A/UX® Programming Languages and Tools, Volume 2

031-0127

Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, California 95014
(408) 996-1010

Simultaneously published in the United States and Canada.

031-0127

031-0127

A/UX Programming Languages and Tools
Volume 2

Contents

Preface

Conventions Used in This Manual

Throughout the A/UX manuals, words that must be typed exactly as shown or that would actually appear on the screen are in `Courier` type. Words that you must replace with actual values appear in *italics* (for example, *user-name* might have an actual value of `joe`). Key names appear in CAPS (for example, RETURN). Special terms are in **bold** type when they are introduced; many of these terms are also defined in the glossary in the *A/UX System Overview*.

Syntax notation

All A/UX manuals use the following conventions to represent command syntax. A typical A/UX command has the form

`command` [*flag-option*] [*argument*]...

where:

`command`	Command name (the name of an executable file).
flag-option	One or more flag options. Historically, flag options have the form −[*opt*...] where *opt* is a letter representing an option. The form of flag options varies from program to program. Note that with respect to flag options, the notation `[-a][-b][-c]` means you can select one or more letters from the list enclosed in brackets. If you select more than one letter you use only one hyphen, for example, `−ab`.
argument	Represents an argument to the command, in this context usually a filename or symbols representing one or more filenames.

`[]`	Surround an optional item.
`...`	Follows an argument that may be repeated any number of times.
`Courier` type	anywhere in the syntax diagram indicates that characters must be typed literally as shown.
italics	for an argument name indicates that a value must be supplied for that argument.

Other conventions used in this manual are:

<CR>	indicates that the RETURN key must be pressed.
^*x*	An abbreviation for CONTROL-*x*, where *x* may be any key.
cmd(*sect*)	A cross-reference to an A/UX reference manual. *cmd* is the name of a command, program, or other facility, and *sect* is the section number where the entry resides. For example, `cat`(1).

Chapter 16
`make` Reference

Contents

Figures

Tables

Chapter 16

make Reference

1. make: a file production tool

The make program automates the production of related sets of files. It simplifies the task of administering libraries, functions, related source and object files, and so on, that must reflect a change when you update one file in the set. Although make is normally used to maintain program code, it can also be used for other batch data processing activities (for example, make is often used to produce technical manuals with troff).

make keeps track of program file dependencies; when you change one part of a program, make recompiles related files with a minimum amount of effort. The required information is maintained by the make program itself (which has built-in "rules" for recompilation), by using certain system information such as the timestamp on the files, and by the description of operations kept in a file called the "description file" or "makefile." Once you have set up a makefile for a large project, make keeps track of your files for you and frees you to concentrate on programming or other tasks.

2. Using make

The simplest use of make is

```
make file1
```

where a file named file1.c resides in the current directory. file1.c can #include other files. This command causes make to find file1.c in the local directory and issue the proper command to compile it into file1.

> *Note:* If file1.c has the same filename prefix (the same filename without the .c suffix) as another file, make may compile that file instead. If, for example, there is a more recent file1.l file, it is compiled instead, and file1.c is

overwritten in the process. If these files are not different incarnations of the same program, losing the `.c` file could be quite dangerous.

As long as only one file is involved and only a standard compilation is required, you do not need to create a makefile to `make` your files.

If, however, your program is spread over multiple files, you do need to create a **makefile,** which is a control file containing the filenames, a description of their interrelations, and actions to be performed on them. When it does not have enough to go on, `make` looks in the current directory for a file named `makefile` (or `Makefile`) that contains the necessary administrative information. In general, you must put an entry in the makefile for any file that has a nonstandard compilation procedure.

2.1 Writing a makefile

To write a makefile, you must determine the following:

- the target filename (see below)
- filenames of related compilation units (files)
- file dependencies (see below)
- related libraries
- the command that will produce the target (including options for the programs to be run)

Targets are filenames, or placeholders for them, that are meant to be compiled.

To the `make` program, the specific meaning of **dependency** is as follows: *file1* depends on *file2* only if *file1* needs to be recompiled whenever *file2* is changed. For example, if file `x.c` contains the line

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

the object file `x.o` depends on `defs.h`. If `defs.h` is changed, the `x.o` file must be remade by compiling `x.c`. Note that the `x.c` (source) file does not depend on `defs.h`, because it does not need to be recreated when `defs.h` changes.

For example, you have a file named `zeke`, which depends on `zeke.o`, and which uses library functions from `libc.a`. To relink `zeke`, you would type

```
cc -lc zeke.o -o zeke
```

The two flag options, `-lc` and `-o`, are both passed to `ld` by `cc`.

`-lc`	causes library `libc.a` to be searched;
`-o`	renames the compiled binary file '`zeke`' (instead of the default '`a.out`').

The following makefile is required:

Figure 16-1. A simple makefile

```
zeke: zeke.o
^I    cc -lc zeke.o -o zeke
```

The first line states the dependency (that `zeke` depends on `zeke.o`). The second line is the command line describing the action that must take place whenever `zeke.o` changes. The command line must begin with a tab (represented by `^I` in the figures).

In a more complicated example, a file named `xavier` depends on three files named `yancy.o`, `quincy.o`, and `wally.o`, all of which depend on `defs.h` and use the library `libc.a`. The command to link `xavier` is

```
cc -o xavier -lc yancy.o quincy.o wally.o
```

The makefile for `xavier` follows:

Figure 16-2. A makefile with multiple objects and **`defs.h`** file

```
xavier: yancy.o quincy.o wally.o
^I    cc -lc yancy.o quincy.o wally.o -o xavier

yancy.o quincy.o wally.o: defs.h
```

When makefiles become more complicated, you can use macros and other features described in the sections that follow.

When you have included the interfile dependencies and command sequences in a makefile, the command

```
make
```

updates the appropriate files, regardless of how many files you have edited since the last `make`. `make` uses the date and time that a file was last modified to find files that are out of date with respect to their targets.

2.2 `make` command syntax

`make` uses the following command syntax:

`make` [*option*...][*macro=def*...][`-f` *filename*][*target*...]

These arguments are interpreted in the following order:

1. First the 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. See ''Macro Definitions'' below for more information.
2. Next, the flag options are examined. See ''Flag Options'' below for details.
3. Finally, the remaining arguments are assumed to be the names of targets to be made, and these are made in left-to-right order. If there are no remaining arguments, the first target in the description file is made.

   Note: `make` finds the first target by scanning the description file for a target that does not represent an internal file transformation rule (see ''Transformation Rules'' below). Since these ''built-in'' rules are of the form

   ```
   .n[.m]:
   ```

   where `n` and `m` are suffixes, any rule begins with a period and contains no slashes (as a full pathname might). Thus, the first target will be the first name in the description file that does not begin with a period or begins with a period but contains a slash.

2.2.1 Flag options

`make` accepts the following flag options:

`-i` Ignore error codes (nonzero exit status) returned by invoked commands. This mode is entered automatically if the fake target name `.IGNORE` appears in the description file. See ''Fake Targets'' below.

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

`-r` *Rule-out* mode. 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 issuing 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` *filename*
Use a different description file. *filename* is the name of a description file. A *filename* of – denotes the standard input. If there are no `-f` arguments, `make` reads the file named `makefile` or `Makefile` in the current directory in the order stated. Failing these, `s.makefile` or `s.Makefile` is sought in the SCCS directory, if such a directory exists. If a description file is present, its contents override the default rules.

`-k` Abandon work on the current entry, but continue work on other branches that do not depend upon that entry. (Entries are described under ''Makefile Entries,'' and branches are discussed in ''The `make` Predecessor Tree.'')

`-e` Cause environment variables to override assignments within makefiles.

`-b` Compatibility mode for old makefiles.

2.2.2 Using `make` on individual files

Individual files mentioned in the makefile can also be used as arguments on the command line, if you want to compile only a single file. For example, with the makefile in Figure 16-2 and the command line

```
make yancy.o
```

`make` remakes only `yancy.o`, including `defs.h` in the process. To `make` both `yancy.o` and `wally.o`, you type

```
make yancy.o wally.o
```

and both files are remade properly.

3. The description file

The description file (often called the makefile) defines the target file and its dependencies.

A description file can contain the following:

- makefile entries, consisting of dependency statements, and commands or command sequences
- comments
- `include` lines
- macro definitions

Note: If you do not supply a description file, `make` uses its default rules to produce the file named on the command line. See "The Default Rules." If you name your description file something other than `makefile` or `Makefile`, you must use the `-f` flag option on the `make` command line. See "Flag Options" for details.

3.1 Makefile entries

A **makefile entry** defines the relationship between a target and its dependent(s) and (usually) stipulates the command as well.

Multiple entries within one description file are permissible and usual.

The general form of a makefile entry is

target1 [*target2* ...] : [:][*dependent1* ...][; *commands*][#*comment*]
[^I *commands*][# ...]
[^I *commands*][# ...]
...

where ^I represents a tab character. Shell metacharacters such as `*` and `?` are expanded in the command sequence only.

For example,

```
zeke: zeke.o
^I    cc -lc zeke.o -o zeke
```

3.1.1 Targets vs. rules

Within a description file, user-defined rules may replace `make`'s built-in rules. User-defined rules can appear in the makefile entry anywhere a target name can be given.

Some aspects of rule syntax are similar to target syntax. A target can be differentiated from a rule by the following criteria:

- A target name either does not begin with a period or does begin with a period and contains slashes.
- A rule begins with a period and does not contain slashes (see ''Transformation Rules'' for more information).

3.1.2 Fake targets

Not all targets correspond to files to be made. `make` has defined certain **fake targets** (targets to which no files correspond) to pass it certain options permanently. Examples of fake targets include

```
.SILENT
.IGNORE
.DEFAULT
.PRECIOUS
.SUFFIXES
```

For more information on `.SUFFIXES`, see the ''Suffixes'' section. For the others, see ''Options.''

3.1.3 Dependency statements

A **dependency statement** in a makefile asserts the logical relation between a target and its dependent(s). The syntax for a dependency statement is

target1 [*target2* . . .] : [:][*dependent1*. . .][; *commands*][#*comment*]

A sample dependency statement would be

```
dancing: music.o
```

A more complex dependency statement with an associated command sequence would be

```
yancy.o wally.o: defs.h ;
^I    echo "defs.h has been changed"
```

A dependency statement can contain either a single or a double colon.

> *Note:* A target name may appear in more than one dependency statement, but each of those statements must be of the same (single-colon or double-colon) type.

For the usual single-colon case, a command sequence may be associated with, at most, one dependency line; that is, a target cannot appear in more than one dependency line if there is a command sequence associated with more than one of them. For example, the fragment

```
yancy.o wally.o: defs.h

yancy.o quincy.o: menus.h
```

is correct because there is no command sequence associated with the dependencies in which `yancy.o` appears.

The following is also correct, because there is only one command sequence associated with the dependencies in which `yancy.o` appears:

```
yancy.o wally.o: defs.h
^I    echo "defs.h has been changed"
yancy.o quincy.o: menus.h
```

If the target is out of date with respect to any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), that command sequence is executed. Otherwise (if a command sequence is not specified) default rules may be invoked.

The following fragment is incorrect, because a target appears in two dependency lines, each of which is associated with a command, and single colons are used:

```
yancy.o wally.o: defs.h
^I    echo "defs.h has been changed"
yancy.o quincy.o: menus.h
^I    echo "menus.h has been changed"
```

In the double-colon case, a command sequence can be associated with each dependency line. For example:

```
yancy.o wally.o:: defs.h
^I    echo "defs.h has been changed"
yancy.o quincy.o:: menus.h
^I    echo "menus.h has been changed"
```

If the target is out of date with respect to any of the files on a particular line, the associated commands are executed, possibly in addition to default rules. If a target must be created, the entire sequence of commands is executed. This detailed form is of particular value in updating archive-type files.

3.1.4 Commands

A **command** is usually the command line required for producing the target(s) from the dependent(s). Syntactically, a command is any string of characters, not including a number sign (#) (except when the # is in quotes) and not including a newline.

> *Note:* When a command appears on a line separate from a dependency statement, *it must be preceded by a tab.* If not preceded by a tab, the command usually results in the

message "`Make: must be a separator on rules line` *x*`. Stop.`"

3.2 Comments

Comments are lines beginning with a number sign (#) and ending with a newline. These lines are ignored by `make`. (Blank lines are also ignored.)

3.3 `include` lines

The C syntax for `include` lines,

```
#include include_file
```

cannot be used in description files, because comments begin with a number sign. Therefore, the following policy was adopted for `include` lines in `make` description files.

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 string following is assumed to be a filename to be read by the current invocation of `make`. Thus, a makefile might contain the following:

```
include macro_defs   #reads in file macro_defs

lunch: supplies      #(entries follow)
```

In this example, `macro_defs` would be a file containing `make` macro definitions. No more than 16 levels of nested `includes` are supported.

3.4 Macro definitions

Macros are defined in `make` command line arguments or in the makefile. In the makefile, a macro definition is a line containing an equal sign, and the line must not begin with a colon or a tab. For example,

```
OBJECTS = x.o y.o z.o
```

The syntax for macro substitution is

```
$(name)
```

The name of the macro is either a single character after the dollar sign or a *name* inside parentheses or braces. Macro names longer than one character must be put inside parentheses or braces. For example, the following are valid macro invocations:

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

The last two invocations listed are functionally identical. Note that two dollar signs (`$$`) may also be used to denote a dollar sign. The following fragment illustrates the assignment and use of some macros:

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

In this example, `make` loads the three object files with the math library. The command line

```
make  "LIBES = -ll -lm"
```

would load them with both the `lex` (`-ll`) and the math (`-lm`) libraries.

Macro definitions on the command line override definitions in the description file, which, in turn, override the default macros.

For example, if you have defined macros in your makefile, you can redefine the library on the command line for a single run of `make`, without changing the meaning of the macros defined in the makefile. For example, the command

```
make "LIBES = -lg"
```

redefines the `LIBES` macro for this run.

To see a listing of the default macros, you can consult the `Macros` part of the listing produced by the command

```
make -np
```

3.4.1 Internal macros

The following are internal macros that change values during the execution of a description file. These internal macros are useful generic terms for current targets and out-of-date dependents. `make` sets these internal macros as follows:

`$@` Current target. The `$@` macro is set to the full target name of the current target. This macro is evaluated only for explicitly named dependencies. For example, in the following makefile, the current target is `zeke`, so `$@` is translated as `zeke`:

```
zeke: zeke.o
^I    cc zeke.o -o $@
```

`$?` Out of date relative to target. The `$?` macro is set to the string of names that were found to be younger than the target. This macro is evaluated when explicit rules from the makefile are evaluated. For example, the following makefile prints all files younger than `springtime`:

```
springtime: lp $?
```

`$<` Related file causing action. If the command was generated by a default rule, the `$<` macro expands to the name of the related dependent that caused the action. For example, the following makefile establishes an implicit rule to create targets from "`.o`" files:

```
.o:
^I    cc $< -o $@
```

`$*` Shared prefix, current and dependent files. If the command was generated by a default rule, the `$*` macro is given the value of the filename prefix shared by the current and dependent filenames. For example, the following makefile sets the prefix `$*` to `zeke` and links `zeke.o`:

```
zeke: zeke.o
^I    cc $*.o -o $*
```

In the following additions, the `D` refers to the directory part of the single-letter macro, and the `F` refers to the filename part of the single-letter macro. These are useful when building hierarchical makefiles.

`$(@D)`	Current target directory
`$(@F)`	Current target filename
`$(*D)`	Shared directory prefix
`$(*F)`	Shared filename prefix
`$(<D)`	Related dependent directory
`$(<F)`	Related dependent filename

For example, the following instruction uses the `D` to gain access to directory names in order to use the `cd` command:

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

3.4.2 Dynamic dependency parameters

The following parameters have meaning *only* within a dependency statement in a makefile.

`$$@` The current item to the left of the colon. The double dollar signs denote a metalevel macro, that is, a macro referring to another macro. Thus, `$$@` is a macro variable for whatever target is current, and `$@` is a macro for the current target. If the target is static, `$@` can be used instead of `$$@`; however, `$$@` allows for use of a dynamic target, a macro defined to denote many files, each of which is processed in turn. This is useful for building a large number of executable files, each of which has only one source file.

For example, the following makefile defines `CMDS` as the stipulated subset of single-file programs in the A/UX software command directory. Each of the programs (or `CMDS`) is compiled correctly in turn using this syntax.

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

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

(See "The Default Macro Settings" for more information on `$(CC)`.)

The dependency statement for the first item in the list of `CMDS` is translated as follows:

1. The target is set to `cat`.
2. The dependent is set to `cat.c` (the current target plus `.c`).
3. The `cc` command (optimized using `-O`) runs on the dependent (`cat.c`) if it is younger than the target.
4. The results are linked into the target file (`cat`).

Note: This syntax cannot be used for multiple-file programs. To deal with multiple-file programs, a directory is usually allocated and a separate makefile written. Then a specific makefile entry is made for files requiring nonstandard compilation.

`$$(@F)`

Another form of `$$@`, representing just the *filename* part of `$$@`. This parameter is also evaluated at execution time. For example, the following makefile maintains the `/usr/include` directory from a makefile in another directory:

```
INCDIR = /usr/include

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

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

The `$$(@F)` macro represents the *filename* prefix part of the current target `$@`. Because the target is also a macro, its value will equal each of the four files named in turn. On the first file's run,

1. The target is `stdio.h`.
2. The macro `$$(@F)` is `stdio` (the target *filename* prefix).
3. The next line copies the younger file (`$?`), if it exists, into the target file.
4. The last line changes the mode of the new target file (`$@`) (in this case, `stdio.h`) to read only.

This pattern is repeated for the other three files stated.

3.5 Options

3.5.1 Suppressing printing of commands

Normally, when `make` processes a description file, each command is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (`make -s`), or if the special name `.SILENT` appears on a line by itself as a target in the makefile, or if the command line begins with an `@` sign. For example,

```
@size make /usr/bin/make
```

If the command line above were in a description file, the printing of the command line itself would be suppressed by the `@` sign, but the output of the command would be printed.

3.5.2 Ignoring errors

The `make` program normally stops if any command signals an error by returning a nonzero exit status. Errors are ignored if any of the following are used:

- The `-i` flag on the `make` command line (where the scope is global)
- The fake target name `.IGNORE` in the description file (where the scope is the description file)

- A hyphen beginning the command string in the description file (where the scope is the command following the hyphen)

Thus, if the `-i` option is used, the target is `file.o`, and the compilation is unsuccessful, `make` effectively pretends that it worked. When `file.o` is found to be a dependent of some other files, `make` tries, for instance, to load all the object files together, and fails with an error message when one (`file.o`) is found to be missing. For all subsequent accesses (within this `make`), `file.o` is treated as though it existed and as though it were up to date. You should beware of this possible consequence of the `-i` option.

Some commands return with nonzero status even though they have worked correctly. For example, `diff` returns 1 to indicate the presence of differences in the compared files, and `rm` returns a nonzero status if the file you remove is already nonexistent. It is safer to use a leading hyphen for commands that may return a nonzero exit status without indicating an error, so `make` can continue processing.

3.5.3 Combining commands

As stated above, when `make` processes a description file, each command or individual command line is printed and then passed to a separate invocation of the shell after substituting for macros. Because the shell to which each command line is passed is a completely new invocation, care must be taken with certain commands (for example, `cd` and shell control commands) that have meaning only within a single shell process. If special means are not taken, the results of these commands will be lost before the next line is executed.

One way to avoid this is to combine two or more shell commands on one line, thus keeping the same shell active on each. This may be done in one of two ways. If both commands are kept on one physical line, a semicolon (`;`) may be inserted between the commands. If the commands are put on separate physical lines, but should form one logical line, a semicolon (`;`) and a backslash (\) should be appended to the first command. In the latter case, the semicolon separates the commands, and the backslash escapes the newline. Examples of these two methods follow:

```
cd ..; cc -c x.o y.o z.o

# with ; both commands can be on the same line


cd ..;\
cc -c x.o y.o z.o

# with ; and \ before <CR>, this is read as one line
```

3.5.4 Default commands

If a file, `prog`, must be made, but there are no explicit commands given or relevant rules to apply, `make` looks for commands dependent on the target `.DEFAULT` to use. If there is no `.DEFAULT` target, `make` prints a message,

```
Don't know how to make prog. Stop
```

and stops. Thus, `.DEFAULT` may be set up by the user to specify default-case treatments for files not covered by `make`'s built-in rules. (For a listing of the types of file compilations covered by these rules, see the ''Transformation Rules'' section.)

3.5.5 Saving files

If a file or files are assigned as dependent to `.PRECIOUS`, those files will not be removed, regardless of any command to the contrary. This is especially helpful to avoid the removal of targets when an interrupt or quit is sent.

3.5.6 Use of selected flag options

`-n` The `-n` option is useful to discover what `make` would do.

