TRS-XENIX SYSTEM

SOFTWARE DEVELOPMENT

ACKNOWLEDGEMENTS

This manual builds on the writing of many others. In many cases, the content here is identical, in whole or in part, to papers and manuals written at Bell Laboratories. In particular, Chapter 2 and Appendix B are adapted from papers written by Brian Kernighan and D.M. Ritchie. Chapters 5 and 10 are adapted from papers written by S.C. Johnson. Chapter 6 is derived from a paper by S.I. Feldman, Chapter 7 from a a paper by J.F. Marazano and S.R. Bourne, and Chapter 9 from a paper by M.E. Lesk and E. Schmidt. In addition, Appendix A is adapted from material written by Bill Joy and Mark Horton, while at the University of California at Berkeley. The work of those mentioned above, and countless others, is gratefully acknowledged.

CONTENTS

1.0 Introduction

1.1 Overview..... 1-1
1.2 Manual Organization..... 1-1
1.3 Notational Conventions..... 1-3

2.0 XENIX Programming

2.1 Introduction..... 2-1
2.2 The C Interface To The XENIX System..... 2-1
2.2.1 Program Arguments..... 2-1
2.2.2 The 2-2
2.2.3 The Standard I/O Library..... 2-4
2.2.4 Low-Level I/O..... 2-9
2.2.5 Processes..... 2-15
2.2.6 Signals and Interrupts..... 2-22
2.3 The Standard I/O Library..... 2-27
2.3.1 General Usage..... 2-27
2.3.2 File Access..... 2-29
2.3.3 File Status..... 2-33
2.3.4 Input Function..... 2-35
2.3.5 Output Functions..... 2-39
2.3.6 String Functions..... 2-44
2.3.7 Character Classification..... 2-47
2.3.8 Character Translation..... 2-49
2.3.9 Space Allocation..... 2-50
2.4 Include Files..... 2-52
2.4.1 ctype.h..... 2-52
2.4.2 signal.h..... 2-53
2.4.3 stdio.h..... 2-54
2.5 XENIX MC68000 Assembly Language Interface..... 2-55
2.5.1 Registers and Return Values..... 2-55
2.5.2 Calling Sequence..... 2-56
2.5.3 Stack Probes..... 2-57

3.0 Software Tools

3.1 Introduction..... 3-1
3.2 Basic Tools..... 3-1
3.3 Other Tools..... 3-2

4.0 Cc: A C Compiler

4.1 Introduction..... 4-1
4.2 Invocation Switches..... 4-2
4.3 The Loader..... 4-3

5.0 Lint: A C Program Checker

5.1 Introduction..... 5-1
5.2 A Word About Philosophy..... 5-2
5.3 Unused Variables and Functions..... 5-2
5.4 Set/Used Information..... 5-3
5.5 Flow of Control..... 5-4
5.6 Function Values..... 5-4
5.7 Type Checking..... 5-5
5.8 Type Casts..... 5-6
5.9 Nonportable Character Use..... 5-7
5.10 Assignments of longs to ints..... 5-7
5.11 Strange Constructions..... 5-8
5.12 History..... 5-9
5.13 Pointer Alignment..... 5-10
5.14 Multiple Uses and Side Effects..... 5-10
5.15 Shutting Lint Up..... 5-11
5.16 Library Declaration Files..... 5-12
5.17 Notes..... 5-13
5.18 Current Lint Options..... 5-14

6.0 ADB: A Program Debugger

6.1 Introduction..... 6-1
6.2 Invocation..... 6-1
6.3 The Current Address - Dot..... 6-2
6.4 Formats..... 6-3
6.5 General Request Meanings..... 6-3
6.6 Debugging C Programs..... 6-4
6.6.1 Debugging A Core Image 6-4
6.6.2 Multiple Functions..... 6-6
6.6.3 Setting Breakpoints..... 6-7
6.6.4 Other Breakpoint Facilities..... 6-9
6.7 Maps..... 6-10
6.8 Advanced Usage..... 6-11
6.8.1 Formatted dump..... 6-11
6.8.2 Directory Dump..... 6-14
6.8.3 Ilist Dump..... 6-14
6.8.4 Converting values..... 6-15
6.9 Patching..... 6-15
6.10 Notes..... 6-16
6.11 Figures..... 6-18
6.12 ADB Summ.ary..... 6-31

6.12.1 Format Summary..... 6-32
6.12.2 Expression Summary..... 6-32

7.0 Make: A Maintenance Program

7.1 Introduction..... 7-1
7.2 Description Files and Substitutions..... 7-5
7.3 Command Usage..... 7-7
7.4 Implicit Rules..... 7-8
7.5 Example..... 7-10
7.6 Suggestions and Warnings..... 7-11
7.7 Suffixes and Transformation Rules..... 7-13

8.0 As: An Assembler

8.1 Introduction..... 8-1
8.2 Invocation..... 8-1
8.3 Invocation Options..... 8-2
8.4 Source Program Format..... 8-3
8.4.1 Label Field..... 8-4
8.4.2 Opcode Field..... 8-4
8.4.3 Operand-Field..... 8-5
8.4.4 Comment Field..... 8-5
8.5 Symbols and Expressions..... 8-5
8.5.1 Symbols..... 8-5
8.5.2 Assembly Location Counter..... 8-8
8.5.3 Program Sections..... 8-9
8.5.4 Constants..... 8-9
8.5.5 Operators..... 8-11
8.5.6 Terms..... 8-12
8.5.7 Expressions..... 8-12
8.6 Instructions and Addressing Modes..... 8-13
8.6.1 Instruction Mnemonics..... 8-13
8.6.2 Operand Addressing Modes..... 8-14
8.7 Assembler Directives..... 8-17
8.7.1 .ascii .asciz..... 8-17
8.7.2 .blkb .blkw .blkl..... 8-18
8.7.3 .byte .word .long..... 8-19
8.7.4 .end..... 8-19
8.7.5 .text .data .bss..... 8-19
8.7.6 .globl .comm..... 8-20
8.7.7 .even..... 8-21
8.8 Operation Codes..... 8-22
8.9 Error Messages..... 8-23

9.0 Lex: A Lexical Analyzer

9.1 Introduction..... 9-1
9.2 Lex Source..... 9-3
9.3 Lex Regular Expressions..... 9-4
9.3.1 Character classes..... 9-5
9.3.2 Arbitrary character..... 9-6
9.3.3 Optional Expressions..... 9-6
9.3.4 Repeated Expressions..... 9-6
9.3.5 Alternation and Grouping..... 9-7
9.3.6 Context Sensitivity..... 9-7
9.3.7 Repetitions and Definitions..... 9-8
9.4 Lex Actions..... 9-9
9.5 Ambiguous Source Rules..... 9-13
9.6 Lex Source Definitions..... 9-16
9.7 Usage..... 9-17
9.8 Lex and Yacc..... 9-18
9.9 Left Context Sensitivity..... 9-22
9.10 Character Set..... 9-24
9.11 Summary of Source Format..... 9-25
9.12 Notes..... 9-27

10.0 YACC: A Compiler-Compiler

10.1 Introduction..... 10-1
10.2 Basic Specifications..... 10-4
10.3 Actions..... 10-6
10.4 Lexical Analysis..... 10-9
10.5 How the Parser Works..... 10-11
10.6 Ambiguity and Conflicts..... 10-17
10.7 Precedence..... 10-22
10.8 Error Handling..... 10-25
10.9 The Yacc Environment..... 10-27
10.10 Hints for Preparing Specifications..... 10-28
10.11 Advanced Topics..... 10-32
10.12 A Simple Example..... 10-35
10.13 Yacc Input Syntax..... 10-38
10.14 An Advanced Example..... 10-40
10.15 Old Features..... 10-47

Appendix A: The C Shell
Appendix B: M4
Appendix C: Portable C Programming

CHAPTER 1

INTRODUCTION

CONTENTS

1.1 Overview..... 1-1

1.2 Manual Organization..... 1-1

1.3 Notational Conventions..... 1-3

1.1 Overview

One of the primary uses of the XENIX system is as an environment for software development. This manual describes this programming environment and the available tools. Since nearly all of the XENIX system is written in the C programming language, C is the ideal language for creating new XENIX applications. However, no attempt is made here to teach C programming. For that, see the excellent tutorial and reference The C Programming Language, by Kernighan and Ritchie. For more information about the basic concepts and software that underly XENIX itself, see the XENIX Fundamentals manual.

1.2 Manual Organization

This manual is organized as follows:

CHAPTER 1: Introduction

The chapter you are now reading contains a word about the development of software on the XENIX system

CHAPTER 2: Xenix Programming

Discusses the standard XENIX environment and how this environment can be accessed either from C or from assembly language.

CHAPTER 3: Software Tools

Describes each of the tools that you are likely to use either directly or indirectly, in programming on the XENIX system, with emphasis on how the the software tools discussed in this manual fit together.

CHAPTER 4: Cc: The C Compiler

Describes use of the XENIX C compiler, cc. Also describes the preprocessing, linking, and assembly stages in compiling C programs to executable files.

CHAPTER 5: Lint: The C Program Checker

Describes use of **lint**, the XENIX C program checker. **Lint** analyzes C program syntax and language usage, reporting anomalies to the user.

CHAPTER 6: Make: A Program Maintainer

Describes use of make, a program for controlling software generation, update, and installation.

CHAPTER 7: ADB: A Program Debugger

Describes use of the debugger, ADB, a program for debugging and analyzing both programs while they execute.

CHAPTER 8: As: The XENIX Assembler

Describes how as, the XENIX assembler can be used to assemble machine language programs and routines.

CHAPTER 9: Lex: A Lexical Analyzer

Describes use of lex, a lexical analyzer useful in reading input languages.

CHAPTER 10: YACC: A Compiler-Compiler

Describes use of YACC, a complex utility for creating language translators. Useful in conjunction with lex, above.

APPENDIX A: The C Shell

Describes use of the alternate shell command interpreter, csh. The C shell command language has a syntax similar to that of the C programming language. Aliases and a command history mechanism are also provided.

APPENDIX B: M4

Describes use of the macro preprocessor, M4.

APPENDIX C: C Program Portability

Explains how to write C programs that are portable across different processors and XENIX systems.

1.3 Notational Conventions

Throughout this manual, the following notational conventions are used:

boldface — Command names are given in boldface in the text of this manual; no boldface occurs in displays, except in syntax specifications for literal text. For example, **ls**, **date**, and **cd** are all the names of commands that you might type at the keyboard, and therefore all are in bold. An exception to this rule occurs for long chapters about a single command. In this case, the command name is made less conspicuous by either underlining or capitalization.

underlining — All filenames and pathnames are underlined. For example, text.file is a filename and /usr/mary is a pathname. Most command arguments are underlined as well, although in some cases these are in boldface. Words and phrases also may be underlined for emphasis. References to entries in the XENIX Reference Manual are underlined and include a section number in parentheses. For example, ls(1) refers to the entry for the **ls** command in Section 1, "Commands".

[brackets] — Brackets enclose optional arguments in syntax specifications.

<angle-brackets> — Angle brackets enclose the names of control characters and special function keys. Examples are <CONTROL-D>, <CONTROL-S>, <RETURN>, <INTERRUPT>, and <BKSP>.

ellipses... — Ellipses are used to indicate one or more entries of an argument in a syntax specification. For example, in the following syntax for the **mail** command, the ellipses indicate that one or more persons can be sent mail:

mail person ...

quotation marks — Quotation marks are used to set off multiple keystroke input. For example,

"ls -la ; date" is an example of a command line appearing in the body of the text.

Common abbreviations for ASCII characters are listed below:

<ESC>	Escape, Control-[
<RETURN>	Carriage return, Control-M
<LF>	Newline, Linefeed, Control-J
<NL>	Newline, Linefeed, Control-J
<BKSP>	Backspace, Control-H
<TAB>	Tab, Control-I
<BELL>	Bell, Control-G
<FF>	Formfeed, Control-L
<SPACE>	Space, octal 040
<DEL>	Delete, octal 0177

CHAPTER 2

XENIX PROGRAMMING

CONTENTS

2.1 Introduction..... 2-1

2.2 The C Interface To The XENIX System..... 2-1
2.2.1 Program Arguments..... 2-1
2.2.2 The 2-2
2.2.3 The Standard I/O Library..... 2-4
2.2.4 Low-Level I/O..... 2-9
2.2.5 Processes..... 2-15
2.2.6 Signals and Interrupts..... 2-22

2.3 The Standard I/O Library..... 2-27
2.3.1 General Usage..... 2-27
2.3.2 File Access..... 2-29
2.3.3 File Status..... 2-33
2.3.4 Input Function..... 2-35
2.3.5 Output Functions..... 2-39
2.3.6 String Functions..... 2-44
2.3.7 Character Classification..... 2-47
2.3.8 Character Translation..... 2-49
2.3.9 Space Allocation..... 2-50

2.4 Include Files..... 2-52
2.4.1 ctype.h..... 2-52
2.4.2 signal.h..... 2-53
2.4.3 stdio.h..... 2-54

2.5 XENIX MC68000 Assembly Language Interface..... 2-55
2.5.1 Registers and Return Values..... 2-55
2.5.2 Calling Sequence..... 2-56
2.5.3 Stack Probes..... 2-57

2.1 Introduction

The C programming language is designed to be used in a computing environment. Because of the power and flexibility of the XENIX environment, it is important for the programmer to to take advantage of its many capabilities. For example, from within some C programs, you may want to execute other programs, or make calls to perform system functions. Or, you may want to write assembly language routines that interface to C programs. Before you can perform any of these programming tasks, you must know the environment. In the case of the XENIX system, this environment includes low-level system calls, available C libraries, and compiler calling conventions. Because you may also want to write C programs that are portable to other XENIX systems and other processors, a section in this chapter discusses portable C programming.

2.2 The C Interface To The XENIX System

This section shows how to interface C programs to the XENIX system, either directly or through the standard I/O library. The topics discussed include:

- Handling command arguments
- Rudimentary I/O
- The standard input and output
- The standard I/O library
- File system access
- Low-level I/O: open, read, write, close, seek
- Processes: exec, fork, pipes
- Signals and interrupts

2.2.1 Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function main as an argument count argc and an array argv of pointers to character strings that contain the arguments. By convention, argv[0] is the command name itself, so argc is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the echo command.)

```
main(argc, argv)        /* echo arguments */
int argc;
char *argv[];
{
        int i;

        for (i = 1; i < argc; i++)
                printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

argv is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by \0, so they can be treated as strings. The program starts by printing argv[1] and loops until it has printed them all.

The argument count and the arguments are parameters to main. If you want to keep them so other routines can get at them, you must copy them to external variables.

2.2.2 The Standard

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function getchar returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention:

```
prog <file
```

This causes prog to read file instead of the terminal. The program itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the "pipe" mechanism. For example

```
otherprog | prog
```

provides the standard input for prog from the standard output of otherprog.

Getchar returns the value EOF when it encounters the end of file (or an error) on whatever you are reading. The value of EOF is normally defined to be -1, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, putchar(c) puts the character c on the "standard output," which is also by default the terminal. The output can be captured on a file by using >. If prog uses putchar,

```
prog >outfile
```

writes the standard output on outfile instead of the terminal. Outfile is created if it doesn't exist; if it already exists, its previous contents are overwritten.

The function printf, which formats output in various ways, uses the same mechanism as putchar does, so calls to printf and putchar may be intermixed in any order: the output appears in the order of the calls.

Similarly, the function scanf provides for formatted input conversion; it reads the standard input and breaks it up into strings, numbers, etc., as desired. Scanf uses the same mechanism as getchar, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with getchar, putchar, scanf, and printf may be entirely adequate, and it is almost always enough to get started. This is particularly true if the XENIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ASCII control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()  /* ccstrip: strip non-graphic characters */
{
        int c;
        while ((c = getchar()) != EOF)
                if ((c >= ' ' && c < 0177) \ ||
                        c == '\t' || c == '\n')
                        putchar(c);
        exit(0);
}
```

The line

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

should appear at the beginning of each source file. It causes the C compiler to read a file (/usr/include/stdio.h) of standard routines and symbols that includes the definition of EOF.

If it is necessary to treat multiple files, you can use cat to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to exit at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Status returns are discussed later in more detail.

2.2.3 The Standard I/O Library

The Standard I/O Library is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. Section 2.3 contains a more complete description of its capabilities.

2.2.3.1 File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is not already connected to the program. One simple example is wc, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in x.c and y.c and the totals.

The question is how to arrange for the named files to be read-that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be opened by the standard library function fopen. Fopen takes an external name (like x.c or y.c), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a file pointer, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including stdio.h is a structure definition called FILE. The only declaration needed for a file pointer is exemplified by

```
FILE    *fp, *fopen();
```

This says that fp is a pointer to a FILE, and fopen returns a pointer to a FILE, which is a type name, like int, not a structure tag.

The actual call to fopen in a program is

```
fp = fopen(name, mode);
```

The first argument of fopen is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read (r), write (w), or append (a).

