Support Tools Guide

UNIX System

Trademarks:

MUNIX, CADMUS	for PCS
UNIX	for Bell Laboratories
DEC, PDP, VAX	for DEC
MASSBUS, UNIBUS	
KODAK, EKTAMATIC	for Eastman Kodak Company
Mohrflow, Mohrdry, Mohrchem	for Mohr Lino-Saw Comp.
TEKTRONIX	for Tektronik, Inc.
TELETYPE	for Teletype Corporation
TRENDATA 4000A®	for Trendata Corporation
Versatec	for Versatec Corporation
DIABLO	for Xerox Corporation

Copyright 1984 by
PCS GmbH, Pfälzer-Wald-Strasse 38, D-8000 München 90, tel. (089) 67604-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.

SUPPORT TOOLS GUIDE

UNIX SYSTEM

CONTENTS PAGE

1. INTRODUCTION 9

2. A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make) 11

GENERAL 11

BASIC FEATURES 13

DESCRIPTION FILES AND SUBSTITUTIONS 15

COMMAND USAGE 16

SUFFIXES AND TRANSFORMATION RULES 17

IMPLICIT RULES 18

SUGGESTIONS AND WARNINGS 19

3. AUGMENTED VERSION OF MAKE 21

GENERAL 21

THE ENVIRONMENT VARIABLES 21

RECURSIVE MAKEFILES 22

FORMAT OF SHELL COMMANDS WITHIN make 23

ARCHIVE LIBRARIES 23

SOURCE CODE CONTROL SYSTEM FILE NAMES: THE TILDE 24

THE NULL SUFFIX 25

INCLUDE FILES 26

INVISIBLE SCCS MAKEFILES 26

DYNAMIC DEPENDENCY PARAMETERS 26

CONTENTS PAGE

EXTENSIONS OF $*, $@, AND $< . . . 27

OUTPUT TRANSLATIONS . . . 27

4. SOURCE CODE CONTROL SYSTEM USER'S GUIDE . . . 41

GENERAL . . . 41

SCCS FOR BEGINNERS . . . 41

A. Terminology . . . 42

B. Creating an SCCS File via "admin" . . . 42

C. Retrieving a File via "get" . . . 42

D. Recording Changes via "delta" . . . 43

E. Additional Information About "get" . . . 44

F. The "help" Command . . . 45

DELTA NUMBERING . . . 45

SCCS COMMAND CONVENTIONS . . . 47

SCCS COMMANDS . . . 48

A. The "get" Command . . . 49

B. The "delta" Command . . . 55

C. The "admin" Command . . . 57

D. The "prs" Command . . . 59

E. The "help" Command . . . 61

F. The "rmdel" Command . . . 61

G. The "cdc" Command . . . 62

H. The "what" Command . . . 62

I. The "sccsdiff" Command . . . 63

J. The "comb" Command . . . 63

K. The "val" Command . . . 64

SCCS FILES . . . 64

CONTENTS PAGE

A. Protection 64
B. Formatting 65
C. Auditing 65
AN SCCS INTERFACE PROGRAM 66
A. General 66
B. Function 66
C. A Basic Program 67
D. Linking and Use 67
5. THE M4 MACRO PROCESSOR 71
GENERAL 71
DEFINING MACROS 71
ARGUMENTS 74
ARITHMETIC BUILT-INS 74
FILE MANIPULATION 75
SYSTEM COMMAND 75
CONDITIONALS 76
STRING MANIPULATION 76
PRINTING 77
6. THE "awk" PROGRAMMING LANGUAGE 81
GENERAL 81
A. Usage 81
B. Program Structure 81
C. Records and Fields 82
D. Printing 82
PATTERNS 83
A. "BEGIN" and "END" 83

CONTENTS PAGE

B. Regular Expressions 84
C. Relational Expressions 85
D. Combinations of Patterns 85
E. Pattern Ranges 85
ACTIONS 86
A. Built-in Functions 86
B. Variables, Expressions, and Assignments 86
C. Field Variables 87
D. String Concatenation 88
E. Arrays 88
F. Flow-of-Control Statements 89
7. ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (BC) 91
GENERAL 91
SIMPLE COMPUTATIONS WITH INTERGERS 91
BASES 92
SCALING 93
FUNCTIONS 94
SUBSCRIPTED VARIABLES 95
CONTROL STATEMENTS 95
ADDITIONAL FEATURES 97
8. INTERACTIVE DESK CALCULATOR (DC) 105
GENERAL 105
DC COMMANDS 105
INTERNAL REPRESENTATION OF NUMBERS 107
THE ALLOCATOR 107
INTERNAL ARITHMETIC 108

CONTENTS PAGE

ADDITION AND SUBTRACTION 108
MULTIPLICATION 108
DIVISION 108
REMAINDER 109
SQUARE ROOT 109
EXPONENTIATION 109
INPUT CONVERSION AND BASE 109
OUTPUT COMMANDS 109
OUTPUT FORMAT AND BASE 110
INTERNAL REGISTERS 110
STACK COMMANDS 110
SUBROUTINE DEFINITIONS AND CALLS 110
INTERNAL REGISTERS—PROGRAMMING DC 110
PUSHDOWN REGISTERS AND ARRAYS 110
MISCELLANEOUS COMMANDS 110
DESIGN CHOICES 111
9. LEXICAL ANALYZER GENERATOR (LEX) 113
GENERAL 113
LEX SOURCE 115
LEX REGULAR EXPRESSIONS 115
A. Operators 116
B. Character Classes 116
C. Arbitrary Character 117
D. Optional Expressions 117
E. Repeated Expressions 117
F. Alternation and Grouping 117

CONTENTS PAGE

G. Context Sensitivity . . . 118
H. Repetitions and Definitions . . . 118
LEX ACTIONS . . . 119
AMBIGUOUS SOURCE RULES . . . 121
LEX SOURCE DEFINITIONS . . . 123
USAGE . . . 124
LEX AND YACC . . . 124
EXAMPLES . . . 125
LEFT CONTEXT SENSITIVITY . . . 126
CHARACTER SET . . . 128
SUMMARY OF SOURCE FORMAT . . . 128
CAVEATS AND BUGS . . . 129
10. YET ANOTHER COMPLIER—COMPLIER (yacc) . . . 131
GENERAL . . . 131
BASIC SPECIFICATIONS . . . 133
ACTIONS . . . 134
LEXICAL ANALYSIS . . . 137
PARSER OPERATION . . . 138
AMBIGUITY AND CONFLICTS . . . 141
PRECEDENCE . . . 145
ERROR HANDLING . . . 147
THE "yacc" ENVIRONMENT . . . 149
HINTS FOR PREPARING SPECIFICATIONS . . . 150
A. Input Style . . . 150
B. Left Recursion . . . 150
C. Lexical Tie-ins . . . 151

CONTENTS PAGE

D. Reserved Words 152

ADVANCED TOPICS 152

A. Simulating Error and Accept in Actions 152

B. Accessing Values in Enclosing Rules 152

C. Support for Arbitrary Value Types 153

NOTES

1. INTRODUCTION

The SUPPORT TOOLS volume is a description of the various software "tools" which may aid the UNIX operating system user. The following paragraphs contain a brief description of each section.

The section 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 simplier by the **make** program.

The section AUGMENTED VERSION OF "make" describes the modifications made to handle many of the problems within the original **make** program.

The section SOURCE CODE CONTROL SYSTEM USER'S GUIDE describes the collection of SCCS programs under the UNIX operating system. The SCCS programs act as a "custodian" over the UNIX system files.

The section THE M4 MACRO PROCESSOR describes the front end for rational Fortran and C programming language.

The section 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.

The section ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (BC) describes a compiler for doing arbitrary precision arithmetic on the UNIX operating system.

The section INTERACTIVE DESK CALCULATOR (DC) describes a program implemented on the UNIX operating system to do arbitrary-precision integer arithmetic.

The section LEXICAL ANALYZER GENERATOR (Lex) describes a software tool designed for lexical processing of character input streams.

The section 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 UNIX software commands. The "tools" described will enable the user to fully utilize the UNIX operating system.

NOTES

2. A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make)

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 may be 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 which 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.

A programmer can easily forget which files depend on which other files, which files have been modified recently, which files need to be reprocessed or recompiled after a change in some part of the source, and the exact sequence of operations needed to make or 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 is used to maintain, update, and regenerate groups of computer programs.

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 will create 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 a needed target file in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target file if it has not been modified since its generators were. 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 dependences and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files regardless of the number 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 = und -3 | opr -r2    # send to be printed
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
        rm 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 file:

```
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 = opr -sp "
              or
make print " P= cat >zap "
```

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. The **make** program does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, 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 IS 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 our 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).

Not taking advantage of **make**'s innate knowledge results in the following longer description file using the same example:

```
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 had changed since the last time **prog** was made, all of the files would be current, and the command

```
make
```

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

```
make x.o
```

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

If the file exists after the commands are executed, 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 futher decisions. Often a method 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 "clean-up" 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 UNIX software commands.

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 has been specified on the **make** command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX 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.

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:

–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".

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 will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed. The following is an excerpt from the default rules file:

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

```
.s.o :
        $(AS) -o $@ $<
.y.o :
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

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:

.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

Figure 2.1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

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 as shown in Fig. 2.1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros **AS**, **CC**, **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=-0"
```

causes the optimizing C language compiler to be used.

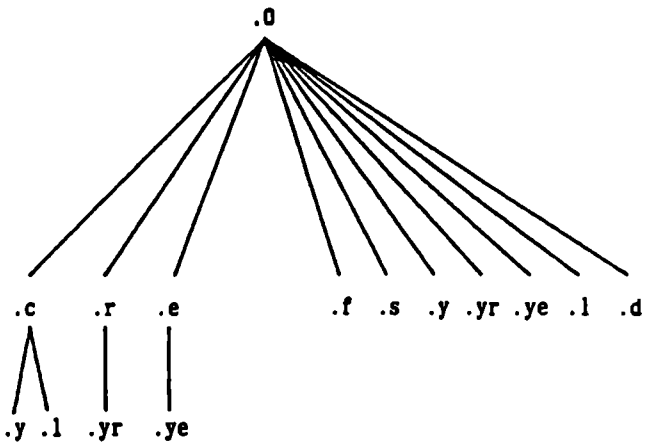

Fig. 2.1 — Summary of Default Transformation Path

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.

NOTES

3. AUGMENTED VERSION OF MAKE

GENERAL

This section describes an augmented version of the **make** command of the UNIX operating system. The augmented version is upward compatible with the old version. This section describes and gives examples of only the additional features. Further possible developments for **make** are also discussed. Some justification will be given for the chosen implementation, and examples will demonstrate the additional features.

The **make** command was perceived as an excellent program administrative tool and has been used extensively in at least one project for over 2 years. However, **make** had the following shortcomings:

- handling of libraries was tedious
- handling of the Source Code Control System (SCCS) file name format was difficult or impossible
- environment variables were completely ignored by **make**
- the general lack of ability to maintain files in a remote directory.

These shortcomings hindered large scale use of **make** as a program support tool.

The **make** program has been modified to handle the problems above. The additional features are within the original syntactic framework of **make** and few if any new syntactical entities have been introduced. A notable exception is the *include* file capability. Further, most of the additions result in a "Don't know how to make ..." message from the old version of **make**.

The following paragraphs describe with examples the additional features of the **make** program. In general, the examples are taken from existing *makefiles*. Also, the tables are examples of working *makefiles*.

THE ENVIRONMENT VARIABLES

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

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

1. Read the **MAKEFLAGS** environment variable. If it is not present or null, the internal **make** variable **MAKEFLAGS** is set to the null string. Otherwise, each letter in **MAKEFLAGS** is assumed to be an input flag argument and is processed as such. (The only exceptions are the **−f**, **−p**, and **−r** flags.)

2. Read and set the input flags from the command line. The command line adds to the previous settings from the **MAKEFLAGS** environment variable.

3. Read macro definitions from the command line. These are made *not resettable*. Thus, any further assignments to these names are ignored.

4. Read the internal list of macro definitions. These are found in the file *rules.c* of the source for **make**. (See Table 3.A for the complete makefile which represents the internally defined macros and rules of the

current version of **make**. Thus, if **make −r ...** is typed and a *makefile* includes the *makefile* in Table 3.A, the results would be identical to excluding the **−r** option and the *include* line in the *makefile*. The Table 3.A output can be reproduced by the following:

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

The output will appear on the standard output.) They give default definitions for the C language compiler (CC=cc), the assembler (AS=as), etc.

5. Read the environment. The environment variables are treated as macro definitions and marked as *exported* (in the shell sense). (***Note:*** **MAKEFLAGS** will be read and set again.) However, since **MAKEFLAGS** is not an internally defined variable (in *rules.c*), this has the effect of doing the same assignment twice. The exception to this is when **MAKEFLAGS** is assigned on the command line. (The reason it was read previously was to turn the debug flag on before anything else was done.)

6. Read the *makefile(s)*. The assignments in the *makefile(s)* will override the environment. This order was chosen so when a *makefile* is read and executed the user knows what to expect. That is, the user gets what is seen unless the **−e** flag is used. The **−e** is an additional command line flag which tells **make** to have the environment override the *makefile* assignments. Thus, if **make −e ...** is typed, the variables in the environment override the definitions in the *makefile*. (***Note:*** There is no way to override the command line assignments.) Also note that **MAKEFLAGS** will override the environment if assigned. (This would be useful for further invocations of **make** from the current *makefile*.)

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

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

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

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

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

RECURSIVE MAKEFILES

Another feature was added to **make** concerning the environment and recursive invocations. If the sequence "$(MAKE)" appears anywhere in a shell command line, the line will be executed even if the **−n** flag is set. Since the **−n** flag is exported across invocations of **make** (through the **MAKEFLAGS** variable), the only thing which will actually get executed is the **make** command itself. This feature is useful when a hierarchy of

makefile(s) describes a set of software subsystems. For testing purposes, **make -n ...** can be executed and everything that would have been done will get printed out including output from lower level invocations of **make**.

FORMAT OF SHELL COMMANDS WITHIN make

The **make** program remembers embedded new lines and tabs in shell command sequences. Thus, if the programmer puts a *for* loop in the makefile with indentation, when **make** prints it out, it retains the indentation and backslashes. The output can still be piped to the shell and is readable. This is obviously a cosmetic change; no new function is gained.

ARCHIVE LIBRARIES

The **make** program has an improved interface to archive libraries. Due to a lack of documentation, most people are probably not aware of the current syntax of addressing members of archive libraries. The previous version of **make** allows a user to name a member of a library in the following manner:

```
lib(object.o)
      or
lib((_localtime))
```

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

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

```
lib:: lib(ctime.o)
      $(CC) -c -O ctime.c
      ar rv lib ctime.o
      rm ctime.o
lib:: lib(fopen.o)
      $(CC) -c -O fopen.c
      ar rv lib fopen.o
      rm fopen.o
...and so on for each object ...
```

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

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

The above 2-member library is then maintained with the following shorter makefile:

```
lib: lib(ctime.o) lib(fopen.o)
 echo lib up-to-date.
```

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

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

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

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

```
lib: lib (ctime.o)
@echo lib up-to-date
```

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

1. Do *lib*.
2. To do *lib*, do each dependent of *lib*.
3. Do *lib* (*ctime.o*).
4. To do *lib*(*ctime.o*), do each dependent of *lib* (*ctime.o*). (There are none.)
5. Use internal rules to try to build *lib*(*ctime.o*). (There is no explicit rule.) Note that *lib*(*ctime.o*) has a parenthesis in the name so identify the target suffix as ".a". This is the key. There is no explicit ".a" at the end of the *lib* library name. The parenthesis forces the ".a" suffix. In this sense, the ".a" is hard-wired into **make**.
6. Break the name *lib*(*ctime.o*) up into *lib* and *ctime.o*. Define two macros, $@ (=*lib*) and $* (=*ctime*).
7. Look for a rule ".X.a" and a file $*.X. The first ".X" (in the .SUFFIXES list) which fulfills these conditions is ".c" so the rule is ".c.a" and the file is *ctime.c*. Set $< to be *ctime.c* and execute the rule. (In fact, **make** must then do *ctime.c*. However, the search of the current directory yields no other candidates, whence, the search ends.)
8. The library has been updated. Do the rule associated with the "lib:" dependency; namely:

```
echo lib up-to-date
```

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

```
lib(ctime.o):        $(INCDIR)/stdio.h
```

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

An example *makefile* for a larger library is given in Table 3.B. The reader will note also that there are no lingering "*.o" files left around. The result is a library maintained directly from the source files (or more generally from the SCCS files).

SOURCE CODE CONTROL SYSTEM FILE NAMES: THE TILDE

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

To allow **make** easy access to the prefix "s." requires either a redefinition of the rule naming syntax of **make** or a trick. The trick is to use the tilde (~) as an identifier of SCCS files. Hence, ".c~ .o" refers to the rule which transforms an SCCS C language source file into an object. Specifically, the internal rule is:

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

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

The following SCCS suffixes are internally defined:

```
.c~
.y~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

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

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

THE NULL SUFFIX

In the UNIX system source code, there are many commands which consist of a single source file. It was wasteful to maintain an object of such files for **make's pleasure.** The current implementation supports single suffix rules (a null suffix). Thus, to maintain the program **cat**, a rule in the *makefile* of the following form is needed:

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

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

```
make cat dd echo date
```

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

```
.c:
.c~:
.sh:
.sh~:
```

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

INCLUDE FILES

The **make** program has an include file capability. If the string *include* appears as the first seven letters of a line in a *makefile* and is followed by a blank or a tab, the following string is assumed to be a file name which the current invocation of **make** will read. The file descriptors are stacked for reading *include* files so no more than about 16 levels of nested *includes* are supported.

INVISIBLE SCCS MAKEFILES

The SCCS *makefiles* are invisible to **make**. That is, if **make** is typed and only a file named *s.makefile* exists, **make** will do a **get** on the file, then read and remove the file. Using the **−f**, **make** will get, read, and remove arguments and *include* files.

DYNAMIC DEPENDENCY PARAMETERS

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

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

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

```
CMDS = cat dd echo date cc cmp comm ar ld chown

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

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

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

```
INCDIR = /usr/include

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

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

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

EXTENSIONS OF $*, $@, AND $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(*D)**,**$(*F)**,**$(<D)**,and **$(<F)**. The "D" refers to the directory part of the single letter macro. The "F" refers to the file name part of the single letter macro. These additions are useful when building hierarchical makefiles. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a shell command can be:

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

An interesting example of the use of these features can be found in the set of *makefiles* in Table 3.C. Each *makefile* is named "70.mk". The following command forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is *uch*. The FRC.is a convention for *FoRC*ing **make** to completely rebuild a target starting from scratch.

OUTPUT TRANSLATIONS

Macros in shell commands can now be translated when evaluated. The form is as follows:

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

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

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

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

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

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

TABLE 3.A

EXAMPLE OF INTERNAL DEFINITIONS

```
#               LIST OF SUFFIXES

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


#               PRESET VARIABLES

MAKE=make
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
CC=cc
CFLAGS=-0
AS=as
ASFLAGS=
GET=get
GFLAGS=


#               SINGLE SUFFIX RULES

.c:
                $(CC) -n -0 $< -o $@
.c~:
                $(GET) $(GFLAGS) -p $< > $*.c
                $(CC) -n -0 $*.c -o $*
                -rm -f $*.c
.sh:
                cp $< $@
.sh~:
                $(GET) $(GFLAGS) -p $< > $*.sh
                cp $*.sh $*
                -rm -f $*.sh
```

TABLE 3.A (Contd)

EXAMPLE OF INTERNAL DEFINITIONS

```
#
#               DOUBLE SUFFIX RULES
#

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

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

.c~.c:
        $(GET) $(GFLAGS) -p $< > $*.c

.s.o:
        $(AS) $(ASFLAGS) -o $@ $<

.s~.o:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s
        -rm -f $*.s

.y.o:
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@

.y~.o:
        $(GET) $(GFLAGS) -p $< > $*.y
        $(YACC) $(YFLAGS) $*.y
        $(CC) $(CFLAGS) -c y.tab.c
        rm -f y.tab.c $*.y
        mv y.tab.o $*.o

.l.o:
        $(LEX) $(LFLAGS) $<
        $(CC) $(CFLAGS) -c lex.yy.c
        rm lex.yy.c
        mv lex.yy.o $@

.l~.o:
        $(GET) $(GFLAGS) -p $< > $*.l
        $(LEX) $(LFLAGS) $*.l
        $(CC) $(CFLAGS) -c lex.yy.c
        rm -f lex.yy.c $*.l
        mv lex.yy.o $*.o
```

TABLE 3.A (Contd)

EXAMPLE OF INTERNAL DEFINITIONS

```
.y.c:
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
.y~.c:
        $(GET) $(GFLAGS) -p $< > $*.y
        $(YACC) $(YFLAGS) $*.y
        mv y.tab.c $*.c
        -rm -f $*.y
.l.c:
        $(LEX) $<
        mv lex.yy.c $@
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
.c~.a:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) -c $(CFLAGS) $*.c
        ar rv $@ $*.o
        rm -f $*.[co]
.s~.a:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s
        ar rv $@ $*.o
        -rm -f $*.[so]
.h~.h:
        $(GET) $(GFLAGS) -p $< > $*.h
```

TABLE 3.B

EXAMPLE OF LIBRARY MAKEFILE

```
#                   @(#)/usr/src/cmd/make/make.tm   3.2


LIB = lsxlib


PR = vpr -b LSX


INSDIR = /r1/flop0/
INS = eval


lsx:                $(LIB) low.o mch.o
                    ld -x low.o mch.o $(LIB)
                    mv a.out lsx
                    @size lsx


#                   Here, $(INS) as either ":" or "eval".


lsx:
                    $(INS) 'cp lsx $(INSDIR)lsx . .
                            strip $(INSDIR)lsx . .
                            ls -l $(INSDIR)lsx'


print:
                    $(PR) header.s low.s mch.s *.h *.c Makefile
```

TABLE 3.B (Contd)

EXAMPLE OF LIBRARY MAKEFILE

```
$(LIB):
                $(LIB)(clock.o)
                $(LIB)(main.o)
                $(LIB)(tty.o)
                $(LIB)(trap.o)
                $(LIB)(sysent.o)
                $(LIB)(sys2.o)
                $(LIB)(sys3.o)
                $(LIB)(sys4.o)
                $(LIB)(sys1.o)
                $(LIB)(sig.o)
                $(LIB)(fio.o)
                $(LIB)(kl.o)
                $(LIB)(alloc.o)
                $(LIB)(nami.o)
                $(LIB)(iget.o)
                $(LIB)(rdwri.o)
                $(LIB)(subr.o)
                $(LIB)(bio.o)
                $(LIB)(decfd.o)
                $(LIB)(slp.o)
                $(LIB)(space.o)
                $(LIB)(puts.o)
                @echo $(LIB) now up-to-date.

.s.o:
                as -o $*.o header.s $*.s
.o.a:
                ar rv $@ $<
                rm -f $<

.s.a:
                as -o $*.o header.s $*.s
                ar rv $@ $*.o
                rm -f $*.o

.PRECIOUS:      $(LIB)
```

TABLE 3.C

RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
                                        ./ucb makefile


#       @(#)/usr/src/cmd/make/make.tm   3.2
#       ucb/~0.mk makefile


VERSION = ~0


DEPS =
            os/low.$(VERSION).o
            os/mch.$(VERSION).o
            os/conf.$(VERSION).o
            os/lib1.$(VERSION).a
            io/lib2.$(VERSION).a



#           This makefile will reload the UNIX system file
#           unix.$(VERSION) if any of the $(DEPS) is out-of-date
#           [wrt unix.$(VERSION)]. (Note: It will not go out and
#           check each member of the libraries. To do this, the FRC
#           macro must be defined.)
#


unix.$(VERSION):        $(DEPS) $(FRC)
        load -s $(VERSION)



$(DEPS):                $(FRC)
        cd $(@D); $(MAKE) -f $(VERSION).mk $(@F)



all:                    unix.$(VERSION)
        @echo unix.$(VERSION) up-to-date.
```

TABLE 3.C (Contd)

RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
includes:
        cd head/sys; $(MAKE) -f $(VERSION).mk

FRC:    includes;

#       @(#)/usr/src/cmd/make/make.tm   3.2
#       ucb/os/~0.mk makefile


VERSION = ~0


LIB = lib1.$(VERSION).a


COMPOOL =


LIBOBJS =


                $(LIB)(main.o)
                $(LIB)(alloc.o)
                $(LIB)(iget.o)
                $(LIB)(prf.o)
                $(LIB)(rdwri.o)
                $(LIB)(slp.o)
                $(LIB)(subr.o)
                $(LIB)(text.o)
                $(LIB)(trap.o)
                $(LIB)(sig.o)
                $(LIB)(sysent.o)
                $(LIB)(sys1.o)
                $(LIB)(sys2.o)
                $(LIB)(sys3.o)
                $(LIB)(sys4.o)
                $(LIB)(sys5.o)
                $(LIB)(syscb.o)
                $(LIB)(maus.o)
                $(LIB)(messag.o)
                $(LIB)(nami.o)
                $(LIB)(fio.o)
                $(LIB)(clock.o)
                $(LIB)(acct.o)
                $(LIB)(errlog.o)
```

TABLE 3.C (Contd)

RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
VERSION = `0


LIB = lib2.$(VERSION).a


COMPOOL =


LIB2OBJS =


        $(LIB)(mx1.o)
        $(LIB)(mx2.o)
        $(LIB)(bio.o)
        $(LIB)(tty.o)
        $(LIB)(malloc.o)
        $(LIB)(pipe.o)
        $(LIB)(dhdm.o)
        $(LIB)(dh.o)
        $(LIB)(dhfdm.o)
        $(LIB)(dj.o)
        $(LIB)(dn.o)
        $(LIB)(ds40.o)
        $(LIB)(dz.o)
        $(LIB)(alarm.o)
        $(LIB)(hf.o)
        $(LIB)(hps.o)
        $(LIB)(hpmap.o)
        $(LIB)(hp45.o)
        $(LIB)(hs.o)
        $(LIB)(ht.o)
        $(LIB)(jy.o)
        $(LIB)(kl.o)
        $(LIB)(lfh.o)
        $(LIB)(lp.o)
        $(LIB)(mem.o)
        $(LIB)(nmpipe.o)
        $(LIB)(rf.o)
        $(LIB)(rk.o)
        $(LIB)(rp.o)
        $(LIB)(rx.o)
        $(LIB)(sys.o)
        $(LIB)(trans.o)
        $(LIB)(ttdma.o)
```

TABLE 3.C (Contd)

RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
ALL =
        conf.$(VERSION).o
        low.$(VERSION).o
        mch.$(VERSION).o
        $(LIB)


all:        $(ALL)
        @echo '$(ALL)' now up-to-date.


$(LIB)::                $(LIBOBJS)


$(LIBOBJS):             $(FRC);


FRC:
        rm -f $(LIB)


clobber:                cleanup
        -rm -f $(LIB)


clean cleanup:;


install:                all;


.PRECIOUS:              $(LIB)
#       @(#)/usr/src/cmd/make/make.tm   3.2
#       ucb/io/~0.mk makefile
```

TABLE 3.C (Contd)

RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
                $(LIB)(tec.o)
                $(LIB)(tex.o)
                $(LIB)(tm.o)
                $(LIB)(vp.o)
                $(LIB)(vs.o)
                $(LIB)(vtlp.o)
                $(LIB)(vtll.o)
                $(LIB)(fakevtlp.o)
                $(LIB)(vt61.o)
                $(LIB)(vt100.o)
                $(LIB)(vtmon.o)
                $(LIB)(vtdbg.o)
                $(LIB)(vtutil.o)
                $(LIB)(vtast.o)
                $(LIB)(partab.o)
                $(LIB)(rh.o)
                $(LIB)(devstart.o)
                $(LIB)(dmcll.o)
                $(LIB)(rop.o)
                $(LIB)(ioctl.o)
                $(LIB)(fakemx.o)

all:                    $(LIB)
        @echo $(LIB) is now up-to-date.