```
make -n
```

instructs `make` to print out the commands it would issue, without actually executing them.

`-t` The `-t` (''touch'') option updates the modification times on the affected file, and thereby can avoid a large number of superfluous recompilations. Be careful when using this option.

-d The -d (''debug'') flag prints out a detailed description of what it is doing, including the file times. The output is verbose and potentially confusing. This is therefore recommended as a last resort.

4. Suffixes and rules

The make program uses a table of significant suffixes and a set of transformation rules to supply default dependency information and implied commands. All of this information is stored in an internal table (the default rules) that has the form of a description file. (If the -r flag option is specified, this internal table is not used.)

4.1 Suffixes

The list of suffixes is actually the dependency list for the fake target .SUFFIXES in the description file. The make program searches for a file with any of the suffixes on the list. If such a file exists and there is a transformation rule for that combination, make transforms a file with one suffix into a file with another suffix.

The order of the suffix list is significant because the list is scanned from left to right. The first name formed that is associated with both a file (in the directory) and a rule (in the makefile or default rules) is made, *and no others.*

Note: You should know the order of the default suffix list if you are not specifying a command in the makefile. Otherwise, you may make an unexpected file.

The default suffix list is as follows:

Table 16-1. Default suffix list

Suffix	File type
.o	Object file
.c	C source file
.e	EFL source file
.r	ratfor source file
.f	Fortran source file
.s	Assembler (as(1)) source file
.y	yacc-C source grammar
.yr	yacc-ratfor source grammar
.ye	yacc-EFL source grammar
.l	lex source grammar

4.2 Transformation rules

make has an internal table of **transformation rules** that perform certain default commands if there is no command specified in the makefile. Note that the default rules are also known as the ''implicit rules.'' There are two types of transformation rules, ''double suffix rules'' and ''single suffix rules.'' In double suffix rules, the stage of compilation is discerned from the suffix (for example, x.c is a source file and x.o is an object file). These rules are phrased in terms of transformations from one type of suffix to another. The names of these rules are formed by concatenating the two filename suffixes; for example, the name of the rule to transform a .r file to a .o file is .r.o.

Single suffix rules describe the transformation of a file with a given suffix into one with no suffixes or a null suffix.

If a rule is listed in the internal table and there is no command sequence given in the description file, the rule is used. Thus, standard transformations (from one type of file to another; for example, from a source file to an object file) do not call for a makefile entry unless nonstandard treatment is required.

If a rule is used (that is, if a default command is generated), the $* macro is given the value of the *filename* prefix of the file to be maintained. Then the $< macro is the name of the dependent that caused the command.

make has all the required information for compiling programs written in languages supported by A/UX. For example, after the command

```
make x.o
```

where x.o is a C language object file, make searches for a file called x.c (a C language source file) in the local directory. If it finds x.c, make consults its default rules for compilation. make finds the rule .c.o, which states the default command

```
cc -O -c x.c
```

which make then issues to produce x.o.

make uses the default suffix list (see "Suffixes") to decide when to invoke which rules. This list tells the order in which to search for certain suffixes.

Within make's default rules file, the name of the rule to follow appears in the place of the target filename. Thus, the .c.o rule is represented by

```
.c.o:
^I    cc -O -c [filename].c
```

The contents of the current default rules file used by make can be directed to standard output with the command

```
make -np
```

Any error messages produced at the end of this output should be ignored. The example that follows shows a representative file, giving one version of the default rules used by make.

Figure 16-3. Sample listing of default rules file

```
#   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=-o
     AS=as
     ASFLAGS=
     GET=get
     GFLAGS=

#   SINGLE SUFFIX RULES

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

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

.sh:
     cp $< $@

.sh~:
     $(GET) &(GFLAGS) -p $< > .sh
     cp $* .sh $*
     -rm -f $* .sh
```

Figure 16-3. Sample listing of default rules file (continued)

```
#   DOUBLE SUFFIX RULES

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

.c~.o:
     $(GET) $(CFLAGS) -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) $(CFLAGS) -c y.tab.c
     rm y.tab.o$@

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


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

Figure 16-3. Sample listing of default rules file (continued)

```
.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

.y.c:
     $(YACC) $(YFLAGS) $<
     mv y.tab.c $@

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

.l.c:
     $(LEX) $<
     mv lex.yy.c$@

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

.c~.a:
     $(GET) $(GFLAGS) -p $< > $*.c
     $(CC) -c $(CFLAGS) $*.c
     ar rv $@ $*.o

.s~.a:
     $(GET) $(GFLAGS) -p $< > $*.s
     $(AS) $(ASFLAGS) -o $*.o $*.s
     ar rv $@ $*.o
     -rm -f $*.[so]

.h~.h
     $(GET) $(GFLAGS) -p $< > $*.h
```

If there are two paths in the rules connecting a pair of suffixes, the longer one is used only if the intermediate file exists or if it is named in the description file. The following are two examples illustrating how this works:

1. If an `x.o` file is needed and a file called `x.c` is found in the current directory or specified in the description file, the `x.o` file is compiled using `x.c`. If an `x.l` also exists and is out of date with respect to `x.c`, that file is processed through `lex` before compiling the result. This is a case of the longer path (`x.l` to `x.c` to `x.o`) being used since the intermediate file (`x.c`) exists.

2. If the file `x.o` is needed and `x.l` but not `x.c` is found, `make` discards the intermediate C language file (in this case, `x.yy.c`) and uses the shorter path (`x.l` to `x.o`).

4.2.1 The default macro settings

You can change the names of some of the compilers used in the default rules, or the flag arguments with which they are invoked, by knowing the macro names used. These macro names, the default compilers they denote, and their associated flags are as follows:

Table 16-2. Macro names and default compilers

Compiler	Macro	Flags
`make` command	`MAKE`	`MAKEFLAGS`
Assembler (`as`)	`AS`	–
C compiler (`cc`)	`CC`	`CFLAGS`
`ratfor` compiler	`RC`	`RFLAGS`
EFL compiler	`EC`	`EFLAGS`
`yacc`-C compiler	`YACC`	`YFLAGS`
`yacc-ratfor` compiler	`YACCR`	`YFLAGS`
`yacc`-EFL compiler	`YACCE`	`YFLAGS`
`lex` compiler	`LEX`	`LFLAGS`
`get` command	`GET`	`GFLAGS`

These macros can be used as arguments on the command line to change defaults for one run of `make`. For example, the command

```
make CC=newcc ...
```

causes the `newcc` compiler to be used instead of the usual C language compiler. An example of the use of flags follows:

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

passes the `-O` flag to the C compiler, `cc`, causing the C language optimizer to be used.

Sometimes it is possible to use macro redefinition instead of stating a local version of the default rule. (Of course, this change is temporary, because it takes place on the command line, and must be restated, whenever desired, every time the file is remade.) To change the `.c.o` rule you can say

```
make "CFLAGS=-V" thorax.o
```

and the flag option `-V` will replace the default setting for `CFLAGS` for this one run.

4.3 Changing default suffixes and rules

4.3.1 The default suffix list

You can add suffixes to the end of the default suffix list, change the order of the list, or change the contents of the list.

If you append new names to the suffix list, an entry can be included for `.SUFFIXES` in the description file. The dependents to `.SUFFIXES` are then added onto the end of the default list.

To change the order or contents of the list, you must be aware that a `.SUFFIXES` line without any dependents deletes the current list of suffixes. Therefore, you must clear the current list to change the order of names. Thus, to install a new list, include lines such as

```
.SUFFIXES :                      # removes old list
.SUFFIXES : .n .n~ .l .l~        # installs new list
```

4.3.2 The default rules

You can modify or replace a default rule in a makefile. For example, if you define a `.c.o` rule in a makefile, your definition overrides the default one. For example, Figure 16-4 defines a new `.c.o` rule:

Figure 16-4. Replacing a default rule

```
.c.o: cc -V -c $<         #Rule, not target

stomach.c: stomach.l      #First target

stomach.l: defs.h
```

This invokes the `-V` option of `cc` every time a `.o` file is linked, printing the version of the assembler that was used.

5. Operation

5.1 Environment variables

Environment variables from the shell are read by `make` and considered in processing makefiles. These variables include `PATH`, `HOME`, `TERM`, `SHELL`, `TERMCAP`, and `LOGNAME` (see *A/UX User Interface* for more information on environment variables). Thus, a reference to `$(HOME)`, otherwise undefined in a makefile, will be translated correctly into the full pathname for the user's home directory.

> *Note:* The value of the `SHELL` variable determines which shell is used to execute commands in the makefile (by default, your login shell). If you wish to include shell scripts that require a different shell (for example, a Bourne shell script when your login shell is the C shell), you must specify the new shell either on the command line:
>
> make [*options*] SHELL=/bin/sh
>
> Or you can do it by including the following line at the beginning of your description file:
>
> ```
> SHELL=/bin/sh
> ```

To see which environment variables `make` recognizes in the present directory (directed to standard output), give the command

```
make -np | head -50 | more
```

The first part of this command's output prints the environment variables.

`make` also maintains an environment variable named `MAKEFLAGS`. The value of this variable is all command line flag arguments (without minus signs). The macro is ''exported'' and accessible to further invocations of `make`. Command line flags and assignments in the makefile update `MAKEFLAGS`. `MAKEFLAGS` is read and set again when the environment settings are read by `make`.

The first part of this command's output prints the environment variables.

5.2 Precedence

Environment variables are read and added to the macro definitions each time that `make` executes. Precedence is a prime consideration in doing this properly. The following is the default precedence of assignments:

1. Command line
2. Makefile(s)
3. Environment
4. Default macros

When executed, `make` assigns macro definitions in the order stated, by doing the following:

- Reading the `MAKEFLAGS` environment variable.

  Each letter in `MAKEFLAGS` is processed as an input flag argument, unless the letter is `-f`, `-p`, or `-r`. These flag options give directions to `make` involving overall processing, as follows:

`-f`	Precedes the makefile filename
`-r`	Leaves out the built-in rules
`-p`	Prints out all macro definitions and target descriptions

  If the `MAKEFLAGS` variable is null, or is not present, `MAKEFLAGS` is set to the null string. This pass establishes if the debug (-d) flag is set, in time for this to be of use.

- Reading and setting the input flags from the command line. The command line adds to the previous settings in the MAKEFLAGS environment variable.

- Reading macro definitions from the command line. Any macro definitions set from the command line cannot be reset. Further assignments to these macro names are ignored.

- Reading the internal list of macro definitions. make reads its default rules file, which contains the internal list of macro definitions. For example, if the command

  ```
  make -r ...
  ```

  is given, and a makefile already includes all of the rules that are found in make's default rules file (for instance, by means of an include line; see "include lines"), the -r option would not have the stated effect of "ruling out" the rules. It would do the right thing, namely, not go to its default rules itself, but it is not bright enough to undo an include line in a makefile. In fact, the effect would be identical to that occurring if both the -r option and the include line in the makefile were excluded, since they cancel each other out.

- Reading the environment settings in the shell. The environment variables are treated as macro definitions and marked as exported.

  > *Note:* Because MAKEFLAGS is not a variable in make's default rules file, this step has the effect of doing the same assignment twice. (The exception to this is when MAKEFLAGS is assigned on the command line.)

  The MAKEFLAGS variable is read and set again.

- Reading the makefile(s). Assignments in the makefile(s) override the environment unless the -e flag is used. The command line option -e instructs make to override the makefile assignments with the environment settings.

If assigned, the `MAKEFLAGS` variable overrides the environment. This is useful for further invocations of `make` from the current makefile. There is no way to override command line assignments. For example, if the command

```
make -e ...
```

is given, the variables in the environment override the definitions in the makefile and reset the precedence of assignments to the following:

1. Command line
2. Environment
3. Makefile(s)
4. Default macros

This has the effect of giving the environment priority over the makefile, as opposed to the reverse in the default case.

5.3 Archive libraries

A `.a` suffix rule builds libraries. (There is no actual `.a` suffix appended to the filename, however; see below for how to recognize candidates for this rule.) For example, the `.c.a` rule is the rule for all of the following:

- Compiling a C language source file
- Adding a C language source file to the library
- Removing the `.o` cadaver of the C language source file

The `.y.a` rule is the rule for performing the same functions on a `yacc` file; the `.s.a` rule, for an assembler file; and the `.l.a` rule, for a `lex` file.

The current archive rules defined internally are `.c.a`, `.c~.a`, and `.s~.a`. (See the section on ''SCCS Filename Prefixes'' for an explanation of the tilde (~) syntax.)

Programmers may choose to define additional rules in the makefile(s).

A library is then maintained with the following makefile:

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

Note: The first parenthesis in the filename identifies the target suffix rule, not an explicit `.a` suffix.

For example, the actual rule `.c.a` is defined as follows:

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

In the `.c.a` rule:

`$@`	This macro is the `.a` target. (Using the library example, this macro would be defined as `lib`.)
`$<` and `$*`	These macros are set to the out-of-date C language file, and the filename without the suffix, respectively. Using the previous example, these macros would be defined as `ctime.c` and `ctime`. Using this example, the `$<` macro could have been changed to `$*.c`.

When `make` sees the `lib(ctime.o)` instruction in the makefile (assuming the object in the library is out of date with respect to `ctime.c`, and there is no `ctime.o` file), it translates that construct into the following sequence of operations:

1. `make lib`.
2. To `make lib`, `make` each dependent of `lib`.
3. `make lib(ctime.o)`.
4. To `make lib(ctime.o)`, `make` each dependent of `lib(ctime.o)`. (There are none in this example.)

   To allow `ctime.o` to have dependencies, the following syntax is required:

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

Thus, explicit references to `.o` files are unnecessary.

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

5. Use default rules to try to build `lib(ctime.o)`. (There is no explicit rule.)

   > *Note:* It is the first parenthesis in the name `lib(ctime.o)` which identifies the (`.a`) target suffix. 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 suffix 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 `$*.X`. The first `.X` (in the `.SUFFIXES` list in the default rules file) that fulfills these conditions is `.c`, so the rule is `.c.a` and the file is `ctime.c`.
8. Set `$<` to `ctime.c` and execute the rule.

   In fact, `make` must then make `ctime.c`. The search of the current directory yields no other candidates, however, and the search ends.
9. The library has been updated. Perform the next instruction associated with the `lib:` dependency. Therefore, `make` will echo

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

5.4 SCCS files

make can be used on SCCS files and knows to run get on them, if required, before otherwise processing them. Those unfamiliar with SCCS (Source Code Control System) should refer to Chapter 17, ''SCCS Reference.''

5.4.1 SCCS filename prefixes

make syntax does not allow for direct prefix references. SCCS files constitute the one important exception to this rule.

SCCS filenames are preceded by a s. prefix. make uses a tilde (~) appended to the suffix to identify SCCS files. The expression .c~.o refers to the rule that transforms an SCCS C language source file into an object file.

The following example shows a transformation from an SCCS filename to a name with a suffix already fixed for make: the SCCS filename s.file1.c into the non-SCCS, make-ready filename file1.c~. This file is then assembled using the command

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

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

5.4.2 SCCS filename suffixes

The following SCCS suffixes are internally defined:

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

5.4.3 SCCS transformation rules

The following rules involving SCCS transformations are internally defined:

```
.c~:    .l~.o:    .sh~:    .y~.c:    .c~.o:

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

These rules transform SCCS files into non-SCCS format and perform the compilations indicated by the letter combinations in the rule names. (See "Transformation Rules" for how to translate rules names into the rules they designate.)

Other rules and suffixes that may prove useful can be defined using the tilde as a handle on the SCCS filename format.

5.4.4 SCCS makefiles

SCCS makefiles are "invisible" to `make`, in that, if you give the command

```
make
```

and only a makefile named `s.makefile` resides in the current directory, `make` will `get`, read, and remove the file. `get` creates a file called `makefile` which remains in the directory (in addition to the *p-file*, `p.makefile`). If the `-f` option is used, `make` will `get`, read, and remove the specified makefile (as well as `include` files), creating a non-SCCS makefile named the same as the old SCCS version, except that the `s.` prefix is removed.

6. Advanced topics

6.1 Walking the directory tree

It is possible to get `make` to walk the directory tree, either by guiding it explicitly or by including a shell script that will discover, implicitly, what directories are there, so that it can visit them. While `make` is in each directory, it can make the files specified in the directory's makefile. This allows you to bring whole systems up to date without yourself changing directories by having `make` follow directions in one local (meta-)makefile.

The explicit route is by far the easiest. If you know the structure of your tree and the names of all the directories you intend to visit, you can include commands in a makefile in the directory at the top of your tree. If, below your current directory, you have directories named `io`, `os`, and others, you can include lines like the following in your makefile

```
all:
^I    cd io; make -f io.mk; \
^I    cd ../os; make -f os.mk;
```

Note: The backslash (\) at the end of command lines is necessary if you want to keep the same invocation of the shell active for a group of commands. If a different shell is invoked, the knowledge of being in a new directory is lost.

If, for example, no backslash terminated the first command line, and so a different shell was invoked on the second line, the second `cd` would be executed from the parent directory for `io` and `os` instead of from the `io` directory. In this case, to keep the same effect, the line should read

```
^I    cd os; make -f os.mk;
```

As this shows, it is possible to write a script that does invoke a new shell with each line and still travels the directory tree. This just changes the mode of travel: With the one-shell-per-journey method, you state explicit directions for going to each directory from where you are relative to that directory *and for going back to the originating directory afterward.* With the one-shell-per-command method, you state explicit directions (that is, a full pathname) for going to the directory, and the return trip is done for you when the shell you are using quits.

To travel a tree of unknown structure but with fairly standard makefile names (like *dirname*`.mk`, where *dirname* stands for the name of the directory where the file is located), you could use a fragment like the following in your makefile:

```
subdirs:
^I    for i in `find /pathname -type d -print`; \
^I    do \
^I        if test -f $$i/$$i.mk; \
^I        then  \
^I            cd $$i; \
^I            $(MAKE) -f $$i.mk; \
^I        fi \
^I    done
```

Note: The above is a Bourne shell script, and it will work only if your login shell is `/bin/sh` or your `SHELL` environment variable is set to /bin/sh. See "Environment Variables" for more information on using different shells to execute a makefile.

6.2 The make predecessor tree

The $! macro represents the current predecessor tree. A make predecessor tree contains the series of files linked through the dependency relation for one run of make. For example, using the makefile

```
all:     cat
^I     @echo cat up-to-date
cat:     cat.c
^I     echo $!
```

when the command echo $! is executed, the variable $! evaluates to

```
cat.c cat all
```

which is the current predecessor tree of this run of make, read from left to right (leaf to root, respectively). The connection constituting branches is the "is depended on by" relation: The leftmost file is depended on by the next file to the right, and so on. Thus, the nodes are dependents of their right neighbors and are targets of their left neighbors (except for the leaf). The predecessor tree can be useful as a debugging tool for make itself, if what it has done does not make sense. Examination of the tree can reveal why certain files were updated, or which files were touched in this run of make.

Another means of debugging must be found if make prints the following message:

```
$! nulled, predecessor circle
```

If the predecessors of a file are circular, they cannot form a tree, and one will not be printed. The actual evaluation of the $! macro is terminated, and the macro's value is set to null.

6.3 The makefile as shell script

If a target cannot be found in the local or specified directory, make attempts to create it. This feature of make's processing may be exploited by the advanced user. When make discovers the absence of the file corresponding to *target*, it considers *target* to be out of date and so executes the specified command sequence. If the results do not include creating the target, this leaves the directory in question in the same state, ready for the same scenario to take place whenever the make command is invoked.

This allows a makefile to function more like a shell script, with each absent target causing make to try to create it, using the command sequence specified.

6.3.1 Unintended targets

make considers missing files to be out of date and processes them. Conversely, existing files may be deemed up to date wrongfully (due to user error) and skipped for processing by make. This might happen in the situation described in ``The Makefile as a Shell Script'' if one of the targets was

```
print:
^I      lp foo bazz fizz
```

Here the command sequence creates no file called print, so the same description file can be used over and over for maintenance, each time executing this line. If, however, you inadvertently name a program in that directory print, this latter file's modification information will be checked to determine if print needs to be remade, probably finding it to be up to date, and telling you so on the screen. Failure to note this might cause a bug that is hard to trace in the working of the ``shell script'' description file, even though the entry for print is correct.

6.3.2 Mnemonic targets

A useful method is to include targets with mnemonic names and commands that do not actually produce a file with the same name as the label in the shell script. These entries can take advantage of make's ability to generate files and substitute macros. For example, save might be included to copy a certain set of files, or an entry cleanup might be used to throw away unneeded intermediate files. It is also possible to maintain a zero-length file purely to keep track of the time at which certain commands were performed. For example,

```
print: $(FILES)
^I    pr $? | lp
^I    touch print
```

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 time since the file `print` was last touched. The `$?` macro in the command sequence then picks up only the names of those files changed since `print` was touched. The `touch` command creates this zero-length file if no file called `print` exists in this directory.

6.3.3 Macro translation

To supplement macro definition and substitution, `make` also provides a macro translation facility. As a macro is evaluated, the translation takes place within the set of names of items to which the macro refers. (Such item names are probably filenames; in any case, they are considered as strings, where a string is delimited by blanks or tabs.) Thus, the macro translation facility allows for more refined and narrow macro definitions and for more concise code in description file command sequences.

The format for macro translation follows:

`$(`*macro-name*`:`*string1*`=`*string2*`)`

This tells `make` to substitute *string2* for *string1* everywhere among the item names produced on evaluation of *macro-name*. (`make` assumes that these substitution strings are suffixes.) Thus,

[*process*] `$(?:.o=.c)`

results in *process*ing of all files younger than target, except that, in this list of files, wherever there was a `.o` file, a `.c` file will be *process*ed instead.

To illustrate the usefulness of this facility, consider the following example situation: To maintain an archive library, the out-of-date members must be accumulated and a shell script must be written to handle all the C programs. The following fragment will optimize the executions of make for archive libraries:

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

The translation (`$(?:.o=.c)`) tells `make` to compile from the `.c` file, every time it finds a `.o` file younger than the target library. (This would act as an added check to ensure that all changes were incorporated, as the `.c` files might have been altered without being subsequently recompiled.) This results in the rule desired (`.c.a`), rather than a nonstandard `.o.a` rule.

6.4 A warning for system administrators

If the system's setting for `date` is wrong (especially if it is very far behind the actual date), `make` can get very confused. Since `make` works by comparing previous dates with the current one, it is important to make sure that what it is given as the current date is accurate. Therefore, to ensure proper functioning of `make`, the accuracy of `date` should be checked frequently.

Chapter 17
SCCS Reference

Contents

Figures

Tables

Chapter 17

SCCS Reference

1. Introduction

The source code control system (SCCS) is a collection of A/UX commands that controls and reports on changes to files of text. SCCS is a valuable tool for version management of program source code or ordinary text files. In large group projects, SCCS prevents multiple, inconsistent versions of files from accumulating in several places. For a single user, multiple versions of a file can be stored without using a lot of disk space, previous versions can be easily reconstructed, and versions can be kept track of with a simple, consistent numbering scheme.

SCCS provides facilities for

- Efficient storage of multiple versions of files
- Retrieving earlier versions of files
- Controlling update privileges to files
- Identifying the version of a retrieved file
- Recording when, where, why, and by whom each change was made to a file

SCCS stores the original file on disk. Whenever changes are made to the file, SCCS stores only the changes. Each set of changes is called a **delta.** When you retrieve a particular version of the file (the default is the most recent version), SCCS applies the appropriate deltas to the original file to reconstruct that version.

This chapter provides an introduction and a general reference guide to SCCS. The following topics are covered here:

- SCCS for beginners: A step-by-step guide to creating SCCS files, updating them, and retrieving a version of a file.

- SCCS files: A description of the protection mechanisms, format, auditing, and delta numbering of SCCS files. The differences between individual SCCS use and group or project SCCS use are discussed, and the role of the SCCS administrator in a group project is introduced.
- SCCS command conventions: A description of the conventions that generally apply to SCCS commands and the temporary files created by SCCS commands.
- SCCS command summary: A summary of SCCS commands and their arguments.

In addition to the programs described in this chapter, the `sccs` command provides a front end to SCCS functionality. Basically, the `sccs` front end runs the SCCS commands documented in the ''SCCS Command Summary'' as well as several commands that are equivalent but easier to use than the most frequently used SCCS commands. See `sccs(1)` in *A/UX Command Reference* for more information on the `sccs` front end.

2. SCCS for beginners

2.1 Creating an SCCS file

Using a text editor, create an ordinary text file named `lang` that contains a list of some programming languages:

```
C
PL/I
FORTRAN
COBOL
ALGOL
```

To bring the tools of SCCS into play, you need to create a (different) file that various SCCS commands will read and modify. You can do this with the `admin` command, as follows:

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

The `admin` command with the `-i` keyletter (and its value, `lang`) creates a new SCCS file and initializes it with the contents of the file named `lang`. An initial SCCS delta is created by applying a set of changes (the contents of `lang`) to a new (null) SCCS file (`s.lang`).

All SCCS files must have names that begin with "`s.`". This effectively limits SCCS filenames to 12 characters.

Each delta is assigned a name called the SCCS Identification string, or **SID.** The SID is normally composed of two components (the release number and the level number) separated by a period. For example, the initial version of a file is delta 1.1 (that is, release 1, level 1). SCCS keeps track of subsequent versions of a file by incrementing the level number whenever you create a new delta. The release number can also be changed (allowing, for example, deltas 2.1, 3.1, and so on) to indicate a major change to the file.

The `admin` command returns a warning message (which may also be issued by other SCCS commands):

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

The absence of keywords is not a fatal error under most conditions, and this warning message does not affect the SCCS file you have just created. In the following examples, this warning message is not shown although it may actually be issued by the commands.

You should now remove the `lang` file from your directory:

```
rm lang
```

2.2 Retrieving a file and storing a new version

To reconstruct the `lang` file you just deleted, use the SCCS `get` command:

```
get s.lang
```

This retrieves the most recent version of file `s.lang` and prints the messages

```
1.1
5 lines
```

(the SID of the version retrieved, and the length of the retrieved text). The retrieved text is placed in another file called the *g-file*. The name of the *g-file* is formed by deleting the `s.` prefix from the name of the SCCS file. Hence, the file `lang` is reconstructed.

When you use the `get` command with no keyletters (in the format above) the `lang` file is created with read only mode (mode 440), and no information about the SCCS file is retained. If you want to be able to change an SCCS file and create a new version, use the `-e` (edit) keyletter on the `get` command line:

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

The `-e` keyletter causes `get` to create `lang` with read-write permission and places certain information about the SCCS file in another file called the *p-file*, which will be read by the `delta` command when the time comes to create a new delta.

The same messages are displayed, as well as the SID of the next delta (to be created). For example,

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

produces

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

After this command, you can edit the `lang` file and make changes. For example, suppose that you use `vi` to create the following new version of the file:

```
C
PL/I
FORTRAN
COBOL
ALGOL
ADA
PASCAL
```

The command

```
delta s.lang
```

records the changes you made to the `lang` file within the SCCS file. SCCS prints the message

```
comments?
```

Your response 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`. When this process is complete, the changes to `lang` are stored in `s.lang`, and `delta` displays

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

The number 1.2 is the SID of the new delta, and the next three lines refer to the changes recorded in the `s.lang` file.

2.3 Retrieving versions

The `-r` keyletter allows you to retrieve a particular delta by specifying its SID on the `get` command line. For the previous example, 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.

The first command retrieves the most recent version of the SCCS file, because no SID is specified. When you omit the level number of the SID (as in the second command), SCCS retrieves the most recent level number in that release (in the previous example, the latest version in release 1, namely 1.2). The third command explicitly requests the retrieval of a particular version (in this case, also 1.2).

Whenever a major change is made to a file, the significance of that change is usually indicated by changing the release number (the first component of the SID) of the delta being made. Because normal automatic numbering of deltas proceeds by incrementing the level number (the second component of the SID), you must explicitly change the release number as follows:

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

Because release 2 does not yet exist, `get` retrieves the latest version *before* release 2 and changes the release number of the next delta to 2, naming it 2.1 rather than 1.3. This information is stored in the *p-file* so the next execution of the `delta` command will produce a delta with the new release number. The `get` command then produces

```
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. Subsequent versions of the file will be created in release 2 (deltas 2.2, 2.3, and so on).