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, fopen returns the null pointer value NULL (which is defined as zero in stdio.h).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which getc and putc are the simplest. Getc returns the next character from a file. It needs the file pointer to tell it what file. Thus:

```
c = getc(fp)
```

places in c the next character from the file referred to by fp; it returns EOF when it reaches end of file. Putc is the inverse of getc. For example

```
putc(c, fp)
```

puts the character c on the file fp and returns c. Getc and putc return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called stdin, stdout, and stderr. Normally these are all connected to the terminal, but may be redirected to files or pipes. Stdin, stdout and stderr are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type

```
FILE *
```

can be. They are constants, however, not variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write **wc**. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order; if there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)        /* wc: count lines, words, chars */
int argc;
char *argv[];

{
int c, i, inword;
FILE *fp, *fopen();
long linect, wordct, charct;
long tlinect = 0, twordct = 0, tcharct = 0;

i = 1;
fp = stdin;
do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
                fprintf (stderr, "wc: can't open %s\n", argv[i]);
                continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
                charct++;
                if (c == '\n')
                        linect++;
                if (c == ' ' || c == '\t' || c == '\n')
                        inword = 0;
                else if (inword == 0) {
                        inword = 1;
                        wordct++;
                }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
} while (++i < argc);
if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
exit(0);
}
```

The function fprintf is identical to printf, save that the first argument is a file pointer that specifies the file to be written.

The function fclose is the inverse of fopen; it breaks the connection between the file pointer and the external name that was established by fopen, freeing the file pointer for another file. Since there is a limit on the number of files

that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call fclose on an output file-it flushes the buffer in which putc is collecting output. fclose(is called automatically for each open file when a program terminates normally.)

2.2.3.2 Error Handling-Stderr and Exit

Stderr is assigned to a program in the same way that stdin and stdout are. Output written on stderr appears on the user's terminal even if the standard output is redirected. **Wc** writes its diagnostics on stderr instead of stdout so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function exit to terminate program execution. The argument of exit is available to whatever process called it, so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

Exit itself calls fclose for each open output file, to flush out any buffered output, then calls a routine named _exit. The function _exit causes immediate termination without any buffer flushing; it may be called directly if desired.

2.2.3.3 Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with putc, etc., is buffered (except to stderr); to force it out immediately, use fflush(fp).

fscanf is identical to scanf, except that its first argument is a file pointer (as with fprintf) that specifies the file from which the input comes; it returns EOF at end of file.

The functions sscanf and sprintf are identical to fscanf and fprintf, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for sscanf and into it for sprintf.

fgets(buf, size, fp) copies the next line from fp, up to and including a newline, into buf; at most size-1 characters are copied; it returns NULL at end of file. fputs(buf, fp) writes the string in buf onto file fp.

The function ungetc(c, fp) "pushes back" the character c onto the input stream fp; a subsequent call to getc, fscanf, etc., will encounter c. Only one character of push-back per file is permitted.

2.2.4 Low-Level I/O

This section describes the bottom level of I/O on the XENIX system. The lowest level of I/O in XENIX provides neither buffering nor any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

2.2.4.1 File Descriptors

In the XENIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a file descriptor. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of and in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

File pointers are similar in concept to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O

without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

2.2.4.2 Read and Write

All input and output is done by two functions called read and write. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than n bytes remained to be read. (When the file is a terminal, read normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and -1 indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical block size on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program copies anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 512

main()  /* copy input to output */
{
        char    buf[BUFSIZE];
        int     n;

        while ((n = read(0, buf, BUFSIZE)) > 0)
                write(1, buf, n);
        exit(0);
}
```

If the file size is not a multiple of BUFSIZE, the last read will return a smaller number of bytes to be written by write; the next call to read after that will return zero.

It is instructive to see how read and write can be used to construct higher level routines like getchar, putchar, etc. For example, here is a version of getchar which does unbuffered input.

```
#define CMASK   0377    /* for making char's > 0 */

getchar()       /* unbuffered single character input */
{
        char c;

        return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c must be declared char, because read accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. (The constant 0377 is machine dependent and thus varies from machine to machine.)

The second version of getchar does input in big chunks, and hands out the characters one at a time.

```
#define CMASK   0377    /* for making char's > 0 */
#define BUFSIZE 512

getchar()       /* buffered version */
{
        static char     buf[BUFSIZE];
        static char     *bufp = buf;
        static int      n = 0;

        if (n == 0) {   /* buffer is empty */
                n = read(0, buf, BUFSIZE);
                bufp = buf;
        }
        return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

2.2.4.3 Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, open and creat [sic].

open is rather like the fopen discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an int.

```
int fd;

fd = open(name, rwmode);
```

As with fopen, the name argument is a character string corresponding to the external file name. The access mode argument is different, however: rwmode is 0 for read, 1 for write, and 2 for read and write access. open returns -1 if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to open a file that does not exist. The entry point creat is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called name, and -1 if not. If the file already exists, creat will truncate it to zero length; it is not an error to creat a file that already exists.

If the file is brand new, creat creates it with the protection mode specified by the pmode argument. In the

XENIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the XENIX utility **cp**, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)        /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
        int     f1, f2, n;
        char    buf[BUFSIZE];

        if (argc != 3)
                error("Usage: cp from to", NULL);
        if ((f1 = open(argv[1], 0)) == -1)
                error("cp: can't open %s", argv[1]);
        if ((f2 = creat(argv[2], PMODE)) == -1)
                error("cp: can't create %s", argv[2]);

        while ((n = read(f1, buf, BUFSIZE)) > 0)
                if (write(f2, buf, n) != n)
                        error("cp: write error", NULL);
        exit(0);
}

error(s1, s2)   /* print error message and die */
char *s1, *s2;
{
        printf(s1, s2);
        printf("\n");
        exit(1);
}
```

There is a limit (typically 20) on the number of files which a program may have open simultaneously. Therefore, any program which intends to process many files must be prepared to reuse file descriptors. The routine close breaks the

connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via exit or return from the main program closes all open files.

The following function removes the file filename from the file system:

```
unlink (filename)
```

2.2.4.4 Random Access-Seek and Lseek

File I/O is normally sequential: each read or write takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call lseek provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is fd to move to position offset, which is taken relative to the location specified by origin. Subsequent reading or writing will begin at that position. offset is a long; fd and origin are int's. origin can be 0, 1, or 2 to specify that offset is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"):

```
lseek(fd, 0L, 0);
```

Notice the 0L argument; it could also be written as

```
(long) 0
```

With lseek, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file:

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
        lseek(fd, pos, 0);      /* get to pos */
        return(read(fd, buf, n));
}
```

2.2.4.5 Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of -1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell errno. The meanings of the various error numbers are listed in the introduction to Section II of the XENIX Reference Manual, so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine perror will print a message associated with the value of errno; more generally, sys errno is an array of character strings which can be indexed by errno and printed by your program.

2.2.5 Processes

It is often easier to use a program written by someone else than to invent your own. This section describes how to execute a program from within another.

2.2.5.1 The System"

The easiest way to execute a program from another is to use the standard library routine system. System takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
        system("date");
        /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of sprintf may be useful.

Remember than getc and putc normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use fflush; for input, see setbuf in the appendix.

2.2.5.2 Low-Level Process Creation-Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's system routine is based on.

The most basic operation is to execute another program without returning, by using the routine execl. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to execl is the filename of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a NULL argument.

The execl call overlays the existing program with the new one, runs that, then exits. There is no return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an execl call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where date is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of execl called execv is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where argp is an array of pointers to the arguments; the

last pointer in the array must be NULL so execv can tell where the list ends. As with execl, filename is the file in which the program is found, and argp[0] is the name of the program. (This arrangement is identical to the argv array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories-you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [] in the argument list. If you want these, use execl to invoke the shell sh, which then does all the work. Construct a string commandline that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, /bin/sh. Its argument -c says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in commandline.

2.2.5.3 Control of Processes - Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with execl or execv. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called fork:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of proc id, the "process id." In one of these processes (the "child"), proc id is zero. In the other (the "parent"), proc id is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
        /* in child */
        execl("/bin/sh", "sh", "-c", cmd, NULL);
```

And in fact, except for handling errors, this is sufficient. The fork makes two copies of the program. In the child, the

value returned by fork is zero, so it calls execl which does the command and then dies. In the parent, fork returns non-zero so it skips the execl. (If there is any error, fork returns -1).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function wait:

```
int status;

if (fork() == 0)
        execl(\ ...\ );
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the execl or fork, or the possibility that there might be more than one child running simultaneously. (The wait returns the process id of the terminated child, if you want to check it against the value returned by fork.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in status). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The status returned by wait encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to exit which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither fork nor the exec calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the execl. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

2.2.5.4 Pipes

A pipe is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of ls to the standard input of pr. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call pipe creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int     fd[2];

stat = pipe(fd);
if (stat == -1)
        /* there was an error ... */
```

fd is an array of two file descriptors, where fd[0] is the read side of the pipe and fd[1] is for writing. These may be used in read, write and close calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent read will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called popen(cmd, mode), which creates a process cmd (just as system does), and returns a file descriptor that will either read or write that process, according to mode. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the pr command; subsequent write calls using the file descriptor fout will send their data to that process through the pipe.

popen first creates the the pipe with a pipe system call; it then forks to create two copies of itself. The child decides whether it is supposed to read or write, closes the

other side of the pipe, then calls the shell (via execl) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define READ    0
#define WRITE   1
#define tst(a, b)       (mode == READ ? (b) : (a))
static  int     popen_pid;

popen(cmd, mode)
char    *cmd;
int     mode;
{
        int p[2];

        if (pipe(p) < 0)
                return(NULL);
        if ((popen_pid = fork()) == 0) {
                close(tst(p[WRITE], p[READ]));
                close(tst(0, 1));
                dup(tst(p[READ], p[WRITE]));
                close(tst(p[READ], p[WRITE]));
                execl("/bin/sh", "sh", "-c", cmd, 0);
                /* disaster has occurred if we get here*/
                _exit(1);
        }
        if (popen_pid == -1)
                return(NULL);
        close(tst(p[READ], p[WRITE]));
        return(tst(p[WRITE], p[READ]));
}
```

The sequence of closes in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first close closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The close closes file descriptor 0, that is, the standard input. Dup is a system call that returns a duplicate of an already open file

descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the dup is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function pclose to close the pipe created by popen. The main reason for using a separate function rather than close is that it is desirable to wait for the termination of the child process. First, the return value from pclose indicates whether the process succeeded. Equally important is that only a finite number of unwaited-for children can exist for a given parent process, even if some of them have terminated. Performing the wait lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)      /* close pipe fd */
int fd;
{
        register r, (*hstat)(), (*istat)(), (*qstat)();
        int      status;
        extern int popen_pid;

        close(fd);
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        hstat = signal(SIGHUP, SIG_IGN);
        while ((r = wait(&status)) != popen_pid && r != -1);
        if (r == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        signal(SIGHUP, hstat);
        return(status);
}
```

The calls to signal make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable popen pid; it really should be an array indexed by file descriptor. A popen function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

2.2.6 Signals and Interrupts

This section is concerned with how to deal gracefully with program faults and with signals and interrupts from the outside world. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: interrupt, which is sent when the character is typed; quit, generated by the character; hangup, caused by hanging up the phone; and terminate, generated by the kill command. When one of these events occurs, the signal is sent to all processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the quit case, a core image file is written for debugging purposes.

The routine which alters the default action is called signal. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file signal.h gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
   ...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, signal returns the previous value of the signal. The second argument to signal may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean

up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
        int onintr();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, onintr);

        /* Process ... */

        exit(0);
}

onintr()
{
        unlink(tempfile);
        exit(1);
}
```

Why the test and the double call to signal? Recall that signals like interrupt are sent to all processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the onintr routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that signal returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>


main()
{
        int (*istat)(), onintr();

        istat = signal(SIGINT, SIG_IGN);
                /* save original status above*/
        setjmp(sjbuf);  /* save current stack position */
        if (istat != SIG_IGN)
                signal(SIGINT, onintr);

        /* main processing loop */
}

onintr()
{
        printf("\nInterrupt\n");
        longjmp(sjbuf); /* return to saved state */
}
```

The include file setjmp.h declares the type jmp_buf an object in which the state can be saved. Sjbuf is such an object; it is an array of some sort. The setjmp routine then saves the state of things. When an interrupt occurs, a call is forced to the onintr routine, which can print a message, set flags, or whatever. Longjmp takes as argument an object stored into by setjmp, and restores control to the location after the call to setjmp, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling exit or longjmp, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the

terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, wait, and pause.) A program whose onintr program just sets intflag, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
        if (intflag)
                /* EOF caused by interrupt */
        else
                /* true end-of-file */
```

One item to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
        execl( ... );
signal(SIGINT, SIG_IGN); /* ignore interrupts */
wait(&status);           /* until the child is done */
signal(SIGINT, onintr);  /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function system:

```
#include <signal.h>

system(s)       /* run command string s */
char *s;
{
        int status, pid, w;
        register int (*istat)(), (*qstat)();

        if ((pid = fork()) == 0) {
                execl("/bin/sh", "sh", "-c", s, 0);
                _exit(127);
        }
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        while ((w = wait(&status)) != pid && w != -1)
                ;
        if (w == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        return(status);
}
```

The function signal obviously has a rather strange second argument. This argument is a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values SIG_IGN and SIG_DFL have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions are sufficiently ugly and nonportable to encourage use of the standard include file:

```
#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1
```

2.3 The Standard I/O Library

A knowledge of the available C libraries is essential to the C programmer, since they defines a common set of macros, types, and functions that can be used in almost any programming project. The most important functions and macros are declared in the standard I/O library, which was was designed with the following goals in mind:

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.

2. It must be simple to use, and free of the magic numbers and mysterious calls whose use can reduce understandability and portability.

3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems.

2.3.1 General Usage

Each program using the library must have the line

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

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore (_) to reduce the possibility of conflict with other names created by the user. The names intended to be visible outside the package are:

stdin The name of the standard input file

stdout The name of the standard output file

stderr The name of the standard error file

EOF The value returned by the read routines on end-of-file or error; usually -1

NULL The null pointer, returned by pointer-valued functions to indicate an error

FILE The name of a macro useful when declaring pointers to streams. It expands to "**struct** _iob".

BUFSIZ The size (usually 512) size suitable for an I/O buffer supplied by the user. See setbuf, below.

Getc, getchar, putc, putchar, feof, ferror, and fileno are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions. Thus, they may not have breakpoints set on them when debugging.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names stdin, stdout, and stderr are in effect constants and may not be assigned to. Stdio.h contains the definitions of NULL, EOF, FILE, and BUFSIZ. The standard input file (stdin), standard output file (stdout), and standard error file (stderr) are also defined in the standard I/O library. These definitions can be incorporated into a C program with the following statement:

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

The file ctype.h provides the macro definitions for the possible character classifications. Any program using those facilities must contain the line:

```
#include <ctype.h>
```

The functions that handle signals need to use the signal definitions, so these definitions must be included if these functions are to be used. This can be done with the line:

```
#include <signal.h>
```

Some function names have changed in order to follow the established convention. To insure that the uniqueness of function names is preserved even if truncation occurs on some systems, those functions dealing with entire strings are named str...; those functions that consider only the first n characters of a string are named strn....

Listed below are some common C library functions. Most of these belong to the standard I/O library -- although other libraries are represented here as well.

2.3.2 File Access

fclose

```
#include <stdio.h>
int fclose(stream)
FILE *stream;
```

Fclose closes a file that was opened by fopen, frees any buffers after emptying them, and returns zero on success, nonzero on error. Exit calls fclose for all open files as part of its processing.

fdopen

```
#include <stdio.h>
FILE *fdopen (fildes, type)
int fildes;
char *type;
```

Fdopen provides a bridge between the low-level input-output (I/O) facilities of XENIX and the standard I/O functions. Fdopen associates a stream with a valid file descriptor obtained from a XENIX system call (e.g., open). "Type" is the same mode ("r", "w", "a", r+ , w+ , a+) that was used in the original creation of a file identified by "fildes". Fdopen returns a pointer to the associated stream, or NULL if unsuccessful.

Example:

```
int fd;
char *name = "myfile";
FILE *strm;

fd = open(name,0);

        .
        .
        .
if((strm = fdopen(fd,"r")) == NULL)
        fprintf(stderr,"Error on %d\n",fd);
```

fileno

```
#include <stdio.h>
int fileno (stream)
FILE *stream;
```

Implemented as a macro on XENIX, (and contained in the file stdio.h), fileno returns an integer file descriptor associated with a valid "stream". Any existing non-XENIX implementations may have different meanings for the integer which is returned. Fileno is used by many other standard functions in the C library.

fopen

```
#include <stdio.h>
FILE *fopen (filename, type)
char *filename, *type;
```

Fopen opens a file named "filename" and returns a pointer to a structure (hereafter referred to as "stream"), containing the data necessary to handle a stream of data. The "type" is one of the following character strings:

- r Used to open for reading.
- w Used to open for writing, which truncates an existing file to zero length or creates a new file.
- a Used to append, that is, open for writing at the end of a file, or create a new file.

For the update options, fseek or rewind can be used to trigger the change from reading to writing, or vice versa. (Reaching EOF on input will also permit writing without further formality.) Fopen returns a NULL pointer if "filename" cannot be opened. The update functions are particularly applicable to stream I/O and allow for the possibility of creating temporary files for both reading and writing.

Example:

```
FILE *fp;
char *file;

if((fp = fopen(file,"r")) == NULL)
        fprintf(stderr, "Cannot open %s\n",file);
```

freopen

```
#include <stdio.h>
FILE *freopen (newfile, type, stream)
char *newfile, *type;
FILE *stream;
```

Freopen accepts a pointer, "stream", to a previously opened file; the old file is closed, and then the new file is opened. The principal motivation for freopen is the desire to attach the names stdin, stdout, and stderr to specified files. On a successful freopen, the stream pointer is returned; otherwise NULL is returned, indicating that while the file closing took place, the reopening failed. Freopen is of limited portability; it cannot be implemented in all environments.

Example:

```
char *newfile;
FILE *nfile;

if((nfile = freopen(newfile,"r",stdout)) == NULL)
        fprintf(stderr,"Cannot reopen %s\n",newfile);
```

fseek

```
#include <stdio.h>
int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;
```

Fseek positions a stream to a location "offset" distance from the beginning, current position or end of a file, depending on the values 0, 1, 2 respectively for "ptrname". On XENIX the offset unit is bytes; other implementations are not necessarily the same. The return values are 0 on success and EOF on failure. Both buffered and unbuffered files may use fseek.

Example:

To position to the end of a file:

```
FILE *stream;

fseek(stream,0L,2);
```

pclose

```
#include <stdio.h>
int pclose (stream)
FILE *stream;
```

Pclose closes a stream opened by popen. It returns the exit status of the command that was issued as the first argument of its corresponding popen, or -1 if the stream was not opened by popen.

popen

```
#include <stdio.h>
FILE *popen (command, type)
char *command, *type;
```

Popen creates a pipe between the calling process and a command to be executed. The first argument is a shell command line; type is the I/O mode for the pipe, and may be either "r" for reading or "w" for writing. The function returns a stream pointer to be used for I/O on the standard input or output of the command. A NULL pointer is returned if an error occurs.