$(LIB)::                $(LIB20BJS)

$(LIB20BJS):            $(FRC)

FRC:
        rm -f $(LIB)
```

TABLE 3.C (Contd)

RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
clobber: cleanup
        -rm -f $(LIB) *.o



clean cleanup:;



install:                all;



.PRECIOUS:              $(LIB)



.s.a:
        $(AS) $(ASFLAGS) -o $*.o $<
        ar rcv $@ $*.o
        rm $*.o



#       @(#)/usr/src/cmd/make/make.tm   3.2
#       ucb/head/sys/~0.mk makefile



COMPOOL = /usr/include/sys



HEADERS =



        $(COMPOOL)/buf.h
        $(COMPOOL)/bufx.h
        $(COMPOOL)/conf.h
        $(COMPOOL)/confx.h
        $(COMPOOL)/crtctl.h
```

TABLE 3.C (Contd)

RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
$(COMPOOL)/dir.h
$(COMPOOL)/dm11.h
$(COMPOOL)/elog.h
$(COMPOOL)/file.h
$(COMPOOL)/filex.h
$(COMPOOL)/filsys.h
$(COMPOOL)/ino.h
$(COMPOOL)/inode.h
$(COMPOOL)/inodex.h
$(COMPOOL)/ioctl.h
$(COMPOOL)/ipcomm.h
$(COMPOOL)/ipcommx.h
$(COMPOOL)/lfsh.h
$(COMPOOL)/lock.h
$(COMPOOL)/maus.h
$(COMPOOL)/mx.h
$(COMPOOL)/param.h
$(COMPOOL)/proc.h
$(COMPOOL)/procx.h
$(COMPOOL)/reg.h
$(COMPOOL)/seg.h
$(COMPOOL)/sgtty.h
$(COMPOOL)/sigdef.h
$(COMPOOL)/sprof.h
$(COMPOOL)/sprofx.h
$(COMPOOL)/stat.h
$(COMPOOL)/syserr.h
$(COMPOOL)/sysmes.h
$(COMPOOL)/sysmesx.h
$(COMPOOL)/systm.h
$(COMPOOL)/text.h
$(COMPOOL)/textx.h
$(COMPOOL)/timeb.h
$(COMPOOL)/trans.h
$(COMPOOL)/tty.h
$(COMPOOL)/ttyx.h
$(COMPOOL)/types.h
$(COMPOOL)/user.h
$(COMPOOL)/userx.h
$(COMPOOL)/version.h
$(COMPOOL)/votrax.h
$(COMPOOL)/vt11.h
$(COMPOOL)/vtmn.h
```

TABLE 3.C (Contd)

RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
all:                    $(FRC) $(HEADERS)
        @echo Headers are now up to date.


$(HEADERS):             s.$$(@F)
        $(GET) -s -p $(GFLAGS) $? > xtemp
        move xtemp 444 src sys $@


FRC:
        rm -f $(HEADERS)


.PRECIOUS:              $(HEADERS)


.h~.h:
        get -s $<


.DEFAULT:
        cpmv $? 444 src sys $@
```

4. SOURCE CODE CONTROL SYSTEM USER'S GUIDE

GENERAL

The Source Code Control System (SCCS) is a collection of the UNIX software commands which help individuals or projects control and account for changes to files of text. The source code and documentation of software systems are typical examples of files of text to be changed. The SCCS is a collection of programs that run under the UNIX operating system. It is convenient to conceive of SCCS as a custodian of files. The SCCS provides facilities for the following:

- Storing files of text
- Retrieving particular versions of the files
- Controlling updating privileges to files
- Identifying the version of a retrieved file
- Recording when, where, and why the change was made and who made each change to a file.

These types of facilities are important when programs and documentation undergo frequent changes because of maintenance and/or enhancement work. It is often desirable to regenerate the version of a program or document as it existed before changes were applied to it. This can be done by keeping copies (on paper or other media), but this method quickly becomes unmanageable and wasteful as the number of programs and documents increases. The SCCS provides an attractive solution because the original file is stored on disk. Whenever changes are made to the file, the SCCS stores only the changes. Each set of changes is called a "delta".

This section, together with relevant portions of the UNIX System User's Manual is a complete user's guide to SCCS. The following topics are covered:

- SCCS for Beginners: How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- How Deltas Are Numbered: How versions of SCCS files are numbered and named.
- SCCS Command Conventions: Conventions and rules generally applicable to all SCCS commands.
- SCCS Commands: Explanation of all SCCS commands, with discussions of the more useful arguments.
- SCCS Files: Protection, format, and auditing of SCCS files including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

Neither the implementation of SCCS nor the installation procedure for SCCS are described in this section.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name**(N), where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a UNIX system, create files, and use the text editor. A number of terminal-session fragments are presented. All of them should be tried since the best way to learn SCCS is to use it.

To supplement the material in this section, the detailed SCCS command descriptions in the UNIX System User's Manual should be consulted.

A. Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *S*CCS *ID*entification string (SID). The SID is composed of at most four components. The first two components are the "release" and "level" numbers which are separated by a period. Hence, the first delta (for the original file) is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.1", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

B. Creating an SCCS File via "admin"

Consider, for example, a file called *lang* that contains a list of programming languages:

```
c
pl/i
fortran
cobol
algol
```

Custody of the *lang* file can be given to SCCS. The following **admin** command (used to "administer" SCCS files) creates an SCCS file and initializes delta 1.1 from the file *lang*:

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

All SCCS files must have names that begin with "s.", hence, *s.lang*. The **-i** keyletter, together with its value *lang*, indicates that **admin** is to create a new SCCS file and "initialize" the new SCCS file with the contents of the file *lang*. This initial version is a set of changes (delta 1.1) applied to the null SCCS file.

The **admin** command replies

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

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described under the **get** command in the part "SCCS COMMANDS". In the following examples, this warning message is not shown, although it may actually be issued by the various commands. The file *lang* should now be removed (because it can be easily reconstructed using the **get** command) as follows:

```
rm lang
```

C. Retrieving a File via "get"

The *lang* file can be reconstructed by using the following **get** command:

```
get s.lang
```

The command causes the creation (retrieval) of the latest version of file *s.lang* and prints the following messages:

```
1.1
5 lines
```

This means that **get** retrieved version 1.1 of the file, which is made up of five lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file. Hence, the file *lang* is created.

The "get s.lang" command simply creates the file *lang* (read-only) and keeps no information regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the **delta** command, the **get** command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The **-e** keyletter causes **get** to create a file *lang* for both reading and writing (so it may be edited) and places certain information about the SCCS file in another new file. The new file, called the *p-file*, will be read by the **delta** command. The **get** command prints the same messages as before except that the SID of the version to be created through the use of **delta** is also issued. For example:

```
get -e s.lang
1.1
new delta 1.2
5 lines
```

The file *lang* may now be changed, for example, by:

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

D. Recording Changes via "delta"

In order to record within the SCCS file the changes that have been applied to *lang*, execute the following command:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made. For example:

```
comments? added more languages
```

The **delta** command then reads the *p-file* and determines what changes were made to the file *lang*. The **delta** command does this by doing its own **get** to retrieve the original version and by applying the **diff**(1) command to the original version and the edited version.

When this process is complete, at which point the changes to *lang* have been stored in *s.lang*, **delta** outputs:

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

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file *s.lang*.

E. Additional Information About "get"

As shown in the previous example, the command

```
get s.lang
```

retrieves the latest version (now 1.2) of the file *s.lang*. This is done by starting with the original version of the file and successively applying deltas (the changes) in order until all have been applied.

In the example chosen, the following commands are all equivalent:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the **-r** keyletter are SIDs. Note that omitting the level number of the SID (as in "get -r1 s.lang") is equivalent to specifying the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the release number (first component of the SID) of the delta being made. Since normal automatic numbering of deltas proceeds by incrementing the level number (second component of the SID), the user must indicate to SCCS the need to change the release number. This is done with the **get** command:

```
get -e -r2 s.lang
```

Because release 2 does not exist, **get** retrieves the latest version *before* release 2. The **get** command also interprets this as a request to change the release number of the delta the user desires to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to **delta** via the *p-file*. The **get** command then outputs

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version **delta** will create. If the file is now edited, for example, by:

```
ed lang
41
/cobol/d
w
35
q
```

and **delta** executed:

```
delta s.lang
comments? deleted cobol from list of languages
```

the user will see by **delta**'s output that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

F. The "help" Command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (co1)
```

The string "co1" is a code for the diagnostic message and may be used to obtain a fuller explanation of that message by use of the **help** command:

```
help co1
```

This produces the following output:

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

Thus, **help** is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Detailed explanations of almost all SCCS messages may be found in this manner.

DELTA NUMBERING

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the release number when making a delta to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas unless specifically changed again. Thus, the evolution of a particular file may be represented as in Fig. 4.1.

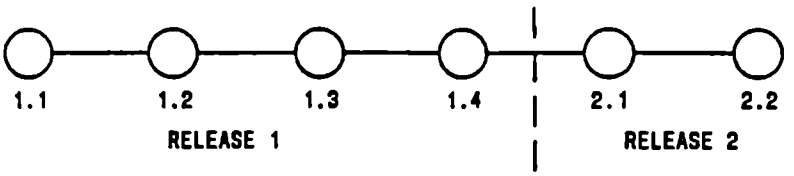

Fig. 4.1— Evolution of an SCCS File

Such a structure may be termed the "trunk" of the SCCS tree. Figure 4.1 represents the normal sequential development of an SCCS file in which changes that are part of any given delta are dependent upon all the preceding deltas.

However, there are situations in which it is necessary to cause a branching in the tree in that changes applied as part of a given delta are not dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3 and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas precisely as shown in Fig. 4.1. Assume that a production user reports a problem in version 1.3 and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a branch of the tree, and its name consists of four components; the release and level numbers, as with trunk deltas, plus the "branch" and "sequence" numbers. The delta name will appear as follows:

release.level.branch.sequence

The branch number is assigned to each branch that is a descendant of a particular trunk delta with the first such branch being 1, the next one 2, etc. The sequence number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Fig. 4.2.

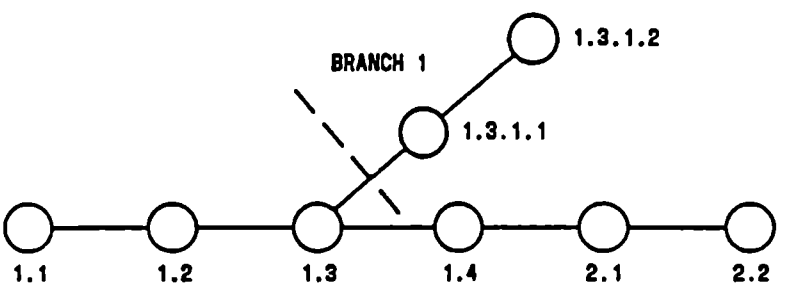

Fig. 4.2 — Tree Structure With Branch Deltas

The concept of branching may be extended to any delta in the tree. The naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain

exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is not possible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.n. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n (see Fig. 4.3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the chronologically second delta on the chronologically second branch whose trunk ancestor is delta 1.3. In particular, it is not possible to determine from the name of delta 1.3.2.2 all the deltas between it and trunk ancestor 1.3.

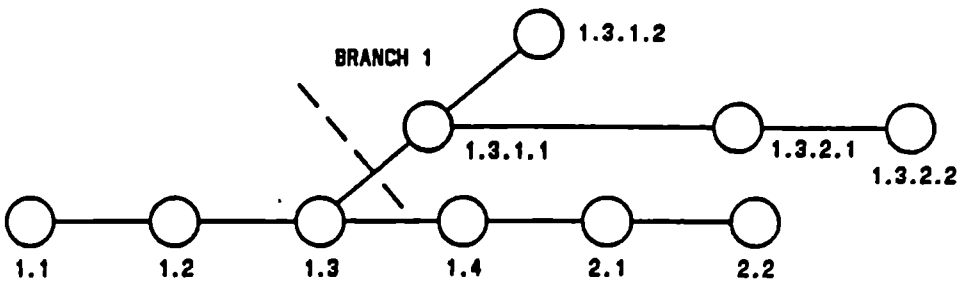

Fig. 4.3 — Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

SCCS COMMAND CONVENTIONS

This part discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to all SCCS commands with exceptions indicated. The SCCS commands accept two types of arguments:

- keyletter arguments
- file arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (−), followed by a lowercase alphabetic character, and in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files [because of permission modes via **chmod**(1)] in the named directories are silently ignored.

In general, file arguments may not begin with a minus sign. However, if the name "−" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line

as the name of an SCCS file to be processed. The standard input is read until end of file. This feature is often used in pipelines with, for example, the **find**(1) or **ls**(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to all file arguments of that command. All keyletters are processed before any file arguments with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right. Somewhat different argument conventions apply to the **help**, **what**, **sccsdiff**, and **val** commands.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags are discussd in this part. For a complete description of all such flags, see **admin**(1) section in the UNIX System User's Manual.

The distinction between the real user [see **passwd**(1)] and the effective user of a UNIX system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a UNIX system). This subject is discussed further in "SCCS FILES".

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file*, which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, given the same mode [see **chmod**(1)] as the SCCS file, and owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file*, called the *z-file*, whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the process number of the command that creates it, and its existence is an indication to other commands that the SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files*. The files may be useful in the event of system crashes or similar situations.

The SCCS commands produce diagnostics (on the diagnostic output) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The code in parentheses may be used as an argument to the **help** command to obtain a further explanation of the diagnostic message. Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of that file and to proceed with the next file, in order, if more than one file has been named.

SCCS COMMANDS

This part describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the UNIX System User's Manual and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

The commands follow in approximate order of importance. The following is a summary of all the SCCS commands and of their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.
admin	Creates SCCS files and applies changes to parameters of SCCS files.

prs	Prints portions of an SCCS file in user specified format.
help	Gives explanations of diagnostic messages.
rmdel	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the **get** command.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
val	Validates an SCCS file.

A. The "get" Command

The **get** command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file*. The *g-file* name is formed by removing the "s." from the SCCS file name. The *g-file* is created in the current directory and is owned by the real user. The mode assigned to the *g-file* depends on how the **get** command is invoked.

The most common invocation of **get** is:

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file.

The generated *g-file* (file "abc") is given mode 444 (read-only) since this particular way of invoking **get** is intended to produce *g-files* only for inspection, compilation, etc., and not for editing (i.e., not for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```

produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)
s.def:
1.7
85 lines
No id keywords (cm7)
```

ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc. within the *g-file,* so this information will appear in a load module when one is eventually created. The SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%). For example:

```
%I%
```

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, **%H%** is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and **%M%** is defined as the name of the *g-file.* Thus, executing **get** on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by **get**, the following message is issued:

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

This message is normally treated as a warning by **get**, although the presence of the **i** flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately 20 ID keywords provided, see **get**(1) in the UNIX System User's Manual.

Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the trunk of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **−r** keyletter of **get**.

The **−r** keyletter is used to specify a SID to be retrieved, in which case the **d** (default SID) flag (if any) is ignored. For example:

```
get −r1.3 s.abc
```

retrieves version 1.3 of file *s.abc* and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly:

```
get -r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3
234 lines
```

When a 2- or 4-component SID is specified as a value for the -r keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get -r3 s.abc
```

causes retrieval of the trunk delta with the highest level number within the given release if the given release exists. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file *s.abc* and that release 7 is actually the highest-numbered release below 9, execution of:

```
get -r9 s.abc
```

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file *s.abc* below release 9. Similarly, omission of the sequence number, as in:

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

The -t keyletter is used to retrieve the latest (top) version in a particular release (i.e., when no -r keyletter is supplied or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get -r3 -t s.abc
```

might produce:

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5
46 lines
```

Retrieval With Intent to Make a Delta

Specification of the **-e** keyletter to the **get** command is an indication of the intent to make a delta, and as such, its use is restricted. The presence of this keyletter causes **get** to check:

1. The user list (a list of login names and/or group IDs of users allowed to make deltas) to determine if the login name or group ID of the user executing **get** is on that list. Note that a null (empty) user list behaves as if it contained all possible login names.

2. The release (R) of the version being retrieved satisfies the relation:

   floor is < or = to R which is
   < or = to ceiling

   to determine if the release being accessed is a protected release. The "floor" and "ceiling" are specified as flags in the SCCS file.

3. The release (R) is not locked against editing. The "lock" is specified as a flag in the SCCS file.

4. Whether or not multiple concurrent edits are allowed for the SCCS file as specified by the **j** flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the **-e** keyletter causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable *g-file* already exists, **get** terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are not substituted by **get** when the **-e** keyletter is specified because the generated *g-file* is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, **get** does not need to check for the presence of ID keywords within the *g-file*, so the message

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

is never output when **get** is invoked with the **-e** keyletter.

In addition, the **-e** keyletter causes the creation (or updating) of a *p-file* which is used to pass information to the **delta** command.

The following is an example of the use of the **-e** keyletter:

```
get -e s.abc
```

which produces (for example) on the standard output:

```
1.3
new delta 1.4
67 lines
```

If the **-r** and/or **-t** keyletters are used together with the **-e** keyletter, the version retrieved for editing is as specified by the **-r** and/or **-t** keyletters.

The keyletters **−i** and **−x** may be used to specify a list [see **get**(1) in the UNIX System User's Manual for the syntax of such a list] of deltas to be included and excluded, respectively, by **get**. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be not applied. This may be used to undo in the version of the SCCS file to be created the effects of a previous delta. Whenever deltas are included or excluded, **get** checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*. Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

Warning: ***The −i and −x keyletters should be used with extreme care.***

The **−k** keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of **get** with the **−e** keyletter or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the **−k** keyletter is identical to one produced by **get** executed with the **−e** keyletter. However, no processing related to the *p-file* takes place.

Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of **get** commands with the **−e** keyletter may be executed on the same file provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The *p-file* (created by the **get** command invoked with the **−e** keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The *p-file* contains the following information for each delta that is still "in progress":

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing **get**.

The first execution of **get −e** causes the creation of the *p-file* for the corresponding SCCS file. Subsequent executions only update the *p-file* with a line containing the above information. Before updating, however, **get** checks that no entry already in the *p-file* specifies as already retrieved the SID of the version to be retrieved unless multiple concurrent edits are allowed.

If both checks succeed, the user is informed that other deltas are in progress and processing continues. If either check fails, an error message results. It is important to note that the various executions of **get** should be carried out from different directories. Otherwise, only the first execution will succeed since subsequent executions would attempt to overwrite a writable *g-file*, which is an SCCS error condition. In practice, such multiple executions are performed by different users so that this problem does not arise since each user normally has a different working directory. See "Protection" under the part "SCCS FILES" for a discussion of how different users are permitted to use SCCS commands on the same files.

Table 4.A shows, for the most useful cases, the version of an SCCS file retrieved by **get**, as well as the SID of the version to be eventually created by **delta**, as a function of the SID specified to **get**.

Concurrent Edits of Same SID

Under normal conditions, **gets** for editing (**−e** keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, **delta** must be executed before a subsequent **get** for editing is executed at

the same SID as the previous **get**. However, multiple concurrent edits (defined to be two or more successive executions of **get** for editing based on the same retrieved SID) are allowed if the **j** flag is set in the SCCS file. Thus:

```
get -e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get -e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of **delta**. In this case, a **delta** command corresponding to the first **get** produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the **delta** command corresponding to the second **get** produces delta 1.1.1.1.

Keyletters That Affect Output

Specification of the **-p** keyletter causes **get** to write the retrieved text to the standard output rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names:

```
get -p s.abc > arbitrary-file-name
```

The **-p** keyletter is particularly useful when used with the "!" or "$" arguments of the **send**(1C) command. For example:

```
send MOD=s.abc REL=3 compile
```

given that file *compile* contains:

```
//plicomp job job-card-information
//step1 exec plickc
//pli.sysin dd *
 ~~s
 ~!get -p -rREL MOD
/*
//
```

will **send** the highest level of release 3 of file *s.abc*. Note that the line "~~s", which causes **send** to make ID keyword substitutions before detecting and interpreting control lines, is necessary if **send** is to substitute "s.abc" for MOD and "3" for REL in the line "~!get -p -rREL MOD".

The **-s** keyletter suppresses all output that is normally directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent nondiagnostic messages from appearing on the user's terminal and is often used in conjunction with the **-p** keyletter to "pipe" the output of **get**, as in:

```
get -p -s s.abc | nroff
```

The **-g** keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get -g -r4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file or it generates an error message if it does not. Another use of the **-g** keyletter is in regenerating a *p-file* that may have been accidentally destroyed:

```
get -e -g s.abc
```

The **-l** keyletter causes the creation of an *l-file*, which is named by replacing the "s." of the SCCS file name with "l.". This file is created in the current directory with mode 444 (read-only) and is owned by the real user. It contains a table (whose format is described in **get**(1) in the UNIX System User's Manual) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get -r2.3 -l s.abc
```

generates an *l-file* showing the deltas applied to retrieve version 2.3 of the SCCS file. Specifying a value of "p" with the **-l** keyletter, as in:

```
get -lp -r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*. The **-g** keyletter may be used with the **-l** keyletter to suppress the actual retrieval of the text.

The **-m** keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The **-n** keyletter causes each line of the generated *g-file* to be preceded by the value of the %M% ID keyword and a tab character. The **-n** keyletter is most often used in a pipeline with **grep**(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get -p -n -s directory |grep pattern
```

If both the **-m** and **-n** keyletters are specified, each line of the generated *g-file* is preceded by the value of the %M% ID keyword and a tab (this is the effect of the **-n** keyletter) and followed by the line in the format produced by the **-m** keyletter. Because use of the **-m** keyletter and/or the **-n** keyletter causes the contents of the *g-file* to be modified, such a *g-file* must not be used for creating a delta. Therefore, neither the **-m** keyletter nor the **-n** keyletter may be specified together with the **-e** keyletter.

See **get**(1) in the UNIX System User's Manual for a full description of additional **get** keyletters.

B. The "delta" Command

The **delta** command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and therefore, a new version of the file.

Invocation of the **delta** command requires the existence of a *p-file*. The **delta** command examines the *p-file* to verify the presence of an entry containing the user's login name. If none is found, an error message results. The **delta** command also performs the same permission checks that **get** performs when invoked with the **-e**

keyletter. If all checks are successful, **delta** determines what has been changed in the *g-file* by comparing it via **diff**(1) with its own temporary copy of the *g-file* as it was before editing. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal **get** at the SID specified in the *p-file* entry.

The required *p-file* entry is the one containing the login name of the user executing **delta** because the user who retrieved the *g-file* must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed **get** with the **−e** keyletter more than once on the same SCCS file), the **−r** keyletter must be used with **delta** to specify an SID that uniquely identifies the *p-file* entry. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of **delta** is

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a new line character. The user's response may be up to 512 characters long with new lines, not intended to terminate the response, escaped by backslash "\".

If the SCCS file has a **v** flag, **delta** first prompts with

```
MRs?
```

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?". In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests [MRs]) and that it is desirable or necessary to record such MR number(s) within each delta.

The **−y** and/or **−m** keyletters may be used to supply the commentary (comments and MR numbers, respectively) on the command line rather than through the standard input:

```
delta −y " descriptive comment "  −m " mrnum1 mrnum2 "  s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The **−m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when **delta** is executed from within a shell procedure [see **sh**(1) in the UNIX System User's Manual.]

The commentary (comments and/or MR numbers), whether solicited by **delta** or supplied via keyletters, is recorded as part of the entry for the delta being created and applies to all SCCS files processed by the same invocation of **delta**. This implies that if **delta** is invoked with more than one file argument and the first file named has a **v** flag all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, **delta** outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

It is possible that the counts of lines reported as inserted, deleted, or unchanged by **delta** do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and **delta** is likely to find a description that differs from the user's perception. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If in the process of making a delta **delta** finds no ID keywords in the edited *g-file*, the message

No id keywords (cm7)

is issued after the prompts for commentary but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by creating a delta from a *g-file* that was created by a **get** without the **-e** keyletter (recall that ID keywords are replaced by **get** in that case) or by accidentally deleting or changing the ID keywords during the editing of the *g-file*. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an **i** flag in the SCCS file indicating that this should be treated as a fatal error. In this last case, the delta is not created.

After processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*. All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file* which is described in the part "SCCS COMMAND CONVENTIONS". If there is only one entry in the *p-file*, then the *p-file* itself is removed.

In addition, **delta** removes the edited *g-file* unless the **-n** keyletter is specified. Thus:

delta -n s.abc

will keep the *g-file* upon completion of processing.

The **-s** (silent) keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **-s** keyletter together with the **-y** keyletter (and possibly, the **-m** keyletter) causes **delta** neither to read the standard input nor to write the standard output.

The differences between the *g-file* and the *d-file* (see above), which constitute the delta, may be printed on the standard output by using the **-p** keyletter. The format of this output is similar to that produced by **diff**(1).

C. The "admin" Command

The **admin** command is used to administer SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files. (Discussed in "Auditing" under the part "SCCS FILES".) Newly created SCCS files are given mode 444 (read-only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command upon that file.

