how to compile C language programs using the cc command.

Chapter 3 explains how to check C language programs for syntactic and semantic correctness using the C program checker *lint*.

Chapter 4 explains how to automate the development of a program or other project using the *make* program.

Chapter 5 explains how to control and maintain all versions of a project's source files using the SCCS commands.

Chapter 6 explains how to debug C and assembly language programs using the XENIX debugger *adb*.

Chapter 7 explains how to assemble assembly language programs using the XENIX assembler *as*.

Chapter 8 explains how to create lexical analyzers using the program generator *lex*.

Chapter 9 explains how to create parsers using the program generator *yacc*.

Appendix A explains how to write C langugae programs that can be compiled on other XENIX systems.

Appendix B explains how to use to create and process macros using the *m4* macro processor.

1.8 Notational Conventions

This guide uses a number of special symbols to describe the syntax of XENIX commands. The following is a list of these symbols and their meaning.

[ ]	Brackets indicate an optional command argument.
...	Ellipses (three dots) indicate that the preceding argument may be repeated one or more times.

SMALL	Small capitals indicate a key to be pressed.
bold	Boldface characters indicate a command name.
italics	Italic characters indicate a placeholder for a command argument. When typing a command, a placeholder must be replaced with an appropriate filename, number, or option.

Chapter 2
Cc: A C Compiler

2.1 Introduction

This chapter explains how to use the cc command to create executable programs from C language source files. The command compiles C source files by invoking the XENIX C compiler, the C preprocessor, and in some cases the C optimizer. It then invokes other programs, such as the XENIX assembler *as* and linker *ld*, to complete the creation of the executable program.

The cc command accepts as C source files any file containing a complete C program or one or more complete C functions. The command processes the source files in five phases: preprocessing, assembly source generation, optimization (if necessary), machine code generation, and linking.

In the preprocessing phase, the cc command invokes the C preprocessor, which searches the source file for C directives. The preprocessor replaces each directive with a corresponding value or meaning. For example, it replaces each occurrence of a macro name with its defined value and each include directive with the contents of its corresponding include file. It then copies the expanded version of the source file to a temporary file. The preprocessor also allows conditional compilation.

In the assembly source generation phase, the cc command invokes the C compiler which translates the C program statements in the temporary file into equivalent assembly language instructions. These instructions form a complete assembly language source file that performs the same tasks as the statements in the C source file. The compiler copies the assembly instructions to a temporary file.

In the optional optimization phase, the cc -O command invokes the C optimizer which modifies the temporary assembly language file, making it smaller and faster without altering the tasks its performs. Programs of all sizes benefit from optimization.

In the machine code generation phase, the command invokes the XENIX assembler *as* which assembles the temporary assembly language file. The assembler creates an "object file" containing relocatable machine instructions that can be prepared for execution. If more than one source file is processed, a permanent object file is created for each source file.

In the linking phase, the command invokes the XENIX linker *ld*, which resolves all unresolved references to variables and functions in the object file. If necessary, *ld* searches the appropriate program libraries to link the contents of other object files to the given file. The linker then writes the linked instructions to a file. This file, called an "executable binary" file, contains the program's machine instructions in executable binary form. The file *x.out* is used by default.

This chapter assumes that you are familiar with the C programming language and that you can create C program source files using a XENIX text editor.

2.2 Invoking the C Compiler

You can invoke the C compiler with the cc command. The command has the form

cc [*option*] ... *filename* ...

where *option* is a command option, and *filename* is the name of a C language source file, an assembly language source file, or an object file. You may give more than one option or filename, if desired, but you must separate each item with one or more whitespace characters.

The cc command options let you control and modify command operation. For example, you can direct the command to skip the optimization phase or create a permanent copy of the file created during the assembly source generation phase. The options also let you specify additional information about the compilation, such as which program libraries to examine and what the name of the executable file should be. The options are described in detail in the following sections.

The cc command lets you name three different kinds of files: C source, assembly language source, and object files. A file's contents are identified by the filename extension. C source files have the extension *.c*. Assembly language source files have the extension *.s*. Object files have the extension *.o*. The command delays processing of each type of file until the appropriate phase. Thus C source files are processed immediately, assembly language files are processed in the machine code generation phase, and object files are processed in the linking phase. An assembly language source file may be created by hand using a XENIX text editor, or created using the cc command's assembly source generation phase (see the –S option later in this chapter). An object file must be the output of the XENIX assembler or the cc command's machine code generation phase (see the –c option).

2.3 Compiling a Source File

You can compile a source file containing a complete C program by giving the name of the file when you invoke the cc command. The command reads and compiles the statements in the file, links the compiled program with the standard C library, then copies the program to the default output file *a.out*

To compile a source program, type:

cc filename

where *filename* is the name of the file containing the program. The program must be complete, that is, it must contain a main program function. It may contain calls to functions explicitly defined by the program or by the standard C library. For example, assume the the following program is stored in the file named *main.c*.

```
#include <stdio.h>

main ()
{
        int x,y;

        scanf("%d + %d", &x, &y);
        printf("%d\n", x+y);
}
```

To compile this program, type

```
cc main.c
```

The command first invokes the C preprocessor which adds the statements in the file */usr/include/stdio.h* to the beginning of the program. It then compiles these statements and the rest of the program statements. Next, the command links the program with the standard C library which contains the binary code for the *ecanf* and *printf* functions. Finally, it copies the program to the file *x.out*.

You can execute the new program by typing the command

```
x.out
```

The program waits until you enter a sum, then prints the value of that sum. For example, if you type "3 + 5" the program displays "8".

Note that when the command creates the *x.out* file, it gives the file the permissions defined by your current file creation mask.

2.4 Compiling Several Source Files

Large source programs are often split into several files to make it easier to update and edit. You can compile such a program by giving the names of all the files belonging to the program when you invoke the cc command. The command reads and compiles each file in turn, then links all object files together and copies the new program to the file *x.out*.

To compile several source files, type

```
cc filename ...
```

where each *filename* is separated from the next by whitespace. One of these files (and no more than one) must contain a program function named "main". The others may contain functions that are called by this main function or by other functions in the program.

For example, suppose the following main program function is stored in the file *main*.

```
#include <stdio.h>
extern int add();

main ()
{
        int x,y,z;

        scanf ("%d + %d", &x, &y);
        z = add (x, y);
        printf ("%d \n", z);
}
```

Assume that the following function is stored in the file *add.c*:

```
add (a, b)
int a, b;
{
        return (a + b);
}
```

You can compile these files and create an executable program by typing

```
cc main.c add.c
```

The command compiles the statements in *main.c*, then compiles the statements in *add.c*. Finally, it links the two together (along with the standard C library) and copies the program to *a.out*. This program, like the program in the previous section, waits for a sum, then prints the value of the sum.

Compiling several source files at a time causes the command to create object files to hold the binary code generated for each source file. These object files are then used in the linking phase to create an executable program. The object files have the same basename as the source file, but are given the *.o* file extension. For example, when you compile the two source files above, the compiler produces the object files *main.o* and *add.o*. These files are permanent files, i.e., the command does not delete them after completing its operation. The command deletes the object file only if you compile a single source file.

2.5 Using Object Files

You can use an object file created by the cc command in any later invocation of the command. When you specify an object file, the command does nothing with it until the linking phase, that is, the command does not compile or assemble the file.

Source files containing functions do not need to be recompiled each time they are linked to a new program. The generated object files can be used instead, saving the programmer the time it takes to compile each source file. This is another reason large programs are often split into several modules.

To create a program from both source files and object files, give the object filenames along with the source filenames in the command invocation. Make sure the filenames are separated by whitespace characters. For example, assume that the following main program function is stored in the file *mult.c*:

```
#include <stdio.h>

main ()
{
        int x,y,z,i;

        scanf("%d * %d", &x, &y);
        for (i=0; i<y; i++)
                z = add (z,x) ;
        printf("%d \n", z);
}
```

This program uses the *add* function compiled in the previous section. Since the object file containing this function is named *add.o*, you can compile this program and link the object file to it by typing

```
cc mult.c add.o
```

The compiler compiles the statements in *mult.c* and produces an object file for it, then the compiler links the *add.o* file to the new file and stores the executable program in *x.out*. This program waits for you to enter the values to be multiplied, multiplies the values, then displays the result.

2.6 Naming the Output File

You can change the name of the executable program file from *x.out* to any valid filename by using the –o (for "output") option. The option has the form:

```
–o filename
```

where *filename* is a valid filename or a full pathname. If a filename is given, the program file is stored in the current directory. If a full pathname is given, the file is stored in the given directory. If a file with that name already exists, the compiler removes the old file before creating the new one.

For example, the command

```
cc main.c add.o –o addem
```

causes the compiler to create an executable program file *addem* from the source file *main.c* and object file *add.o*. You can execute this program by typing

```
addem
```

The permissions defined by the file creation mask apply to this file just as they do to *a.out*.

Note that the **-o** option does not affect the *a.out* file. This means that the **cc** command does not change the current contents of this file if the **-o** option has been given.

2.7 Compiling Without Linking

You can compile a source file without linking it by using the **-c** (for "compile") option. This option is useful if you wish to have an object file available for later programs but have no current program that uses it. The option has the form:

```
-c filename
```

where *filename* is the name of the source file. You may give more than one filename if you wish. Make sure each name is separated from the next by a space.

For example, to make object files for the source files *main.c*, *add.c*, and *mult.c*, type

```
cc -c main.c add.c mult.c
```

The command compiles each file in turn and copies the compiled source to the files *main.o*, *add.o*, and *mult.o*.

2.8 Linking to Library Functions

A library is a file that contains useful functions in object file format. You can link a source file to these functions by linking it to the library with the **-l** (for "library") option. The option, used by the linker during the linking phase, causes the linker to search the given library for the functions called in the source file. If the functions are found, the linker links them to the source file.

The option has the form

```
cc -lname
```

where *name* is a shortened version of the library's actual filename. The actual filename has the form

lib*name*.a

Spaces between the name and option are not permitted. The linker builds the library's filename from the given name, then searches the */lib* directory for the library. If not found, it searches the */usr/lib* directory.

For example, the command

cc main.c -lcurses

links the library *libcurses.a* to the source file *main.c*.

A library is a convenient way to store a large collection of object files. The XENIX system provides several libraries. The most common is the standard C library. This library is automatically linked to your program whenever you invoke the compiler. Other libraries, such as *libcurses.a*, must be explicitly linked using the -l<libname> option. Without the -l flag, cc and ld would identify a library by inspecting its first byte. The XENIX libraries and their functions are described in detail in the XENIX *Programmer's Reference Guide*.

Note that you can create your own libraries with the XENIX *ar* and *ranlib* programs. These commands let you copy object files to a library file and then prepare the library for searching by the linker. These commands are described in the XENIX *Reference Manual*.

In general, the linker does not search a library until the -l option is encountered, so the placement of the option is important. The option must follow the names of source files containing calls to functions in the given library.

2.9 Optimizing a Source File

You can optimize a source file, that is, make its corresponding assembly source file more efficient, by using the -O (for "optimize") option. For example, the command

cc -O main.c

optimizes the source file *main.c*.

Optimization only applies to compiled files; the compiler cannot optimize assembly source or object files. Furthermore, the -O option must appear before the names of the files you wish to optimize. Files preceding the option are not optimized. For example, the command

cc add.c -O main.c

optimizes *main.c* but not *add.c*.

You may combine the –O and –c options to compile and optimize source files without linking the resulting object files. For example, the command

```
cc –O –c main.c add.c
```

creates optimized object files from the source files *main.c* and *add.c*.

Although optimization is very useful for large programs, it takes more time than regular compilation. In general, it should be used in the last stage of program development, after the program has been debugged.

2.10 Producing an Assembly Source File

You can direct the compiler to save a copy of the temporary assembly source file by using the –S (for "source") option. The option causes the command to copy the temporary assembly source file to a permanent file. This permanent file has the same basename as the source file, but is given the file extension *.s*.

For example, the command

```
cc –S add.c
```

compiles the source file *add.c* and creates an assembly language instruction file *add.s*.

The –S option applies to source files only; the compiler cannot create a source file from an existing object file. Furthermore, the option must appear before the names of the files for which the assembly source is to be saved.

2.11 Stripping the Symbol Table

You can reduce the size of a program by using the –s, option. This option causes the cc command to strip the symbol table. The symbol table contains information about code relocation and program symbols and is used by the XENIX debugger *adb* to allow symbolic references to variables and functions when debugging. The information in this table is not required for normal execution and can be stripped when the program has been completely debugged.

The –s option strips the entire table, leaving machine instructions only.

For example, the command

```
cc –s main.c add.c
```

creates a executable program that contains no symbol table. It also creates the object files *main.o* and *add.o* which contain no symbol tables.

The –s option may be combined with the –O option to create an optimized and stripped program. An optimized and stripped program has the smallest size possible.

Note that you can also strip a program with the XENIX command **strip**. See the XENIX *Reference Manual* for details.

2.12 Profiling a Program

You can examine the flow of execution of a program by adding "profiling" code to the program with the –p option. The profiling code automatically keeps a record of the number of times program functions are called during execution of the program. This record is written to the *mon.out* file and can be examined with the **prof** command.

For example, the command

```
cc –p main.c
```

adds profiling code to the program created from the source file *main.c*. The profiling code automatically calls the *monitor* function which creates the *mon.out* file at normal termination of the program. The **prof** command and *monitor* function are described in detail in *prof*(CP) and *monitor*(S) in the XENIX *Reference Manual*.

2.13 Saving a Preprocessed Source File

You can save a copy of the temporary file created by the C preprocessor by using the –P (for "preprocessing") option. The temporary file is identical to the source file except that all macro names have been expanded and all include directives have been replaced by the specified files. The command copies this temporary file to a permanent file which has the same basename as the source file and the filename extension *.i*.

For example, the command

```
cc –P main.c
```

creates a preprocessed file for the source file *main.c*.

You may also display a copy of the preprocessed source file by using the –E option. This option invokes the C preprocessor only and directs the preprocessor to send the preprocessed file to the standard output.

2.14 Defining a Macro

You can define the value or meaning of a macro used in a source file by using the –D (for "define") option. The option lets you assign a value to a macro when you invoke the compiler and is useful if you have used if directives in your source files.

The option has the form

–Dname=def

where *name* is the name of the macro and *def* is its value or meaning. For example, the command

cc –DNEED=2 main.c

sets the macro "NEED" to the value "2". The command compiles the source file *main.c*, replacing every occurrence of "NEED" with "2". If a name is given but no definition, the compiler assigns the value 1 by default.

You can also remove the initial definition of a macro by using the –U (for "undefine") option. Removing the initial definition is required if you wish to use the -D option twice in the same command line. The option has the form

cc –U*name*

where *name* is the macro name. For example, in the command

cc -DNEED=2 main.c -UNEED -DNEED=3 add.c

the -U options removes the previous definition of "NEED" and allows a new one.

2.15 Defining the Include Directories

You can explicitly define the directories containing include files by using the –I (for "include") option. This option adds the given directory to the list of directories containing include files. These directories are automatically searched whenever you give an include directive in which the filename is enclosed in angle brackets. The option has the form

–Idirectoryname

where *directoryname* is a valid pathname to a directory containing include files. For example, the command

cc –Imyinclude main.c

causes the compiler to search the directory *myinclude* for include files requested by the source file *main.c*.

The directories are searched in the order they are given and only until the given include file is found. The */usr/include* directory is the default include directory and is always searched first.

2.16 Error Messages

The cc command itself produces error messages. It also lets the XENIX C compiler, C preprocessor, C optimizer, assembler, and linker programs detect and announce any errors found in the source files or command options. The error messages are usually preceded by the name of the program which detected the error. If the error is severe, the cc command terminates and leaves all files unchanged. Otherwise, it proceeds with the compilation and linking of the given source files if you have given the appropriate commands.

Most error messages are generated by the C compiler. This displays messages about errors found during compilation such as incorrect syntax, undefined variables, and illegal use of operators. Error messages from the compiler begin with the name of the source file and list the number of the line containing the error.

The XENIX linker also generates many error messages. It displays messages about errors found during linking such as undefined symbols and misnamed libraries. The preprocessor, optimizer, and assembler also display messages if errors are found. For example, the preprocessor displays an error message if it cannot find an include file.

For convenience, you should use the XENIX C program checker *lint* before compiling your C source files. *Lint* performs detailed error checking on a source file and provide a list of actual errors and possible problems which may affect execution of the program. See Chapter 3, "Lint: A C Program Checker" for a description of *lint*.

Chapter 3
Lint: A C Program Checker

3.1 Introduction

This chapter explains how to use the C program checker *lint*. The program examines C source files and warns of errors or misconstructions that may cause errors during compilation of the file or during execution of the compiled file.

In particular, *lint* checks for:

Unused functions and variables

Unknown values in local variables

Unreachable statements and infinite loops

Unused and misused return values

Inconsistent types and type casts

Mismatched types in assignments

Nonportable and old fashioned syntax

Strange constructions

Inconsistent pointer alignment and expression evaluation order

The *lint* program and the C compiler are generally used together to check and compile C language programs. Although the C compiler compiles C language source files, it does not perform the sophisticated type and error checking required by many programs, though syntax is gone over. The *lint* program, provides additional checking of source files without compiling.

3.2 Invoking *lint*

You can invoke *lint* program by typing

lint [*option*] ... *filename* ... *lib* ...

where *option* is a command option that defines how the checker should operate, *filename* is the name of the C language source file to be checked, and *lib* is the name of a library to check. You can give more than one option, filename, or library name in the command. If you give two or more filenames, *lint* assumes that the files belong to the same program and checks the files accordingly. For example, the command

lint main.c add.c

treats *main.c* and *add.c* as two parts of a complete program.

If *lint* discovers errors or inconsistencies in a source file, it produces messages describing the problem. The message has the form

filename (*num*): *description*

where *filename* is the name of the source file containing the problem, *num* is the number of the line in the source containing the problem, and *description* is a description of the problem. For example, the message

main.c (3): warning: x unused in function main

shows that the variable "x", defined in line three of the source file *main.c*, is not used anywhere in the file.

3.3 Checking for Unused Variables and Functions

The *lint* program checks for unused variables and functions by seeing if each declared variable and function is used in at least once in the source file. The program considers a variable or function used if the name appears in at least one statement. It is not considered used if it only appears on the left side of on assignment. For example, in the following program fragment

```
main ()
{
        int x,y,z;

        x=1; y=2; z=x+y;
```

the variables "x" and "y" are considered used, but variable "z" is not.

Unused variables and functions often occur during the development of large programs. It is not uncommon for a programmer to remove all references to a variable or function from a source file but forget to remove its declaration. Such unused variables and functions rarely cause working programs to fail, but do make programs larger, harder to understand and change. Checking for unused variables and functions can also help you find variables or functions that you intended to used but accidentally have left out of the program.

Note that the *lint* program does not report a variable or function unused if it is explicitly declared with the **extern** storage class. Such a variable or function is assumed to be used in another source file.

You can direct *lint* to ignore all the external declarations in a source file by using the **-x** (for "external") option. The option causes the program checker to skip any declaration that begins with the **extern** storage class.

The option is typically used to save time when checking a program, especially if all external declarations are known to be valid.

Some programming styles require functions that perform closely related tasks to have the same number and type of arguments regardless of whether or not these arguments are used. Under normal operation, *lint* reports any argument not used as an unused variable, but you can direct *lint* to ignore unused arguments by using the –v option. The –v option causes *lint* to ignore all unused function arguments except for those declared with **register** storage class. The program considers unused arguments of this class to be a preventable waste of the register resources of the computer.

You can direct *lint* to ignore all unused variables and functions by using the –u (for "unused") option. This option prevents *lint* from reporting variables and functions it considers unused.

This option is typically used when checking a source file that contains just a portion of a large program. Such source files usually contain declarations of variables and functions that are intended to be used in other source files and are not explicitly used within the file. Since *lint* can only check the given file, it assumes that such variables or functions are unused and reports them as such.

3.4 Checking Local Variables

The *lint* program checks all local variables to see that they are set to a value before being used. Since local variables have either automatic or register storage class, their values at the start of the program or function cannot be known. Using such a variable before assigning a value to it is an error.

The program checks the local variables by searching for the first assignment in which the variable receives a value and the first statement or expression in which the variable is used. If the first assignment appears later than the first use, *lint* considers the variable inappropriately used. For example, in the program fragment

```
char c;

if( c != EOT )
        c = getchar();
```

lint warns that the the variable "c" is used before it is assigned.

If the variable is used in the same statement in which it is assigned for the first time, *lint* determines the order of evaluation of the statement and displays an appropriate message. For example, in the program fragment

```
int i,total;

scanf("%d", &i);
total = total + i;
```

lint warns that the variable "total" is used before it is set since it appears on the

rightside of the same statement that assigns its first value.

3.5 Checking for Unreachable Statements

The *lint* program checks for unreachable statements, that is, for unlabeled statements that immediately follow a **goto**, **break**, **continue**, or **return** statement. During execution of a program, the unreachable statements never receive execution control and are therefore considered wasteful. For example, in the program fragment

```
int x,y;

return (x+y);
exit (1);
```

the function call *exit* after the return statement is unreachable.

Unreachable statements are common when developing programs containing large case constructions or loops containing break and continue statements.

During normal operation, *lint* reports all unreachable break statements. Unreachable break statements are relatively common (some programs created by the *yacc* and *lex* programs contain hundreds), so it may be desirable to suppress these reports. You can direct *lint* to suppress the reports by using the –b option.

Note that *lint* assumes that all functions eventually return control, so it does not report as unreachable any statement that follows a function that takes control and never returns it. For example:

```
exit (1);
return;
```

the call to *exit* causes the **return** statement to become an unreachable statement, but *lint* does not report it as such.

3.6 Checking for Infinite Loops

The *lint* program checks for infinite loops and for loops which are never executed. For example, the statement

```
while (1) { }
```

and

```
for (;;) {}
```

are both considered infinite loops. While the statements

```
while (0) { }
```

or

```
for (0;0;) { }
```

are never executed.

It is relatively common for valid programs to have such loops, but they are generally considered errors.

3.7 Checking Function Return Values

The *lint* program checks that a function returns a meaningful value if necessary. Some functions return values which are never used; some programs incorrectly use function values that have never been returned. *Lint* addresses these problems in a number of ways.

Within a function definition, the appearance of both

```
return (expr);
```

and

```
return ;
```

statements is cause for alarm. In this case, *lint* produces the following error message:

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

It is difficult to detect when a function return is implied by the flow of control reaching the end of the given function. This is demonstrated with a simple example:

```
f (a)
{
        if (a)
                return (3);
        g ();
}
```

Note that if the variable "a" tests false, then *f* will call the function *g* and then return with no defined return value. This will trigger a report from *lint*. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong. In practice, potentially serious bugs can be discovered with this feature. It also accounts for a some of the noise messages produced by *lint*.

3.8 Checking for Unused Return Values

The *lint* program checks for cases where a function returns a value, but the value is usually ignored. *Lint* considers functions that return unused values to be inefficient, and functions that return rarely used values to be a result of bad programming style.

Lint also checks for cases where a function does not return a value but the value is used anyway. This is considered a serious error.

3.9 Checking Types

Lint enforces the type checking rules of C more strictly than the C compiler. The additional checking occurs in four major areas:

1. Across certain binary operators and implied assignments
2. At the structure selection operators
3. Between the definition and uses of functions
4. In the use of enumerations

There are a number of operators that have an implied balancing between types of operands. The assignment, conditional, and relational operators have this property. The argument of a **return** statement, and expressions used in initialization also 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 be intermixed with pointers to x's.

The type checking rules also require that, in structure references, the left operand of a pointer arrow symbol (->) be a pointer to a structure, the left operand of a period (.) 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**. Pointers can also be matched with the associated arrays. Aside from these relaxations in type checking, all actual arguments must agree in type with their declared counterparts.

For 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 assignment (=), initialization, equals (==), and not-equals (!=). Enumerations may also be function arguments and return values.

3.10 Checking Type Casts

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

```
p = 1 ;
```

where "p" is a character pointer. *Lint* reports this as suspect. But 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. On the other hand, if this code is moved to another machine, it should be looked at carefully. The –c option controls the printing of comments about casts. When –c is in effect, casts are not checked and all legal casts are passed without comment, no matter how strange the type mixing seems to be.

3.11 Checking for Nonportable Character Use

Lint flags certain comparisons and assignments as illegal or nonportable. For example, the fragment

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

works on some machines, but fails on machines where characters always take on positive values. The solution is to declare "c" an integer, since *getchar* is actually returning integer values. In any case, *lint* issues the message:

```
nonportable character comparison
```

A similar issue arises with bitfields. When assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true where on some machines bitfields are considered as signed quantities. While it may seem counter-intuitive to consider that a 2-bit field declared of type int cannot hold the value 3, the problem disappears if the bitfield is declared to have type unsigned.

3.12 Checking for Assignment of longs to ints

Bugs may arise from the assignment of a long to an int, because of a loss in

accuracy in the process. This may happen in programs that have been incompletely converted by changing type definitions with typedef. When a typedef variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to integer values, losing accuracy. Since there are a number of legitimate reasons for assigning longs to integers, you may wish to suppress detection of these assignments by using the **-a** option.