Example:

```
FILE *pstrm;

if((pstrm=popen("tr mvp MVP","w"))== NULL)
        fprintf(stderr,"popen error\n");
fprintf(pstrm,"a message via the pipe...\n");
if(pclose(pstrm) == -1)
        fprintf(stderr,"Pclose error\n");
```

results in:

```
a message via the pipe
```

rewind

```
#include <stdio.h>
int rewind(stream)
FILE *stream;
```

Rewind sets the position of the next operation at the beginning of the file associated with "stream", retaining the current mode of the file. It is the equivalent of fseek (stream,0L,0);.

setbuf

```
#include <stdio.h>
setbuf (stream, buf)
FILE *stream;
char *buf;
```

This function allows the user to choose his own buffer for I/O or choose no buffering at all. Use it after opening and before reading or writing. The function is often used to eliminate the single character writes to a file that result from the execution of putc to standard output that is not redirected. The choice to buffer I/O brings with it the responsibility for flushing any data that may remain in a last, partially-filled buffer. Fflush or fclose perform this task. The constant BUFSIZ in stdio.h tells how big the character array "buf" is. It is well-chosen for the machine on which XENIX is running. When "buf" is set to NULL, the I/O is completely unbuffered.

Example:

```
setbuf (stdout, malloc(BUFSIZ));
```

2.3.3 File Status

clearerr

```
#include <stdio.h>
clearerr(stream)
FILE *stream;
```

Clearerr resets the error condition on "stream". The need for clearerr arises in XENIX implementations where the error indicator is not reset after a query.

feof

```
#include <stdio.h>
int feof (stream)
FILE *stream;
```

Feof, which is implemented as a macro, returns nonzero if an input operation on "stream" has reached end of file; otherwise a zero is returned. Feof should be used in conjunction with any I/O function whose return value is not a clear indicator of an end-of-file condition. Such functions are fread and getw.

Example:

```
int *x;
FILE *stream;

do
        *x++ = getw(stream);
while(!feof(stream));
```

ferror

```
#include <stdio.h>
int ferror (stream)
FILE *stream;
```

Ferror tests for an indication of error on "stream". It returns a nonzero value (true) when an error is found, and a zero otherwise. Calls to ferror do not clear the error condition, hence the clearerr function is needed for that purpose. The user should be aware that, after an error, further use of the file may cause strange results. On XENIX ferror is implemented as a macro.

Example:

```
FILE *stream;
int *x;

while(!ferror(stream))
        putw(*x++,stream);
```

ftell

```
#include <stdio.h>
long ftell (stream)
FILE *stream;
```

Ftell determines the current offset relative to the beginning of the file associated with "stream". It returns the current value of the offset in bytes. On error, a value of -1 is returned. This function is useful in obtaining an offset for subsequent fseek calls.

2.3.4 Input Function

fgetc

```
#include <stdio.h>
int fgetc (stream)
FILE *stream;
```

This is the function version of the macro getc and acts identically to getc. Because fgetc is a function and not a macro, it can be used in debugging to set breakpoints on fgetc and when the side effects of macro processing of the argument is a problem. Furthermore, it can also be passed as an argument.

fgets

```
#include <stdio.h>
char *fgets (s,n,stream)
char *s;
int n;
FILE *stream;
```

Fgets reads from "stream" into the area pointed to by "s" either n-1 characters or an entire string including its newline terminator, whichever comes first. A final null character is affixed to the data read. Fgets returns the pointer "s" on success, and NULL on end-of-file or error. Fgets differs from the function gets in three ways: it can read from other than stdin; it appends the newline at the end of input when the size of the string is longer than or equal to "n"; and even more important, it provides control, not available with gets, over the size of the string to be read.

Example:

```
char msg[MAX];
FILE *myfile;

        while(fgets(msg,MAX,myfile) != NULL)
                printf("%s\n",msg);
```

fread

```
#include <stdio.h>
int fread((char *)ptr, sizeof (*ptr), nitems, stream)
FILE *stream;
```

This function reads from "stream" the next "nitems" whose size is the same as the size of the item pointed to by "ptr", into a sufficiently large area starting at "ptr". It returns the number of items read. In XENIX, fread makes use of the function getc. It is often used in combination with feof and ferror to obtain a clear indication of the file status.

Example:

```
FILE *pstm;
char mesg[100];

while(fread((char *)mesg,sizeof(*mesg),1,pstm) == 1)
        printf("%s\n",mesg);
```

fscanf

```
#include <stdio.h>
int fscanf (stream, format[, argptr]...)
char *format;
FILE *stream;
```

Fscanf accepts input from the file associated with "stream", and deposits it into the storage area pointed to by the respective argument pointers according to the specified formats. Fscanf differs from scanf in that it can read from other than stdin. The function returns the number of successfully handled input arguments, or EOF on end of input.

Example:

```
FILE *file;
long pay;
char name[15];
char pan[7];

fscanf(file,"%6s%14s%ld\n",pan,name,&pay);
if(pay<50000)
        printf("$%ld raise for %s.\n",pay/10,name);
```

If the input data is:

```
020202MaryJones 15000
```

the resulting output is:

$1500 raise for MaryJones.

getc

```
#include <stdio.h>
int getc (stream)
FILE *stream;
```

Getc returns the next character from the named "stream". It is implemented as a macro to avoid the overhead of a function call. On error or end-of-file it returns an EOF. Fgetc should be used if it is necessary to avoid the side effects of argument processing by the macro getc.

getchar

```
#include <stdio.h>
int getchar()
```

This is identical to getc (stdin).

gets

```
#include <stdio.h>
char *gets(s)
char *s;
```

Gets reads a string of characters up to a newline from stdin and places them in the area pointed to by "s". The newline character which ended the string is replaced by the null character. The return values are "s" on success, NULL on error or end-of-file. The simple example below presumes the size of the string read into "msg" will not exceed SIZE in length. If used in conjunction with strlen, a dangerous overflow can be detected, though not prevented.

Example:

```
char msg[SIZE];
char *s;
        s = msg;
        while (gets(s) != NULL)
                printf("%s\n",s);
```

getw

```
#include <stdio.h>
int getw (stream)
FILE *stream;
```

Getw reads the next word from the file associated with "stream". If successful, it returns the word; on error or end-of-file, it returns EOF. However, because EOF could be a valid word, this function is best used with feof and ferror.

Example:

```
FILE *stream;
int *x;
        do
                *x++ = getw(stream);
        while (!feof(stream));
```

scanf

```
#include <stdio.h>
int scanf (format[, argptr]...)
char *format;
```

Scanf reads input from stdin, delivers the input according to the specified formats, and deposits the input in the storage area pointed to by the respective argument pointers. For input from other streams than stdin use fscanf; for input from a character array use sscanf. Scanf returns the number of successfully handled input arguments, or EOF on end-of-input.

Example:

```
long number;

scanf("%ld",&number);
(printf(number%2?"%ld is odd":"%ld is even",number));
```

sscanf

```
#include <stdio.h>
sscanf (s, format [, pointer]...)
char *s;
char *format;
```

Sscanf accepts input from character string "s", delivers the input according to the specified formats, and deposits it into the storage area pointed to by the respective argument pointers. This function returns the number of successfully handled input arguments.

Example:

```
char datestr[] = {"THU MAR 29 11:04:40 EST 1983"};
char month[4];
char year[5];

sscanf(datestr,"%*3s%3s%*2s%*8s%*3s%4s",month,year);
printf("%s, %s\n",month,year);
```

The result is:

```
MAR, 1983
```

ungetc

```
#include <stdio.h>
int ungetc (c, stream)
int c;
FILE *stream;
```

Ungetc puts the character "c" back on the file associated with "stream". One character (but never EOF) is assured of being put back. If successful, the function returns "c", otherwise EOF.

Example:

```
while(isspace (c = getc(stdin)))
        ;
ungetc(c,stdin);
```

This code puts the first character that is not white space back onto the standard input stream.

2.3.5 Output Functions

fflush

```
#include <stdio.h>
int fflush (stream)
FILE *stream;
```

Fflush takes action to guarantee that any data contained in file buffers and not yet written out will be written. It is used by fclose to flush a stream. No action is taken on files not open for writing. The return values are zero for success, EOF on error.

fprintf

```
#include <stdio.h>
int fprintf (stream, format[, arg ]...)
FILE *stream;
char *format;
```

Fprintf provides formatted output to a named stream. The function printf may be used if the destination is stdout. Fprintf returns nonzero on error, otherwise zero.

Example:

```
int *filename;
int c;

if(c==EOF)
        fprintf(stderr,"EOF on %s\n",filename);
```

fputc

```
#include <stdio.h>
int fputc (c,stream)
int c;
FILE *stream;
```

Fputc performs the same task as putc; that is, it writes the character "c" to the file associated with "stream", but is implemented as a function rather than a macro. Fputc is preferred to putc when the side effects of macro processing of arguments are a problem. On success, it returns the character written; on failure it returns EOF.

Example:

```
FILE *in, *out;
int c;

        while ((c = fgetc(in)) != EOF)
                fputc(c,out);
```

fputs

```
#include <stdio.h>
int fputs(s,stream)
char *s;
FILE *stream;
```

Fputs copies a string to the output file associated with "stream", using the function putc

to do this. It is different from puts in two ways: fputs allows any output stream to be specified, and does not affix a newline to the output. For an example, see puts.

fwrite

```
#include <stdio.h>
int fwrite ((char *)ptr, sizeof (*ptr),nitems,stream)
FILE *stream;
```

Beginning at "ptr", this function writes up to "nitems" of data of the type pointed to by "ptr" into output "stream". It returns the number of items actually written. Like fread, this function should be used in conjunction with ferror to detect the error condition.

Example:

```
char mesg[] ={"My message to write out\n"};
FILE *pstrm;

if(fwrite(mesg,(sizeof(*mesg)-1),1,pstrm) != 1)
        fprintf(stderr,"Output error\n");
```

printf

```
#include <stdio.h>
int printf(format[, arg]...)
char *format;
```

Printf provides formatted output on stdout. Fprintf and sprintf are related functions that write output onto other than the standard output device. In case of error, implementations are not consistent in their output. On error, printf returns nonzero, otherwise zero. In later releases of the C library, printf returns the number of characters transmitted, or a negative value on error.

Example:

```
int num = 10;
char msg[] = {"ten"};
printf("%d - %o - %s\n", num, num, msg);
```

results in the line:

```
10 - 12 - ten;
```

putc

```
#include <stdio.h>
int putc (c,stream)
int c;
FILE *stream;
```

Putc writes the character c to the file associated with stream. On success, it returns the character written; on error it returns EOF. Because it is implemented as a macro, side effects may result from argument processing. In such cases, the equivalent function fputc should be used.

Example:

```
#define PROMPT()        putc('\7',stderr)
/* Prompt is BELL character */
```

putchar

```
#include <stdio.h>
int putchar(c)
int c;
```

Putchar is defined as:

```
putc (c, stdout)
```

Putchar returns the character written or EOF if an error occurs.

Example:

```
char *cp;
char x[SIZE];

        for(cp=x;cp<(x+SIZE);cp++)
                putchar(*cp);
```

puts

```
#include <stdio.h>
int puts(s)
char *s;
```

The function puts copies the string pointed to by "s" without its terminating null character to stdout. A newline character is appended. XENIX

uses the macro putchar (which calls putc).

Example:

```
puts("I will append a newline");
fputs("\tsome more data ", stdout);
puts("and now a newline");
```

The resulting output is:

```
I will append a newline
        some more data and now a newline
```

putw

```
#include <stdio.h>
int putw(w,stream)
FILE *stream;
int w;
```

Putw appends word "w" to the output "stream". As with getw, the proper way to check for an error or end-of-file is to use the feof and ferror functions.

Example:

```
int info;

while(!feof(stream))
        putw(info,stream);
```

sprintf

```
#include <stdio.h>
int sprintf(s, format, [, arg]...)
char *s;
char *format;
```

Sprintf allows formatted output to be placed in a character array pointed to by "s". Sprintf adds a null at the end of the formatted output. It is the user's responsibility to provide an array of sufficient length. The related functions printf and fprintf handle similar kinds of formatted output. The comparable input function is sscanf. On error, sprintf returns nonzero, otherwise zero.

Example:

```
char cmd[100];
char *doc = "/usr/src/cmd/cp.c"
int width = 50;
int length = 60;

sprintf(cmd,"pr -w%d -l%d %s\n",width,length,doc);
system(cmd);
```

The above code executes the pr command to print the source of the cp command.

2.3.6 String Functions

strcat

```
char *strcat(dst,src)
char *dst, *src;
```

Strcat appends characters in the string pointed to by "src" to the end of the string pointed to by "dst", and places a null character after the last character copied. It returns a pointer to "dst". To concatenate strings up to a maximum number of characters, use strncat.

Example:

```
char *myfile;
char dir[L_cuserid+5] = "/usr/";
        myfile = (strcat(dir,cuserid(0)));
```

The result is the concatenation of the login name onto the end of the string "dir".

strcmp

```
char *strcmp(s1,s2)
char *s1, *s2;
```

Strcmp compares the characters in the string "s1" and "s2". It returns an integer value, greater than, equal to, or less than zero, depending on whether "s1" is lexicographically greater than, equal to, or less than "s2".

Example:

```
#define EQ(x,y)  !strcmp(x,y)
```

strcpy

```
char *strcpy(dst, src)
char *dst, *src;
```

Strcpy copies the characters (including the null terminator) from the string pointed to by "src" into the string pointed to by "dst". A pointer to "dst" is returned.

Example:

```
char dst[] = "UPPER CASE";
char src[] = "this is lowercase";

        printf("%s\n",strcpy(dst,src+8));
```

results in:

```
lowercase
```

strlen

```
int strlen(s)
char *s;
```

Strlen counts the number of characters starting at the character pointed to by "s" up to, but not including, the first null character. It returns the integer count.

Example:

```
char nextitem[SIZE];
char series[MAX];

if(strlen(series)) strcat(series,",");
strcat(series,nextitem);
```

strncat

```
char *strncat(dst, src, n)
char *dst, *src;
int n;
```

Strncat appends a maximum of "n" characters of the string pointed to by "src" and then a null

character to the string pointed to by "dst". It returns a pointer to "dst".

Example:

```
char dst[] = "cover";
char src[] = "letter";

        printf("%s\n",strncat(dst,src,3));
```

The output is:

```
coverlet
```

strncmp

```
int strncmp(s1,s2,n)
char *s1, *s2;
int n;
```

Strncmp compares two strings for at most "n" characters and returns an integer value greater than, equal to, or less than zero depending on whether "s1" is lexicographically greater than, equal to or less than "s2".

Example:

```
char filename [] = "/dev/ttyx";

if(strncmp (filename+5, "tty",3) == 0)
        printf("success\n");
```

strncpy

```
char *strncpy(dst,src,n)
char *dst, *src;
int n;
```

Strncpy copies "n" characters of the string pointed to by "src" into the string pointed to by "dst". Null padding or truncation of "src" occurs as necessary. A pointer to "dst" is returned.

Example:

```
char buf [MAX];
char date [29] = {"Fri Dec 29 09:35:44 EDT 1982"};
char *day = buf;

        strncpy(day,date,3);
```

After executing this code, "day" points to the string "Fri".

2.3.7 Character Classification

isalnum

```
#include <ctype.h>
int isalnum(c)
int c;
```

This macro determines whether or not the character "c" is an alphanumeric character ([A-Za-z0-9]). It returns zero for false and nonzero for true.

isalpha

```
#include <ctype.h>
int isalpha(c)
int c;
```

This macro determines whether or not the character "c" is an alphabetic character ([A-Za-z]). It returns zero for false and nonzero for true.

isascii

```
#include <ctype.h>
int isascii(c)
int c;
```

This macro determines whether or not the integer value supplied is an ASCII character; that is, a character whose octal value ranges from 000 to 177. It returns zero for false and nonzero for true.

iscntrl

```
#include <ctype.h>
int iscntrl(c)
int c;
```

This macro determines whether or not the character "c" when mapped to ASCII is a control

character (that is, octal 177 or 000-037). It returns zero for false and nonzero for true.

isdigit

```
#include <ctype.h>
int isdigit(c)
int c;
```

This macro determines whether or not the character "c" is a digit. It returns zero for false and nonzero for true. (that is, is an ASCII code between octal 041 and 176 inclusive).

islower

```
#include <ctype.h>
int islower(c)
int c;
```

This macro determines whether or not the character "c" is a lowercase letter. It returns zero for false and nonzero for true.

isprint

```
#include <ctype.h>
int isprint(c)
int c;
```

This macro determines whether or not the character "c" is a printable character. (This includes spaces.) It returns zero for false and nonzero for true.

ispunct

```
#include <ctype.h>
int ispunct(c)
int c;
```

This macro determines whether or not the character "c" is a punctuation character (neither a control character nor an alphanumeric). It returns zero for false and nonzero for true.

isspace

```
#include <ctype.h>
int isspace(c)
int c;
```

This macro determines whether or not the character "c" is a form of white space (that is, a blank, horizontal or vertical tab, carriage return, form-feed or newline). It returns zero

for false and nonzero for true.

isupper

```
#include <ctype.h>
int isupper(c)
int c;
```

This macro determines whether or not the character "c" is an uppercase letter. It returns zero for false and nonzero for true.

2.3.8 Character Translation

toascii

```
#include <ctype.h>
int toascii (c)
int c;
```

The macro toascii usually does nothing: its purpose is to map the input character into its ASCII equivalent.

Example:

```
FILE *oddstrm;

        if(!isdigit (toascii(getw(oddstrm))))
                fprintf(stderr,"bad data\n");
```

tolower

```
#include <ctype.h>
int tolower (c)
int c;
```

If the argument "c" passed to the function tolower is an uppercase letter, the lowercase representation of "c" is returned, otherwise "c" is returned unchanged. For a faster routine, use _tolower, which is implemented as a macro; however, the argument must already be an uppercase letter.

Example:

```
if(tolower(getchar()) != 'y')
        exit(0);
```

toupper

```
#include <ctype.h>
int toupper (c)
int c;
```

If the argument "c" passed to the function toupper is a lowercase letter, the uppercase representation of "c" is returned, otherwise "c" is returned unchanged. For a faster routine, use _toupper, however, the argument must already be a lowercase letter.

