PRO/VENIX™

UNIX™
for

Professional™

Support Tools Guide

digital
software

PRO/VENIX™
for the Professional

Support Tools Guide

Developed by:

VenturCom, Inc.
215 First Street
Cambridge, MA 02142

Digital Equipment Corporation
Maynard, MA 01754

First Printing

The software described in this manual is distributed as part of Digital Equipment Corporation's Digital Classified Software (DCS) Program. This program enables software developers to submit their software products to Digital for testing according to Digital quality standards for third party software. This software product has met the DCS standard specified in the software product description (SPD) for this product. You should refer to the SPD for information about these standards, the hardware and software required to run this product, and warranties (if any warranty is available).

The software described in this manual is furnished under a license and may only be used or copied in accordance with the terms of that license. This manual is reproduced with the permission of VenturCom, Inc.

Except as may be stated in the SPD for this product, no responsibility is assumed by Digital or its affiliated companies for use or reliability of this software, or for errors in this manual or in the software. Additional support and/or warranty services may be available from the developer of this software product. Digital has no connection with, and assumes no responsibility or liabilities in connection with these services.

This manual is subject to change without notice and does not constitute a commitment by Digital.

VENIX is a trademark of VenturCom, Inc.
UNIX is a trademark of AT&T Technology, Inc.

The following are trademarks of Digital Equipment Corporation:

DEC
DECmate
DECnet
DECsystem-10
DECSYSTEM-20
DECUS
DECwriter
DIBOL
MASSBUS
PDP
P/OS
Professional
Rainbow
RSTS
RSX
UNIBUS
VAX
VMS
VT
Work Processor
digital

The PRO/VENIX† Documentation Set

The PRO/VENIX documentation set consists of the following manuals:

PRO/VENIX Installation and System Manager's Guide

The set up and maintenance of PRO/VENIX are described in the installation sections. Other articles explain the UNIX-to-UNIX‡ communications systems. The "System Maintenance Reference Manual" contains reference pages for devices and system maintenance procedures (sections (7) and (8)).

PRO/VENIX User Guide

The *User Guide* contains tutorials for newcomers to PRO/VENIX, covering basic use of the system, the editor **vi** and use of the command language interpreters.

PRO/VENIX Document Processing Guide

The line and screen editors and **nroff**-related text formatting utilities are described in the Document Processing Guide. Topics include: line editor **ed**, and stream editor **sed**; the text formatter **nroff**; the **nroff**-preprocessors **tbl** and **neqn**.

PRO/VENIX Programming Guide

The chapters in the *Programming Guide* explicate the different programming languages for VENIX.

† VENIX is a trademark of VenturCom, Inc.

‡ UNIX is a trademark of Bell Laboratories.

PRO/VENIX Support Tools Guide

This guide includes tools for programming, such as the compiler-writing languages Yacc and Lex, the M4 Macro processor, the program development utility Make, and the desk calculator programs DC and BC.

PRO/VENIX User Reference Manual

This is a complete and concise reference for the PRO/VENIX system. This volume contains write-ups on all PRO/VENIX commands.

PRO/VENIX Progammer Reference Manual

The reference pages in this volume include system calls, library functions, file formats, miscellaneous functions and games.

Contents

Introduction

The *Support Tools Guide* is a description of the various software ''tools'' that aid the VENIX operating system user. The following paragraphs contain a brief description of each chapter.

The first chapter, A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make), describes a software tool for maintaining, updating, and regenerating groups of computer programs. The many activities of program development and maintenance are made simpler by the **make** program.

Chapter 2, THE M4 MACRO PROCESSOR, describes a general purpose macro processor that may be used as a front end for rational Fortran, C, and other programming languages.

Chapter 3, THE awk PROGRAMMING LANGUAGE, describes a software tool designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

Chapter 4, ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (bc), describes a compiler for doing arbitrary precision arithmetic on the VENIX operating system.

Chapter 5, INTERACTIVE DESK CALCULATOR (dc), describes a program implemented on the VENIX operating system to do arbitrary-precision integer arithmetic.

Chapter 6, LEXICAL ANALYZER GENERATOR (lex), describes a software tool that lexically processes character input streams.

Chapter 7, YET ANOTHER COMPILER-COMPILER (yacc), describes the **yacc** program. The **yacc** program provides a general tool for imposing structure on the input to a computer program.

The support tools provide an added dimension to the basic VENIX software commands. The ''tools'' described enable the user to fully utilize the VENIX operating system.

Introduction to the Programmer Reference Manual

This volume describes features of the VENIX† system, a licensed UNIX‡, time-sharing operating system. It does not provide an overview or tutorial information on the VENIX operating system, its facilities, or implementation. Various documents on these topics are contained in other VENIX manuals. For a general tutorial see 'VENIX for Beginners' in the *User Guide*. Those with previous UNIX experience will find VENIX is quite similar to other versions of UNIX.

The *Programmer Reference Manual* attempts to be timely, complete and concise. A lot of information is packed into these pages in a concentrated form. Examples have been included to assist the reader to understand some of the more terse descriptions. We hope that the user will be able to flip open the manual, refer to the appropriate section and find the needed information.

The *Programmer Reference Manual* contains sections 2 through 6. Commands (section 1) are contained in the *User Reference Manual*. The user should be familiar with the basic layout:

User Reference Manual

1. Commands

Programmer Reference Manual

2. System calls
3. Subroutines
4. File Formats and Conventions
5. Miscellaneous Facilities
6. Games

When referring to a particular item in the manual, it is common practice to append the section number in parentheses: for instance, **open**(2) refers to the **open** system call in section 2.

†VENIX is a trademark of VenturCom, Inc.
‡UNIX is a trademark of Bell Laboratories

Introduction to the Programmer Reference Manual

At the beginning of this volume is a table of contents, organized by section and alphabetically within each section.

Following is a short overview of the sections in both the *Programmer Reference Manual* and the *User Reference Manual*:

Commands are programs intended to be invoked directly by the user. Commands generally reside in directory */bin* (for *bin* ary programs). Some programs also reside in */usr/bin,* to save space in */bin.* These directories are searched automatically by the command interpreter.

System calls are entries into the VENIX supervisor. Every system call has one or more C language interfaces described in section 2.

An assortment of *subroutines* is available; they are described in section 3. The primary libraries in which they are kept are described in **intro**(3). The functions are described in terms of C, but most will work with Fortran as well.

Section 4 *file formats and conventions* documents the structure of a particular kind of files, for example, the form of the output of the loader and assembler.

Miscellaneous facilities in section 5 include user environment, graphics interface and terminal capability database.

If you find that the *Programmer Reference Manual* is rather prosaic reading, see section 6 for *games*.

Each section consists of a number of independent entries. The name of the entry is in the upper corner of the page, together with the section number, and sometimes a letter characteristic of a subcategory, e.g. graphics is 3G, and the math library is 3M. Entries within each section are alphabetized and the page numbers of each entry start at 1. The introductions to sections 2 and 3 should be perused before reading further in the section. These introductions also include a summary by subject of the relevant routines and cross references.

All entries are based on a common format, though not all subsections will always appear:

> The *name* subsection lists the exact names of the commands and subroutines covered under the entry and gives a very short description of their purpose.
>
> The *synopsis* summarizes the use of the program being described. A few conventions are used, particularly in the Commands subsection:

Boldface is reserved for literals, typed just as they appear, e.g. **termcap**(5). *Italics* are used for generic arguments which are to be replaced by real parameters.

Square brackets [ ] around an argument indicate that the argument is optional. When an argument is given as 'name', it always refers to a file name.

Ellipses '...' are used to show that the previous argument-prototype may be repeated.

A final convention is used by the commands themselves. An argument beginning with a minus sign '−' is often taken to mean some sort of option-specifying argument even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with '−'.

The *description* subsection discusses in detail the subject at hand.

The *files* subsection gives the names of files which are built into the program.

A *see also* subsection gives pointers to related information, such as other commands or tutorials.

A *diagnostics* subsection discusses the diagnostic indications which may be produced. Messages which are intended to be self-explanatory are not listed.

The *bugs* subsection gives known bugs and sometimes deficiencies. Occasionally the suggested fix is also described.

Contents

Chapter 1
A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make)

1.1 GENERAL

In a programming project, a common practice is to divide large programs into smaller pieces that are more manageable. The pieces may require several different treatments such as being processed by a macro processor or sophisticated program generators (e.g., **yacc** or **lex**). The project continues to become more complex as the output of these generators are compiled with special options and with certain definitions and declarations. A sequence of code transformations develops which is difficult to remember. The resulting code may need further transformation by loading the code with certain libraries under control of special options. Related maintenance activities also complicate the process further by running test scripts and installing validated modules. Another activity that complicates program development is a long editing session. A programmer may lose track of the files changed and the object modules still valid especially when a change to a declaration can make a dozen other files obsolete. The programmer must also remember to compile a routine that has been changed or that uses changed declarations.

The ''make'' is a software tool that maintains, updates, and regenerates groups of computer programs.

A programmer can easily forget

- Files that are dependent upon other files.
- Files that were modified recently.
- Files that need to be reprocessed or recompiled after a change in the source.
- The exact sequence of operations needed to make and exercise a new version of the program.

The many activities of program development and maintenance are made simpler by the **make** program.

The **make** program provides a method for maintaining up-to-date versions of programs that result from many operations on a number of files. The **make** program can keep track of 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 a program, the **make** command creates the proper files simply, correctly, and with a minimum amount of effort. The **make** program also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

The basic operation of **make** is to

- Find the name of the needed target file in the description.
- Ensure that all of the files on which it depends exist and are up to date.
- Create the target file if it has not been modified since its generators were modified.

The descriptor file really defines the graph of dependencies. The **make** program determines the necessary work by performing a depth-first search of the graph of dependencies.

If the information on interfile dependencies and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files regardless of the number edited since the last **make**. 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 . . .

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

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

```
# Description file for the Make command

p = lp
FILES = Makefile version.c defs main.c doname.c misc.c
        files.c dosys.c gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lS
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 version.c
        gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c
        version.c gram.c

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

The **make** program 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 files:

```
cc  −O  −c version.c
cc  −O  −c main.c
cc  −O  −c doname.c
cc  −O  −c misc.c
cc  −O  −c files.c
cc  −O  −c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  −O  −c gram.c
cc  version.o main.o doname.o misc.o files.o dosys.o
    gram.o  −lS  −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 entries in the description file are useful maintenance sequences. The "print" entry prints only the files 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 as follows:

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

1.2 BASIC FEATURES

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

To illustrate, consider a simple example in which a program named *prog* is made by compiling and loading three C language files *x.c*, *y.c*, and *z.c* with the **lS** library. By convention, the output of the C language compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, 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  -lS  -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*.

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

- A user-supplied description file
- File names and "last-modified" times from the file system
- Built-in rules to bridge some of the gaps.

In the example, the first line states that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line states that *x.o* and *y.o* depend on the file *defs*. 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).

By not taking advantage of **make**'s innate knowledge, the following longer descriptive file results.

```
prog : x.o  y.o  z.o
       cc  x.o  y.o  z.o  −lS  −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 have changed since the last time *prog* was made, all of the files are current, and the command

```
make
```

announces this fact and stops. If, however, the *defs* file has been edited, *x.c* and *y.c* (but not *z.c*) is recompiled; and then *prog* is created from the new ''*.o*'' files. If only the file *y.c* had changed, only it is recompiled; but it is still necessary to reload *prog*. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

```
make x.o
```

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

If the file exists after the commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions. A method, often useful to programmers, is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros. Thus, an entry ''save'' might be included to copy a certain set of files, or an entry ''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.

The **make** program 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. A $$ is a dollar sign.

The **$***, **$@**, **$?**, and **$<** are four special macros which change values during the execution of the command. (These four macros are described in the part "DESCRIPTION FILES AND SUBSTITUTIONS".) The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = −lS
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  −o prog
  . . .
```

The **make** command loads the three object files with the **lS** library. The command

```
make "LIBES= −ll −lS"
```

loads them with both the Lex (−ll) and the standard (−lS) libraries since macro definitions on the command line override definitions in the description. Remember to quote arguments with embedded blanks in VENIX software commands.

1.3 DESCRIPTION FILES AND SUBSTITUTIONS

A description file contains the following information:

- macro definitions
- dependency information
- executable commands.

The comment convention is that a sharp (#) and all characters on the same line after a sharp are ignored. Blank lines and lines beginning with a sharp (#) are totally ignored. If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

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 −lS
LIBES =
```

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

Macro definitions may also appear on the **make** command line while other lines give information about target files. The general form of an entry is

```
target1 [target2 . .] :[:] [dependent1 . .] [; commands] [# . .]
[(tab) commands] [# . . .]
 . . .
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as "*" and "?" are expanded. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a sharp (#) except when the sharp is in quotes or not including a new 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. For the usual single-colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default creation rule may be invoked. 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, 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 the silent mode or if the command line begins with an @ sign. **Make** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the −**i** flags have 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 VENIX software 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. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The **$@** macro is set to the full target name of the current target. The **$@** macro is evaluated only for explicitly named dependencies. The **$?** macro is set to the string of names that were found to be younger than the target. The **$?** macro is evaluated when explicit rules from the *makefile* are evaluated. If the command was generated by an implicit rule, the **$<** macro is the name of the related file that caused the action; and the **$*** macro is the prefix shared by the current and the dependent 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.

1.4 COMMAND USAGE

The **make** command takes macro definitions, flags, description file names, and target file names as arguments in the form:

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

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

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

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

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

−**r** Do not use the built-in rules.

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

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

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

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

−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, and the arguments 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".

1.5 SUFFIXES AND TRANSFORMATION RULES

The **make** program 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, the internal table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES". The **make** program searches 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** transforms a file with one suffix into a file with another suffix. 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 the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for ".SUFFIXES" in his own description file. The dependents are added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed. 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 $@
```

1.6 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 default suffix list is as follows:

.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.

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* were needed and there were an *x.c* in the description or directory, the *x.o* file would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, **make** would discard the intermediate C language file and use the direct link.

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
```

will cause the **newcc** command to be used instead of the usual C language 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 language compiler to be used.

1.7 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, nothing is done to the file *x.c* while file *x.o* must be recreated.

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

```
make −n
```

orders **make** to print out the commands which **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a 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.

Contents

Chapter 2
THE M4 MACRO PROCESSOR

2.1 GENERAL

The M4 macro processor is a front end for rational Fortran (Ratfor) and the C programming languages. The ‘‘#define’’ statement in C language and the analogous ‘‘define’’ in Ratfor are examples of the basic facility provided by any macro processor. At the beginning of a program, a symbolic name or symbolic constant can be defined as a particular string of characters. The compiler will then replace later unquoted occurrences of the symbolic name with the corresponding string. Besides the straightforward replacement of one string of text by another, the M4 macro processor provides the following features:

- arguments
- arithmetic capabilities
- file manipulation
- conditional macro expansion
- string and substring functions.

The basic operation of M4 is to read every alphanumeric token (string of letters and digits) input and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is rescanned. The user also has the capability to 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. A list of 21 built-in macros provided by the M4 macro processor can be found in Figure 2-1.

Macro Name	Function
changequote	Restores original characters or makes new quote characters the left and right brackets.
changescom	Changes left and right comment markers from the default # and new line.
deer	Returns the value of its argument decremented by 1.
define	Defines new macros.
defn	Returns the quoted definition of its argument(s).
divert	Diverts output to 1-out-of-10 diversions.