3.13 Checking for Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint*. The generated messages encourage better code quality, clearer style, and may even point out bugs. For example, in the statement

```
*p++ ;
```

the star (*) does nothing and *lint* prints:

```
null effect
```

The program fragment

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

is also strange since the test will never succeed. Similarly, the test

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

is equivalent to

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

which may not be the intended action. In these cases, *lint* prints the message:

```
degenerate unsigned comparison
```

If you use

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

then *lint* reports

```
constant in conditional context
```

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

Another construction detected by *lint* involves operator precedence. Bugs that 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 ) ...
```

or

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

probably do not do what is intended. The best solution is to parenthesize such expressions. *Lint* encourages this by printing an appropriate message.

Finally, *lint* checks variables that are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered bad style, usually unnecessary, and frequently a bug.

If you do not wish these heuristic checks, you can suppress them by using the **-h** option.

3.14 Checking for Use of Older C Syntax

Lint checks for older C constructions. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, ...) can 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* checks for occurrences of these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize "x" to 1. This causes syntactic difficulties. For example

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

looks somewhat like the beginning of a function declaration

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

and the compiler must read past "x" to determine what the declaration really is. The problem is even more perplexing when the initializer involves a macro. The current C syntax places an equal sign between the variable and the initializer:

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

This form is free of any possible syntactic ambiguity.

3.15 Checking Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due to alignment restrictions. For example, on some machines it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On other machines, however, double precision values must begin on even word boundaries; thus, not all such assignments make sense. *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.

3.16 Checking Expression Evaluation Order

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 up, function arguments will probably be best evaluated from right to left; on machines with a stack running down, left to right is probably best. Function calls embedded as arguments of other functions may or may not be treated in the same way as 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 on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the compiler, and 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 draw the comment:

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

3.17 Embedding Directives

There are occasions when the programmer is smarter than *lint*. There may be valid reasons for illegal type casts, functions with a variable number of arguments, and other constructions that *lint* flags. Moreover, as specified in the above sections, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Some way of communicating with *lint*, typically to turn off its output, is desirable. Therefore, a number of words are recognized by *lint* when they are embedded in comments in a C source file. These words are called directives. *Lint* directives are invisible to the compiler.

The first directive discussed concerns flow of control information. If a particular place in the program cannot be reached, this can be asserted at the appropriate spot in the program with the directive:

```
/* NOTREACHED */
```

Similarly, if you desire to turn off strict type checking for the next expression, use the directive:

```
/* NOSTRICT */
```

The situation reverts to the previous default after the next expression. The –v option can be turned on for one function with the directive:

```
/* ARGSUSED */
```

Comments about a variable number of arguments in calls to a function can be turned off by preceding the function definition with the directive:

```
/* VARARGS */
```

In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. Do this by following the VARARGS keyword immediately with a digit giving the number of arguments that should be checked. Thus:

```
/* VARARGS2 */
```

causesonly the first twoarguments to be checked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file, discussed in the next section.

3.18 Checking For Library Compatibility

Lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This testing is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* 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" directives can be used to specify features of the library functions.

Lint library files are processed like ordinary source files. The only difference is that functions that are defined in a library file, but are not used in a source file, draw no comments. *Lint* does not simulate a full library search algorithm, and checks to see if the source files contain redefinitions of library routines.

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

Lint library files are named "/usr/lib/ll*". The programmer may wish to examine the lint libraries directly to see what lint thinks a function should passed and return. Printed out, lint libraries also make satisfactory skeleton quick-reference cards.

Chapter 4
Make: A Program Maintainer

4.1 Introduction

The **make** program provides an easy way to automate the creation of large programs. Make reads commands from a user-defined "makefile" that lists the files to be created, the commands that create them, and the files from which they are created. When you direct **make** to create a program, it verifies that each file on which the program depends is up to date, then creates the program by executing the given commands. If a file is not up to date, **make** updates it before creating the program. Make updates a program by executing explicitly given commands, or one of the many built-in commands.

This chapter explains how to use **make** to automate medium-sized programming projects. It explains how to create makefiles for each project, and how to invoke **make** for creating programs and updating files. For more details about the program, see *make* (CP) in the XENIX *Reference Manual*.

4.2 Creating a Makefile

A makefile contains one or more lines of text called dependency lines. A dependency line shows how a given file depends on other files and what commands are required to bring a file up to date. A dependency line has the form

target ... : [*dependent* ...] [; *command* ...]

where *target* is the filename of the file to be updated, *dependent* is the filename of the file on which the target depends, and *command* is the XENIX command needed to create the target file. Each dependency line must have at least one command associated with it, even if it is only the null command (;).

You may give more than one target filename or dependent filename if desired. Each filename must be separated from the next by at least one space. The target filenames must be separated from the dependent filenames by a colon (:). Filenames must be spelled as defined by the XENIX system. Shell metacharacters, such as star (*) and question mark (?), can also be used.

You may give a sequence of commands on the same line as the target and dependent filenames, if you precede each command with a semicolon (;). You can give additional commands on following lines by beginning each line with a tab character. Commands must be given exactly as they would appear on a shell command line. The at sign (@) may be placed in front of a command to prevent **make** from displaying the command before executing it. Shell commands, such as *cd*(C), must appear on single lines; they must not contain the backslash (\) and newline character combination.

You may add a comment to a makefile by starting the comment with a number sign (#) and ending it with a newline character. All characters after the number sign are ignored. Comments may be place at the end of a dependency

line if desired. If a command contains a number sign, it must be enclosed in double quotation marks (").

If a dependency line is too long, you can continue it by typing a backslash (\) and a newline character.

The makefile should be kept in the same directory as the given source files. For convenience, the filenames **makefile**, **Makefile**, *s.makefile*, and *s.Makefile* are provided as default filenames. These names are used by **make** if no explicit name is given at invocation. You may use one of these names for your makefile, or choose one of your own. If the filename begins with the *s.* prefix, **make** assumes that it is an SCCS file and invokes the appropriate SCCS command to retrieve the lastest version of the file.

To illustrate dependency lines, consider the following example. A program named *prog* is made by linking three object files, *x.o*, *y.o*, and *z.o*. These object files are created by compiling the C language source files *x.c*, *y.c*, and *z.c*. Furthermore, the files *x.c* and *y.c* contain the line

```
#include "defs"
```

This means that *prog* depends on the three object files, the object files depend on the C source files, and two of the source files depend on the include file *defs*. You can represent these relationships in a makefile with the following lines.

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

In the first dependency line, *prog* is the target file and *x.o*, *y.o*, and *z.o* are its dependents. The command sequence

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

on the next line tells how to create *prog* if it is out of date. The program is out of date if any one of its dependents has been modified since *prog* was last created.

The second, third, and fourth dependency lines have the same form, with the *x.o*, *y.o*, and *z.o* files as targets and *x.c*, *y.c*, *z.c*, and *defs* files as dependents. Each dependency line has one command sequence which defines how to update the given target file.

4.3 Invoking Make

Once you have a makefile and wish to update and modify one or more target files in the file, you can invoke **make** by typing its name and optional arguments. The invocation has the form

make [*option*] ... [*macdef*] ... [*target*] ...

where *option* is a program option used to modify program operation, *macdef* is a macro definition used to give a macro a value or meaning, and *target* is the filename of the file to be updated. It must correspond to one of the target names in the makefile. All arguments are optional. If you give more than one argument, you must separate them with spaces.

You can direct **make** to update the first target file in the makefile by typing just the program name. In this case, **make** searches for the files **makefile**, **Makefile**, *s.makefile*, and *s.Makefile* in the current directory, and uses the first one it finds as the makefile. For example, assume that the current makefile contains the dependency lines given in the last section. Then the command

```
make
```

compares the current date of the *prog* program with the current date each of the object files *x.o*, *y.o*, and *z.o*. It recreates *prog* if any changes have been made to any object file since *prog* was last created. It also compares the current dates of the object files with the dates of the four source files *x.c*, *y.c*, *z.c*, or *defs*, and recreates the object files if the source files have changed. It does this before recreating *prog* so that the recreated object files can be used to recreate *prog*. If none of the source or object files have been altered since the last time *prog* was created, **make** announces this fact and stops. No files are changed.

You can direct **make** to update a given target file by giving the filename of the target. For example,

```
make x.o
```

causes **make** to recompile the *x.o* file, if the *x.c* or *defs* files have changed since the object file was last created. Similarly, the command

```
make x.o z.o
```

causes **make** to recompile *x.o* and *z.o* if the corresponding dependents have been modified. **Make** processes target names from the command line in a left to right order.

You can specify the name of the makefile you wish **make** to use by giving the **-f** option in the invocation. The option has the form

-f *filename*

where *filename* is the name of the makefile. You must supply a full pathname if the file is not in the current directory. For example, the command

make -f makeprog

reads the dependency lines of the makefile named **makeprog** found in the current directory. You can direct **make** to read dependency lines from the standard input by giving "-" as the *filename*. **Make** reads the standard input until the end-of-file character is encountered.

You may use the program options to modify the operation of the **make** program. The following list describes some of the options.

-p Prints the complete set of macro definitions and dependency lines in a makefile.

-i Ignores errors returned by XENIX commands.

-k Abandons work on the current entry, but continues on other branches that do not depend on that entry.

-s Executes commands without displaying them.

-r Ignores the built-in rules.

-n Displays commands but does not execute them. **Make** even displays lines beginning with the at sign (@).

-e Ignores any macro definitions that attempt to assign new values to the shell's environment variables.

-t Changes the modification date of each target file without recreating the files.

Note that **make** executes each command in the makefile by passing it to a separate invocation of a shell. Because of this, care must be taken with certain commands (e.g., cd and shell control commands) that have meaning only within a single shell process; the results are forgotten before the next line is executed. If an error occurs, **make** normally stops the command.

4.4 Using Pseudo-Target Names

It is often useful to include dependency lines that have pseudo-target names, i.e., names for which no files actually exist or are produced. Pseudo-target

names allow **make** to perform tasks not directly connected with the creation of a program, such as deleting old files or printing copies of source files. For example, the following dependency line removes old copies of the given object files when the pseudo-target name "cleanup" is given in the invocation of **make**.

```
cleanup :
        rm  x.o y.o z.o
```

Since no file exists for a given pseudo-target name, the target is always assumed to be out of date. Thus the associated command is always executed.

Make also has built-in pseudo-target names that modify its operation. The pseudo-target name ".IGNORE" causes **make** to ignore errors during execution of commands, allowing **make** to continue after an error. This is the same as the –i option. (*Make* also ignores errors for a given command if the command string begins with a hyphen (-).)

The pseudo-target name ".DEFAULT" defines the commands to be executed either when no built-in rule or user-defined dependency line exists for the given target. You may give any number of commands with this name. If ".DEFAULT" is not used and an undefined target is given, **make** prints a message and stops.

The pseudo-target name ".PRECIOUS" prevents dependents of the current target from being deleted when **make** is terminated using the INTERRUPT or QUIT key, and the pseudo-target name ".SILENT" has the same effect as the –**s** option.

4.5 Using Macros

An important feature of a makefile is that it can contain macros. A macro is a short name that represents a filename or command option. The macros can be defined when you invoke **make**, or in the makefile itself.

A macro definition is a line containing a name, an equal sign (=), and a value. The equal sign 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
LIBES =
```

The last definition assigns "LIBES" the null string. A macro that is never explicitly defined has the null string as its value.

A macro is invoked by preceding the macro name with a dollar sign; macro names longer than one character must be placed in parentheses. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations.

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

The last two invocations are identical.

Macros are typically used as placeholders for values that may change from time to time. For example, the following makefile uses a macro for the names of object files to be link and one for the names of the library.

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

If this makefile is invoked with the command

```
make
```

it will load the three object files with the *lex* library specified with the **-lln** option.

You may include a macro definition in a command line. A macro definition in a command line has the same form as a macro definition in a makefile. If spaces are to be used in the definition, double quotation marks must be used to enclose the definition. Macros in a command line override corresponding definitions found in the makefile. For example, the command

```
make "LIBES=-lln -lm"
```

loads assigns the library options **-lln** and **-lm** to "LIBES".

You can modify all or part of the value generated from a macro invocation without changing the macro itself by using the "substitution sequence". The sequence has the form

name : *st1* =[*st2*]

where *name* is the name of the macro whose value is to be modified, *st1* is the character or characters to be modified, and *st2* is the character or characters to replace the modified characters. If *st2* is not given, *st1* is replaced by a null character.

The substitution sequence is typically used to allow user-defined metacharacters in a makefile. For example, suppose that ".x" is to be used as a metacharacter for a prefix and suppose that a makefile contains the definition

```
FILES = prog1.x prog2.x prog3.x
```

Then the macro invocation

```
$(FILES : .x=.o)
```

generates the value

```
prog1.o prog2.o prog3.o
```

The actual value of "FILES" remains unchanged.

Make has five built-in macros that can be used when writing dependency lines. The following is a list of these macros.

$* Contains the name of the current target with the suffix removed. Thus if the current target is *prog.o*, **$*** contains *prog*. It may be used in dependency lines that redefine the built-in rules.

$@ Contains the full pathname of the current target. It may be used in dependency lines with user-defined target names.

$< Contains the filename of the dependent that is more recent than the given target. It may be used in dependency lines with built-in target names or the **.DEFAULT** pseudo-target name.

$? Contains the filenames of the dependents that are more recent than the given target. It may be used in dependency lines with user-defined target names.

$% Contains the filename of a library member. It may be used with target library names (see the section "Using Libraries" later in this chapter). In this case, **$@** contains the name of the library and **$%** contains the name of the library member.

You can change the meaning of a built-in macro by appending the **D** or **F** descriptor to its name. A built-in macro with the **D** descriptor contains the name of the directory containing the given file. If the file is in the current directory, the macro contains ".". A macro with the **F** descriptor contains the name of the given file with the directory name part removed. The **D** and **F** descriptor must not be used with the **$?** macro.

4.6 Using Shell Environment Variables

Make provides access to current values of the shell's environment variables such as "HOME", "PATH", and "LOGIN". Make automatically assigns the value of each shell variable in your environment to a macro of the same name. You can access a variable's value in the same way that you access the value of explicitly defined macros. For example, in the following dependency line, "$(HOME)" has the same value as the user's "HOME" variable.

```
prog :
        cc $(HOME)/x.o $(HOME)/y.o /usr/pub/z.o
```

Make assigns the shell variable values after it assigns values to the built-in macros, but before it assigns values to user-specified macros. Thus, you can override the value of a shell variable by explicitly assigning a value to the corresponding macro. For example, the following macro definition causes make to ignore the current value of the "HOME" variable and use */usr/pub* instead.

```
HOME = /usr/pub
```

If a makefile contains macro definitions that override the current values of the shell variables, you can direct make to ignore these definitions by using the –e option.

Make has two shell variables, "MAKE" and "MAKEFLAGS", that correspond to two special-purpose macros.

The "MAKE" macro provides a way to override the –n option and execute selected commands in a makefile. When "MAKE" is used in a command, make will always execute that command, even if –n has been given in the invocation. The variable may be set to any value or command sequence.

The "MAKEFLAGS" macro contains one or more make options, and can be used in invocations of make from within a makefile. You may assign any make options to "MAKEFLAGS" except –f, –p, and –d. If you do not assign a value to the macro, make automatically assigns the current options to it, i.e., the options given in the current invocation.

The "MAKE" and "MAKEFLAGS" variables, together with the –n option, are typically used to debug makefiles that generate entire software systems. For example, in the following makefile, setting "MAKE" to "make" and invoking this file with the –n options displays all the commands used to generate the programs *prog1*, *prog2*, and *prog3* without actually executing them.

```
system : prog1 prog2 prog3
        @echo System complete.

prog1 : prog1.c
        $(MAKE) $(MAKEFLAGS) prog1

prog2 : prog2.c
        $(MAKE) $(MAKEFLAGS) prog2

prog3 : prog3.c
        $(MAKE) $(MAKEFLAGS) prog3
```

4.7 Using the Built-In Rules

Make provides a set of built-in dependency lines, called built-in rules, that automatically check the targets and dependents given in a makefile, and create up-to-date versions of these files if necessary. The built-in rules are identical to user-defined dependency lines except that they use the suffix of the filename as the target or dependent instead of the filename itself. For example, make automatically assumes that all files with the suffix *.o* have dependent files with the suffixes *.c* and *.s*.

When no explicit dependency line for a given file is given in a makefile, make automatically checks the default dependents of the file. It then forms the name of the dependents by removing the suffix of the given file and appending the predefined dependent suffixes. If the given file is out of date with respect to these default dependents, make searches for a built-in rule that defines how to create an up-to-date version of the file, then executes it. There are built-in rules for the following files.

.o	Object file
.c	C source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.l	Lex source grammar

For example, if the file *x.o* is needed and there is an *x.c* in the description or directory, it is compiled. If there is also an *x.l*, that grammar would be run through *lex* before compiling the result.

The built-in rules are designed to reduce the size of your makefiles. They provide the rules for creating common files from typical dependents. Reconsider the example given in the section "Creating a Makefile". In this example, the program *prog* depended on three object files *x.o*, *y.o*, and *z.o*. These files in turn depended on the C language source files *x.c*, *y.c*, and *z.c*.

The files *x.c* and *y.c* also depended on the include file *defs*. In the original example each dependency and corresponding command sequence was explicitly given. Many of these dependency lines were unnecessary, since the built-in rules could have been used instead. The following is all that is needed to show the relationships between these files.

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

x.o y.o: defs
```

In this makefile, *prog* depends on three object files, and an explicit command is given showing how to update *prog*. However, the second line merely shows that two objects files depend on the include file *defs*. No explicit command sequence is given on how to update these files if necessary. Instead, **make** uses the built-in rules to locate the desired C source files, compile these files, and create the necessary object files.

4.8 Changing the Built-in Rules

You can change the built-in rules by redefining the macros used in these lines or by redefining the commands associated with the rules. You can display a complete list of the built-in rules and the macros used in the rules by typing

```
make -fp - 2>/dev/null </dev/null
```

The rules and macros are displayed at the standard output.

The macros of the built-in dependency lines define the names and options of the compilers, program generators, and other programs invoked by the built-in commands. Make automatically assigns a default value to these macros when you start the program. You can change the values by redefining the macro in your makefile. For example, the following built-in rule contains three macros, "CC", "CFLAGS", and "LOADLIBES".

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

You can redefine any of these macros by placing the appropriate macro definition at the beginning of the makefile.

You can redefine the action of a built-in rule by giving a new rule in your makefile. A built-in rule has the form

```
suffix-rule :
        command
```

where *suffix-rule* is a combination of suffixes showing the relationship of the implied target and dependent, and *command* is the XENIX command required

to carry out the rule. If more than one command is needed, they are given on separate lines.

The new rule must begin with an appropriate *suffix-rule*. The available *suffix-rules* are

.c	.c
.sh	.sh
.c.o	.c .o
.c .c	.s.o
.s .o	.y.o
.y .o	.l.o
.l .o	.y.c
.y .c	.l.c
.c.a	.c .a
.s.a	.h .h

A tilde ( ) indicates an SCCS file. A single suffix indicates a rule that makes an executable file from the given file. For example, the suffix rule ".c" is for the built-in rule that creates an executable file from a C source file. A pair of suffixes indicates a rule that makes one file from the other. For example, ".c.o" is for the rule that creates an object file (*.o*) file from a corresponding C source file (*.c*).

Any commands in the rule may use the built-in macros provided by make. For example, the following dependency line redefines the action of the *.c.o* rule.

```
.c.o :
        cc68 $< -c $*.o
```

If necessary, you can also create new *suffix-rules* by adding a list of new suffixes to a makefile with ".SUFFIXES". This pseudo-target name defines the suffixes that may be used to make *suffix-rules* for the built-in rules. The line has the form

```
.SUFFIXES: suffix ...
```

where *suffix* is usually a lowercase letter preceded by a dot (.). If more than one suffix is given, you must use spaces to separate them.

The order of the suffixes is significant. Each suffix is a dependent of the suffixes preceding it. For example, the suffix list

```
.SUFFIXES: .o .c .y .l .s
```

causes *prog.c* to be a dependent of *prog.o*, and *prog.y* to be a dependent of *prog.c*.

You can create new *suffix-rules* by combining dependent suffixes with the suffix of the intended target. The dependent suffix must appear first.

f a ".SUFFIXES" list appears more than once in a makefile, the suffixes are ombined into a single list. If a ".SUFFIXES" is given that has no list, all uffixes are ignored.

1.9 Using Libraries

ou can direct **make** to use a file contained in an archive library as a target or lependent. To do this you must explicitly name the file you wish to access by ising a library name. A library name has the form

lib(*member-name*)

here *lib* is the name of the library containing the file, and *member-name* is the ame of the file. For example, the library name

libtemp.a(print.o)

efers to the object file *print.o* in the archive library *libtemp.a*.

ou can create your own built-in rules for archive libraries by adding the *.a* uffix to the suffix list, and creating new suffix combinations. For example, the ombination ".c.a" may be used for a rule that defines how to create a library ember from a C source file. Note that the dependent suffix in the new ombination must be different than the suffix of the ultimate file. For example, he combination ".c.a" can be used for a rule that creates *.o* files, but not for one at creates *.c* files.

he most common use of the library naming convention is to create a makefile at automatically maintains an archive library. For example, the following ependency lines define the commands required to create a library, named *lib*, ontaining up to date versions of the files *file1.o*, *file2.o*, and *file3.o*.

```
lib:
        lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up to date
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

he *.c.a* rule shows how to redefine a built-in rule for a library. In the following ample, the built-in rule is disabled, allowing the first dependency to create e library.

```
lib:
        lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv lib $?
        rm $?
        @echo lib is now up to date
.c.a:;
```

In this example, a substitution sequence is used to change the value of the "$?" macro from the names of the object files "file1.o", "file2.o", and "file3.o" to "file1.c", "file2.c", and "file3.c".

4.10 Troubleshooting

Most difficulties in using make arise from make's specific meaning of dependency. If the file *x.c* has the line

```
#include "defs"
```

then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To determine which commands make will execute, without actually executing them, use the –n option. For example, the command

```
make -n
```

prints out the commands make would normally execute without actually executing them.

The debugging option –d causes make to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

If a change to a file is absolutely certain to be benign (e.g., adding a new definition 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
```

which stands for touch silently, causes the relevant files to appear up to date.

4.11 Using Make: An Example

As an example of the use of make, examine the makefile, given in Figure 4-1, used to maintain the make itself. The code for make is spread over a number

of C source files and a *yacc* grammar.

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and makefile:

```
cc -c vers.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc vers.o main.o ... dosys.o gram.o -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the makefile, **make** found them by using its suffix rules and issued the needed commands. The string of digits results from the **size make** command.

The last few targets in the makefile are useful maintenance sequences. The *print* target prints only the files that have been changed since the last **make print** command. A zero-length file, *print*, is maintained to keep track of the time of the printing; the **$?** macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the **P** macro.