Example:

```
if(toupper (getchar()) != 'Y')
        exit(0);
```

2.3.9 Space Allocation

calloc

```
char *calloc(n, size)
unsigned n, size;
```

Calloc allocates enough storage for an array of "n" items aligned for any use, each of "size" bytes. The space is initialized to zero. Calloc returns a pointer to the beginning of the allocated space, or a NULL pointer on failure.

Example:

```
char *t;
int n;
unsigned size;

        if(t=calloc((unsigned)n, size) == NULL)
                fprintf(stderr,"Out of space.\n");
```

free

```
free(ptr)
char *ptr;
```

Free is used in conjunction with the space allocating functions malloc, calloc, or realloc. "Ptr" is a pointer supplied by one of these routines. The function frees the space previously allocated.

malloc

```
char *malloc(size)
unsigned size;
```

Malloc allocates "size" bytes of storage beginning on a word boundary. It returns a pointer to the beginning of the allocated space, or a NULL pointer on failure to acquire space. For space initialized to zero, see calloc.

Example:

```
int n;
char *t;
unsigned size;

        if(t=malloc((unsigned)n) == NULL)
                fprintf(stderr,"Out of space.\n");
```

realloc

```
char *realloc (ptr, size)
char *ptr;
unsigned size;
```

Given "ptr" which was supplied by a call to malloc or calloc, and a new byte size, "size", realloc returns a pointer to the block of space of "size" bytes. This function is used to compact storage, and is used with the functions malloc and free.

2.4 Include Files

The following pages contain the contents of the three most important include files: ctype.h, stdio.h, and signal.s. These files are well worth some study, as the define a standard interface to the internals of the XENIX system.

2.4.1 ctype.h

```
#define _U      01
#define _L      02
#define _N      04
#define _S     010
#define _P     020
#define _C     040
#define _B    0100

extern  char    _ctype_[];

#define isalpha(c)      ((_ctype_+1)[c]&(_U|_L))
#define isupper(c)      ((_ctype_+1)[c]&_U)
#define islower(c)      ((_ctype_+1)[c]&_L)
#define isdigit(c)      ((_ctype_+1)[c]&_N)
#define isspace(c)      ((_ctype_+1)[c]&(_S|_B))
#define ispunct(c)      ((_ctype_+1)[c]&_P)
#define isalnum(c)      ((_ctype_+1)[c]&(_U|_L|_N))
#define isprint(c)      ((_ctype_+1)[c]&(_P|_U|_L|_N|_B))
#define iscntrl(c)      ((_ctype_+1)[c]&_C)
#define isascii(c)      ((unsigned)(c)<=0177)
#define _toupper(c)     ((c)-'a'+'A')
#define _tolower(c)     ((c)-'A'+'a')
#define toascii(c)      ((c)&0177)
```

2.4.2 signal.h

```
#define     NSIG        16

#define     SIGHUP       1   /* hangup */
#define     SIGINT       2   /* interrupt */
#define     SIGQUIT      3   /* quit */
#define     SIGILL       4   /* illegal instruction */
                             /* (not reset when caught) */
#define     SIGTRAP      5   /* trace trap */
                             /* (not reset when caught) */
#define     SIGIOT       6   /* IOT instruction */
#define     SIGEMT       7   /* EMT instruction */
#define     SIGFPE       8   /* floating point exception */
#define     SIGKILL      9   /* kill (cannot be */
                             /* (caught or ignored) */
#define     SIGBUS      10   /* bus error */
#define     SIGSEGV     11   /* segmentation violation */
#define     SIGSYS      12   /* bad argument to system call */
#define     SIGPIPE     13   /* write on a pipe */
                             /* with no one to read it */
#define     SIGALRM     14   /* alarm clock */
#define     SIGTERM     15   /* software termination */
                             /* signal from kill */

int         (*signal())();
#define     SIG_DFL     (int (*)())0
#define     SIG_IGN     (int (*)())1
```

2.4.3 stdio.h

```
#define         BUFSIZ          512
#define         _NFILE           20
# ifndef FILE
extern  struct  _iobuf {
        char    *_ptr;
        int     _cnt;
        char    *_base;
        char    _flag;
        char    _file;
} _iob[_NFILE];
# endif

#define         _IOREAD          01
#define         _IOWRT           02
#define         _IONBF           04
#define         _IOMYBUF        010
#define         _IOEOF          020
#define         _IOERR          040
#define         _IOSTRG        0100
#define         _IORW          0200

#define         NULL             0
#define         FILE            struct  _iobuf
#define         EOF             (-1)

#define         L_ctermid        9
#define         L_cuserid        9
#define         L_tmpnam        19

#define         stdin           (&_iob[0])
#define         stdout          (&_iob[1])
#define         stderr          (&_iob[2])
#define         getc(p)         (--(p)->_cnt>=0?\
                                *(p)->_ptr++&0377:_filbuf(p))
#define         getchar()       getc(stdin)
#define         putc(x,p)       (--(p)->_cnt>=0?\
                                ((int)(*(p)->_ptr++=(unsigned)(x))):\
                                _flsbuf((unsigned)(x),p))
#define         putchar(x)      putc(x,stdout)
#define         feof(p)         (((p)->_flag&_IOEOF)!=0)
#define         ferror(p)       (((p)->_flag&_IOERR)!=0)
#define         fileno(p)       p->_file

FILE    *fopen();
FILE    *freopen();
FILE    *fdopen();
long    ftell();
char    *fgets();
```

2.5 XENIX MC68000 Assembly Language Interface

The XENIX system is designed so that there should be little need to program in assembly language. Occasionally, however, the need does arise, and you may need to know the conventions for storing words in memory and for accessing parameters on the stack in a way compatible with the C runtime environment. Remember, however, that programming in assembly language is highly machine-dependent, and that you sacrifice portability whenever you forsake C for whatever low-level advantages you might gain.

If you do choose to mix MC68000 assembly language routines and compiled C routines, there are several things to be aware of:

- Registers and return values
- Calling sequence
- Stack probes

With an understanding of these three topics, you should be able to write both C programs that call MC68000 assembly language routines and assembly language routines that call compiled C routines.

2.5.1 Registers and Return Values

Function return values are passed in registers if possible. The set of machine registers used is called the save set, and includes the registers from d2-d7 and a2-a7 that are modified by a routine. The compiler assumes that these registers are preserved by the callee, and saves them itself when it is generating code for the callee. (When a C compatible routine is called by another routine, we refer to the calling routine as the caller. We refer to the called routine as the callee.) Note that a6 and a7 are in effect saved by a link instruction at procedure entry.

The function return value is in d0. The current floating point implementation returns the high order 32 bits of doubles in d1, and the low order 32 bits in d0. Functions return structure values (not pointers to the values) by loading d0 with a pointer to a static buffer containing the

structure value.

This makes the following two functions equivalent:

```
struct list proc (){
        struct list this;
        ...
        return (this);
}

struct list *proc (){
        struct list this;
        static struct list temp;
        ...
        temp = this;
        return (&temp);
}
```

This implementation allows recursive reentrancy (as long as the explicit form is not used, since the first sequence is indivisible but not the second). However, this implementation does not permit multitasking reentrancy. Note that the latter includes the XENIX signal(3) call.

Setjmp(3) and longjmp(3) cannot be implemented as they are on the PDP-11, because each procedure saves only the registers from the save set that it will modify. This makes it difficult to get back the current values of the register variables of the procedure that is being setjmped to. Hence, register variable values after a longjmp are the same as before a corresponding setjmp is called. If you need local variables to change between the call of setjmp and longjmp, they cannot be register variables.

2.5.2 Calling Sequence

The calling sequence is straightforward: arguments are pushed on the stack from the last to first: i.e., from right to left as you read them in the C source. The push quantum is 4 bytes, so if you are pushing a character, you must extend it appropriately before pushing. Structures and floating point numbers that are larger than 4 bytes are pushed in increments of 4 bytes so that they end up in the same order in stack memory as they are in any other memory. This means pushing the last word first and long-word padding the last word (the first pushed) if necessary. The caller is responsible for removing his own arguments. Typically, an

```
addql   #constant,sp
```

is done. It is not really important whether the caller actually pushes and pops his arguments or just stores them in a static area at the top of the stack, but the debugger, adb, examines the **addql** or **addw** from the sp to decide how many arguments there were.

2.5.3 Stack Probes

XENIX is designed to dynamically allocate space on the stack for local variables, function arguments, return addresses, and other information. When additional space is needed and an instruction causes a memory fault, the XENIX kernel checks the offending instruction. If the instruction is a stack reference, the kernel maps enough stack memory for the instruction to complete its execution successfully. Then the procedure continues execution where it left off. Generally, this means restarting the offending memory reference instruction (usually a push or store). The MC68000 does not provide a way to restart instructions; therefore, we need to perform a special instruction that can trigger the memory fault, but that has no ill effect other than triggering the fault. This instruction we call a stack probe.

When we perform a stack probe and a memory fault occurs, the kernel allocates additional memory for the stack. The XENIX kernel can allocate this needed stack memory, ignore the fact that the stack probe instruction did not complete, and then continue on to the next instruction.

The stack probe instruction for the MC68000 XENIX is

```
tstb -value(sp)
```

The value argument must be negative, because a negative index from the stack pointer is above the top of the stack. This is an otherwise absurd reference that XENIX recognizes as a stack probe.

For the general case, use the following procedure entry sequence:

```
procedure_entry:
        link    a6,#-savesize
        tstb    -pushsize-extra-8(sp)
```

Any registers among d2-d7 and a2-a5 that are used in this procedure are saved with a **moveml** instruction after this sequence. The number of registers saved in the **moveml** instruction needs to be accounted for in the push size. Thus, pushsize is the sum of the number of bytes pushed as

temporaries, save areas, and arguments by the whole procedure. The 8 bytes are the space for the return address and frame pointer save (by the link instruction) of a nested call. The extra is tolerance so that extremely short runtimes that use little stack do not need to perform a stack probe. The tolerance is intentionally kept small to conserve memory, so make sure you understand what you are doing before you consider leaving out a stack probe in your assembly procedures.

In most cases, unless you are pushing huge structures or doing tricks with the stack within your procedure, you can use the following instruction for your stack probe:

```
tstb    -100(sp)
```

This instruction makes sure that enough space has been allocated for most of the usual things you might do with the stack and is enough for the XENIX runtimes that do not perform stack probes. Note that you do not need to consider space allocated by the link instruction in this stack probe, since it is already added by indexing off the stack pointer.

CHAPTER 3

SOFTWARE TOOLS

CONTENTS

3.1 Introduction..... 3-1

3.2 Basic Tools..... 3-1

3.3 Other Tools..... 3-2

3.1 Introduction

This chapter discusses the tools available for use in software development on the XENIX System. The wide variety of available tools makes for a rich environment for programmers and software engineers. Often, tools are combined in shell procedures to perform whatever programming task desired. However, because their are so many different commands available to the programmer, it is often difficult to know what subset is especially useful in software development. To solve this problem, this chapter circumscribes a set of commands that are known to be of use in software development, and then summarizes the function of each command. In addition, this chapter contains a section describing the basic tools required in developing software on the XENIX system. Most of these basic tools have late chapters devoted to them.

3.2 Basic Tools

The tools used to create executable C programs are:

cc The XENIX C compiler.

lint The XENIX C program checker.

ld The XENIX loader.

as The XENIX assembler.

adb The XENIX debugger.

make The XENIX program maintainer.

Note that **cc** automatically invokes both the loader and the assembler so that use of either is optional. **Lint** is normally used in the early stages of program development to check for illegal and improper usage of the C language. The program **adb** is used to debug executable programs. The program **make** is used with the above tools to automatically maintain and regenerate software in medium scale programming projects.

All the above tools are used in creating executable C programs. These programs are created to run in the XENIX environment. This environment is manifested in the various subroutines and system calls available in several subroutine libraries.

Note that not all programming projects are best implemented in C, even if they are programs written for XENIX. Often, simple programs can be written in the shell command language much more quickly than they can be in C. For some complicated programs, **lex** and **yacc** may be just what is required. **Lex** is a lexical analyzer that can be used to accept a given input language. **Yacc** is a program designed to compile grammars into a parsing program. Typically, it is used to compile languages that are recognized by **lex**. For this reason, lex and **yacc** are often used together, although either can be used separately.

3.3 Other Tools

Other tools useful in software development are described below:

ar Archives files and maintains libraries. Useful when constructing or maintaining large object libraries.

at Execute commands at a later time. Used to execute time consuming compilations, printouts, and **makes**, so that they execute when the system isn't busy.

awk Recognizes text patterns and performs transformation operations based on the **awk** language.

basename Strips directory affixes and filename prefixes from a filename or pathname. Useful in shell scripts to obtain the filename part of a pathname or to strip off filename extensions.

cb Beautifies C programs, improving their readability. Note that the output from cb is not necessarily attractive to all programmers.

chgrp Changes the group affiliation of a file, so that it has the proper group permission when it is accessed or, if it is an executable file, when it is executed.

chown Changes the owner of a file, so that it has the proper owner when it is accessed or, if it is an executable file, when it is executed.

cmp Compares two sorted files. Useful when comparing object files and other binary images.

comm Compares sorted lists by either selecting or rejecting common lines.

copy Copies groups of files. Useful in recursively copying directories or in creating files that are linked to another set of files. Note that cp does not copy recursively.

csh Interprets and executes commands taken from the keyboard or from a command file. The "C shell" supports a C-like command language, an aliasing facility, and a command history mechanism.

ctags Creates a tags file so that C functions can be quickly found in a set of related C source files.

dd Converts and copies a file to the standard output. Used to read in files from various media. A variety of formats and conversions are available.

df Reports the amount of space that is free and available in a given file system.

diff Compares two text files, line by line.

diff3 Compares three text files, line by line.

du Summarizes disk usage. Used to determine how much disk space you are using.

file Determines the file type of given files Use this to examine files to determine whether they are directories, special files, or executable files.

find Finds filenames in a filesystem and optionally may perform commands that affect the found files. Used in performing complex operations on a selected set of files.

fsck Checks file system consistency and if possible, interactively repairs the file system when inconsistencies are found.

ln Creates a link of a file to another file so that file contents are shared and both filenames refer to the same file.

lorder Finds ordering relation for an object library.

m4	Processes input text performing several functions, including macro definition and invocation.
mkfs	Constructs a file system.
mknod	Creates a special file.
mkstr	Creates an error message file by examining a C source file.
mount	Mounts a file system on the given directory.
ncheck	Generate file names from inode numbers.
nice	Runs a command at a lower priority.
nm	Prints the list of names in a program.
od	Performs an "octal dump" of given files, printing files in a variety of formats, one of which is octal.
printenv	Prints out the environment.
prof	Displays profile data.
pstat	Prints system facts.
quot	Summarizes file system ownership.
ranlib	Converts archive libraries to random libraries by placing a table of contents at the front of each library.
settime	Change the access and modification dates of files.
size	Reports the size of an object file.
sleep	Suspends execution for a given period of time.
strings	Finds and prints readable text (strings) in an object or other binary file.
strip	Removes symbols and relocation bits from executable files.
su	Logs in user as super-user or other user.

sum	Computes check sum for a file and counts blocks. Useful in looking for bad spots in a file and in verifying transmission of data between systems.
sync	Updates the super block so that all input and output to the disk is completed before the **sync** command finishes.
tar	Archives files to tape or other similar device. Also used in moving large sets of files.
time	Times a given command. Used in taking benchmarks for execution-time of programs.
touch	Updates the modification date of a file without altering the contents of the file.
tr	Translates a one given set of characters to another set for all characters in a file.
tsort	Topologically sorts object libraries so that dependencies are apparent.
umount	Unmounts a mounted file system.
xstr	Extracts strings from C programs to implement shared strings.

CHAPTER 4

CC: A C COMPILER

CONTENTS

4.1 Introduction..... 4-1

4.2 Invocation Switches..... 4-2

4.3 The Loader..... 4-3

4.4 Files..... 4-5

4.1 Introduction

Cc is the command used to invoke the XENIX C compiler. Since the entire XENIX system is written in the C language, cc is the fundamental XENIX program development tool. The emphasis in this chapter is on giving insight into cc's operation and use. Special emphasis is given to input and output files and and to the available compiler options. Throughout, familiarity with compilers and with the C language is assumed. For more information on programming in C, see The C Programming Language, by Kernighan and Ritchie.

The fundamental function of the C compiler is to produce executable programs by processing C source files. The word "processing" is the key here, since the compilation process involves several distinct phases: These phases are described below:

Preprocessing

In this phase of compilation, your C source program is examined for macro definitions and include file directives. Any include files are processed at the point of the include statement; then occurrences of macros are expanded throughout the text. Normally, standard include files found in the /usr/include directory are included at the beginning of C programs. These standard include files normally are named with a ".h" extension. For example, the following statement includes the definitions for functions in the standard I/O library:

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

Note that the angle brackets indicate that the file is presumed to exist in /usr/include. The effects of preprocessing on a file can be captured in a file by specifying the -P switch on the cc command line. The -E switch performs a similar function useful for debugging when you suspect that an include file or macro is not expanding as desired.

Optimization

Optimization of generated code can be specified on the cc command line with the -O switch. This option should be used to increase execution speed or to decrease size of the executing program. Since programs will take longer to compile with this option, you may want to use this option only after you have a working debugged program.

Generation of Assembly Source Code

The C compiler generates assembly source code that is later assembled by the XENIX assembler, as. Cc's assembly output can be saved in a file by specifying the -S switch when the compiler is invoked. Assembly source output is saved in a file whose name has the ".s" extension.

Assembly

To assemble the generated assembly code, cc calls as to create a ".o" file. The ".o" file is used in the next step, linking and loading.

Linking and Loading

The final phase in the compilation of a C program is linking and loading. The program responsible for this is the XENIX loader, ld. Loader options can be specified on the cc command line. These options are discussed later in the section on the loader.

It is important to realize that all of the above phases can be controlled at the cc command level: they do not have to be invoked separately. What normally happens when you execute a cc command is that a sequence of programs processes the original C source file. Each program creates a temporary file that is used by the next program in the sequence. The final output is the load image that is loaded into memory when the final executable file is run.

4.2 Invocation Switches

A list of some of the available switches follows. Other switches may be described in cc(1S).