Creation of SCCS Files

An SCCS file may be created by executing the command

admin -ifirst s.abc

in which the value "first" of the −i keyletter specifies the name of a file from which the text of the initial delta of the SCCS file *s.abc* is to be taken. Omission of the value of the −i keyletter indicates that **admin** is to read the standard input for the text of the initial delta. Thus, the command

```
admin −i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message

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

is issued by **admin** as a warning. However, if the same invocation of the command also sets the **i** flag (not to be confused with the −i keyletter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using the −i keyletter.

When an SCCS file is created, the release number assigned to its first delta is normally "1", and its level number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The −r keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin −ifirst −r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the −i keyletter.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (−y keyletter) and/or MR numbers (−m keyletter) in exactly the same manner as for **delta**. The creation of an SCCS file may sometimes be the direct result of an MR. If comments (−y keyletter) are omitted, a comment line of the form

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

is automatically generated.

If it is desired to supply MR numbers (−m keyletter), the **v** flag must also be set (using the −f keyletter described below). The **v** flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a "delta commentary" [see **sccsfile**(4) in the UNIX System User's Manual] in the SCCS file. Thus:

```
admin −ifirst −mmrnum1 −fv s.abc
```

Note that the −y and −m keyletters are only effective if a new SCCS file is being created.

Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for descriptive text may be initialized or changed through the use of the −t keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file.

When an SCCS file is being created and the −t keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

```
admin −ifirst −tdesc s.abc
```

specifies that the descriptive text is to be taken from file *desc*.

When processing an *existing* SCCS file, the **-t** keyletter specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
admin -tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of *desc*; omission of the file name after the **-t** keyletter as in

```
admin -t s.abc
```

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized, changed, or deleted through the use of the **-f** and **-d** keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See **admin**(1) in the UNIX System User's Manual for a description of all the flags. For example, the **i** flag specifies that the warning message stating there are no ID keywords contained in the SCCS file should be treated as an error, and the **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command. The **-f** keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin -ifirst -fi -fmmodname s.abc
```

sets the **i** flag and the **m** (module name) flag. The value "modname" specified for the **m** flag is the value that the **get** command will use to replace the %M% ID keyword. (In the absence of the **m** flag, the name of the *g-file* is used as the replacement for the %M% ID keyword.) Note that several **-f** keyletters may be supplied on a single invocation of **admin** and that **-f** keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The **-d** keyletter is used to delete a flag from an SCCS file and may only be specified when processing an existing file. As an example, the command

```
admin -dm s.abc
```

removes the **m** flag from the SCCS file. Several **-d** keyletters may be supplied on a single invocation of **admin** and may be intermixed with **-f** keyletters.

The SCCS files contain a list (user list) of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default which implies that anyone may create deltas. To add login names and/or group IDs to the list, the **-a** keyletter is used. For example:

```
admin -axyz -awql -a1234 s.abc
```

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The **-a** keyletter may be used whether **admin** is creating a new SCCS file or processing an existing one and may appear several times. The **-e** keyletter is used in an analogous manner if one wishes to remove (erase) login names or group IDs from the list.

D. The "prs" Command

The **prs** command is used to print on the standard output all or parts of an SCCS file in a format, called the output "data specification," supplied by the user via the **-d** keyletter. The data specification is a string consisting of SCCS file data keywords (not to be confused with **get** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example:

```
:I:
```

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, :F: is defined as the data keyword for the SCCS file name currently being processed, and :C: is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see **prs**(1) in the UNIX System User's Manual.

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example:

```
prs -d ":I: this is the top delta for :F:  :I: "  s.abc
```

may produce on the standard output

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying the SID of that delta using the **-r** keyletter. For example:

```
prs -d ":F: : :I: comment line is: :C: "  -r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If the **-r** keyletter is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained by specifying the **-l** or **-e** keyletters. The **-e** keyletter substitutes data keywords for the SID designated via the **-r** keyletter and all deltas created earlier. The **-l** keyletter substitutes data keywords for the SID designated via the **-r** keyletter and all deltas created later. Thus, the command

```
prs -d:I: -r1.4 -e s.abc
```

may output

```
1.4
1.3
1.2.1.1
1.2
1.1
```

and the command

```
prs -d:I: -r1.4 -l s.abc
```

may produce

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both the −e and −l keyletters.

E. The "help" Command

The **help** command prints explanations of SCCS commands and of messages that these commands may print. Arguments to **help**, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, **help** prompts for one. The **help** command has no concept of keyletter arguments or file arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will not terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

```
help ge5 rmdel
```

produces

```
ge5:
" nonexistent sid "
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
   rmdel −rSID name ...
```

F. The "rmdel" Command

The **rmdel** command is provided to allow removal of a delta from an SCCS file. Its use should be reserved for those cases in which incorrect global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Fig. 4.3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed then deltas 1.3.2.1 and 2.1 can be removed, etc.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed or be the owner of the SCCS file and its directory.

The −r keyletter, which is mandatory, is used to specify the complete SID of the delta to be removed (i.e., it must have two components for a trunk delta and four components for a branch delta). Thus:

```
rmdel −r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, **rmdel** checks that the release number (R) of the given SID satisfies the relation:

```
floor <= R <= ceiling
```

The **rmdel** command also checks that the SID specified is not that of a version for which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's "user list", or the "user list" must be empty. Also, the release specified can not be locked against editing. That is, if the **l** flag is set [see **admin**(1) in the UNIX System User's Manual], the release specified *must* not be contained in the list]. If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the "delta table" of the SCCS file is changed from "D" (delta) to "R" (removed).

G. The "cdc" Command

The **cdc** command is used to change a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the **rmdel** command, except that the delta to be processed is not required to be a leaf delta. For example:

```
cdc -r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The new commentary is solicited by **cdc** in the same manner as that of **delta**. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for delta 1.4.

H. The "what" Command

The **what** command is used to find identifying information within any UNIX system file whose name is given as an argument to **what**. Directory names and a name of "-" (a lone minus sign) are not treated specially, as they are by other SCCS commands, and no keyletters are accepted by the command.

The **what** command searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the %Z% ID keyword [see **get**(1)], and prints (on the standard output) what follows that string until the first double quote ("), greater than (>), backslash (\), new line, or (nonprinting) NUL character. For example, if the SCCS file *s.prog.c* (a C language program) contains the following line:

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

and then the command

```
get -r3.4 s.prog.c
```

is executed, the resulting *g-file* is compiled to produce "prog.o" and "a.out". Then the command

```
what prog.c prog.o a.out
```

produces

```
prog.c:
  prog.c:3.4
prog.o:
  prog.c:3.4
a.out:
  prog.c:3.4
```

The string searched for by **what** need not be inserted via an ID keyword of **get**; it may be inserted in any convenient manner.

I. The "sccsdiff" Command

The **sccsdiff** command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the **−r** keyletter, whose format is the same as for the **get** command. The two versions must be specified as the first two arguments to this command in the order they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the **pr**(1) command (which actually prints the differences) and must appear before any file names. The SCCS files to be processed are named last. Directory names and a name of "−" (a lone minus sign) are not acceptable to **sccsdiff**.

The differences are printed in the form generated by **diff**(1). The following is an example of the invocation of **sccsdiff**:

```
sccsdiff −r3.4 −r5.6 s.abc
```

J. The "comb" Command

The **comb** command generates a "shell procedure" [see **sh**(1) in the UNIX System User's Manual] which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated shell procedure is written on the standard output. Named SCCS files are reconstructed by discarding unwanted deltas and combining other specified deltas. The SCCS files that contain deltas no longer useful should be discarded. It is not recommended that **comb** be used as a matter of routine; its use should be restricted to a very small number of times in the life of an SCCS file.

In the absence of any keyletters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Fig. 4.3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The **−p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The **−c** keyletter specifies a list [see **get**(1) in the UNIX System User's Manual for the syntax of such a list] of deltas to be preserved. All other deltas are discarded.

The **−s** keyletter causes the generation of a shell procedure, which when run, produces only a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that **comb** be run with this keyletter (in addition to any others desired) before any actual reconstructions.

It should be noted that the shell procedure generated by **comb** is not guaranteed to save space. In fact, it is possible for the reconstructed file to be larger than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

K. The "val" Command

The **val** command is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

The **val** command checks for the existence of a particular delta when the SID for that delta is explicitly specified via the **−r** keyletter. The string following the **−y** or **−m** keyletter is used to check the value set by the **t** or **m** flag, respectively [see **admin**(1) in the UNIX System User's Manual for a description of the flags].

The **val** command treats the special argument "−" differently from other SCCS commands. This argument allows **val** to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end of file. This capability allows for one invocation of **val** with different values for the keyletter and file arguments. For example:

```
val −
−yc −mabc s.abc
−mxyz −ypl1 s.xyz
```

first checks if file *s.abc* has a value "c" for its "type" flag and value "abc" for the "module name" flag. Once processing of the first file is completed, **val** then processes the remaining files, in this case, *s.xyz*, to determine if they meet the characteristics specified by the keyletter arguments associated with them.

The **val** command returns an 8-bit code; each bit set indicates the occurrence of a specific error [see **val**(1) for a description of possible errors and the codes]. In addition, an appropriate diagnostic is printed unless suppressed by the **−s** keyletter. A return code of "0" indicates all named files met the characteristics specified.

SCCS FILES

This part discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

A. Protection

The SCCS relies on the capabilities of the UNIX software for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the "release lock" flag, the "release floor" and "ceiling" flags, and the "user list".

New SCCS files created by the **admin** command are given mode 444 (read-only). It is recommended that this mode *not* be changed as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755 which allows only the owner of the directory to modify its contents.

The SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files. The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.

The SCCS files must have only one link (name) because the commands that modify SCCS files do so by creating a copy of the file (the *x-file*, see "SCCS COMMAND CONVENTIONS") and, upon completion of processing, remove the old file and rename the *x-file*. If the old file has more than one link, this would break such additional links. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with "s.".

When only one user uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (e.g., in large software development projects), one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the **admin** command). This user is termed the "SCCS administrator" for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and if desired, **rmdel** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the "set user ID on execution" bit "on" [see **chmod**(1) in the UNIX System User's Manual], so that the effective user ID is the user ID of the administrator. This program invokes the desired SCCS command and causes it to inherit the privileges of the interface program for the duration of that command's execution. Thus, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the "user list" for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. These other users are thus able to modify the SCCS files only through the use of **delta** and, possibly, **rmdel** and **cdc**. The project-dependent interface program, as its name implies, must be custom-built for each project.

B. Formatting

The SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	A line containing the "logical" sum of all the characters of the file (*not* including this checksum itself).
Delta Table	Information about each delta, such as type, SID, date and time of creation, and commentary.
User Names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	Indicators that control certain actions of various SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in **sccsfile**(5). The checksum is the only portion of the file which is of interest below.

It is important to note that because SCCS files are ASCII files they may be processed by various UNIX software commands, such as **ed**(1), **grep**(1), and **cat**(1). This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly) or when it is desired to simply look at the file.

> ***Caution: Extreme care should be exercised when modifying SCCS files with non-SCCS commands.***

C. Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file or portions of it (i.e., one or more "blocks") can be destroyed. The SCCS commands (like most UNIX software commands)

issue an error message when a file does not exist. In addition, SCCS commands use the checksum stored in the SCCS file to determine whether a file has been corrupted since it was last accessed [possibly by having lost one or more blocks or by having been modified with **ed**(1)]. No SCCS command will process a corrupted SCCS file except the **admin** command with the **−h** or **−z** keyletters, as described below.

It is recommended that SCCS files be audited for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the **admin** command with the **−h** keyletter on all SCCS files:

```
admin −h s.file1 s.file2 ...
          or
admin −h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

```
corrupted file (co6)
```

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not detect missing files. A simple way to detect whether any files are missing from a directory is to periodically execute the **ls**(1) command on that directory and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX system operations group and request the file be restored from a backup copy. In the case of minor damage, repair through use of the editor **ed**(1) may be possible. In the latter case after such repair, the following command must be executed:

```
admin −z s.file
```

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption that existed in the file will no longer be detectable.

AN SCCS INTERFACE PROGRAM

A. General

In order to permit UNIX system users with different user identification numbers (user IDs) to use SCCS commands upon the same files, an SCCS interface program is provided to temporarily grant the necessary file access permissions to these users. This part discusses the creation and use of such an interface program. The SCCS interface program may also be used as a preprocessor to SCCS commands since it can perform operations upon its arguments.

B. Function

When only one user uses SCCS, the real and effective user IDs are the same; and that user's ID owns the directories containing SCCS files. However, there are situations (e.g., in large software development projects) in which it is practical to allow more than one user to make changes to the same set of SCCS files. In these cases, one user must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the **admin** command). This user is termed the "SCCS administrator" for that project. Since other users of SCCS do not have the same privileges and permissions as the SCCS administrator, the other users are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and if desired, **rmdel**, **cdc**, and **unget** commands. Other SCCS commands either do not require write permission in the directory containing SCCS files or are (generally) reserved for use only by the administrator.

The interface program must be owned by the SCCS administrator, must be executable by nonowners, and must have the "set user ID on execution" bit "on" [see **chmod**(1) in the UNIX System User's Manual] so that, when executed, the effective user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to inherit the privileges of the SCCS administrator for the duration of that command's execution. In this manner, the owner of an SCCS file (the administrator) can modify it at will. Other users whose login names are in the user list for that file (but who are not its owners) are given the necessary permissions only for the duration of the execution of the interface program. They are thus able to modify the SCCS files only through the use of **delta** and, possibly, **rmdel** and **cdc**.

C. A Basic Program

When a UNIX program is executed, the program is passed as argument 0, which is the name that invoked the program, and followed by any additional user-supplied arguments. Thus, if a program is given a number of links (names), the program may alter its processing depending upon which link invokes the program. This mechanism is used by an SCCS interface program to determine which SCCS command it should subsequently invoke [see **exec**(2) in the UNIX System User's Manual].

A generic interface program (*inter.c*, written in C language) is shown in Table 4.B. Note the reference to the (unsupplied) function "filearg". This is intended to demonstrate that the interface program may also be used as a preprocessor to SCCS commands. For example, function "filearg" could be used to modify file arguments to be passed to the SCCS command by supplying the full pathname of a file, thus avoiding extraneous typing by the user. Also, the program could supply any additional (default) keyletter arguments desired.