Figure 2-1. Built-in Macros (Sheet 1 of 4)

Macro Name	Function
divnum	Returns the number of the currently active diversion.
dnl	Reads and discards characters up to and including the next new line.
dumpdef	Dumps the current names and definitions of items named as arguments.
errprint	Prints its arguments on the standard error file.
eval	Prints arbitrary arithmetic on integers.
ifdef	Determines if a macro is currently defined.
ifelse	Performs arbitrary conditional testing.
include	Returns the contents of the file named in the argument. A fatal error occurs if the file name cannot be accessed.

Figure 2-2. Built-in Macros (Sheet 2 of 4)

Macro Name	**Function**
iner	Returns the value of its argument incremented by 1.
index	Returns the position where the second argument begins in the first argument of index.
len	Returns the number of characters that makes its argument.
m4exit	Causes immediate exit from M4.
m4wrap	Pushes the exit code back at final EOF.
maketemp	Facilitates making unique file names.
popdef	Removes current definition of its argument(s) exposing any previous definitions.
pushdef	Defines new macros but saves any previous definition.

Figure 2-3. Built-in Macros (Sheet 3 of 4)

Macro Name	Function
shift	Returns all arguments of shift except the first argument.
sinclude	Returns the contents of the file named in the arguments. The macro remains silent and continues if the file is inaccessible.
substr	Produces substrings of strings.
syscmd	Executes the VENIX System command given in the first argument.
traceoff	Turns macro trace off.
traceon	Turns the macro trace on.
translit	Performs character transliteration.
undefine	Removes user-defined or built-in macro definitions.
undivert	Discards the diverted text.

Figure 2-4. Built-in Macros (Sheet 4 of 4)

To use the M4 macro processor, input the following command:

```
m4 [optional files]
```

Each argument file is processed in order. If there are no arguments or if an argument is "–", the standard input is read at that point. The processed text is written on the standard output which may be captured for subsequent processing with the following input:

```
m4 [files] >outputfile
```

2.2 DEFINING MACROS

The primary built-in function of M4 is **define**. **Define** is used to define new macros. The following 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. Use of a slash may stretch *stuff* over multiple lines. Thus, as a typical example,

```
define(N, 100)
 ...
if (i > N)
```

defines *N* to be 100 and uses the symbolic constant *N* in a later **if** statement. The left parenthesis must immediately follow the word **define** to signal that **define** has arguments. If a user-defined macro or built-in name is not followed immediately by "(", it is assumed to have no arguments. Macro calls have the following general form:

```
name(arg1,arg2,...argn)
```

A macro name is only recognized as such if it appears surrounded by nonalphanumerics. Using the following example:

```
define(N, 100)
 ...
if (NNN > 100)
```

the variable *NNN* is absolutely unrelated to the defined macro *N* even though the variable contains a lot of *N*s. Macros may be defined in terms of other names. For example,

```
define(N, 100)
define(M, N)
```

defines both *M* and *N* to be 100. If *N* is redefined and subsequently changes, *M* retains the value of 100 not *N*. The M4 macro processor expands macro names into their defining text as soon as possible. The string *N* is immediately replaced by 100. Then the string *M* is also immediately replaced by 100. The overall result is the same as using the following input in the first place:

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now *M* is defined to be the string *N*, so when the value of *M* is requested later, the result is the value of *N* at that time (because the *M* will be replaced by *N* which will be replaced by 100). The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by left and right single quotes is not expanded immediately but has the quotes stripped off. The value of a quoted string is the string stripped of the quotes. If the input is

```
define(N, 100)
define(M, ‘N’)
```

the quotes around the *N* are stripped off as the argument is being collected. The results of using quotes is to define *M* 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 the word **define** is to appear in the output, the word must be quoted in the input as follows:

```
‘define’ = 1;
```

Another example of using quotes is redefining *N*. To redefine *N*, the evaluation must be delayed by quoting

```
define(N, 100)
 ...
define(‘N’, 200)
```

In M4, it is often wise to quote the first argument of a macro. The following example will not redefine *N*:

```
define(N, 100)
 ...
define(N, 200)
```

The *N* in the second definition is replaced by 100. The result is equivalent to the following statement:

```
define(100, 200)
```

This statement is ignored by M4 since only things that look like names can be defined. If left and right single quotes are not convenient for some reason, the quote characters can be changed with the following built-in macro:

```
changequote([, ])
```

The built-in **changequote** makes the new quote characters the left and right brackets. The original characters can be restored by using **changequote** without arguments as follows:

```
changequote
```

There are two additional built-ins related to **define**. The **undefine** macro removes the definition of some macro or built-in as follows:

```
undefine(‘N’)
```

The macro removes the definition of *N*. Built-ins can be removed with **undefine**, as follows:

```
undefine(‘define’)
```

But once removed, the definition cannot be reused. The built-in **ifdef** provides a way to determine if a macro is currently defined. Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef(‘pdp11’, ‘define(wordsize,16)’)
ifdef(‘u3b’, ‘define(wordsize,32)’)
```

Remember to use the quotes. The **ifdef** macro actually permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument. If the first argument is not defined, the value of **ifdef** is the third argument. If there is no third argument, the value of **ifdef** is null. If the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef(‘venix’, on VENIX, not on VENIX)
```

2.3 ARGUMENTS

So far the simplest form of macro processing has been discussed which is 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** is replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump** defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1. The ‘bump(x)’ statement is equivalent to ‘x = x + 1.’ A macro can have as many arguments as needed, but only the first nine are accessible (**$1** through **$9**). The macro name is **$0** although that is less commonly used. Arguments that are not supplied are replaced by null strings, so a macro can be defined which simply concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus, 'cat(x, y, z)' is equivalent to 'xyz'. Arguments **$4** through **$9** are null since no corresponding arguments were provided. Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a,   b   c)
```

defines 'a' to be 'b   c'. Arguments are separated by commas; however, when commas are within parentheses, the argument is not terminated nor separated. For example,

```
define(a, (b,c))
```

has only two arguments. The first argument is **a**. The second is literally **(b,c)**. A bare comma or parenthesis can be inserted by quoting it.

2.4 ARITHMETIC BUILT-INS

The M4 provides three built-in functions for doing arithmetic on integers (only). The simplest is **incr** which increments its numeric argument by 1. The built-in **decr** decrements by 1. Thus to handle the common programming situation where a variable is to be defined as "one more than *N*", use the following:

```
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. The operators in decreasing order of precedence are

```
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 32 bits under the VENIX operating system. As a simple example, define *M* to be "2==N+1" using **eval** as follows:

```
define(N, 3)
define(M, 'eval(2= =N+1)')
```

The defining text for a macro should be quoted unless the text is very simple. Quoting the defining text usually gives the desired result and is a good habit to get into.

2.5 FILE MANIPULATION

A new file can be included in the input at any time by the built-in function **include**. For example,

```
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 (include**'s replacement text) is the contents of the file. If needed, the contents can be captured in definitions, etc. A fatal error occurs if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used. The built-in **sinclude** (silent include) says nothing and continues if the file named cannot be accessed. The output of M4 can be diverted to temporary files during processing, and the collected material can be output upon command. The M4 maintains nine of these diversions, numbered 1 through 9. If the built-in macro

```
divert(n)
```

is used, all subsequent output is put onto the end of a temporary file referred to as *n*. Diverting to this file is stopped by the **divert** or **divert(0)** command which resumes the normal output process. Diverted text is normally output all at once at the end of processing with the diversions output in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The built-in **undivert** brings back all diversions in numerical order. The built-in **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted text (as does diverting) into a diversion whose number is not between 0 and 9, inclusive. The value of **undivert** is *not* the diverted text. Furthermore, the diverted material is *not* rescanned for macros. The built-in **divnum** returns the number of the currently active diversion. The current output stream is zero during normal processing.

2.6 SYSTEM COMMAND

Any program in the local operating system can be run by using the **syscmd** built-in. For example,

```
syscmd(date)
```

on the VENIX system 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*. The **maketemp** macro fills in a string of XXXXX in the argument with the process id of the current process.

2.7 CONDITIONALS

Arbitrary conditional testing is performed via built-in **ifelse**. In the simplest form

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, **ifelse** returns the string *c*. Otherwise, string *d* is returned. Thus, a macro called **compare** can be defined as one which compares two strings and returns "yes" or "no" if they are the same or different as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes which prevents evaluation of **ifelse** occurring too early. If the fourth argument is missing, it is treated as empty. The built-in **ifelse** can actually have any number of arguments and provides a limited form of multiway 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.

2.8 STRING MANIPULATION

The built-in **len** returns the length of the string (number of characters) 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. Using input, **substr(s, i, n)** returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. Inputting

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time.
```

If *i* or *n* are out of range, various actions occur. The built-in **index(s1, s2)** returns the index (position) in *s1* where the string *s2* occurs or −1 if it does not occur. As with **substr**, the origin for strings is 0. The built-in **translit** performs character transliteration and has the general form

```
translit(s, f, t)
```

which modifies *s* by replacing any character found in *f* by the corresponding character of *t*. Using input

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not 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)
```

would delete vowels from *s*. There is also a built-in called **dnl** that deletes all characters that follow it up to and including the next new line. The **dnl** macro is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. Using input

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new line at the end of each line that is not part of the definition. So the new line is copied into the output where it may not be wanted. If the built-in **dnl** is added to each of these lines, the newlines will disappear. Another method of achieving the same results is to input

```
divert(−1)
define(...)
 ...
divert.
```

2.9 PRINTING

The built-in **errprint** writes its arguments out on the standard error file. An example would be

```
errprint('fatal error')
```

The built-in **dumpdef** is a debugging aid that dumps the current names and definitions of items named as arguments. If no arguments are given, then all current names and definitions are printed. Do not forget to quote the names.

Contents

Chapter 3
THE awk PROGRAMMING LANGUAGE

3.1 GENERAL

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

- Generates reports
- Matches patterns
- Validates data
- Filters data for transmission.

3.2 PROGRAM STRUCTURE

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

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

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

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

For example, this *awk* program

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

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

An *awk* program has the following structure:

- a <BEGIN> section
- a <record> or main section
- an <END> section.

The <BEGIN> section is run before any input lines are read, and the <END> section is run after all the data files are processed. The <record> section is data driven. That is, it is the section that is run over and over for each separate line of input.

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

The words ''BEGIN'' and ''END'' are actually patterns recognized by *awk*. These are discussed further in the pattern section of this guide.

3.3 LEXICAL CONVENTION

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

1. *numeric constants*
2. *string constants*
3. *keywords*
4. *identifiers*
5. *operators*
6. *record and file tokens*
7. *comments*
8. *separators.*

3.3.1 Numeric Constants

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

3.3.2 String Constants

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

3.3.3 Keywords

Strings used as keywords are shown in Figure 3-1.

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

Figure 3-1. Strings Used as Keywords

3.3.4 Identifiers

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

3.3.5 Operators

The *awk* has assignment, arithmetic, relational, and logical operators similar to those in the C programming language and regular expression pattern matching operators similar to those in the VENIX operating system program *egrep* and *lex.*

Assignment operators are shown in Figure 3-2.

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

Figure 3-2. Symbols and Descriptions for Assignment Operators

Arithmetic operators are shown in Figure 3-3.

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

Figure 3-3. Symbols and Descriptions for Arithmetic Operators

Relational operators are shown in Figure 3-4.

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

Figure 3-4. Symbols and Descriptions for Relational Operators

Logical operators are shown in Figure 3-5.

Logical Operators	
Symbol	**Description**
&&	and
! !	or
!	not

Figure 3-5. Symbols and Descriptions for Logical Operators

Regular expression matching operators are shown in the Figure 3-6.

Regular Expression Pattern Matching Operators	
Symbol	**Description**
–	matches
!–	does not match

Figure 3-6. Symbols and Descriptions for Regular Expression Pattern

3.3.6 Record and Field Tokens

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

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

3.3.7 Record Separators

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

3.3.8 Field Separator

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

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

3.3.9 Multiline Records

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

3.3.10 Output Record and Field Separators

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

3.3.11 Comments

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

```
#    part of the line is a comment
```

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

3.3.12 Separators and Brackets

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

3.4 PRIMARY EXPRESSIONS

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

numeric constants
string constant
var
function

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

3.4.1 Numeric Constants

The format of a numeric constant was defined previously in **LEXICAL CONVENTIONS**. Numeric values are stored as floating point numbers. Both the numeric and string value of a numeric constant is the decimal number represented by the constant. The preferred value is the numeric value.

Numeric values for string constants are in Figure 3-7.

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

Figure 3-7. Numeric Values for String Constants

3.4.2 String Constants

The format of a string constant was defined previously in **LEXICAL CONVENTIONS**. The numeric value of a string constant is **0** unless the string is a numeric constant enclosed in double quotes. In this case, the numeric value is the number represented. The preferred value of a string constant is its string value. The string value of a string constant is always the string itself.

String values for string constants are in Figure 3-8.

String Constants		
String Constant	**Numeric Value**	**String Value**
’’’’	0	empty space
"a"	0	a
"XYZ"	0	xyz
"o"	0	0
"1"	1	1
".5"	0.5	.5
".5e2"	0.5	.5e2a

Figure 3-8. String Values for String Constants

3.4.3 Vars

A *var* is one of the following:

identifier
identifier{expression}
*$*term

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

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

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

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

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

3.4.4 Function

The *awk* has a number of built-in functions that perform common arithmetic and string operations.

The arithmetic functions are in Figure 3-9.

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

Figure 3-9. Built-in Functions for Arithmetic and String Operations

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

String functions are shown in Figure 3-10.

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

Figure 3-10. Expressions for String Functions

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

The function index (e1,e2) takes the string value of expressions e1 and e2 and returns the first position of where e2 occurs as a substring in e1. If e2 does not occur in e1, index returns 0. For example, **index ("abc", "bc")=2** and **index ("abc", "ac")=0.**

The function length without an argument returns the number of characters in the current input record. With an expression argument, length (e) returns the number of characters in the string value of e. For example, **length ("abc")=3** and **length (17)=2.**

The function split *(e array, sep)* splits the string value of expression e into fields that are then stored in *array [1], array [2],..., array [n]* using the string value of *sep* as the field separator. Split returns the number of fields found in *e.* The function split *(e, array)* uses the current value of **FS** to indicate the field separator. For example, after invoking n = split **($0)**, **a**[1], **a**[2],..., **a**[n] is the same sequence of values as **$1, $2..., $NF.**

The function splitf **(f, e1, e2...)** produces the value of expressions **e1, e2...** in the format specified by the string value of the expression **f.** The format control conventions are those of the printf statement in the C programming language **[KR].**

The function substr *(string, pos)* returns the suffix of *string* starting at position *pos*. The function substr *(string, pos, length)* returns the substring of *string* that begins at position *pos* and is *length* characters long. If pos + length is greater than the length of *string* then substr *(string, pos, length)* is equivalent to substr *(string, pos)*. For example, substr (″**abc**″, **2, 1)** = ″**b**″, substr (″**abc**″, **2, 2)** = ″**bc**″, and substr (″**abc**″, **2, 3)** = ″**bc**″. Positions less than 1 are taken as 1. A negative or zero length produces a null result.

The preferred value of sprintf and substr is string. The preferred value of the remaining string functions is numeric.

3.5 TERMS

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

primary expression
term binop term
unop term
incremented var
(term)

3.5.1 Binary Terms

In a *term* of the form

term1
binop
term2

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

3.5.2 Unary Term

In a *term* of the form

unop term

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

3.5.3 Incremented Vars

An *incremented var* has one of the forms

+ + *var*
− − *var*
var + +
var − −

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

3.5.4 Parenthesized Terms

Parentheses are used to group terms in the usual manner.