-c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

-p Arrange for the compiler to produce code which counts the number of times each routine is called. Also, if loading takes place, replace the standard startup routine by one that automatically calls monitor(3) at the start and arranges to write out a mon.out file at normal termination of execution of the object program. An execution profile can then be generated by use of prof(1).

-O Invoke an object-code optimizer.

-S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed ".s".

-P Run only the macro preprocessor and place the result for each ".c" file in a corresponding ".i" file. The resultant file has no "#" lines in it.

-o *output*
Give the final output file the name specified by *output*. If this option is used the file *a.out* will be left undisturbed.

-D*name*=*def*
Define the *name* to the compiler preprocessor, as if by "#define". If no definition is given, then *name* is assigned the value 1.

-U*name*
Remove any initial definition of *name*.

-I*dir*
Any "#include" files whose names do not begin with "/" and that are enclosed within angle brackets (< and >) are searched for first in the directory of the *file* argument, then in directories named in **-I** options, then in directories given by a standard list.

Other arguments are taken to be either loader option arguments, or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines created with the assembler. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with the name *a.out*.

Note that some versions of the C compiler support additional switches. These switches and their function are described in the reference section of this manual.

4.3 The Loader

As mentioned in the above sections, the XENIX loader, **ld**, plays a fundamental role in the development of any C program. For this reason it is discussed as part of **cc**; it can however, be used as a stand-alone processor of object files. Note that arguments to **ld** can be given on the **cc**

command line and are a part of the syntax of the cc command.

Some of the available loader switches are listed below. Except for -l, they should appear before filename arguments. Other switches are described in ld(1S).

-s "Strip" the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by strip(1S).

-u Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-lx This option is an abbreviation for the library name /lib/libx.a, where x is a string. If that does not exist, ld tries /usr/lib/libx.a. A library is searched when its name is encountered, so the placement of a -l is significant.

-x Do not preserve local (non-.globl) symbols in the output symbol table: enter only external symbols. This option saves some space in the output file.

-X Save local symbols except for those whose names begin with "L". This option is used by cc to discard internally generated labels while retaining symbols local to routines.

-n Arrange that when the output file is executed, the text portion will be read-only and shared among all users executing the file.

-i When the output file is executed, the program text and data areas will live in separate address spaces. The only difference between this option and -n is that here the data starts at location 0.

-o The name argument after -o is used as the name of the ld output file, instead of a.out.

For more information on the loader, see ld in the reference section of this manual.

4.4 Files

The files making up the compiler, as well as those files needed, used, or created by cc are listed below:

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporaries for cc
/lib/cpp	preprocessor
/lib/c[01]	compiler for cc
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/lib/libc.a	standard library
/usr/include	standard directory for "#include" files

CHAPTER 5

LINT: A C PROGRAM CHECKER

CONTENTS

5.1 Introduction..... 5-1

5.2 A Word About Philosophy..... 5-2

5.3 Unused Variables and Functions..... 5-2

5.4 Set/Used Information..... 5-3

5.5 Flow of Control..... 5-4

5.6 Function Values..... 5-4

5.7 Type Checking..... 5-5

5.8 Type Casts..... 5-6

5.9 Nonportable Character Use..... 5-7

5.10 Assignments of longs to ints..... 5-7

5.11 Strange Constructions..... 5-8

5.12 History..... 5-9

5.13 Pointer Alignment..... 5-10

5.14 Multiple Uses and Side Effects..... 5-10

5.15 Shutting Lint Up..... 5-11

5.16 Library Declaration Files..... 5-12

5.17 Notes..... 5-13

5.18 Current Lint Options..... 5-14

5.1 Introduction

Lint is a program that examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

The separation of function between Lint and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not perform sophisticated type checking, especially between separately compiled programs. Lint takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This section discusses the use of Lint, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

Suppose there are two C source files, file1.c and file2.c, which are ordinarily compiled and loaded together. Then the command

```
lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint -p file1.c file2.c
```

produces, in addition to the above messages, additional messages that relate to the portability of the programs to other operating systems and machines. Replacing the **-p** by **-h** produces messages about various error-prone or wasteful constructions that, strictly speaking, are not bugs. Saying **-hp** gets the whole works.

The next several sections describe the major messages; the discussion of Lint closes with sections discussing the implementation and giving suggestions for writing portable C. The final section gives a summary of Lint command line options.

5.2 A Word About Philosophy

Many of the facts about a particular C program that Lint needs may be impossible for it to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether exit is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the Lint algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, Lint assumes it can be called: this is not necessarily so, but in practice it is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form "xxx might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages that Lint produces.

5.3 Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs. If a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions that are defined but not otherwise mentioned. An exception is made for variables that are declared through explicit **extern** statements but are never referenced. Thus, the statement

```
extern float sin();
```

will evoke no comment if sin is never used. This agrees with the semantics of the C compiler.

In some cases, these unused external declarations might be of some interest: they can be discovered by adding the **-x** flag to the Lint invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **-v** option is available to suppress the printing of complaints about unused arguments. When **-v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when Lint is applied to some, but not all, files out of a collection that are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The **-u** flag may be used to suppress the spurious messages that might otherwise appear.

5.4 Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well: many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. Lint detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that Lint can complain about some legal programs, but these programs would probably be considered bad on stylistic grounds (for example, they might contain at least two **goto**'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

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

5.5 Flow of Control

Lint attempts to detect unreachable portions of program code. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops that can never be left at the bottom, detecting the special cases **while(1)** and **for(;;)** as infinite loops. Lint also complains about loops which cannot be entered at the top: some valid programs may have such loops, but at best they are bad style and at worst, bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to exit may cause unreachable code which Lint does not detect; the most serious effects of this are in the determination of returned function values, discussed in the next section.

One form of unreachable statement is not usually complained about by Lint: a **break** statement that cannot be reached causes no message. Programs generated by yacc and especially lex may have literally hundreds of unreachable **break** statements. Using the **-O** switch with the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the Lint output. If these messages are desired, Lint can be invoked with the **-b** option.

5.6 Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function "values" which have never been returned. Lint addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return ( expr );
```

and

```
return ;
```

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

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

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

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

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

In practice, some potentially serious bugs have been discovered by this feature. It also accounts for a substantial fraction of the "noise" messages produced by Lint.

On a global scale, Lint detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

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

5.7 Type Checking

Lint enforces the type checking rules of C more strictly than do the compilers. The additional checking is 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 the 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**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

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

5.8 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 quite rightly complains. Now, 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. It seems harsh for Lint to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **-c** flag controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to complaint. Otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

5.9 Nonportable Character Use

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

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

works on some machines, but will fail on machines where characters always take on positive values. The real 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 because on some machines bitfields are considered as signed quantities. While it may seem counter-intuitive to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

5.10 Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use typedefs. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to integer values, losing accuracy. Since there are a number of legitimate reasons for assigning longs to integers, the detection of these assignments is enabled with the **-a** flag.

5.11 Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by Lint. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **-h** flag is used to enable these checks. For example, in the statement

```
*p++ ;
```

the star `*' does nothing. This provokes the message "null effect" from Lint. The program fragment

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

is clearly somewhat strange. 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 one says

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

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 which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

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

or

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

probably do not do what is intended. The best solution is to parenthesize such expressions, and Lint encourages this

by an appropriate message.

Finally, when the **-h** flag is in force, Lint complains about variables which 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.

5.12 History

There are several forms of older syntax that are discouraged by Lint . These fall into two classes, assignment operators and initialization.

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

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, -=, etc.) have no such ambiguities. To spur the abandonment of the older forms, Lint complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize x to 1. This also caused syntactic difficulties. For example

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

looks somewhat like the beginning of a function declaration:

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

and the compiler must read a fair ways past x in order to sure what the declaration really is. Again, the problem is even more perplexing when the initializer involves a macro.

The current syntax places an equals sign between the variable and the initializer:

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

This is free of any possible syntactic ambiguity.

5.13 Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely 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 others, 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 whenever either the **-p** or **-h** flags are in effect.

5.14 Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which 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 local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

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

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

will draw the complaint:

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

5.15 Shutting Lint Up

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, etc. Moreover, as specified above, the flow of control information produced by Lint often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with Lint, typically to shut it up, is desirable. Therefore, a number of words are recognized by Lint when they were embedded in comments. Thus, Lint directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the Lint directives don't work.

The first directive is concerned with flow of control information. If a particular place in the program cannot be reached, but this is not apparent to Lint, this can be asserted at the appropriate spot in the program by the directive:

```
/* NOTREACHED */
```

Similarly, if it is desired 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 flag can be turned on for one function by the directive:

```
/* ARGSUSED */
```

Complaints about 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. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments that should be

checked. Thus:

```
/* VARARGS2 */
```

causes the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

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

5.16 Library Declaration Files

Lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This 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 almost exactly like ordinary source files. The only difference is that functions that are defined on a library file, but are not used on a source file, draw no complaints. Lint does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

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

5.17 Notes

Lint is by no means perfect. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked; and no attempt is made to match up structure and union declarations across files.

5.18 Current Lint Options

The command currently has the form

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

The options are

- **h** Perform heuristic checks
- **p** Perform portability checks
- **v** Don't report unused arguments
- **u** Don't report unused or undefined externals
- **b** Report unreachable **break** statements.
- **x** Report unused external declarations
- **a** Report assignments of **long** to **int** or shorter.
- **c** Complain about questionable casts
- **n** No library checking is done
- **s** Same as **h** (for historical reasons)

CHAPTER 6

ADB: A PROGRAM DEBUGGER

CONTENTS

6.1 Introduction..... 6-1

6.2 Invocation..... 6-1

6.3 The Current Address - Dot..... 6-2

6.4 Formats..... 6-3

6.5 General Request Meanings..... 6-3

6.6 Debugging C Programs..... 6-4
- 6.6.1 Debugging A Core Image 6-4
- 6.6.2 Multiple Functions..... 6-6
- 6.6.3 Setting Breakpoints..... 6-7
- 6.6.4 Other Breakpoint Facilities..... 6-9

6.7 Maps..... 6-10

6.8 Advanced Usage..... 6-11
- 6.8.1 Formatted dump..... 6-11
- 6.8.2 Directory Dump..... 6-14
- 6.8.3 Ilist Dump..... 6-14
- 6.8.4 Converting values..... 6-15

6.9 Patching..... 6-15

6.10 Notes..... 6-16

6.11 Figures..... 6-18

6.12 ADB Summary..... 6-31
- 6.12.1 Format Summary..... 6-32
- 6.12.2 Expression Summary..... 6-32

6.1 Introduction

ADB is an indispensable tool for debugging programs or crashed systems. ADB provides capabilities 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 provides 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

Here 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(1). 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 from 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 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, a user specifies 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:

Letter	Format
b	one byte in octal
c	one byte as a character
o	one word in octal
d	one word in decimal
x	one word in hexadecimal
f	two words in floating point
i	machine instruction
s	a null terminated character string
a	the value of dot
u	one word as unsigned integer
n	print a newline
r	print a blank space
^	backup dot

(Format letters are also available for "long" values, for example, D for long decimal, and F for double floating point.)

6.5 General Request Meanings

The general form of a request is:

```
address,count command modifier
```

which sets "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.

6.6 Debugging C Programs

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

6.6.1 Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case 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.

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 0xlfff90 respectively. Both of these values look reasonable; 0x2 = two arguments, 0xlfff90 = 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. More about these maps later.

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 says 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. Using ADB similarly, we could print information about the arguments to a function.

```
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 double quote mark (") means ditto, i.e., the 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 (not its contents) in hex which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

```
.-10/d
```

6.6.2 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 you can enter the debugger via:

```
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 (typing <DEL> will terminate the output and bring you back to ADB request level. Additionally, some versions, will automatically quit after 15 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 two word format D.

ADB

6.6.3 Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from Software Tools by Kernighan and Plauger, pp. 18-27.

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 a 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 since 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 one types:

```
$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.

Notice 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 one 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
```

to display a stack trace, or:

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

to print 6 lines of 4 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.

6.6.4 Other Breakpoint Facilities

- Arguments and change 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 entered at a specific address by typing:

  ```
  address:r
  ```

- The count field can be used to skip the first n breakpoints as:

  ```
  ,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
```

6.7 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 is generated by a C compiler invocation such as:

```
cc pgm.c
```

A shared file is produced by a C compiler command 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. The "?*" 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 02000 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 (407,410,411)
```

In Figure 7 those variables not present are zero. Use can be made of these variables by 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
```

that 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.

6.8 Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

6.8.1 Formatted dump

The line

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

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down,

the various 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 broken down as follows:

4o	Print 4 octal locations.
4^	Backup the current address 4 locations (to the original start of the field).
8C	Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as an at-sign (@) followed by the corresponding character in the range 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 the 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 as:

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

to read in a script file, dump, of requests. 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 of the form:

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

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

```
$v
```

prints all non-zero 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 octal 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.

6.8.2 Directory Dump

As another illustration (Figure 10) consider a 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, the u prints the inumber as an unsigned decimal integer, the "8t" means that ADB will space to the next multiple of 8 on the output line, and the "14c" prints the 14 character file name.

6.8.3 Ilist Dump

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

```
adb /dev/src -
02000>b
?m <b
<b,-1?"flags"8ton"links,uid,gid"8t3bn",...
```

(Note that the two lines separated by ellipses should be entered as one line with no intervening space. The line is broken here so that it will fit on the page.) 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. "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.

6.8.4 Converting values

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

```
072 = odx
```

prints

```
072     58      #3a
```

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

prints

```
a       0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6.9 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. In general, the request syntax for l and w are similar:

```
?l value
```

The request l is used to match on two bytes; L is used for four bytes. The request w is used to write two bytes, whereas W writes four 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 as:

```
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 a.out file. The form "?*" would have been used for a 411 file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

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 a.out -
:s argl 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 a.out as a subprocess with arguments argl and arg2. If there is a subprocess running ADB writes to it rather than to the file so the w request causes flag to be changed in the memory of the subprocess.

6.10 Notes

Below is a list of some things that users should be aware of:

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.

2. When printing addresses, ADB uses either text or data symbols from the a.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 for text (instructions) and slash (/) for data.

3. Local variables cannot be addressed.

6.11 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 argc;
char **argv;
{
        char    cc;
        FILE *file;

        if(argc < 2) {
                printf("Input file missing\n");
                exit(8);
        }

        if((file = fopen(argv[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);
}
```

Figure 2: ADB output for C program of figure 1

```
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   `a.out'
b1 = 0x8000     e1 = 0x970C     f1 = 0x20
b2 = 0x8000     e2 = 0x970C     f2 = 0x20
/ map   `-'
b1 = 0x0        e1 = 0x1000000  f1 = 0x0
b2 = 0x0        e2 = 0x0        f2 = 0x0
*charp/s
0x55:
data address not found
0x1fff90,3/X
0x1FFF90:       0x1FFFB0        0x1FFFB6        0x0
0x1fffb0/s
0x1FFFB0:       a.out
/s
0x1FFFB0:       a.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);
}



q(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);
}
```

Figure 4: ADB output for C program of Figure 3

```
$c
_h+46:          _f      (0x2, 0x92D)
_g+48:          _h      (0x92C, 0x92B)
_f+70:          _g      (0x92D, 0xl258)
_h+46:          _f      (0x2, 0x92B)
_g+48:          _h      (0x92A, 0x929)
_f+70:          _g      (0x92B, 0xl254)
_h+46:          _f      (0x2, 0x929)
_g+48:          _h      (0x928, 0x927)
<INTERRUPT>
adb
,5$c
_h+46:          _f      (0x2, 0x92D)
_g+48:          _h      (0x92C, 0x92B)
_f+70:          _g      (0x92D, 0xl258)
_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 a.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
a.out:running
breakpoint      _settab+8:      moveml  #<>,-(a7)
settab+8:d
:c
a.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
```

Figure 7: ADB output for maps

```
adb a.out.unshared core.unshared
$m
? map   `a.out.unshared'
b1 = 0x8000      e1 = 0x83E4      f1 = 0x20
b2 = 0x8000      e2 = 0x83E4      f2 = 0x20
/ 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 a.out.shared core.shared
$m
? map   `a.out.shared'
b1 = 0x8000      e1 = 0x8390      f1 = 0x20
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    strl[]  = "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 a.out.shared core.shared
<b,-1/8ona
_strl:      052150 064563 020151 071440 060440 061550 060562 060543

_strl+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
_strl:      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
_strl:      052150 064563 020151 071440          This is
_strl+8:    060440 061550 060562 060543          a charac
_strl+16:   072145 071040 071564 071151          ter stri
_strl+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
_strl:      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,-1?utl4cn

                Inode   Name
0:              652     .
                82      ..
                5971    cap.c
                5323    cap
                0       pp



adb /dev/src -
02000>b
?m<b
new map     `/dev/src'
b1 = 02000          e1 = 0100000000     f1 = 0
b2 = 0              e2 = 0              f2 = 0
$v
variables
b = 02000
<b,-1?"flags"8ton"links,uid,gid"8t3bn"
size"8tbrdn"addr"8t8un"times"8t2Y2na
(type above two lines all on one line)
02000:  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

02040:  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

02100:  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
```

6.12 ADB Summary

Command Summary

a. Formatted printing

? format	print from a.out file according to format
/ format	print from core file according to format
= format	print the value of dot
?w expr	write expression into a.out file
/w expr	write expression into core file
?l expr	locate expression in a.out file

b. Breakpoint and program control

:b	set breakpoint at dot
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run a.out file under ADB control
:s	single step

c. Miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$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

d. Calling the shell

!	call shell to read rest of line

e. Assignment to variables

>name	assign dot to variable or register name

6.12.1 Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	machine instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
nt	move to next n space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

6.12.2 Expression Summary

a. Expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. flag _main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

b. Dyadic operators

+	add
-	subtract
*	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

c. Monadic operators

~	not
*	contents of location
-	integer negation

CHAPTER 7

MAKE: A PROGRAM MAINTAINER

CONTENTS

7.1 Introduction..... 7-1

7.2 Description Files and Substitutions..... 7-5

7.3 Command Usage..... 7-7

7.4 Implicit Rules..... 7-8

7.5 Example..... 7-10

7.6 Suggestions and Warnings..... 7-11

7.7 Suffixes and Transformation Rules..... 7-13

7.1 Introduction

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. Make provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell Make the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the make command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of Make is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file defines the graph of dependencies. Make does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc or Lex). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last **make** command. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the **make** command than to issue even one of the needed operations, so the typical cycle of program development operations becomes think, edit, make, test.