D. Linking and Use

In general, the following demonstrates the steps to be performed by the SCCS administrator to create the SCCS interface program. It is assumed, for the purposes of the discussion, that the interface program **inter.c** resides in directory "/x1/xyz/sccs". Thus, the command sequence

```
cd /x1/xyz/sccs
cc ... inter.c -o inter ...
```

compiles **inter.c** to produce the executable module **inter** (the "..." represent other arguments that may be required). The proper mode and the "set user ID on execution" bit are set by executing:

```
chmod 4755 inter
```

For example, new links are created by:

```
ln inter get
ln inter delta
ln inter rmdel
```

The names of the links may be arbitrary provided the interface program is able to determine from them the names of SCCS commands to be invoked. Subsequently, any user whose shell parameter PATH [see **sh**(1) in the UNIX System User's Manual] specifies directory "/x1/xyz/sccs" as the one to be searched first for executable commands may execute, e.g.:

```
get -e /x1/xyz/sccs/s.abc
```

from any directory to invoke the interface program (via its link "get"). The interface program then executes "/usr/bin/get" (the actual SCCS **get** command) upon the named file. As previously mentioned, the interface program could be used to supply the pathname "/x1/xyz/sccs" so that the user would only have to specify

```
get -e s.abc
```

to achieve the same results.

TABLE 4.A

DETERMINATION OF NEW SID

CASE	SID SPECIFIED*	-b KEYLETTER USED†	OTHER CONDITIONS	SID RETRIEVED	SID OF DELTA TO BE CREATED
1	none‡	no	R defaults to mR	mR.mL	mR.(mL + 1)
2	none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB + 1).1
3	R	no	R > mR	mR.mL	R.1§
4	R	no	R = mR	mR.mL	mR.(mL + 1)
5	R	yes	R > mR	mR.mL	mR.mL.(mB + 1).1
6	R	yes	R = mR	mR.mL	mR.mL.(mB + 1).1
7	R	—	R < mR and R does not exist	hR.mL**	hR.mL.(mB + 1).1
8	R	—	Trunk successor in release > R and R exists	R.mL	R.mL.(mB + 1).1
9	R.L	no	No trunk successor	R.L	R.(L + 1)
10	R.L	yes	No trunk successor	R.L	R.L.(mB + 1).1
11	R.L	—	Trunk successor in release ≥ R	R.L	R.L.(mB + 1).1
12	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS + 1)
13	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB + 1).1
14	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S + 1)
15	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB + 1).1
16	R.L.B.S	—	Branch successor	R.L.B.S	R.L.(mB + 1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB + 1).1" means "the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components must exist.

† The -b keyletter is effective only if the **b** flag [see admin(1)] is present in the file. In this table, an entry of "—" means "irrelevant".

‡ This case applies if the **d** (default SID) flag is not present in the file. If the **d** flag is present in the file, the SID obtained from the **d** flag is interrupted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the first delta in a new release.

** "hR" is the highest existing release that is lower than the specified, nonexistent, release R.

TABLE 4.B

SCCS INTERFACE PROGRAM "inter.c"

```
main(argc, argv)
int argc;
char *argv[];
{

        register int i;
        char cmdstr[LENGTH]

        /*
        Process file arguments (those that don't begin with "-").
        */
        for (i = 1; i < argc; i++)
                if (argv[i][0] != '-')
                        argv[i] = filearg(argv[i]);

        /*
        Get "simple name" of name used to invoke this program
        (i.e., strip off directory-name prefix, if any).
        */
        argv[0] = sname(argv[0]);

        /*
        Invoke actual SCCS command, passing arguments.
        */
        sprintf(cmdstr, "/usr/bin/%s", argv[0]);
        execv(cmdstr, argv);
}
```

NOTES

5. THE M4 MACRO PROCESSOR

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, in which case the arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

A list of 21 built-in macros provided by the M4 macro processor can be found in Table 5.A. 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.

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
```

DEFINING MACROS

The primary built-in function of M4 is **define**, which 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 can not be reused.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has predefined the names *pdp11* and *u3b* on the corresponding systems. 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, otherwise the third. 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('unix', on UNIX, not on UNIX)
```

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** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump** defined as

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

generates code to increment its argument by 1. 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 new lines that occur during argument collection are discarded. All other white space is retained. Thus:

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

defines 'a' to be 'b c'.

Arguments are separated by commas, but parentheses are counted properly so a comma protected by parentheses does not terminate an argument. 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.

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 UNIX 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.

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 can not 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.

SYSTEM COMMAND

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

```
syscmd(date)
```

on the UNIX 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.

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.

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 doesn't 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 which 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** which 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 new lines will disappear. Another method of achieving the same results is to input

```
divert(-1)
define(...)
  ...
divert.
```

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 which 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.

TABLE 5.A

BUILT-IN MACROS

MACRO NAME	FUNCTION
changequote	Restores original characters or makes new quote characters the left and right brackets
changecom	Changes left and right comment markers from the default # and new line
decr	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 one-of-ten diversions
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	Performs 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 named can not be accessed.
incr	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 up 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 definition

TABLE 5.A (Contd)

BUILT-IN MACROS

MACRO NAME	FUNCTION
pushdef	Defines new macros, but saves any previous definition
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 UNIX 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 defintions
undivert	Discards the diverted text

NOTES

6. THE "awk" PROGRAMMING LANGUAGE

GENERAL

The **awk** programming language is designed to scan a set of files for lines that match any of a set of patterns which the user has specified. For each pattern, an action can be specified. The specified action will be performed on each line or fields of lines that match the pattern. The **awk** language is designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

Readers familiar with the program **grep** will recognize the approach, although in **awk** the patterns may be more general than in **grep**, and the actions allowed are more involved than printing the matching line. For example, the **awk** program

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

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev    { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

A. Usage

The command

```
awk program [files]
```

scans each input file, or on the standard input if there are no files, for lines that match any of a set of patterns specified in *program*. With each pattern in *program*, there may be an associated action that will be performed when a line of the input matches the pattern. The **awk** commands may appear literally in *program*, or the statements can also be placed in a file *pfile* and executed by the command:

```
awk -f pfile [files]
```

B. Program Structure

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

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

Each line of input is matched against each of the *patterns* in succession. For each pattern that matches, the associated *action* is executed. When all the patterns have been tested, the next line of input is fetched and the matching process starts over.

Either the pattern or the action may be omitted, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (A line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which does not match a pattern is ignored.

Since patterns or actions may be omitted, actions must be enclosed in braces to distinguish them from patterns in the program.

The awk patterns may include arbitrary boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, if-else, while, for statements, and multiple output streams.

C. Records and Fields

The variable *FILENAME* contains the name of the current input file. The input to awk is divided into "records" terminated by a record separator. The default record separator is a newline. However, the input record separator may be changed; so by default, awk processes its input a line at a time. The number of the current record is available in a variable named *NR*.

Each input record is considered to be divided into "fields". The default field separator is white space (blanks or tabs); however, the input field separator may be changed. Fields are referred to as $1, $2, etc. where $1 is the first field, and $0 is the entire input record. Fields may be assigned to a numeric or string value. The number of fields in the current record is available in the variable *NF*.

The variables *FS* and *RS* refer to the input field and record separators, respectively; they may be changed at any time to any single character. The optional command-line argument **−F***c* may also be used to set *FS* to the character *c*.

If the record separator is empty, an empty input line is taken as the record separator; and blanks, tabs, and newlines are treated as field separators.

D. Printing

When there is no pattern for an action, the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the awk command print. The awk program

```
{ print }
```

prints each record copying the input to the output. A more useful program is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator, referenced by the variable *OFS*, when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

prints the first and second fields together.

The predefined variables *NF* and *NR* can be used in the print command; for example:

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

prints each record preceded by the record number and the number of fields in the record.

Output may be diverted to multiple files; the program

```
{ print $1 >" fool" ; print $2>"foo2"}
```

writes the first field, $1, on file *fool*, and the second field on file *foo2*. The >> notation can also be used:

```
print $1 >> " foo "
```

appends the first field, $1, to file *foo*. (In each case, the output files are created if necessary.) The file name can be a variable, a field, or a constant; for example:

```
print $1 >$2
```

uses the contents of field 2 as a file name.

There is a limit on the number of output files; currently the limit is 10.

Similarly, output can be piped into another process (on the UNIX operating system only); for instance:

```
print | " mail bwk "
```

mails the output to **bwk**.

The variables *OFS* and *ORS* may be used to change the current output field separator and output record separator. The default output field separator is a blank, and the default output record separator is a newline. The output record separator is appended to the output of the **print** statement.

The **awk** language also provides the **printf** statement for output formatting:

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

formats the expressions in the list according to the specification in **format** and prints them. The statement

```
printf " %8.2f %10ld\n " , $1, $2
```

prints the first field, $1, as a floating point number eight digits wide with two after the decimal point and prints the second field, $2, as a 10-digit long decimal number; followed by a new line. No output separators are produced automatically. In this example, the two fields will be separated by the current output field separator. The version of **printf** used in the **awk** programming language is identical to that used in the C programming language.

PATTERNS

The "pattern" which precedes an "action" in an **awk** program acts as a selector to determine whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

A. "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 record has been processed. **BEGIN** and **END** provides a way to gain control of the program before and after processing.

As an example, using the variable *FS*, the field separator can be set to a colon by

```
BEGIN    { FS = ":" }
  ... rest of program ...
```

or, using the variable *NR*, the input lines may be counted by

```
END    { print NR }
```

If **BEGIN** is present, it must be the first pattern; if **END** is present, it must be the last pattern in the program.

B. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, for example:

```
/smith/
```

This is a complete **awk** program that prints all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

```
blacksmithing
```

The **awk** regular expressions include the regular expression forms found in the UNIX text editor **ed** and **grep** (without back referencing). In addition, **awk** allows parentheses for grouping, | for alternatives, + for "one or more", and ? for "zero or one", all as in the **lex** programming language. Character classes may be abbreviated as **[a-zA-Z0-9]** to represent the set of all letters and digits. As an example, the **awk** program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

prints all lines containing any of the names "Aho", "Weinberger", or "Kernighan", whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in the programs **ed** and **sed**. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn off the special meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern:

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

which matches any string of characters enclosed in slashes.

Any field or variable can be specified to match (or not match) a regular expression with the operators ~ and !~. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches "john" or "John". Notice that the program will also match "Johnson", "St. Johnsbury", etc. To restrict the program to match lines or fields that contain only "[jJ]ohn", use

```
$1 ~ /^[jJ]ohn$/
```

The caret "^" refers to the beginning of a line or field; the dollar sign $ refers to the end of a line or field.

C. Relational Expressions

An **awk** pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines that contain an even number of fields.

In relational tests if neither operand is numeric, a string comparison is made; otherwise, the operand is numeric. Thus:

```
$1 >= "s"
```

selects lines that begin with an "s", "t", "u", etc. In the absence of any other information, fields are treated as strings; therefore, the program

```
$1 > $2
```

will perform a string comparison.

D. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example:

```
$1 >= "s" && $1< "t" && $1 != "smith"
```

selects lines where the first field begins with "s" but is not "smith". The && and || guarantee that the operands will be evaluated from left to right; evaluation stops when the truth or falsehood is determined.

E. Pattern Ranges

A "pattern" may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second. For instance,

```
pat1, pat2 { ... }
```

will perform the action for each line between an occurrence of "pat1" and the next occurrence of "pat2" (inclusive). An example is:

```
/start/, /stop/
```

which prints all lines between the patterns "start" and "stop", while:

```
NR == 100, NR == 200 { ... }
```

performs the action for lines 100 through 200 of the input.

ACTIONS

An **awk** action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used for a variety of bookkeeping and string manipulating tasks.

A. Built-in Functions

The **awk** language provides a "length" function to compute the length of a string of characters. The following program prints each record preceded by its length:

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

The function **length** is a "pseudo-variable" which yields the length of the current record; **length(argument)** is a function which yields the length of its argument. The argument may be any expression. The following is equivalent to the previous program:

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

Also provided by **awk** are the arithmetic functions **sqrt**, **log**, **exp**, and **int** for square root, base **e** logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the entire record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **fBbstr(s, m, n)** produces the substring of "s" that begins at position "m" (origin 1) and is at most "n" characters long. If "n" is omitted, the substring goes to the end of "s". The function **index(s1, s2)** returns the position where the string "s2" occurs in "s1" or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions "e1", "e2", etc., in the **printf** format specified by "f". For example

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets *x* to the string produced by formatting the values of the first field, $1, and the second field, $2.

B. Variables, Expressions, and Assignments

The **awk** variables take on numeric (floating point) or string values according to context. For example, in:

```
x = 1
```

x is clearly a number, while in:

```
x = "smith"
```

x is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance:

```
x = "3" + "4"
```

assigns 7 to *x*. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero. To force an expression to be treated as a number, add 0 (zero) to it; to force an expression to be treated as a string, concatenate the null string ("") to it.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most **BEGIN** sections. For example, the sums of the first two fields can be computed by:

```
                { s1 += $1; s2 += $2 }
END             { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, and % (mod). The C language increment ++ and decrement −− operators are available; also the assignment operators +=, −=, *=, /=, and %= are available. These operators may all be used in expressions.

C. Field Variables

Fields in **awk** share essentially all of the properties of variables—they may be used in arithmetic or string operations and may be assigned to a numeric or string value. One can replace the first field with a sequence number with:

```
{ $1 = NR; print }
```

or accumulate two fields into a third:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
      $3 = "too big"
  print
}
```

which replaces the third field with "too big" when the third field is greater than 1000 and prints the record.