2.4 On-line information

The `help` command is useful whenever there is any doubt about the meaning of an SCCS message. Detailed explanations of almost all SCCS messages can be found using the `help` command and the code printed in parentheses after the message.

If you give the command

```
get abc
```

SCCS prints the message

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

The string `co1` is a code that can be used to obtain a fuller explanation of that message using the `help` command. The command

```
help co1
```

produces

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

3. SCCS files

This section discusses the protection mechanisms used by SCCS, the format of SCCS files, recommended procedures for auditing SCCS files, and how deltas are numbered.

3.1 Standard A/UX protection

In addition to the special SCCS flags and keyletters described in ``SCCS Protection Mechanisms,'' SCCS uses standard A/UX protection mechanisms to prevent you from making changes to SCCS files using non-SCCS commands. The following precautions are automatically taken by SCCS:

- When you create an SCCS file (using `admin`), it is automatically given mode 444 (read only) if your `umask` is less than or equal to 333. If your `umask` is 334 the SCCS file will be created with mode 440 (no read permission for others). If your `umask` is 344 the SCCS file will be created with mode 400 (read permission for the owner only). If your `umask` is 444 or higher, the SCCS file will be created with no permissions across the board, and a lock file, also called a *z-file*, will be created. The preferred mode for an SCCS file is 444; this protects against modifying SCCS files using non-SCCS commands and should not be changed.
- If you make a hard link from an SCCS file to another file, SCCS commands will not process the SCCS file. SCCS commands produce an error message rather than process a file that has been linked. The reason for this is the same: Protection is provided against using non-SCCS commands to modify SCCS files.

3.2 SCCS protection mechanisms

SCCS provides the following protection features directly: three SCCS file flags (release ceiling, release floor, and release lock) and a user list for SCCS files.

The SCCS file flags are set using the `-f` keyletter with the `admin` command. This keyletter specifies a flag and possibly a value for the flag, to be placed in the SCCS file. Several `-f` keyletters may be supplied on a single `admin` command line (see ''SCCS Flags'' under ''Create SCCS Files: `admin`'').

The flags used for file protection are

`c` *ceiling*	The highest release (''ceiling'') that can be retrieved by a `get` command for editing. *ceiling* is a number less than or equal to 9999. If this flag is not used, the default value for *ceiling* is 9999, which allows all releases up to and including 9999 to be retrieved for editing.
`f` *floor*	The lowest release (''floor'') that can be retrieved by a `get` command for editing. *floor* is a number less than

9999 and greater than 0. If this flag is not used, the default value for *floor* is 1, which allows the first release to be retrieved for editing.

`l` *list*	A list of "locked" releases to which deltas can no longer be made. (See `admin`(1) in *A/UX Command Reference* for the complete syntax of this list.) The `get -e` command fails if you attempt to retrieve one of these locked releases for editing. The character `a` in *list* can be specified to protect all releases for the named SCCS file.

SCCS files may also contain a user list of login names and/or group IDs of users who are or who are not allowed to create deltas of that file. This list is empty by default, which means that anyone may create deltas. To add names to the list (either to allow permission or to deny it) the `-a` keyletter is used with the `admin` command. The argument to the `-a` keyletter can be

login-name	A login name or numerical group ID may be specified; a group ID is equivalent to specifying all login names common to that ID.
! *login-name*	If a login or group ID is preceded by an exclamation character (`!`), these ID's are denied permission to make deltas.

These features are described in more detail under the `admin` command.

3.3 Administering SCCS

If you are using SCCS to manage personal files, the protection mechanisms described above should be used to keep certain releases from being modified, or to prevent you from accidentally modifying your files without using SCCS.

Aside from these protections, you can simply use SCCS directly. See "Delta Numbering" for information on storing and retrieving different releases.

3.3.1 Group project administration

If you are using SCCS to manage and protect files in a large project with several users having access to the same files, a single user should own the SCCS files and directories. This single user will be the only one to administer the SCCS files.

The following precautions are recommended:

- Directories containing SCCS files should be mode 755. This allows only the owner of the directory to modify its contents.
- 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, for example, subsystems of a large project.
- No SCCS users other than the SCCS administrator should be able to use those commands that require write permission in the directory containing the SCCS files. Instead, a project-dependent program should be written to provide an interface to certain SCCS commands, usually the `get`, `delta`, and, if desired, `rmdel` and `cdc` commands.

This last precaution requires that you write an interface program (usually specific to the project) that invokes the desired SCCS command and gives other users (who are not the owners of the SCCS files) the permissions they need to modify specific SCCS files, using only those commands that are linked to the interface program.

> *Note:* If you are not using the `sccs` front end (see `sccs`(1) in *A/UX Command Reference*), you may need to write an interface program such as the sample program shown in Figure 17-1 to handle special file permissions for a particular project.

The sample program in Figure 17-1 causes the invoked command to inherit the privileges of the interface program for the duration of that command's execution. Users whose login names or group IDs are in the user list for that file (but who are not the owner), and who have the path to the executable interface program in their `PATH` variable, are given the necessary permissions only for the duration of the execution of the interface program. They can modify the SCCS files only through the use of those commands that are linked to the interface program.

Figure 17-1. Sample interface program for group projects

```
main (argc, argv)
int argc;
char *argv[];
{
   register int i;
   char cmdstr [BUFSIZ];

   /* 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 program
      (strip off directory prefix, if any) */
   argv[0] = sname (argv[0]);

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

This sample interface program is an example only; the functions `sname` and `filearg` are not standard functions. You should write these and any other functions required by your project.

Such an interface program must be owned by the SCCS administrator, must be executable by the new owner, and must have the `setuid` (set user ID on execution) bit on (see `setuid`(2)).

Links can then be created between the executable interface program and the command names. For example, if the path to the file is

```
/sccs/interface.c
```

then the commands

```
cd /sccs
cc interface.c -o inter
```

compile the program into the executable module `inter`. At this point, the command

```
chmod 4755 inter
```

sets the proper mode and `setuid` bit. You can then create links from any directory with the commands

```
ln /sccs/inter get
ln /sccs/inter delta
ln /sccs/inter rmdel
ln /sccs/inter cdc
```

The full pathname of the directory containing the links must then be included prior to the `/usr/bin` directory in the `PATH` variable (in the `.profile` or `.login` files of all SCCS users who need to use the desired SCCS commands). For example,

```
PATH=(.:/usr/new:/bin:/sccs:/usr/bin)
```

Depending on the type of interface program you have written, the names of the links can be arbitrary (if the program can determine from them the names of the commands to be invoked), the pathname to your project can be supplied, and so on. If the pathname to your project is supplied in the interface program, the user can use the syntax

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

regardless of where the user is currently located in the file system.

3.4 SCCS file formats

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

checksum	This part of the file contains the sum of the ASCII values of all characters in the file (not including the checksum itself). The SCCS checksum is described in ''SCCS File Auditing.''

delta table	This part contains information about each delta, such as type, SID, date and time of creation, and commentary.
user list	This is a list of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas. The user list is described under "SCCS Protection Mechanisms."
flags	This part contains indicators that control certain actions of SCCS commands. Flags are discussed under "Create SCCS Files: `admin`."
descriptive text	This is arbitrary text provided by the user, usually comments that provide a summary of the contents and purpose of the file. Descriptive text is discussed under "Create SCCS Files: `admin`."
body	This is the actual text of the ASCII file being administered by SCCS, intermixed with internal SCCS control lines.

For information regarding the physical layout of SCCS files, see `sccsfile`(4) in *A/UX Command Reference*.

Note: Because SCCS files are ASCII files, they can be processed by other A/UX commands such as `vi`, `grep`, and `cat`. This can be convenient when an SCCS file must be modified manually or when you simply want to look at the file. However, it is extremely important to be careful about introducing changes that will affect future deltas. It is wise to make a backup copy first.

3.5 SCCS file auditing

On rare occasions (such as a system crash) an SCCS file may be destroyed or corrupted (that is, one or more blocks of it may be destroyed). If the entire SCCS file has been trashed, the SCCS commands issue an error message when you attempt to process that file. In this case, you need to restore the file from your most recent backup copy.

If one or more blocks of an SCCS file have been trashed by a system crash, the SCCS commands will recognize this through an inconsistent checksum. In this case, the only SCCS command that will process the file is the `admin` command with the `-h` or `-z` keyletter:

```
   admin -h s.file1 s.file2 ...
```

It is a good idea to use these commands routinely to audit your SCCS files to detect any inconsistent checksums (indicating file corruptions). If the new checksum of any file is not equal to the checksum in the first line of that file, SCCS prints the message

```
   corrupted file (co6)
```

This process continues until all the files have been examined. The `admin -h` command can also be applied to directories:

```
   admin -h directory1 directory2 ...
```

This prints an error message for any corrupted files, but does not automatically report missing SCCS files. To determine whether any of your SCCS files are missing, list the contents of each directory (`ls`).

If you have an SCCS file that has been extensively corrupted, the best solution is to restore the file from your most recent backup copy. If there is only minor damage, you may be able to repair it using a text editor. In this case, after you have repaired the file, use the command

```
   admin -z s.file
```

This recomputes the file's checksum so that it agrees with the file contents. After you use `admin -z`, any corruption that existed in the file will no longer be detectable by the `admin -h` command.

3.6 Delta numbering

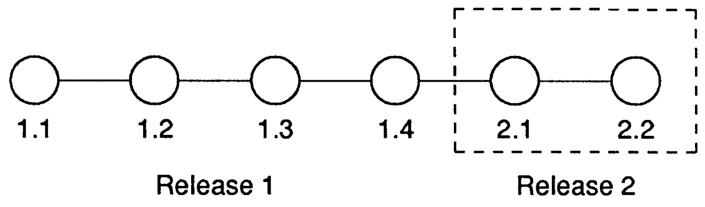
SCCS deltas are changes applied to an original (null) file to produce different versions and releases of your file.

SCCS names deltas with an SCCS Identification string (a SID). SIDs have exactly two components (the *release* number and the *level* number) separated by a period:

   release . level

SCCS names the initial delta 1.1. This is considered a set of changes applied to the null file. Subsequent deltas are named by incrementing the level number (1.2, 1.3, and so on) when the delta is created. If you make a major change to the file, you may want to specify a new release number when you create the new delta. In this case, SCCS assigns a new release number (2.1) and subsequent deltas are incremented as in release 1. This is shown in Figure 17-2.

Figure 17-2. A linear progression of versions



In this simplest case the deltas progress linearly; that is, any delta is dependent on all preceding deltas. When SCCS reconstructs a particular version of your SCCS file, it applies all deltas up to and including the number you specify. In most cases, this is all you will need to know about SCCS delta numbering.

3.6.1 Branch deltas

The linear progression of file versions shown above is sometimes called the "trunk" of the SCCS tree for that file. Under special conditions, you may need to use a "branch" in the tree: an independent progression of deltas that does *not* depend on all previous deltas for that file.

For example, suppose you have a program at version 1.3 that is being used in a production environment. You are developing a new release (release 2) of the program, and already have several deltas of that release. This situation uses the simple linear organization shown above.

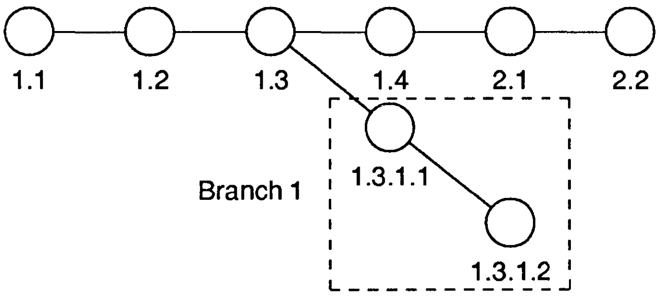
Now suppose that a user reports a problem in version 1.3 which requires changes only to version 1.3 but does not affect subsequent deltas. This requires a branch from the previous linear ordering. The new (branch) delta's name consists of exactly four components: release and level numbers (as in the trunk delta) plus a branch number and sequence number, all separated by periods:

release . level . branch . sequence

Thus, a branch delta can always be identified as such from its name.

Once you have created a branch delta, SCCS increments subsequent deltas on that branch by incrementing the sequence number. This is shown in Figure 17-3.

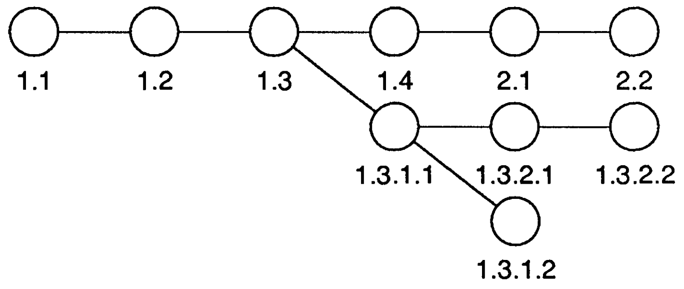
Figure 17-3. A branching SCCS tree



While SCCS increments the sequence number on each branch, it increments the branch number according to *when you create the branch.* If you need to complicate your SCCS branch structure, consider this carefully. While the trunk delta (the initial linear progression) can always be identified by the branch delta's name (by the release and level numbers), it is not possible to determine the entire path leading from the trunk delta to the particular branch delta you may have retrieved.

For example, if delta 1.3 has one branch, all deltas on that branch will be named 1.3.1.*n*. If a delta on this branch (for example, delta 1.3.1.1) has a branch, all deltas on the new branch will be named 1.3.2.*n*. This is shown in Figure 17-4.

Figure 17-4. A complicated branch structure



If you retrieve version 1.3.2.2, you know that (chronologically) it is the second delta on the second branch from delta 1.3. You are not able to deduce how many deltas there are between version 1.3.2.2 and version 1.3. Thus, although the branching capability has been provided for managing files under certain special conditions, it is much easier to manage your files if you keep the SCCS organization as linear and simple as possible.

4. SCCS command conventions

This section discusses the conventions and rules that apply to SCCS commands. Except where noted otherwise, these conventions apply to all SCCS commands. A list of the temporary files generated by various commands (and referred to in the "SCCS Command Summary") is also provided.

4.1 SCCS command arguments

SCCS commands accept two types of arguments: keyletters and file arguments.

Keyletters consist of a minus sign followed by a lowercase character, which may be followed by a value. For example, `-a` is a keyletter. Keyletters control the execution of the command to which they are supplied. All keyletters specified for a given command apply to all file arguments of that command. Keyletters are processed before any file arguments, with the result that the placement of keyletters is arbitrary (that is, keyletters may be interspersed with file arguments). Somewhat different argument conventions apply to the `help`, `what`, `sccsdiff`, and `val` commands.

> *Note:* Keyletters are command-line options equivalent to A/UX flag options. Do not confuse keyletters with SCCS flags, discussed in "SCCS Flags."

File arguments (names of files and/or directories) specify the file(s) to be processed by the given SCCS command. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files in the named directories are silently ignored. In general, file arguments may not begin with a minus sign, but if the name – (a single minus sign) is specified as an argument to a command, the command reads the standard input (until end-of-file) and takes each line as the name of an SCCS file to be processed. This feature is often used in pipelines. File arguments are processed left to right.

4.2 Flags

Certain actions of SCCS commands can be controlled by flags, which appear in SCCS files. These flags are discussed in "SCCS Flags."

4.3 Diagnostics

SCCS commands produce diagnostics (on the standard error output) that use this format:

`ERROR [`*filename*`] :` *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 stop processing that file and to proceed with the next file, in order, if more than one file has been named.

Certain SCCS commands check both the *real* and *effective* user IDs (see `passwd`(1) in *A/UX Command Reference*). If you are using SCCS to manage your personal files, these two IDs are the same; if you are working in a group project, see "SCCS Protection Mechanisms."

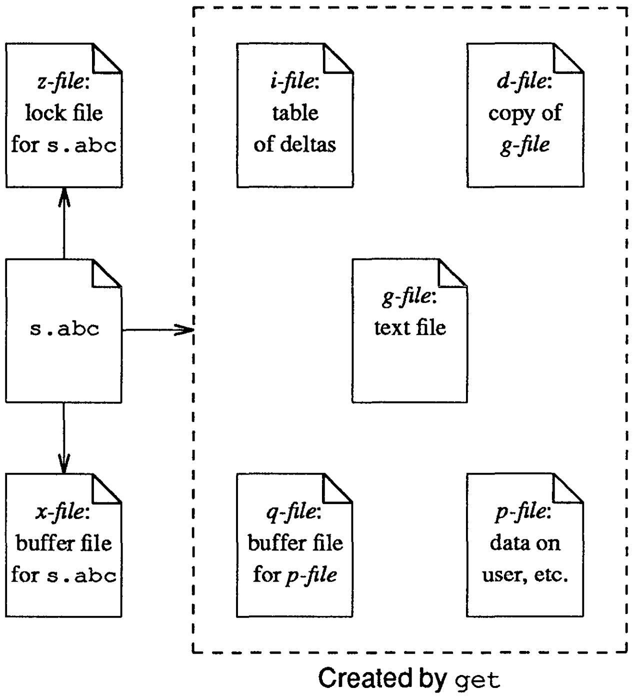
4.4 Temporary files

Several SCCS commands generate temporary files and file copies during the process of creating, retrieving, and updating SCCS files.

The temporary files are normally named by stripping off the `s.` prefix of the SCCS filename and replacing it with another single alphabetic character.

The *g-file* is named by simply deleting the `s.` prefix. Thus, if the SCCS file is named `s.abc` the *g-file* will be named `abc`. The *p-file* will be named `p.abc`.

Figure 17-5. Relationships among temporary files


z-file: lock file for s.abc
i-file: table of deltas
d-file: copy of *g-file*
s.abc
g-file: text file
x-file: buffer file for s.abc
q-file: buffer file for *p-file*
p-file: data on user, etc.
Created by get

These temporary files are as follows:

g-file This is the text file created by a `get` command. It contains a particular version of an SCCS file, and its name is formed by stripping off the `s.` prefix from the SCCS file.

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 version it contains also depends on how the `get` command is invoked. The default version is the most recent trunk delta (that is, excluding branches).

d-file When you invoke a `get` command, SCCS creates its own temporary copy of the *g-file* by performing an internal `get` at the SID specified in the *p-file* entry. This temporary copy is called the *d-file*.

When you record your changes in a new version, the `delta` command compares the *d-file* to to the *g-file* (using the `diff` command). The differences between the *g-file* and the *d-file* are the changes that constitute the delta.

p-file When the `get -e` command creates a *g-file* with read-write permission (so you can edit it), it places certain information about the SCCS file (that is, the SID of the retrieved version, the SID to be given to the new delta when it is created, and the login name of the user executing `get`) in another new file called the *p-file*.

When you record your changes in a new version, the `delta` command reads the *p-file* for the SID and the login name of the user creating the new delta.

When the new delta has been made, the *p-file* is updated by removing the relevant entry. If there is only a single entry in the *p-file*, then the *p-file* itself is removed.

q-file 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*.

x-file All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file* (to ensure that the SCCS file is not damaged if processing terminates abnormally). When processing is complete, the old SCCS file is removed and the *x-file* is renamed (with the `s.` prefix) to be the SCCS file.

The *x-file* is created in the directory containing the SCCS file, given the same mode as the SCCS file, and owned by the effective user.

z-file To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a "lock file" called the *z-file*. This file exists only for the duration of the execution of the command that creates it. The *z-file* contains the process number of the command that creates it. While the *z-file* exists, it indicates to other commands that the SCCS file is being updated. SCCS commands that modify SCCS files will not process a file if the corresponding *z-file* exists.

The *z-file* is created with read-only mode (mode 444, possibly modified by the user's `umask`) in the directory containing the SCCS file. It is owned by the effective user.

l-file The `get -l` command creates an *l-file* containing a table showing the deltas used in constructing a particular version of the SCCS file. This file is created in the current directory with mode 444 (read only) and is owned by the real user.

In general, users can ignore most of these temporary files, although they can be useful in the event of system crashes or similar situations.

4.5 SCCS ID keywords

When you retrieve an SCCS file to compile it, it is useful to record the date and time of creation, the version retrieved, the module's name, and so forth, within the *g-file*. This information appears in a load module when one is eventually created.

SCCS uses ID keywords for recording such information about deltas automatically. ID keywords can appear anywhere in the generated file and will be replaced by appropriate values.

The format of an ID keyword is an uppercase letter enclosed by percent signs (`%`). When these appear in the generated SCCS file they are replaced by the values defined for that keyword. For example,