Figure 4-1. Makefile Contents

```
# Description file for the make command

# Macro definitions below
P = lpr
FILES = Makefile vers.c defs main.c doname.c misc.c files.c dosys.c\
        gram.y lex.c
OBJECTS = vers.o main.o ... dosys.o gram.o
LIBES=
LINT = lint -p
CFLAGS = -O

#targets: dependents
#<TAB>actions

make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS): defs
gram.o: lex.c

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

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

print: $(FILES)  # print recently changed files
        pr $? | $P
        touch print

test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

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

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Chapter 5
SCCS: A Source Code Control System

5.1 Introduction

The Source Code Control System (SCCS) is a collection of XENIX commands that create, maintain, and control special files called SCCS files. The SCCS commands let you create and store multiple versions of a program or document in a single file, instead of one file for each version. The commands let you retrieve any version you wish at any time, make changes to this version, and save the changes as a new version of the file in the SCCS file.

The SCCS system is useful wherever you require a compact way to store multiple versions of the same file. The SCCS system provides an easy way to update any given version of a file and explicitly record the changes made. The commands are typically used to control changes to multiple versions of source programs, but may also be used to control multiple versions of manuals, specifications, and other documentation.

This chapter explains how to make SCCS files, how to update the files contained in SCCS files, and how to maintain the SCCS files once they are created. The following sections describe the basic information you need to start using the SCCS commands. Later sections describe the commands in detail.

5.2 Basic Information

This section provides some basic information about the SCCS system. In particular, it describes

- Files and directories
- Deltas and SIDs
- SCCS working files
- SCCS command arguments
- File administration

5.2.1 Files and Directories

All SCCS files (also called s-files) are originally created from text files containing documents or programs created by a user. The text files must have been created using a XENIX text editor such as vi. Special characters in the files are allowed only if they are also allowed by the given editor.

To simplify s-file storage, all logically related files (e.g., files belonging to the same project) should be kept in the same directory. Such directories should contain s-files only, and should have read and examine permission for everyone, and write permission for the user only.

Note that you must not use the XENIX **link** command to create multiple copies of an s-file.

5.2.2 Deltas and SIDs

Unlike an ordinary text file, an SCCS file (or s-file for short) contains nothing more than lists of changes. Each list corresponds to the changes needed to construct exactly one version of the file. The lists can then be combined to create the desired version from the original.

Each list of changes is called a "delta". Each delta has an identification string called an "SID". The SID is a string of at least two, and at most four, numbers separated by periods. The numbers name the version and define how it is related to other versions. For example, the first delta is usually numbered 1.1 and the second 1.2.

The first number in any SID is called the "release number". The release number usually indicates a group of versions that are similar and generally compatible. The second number in the SID is the "level number". It indicates major differences between files in the same release.

An SID may also have two optional numbers. The "branch number", the optional third number, indicates changes at a particular level, and the "sequence number", the fourth number, indicates changes at a particular branch. For example, the SIDs 1.1.1.1 and 1.1.1.2 indicate two new versions that contain slight changes to the original delta 1.1.

An s-file may at any time contain several different releases, levels, branches, and sequences of the same file. In general, the maximum number of releases an s-file may contain is 9999, that is, release numbers may range from 1 to 9999. The same limit applies to level, branch, and sequence numbers.

When you create a new version, the SCCS system usually creates a new SID by incrementing the level number of the original version. If you wish to create a new release, you must explicitly instruct the system to do so. A change to a release number indicates a major new version of the file. How to create a new version of a file and change release numbers is described later.

The SCCS system creates a branch and sequence number for the SID of a new version, if the next higher level number already exists. For example, if you change version 1.3 to create a version 1.4 and then change 1.3 again, the SCCS system creates a new version named 1.3.1.1.

Version numbers can become quite complicated. In general, it is wise to keep the numbers as simple as possible by carefully planning the creation of each new version.

5.2.3 SCCS Working Files

The SCCS system uses several different kinds of files to complete its tasks. In general, these files contain either actual text, or information about the commands in progress. For convenience, the SCCS system names these files by placing a prefix before the name of the original file from which all versions were made. The following is a list of the working files.

s-file — A permanent file that contains all versions of the given text file. The versions are stored as deltas, that is, lists of changes to be applied to the original file to create the given version. The name of an s-file is formed by placing the file prefix *s.* at the beginning of the original filename.

x-file — A temporary copy of the s-file. It is created by SCCS commands which change the s-file. It is used instead of the s-file to carry out the changes. When all changes are complete, the SCCS system removes the original s-file and gives the x-file the name of the original s-file. The name of the x-file is formed by placing the prefix *x.* at the beginning of the original file.

g-file — An ordinary text file created by applying the deltas in a given s-file to the original file. The g-file represents a copy of the given version of the original file, and as such receives the same filename as the original. When created, a g-file is placed in the current working directory of the user who requested the file.

p-file — A special file containing information about the versions of an s-file currently being edited. The p-file is created when a g-file is retrieved from the s-file. The p-file exists until all currently retrieved files have been saved in the s-file; it is then deleted. The p-file contains one or more entries describing the SID of the retrieved g-file, the proposed SID of the new, edited g-file, and the login name of the user who retrieved the g-file. The p-file name is formed by placing the prefix *p.* at the beginning of the original filename.

z-file — A lock file used by SCCS commands to prevent two users from updating a single SCCS file at the same time. Before a command modifes an SCCS file, it creates a z-file and copies its own process ID to it. Any other command which attempts to access the file while the z-file is present displays an error message and stops. When the original command has finished its tasks, it deletes the z-file before stopping. The z-file name is formed by placing the prefix *z.* at the beginning of the original filename.

l-file — A special file containing a list of the deltas required to create a given version of a file. The l-file name is formed by placing the prefix *l.* at the beginning of the original filename.

d-file A temporary copy of the g-file used to generate a new delta.

q-file A temporary file used by the **delta** command when updating the p-file. The file is not directly accessible.

In general, a user never directly accesses x-files, z-files, d-files, or q-files. If a system crash or similar situation abnormally terminates a command, the user may wish delete these files to ensure proper operation of subsequent SCCS commands.

5.2.4 SCCS Command Arguments

Almost all SCCS commands accept two types of arguments: options and filenames. These appear in the SCCS command line immediately after the command name.

An option indicates a special action to be taken by the given SCCS command. An option is usually a lowercase letter preceded by a minus sign (-). Some options require an additional name or value.

A filename indicates the file to be acted on. The syntax for SCCS filenames is like other XENIX filename syntax. Appropriate pathnames must be given if required. Some commands also allow directory names. In this case, all files in the directory are acted on. If the directory contains non-SCCS and unreadable files, these are ignored. A filename must not begin with a minus sign (-).

The special symbol – may be used to cause the given command to read a list of filenames from the standard input. These filenames are then used as names for the files to be processed. The list must terminate with an end-of-file character.

Any options given with a command apply to all files. The SCCS commands process the options before any filenames, so the options may appear anywhere on the command line.

Filenames are processed left to right. If a command encounters a fatal error, it stops processing the current file and, if any other files have been given, begins processing the next.

5.2.5 File Administrator

Every SCCS file requires an administrator to maintain and keep the file in order. The administrator is usually the user who created the file and therefore owns it. Before other users can access the file, the administrator must ensure that they have adequate access. Several SCCS commands let the administrator define who has access to the versions in a given s-file. These are described later.

5.3 Creating and Using S-files

The s-file is the key element in the SCCS system. It provides compact storage for all versions of a given file and automatic maintenance of the relationships between the versions.

This section explains how to use the **admin**, **get**, and **delta** commands to create and use s-files. In particular, it describes how to create the first version of a file, how to retrieve versions for reading and editing, and how to save new versions.

5.3.1 Creating an S-file

You can create an s-file from an existing text file using the –i (for "initialize") option of the **admin** command. The command has the form

admin –i*filename* *s.filename*

where –i*filename* gives the name of the text file from which the s-file is to be created, and *s.filename* is the name of the new s-file. The name must begin with *s.* and must be unique; no other s-file in the same directory may have the same name. For example, suppose the file named *demo.c* contains the short C language program

```
#include <stdio.h>

main ()
{
printf("This is version 1.1 \n");
}
```

To create an s-file, type

admin –idemo.c s.demo.c

This command creates the s-file *s.demo.c*, and copies the first delta describing the contents of *demo.c* to this new file. The first delta is numbered 1.1.

After creating an s-file, the original text file should be removed using the **rm** command, since it is no longer needed. If you wish to view the text file or make changes to it, you can retrieve the file using the **get** command described in the next section.

When first creating an s-file, the **admin** command may display the warning message

No id keywords (cm7)

In general, this message can be ignored unless you have specifically included keywords in your file (see the section, "Using Identification Keywords" later in this chapter).

Note that only a user with write permission in the directory containing the s-file may use the **admin** command on that file. This protects the file from administration by unauthorized users.

5.3.2 Retrieving a File for Reading

You can retrieve a file for reading from a given s-file by using the **get** command. The command has the form

get *s.filename* ...

where *s.filename* is the name of the s-file containing the text file. The command retrieves the lastest version of the text file and copies it to a regular file. The file has the same name as the s-file but with the *s.* removed. It also has read-only file permissions. For example, suppose the s-file *s.demo.c* contains the first version of the short C program shown in the previous section. To retrieve this program, type

```
get s.demo.c
```

The command retrieves the program and copies it to the file named *demo.c*. You may then display the file just as you do any other text file.

The command also displays a message which describes the SID of the retrieved file and its size in lines. For example, after retrieving the short C program from *s.demo.c*, the command displays the message

```
1.1
6 lines
```

You may also retrieve more than one file at a time by giving multiple s-file names in the command line. For example, the command

```
get s.demo.c s.def.h
```

retrieves the contents of the s-files *s.demo.c* and *s.def.h* and copies them to the text files *demo.c* and *def.h*. When giving multiple s-file names in a command, you must separate each with at least one space. When the **get** command displays information about the files, it places the corresponding filename before the relevent information.

5.3.3 Retrieving a File for Editing

You can retrieve a file for editing from a given s-file by using the –e (for "editing") option of the **get** command. The command has the form

get –e *s.filename* ...

where *s.filename* is the name of the s-file containing the text file. You may give more than one filename if you wish. If you do, you must separate each name with a space.

The command retrieves the lastest version of the text file and copies it to an ordinary text file. The file has the same name as the s-file but with the *s.* removed. It has read and write file permissions. For example, suppose the s-file *s.demo.c* contains the first version of a C program. To retrieve this program, type

get –e s.demo.c

The command retrieves the program and copies it to the file named *demo.c*. You may edit the file just as you do any other text file.

If you give more than one filename, the command creates files for each corresponding s-file. Since the –e option applies to all the files, you may edit each one.

After retrieving a text file, the command displays a message giving the SID of the file and its size in lines. The message also displays a proposed SID, that is, the SID for the new version after editing. For example, after retrieving the six-line C program in *s.demo.c*, the command displays the message

1.1
new delta 1.2
6 lines

The proposed SID is 1.2. If more than one file is retrieved, the corresponding filename precedes the relevant information.

Note that any changes made to the text file are not immediately copied to the corresponding s-file. To save these changes you must use the **delta** command described in the next section. To help keep track of the current file version, the **get** command creates another file, called a p-file, that contains information about the text file. This file is used by a subsequent **delta** command when saving the new version. The p-file has the same name as the s-file but begins with a *p.*. The user must not access the p-file directly.

5.3.4 Saving a New Version of a File

You can save a new version of a text file by using the **delta** command. The command has the form

delta *s.filename*

where *s.filename* is the name of the s-file from which the modified text file was retrieved. For example, to save changes made to a C program in the file *demo.c* (which was retrieved from the file *s.demo.c*), type

```
delta s.demo.c
```

Before saving the new version, the **delta** command asks for comments explaining the nature of the changes. It displays the prompt

```
comments?
```

You may type any text you think appropriate, up to 512 characters. The comment must end with a newline character. If necessary, you can start a new line by typing a backslash (\) followed by a newline character. If you do not wish to include a comment, just type a newline character.

Once you have given a comment, the command uses the information in the corresponding p-file to compare the original version with the new version. A list of all the changes is copied to the s-file. This is the new delta.

After a command has copied the new delta to the s-file, it displays a message showing the new SID and the number of lines inserted, deleted, or left unchanged in the new version. For example, if the C program has been changed to

```
#include <stdio.h>

main ()
{
int i = 2;

printf("This is version 1.%d 0, i);
}
```

the command displays the message

```
1.2
3 inserted
1 deleted
5 unchanged
```

Once a new version is saved, the next **get** command retrieves the new version.

The command ignores previous versions. If you wish to retrieve a previous version, you must use the –r option of the get command as described in the next section.

5.3.5 Retrieving a Specific Version

You can retrieve any version you wish from an s-file by using the –r (for "retrieve") of the get command. The command has the form

get [–e] –r*SID* *s.filename* ...

where –e is the edit option, –r*SID* gives the SID of the version to be retrieved, and *s.filename* is the name of the s-file containing the file to be retrieved. You may give more than one filename. The names must be separated with spaces.

The command retrieves the given version and copies it to the file having the same name as s-file but with the *s.* removed. The file has read-only permission unless you also give the –e option. If multiple filenames are given, one text file of the given version is retrieved from each. For example, the command

```
get –r1.1 s.demo.c
```

retrieves version 1.1 from the s-file *s.demo.c*, but the command

```
get –e –r1.1 s.demo.c  s.def.h
```

retrieves for editing a version 1.1 from both *s.demo.c* and *s.def.h*. If you give the number of a version that does not exist, the command displays an error message.

You may omit the level number of a version number if you wish, that is, just give a release number. If you do, the command automatically retrieves the most recent version having the same release number. For example, if the most recent version in the file *s.demo.c* is numbered 1.4, the command

```
get –r1 s.demo.c
```

retrieves the version 1.4. If there is no version with the given release number, the command retrieves the most recent version in the previous release.

5.3.6 Changing the Release Number of a File

You can direct the delta command to change the release number of a new version of a file by using the –r option of the get command. In this case, the get command has the form

get –e –r*rel-num* *s.filename* ...

where –e is the required edit option, *–rrel-num* gives the new release number of the file, and *s.filename* gives the name of the s-file containing the file to be retrieved. The new release number must be an entirely new number, that is, no existing version may have this number. You may give more than one filename.

The command retrieves the most recent version from the s-file, then copies the new release number to the p-file. On the subsequent **delta** command, the new version is saved using the new release number and level number 1. For example, if the most recent version in the s-file *s.demo.c* is 1.4, the command

```
get –e –r2 s.demo.c
```

causes the subsequent **delta** to save a new version 2.1, not 1.5. The new release number applies to the new version only; the release numbers of previous versions are not affected. Therefore, if you edit version 1.4 (from which 2.1 was derived) and save the changes, you create a new version 1.5. Similarly, if you edit version 2.1, you create a new version 2.2.

As before, the **get** command also displays a message showing the current version number, the proposed version number, and the size of the file in lines. Similarly, the subsequent **delta** command displays the new version number and the number of lines inserted, deleted, and unchanged in the new file.

5.3.7 Creating a Branch Version

You can create a branch version of a file by editing a version that has been previously edited. A branch version is simply a version whose SID contains a branch and sequence number.

For example, if version 1.4 already exists, the command

```
get –e –r1.3 s.demo.c
```

retrieves version 1.3 for editing and gives 1.3.1.1 as the proposed SID.

In general, whenever **get** discovers that you wish to edit a version that already has a succeeding version, it uses the first available branch and sequence numbers for the proposed SID. For example, if you edit version 1.3 a third time, **get** gives 1.3.2.1 as the proposed SID.

You can save a branch version just like any other version by using the **delta** command.

5.3.8 Retrieving a Branch Version

You can retrieve a branch version of a file by using the **–r** option of the **get** command. For example, the command

```
get -r1.3.1.1 s.demo.c
```

retrieves branch version 1.3.1.1.

You may retrieve a branch version for editing by using the -e option of the get command. When retrieving for editing, get creates the proposed SID by incrementing the sequence number by one. For example, if you retrieve branch version 1.3.1.1 for editing, get gives 1.3.1.2 as the proposed SID.

As always, the command displays the version number and file size. If the given branch version does not exist, the command displays an error message.

You may omit the sequence number if you wish. In this case, the command retrieves the most recent branch version with the given branch number. For example, if the most recent branch version in the s-file *s.def.h* is 1.3.1.4, the command

```
get -r1.3.1 s.def.h
```

retrieves version 1.3.1.4.

5.3.9 Retrieving the Most Recent Version

You can always retrieve the most recent version of a file by using the -t option with the get command. For example, the command

```
get -t s.demo.c
```

retrieves the most recent version from the file *s.demo.c*. You may combine the -r and -t options to retrieve the most recent version of a given release number. For example, if the most recent version with release number 3 is 3.5, then the command

```
get -r3 -t s.demo.c
```

retrieves version 3.5. If a branch version exists that is more recent than version 3.5 (e.g., 3.2.1.5), then the above command retrieves the branch version and ignores version 3.5.

5.3.10 Displaying a Version

You can display the contents of a version at the standard output by using the -p option of the get command. For example, the command

```
get -p s.demo.c
```

displays the most recent version in the s-file *s.demo.c* at the standard output. Similarly, the command

```
get -p -r2.1 s.demo.c
```

displays version 2.1 at the standard output.

The **-p** option is useful for creating g-files with user-supplied names. This option also directs all output normally sent to the standard output, such as the SID of the retrieved file, to the standard error file. Thus, the resulting file contains only the contents of the given version. For example, the command

```
get -p s.demo.c >version.c
```

copies the most recent version in the s-file *s.demo.c* to the file *version.c*. The SID of the file and its size is copied to the standard error file.

5.3.11 Saving a Copy of a New Version

The **delta** command normally removes the edited file after saving it in the s-file. You can save a copy of this file by using the **-n** option of the **delta** command. For example, the command

```
delta -n s.demo.c
```

first saves a new version in the s-file *s.demo.c*, then saves a copy of this version in the file *demo.c*. You may display the file as desired, but you cannot edit the file.

5.3.12 Displaying Helpful Information

An SCCS command displays an error message whenever it encounters an error in a file. An error message has the form

ERROR [*filename*]: *message* (*code*)

where *filename* is the name of the file being processed, *message* is a short description of the error, and *code* is the error code.

You may use the error code as an argument to the **help** command to display additional information about the error. The command has the form

```
help code
```

where *code* is the error code given in an error message. The command displays one or more lines of text that explain the error and suggest a possible remedy. For example, the command

```
help co1
```

displays the message

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

The **help** command can be used at any time.

5.4 Using Identification Keywords

The SCCS system provides several special symbols, called identification keywords, which may be used in the text of a program or document to represent a predefined value. Keywords represent a wide range of values, from the creation date and time of a given file, to the name of the module containing the keyword. When a user retrieves the file for reading, the SCCS system automatically replaces any keywords it finds in a given version of a file with the keyword's value.

This section explains how keywords are treated by the various SCCS commands, and how you may use the keywords in your own files. Only a few keywords are described in this section. For a complete list of the keywords, see the section *get*(CP) in the XENIX *Reference Manual*.

5.4.1 Inserting a Keyword into a File

You may insert a keyword into any text file. A keyword is simply an uppercase letter enclosed in percent signs (%). No special characters are required. For example, "%I%" is the keyword representing the SID of the current version, and "%H%" is the keyword representing the current date.

When the program is retrieved for reading using the **get** command, the keywords are replaced by their current values. For example, if the "%M%", "%I%", and "%H" keywords are used in place of the module name, the SID, and the current data in a program statement

```
char header(100) = {" %M%  %I%  %H% "};
```

then these keywords are expanded in the retrieved version of the program

```
char header(100) = {" MODNAME  2.3  07/07/77 "};
```

The **get** command does not replace keywords when retrieving a version for editing. The system assumes that you wish keep the keywords (and not their values) when you save the new version of the file.

To indicate that a file has no keywords, the **get**, **delta**, and **admin** commands display the message

No id keywords (cm7)

This message is normally treated as a warning, letting you know that no keywords are present. However, you may change the operation of the system to make this a fatal error, as explained later in this chapter.

5.4.2 Assigning Values to Keywords

The values of most keywords are predefined by the system, but some, such as the value for the "%M%" keyword can be explicitly defined by the user. To assign a value to a keyword, you must set the corresponding s-file flag to the desired value. You can do this by using the **-f** option of the **admin** command.

For example, to set the %M% keyword to "cdemo", you must set the **m** flag as in the command

admin -fmcdemo s.demo.c

This command records "cdemo" as the current value of the %M% keyword. Note that if you do not set the **m** flag, the SCCS system uses the name of the original text file for %M% by default.

The **t** and **q** flags are also associated with keywords. A description of these flags and the corresponding keywords can be found in the section *get*(CP) in the XENIX *Reference Manual*. You can change keyword values at any time.

5.4.3 Forcing Keywords

If a version is found to contain no keywords, you can force a fatal error by setting the **i** flag in the given s-file. The flag causes the **delta** and **admin** commands to stop processing of the given version and report an error. The flag is useful for ensuring that keywords are used properly in a given file.

To set the **i** flag, you must use the **-f** option of the **admin** command. For example, the command

admin -fi s.demo.c

sets the **i** flag in the s-file *s.demo.c*. If the given version does not contain keywords, subsequent **delta** or **admin** commands that access this file print an error message.

Note that if you attempt to set the **i** flag at the same time as you create an s-file, and if the initial text file contains no keywords, the **admin** command displays a fatal error message and stops without creating the s-file.

5.5 Using S-file Flags

An s-file flag is a special value that defines how a given SCCS command will operate on the corresponding s-file. The s-file flags are stored in the s-file and are read by each SCCS command before it operates on the file. S-file flags affect operations such as keyword checking, keyword replacement values, and default values for commands.

This section explains how to set and use s-file flags. It also describes the action of commonly-used flags. For a complete description of all flags, see the section *admin*(CP) in the XENIX *Reference Manual*.

5.5.1 Setting S-file Flags

You can set the flags in a given s-file by using the –f option of the **admin** command. The command has the form

admin –f*flag s.filename*

where –f*flag* gives the flag to be set, and *s.filename* gives the name of the s-file in which the flag is to be set. For example, the command

admin –fi s.demo.c

sets the i flag in the s-file *s.demo.c*.

Note that some s-file flags take values when they are set. For example, the **m** flag requires that a module name be given. When a value is required, it must immediately follow the flag name, as in the command

admin –fmdmod s.demo.c

which sets the **m** flag to the module name "dmod".

5.5.2 Using the i Flag

The **i** flag causes the **admin** and **delta** commands to print a fatal error message and stop, if no keywords are found in the given text file. The flag is used to prevent a version of a file, which contains expanded keywords, from being saved as a new version. (Saving an expanded version destroys the keywords for all subsequent versions).

When the i flag is set, each new version of a file must contain at least one keyword. Otherwise, the version cannot be saved.

5.5.3 Using the d Flag

The d flag gives the default SID for versions retrieved by the **get** command. The flag takes an SID as its value. For example, the command

```
admin -fd1.1 s.demo.c
```

sets the default SID to 1.1. A subsequent **get** command which does not use the **-r** option will retrieve version 1.1.

5.5.4 Using the v Flag

The v flag allows you to include modification requests in an s-file. Modification requests are names or numbers that may be used as a shorthand means of indicating the reason for each new version.

When the v flag is set, the **delta** command asks for the modification requests just before asking for comments. The v flag also allows the **-m** option to be used in the **delta** and **admin** commands.

5.5.5 Removing an S-file Flag

You can remove an s-file flag from an s-file by using the **-d** option of the **admin** command. The command has the form

```
admin -dflag s.filename
```

where **-d***flag* gives the name of the flag to be removed and *s.filename* is the name of the s-file from which the flag is to be removed. For example, the command

```
admin -di s.demo.c
```

removes the i flag from the s-file *s.demo.c*. When removing a flag which takes a value, only the flag name is required. For example, the command

```
admin -dm s.demo.c
```

removes the m flag from the s-file.

The **-d** and **-i** options must not be used at the same time.

5.6 Modifying S-file Information

Every s-file contains information about the deltas it contains. Normally, this information is maintained by the SCCS commands and is not directly accessible

by the user. Some information, however, is specific to the user who creates the s-file, and may be changed as desired to meet the user's requirements. This information is kept in two special parts of the s-file called the "delta table" and the "description field".

The delta table contains information about each delta, such as the SID and the date and time of creation. It also contains user-supplied information, such as comments and modification requests. The description field contains a user-supplied description of the s-file and its contents. Both parts can be changed or deleted at any time to reflect changes to the s-file contents.

5.6.1 Adding Comments

You can add comments to an s-file by using the –y option of the **delta** and **admin** commands. This option causes the given text to be copied to the s-file as the comment for the new version. The comment may be any combination of letters, digits, and punctuation symbols. No embedded newline characters are allowed. If spaces are used, the comment must be enclosed in double quotes. The complete command must fit on one line. For example, the command

```
delta –y"George Wheeler"  s.demo.c
```

saves the comment "George Wheeler" in the s-file *s.demo.c*.

The –y option is typically used in shell procedures as part of an automated approach to maintaining files. When the option is used, the **delta** command does not print the corresponding comment prompt, so no interaction is required. If more than one s-file is given in the command line, the given comment applies to them all.

5.6.2 Changing Comments

You can change the comments in a given s-file by using the **cdc** command. The command has the form

```
cdc –rSID  s.filename
```

where –r*SID* gives the SID of the version whose comment is to be changed, and *s.filename* is the name of the s-file containing the version. The command asks for a new comment by displaying the prompt

```
comments?
```

You may type any sequence of characters up to 512 characters long. The sequence may contain embedded newline characters if they are preceded by a backslash (\). The sequence must be terminated with a newline character. For example, the command

```
cdc -r3.4 s.demo.c
```

prompts for a new comment for version 3.4.

Although the command does not delete the old comment, it is no longer directly accessible by the user. The new comment contains the login name of the user who invoked the cdc command and the time the comment was changed.

5.6.3 Adding Modification Requests

You can add modification requests to an s-file, when the v flag is set, by using the **-m** option of the **delta** and **admin** commands. A modification request is a shorthand method of describing the reason for a particular version. Modification requests are usually names or numbers which the user has chosen to represent a specific request.

The **-m** option causes the given command to save the requests following the option. A request may be any combination of letters, digits, and punctuation symbols. If you give more than one request, you must separate them with spaces and enclose the request in double quotes. For example, the command

```
delta -m"error35 optimize10" s.demo.c
```

copies the requests "error35" and "optimize10" to *s.demo.c*, while saving the new version.

The **-m** option, when used with the **admin** command, must be combined with the **-i** option. Furthermore, the **v** flag must be explicitly set with the **-f** option. For example, the command

```
admin -idef.h -m"error0" -fv s.def.h
```

inserts the modification request "error0" in the new file *s.def.h*.

The **delta** command does not prompt for modification requests if you use the **-m** option.

5.6.4 Changing Modification Requests

You can change modification requests, when the **v** flag is set, by using the **cdc** command. The command asks for a list of modification requests by displaying the prompt

```
MRs?
```

You may type any number of requests. Each request may have any combination of letters, digits, or punctuation symbols. No more than 512 characters are allowed, and the last request must be terminated with a newline

character. If you wish to remove a request, you must precede the request with an exclamation mark (!). For example, the command

```
cdc -r1.4 s.demo.c
```

asks for changes to the modification requests. The response

```
MRs? error36 !error35
```

adds the request "error36" and removes "error35".

5.6.5 Adding Descriptive Text

You can add descriptive text to an s-file by using the -t option of the **admin** command. Descriptive text is any text that describes the purpose and reason for the given s-file. Descriptive text is independent of the contents of the s-file and can only be displayed using the **prs** command.

The -t option directs the **admin** to copy the contents of a given file into the description field of the s-file. The command has the form

```
admin -tfilename  s.filename
```

where -t*filename* gives the name of the file containing the descriptive text, and *s.filename* is the name of the s-file to receive the descriptive text. The file to be inserted may contain any amount of text. For example, the command

```
admin -tcdemo s.demo.c
```

inserts the contents of the file *cdemo* into the description field of the s-file *s.demo.c*.

The -t option may also be used to initialize the description field when creating the s-file. For example, the command

```
admin -idemo.c -tcdemo s.demo.c
```

inserts the contents of the file *cdemo* into the new s-file *s.demo.c*. If -t is not used, the description field of the new s-file is left empty.

You can remove the current descriptive text in an s-file by using the -t option without a filename. For example, the command