Field references may be numerical expressions, as in:

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields;

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

splits the string "s" into "array[1], ..., array[n]". The number of elements found is returned. If the "sep" argument is provided, it is used as the field separator; otherwise, the field separator is that which is referenced by the variable *FS*.

D. String Concatenation

Strings may be concatenated. For example:

```
length($1 $2 $3)
```

returns the length of the first three fields; or in a **print** statement,

```
print $1 " is " $2
```

prints the two fields separated by "is". Variables and numeric expressions may also appear in concatenations.

E. Arrays

Array elements are not declared; the elements are initialized when 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 record to the *NR*-th element of the array *x*. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the **awk** program

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

The first action records each input line in the array *x*.

Array elements may be named by non-numeric values, which gives **awk** a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like "apple", "orange", etc. Then the program

```
/apple/   { x[ " apple" ]++ }
/orange/  { x[ " orange" ]++ }
END       { print x[ " apple" ], x[ " orange " ] }
```

increments counts for the named array elements and prints them at the end of the input.

Any expression can be used as a subscript in an array reference. Thus:

```
x[$1] = $2
```

uses the first field of a record (as a string) to index the array *x*.

Suppose each line of input contains two fields — a name and a nonzero value. Names may be repeated; the task is to print a list of each unique name followed by the sum of all the values for that name. This can be done with the program

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

To sort the output, replace the last line by

```
print name, amount[name] | " sort "
```

F. Flow-of-Control Statements

The **awk** language also provides the basic flow-of-control statements **if-else**, **while**, **for**, and statement grouping with braces, as in the C programming language. The **if-else** statement is exactly like that of the C language and was previously shown in the subpart "Field Variables". The condition in parentheses of an **if-else** statement is evaluated; if it is true, the statement following the **if** is performed. The **else** part is optional.

The **while** statement is exactly like that of the C language. For example, to print all input fields one per line:

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

The **for** statement is also like that of the C language;

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

performs the same task as the **while** statement above.

There is an alternate form of the **for** statement which is suited for accessing the elements of an associative array:

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

does statement with *i* set in turn to each element of **array**. The elements are accessed in an apparently random order. Confusion will develop if *i* is altered or if any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while**, or **for** can include relational operators like <, <=, >, >=, == (equal to), and != (not equal to); regular expression matches with the match operators ~ and !~; the logical operators ||, &&, and !; and parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for** loop; the **continue** statement causes the next iteration of the enclosing loop to begin.

The statement **next** causes the **awk** program to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in **awk** programs, but they must begin with the character # and terminate with the end of the line, for example:

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

NOTES

7. ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (BC)

GENERAL

The arbitrary precision desk calculator language (BC) is a language and compiler for doing arbitrary precision arithmetic under the UNIX operating system. The output of the compiler is interpreted and executed by a collection of routines which 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) which 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

- to do computation with large intergers
- to do computation accurate to many decimal places
- conversion of 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. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of the UNIX 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.

SIMPLE COMPUTATIONS WITH INTERGERS

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 exponentiation, respectively. Division of integers produces an 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 Fortran, with ^ having the greatest binding 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. The two expressions

```
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.
```

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
```

will produce 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 will have 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
```

will change 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
```

will produce 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.

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.

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, will be 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: b().

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

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

would cause the result 21.98 to be printed, and the line

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

would cause the result 60 to be printed.

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.

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 will print the integers from one to ten. 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)
```

will print "a" factorial if "a" is a positive integer. The following is the definition of a function which will compute 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
        }
}
```

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.

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 Appendix 7.1 or 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
```

***Warning:** 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 which may be obtained by typing at command level:

  ```
  bc -l
  ```

  This command will load a set of library functions which 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 will read and execute the named file or files before accepting commands from the keyboard. In this way, programs and function definitions may be loaded.

APPENDIX 7.1

NOTATION

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

TOKENS

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

A. Comments

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

B. Identifiers

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.

C. Keywords

The following are reserved keywords:

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

D. Constants

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.

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.

A. Primative Expressions

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.

identifiers

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

array-name[expression]

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

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.

Function Calls

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.

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.

length(expression)

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

scale(expression)

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

Constants

Constants are primitive expressions.

Parentheses

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

B. Unary Operators

The unary operators bind right to left.

-expression

The result is the negative of the expression.

++named-expression

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

--named-expression

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

named-expression++

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

named-expression--

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

C. Exponentiation Operator

The exponentiation operator binds right to left.

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(axb,max(scale,a))

D. Multiplicative Operators

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

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))

expression / expression

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

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.

E. Additive Operators

The additive operators bind left to right.

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.

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.

F. Assignment Operators

The assignment operators bind right to left.

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.

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

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.

STATEMENTS

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

A. Expression Statements

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

B. Compound Statements

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

C. Quoted String Statements

The following statement prints the string inside the quotes.

" any string "

D. The "if" Statement

if(*relation*)*statement*

The substatement is executed if the relation is true.

E. The "while" Statement

while(*relation*)*statement*

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

F. The "for" 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.

G. The "break" Statement

break

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

H. The "auto" 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 by empty square brackets. The **auto** statement must be the first statement in a function definition.

I. The "define" Statement

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.

J. The "return" Statement

return
return(*expression*)

The **return** statement causes termination of a function, popping of its auto variables on the stack, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

K. The "quit" Statement

The **quit** statement stops execution of a BC program and returns control to the UNIX 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.

8. INTERACTIVE DESK CALCULATOR (DC)

GENERAL

The DC program is an interactive desk calculator program implemented on the UNIX 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 UNIX 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 from the standard input.

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

The value of the number is pushed onto the main 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. Numbers may contain decimal points.

+ − * / % ^

The top two values on the stack are added (+), subtracted (−), multiplied (*), divided (/), remaindered (%), or exponentiated (^). 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 into 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 new line, is a valid register name.

l*x*

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

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 UNIX 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.

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.

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 can not 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.

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 will include the exact effect of **scale** on the computations.

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 which are not in the range 0 through 99 must be brought into that range, propagating any carries or borrows that result.

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.

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 be one unit too low, the next trial quotient will be larger than 99; and this will be 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.

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.

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

$$X_{n+1} = \tfrac{1}{2}(X_n + \frac{Y}{X_n})$$

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

EXPONENTIATION

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

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 will push the value of the input base on the stack.

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 can be output by typing the command **f**. The **o** command can be 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 will work correctly for any base. The command O pushes the value of the output base on the stack.

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 can be used for decimal-octal or decimal-hexadecimal conversions.

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.

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.

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.

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
```

PUSHDOWN REGISTERS AND ARRAYS

These commands were 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.

MISCELLANEOUS COMMANDS

The command **!** interprets the rest of the line as a UNIX software command and passes it to the UNIX 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.

DESIGN CHOICES

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) 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 was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was 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 was 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** were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause 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 and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were 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 and 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.

NOTES

9. LEXICAL ANALYZER GENERATOR (LEX)

GENERAL

The **Lex** program generator is designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by 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 which reads an input stream, copies the input stream to an output stream, and partitions the input into strings which 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 the user 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 code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired.

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 UNIX operating system, but the code generated by **Lex** may be taken anywhere the appropriate compilers exist.

The **Lex** program generator turns the user's expressions and actions (called **source** in this section) into the host general purpose language; the generated program is named **yylex**. The **yylex** program will recognize expressions in a stream (called **input** in this section) and perform the specified actions for each expression as it is detected. See Fig. 9.1.

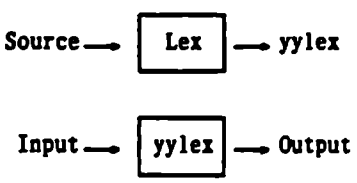

Fig. 9.1 — An Overview of Lex

For an example, consider a program to delete from the input all blanks or tabs at the ends of lines:

```
%%
[\t]+$  ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written for visibility, in accordance with the C language convention) which 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()* will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

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

The coded instructions generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a new-line character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule 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 require a lower level analyzer to recognize input tokens. Thus, a combination of **Lex** and **yacc** is often appropriate. When used as a preprocessor for a later parser generator, **Lex** is used to partition the input stream; and the parser generator assigns structure to the resulting pieces. The flow of control in such a case is shown in Fig. 9.2. Additional programs, written by other generators or by hand, can be added easily to programs written by **Lex**. The **yacc** compiler users will realize that the name **yylex** is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **Lex** simplifies interfacing.

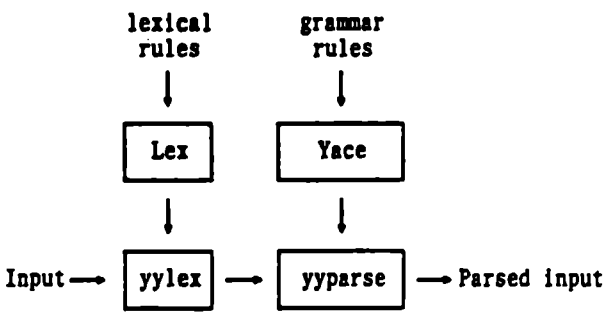

Fig. 9.2 — Lex With Yacc

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 source which 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** will recognize "ab" and leave the input pointer just before "cd ...". Such backup is more costly than the processing of simpler languages.

LEX SOURCE

The general format of **Lex** source is

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

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

```
%%
```

(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 the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear:

```
integer        printf( " found keyword INT " );
```

to look for the string **integer** in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C, and the C 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 it is desired 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".

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".

A. 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 new line, \t is tab, and \b is backspace. To enter \ itself, use \\. Since new line is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character except blank, tab, new line, and the list of operator characters above is always a text character.

B. 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 will get 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 which is not a letter. The \ character provides the usual escapes within character class brackets.

C. Arbitrary Character

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

.

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

[\40-\176]

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

D. Optional Expressions

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

ab?c

matches either "ac" or "abc".

E. 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.

F. 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".

G. Context Sensitivity

The **Lex** program will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a new-line character or at the beginning of the input stream). This can never conflict with the other meaning of ^ (complementation of character classes) since that only applies within the [ ] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by new line). 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.

H. 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 one to five occurrences of "a".

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

LEX ACTIONS

When an expression written as above is matched, **Lex** executes the corresponding action. This part describes some features of **Lex** which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus, the **Lex** user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When **Lex** is being used with **yacc**, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C language null statement, ; as an action causes this result. A frequent rule is

```
[ \t\n]   ;
```

which causes the three spacing characters (blank, tab, and new line) 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, the user will 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);
```

will print 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[]*. So 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 which is not desired. For example, if there is a rule which matches **read**, it will normally match the instances of **read** contained in **bread** or **readjust**; to avoid this, a rule of the form "[a-z]+" is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence **Lex** also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might 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 which 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
             }
```

which 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 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 = ...
             }
```

will perform 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 would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3", however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, **Lex** also permits access to the I/O routines it uses. They are:

1. *input()* 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, a value of zero returned by *input* must mean end of file, and the relationship between *unput* and *input* must be retained or the **Lex** 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. See below for a discussion of the character set used by **Lex**. The standard **Lex** library imposes a 100-character limit on backup.

Another **Lex** library routine that the user will 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 which recognizes end of file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied, a file containing nulls cannot be handled since a value of 0 returned by *input* is taken to be end of file.

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 which 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") will 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 seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input:

'first' quoted string here, 'second' here

the above expression 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, will stop after ('first'). The consequences of errors like this are mitigated by the fact that the dot (.) operator will not match new line. Thus expressions like .* stop on the current line. Do not try to defeat this with expressions like [.\n]+ or equivalents; the **Lex** generated program will try to read the entir 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 new line. Since "she" includes "he", **Lex** will normally not recognize the instances of "he" included in "she" since once it has passed a "she" those characters are gone.

Sometimes the user would like to override this choice. The action *REJECT* means "go do the next alternative". It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of "he":

```
she             {s++; REJECT;}
he              {h++; REJECT;}
\n              |
.               ;
```

these rules are one way of changing the previous example to accomplish the task. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that "she" includes "he" but not vice versa and omit the *REJECT* action on "he". In other cases, it would not be possible to state which input characters were in both classes.

Consider the two rules

```
a[bc]+          { ... ; REJECT;}
a[cd]+          { ... ; REJECT;}
```

If the input is "ab", only the first rule matches, and on "ad" only the second matches. The input string "accb" matches the first rule for four characters and then the second rule for three characters. In contrast, the input "accd" agrees with the second rule for four characters and then the first rule for three.

In general, *REJECT* is useful whenever the purpose of **Lex** is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other.

Suppose a digram table of the input is desired; normally the digrams overlap, that is the word "the" is considered to contain both "th" and "he". Assuming a 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.

LEX SOURCE DEFINITIONS

Recalling the format of the **Lex** source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options though 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 will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by **Lex** which contains the actions. This material must look like program fragments and should precede the first **Lex** rule.

   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.

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 UNIX 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 can be avoided.

LEX AND YACC

To use **Lex** with **yacc**, note that what **Lex** writes is a program named *yylex()*, the name required by **yacc** for its analyzer. Normally, the default main program on the **Lex** library calls this routine; but if **yacc** is loaded and its main program is used, **yacc** will call *yylex()*. In this case, each **Lex** rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to **yacc's** names for tokens is to compile the **Lex** output file as part of the **yacc** output file by placing the line

```
#include " lex.yy.c "
```

in the last section of **yacc** input. Supposing the grammar to be named "good" and the lexical rules to be named "better", the UNIX software command sequence can just be

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The **yacc** library (**-ly**) should be loaded before the **Lex** library to obtain a main program which invokes the **yacc** parser. The generations of **Lex** and **yacc** programs can be done in either order.

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 will alter 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 which 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. To provide a *yywrap* that never returns true causes an infinite loop.

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 will only be recognized when **Lex** is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word "magic" to "first" on every line which began with the letter "a", changing "magic" to "second" on every line which began with the letter "b", and changing "magic" to "third" on every line which began with the letter "c". All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
    int flag;
%%
^a    {flag = 'a'; ECHO;}
^b    {flag = 'b'; ECHO;}
^c    {flag = 'c'; ECHO;}
\n    {flag = 0 ; ECHO;}
magic    {
    switch (flag)
    }
    case 'a': printf( " first " ); break;
    case 'b': printf( " second " ); break;
    case 'c': printf( " third " ); break;
    default: ECHO; break;
    }
 }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to **Lex** in the definitions section with a line reading:

```
%Start    name1 name2 ...
```

where the conditions may be named in any order. The word "Start" may be abbreviated to "s" or "S". The conditions may be referenced at the head of a rule with <> brackets;

```
<name1>expression
```

is a rule which is only recognized when **Lex** is in the start condition **name1.** To enter a start condition, execute the action statement

```
BEGIN 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:

```
%START AA BB CC
%%
^a                          {ECHO; BEGIN AA;}
^b                          {ECHO; BEGIN BB;}
^c                          {ECHO; BEGIN CC;}
\n                          {ECHO; BEGIN 0;}
<AA>magic                   printf( " first " );
<BB>magic                   printf( " second " );
<CC>magic                   printf( " third " );
```

where the logic is exactly the same as in the previous method of handling the problem, but **Lex** does the work rather than the user's code.