```
%I%
```

is replaced by the SID of the retrieved version of a file. Similarly,

```
%H%
```

is replaced by the current date (in the form *mm/dd/yy*).

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`, unless the `i` flag is present in the SCCS file (see ''SCCS Flags'').

Here is a complete list of the ID keywords:

Table 17-1. SCCS ID Keywords

Keyword	**Value**
`%M%`	Module name: either the value of the `m` flag in the file (see `admin`(1)), or the name of the SCCS file with the leading `s.` removed.
`%I%`	SCCS identification (SID) (`%R%.%L%.%B%.%S%`) of the retrieved text.
`%R%`	Release.
`%L%`	Level.
`%B%`	Branch.
`%S%`	Sequence.
`%D%`	Current date (*yy/mm/dd*).
`%H%`	Current date (*mm/dd/yy*).
`%T%`	Current time (*hh:mm:ss*).
`%E%`	Date newest applied delta was created (*yy/mm/dd*).
`%G%`	Date newest applied delta was created (*mm/dd/yy*).
`%U%`	Time newest applied delta was created (*hh:mm:ss*).
`%Y%`	Module type: the value of the `t` flag in the SCCS file (see `admin`(1)).
`%F%`	SCCS file name.
`%P%`	Fully qualified SCCS filename.
`%Q%`	Value of the `q` flag in the file (see `admin`(1)).

`%C%`	Current line number. This keyword is intended for identifying messages sent by the program. It is not intended to be used on every line to provide sequence numbers.
`%Z%`	Four-character string `@(#)` recognizable by `what`.
`%W%`	Shorthand notation for constructing `what` strings for A/UX system program files. `%W%` = `%Z%%M%^I%I%` (where `^I` is the tab character).
`%A%`	Another shorthand notation for constructing `what` strings for non-A/UX system program files. `%A%` = `%Z%%Y%%M%%I%%Z%`

5. SCCS command summary

This section describes the features of all the SCCS commands. The SCCS commands are as follows:

`admin`	Creates SCCS files and applies changes to characteristics of SCCS files.
`cdc`	Changes the commentary associated with a delta.
`comb`	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
`delta`	Applies changes (deltas) to the text of SCCS files, that is, creates new versions.
`get`	Retrieves versions of SCCS files.
`unget`	"Undoes" a `get -e` command if invoked before the new delta is created.
`help`	Prints explanations of diagnostic messages.
`prs`	Prints portions of an SCCS file in user-specified format.
`rmdel`	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
`sact`	Accounts for SCCS files in the process of being changed.

`sccsdiff`	Shows the differences between any two versions of an SCCS file.
`val`	Validates an SCCS file.
`what`	Searches any A/UX system file(s) for all occurrences of a special pattern and prints out what follows it; `what` is useful in finding identifying information inserted by the `get` command.

5.1 Create SCCS files: `admin`

`admin` creates new SCCS files or changes characteristics of existing ones. You can create an SCCS file with the command

```
admin -ifilename s.filename
```

where *filename* is a file from which the text of the initial delta of the SCCS file `s.`*filename* is to be taken.

Note: There is no space between the `-i` keyletter and the *filename* argument.

SCCS files are created in read-only mode (444) and are owned by the effective user (see `passwd`(1) in *A/UX Command Reference*). Only a user with write permission in a directory containing SCCS files can use the `admin` command on a file in that directory.

If you omit the value of the `-i` keyletter, `admin` reads the standard input for the text of the initial delta. Thus, the command

```
admin -is.filename < filename
```

is also valid. Only one SCCS file can be created at a time using the `-i` keyletter.

If the text of the initial delta does not contain ID keywords, the message

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

is issued as a warning. See "SCCS ID Keywords" for more information.

If you set the `i` flag in the SCCS file (using the `-f` keyletter with the `admin` command; see "SCCS Flags"), the above message is treated as a fatal error and the SCCS file is not created.

The first delta of an SCCS file is normally 1.1. The `-r` keyletter to the `admin` command is used to specify a different release number for the initial delta. Because it is only meaningful in creating the first delta (with `admin`), its use is permitted only with the `-i` keyletter. The command

```
admin -ifilename -r3 s.filename
```

specifies that the first delta should be named 3.1 rather than 1.1.

5.1.1 SCCS flags

SCCS file flags are used to direct certain actions of SCCS commands.

The flags of an SCCS file are initialized or changed using the `-f` keyletter, and deleted using the `-d` keyletter. When you create an SCCS file, flags are either initialized by the `-f` keyletter on the command line or assigned default values.

For example, the following command sets the `i` flag and the `m` (module name) flag:

```
admin -ifilename -fi -fmmodname s.filename
```

The `i` flag specifies that a warning message stating that there are no ID keywords contained in the SCCS file should be treated as a fatal error.

The value *modname* specified for the `m` flag is the value that the `get` command will use to replace the `sccs` ID keyword. (In the absence of the `m` flag, the name of the *g-file* is used as the replacement for the `sccs` ID keyword.)

Note that several `-f` keyletters may be supplied on the `admin` command line 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 be specified only when processing an existing file. For example, the following command removes the `m` flag from the SCCS file:

```
admin -dm s.filename
```

Several `-d` keyletters may be supplied on a single invocation of `admin` and may be intermixed with `-f` keyletters.

A user list of login names and/or group IDs of users who are allowed to create deltas of that file is checked by several SCCS commands to ensure that the delta is authorized. This list is empty by default, which means that anyone may create deltas. The `-a` keyletter is used to specify users who are given permission or denied permission to create deltas. You can use the `-a` keyletter whether `admin` is creating a new SCCS file or processing an existing one, and it can appear several times on a command line.

For example, the command

```
admin -avz -aram -a1234 s.filename
```

gives permission to create deltas to the login names `vz` and `ram` and the group ID `1234`. The command

```
admin -a!vz s.filename
```

denies permission to create deltas to the login name `vz`. Similarly, the `-e` keyletter is used to remove (erase) login names or group IDs from the list. For example,

```
admin -evz s.filename
```

removes the login name `vz` from the user list of `s`.*filename*.

5.1.2 Comments and MR numbers

When an SCCS file is created, you may insert comments stating your reasons for creating the file. In a controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, and so forth, all of which are collectively called MRs (for "modification request").

The creation of an SCCS file may sometimes be the direct result of an MR. MRs can be recorded by number in a delta via the `-m` keyletter, which can be supplied on the `admin` (or `delta`) command line.

The `-y` keyletter can also be used to supply comments on the command line rather than through the standard input.

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 you want to supply an MR number (using the `-m` keyletter), the `v` flag must also be set (using the `-f` keyletter described below), as in the command

```
   admin -ifilename -mmrlist -fv s.filename
```

The `v` flag causes the `delta` command to prompt for MR numbers as the reason for creating a delta. (See `sccsfile`(4) in *A/UX Programmer's Reference*.) Note that the `-y` and `-m` keyletters are effective only if a new SCCS file is being created.

5.1.3 Descriptive text

The portion of the SCCS file reserved for descriptive text can be initialized or changed using the `-t` keyletter. Descriptive text is intended as a summary of the contents and purpose of the SCCS file.

To insert descriptive text in a file you are creating, the `-t` keyletter is followed by the name of a file from which the descriptive text is to be taken. For example, when a new SCCS file is being created, the following command takes descriptive text from *description-file:*

```
   admin -ifilename -tdescription-file s.filename
```

When processing an existing SCCS file, the `-t` keyletter specifies that text found in *description-file* should overwrite current descriptive text (if any). If you omit the file name after the `-t` keyletter, as in

```
   admin -t s.filename
```

the descriptive text currently in the SCCS file is removed.

5.2 Change comments in an SCCS file: `cdc`

`cdc` changes the comments or MR numbers that were supplied when a delta was created. It is invoked as follows:

```
   cdc -r3.4 s.filename
```

This specifies that you want to change the comments of delta 3.4 of `s`*.filename*. You can also use `cdc` to delete selected MR numbers by preceding the selected MR numbers by the exclamation character (`!`).

`cdc` prompts for MR numbers and new comments:

```
cdc -r3.4 s.filename
MRs? mrlist!mrlist
comments? deleted wrong MR number and inserted\
correct MR number
```

The new MR number(s) in the first `mrlist` are inserted, and the old MR number(s) (preceded by the exclamation character) are deleted. The old comments are kept and preceded by a line, indicating that they have been changed. The inserted comment line records the login name of the user executing `cdc` and the time of its execution.

5.3 Combine deltas to save space: `comb`

The `comb` command generates a shell script (see `sh`(1) in *A/UX Command Reference*) that is written to standard output. When executed, the script attempts to save space by discarding deltas that are no longer useful and combining other specified deltas.

> *Note:* `comb` should be used only a few times in the life of an SCCS file. Before any actual reconstructions, `comb` should be run with the `-s` keyletter (in addition to any other keyletters desired).

In the absence of any keyletters, `comb` preserves only the most recent deltas and the minimum number of ''ancestor'' deltas necessary to preserve the SCCS file tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree.

Some of `comb`'s keyletters are as follows:

`-p`	Specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.
`-c`	Specifies a list of deltas to be preserved (see `get`(1) in *A/UX Command Reference* for the syntax of this list). All other deltas are discarded.
`-s`	Causes the generation of a shell script that, when run, produces only a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. You should run

comb with this keyletter (in addition to any others desired) before any actual reconstructions.

Note that the shell script 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.

5.4 Store a new SCCS file version: `delta`

delta creates a new delta by recording the changes made to a *g-file*. The differences between the *g-file* and the *d-file* are the changes that constitute the delta. These changes are normally stored as a delta; they may also be printed on the standard output by using the -p keyletter. The format of this output is similar to that produced by diff.

5.4.1 Required temporary files

All temporary files used by the delta command are described in ‘‘Temporary Files.’’ There must be a *p-file* and a *d-file* for delta to work.

delta looks in the *p-file* for the user's login name and a valid SID for the next delta. There should be just one entry for the user (created when the user does a get -e) and it should be the same user who is trying to create a delta. Otherwise, delta will print an error message and stop. If the user's login name appears in more than one entry in the *p-file*, the same user has executed more than one get -e on the SCCS file. In this case the -r keyletter must then be used with delta to specify the SID that uniquely identifies the *p-file* entry. This entry is the one used to obtain the SID of the delta to be created.

The delta command also performs the same permission checks performed by get -e. If all checks are successful, delta performs a diff on the *g-file* and the *d-file* and records the changes as a new delta.

5.4.2 Comments and MR numbers

In practice, the most common use of delta is

```
delta s.filename
```

which prompts

```
comments?
```

on the screen. Your response can be up to 512 characters long if you escape all newlines with a backslash (\). The response is terminated by a newline character.

In a controlled environment, deltas are usually created only as a result of some trouble report, change request, trouble ticket, and the like. These are collectively called MRs (modification requests) and can be recorded in each delta. If the SCCS file has a `v` flag set, `delta` first prompts with

```
MRs?
```

on the screen. The standard input is then read for MR numbers, separated by blanks and/or tabs. Your response can be up to 512 characters long if you escape all newlines with a backslash (\). The response is terminated by a newline character.

The `-y` and/or `-m` keyletters on the `delta` command line can also be used to supply comments and MR numbers, respectively, instead of supplying these through the standard input. The format of the `delta` command is then

`delta -y`*descriptive comment* `-m`*mrlist* `s.`*filename*

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 script (see `sh(1)` in *A/UX Command Reference*).

The `-s` keyletter suppresses all output that is normally directed to the standard output except for the prompts `comments?` and `MRs?`. Use of the `-s` keyletter together with the `-y` keyletter (and possibly the `-m` keyletter) causes `delta` to neither read standard input nor write to standard output.

The comments and/or MR numbers are recorded as part of the entry for the delta being created and apply to all SCCS files processed by the same invocation of `delta`. If `delta` is invoked with more than one file argument and the first file named has a `v` flag, all files named must have the `v` 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, 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 are written to the standard output.

Thus, a typical output might be

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

Note: The counts of lines reported as inserted, deleted, or unchanged by `delta` may not agree with your perception of the changes applied to the *g-file*. There are usually several ways to describe a set of changes, especially if lines are moved around in the *g-file*, and `delta` is likely to find a description that differs from your 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*.

5.4.3 Keywords

If `delta` finds no ID keywords in the edited *g-file*, it prints the message

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

after it prompts for comments, 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` command without the `-e` keyletter (ID keywords are replaced by `get` in that case)
- Accidentally deleting or changing the ID keywords while you are editing the *g-file*
- The file's having no ID keywords to begin with

In any case, it is left up to the user to determine what to do about it. The delta is created whether or not ID keywords are present, 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 until the ID keywords are inserted in the *g-file* and the `delta` command is executed again.

See "SCCS ID Keywords" for more information.

5.4.4 Removal of temporary files

When 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 called the *q-file*. If there is only one entry in the *p-file*, then the *p-file* itself is removed.

When processing of the corresponding SCCS file is complete, `delta` also removes the edited *g-file* unless the `-n` keyletter is specified. The command

```
delta -n s.filename
```

keeps the *g-file* upon completion of processing.

5.5 Retrieve an SCCS file version: get

`get` creates a text file containing a particular version of an SCCS file. The `get` command applies deltas to the initial version of the file to obtain the version you specify or the most recent version (excluding branch versions, which must be retrieved specifically).

The resulting text file is called the *g-file* (see "Temporary Files"). The mode of the *g-file* depends on how the `get` command is invoked.

For example, the command

```
get s.filename
```

produces

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

on the standard output. This indicates that version 1.3 (the most recent delta) was retrieved, that there are 67 lines of text in this version, and that no ID keywords were substituted in the file.

The generated *g-file* is assigned mode 444 (read only), which does not allow you to modify the file, although you can read the file or compile it, and so on. The file is not intended for editing (that is, for making deltas).

If you specify several file arguments (or directory-name arguments) on the `get` command line, similar information is displayed for each file processed, preceded by the SCCS filename. For example, the command

```
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)
```

See "SCCS ID Keywords."

5.5.1 Retrieving different versions

By default the `get` command retrieves the most recent delta of the highest-numbered release on the basic trunk of the SCCS file tree (exclusive of branches). To change this default, you can

- Set the `d` flag in the SCCS file. Then the SID specified as the value of this flag is used as a default.
- Use the `-r` keyletter on the `get` command line to specify which SID you want to retrieve. (If the version you specify does not exist, an error message results.) For example,

  ```
  get -r1.3 s.filename
  ```

  In this case, the `d` flag (if any) is ignored. A branch delta can be retrieved similarly:

```
get -r1.5.2.3 s.filename
```

If you omit the level number

```
get -r3 s.filename
```

the highest level number (most recent delta) within the given release will be retrieved. If the given release does not exist, `get` retrieves the most recent trunk delta (not in a branch) with the highest level number within the highest-numbered existing release that is lower than the release you specify.

- Use the `-t` keyletter to retrieve the most recent (top) version in a particular release (when no `-r` keyletter is supplied or when its value is simply a release number). *Most recent* is independent of location in the SCCS tree (see "Delta Numbering"). For example, if the most recent delta in release 3 is 3.5,

  ```
  get -r3 -t s.filename
  ```

  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
  ```

5.5.2 Retrieving a file to create a new delta

When you specify the `-e` keyletter to `get`, the retrieved file has read-write permission and can be edited to make a new delta. For example, the command

```
get -e s.filename
```

produces

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

on the standard output. The use of `get -e` is restricted (because a new delta can be created), causing a check of the SCCS protection mechanisms (user list and protection flags; see "SCCS Protection Mechanisms"). SCCS also checks for permission to make concurrent edits (specified by the `j` flag in the SCCS file; see "Concurrent Edits of Same SID").

If the permission checks succeed, `get -e` creates a *g-file* with mode 644 (readable by everyone, writable only by the owner) in the current directory. This mode may be modified by the user's `umask`.

If a writable *g-file* already exists, `get -e` 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.

ID keywords appearing in the *g-file* are not substituted by `get -e` because the generated *g-file* is subsequently used to create another delta. Replacement of ID keywords causes them to be permanently changed within the SCCS file.

The following keyletters may be used with `get -e`:

`-r` Used to specify a particular version to be retrieved for editing. If the number specified to `-r` does not exist, it will be assigned to the new delta.

`-t` Specifies that the most recent version in a given release be retrieved for editing.

`-i` Used to specify a list of deltas to be included 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 you want to apply the same changes to more than one version of the SCCS file. When a delta is included, `get` checks for possible interference between those deltas and 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 do whatever is necessary (for example, edit the file). The `-i` keyletter should be used with extreme care.

-x Used to specify a list of deltas to be excluded by `get`. Excluding a delta means forcing it not to be 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 excluded, `get` checks for possible interference between those deltas and deltas that are normally used in retrieving the particular version of the SCCS file. (See the explanation under `-i`.) The `-x` keyletter should be used with extreme care.

-k Facilitates regeneration of a *g-file* that may have been accidentally removed or ruined after a `get -e` command, or the simple generation of a *g-file* in which the replacement of ID keywords has been suppressed. A *g-file* generated by the `-k` keyletter is identical to one produced by `get -e` except that no processing related to the *p-file* takes place (see "Temporary Files").

5.5.3 Concurrent edits of different versions

There is a possibility (in a group project) that several `get -e` commands may be executed at the same time on the same file. However, unless concurrent edits are explicitly allowed (see "Concurrent Edits of Same SID"), no two `get -e` executions can retrieve the same version of an SCCS file. This protection uses information from the *p-file* (see "Temporary Files").

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 to ensure that no entry (already in the *p-file*) specifies that the SID (of the version to be retrieved) is already retrieved, unless multiple concurrent edits are allowed. (See "Concurrent Edits of Same SID.")

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 succeeds because 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 (each user normally has a different working directory). (See the section "SCCS Protection Mechanisms" for a discussion about how different users are permitted to use SCCS commands on the same files.)

Figure 17-6 shows a sample SCCS file retrieved by `get -e` and the SID of the version that will subsequently be created by `delta`, as a function of the SID specified to `get`.

In Figure 17-6, R, L, B, and S are release, level, branch, and sequence components of the SID. 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 (maximum branch number plus 1) of level L within release R."

Also 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 is present in the file (see `admin`(1)). In this state, an entry of `-i` means "irrelevant."

The cases marked * apply 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 figure applies.

The case marked with a † is used to force the creation of the first delta in the new release.

hR is the highest existing release that is lower than the specified, nonexisting, release R.

Figure 17-6. Determination of new SID

SID specified	-b keyletter used	Other conditions	SID retrieved	SID of delta to be created
none*	no	R default to mR	mR.mL	mR.(mL+1)
none*	yes	R default to mR	mR.mL	mR.mL.(mB+1)
R R R R	no no yes yes	R > mR R == mR R > mR R == mR	mR.mL mR.mL mR.mL mR.mL	R.1† mR.(mL+1) mR.mL.(mB+1).1 mR.mL.(mB+1).1
R	-	R< mR and does not exist	hR.mL	hR.mL.(mB+1).1
R	-	Trunk successor in release >R and R exists	R.mL	R.mL.(mB+1).1

Figure 17-6. Determination of new SID (continued)

SID specified	-b keyletter used	Other conditions	SID retrieved	SID of delta to be created
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	-	Trunk in release >= R	R.L	R.L.(mB+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	no	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	-	Branch successor	R.L.B.S	R.L.(mB+1).1

5.5.4 Concurrent edits of same SID

Unless the `j` flag is set in the SCCS file (see "SCCS Flags"), `get -e` commands are not permitted to occur concurrently on the same SID. That is, `delta` must be executed before another `get -e` is executed on the same SID. If the `j` flag is set in the SCCS file, two or more successive executions of `get -e` on the same SID are allowed. The command

```
admin -fj s.filename
```

sets the `j` flag. Then, the command

```
get -e s.filename
```

may produce

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

which may be immediately followed by the commands

```
mv filename new-filename
get -e s.filename
```

The second edit request without an intervening execution of `delta` causes a warning to be generated:

```
1.1
WARNING: being edited: '1.1 1.2 username date-stamp' (ge18)
new delta 1.1.1.1
5 lines
```

In this case, a `delta` command corresponding to the first `get` produces delta 1.2 (assuming 1.1 is the latest (most recent) delta), and the `delta` command corresponding to the second `get` produces delta 1.1.1.1.

5.5.5 Keyletters that affect output

The following keyletters affect output:

`-p` The retrieved text is written on standard output rather than on a *g-file*. In this case, 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 standard error output. The `-p` keyletter is used, for example, to create *g-files* with arbitrary names:

```
get -p s.filename >filename
```

`-s` Suppresses all output that is normally directed to the standard output (the SID of the retrieved version, the number of lines retrieved, and so forth, are not written). This does not affect messages to the standard error 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`. For example,

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

`-g` Suppresses the actual retrieval of the text of a version of the SCCS file. This can be used in a number of ways, for example, to verify the existence of a particular SID in an SCCS file:

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

This prints the given SID if it exists in the SCCS file or generates an error message if it does not exist. The `-g` keyletter is also used to regenerate a *p-file* that has been accidentally destroyed. For example,

```
get -e -g s.filename
```

`-l` Creates an *l-file* named by replacing the `s.` of the SCCS file name with `l.`. See "Temporary Files." For example, the command

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

generates an *l-file* that shows the deltas applied to retrieve version 2.3 of the SCCS file. Specifying a value of `p` with the `-l` keyletter

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

causes the generated output to be written to the standard output rather than to the *l-file*. You can use the `-g` keyletter with the `-l` keyletter to suppress the actual retrieval of the text.

`-m` Identifies the changes applied to an SCCS file, line by line. When you specify this keyletter to the `get` command, each line of the generated *g-file* is 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.

`-n` Causes each line of the generated *g-file* to be preceded by the value of the `%M%` ID keyword (the module name) and a tab character. The `-n` keyletter is most often used in a pipeline with the `grep` command. For example,

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

searches the latest version of each SCCS file in a directory for all lines that match a given pattern. If both the `-m` and `-n` keyletters are specified, each line of the generated *g-file* is preceded by the value of the `sccs` ID keyword and a tab (caused by the `-n` keyletter) and shown in the format produced by the `-m` keyletter.

Because the contents of the *g-file* are modified when you use the `-m` and/or `-n` keyletters, this *g-file* cannot be used for creating a delta, and neither `-m` nor `-n` can be used with the `-e` keyletter.

5.6 Restore a version unchanged: `unget`

If invoked before a `delta`, `unget` undoes a `get -e` command. The following keyletters can be used with `unget`:

`-r`*SID* Uniquely identifies the delta that is no longer intended (the SID for the new delta is included in the *p-file*). This is necessary only if two or more `get -e` commands of the same SCCS file are in progress.

`-s` Suppresses the display of the intended SID of the delta on standard output.

`-n` Retains the *g-file* in the current directory instead of removing it.

For example, the command

```
get -e s.filename
```

followed by

```
unget s.filename
```

causes the last version to be unchanged.

5.7 On-line explanations: `help`

The `help` command prints explanations of SCCS commands and the messages printed by some of these commands. If you use `help` without an argument, it prompts for one. Valid arguments are names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. Keyletter arguments or file arguments are not valid arguments to `help`.