```
admin -t s.demo.c
```

removes the descriptive text from the s-file *s.demo.c*.

5.7 Printing from an S-file

This section explains how to use the **prs** command to display information contained in an s-file. The **prs** command has a variety of options which control the display format and content.

5.7.1 Using a Data Specification

You can explicitly define the information to be printed from an s-file by using the **-d** option of the **prs** command. The command copies user-specified information to the standard output. The command has the form

prs -d*spec* *s.filename*

where -d*spec* is the data specification, and *s.filename* is the name of the s-file from which the information is to be taken.

The data specification is a string of data keywords and text. A data keyword is an uppercase letter, enclosed in colons (:). It represents a value contained in the given s-file. For example, the keyword **:I:** represents the SID of a given version, **:F:** represent the filename of the given s-file, **:C:** represents the comment line associated with a given version. Data keywords are replaced by these values when the information is printed.

For example, the command

```
prs -d" version: :I:  filename: :F: " s.demo.c
```

may produce the line

```
version: 2.1  filename: s.demo.c
```

A complete list of the data keywords is given in the section *prs*(CP) in the XENIX *Reference Manual*.

5.7.2 Printing a Specific Version

You can print information about a specific version in a given s-file by using the **-r** option of the **prs** command. The command has the form

prs -r*SID* *s.filename*

where -r*SID* gives the SID of the desired version, and *s.filename* is the name of the s-file containing the version. For example, the command

```
prs -r2.1 s.demo.c
```

prints information about version 2.1 in the s-file *s.demo.c*.

If the **-r** option is not specified, the command prints information about the most recently created delta.

5.7.3 Printing Later and Earlier Versions

You can print information about a group of versions by using the **-l** and **-e** options of the **prs** command. The **-l** option causes the command to print information about all versions immediately succeeding the given version. The **-e** option causes the command to print information about all versions immediately preceding the given version. For example, the command

```
prs -r1.4 -e s.demo.c
```

prints all information about versions which precede version 1.4 (e.g., 1.3, 1.2, and 1.1). The command

```
prs -r1.4 -l s.abc
```

prints information about versions which succeed version 1.4 (e.g., 1.5, 1.6, and 2.1).

If both options are given, information about all versions is printed.

5.8 Editing by Several Users

The SCCS system allows any number users to access and edit versions of a given s-file. Since users are likely to access different versions of the s-file at the same time, the system is designed to allow concurrent editing of different versions. Normally, the system allows only one user at a time to edit a given version, but you can allow concurrent editing of the same version by setting the **j** flag in the given s-file.

The following sections explain how to perform concurrent editing and how to save edited versions when you have retrieved more than one version for editing.

5.8.1 Editing Different Versions

The SCCS system allows several different versions of a file to be edited at the same time. This means a user can edit version 2.1 while another user edit version 1.1. There is no limit to the number of versions which may be edited at any given time.

When several users edits different versions concurrently, each user must begin work in his own directory. If users attempt to share a directory and work on versions from the same s-file at the same time, the **get** command will refuse to

retrieve a version.

5.8.2 Editing a Single Version

You can let a single version of a file be edited by more than one user by setting the **j** flag in the given s-file. The flag causes the **get** command to check the p-file and create a new proposed SID if the given version is already being edited.

You can set the flag by using the **–f** option of the **admin** command. For example, the command

```
admin –fj s.demo.c
```

sets the flag for the s-file *s.demo.c*.

When the flag is set, the **get** command uses the next available branch SID for each new proposed SID. For example, suppose a user retrieves for editing version 1.4 in the file *s.demo.c*, and that the proposed version is 1.5. If another user retrieves version 1.4 for editing before the first user has saved his changes, the the proposed version for the new user will be 1.4.1.1, since version 1.5 is already proposed and likely to be taken. In no case will a version edited by two separate users result in a single new version.

5.8.3 Saving a Specific Version

When editing two or more versions of a file, you can direct the **delta** command to save a specific version by using the –r option to give the SID of that version. The command has the form

```
delta –rSID  s.filename
```

where –r*SID* gives the SID of the version being saved, and *s.filename* is the name of the s-file to receive the new version. The *SID* may be the SID of the version you have just edited, or the proposed SID for the new version. For example, if you have retrieved version 1.4 for editing (and no version 1.5 exists), both commands

```
delta –r1.5 s.demo.c
```

and

```
delta –r1.4 s.demo.c
```

save version 1.5.

5.9 Protecting S-files

The SCCS system uses the normal XENIX system file permissions to protect s-files from changes by unauthorized users. In addition to the XENIX system protections, the SCCS system provides two ways to protect the s-files: the "user list" and the "protection flags". The user list is a list of login names and group IDs of users who are allowed to access the s-file and create new versions of the file. The protection flags are three special s-file flags that define which versions are currently accessible to otherwise authorized users. The following sections explain how to set and use the user list and protection flags.

5.9.1 Adding a User to the User List

You can add a user or a group of users to the user list of a given s-file by using the **–a** option of the **admin** command. The option causes the given name to be added to the user list. The user list defines who may access and edit the versions in the s-file. The command has the form

admin –a*name* *s.filename*

where –a*name* gives the login name of the user or the group name of a group of users to be added to the list, and *s.filename* gives the name of the s-file to receive the new users. For example, the command

admin –ajohnd –asuex –amarketing s.demo.c

adds the users "johnd" and "suex" and the group "marketing" to the user list of the s-file *s.demo.c*.

If you create an s-file without giving the **–a** option, the user list is left empty, and all users may access and edit the files. When you explicitly give a user name or names, only those users can access the files.

5.9.2 Removing a User from a User List

You can remove a user or a group of users from the user list of a given s-file by using the –e option of the **admin** command. The option is similar to the **–a** option but performs the opposite operation. The command has the form

admin –e*name* *s.filename*

where –e*name* gives the login name of a user or the group name of a group of users to be removed from the list, and *s.filename* is the name of the s-file from which the names are to be removed. For example, the command

admin –ejohnd –emarketing s.demo.c

removes the user "johnd" and the group "marketing" from the user list of the s-file *s.demo.c*.

5.9.3 Setting the Floor Flag

The floor flag, **f**, defines the release number of the lowest version a user may edit in a given s-file. You can set the flag by using the **–f** option of the **admin** command. For example, the command

```
admin -ff2 s.demo.c
```

sets the floor to release number 2. If you attempt to retrieve any versions with a release number less than 2, an error will result.

5.9.4 Setting the Ceiling Flag

The ceiling flag, **c**, defines the release number of the highest version a user may edit in a given s-file. You can set the flag by using the **–f** option of the **admin** command. For example, the command

```
admin -fc5 s.demo.c
```

sets the ceiling to release number 5. If you attempt to retrieve any versions with a release number greater than 5, an error will result.

5.9.5 Locking a Version

The lock flag, **l**, lists by release number all versions in a given s-file which are locked against further editing. You can set the flag by using the **–f** flag of the **admin** command. The flag must be followed by one or more release numbers. Multiple release numbers must be separated by commas (,). For example, the command

```
admin -fl3 s.demo.c
```

locks all versions with release number 3 against further editing. The command

```
admin -fl4,5,9 s.def.h
```

locks all versions with release numbers 4, 5, and 9.

Note that the special symbol "a" may be used to specify all release numbers. The command

```
admin -fla  s.demo.c
```

locks all versions in the file *s.demo.c*.

5.10 Repairing SCCS Files

The SCCS system carefully maintains all SCCS files, making damage to the files very rare. However, damage can result from hardware malfunctions, which cause incorrect information to be copied to the file. The following sections explain how to check for damage to SCCS files, and how to repair the damage or regenerate the file.

5.10.1 Checking an S-file

You can check a file for damage by using the **–h** option of the **admin** command. This option causes the checksum of the given s-file to be computed and compared with the existing sum. An s-file's checksum is an internal value computed from the sum of all bytes in the file. If the new and existing checksums are not equal, the command displays the message

```
corrupted file (co6)
```

indicating damage to the file. For example, the command

```
admin -h s.demo.c
```

checks the s-file *s.demo.c* for damage by generating a new checksum for the file, and comparing the new sum with the existing sum.

You may give more than one filename. If you do, the command checks each file in turn. You may also give the name of a directory, in which case, the command checks all files in the directory.

Since failure to repair a damaged s-file can destroy the file's contents or make the file inaccessible, it is a good idea to regularly check all s-files for damage.

5.10.2 Editing an S-file

When an s-file is discovered to be damaged, it is a good idea to restore a backup copy of the file from a backup disk rather than attempting to repair the file. (Restoring a backup copy of a file is described in the XENIX *Operations Guide*.) If this is not possible, the file may be edited using a XENIX text editor.

To repair a damaged s-file, use the description of an s-file given in the section *sccsfile* (F) in the XENIX *Reference Manual*, to locate the part of the file which is damaged. Use extreme care when making changes; small errors can cause unwanted results.

5.10.3 Changing an S-file's Checksum

After repairing a damaged s-file, you must change the file's checksum by using the –z option of the **admin** command. For example, to restore the checksum of the repaired file *s.demo.c*, type

```
admin -z s.demo.c
```

The command computes and saves the new checksum, replacing the old sum.

5.10.4 Regenerating a G-file for Editing

You can create a g-file for editing without affecting the current contents of the p-file by using the –k option of the **get** command. The option has the same affect as the –e option, except that the current contents of the p-file remain unchanged. The option is typically used to regenerate a g-file that has been accidentally removed or destroyed before it has been saved using the **delta** command.

5.10.5 Restoring a Damaged P-file

The –g option of the **get** command may be used to generate a new copy of a p-file that has been accidentally removed. For example, the command

```
get -e -g s.demo.c
```

creates a new p-file entry for the most recent version in *s.demo.c*. If the file *demo.c* already exists, it will not be changed by this command.

5.11 Using Other Command Options

Many of the SCCS commands provide options that control their operation in useful ways. This section describes these options and explains how you may use them to perform useful work.

5.11.1 Getting Help With SCCS Commands

You can display helpful information about an SCCS command by giving the name of the command as an argument to the **help** command. The **help** command displays a short explanation of the command and command syntax. For example, the command

```
help rmdel
```

displays the message

```
rmdel:
        rmdel -rSID name ...
```

5.11.2 Creating a File With the Standard Input

You can direct **admin** to use the standard input as the source for a new s-file by using the –i option without a filename. For example, the command

```
admin -i s.demo.c <demo.c
```

causes **admin** to create a new s-file named *s.demo.c* which uses the text file *demo.c* as its first version.

This method of creating a new s-file is typically used to connect **admin** to a pipe. For example, the command

```
cat mod1.c mod2.c | admin -i s.mod.c
```

creates a new s-file *s.mod.c* which contains the first version of the concatenated files *mod1.c* and *mod2.c*.

5.11.3 Starting At a Specific Release

The **admin** command normally starts numbering versions with release number 1. You can direct the command to start with any given release number by using the –r option. The command has the form

```
admin -rrel-num s.filename
```

where *–rrel-num* gives the value of the starting release number, and *s.filename* is the name of the s-file to be created. For example, the command

```
admin -idemo.c -r3 s.demo.c
```

starts with release number 3. The first version is 3.1.

5.11.4 Adding a Comment to the First Version

You can add a comment to the first version of file by using the –y option of the **admin** command when creating the s-file. For example, the command

```
admin -idemo.c -y"George Wheeler" s.demo.c
```

inserts the comment "George Wheeler" in the new s-file *s.demo.c*.

The comment may be any combination of letters, digits, and punctuation symbols. If spaces are used, the comment must be enclosed in double quotes. The complete command must fit on one line.

If the –y option is not used when creating an s-file, a comment of the form

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically inserted.

5.11.5 Suppressing Normal Output

You can suppress the normal display of messages created by the **get** command by using the –s option. The option prevents information, such as the SID of the retrieved file, from being copied to the standard output. The option does not suppress error messages.

The –s option is often used with the –p option to pipe the output of the **get** command to other commands. For example, the command

```
get -p -s s.demo.c | lpr
```

copies the most recent version in the s-file *s.demo.c* to the line printer.

You can also suppress the normal output of the **delta** command by using the –s option. This option suppresses all output normally directed to the standard output, except for the normal comment prompt.

5.11.6 Including and Excluding Deltas

You can explicitly define which deltas you wish to include and which you wish to exclude when creating a g-file, by using the –i and –x options of the **get** command.

The –i option causes the command to apply the given deltas when constructing a version. The –x option causes the command to ignore the given deltas when constructing a version. Both options must be followed by one or more SIDs. If multiple SIDs are given they must be separated by commas (,). A range of SIDs may be given by separating two SIDs with a hyphen (–). For example, the command

```
get -i1.2,1.3 s.demo.c
```

causes deltas 1.2 and 1.3 to be used to construct the g-file. The command

```
get -x1.2-1.4 s.demo.c
```

causes deltas 1.2 through 1.4 to be ignored when constructing the file.

The –i option is useful if you wish to automatically apply changes to a version while retrieving it for editing. For example, the command

```
get –e –i4.1 –r3.3 s.demo.c
```

retrieves version 3.3 for editing. When the file is retrieved, the changes in delta 4.1 are automatically applied to it, making the g-file the same as if version 3.3 had been edited by hand using the changes in delta 4.1. These changes can be saved immediately by issuing a **delta** command. No editing is required.

The –x option is useful if you wish to remove changes performed on a given version. For example, the command

```
get –e –x1.5 –r1.6 s.demo.c
```

retrieves version 1.6 for editing. When the file is retrieved, the changes in delta 1.5 are automatically left out of it, making the g-file the same as if version 1.4 had been changed according to delta 1.6 (with no intervening delta 1.5). These changes can be saved immediately by issuing a **delta** command. No editing is required.

When deltas are included or excluded using the –i and –x options, **get** compares them with the deltas that are normally used in constructing the given version. If two deltas attempt to change the same line of the retrieved file, the command displays a warning message. The message shows the range of lines in which the problem may exist. Corrective action, if required, is the responsibility of the user.

5.11.7 Listing the Deltas of a Version

You can create a table showing the deltas required to create a given version by using the –l option. This option causes the **get** command to create an l-file which contains the SIDs of all deltas used to create the given version.

The option is typically used to create a history of a given version's development. For example, the command

```
get –l s.demo.c
```

creates a file named *l.demo.c* containing the deltas required to create the most recent version of *demo.c*.

You can display the list of deltas required to create a version by using the **–lp** option. The option performs the same function as the –I options except it copies the list to the standard output file. For example, the command

```
get –lp –r2.3 s.demo.c
```

copies the list of deltas required to create version 2.3 of *demo.c* to the standard

output.

Note that the –l option may be combined with the –g option to create a list of deltas without retrieving the actual version.

5.11.8 Mapping Lines to Deltas

You can map each line in a given version to its corresponding delta by using the –m option of the get command. This option causes each line in a g-file to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the beginning of the line by a tab character. The –m option is typically used to review the history of each line in a given version.

5.11.9 Naming Lines

You can name each line in a given version with the current module name (i.e., the value of the %M% keyword) by using the –n option of the get command. This option causes each line of the retrieved file to be preceded by the value of the %M% keyword and a tab character.

The –n option is typically used to indicate that a given line is from the given file. When both the –m and –n options are specified, each line begins with the %M% keyword.

5.11.10 Displaying a List of Differences

You can display a detailed list of the differences between a new version of a file and the previous version by using the –p option of the delta command. This option causes the command to display the differences, in a format similar to the output of the XENIX diff command.

5.11.11 Displaying File Information

You can display information about a given version by using the –g option of the get command. This option suppresses the actual retrieval of a version and causes only the information about the version, such as the SID and size, to be displayed.

The –g option is often used with the –r option to check for the existence of a given version. For example, the command

```
get -g -r4.3 s.demo.c
```

displays information about version 4.3 in the s-file *s.demo.c*. If the version does not exist, the command displays an error message.

5.11.12 Removing a Delta

You can remove a delta from an s-file by using the **rmdel** command. The command has the form

rmdel -r*SID* *s.filename*

where -r*SID* gives the SID of the delta to be removed, and *s.filename* is the name of the s-file from which the delta is to be removed. The delta must be the most recently created delta in the s-file. Furthermore, the user must have write permission in the directory containing the s-file, and must either own the s-file or be the user who created the delta.

For example, the command

rmdel -r2.3 s.demo.c

removes delta 2.3 from the s-file *s.demo.c*.

The **rmdel** command will refuse to remove a protected delta, that is, a delta whose release number is below the current floor value, above the current ceiling value, or equal to a current locked value (see the section "Protecting S-files" given earlier in this chapter). The command will also refuse to remove a delta which is currently being edited.

The **rmdel** command should be reserved for those cases in which incorrect, global changes were made to ans-file.

Note that **rmdel** changes the type indicator of the given delta from "D" to "R". A type indicator defines the type of delta. Type indicators are described in full in the section *delta*(CP) in the XENIX *Reference Manual*.

5.11.13 Searching for Strings

You can search for strings in files created from an s-file by using the **what** command. This command searches for the symbol #(@) (the current value of the %Z% key word) in the given file. It then prints, on the standard output, all text immediately following the symbol, up to the next double quote ("), greater than (>), backslash (\), newline, or (non-printing) NULL character. For example, if the s-file *s.demo.c* contains the following line

char id[] = "%Z%%M%:%I%";

and the command

get -r3.4 s.prog.c

is executed, then the command

```
what prog.c
```

displays

```
prog.c:
        prog.c:3.4
```

You may also use **what** to search files that have not been created by SCCS commands.

5.11.14 Comparing SCCS Files

You can compare two versions from a given s-file by using the **sccsdiff** command. This command prints on the standard output the differences between two versions of the s-file. The command has the form

```
sccsdiff -rSID1 -rSID2 s.filename
```

where *-rSID1* and *-rSID2* give the SIDs of the versions to be compared, and *s.filename* is the name of the s-file containing the versions. The version SIDs must be given in the order in which they were created. For example, the command

```
sccsdiff -r3.4 -r5.6 s.demo.c
```

displays the differences between versions 3.4 and 5.6. The differences are displayed in a form similar to the XENIX **diff** command.

Chapter 6
Adb: A Program Debugger

6.1 Introduction

Adb is an indispensable tool for debugging programs or crashed systems. It allows you to look at *core* files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This chapter is an introduction to *adb* with examples of its use. It explains the various formatting options, techniques for debugging C programs, and gives examples of printing file system information, and of patching.

6.2 Invocation

The *adb* invocation syntax is as follows:

adb *objectfile corefile*

where *objectfile* is an executable XENIX file and *corefile* is a core image file. Often this will look like:

adb a.out core

or more simply:

adb

where the defaults are *a.out* and *core*, respectively. The filename minus (−) means ignore this argument as in:

adb − core

Adb has requests for examining locations in either file. A question mark (?) request examines the contents of *objectfile*; a slash (/) request examines the *corefile*. The general form of these requests is:

address ? *format*

or

address / *format*

6.3 The Current Address − Dot

Adb maintains a pointer to the current address, called dot, similar in function to the current pointer in the editor, **ed**(C). When an address is entered, the current address is set to that location, so that:

0126?i

sets dot to octal 126 and prints the instruction at that address. The request

.,10/d

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the question mark (?) or slash (/) request, the current address can be advanced by typing a newline; it can be decremented by typing a caret (^).

Addresses are represented by expressions. Expressions are made up of decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be

combined with the following operators:

+	Addition
−	Subtraction
*	Multiplication
%	Integer division
&	Bitwise AND
|	Bitwise inclusive OR
#	Round up to the next multiple
~	Not

Note that all arithmetic within *adb* is 32−bit arithmetic. When typing a symbolic address for a C program, type either "name" or "_name"; *adb* recognizes both forms. Because *adb* will find only one instance of "name" and "_name" (generally the first to appear in the source) one will mask the other if they both appear in the same source file.

6.4 Formats

To print data, you can specify a collection of letters and characters that describe the format of the printout. Formats are remembered in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters; for a complete list see **adb(CP)**

Letter	Format
b	1 byte in octal
c	1 byte as a character
o	1 word in octal
d	1 word in decimal
x	1 word in hexadecimal
D	2 words (1 longword) in decimal
X	2 words (1 longword) in hexadecimal
i	machine instruction
s	a null terminated character string
a	the value of dot
u	1 word in unsigned decimal
n	print a newline
r	print a blank space
^	backup dot

equest is:

address [,*count*] *command* [*modifier*]

which sets the current address (dot) to *address* and executes the command *count* times.

The following table illustrates some general *adb* command meanings:

Command	Meaning
?	Print contents from *a.out* file
/	Print contents from *core* file
=	Print value of "dot"
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

Adb catches signals, so a user cannot use a quit signal to exit from *adb*. The request $q or $Q (or <CONTROL-D>) must be used to exit from *adb*.

6.5 Debugging C Programs

The following subsections describe use of *adb* in debugging the C programs given in the numbered figures at the end of this chapter. Refer to these figures as you work your way through the examples.

6.5.1 Debugging a Core Image

Consider the C program in Figure 1. This program illustrates a common error made by C programmers. The object of the program is to change the lowercase "t" to uppercase "T" in the string pointed to by "charp" and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer "charp" instead of the string pointed to by "charp." Executing the program produces a core file because of an out-of-bounds memory reference. (Note that a core file may not be produced on all systems.)

Adb is invoked by typing:

```
adb a.out core
```

The first debugging request

```
$c
```

is used to give a C backtrace through the subroutines called. As shown in Figure 2, only one function, *main*, was called and the arguments "argc" and "argv" have hex values 0x2 and 0x1fff90 respectively. Both of these values look reasonable; 0x2 = two arguments, 0x1fff90 = address on stack of parameter vector. These values may be different on your system due to a different mapping of memory.

The next request

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

```
$e
```

prints out the values of all external variables.

A map exists for each file handled by *adb*. The map for the *a.out* file is referenced with a question mark (?), whereas the map for the *core* file is referenced with a slash (/). Furthermore, a good rule of thumb is to use question mark for instructions and slash for data when looking at programs. To print out information about the maps, type:

```
$m
```

This produces a report of the contents of the maps.

In our example, it is useful to see the contents of the string pointed to by "charp." This is done by typing

```
*charp/s
```

which means use "charp" as a pointer in the *core* file and print the information as a character string. This printout shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around "charp" shows that the buffer is unchanged but that the pointer is destroyed. Similarly, we could print information about the arguments to a function. For example

```
0x1fff90,3/X
```

prints the hex values of the three consecutive cells pointed to by "argv" in the function *main*. Note that these values are the addresses of the arguments to main. Therefore:

```
0x1fffb6/s
```

prints the ASCII value of the first argument. Another way to print this value would have been

```
*"/s
```

The quotation mark (") means ditto, i.e., the last address typed, in this case "0x1fff90 ;" the star (*) instructs *adb* to use the address field of the *core* file as a pointer.

The request

```
.=x
```

prints the current address in hex (and not its contents). This has been set to the address of the first argument. The current address, dot, is used by *adb* to remember its current location. Dot allows the user to reference locations relative to the current address, for example:

```
.-10/d
```

6.5.2 Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f*, *g*, and *h* until the stack is exhausted and a core image is produced.

Again, enter *adb* by typing

```
adb
```

which assumes the names *a.out* and *core* for the executable file and core image file, respectively. The request

```
$c
```

fills a page of backtrace references to *f*, *g*, and *h*. Figure 4 shows an abbreviated list. Pressing the INTERRUPT key terminates the output and brings you back to the *adb* request level. Additionally, some versions of *adb* will automatically quit after fifteen levels unless told otherwise with the command:

```
,levelcount$c
```

The request

```
,5$c
```

prints the five most recent activations.

Notice that each function (*f*, *g*, and *h*) has a counter that counts the number of times each has been called.

The request

```
fcnt/D
```

prints the decimal value of the counter for the function *f*. Similarly, "gcnt" and "hcnt" could be printed. Notice that because "fcnt", "gcnt", and "hcnt" are **int** variables, and on the MC68000 **int** is implemented as **long**, to print its value you must use the D two-word format.

6.5.3 Setting Breakpoints

Consider the C program in Figure 5. This program changes tabs into blanks. We will run this program under the control of *adb* (see Figure 6) by typing:

```
adb a.out -
```

Breakpoints are set in the program as:

address:b [*request*]

The requests

```
settab+8:b
fopen+8:b
tabpos+8:b
```

set breakpoints at the start of these functions. C does not generate statement labels. Therefore, it is currently not possible to plant breakpoints at locations other than function entry points without knowledge of the code generated by the C compiler. The above addresses are entered as

```
symbol+8
```

so that they will appear in any C backtrace, because the first two instructions of each function are used to set up the local stack frame. Note that some of the functions are from the C library.

To print the location of breakpoints, type:

```
$b
```

The display indicates a *count* field. A breakpoint is bypassed *count*−1 times before causing a stop. The *command* field indicates the *adb* requests to be executed each time

the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the **tstb** instruction, which is the stack probe. We can display the instructions using the *adb* request:

```
settab,5?ai
```

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is

```
settab,5?i
```

which displays the instructions with only the starting address.

Note that we accessed the addresses from the *a.out* file with the question (?) command. In general, when asking for a printout of multiple items *adb* advances the current address the number of bytes necessary to satisfy the request. In the above example, five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program type:

```
:r
```

To delete a breakpoint, for instance the entry to the function *settab*, type:

```
settab+8:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for *fopen*), *adb* requests can be used to display the contents of memory. For example

```
$c
```

displays a stack trace or

```
tabs,6/4X
```

prints six lines of four locations each from the array called "tabs". By this time (at location *fopen*) in the C program, *settab* has been called and should have set a one in every eighth location of "tabs".

The XENIX quit and interrupt signals act on *adb* itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to *adb*. The signal is saved by *adb* and is passed on to the test program if

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if

```
:c 0
```

is typed.

6.5.4 Other Breakpoint Facilities

Arguments and changes of standard input and output are passed to a program as:

:r *arg1 arg2 ...<infile >outfile*

This request kills any existing program under test and starts the *a.out* afresh.

The program being debugged can be single-stepped by typing:

:s

If necessary, this request starts up the program being debugged and stops after executing the first instruction.