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.

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 new line.
^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

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.

NOTES

10. YET ANOTHER COMPILER—COMPILER (yacc)

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 *grammer 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 would only need to recognize individual letters, and "month name" would be a nonterminal symbol. Such low-level rules tend to waste time and space and may complicate the specification beyond the ability of yacc to deal with it. Usually, the lexical analyzer would recognize the month names and return an indication that a "month name" was seen. In this case, "month name" would be a "token".

Literal characters such as 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 appendicies. Appendix 10.1 is a brief example, and Appendix 10.2 is a summary of the **yacc** input syntax. Appendix 10.3 gives an example using some of the more advanced features of **yacc**,

and Appendix 10.4 describes mechanisms and syntax no longer actively supported but provided for historical continuity with older versions of yacc.

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. Moreover, if the programs section is omitted, the second %% mark may be omitted also. The smallest legal yacc specification is

```
%%
rules
```

since the other two sections may be omitted.

Blanks, tabs, and new lines are ignored except that they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in /* ... */, as in C 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' new-line
'\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 *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen and accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate. Usually the end marker represents some reasonably obvious I/O status, such as "end of file" or "end of record".

ACTIONS

With each grammar rule, the user may associate actions to be performed 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 will be found in the part "ADVANCED TOPICS".

LEXICAL ANALYSIS

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by **yacc** or the user. In either case, the **# define** mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like

```
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 will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C 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 endmarker 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 negative as a token number 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.

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 whether it needs a look-ahead token to decide 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 but usually it is not. 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 endmarker and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the 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 endmarker is obtained, indicated by $end in the *y.output* file. The action in state 1 when the end marker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG, DING DONG, DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

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, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule and continue reading the input until

```
expr - expr - expr
```

has been 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 two legal things, a shift or a reduction, and has no way of deciding between them. This is called a "shift/reduce conflict". It may also happen that the parser has a choice of two legal reductions. This is called a "reduce/reduce conflict". Note that there are never any shift/shift conflicts.

When there are shift/reduce or reduce/reduce conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing 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 disamiguating 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. Back in state 23 the alternative action, described a dot (.), is to be done if the input symbol is not mentioned explicitly in the above actions. In this case if the input symbol is not "ELSE", the parser reduces to

```
stat : IF '(' cond ')' stat
```

by grammer 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 will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

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.

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
```

would mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Rules such as

```
stat : error ';'
```

are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but will do 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 an error has been fully recovered from. 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 was called, the next token returned by *yylex* would presumably be the first token in a legal statement. The old, illegal token must be discarded, and the error state reset. A rule similar to

```
stat  :  error
      {
        resynch() ;
        yyerrok  ;
        yyclearin ;
      }
      ;
```

could perform this.

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

THE "yacc" ENVIRONMENT

When the user inputs a specification to **yacc**, the output is a file of C language 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 part "LEXICAL ANALYSIS") to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse()* returns the value 1, or the lexical analyzer returns the 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 code

```
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 will want to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the 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.

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.

A. 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 10.1 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.

B. 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 would be a bit bigger; and the items would be seen and reduced from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification 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!

C. 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 which 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.

D. Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**. It is difficult to pass information to the lexical analyzer telling it "this instance of *if* is a keyword and that instance is a variable". The user can make a stab at it using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*, i.e., be forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

ADVANCED TOPICS

This part discusses a number of advanced features of **yacc**.

A. 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.

B. 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;
}
;
...
```

but 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 will save a great deal of trouble especially when a few combinations are to be excluded from an otherwise regular structure.

C. 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 will be strictly type checked. The **yacc** value stack is declared to be a *union* of the various types of values desired. The user declares the union and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, **yacc** will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as **lint** will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union. This must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** can not easily determine the type.

To declare the union, the user includes

```
%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> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype.* Another keyword, **%type,** is used 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 10.3. 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.

APPENDIX 10.1

A SIMPLE EXAMPLE

This example gives the complete **yacc** specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise, it is. As in C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately line by line. Note the way that decimal and octal integers are read in by the grammar rules; this job is probably better done by the lexical analyzer.

```
%{
#  include <stdio.h>
#  include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS     /* supplies precedence for unary minus */

%%     /* beginning of rules section */

list     :     /* empty */
         |     list stat '\n'
         |     list error '\n'
         {
                  yyerrok;
         }
         ;

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
            {
                $$ = $1 / $3;
            }
            |   expr '%' expr
            {
                $$ = $1 % $3;
            }
            |   expr '&' expr
            {
                $$ = $1 & $3;
            }
            |   expr '|' expr
            {
                $$ = $1 | $3;
            }
            |   '-' expr    %prec UMINUS
            {
                $$ = - $2;
            }
            |   LETTER
            {
                $$ = regs[$1];
            }
            |   number
            ;

number      :     DIGIT
            {
```

```
                $$ = $1;  base = ($1==0) ? 8 : 10;
        }
        |       number DIGIT
        {
                $$ = bas * $1 + $2;
        }
        ;

%% /*  start of programs */

yylex( )        /*  lexical analysis routine */
{
        /*  returns LETTER for a lowercase letter, yylval = 0 through 25*/
        /*  returns DIGIT for a 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 );
}
```

APPENDIX 10.2

YACC INPUT SYNTAX

This appendix has a description of the yacc input syntax as a yacc specification. Context dependencies, etc., are not considered. Ironically, the yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, new lines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS but never as part of C_IDENTIFIERs.

```
    /*  grammar for the input to Yacc */

    /*  basic entities */
%token      IDENTIFIER    /* includes identifiers and literals */
%token      C_IDENTIFIER    /* identifier (but not literal) followed by colon */
%token      NUMBER        /* [0-9]+ */

    /*  reserved words: %type => TYPE, %left => LEFT, etc. */

%token      LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token      MARK    /* the %% mark */
%token      LCURL     /* the %{ mark */
%token      RCURL     /* the %} mark */

    /*  ASCII character literals stand for themselves */

%start      spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK
        {
                    In this action, eat up the rest of the file
        }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :   START IDENTIFIER
        |   UNION
        {
                Copy union definition to output
```

```
        }
        |  LCURL
        {
                Copy C code to output file
                RCURL
        }
        |  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 */
        ;

    /* rules section */

rules   :  C_IDENTIFIER rbody prec
        |  rules rule
        ;

rule    :  C_IDENTIFIER rbody prec
        |  '|' rbody prec
        ;

rbody   :  /* empty */
        |  rbody IDENTIFIER
        |  rbody act
        ;

act     :  '{'
              {
                Copy action, translate $$, etc.
              }
             '}'
        ;

prec    :  /* empty */
        |  PREC IDENTIFIER
        |  PREC IDENTIFIER act
        |  prec ';'
        ;
```

APPENDIX 10.3

AN ADVANCED EXAMPLE

This appendix gives an example of a grammar using some of the advanced features. The desk calculator example in Appendix 10.1 is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants; the arithmetic operations +, −, *, /, unary −, and = (assignment); and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals,* written

(x , y)

where *x* is less than or equal to *y*. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix 10.1; assignments return no value and print nothing while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C language. Intervals are represented by a structure consisting of the left and right endpoint values stored as *double's.* This structure is given a type name, INTERVAL, by using *typedef.* The **yacc** value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C 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 (e.g., scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**—18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines

2.5 + (3.5 − 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is 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 conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive but not very general. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C 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 lines

%union
{
    int ival;
    double dval;
    INTERVAL vval;
}

%token <ival> DREG VREG     /* indices into dreg, vreg arrays */

%token <dval> CONST         /* floating point constant */

%type <dval> dexp           /* expression */

%type <vval> vexp           /* interval expression */

    /* precedence information about the operators */

%left    '+' '-'
%left    '*' '/'
%left    UMINUS  /* precedence for unary minus */

%%

lines    :  /* empty */
         |  lines line
         ;
```

```
line    :  dexp '\n'
           {
                   printf( "%15.8f\n", $1 );
           }
           |  vexp '\n'
           {
                   printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi );
           }
           |  DREG '=' dexp '\n'
           {
                   dreg[$1] = $3;
           }
           |  VREG '=' vexp '\n'
           {
                   vreg[$1] = $3;
           }
           |  error '\n'
           {
                   yyerrok;
           }
           ;

dexp    :  CONST
           |  DREG
           {
                   $$ = dreg[$1]
           }
           |  dexp '+' dexp
           {
                   $$ = $1 + $3
           }
           |  dexp '-' dexp
           {
                   $$ = $1 - $3
           }
           |  dexp '*' dexp
           {
                   $$ = $1 * $3
           }
           |  dexp '/' dexp
           {
                   $$ = $1 / $3
           }
           |  '-' dexp     %prec UMINUS
           {
                   $$ = - $2
           }
           |  '(' dexp ')'
           {
                   $$ = $2
           }
           ;
```

```
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;
                   $$.lo = $1.lo - $3.hi
           }
           |  dexp '-' vexp
           {
                   $$.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, $1, $3 )
```

```
        }
        |  '-' vexp     %prec UMINUS
        {
                $$.hi = -$2.lo; $$.lo = -$2.hi
        }
        |  '(' vexp ')'
        {
                $$ = $2
        }
        ;

%%

# define BSZ 50     /* buffer size for floating point numbers */

        /* lexical analysis */

yylex( )
{
        register c;

                                        /* skip over blanks */
        while( (c=getchar( ) ) == ' ' )
                ;
        if( isupper( c ) )
        {
           yylval.ival = c - 'A';
           return( VREG );
        }
        if( islower( c ) )
        {
           yylval.ival = c - 'a';
           return( DREG );
        }

           /* gobble up digits, points, exponents */
        if( isdigit( c ) || c== '.' )
        {
           char buf[BSZ+1], *cp = buf;
           int dot = 0, exp = 0;

           for( ; (cp-buf)<BSZ ; ++cp,c=getchar( ) )
           {

               *cp = c;
               if( isdigit( c ) )
                 continue;
               if( c == '.' )
               {
                 if( dot++ || exp )
                       return( '.' ); /* will cause syntax error */
                 continue;
               }
```

```
                if( c == 'e' )
                {
                    if( exp++ )
                        return( 'e' );  /* will cause syntax error */
                    continue;
                }
                /* end of number */
                break;
            }
            *cp = '\0';
            if( (cp-buf) >= BSZ )
                printf( "constant too long: truncated\n" );
            else
                ungetc( c, stdin );  /* push back last char read */
            yylval.dval = atof( buf );
            return( CONST );
        }
        return( c );
}

INTERVAL
hilo( a, b, c, d )
        double a, b, c, d;
{
        /* returns the smallest interval containing a, b, c, and d */

        /* used by *,/ routines */
        INTERVAL v;

        if( a>b )
        {
            v.hi = a;
            v.lo = b;
        }
        else
        {
            v.hi = b;
            v.lo = a;
        }
        if( c>d )
        {
            if( c>v.hi )
                v.hi = c;
            if( d<v.lo )
                v.lo = d;
        }
        else
        {
            if( d>v.hi )
                v.hi = d;
            if( c<v.lo )
                v.lo = c;
        }
        return( v );
```

```
}

INTERVAL vmul( a, b, v )
            double a, b;
            INTERVAL v;
{
               return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
}
dcheck( v )
            INTERVAL v;
{
            if( v.hi >= 0. && v.lo <= 0. )
            {
                  printf( "divisor interval contains 0.\n" );
                  return( 1 );
            }
            return( 0 );
{

INTERVAL vdiv( a, b, v )
            double a, b;
            INTERVAL v;
{
   return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );

}
```

APPENDIX 10.4

OLD FEATURES SUPPORTED BUT NOT ENCOURAGED

This appendix mentions synonyms and features which are supported for historical continuity but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes.

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multicharacter literals is likely to mislead those unfamiliar 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. Actions 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.

NOTES