3.6 EXPRESSIONS

An *awk* expression is one of the following:

term
term term ...
var asgnop expression

3.6.1 Concatenation of Terms

In an expression of the form *term1 term2 ...,* the string value of the terms are concatenated. The preferred value of the resulting expression is a string value that can be interpreted as a numeric value. Concatenation of terms has lower precedence than binary + and −. For example, 1+2 3+4 has the string (and numeric) value 37.

3.6.2 Assignment Expressions

An *assignment expression* is one of the forms *var asgnop expression*

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

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

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

In an expression of the form

var = expression

the numeric and string value of *var* becomes those of *expression.*

var op = expression

is equivalent to

var = var op expression

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

```
b = b * (c−2)
a = a + b
```

3.7 USING *awk*

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

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

   awk ' program ' files

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

   awk ' /x/ {print } ' files

   to run the *awk* script /x/ {print} on the input file "files". If no input files are specified, *awk* takes input from the standard input **stdin.** You can also specify that input comes from **stdin** by using ''-'' (the hyphen) as one of the files. The pattern-action statement

```
awk 'program' files -
```

looks for input from ''files'' and from **stdin** and processes first from ''files'' and then from **stdin.**

2. Alternately, if your *awk* program is long, it is more convenient to put the program in a separate file, awkprog, and tell *awk* to fetch it from there. This is done by using the ''–f'' option after the *awk* command as follows:

   ```
   awk –f awkprog files
   ```

   where ''files'' is an optional list of input files that may include **stdin** as is indicated by a hyphen (-).

For example:

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

prints

```
            hello, world
```

on the standard output when given to the shell. Recall that the word ''BEGIN'' is a special pattern indicating that the action following in braces is run before any data is read. Words ''print'' and ''exit'' are both discussed in later sections.

This *awk* program could be run by putting

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

in a file named awkprog , and then the command

```
            awk – f awkprog
```

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

3.8 INPUT: RECORDS AND FIELDS

The *awk* reads its input one record at a time unless changed by you. A record is a sequence of characters from the input ending with a newline character or with an end of file. Thus, a record is a line of input. The *awk* program reads in characters until it encounters a newline or end of file. The string of characters, thus read, is assigned to the variable **$0**. You can change the character that indicates the end of a record by assigning a new character to the special variable **RS** (the record separator). Assignment of values to variables and these special variables such as **RS** are discussed later.

Once *awk* has read in a record, it then splits the record into "fields". A field is a string of characters separated by blanks or tabs, unless you specify otherwise. You may change field separators from blanks or tabs to whatever characters you choose in the same way that record separators are changed. That is, the special variable FS is assigned a different value.

As an example, let us suppose that the file "countries" contains the area in thousands of square miles, the population in millions, and the continent for the ten largest countries in the world. (Figures are from 1978; Russia is placed in Asia.)

Sample Input File "countries":

```
Russia      8650   262   Asia
Canada      3852   24    North America
China       3692   866   Asia
USA         3615   219   North America
Brazil      3286   116   South America
Australia   68     14    Australia
India       1269   637   Asia
Argentina   72     26    South America
Sudan       968    19    Africa
Algeria     920    18    Africa
```

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

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

```
Russia  8650    262     Asia
```

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

For example, the following input:

```
{print $0}
```

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

$1 is equal to the string ″Russia″
$2 is equal to the integer 8650
$3 is equal to the integer 262
$4 is equal to the string ″Asia″
$5 is equal to the null string
... and so forth.

To print the continent, followed by the name of the country, followed by its population, use the following *awk* script:

```
{print $4, $1, $3}
```

Note that *awk* does not require type declarations.

3.9 INPUT: FROM THE COMMAND LINE

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

```
x=5
```

This statement in an *awk* program assigns the value 5 to the variable x. It is also possible to assign values to variables from the command line. This provides another way to supply input values to *awk* programs.

For example

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

will print the value 5 on the standard output. The minus sign at the end of this command is necessary to indicate that input is coming from **stdin** instead of a file called "x=5". Similarly if the input comes from a file named "file", the command is

```
awk '{print x}' file
```

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

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

```
awk -f awk.program RS=":" file
```

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

This operation is so common that there is yet another way to change the field separator from the command line. There is a separate option "–Fx" that is placed directly after the command *awk*. This changes the field separator from blank or tab to the character "x".

For example

```
awk -F:   -f  awk.program file
```

changes the field separator FS to the character ":". Note that if the field separator is specifically set to a tab, (that is, with the –F option or by making a direct assignment to FS) then blanks are recognized by *awk* as separating fields. However, even if the field separator is specifically set to a blank, tabs are STILL recognized by *awk* as separating fields.

An exercise:

Using the input file (“countries” described earlier) write an *awk* script that prints the name of a country followed by the continent that it is on. Do this in such a way that continents composed of two words (e. g., North America) are processed as only one field and not two.

3.10 OUTPUT: PRINTING

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

```
{print}
```

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

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

prints the name of the country and the population:

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

Note that the use of a semicolon at the end of statements in *awk* programs is optional. *Awk* accepts

```
{print $1 }
```

and

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

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

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

Parentheses are also optional with the print statement.

```
print $3, $2
```

is the same as

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

Items separated by a comma in a print statement are separated by the current output field separators (normally spaces, even though the input is separated by tabs) when printed. The **OFS** is another special variable that can be changed by you. These special variables are summarized in a later section.

An exercise:

Using the input file, ‘‘countries’’, print the continent followed by the country followed by the population for each input record. Then pipe the output to the VENIX operating system command ‘‘sort’’ so that all countries from a given continent are printed together.

Print also prints strings directly from your programs with the *awk* script

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

from an earlier section.

An exercise:

Print a header to the output of the previous exercise that says ‘‘Population of Largest Countries’’ followed by headers to the columns that follow describing what is in that column, for example, Country or Population.

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

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

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

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

and the program

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

prints

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

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

```
print ""
```

to print the empty line.

Awk also provides the statement printf so that you can format output as desired. Print uses the default format ''%.6g'' for each variable printed.

```
printf format, expr, expr, ...
```

formats the expressions in the list according to the specification in the string, format, and prints them. The format statement is exactly that of the printf in the C library. For example,

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

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

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

With printf, no output separators or newlines are produced automatically. You must add them as in this example. In the C library version of printf, the various escape characters '**\n**', '**\t**', '**\b**' (backspace) and '**\r**' (carriage return) are valid with the *awk* printf.

There is a third way that printing can occur on standard output when a pattern is specified but there is no action to go with it. In this case, the entire record **$0** is printed. For example, the program

```
/x/
```

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

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

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

which prints

```
                    hello world
```

However, without the comma in the print statement as

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

you get

```
                    helloworld
```

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

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

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

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

both of these last two scripts yields

```
                    hello, world
```

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

3.11 OUTPUT: TO DIFFERENT FILES

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

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

The flow of control statements (for example, "if") are discussed later.

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

```
print > "FILE"
```

where FILE is the name of the file receiving the data, and the print statement may have any legal arguments to it.

Notice that the file names are quoted. Without quotes, the file names are treated as uninitialized variables and all output then goes to the same file.

If > is replaced by >>, output is appended to the file rather than overwriting it.

Users should also note that there is an upper limit to the number of files that are written in this way. At present it is ten.

3.12 OUTPUT: TO PIPES

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

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

where "mary" is someone's login name, any record is sent (with the second field equal to "XX") to the user, mary, as mail. Awk waits until the entire program is run before it executes the command that was piped to, in this case the "mail" command.

For example:

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

takes the first field of each input record, sorts these fields, and then prints them. The command in parentheses is any VENIX operating system command.

An exercise:

Write an *awk* script that uses the input file to

- List countries that were used previously
- Print the name of the countries
- Print the population of each country
- Sort the data so that countries with the largest population appear first
- Mail the resulting list to yourself.

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

```
print ... | "cat -u > /dev/tty"
```

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

3.13 COMMENTS

Comments are placed in *awk* programs; they begin with the character # and end with the end of the line as in

```
print x, Y        # this is a comment
```

3.14 PATTERNS

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

- Regular expressions
- Arithmetic relational expressions
- String valued expressions
- Combinations of these.

3.14.1 BEGIN and END

The special pattern, BEGIN, matches the beginning of the input before the first record is read. The pattern, END, matches the end of the input after the last line is processed. BEGIN and END thus provide a way to gain control before and after processing for initialization and wrapping up.

An example:

As you have seen, you can use BEGIN to put column headings on the output

```
BEGIN {print "Country", "Area", "Population", "Continent"}
      {print}
```

which produces

```
Country Area Population Continent

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

Formatting is not very good here; printf would do a better job and is usually mandatory if you really care about appearance.

Recall also, that the BEGIN section is a good place to change special variables such as **FS** or **RS**.

Example:

```
BEGIN { FS= "   "
      print "Countries", "Area", "Population", "Continent"
      }
      {print}
END     {print "The number of records is", NR}
```

In this program, **FS** is set to a tab in the BEGIN section and as a result all records (in the file countries) have exactly four fields.

Note that if BEGIN is present it is the first pattern; END is the last if it is used.

3.14.2 Relational Expressions

An *awk* pattern is any expression involving comparisons between strings of characters or numbers. For example, if you want to print only countries with more than 100 million population, use

```
   $3 >100
```

This tiny *awk* program is a pattern without an action so it prints each line whose third field is greater than 100 as follows:

```
Russia   8650   262   Asia
China    3692   866   Asia
USA      3615   219   North America
Brazil   3286   116   South America
India    1269   637   Asia
```

To print the names of the countries that are in Asia, type

```
   $4 == "Asia" {print $1}
```

which produces

```
Russia
China
India
```

The conditions tested are <, < =, = =, ! =, > =, and >. In such relational tests if both operands are numeric, a numerical comparison is made. Otherwise, the operands are compared as strings. Thus,

```
$1 > = "S"
```

selects lines that begin with S, T, U, and so forth which in this case is

```
USA    3615  219   North America
Sudan  968   19    Africa
```

In the absence of other information, fields are treated as strings, so the program

```
$1 = = $4
```

compares the first and fourth fields as strings of characters and prints the single line

```
Australia      2968     14 Australia
```

If fields appear as numbers, the comparisons are done numerically.

3.14.3 Regular Expressions

Awk provides more powerful capabilities for searching for strings of characters than were illustrated in the previous section. These are regular expressions. The simplest regular expression is a literal string of characters enclosed in slashes.

```
/Asia/
```

This is a complete *awk* program that prints all lines which contain any occurrence of the name ‘‘Asia’’. If a line contains ‘‘Asia’’ as part of a larger word like ‘‘Asiatic’’, it is also printed (but there are no such words in the countries file.)

Awk regular expressions include

- Regular expression forms found in the text editor
- **ed** and the pattern finder
- *grep* in which certain characters have special meanings.

For example, we could print all lines that begin with **A** with

```
/^A/
```

or all lines that begin with A, B, or C with

```
/^[ABC]/
```

or all lines that end with "ia" with

```
/ia$/
```

In general, the circumflex (^) indicates the beginning of a line. The dollar sign (**$**) indicates the end of the line and characters enclosed in brackets ,{ }, match any one of the characters enclosed. In addition, *awk* allows parentheses for grouping, the pipe (|) for alternatives, + for "one or more" occurrences, and ? for "zero or one" occurrences. For example,

```
/x|y/ {print}
```

prints all records that contain either an "x" or a "y".

```
/ax+b/      {print}
```

prints all records that contain an "a" followed by one or more "x's" followed by a "b". For example, axb, Paxxxxxxxb, QaxxbR.

```
/ax?b/      {print}
```

prints all records that contain an "a" followed by zero or one "x" followed by a "b". For example: ab, axb, yaxbPPP, CabD.

The two characters “.” and “*” have the same meaning as they have in *ed*: namely, “.” can stand for any character and “*” means zero or more occurrences of the character preceding it. For example,

```
    /a.b/
```

matches any record that contains an “a” followed by any character followed by a “b”. That is, the record must contain an “a” and a “b” separated by exactly one character. For example, /a.b/ matches axb, aPb and xxxxaXbxx, but NOT ab, axxb.

```
    /ab*c/
```

matches a record that contains an “a” followed by zero or more “b”’s followed by a “c”. For example, it matches

```
    ac
    abc
    pqrabbbbbbbbbbc901
```

Just as in *ed,* it is possible to turn off the special meaning of these metacharacters such as “^” and “*” by preceding these characters with a backslash. An example of this is the pattern

```
    /\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) by using the operators ~ or !~. For example, with the input file countries as before, the program

```
$1 ~ /ia$/      {print $1}
```

prints all countries whose name ends in “ia”:

```
    Russia
    Australia
    India
    Algeria
```

that is indeed different from *lines* which end in "ia".

3.14.4 Combinations of Patterns

A pattern is made up of similar patterns combined with the operators || (OR), && (AND), ! (NOT), and parentheses. For example,

```
    $2 >= 3000 && $3 >=100
```

selects lines where both area AND population are large. For example,

```
Russia   8650   262   Asia
China    3692   866   Asia
USA      3615   219   North America
Brazil   3286   116   South America
```

while

```
    $4 == "Asia" || $4 == "Africa"
```

selects lines with Asia or Africa as the fourth field. An alternate way to write this last expression is with a regular expression:

```
    $1 ~ /^(Asia|Africa))$/
```

&& and || guarantee that their operands are evaluated from left to right; evaluation stops as soon as truth or falsehood is determined.

3.14.5 Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma as in

```
    pattern1, pattern2        { ... }
```

In this case, the action is performed for each line between an occurrence of pattern1 and the next occurrence of pattern2 (inclusive). As an example with no action

```
    /Canada/,/Brazil/
```

prints all lines between the one containing "Canada" and the line containing "Brazil". For example,

```
Canada   3852   24    North America
China    3692   866   Asia
USA      3615   219   North America
Brazil   3286   116   South America
```

while

```
    NR = = 2, NR = = 5 { ... }
```

does the action for lines 2 through 5 of the input. Different types of patterns are mixed as in

```
    /Canada/, $4 = = "Africa"
```

and prints all lines from the first line containing "Canada" up to and including the next record whose fourth field is "Africa".

Users should note that patterns in this form occur OUTSIDE of the action parts of the *awk* programs (outside of the braces that define *awk* actions). If you need to check patterns inside an awk action (inside the braces), use a flow of control statement such as an "if" statement or a "while" statement. Flow of control statements are discussed in the part "BUILT-IN FUNCTIONS".

3.15 ACTIONS

An *awk* action is a sequence of action statements separated by newlines or semicolons. These action statements do a variety of bookkeeping and string manipulating tasks.