Adb allows a program to be executed beginning at a specific address by typing:

address:r

The count field can be used to skip the first *n* breakpoints with:

,*n*:r

The request

,*n*:c

may also be used for skipping the first *n* breakpoints when continuing a program.

A program can be continued at an address different from the breakpoint by typing:

address:c

The program being debugged runs as a separate process and can be killed by typing:

:k

6.6 Maps

XENIX supports several executable file formats. These are used to tell the loader how to load the program file. Nonshared program files are the most common and are generated by a C compiler invocation such as:

cc pgm.c

A shared file is produced by a C compiler command line of the form

cc -n pgm.c

Note that separate instruction/data files are not supported on the MC68000.

Adb interprets these different file formats and provides access to the different segments through a set of maps. To print the maps type:

$m

In nonshared files, both text (instructions) and data are intermixed. This makes it impossible for *adb* to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In shared text, the instructions are separated from data and the

```
?*
```

accesses the data part of the *a.out* file. This request tells *adb* to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution. In shared files the corresponding *core* file does not contain the program text.

Figure 7 shows the display of three maps for the same program linked as a nonshared and shared respectively. The b, e, and f fields are used by *adb* to map addresses into file addresses. The *f1* field is the length of the header at the beginning of the file (0x34 bytes for an *a.out* file and 0x800 bytes for a *core* file). The *f2* field is the displacement from the beginning of the file to the data. For unshared files with mixed text and data this is the same as the length of the header; for shared files this is the length of the header plus the size of the text portion.

The *b* and *e* fields are the starting and ending locations for a segment. Given an address, *A*, the location in the file (either *a.out* or *core*) is calculated as:

```
b1≤A≤e1 ⇒ file address = (A−b1)+f1
b2≤A≤e2 ⇒ file address = (A−b2)+f2
```

A user can access locations by using the *adb* defined variables. The

```
$v
```

request prints the variables initialized by *adb*:

b	Base address of data segment
d	Length of the data segment
s	Length of the stack
t	Length of the text
m	Execution type

In Figure 7 those variables not present are zero. These variables can be used in expressions such as

```
<b
```

in the address field. Similarly, the value of the variable can be changed by an assignment request such as

```
02000>b
```

which sets "b" to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

Adb reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, or if it is missing, then the header of the executable file is used instead.

6.7 Advanced Usage

With *adb* it is possible to combine formatting requests to provide elaborate displays. Below are several examples.

6.7.1 Formatted Dump

The line

```
<b,-1/4o4^8Cn
```

prints four octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the request pieces mean:

- `<b` The base address of the data segment.
- `<b,-1` Print from the base address to the end-of-file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end-of-file) is detected.

The format "4o4^8Cn" is interpreted as follows:

- `4o` Print four octal locations.
- `4^` Backup the current address four locations (to the original start of the field).
- `8C` Print eight consecutive characters using an escape convention; each character in the range octal 0 to 037 is printed as an at-sign (@) followed by the corresponding character in the range octal 0140 to 0177. An at-sign is printed as "@@".
- `n` Print a newline.

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow printing to stop at the end of the data segment (`<d` provides the data segment size in bytes).

The formatting requests can be combined with *adb*'s ability to read in a script to produce a *core* image dump script. *Adb* is invoked with the command line

```
adb a.out core < dump
```

to read in a script file containing requests named *dump*. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request

```
120$w
```

sets the width of the output to 120 characters (normally, the width is 80 characters). *Adb* attempts to print addresses as:

```
symbol + offset
```

The request

```
4095$s
```

increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The equal sign request (=) can be used to print literal strings. Thus, headings are provided in this *dump* program with requests such as:

```
=3n"C Stack Backtrace"
```

This spaces three lines and prints the literal string. The request

```
$v
```

prints all nonzero *adb* variables. The request

```
0$s
```

sets the maximum offset for symbol matches to zero, thus suppressing the printing of symbolic labels in favor of hexadecimal values. Note that this is only done for the printing of the data segment. The request

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end-of-file with an octal address field and eight octal numbers per line.

Figure 9 shows the results of some formatting requests on the C program of Figure 8.

6.7.2 Directory Dump

Figure 10 illustrates another set of requests to dump the contents of a directory (which is made up of an integer "inumber" followed by a 14-character name):

```
adb dir -
=n8t"inum"8t"Name"
0,-1? u8t14cn
```

In this example, "u" prints the inumber as an unsigned decimal integer, "8t" means that *adb* will space to the next multiple of 8 on the output line, and "14c" prints the 14-character filename.

6.7.3 Ilist Dump

Similarly the contents of the *ilist* of a file system (e.g., */dev/root*) can be dumped with the following set of requests:

```
adb /dev/root -
02000>b
?m <b
<b,-1?"flags"8ton"links,uid,gid"8t3bn",size"8tbrdn"addr"8t8un"times"8t2Y2na
```

In this example the value of the base for the map was changed to 02000 by typing

```
?m<b
```

since that is the start of an *ilist* within a file system. The request "brd" above was used to print the 24-bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the "2Y" operator. Figure 10 shows portions of these requests as applied to a directory and file system.

6.7.4 Converting Values

Adb may be used to convert values from one representation to another. For example

```
072 = odx
```

prints

```
072     58      0x3a
```

which are the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers prints them in the given formats. Character values can be converted in a similar way; for example

```
'a' = co
```

prints

```
a       0141
```

It may also be used to evaluate expressions. However, be forewarned that all binary operators have the same precedence, a precedence that is lower than that for unary operators.

6.8 Patching

Patching files with *adb* is accomplished with the write (**w** or **W**) request. This is often used in conjunction with the locate (**l** or **L**) request. The request syntax for **l** and **w** are similar:

```
?l value
```

The request **l** is used to match on 2 bytes; **L** is used for 4 bytes. The request **w** is used to write 2 bytes, whereas **W** writes 4 bytes. The *value* field in either locate or write requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, *adb* must be called with the **-w** switch:

```
adb -w file1 file2
```

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 8. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

```
adb -w ex7 -
?l 'Th'
?W 'The'
```

The request

```
?l
```

starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of the question mark (?) to write to the *x.out* file. The form

```
?*
```

would have been used for a shared file.

More frequently the request is typed as:

```
?l 'Th'; ?s
```

This locates the first occurrence of "Th" and prints the entire string. Execution of this request sets dot to the address of the characters "Th".

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through *adb* and the program run. For example:

```
adb x.out -
:s arg1 arg2
flag/w 1
:c
```

The **:s** request is normally used to single-step through a process or start a process in single-step mode. In this case it starts *x.out* as a subprocess with arguments "arg1" and "arg2". If there is a subprocess running, *adb* writes to it rather than to the file so the **w** request causes "flag2" to be changed in the memory of the subprocess.

6.9 Notes

Below is a list of some things that users should be aware of:

The stack frame i s allocated by teh first two instructions at the beginning of every C routine. Thus, putting breakpoints at the entry point of routines means that the function appears not to have been called when the breakpoint

occurs. Try placing the breakpoint at "routine"+ instead.

1. When printing addresses, ADB uses ither text or data symbols from the *x.out* file. This sometimes causes unexpected symbol names to be printed with data (e.g., "savr5+022"). This does not happen if question mark (?) is used fo text (instructions) and slash (/) for data.

2. Local variables cannot be addressed.

6.10 Figures

Figure 1: C program with pointer bug

```
#include <stdio.h>
struct buf {
        int fildes;
        int nleft;
        char *nextp;
        char buff[512];
        }bb;
struct buf *obuf;

char *charp = "this is a sentence.";

main(argc,argv)
int a gc;
char **argv;
{
        char    cc;
        FILE *file;

        if(argc < 2) {
                printf("Input file missing\n");
                exit(8);
        }

        if((file = fopen(a gv[1],"w")) == NULL){
                printf("%s : can't open\n", argv[1]);
                exit(8);
        }
        charp = 'T';
printf("debug 1 %s\n",charp);
        while(cc=  *charp++)
                putc(cc,file);
        fflush(file);
}
```

Figure2: Adboutputfor C programoffigure1

```
adb
$c
start+44: _main     (0x2, 0x1FFF90)
$r
d0      0x0                         a0      0x54
d1      0x8                         a1      0x1FFF90
d2      0x0                         a2      0x0
d3      0x0                         a3      0x0
d4      0x0                         a4      0x0
d5      0x0                         a5      0x0
d6      0x0                         a6      0x1FFF7C
d7      0x0                         sp      0x1FFF74

ps      0x0
pc      0x80E4  _main+160:          movb    (a0),-1.(a6)
$e
_environ: 0x1FFF9C
_errno:   0x19
_bb:      0x0
_obuf:    0x0
_charp:   0x55
_iob:     0x9B1C
_sobuf:   0x64656275
_lastbu:  0x96F8
_sibuf:   0x0
_allocs:  0x0
_allocp:  0x0
_alloct:  0x0
_allocx:  0x0
_end:     0x0
_edata:   0x0
$m
? map    'x.out'
b1 = 0x8000      e1 = 0x970C      f1 = 0x20
b2 = 0x8000      e2 = 0x970C      f2 = 0x20
/ map    '-'
b1 = 0x0e1 = 0x1000000   f1 = 0x0
b2 = 0x0e2 = 0x0f2 = 0x0
*charp/s
0x55:
data address not found
0x1fff90,3/X
0x1FFF90:       0x1FFFB0        0x1FFFB6.       0x0
0x1fffb0/s
0x1FFFB0:       x.out
/s
0x1FFFB0:       x.out
.=X
                0x1FFFB0
.-10/d
0x1FFFA6:       65497
```

```
$q
```

Figure 3: Multiple function C program

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q-p;
        gr = q-p+1;
        gcnt++;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

Figure4: Adboutputfor C programofFigure3

```
adb
$c
_h+46:          _f      (0x2, 0x92D)
_g+48:          _h      (0x92C,  0x92B)
_f+70:          _g      (0x92D,  0x1258)
_h+46:          _f      (0x2, 0x92B)
_g+48:          _h      (0x92A,  0x929)
_f+70:          _g      (0x92B,  0x1254)
_h+46:          _f      (0x2, 0x929)
_g+48:          _h      (0x928,  0x927)
<INTERRUPT>
adb
,5$c
_h+46:          _f      (0x2, 0x92D)
_g+48:          _h      (0x92C,  0x92B)
_f+70:          _g      (0x92D,  0x1258)
_h+46:          _f      (0x2, 0x92B)
_g+48:          _h      (0x92A,  0x929)
fcnt/D
_fcnt:          1175
gcnt/D
_gcnt:          1174
hcnt/D
_hcnt:          1174
$q
```

Figure 5: C program to decode tabs

```
#include <stdio.h>
#define MAXLINE         80
#define YES             1
#define NO              0
#define TABSP           8
char    input[] = "data";
char    ibuf[518];
int     tabs[MAXLINE];

main()
{
        int col, *ptab;
        char c;

        ptab = tabs;
        settab(ptab);       /*Set initial tab stops */
        col = 1;
        if(fopen(input,ibuf) < 0) {
                printf("%s : not found\n",input);
                exit(8);
        }
        while((c = getch(ibuf)) != -1) {
                switch(c) {
                        case '\t': /* TAB */
                                while(tabpos(col) != YES) {
                                        /* put BLANK */
                                        putchar(' ');
                                        col++ ;
                                }
                                break;
                        case '\n': /*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++ ;
                }
        }
}
/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}
```

```
/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}

/* getch(ibuf) - Just do a getc call, but not a macro */
getch(ibuf)
FILE *ibuf;
{
        return(getc(ibuf));
}
```

Figure 6: Adb output for C program of Figure 5

```
adb x.out
settab+8:b
fopen+8:b
getch+8:b
tabpos+8:b
$b
breakpoints
count    bkpt              command
1        _tabpos+8
1        _getch+8
1        _fopen+8
1        _settab+8
settab,5?ia
_settab:        link    a6,#0xFFFFFFFC
_settab+4:              tstb    -132.(a7)
_settab+8:              moveml  #<>,-(a7)
_settab+12:             clrl    -4.(a6)
_settab+16:             cmpl    #0x50,-4.(a6)
_settab+24:
settab,5?i
_settab:        link    a6,#0xFFFFFFFC
                        tstb    -132.(a7)
                        moveml  #<>,-(a7)
                        clrl    -4.(a6)
                        cmpl    #0x50,-4.(a6)
:r
x.out:running
breakpoint      _settab+8:      moveml  #<>,-(a7)
settab+8:d
:c
x.out:running
breakpoint      _fopen+8:       jsr     _findio
$c
_main+52:       _fopen  (0x9750, 0x9958)
start+44: _main (0x1, 0x1FFF98)
tabs,6/4X
_tabs:  0x1     0x0     0x0     0x0
        0x0     0x0     0x0     0x0
        0x1     0x0     0x0     0x0
        0x0     0x0     0x0     0x0
        0x1     0x0     0x0     0x0
        0x0     0x0     0x0     0x0
```

Figore7: Adboutputformaps

```
adb x.out.unshared core.unshared
$m
? map    'x.out.unshared'
b1 = 0x8000      e1 = 0x83E4      f1 = 0x34
b2 = 0x8000      e2 = 0x83E4      f2 = 0x34
/ map    'core.unshared'
b1 = 0x8000      e1 = 0x8800      f1 = 0x800
b2 = 0x1EB000    e2 = 0x200000    f2 = 0x1000
$v
variables
b = 0x8000
d = 0x800
e = 0x8000
m = 0x107
s = 0x15000
$q


adb x.out.shared core.shared
$m
? map    'x.out.shared'
b1 = 0x8000      e1 = 0x8390      f1 = 0x34
b2 = 0x10000     e2 = 0x10054     f2 = 0x3B0
/ map    'core.shared'
b1 = 0x10000     e1 = 0x10108     f1 = 0x800
b2 = 0x1EB000    e2 = 0x200000    f2 = 0x1000
$v
variables
b = 0x10390
d = 0x800
e = 0x8000
m = 0x108
s = 0x15000
$q
```

Figure 8: Simple C program illustrating formatting and patching

```
char    str1[]  = "This is a character string";
int     one     = 1;
int     number  = 456;
long    lnum    = 1234;
float   fpt     = 1.25;
char    str2[]  = "This is the second character string";
main()
{
        one = 2;
}
```

Figure 9: Adb output illustrating fancy formats

```
adb x.out.shared core.shared
<b,-1/8ona
_str1:      052150 064563 020151 071440 060440 061550 060562 060543

_str1+16:   072145 071040 071564 071151 067147 0      0      01

_number:
_number:    0      0710   0      02322  037640 0      052150 064563

_str2+4:    020151 071440 072150 062440 071545 061557 067144 020143

_str2+20:   064141 071141 061564 062562 020163 072162 064556 063400

$nd:
$nd:        01     0140
<b,20/4o4^8Cn
_str1:      052150 064563 020151 071440  This is
            060440 061550 060562 060543  a charac
            072145 071040 071564 071151  ter stri
            067147 0      0      01      ng@'@'@'@'@'@a

_number:    0      0710   0      02322   @'@'@aH@'@'@dR

_fpt:       037640 0      052150 064563  ? @'@'This
            020151 071440 072150 062440  is the
            071545 061557 067144 020143  second c
            064141 071141 061564 062562  haracter
            020163 072162 064556 063400   string@'

$nd:        01     0140
data address not found
<b,20/4o4^8t8Cna
_str1:      052150 064563 020151 071440          This is
_str1+8:    060440 061550 060562 060543            a charac
_str1+16:   072145 071040 071564 071151            ter stri
_str1+24:   067147 0      0      01               ng@'@'@'@'@'@a
_number:
_number:    0      0710   0      02322             @'@'@aH@'@'@dR
_fpt:
_fpt:       037640 0      052150 064563           ? @'@'This
_str2+4:    020151 071440 072150 062440             is the
_str2+12:   071545 061557 067144 020143             second c
_str2+20:   064141 071141 061564 062562             haracter
_str2+28:   020163 072162 064556 063400              string@'
$nd:
$nd:        01     0140
data address not found
<b,10/2b8t^2cn
_str1:      0124   0150          Th
            0151   0163          is
            040    0151          i
```

```
        0163  040     s
        0141  040     a
        0143  0150    ch
        0141  0162    ar
        0141  0143    ac
        0164  0145    te
        0162  040     r
$q
```

Figure 10: Directory and inode dumps

```
adb dir -
=nt"Inode"t"Name"; 0,-17ut14cn

          Inode  Name
0x0:       652    .
           82     ..
           5971   cap.c
           5323   cap
           0      pp



adb /dev/root -
/dev/root - not in a.out format
02000>b
?m<b
$v
variables
b = 0x400
<b,-1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
0x400:  flags 073145
        links,uid,gid 0163  0164    0141
        size 0162    10356
        addr 28770   8236  25956 27766 25455 8236  25956 25206
        times 1976 Feb 5 08:34:56     1975 Dec 28 10:55:15

0x420:  flags 024555
        links,uid,gid 012   0163  0164
        size 0162    25461
        addr 8308    30050 8294  25130 15216 26890 29806 10784
        times 1976 Aug 17 12:16:51    1976 Aug 17 12:16:51

0x440:  flags 05173
        links,uid,gid 011   0162  0145
        size 0147    29545
        addr 25972   8306  28265 8308  25642 15216 2314  25970
        times 1977 Apr 2 08:58:01     1977 Feb 5 10:21:44
```

6.11 Adb Summary

6.11.1 Command Summary

Formatted printing

?*format*	print from *x.out* file according to *format*
/*format*	print from *core* file according to *format*
=*format*	print the value of *dot*
?wexpr	write expression into *x.out* file
/wexpr	write expression into *core* file
?lexpr	locate expression in *x.out* file

Breakpoint and program control

:b	set breakpoint at *dot*
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run *x.out* file under adb control
:s	single step

Miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$m	print adb segment maps
$q	exit from adb
$r	general registers
$s	set offset for symbol match
$v	print adb variables
$w	set output line width

Calling the shell

!	call *sh* (shell) to read rest of line

Assignment to variables

>*name*	assign dot to variable or register *name*

6.11.2 Incomplete Format Summary

a	the value of dot
b	1 byte in octal
c	1 byte as a character
d	1 word in decimal
i	machine instruction
o	1 word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
***n*t**	move to next *n* space tab
u	1 word as unsigned integer
x	1 word in hexadecimal
X	2 words (1 longword) in hexadecimal
D	2 words (1 longword) in decimal
Y	date
^	backup dot
"..."	print string

6.11.3 Expression Summary

Expression components

decimal integer	e.g., 256
octal integer	e.g., 0277
hexadecimal	e.g., 0xff
symbols	e.g., flag _main main.argc
variables	e.g., <b
registers	e.g., <pc <d0 <a0
(expression)	expression grouping

Dyadic operators

+	add
-	subtract
*	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

Monadic operators

~	not
*	contents of location
-	integer negation

hapter 7
s: An Assembler

7.1 Introduction

This chapter describes the use of the XENIX assembler, named *as*, for the Motorola MC68000 microprocessor. It is beyond the scope of this chapter to describe the instruction set of the MC68000 or to discuss assembly language programming in general. For information on these topics, refer to the "MC68000 16-Bit Microprocessor User's Manual", 3rd Edition, Englewood Cliffs: Prentice-Hall, 1982.

This chapter describes the following:

- Command Usage
- Source Program Format
- Symbols and Expressions
- Instructions and Addressing Modes
- Assembler Directives
- Operation Codes
- Error Messages

7.2 Command Usage

As can be invoked with one or more arguments. Except for option arguments, which must appear first on the command line, arguments may appear in any order on the command line. The source filename argument is traditionally named with an ".s" extension. Except as specified below, flags may be grouped. For example

```
as -glo that.o this.s
```

will have the same effect as

```
as -g -l -o that.o this.s
```

7.3 Invocation Options

The various options and their functions are described below:

-o *relname* The default output name is *filename*.o. This can be overridden by giving *as* the **-o** flag and giving the new filename in the argument following the **-o**. For example

```
as -o that.o this.s
```

assembles the source *this.s* and puts the output in the file *that.o*.

-l By default, no output listing is produced. A listing may be produced by giving the **-l** flag. The listing filename extension is ".L". The filename for the list file is based on the output file. So the command line

as −l −o output.x input.s

produces a listing named *output.L*.

−e By default, all symbols go into the symbol table of the *a.out*(F) file that is produced by the assembler, including locals. If you want only symbols that are defined as **.globl** or **.comm** to be included, use the **−e** (externals only) flag.

−g By default, if a symbol is undefined in an assembly, an error is flagged. This may be changed with the **−g** flag. If this is done, undefined symbols will be interpreted as external.

:−v **By default, the a.out file is for XENIX version 3.0 systems; the number 2 or 3 specifies which version the output is intended for.**

7.4 Source Program Format

An *as* program consists of a series of statements, each of which occupies exactly one line, i.e., a sequence of characters followed by the newline character. Form feed, ASCII <CONTROL−L>, also serves as a line terminator. Continuation lines are not allowed, and the maximum line length is 132 characters. However, several statements may be on a single line, separated by semicolons. Remember though, that anything after a comment character is considered a comment. The format of an *as* assembly language statement is:

[*label−field*] [*opcode* [*operands*] [| *comment*]

Most of the fields may be omitted under certain circumstances. In particular:

1. Blank lines are permitted.

2. A statement may contain only a label field. The label defined in this field has the same value as if it were defined in the label field of the next statement in the program. As an example, the two statements

```
name:
addl    d0,d1
```

are equivalent to the single statement

```
name:   addl    d0,d1
```

3. A line may consist of only the comment field. The two statements below are allowed as comments occupying full lines:

```
| This is a comment field.
| So is this.
```

4. Multiple statements may be put on a line by separating them with a semicolon (;). Remember, however, that anything after a comment character (including statement separators) is a comment.

n general, blanks or tabs are allowed anywhere in a statement; that is, multiple blanks re allowed in the operand field to separate symbols from operators. Blanks are

significant only when they occur in a character string (e.g., as the operand of an **.ascii** pseudo–op) or in a character constant. At least one blank or tab must appear between the opcode and the operand field of a statement.

7.4.1 Label Field

A label is a user–defined symbol that is assigned the value of the current location counter, both of which are entered into the assembler's symbol table. The value of the label is relocatable.

A label is a symbolic means of referring to a specific location within a program. If present, a label always occurs first in a statement and must be terminated by a colon. A maximum of ten labels may be defined by a single source statement. The collection of label definitions in a statement is called the "label–field."

The format of a label–field is:

symbol: [*symbol*:] ...

Examples:

```
start:
name: name2:        | Multiple symbols
7$:                 | A local symbol (see below)
```

7.4.2 Opcode Field

The opcode field of an assembly language statement identifies the statement as either a machine instruction, or an assembler directive (pseudo–op). One or more blanks (or tabs) must separate the opcode field from the operand field in a statement. No blanks are necessary between the label and opcode fields, but they are recommended to improve readability of programs.

A machine instruction is indicated by an instruction mnemonic. Conventions used in *as* for instruction mnemonics are described in a later section, along with a complete list of opcodes.

An assembler directive, or pseudo–op, performs some function during the assembly process. It does not produce any executable code, but it may assign space in a program for data.

As is case–sensitive. Operators and operands may only be lowercase.

7.4.3 Operand–Field

As makes a distinction between operand–field and operand. Several machine instructions and assembler directives require one or more arguments, and each of these is referred to as an "operand". In general, an operand field consists of zero, one, or two operands, and in all cases, operands are separated by a comma. In other words, the format for an operand–field is:

[*operand* [, *operand*]...]

The format of the operand field for machine instruction statements is the same for all

instructions. The format of the operand field for assembler directives depends on the directive itself.

7.4.4 Comment Field

The comment delimiter is the vertical bar, (|), not the semicolon, (;). The semicolon is the statement separator. The comment field consists of all characters on a source line following and including the comment character. These characters are ignored by the assembler. Any character may appear in the comment field, with the exception of the newline character, which starts a newline.

7.5 Symbols and Expressions

This section describes the various components of *as* expressions: symbols, numbers, terms, and expressions.

7.5.1 Symbols

A symbol consists of 1 to 32 characters, with the following restrictions:

1. Valid characters include A–Z, a–z, 0–9, period (.), underscore (_), and dollar sign ($).

2. The first character must not be numeric, unless the symbol is a local symbol.

There is no limit to the size of symbols, except the practical issue of running out of symbol memory in the assembler. However, be aware that the current C compiler only generates eight–character symbol names, so a symbol greater than eight–characters in length that you think is the same in both C and assembly may not match. Uppercase and lowercase are distinct (e.g., "Name" and "name" are separate symbols). The period (.) and dollar sign ($) characters are valid symbol characters, but they are reserved for system software symbols such as system calls and should not appear in user–defined symbols.

A symbol is said to be "declared" when the assembler recognizes it as a symbol of the program. A symbol is said to be "defined" when a value is associated with it. With the exception of symbols declared by a **.globl** directive, all symbols are defined when they are declared. A label symbol (which represents an address in the program) may not be redefined; other symbols are allowed to receive a new value.

There are several ways to declare a symbol:

1. As the label of a statement

2. In a direct assignment statement

3. As an external symbol via the **.globl** directive

4. As a common symbol via the **.comm** directive

5. As a local symbol

7.5.1.1 Direct Assignment Statements

A direct assignment statement assigns the value of an arbitrary expression to a specified symbol. The format of a direct assignment statement is:

symbol = [*symbol* =] ... *expression*

Examples of valid direct assignments are:

```
vect_size =     4
vectora   =     /fffe
vectorb   =     vectora-vect_size
CRLF      =     /0D0A
```

Any symbol defined by direct assignment may be redefined later in the program, in which case its value is the result of the last such statement. A local symbol may be defined by direct assignment; a label or register symbol may not be redefined.