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

```
help ge5 rmdel
```

produces

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

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

This is printed on standard output by default. If no information is found, `help` prints an error message. Note that `help` processes each argument independently, and an error resulting from one argument will not terminate the processing of the other arguments on the command line.

5.8 Print part(s) of an SCCS file: `prs`

The `prs` command is used to print on the standard output all or part(s) of an SCCS file in a format you specify. The format is called the output "data specification." It is a string consisting of SCCS file data keywords (not to be confused with `get` ID keywords), supplied using the `-d` keyletter on the `prs` command line. These keywords can (optionally) be interspersed with text.

Data keywords specify which parts of an SCCS file are to be retrieved and produced. All parts of an SCCS file (see `sccsfile`(4)) have an associated data keyword. Data keywords are an uppercase character, two uppercase characters, or an uppercase and a lowercase character, enclosed by colons. For example,

```
:I:
```

is the keyword replaced by the SID of a specified delta. Similarly,

```
:F:
```

is the keyword replaced by the SCCS filename currently being processed, and

```
:C:
```

is replaced by the comment line associated with a specified delta. For a complete list of the data keywords, see `prs`(1) in *A/UX Command Reference*.

There is no limit to the number of times a data keyword can appear in a data specification. For example, the command

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

may produce on the standard output (for example)

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

Information can 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.filename
```

may produce the following output:

```
s.filename: 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.

Information may be obtained from a range of deltas by specifying the `-e` or `-l` keyletters.

The `-e` keyletter substitutes data keywords for the SID designated by the `-r` keyletter and all earlier deltas.

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

may produce

```
1.4
1.3
1.2.1.1
1.2
1.1
```

The `-l` keyletter substitutes data keywords for the SID designated by the `-r` keyletter and all later deltas.

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

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.

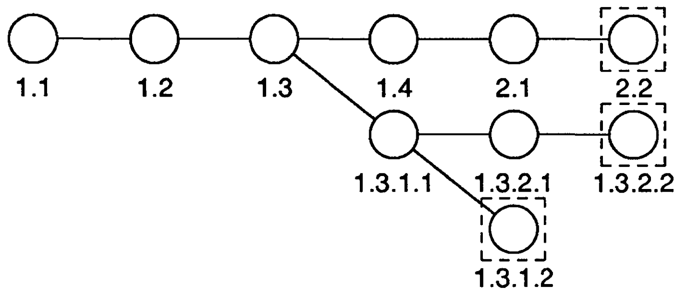
5.9 Remove a specific delta: `rmdel -r`

`rmdel` removes a delta from an SCCS file. Normally, you should use it only if incorrect global changes were incorporated in a delta.

The `-r` keyletter is required to specify the complete SID of the delta to be removed.

The delta to be removed must be the most recent delta on its branch or on the trunk of the SCCS file tree. In Figure 17-7, 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.

Figure 17-7. Removing a delta



The command

```
rmdel -r2.2 s.filename
```

specifies that delta 2.2 of the SCCS file should be removed. Before removing it, `rmdel` checks that the release number (R) of the given SID satisfies the relation

floor <= *R* <= *ceiling*

and that the SID specified is not a version that is being changed (for which a `get -e` has been executed and whose associated `delta` has not yet been made).

The A/UX and SCCS protection mechanisms are also checked (see appropriate sections above). If the checks are not successful, processing is terminated and the delta is not removed.

If the checks are successful, the delta is removed and its type indicator in the delta table of the SCCS file is changed from `D` (''delta'') to `R` (''removed'').

5.10 Account for open SCCS files: `sact`

The `sact` command reports any impending deltas to an SCCS file. An impending delta is a change that has not yet been incorporated into the SCCS file with the `delta` command. This would occur if a `get -e` has been executed but an associated `delta` has not yet been made.

`sact` reports five fields for each named file:

field 1	The SID of the existing SCCS file being changed
field 2	The SID of the new delta to be created
field 3	The login name of the user who executed the `get -e` command
field 4	The date the `get -e` command was executed
field 5	The time the `get -e` command was executed

The command

```
   sact s.filename
```

produces a display such as

```
   1.2 1.3 john 85/06/20 16:15:15
```

5.11 Compare two SCCS files: `sccsdiff`

`sccsdiff` compares two specified versions of one or more SCCS files and prints the differences on standard output. The versions to be compared are specified using the `-r` keyletter in the same format used for the `get` command. For example,

```
   sccsdiff -r3.4 -r5.6 s.filename
```

The two versions must be specified as the first two arguments to this command in the order in which they were created (the older version is specified first). Any following keyletters are interpreted as arguments to the `pr` command (which prints the differences on standard output in `diff` format) and must appear before any filenames.

The SCCS files to be processed are named last. Directory names and a name of a single minus sign (-) are not acceptable to `sccsdiff`.

5.12 Check an SCCS file's characteristics: `val`

`val` 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 *A/UX Command Reference* for a description of the flags).

The val command treats the special argument - differently than 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 an 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.filename
-mxyz -ypl1 s.xyz
(EOF)
```

first checks if the s.*filename* file has a value c for its type flag and value *filename* 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). The appropriate diagnostic is also printed unless suppressed by the -s keyletter. A return code of 0 indicates all named files met the characteristics specified.

5.13 Find identifying information: **what**

what is used to find identifying information within any A/UX system file whose name is given as an argument to what. Directory names and a name of - (a single 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 @(#) ID keyword) and prints (on the standard output) the balance following that string until the first double quote ("), greater than (>), backslash (\), newline, or (nonprinting) null character. For example, if the SCCS file s.prog.c (a C language program) contains the following line:

```
char id[] = "@(#)%Z%%M%:%I%";
```

The command

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

is executed, and 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` does not need to be inserted in the SCCS file via an ID keyword of `get`; it can be inserted in any convenient way.

Chapter 18
awk Reference

Contents

Tables

Chapter 18

awk Reference

1. awk: a programming language

The awk programming language is a file-processing language designed to make many common information retrieval and manipulation tasks easy to state and perform. The awk language can be used to

- Generate reports
- Match patterns
- Validate data
- Filter data for transmission

2. Program structure

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

```
BEGIN   { action }
pattern { action }
          .
          .
          .
END     { action }
```

The awk program is run on a set of input files. The basic operation of awk is to scan a set of input lines, in order, one at a time. In each line, awk searches for the *pattern* described; if that *pattern* is found in the input line, the corresponding *action* is performed. An **action** is a sequence of action statements separated by newlines or semicolons. A **pattern** in front of an action acts as a selector that determines whether the action executes. When all the patterns are tested, the next input line is fetched and the awk program is once again executed from the beginning.

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

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

prints every input line that has the letter x in it, as will

```
/x/
```

The patterns recognized by awk, such as the /x/ just above, include regular expressions recognized by other A/UX utilities such as egrep or vi. There are also two special patterns, BEGIN and END. The BEGIN section is run before any input lines are read, and the END section is run after all the data files are processed.

The action may be quite simple or very complex: awk provides conditional execution of statements and full flow-control constructions. Variables may be created and assigned values in any of the three sections of an awk program; values may also be assigned from the awk command line, although the BEGIN section is run before these assignments are made.

3. Invoking awk

There are three ways in which to present an awk program to awk for processing:

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

   awk '*program*' *files*

   where *files* is an optional list of input files and *program* is your awk program. Note that there are single quotes around the program name to make the shell accept the entire string (*program*) as the first argument to awk. For example, you may write the following to the shell:

   ```
   awk '/x/' chap.1
   ```

   to run the awk program /x/ on the input file chap.1. If no input files are specified, awk takes input from the standard input stdin. You can also specify that input comes from stdin by using the hyphen (–) as one of the files. The command

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

looks for input first from *files* and then from `stdin`.

2. Alternatively, if your `awk` program is long, it is more convenient to put the program into a separate file, say `awkprog`, and then to tell `awk` to fetch it from there. This is done by using the `-f` option with the `awk` command, as follows:

   ```
   awk -f awkprog files
   ```

   where *files* is an optional list of input files that may include `stdin` as indicated by a hyphen (–). For example, suppose that you put the following text into a file called `awkprog`:

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

   Then you may give the command

   ```
   awk -f awkprog
   ```

   to the shell. This yields

   ```
   hello, world
   ```

   on the standard output. Recall that the word `BEGIN` is a special pattern indicating that the action following in braces is run before any data is read. `print` and `exit` are both discussed in later sections, but their effects here are obvious.

3. Finally, the `awk` program may be put into a file together with the `awk` invocation for use as a shell script. This is most useful if the input to `awk` needs to be processed first by other A/UX utilities such as `sort` or `m4`; but a single `awk` statement will also work:

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

If this text is put into a file, for example `greet`, and made executable, then typing `greet` to the shell has the same output as the previous example.

This method also allows passing command line arguments to `awk`. If the following is put instead in the file called `greet`,

```
awk ' BEGIN {
                print "hello, '$1' "
                exit
              } '
```

you can invoke it from the shell with the command line

```
greet Jim
```

and the output will be

```
hello, Jim
```

4. Input: records and fields

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

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

As an example, suppose that the file `countries` contains the area in thousands of square miles, the population in millions, and the continent for the ten largest countries in the world. (Figures are from 1978; Russia is placed in Asia.) The sample input file `countries` looks like this:

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

The wide spaces are tabs in the original input, while a single blank separates `N.` and `S.` from `America`. This sample file will be used as the input for many of the `awk` programs in this guide because it is typical of the kind of material that `awk` is best at processing (a mixture of words and numbers separated into fields or columns separated by blanks and tabs).

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

```
Russia   8650     262      Asia
```

When this record is read by `awk`, it is stored in the variable `$0`. If you want to refer to this entire record, you do so through the variable `$0`. For example, the following action

```
{ print $0 }
```

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

$1 is equal to the string Russia, $2 is equal to the integer 8650, and so on.

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

```
{ print $4, $5, $1, $3 }
```

Using $4 and $5 allows for those countries whose names consist of two fields (N. America, for example). Note that awk does not require type declarations.

5. Input from the command line

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

```
x=5
```

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

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

will print the value 5 on the standard output once for each input line. It will terminate only when an end-of-file is received. If input from the keyboard must be given, the minus sign at the end of this command is necessary to indicate that input is coming from standard input. Similarly, if the input comes from a file named chap.1, the command is

```
awk '{ print x }' x=5 chap.1
```

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

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

```
awk -f awk.program RS=":" chap.1
```

Here, the record separator is changed to the colon (`:`). This causes your program in the file `awk.program` to run with records separated by the colon instead of the newline character. Also, its input comes from the file `chap.1`.

It is similarly useful to change the field separator from the command line. This operation is so common that there is another way to do it. There is a separate option `-F`*c* that is placed directly after the command `awk`. This changes the field separator from blank or tab to the character *c*. For example,

```
   awk -F: -f awk.program chap.1
```

changes the field separator `FS` to the colon. Note that if the field separator is specifically set to a tab (that is, with the `-F` option or by making a direct assignment to `FS`) then blanks are not recognized as separating fields. However, even if the field separator is specifically set to a blank, tabs are still recognized as separating fields. Certain characters must be quoted to protect them from interpretation by the shell (for example, blank, tab, asterisk, and so forth).

As an exercise, using the input file `countries` described earlier, write an `awk` program that prints the name of a country followed by the continent that it is on. Do this in such a way that continent names composed of two words (for example, `N. America`) are processed as only one field and not two.

6. Output: printing

6.1 `print`

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

```
   { print }
```

This is one of the simplest actions performed by `awk`. It prints each line of the input to the output. Generally, it is more useful to print one or more fields from each line. For instance, using the file `countries`, the command line

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

prints the name of the country and the population:

```
Russia 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 26
Sudan 19
Algeria 18
```

If you want to specify two `awk` actions for the same pattern, a semicolon must separate each action. For example, if you want the number 5 printed, you might give the following program:

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

or even

```
{ print 5 }
```

Note that the use of a semicolon after the final action statement in an `awk` program line is optional. `awk` accepts

```
{ print $1 }
```

and

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

equally and takes them to mean the same thing. Parentheses are also optional with the `print` statement.

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

is the same as

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

Items separated by a comma in a `print` statement are separated by the current output field separator (normally a space, even when the input is separated by tabs) when printed. The output field separator (`OFS`) is another special variable that you can change. These special variables are summarized in a later section.

As an exercise, using the input file countries, print the continent followed by the country, followed by the population for each input record. Then pipe the output to the A/UX command sort so that all countries from a given continent are printed together.

print also prints strings directly from your programs. Recall the awk program

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

from an earlier section.

As an exercise, print a header to the output of the previous exercise that says "Population of Largest Countries" followed by a header to each column that follows describing what is in that column; for example, "Country" or "Population" (see "BEGIN and END" for more information).

6.2 NR and NF

As you have already seen, awk makes available a number of special variables with useful values, for example, FS and RS. Two more special variables are introduced in the next example. NR and NF are both integers that contain the number of the present record and the number of fields in the present record, respectively. Thus,

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

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

```
1 4 Russia      8650    262     Asia
2 5 Canada      3852    24      N. America
3 4 China       3692    866     Asia
4 5 USA 3615    219     N. America
5 5 Brazil      3286    116     S. America
6 4 Australia   2986    14      Australia
7 4 India       1269    637     Asia
8 5 Argentina   1072    26      S. America
9 4 Sudan       968     19      Africa
10 4 Algeria    920     18      Africa
```

And the program

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

yields

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

This is an easy way to supply sequence numbers to a list. By itself, `print` prints the entire input record. Use

```
print ""
```

to print an empty line.

6.3 printf

`awk` also provides the statement `printf` so that you can format output as desired. (`print` uses the default format `%.6g` for each variable printed.)

printf *format*, *expr*, *expr*, ...

formats the expressions in the list according to the specification in the string *format*, and prints them. The *format* specifier is exactly like that used with `printf` in the C library (with the following exceptions: the format specifiers `*`, `u`, `X`, `E`, and `G` are not supported in `awk`). For example,

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

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

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

With `printf`, no output separators or newlines are produced automatically. You must add them, as in this example. As in the C library version of `printf`, the escape characters `\n` (newline) and `\t` (tab) are valid with the `awk` `printf`.

6.4 OFS and ORS

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

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

which prints

```
hello world
```

However, without the comma in the `print` statement, as in

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

you get

```
helloworld
```

White space is the `awk` concatenation operator, so `print` recognizes only one argument here.

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

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

or you can change OFS in a BEGIN section as in

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

Both of these last two programs yield

```
hello, world
```

Wherever a comma appears in a print statement, it is replaced by the output field separator. If you have not defined the output field separator, the default (a space) is used. If there are no commas, as in

```
{ print $0 $1}
```

no output field separator is used. However, in

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

the output field separator is used between the two $0s.

7. Output to different files

The A/UX shell programs allow you to redirect the standard output to a file. awk also lets you direct output to many different files from within your awk program. For example, with the input file countries, you might want to print all the data from countries of Asia in a file called ASIA, all the data from countries in Africa in a file called AFRICA, and so forth. This is done with the following awk program:

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

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

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

print > "*file*"

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

Notice that the filenames are quoted strings or variables containing strings. Without quotes, the filenames are treated as uninitialized variables and all output then goes to the same file. Also, if the redirection symbol > is replaced by >>, output is appended to the file rather than overwriting it.

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

8. Output to pipes

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

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

(where `harry` is someone's login name), any record with the second field equal to `XX` is sent to the user `harry` as mail. But instead of passing each such record across the pipe to `mail` individually, `awk` waits until the entire `print` input is processed before passing its output on to `mail`. Also,

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

takes the first field of each input record, accumulates them until the input to `print` is exhausted, and then passes the entire list to `sort`, which then generates the sorted list. The command in double quotes may be any A/UX command.

As an exercise, write an `awk` program that uses the input file `countries` to

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

Here is another example of using a pipe for output, which guarantees that its output always goes to your terminal:

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

`cat -u` allows output to be displayed as soon as it is produced (that is, it is unbuffered). Otherwise, you have to wait for a bufferful of data before you see anything.

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

9. Comments

Comments may be placed in `awk` programs; they begin with the character # and end with the end of the line, as in

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

10. Patterns

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

- The special patterns `BEGIN` and `END`
- Regular expressions
- Arithmetic relational expressions
- String-valued expressions
- Combinations of these

10.1 `BEGIN` and `END`

The action corresponding to the special pattern `BEGIN` is executed before the input is read. The action corresponding to the special pattern `END` is executed after all the input has been processed. `BEGIN` and `END` thus provide a way to gain control before processing for initialization and after processing for wrapping up.

You can use `BEGIN` to put column headings on the output. For example, if you put the following `awk` program in the file `awkprog`,

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

and invoked awk with the command line

```
awk -f awkprog countries
```

awk would produce

```
Country Area Population Continent
Russia  8650    262     Asia
Canada  3852    24      N. America
China   3692    866     Asia
USA     3615    219     N. America
Brazil  3286    116     S. America
Australia       2986    14      Australia
India   1269    637     Asia
Angentina       1072    26      South Africa
Sudan   968     19      Africa
Algeria 920     18      Africa
```

Formatting is obviously not very good here; `printf` would do a better job and is usually mandatory if you really care about appearance (see `printf`(3S) in *A/UX Programmer's Reference*).

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

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

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

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

10.2 Relational expressions

Any expression involving comparisons between strings of characters or numbers can be an awk pattern. For example, if you want to print only countries with more than 100 million population, use

```
$3 > 100
```

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

```
Russia  8650    262     Asia
China   3692    866     Asia
USA     3615    219     N. America
Brazil  3286    116     S. America
India   1269    637     Asia
```

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

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

which produces

```
Russia
China
India
```

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

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

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

```
USA     3615    219     N. America
Sudan   968     19      Africa
```

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

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

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

```
Australia    2968    14    Australia
```

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

10.3 Regular expressions

awk provides more powerful capabilities for searching for character strings than were illustrated in the previous section. These additional search capabilities make use of regular expressions. The simplest regular expression is a literal string of characters enclosed in slashes.

```
/Asia/
```

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

awk regular expressions include regular expression forms like those found in the text editor ed and the pattern finder egrep, in which certain characters have special meanings. For example, you could print all lines that begin with A using

```
/^A/
```

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

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

or all lines that end with ia using

```
/ia$/
```

The circumflex (^) means "match the beginning of a line." The dollar sign ($) means "match the end of the line," and enclosing characters in brackets ([and]) means "match any of the characters enclosed." In addition, awk allows parentheses for grouping, the vertical bar (|) for alternatives, the plus sign (+) for "one or more" occurrences, and the question mark (?) for "zero or one" occurrences. For example,

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

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

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

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

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

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

The two characters . and * have the same meaning as they have in ed or grep: namely, . matches any character and * matches zero or more occurrences of the character preceding it. For example,

```
/a.b/
```

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

```
/ab*c/
```

matches a record that contains an a followed by zero or more b's followed by a c. For example, it matches ac, abc, and pqrabbbbbbbbbbc901.

It is possible to turn off the special meaning of metacharacters such as ^ and * by preceding these characters with a backslash. An example of this is the pattern

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

which matches any string of characters enclosed in slashes.

You can also specify that any field or variable must match a regular expression, or must not match it, by using the operators ~ or !~, respectively. For example, with the input file countries as before, the program

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

prints all countries whose names end in ia:

```
Russia
Australia
India
Algeria
```

10.4 Combinations of patterns

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

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

selects lines where both area and population are large:

```
Russia  8650    262     Asia
China   3692    866     Asia
USA     3615    219     N. America
Brazil  3286    116     S. America
```

The program

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

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

```
$4 ~ /(Asia|Africa)/
```

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

10.5 Pattern ranges

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

pattern1 , *pattern2*   { *action* }

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

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

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

```
Canada  3852    24      N. America
China   3692    866     Asia
USA     3615    219     N. America
Brazil  3286    116     S. America
```

while

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

does the *action* for lines 2 through 5 of the input. Different types of patterns may be mixed, as in

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

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

Note that patterns in this form occur *outside* the action parts of the `awk` programs (outside the braces that define `awk` actions). If you need to check patterns *inside* an `awk` action (inside the braces), use a flow-of-control statement such as an `if` statement or a `while` statement. Flow-of-control statements are discussed in the section ''Built-in Functions.''

11. Actions

An `awk` action is a sequence of action statements separated by newlines or semicolons. These action statements do a variety of bookkeeping, arithmetic, and string-manipulation tasks.

11.1 Variables, expressions, and assignments

`awk` provides the ability to do arithmetic and to store the results in variables for later use in the program. As an example, consider printing the population density for each country in the file `countries`.

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

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

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

The formatting is pretty bad; using `printf` instead gives the program

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

and the output

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

Here the output format is much more readable.

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

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

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

which produces

```
   total population of 3 Asian countries is 1765
```

Actually, no experienced C programmer would write

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

because both assignments can be written more concisely. A better way is

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

Indeed, the operators ++, --, -=, /=, *=, +=, and %= are available in awk as they are in C. The statement

```
   x += y
```

has the same effect as

```
   x = x + y
```

but += is shorter and runs slightly faster. The same is true of the ++ operator; it adds one to the value of a variable. The increment and decrement operators ++ and -- (as in C) may be used as prefix or as postfix operators. These operators are also used in expressions.

11.2 Initialization of variables

In the previous example, neither pop nor n was initialized, yet everything worked properly. This is because (by default) variables are initialized to the null string, which has a numeric value of 0. This eliminates the need for most initialization of variables in BEGIN sections. You can use default initialization to advantage in the following program, which finds the country with the largest population:

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

which produces

```
   China 866
```

11.3 Field variables

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

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

or process two fields into a third with

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

or assign strings to a field, as in

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

which replaces `USA` by `United States` and prints the affected line:

```
United States 3615 219 N. America
```

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

11.4 String concatenation

Variables can also store strings of characters. You cannot do arithmetic on character strings, but you can join them. Strings are concatenated by writing them one after the other, as in the following example:

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

This prints the usual:

```
hello, world
```

With input from the file `countries`, the program

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

prints

```
Australia Argentina Algeria
```

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

11.5 Special variables

Some variables in `awk` have special meanings. These are

`NR`	Number of the current record.
`NF`	Number of fields in the current record.
`FS`	Input field separator; by default it is set to a blank or a tab.
`RS`	Input record separator; by default it is set to the newline character.
`$`*i*	The *i*th input field of the current record.
`$0`	The entire current input record.
`OFS`	Output field separator; by default it is set to a blank.
`ORS`	Output record separator; by default it is set to the newline character.
`OFMT`	The format for printing numbers; with the `print` statement, by default it is `%.6g`.
`FILENAME`	The name of the input file currently being read. This is useful because `awk` commands often read multiple files, as in `awk -f` *program file1 file2 file3* ...

11.6 Type

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

```
pop += $3
```

`pop` is presumably a number, while in

```
country = $1
```

`country` is a string. In

```
maxpop < $3
```

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

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

```
var = expr
```

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

```
v1 = v2
```

then the type of `v1` becomes that of `v2`.

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

The type of any expression may be coerced to numeric by maneuvers such as

```
expr + 0
```

and to string by

```
expr ""
```

This last expression is a string concatenated with the null string. If a string cannot be converted to a number without errors, `awk` converts it to zero.

11.7 Arrays

`awk` provides one-dimensional arrays as well as ordinary variables. A name may not be both a variable and an array, however.

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

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

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