3.15.1 Variables, Expressions, and Assignments

The *awk* provides the ability to do arithmetic and to store the results in variables for later use in the program. However, variables can also store strings of characters. You cannot do arithmetic on character strings, but you can stick them together and pull them apart as shown. As an example, consider printing the population density for each country in the file countries.

```
    {print $1, (1000000 * $3)/($2 * 1000) }
```

(Recall that in this file the population is in millions and the area in thousands.) The result is population density in people per square mile.

```
Russia 30.289
Canada 6.23053
China 234.561
USA 60.5809
Brazil 35.3013
Australia 4.71698
India 501.97
Argentina 24.2537
Sudan 19.6281
Algeria 19.5652
```

The formatting is bad; so using printf instead gives the program

```
    {printf "%10s %6.1f0, $1, (1000000 * $3)/($2 * 1000) }
```

and the output

```
   Russia     30.3
   Canada      6.2
    China    234.6
      USA     60.6
   Brazil     35.3
Australia      4.7
    India    502.0
Argentina     24.3
    Sudan     19.6
  Algeria     19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are +. −, *, / and % (mod or remainder).

To compute the total population and number of countries from Asia, we could write

```
/Asia/   { pop = pop + $3; n = n + 1 }
END      {print "total population of", n, "Asian countries is", pop }
```

which produces total population of three Asian countries is 1765.

Actually, no experienced programmer would write

```
   {pop = pop + $3; n = n + 1 }
```

since both assignments are written more clearly and concisely. The better way is

```
   {pop += $3; ++n }
```

Indeed, these operators, ++, −−, −=, /=, * =, +=, and %= are available in *awk* as they are in C. Operator x += y has the same effect as x = x + y but += is shorter and runs faster. The same is true of the ++ operator; it adds one to the value of a variable. The increment operators ++ and −− (as in C) are used as prefix or as postfix operators. These operators are also used in expressions.

3.15.2 Initialization of Variables

In the previous example, we did not initialize pop nor n; yet, everything worked properly. This is because (by default) variables are initialized to the null string which has a numerical value of 0. This eliminates the need for most initialization of variables in BEGIN sections. We can use default initialization to advantage in this program which finds the country with the largest population.

```
maxpop < $3 {
        maxpop = $3
        country = $1
        }
END     {print country, maxpop}
```

which produces

```
China 866
```

3.15.3 Field Variables

Fields in *awk* share essentially all of the properties of variables. They are used in arithmetic and string operations and may be assigned to and initialized to the null string. Thus, divide the second field by 1000 to convert the area to millions of square miles by

```
{ $2 /= 1000; print }
```

or process two fields into a third with

```
BEGIN  { FS = "       "}
        { $4 = 1000 * $3 / $2; print }
```

or assign strings to a field as in

```
/USA/   { $1 = "United States" ; print }
```

which replaces USA by United States and prints the effected line

```
    United States 3615 219 North America
```

Fields are accessed by expressions; thus, $NF is the last field and $(NF−1) is the second to the last. Note that the parentheses are needed since $NF−1 is 1 less than the values in the last field.

3.15.4 String Concatenation

Strings are concatenated by writing them one after the other as in the following example:

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

prints the usual

```
                    hello, world
```

With input from the file ''countries'', the following program:

```
    /A/       { s = s " " $1 }
    END        { print s }
```

prints

```
    Australia Argentina Algeria
```

Variables, string expressions, and numeric expressions may appear in concatenations; the numeric expressions are treated as strings in this case.

3.15.5 Special Variables

Some variables in *awk* have special meanings. These are detailed here and the complete list given.

NR   Number of the current record.

NF Number of fields in the current record.

FS Input field separator, by default it is set to a blank or tab.

RS Input record separator, by default it is set to the newline character.

$i The ith input field of the current record.

$0 The entire current input record.

OFS Output field separator, by default it is set to a blank.

ORS Output record separator, by default it is set to the newline character.

OFMT The format for printing numbers, with the print statement, by default is "%.6g".

FILENAME The name of the input file currently being read. This is useful because *awk* commands are typically of the form

```
awk -f program file1 file2 file3 ...
```

3.15.6 Type

Variables (and fields) take on numeric or string values according to context. For example, in

```
pop += $3
```

pop is presumably a number, while in

```
country = $1
```

country is a string. In

```
maxpop < $3
```

the type of maxpop depends on the data found in $3. It is determined when the program is run.

In general, each variable and field is potentially a string or a number or both at any time. When a variable is set by the assignment

```
v = expr
```

its type is set to that of expr. (Assignment also includes +=, ++, −=, and so forth.) An arithmetic expression is of the type, “number”; a concatenation of strings is of type “string”. If the assignment is a simple copy as in

```
v1 = v2
```

then the type of v1 becomes that of v2.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to strings if necessary and the comparison is made on strings.

The type of any expression is coerced to numeric by subterfuges such as

```
expr + 0
```

and to string by

```
expr ""
```

This last expression is string concatenated with the null string.

3.15.7 Arrays

As well as ordinary variables, *awk* provides 1-dimensional arrays. Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value including non-numeric strings.

As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input line to the NRth element of the array x. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the following *awk* program:

```
        { x[NR] = $0 }
   END    { ... program ... }
```

The first line of this program records each input line into the array x. In particular, the following program

```
   { x[NR] = $1}
```

(when run on the file countries) produces an array of elements with

```
            x[1] = "Russia"
            x[2] = "Canada"
            x[3] = "China"
                 ... and so forth.
```

Arrays are also indexed by non-numeric values that give *awk* a capability rather like the associative memory of Snobol tables. For example, we can write

```
/Asia/    {pop["Asia"] += $3 }
/Africa/{pop[Africa] += $3 }
END       print "Asia=" pop["Asia"], "Africa="pop["Africa"] }
```

which produces

```
   Asia=1765 Africa=37
```

Notice the concatenation. Also, any expression can be used as a subscript in an array reference. Thus,

```
   area[$1] = $2
```

uses the first field of a line (as a string) to index the array area.

3.16 BUILT IN FUNCTIONS

The function

```
length
```

is provided by *awk* to compute the length of a string of characters. The following program prints each record preceded by its length:

```
{print length, $0 }
```

In this case (the variable) length means length($0), the length of the present record. In general, length(x) will return the length of x as a string.

Example:

With input from the file countries, the following *awk* program will print the longest country name:

```
length($1) > max  {max = length($1); name = $1 }
END               {print name}
```

The function

split

split (s, array) assigns the fields of the string ‘‘s’’ to successive elements of the array, ‘‘array’’.

For example;

```
split("Now is the time", w)
```

assigns the value ‘‘Now’’ to w[1], ‘‘is’’ to w[2], ‘‘the’’ to w[3] and ‘‘time’’ to w[4]. All other elements of the array w[ ], if any, are set to the null string. It is possible to have a character other than a blank as the separator for the elements of w. For this, use split with three elements.

```
n = split(s, array, sep)
```

This splits the string s into array[1], ..., array[n]. The number of elements found is returned as the value of split. If the sep argument is present, its first character is used as the field separator; otherwise, FS is used. This is useful if in the middle of an *awk* script, it is necessary to change the record separator for one record.

Also provided by the *awk* are the

Math Functions

sqrt,
log,
exp
int,

They provide the square root function, the base e logarithm function, exponential and integral part functions. This last function returns the greatest integer less than or equal to its argument. These functions are the same as those of the C library (*int* corresponds to the libc *floor* function) and so they have the same return on error as those in libc. (See the *Programmer Reference Manual*.) The substring function

substr

substr(s,m,n) produces the substring of s that begins at position m and is at most n characters long. If the third argument (n in this case) is omitted, the substring goes to the end of s. For example, we could abbreviate the country names in the file countries by

```
{ $1 = substr($1, 1, 3); print }
```

which produces

Rus	8650	262	Asia
Can	3852	24	North America
Chi	3692	866	Asia
USA	3615	219	North America
Bra	3286	116	South America
Aus	2968	14	Australia
Ind	1269	637	Asia
Arg	1072	26	South America
Sud	968	19	Africa
Alg	920	18	Africa

If s is a number, substr uses its printed image; substr(123456789,3,4) = 3456.

The function

```
index:
```

index (s1,s2) returns the leftmost position where the string s2 occurs in s1 or zero if s2 does not occur in s1.

The function

```
sprintf
```

formats expressions as the printf statement does but will assign the resulting expression to a variable instead of sending the results to stdout. For example,

```
x = sprintf( "%10s %6d ", $1, $2 )
```

sets x to the string produced by formatting the values of $1 and $2. The x is then used in subsequent computations.

The function

```
getline
```

immediately reads the next input record. Fields NR and $0 are all set but control is left at exactly the same spot in the *awk* program. Getline returns 0 for the end of file and a 1 for a normal record.

3.17 FLOW OF CONTROL

The *awk* provides the basic flow of control statements

- **if-else**
- **while**
- **for**

with statement grouping as in C language.

The **if** statement is used as follows:

```
if ( condition ) statement1 else statement2
```

The condition is evaluated; and if it is true, statement1 is executed; otherwise, statement2 is executed. The else part is optional. Several statements enclosed in braces ({,}) are treated as a single statement. Rewriting the maximum population computation from the pattern section with an if statement results in

```
{       if (maxpop < $3) {
                maxpop= $3
                country= $1
                }       }
END     { print country, maxpop }
```

There is also a while statement in *awk*.

```
while ( condition ) statement
```

The condition is evaluated; if it is true, the statement is executed. The condition is evaluated again, and if true, the statement is executed. The cycle repeats as long as the condition is true. For example, the following prints all input fields one per line:

```
{    i = 1
     while (i < = NF) {
           pint $i
           + +i
           }
     }
```

Another example is the Euclidean algorithm for finding the greatest common divisor of $1 and $2:

```
{printf "the greatest common divisor of " $1 "and ", $2, "is"
while ($1 ! = $2) {
                                  if ($1 > $2) $1 = $1 - $2
                                  else    $2 = $2 - $1
                                  }
printf $1 "\n"
}
```

The **for** statement is like that of C.

```
for ( expression1 ; condition ; expression2 ) statement
```

has the same effect as

```
expression1
while (condition) {
          statement
          expression2
          }
```

so

```
{     for (i=1 ; i < = NF; i+ +)
          print $i
}
```

is another *awk* program that prints all input fields one per line.

This is an alternate form of the **or** statement that is suited for accessing the elements of an associative array as is in *awk*.

```
for (i in array) statement
```

executes statement with the variable i set in turn to each subscript of array. The subscripts are each accessed once but in random order. Chaos will ensue if the variable i is altered or if any new elements are created within the loop. For example, you could use the "for" statement to print the record number followed by the record of all input records after the main program is executed.

```
      { x[NR] = $0 }
END     { for(i in x) { print i, x[i] }
```

A more practical example is the following use of strings to index arrays to add the populations of countries by continents:

```
BEGIN  {FS=""}
      {population[$4] = + $3}
END     {for(i in population)
             print i, population[i]
      }
```

In this program, the body of the **for** loop is executed for *i* equal to the string "Asia", then for *i* equal to the string "North America", and so forth until all the possible values of i are exhausted; that is, until all the strings of names of countries are used. Note, however, the order the loops are executed is not specified. If the loop associated with "Canada" is executed before the loop associated with the string "Russia", such a program produces

```
South America 26
Africa 16
Asia 637
Australia 14
North America 219
```

Note that the expression in the condition part of an **if, while**, or, **for** statement can include relational operators like <, <=, >, >=, ==, and !=; it can include regular expressions that are used with the "matching" operators ~ and !~; it can include the logical operators ||, &&, and !; and it also include parentheses for grouping.

The **break** statement (when it occurs within a **while** or **for** loop) causes an immediate exit from the **while** or **for** loop.

The **continue** statement (when it occurs within a **while** or **for** loop) causes the next iteration of the loop to begin.

The **next** statement in an *awk* program causes awk to skip immediately to the next record and begin scanning patterns from the top of the program. (Note the difference between getline and next. Getline does not skip to the top of the *awk* program.)

If an **exit** statement occurs in the BEGIN section of an *awk* program, the program stops executing and the END section is not executed (if there is one).

An **exit** that occurs in the main body of the *awk* program causes execution of the main body of the *awk* program to stop. No more records are read, and the END section is executed.

An **exit** in the END section causes execution to terminate at that point.

3.18 REPORT GENERATION

The flow of control statements in the last section are especially useful when *awk* is used as a report generator. *Awk* is useful for tabulating, summarizing, and formatting information. We have seen an example of *awk* tabulating in the last section with the tabulation of populations. Here is another example of this. Suppose you have a file "prog.usage" that contains lines of three fields; name, program, and usage:

```
Smith   draw    3
Brown   eqn     1
Jones   nroff   4
Smith   nroff   1
Jones   spell   5
Brown   spell   9
Smith   draw    6
```

The first line indicates that Smith used the draw program three times. If you want to create a program that has the total usage of each program along with the names in alphabetical order and the total usage, use the following program, called *list.a*:

```
      { use[$1 "" $2] += $3}
END     {for (np in use)
           print np "         " use[np] | "sort +0 +2nr" }
```

This program produces the following output when used on the input file, *prog.usage*.

```
Brown    eqn     1
Brown    spell   9
Jones    nroff   4
Jones    spell   5
Smith    draw    9
Smith    nroff   1
```

If you would like to format the previous output so that each name is printed only once, pipe the output of the previous *awk* program into the following program, called ``format.a:

```
   {      if ($1 != prev) {
                print $1 ":"
                rev = $1
                }
          print " " $2 " " $3
   }
```

The variable *prev* prints the unique values of the first field. The command

```
   awk -f list.a prog.usage | awk -f format.a
```

gives the output

```
Brown:
        eqn     1
        spell   9
Jones:
        nroff   4
        spell   5
Smith:
        draw    9
        nroff   1
```

It is often useful to combine different *awk* scripts and other shell commands such as *sort* as was done in the last script.

3.19 COOPERATION WITH THE SHELL

Normally, an *awk* program is either contained in a file or enclosed within single quotes as in

```
awk '{print $1}' ...
```

Awk uses many of the same characters that the shell does, such as $ and the double quote. Surrounding the program by ' ... ' ensures that the shell passes the *awk* program to *awk* intact.

Consider writing an *awk* program to print the nth field, where n is a parameter determined when the program is run. That is, we want a program called field such that

```
field n
```

runs the *awk* program

```
awk '{print $n}'
```

How does the value of n get into the *awk* program?

There are several ways to do this. One is to define field as follows:

```
awk '{print $'$1'}'
```

Spaces are critical here: as written there is only one argument, even though there are two sets of quotes. The $1 is outside the quotes, visible to the shell, and therefore substituted properly when field is invoked.

Another way to do this job relies on the fact that the shell substitutes for $ parameters within double quotes.

```
awk "{print $1}"
```

Here the trick is to protect the first $ with a \ \; the $1 is again replaced by the number when field is invoked.

This kind of trickery is extended in remarkable ways, but it is hard to understand quickly.

3.20 MISCELLANEOUS HINTS

You can simulate the effect of multidimensional arrays by creating your own subscripts. For example,

```
for ( i = 1; i < = 10; i+ +)
        for ( j = 1; j < = 10; j+ +)
                mult[i "," j] = . . .
```

creates an array whose subscripts have the form i,j; that is, 1,1; 1,2; and so forth and thus simulate a 2-dimensional array.

Contents

Chapter 4
ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (bc)

4.1 GENERAL

The arbitrary precision desk calculator language (**bc**) is a language and compiler for doing arbitrary precision arithmetic under the VENIX operating system. The output of the compiler is interpreted and executed by a collection of routines that can input, output, and do arithmetic on infinitely large integers and on scaled fixed-point numbers. These routines are based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The **bc** language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution. A small collection of library functions is also available, including *sin, cos, arctan, log, exponential,* and *Bessel* functions of integer order.

The **bc** compiler was written to make conveniently available a collection of routines (called **dc**) that are capable of doing arithmetic on integers of arbitrary size. The compiler is not intended to provide a complete programming language. It is a minimal language facility.

Some of the uses of this compiler are:

- Compile large integers
- Compute accurately to many decimal places
- Convert numbers from one base to another base.

There is a scaling provision that permits the use of decimal point notation. Provision is also made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal eight.

The actual limit on the number of digits that can be handled depends on the amount of core storage available. This is possible even on the smallest versions of the VENIX operating system.

The syntax of **bc** is very similar to that of the C language. This enables users who are familiar with C language to easily work with **bc**.

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the addition of two numbers (with the + operator) such as

```
142857 + 285714
```

the program responds immediately with the sum

```
428571.
```

The operators −, *, /, %, and ^ can also be used. They indicate subtraction, multiplication, division, remaindering, and integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the **unary** minus sign). The expression

```
        7+−3
```

is interpreted to mean that −3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in power, then *, %, and /, and finally, + and −. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right.