If the expression is absolute, then the symbol is also absolute, and may be treated as a constant in subsequent expressions. If the expression is relocatable, however, then symbol is also relocatable, and is considered to be declared in the same program section as the expression. See the discussion in a later section of absolute and relocatable expressions.

7.5.1.2 Register Symbols

Register symbols are symbols used to represent machine registers. Register symbols are usually used to indicate the register in the register field of a machine instruction. The register symbols known to the assembler are given at the end of this chapter.

7.5.1.3 External Symbols

A program may be assembled in separate modules, and then linked together to form a single program (see *ld*(CP)). External symbols may be defined in each of these separate modules. A symbol that is declared (given a value) in one module may be referenced in another module by declaring the symbol to be external in both modules. There are two forms of external symbols: those defined with the **.globl** directive and those defined with the **.comm** directive. See Section 8.7.6 for more information on these directives.

7.5.1.4 Local Symbols

Local symbols provide a convenient means of generating labels for branch instructions. Use of local symbols reduces the possibility of multiply-defined

symbols in a program, and separates entry point symbols from local references, such as the top of a loop. Local symbols cannot be referenced by other object modules.

Local symbols are of the form *n* $ where *n* is any integer. Valid local symbols include:

```
27$
394$
```

A local symbol is defined and referenced only within a single local symbol block (lsb). A new local symbol block is entered when either:

1. A label is declared, or
2. A new program section is entered.

There is no conflict between local symbols with the same name that appear in different local symbol blocks.

7.5.2 Assembly Location Counter

The assembly location counter is the period character (.); hence its name "dot". When used in the operand field of any statement, dot represents the address of the first byte of the statement. Even in assembly directives, it represents the address of the start of the directive. A dot appearing as the third argument in a .byte directive would have the value of the address where the first byte was loaded; it is not updated "during" the directive.

For example:

```
movl .,d1    | load value of program counter into d1
```

At the beginning of each assembly pass, the assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of generated code. However, the location where the code is stored may be changed by a direct assignment altering the location counter:

. = *expression*

This *expression* must not contain any forward references, must not change from one pass to another, and must not have the effect of reducing the value of dot. Note that setting dot to an absolute position may not have quite the effect you expect if you are linking an *as* output file with other files, since dot is maintained relative to the origin of the output file and not the resolved position in memory. Storage area may also be reserved by advancing dot. For example, if the current value of dot is 1000, the direct assignment statement:

```
TABLE: . = . + /100
```

would reserve 100 (hex) bytes of storage, with the address of the first byte as the value of TABLE. The next instruction would be stored at address 1100. Note that

```
.blkb 100
```

is a substantially more readable way of doing the same thing.

The :p operator, discussed in a later section, allows you to assemble values that are location–relative, both locally (within a module) and across module boundaries, without explicit address arithmetic.

7.5.3 Program Sections

As in XENIX, programs to *as* are divided into two sections: text and data. These sections are interpreted as instruction space and initialized data space, respectively.

In the first pass of the assembly, *as* maintains a separate location counter for each section. Thus, for code like the following:

```
        .text
LABEL1: movw  d1,d2
        .data
LABEL2: .word 27
        .text
LABEL3: addl  d2,d1
        .data
LABEL4: .byte 4
```

LABEL1 will immediately precede LABEL3, and LABEL2 will immediately precede LABEL4 in the output. At the end of the first pass, *as* rearranges all the addresses so that the sections will be output in the following order: text, then data. The resulting output file is an executable image with all addresses correctly resolved, with the exception of **.comm** variables and undefined **.globl** variables. For more information on the format of the output file, consult *a.out*(F).

7.5.4 Constants

All constants are considered absolute quantities when appearing in an expression.

7.5.4.1 Numeric Constants

Any symbol beginning with a digit is assumed to be a number, and will be interpreted in the default decimal radix. Individual numbers may be evaluated in any of the five valid radices: decimal, octal, hexadecimal, character, and binary. The default decimal radix is only used on "bare" numbers, i.e., sequences of digits. Numbers may be represented in other radices as defined by the following table. The other three radices

require a prefix:

Radix	*Prefix*	*Example*	
octal	^ (up-arrow)	^17	equals 15 base 10.
octal	0	017	equals 15 base 10.
hex	/ (slash)	/A1	equals 161 base 10.
hex	0x	0xA1	equals 161 base 10.
char	' (quote)	'a	equals 97 base 10.
char	' (quote)	'\n	equals 10 base 10.
binary	% (percent)	%11011	equals 27 base 10.

Letters in hex constants may be uppercase or lowercase; e.g., /aa=/Aa=/AA=170. Illegal digits for a particular radix generate an error (e.g., ^018). While the C character constant syntax is supported,
you cannot define character constants with a number (e.g., '\27) as this is more easily represented in one of the other formats.

7.5.5 Operators

An operator is either a unary operator requiring a single operand, or a binary operator requiring two operands. Operators of each type are described below.

7.5.5.1 Unary Operators

There are three unary operators in *as*:

Operator	*Function*
+	unary plus, has no effect.
-	unary minus.
:p	program displacement

The ":p" operator is a suffix that can be applied to a relocatable expression. It replaces the value of the expression with the displacement of that value from the current location (not dot). This is implemented with displacement relocation, so that it also works

across modules.

7.5.5.2 Binary Operators

Binary operators include:

Operator	*Description*	*Example*	*Value*
+	Addition	3+4	7.
−	Subtraction	3−4	−1., or/FFFF
*	Multiplication	4*3	12.
/	Division	12/4	3.
|	Logical OR	%01101 | %00011	%01111
&	Logical AND	%01101&%00011	%00001
^	Remainder	5^3	2.

Each operator is assumed to work on a 32−bit number. If the value of a particular term occupies only 8 or 16 bits, the sign bit is extended into the high byte.

Sometimes errors in expressions can be fixed by breaking the expressions into multiple statements using direct assignment statements.

7.5.6 Terms

A term is a component of an expression. A term may be one of the following:

1. A number whose 32−bit value is used
2. A symbol
3. A term preceded by a unary operator. For example, both "term" and "term" may be considered terms. Multiple unary operators are allowed; e.g. "+−−+A" has the same value as "A".

7.5.7 Expressions

Expressions are combinations of terms joined together by binary operators. An expression is always evaluated to a 32−bit value. If the instruction calls for only 1 byte (e.g., **.byte**), then the low−order 8 bits are used.

Expressions are evaluated left to right with no operator precedence. Thus "1 + 2 * 3" evaluates to 9, not 7. Unary operators have precedence over binary operators since they are considered part of a term, and both terms of a binary operator must be evaluated before the binary operator can be applied.

A missing expression or term is interpreted as having a value of zero. In this case, the following error message is generated:

Invalid Expression

An "Invalid Operator" error means that a valid end-of-line character or binary operator was not detected after the assembler processed a term. In particular, this error will be generated if an expression contains a symbol with an illegal character, or if an incorrect comment character was used.

Any expression, when evaluated, is either absolute, relocatable, or external:

1. An expression is absolute if its value is fixed. Absolute expressions are those whose terms are constants, or symbols assigned constants with an assignment statement. Also absolute is a relocatable expression minus a relocatable term, where both items belong to the same program section.

2. An expression is *relocatable* if its value is fixed relative to a base address, but will have an offset value when it is linked, or loaded into core. All labels of a program defined in relocatable sections are relocatable terms, and any expression that contains them must only add or subtract constants to their value. For example, assume the symbol "sym" was defined in a relocatable section of the program. Then the following demonstrates the use of relocatable expressions:

sym	Relocatable
sym+5	Relocatable
sym-'A	Relocatable
sym*2	Not relocatable
2-sym	Not relocatable, since the expression cannot be linked by adding sym's offset to it.
sym-sym2	Absolute, since the offsets added to *sym* and *sym2* cancel each other out.

3. An expression is "external" (i.e., or global) if it contains an external symbol not defined in the current program. The same restrictions on expressions containing relocatable symbols apply to expressions containing external symbols.

   An important exception is the expression *sym-sym2* where both *sym* and *sym2* are external symbols. Expressions of this kind are disallowed.

7.6 Instructions and Addressing Modes

This section describes the conventions used in *as* to specify instruction mnemonics and addressing modes.

7.6.1 Instruction Mnemonics

The instruction mnemonics used by *as* are described in the Motorola MC68000 User's Manual with a few variations. Most of the MC68000 instructions can apply to byte,

word or to long operands, thus in *as* the normal instruction mnemonic is suffixed with **b**, **w**, or **l** to indicate which length of operand was intended. For example, there are three mnemonics for the **add** instruction: **addb**, **addw**, and **addl**.

Branch and call instructions come in 3 forms: the **bra**, **jra**, **bsr** and **jbsr** forms may only take a label as argument. For the **bra** and **bsr** forms, the assembler will always produce a long (16-bit) pc relative address. For the **jra** and **jbsr** forms, the assembler will produce the shortest form of binary it can. This may be 8-bit or 16-bit pc relative, or 32-bit absolute. The 32-bit absolute is implemented for conditional branches by inverting the sense of the condition and branching around a 32-bit **jmp** instruction. The 32-bit form will be generated whenever the assembler can't figure out how far away the addressed location is; for example, branching to an undefined symbol or a calculated value such as branching to a constant location.

7.6.2 Operand Addressing Modes

These effective addressing modes specify the operand(s) of an instruction. For details of the effective addressing modes, see the "MC68000 User's Manual." Note also that not all instructions allow all addressing modes. Details are given in the "MC68000 User's Manual" in Appendix B under the specific instruction.

In the examples that follow, when two examples are given, the first example is based on the assembly format suggested by Motorola. The second example is in what is called "Register Transfer Language" or RTL and is used to describe the register transfers that are occurring within the machine. It is provided for compatibility. Either syntax is accepted, and it is permissible to mix the two types of syntax within a module or even within a line when two effective address fields are allowed. Beware, however, that a warning message will be generated when the assembler notices such a mix.

Many of the effective address modes have other names, by which they may be more commonly known. In the following descriptions, this name appears to the right of the Motorola name in parentheses.

Data Register Direct

```
addl    d0,d1
```

Address Register Direct

```
addl    a0,a0
```

Address Register Indirect (indirect)

```
addl    (a0),d1
addl    a0@,d1
```

Address Register Indirect With Postincrement (autoinc)

```
movl    (a7)+,d1
movl    a7@+,d1
```

Address Register Indirect With Predecrement (autodec)

```
movl    dl, -(a7)
movl    dl,a7@-
```

\ddress Register Indirect With Displacement(indexed)

This form includes a signed 16-bit displacement. These displacements may be symbolic.

```
movl    12(a6),dl
movl    a6@(12),dl
```

\ddress Register Indirect With Index(double-indexed)

This form includes a signed 8-bit displacement and an index register. The size of the index register is given by following its specification with a ":w" or a ":l". If neither is specified, ":l" is assumed.

```
movl    12(a6,d0:w),dl
movl    a6@(12,d0:w),dl
```

\bsolute Short Address

```
movl    xx:w,dl
```

\bsolute Long Address (absolute)

This is the assumed addressing mode should the given value be a constant. This is not true of branch and call instructions. Note also that the second example here is not RTL syntax, but is provided only because it is also allowed.

```
movl    xx,dl
movl    xx:l,dl
```

rogram Counter With Displacement (pc relative)

When pc relative addressing is used, such as

```
pea name(pc)
```

the assembler will assemble a value that is equal to "name-.", where dot(.) is the position of the value, whether "name" is in the current module or not. You may also cause an expression to be pc relative by suffixing it with a ":p".

```
movl    10(pc),dl
movl    pc@(10),dl
```

Note that if a symbol appears in the above addressing mode (where the 10 is in the example), the symbol's displacement from the extension word will be used in the instruction.

·ogram Counter With Index

```
jmp     switchtab(pc,d0:l)
jmp     pc@(switchtab,d0:l)
switchtab:
```

mediate Data

Note that this is the way to get immediate data. If a number is given with no number sign (#), you get absolute addressing. This does not hold for **jsr** and **jmp** instructions.

```
movl    #47,dl
jmp     somewhere
moveq   #7,dl
```

In the **movem** instruction's register mask field, a special kind of immediate is allowed: the register list. Its syntax is as follows:

<reg [,reg]>

Here, *reg* is any register name. Register names may be given in any order. The assembler automatically takes care of reversing the mask for the auto-decrement addressing mode. Normal immediates are also allowed.

7.7 Assembler Directives

The following assembler directives are available in *as*:

.ascii **.asciz**	stores character strings stores null−appended character strings
.blkb **.blkw** **.blkl**	saves blocks of bytes/words/longs
.byte **.word** **.long**	stores bytes/words/longs
.end	terminates program and identifies execution address
.text **.data** **.bss**	Text program section Data program section Bss program section
.globl **.comm**	declares external symbols declares communal symbols
.even	forces location counter to next word boundary

7.7.1 .ascii .asciz

The **.ascii** directive translates character strings into their 7−bit ASCII (represented as 8−bit bytes) equivalents for use in the source program. The format of the **.ascii** directive is as follows:

.ascii *"character−string"*

where *character−string* contains any character valid in a character constant. Obviously, a newline must not appear within the character string. (It can be represented by the escape sequence "\n" as described below). The quotation mark (") is the delimiter character, which must not appear in the string unless preceded by a backslash (\).

The following escape sequences are also valid as single characters:

X	*Value of X*	
\b	<*backspace*>,	hex /08
\t	<*tab*>,	hex /09
\n	<*newline*>,	hex /0A
\f	<*form−feed*>,	hex /0C
\r	<*return*>,	hex /0D
nnn	hex value of *nnn*	

Several examples follow:

Hex Code Generated:	*Statement:*
22686 5 6C6C6F2074 6865726522	**.ascii "hello there"**
7761 72 6E696E 67 20 2D0707200A	**.ascii "Warning−\007\007\n"**

The **.asciz** directive is equivalent to the **.ascii** directive with a zero (null) byte automatically inserted as the final character of the string. Thus, when a list or text string is to be printed, a search for the null character can terminate the string. Null terminated strings are often used as arguments to XENIX system calls.

7.7.2 .blkb .blkw .blkl

The **.blkb**, **.blkw**, and **.bkkl** directives are used to reserve blocks of storage: **.blkb** reserves bytes, **.blkw** reserves words and **.blkl** reserves longs.

The format is:

[*label:*]	**.blkb**	*expression*
[*label:*]	**.blkw**	*expression*
[*label:*]	**.blkl**	*expression*

where *expression* is the number of bytes or words to reserve. If no argument is given a value of 1 is assumed. The expression must be absolute, and defined during pass 1 (i.e. no forward references).

This is equivalent to the statement "**. = . +** *expression*", but has a much more transparent meaning.

7.7.3 .byte .word .long

The **.byte**, **.word**, and **.long** directives are used to reserve bytes and words and to initialize them with values.

The format is:

[*label:*]	**.byte**	[*expression*]	[, *expression*]...
[*label:*]	**.word**	[*expression*]	[, *expression*]...
[*label:*]	**.long**	[*expression*]	[, *expression*]...

The **.byte** directive reserves 1 byte for each expression in the operand field and initializes the value of the byte to be the low-order byte of the corresponding expression. Note that multiple expressions must be separated by commas. A blank expression is interpreted as zero, and no error is generated.

For example,

.byte a,b,c,s	reserves 4 bytes.
.byte,,,,	reserves 5 bytes, each with a value of zero.
.byte	reserves 1 byte, with a value of zero.

The semantics for **.word** and **.long** are identical, except that 16-bit or 32-bit words are reserved and initialized. Be forewarned that the value of dot within an expression is that of the beginning of the statement, not of the value being calculated.

7.7.4 .end

The **.end** directive indicates the physical end of the source program. The format is:

.end

The **.end** is not required; reaching the end of file has the same effect.

7.7.5 .text .data .bss

These statements change the "program section" where assembled code will be loaded.

7.7.6 .globl .comm

Two forms of external symbols are defined with the **.globl** and **.comm** directives.

External symbols are declared with the **.globl** assembler directive. The format is:

.globl *symbol* [, *symbol* ...]

For example, the following statements declare the array TABLE and the routine SRCH to be external symbols:

```
        .globl TABLE, SRCH
TABLE:  .blkw 10.
SRCH:   movw   TABLE,a0
```

External symbols are only declared to the assembler. They must be defined (i.e., given a value) in some other statement by one of the methods mentioned above. They need not be defined in the current program; in this case they are flagged as "undefined" in the symbol table. If they are undefined, they are considered to have a value of zero in expressions.

It is generally a good idea to declare a symbol as **.globl** before using it in any way. This is particularly important when defining absolutes.

The other form of external symbol is defined with the **.comm** directive. The **.comm** directive reserves storage that may be communally defined, i.e., defined mutually by several modules. The link editor, *ld* (CP) resolves allocation of **.comm** regions. The syntax of the .comm directive is:

.comm *name constant−expression*

which causes *as* to declare the *name* as a common symbol with a value equal to the expression. For the rest of the assembly this symbol will be treated as though it were an undefined global. *As* does not allocate storage for common symbols; this task is left to the loader. The loader computes the maximum size of each common symbol that may appear in several load modules, allocates storage for it in the *bss* section, and resolves linkages.

7.7.7 .even

This directive advances the location counter if its current value is odd. This is useful for forcing storage allocation on a word boundary after a **.byte** or **.ascii** directive. Note that many things may not be on an odd boundary in *as*, including instructions, and word and long data.

7.8 Operation Codes

Below are all opcodes recognized by *as*:

abcd	bmi	dbra	movb	rte
addb	bmis	dbt	movw	rtr
addw	bne	dbvc	movl	rts
addl	bnes	dbvs	movemw	sbcd
addqb	bpl	divs	moveml	scc
addqw	bpls	divu	movepw	scs
addql	bra	eorb	movepl	seq
addxb	bras	eorw	moveq	sf
addxw	bset	eorl	muls	sge
addxl	bsr	exg	mulu	sgt
andb	bsrs	extw	nbcd	shi
andw	btst	extl	negb	sle
andl	bvc	jbsr	negw	sls
aslb	bvcs	jcc	negl	slt
aslw	bvs	jcs	negxb	smi
asll	bvss	jeq	negxw	sne
asrb	chk	jge	negxl	spl
asrw	clrb	jgt	nop	st
asrl	clrw	jhi	notb	stop
bcc	clrl	jle	notw	subb
bccs	cmpb	jls	notl	subw
bchg	cmpw	jlt	orb	subl
bclr	cmpl	jmi	orw	subqb
bcs	cmpmb	jmp	orl	subqw
bcss	cmpmw	jne	pea	subql
beq	cmpml	jpl	reset	subxb
beqs	dbcc	jra	rolb	subxw
bge	dbcs	jsr	rolw	subxl
bges	dbeq	jvc	roll	svc
bgt	dbf	jvs	rorb	svs
bgts	dbge	lea	rorw	swap
bhi	dbgt	link	rorl	tas
bhis	dbhi	lslb	roxlb	trap
ble	dble	lslw	roxlw	trapv
bles	dbls	lsll	roxll	tstb
bls	dblt	lsrb	roxrb	tstw
blss	dbmi	lsrw	roxrw	tstl
blt	dbne	lsrl	roxrl	unlk
blts	dbpl			

The following pseudooperations are recognized:

.ascii
.asciz
.blkb
.blkl
.blkw
.bss
.byte
.comm
.data
.end
.even
.globl
.long
.text
.word

The following registers are recognized:

d0 d1 d2 d3 d4 d5 d6 d7
a0 a1 a2 a3 a4 a5 a6 a7
sp pc cc sr

7.9 Error Messages

If there are errors in an assembly, an error message appears on the standard error output (usually the terminal) giving the type of error and the source line number. If an assembly listing is requested, and there are errors, the error message appears before the offending statement. If there were no assembly errors, then there are no messages, thus indicating a successful assembly. Some diagnostics are only warnings and the assembly is successful despite the warnings.

The common error codes and their probable causes, appear below:

Invalid character
An invalid character for a character constant or character string was encountered.

Multiply defined symbol
A symbol has appeared twice as a label, or an attempt has been made to redefine a label using an = statement. This error message may also occur if the value of a symbol changes between passes.

Offset too large
A displacement cannot fit in the space provided for by the instruction.

Invalid constant
An invalid digit was encountered in a number.

Invalid term
The expression evaluator could not find a valid term that was either a symbol, constant or expression. An invalid prefix to a number or a bad symbol name in an operand will generate this.

Nonrelocatable expression
A required relocatable expression was not found as an operand. It was not provided.

Invalid operand
An illegal addressing mode was given for the instruction.

Invalid symbol
A symbol was given that does not conform to the rules for symbol formation.

Invalid assignment
An attempt was made to redefine a label with an = statement.

Invalid opcode
A symbol in the opcode field was not recognized as an instruction mnemonic or directive.

Bad filename
An invalid filename was given.

Wrong number of operands
An instruction has either too few or too many operands as required by the syntax of the instruction.

Invalid register expression
An operand or operand element that must be a register is not, or a register name is used where it may not be used. For example, using an address register in a **moveq** instruction, which only allows data registers will produce this error message; as will using a register name as a label with a **bra** instruction.

Odd address
An instruction or data item that must start at an even address does not.

Inconsistent effective address syntax
Both assembly and RTL syntax appear within a single module.

Nonword memory shift
An in-memory shift instruction was given a size other than 16 bits.

Chapter 8
Lex: A Lexical Analyzer

8.1 Introduction

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a C program that recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to lex. The lex code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the user are executed. The lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by lex, the corresponding fragment is executed.

The user supplies the additional code needed to complete his tasks, including code written by other generators. The program that recognizes the expressions is generated in the from the user's C program fragments. Lex is not a complete language, but rather a generator representing a new language feature added on top of the C programming language.

Lex turns the user's expressions and actions (called *source* in this chapter) into a C program named *yylex*. The *yylex* program recognizes expressions in a stream (called input in this chapter) and performs the specified actions for each expression as it is detected.

Consider a program to delete from the input all blanks or tabs at the ends of lines. The following lines

```
%%
[\t]+$   ;
```

are all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression that matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates one or more of the previous item; and the dollar sign ($) indicates the end of the line. No action is specified, so the program generated by lex will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[\t]+$   ;
[\t]+    printf(" ");
```

The finite automaton generated for this source scans for both rules at once, observes at the termination of the string of blanks or tabs whether or not there is a newline character, and then executes the desired rule's action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule matches all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is especially easy to interface lex and yacc. Lex programs recognize only regular expressions; yacc writes parsers that accept a large class of context-free grammars, but that require a lower level analyzer to recognize input tokens. Thus, a combination of lex and yacc is often appropriate. When used as a preprocessor for a later parser generator, lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. Additional programs, written by other generators or by hand, can be added easily to programs written by lex. Yacc users will realize that the name *yylex* is what yacc expects its lexical analyzer to be named, so that the use of this name by lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a lex program to recognize and partition an input stream is proportional to the length of the input. The number of lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by lex.

In the program written by lex, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source that can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, lex will recognize *ab* and leave the input pointer just before *cd*. Such backup is more costly than the processing of simpler languages.

8.2 Lex Source Format

The general format of lex source is:

{definitions}
%%
{rules}
%%
{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program that copies the input to the output unchanged.

In the lex program format shown above, the rules represent the user's control decisions. They make up a table in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. Thus the following individual rule might appear:

```
integer    printf("found keyword INT");
```

This looks for the string *integer* in the input stream and prints the message

```
found keyword INT
```

whenever it appears in the input text. In this example the C library function *printf*() is used to print the string. The end of the lex regular expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour          printf("color");
mechanise       printf("mechanize");
petrol          printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gasoleum*; a way of dealing with such problems is described in a later section.

8.3 Lex Regular Expressions

A regular expression specifies a set of strings to be matched. It contains text characters (that match the corresponding characters in the strings being compared) and operator characters (these specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters. Thus, the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D*.

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

If any of these characters are to be used literally, they needed to be quoted individually with a backslash (\) or as a group within quotation marks ("). The quotation mark operator (") indicates that whatever is contained between a pair of quotation marks is to be taken as text characters. Thus

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every nonalphanumeric character being used as a text character, you need not memorize the above list of current operator characters.

An operator character may also be turned into a text character by preceding it with a backslash (\) as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. The quoting mechanism can also be used to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within brackets must be quoted. Several normal C escapes with the backslash (\ ) are recognized:

\n newline

\t tab

\b backspace

\\ backslash

Since newline is illegal in an expression, a \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

8.4 Invoking *lex*

There are two steps in compiling a lex source program. First, the lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of lex

subroutines. The generated program is in a file named lex.yy.c. The I/O library is defined in terms of the C standard library.

The library is accessed by the loader flag *-ll*. So an appropriate set of commands is

```
lex source
cc lex.yy.c -ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use lex with yacc see the section "Lex and Yacc" in this chapter and Chapter 9, "Yacc: A Compiler-Compiler"". Although the default lex I/O routines use the C standard library, the lex automata themselves do not do so. If private versions of *input*, *output*, and *unput* are given, the library can be avoided.

8.5 Specifying Character Classes

Classes of characters can be specified using brackets: [and]. The construction

```
[abc]
```

matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are the backslash (\), the dash (-), and the caret (^). The dash character indicates ranges. For example

```
[a-z0-9<>_]
```

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using the dash between any pair of characters that are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and causes a warning message. If it is desired to include the dash in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the plus and minus signs.

In character classes, the caret (^) operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

```
[^abc]
```

matches all characters except *a*, *b*, or *c*, including all special or control characters; or

```
[^a-zA-Z]
```

is any character which is not a letter. The backslash (\) provides an escape mechanism within character class brackets, so that characters can be entered literally by preceding them with this character.