```
      { x[NR] = $0 }
END   { action }
```

The first line of this program reads each input line into the array x.

When run on the file countries, the program

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

produces an array of elements with

```
x[1] = "Russia"
x[2] = "Canada"
x[3] = "China"
```

and so forth. Arrays may also be indexed by non-numeric values, thus giving awk a capability rather like the associative memory of Snobol tables. For example, you can write

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

which produces

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

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

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

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

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

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

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

12. Built-in functions

`awk`'s `length` function computes the length of a string of characters. If you don't include an argument, `length` returns the length of the current input record. The following program prints each record preceded by its length:

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

In this case (the variable) `length` is equivalent to `length($0)`, the length of the present input record. In general, `length(`*x*`)` will return the length of *x* as a string. For example, with input taken from the file `countries`, the following `awk` program prints the longest country name:

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

The function

`split(`*s*`,` *array*`)`

assigns the fields of the string *s* to successive elements of the array *array*. For example,

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

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

`split(`*s*`,` *array*`,` *sep*`)`

This splits the string *s* into *array*`[1]`,..., *array*`[`*n*`]`. The number of elements found is returned as the value of `split`. Thus, you may write

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

if you need to know how many elements the string was split into.

When the *sep* argument is present, it must be a string enclosed by double quotes. but only its first character is used as the field separator. When there is no *sep* argument present, `FS` is used. Specifying a *sep* is especially useful if in the middle of an `awk` program it is necessary to change the record separator for one or more records. For instance, if you use the following three lines,

```
{split("Now is+the time", w, "+")}
{split("This\tis\tnot\tthe\tend", x , "\t")}
{print w[1],x[3], w[2] }
```

(`\t` is the tab character) the output will be

```
Now is not the time
```

`awk` also provides the following mathematical functions:

```
sqrt
log
exp
int
```

They provide the square root function, the base *e* logarithm function, and exponential and integer conversion (that is, floating point to integer) functions. The `int` function returns the greatest integer less than or equal to its argument. These functions are the same as those of the C library (except that `int` corresponds to the C library `floor` function) and return the same errors as those in `libc`. (See "C Math Library" in *A/UX Programming Languages and Tools, Volume 1*.)

The substring function

```
substr(s,m,n)
```

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

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

which produces

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

If *s* is a number, `substr` uses its printed image; for example, `substr(123456789,3,4)` is `3456`.

The function

`index(`*s1*`,` *s2*`)`

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

The function

```
sprintf
```

formats expressions as the `printf` statement does (`X`, `E`, `G`, `*` and `u` do not apply), but will assign the resulting expression to a variable instead of sending the results to `stdout`. For example,

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

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

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

13. Flow of control

The `awk` language provides the basic flow-of-control statements:

- `if-else`
- `while`
- `for`

with statement grouping as in the C language.

The `if` statement is used as follows:

`if ( `*condition*` ) `*statement1*` [else `*statement2*`]`

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

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

There is also a `while` statement in `awk`:

`while ( `*condition*` ) `*statement*

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

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

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

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

The `for` statement is like that of C.

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

has the same effect as

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

so that the action

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

is another `awk` program that prints all input fields one per line. Note that multiple initializations are not permitted, as in

```
for (i=1,j=2; ...; ...)
```

There is an alternate form of the `for` statement in `awk` that is suited for accessing the elements of an associative array:

`for (` *var* `in` *array* `)` *statement*

executes *statement* with the variable *var* set in turn to each subscript of *array*. The subscripts are each accessed once but in no predictable order. Chaos ensues if the variable *var* is altered or if any new elements are created within the loop. You could use the `for` statement to print each record preceded by its record number (NR) after the input part of the program is executed:

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

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

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

In this program, the body of the `for` loop is executed for `i` equal to the string `Asia`, then for `i` equal to the string `N. America`, and so forth until all the possible values of `i` are exhausted; that is, the program is repeated until all the strings of names of continents are used. Note, however, that the order in which the loops are executed is not specified. If the iteration associated with `N. America` is executed before the iteration associated with the string `Asia`, such a program might produce

```
S. America 142
Africa 37
N. America 243
Asia 1765
Australia 14
```

The expression in the *condition* part of an `if`, `while`, or `for` statement can include relational operators (see "Operators").

The *condition* can also include regular expressions that are used with the pattern-matching operators and the logical operators (see "Operators"). Finally, it can include parentheses for grouping.

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

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

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

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

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

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

14. Report generation

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

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

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

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

This program produces the following output when used on the input file `prog.usage`:

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

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

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

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

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

gives the output

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

It is often useful to combine different `awk` scripts and other shell commands such as `sort`, as was done in the `list.a` script on the preceding page.

15. Cooperation with the shell

Normally, an awk program is either contained in a file (as a shell script) or enclosed within single quotes, as in

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

Because awk uses many of the same characters that the shell does (such as $ and the double quote), it may take some work to get parameters passed from the shell to awk.

If an awk program is invoked from the command line, surrounding the program by single quotes ensures that the $1 is not interpreted by the shell. The shell passes the awk program to awk intact.

Passing parameters to an awk program contained in a script is slightly more complicated. Suppose you wanted to write an awk program to print the *n*th field of each input record, where *n* is a parameter determined when the program is run. That is, you want a program called field such that

```
field 5 chap.1
```

runs the awk program

```
awk '{print $5}' chap.1
```

How does the value of 5 get from the command line into the awk program? There are several ways to do this. One is to define field as a shell script, as follows:

```
:
# field: print the nth field of each record
awk '{print $'$1'}' $2
```

(This file begins with a colon (:) so that it is interpreted as a Bourne shell script.) When the shell parses this script, it does not interpret anything contained within the first pair of single quotes ({print $), but passes it as input to awk. The shell does recognize the $1, and substitutes its value, the first argument given on the command line. The shell then passes the brace within the second set of single quotes to awk. When awk encounters the end of its first argument, it recognizes that the awk program specification has ended and that the next argument is the file in which the input is found.

Another way to write `field` relies on the fact that the shell substitutes for `$` parameters within double quotes. So you could rewrite the script above as

```
:
# field: print the nth field of each record
awk "{print \$$1}" $2
```

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

This kind of trickery can be extended in remarkable ways, but it may be hard to understand quickly. Experimentation and a playful spirit are encouraged.

16. Lexical conventions

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

- numeric constants
- string constants
- keywords and built-in variables
- identifiers
- operators
- record and field tokens
- comments (discussed previously)
- separators

Precise specifications of each of these tokens are given in the following sections.

16.1 Numeric constants

A numeric constant is either a decimal constant or a floating constant. A decimal constant is a non-null sequence of digits containing at most one decimal point, as in

```
12
12.
1.2
.12
```

A floating constant is a decimal constant followed by `e` or `E` followed by an optional + or − sign followed by a non-null sequence of digits, as in

```
12e3
1.2e3
1.2e-3
1.2E+3
```

16.2 String constants

A string constant is a sequence of zero or more characters surrounded by double quotes, as in

```
"armadillo"
"a"
"ab"
"12"
```

A double quote may be put into a string by preceding it with the backslash (\), as in

```
"He said, \"Sit!\""
```

A newline is put in a string by using `\n` in its place. No other characters need to be escaped except \ itself. Strings can be (almost) any length.

16.3 Predefined variables, reserved keywords, and reserved function names

Table 18-1 lists certain character strings which have special meaning to `awk`. There are three types of these character strings:

1. **Predefined variables** are variables defined by `awk` which have special meanings. The meaning of these variables is explained in "Special Variables."

2. **Reserved keywords** are a special set of character strings used in `awk` statements. Reserved keywords cannot be used as variables.

3. **Reserved function names** are a special set of character strings used to invoke built-in awk functions. These functions are discussed in ''Built-in Functions.''

Table 18-1. Reserved Strings

Predefined Variables	**Reserved Keywords**	**Reserved Function Names**
BEGIN	break	exp
END	close	getline
FILENAME	continue	index
FS	exit	int
NF	for	length
NR	if	log
OFS	in	split
ORS	next	sprintf
OFMT	number	sqrt
RS	print	substr
$0	printf	
$*i*	string	
	while	

16.4 Identifiers

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

16.5 Operators

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

Table 18-2. Assignment operators

Symbol	Usage	Description
=	Assignment	Assign right side value to left side
+=	Plus-equals	Increment left side by value of right side
-=	Minus-equals	Decrement left side by value of right side
*=	Times-equals	Multiply left side by value of right side
/=	Divide-equals	Divide left side by value of right side
%=	Mod-equals	Take modulus of left side by value of right side
++	Prefix/postfix increment	Increment operand by one before/after taking current value
--	Prefix/postfix decrement	Decrement operand by one before/after taking current value

Table 18-3. Arithmetic operators

Symbol	Description
+	Unary and binary plus
-	Unary and binary minus
*	Multiplication
/	Division
%	Modulus
(...)	Grouping

Table 18-4. Relational operators

Symbol	Description
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal
>=	Greater than or equal
>	Greater than

Table 18-5. Logical operators

Symbol	Description
||	OR
&&	AND
!	NOT

Table 18-6. Regular expression pattern-matching operators

Symbol	Description
~	Matches
!~	Does not match

16.6 Record and field tokens

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

The keyword NR (number of records) is a variable whose value is the number of input records read so far. The first input record read is 1. At END it contains the total number of input lines.

16.7 Separators

16.7.1 Record separators

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

```
RS = "c"
```

in an action.

16.7.2 Field separators

The keyword FS (field separator) is a variable indicating the current field separator. Initially, the value of FS is a blank, indicating that fields are separated by white space, that is, any sequence of blanks and tabs. Keyword FS may be changed to any single character *c* by including the assignment statement

```
FS = "c"
```

in an action or by using the flag option `-Fc`. Two values of *c*, space and `\t`, have special meaning. The assignment statement

```
FS = " "
```

makes white space (blank spaces or tabs) the field separator; and on the command line, `-F\t` makes a tab the field separator.

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

16.8 Multiline records

The assignment

```
RS = ""
```

as part of the action associated with a `BEGIN` pattern makes an empty line the record separator. It also makes a sequence of blanks, tabs, and possibly a newline, the field separator. With this setting, none of the first fields of any record is null, as discussed above.

16.9 Output record and field separators

The value of `OFS` (output field separator) is the character or string separating output fields. It is put between fields by `print`. The value of `ORS` (output record separator) is put after each record by `print`. Initially, `ORS` is set to a newline and `OFS` to a space. These values may be changed to any string by assignments such as the following two:

```
ORS = "abc"
OFS = "xyz"
```

16.10 Separators and braces

Tokens in `awk` are usually separated by non-null sequences of blanks, tabs, and newlines, or by other punctuation symbols such as commas and semicolons. Braces (`{ }`) surround actions, slashes (`/ /`) surround regular expression patterns, and double quotes (`" "`) surround strings. Braces may also be used to group statements within actions.

17. Primary expressions

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

- numeric constants
- string constants
- variables
- functions

Each expression has both a numeric and a string value, and will default to one or the other, depending on context. The rules for determining the default value of an expression are explained below.

17.1 Numeric constants

A numeric constant is simply a number. The format of a numeric constant was previously defined in the section “Lexical Conventions.”

The value of a numeric constant is always its numeric value in decimal unless it is coerced to type string. Table 18-7 shows the result of coercing various numeric constants to type string. Coercion of a numeric constant may occur explicitly as defined in ''Type'' or implicitly within the context of an expression.

Table 18-7. Values for sample numeric constants

Numeric Constant	Numeric Value	String Value
`0`	`0`	`0`
`1`	`1`	`1`
`.5`	`0.5`	`.5`
`.5e2`	`50`	`50`

17.2 String constants

A string constant is simply a series of characters enclosed in double quotes. The format of a string constant was defined in ''Lexical Conventions.''

The value of a string constant is the contents of the string itself unless it has been coerced to type numeric. The numeric value of a string coerced to type numeric depends on the contents of the string: If the string is composed entirely of numbers (either decimal or floating point format), its numeric value is the number contained in the string. If the string does not contain a recognizable decimal or floating point number, its numeric value is zero. Table 18-8 shows the result of coercing various string constants to type numeric. Coercion of a string constant may occur explicitly as defined in ''Type'' or implicitly within the context of an expression.

Table 18-8. Values for sample string constants

String constant	Numeric value	String value
`""`	0	null string
`" "`	0	space
`"a"`	0	`a`
`"XYZ"`	0	`XYZ`
`"0"`	0	`0`
`"1"`	1	`1`
`".5"`	0.5	`.5`
`".5e2"`	50	`.5e2`

17.3 Variables

A **variable** or **var** is in one of the following forms:

identifier
identifier[*expression*]
`$`*term*

The numeric value of any uninitialized variable is `0`, and the string value is the empty string. An *identifier* by itself is a simple variable. A variable of the form

identifier[*expression*]

represents an element of an associative array named by *identifier*.

The string value of *expression* is used as the index into the array. The default value of *identifier* or *identifier* [*expression*] is determined by context.

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

The variables `$1` and `$2` refer to fields 1 and 2 of the current input record. The string and numeric values of `$`*i* for `1<=`*i*`<=NF` are those of the *i*th field of the current input record. As with `$0`, if the *i*th field represents a number, then the numeric value of `$`*i* is the number and the string value is the literal string. The default value of `$`*i* is string unless the *i*th field is a number. The `$`*i* may be changed by assignment. The value of `$0` is then changed accordingly, but the results may not be apparent unless `NF` is changed to at least *i*.

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

17.4 Functions

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

```
exp [(expression)]
int [(expression)]
log [(expression)]
sqrt [(expression)]
```

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

```
getline
index (expression1, expression2)
length [(expression)]
split (expression, identifier [, "separator"])
sprintf [("format", expression1 [,expression2 ...])]
substr (expression1, expression2 [, expression3 ])
```

These functions (`getline`, `index`, `length`, `split`, `sprintf`, and `substr`) perform spring operations. See ''Built-in Functions'' for more details.

18. Terms

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

primary expression
term1 binop term2
unop term
incremented var
(*term*)

18.1 Binary terms

In a term of the form

term1 binop term2

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

18.2 Unary terms

In a term of the form

unop term

unop can be unary + or –. The unary operator is applied to the numeric value of *term*, and the resulting numeric value is the default value. However, it can be interpreted as a string value. Unary + and – have higher precedence than `*`, `/`, and `%`.

18.3 Incremented variables

An incremented variable has one of the following forms:

++*var*
--*var*
var++
var--

That is, it can be either *pre-* or *post-incremented.*

The form ++*var* has the effect of the assignment

var = *var* + 1

and so has the value *var*+1 before it is further evaluated or assigned.

Similarly, the form --*var* has the effect of the assignment

```
var = var - 1
```

and so has the value *var*-1 before it is further evaluated or assigned.

The form *var*++ has the same value as *var* before it is evaluated or assigned, and after that it has the effect of the assignment

```
var = var + 1
```

Similarly, the form *var*-- has the same value as *var* before it is evaluated or assigned, and after that it has the effect of the assignment

```
var = var - 1
```

The default value of an *incremented var* is numeric. You shouldn't use the ++ or -- operators where the incremented variable is used more than once (such as `a = b++ * b`), since the results are indeterminate.

18.4 Parenthesized terms

Parentheses are used to group terms in the usual manner.

19. Expressions

An `awk` expression is one of the following:

term
term1 term2 ...
var asgnop expression

19.1 Concatenation of terms

In an expression of the form *term1 term2* the string values of the terms are concatenated. If the terms are numeric expressions they are first evaluated and then also treated as strings; that is, the default value of the resulting expression is a string value that can be interpreted as a numeric value. Concatenation of terms has lower precedence than binary + and –. For example, the expression

```
1+2 3+4
```

has the string (and numeric) value 37.

19.2 Assignment expressions

An *assignment expression* is one of the form

    var asgnop expression

where *asgnop* is one of the six assignment operators (=, +=, −=, *=, /=, %=, ++, --) (see ''Operators'').

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

In an expression of the form

    var = expression

the numeric and string values of *var* become those of *expression.*

An expression of the form

    var op = expression

is equivalent to

    var = var op expression

where *op* is one of the arithmetic operators (see ''Operators'').

The *asgnops* are right associative and have the lowest precedence of any operator. Thus, the assignment

```
a += b *= c - 2
```

is interpreted as

```
a += ( b *= ( c - 2 ) )
```

which is equivalent to the sequence of assignments

```
b = b * (c - 2)
a = a + b
```

Chapter 19
lex Reference

Contents

Figures

Tables

Chapter 19

lex Reference

1. lex: a lexical analyzer

lex is a program generator that produces a program in a general-purpose language that recognizes regular expressions. It is designed for lexical processing of character input streams. It accepts high-level, problem-oriented specifications for character string matching.

Input to lex is a table of regular expressions and corresponding program fragments. The table is translated to a program that reads an input stream, copies the input stream to an output stream, and partitions the input into strings that match the given expressions. As each such string is recognized, the corresponding program fragment is executed.

The recognition of the regular expressions is performed by a deterministic finite automaton generated by lex. The program fragments are executed in the order in which the corresponding regular expressions occur in the input stream.

The code written by lex is not itself a complete language, but rather a generator representing a new language feature that can be added to different programming languages, called ''host languages.'' For example, one high-level language may be used for recognizing patterns, while a more general-purpose language is used for action statements.

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 (for example, yacc) to perform the lexical analysis phase.

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 the program fragments that comprise the lex source program.

Compatible run-time libraries for the different host languages are provided, making `lex` adaptable to many environments and users. However, at present, the only supported host language is the C language.

2. Overview of `lex` usage

The program generated by `lex` is called `yylex`. The `yylex` program recognizes expressions in an input stream and performs the specified actions for each expression as it is detected. See Figure 19-1.

Figure 19-1. Overview of **`lex`**

$$\begin{tikzcd}
\text{Source} \arrow[r] & \boxed{\texttt{lex}} \arrow[r] & \texttt{yylex()} \\
\text{Input} \arrow[r] & \boxed{\texttt{yylex}} \arrow[r] & \text{Output}
\end{tikzcd}$$

For example,

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

This sample `lex` source program is all that is required to generate a program to delete all blanks or tabs at the ends of the input lines. The `%%` delimiter is a `lex` convention to mark the beginning of the rules, the pattern-matching expressions. The rule itself,

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

matches one or more instances of the characters blank and tab. The brackets enclose the character class consisting of blank and tab; the + indicates ''one or more instance of the previous character(s) or character class'' and the `$` indicates end-of-line. No action is specified, so the `yylex()` program (generated by `lex`) ignores these characters. Everything else is copied.

Consider this next example:

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

The coded instructions in yylex scan for both rules at once. Once a string of blanks or tabs is recognized, yylex determines whether the string is followed by a newline character. If it is, then the first rule has been matched so that the corresponding action is performed; yylex does not copy the string to output. The second rule matches strings of one or more blanks and tabs not already satisfying the first rule, and causes yylex to replace a string of one or more blanks and tabs with a single space.

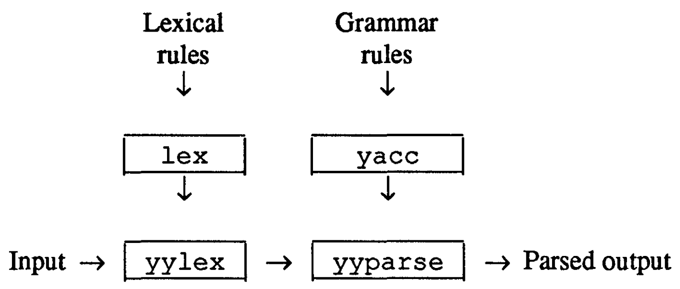
In yylex, the program generated by lex, the actions to be performed as each regular expression is found are gathered as cases of a switch. The automaton interpreter directs the control flow. It is possible to insert either declarations or additional statements in the routine containing the actions and to add subroutines outside this action routine, should you need to do so.

The lex program generator is not limited to one-character look-ahead. For example, if there are two rules, one looking for ab and another for abcdefg, and the input stream is abcdefh, lex recognizes ab and leaves the input pointer just before cdefh.

3. lex and yacc

It is particularly easy to use lex and yacc together. The lex program recognizes only regular expressions; yacc writes parsers that accept a large class of context-free grammars but requires a lower level analyzer to recognize input tokens. Thus, a combination of lex and yacc is often appropriate. When used as a preprocessor for a later parser generator, lex is used to partition the input stream; and the parser generator assigns structure to the resulting pieces. The flow of control in such a case is shown in Figure 19-2. Additional programs, written by other generators or by hand, can be added easily to programs written by lex. The name "yylex" is what yacc expects its lexical analyzer to be named. If lex uses this name, it simplifies interfacing.

Figure 19-2. `lex` with `yacc`



To use `lex` with `yacc`, observe that `lex` writes a function named `yylex`, which is the name required by `yacc` for its analyzer. Normally, the default main program on the `lex` library calls the `yylex` routine, but if `yacc` is loaded and its main program is used, `yacc` calls `yylex`. In this case, each `lex` rule ends with

```
return(token);
```

where the appropriate token value is returned. An easy way to gain 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 the `yacc` input. If the grammar is to be named `good` and the lexical rules are to be named `better`, the command sequence could be

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

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

4. Program syntax

The general format of `lex` input is

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

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

```
%%
```

This `lex` source would generate a program that copies the input to output unchanged.

In the `lex` program format shown above, the *rules* consist of two parts:

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

For example,

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

The sample rule above gives the instructions to look for the string `integer`, and, when found, it produces the statement

```
found keyword INT
```

In this example, because the host procedural language is C, 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 a single C language expression, it can just be given in the right column, as illustrated in the example. If the action is compound or requires more than one line, it should be enclosed in braces.

Consider the following example:

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

This lex source segment could be used to generate a program to change a number of words from British to American spelling. It should be noted, however, that these rules would have to be changed somewhat to be really useful. For example, if the word petroleum appeared in the input stream, the program generated by this segment would change it to gaseum.

5. Character set

Internally, a character is represented as a small integer. If the standard library is used, a character's value is equal to the integer value of the bit pattern representing the character on the host computer. For example, the character A has the value \101 (octal) in ASCII.

Of course, you need not use the integer value of a character to access the value. The character 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 of the source, and this translation table must be bracketed by lines containing only %T. The translation table, then, contains lines of the form

```
%T
{ integer } { character string }
%T
```

which indicate the value associated with each character.

5.1 Character classes

Classes of characters can be specified using the operator pair [and]. For example, the construction [abc] matches a single character, which may be a, b, or c.

Within brackets, most operator meanings are ignored. Only three characters are special:

```
-
^
\
```

The – character indicates a range. For example,

```
[a-z0-9<>_]
```

specifies the character class containing all the lowercase letters (`a` to `z`), digits (`0` through `9`), angle brackets (< and >), and the underline character (_).

Using – between any pair of characters that are not both uppercase letters, both lowercase letters, or both digits is sometimes acceptable to `lex`, but this is implementation dependent (it works on A/UX, but it may not be portable to other systems.) Therefore, if such a range is declared, `lex` issues a warning message. One reason for this is that `[0-z]` matches many more characters in ASCII than in EBCDIC.

If it is necessary to include the character – in a character class, it should either be first or last within the brackets. For example,