```
a^b^c  and  a^(b^c)
```

are equivalent as are the two expressions

```
a*b*c and (a*b)*c.
```

However, **bc** shares with Fortran and C language the undesirable convention that

```
a/b*c  is equivalent to  (a/b)*c.
```

Internal storage registers to hold numbers have single lowercase letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named *x*. When, as in this case, the outermost operator is an "=", the assignment is performed; but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (see the part on "SCALING"). Entering the lines

```
x = sqrt(191)
x
```

produces the printed result

```
13
```

4.2 BASES

There are two special internal quantities; **ibase** (input base) and **obase** (output base). The contents of **ibase**, initially set to 10 (decimal), determines the base used for interpreting numbers read in. For example, the input lines

```
ibase = 8
11
```

produces the output line

```
9
```

and the system is ready to do octal to decimal conversions. Beware, however, of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement has no effect. For dealing in hexadecimal notation, the characters A through F are permitted in numbers (regardless of what base is in effect) and are interpreted as digits having values 10 through 15, respectively. The statement

```
ibase = A
```

changes the base to decimal regardless of what the current input base is. Negative and large positive input bases are permitted but are useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The content of **obase**, initially 10 (decimal), is used as the base for output numbers. The input lines

```
obase = 16
1000
```

produces the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted and are sometimes useful. For example, large numbers can be output in groups of five digits by setting **obase** to 100000. Strange output bases (i.e., 1, 0, or negative) are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with a backslash (\). Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Nondecimal output conversion of a 100-digit number takes about 3 seconds.

The **ibase** and **obase** have no effect on the course of internal computation or on the evaluation of expressions. They only affect input and output conversions, respectively.

4.3 SCALING

A third special internal quantity called **scale** is used to determine the scale of calculated quantities. The number of digits after the decimal point of a number is referred to as its scale. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations.

The contents of **scale** must be no greater than 99 and no less than 0. It is initially set to 0. However, appropriate scaling can be arranged when more than 99 fraction digits are required.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

- Addition and subtraction—The scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result.
- Multiplication—The scale of the result is never less than the maximum of the two scales of the operands and never more than the sum of the scales of the operands. Subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity **scale**.
- Division—The scale of a quotient is the contents of the internal quantity **scale**. The scale of a remainder is the sum of the scales of the quotient and the divisor.
- Exponentiation—The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.
- Square root—The scale of a square root is set to the maximum of the scale of the argument and the contents of **scale**.

All of the internal operations are actually carried out in terms of integers with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The internal quantities **scale**, **ibase**, and **obase** can be used in expressions just like other variables. The input line

```
scale = scale + 1
```

increases the value of **scale** by one, and the input line

```
scale
```

causes the current value of **scale** to be printed.

The value of **scale** retains its meaning as a number of decimal digits to be retained in internal computation even when **ibase** or **obase** are not equal to 10. The internal computations (which are still conducted in decimal regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal, octal, or any other kind of digits.

4.4 FUNCTIONS

The name of a function is a single lowercase letter. Function names are permitted to coincide with simple variable names. Twenty-six different defined functions are permitted in addition to the 26 variable names. The input line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements which make up the body of the function ending with a right brace (}). The general form of a function is

```
define a(x) {
        . . .
        . . .
        return
}
```

Return of control from a function occurs when a **return** statement is executed or when the end of the function is reached. The **return** statement can take either of the two forms:

```
return
return(x)
```

In the first case, the value of the function is 0; and in the second, the value of the function is the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one **auto** statement in a function, and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return (exit). The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function *a*, when called, is the product of its two arguments, ‘‘x’’ and ‘‘y’’.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: ().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

causes the result 21.98 to be printed, and the line

```
z = a(a(3,4),5)
```

causes the result 60 to be printed.

4.5 SUBSCRIPTED VARIABLES

A single lowercase letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name, and the expression in brackets is called the subscript. Only 1-dimensional arrays are permitted. The names of arrays are permitted to coincide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to 0 and less than or equal to 2047.

Subscripted variables may be used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function and thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other contexts.

4.6 CONTROL STATEMENTS

The **if**, **while**, and **for** statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way:

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the following six relational operators:

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

Beware of using "=" instead of "= =" as a relational operator. Unfortunately, both of these are legal, so there will be no diagnostic message, but "=" will not do a comparison.

The **if** statement causes execution of its range if and *only if* the relation is true. Then control passes to the next statement in sequence.

The **while** statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range; and if the relation is false, control passes to the next statement beyond the range of the **while** statement.

The **for** statement begins by executing **expression1**. Then the relation is tested; and if true, the statements in the range of the **for** are executed. Then **expression2** is executed. The relation is then tested, etc. The typical use of the **for** statement is for a controlled iteration, as in the statement

```
for(i=1; i< =10; i=i+1) i
```

which prints the integers from one to ten. The following are some examples of the use of the control statements:

```
define f(n){
auto i, x
x=1
for(i=1; i< =n; i=i+1) x=x*i
return(x)
}
```

The input line

```
f(a)
```

prints ‘‘a’’ factorial if ‘‘a’’ is a positive integer. The following is the definition of a function that computes values of the binomial coefficient (m and n are assumed to be positive integers):

```
define b(n,m){
auto x, j
x=1
for(j=1; j< =m; j=j+1) x=x*(n−j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1 = = 1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c = = d) return(c)
                d = c
        }
}
```

4.7 ADDITIONAL FEATURES

There are some additional language features that every user should know.

Normally, statements are typed one to a line. It is also permissible, however, to type several statements on a line by separating the statements by semicolons.

If an assignment statement is parenthesized, it then has a value; and it can be used anywhere that an expression can. For example, the input line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

The following is an example of a use of the value of an assignment statement even when it is not parenthesized. The input line

```
x = a[i=i+1]
```

causes a value to be assigned to *x* and also increments *i* before it is used as a subscript.

The following constructs work in **bc** in exactly the same manner as they do in the C language. Refer to the C language programming documents for more details.

```
x=y=z  is the same as  x=(y=z)
x =+ y      "         x = x+y
x =- y      "         x = x-y
x =* y      "         x = x*y
x =/ y      "         x = x/y
x =% y      "         x = x%y
x =^ y      "         x = x^y
x++         "         (x=x+1)-1
x--         "         (x=x-1)+1
++x         "         x = x+1
--x         "         x = x-1
```

In some of these constructions, spaces are significant. There is a real difference between x=−y and x= −y. The first replaces x by x−y and the second by −y.

The following are three important things to remember when using **bc** programs:

- To exit a **bc** program, type **quit**.
- There is a comment convention identical to that of the C language. Comments begin with /* and end with */.
- There is a library of math functions that may be obtained by typing at command level:

  bc −1

This command loads a set of library functions that includes sine (**s**), cosine (**c**), arctangent (**a**), natural logarithm (**l**), exponential (**e**), and Bessel functions of integer order [**j(n,x)**]. The library sets the scale to 20, but it can be reset to another value.

If you type

```
bc file ...
```

the **bc** program reads and executes the named file or files before accepting commands from the keyboard. In this way, programs and function definitions are loaded.

4.8 NOTATIONS

In the following pages, syntactic categories are in *italics* and literals are in **bold**. Material in brackets “[ ]” is optional.

4.9 TOKENS

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs, or comments. Newline characters or semicolons separate statements.

Comments are introduced by the characters /* and terminated by */.

There are three kinds of identifiers-ordinary, array, and function. All three types consist of single lowercase letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict. A program can have a variable named **x**, an array named **x**, and a function named **x**; all of which are separate and distinct.

The following are reserved keywords:

ibase
obase
scale
sqrt
length
while
for
if
break
define
auto
return
quit

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits A through F are also recognized as digits with values 10 through 15, respectively.

4.10 EXPRESSIONS

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

4.10.1 Named Expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

4.10.2 Identifiers

Simple identifiers are named expressions. They have an initial value of zero.

4.10.3 array-name[expression]

Array elements are named expressions. They have an initial value of zero.

4.10.4 scale, ibase, and obase

The internal registers **scale**, **ibase**, and **obase** are all named expressions. The **scale** register is the number of digits after the decimal point to be retained in arithmetic operations. It has an initial value of zero. The **ibase** and **obase** registers are the input and output number radix, respectively. Both **ibase** and **obase** have initial values of ten.

4.11 FUNCTION CALLS

4.11.1 function name ([expression[,expression..]])

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a **return** statement, the value of the function is the value of the expression in the parentheses of the **return** statement or is zero if no expression is provided or if there is no **return** statement.

4.11.2 sqrt(expression)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

4.11.3 length(expression)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

4.11.4 scale(expression)

The result is the scale of the expression. The scale of the result is zero.

4.11.5 Constants

Constants are primitive expressions.

4.11.6 Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

The unary operators bind right to left.

4.11.7 −expression

The result is the negative of the expression.

4.11.8 + +named-expression

The named expression is incremented by one. The result is the value of the named expression after incrementing.

4.11.9 − −named-expression

The named expression is decremented by one. The result is the value of the named expression after decrementing.

4.11.10 named-expression+ +

The named expression is incremented by one. The result is the value of the named expression before incrementing.

4.11.11 named-expression− −

The named expression is decremented by one. The result is the value of the named expression before decrementing.

The exponentiation operator binds right to left.

4.11.12 expression ^ expression

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If *a* is the scale of the left expression and *b* is the absolute value of the right expression, then the scale of the result is

min(a×b,max(scale,a))

The operators *, /, and % bind left to right.

4.11.13 expression * expression

The result is the product of the two expressions. If *a* and *b* are the scales of the two expressions, then the scale of the result is

min(a + b,max(scale,a,b))

4.11.14 expression / expression

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

4.11.15 expression % expression

The % operator produces the remainder of the division of the two expressions. More precisely, *a%b* is *a−a/b*b*.

The scale of the result is the sum of the scale of the divisor and the value of **scale**.

The additive operators bind left to right.

4.11.16 expression + expression

The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

4.11.17 expression − expression

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

The assignment operators bind right to left.

4.11.18 named-expression = expression

This expression results in assigning the value of the expression on the right to the named expression on the left.

named-expression =+ *expression*
named-expression =− *expression*
named-expression =* *expression*
named-expression =/ *expression*
named-expression =% *expression*
named-expression =^ *expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

4.12 RELATIONAL OPERATORS

Unlike all other operators, the relational operators are only valid as the object of an **if** or **while** statement or inside a **for** statement.

expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*
expression == *expression*
expression != *expression*

4.13 STORAGE CLASSES

There are only two storage classes in **bc**-global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. The **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in **bc** do not work in exactly the same way as in C language. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

4.14 STATEMENTS

Statements must be separated by a semicolon or newline. Except where altered by control statements, execution is sequential.

When a statement is an expression unless the main operator is an assignment, the value of the expression is printed followed by a newline character.

Statements may be grouped together and used when one statement is expected by surrounding them with braces { }.

The following statement prints the string inside the quotes.

```
"any string"
```

if(*relation*)*statement*

The substatement is executed if the relation is true.

while(*relation*)*statement*

The **while** statement is executed while the relation is true. The test occurs before each execution of the statement.

for(*expression*; *relation*; *expression*)*statement*

The **for** statement is the same as

```
first-expression
while(relation) {
    statement
    last-expression
}
```

All three expressions must be present.