Make is most useful for medium-sized programming projects. It does not solve the problems of maintaining multiple source versions or of describing huge programs.

Basic Features The basic operation of Make is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. Make does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example. A program named prog is made by compiling and loading three C-language files x.c, y.c, and z.c. By convention, the output of the C compilations is found in files named x.o, y.o, and z.o. Assume that the files x.c and y.c share some declarations in a file named defs, but that z.c does not. That is, x.c and y.c have the line

```
#include "defs"
```

The following text describes the relationships and operations:

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

x.o y.o: defs
```

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

```
make
```

would perform the operations needed to recreate prog after any changes had been made to any of the four source files x.c, y.c, z.c, or defs.

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that prog depends on three ".o" files. Once these object files are current, the second line describes how to load them to create prog. The third line says that x.o and y.o depend on the file defs. From the file system, Make discovers that there are three ".c" files corresponding to the needed ".o" files, and uses built-in information on how to generate an object from a source file (i.e., issue a "cc -c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of Make's default rules:

```
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
```

If none of the source or object files had changed since the last time prog was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the defs file had been edited, x.c and y.c (but not z.c) would be recompiled, and then prog would be created from the new ".o" files. If only the file y.c had changed, only it would be recompiled, but it would still be necessary to reload prog.

If no target name is given on the Make command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile x.o if x.c or defs had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These targets can take advantage of Make's ability to generate files and substitute macros. Thus, a target "save" might be included to copy a certain set of files, or a target "cleanup" might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. 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. $$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: $*, $@, $?, and $<. They will be discussed later. The following fragment shows the use:

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

The command

```
make
```

loads the three object files with the Lex -lln library. The command

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

loads them with both the Lex (-ll) and the math (-lm)

libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in XENIX commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

7.2 Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not 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 value. Macro definitions may also appear on the Make command line (see below).

Other lines give information about target files. The general form of a target is:

```
target ... :[:] [dependent ...] [; commands] [# ...]
[(tab) commands] [# ...]
 ...
```

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

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed. Otherwise a default creation rule may be invoked.

2. In the double-colon case, a command sequence may be associated with each dependency line. If the target is out of date with any of the files on a particular line, then the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

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

Before issuing any command, certain macros are set. $@ is set to the name of the file to be "made". $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, Make prints a message and stops.

7.3 Command Usage

The make command takes four kinds of arguments: macro definitions, flags, description file names, and target file names. The syntax is as follows:

```
make [ flags ]  [ macro definitions ]  [ targets ]
```

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

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

- -i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.
- -s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.
- -r Do not use the built-in rules.
- -n No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.
- -t Touch the target files (causing them to be up to date) rather than issue the usual commands.
- -q Question. The make command returns a zero or non-zero status code depending on whether the target file is or is not up to date.
- -p Print out the complete set of macro definitions and target descriptions
- -d Debug mode. Print out detailed information on files and times examined.
- -f Description file name. The next argument is assumed to be the name of a description file. A file name of "-" denotes the standard input. If there are no "-f" arguments, the file named makefile or Makefile

in the current directory is read. The contents of the description files override the built-in rules if they are present).

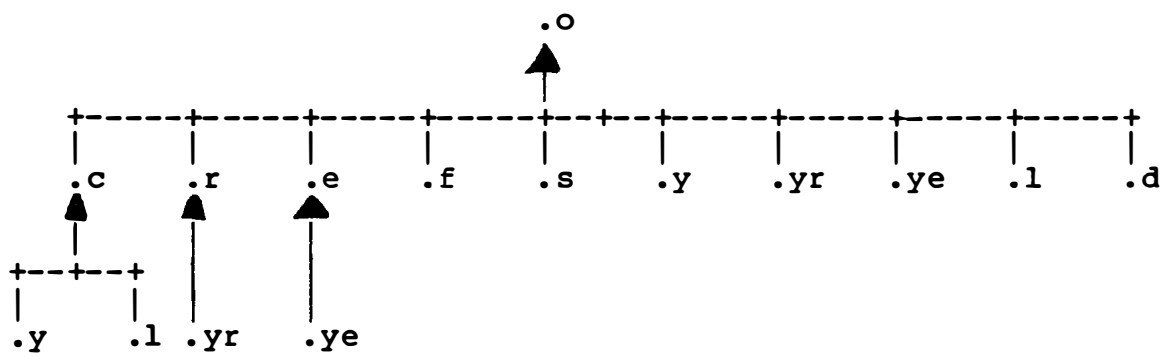
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

7.4 Implicit Rules

The Make program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

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

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



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. However, if there is no x.c but there is an x.l, Make then discards the intermediate C-language file and uses the direct link in the graph above.

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

```
make CC=newcc
```

causes the newcc command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus

```
make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

7.5 Example

As an example of the use of Make, we will present the description file used to maintain the make command itself. The code for Make is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

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

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)
```

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 description file:

```
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 description file, Make found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the size command in the description file suppressed the printing of the command, so only the sizes are written.

The last few targets in the description file 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:

```
make print "P = lpr"
```

or

```
make print "P=  cat >zap"
```

7.6 Suggestions and Warnings

The most common difficulties arise from Make's specific meaning of dependency. If file x.c has a "#include "defs"" line, 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 discover what Make would do, the **-n** option is very useful. The command

```
make -n
```

orders Make to print out the commands it would issue without actually taking the time to execute them. 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
```

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

The debugging flag **-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.

7.7 Suffixes and Transformation Rules

The Make program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the "-r" flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; Make looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, Make acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a ".r" file to a ".o" file is thus ".r.o". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

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

is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
    $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
    $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
    $(AS) -o $@ $<
.y.o :
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.y.c :
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@
```

CHAPTER 8

AS: AN ASSEMBLER

CONTENTS

8.1 Introduction..... 8-1

8.2 Invocation..... 8-1

8.3 Invocation Options..... 8-2

8.4 Source Program Format..... 8-3
8.4.1 Label Field..... 8-4
8.4.2 Opcode Field..... 8-4
8.4.3 Operand-Field..... 8-5
8.4.4 Comment Field..... 8-5

8.5 Symbols and Expressions..... 8-5
8.5.1 Symbols..... 8-5
8.5.2 Assembly Location Counter..... 8-8
8.5.3 Program Sections..... 8-9
8.5.4 Constants..... 8-9
8.5.5 Operators..... 8-11
8.5.6 Terms..... 8-12
8.5.7 Expressions..... 8-12

8.6 Instructions and Addressing Modes..... 8-13
8.6.1 Instruction Mnemonics..... 8-13
8.6.2 Operand Addressing Modes..... 8-14

8.7 Assembler Directives..... 8-17
8.7.1 .ascii .asciz..... 8-17
8.7.2 .blkb .blkw .blkl..... 8-18
8.7.3 .byte .word .long..... 8-19
8.7.4 .end..... 8-19
8.7.5 .text .data .bss..... 8-19
8.7.6 .globl .comm..... 8-20
8.7.7 .even..... 8-21

8.8 Operation Codes..... 8-22

8.9 Error Messages..... 8-23

8.1 Introduction

This chapter describes the use of the XENIX assembler, named **as**, for the Motorola MC68000 microprocessor. It is beyond the scope of this manual to describe the instruction set of the 68000 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, N.J: Prentice-Hall, Inc., 1982.

This chapter is organized as follows:

The Command Line
: Discusses assembler invocation and command line options.

Source Program Format
: Discusses the proper layout of an assembly language program, including specification of the label, opcode, operand, and comment fields.

Symbols and Expressions
: Discusses the symbols and expressions used in writing assembly language programs.

Instructions and Addressing Modes
: Discusses the available instructions and addressing modes.

Assembler Directives
: Discusses assembler directives.

Operation Codes
: Lists the available 68000 operation codes.

Error Messages
: Lists error messages that can be generated by **as**.

8.2 Invocation

As can be invoked with one or more arguments options. 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 should be named filename.s. If a filename does not have the ".s" extension, the assembler prints a warning message, but still assembles the specified file. Note, that 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
```

8.3 Invocation Options

The various flags and their function are:

-o relname — The default output name is filename.o if assembling on an MC68000, and filename.b if cross assembling. This can be overridden by giving **as** the **-o** flag and giving the new filename in the argument following the **-o**. The **-o** must be the last argument in a flag bunch. Subsequent flags are ignored. 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(5) format file that is produced by the assembler, including locals. If you want only symbols that are defined as **.globl** or **.comm** to be included, you can give 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.

8.4 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 [operand-field] [;]] [| 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.

In general, blanks or tabs are allowed anywhere in a statement; that is, multiple blanks are 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.

8.4.1 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 10 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, defined below
```

8.4.2 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 the program.

A machine instruction is indicated by an instruction mnemonic. Some conventions used in as for instruction mnemonics are be described later in a later section. A complete list of the opcodes is also presented.

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 lower case.

8.4.3 Operand-Field

A distinction is made between operand-field and operand in as. 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.

8.4.4 Comment Field

The comment delimiter in as 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 obvious exception of the newline character, which starts a newline.

8.5 Symbols and Expressions

This section describes the various components of as expressions: symbols, numbers, terms, and expressions.

8.5.1 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 emits 8 characters so a 9 or more character symbol that you think is the same in C and assembly may not match. Upper and lower cases are distinct, ("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

8.5.1.1 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 it is considered to be declared in the same program section as the expression. See the discussion in a later section for an explanation of absolute and relocatable expressions.

8.5.1.2 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.

8.5.1.3 External Symbols

A program may be assembled in separate modules, and then linked together to form a single program (see ld(1)). External symbols are defined in each of these separate modules. A symbol which 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.

8.5.1.4 Local Symbols

Local symbols provide a convenient means of generating labels for branch instructions, etc. 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.

8.5.2 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 pseudo-op.

For example:

```
movl .,dl       | load value of program counter into dl
```

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 as's 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.

Note that the :p operator allows you to assemble values that are location relative both locally (within a module) and across module boundaries, without needing to do explicit address arithmetic.

8.5.3 Program Sections

As in XENIX, programs to as are divided into two sections: text and data. The normal interpretation of these sections is: 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

```
        .text
LABEL1: movw    d1,d2
        .data
LABEL2: .word 27
        .text
LABEL3: addl  d2,d1
        .data
LABEL4: .byte 4
```

in the output, LABEL1 will immediately precede LABEL3, and LABEL2 will immediately precede LABEL4. 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 file with all addresses correctly resolved, with the exception of .comm's and undefined .globl's. For more information on the format of the output file, consult a.out(5).

8.5.4 Constants

All constants are considered absolute quantities when appearing in an expression.

8.5.4.1 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 upper or lower case; 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 by a number, (e.g., '\27) as this is more easily represented in one of the other formats.

8.5.5 Operators

8.5.5.1 Unary Operators

There are three unary operators in as:

Operator	Function
+	unary plus, has no effect.
-	unary minus.
~	logical negation.
: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.

8.5.5.2 Binary Operators

Binary operators in as 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 error messages in expressions can be fixed by breaking the expressions into multiple statements using direct assignment statements.

8.5.6 Terms

A term is a component of an expression. A term may be one of the following:

A. A number whose 32-bit value is used

B. A symbol

C. A term preceded by a unary operator. For example, both "term" and "~term" may be considered to be terms. Multiple unary operators are allowed; e.g. "+--+A" has the same value as "A".

8.5.7 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 one 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, an "Invalid Expression" error will be generated. 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:

A. An expression is absolute if its value is fixed. An expression whose terms are constants, or symbols whose values are constants via a direct assignment statement, is absolute. A relocatable expression minus a relocatable term, where both items belong to the same program section is also absolute.

B. 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 which 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.

C. An expression is external (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. Exception: the expression "sym-sym2" where both sym and sym2 are external symbols is not allowed.

8.6 Instructions and Addressing Modes

This section describes the conventions used in as to specify instruction mnemonics and addressing modes.

8.6.1 Instruction Mnemonics

The instruction mnemonics used by as are described in the previously mentioned 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 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.

8.6.2 Operand Addressing Modes

These effective addressing modes specify the operand(s) of an instruction. For details of the effective addressing modes, see section 2.10 of 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. This is the format used by MIT and a number of other M68000 UNIX vendors, and is used by Motorola to describe technically the register transfers that are occurring within the machine, so 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. Be aware, 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. These name or names appear to the right of the Motorola name in parenthesis.

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)+,dl
movl    a7@+,dl
```

Address Register Indirect with Predecrement (autodec)

```
movl    dl,(a7)-
movl    dl,a7@-
```

Address 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
```

Address 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
```

Absolute Short Address

```
movl    xx:w,dl
```

Absolute Long Address (absolute)

Note that this is the address mode assumed 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 it is provided because it is also allowed.

```
movl    xx,dl
movl    xx:l,dl
```

Program 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". See also the displacement relocation mode in a.out(5).

```
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.

Program Counter with Index

```
jmp     switchtab(pc,d0:l)
jmp     pc@(switchtab,d0:l)
switchtab:
```

Immediate 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. Also note that the second and third examples are not RTL syntax in particular.

```
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 th mask for the auto-decrement addressing mode. Normal immediates are also allowed.

8.7 Assembler Directives

The following pseudo-ops are available in as:

.ascii .asciz	stores 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 psect Data psect Bss psect
.globl .comm	declares external symbols declares communal symbols
.even	forces location counter to next word boundary

8.7.1 .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 proceeded 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:
22 68 65 6C 6C 6F 20 74	.ascii "hello there"
68 65 72 65 22	
77 61 72 6E 69 6E 67 20	.ascii "Warning-\007\007 \n"
2D 07 07 20 0A	

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 sometimes used as arguments to XENIX system calls.

8.7.2 .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.

8.7.3 .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 one 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 five bytes, each with
                 a value of zero.
.byte            reserves a single 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.

8.7.4 .end

The .end directive indicates the physical end of the source program. The format is:

```
.end
```

The .end is not really required; reaching the end of file has the same effect.

8.7.5 .text .data .bss

These statements change the "program section" where assembled code will be loaded.

8.7.6 .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 anyway. 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(1) 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 was 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 final bss section, and resolves linkages.

8.7.7 .even

This directive advances the location counter if its current value is odd. This is useful for forcing storage allocation to be 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.

8.8 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 pseudo operations 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
```

8.9 Error Messages

If there are errors in an assembly, an error message appears on the standard error channel (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.

If an assembly listing was not requested, any source lines which caused an assembly diagnostic are displayed on the terminal (the standard error file). In addition, a list of assembly errors and their description is also displayed on the terminal.

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.

Non-relocatable expression
Some instructions require relocatable expressions as operands. 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
Something which must start at an even address does not.

Inconsistent effective address syntax
Both assembly and RTL syntax appear within a single module.

Non-word memory shift
An in memory shift instruction was given a size other than 16 bits.

CHAPTER 9

LEX: A LEXICAL ANALYZER

CONTENTS

9.1 Introduction..... 9-1

9.2 Lex Source..... 9-3

9.3 Lex Regular Expressions..... 9-4
- 9.3.1 Character classes..... 9-5
- 9.3.2 Arbitrary character..... 9-6
- 9.3.3 Optional Expressions..... 9-6
- 9.3.4 Repeated Expressions..... 9-6
- 9.3.5 Alternation and Grouping..... 9-7
- 9.3.6 Context Sensitivity..... 9-7
- 9.3.7 Repetitions and Definitions..... 9-8

9.4 Lex Actions..... 9-9

9.5 Ambiguous Source Rules..... 9-13

9.6 Lex Source Definitions..... 9-16

9.7 Usage..... 9-17

9.8 Lex and Yacc..... 9-18

9.9 Left Context Sensitivity..... 9-22

9.10 Character Set..... 9-24

9.11 Summary of Source Format..... 9-25

9.12 Notes..... 9-27

9.1 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 written 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 beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." At present, the only supported host language is C.

Lex turns the user's expressions and actions (called source in this section) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input here) and perform the specified actions for each expression as it is detected.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$ ;
```

is 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 which 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 ..."; and the $ indicates "end of line." No action is specified, so the program generated by Lex (yylex) 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 will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule 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 particularly 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 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 which 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.

9.2 Lex Source

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 which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are 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 an individual rule might appear

```
integer printf("found keyword INT");
```

to look for the string integer in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and 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 just 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 "gaseum"; a way of dealing with this is described later.

9.3 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

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes 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 non-

alphanumeric character being used as a text character, the user need not memorize the list above 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 \ 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.

9.3.1 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 up-arrow (^). The dash character indicates ranges. For example

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using the dash between any pair of characters which are not both upper case letters, both lower case 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 two signs.

In character classes, the up-arrow (^) 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.

9.3.2 Arbitrary character

To match almost any character, the period (.) designates the class of all characters except a newline. Escaping into octal is possible although non-portable. For example

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

9.3.3 Optional Expressions

The question mark (?) operator indicates an optional element of an expression. Thus

```
ab?c
```

matches either "ac" or "abc".

9.3.4 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.

9.3.5 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 can 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".

9.3.6 Context Sensitivity

Lex recognizes a small amount of surrounding context. The two simplest operators for this are the up-arrow (^) and the dollar sign ($). If the first character of an expression is an up-arrow, 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 up-arrow, 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 later. 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 up-arrow (^) operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

9.3.7 Repetitions and Definitions

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. 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.

9.4 Lex Actions

When an expression written as above is matched, 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. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

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 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" (% indicating data conversion, and s indicating 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 which 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 by

```
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 would 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.

Example: Consider a language which 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 quote 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 C problem of distinguishing the ambiguity of "=-a". Suppose it is desired to treat this as "=- a" but print a message. A rule might be

```
=-[a-zA-Z]      {i
        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 are:

1. input() which returns the next input character;
2. output(c) which writes the character c on the output; and
3. unput(c) 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 Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / 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 which 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 which 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.

9.5 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 which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer keyword action ...;
[a-z]+  identifier action ...;
```

to be given in that order. 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, only 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.

9.6 Lex 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 which 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 which 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, etc., 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 the selection of a host language, 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 below under "Summary of Source Format."