```
[-+0-9]
```

matches *all* digits (`0` through `9`) and the two symbols – and +.

The \ character acts as an escape character within class brackets. For example,

```
[a-z\*]
```

matches all lowercase letters (`a` to `z`) *and* the character `*`.

If the ^ operator appears as the first character after the left bracket, `lex` *ignores* the characters within the brackets, therefore matching all characters *except* those within the designated character class range. If an operation is to be performed on recognition of a string expressed using this construction, it will be done on strings *other than* those within the brackets. For example,

```
[^abc]
```

matches all characters *except* `a`, `b`, or `c`, including all special and control characters. Also,

```
[^a-zA-Z]
```

matches any character that is *not* a letter (neither in the range `a` through `z` nor in the range `A` through `Z`).

5.2 Arbitrary characters

There are several other ways to specify characters to `lex`. The period operator (`.`) instructs `lex` to match any character except a newline.

The meaning of the period does not change within brackets.

Also, all characters and ranges can be designated using the octal representations of those characters. This method, however, is difficult to read and most likely not portable. Nonetheless, the character class range

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

can be used to match all printable ASCII characters from octal 40 (blank) to octal 176 (tilde: ~).

5.3 Operators

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

If these are to be used as text characters, an appropriate ''escape'' should be used. For example, to get the character \, you must escape its significance as an operator. You can do so easily with another backslash: \\. For more information on escaping, refer to *A/UX User Interface*.

The quotation mark operator (") indicates that whatever characters follow, up to a second " character, are to be taken as text characters without any ''magic'' meaning or operator significance. The quotation mark, then, is another way to escape the special meaning of a character. For example,

```
xyz"++"
```

matches the string `xyz++` wherever it appears. Of course, it is unnecessary, though harmless, to quote an ordinary text character. Consequently, the expression

```
"xyz++"
```

is equivalent to the one that quoted only the ++. However, by quoting every character being used as a text character, you can avoid remembering the list of current operator characters, and avoid problems should further extensions to `lex` lengthen the list.

Another use of the quoting mechanism is for forcing a blank into an expression. Normally, as explained above, blanks or tabs end a rule. Any blank character not contained within brackets *must* be quoted.

There is also a third way to match the literal value of these operators, using the \ escape character. You could specify the string discussed above as

```
xyz\+\+
```

Several C language escapes using \ are recognized:

C language escapes

`\n`	Newline
`\t`	Tab
`\b`	Backspace
`\\`	Backslash

Since newline is illegal in an expression, `\n` must be used.

6. Definitions

Recall that the basic format of a `lex` source is

{ *definitions* }
%%
{ *rules* }
%%
{ *user subroutines* }

In addition to the *rules* (discussed below), `lex` includes options to define variables. Variables can occur either in the *definitions* section or in the *rules* section.

Remember, `lex` is generating the rules into a program, and any source not intercepted by `lex` is copied into the generated program. Also,

- Any line not part of a `lex` rule or action and that begins with a blank or tab is copied into the `lex` generated program.
- Any line not part of a `lex` rule or action that begins with a blank or tab and is found prior to the first `%%` delimiter is ''external'' to any function in the code.
- Any line not part of a `lex` rule or action that begins with a blank or tab and is found immediately after the first `%%` appears in an appropriate place for declarations in the function written by `lex` that 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.
- Anything included *between* lines containing only `%{` and `%}` is copied to output. 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.
- Anything after the third `%%` delimiter, regardless of formats, and so on, is copied to output *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

This facility enables the string given as *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 be called by the {*name*} syntax in a rule. Using `{D}` for the digits and `{E}` for an exponent field, you might have

```
D                       [0-9]
E                       [DEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?           |
{D}*"."{D}+({E})?           |
{D}+{E}                 printf("real");
```

This example abbreviates rules to recognize numbers. 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 (`{D}+"."{D}*({E})?`), and the second requires at least one digit after the decimal point (`{D}*"."{D}+({E})?`). To correctly handle the the Fortran expression `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 (see "Context Sensitivity").

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.

6.1 Repetitions and definitions

The operators { and } specify either

- repetitions (if they enclose numbers)
- 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. On the other hand, the expression

```
a{1,5}
```

looks for one to five occurrences of `a`.

An initial `%` is not an ordinary character, but has a special meaning to `lex` as the the separator for source program segments.

7. Rules

7.1 Regular expressions

The regular expressions in `lex` function just as do those in the A/UX text editors (`vi`, `ed`, and so on). A regular expression specifies a set of strings to be matched. It contains "text characters" (which match characters in the input stream) and "operator characters" (which, together with those "text characters," express a string that is to be recognized before the action in the right column takes place).

Letters of the alphabet and digits are always text characters. For example,

```
integer
```

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

```
a57D
```

looks for the string `a57D`.

7.2 Optional expressions

The question mark operator (`?`) indicates that what immediately precedes it is an optional element of an expression. Thus,

```
ab?c
```

matches either `ac` or `abc`.

7.3 Repeated expressions

Repetitions of classes are indicated by the operators `*` and `+`. The expression

```
a*
```

matches zero or more consecutive `a` characters. The expression

```
a+
```

matches one or more instances of `a` characters. The expression

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

matches all strings of lowercase letters. The expression

```
[A-Za-z][A-Za-z0-9]*
```

matches all alphanumeric strings that have a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

7.4 Alternation and grouping

The operator `|` indicates *alternation.* For example,

```
(ab|cd)
```

matches either `ab` or `cd`. The parentheses are used here for grouping only. They are not required in such a simple and clear-cut example, but are often used for clarity or to force correct interpretation of more complex expressions. For example,

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

matches such strings as

```
abefef
efefef
cdef
cddd
```

but not

```
abc
abcd
abcdef
```

7.5 Context sensitivity

The `lex` program recognizes a small amount of surrounding context. The two simplest operators for this are ^ and `$`.

As in the A/UX text editors, if the first character of an expression is ^, the expression is matched only if found at the beginning of a line, either after a newline character or at the beginning of the input stream. Do not confuse this with the use of the ^ operator within brackets, which instructs `lex` to match any character except those in the designated character class range. If you want to use `lex` to find occurrences of a particular range of characters, but only if they occur as the first character on a line, you must use the ^ operator on the *outside* of the brackets. For example, the expression

```
^[0-9]
```

matches lines whose first character is a digit, `0` through `9`. The expression

```
^[^0-9]
```

matches lines whose first character is not a digit `0` through `9`.

The operator `$` is matched only at the end of a line, immediately followed by newline. This operator is a special case of the / operator character, which indicates "trailing context." The expression

```
ab/cd
```

matches the string ab only if followed by cd. Therefore, the expression

```
ab$
```

could also be expressed

```
ab/\n
```

That is, the use of the $ operator could be interpreted as an instruction to match the character(s) only when followed by a newline.

Left context is handled in lex by ''start conditions.'' If a rule is only to be executed when the lex automaton interpreter is in ''start condition'' x, the rule should be enclosed within the angle-bracket operator characters:

```
<x>
```

If ''being at the beginning of a line'' was considered to be start condition ONE, then the ^ operator would be equivalent to

```
<ONE>
```

See the sections entitled ''Left Context Sensitivity,'' ''Examples,'' and ''Summary'' for further explanation and illustration of start conditions.

7.5.1 Left context sensitivity

Sometimes it is desirable to have several sets of lexical rules 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. There are several ways of handling such occurrences. For example, the ^ operator is a ''prior context operator'' because it must recognize the immediately preceding left context in order to discern if a character appears at the beginning of a line, just as the $ operator must recognize the immediately following right context in order to discern if a character appears at the end of a line.

Adjacent left context could be extended to produce a facility similar to that for adjacent right context. This is likely to be less useful, however, since often the relevant left context, such as the beginning of a line, appeared some time earlier.

There are three basic ways of dealing with different environments so as to achieve a lexical analysis with a greater degree of context sensitivity.

- A use of flags. This is most useful when only a few rules change from one environment to another.
- A use of ''start conditions'' on rules.
- The possibility of making multiple lexical analyzers all run together. If the sets of rules for the different environments are very dissimilar, clarity may best be achieved by writing several distinct lexical analyzers and switching from one to another as necessary.

In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed and a parameter is set to reflect the change. The remainder of this section describes in greater detail the first two ways of dealing with different environments.

7.5.2 Flags

The simplest way of changing the environment in which input is analyzed is by use of a ''flag'' explicitly tested by the user's action code. If done in this way, `lex` is not involved at all.

To illustrate, consider the following program requirements:

- Copy the input to the output
- Change the word `magic` to `first` on every line that begins with the letter `a`
- Change `magic` to `second` on every line that begins with the letter `b`
- Change `magic` to `third` on every line that begins 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. For example,

```
        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;
          }
        }
```

7.5.3 Start conditions

It may be more convenient to have `lex` "remember" the flags as "start conditions" on the rules. Any rule may be associated with a start condition. That rule, then, would be recognized only when `lex` is in that start condition. The current start condition may be changed at any time. To handle the same problem using start conditions, begin by introducing each start condition to `lex` in the *definitions* section with a line reading

`%Start` *name1 name2* ...

where the conditions (*name1*, *name2*, and so on), may be named in any order. The word `Start` may be abbreviated to `s` or `S`.

Then, to reference the conditions use angle brackets:

`<name1>` *expression*

The rule illustrated above will be recognized *only* when `lex` is in the "start condition" *name1*. To enter that start condition, execute the following action statement:

`BEGIN` *name1*;

The action statement

```
BEGIN 0;
```

resets the initial condition of the `lex` automaton interpreter.

A rule may be active in several start conditions.

<name1, name2, name3> expression

is a legal expression.

Any rule *not* beginning with the < prefix operator is always active.

The following example illustrates the use of start conditions:

```
%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");
```

Obviously, the above is a rewrite of the previous example; the problem-solving logic is exactly the same. However, in this case, `lex` has been instructed to do the work instead of the host language code.

7.6 Ambiguous rules

The `lex` program can handle ambiguous specifications. When more than one expression can match the current input, the longest match is preferred, and among rules that matched the same number of characters, the rule given first is preferred.

For example, using the rules

```
integer    keyword-action ;
[a-z]+     identifier-action ;
```

(if the input was integers), `lex` would interpret the input as an identifier because `[a-z]+` matches all eight characters (including the final `s`), while `integer` matches only seven characters.

If the input were `integer`, both rules would match the seven characters. In that case, `lex` would select the keyword rule because it was given first. If the input were anything shorter (for example, `int`), the input would not match the expression `integer`. It would, however, match the `[a-z]+` expression, so the identifier interpretation would be used.

The principle of preferring the longest match makes rules containing expressions like `.*` dangerous. For example,

```
'.*'
```

appears to instruct `lex` to find a match for a string in single quotes. However, it is an instruction for the program to read far ahead looking for a distant single quote. For example, if the above expression were given the following input:

```
'first' quoted string here, 'second' here
```

the expression would match almost the entire input line:

```
'first' quoted string here, 'second'
```

which is most likely *not* the desired result. A better rule for matching strings within single quotes might be

```
'[^'\n]*'
```

which, given the same input, will match `'first'`.

The consequences of errors like this are greatly lessened by the fact that the period (`.`) operator does not match newline. Expressions like `.*` stop on the current line.

> *Note:* Do not try to defeat the protection of `.` not matching the newline character with expressions such as `[.\n]+` or an equivalent, because the program generated by `lex` will then try to read the entire input file, causing internal buffer overflows.

8. Actions

When an expression written as above is matched, `yylex` executes the corresponding action. The default action for `yylex` is to copy input to output, and is performed on all strings not otherwise matched. Therefore, a rule that merely copies can be omitted. If you want to absorb the entire input without producing any output, you must provide rules to match everything. (When `yylex` is being used with `yacc`, this is the normal situation.) In other words, by default, a character combination in input that was omitted from the rules will be printed on the output.

8.1 The null statement

One of the simplest things that can be done is to ignore the input. To accomplish this, use a semicolon (`;`) as the action (a semicolon is the C language ''null statement'').

The rule

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

causes the spacing characters (that is, blank, tab and newline) to be ignored because it gives the null statement as its associated action.

8.2 The repetition character

The vertical bar character (|) represents the instruction to use the action designated for the next rule for the current rule as well. For example,

```
" "     |
"\t"    |
"\n"    ;
```

This example instructs `yylex` to ignore the spacing characters, as did the previous example. The first line gives the rule ''match blank characters'' and instructs the program to perform the action indicated for the next rule. Then, the second line gives the rule ''match `\t` characters'' and instructs the program to perform the action indicated for the next rule. Finally, the third line gives the rule ''match `\n` characters,'' and gives the action `;`, the null statement. Therefore, the action for all three rules is the null statement.

8.3 `printf` and `ECHO`

In more complex actions, you may often want to know the actual text that matched a regular expression. The `yylex` program leaves this text in an external character array, named `yytext`. Consider the following example:

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

This example illustrates a way of accessing the characters matching a regular expression. Using this example, the rule given is to find the strings matching the regular expression `[a-z]+` and the action is to print those strings in the character array `yytext` using the C language function `printf`.

The `printf` function accepts a format argument and data to be printed. Still using this example, the format is `%s` (print string). The `%` character indicates data conversion, and `s` indicates data type string, in this case the character array, `yytext`. This places the matched string on the output.

The action of printing the strings matching the regular expressions is so common that it may be written simply as `ECHO`. For example,

```
[a-z]+   ECHO;
```

This example accomplishes the same action as the previous one using the `printf` statement.

Even though the default action is to copy input to output, the `ECHO` facility is included explicitly to provide a more discriminating copy function. For example, a rule that matches `read` will normally match all instances of `read`, even those contained in other words (`bread`, `treadmill`, and so on). To avoid this, a rule of the form `[a-z]+` is needed. This is explained further below.

8.4 yyleng

Sometimes it is necessary to know what is at the end of a matched pattern. To facilitate this, `lex` provides a count of the number of characters matched, `yyleng`. To count both the number of words in the input and the number of characters in those words, you might write

```
[a-zA-Z]+   {words++; chars += yyleng;}
```

This instruction takes the strings that match the regular expression `[a-zA-Z]+` and accumulates the number of characters in these strings in `chars`. Then, the action instruction

```
yytext[yyleng-1]
```

could be used to access the last character in the string matched.

8.5 yymore and yyless

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:

`yymore()`	This routine instructs `yylex` to tack the next input expression recognized on to the end of this input. Normally, the next input string would overwrite the current entry in `yytext`.
`yyless(`*n*`)`	This routine instructs `yylex` to retain in `yytext` only *n* (a number) of those characters resulting from the current expression. Further characters previously matched are returned to the input. This provides the same sort of look-ahead offered by the / operator, though in a very different form.

Consider a language that defines a string as a set of characters between quotation marks ("), and requires that the " character be preceded by a \ to be included in a string. The regular expression which matches that is somewhat confusing, so it might be preferable to write the following:

```
\"[^"]* {
              if (yytext[yyleng-1] == '\\')
                yymore();
              else
                ...normal user processing
            }
```

The above `lex` segment will, when it finds the string

```
"abc\"def"
```

first match the five characters `"abc\` and then call the `yymore` routine, which will cause the next part of the string, `"def`, to be tacked on the end of the input. Note that the final quote terminating the string should be picked up in the code labeled *normal user processing*.

The function `yyless` might be used to reprocess text in various circumstances. Consider, for example, the problem of disambiguating a C language statement such as

```
s=-a
```

One way to parse this statement treats the – as part of the operator:

```
=-[a-zA-Z] {
               printf("Operator (=-) ambiguous\n");
               yyless(yyleng-1);
               action for =-
           }
```

This `lex` segment will print a message, treat the operator as =–, and return the letter found after the operator to the input stream. However, you might want to treat this syntax as = `-a`. In that case

```
=-[a-zA-Z] {
               printf("Operator (=-) ambiguous\n");
               yyless(yyleng-2);
               action for =
           }
```

will print a message, treat the operator as =, and return `-a` to the input stream.

It is possible to avoid the misinterpretation of operators by rewriting the regular expression. To indicate that the operator is =–, using the same example, use the following rule:

```
=-/[A-Za-z]
```

To indicate that the operator is =, use the following rule:

```
=/-[A-Za-z]
```

No backup is required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. However, the possibility of `=-3` makes

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

a still better rule.

8.6 `lex` input and output routines

The programs generated by `lex` handle character I/O only through the routines `input`, `output`, and `unput`. The character representation provided in these routines is accepted by `lex` and used to return values in yytext. These are provided as lex macro definitions:

input()	Returns the next input character
output(*c*)	Writes the character *c* on the output
unput(*c*)	Pushes the character *c* back onto the input stream to be read later by input

(As shown previously, you can use printf to generate error messages.) These routines are provided by default, but you can override them by providing your own versions. To redefine or override a lex routine, include your own version in the *user subroutines* section. These routines must be standard C and be named according to the lex routine you want to replace. However, because these routines define the relationship between external files and internal characters, they must all be retained and/or modified consistently.

These routines may be redefined to cause input or output to be transmitted to or from other programs or internal memory. The character set used must be consistent in all routines and a value of 0 returned by input must mean end-of-file.

The relationship between unput and input must be retained or the lex look-ahead will not work. The lex program does not look ahead at all if it does not have to; rules ending in +, *, ?, or $, or those containing a /, however, will force look-ahead. Look-ahead is necessary to match an expression that is a prefix of another expression. The standard lex library imposes a 100-character limit on backup.

8.7 yywrap

Another lex library routine that you may sometimes want to redefine is yywrap. To redefine or override a lex routine, include your own version in the *user subroutines* section. These routines must be standard C and be named according to the lex routine you want to replace. This routine is called whenever lex reaches an end-of-file. If yywrap returns a 1, which it does by default, lex continues with the normal wrapup on end of input.

It is sometimes convenient to arrange for input to continue from a new source. In this case, yywrap could be redefined to arrange for new input and return 0. This would then instruct lex to continue processing.

This routine provides a convenient way to print tables, summaries, and so on, at the end of a program. It is not possible to write a normal rule that recognizes end-of-file. The only access to this condition is through yywrap. In fact, unless a private version of input is supplied, a file containing nulls cannot be handled because a value of 0 returned by input is taken to be end-of-file by yywrap.

8.8 REJECT

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. Consider the following example:

```
she    s++;
he     h++;
\n     |
.      ;
```

The first rule matches all occurrences of the string she and the action increments s for each one found. The second matches all occurrences of the string he and its action increments h for each one found. The last two rules match newline and everything else and take the action of ignoring them. Normally, lex would not recognize the instances of he included in she, because once it has passed a she those characters are gone. To override this default, the action REJECT could be used to instruct lex to go do the next alternative. REJECT causes the rule *after* the current rule to be executed. The position of the input pointer is adjusted accordingly.

Suppose you want to count the instances of he included in she. To do that, use the following rules:

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

In this example, after counting each expression, the expression is "rejected" (whenever appropriate), and the other expression is evaluated. In this example, because he does not include she the REJECT action on he could be eliminated. In other cases, it is not possible to state which input characters are in both classes.

Consider the following two rules:

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

- If the input to the rules above were ab, only the first rule would match.
- If the input to these same rules were ad, only the second would match.
- If the input were accb, the first rule would match four characters, and the second rule would match three characters.
- If the input were accd, however, the second rule would match four characters, and the first rule would match three characters.

In general, REJECT is useful whenever the purpose of lex is to detect all examples of some items in the input for which the instances of these items may overlap or include one another, instead of lex's usual purpose of partitioning the input stream.

Suppose you want a digram of some input. Normally, the digrams overlap, that is, the word the is considered to contain both th and he. Assuming a two-dimensional array named digram[] to be incremented, an appropriate lex procedure would be

```
%%
[a-z][a-z] {digram[yytext[0]][yytext[1]]++; REJECT;}
.            |
\n           ;
```

In this example, REJECT is used 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. Therefore, if yylex is instructed to find a rule with trailing context and execute REJECT, unput cannot have been called 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.

9. Compilation

The following steps are involved in compiling a `lex` source file:

1. The `lex` source must be transformed into a program in the host general-purpose language. The generated program is put into a file named `lex.yy.c`.
2. That program must then be compiled and loaded, usually with a library of `lex` subroutines. The I/O library is defined in terms of the C language standard library. On the A/UX operating system, the library is accessed by the loader flag `-ll`. In this case, an appropriate set of commands is

   ```
   lex inputfile
   cc lex.yy.c -ll
   ```

   The resulting program is placed in the file `a.out` for later execution.

Although the default `lex` I/O routines use the C language standard library, `lex` routines such as `input`, `output`, and `unput` do not. Therefore, if your own versions of these routines are given, the library is avoided.

10. Examples

For the sake of example, consider copying an input file while adding three to every positive number divisible by 7. 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, 0 through 9; `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 3 as it is written out. It may be objected that this program alters such input items as `49.63` or `X7`. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, add a few more rules after the active one. For example,

```
%%
   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;
```

Numeric strings containing a period (`.`), 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 expression *a* `?` *b* `:` *c* is evaluated as "if *a* then *b* else *c*."

The following is an example using `lex` for statistics gathering. This program reports how many words of various lengths there are. (A word is defined here 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);
}
```

In the above example, the data is accumulated, but no output is generated until, at the end of the input, the table is printed. The final statement, `return(1);`, indicates that `lex` is to perform wrapup. If `yywrap` returns 0 (false), it implies that further input is available and the program is to continue reading and processing. Remember, providing a `yywrap` that never returns true causes an infinite loop.

11. Summary

The general form of a `lex` source file is

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

The *definitions* section contains a combination of the following:

- Definitions in the form

  name translation

- Included code in the form

  code

  where a space (or tab) must precede *code*

- Included code in the form

  `%{`
  code
  `%}`

- Start conditions given in the form

  `%S` *name1 name2 ...*

- Character set tables in the form

  `%T`
  number character-string
  ...
  `%T`

- Changes to internal array sizes in the form

```
%x    nnn
```

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	**Parameter**
`p`	Positions
`n`	States
`e`	Tree nodes
`a`	Transitions
`k`	Packed character classes
`o`	Output array size

Lines in the *rules* section have the form

expression action

where the *action* may be continued on succeeding lines by using braces to delimit it.

Regular expressions in `lex` use the following operators:

Table 19-1. Regular expression operators

Expression	Meaning
x	The character x
"x"	An x, even if it is an operator
\x	An x, even if it is an operator
[xy]	The character x or y
[x-z]	The characters x, y, or z
[^x]	Any character but x
.	Any character but newline
^x	An x at the beginning of a line
<y>x	An x when `lex` is in start condition y
x$	An x at the end of a line
x?	An optional x
x*	0 or more instances of x
x+	1 or more instances of x
x|y	An x or a y
(x)	An x
x/y	An x, but only if followed by y
{xx}	Expands to xx definition in `lex` definition section
x{*m*,*n*}	*m* through *n* occurrences of x

Chapter 20
yacc Reference

Contents

Tables

Chapter 20

yacc Reference

1. yacc: a compiler-writing system

The yacc program is a general tool for imposing structure on the input to a computer program. The first step in using yacc is to create a specification of the input process, which includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. yacc then generates a function to control the input process. This function, called a "parser," calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called "tokens") from the input stream.