```
break
```

The **break** statement causes termination of a **for** or **while** statement.

auto *identifier*[,*identifier*]

The **auto** statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name with empty square brackets. The **auto** statement must be the first statement in a function definition.

define(*[parameter*[*,parameter...*]]){
statements}

The **define** statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

return
return(*expression*)

The **return** statement causes the following:

- Termination of a function
- Popping of the auto variables on the stack
- Specifies the results of the function.

The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

The **quit** statement stops execution of a **bc** program and returns control to the VENIX system software when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.

Contents

Chapter 5
INTERACTIVE DESK CALCULATOR (dc)

5.1 GENERAL

The **dc** program is an interactive desk calculator program implemented on the VENIX operating system to do arbitrary-precision integer arithmetic. It has provisions for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated by **dc** is limited only by available core storage. On typical implementations of the VENIX system, the size of numbers that can be handled varies from several hundred on the smallest systems to several thousand on the largest.

The **dc** program works like a stacking calculator using reverse Polish notation. Ordinarily, **dc** operates on decimal integers; but an input base, output base, and a number of fractional digits to be maintained can be specified.

A language called **bc** has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles the output which is interpreted by **dc**. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into **dc** are put on a pushdown stack. The **dc** commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then it is taken from the standard input.

5.2 dc COMMANDS

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number (e.g. 244)

The value of a number is pushed onto the stack. A number is an unbroken string of digits 0 through 9 and uppercase letters A through F (treated as digits with values 10 through 15, respectively). The number may be preceded by an underscore (_) to input a negative number and numbers may contain decimal points.

The top two values on the stack are added (+), subtracted (−), multiplied (*), divided (/), remaindered (%), or exponentiated (^) by using

+ − * / % ^

The two entries are popped off the stack, and the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. An exponent must not have any digits after the decimal point.

s*x*

The top of the main stack is popped and stored in a register named *x* (where *x* may be any character). If s is uppercase, *x* is treated as a stack; and the value is pushed onto it. Any character, even blank or newline, is a valid register name.

The value of register x is pushed onto the stack. Register x is not altered. If the l in

l*x*

is uppercase, register x is treated as a stack, and its top value is popped onto the main stack. All registers start with empty value which is treated as a zero by the command **l** and is treated as an error by the command **L**.

The following characters perform the stated tasks:

d

The top value on the stack is duplicated.

p

The top value on the stack is printed. The top value remains unchanged.

f

All values on the stack and in registers are printed.

x

Treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of **dc** commands.

[...]

Puts the bracketed character string onto the top of the stack.

q

Exits the program. If executing a string, the recursion level is popped by two. If **q** is uppercase, the top value on the stack is popped; and the string execution level is popped by that value.

<*x* >*x* =*x* !<*x* !>*x* !=*x*

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

v

Replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer.

!

Interprets the rest of the line as a VENIX software command. Control returns to **dc** when the command terminates.

c

All values on the stack are popped; the stack becomes empty.

i

The top value on the stack is popped and used as the number radix for further input. If **i** is uppercase, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o

The top value on the stack is popped and used as the number radix for further output. If **o** is uppercase, the value of the output base is pushed onto the stack.

k

The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If **k** is uppercase, the value of the scale factor is pushed onto the stack.

z

The value of the stack level is pushed onto the stack.

?

A line of input is taken from the input source (usually the console) and executed.

5.3 INTERNAL REPRESENTATION OF NUMBERS

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0 to 99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100s complement notation, which is analogous to twos complement notation for binary numbers. The high-order digit of a negative number is always −1 and all other digits are in the range 0 to 99. The digit preceding the high-order −1 digit is never a 99. The representation of −157 is 43,98,−1. This is called the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when it is convenient.

An additional byte is stored with each number beyond the high-order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high-order digit. The value of this extra byte is called the **scale factor** of the number.

5.4 THE ALLOCATOR

The **dc** program uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is through the allocator. Associated with each string in the allocator is a 4-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and **dc** is via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of two. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Leftover strings are put on the free list. If there are no larger strings, the allocator tries to combine smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long.

If a string of the proper length cannot be found, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in **dc**. If the allocator runs out of headers at any time in the process of trying to allocate a string, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end of string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

5.5 INTERNAL ARITHMETIC

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. The **scale** register limits the number of decimal places retained in arithmetic computations. The **scale** register may be set to the number on the top of the stack truncated to an integer with the **k** command. The **K** command may be used to push the value of **scale** on the stack. The value of **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations includes the exact effect of **scale** on the computations.

5.6 ADDITION AND SUBTRACTION

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

The addition is performed digit by digit from the low-order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers, replacing the high-order configuration 99, −1 by the digit −1. In any case, digits that are not in the range 0 through 99 must be brought into that range, propagating any carries or borrows that result.

5.7 MULTIPLICATION

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly follows the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low-order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

5.8 DIVISION

The scales are removed from the two operands. Zeros are appended, or digits are removed from the dividend to make the scale of the result of the integer division equal to the internal quantity **scale**. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise, the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. If it turns out to be one unit too low, the next trial quotient is larger than 99; and this is adjusted at the end of the process. The trial digit is multiplied by the divisor, the result subtracted from the dividend, and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form with propagation of carry as needed. The sign is set from the sign of the operands.

5.9 REMAINDER

The division routine is called, and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

5.10 SQUARE ROOT

The scale is removed from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity **scale** and the scale of the operand. The method used to compute the square root is Newton's method with successive approximations by the rule.

$$X_{n+1}=(X_n+Y/X_n)$$

The initial guess is found by taking the integer square root of the top two digits.

5.11 EXPONENTIATION

Only exponents with 0 scale factor are handled. If the exponent is 0, then the result is 1. If the exponent is negative, then it is made positive; and the base is divided into 1. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared, and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

5.12 INPUT CONVERSION AND BASE

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with an underscore (_). The hexadecimal digits A through F correspond to the numbers 10 through 15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base **(ibase)** is initialized to 10 (decimal) but may, for example, be changed to 8 or 16 for octal or hexadecimal to decimal conversions. The command **I** pushes the value of the input base on the stack.

5.13 OUTPUT COMMANDS

The command **p** causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers are output by typing the command **f**. The **o** command is used to change the output base (**obase**). This command uses the top of the stack truncated to an integer as the base for all further output. The output base in initialized to 10 (decimal). It works correctly for any base. The command **O** pushes the value of the output base on the stack.

5.14 OUTPUT FORMAT AND BASE

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a backslash (\) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 are used for decimal-octal or decimal-hexadecimal conversions.

5.15 INTERNAL REGISTERS

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands s and **l**. The command s*x* pops the top of the stack and stores the result in register *x*. The *x* can be any character. The command **l***x* puts the contents of register *x* on the top of the stack. The **l** command has no effect on the contents of register *x*. The s command, however, is destructive.

5.16 STACK COMMANDS

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack onto the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

5.17 SUBROUTINE DEFINITIONS AND CALLS

Enclosing a string in brackets "[]" pushes the ASCII string on the stack. The **q** command quits or (in executing a string) pops the recursion levels by two.

5.18 INTERNAL REGISTERS—PROGRAMMING DC

The load and store commands, together with "[]" to store strings, the **x** command to execute, and the testing commands (<, >, =, !<, !>, !=), can be used to program **dc**. The **x** command assumes the top of the stack is a string of **dc** commands and executes it. The testing commands compare the top two elements on the stack and, if the relation holds, execute the register that follows the relation. For example, to print the numbers 0 through 9,

```
[lip1+  si  li10>a]sa
0si  lax
```

5.19 PUSHDOWN REGISTERS AND ARRAYS

These commands are designed for use by a compiler, not directly by programmers. They involve pushdown registers and arrays. In addition to the stack that commands work on, **dc** can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**. **S***x* pushes the top value of the main stack onto the stack for the register *x*. **L***x* pops the stack for register *x* and puts the result on the main stack. The commands **s** and **l** also work on registers but not as pushdown stacks. The command **l** does not affect the top of the register stack, but **s** destroys what was there before.

The commands to work on arrays are **:** and **;**. The command **:***x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. The command **;***x* loads the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

5.20 MISCELLANEOUS COMMANDS

The command **!** interprets the rest of the line as a VENIX software command and passes it to the VENIX operating system to execute. One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.

5.21 DESIGN CHOICES

The real reason for the use of a dynamic storage allocator is that a general purpose program can be used for a variety of other tasks. The allocator has some value for input and for compiling (i.e., the bracket [...] commands) where it cannot be known in advance how long a string will be. The result is that at a modest cost in execution time:

- All considerations of string allocation and sizes of strings are removed from the remainder of the program.
- Debugging is made easier.
- The allocation method used wastes approximately 25 percent of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5 percent in space debugging was made a great deal easier, and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all **dc** commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases is to provide an understandable means of proceeding after a change of base or scale (when numbers had already been entered). An earlier implementation which had global notions of scale and base did not work out well. If the value of **scale** is interpreted in the current input or output base, then a change of base or scale in the midst of a computation causes great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program. It is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The rationale for the choices for the scales of the results of arithmetic is that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give them the result 5.017 without requiring to unnecessarily specify rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands. It seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places, and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

Contents

Chapter 6
LEXICAL ANALYZER GENERATOR (lex)

6.1 GENERAL

The **Lex** is a program generator that produces a program in a general purpose language that recognizes regular expressions. It is designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching. The regular expressions are specified by you (the user) in the source specifications given to **Lex.** The **Lex** program generator source is a table of regular expressions and corresponding program fragments. The table is translated to a program that reads an input stream, copies the input stream to an output stream, and partitions the input into strings that match the given expressions. As each such string is recognized, the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by **Lex.** The program fragments written by you are executed in the order in which the corresponding regular expressions occur in the input stream.

The user supplies the additional code beyond expression matching needed to complete the tasks, possibly including codes written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for your program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while your freedom to write actions is unimpaired.

The **Lex** written code 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''. Just as general purpose languages can produce code to run on different computer hardware, **Lex** can write code in different host languages. The host language is used for the output code generated by **Lex** and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes **Lex** adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is the C language, although Fortran (in the form of Ratfor) has been available in the past. The **Lex** generator exists on the VENIX operating system, but the codes generated by **Lex** may be taken anywhere the appropriate compilers exist.

The **Lex** program generator turns the user's expressions and actions (called **source**) into the host general purpose language; the generated program is named **yylex.** The **yylex** program recognizes expressions in a stream (called **input**) and performs the specified actions for each expression as it is detected.

For 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. This rule contains a regular expression that matches one or more instances of the characters blank or tab (written for visibility, in accordance with the C language convention) and occurs 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," as in **QED**. No action is specified, so the program generated by **Lex** *yylex()* ignores these characters. Everything else is copied. To change any remaining string of blanks or tabs to a single blank, add another rule.

```
%%
[ \t]+$        ;
[ \t]+         printf(" ");
```

The coded instructions (generated for this source) scans for both rules at once, observes (at the termination of the string of blanks or tabs) whether or not there is a newline character, and then executes the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule matches all remaining strings of blanks or tabs.

The **Lex** program generator can be used alone for simple transformations or for analysis and statistics gathering on a lexical level. The **Lex** generator can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface **Lex** and **yacc.** The **Lex** program recognizes only regular expressions; **yacc** writes parsers that accept a large class of context free grammars but requires 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.** You 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.

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.

The **Lex** program generator is not limited to a source that can be interpreted on the basis of one character look-ahead. For example, if there are two rules, one looking for "ab" and another for "abcdefg" and the input stream is "abcdefh", **Lex** recognizes "ab" and leaves the input pointer just before "cd ...". Such backup is more costly than the processing of simpler languages.

6.2 LEX SOURCE

The general format of **Lex** source is

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The first %% is required to mark the beginning of the rules, but the second %% is optional. The absolute minimum **Lex** program is

```
%%
```

(no definitions, no rules) which translates into a program that copies the input to the output unchanged.

In the outline of **Lex** programs shown above, the rules represent your control decisions. They are in a table containing

- A left column with regular expressions
- A right column with actions and program fragments to be executed when the expressions are recognized.

Thus an individual rule might be

```
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 language library function **printf** is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C language 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 more useful example, suppose you desire to change a number of words from British to American spelling. The **Lex** rules such as:

```
colour       printf("color");
mechanise    printf("mechanize");
petrol       printf("gas");
```

would be a start. These rules are not sufficient since the word "petroleum" would become "gaseum".

6.3 LEX REGULAR EXPRESSIONS

The definitions of regular expressions are very similar to those in **QED**. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; the regular expression

```
integer
```

matches the string "integer" wherever it appears, and the expression

```
a57D
```

looks for the string "a57D".

6.3.1 Operators

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 equivalent to the one above. Thus, by quoting every nonalphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters and is safe should further extensions to **Lex** lengthen the list.

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. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within **[]** (see below) must be quoted. Several normal C language escapes with \ are recognized: **\n** is newline, **\t** is tab, and **\b** is backspace. To enter \ itself, use **\\.** Since newline is illegal in an expression, **\n** must be used; it is not required to escape tab and backspace. Every character except blank, tab, newline, and the list of operator characters above is always a text character.

6.3.2 Character Classes

Classes of characters can be specified using the operator pair **[]**. 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 \, –, and ^. The – character indicates ranges. For example,

```
[a–z0–9<>_]
```

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and gets a warning message (e.g., [0-z] in ASCII is many more characters than is in EBCDIC). If it is desired to include the character - 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 ^ operator must appear as the first character after the left bracket to indicate that the resulting string is 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 that is not a letter. The \ character provides the usual escapes within character class brackets.

6.3.3 Arbitrary Character

To match almost any character, the operator character (dot)

```
.
```

is the class of all characters except newline. Escaping into octal is possible although nonportable.

```
[\40-\176]
```

matches all printable ASCII characters from octal 40 (blank) to octal 176 (tilde).

6.3.4 Optional Expressions

The operator **?** indicates an optional element of an expression. Thus:

```
ab?c
```

matches either “ac” or “abc”.

6.3.5 Repeated Expressions

Repetitions of classes are indicated by the operators * and +. For example,

```
a*
```

is any number of consecutive ``a'' characters, including zero; while

```
a+
```

is one or more instances of ``a''. For example,

```
[a-z]+
```

is all strings of lowercase letters. And

```
[A−Za−z][A−Za−z0−9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

6.3.6 Alternation and Grouping

The operator | indicates alternation

```
(ab|cd)
```

matches either ``ab'' or ``cd''. Note that parentheses are used for grouping; although they are not necessary on the outside level,

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions.

```
(ab|cd+)?(ef)*
```

matches such strings as ``abefef'', ``efefef'', ``cdef'', or ``cddd''; but not ``abc'', ``abcd'', or ``abcdef''.

6.3.7 Context Sensitivity

The **Lex** program recognizes a small amount of surrounding context. The two simplest operators for this are ^ and **$**. If the first character of an expression is ^, the expression is only matched at the beginning of a line (after a newline character or at the beginning of the input stream). This never conflicts with the other meaning of ^ (complementation of character classes) since that only applies within the **[ ]** operators. If the very last character is **$**, the expression is only matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character 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 ''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 prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered ''being at the beginning of a line'' to be start condition **ONE**, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

6.3.8 Repetitions and Definitions

The operators { } 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 ''a''.

Finally, initial % is special being the separator for **Lex** source segments.

6.4 LEX ACTIONS

When an expression written as above is matched, **Lex** executes the corresponding action. This part describes some features of **Lex** that aid in writing actions. Note that there is a default action that 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. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule that merely copies can be omitted. Also, a character combination that is omitted from the rules and that 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 language 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 the 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 may often want to know the actual text that matched some expression like ``[a-z]+''. The **Lex** program leaves this text in an external character array. Thus, to print the name found, a rule like

```
[a-z]+    printf("%s", yytext);
```

prints the string in *yytext[].* The C language 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[].* This places the matched string on the output. This action is so common that it may be written as ECHO.

```
[a-z]+    ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule like this one which merely specifies the default action. Such rules are often required to avoid matching some other rule that is not desired. For example, if there is a rule that matches **read,** it normally matches 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 *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, write

```
[a-zA-Z]+      {words++; chars += yyleng;}
```

which accumulates in *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 look ahead offered by the / operator but in a different form.

Example:

Consider a language that defines a string as a set of characters between quotation (") marks and provides that to include a (") in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*  {
         if (yytext[yyleng-1] == '\\')
             yymore();
         else
             ... normal user processing
         }
```

will, when faced with a string such as "**abc**\"**def**", first match the five characters "**abc**\; 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 language problem of distinguishing the ambiguity of "=−a ". Suppose it is desired to treat this as "=− a" but also to print a message: a rule might be

```
=−[a−zA−Z]  {
        printf("Operator (=−) ambiguous\n");
        yyless(yyleng−1);
        ... action for =− ...
        }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=− ". Alternatively, it might be desired to treat this as "= −a ". To do this, just return the minus sign as well as the letter to the input.

```
=−[a−zA−Z]  {
        printf("Operator (=−) ambiguous\n");
        yyless(yyleng−2);
        ... action for = ...
        }
```

performs the other interpretation. 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 is 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 as follows:

1. *input()* returns the next input character.
2. *output(c)* writes the character "c" on the output.

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. The character set used must be consistent in all routines and a value of zero returned by *input* must mean end of file. The relationship between *unput* and *input* must be retained or the **Lex** look ahead will not work. The **Lex** program does not look ahead at all if it does not have to, but every rule ending in **+**, *****, **?**, or **$** or containing / implies look ahead. Look ahead is also necessary to match an expression that is a prefix of another expression. The standard **Lex** library imposes a 100-character limit on backup.

Another **Lex** library routine that you may 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 wrap up 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 that recognizes end of file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied, a file containing nulls cannot be handled since a value of 0 returned by *input* is taken to be end of file.

6.5 AMBIGUOUS SOURCE RULES

The **Lex** program can handle ambiguous specifications. When more than one expression can match the current input, **Lex** chooses as follows:

1. The longest match is preferred.
2. Among rules that matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer    keyword action ...;
[a-z]+     identifier action ...;
```

are to be given in that order. If the input is “integers”, it is taken as an identifier because

“[a-z]+”

matches eight characters while “integer” matches only seven. If the input is “integer”, both rules match seven characters; and the keyword rule is selected because it was given first. Anything shorter (e.g., “int”) does not match the expression “integer” and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like **.*** dangerous. For example:

```
    ’.*’
```

might appear to be a good way of recognizing a string in single quotes. However, 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 will match

```
    ’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 newline. Thus expressions like **.*** stop on the current line. Do not try to defeat this with expressions like **[.\n]+** or equivalents; the **Lex** generated program tries 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 dot (.) does not include newline. Since ''she'' includes ''he'', **Lex** normally *does not* recognize the instances of ''he'' included in ''she'' since once it has passed a ''she'' those characters are gone.

Sometimes the user desires 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 you really want to count the included instances of ''he''. Use the following rule to change the previous example to accomplish the task.

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

After counting each expression, it is rejected; whenever appropriate, the other expression is then counted. In this example, you could note that ''she'' includes ''he'' but not vice versa and omit the *REJECT* action on ''he''. In other cases, it is not possible to state which input characters are 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 2-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.