8.6 Specifying an Arbitrary Character

To match almost any character, the period (.) designates the class of all characters except a newline. Escaping into octal is possible although nonportable. For example

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

8.7 Specifying Optional Expressions

The question mark (?) operator indicates an optional element of an expression. Thus

```
ab?c
```

matches either *ac* or *abc*. Note that the meaning of the question mark here differs from its meaning in the shell.

8.8 Specifying Repeated Expressions

Repetitions of classes are indicated by the asterisk (*) and plus (+) operators. For example

```
a*
```

matches any number of consecutive *a* characters, including zero; while *a+* matches one or more instances of *a*. For example,

```
[a-z]+
```

matches all strings of lowercase letters, and

```
[A-Za-z][A-Za-z0-9]*
```

matches all alphanumeric strings with a leading alphabetic character; this is a typical expression for recognizing identifiers in computer languages.

8.9 Specifying Alternation and Grouping

The vertical bar (|) operator indicates alternation. For example

```
(ab|cd)
```

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary at the outside level. For example

```
ab|cd
```

would have sufficed in the preceding example. Parentheses should be used for more complex expressions, such as

```
(ab|cd+)?(ef)*
```

which matches such strings as *abefef*, *efefef*, *cdef*, and *cddd*, but not *abc*, *abcd*, or *abcdef*.

8.10 Specifying Context Sensitivity

Lex recognizes a small amount of surrounding context. The two simplest operators for this are the caret (^) and the dollar sign ($). If the first character of an expression is a caret, then the expression is only matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of the caret, complementation of character classes, since complementation only applies within brackets. If the very last character is dollar sign, the expression only matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the slash (/) operator, which indicates trailing context. The expression

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in lex by specifying start conditions as explained in the section "Specifying Left Context Sensitivity". If a rule is only to be executed when the lex automaton interpreter is in start condition *x*, the rule should be enclosed in angle brackets:

```
<x>
```

If we considered being at the beginning of a line to be start condition ONE, then the caret (^) operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

8.11 Specifying Expression Repetition

The curly braces ({ and }) specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression.

8.12 Specifying Definitions

The definitions are given in the first part of the lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of the character *a*.

Finally, an initial percent sign (*%*) is special, since it is the separator for lex source segments.

8.13 Specifying Actions

When an expression is matched by a pattern of text in the input, lex executes the corresponding action. This section describes some features of lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When lex is being used with yacc, this is the normal situation. You may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement *;* as an action causes this result. A frequent rule is

```
[ \t\n]   ;
```

which causes the three spacing characters (blank, tab, and newline) to be

ignored.

Another easy way to avoid writing actions is to use the repeat action character, |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "          |
"\t"         |
"\n"         ;
```

with the same result, although in a different style. The quotes around \n and \t are not required.

In more complex actions, you often want to know the actual text that matched some expression like:

```
[a-z]+
```

Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a-z]+    printf("%s", yytext);
```

prints the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is *print string* where the percent sign (%) indicates data conversion, and the *s* indicate string type, and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO. For example

```
[a-z]+    ECHO;
```

is the same as the preceding example. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule that is not desired. For example, if there is a rule that matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form

```
[a-z]+
```

is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence lex also provides a count of the number of characters matched in the variable, *yyleng*. To count both the number of words and the number of characters in words in the input, you might write

```
[a-zA-Z]+        {words++; chars += yyleng;}
```

which accumulates in the variables *chars* the number of characters in the words

recognized. The last character in the string matched can be accessed with:

```
yytext[yyleng-1]
```

Occasionally, a lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore*() can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string will overwrite the current entry in *yytext*. Second, *yyless*(n) may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the slash (/) operator, but in a different form.

For example, consider a language that defines a string as a set of characters between quotation marks ("), and provides that to include a quotation mark in a string, it must be preceded by a backslash (\). The regular expression that matches this is somewhat confusing, so that it might be preferable to write

```
\"[^"]*  {
         if (yytext[yyleng-1] == '\\')
             yymore();
         else
             ... normal user processing
         }
```

which, when faced with a string such as

```
"abc\"def"
```

will first match the five characters

```
"abc\
```

and then the call to *yymore*() will cause the next part of the string,

```
"def
```

to be tacked on the end. Note that the final quotation mark terminating the string should be picked up in the code labeled normal processing.

The function *yyless*() might be used to reprocess text in various circumstances. Consider the problem in the older C syntax of distinguishing the ambiguity of =-*a*. Suppose it is desired to treat this as =- *a* and to print a message. A rule might be

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as =-.

Alternatively it might be desired to treat this as = -*a*. To do this, just return the minus sign as well as the letter to the input. The following performs the interpretation:

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```

Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second: no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of =-*3*, however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, lex also permits access to the I/O routines it uses. They include:

1. *input*() which returns the next input character;
2. *output*(c) which writes the character *c* on the output; and
3. *unput*(c) which pushes the character *c* back onto the input stream to be read later by *input*().

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or

output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end-of-file; and the relationship between *unput* and *input* must be retained or the lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule containing a slash (/) or ending in one of the following characters implies lookahead:

```
+ * ? $
```

Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by lex. The standard lex library imposes a 100 character limit on backup.

Another lex library routine that you sometimes want to redefine is *yywrap*() which is called whenever lex reaches an end-of-file. If *yywrap* returns a 1, lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* that arranges for new input and returns 0. This instructs lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule that recognizes end-of-file; the only access to this condition is through *yywrap*(). In fact, unless a private version of *input*() is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

8.14 Handling Ambiguous Source Rules

Lex can handle ambiguous specifications. When more than one expression can match the current input, lex chooses as follows:

- The longest match is preferred.
- Among rules that match the same number of characters, the first given rule is preferred.

For example, suppose the following rules are given:

```
integer   keyword action ...;
[a-z]+    identifier action ...;
```

If the input is *integers*, it is taken as an identifier, because

```
[a-z]+
```

matches 8 characters while

```
integer
```

matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g., *int*) does not match the expression *integer*, so the identifier interpretation is used.

The principle of preferring the longest match makes certain constructions dangerous, such as the following:

```
.*
```

For example

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression matches

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, stops after 'first'. The consequences of errors like this are mitigated by the fact that the dot (.) operator does not match a newline. Therefore, no more than one line is ever matched by such expressions. Don't try to defeat this with expressions like

```
[.\n]+
```

or their equivalents: the lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that the period (.) does not include the newline. Since *she* includes *he*, lex will normally not recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means go do the next alternative. It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

These rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that *she* includes *he*, but not vice versa, and omit the REJECT action on *he*; in other cases, however, it would not be possible to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+  { ... ; REJECT;}
a[cd]+  { ... ; REJECT;}
```

If the input is *ab*, only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word *the* is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```
%%
[a-z][a-z] {digram[yytext[0]][yytext[1]]++; REJECT;}
.       ;
\n      ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

Remember that REJECT does not rescan the input. Instead it remembers the results of the previous scan. This means that if a rule with trailing context is

found, and REJECT executed, you must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction to ability to manipulate the not-yet-processed input.

8.15 Specifying Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The caret (^) operator, for example, is a prior context operator, recognizing immediately preceding left context just as the dollar sign ($) recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments:

1. The use of flags, when only a few rules change from one environment to another
2. The use of start conditions with rules
3. The use multiple lexical analyzers running together.

In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since lex is not involved at all. It may be more convenient, however, to have lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line that began with the letter *a*, changing *magic* to *second* on every line that began with the letter *b*, and changing *magic* to *third* on every line that began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag = 0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to lex in the definitions section with a line reading

```
%Start  name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with angle brackets. For example

```
<name1>expression
```

is a rule that is only recognized when lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To return to the initial state

```
BEGIN 0;
```

resets the initial condition of the lex automaton interpreter. A rule may be active in several start conditions; for example:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>magic    printf("first");
<BB>magic    printf("second");
<CC>magic    printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but lex does the work rather than the user's code.

8.16 Specifying Source Definitions

Remember the format of the lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. You will need additional options, though, to define variables for use in your program and for use by lex. These can go either in the definitions section or in the rules section.

Remember that lex is turning the rules into a program. Any source not intercepted by lex is copied into the generated program. There are three classes of such things:

1. Any line that is not part of a lex rule or action which begins with a blank or tab is copied into the lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by lex which contains the actions. This material must look like program fragments, and should precede the first lex rule.

   As a side effect of the above, lines that begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the lex source or the generated code. The comments should follow the conventions of the C language.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column

1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of formats, is copied out after the lex output.

Definitions intended for lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and beginning in column 1, is assumed to define lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                       [0-9]
E                       [DEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}                         printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a FORTRAN expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including a character set table, a list of start conditions, or adjustments to the default size of arrays within lex itself for larger source programs. These possibilities are discussed in the section "Source Format".

8.17 Lex and Yacc

If you want to use lex with yacc, note that what lex writes is a program named *yylex*(), the name required by yacc for its analyzer. Normally, the default main program on the lex library calls this routine, but if yacc is loaded, and its main program is used, yacc will call *yylex*(). In this case, each lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to yacc's names for tokens is to compile the lex output file as part of the yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of yacc input. Supposing the grammar to be named *good* and the lexical rules to be named *better* the XENIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The yacc library (–ly) should be loaded before the lex library, to obtain a main program which invokes the yacc parser. The generation of lex and yacc programs can be done in either order.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable lex source program to do just that:

```
%%
        int k;
[0-9]+  {
        k = atoi(yytext);
        if (k%7 == 0)
            printf("%d", k+3);
        else
            printf("%d",k);
        }
```

The rule [0–9]+ recognizes strings of digits; *atoi*() converts the digits to binary and stores the result in *k*. The remainder operator (%) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
        int k;
-?[0-9]+  {
        k = atoi(yytext);
        printf("%d", k%7 == 0 ? k+3 : k);
        }
-?[0-9.]+       ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a decimal point or preceded by a letter will be

picked up by one of the last two rules, and not changed. The **if–else** has been replaced by a C conditional expression to save space; the form *a?b:c* means: if *a* then *b* else *c*.

For an example of statistics gathering, here is a program which makes histograms of word lengths, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return*(1); indicates that lex is to perform wrapup. If *yywrap*() returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap*() that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written to convert double precision FORTRAN to single precision FORTRAN. Because FORTRAN does not distinguish between upper- and lowercase letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
.       .
.       .
.       .
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes *double precision* to *real*, or *DOUBLE PRECISION* to *REAL*.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
"     "[^ 0]        ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of the caret (^) here. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+          |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+    |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+    {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' || *p == 'D')
            *p+= 'e'- 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the **for** loop to find the letter "d" or "D". The program then adds "'e'-'d'" which converts it to the next letter of the alphabet. The modified constant, now single precision, is written out again. There follow a series of names which must be respelled to remove their initial "d". By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}      |
{d}{c}{o}{s}      |
{d}{s}{q}{r}{t}   |
{d}{a}{t}{a}{n}   |
...
{d}{f}{l}{o}{a}{t}        printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}      |
{d}{l}{o}{g}10    |
{d}{m}{i}{n}1     |
{d}{m}{a}{x}1     {
        yytext[0] += 'a' - 'd';
        ECHO;
        }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}        {
        yytext[0] += 'r' - 'd';
        ECHO;
}
```

To avoid such names as ***dsinx*** being detected as instances of ***dsin***, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
.       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

8.18 Specifying Character Sets

The programs generated by lex handle character I/O only through the routines *input*, *output*, and *unput*. Thus the character representation provided in these routines is accepted by lex and employed to return values in ***yytext***. For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant:

```
'a'
```

If this interpretation is changed, by providing I/O routines which translate the characters, lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only *%T*. The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. For example:

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

This table maps the lowercase and uppercase letters together into the integers 1 through 26, newline into 27, plus (+) and minus (-) into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a larger number than the size of the hardware character set.

8.19 Source Format

The general form of a lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1. Definitions, in the form "name space translation"
2. Included code, in the form "space code"
3. Included code, in the form

   ```
   %{
   code
   %}
   ```

4. Start conditions, given in the form

   ```
   %S name1 name2 ...
   ```

5. Character set tables, in the form

```
%T
number space character-string
%T
```

6. Changes to internal array sizes, in the form

```
%x  nnn
```

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form:

expression action

where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in lex use the following operators:

x	The character "x"
"x"	An "x", even if x is an operator.
\x	An "x", even if x is an operator.
[xy]	The character x or y.
[x-z]	The characters x, y or z.
[^x]	Any character but x.
.	Any character but newline.
^x	An x at the beginning of a line.
<y>x	An x when lex is in start condition y.
x$	An x at the end of a line.

x?	An optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	An x or a y.
(x)	An x.
x/y	An x but only if followed by y.
{xx}	The translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x.

Chapter 9
Yacc: A Compiler-Compiler

9.1 Introduction

Computer program input generally has some structure; every computer program that does input can be thought of as defining an input language which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The name yacc itself stands for "yet another compiler-compiler". The yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification. The class of specifications accepted is a very general one: LALR grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

Yacc provides a general tool for imposing structure on the input to a computer program. The yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (called the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be:

```
date : month_name day ',' year  ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma (,) is enclosed in single quotation marks; this implies 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 controlling the input. Thus, with proper definitions, the input:

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. A structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures 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. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters, such as the comma, must also be passed through the lexical analyzer and are 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

In most cases, this new rule could be slipped in to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. 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 can often 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 constructions which 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 next several sections describe:

- The preparation of grammar rules
- The preparation of the user supplied actions associated with the grammar rules
- The preparation of lexical analyzers
- The operation of the parser
- Various reasons why yacc may be unable to produce a parser from a specification, and what to do about it.
- A simple mechanism for handling operator precedences in arithmetic expressions.
- Error detection and recovery.
- The operating environment and special features of the parsers yacc produces.
- Some suggestions which should improve the style and efficiency of the specifications.

9.2 Specifications

Names refer to either tokens or nonterminal symbols. **yacc** requires token names to be declared as such. In addition, for reasons discussed later, it is often desirable to include the lexical analyzer as part of the specification file. It may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent *%%* marks. (The percent sign (%) is generally used in **yacc** specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal **yacc** specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* ... */, as in C.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of arbitrary length, and may be made up of letters, dot (.), the underscore (_), and noninitial digits. 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 quotation marks ('). As in C, the backslash (\) is an escape character within literals, and all the C escapes are recognized. Thus

'\n'	Newline
'\r'	Return
'\"'	Single quotation mark
'\\'	Backslash
'\t'	Tab
'\b'	Backspace
'\f'	Form feed
'\xxx'	"xxx" in octal

For a number of technical reasons, the ASCII NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, then the vertical bar (|) can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be 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
  ;
```

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 much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token name1 name2 ...
```

in the declarations section. (See Sections 3, 5, and 6 for much more discussion). Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. 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 in fact 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 endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as the end of the file or end of the record.

3.3 Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These 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 statement, and as such can do input and output, call subprograms, and alter external vectors 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
        { printf("a message\n");
          flag = 25;}
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign ($) is used as a signal to yacc in this context.

To return a value, the action normally sets the pseudo-variable $$ to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, ..., which refer to the values returned by the components of the right side of a rule, reading from left to

right. Thus, if the rule is

```
A : B C D ;
```

for example, then **$2** has the value returned by C, and **$3** the value returned by D.

As a more concrete example, consider the rule

```
expr : '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This 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 the examples above, all the actions came at the end of their 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 rule 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

```
A : B
    { $$ = 1; }
    C
    { x = $2;  y = $3; }
    ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually 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 off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT : /* empty */
            { $$ = 1; }
     ;

A    : B $ACT C
            { x = $2;  y = $3; }
     ;
```

In many applications, output is not done directly by the actions; rather, 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 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 the lexical analyzer. For example,

```
%{ int variable = 0; %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The yacc parser uses only names beginning in *yy*; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in a later section.

9.4 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, called 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 chosen by the user. In either case, the *# define* mechanism of C 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:

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
                ...
        case '0':
        case '1':
          ...
        case '9':
                yylval = c-'0';
                return( DIGIT );
                ...
                }
        ...
```

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 programs section of the specification file, the identifier *DIGIT* will be defined as the token number associated with the token *DIGIT*.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C 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.

As mentioned above, the token numbers may be chosen by yacc or by the user. In the default situation, the 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.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user. Hence, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the

end of their input.

A very useful tool for constructing lexical analyzers is lex, discussed in a previous section. These lexical analyzers 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.

9.5 How the Parser Works

Yacc turns the specification file into a C program, 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 (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

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 lookahead 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 lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.

2. Using the current state, and the lookahead 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 lookahead 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 lookahead token. For example, in state 56 there may be an action:

```
IF    shift 34
```

which says, in state 56, if the lookahead 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 lookahead 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 lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a .) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading 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 being reduced is

```
A : x y z ;
```

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 lookahead token is cleared by a shift, and 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 stack: 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 lookahead token is the endmarker, 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 lookahead 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 in a later section.

Consider the following example:

```
%token DING DONG DELL
%%
rhyme : sound  place
      ;
sound : DING  DONG
      ;
place : DELL
      ;
```

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) is:

```
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
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The underscore character (_) is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

DING DONG DELL

It is instructive to follow the steps of the parser while processing this input.

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, becoming the lookahead token. The action in state 0 on *DING* is ***shift 3***, so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is ***shift 6***, so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

sound : DING DONG

This rule has two symbols on the right hand side, so 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; thus state 2 is pushed onto the stack, becoming 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 lookahead 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; the endmarker is obtained, indicated by ***$end*** in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending 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 spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

9.6 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. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has 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, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr - expr
```

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 two legal things, a shift or a reduction, and 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 which choice to make in a given situation is called a disambiguating rule.

Yacc invokes two disambiguating rules by default:

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 input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever 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 a fragment from a programming language involving an if-then-else construction:

```
stat : IF '(' cond ')' stat
     | IF '(' cond ')' stat ELSE stat
     ;
```

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, since 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
        }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last *IF* immediately preceding the *ELSE*. 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 already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

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
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving 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 5. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by "." , is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol

is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

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 after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be 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. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references might be consulted; the services of a local guru might also be appropriate.

9.7 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) ) )
```

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 '-'; 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 %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, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr  : expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | '-' expr %prec '*'
      | NAME
      ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by **yacc** to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The 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 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 the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and 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 an essentially 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.

9.8 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, 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 perform 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 a simple, but reasonably general feature. The token name *error* is reserved for error handling. 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 lookahead token, and performs the action encountered. The lookahead 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
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan 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. Somewhat easier are rules such as

```
stat : error ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before

the next ';' cannot be shifted, and are discarded. When the ';' 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. A possible error rule might be

```
input : error '\n' { printf( "Reenter line: "); } input
                { $$ = $4;}
```

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 be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input : error '\n'
              { yyerrok;
                printf( "Reenter last line: " ); }
              input
              { $$ = $4; }
              ;
```

As mentioned above, 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 lookahead 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 was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat : error
        { resynch();
          yyerrok ;
          yyclearin ; }
        ;
```

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.

9.9 The Yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C programs, called *y.tab.c* on most systems. 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 to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker 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 program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints 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 *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a **–ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string *syntax error*. The average application will want 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 lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program 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 which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9.10 Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

9.11 Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file.

1. Use uppercase letters for token names, lowercase letters for nonterminal names. This rule helps you to know who to blame when things go wrong.

2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

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 two tab stops, and action bodies by three tab stops.

The examples in the text of this section follow this style (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

9.12 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 ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list : item
     | list ',' item
     ;
```

and

```
seq : item
    | seq item
    ;
```

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 would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq : /* empty */
    | seq item
    ;
```

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!

9.13 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 this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
  int dflag;
%}
  ... other declarations ...

%%

prog    : decls  stats
        ;

decls   : /* empty */
                {       dflag = 1; }
        | decls declaration
        ;

stats   : /* empty */
                {       dflag = 0; }
        | stats statement
        ;

    ... other rules ...
```

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 over done. Nevertheless, it represents a way of doing some things that are difficult to do otherwise.

9.14 Handling Reserved Words

Some programming languages permit the user 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, but it is difficult. It is best that keywords be reserved; that is, be forbidden for use as variable names.

9.15 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*. *YYACCEPT* 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 endmarkers or context-sensitive syntax checking.

9.16 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, but in this case the digit may be 0 or negative. Consider

```
sent    : adj noun verb adj noun
              { look at the sentence ... }
        ;

adj     : THE   { $$ = THE; }
        | YOUNG { $$ = YOUNG; }
        ...
        ;

noun    : DOG   { $$ = DOG; }
        | CRONE { if( $0 == YOUNG ){
                      printf( "what?\n" );
                    }
                  $$ = CRONE;
                }
        ;
        ...
```

In the action following the word *CRONE*, a check is made 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 will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

9.17 Supporting 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 will be strictly type checked. The 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 to 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 will take place. In addition, type checking commands such as lint(C) will be 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 programs, 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 in the declaration section:

```
%union {
        body of union ...
        }
```

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. Alternatively, the union may be declared in a header file, and a typedef used to define the variable *YYSTYPE* to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

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> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr 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 predefined type. Similarly, reference to left context values (such as $0 – see the previous subsection) 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 $. An example of this usage is

```
rule : aaa { $<intval>$ = 3; } bbb
            { fun( $<intval>2, $<other>0 ); }
     ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in a later section. 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 *int*'s, as was true historically.

9.18 A Small Desk Calculator

This example gives the complete yacc specification for a small desk calculator: the desk calculator has 26 registers, labeled *a* through *z*, and accepts arithmetic expressions made up of the operators +, –, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, 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 precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase 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 the 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 /* precedence for unary minus */

%%    /* beginning of rules section */

list : /* empty */
     | list stat '\n'
     | list error '\n'
          { yyerrok; }
     ;

stat : expr
          { 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
          { $$ = $1 / $3; }
     | expr '%' expr
          { $$ = $1 % $3; }
     | expr '&' expr
          { $$ = $1 & $3; }
     | expr '|' expr
          { $$ = $1 | $3; }
```

```
        | '-' expr %prec UMINUS
            { $$ = - $2; }
        | LETTER
            { $$ = regs[$1]; }
        | number
        ;

number  : DIGIT
            { $$ = $1; base = ($1==0) ? 8 : 10; }
        | number DIGIT
            { $$ = base * $1 + $2; }
        ;

%%      /* start of programs */

yylex() {     /* lexical analysis routine  */
              /* returns LETTER for a lowercase letter, */
              /* yylval = 0 through 25  */
              /* return DIGIT for a digit, */
              /* yylval = 0 through 9  */
              /* all other characters */
              /* are returned immediately  */

        int c;

        while( (c=getchar()) == ' ' ) { /* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
              yylval = c - 'a';
              return ( LETTER );
              }
        if( isdigit( c ) ) {
              yylval = c - '0';
              return( DIGIT );
              }
        return( c );
        }
```

9.19 Yacc Input Syntax

This section has a description of the yacc input syntax, as a yacc specification. Context dependencies, etc., are not considered. Ironically, 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

on embedded in it. As implemented, the lexical analyzer looks ahead after
ng an identifier, and decide whether the next token (skipping blanks,
lines, comments, etc.) is a colon. If so, it returns the token
DENTIFIER. Otherwise, it returns *IDENTIFIER*. Literals (quoted
ngs) are also returned as *IDENTIFIER*, but never as part of
DENTIFIER.

```
    /* grammar for the input to Yacc */

  /* basic entities */
oken IDENTIFIER      /* includes identifiers and literals */
oken C_IDENTIFIER    /* identifier followed by colon      */
oken NUMBER          /* [0-9]+    */

  /* reserved words:  %type => TYPE, %left => LEFT, etc. */

oken LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

oken MARK   /* the %% mark */
oken LCURL  /* the %{ mark */
oken RCURL  /* the %} mark */

  /* ascii character literals stand for themselves */

art spec

;

:  : defs MARK rules tail
   ;

   : MARK   { Eat up the rest of the file }
   | /* empty: the second MARK is optional */
   ;

;  : /* empty */
   | defs def
   ;

   : START IDENTIFIER
   | UNION { Copy union definition to output }
   | LCURL { Copy C code to output file } RCURL
   | ndefs rword tag nlist
   ;

rd  : TOKEN
   | LEFT
   | RIGHT
   | NONASSOC
```

```
      | TYPE
      ;

tag   : /* empty: union tag is optional */
      | '<' IDENTIFIER '>'
      ;

nlist : nmno
      | nlist nmno
      | nlist ',' nmno
      ;

nmno  : IDENTIFIER        /* Literal illegal with %type */
      | IDENTIFIER NUMBER /* Illegal with %type */
      ;

      /* rules 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 ';'
      ;
```

9.20 An Advanced Example

This section gives an example of a grammar using some of the advanced features discussed in earlier sections. The desk calculator example is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, -, *, /, unary –, and = (assignment), and has 26 floating point variables, *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. 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 a double precision values. This structure is given a type name, *INTERVAL*, by using *typedef*. The yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. 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. In effect, 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 (e.g., scalar or interval) of intermediate expressions. Note that a 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 valued 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 circumvented 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. However, 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 conflicts 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, but not very general. 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 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   lines

%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 */

%%

lines  : /* empty */
       | lines line
       ;

line   : dexp '\n'
              { printf( "%15.8f\n", $1 ); }
       | vexp '\n'
              { 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 ){
                      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;
          $$.lo = $1.lo - $3.hi; }
    | dexp '-' vexp
        { $$.hi = $1 - $3.lo;
          $$.lo = $1 - $3.hi;}
    | vexp '*' vexp
        { $$ = vmul( $1.lo, $1.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;
          $$ = vdiv( $1, $1, $3 ); }
    | '-' vexp  %prec UMINUS
        { $$.hi = -$2.lo; $$.lo = -$2.hi; }
    | '(' vexp ')'
        {       $$ = $2; }
    ;