9.7 Usage

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 on 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 -lln. So an appropriate set of commands is

```
lex source
cc lex.yy.c -lln
```

The resulting program is placed on the usual file a.out for later execution. To use Lex with Yacc see the following section and Chapter 10 "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.

9.8 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 -lln
```

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations 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 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 operator % (remainder) 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 upper and lower case 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 (upper or lower) 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.

```
{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}l{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.

9.9 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 circumflex (^) 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: a simple use of flags, when only a few rules change from one environment to another, the use of start conditions on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which 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 which began with the letter "a", changing "magic" to "second" on every line which began with the letter "b", and changing "magic" to "third" on every line which 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 (< and >):

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition "name1". To enter a start condition, execute the action statement

```
BEGIN namel;
```

which changes the start condition to namel. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<namel,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.

9.10 Character Set

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:

```
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
```

This table maps the lower and upper case 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.

9.11 Summary of 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.

9.12 Notes

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

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, the user must not have used unput to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

CHAPTER 10

YACC: A COMPILER-COMPILER

CONTENTS

10.1 Introduction..... 10-1

10.2 Basic Specifications..... 10-4

10.3 Actions..... 10-6

10.4 Lexical Analysis..... 10-9

10.5 How the Parser Works..... 10-11

10.6 Ambiguity and Conflicts..... 10-17

10.7 Precedence..... 10-22

10.8 Error Handling..... 10-25

10.9 The Yacc Environment..... 10-27

10.10 Hints for Preparing Specifications..... 10-28

10.11 Advanced Topics..... 10-32

10.12 A Simple Example..... 10-35

10.13 Yacc Input Syntax..... 10-38

10.14 An Advanced Example..... 10-40

10.15 Old Features..... 10-47

10.1 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 (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.

in this case, month name would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7/4/1776
```

as a synonym for

```
July 4, 1776
```

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

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 quotes; 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;

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character 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 ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "*''. As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule 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:

- 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.
- gives some suggestions which should improve the style and efficiency of the specifications.

10.2 Basic 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 `%' 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
```

```
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 symbol "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

```
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 name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, 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 "end-of-file" or "end-of-record".

10.3 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

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.

10.4 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, 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:

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.

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.

10.5 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

```
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

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:

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

```
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

```
%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:

10.6 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.

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 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.

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 preceding "un-ELSE'd" IF. In this

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

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 45. 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

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,

```
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:

local guru might also be appropriate.

10.7 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 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.

10.8 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

```
%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

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.

10.9 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

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

Input Style It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. 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.

d. 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.

e. 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.

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
    ;
```

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.

10.10 Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

```
%{
  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.

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, using the mechanism described in the last subsection, but it is difficult.

For stylistic (and other) reasons, it is best that keywords be reserved; that is, be forbidden for use as variable names.

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!

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:

Support for Arbitrary Value Types By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser 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(1) 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 can not 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

10.11 Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. 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.

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 that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol noun in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

10.12 A Simple Example

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
```

```
< 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 a priori 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.

```
        yylval = c - 'a';
        return ( LETTER );
        }
if( isdigit( c ) ) {
        yylval = c - '0';
        return(  DIGIT  );
        }
return( c );
}
```

```
                { 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 ) ) {
```

```
def     : START  IDENTIFIER
        | UNION  { Copy union definition to output }
        | LCURL  { Copy C code to output file }  RCURL
        | ndefs  rword  tag  nlist
        ;

rword   : 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  ';'
        ;
```

10.13 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 action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
            /* grammar for the input to Yacc */

        /*  basic  entities  */
%token IDENTIFIER       /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier followed by colon      */
%token  NUMBER          /* [0-9]+    */

        /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

%token  MARK    /* the %% mark */
%token  LCURL   /* the %{ mark */
%token  RCURL   /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    : defs  MARK  rules  tail
        ;

tail    : MARK    { Eat up the rest of the file }
        | /* empty: the second MARK is optional */
        ;

defs    : /* empty */
        | defs  def
        ;
```

```
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 "," 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.

10.14 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 double's. 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

```
        | 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
```

```
%{

#  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 ); }
```

```
                        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);
        }
```

```
                { $$ = vmul( $1.lo, $1.hi, $3 ); }
        | dexp '*' vexp
                { $$ = 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;
                                }
```

10.15 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 quotes """.

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 quotes around it. Otherwise, it is difficult to find the value for such literals. The use of multi-character 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, "\\" 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.

```
INTERVAL  vdiv( a, b, v )  double a, b;  INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

Appendix A: The C Shell

Csh is an alternate command language interpreter. It incorporates good features of other shells and a history mechanism. most of the features unique to csh are designed more for the interactive XENIX user, although some features of other shells have been incorporated to make writing shell procedures easier.

XENIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with csh is possible after reading just the first section of this document. The second section describes the capabilities you can explore after you have begun to become acquainted with the Cshell. Later sections introduce features which are useful, but not necessary for all users of the shell.

The final section of this chapter lists special characters of the Cshell.

A shell is a command language interpreter. Csh is the name of one particular command interpreter on XENIX. The primary purpose of csh is to translate command lines typed at a terminal into system actions, such as invocations of other programs. Csh is a user program just like any you might write.

This document provides a full description of all features of the shell and is a final reference for all questions.

A.1 Details on the shell for terminal users

A.1.1 Shell startup and termination

When you login, the shell is started by the system in your home directory and begins by reading commands from a file .cshrc in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A login shell, executed after you login to the system, will, after it reads commands from .cshrc, read commands from a file .login also in your home directory. This file contains commands which you wish to do each time you login to the XENIX system. A typical .login file might look something like this:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
        'set noglob ; eval `tset -s -m dialup:c100rv4pna \
        -m plugboard:?hp2621nl *`';
ts; stty intr ^C kill ^U crt
set time=15 history=10
if (-e $mail) then
        echo "${prompt}mail"
        mail
endif
```

This above file contains several commands to be executed by XENIX at each login. The first is a set command which is interpreted directly by the shell. It sets the shell variable ignoreeof which shields the shell from log off if <CONTROL-D> is hit. Instead of <CONTROL-D>, the logout command is used to log off the system. By setting the mail variable, the shell is notified that it is to watch for incoming mail and to notifiy the user if new mail arrives.

Next the shell variable time is set to "15" causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable "history" is set to 10 indicating that the shell will remember the last 10 commands types in its history list, (described later).

Next, an alias, "ts", is created which executes a tset(1) command setting up the modes of the terminal. The parameters to tset indicate the kinds of terminal normally used when not on a hardwired port. Then "ts" is executed, and the stty command is used to change the interrupt character to <CONTROL-C> and the line kill character to <CONTROL-U>.

Finally, if my mailbox file exists, then I run the **mail** program to process my mail.

When the **mail** programs finish, the shell will finish processing my .login file and begin reading commands from the terminal, prompting for each with "% ". When I log off (by giving the logout command) the shell will print "logout" and execute commands from the file .logout if it exists in my home directory. After that, the shell will terminate and XENIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the logout message the shell is committed to terminating and will take no further input from my terminal.

A.1.2 Shell variables

The shell maintains a set of variables. We saw above the variables history and time which had the values 10 and 15. In fact, each shell variable has as value an array of zero or more strings. Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable path. This variable contains a sequence of directory names where the shell searches for commands. The set command with no arguments shows the value of all variables currently defined (we usually say set) in the shell. The default value for path will be shown by set to be

```
% set
argv    ()
cwd     /usr/bill
home    /usr/bill
path    (. /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
term    cl00rv4pna
user    bill
%
```

This output indicates that the variable path points to the current directory indicated by dot (.) and then /bin, and /usr/bin. Your own local commands may be in dot. Normal XENIX commands live in /bin and /usr/bin.

Often a number of locally developed programs on the system live in the directory /usr/local. If we wish that all shells which we invoke to have access to these new programs we can place the command

```
set path=(. /bin /usr/bin /usr/local)
```

in our file .cshrc in our home directory. Try doing this and then logging out and back in. Then type

```
set
```

again to see that the value assigned to path has changed.

You should be aware that the shell examines each directory that you insert into your path and determines which commands are contained there. Except for the current directory, dot (.), which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found. If you wish to use a command which has been added in this way, you should give the command

```
rehash
```

to the shell, which causes it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory . on each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

Other useful built in variables are the variable home which shows your home directory, cwd which contains your current working directory, the variable ignoreeof which can be set in your .login file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable "ignoreeof" is one of several variables which the shell does not care about the value of, only whether they are set or unset. Thus to set this variable you simply do

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

These give the variable "ignoreeof" no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables noclobber and mail. The metasyntax

```
>filename
```

which redirects the standard output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

```
set noclobber
```

in your .login file. Then trying to do

```
date > now
```

would cause a diagnostic if "now" existed already. You could type

```
date >! now
```

if you really wanted to overwrite the contents of now. The ">!" is a special metasyntax indicating that overwriting or "clobbering" the file is ok. (The space between the exclamation (!) and the word "now" is critical here, as "!now" would be an invocation of the history mechanism, and have a totally different effect.)

A.1.3 The Shell's History List

The shell can maintain a history list into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the shell.

```
% cat bug.c
main()

{
        printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
        printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
        printf("hello\n");
w
32
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill       3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill       3932 Dec 19 09:42 bug
% bug
hello
% num bug.c | spp
spp: Command not found.
% ^spp^ssp
num bug.c | ssp
    1   main()
    3   {
    4           printf("hello\n");
    5   }
% !! | lpr
num bug.c | ssp | lpr
%
```

In this example, we have a very simple C program which has a bug (or two) in it in the file bug.c, which we **cat** out on our terminal. We then try to run the C compiler on it, referring to the file again as "!$", meaning the last argument to the previous command. Here the exclamation mark (!) is the history mechanism invocation metacharacter, and the dollar sign ($) stands for the last argument, by analogy to the dollar sign in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics, so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as "!c", which repeats the last command which started with the letter "c". If there were other commands starting with "c" done recently we could have said "!cc" or even "!cc:p" which would have printed the last command starting with "cc" without executing it.

After this recompilation, we ran the resulting a.out file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra "-o bug" telling the compiler to place the resultant binary in the file bug rather than a.out. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the **size** command to see how large the binary program images we have created were, and then an "ls -l" command with the same argument list, denoting the argument list "*". Finally, we ran the program bug to see that its output is indeed correct.

To make a numbered listing of the program, we ran the **num** command on the file bug.c. In order to filter out blank lines in the output of **num** we ran the output through the filter **ssp**, but misspelled it as "spp". To correct this we used a shell substitute, placing the old text and new text between up arrow (^) characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with "!!", but sent its output to the line printer.

There are other mechanisms available for repeating commands. The history command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete

description of all these mechanisms is given in the C shell manual pages in the XENIX Programmers Manual.

A.1.4 Aliases

The shell has an alias mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment or involve commands such as cd which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called "newmail" you wish to use, rather than the standard mail program which is called "mail". If you place the shell command

```
alias mail newmail
```

in your .cshrc file, the shell will transform an input line of the form

```
mail bill
```

into a call on "newmail". More generally, suppose we wish the command **ls** to always show sizes of files, that is to always do **-s**. We can do

```
alias ls ls -s
```

or even

```
alias dir ls -s
```

creating a new command named "dir" which does an "ls -s". If we say

```
dir ~bill
```

then the shell will translate this to

```
ls -s /usr/bill
```

Thus the alias mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands

or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

would do an ls command after each change directory cd command. We enclosed the entire alias definition in single quotes (') to prevent most substitutions from occurring and the semicolon (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from being interpreted when the alias command is typed in. The "\!*" here substitutes the entire argument list to the pre-aliasing cd command, without giving an error if there were no arguments. The semicolon (;) separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

```
alias whois 'grep \!^ /etc/passwd'
```

defines a command which looks up its first argument in the password file.

Warning: The shell currently reads the .cshrc file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

A.1.5 More redirection; >> and >&

There are a few more notations useful to the terminal user which have not been introduced yet. In addition to the standard output, commands also have a diagnostic output which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can type

```
command >& file
```

The ">&" here tells the shell to route both the diagnostic output and the standard output into file. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon lpr. A command form

```
command >&! file
```

exists, and is used when noclobber is set and file already exists.

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file. If noclobber is set, then an error will result if file does not exist, otherwise the shell will create file if it doesn't exist. A form

```
command >>! file
```

makes it not be an error for file to not exist when noclobber is set.

A.1.6 Jobs: Background and Foreground

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single job is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If the ampersand metacharacter (&) is typed at the end of the commands, then the job is started as a background job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

would run the du program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file usage and return

immediately with a prompt for the next command without out waiting for du to finish. The du program would continue executing in the background until it finished, even though you can type and execute more commands in the mean time. Background jobs are unaffected by any signals from the keyboard like the <INTERRUPT> or <QUIT> signals mentioned earlier.

The kill command terminates a background job immediately. It may be given process numbers as arguments, as printed by ps.

A.1.7 Useful Built-In Commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The **alias** command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

```
alias ls
```

to show the current alias for, e.g., ls.

The **echo** command prints its arguments. It is often used in shell scripts or as an interactive command to see what filename expansions will produce.

The **history** command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called prompt. By placing an exclamation mark (!) in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

```
set prompt='\! % '
```

Note that the exclamation mark (!) had to be escaped here even within backslashes.

The **logout** command can be used to terminate a login shell which has ignoreeof set.

The **rehash** command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shell's search path

and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command can be used to repeat a command several times. Thus to make 5 copies of the file one in the file five you could do

```
repeat 5 cat one >> five
```

The **setenv** command can be used to set variables in the environment. Thus

```
setenv TERM adm3a
```

sets the value of the environment variable TERM to "adm3a". A user program printenv exists which will print out the environment. It might then show:

```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

The **source** command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the .cshrc file which you wish to take effect before the next time you login.

The time command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8%
% time wc /etc/rc /usr/bill/rc
      52     178    1347 /etc/rc
      52     178    1347 /usr/bill/rc
     104     356    2694 total
0.1u 0.1s 0:00 13%
%
```

indicates that the cp command used a negligible amount of user time (u) and about 1/10th of a second system time (s); the elapsed time was 1 second (0:01). The word count

command, **wc**, on the other hand, used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage "13%" indicates that over the period when it was active the command **wc** used an average of 13 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands can be used to remove aliases and variable definitions from the shell, and **unsetenv** removes variables from the environment.

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the **foreach** built-in command which can be used to run the same command sequence with a number of different arguments.

A.2 Shell Control Structures and Command Scripts

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called shell scripts. We here detail those features of the shell useful to the writers of such scripts.

It is important to first note what shell scripts are not useful for. There is a program called make which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a makefile which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this makefile. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a makefile may be created which defines how different versions of the document are to be created and which options of nroff or troff are appropriate.

A.2.1 Invocation and the argv variable

A csh command script may be interpreted by saying

```
% csh script ...
```

where script is the name of the file containing a group of

csh commands and "..." is replaced by a sequence of arguments. The shell places these arguments in the variable argv and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file script executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a pound sign (#)) then a /bin/csh will automatically be invoked to execute script when you type

```
script
```

If the file does not begin with a pound sign (#) then the standard shell /bin/sh will be used to execute it. This allows you to convert your older shell scripts to use csh at your convenience.

A.2.2 Variable substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as variable substitution is done on these words. Keyed by the dollar sign ($), this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable argv to be echoed to the output of the shell script. It is an error for argv to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to 1 if name is set or to 0 if name is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable name. Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

```
$argv[1]
```

gives the first component of argv or in the example above "a". Similarly

```
$argv[$#argv]
```

would give "c", and

```
$argv[1-2]
```

would give:

```
a b
```

Other notations useful in shell scripts are

```
$n
```

where n is an integer as a shorthand for

```
$argv[n]
```

the nth parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

```
$<
```

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
echo 'yes or no?\c'
set a=($<)
```

would write out the prompt "yes or no?" without a newline and then read the answer into the variable a. In this case "$#a" would be 0 if either a blank line or <CONTROL-D> was typed.

One minor difference between "$n" and "$argv[n]" should be noted here. The form "$argv[n]" will yield an error if n is not in the range "1-$#argv" while "$n" will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form "n-"; if there are less than "n" components of the given variable then no words are substituted. A range of the form "m-n" likewise returns an empty vector without giving an error when "m" exceeds the number of elements of the given variable, provided the subscript "n" is in range.

A.2.3 Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations "==" and "!=" compare strings and the operators "&&" and "||" implement the boolean AND and OR operations. The special operators "=~" and "!~" are similar to "==" and "!=" except that the string on the right side can have pattern matching characters (like *, ? or [and]) and the

test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

-? filename

where question mark (?) is replaced by a number of single characters. For instance the expression primitive

-e filename

tell whether the file filename exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form

{ command }

which returns true, i.e. 1 if the command succeeds exiting normally with exit status 0, or 0 if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable "$status" examined in the next command. Since "$status" is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

A.2.4 Sample shell script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script makes use of the **foreach** command, which causes the shell to execute the commands between the **foreach** and the matching **end** for each of the values given between parentheses with the named variable, in this case "i" set to successive values in the list. Within this loop we may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop the iteration variable (i in this case) has the value at the last iteration.

We set the variable noglob here to prevent filename expansion of the members of argv. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a "$" variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords here is not flexible due to

the current implementation of the shell. The following two formats are not acceptable to the shell:

```
if (expression) # Won't work!
then
        command
        ...
endif
```

and

```
if (expression) then command endif # Won't work
```

The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve "|", "&" or ";" and must not be another control command. The second form requires the final backslash (\) to immediately precede the end-of-line.