The action *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.

6.6 LEX SOURCE DEFINITIONS

Recalling the format of the **Lex** source,

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far, only the rules have been described. You need additional options to define variables for use in the program and for use by **Lex**. Variables can go either in the definitions section or in the rules section.

Remember **Lex** is generating 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 not part of a **Lex** rule or action that begins with a blank or tab is copied into the **Lex** generated program. Such source input prior to the first %% delimiter is 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.

   Lines that begin with a blank or tab and that 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 host language convention.

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, 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. 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 later.

6.7 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 language standard library.

On the VENIX operating system, the library is accessed by the loader flag **−ll.** So an appropriate set of commands is

```
lex source
cc lex.yy.c −ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use **Lex** with **yacc,** see part "LEX AND YACC". Although the default **Lex** I/O routines use the C language standard library, the **Lex** automata themselves do not do so; if private versions of *input, output,* and *unput* are given, the library is avoided.

6.8 LEX AND YACC

To use **Lex** with **yacc,** observe that **Lex** writes 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** calls *yylex().* In this case, each **Lex** rule ends 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. If the grammer is to be named "good" and the lexical rules are to be named "better", the VENIX software command sequence could be

```
yacc good
lex better
cc y.tab.c −ly −ll
```

The **yacc** library **(−ly)** should be loaded before the **Lex** library to obtain a main program that invokes the **yacc** parser. The generations of **Lex** and **yacc** programs can be done in either order.

6.9 EXAMPLES

As a problem, consider copying an input file while adding three to every positive number divisible by seven. A suitable **Lex** source program follows:

```
%%
          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 seven; if it is, "k" is incremented by three as it is written out. It may be objected that this program alters such input items as "49.63" or "X7". Furthermore, it increments the absolute value of all negative numbers divisible by seven. To avoid this, 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 dot (.) 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 language 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 that histograms the lengths of words, 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 wrap up. If *yywrap* returns zero (false), it implies that further input is available and the program is to continue reading and processing. Providing a *yywrap* (that never returns true) causes an infinite loop.

6.10 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 ^ operator, for example, is a prior context operator recognizing immediately preceding left context just as **$** 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 part 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 that recognize the need to change the environment in which the following input text is analyzed and that set a parameter to reflect the change. This may be a flag explicitly tested by the user's action code; this 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 is only 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 < > brackets;

```
<name1>expression
```

is a rule that is only recognized when **Lex** is in the start condition **name1.** To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to **name1.** To resume the normal state

```
BEGIN 0;
```

resets the initial condition of the **Lex** automaton interpreter. A rule may be active in several start conditions.

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the < > prefix operator is always active.

The same example as before can be written as follows:

```
%START AA BB CC
%%
^a              {ECHO; BEGIN AA;}
^b              {ECHO; BEGIN BB;}
^c              {ECHO; BEGIN CC;}
\n              {ECHO; BEGIN O;}
<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.

6.11 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 used 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 in the same form as the character constant **'a'**. If this interpretation is changed by providing I/O routines that translate the characters, **Lex** must be given a translation table that is in the definitions section and must be bracketed by lines containing only %**T**; the translation table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character.

6.12 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 ‘‘a’’ 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.

6.13 CAVEATS AND BUGS

There are pathological expressions that 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.

Contents

Chapter 7
YET ANOTHER COMPILER-COMPILER (yacc)

7.1 GENERAL

The **yacc** program 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. The **yacc** program 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.

The **yacc** program is written in a portable dialect of the C language, and the actions and output subroutine are in the C language as well. Moreover, many of the syntactic conventions of **yacc** follow the C language.

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 ;
```

where ''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. With proper definitions, the input

```
July 4, 1776
```

might be matched by the rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizes the lower-level structures, and communicates these tokens to the parser. For historical reasons, 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 only needs to recognize individual letters, and ''month name'' is a nonterminal symbol. Such low-level rules tend to waste time and space and may complicate the specification beyond the ability of **yacc** to deal with it. Usually, the lexical analyzer recognizes the month names and returns an indication that a ''month name'' is seen. In this case, ''month name'' is a ''token''.

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

on input. In most cases, this new rule could be ''slipped 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 **yacc** program has been extensively used in numerous practical applications, including **lint**, the Portable C Compiler, and a system for typesetting mathematics.

The remainder of this document describes the following subjects as they relate to **yacc**

- Basic process of preparing a **yacc** specification
- Parser operation
- Handling ambiguities
- Handling operator precedences in arithmetic expressions
- Error detection and recovery
- The operating environment and special features of the parsers **yacc** produces
- Suggestions to improve the style and efficiency of the specifications
- Advanced topics.

In addition, there are four appendices. Appendix A is a brief example, and Appendix B is a summary of the **yacc** input syntax. Appendix C gives an example using some of the more advanced features of **yacc**, and Appendix D describes mechanisms and syntax no longer actively supported but provided for historical continuity with older versions of **yacc**.

7.2 BASIC SPECIFICATIONS

Names refer to either tokens or nonterminal symbols. The **yacc** program requires token names to be declared as such. In addition, 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 symbol is generally used in **yacc** specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

when each section is used.

The declaration section may be empty, and if the programs section is omitted, the second %% mark may also be omitted. The smallest legal **yacc** specification is

```
%%
rules
```

since the other two sections may be omitted.

Blanks, tabs, and newlines are ignored, but they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in /* ... */, as in C language.

The rules section is made up of one or more grammar rules. A grammar rule has the form

```
A : BODY ;
```

where "A" represents a nonterminal 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, dots, underscores, and noninitial digits. Uppercase and lowercase letters are distinct.

The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ('). As in C language, the backslash (\) is an escape character within literals, and all the C language escapes are recognized. Thus:

'\n'	newline
'\r'	return
'\''	single quote (')
'\\'	backslash (\ )
'\t'	tab
'\b'	backspace
'\f'	form feed
'\xxx'	"xxx" in octal

are understood by **yacc**. 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
   ;
```

by using the vertical bar. It is not necessary that all grammar rules with the same left side appear together in the grammar rules section although it makes the input much more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by

```
empty :  ;
```

which is understood by **yacc**.

Names representing tokens must be declared. This is most simply done by writing

```
    %token   name1  name2 ...
```

in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the *start symbol* has particular importance. 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 desirable to declare the start symbol explicitly in the declarations section using the **%start** keyword

```
    %start   symbol
```

to define the start symbol.

The end of the input to the parser is signaled by a special token, called the *end-marker*. If the tokens up to but not including the end-marker form a structure that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually the end-marker represents some reasonably obvious I/O status, such as ''end of file'' or ''end of record''.

7.3 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 language statement and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements enclosed in curly braces ({) and (}). For example:

```
A  : '(' B ')'
     {
        hello( 1, "abc" );
     }
```

and

```
XXX  : YYY  ZZZ
        {
           printf("a message\n");
           flag = 25;
        }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign symbol (**$**) 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, the action

```
{  $$ = 1;  }
```

does nothing but return the value of one.

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. If the rule is

```
A  : B  C  D  ;
```

then **$2** has the value returned by C, and **$3** the value returned by D.

The rule

```
expr  :  '('  expr  ')'  ;
```

provides a more concrete example. 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 rules. Sometimes, it is desirable to get control before a rule is fully parsed. The **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. The **yacc** program actually treats the above example as if it had been written

```
$ACT  :  /* empty */
      {
         $$ = 1;
      }
      ;

A     :  B  $ACT  C
      {
         x = $2;
         y = $3;
      }
      ;
```

where **$ACT** is an empty action.

In many applications, output is not done directly by the actions. A data structure, such as a parse tree, is constructed in memory and transformations are applied to it before output is generated. Parse trees are particularly easy to construct given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node* written so that the call

```
node( L, n1, n2 )
```

creates a node with label L and descendants n1 and n2 and returns the index of the newly created node. Then parse tree can be built by supplying actions such as

```
expr  :  expr  '+'  expr
      {
         $$ = node( '+', $1, $3 );
      }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example:

```
%{  int variable = 0;  %}
```

could be placed in the declarations section making ''variable'' accessible to all of the actions. The **yacc** parser uses only names beginning with **yy**. The user should avoid such names.

In these examples, all the values are integers. A discussion of values of other types is found in the part ''ADVANCED TOPICS''.

7.4 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 the user. In either case, the **#define** mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like

```
yylex()
{
  extern int yylval;
  int c;
  ...
  c = getchar();
  ...
  switch( c )
  {
    ...
  case ’0’:
  case ’1’:
    ...
  case ’9’:
    yylval = c-’0’;
    return( DIGIT );
    ...
  }
  ...
```

to return the appropriate token.

The intent is to return a token number of DIGIT and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT is defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling and should not be used naively.

As mentioned above, the token numbers may be chosen by **yacc** or 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 end-marker must have token number 0 or negative. This token number cannot be redefined by the user. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the **lex** program. 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.

7.5 PARSER OPERATION

The **yacc** program turns the specification file into a C language 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. The parser itself, however, is relatively simple and understanding how it works will 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 *look-ahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0 (the stack contains only state 0) and no look-ahead token has been read.

The machine has only four actions available—*shift*, *reduce*, *accept*, and *error*. A step of the parsor is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. If it needs one and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off of the stack and in the look-ahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a look-ahead token. For example, in state 56 there may be an action

```
IF   shift 34
```

which says, in state 56, if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right-hand side by the left-hand side. It may be necessary to consult the look-ahead token to decide whether to reduce or not (usually it is not necessary). In fact, the default action (represented by a dot) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. The action

```
.   reduce 18
```

refers to grammar rule 18, while the action

```
IF   shift 34
```

refers to state 34.

Suppose the rule

```
A  :  x y z   ;
```

is being reduced. The reduce action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z* and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the look-ahead token is cleared by a shift but is not affected by a goto. In any case, the uncovered state contains an entry such as

```
A   goto 20
```

causing state 20 to be pushed onto the stack and become the current state.

In effect, the reduce action "turns back the clock" in the parse popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stacks. The uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable "yylval" is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable "yyval" is copied onto the value stack. The pseudo-variables **$1**, **$2**, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) will be discussed later.

Consider:

```
%token  DING  DONG  DELL
%%
rhyme  :  sound  place
       ;
sound  :  DING  DONG
       ;
place  :  DELL
       ;
```

as a **yacc** specification.

When **yacc** is invoked with the −**v** option, a file called *y.output* is produced with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

state 3
        sound   :   DING_DONG

        DONG  shift 6
```

```
          .  error

     state 4
          rhyme  :  sound  place_    (1)

          .  reduce  1

     state 5
          place  :  DELL_    (3)

          .  reduce  3

     state 6
          sound   :  DING  DONG_    (2)

          .  reduce  2
```

where the actions for each state are specified and there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen and what is yet to come in each rule. The following input

```
DING  DONG  DELL
```

can be used to track the operations of the parser. Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read and becomes the look-ahead token. The action in state 0 on *DING* is *shift 3*, state 3 is pushed onto the stack, and the look-ahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read and becomes the look-ahead token. The action in state 3 on the token *DONG* is *shift 6*, state 6 is pushed onto the stack, and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, the parser reduces by

```
sound  :  DING  DONG
```

which is rule 2. Two states, 6 and 3, are popped off of the stack uncovering state 0. Consulting the description of state 0 (looking for a goto on *sound*),

```
sound   goto 2
```

is obtained. State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token, *DELL*, must be read. The action is *shift 5*, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place* (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read and the end-marker is obtained indicated by **$end** in the *y.output* file. The action in state 1 (when the end-marker is seen) successfully ends the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spent with this and other simple examples is repaid when problems arise in more complicated contexts.

7.6 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".)

The **yacc** program detects such ambiguities when it is attempting to build the parser. Given the input

```
expr  -  expr  -  expr
```

consider the problem that confronts the parser. When the parser has read the second expr, the input seen

```
expr  -  expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to ‘‘expr’’ (the left side of the rule). The parser would then read the final part of the input

```
    –  expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, if the parser sees

```
    expr  –  expr
```

it could defer the immediate application of the rule and continue reading the input until

```
    expr  –  expr  –  expr
```

is seen. It could then apply the rule to the rightmost three symbols reducing them to ‘‘expr’’ which results in

```
    expr  –  expr
```

being left. Now the rule can be reduced once more. The effect is to take the right associative interpretation. Thus, having read

```
    expr  –  expr
```

the parser can do one of two legal things, a shift or a reduction. It has no way of deciding between them. This is called a ‘‘shift/reduce conflict’’. It may also happen that the parser has a choice of two legal reductions. This is called a ‘‘reduce/reduce conflict’’. Note that there are never any shift/shift conflicts.

When there are shift/reduce or reduce/reduce conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing the choice to make in a given situation is called a ‘‘disambiguating rule’’.

The **yacc** program 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 in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided when possible.

Conflicts may arise because of mistakes in input or logic or because the grammar rules (while consistent) require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, **yacc** always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural and produces slower parsers. Thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider

```
stat:  IF '(' cond ')' stat
|  IF '(' cond ')' stat ELSE stat
;
```

which is a fragment from a programming language involving an "if-then-else" statement. In these rules, "IF" and "ELSE" are tokens, "cond" is a nonterminal symbol describing conditional (logical) expressions, and "stat" is a nonterminal symbol describing statements. The first rule will be called the "simple-if" rule and the second the "if-else" rule.

These two rules form an ambiguous construction 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
}
```

where the second interpretation is the one given in most programming languages having this construct. Each ''ELSE'' is associated with the last preceding ''un-ELSE'd'' IF. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the ''ELSE''. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the ''ELSE'' may be shifted, ''S2'' read, and then the right-hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input which is usually desired.

Once again, the parser can do two valid things—there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, ''ELSE'', and particular inputs, such as

```
IF ( C1 ) IF ( C2 ) S1
```

have already been seen. In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (**−v**) option output file. For example, the output corresponding to the above conflict state might be

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

   stat : IF ( cond ) stat_        (18)
   stat : IF ( cond ) stat_ELSE stat

   ELSE    shift 45
   .       reduce 18
```

where the first line describes the conflict—giving the state and the input symbol. The ordinary state description gives the grammar rules active in the state and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is "ELSE", it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the "ELSE" will have been shifted in this state. In state 23, the alternative action [describing a dot (.)] is to be done if the input symbol is not mentioned explicitly in the actions. In this case, if the input symbol is not "ELSE", the parser reduces to

```
stat : IF '(' cond ')' stat
```

by grammar rule 18.

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there is reduce action possible in the state and this is the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

7.7 PRECEDENCE

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts. As disambiguating rules, the user specifies the precedence or binding strength of all the operators and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative and have lower precedence than star and slash, which are also left associative. The keyword **%right** is used to describe right associative operators, and the keyword **%nonassoc** is used to describe operators, like the operator **.LT.** in FORTRAN, that may not associate with themselves. Thus:

```
A .LT. B .LT. C
```

is illegal in FORTRAN and such an operator would be described with the keyword **%nonassoc** in **yacc**. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr:  expr '=' expr
|  expr '+' expr
|  expr '-' expr
|  expr '*' expr
|  expr '/' expr
|  NAME
;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

in order to perform the correct precedence of operators. When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary "–". Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, **%prec**, changes the precedence level associated with a particular grammar rule. The keyword **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules

```
    %left '+' '-'
    %left '*' '/'

    %%

    expr:  expr '+' expr
    |  expr '-' expr
    |  expr '*' expr
    |  expr '/' expr
    |  '-' expr     %prec '*'
    |  NAME
    ;
```

might be used to give unary minus the same precedence as multiplication.