%%

# define BSZ 50  /* buffer size for fp numbers */

    /* lexical analysis */

yylex(){
    register  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 );
        }

    if( isdigit( c ) || c=='.' ){
```

```
        /* gobble up digits, points, exponents */

        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( '.' );
                                /* above causes syntax error */
                        continue;
                        }

                if (c == 'e' ) {
                        if ( exp++ ) return( 'e' );
                                /* above causes syntax error */
                        continue;
                        }

                /* end of number */
                break;
                }
        *cp = '\0';
        if( (cp-buf) >= BSZ )
                        printf( "constant too long:  truncated\n");
        else  ungetc( c, stdin );
                  /* above pushes 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 *, / routines */
    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. ){
        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 ) );
    }
```

9.21 Old Features

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotation marks (").

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or underscore, the type number of the literal is defined, just as if the literal did not have the quotation marks around it. Otherwise, it is difficult to find the value for such literals. The use of multicharacter literals is likely to mislead those unfamiliar with yacc, since it suggests that yacc is doing a job that must be actually done by the lexical analyzer.

3. Most places where '%' is legal, backslash (\) may be used. In particular, the double backslash (\\) is the same as *%%*, *\left* the same as *%left*, etc.

4. There are a number of other synonyms:

    %< is the same as %left
    %> is the same as %right
    %binary and %2 are the same as %nonassoc
    %0 and %term are the same as %token
    %= is the same as %prec

5. Actions may also have the form

   ={ ... }

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Appendix A

C Language Portability

A.1 Introduction

The standard definition of the C programming language leaves many details to be decided by individual implementations of the language. These unspecified features of the language detract from its portability and must be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences in either target machine hardware or compilers. C was designed to compile to efficient code for the target machine (initially a PDP-11) and so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This appendix highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment, which is determined by the system calls and library routines it uses during execution, file pathnames it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, from small 8-bit microprocessors to large mainframes. This appendix is concerned with the portability of C code in the XENIX programming environment. This is a more restricted problem to consider since all XENIX systems to date run on hardware with the following basic characteristics:

- ASCII character set
- 8-bit bytes
- 2-byte or 4-byte integers
- Two's complement arithmetic

These features are not formally defined for the language and may not be found in all implementations of C. However, the remainder of this appendix is devoted to those systems where these basic assumptions hold.

The C language definition contains no specification of how input and output is performed. This is left to system calls and library routines on individual systems. Within XENIX systems there are system calls and library routines that can be considered portable. These are described briefly in a later section.

This appendix is not intended as a C language primer. It is assumed that the reader is familiar with C, and with the basic architecture of common microprocessors.

A.2 Program Portability

A program is portable if it can be compiled and run successfully on different machines without alteration. There are many ways to write portable programs. The first is to avoid using inherently nonportable language features. The second is to isolate any nonportable interactions with the environment, such as I/O to nonstandard devices. For example programs should avoid hard-coding pathnames unless a pathname is common to all systems (e.g., */etc/passwd*).

Files required at compiletime (i.e., include files) may also introduce nonportability if the pathnames are not the same on all machines. In some cases include files containing machine parameters can be used to make the source code itself portable.

A.3 Machine Hardware

Differences in the hardware of the various target machines and differences in the corresponding C compilers cause the greatest number of portability problems. This section lists problems commonly encountered on XENIX systems.

A.3.1 Byte Length

By definition, the **char** data type in C must be large enough to hold as positive integers all members of a machine's character set. For the machines described in this appendix, the **char** size is exactly an 8 bit byte.

A.3.2 Word Length

In C, the size of the basic data types for a given implementation are not formally defined. Thus they often follow the most natural size for the underlying machine. It is safe to assume that **short** is no longer than **long**. Beyond that no assumptions are portable. For example on some machines **short** is the same length as **int**, whereas on others **long** is the same length as **int**.

Programs that need to know the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example the maximum positive integer (on a two's complement machine) can be obtained with:

```
#define MAXPOS          ((int)(((unsigned)0) >> 1))
```

This is preferable to something like:

```
#ifdef PDP11
#define MAXPOS 32767
#else
        ...
#endif
```

To find the number of bytes in an int use "sizeof(int)" rather than 2, 4, or some other nonportable constant.

A.3.3 Storage Alignment

The C language defines no particular layout for storage of data items relative to each other, or for storage of elements of structures or unions within the structure or union.

Some CPU's, such as the PDP-11 and M68000 require that data types longer than one byte be aligned on even byte address boundaries. Others, such as the 8086 and VAX-11 have no such hardware restriction. However, even with these machines, most compilers generate code that aligns words, structures, arrays, and long words on even addresses, or even long word addresses. Thus, on the VAX-11, the following code sequence gives "8", even though the VAX hardware can access an int (a 4-byte word) on any physical starting address:

```
struct s_tag {
        char c;
        int  i;
};
printf("%d\n",sizeof(struct s_tag));
```

The principal implications of this variation in data storage are that data accessed as nonprimitive data types is not portable, and code that makes use of knowledge of the layout on a particular machine is not portable.

Thus unions containing structures are nonportable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used simply to have different data in the same space at different times. For example, if the following union were used to obtain 4 bytes from a long word, the code would not be portable:

```
union {
        char c[4];
        long lw;
} u;
```

The *sizeof* operator should always be used when reading and writing structures:

```
struct s_tag st;

   ...

write(fd, &st, sizeof(st));
```

This ensures portability of the source code. It does not produce a portable data file. Portability of data is discussed in a later section.

Note that the *sizeof* operator returns the number of bytes an object would occupy in an array. Thus on machines where structures are always aligned to begin on a word boundary in memory, the *sizeof* operator will include any necessary padding for this in the return value, even if the padding occurs after all useful data in the structure. This occurs whether or not the argument is actually an array element.

A.3.4 Byte Order in a Word

The variation in byte order in a word affects the portability of data more than the portability of source code. However any program that makes use of knowledge of the internal byte order in a word is not portable. For example, on some systems there is an include file *misc.h* that contains the following structure declaration:

```
/*
* structure to access an
* integer in bytes
*/
struct {
        char    lobyte;
        char    hibyte;
};
```

With certain less restrictive compilers this could be used to access the high and low order bytes of an integer separately, and in a completely nonportable way. The correct way to do this is to use mask and shift operations to extract the required byte:

```
#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)
```

Note that even this operation is only applicable to machines with two bytes in an int.

One result of the byte ordering problem is that the following code sequence will not always perform as intended:

```
int c = 0;

read(fd, &c, 1);
```

On machines where the low order byte is stored first, the value of "c" will be the byte value read. On other machines the byte is read into some byte other than the low order one, and the value of "c" is different.

A.3.5 Bitfields

Bitfields are not implemented in all C compilers. When they are, no field may be larger than an **int**, and no field can overlap an **int** boundary. If necessary the compiler will leave gaps and move to the next **int** boundary.

The C language makes no guarantees about whether fields are assigned left to right, or right to left in an **int**. Thus, while bitfields may be useful for storing flags and other small data items, their use in unions to dissect bits from other data is definitely nonportable.

To ensure portability no individual field should exceed 16 bits.

A.3.6 Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers will not object to nonportable pointer operations. The *lint* program is particularly useful for detecting questionable pointer assignments and comparisons.

The common nonportable use of pointers is the use of casts to assign one pointer to another pointer of a different data type. This almost always makes some assumption about the internal byte ordering and layout of the data type, and is therefore nonportable. In the following code, the byte order in the given array is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

The *lint* program will issue warning messages about such uses of pointers. Code like this is very rarely necessary or valid. It is acceptable, however, when using the *malloc* function to allocate space for variables that do not have **char** type. The routine is declared as type **char** * and the return value is cast to the type to be stored in the allocated memory. If this type is not **char** * then *lint* will issue a warning concerning illegal type conversion. In addition, the *malloc* function is written to always return a starting address suitable for storing all types of data. *Lint* does not know this, so it gives a warning about possible data

alignment problems too. In the following example, *malloc* is used to obtain memory for an array of 50 integers.

```
extern char *malloc();
int *ip;

ip = (int *)malloc(50);
```

This example will attract a warning message from *lint*.

A.3.7 Address Space

The address space available to a program running under XENIX varies considerably from system to system. On a small PDP-11 there may be only 64K bytes available for program and data combined. Larger PDP-11's, and some 16 bit microprocessors allow 64K bytes of data, and 64K bytes of program text. Other machines may allow considerably more text, and possibly more data as well.

Large programs, or programs that require large data areas may have portability problems on small machines.

A.3.8 Character Set

The C language does not require the use of the ASCII character set. In fact, the only character set requirements are all characters must fit in the **char** data type, and all characters must have positive values.

In the ASCII character set, all characters have values between zero and 127. Thus they can all be represented in 7 bits, and on an 8-bits-per-byte machine are all positive, whether **char** is treated as signed or unsigned.

There is a set of macros defined under XENIX in the header file */usr/include/ctype.h* that should be used for most tests on character quantities. They provide insulation from the internal structure of the character set and, in most cases, their names are more meaningful than the equivalent line of code. Compare

```
if(isupper(c))
```

to

```
if((c >='A') && (c <='Z'))
```

With some of the other macros, such as *isdigit* to test for a hex digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an 'if' statement

A.4 Compiler Differences

There are a number of C compilers running under XENIX. On PDP-11 systems there is the so-called "Ritchie" compiler. Also on the 11, and on most other systems, there is the Portable C Compiler.

A.4.1 Signed/Unsigned char, Sign Extension

The current state of the signed versus unsigned **char** problem is best described as unsatisfactory.

The sign extension problem is a serious barrier to writing portable C, and the best solution at present is to write defensive code that does not rely on particular implementation features.

A.4.2 Shift Operations

The left shift operator, "**<<**" shifts its operand a number of bits left, filling vacated bits with zero. This is a so-called logical shift. The right shift operator, "**>>**" when applied to an unsigned quantity, performs a logical shift operation. When applied to a signed quantity, the vacated bits may be filled with zero (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code that uses knowledge of a particular implementation is nonportable.

The PDP-11 compilers use arithmetic right shift. To avoid sign extension it is necessary to shift and mask out the appropriate number of high order bits:

```
char c;

c = (c >> 3) & 0x1f;
```

You can also avoid sign extension by using using the divide operator:

```
char c;

c = c / 8;
```

A.4.3 Identifier Length

The use of long symbols and identifier names will cause portability problems with some compilers. To avoid these problems, a program should keep the following symbols as short as possible:

— C Preprocessor Symbols

— C Local Symbols

— C External Symbols

The loader used may also place a restriction on the number of unique characters in C external symbols.

Symbols unique in the first six characters are unique to most C language processors.

On some non-XENIX C implementations, uppercase and lowercase letters are not distinct in identifiers.

A.4.4 Register Variables

The number and type of register variables in a function depends on the machine hardware and the compiler. Excess and invalid register declarations are treated as nonregister declarations and should not cause a portability problem. On a PDP-11, up to three register declarations are significant, and they must be of type **int**, **char**, or pointer. While other machines and compilers may support declarations such as

```
register unsigned short
```

this should not be relied upon.

Since the compiler ignores excess variables of register type, the most important register type variables should be declared first. Thus, if any are ignored, they will be the least important ones.

A.4.5 Type Conversion

The C language has some rules for implicit type conversion; it also allows explicit type conversions by type casting. The most common portability problem in implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type **char** is compared with an **int**.

For example

```
char c;

if(c == 0x80)
        ...
```

will never evaluate true on a machine which sign extends since "c" is sign extended before the comparison with 0x80, an **int**.

The only safe comparison between **char** type and an **int** is the following:

```
char c;

if(c == 'x')
        ...
```

This is reliable because C guarantees all characters to be positive. The use of hard-coded octal constants is subject to sign extension. For example the following program prints "ff80" on a PDP-11:

```
main()
{
        printf("%x\n",'\200');
}
```

Type conversion also takes place when arguments are passed to functions. Types **char** and **short** become **int**. Machines that sign extend **char** can give surprises. For example the following program gives –128 on some machines:

```
char c = 128;
printf("%d\n",c);
```

This is because "c" is converted to **int** before passing o the function. The function itself has no knowledge of the original type of the argument, and is expecting an **int**. The correct way to handle this is to code defensively and allow for the possibility of sign extension:

```
char c = 128;
printf("%d\n", c & 0xff);
```

A.4.6 Functions With Variable Number of Arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is variable too. In such cases the code is dependent upon the size of various data types.

In XENIX there is an include file, */usr/include/varargs.h*, that contains macros for use in variable argument functions to access the arguments in a portable way:

```
typedef char *va_list;
#define va_dcl int va_alist;
#define va_start(list) list = (char *) &va_alist
#define va_end(list)
#define va_arg(list,mode) ((mode *)(list += sizeof(mode)))[-1]
```

The va_end() macro is not currently required. Use of the other macros will be

demonstrated by an example of the *fprintf* library routine. This has a first argument of type **FILE** *, and a second argument of type **char** *. Subsequent arguments are of unknown type and number at compilation time. They are determined at run time by the contents of the control string, argument 2.

The first few lines of *fprintf* to declare the arguments and find the output file and control string address could be:

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

int
fprintf(va_alist)
va_dcl;
{
        va_list ap;         /* pointer to arg list        */
        char *format;
        FILE *fp;

        va_start(ap);       /* initialize arg pointer */
        fp = va_arg(ap, (FILE *));
        format = va_arg(ap, (char *));

                ...
}
```

Note that there is just one argument declared to *fprintf*. This argument is declared by the va_dcl macro to be type int, although its actual type is unknown at compile time. The argument pointer "ap" is initialized by *va_start* to the address of the first argument. Successive arguments can be picked from the stack so long as their type is known using the *va_arg* macro. This has a type as its second argument, and this controls what data is removed from the stack, and how far the argument pointer "ap" is incremented. In *fprintf*, once the control string is found, the type of subsequent arguments is known and they can be accessed sequentially by repeated calls to va_arg(). For example, arguments of type **double**, **int** *, and **short**, could be retrieved as follows:

```
double dint;
int *ip;
short s;

dint = va_arg(ap, double);
ip = va_arg(ap, (int *));
s = va_arg(ap, short);
```

The use of these macros makes the code more portable, although it does assume a certain standard method of passing arguments on the stack. In particular no holes must be left by the compiler, and types smaller than **int** (e.g., **char**, and **short** on long word machines) must be declared as **int**.

A.4.7 Side Effects, Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression, or arguments to a function call. Thus

```
func(i++, i++);
```

is extremely nonportable, and even

```
func(i++);
```

is unwise if *func* is ever likely to be replaced by a macro, since the macro may use "i" more than once. There are certain XENIX macros commonly used in user programs; these are all guaranteed to use their argument once, and so can safely be called with a side-effect argument. The most common examples are *getc*, *putc*, *getchar*, and *putchar*.

Operands to the following operators are guaranteed to be evaluated left to right:

```
,       &&      ||      ?       :
```

Note that the comma operator here is a separator for two C statements. A list of items separated by commas in a declaration list is not guaranteed to be processed left to right. Thus the declaration

```
register int a, b, c, d;
```

on a PDP-11 where only three register variables may be declared could make any three of the four variables register type, depending on the compiler. The correct declaration is to decide the order of importance of the variables being register type, and then use separate declaration statements, since the order of processing of individual declaration statements is guaranteed to be sequential:

```
register int a;
register int b;
register int c;
register int d;
```

A.5 Program Environment Differences

Most programs make system calls and use library routines for various services. This section indicates some of those routines that are not always portable, and those that particularly aid portability.

We are concerned here primarily with portability under the XENIX operating system. Many of the XENIX system calls are specific to that particular operating system environment and are not present on all other operating

system implementations of C. Examples of this are *getpwent* for accessing entries in the XENIX password file, and *getenv* which is specific to the XENIX concept of a process' environment.

Any program containing hard-coded pathnames to files or directories, or user IDs, login names, terminal lines or other system dependent parameters is nonportable. These types of constant should be in header files, passed as command line arguments, obtained from the environment, or obtained by using the XENIX default parameter library routines *dfopen*, and *dfread*.

Within XENIX, most system calls and library routines are portable across different implementations and XENIX releases. However, a few routines have changed in their user interface. The XENIX library routines are usually portable among XENIX systems.

Note that the members of the printf family, *printf*, *fprintf*, *sprintf*, *sscanf*, and *scanf* have changed in several ways during the evolution of XENIX, and some features are not completely portable. The return values of these routines cannot be relied upon to have the same meaning on all systems. Some of the format conversion characters have changed their meanings, in particular those relating to uppercase and lowercase in the output of hexadecimal numbers, and the specification of **long** integers on 16-bit word machines. The reference manual page for *printf* contains the correct specification for these routines.

4.6 Portability of Data

Data files are almost always nonportable across different machine CPU architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

The only way achieve data file portability is to write and read data files as one dimensional character arrays. This avoids alignment and padding problems if the data is written and read as characters, and interpreted that way. Thus ASCII text files can usually be moved between different machine types without too many problems.

4.7 Lint

lint is a C program checker which attempts to detect features of a collection of C source files that are nonportable or even incorrect C. One particular advantage of *lint* over any compiler checking is that *lint* checks function declaration and usage across source files. Neither compiler nor loader do this.

lint will generate warning messages about nonportable pointer arithmetic, assignments, and type conversions. Passage unscathed through *lint* is not a guarantee that a program is completely portable.

A.8 Byte Ordering Summary

The following conventions are used in the tables below:

a0 The lowest physically addressed byte of the data item. a0 + 1, and so on.

b0 The least significant byte of the data item, 'b1' being the next least significant, and so on.

Note that any program that actually makes use of the following information is guaranteed to be nonportable!

Byte Ordering for Short Types

CPU	Byte Order	
	a0	a1
PDP-11	b0	b1
VAX-11	b0	b1
8086	b0	b1
286	b0	b1
M68000	b1	b0
Z8000	b1	b0

Byte Ordering for Long Types

CPU	Byte Order			
	a0	a1	a2	a3
PDP-11	b2	b3	b0	b1
VAX-11	b0	b1	b2	b3
8086	b2	b3	b0	b1
286	b2	b3	b0	b1
M68000	b3	b2	b1	b0
Z8000	b3	b2	b1	b0

Appendix B

M4: A Macro Processor

B.1 Introduction

The *m4* macro processor defines and processes specially defined strings of characters called macros. By defining a set of macros to be processed by *m4*, a programming language can be enhanced to make it:

- More structured
- More readable
- More appropriate for a particular application

The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor—replacement of text by other text.

Besides the straightforward replacement of one string of text by another, *m4* provides:

- Macros with arguments
- Conditional macro expansions
- Arithmetic expressions
- File manipulation facilities
- String processing functions

The basic operation of *m4* is copying its input to its output. As the input is read, each alphanumeric token (that is, string of letters and digits) is checked. If the token is the name of a macro, then the name of the macro is replaced by its defining text. The resulting string is reread by *m4*. Macros may also be called with arguments, in which case the arguments are collected and substituted in the right places in the defining text before *m4* rescans the text.

M4 provides a collection of about twenty built-in macros. In addition, the user can define new macros. Built-ins and user-defined macros work in exactly the same way, except that some of the built-in macros have side effects on the state of the process.

B.2 Invoking m4

The invocation syntax for *m4* is:

```
m4 [files]
```

Each file name argument is processed in order. If there are no arguments, or if

an argument is a dash (-), then the standard is read. The processed text is written to the standard output, and can be redirected as in the following example:

```
m4 file1 file2 - > outputfile
```

Note the use of the dash in the above example to indicate processing of the standard input, *after* the files *file1* and *file2* have been processed by *m4*.

B.3 Defining Macros

The primary built-in function of *m4* is **define**, which is used to define new macros. The input

```
define(name, stuff)
```

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter (the underscore (_) counts as a letter). *Stuff* is any text, including text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example

```
define(N, 100)
.
.
.
if (i > N)
```

defines "N" to be 100, and uses this symbolic constant in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by a left parenthesis, "(", it is assumed to have no arguments. This is the situation for "N" above; it is actually a macro with no arguments. Thus, when it is used, no parentheses are needed following its name.

You should also notice that a macro name is only recognized as such if it appears surrounded by nonalphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable "NNN" is absolutely unrelated to the defined macro "N", even though it contains three N's.

Things may be defined in terms of other things. For example

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if "N" is redefined? Or, to say it another way, is "M" defined as "N" or as 100? In *m4*, the latter is true, "M" is 100, so even if "N" subsequently changes, "M" does not.

This behavior arises because *m4* expands macro names into their defining text as soon as it possibly can. Here, that means that when the string "N" is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now "M" is defined to be the string "N", so when you ask for "M" later, you will always get the value of "N" at that time (because the "M" will be replaced by "N" which, in turn, will be replaced by 100).

B.4 Quoting

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by single quotation marks ` and ´ is not expanded immediately, but has the quotation marks stripped off. If you say

```
define(N, 100)
define(M, `N')
```

the quotation marks around the "N" are stripped off as the argument is being collected, but they have served their purpose, and "M" is defined as the string "N", not 100. The general rule is that *m4* always strips off one level of single quotation marks whenever it evaluates something. This is true even outside of macros. If you want the word "define" to appear in the output, you have to quote it in the input, as in

```
`define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining "N":

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the "N" in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by *m4*, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine "N", you must delay the evaluation by quoting:

```
define(N, 100)
...
define('N', 200)
```

In *m4*, it is often wise to quote the first argument of a macro.

If the forward and backward quotation marks (` and ') are not convenient for some reason, the quotation marks can be changed with the built-in **changequote**. For example:

```
changequote([, ])
```

makes the new quotation marks the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define**. The built-in **undefine** removes the definition of some macro or built-in:

```
undefine('N')
```

removes the definition of "N". Built-ins can be removed with **undefine**, as in

```
undefine('define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. For instance, pretend that either the word "xenix" or "unix" is defined according to a particular implementation of a program. To perform operations according to which system you have you might say:

```
ifdef('xenix', 'define(system,1)' )
ifdef('unix', 'define(system,2)' )
```

Don't forget the quotation marks in the above example.

Ifdef actually permits three arguments: if the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef('xenix', on XENIX, not on XENIX)
```

B.5 Using Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$*n*** will be replaced by the *n*th argument when the macro is actually used. Thus, the macro ***bump***, defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**.) Arguments that are not supplied are replaced by null strings, so we can define a macro *cat* which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

The arguments **$4** through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a,   b   c)
```

defines "a" to be "b c".

Arguments are separated by commas, but parentheses are counted properly, so a comma protected by parentheses does not terminate an argument. That is, in

```
define(a, (b,c))
```

there are only two arguments; the second is literally "(b,c)". And of course a bare comma or parenthesis can be inserted by quoting it.

B.6 Using Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers. The simplest is **incr**, which increments its numeric argument by 1. Thus, to handle the common programming situation where you want a variable to be defined as one more than N, write

```
define(N, 100)
define(N1, 'incr(N)')
```

Then "N1" is defined as one more than the current value of "N".

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the following operators (in decreasing order of precedence):

```
unary + and -
** or ^  (exponentiation)
* / %  (modulus)
+ -
== != < <= > >=
!       (not)
& or &&(logical and)
| or ||    (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is implementation dependent.

As a simple example, suppose we want "M" to be "2**N+1". Then

```
define(N, 3)
define(M, 'eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

B.7 Manipulating Files

You can include a new file in the input at any time by the built-in function **include**:

include(*filename*)

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** (for "silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of *m4* to temporary files during processing, and output the collected material upon command. *M4* maintains nine of these diversions, numbered 1 through 9. If you say

divert(n)

all subsequent output is put onto the end of a temporary file referred to as "n". Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

undivert

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is not the diverted stuff. Furthermore, the diverted material is not rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

B.8 Using System Commands

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function **mktemp**: a string of "XXXXX" in the argument is replaced by the process id of the current process.

B.9 Using Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If these are identical, **ifelse** returns the string *c*; otherwise it returns *d*. Thus, we might define a macro called **compare** which compares two strings and returns "yes" or "no" if they are the same or different.

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotation marks, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

```
ifelse(a, b, c, d, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise the result is *g*. If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is *c* if *a* matches *b*, and null otherwise.

B.10 Manipulating Strings

The built-in **len** returns the length of the string that makes up its argument. Thus

```
len(abcdef)
```

is 6, and

```
len((a,b))
```

is 5.

The built-in substr can be used to produce substrings of strings. For example

```
substr(s,i,n)
```

returns the substring of *s* that starts at position *i* (origin zero), and is *n* characters long. If *n* is omitted, the rest of the string is returned, so

```
substr('now is the time', 1)
```

is

```
ow is the time
```

If *i* or *n* are out of range, various sensible things happen.

The command

```
index(s1,s2)
```

returns the index (position) in *s1* where the string *s2* occurs, or –1 if it doesn't occur. As with substr, the origin for strings is 0.

The built-in transliterate performs character transliteration.

```
translit(s, f, t)
```

modifies *s* by replacing any character found in *f* by the corresponding character of *t*. That is

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that don't have an entry in *t* are deleted; as a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So

```
translit(s, aeiou)
```

deletes vowels from "s".

There is also a built-in called dnl which deletes all characters that follow it up to and including the next newline. It is useful mainly for throwing away empty lines that otherwise tend to clutter up *m4* output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add dnl to each of these lines, the newlines will disappear.

Another way to achieve this, is

```
divert(-1)
        define(...)
        ...
divert
```

B.11 Printing

The built-in errprint writes its arguments out on the standard error file. Thus, you can say

```
errprint('fatal error')
```

Dumpdef is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget the quotation marks.

ndex