The more general if statements above also admit a sequence of else-if pairs followed by a single else and an endif, e.g.:

```
if ( expression ) then
        commands
else if (expression ) then
        commands
...
else
        commands
endif
```

Another important mechanism used in shell scripts is the colon (:) modifier. We can use the modifier ":r" here to extract the root of a filename or `:e' to extract the extension. Thus if the variable i has the value /mnt/foo.bar then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
%
```

shows how the ":r" modifier strips off the trailing ".bar" and the the ":e" modifier leaves only the "bar". Other modifiers will take off the last component of a pathname leaving the head ":h" or all but the last component of a pathname leaving the tail ":t". These modifiers are fully described in the csh(1S) manual pages in the XENIX Reference manual. It is also possible to use the command substitution mechanism described in the next major section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. (It is also important to note that the current implementation of the shell limits the number of colon modifiers on a "$" substitution to 1. Thus

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.)

Finally, we note that the pound sign character (#) lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a pound sign are discarded by the shell. This character can be quoted using "'" or "\" to place it in an argument word.

A.2.5 Other control structures

The shell also has control structures **while** and **switch** similar to those of C. These take the forms

```
while ( expression )
        commands
end
```

and

Appendix A

```
switch ( word )

case strl:
        commands
        breaksw

 ...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the manual section for csh(1S). C programmers should note that we use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in cshell scripts is to use **break** rather than **breaksw** in switches.

Finally, cshell allows a **goto** statement, with labels looking like they do in C, i.e.:

```
loop:
        commands
        goto loop
```

A.2.6 Supplying input to commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under XENIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
%
```

The notation "<< 'EOF'" means that the standard input for the ed command is to come from the text in the shell script file up to the next line consisting of exactly "'EOF'". The fact that the EOF is enclosed in single quotes ('), i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the "<<" which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form "1,$" in our editor script we needed to insure that this dollar sign was not variable substituted. We could also have insured this by preceding the dollar sign ($) with a backslash (\), i.e.:

```
1,\$s/^[ ]*//
```

but quoting the EOF terminator is a more reliable way of achieving the same thing.

A.2.7 Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where label is a label in our program. If an interrupt is received the shell will do a "goto label" and we can remove the temporary files and then do an exit command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status 1.

A.2.8 Other Features

There are other features of the shell useful to writers of shell procedures. The verbose and echo options and the related -v and -x command line options can be used to help trace the actions of the shell. The -n option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that csh will not execute shell scripts which do not begin with the pound sign character (#), that is shell scripts that do not begin with a comment. Similarly, the /bin/sh on your system may well defer to csh to interpret shell scripts which begin with the pound sign (#). This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using the quotation mark ("), which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as the single quote (') does.

A.3 Loops At The Terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if there were three shells in use on a particular system, /bin/sh, /bin/nsh, and /bin/csh, you could count the number of persons using each shell by using the following commands:

```
% grep -c csh$ /etc/passwd
5
% grep -c nsh$ /etc/passwd
3
% grep -c -v sh$ /etc/passwd
20
%
```

Since these commands are very similar we can use **foreach** to do this more easily.

```
% foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
5
3
20
%
```

Note here that the shell prompts for input with "? " when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The set command here gave the variable a a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within back quote characters (`) is converted by the shell to a list of words. You can also place the quoted string within double quote characters (") to take each (non-empty) line as a component of the variable. This prevents the lines from being split into words at blanks and tabs. A modifier ":x" exists which can be used later to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

A.4 Braces { ... } in argument expansion

Another form of filename expansion, alluded to before involves the characters, "{" and "}". These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

```
AstrlB Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common

parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories hdrs, retrofit and csh in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

A.5 Command substitution

A command enclosed in back quotes (`) is replaced, just before filenames are expanded, by the output from that command. Thus, it is possible to do

```
set pwd=`pwd`
```

to save the current directory in the variable pwd or to do

```
vi `grep -l TRACE *.c`
```

to run the editor vi supplying as arguments those files whose names end in ".c" which have the string "TRACE" in them. Command expansion also occurs in input redirected with "<<" and within quotations ("). Refer to csh(1S) in the XENIX Reference manual for more information.

A.6 Other Details Not Covered Here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing XENIX programs and debugging shell scripts. See csh(1S) in the XENIX Reference Manual for a list of these options.

A.7 Special Characters

The following table lists the special characters of csh and the XENIX system. A number of these characters also have special meaning in expressions. See the csh manual section for a complete list.

Syntactic metacharacters

;	Separates commands to be executed sequentially
|	Separates commands in a pipeline
( )	Brackets expressions and variable values
&	Follows commands to be executed without waiting for completion

Filename metacharacters

/	Separates components of a file's pathname
?	Expansion character matching any single character
*	Expansion character matching any sequence of characters
[ ]	Expansion sequence matching any single character from a set of characters
~	Used at the beginning of a filename to indicate home directories
{ }	Used to specify groups of arguments with common parts

Quotation metacharacters

\	Prevents meta-meaning of following single character
'	Prevents meta-meaning of a group of characters
"	Like ', but allows variable and command expansion

Input/output metacharacters

<	Indicates redirected input
>	Indicates redirected output

Expansion/Substitution metacharacters

$	Indicates variable substitution
!	Indicates history substitution
:	Precedes substitution modifiers
^	Used in special forms of history substitution
`	Indicates command substitution

Other metacharacters

#	Begins scratch file names; indicates shell comments
-	Prefixes option (flag) arguments to commands
%	Prefixes job name specifications

APPENDIX B: M4 - A Macro Processor

M4 is the name of the XENIX macro processor. Macro processors are used to define and to process 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:

1. More structured
2. More readable
3. 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, a macro processor provides:

- Macros with arguments
- Conditional macro expansions
- Arithmetic expressions
- File manipulation facilities
- String processing functions

The basic operation of M4 is to copy its input to its output. As the input is read, , each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input it is rescanned by M4. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before M4 rescans the text.

M4 provides a collection of about twenty built-in macros which perform various operations. In addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

Appendix B

B.1 Usage

To invoke M4, type:

```
m4 [files]
```

Each argument file is processed in order. If there are no arguments, or if an argument is a dash (-), the standard input is read at that point. The processed text is written to the standard output.

```
m4 [files] >outputfile
```

B.2 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 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 "(", it is assumed to have no arguments. This is the situation for N above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
  ...
if (NNN > 100)
```

Appendix B

the variable NNN is absolutely unrelated to the defined macro N, even though it contains a lot of 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.3 Quoting

The more general solution is to delay the expansion of the arguments of define by quoting them. Any text surrounded by the single quotes ` and ' is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N')
```

the quotes 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 quotes

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 quote characters (` and ') are not convenient for some reason, the quote characters can be changed with the built-in changequote. For example:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to define. 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 quotes 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.4 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 nth 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, although that is less commonly used.) 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)
```

Appendix B

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$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.5 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.6 File Manipulation

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 ("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.7 System Command

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.8 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 quotes, 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.9 String Manipulation

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. substr(s, i, n) returns the substring of s that starts at the ith position (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.

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 translit 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 which 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.10 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 which 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 quotes.

APPENDIX C: C Language Portability

The C language is defined in the appendix to "The C Programming Language", by Kernighan and Ritchie. This definition leaves many details to be decided by individual implementations of the language. It is those incompletely specified features of the language that detract from its portability and that should 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 document 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, varying from small 8-bit microprocessors to large mainframes. This document is largely 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 or 4 byte integers.
- Two's Complement Arithmetic.

None of these features is required by the formal definition of the language, nor is it true of all implementations of C. However, the remainder of this document is largely 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 a large number of system calls and library

B.11 Summary of Built-ins

```
changequote(L, R)
define(name, replacement)
divert(number)
divnum
dnl
dumpdef(`name', `name', ...)
errprint(s, s, ...)
eval(numeric expression)
ifdef(`name', this if true, this if false)
ifelse(a, b, c, d)
include(file)
incr(number)
index(s1, s2)
len(string)
maketemp(...XXXXX...)
sinclude(file)
substr(string, position, number)
syscmd(s)
translit(str, from, to)
undefine(`name')
undivert(number,number,...)
```

routines which can be considered portable. These are described briefly in a later section.

This document is not intended as a C language primer, for which should be used. It is assumed here that the reader is familiar with C, and with the basic architecture of common microprocessors.

C.1 Source Code Portability

We are concerned here with source code portability, which means that programs can be compiled and run successfully on different machines without alteration.

Programs can be written to achieve this goal using several techniques. The first is to avoid using inherently non-portable language features. Secondly, any non-portable interactions with the environment, such as I/O to non-standard devices should be isolated, and possibly passed as an argument to the program at run time. For example programs should not, in general, contain hard-coded file pathnames except where these are commonly understood to be portable (an example might be /etc/passwd).

Files required at compile time (i.e. include files) may also introduce non-portability if the pathnames are not the same on all machines. However in some cases the use of include files to contain machine parameters can be used to make the source code itself portable.

C.2 Machine Hardware

As mentioned earlier, most non-portable features of the C language are due either to hardware differences in the target machine or to compiler differences. This section lists the more common hardware differences encountered on XENIX systems and some language features to beware of.

C.2.1 Byte Length

The length of the **char** data type is not defined in the language, other than that it must be sufficient to hold all members of the machine's character set as positive numbers. Within the scope of this document we will consider only 8-bit bytes, since this is the byte size on all XENIX systems.

C.2.2 Word Length

The definition of C makes no mention of the size of the basic data types for a given implementation. These generally 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 the PDP-11 **short** is the same length as **int**, whereas on the VAX **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 usually preferable to something like:

```
#ifdef PDP11
#define MAXPOS 32767
#else
        ...
#endif
```

Likewise to find the number of bytes in an **int** use **sizeof(int)** rather than 2, 4, or some other non-portable constant.

C.2.3 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 twofold: 1) data accessed as non-primitive data types is not portable, and 2) neither is code that makes use of

knowledge of the layout on a particular machine.

Thus unions containing structures are non-portable 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 four bytes from a long word, there's no chance of the code being 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.

C.2.4 Byte Order in a Word

The variation in byte order in a word between machines affects the portability of data between machines 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 PDP-11 systems there is an include file misc.h which 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 non-portable 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 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.

C.2.5 Bitfields

Bitfields are not implemented in all C compilers. When they are, a number of restrictions apply:

- No field may be larger than an **int**.
- No field will 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 disect bits from other data is definitely non-portable.

To ensure portability no individual field should exceed 16 bits.

C.2.6 Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers will not object to non-portable pointer operations. The lint program is particularly useful for detecting questionable pointer assignments and comparisons.

The common non-portable use of pointers is where a pointer to one data type is cast to be a pointer to a different data type. This almost always makes some assumption about the internal byte ordering and layout of the data type, and is therefore non-portable. For example, in the following code, the ordering of the bytes from the **long** in the byte array is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0xl2345678L;
```

The lint program will issue warning messages about such uses of pointers. Very occasionally it is necessary and valid to write code like this. An example is when the malloc() library routine is used to allocate memory for something other than type **char**. The routine is declared as type **char *** and so the return value has to be 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() routine is written to always return a starting address suitable for storing all types of data, but 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. The code will attract a warning message from lint. There is nothing which can be done about this.

```
extern char *malloc();
int *ip;

ip = (int *)malloc(50);
```

C.2.7 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 (although this can be increased - see 23fix(1)). 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.

C.2.8 Character Set

We have said that we are concerned here mainly with the ascii character set. The C language does not require this however. The only requirements are:

- All characters fit in the **char** data type.

- All characters 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 regardless of 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 which should be used for most tests on character quantities. Not only do they provide some insulation from the internal structure of the character set, their names are more meaningful than the equivalent line of code in most cases, Compare

```
if(isupper(c))
```

to

```
if((c >= 'A') && (c <= 'Z'))
```

With some of the other macros, such as isxdigit() 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.

C.3 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.

C.3.1 Signed/Unsigned char, Sign Extension

The current state of the signed versus unsigned char problem is best described as unsatisfactory. The problem is completely explained and discussed in Sign Extension and Portability in C, Hans Spiller, Microsoft 1982, so that material is not repeated here.

The sign extension problem is one of the more serious barriers to writing portable C, and the best solution at present is to write defensive code which does not rely on particular implementation features. The above paper suggests some ways.

C.3.2 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 which uses knowledge of a particular implementation is non-portable.

The PDP-11 compilers use arithmetic right shift. Thus to avoid sign extension it is necessary to either shift and mask out the appropriate number of high order bits, or to use a divide operator which will avoid the problem completely:

```
char c;

For  c >> 3;     use:    (c >> 3) & 0x1f;
                 or:     c / 8;
```

C.3.3 Identifier Length

The use of long identifier names will cause portability problems with some compilers. There are three different cases to be aware of:

- 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, upper and lower case letters are not distinct in identifiers.

C.3.4 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 non-register declarations, which 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. (Page 81). Whilst other machines/compilers may support declarations such as "register unsigned short" this should not be relied upon.

Since the compiler ignores excess register keywords, register type variables should always be declared in their importance of being register type. Then the ones the compiler ignores will be the least important.

C.3.5 Type Conversion

The C language has some rules for implicit type conversion; tt also allows explicit type conversions by type casting. The most common portability problem arising from 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 since 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("%x0,'\200');
}
```

Type conversion also takes place when arguments are passed to functions. Types **char** and **short** become **int**. Once again machines that sign extend **char** can give surprises. For example the following program gives -128 on the PDP-11:

```
char c = 128;
printf("%d\n",c);
```

This is because c is converted to **int** before passing on the stack to 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);
```

C.3.6 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]
```

Figure 1. File: /usr/include/varargs.h

The va_end() macro is not currently required. The 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**.

C.3.7 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 non-portable, 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 only use their argument once, and so can safely be called with a side-effect argument. The commonest 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 are 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;
```

For the same reason declaration initializations of the following type are unwise:

```
int  a = 0, b = a;
```

C.4 Program Environment Differences

Most non-trivial programs make system calls and use library routines for various services. The sections below indicate 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's environment.

Any program containing hard-coded pathnames to files or directories, or user id's, login names, terminal lines or other system dependent parameters is non-portable. These types of constant should be in header files, passed as command line arguments, obtained from the environment, or 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.

C.4.1 Libraries

The various XENIX library routines are generally portable among XENIX systems; however, note the following:

printf — The members of the printf family, printf, fprintf, sprintf, sscanf, and scanf have changed in several

small ways during the evolution of XENIX, and some features are not completely portable. The return values from these routines cannot be relied upon to have the same meaning on all systems. Certain of the format conversion characters have changed their meanings, in particular relating to upper/lower case in the output of hexadecimal numbers, and the specification of long integers on 16-bit word machines. The reference manual page for printf(3S) contains the correct specification for the routines.

C.5 Portability of Data

Data files are almost always non-portable 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 to get close to 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 much problem.

C.6 Lint

For a complete description of lint(1) see the discussion in a following chapter.

Lint is a C program checker which attempts to detect features of a collection of C source files which are non-portable or even incorrect C. One particular advantage 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 non-portable pointer arithmetic and dubious assignments and type conversions. Passage unscathed through lint is not a guarantee that a program is completely portable.

C.7 Byte Ordering Summary

The following conventions are used below. 'a0' is the lowest physical addressed byte of the data item. 'a1' has a byte address a0 + 1, etc. 'b0' is the least significant byte of the data item, 'b1' being the next least significant, etc.

Note that any program which actually makes use of the following information is guaranteed to be non-portable!

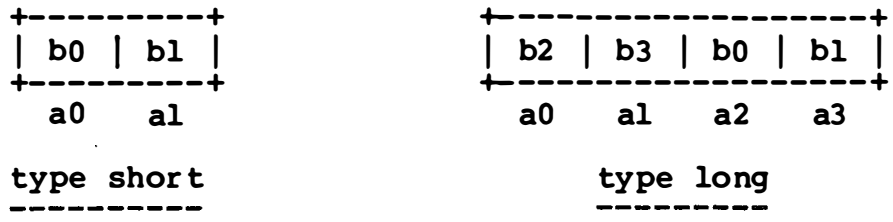


Figure 2. PDP-11 Byte Ordering

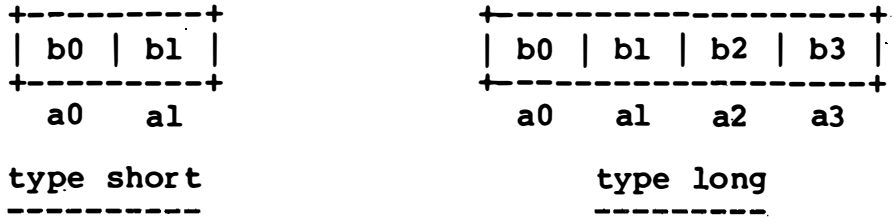


Figure 3. VAX-11 Byte Ordering

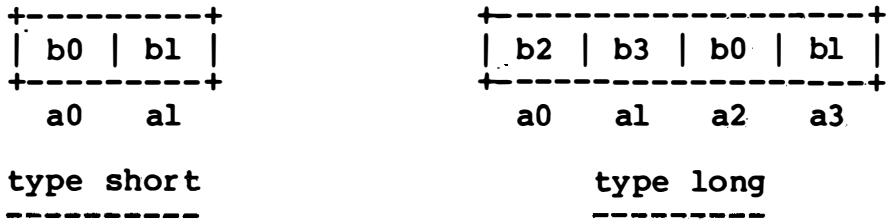


Figure 4. 8086 Byte Ordering

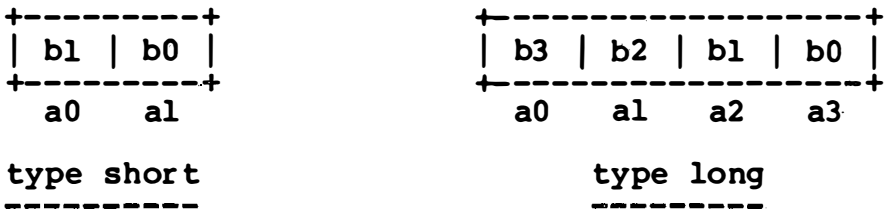


Figure 5. M68000 Byte Ordering

Appendix C

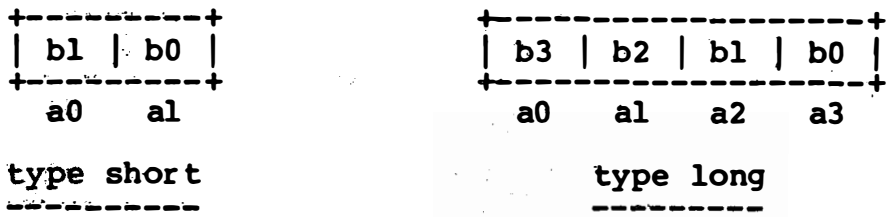


Figure 6. Z8000 Byte Ordering