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

The precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar. It is a good idea to be sparing with precedences and use them in an essentially "cookbook" fashion until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7.8 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 do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides 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 look-ahead token and performs the action encountered. The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
    stat  :  error
```

means that on a syntax error the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Rules such as

```
stat  :  error ';'
```

are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule will be reduced and any ''cleanup'' action associated with it performed.

Another form of error rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example

```
input  :  error '\n'
             {
               printf( "Reenter last line: " );
             }
             input
      {
        $$ = $4;
      }
      ;
```

is one way to do this. There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable. For this reason, there is a mechanism that can force the parser to believe that error recovery has been accomplished. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as

```
input  :  error  '\n'
              {
                 yyerrok;
                 printf( "Reenter last line: " );
              }
              input
          {
             $$ = $4;
          }
          ;
```

which is somewhat better.

As previously mentioned, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous look-ahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by *yylex* is presumably the first token in a legal statement. The old illegal token must be discarded and the error state reset. A rule similar to

```
stat  :  error
      {
         resynch();
         yyerrok ;
         yyclearin;
      }
      ;
```

could perform this.

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

7.9 THE "yacc" ENVIRONMENT

When the user inputs a specification to **yacc**, the output is a file of C language programs, called *y.tab.c* on most systems. (Due to local file system conventions, the names may differ from installation to installation.) The function produced by **yacc** is called *yyparse()*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex()*, the lexical analyzer supplied by the user (see "LEXICAL ANALYSIS"), to obtain input tokens. Eventually, an error is detected, *yyparse()* returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, *yyparse()* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C language program, a program called *main()* must be defined that eventually calls *yyparse()*. In addition, a routine called *yyerror()* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of *main()* and *yyerror()*. The name of this library is system dependent; on many systems, the library is accessed by a **−ly** argument to the loader. The source codes

```
main()
{
    return ( yyparse() );
}
```

and

```
# include <stdio.h>

yyerror(s)
      char *s;
{
      fprintf( stderr, "%s\n", s );
}
```

show the triviality of these default programs. The argument to *yyerror()* is a string containing an error message, usually the string "syntax error". The average application wants to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics. Since the *main()* 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 the input symbols 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.

7.10 HINTS FOR PREPARING SPECIFICATIONS

This part contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

7.10.1 Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following are a few style hints.

1. Use all uppercase letters for token names and all lowercase letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong".
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left-hand side together. Put the left-hand side in only once and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by two tab stops and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in this section (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.

7.10.2 Left Recursion

The algorithm used by the **yacc** parser encourages so called "left recursive" grammar rules. Rules of the form

```
name  :  name  rest_of_rule  ;
```

match this algorithm. These rules such as

```
list:  item
|  list  ','  item
;
```

and

```
seq:  item
|  seq  item
;
```

frequently arise when writing specifications of sequences and lists. In each of these cases, the first rule will be reduced for the first item only; and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq:  item
|  item  seq
;
```

the parser is a bit bigger; and the items are seen and reduced from right to left. More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, the user should use left recursion wherever reasonable.

It is worth considering if a sequence with zero elements has any meaning, and if so, consider writing the sequence specification as

```
seq:  /* empty */
|  seq  item
;
```

using an empty rule. Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen when it hasn't seen enough to know!

7.10.3 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,

```
%{
  int dflag;
%}
  ... other declarations ...

%%

prog  :  decls  stats
      ;

decls :  /* empty */
      {
           dflag = 1;
      }
      |  decls  declaration
      ;

stats :  /* empty */
      {
           dflag = 0;
      }
      |  stats  statement
      ;

   ... other rules ...
```

specifies a program that consists of zero or more declarations followed by zero or more statements. The flag ''dflag'' is now 0 when reading statements and 1 when reading declarations, *except for the first token in the first statement*. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of ''back-door'' approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult if not impossible to do otherwise.

7.10.4 Reserved Words

Some programming languages permit you to use words like ''if'', which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**. It is difficult to pass information to the lexical analyzer telling it ''this instance of *if* is a keyword and that instance is a variable''. The user can make a stab at it using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*, i.e., forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

7.11 ADVANCED TOPICS

This part discusses a number of advanced features of **yacc**.

7.11.1 Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros **YYACCEPT** and **YYERROR**. The **YYACCEPT** macro causes *yyparse()* to return the value 0; **YYERROR** causes the parser to behave as if the current input symbol had been a syntax error; *yyerror()* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple end-markers or context sensitive syntax checking.

7.11.2 Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit.

```
sent  :  adj  noun  verb  adj  noun
      {
         look at the sentence ...
      }
      ;
adj   :  THE
      {
            $$ = THE;
      }
      |  YOUNG
      {
            $$ = YOUNG;
      }
      ...
      ;
noun  :  DOG
      {
         $$ = DOG;
      }
      |  CRONE
      {
         if( $0 = = YOUNG )
         {
            printf( "what?\n" );
         }
         $$ = CRONE;
      }
      ;
      ...
```

In this case, the digit may be 0 or negative. In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol ''noun'' in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism prevents a great deal of trouble especially when a few combinations are to be excluded from an otherwise regular structure.

7.11.3 Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. The **yacc** program can also support values of other types including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type checked. 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 take place. In addition, type checking commands such as **lint** is far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union. This must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes

```
%union
{
  body of union ...
}
```

in the declaration section. This declares the **yacc** value stack and the external variables *yylval* and *yyval* to have type equal to this union. If **yacc** was invoked with the **−d** option, the union declaration is copied onto the *y.tab.h* file. 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 have said

```
typedef union
{
  body of union ...
}
YYSTYPE;
```

instead. The header file must be included in the declarations section by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords **%token**, **%left**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed. Thus, saying

```
    %left <optype> '+' '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, **%type**, is used to associate union member names with nonterminals. Thus, one might say

```
    %type <nodetype> expr stat
```

to associate the union member *nodetype* with the nonterminal symbols "expr" and "stat".

There remains a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between < and > immediately after the first $. The example

```
    rule  :  aaa
                {
                  $<intval>$ = 3;
                }
                bbb
          {
             fun( $<intval>2, $<other>0 );
          }
          ;
```

shows this usage. This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. 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.

7.12 APPENDIX A

7.12.1 A Simple Example

This example gives the complete **yacc** applications for a small desk calculator; the calculator has 26 registers labeled 'a' through 'z' and accepts arithmetic expressions made up of the operators +, −, *,/,% (med operator), & (bitwise and), | (bit wise or), and assignments. If an expression at the top level is an assignment, the value is printed; otherwise, the expression is printed. As in C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a resonable job of showing how precedence and ambiguties are used and demonstrates simple recovery. The major oversimplications are that the lexical analyzer is much simpler for most applications, and the output is produced immediately line by line. Note the way that decimal and octal integers are read in by grammar rules. This job is probably better done by the lexical analyzer.

```
%{
# includes<stdio.h>
# includes<ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /* supplies precedence for unary minus */

% %        /* beginning of rule section */

list      : /* empty */
          | list stat '\n'
          | list error '\n'

          {

            yyerrork;

          }
          ;
```

```
stat      : expr
          {
            printf( ″%dn″, $1 );
          }
          | LETTER ‘=’ expr
          {
            regs[$1] = $3
          }
          ;

expr       : ‘(’ expr ‘)’
             {
                 $$ = $2;
             }
             | expr ‘+’ expr
             {
                 $$ = $1 + $3
             }
             | expr ‘−’ expr
             {
                 $$ = $1 − $3
             {
             | expr ‘*’ expr
             {
                 $$ = $1 * $3;
             }
             | expr ‘/’ expr
             {
                 $$ = $1/$3;
             }
             |  exp ‘%’ expr
             {
                  $$ = $1 % $3
             }
             |  expr ‘&’ expr

             {
                 $$ = $1 & $3;
             }
             |  expr ‘|’ expr
             {
                 $$ = $1 | $3
             }
          | '-' expr  %prec UMINUS
          {
             $$ = - $2;
          }
          | LETTER
          {
             $$ = reg[$1];
          }
          | number
          ;

number       :  DIGIT
          {
              $$ = $1; base = ($1= =0) ? 8 ; 10;
          }
          |  number DIGIT
          {
              $$ = bas * $1 + $2
          }
          ;

%% /* start of program */

yylex( )          /* lexical analysis routine */
{             /* return LETTER for lowercase letter,
          yylval = 0 through 25*/

          /* returns DIGIT for digit, yylval = 0 through 9*/
          /* all other characters are returned immediately */

          int c;
                                /*skip blanks*/
          while (c=getchar( ) )    = = '')
                ;

          /* c is now nonblank */

          if( islower( c ))
          {
                  yylval = c-'a';
                  return( LETTER );
          }
          if( isdigit( c ))
          }
                    yylval = c – ‘0’;
                    return( DIGIT );

          }
          return( c );
}
```

7.13 APPENDIX B

7.13.1 YACC Input Syntax

systax as a **yacc** specification. Contex 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 decides whether the next token (skipping blanks, newlines, and comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS but never as part of C_IDENTIFIERs.

```
   /* grammar for the input to yacc */

   /* basic entries */
%token   IDENTIFIER   /* includes identifiers and literals */
%token    C_IDENTIFIER  /* identifier (but not literal)
                   followed by a 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 */
```

```
%token   spec

%%

spec  :   defs MARK rules tail
      ;

tail  :   MARK
      {
              In this action, eat up the rest of the file
      }

      |       /* empty: the second MARK is optional */
      ;

defs  :        /* empty */
      |       defs def
      ;

defs  :        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         /*Note: literal illegal with % type */
        |  IDENTIFIER NUMBER   /* Note: illegal with % type */
        ;

  /* rule section */

rules  :  C_IDENTIFIER rbody proc
       |  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';'
      ;
```

7.14 APPENDIX C

7.14.1 An Advanced Example

This appendix gives an example of a grammar using some of the advanced features. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants; the arithmetic operations +, − *, /, and unary −; "a" through "z". Moreover, it also understands intervals written

(X,Y)

where X is less than or equal to Y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments returns no value and prints nothing while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C language. Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, INTERVAL, by using *typedef*. The **yacc** value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that the entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions—division by an interval containing 0 and an interval presented in the wrong order. The error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**— 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines.

2.5 + (3.5 − 4.)

and

2.5 + (3.5,4)

Notice that the 2.5 is to be used in an interval value expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator—one when the left operand is a scalar and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflict will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive 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 language library routine *atof()* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar provoking a syntax error in the parser and thence error recovery.

```
%{

#include<stdio.h>
#include<ctype.h>

typedef struct interval
{
   double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv( );

double atof( );

double dreg[ 26 ];
INTERVAL vreg[ 26 ];
```

```
%}

%start line

%union
{
 int ival;
 double dval;
 INTERVAL vval;
}

%token <ival> DREG VREG  /*indices into dreg, vreg arrays */


%token <dval> CONST        /* floating point constant */

%type <dval> dexp          /* expression */

%type <vval> vexp          /* interval expression */

 /* precedence information about the operators */

%left   ‘+’ ‘-’
%left   ‘*’ ‘/’
%left   UMINUS   /* precedence for unary minus */

% %

lines  : /* empty */
       | lines line
       ;
line   : dexp ‘\n’
       {
              printf( "%15.8f\n".$1 );
       }
       | vexp ‘\n’

       {
              printf( "(%15.8f , %15.8f )0,$1.1o,$1.hi );

       }
       | DREG ‘=’ ‘\n’
       {
            dreg[$1] = $3;

        }
        | VREG '=' vexp '\n'
        {

            vreg[$1] = $3;

        }
        | error '\n'
        {

              yyerrork;

        }
        ;

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

        }
        ;
vexpp   : 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

        }
        | dvep ‘-’ vdep

        {

                $$.hi = $1 - $3.lo;
                $$.lo = $1 - $3.hi

        }
        | vexp ‘*’ vexp
        {

                $$ = vmul( $1.lo,$.hi,$3 )

        }
        | dexp ‘*’ vexp
        {

                $$ = vmul( $1, $1, $3 )

        }
        | vexp ‘/’ vexp
        {
                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 )

        }
        | dexp ‘/’ vexp


                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 )
        }
        | ‘-’ vexp    %prec UMINUS
        {
                $$.hi = -$2.lo;$$.lo =-$2.hi
        }
        | ‘(’ vexp ‘)’
        }
                 $$ = $2
        }
        ;

%%

# define BSZ 50  /* buffer size for floating point number */

        /* lexical analysis */

yylex( )
{
        register c;

                                /* skip over blanks */

                ;

        if( isupper( c ) )
        {
           yylval.ival = c - ‘A’
           return( VREG );
        }
        if( islower( c ) )
        {

           yylval.ival = c - ‘a’,
           return( DREG );
        }

           /* gobble up digits. points, exponents */
        if( idigit( c ) || c= ='.' )
        {

           char buf[BSZ+1], *cp = buf;
           int dot = 0, exp = 0;

           for( ; (cp-buf)<BSZ ; + +cp,c=getchar( ) )
       {

              *cp = c;
              if( isdigit( c ) )
                continue;
              if( c = ='.'
              {
              if( dot+ + || exp )
                    return( '.' );/* will cause syntax error */
                continue;
              }
             if( c = = 'e')
             {

                 if( exp+ + )
                       return( 'e' ); */will cause syntax error */
                 continue;
             }
                 /* end of number */
                 break;
        }
        *cp = '\0';
        if(cp-buff) > = BSZ )
          printcf( "constant too long truncated\n" );
        else
           ungetc( c, stdin ); /* push back last char read */
        yylval.dval = atof( buf );
        return( CONST );
        }
        return( c );

}

INTERVAL
hilo( a, b, c, d )
        double a, b, c, d;
```

```
{
        /* returns the smallest interval containing a, b, c, and d */

        /* used by *,/ routine */
        INTERVAL v;

        if( a>b )
        {
             v.hi = a;
             v.lo = b;
        }
        else
        {
             v.hi = b;
             v.lo = a;
        }
        if( c>d )
        {
             if( c>v.hi )
                v.hi = c;
             if( d<v.lo )
                v.lo = d;
        }
        else
        }
             if( d>v.hi )
                v.hi = d;
             if( c<v.lo )
                v.lo = c;
        }
        return( v );
}
INTERVAL vmul( a, b, v )
        double a, b;
        INTERVAL v;
{
         return( hilo( a*v.hi, a*v,lo, b*v.hi, b*v.lo ) );
}
dcheck( v )
        INTERVAL v;
{
        if( v.hi > =0.&& v.lo < =0. )
        {
                printf( "divisor internal contains 0.\n" );
                return( 1);
        }
        return( 0 );
{

INTERVAL vdiv( a, b, v )
        double a, b;
        INTERVAL v;
{
  return( hilo( a/v.hi, a/v,lo, b/v.hi, b/v.lo ) );

}
```

7.15 APPENDIX D

7.15.1 Old Features Supported But Not Encouraged

This appendix mentions synonyms and features that 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 that one character long. If all the characters are alphabetic, numeric, or _, 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 literal.

   The use of multicharacter literals is likely to mislead those unfimiliar with **yacc** since it suggests that **yacc** is doing a job which 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. Action may also have the form

   ={ ... }

and the curly braces can be dropped if the action is a single C language statement.

6. The C language code between %{ and %} use to be permitted at the head of the rules section as well as in the declaration section.

Printed in U.S.A.

AA-BM34A-TH