Tokens are organized according to the input structure rules, called "grammar rules." When one of these rules has been recognized, the user code supplied for this rule (that is, an action) is invoked. Actions have the ability to return values and make use of the values of other actions.

yacc is written in a portable dialect of the C language, and the actions and output subroutine are written in the C language as well. Moreover, many of the syntactic conventions of yacc follow those of 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 serve merely as punctuation in the rule and have no significance in controlling the input.

With proper definitions, the following input might be matched by the rule given above:

```
July 4, 1776
```

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 following rules might be used in the above example:

```
month_name : 'J' 'a' 'n'   ;
month_name : 'F' 'e' 'b'   ;
...
month_name : 'D' 'e' 'c'   ;
```

The lexical analyzer needs to recognize only individual letters, and `month_name` is a nonterminal symbol.

Such low-level rules tend to waste time and space and may complicate the specification beyond the ability of `yacc` to deal with it.

Usually, the lexical analyzer recognizes the month names and returns an indication that a `month_name` is seen. In this case, `month_name` is a token.

Literal characters (such as the comma above) must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. If the rule

```
date : month '/' day '/' year;
```

were added to the above example, entering `7/4/1776` would then be equivalent to `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 that are difficult for `yacc` to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid `yacc` specifications for their input revealed errors of conception or design early in the program development.

`yacc` has been used extensively in numerous practical applications on the A/UX system, including the syntax checker `lint`, the Portable C Compiler, and a system for typesetting mathematics.

The remainder of this chapter describes:

- Basic process of preparing a `yacc` specification
- Parser operation
- Handling ambiguities
- Handling operator precedence 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 sections that illustrate the earlier material:

- "A Desk Calculator" contains a brief example of using `yacc` to design a simple program.
- "`yacc` Input Syntax" contains a summary of the `yacc` input syntax.
- "An Advanced Grammar" contains an example using some of the more advanced features of `yacc`.
- "Backward Compatibility" contains a description of the mechanisms and syntax that, though no longer actively supported, are provided for historical continuity with older versions of `yacc`.

2. Basic specifications

Names refer to either tokens or nonterminal symbols. `yacc` 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.

Every specification file consists of three sections:

- Declarations
- Grammar rules
- Programs

These sections are separated by double percent symbols (`%%`). The percent symbol is generally used in `yacc` specifications as an escape character.

The following is a syntactic description of a `yacc` specification file:

```
declarations
%%
rules
%%
programs
```

The *declarations* section may be empty, and, if the *programs* section is omitted, the second `%%` mark may also be excluded. The smallest legal `yacc` specification is therefore

```
%%
rules
```

Blanks, tabs, and newlines are ignored, but they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in `/*` and `*/`, as in the C language.

The *rules* section is made up of one or more grammar rules. A grammar rule has the following form:

```
a : body;
```

In this example, *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 the C language, the backslash (\) is an escape character within literals, and all the C language escapes are recognized.

Table 20-1. C language escapes recognized by `yacc`

Escape	**Meaning**
`\n`	Newline
`\r`	Return
`\'`	Single quote (`'`)
`\\`	Backslash (\)
`\t`	Tab
`\b`	Backspace
`\f`	Form feed
\xxx	*xxx* in octal

For a number of technical reasons, the null character (`\0` or `0`) should never be used in grammar rules.

If there are several grammar rules with the same left side, the vertical bar (|) can be used to avoid rewriting the left side. 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` using the vertical bar:

```
A : B C D
  | E F
  | G;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much easier to read and to change.

If a nonterminal symbol matches the empty string, this can be indicated by the following:

```
empty : ;
```

Names representing tokens must be declared in the declarations section. For example,

`%token` *name1 name2*

Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Nonterminal symbols must appear on the left side of at least one rule.

The parser is designed to recognize the nonterminal 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 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. For example,

```
%start symbol
```

The end of the input to the parser is signaled by a special token, called the "end-marker." If the tokens up to but not including the end-marker form a structure that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

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

3. Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens if desired.

An action is an arbitrary C language statement and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements enclosed in braces ({ and }). For example,

```
A : '(' B ')'
   {
    hello( 1, "abc" );
   }
```

and the following is an example of grammar rules with actions:

```
XXX : YYY ZZZ
    {
       printf("a message\n");
       flag = 25;
    }
```

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 pseudovariable $$ to some value.

The following action does nothing except return the value of one:

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudovariables `$1`, `$2`, and so on, which refer to the values returned by the components of the right side of a rule, reading from left to right. For example, if the rule is

```
A : B C D;
```

then `$2` has the value returned by `C`, and `$3` the value returned by `D`.

With the following rule, the value returned is usually the value of the *expr* in parentheses:

```
expr : '(' expr ')'
     {
          $$ = $2 ;
     }
```

By default, the value of a rule is the value of the first element in it (`$1`).

Grammar rules of the following form frequently need not have an explicit action:

```
A : B;
```

In the examples above, all the actions came at the end of rules. Sometimes, though, it is desirable to get control before a rule is fully parsed. The `yacc` program permits an action to be written in the middle of a rule as well as at the end.

This kind of 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 the left of the action. For example, in the following rule `x` is set to 1 (the value returned by the action to its left) and `y` is set to the value returned by `C`:

```
A : B
    {
       $$ = 1;
    }
       C
    {
       x = $2;
       y = $3;
    }
;
```

This is because every component of the right side of the rule, including an action, is associated with a positional pseudovariable, so the `$1` refers to `B`, `$2` to the value returned by the action associated with `B`, `$3` to `C`, and so on.

Actions that do not terminate a rule are actually handled by `yacc` by manufacturing a new nonterminal symbol name and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule.

`yacc` actually treats the preceding example as if it had been written like the following (`$ACT` is an empty action):

```
$ACT   : /* empty */
    {
       $$ = 1;
    }
    ;
A      : B $ACT C
    {
       x = $2;
       y = $3;
    }
    ;
```

In many applications, output is not produced 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.

In the following example, the C function `node` creates a node with label *l* and descendants *n1* and *n2* and returns the index of the newly created node:

```
node(l, n1, n2)
```

Then a parse tree is built by supplying the actions following in the yacc specification file:

```
expr : expr '+' expr
     {
        $$ = node('+', $1, $3 );
     }
```

The user may define other variables to be used by the actions.

Declarations and definitions can appear in the declarations section enclosed in the marks `%{` and `%}`. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{ int variable = 0; %}
```

could be placed in the declarations section, making `variable` accessible to all of the actions.

The `yacc` parser uses only names beginning with `yy`. The user should avoid such names. In these examples, all the values are integers. A discussion of values of other types is found in the section "Arbitrary Value Types."

4. Lexical analysis

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

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by `yacc` or by 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 the following:

```
yylex()
{
    extern int yylval;
    int c;
    ...
    c = getchar();
    ...
    switch( c )
    {
    ...
    case '0':
    case '1':
    ...
    case '9':
        yylval = c - '0';
        return( DIGIT );
        ...
    }
    ...
}
```

The intent is to return a token number of DIGIT and a value equal to the numeric value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT is defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in the 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 by the user. In the default situation, the numbers are chosen by `yacc`. The default token number for a literal character is the numeric 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 non-negative 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 definitions. It is important that all token numbers be distinct.

For historical reasons, the end-marker must have token number 0 or be negative. This token number cannot be redefined by the user. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token upon reaching the end of their input.

The `lex` program is a very useful tool for constructing lexical analyzers. 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 easily be used to produce quite complicated lexical analyzers, but there remain some languages (such as Fortran) that do not fit any theoretical framework and whose lexical analyzers must be crafted by hand. See ``` ``lex`` ``` Reference'' in this manual for more information on `lex`.

5. Parser operation

The `yacc` program turns the specification file into a C language program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex and will not be discussed here. The parser itself, however, is relatively simple, and understanding how it works will make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by `yacc` consists of a finite-state machine with a stack. The parser is also capable of reading and remembering the next input token (called the ``look-ahead token''). The current state is always the one on the top of the stack. The states of the finite-state machine are given small integer labels.

Initially, the machine is in state 0 (the stack contains only state 0) and no look-ahead token has been read. The machine has only four actions available:

`shift`	Push current state onto stack, go into specified new state.
`reduce`	Pop some number of states from stack, push new state, execute user code.
`accept`	End of input has been (successfully) reached.
`error`	An unparseable situation has been detected.

A step of the parser is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. If it needs one and does not have one, it calls `yylex` to obtain the next token.
2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may cause states to be pushed onto the stack or popped off the stack and the look-ahead token to be 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. In the following example, 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):

```
IF shift 34
```

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 side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right side by the left side.

It may be necessary to consult the look-ahead token to decide whether to reduce or not (usually it is not necessary). In fact, the default action (represented by a dot) is often a `reduce` action.

reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. For example, in the following display, the action refers to grammar rule 18:

```
. reduce 18
```

While in this example, the action refers to state 34:

```
IF shift 34
```

Suppose the following rule is being reduced:

```
A  : x  y  z ;
```

The reduce action depends on the left symbol (A in this case) and the number of symbols on the right 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 that 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 and pushed onto the stack, and parsing continues.

There are significant differences between the processing of the left 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 the following, which causes state 20 to be pushed onto the stack and become the current state:

```
A  goto 20
```

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 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 side of the rule is empty, no states are popped off 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 pseudovariables $1, $2, and so on refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker and indicates that the parser has successfully done its job.

The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) will be discussed later.

Consider the following example as a yacc specification:

```
%token  DING  DONG  DELL
%%
rhyme    :    sound  place
         ;
sound    :    DING  DONG
         ;
place    :    DELL
         ;
```

When yacc is invoked with the -v option, a file called y.output is produced with a human-readable description of the parser.

The following example is the y.output file corresponding to the above grammar (with some statistics stripped off the end), where the actions for each state are specified and there is a description of the parsing rules being processed in each state:

```
state 0
        $accept : _rhyme $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept :  rhyme_$end

        $end  accept
        .  error


state 2
        rhyme :  sound_place

        DELL  shift 5
        .  error

        place  goto 4

state 3
        sound :  DING_DONG

        DONG  shift 6
        .  error


state 4
        rhyme :  sound place_     (1)

        .  reduce 1
```

```
state 5
        place :  DELL_    (3)

         .  reduce 3


state 6
        sound :  DING DONG_    (2)

         .  reduce 2
```

The underscore character _ is used to indicate what has been seen and what is yet to come in each rule.

The following input can be used to track the operations of the parser:

```
DING  DONG  DELL
```

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 the following, which is rule 2:

```
sound   :   DING  DONG
```

Two states, 6 and 3, are popped off the stack, uncovering state 0. Consulting the description of state 0 (looking for a `goto` on `sound`), the following is obtained:

```
sound goto 2
```

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 side, so one state (5) is popped off and state 2 is uncovered. The `goto` in state 2 on `place` (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4.

In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a `goto` on `rhyme` causing the parser to enter state 1. In state 1, the input is read and the end-marker is obtained indicated by `$end` in the `y.output` file. The action in state 1 (when the end-marker is seen) successfully ends the parse.

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

6. Ambiguity and conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the following grammar rule 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:

```
expr   :  expr  '-'  expr
```

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

```
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.

Consider the problem that confronts the parser when provided with the following input:

expr - *expr* - *expr*

When the parser has read the second *expr*, the input seen matches the right side of the grammar rule above:

expr - *expr*

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 (displayed in the following example) and again reduce:

- *expr*

The effect of this is to take the left associative interpretation. Alternatively, if the parser sees the following:

expr - *expr*

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

expr - *expr* - *expr*

It could then apply the rule to the rightmost three symbols, reducing them to *expr*, which results in the following being left:

expr - *expr*

Now the rule can be reduced once more. The effect is to take the right associative interpretation. The parser can do one of two legal things, a shift or a reduction. It has no way of deciding between them. This is called a "shift/reduce conflict."

It may also happen that the parser has a choice of two legal reductions. This is called a "reduce/reduce conflict." (Note that there are never any "shift/shift" conflicts.) When there are shift/reduce or reduce/reduce conflicts, `yacc` still produces a parser. It does this by selecting one of the valid steps wherever it has a choice.

A rule describing the choice to make in a given situation is called a "disambiguating rule." The `yacc` program invokes two disambiguating rules by default:

- In a shift/reduce conflict, the default is to do the shift.
- In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

The first rule implies that reductions are deferred in favor of shifts when there is a choice. The second rule 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. Experience has suggested that this rewriting is somewhat unnatural and produces slower parsers. Thus, `yacc` will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider

```
stat : IF '(' cond ')' stat
     | IF '(' cond ')' stat ELSE stat
     ;
```

which is a fragment from a programming language involving an if-then-else statement.

In these rules, `IF` and `ELSE` are tokens, `cond` is a nonterminal symbol describing conditional (logical) expressions, and `stat` is a nonterminal symbol describing statements. The first rule will be called the ``simple-if'' rule and the second the ``if-else'' rule. These two rules form an ambiguous construction because input of the following form can be structured according to these rules in two ways:

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

The input can be structured as in the following example or as in the subsequent example, which is the one given in most programming languages having this construct:

```
IF ( C1 )
{
   IF ( C2 )
            S1
}
ELSE
      S2
```

or:

```
IF  ( C1 )
{
      IF  ( C2 )
            S1
      ELSE
            S2
}
```

Each `ELSE` is associated with the preceding "un-`ELSE`'d" `IF`.

In the following example, consider the situation where the parser has seen the `IF-ELSE` construct and is looking at the `ELSE`.

```
IF ( C1 ) IF ( C2 ) S1
```

It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input

```
ELSE  S2
```

and reduce 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 portion can be reduced by the if-else rule to get the following, which can be reduced by the simple-if rule:

```
   IF  ( C1 )  stat
```

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 have already been seen:

```
   IF  ( C1 )  IF  ( C2 )  S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the "state" of the parser. The conflict messages of yacc are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the above conflict state might be

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE
state 23
        stat   :  IF   (   cond   )   stat_          (18)
        stat   :  IF   (   cond   )   stat_ELSE   stat
        ELSE      shift 45
        .         reduce 18
```

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 that 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 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 following line:

```
     stat   :   IF ( cond ) stat ELSE_stat
```

because the ELSE will have been shifted in this state. In state 23, the alternative action (describing a dot (.)) is to be done if the input symbol is not mentioned explicitly in the actions.

- If the input symbol is not ELSE, the parser reduces to

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

  by grammar rule 18.

Once again, notice that the numbers following `shift` commands refer to other states, while the numbers following `reduce` commands refer to grammar rule numbers.

In the `y.output` file, the rule numbers are printed after those rules which can be reduced. In most states, only one `reduce` action is possible, and it 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.

7. Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity.

It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the following two forms for all binary and unary operators desired:

```
expr   :   expr   OP   expr
```

and

```
expr   :   UNARY   expr
```

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 one of the following yacc keywords: %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. For example,

```
%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. For example, the following is illegal in Fortran and such an operator would be described with the keyword %nonassoc in yacc:

```
A  .LT.  B  .LT.  C
```

As an example of the behavior of these declarations, the following description might be used to structure the subsequent input:

```
%right  '='
%left  '+'  '-'
%left  '*'  '/'

%%

expr    :    expr  '='  expr
        |    expr  '+'  expr
        |    expr  '-'  expr
        |    expr  '*'  expr
        |    expr  '/'  expr
        |    NAME
        ;
```

The following is the input to be structured by the above description in order to perform the correct precedence of operators:

```
a = b = c * d - e - f * g
```

The result of the structuring is as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary minus (–). Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication.

The keyword `%prec` changes the precedence level associated with a particular grammar rule. `%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. The keyword causes the precedence of the grammar rule to become that of the following token name or literal. For example, the following rules might be used to give unary minus the same precedence as multiplication:

```
%left   '+'   '-'
%left   '*'   '/'

%%

expr    :   expr  '+'  expr
            |  expr  '-'  expr
            |  expr  '*'  expr
            |  expr  '/'  expr
            |  '-'  expr        %prec  '*'
            |  NAME
            ;
```

A token declared by `%left`, `%right`, and `%nonassoc` need not be, but may be, declared by `%token` as well.

The precedences and associativities are used by `yacc` to resolve parsing conflicts. They give rise to disambiguating rules. Formally, the rules work as follows:

- The precedences and associativities are recorded for those tokens and literals that have them.
- 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.
- 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.
- 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.

8. Error handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output. It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser ''restarted'' after an error.

A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue. To allow the user some control over this process, `yacc` provides a simple but reasonably general feature. The token name `error` is reserved for error handling. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place.

The parser pops its stack until it enters a state where the token `error` is legal. It then behaves as if the token `error` were the current look-ahead token and performs the action encountered. The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

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 following form means that on a syntax error the parser attempts to skip over the statement in which the error is seen:

```
   stat    :   error
```

More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and starts 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, and so on. Error rules such as the above are very general but difficult to control. Rules such as the following are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon:

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

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 is one way to do this:

```
   input   :  error   '\n'
               {
                  printf("Reenter last line: ");
               }
                 input
            {
               $$ = $4;
            }
           ;
```

There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable. For this reason, there is a mechanism that can force the parser to believe that error recovery has been accomplished. The following statement in an action resets the parser to its normal mode:

```
   yyerrok ;
```

The last example can be rewritten, somewhat more usefully, as the following:

```
   input  :  error  '\n'
                {
                   yyerrok;
                   printf("Reenter last line: ");
                }
                input
          {
              $$ = $4;
          }
          ;
```

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 following statement in an action will have this effect:

```
   yyclearin ;
```

For example, suppose the action after `error` were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by `yylex` is presumably the first token in a legal statement. The old illegal token must be discarded and the error state reset. A rule similar to the one following could perform this:

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

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

9. The yacc environment

When the user enters a specification to `yacc`, the output is a file of C language programs, called `y.tab.c`. The function produced by `yacc` is an integer-valued function called `yyparse`. When it is called, it in turn repeatedly calls `yylex`, the lexical analyzer supplied by the user (see "Lexical Analysis"), to obtain input tokens.

Eventually, if an error is detected, `yyparse` returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, `yyparse` returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C language program, a program called `main` must be defined that eventually calls `yyparse`. Also needed is a routine called `yyerror` which prints a message when a syntax error is detected. These two routines (`main` and `yyerror`) 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`. Use `ld`'s `-ly` option to incorporate these routines into your program. The following source code examples show the simplicity of these routines:

```
main()
{
   return ( yyparse() );
}
```

and

```
#include <stdio.h>

yyerror(s)
char *s;
{
   fprintf( stderr, "%s\n", s );
}
```

The argument to `yyerror` is a string containing an error message, usually the string `syntax error`. The average application wants to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected.

The external integer variable `yychar` contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics.

Because the `main` program is probably supplied by the user (to read arguments, and so on), 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 send as 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 `yydebug` by using a debugging system.

10. 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:

- Use all uppercase letters for token names and all lowercase letters for nonterminal names.

- Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
- Put all rules with the same left side together. Put the left side in only once and let all following rules begin with a vertical bar.
- Put a semicolon only after the last rule with a given left side and put the semicolon on a separate line. This allows new rules to be easily added.
- Indent rule bodies by two tab stops and action bodies by three tab stops.

The example in "Example: A Desk Calculator" is written following this style (where space permits). You must make up your own mind about these stylistic questions. The central problem, however, is to make the rules visible through the morass of action code.

11. Left recursion

The algorithm used by the `yacc` parser encourages so called "left recursive" grammar rules. Rules of the following form match this algorithm:

```
name    :    name   rest-of-rule   ;
```

Rules such as the two following 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:

```
list    :    item
        |    list ',' item
        ;
```

and

```
seq     :    item
        |    seq   item
        ;
```

With right recursive rules, such as the following, the parser is a bit bigger, and the items are seen and reduced from right to left:

```
seq     :    item
        |     item   seq
        ;
```

More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. The user should use left recursion wherever reasonable.

It is worth considering if a sequence with zero elements has any meaning; if so, consider writing the sequence specification as in the following, using an empty rule:

```
seq     :    /* empty */
        |     seq   item
        ;
```

Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if `yacc` is asked to decide which empty sequence it has seen when it hasn't seen enough to know.

12. Lexical considerations

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. The following example specifies a program that consists of zero or more declarations followed by zero or more statements. The flag `dflag` is 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.

```
%{
   int dflag;
%}
  ... other declarations ...

%%

prog  :   decls  stats
        ;

decls :   /* empty */
        {
           dflag = 1;
        }
        |   decls  declaration
        ;

stats :   /* empty */
        {
           dflag = 0;
        }
        |   stats  statement
        ;

  ... other rules ...
```

This kind of "back-door" approach can be elaborated to an unpleasant degree. Nevertheless, it represents a way of doing some things that are difficult if not impossible to do otherwise.

13. Reserved words

Some programming languages permit you to use words (like `if`) that 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 section, but it is difficult. A number of ways of making this easier are being studied. For the time being, it is better that the keywords be reserved, that is, forbidden for use as variable names.

14. Simulating error and accept in actions

The parsing actions of error and accept can be simulated in an action by use of the 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. The function `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.

15. Accessing values in enclosing rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is the same as with ordinary actions, a dollar sign followed by a digit.

```
sent     :   adj  noun  verb  adj  noun
         {
            look at the sentence ...
         }
         ;
adj      :   THE
         {
            $$ = THE;
         }
            |   YOUNG
         {
            $$ = YOUNG;
         }
            ...
         ;
noun    :   DOG
         {
            $$ = DOG;
         }
           |   CRONE
         {
            if( $0 == YOUNG )
            {
               printf( "what?\n" );
            }
            $$ = CRONE;
          }
          ;
             ...
```

In this case, the digit may be 0 or negative.

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol noun in the input.

There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism prevents a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

16. 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. The yacc program keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type checked.

The yacc value stack is declared to be a union of the various types of values desired. The user declares the union and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, yacc automatically inserts the appropriate union name so that no unwanted conversions take place. This makes type-checking commands such as lint much quieter.

Three mechanisms are used to provide for this typing:

- First, there is a way of defining the union. This must be done by the user because 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 nonterminal symbols.
- Third, there is a mechanism for describing the type of those few values where `yacc` cannot easily determine the type.

To declare the union, the user includes the following in the declaration section:

```
%union
{
    body of union
}
```

This declares the `yacc` value stack and the external variables `yylval` and `yyval` to have type equal to this union. If `yacc` was invoked with the `-d` option, the union declaration is copied onto the `y.tab.h` file. Alternatively, the union may be declared in a header file, and a `typedef` used to define the variable `YYSTYPE` to represent this union. Thus, the header file might have said the following, instead:

```
typedef union
{
    body of union
}
YYSTYPE;
```

The header file must be included in the declarations section by use of `%{` and `%}`. Once `YYSTYPE` is defined, the union member names must be associated with the various terminal and nonterminal names.

The following construction is used to indicate a union member name:

<name>

If this follows one of the keywords `%token`, `%left`, `%right`, or `%nonassoc`, the union member name is associated with the tokens listed. For example, the following causes any reference to values returned by these two tokens to be tagged with the union member name `optype`:

```
%left  <optype>  '+'  '-'
```

Another keyword, `%type`, is used to associate union member names with nonterminals. For example, the following may be used to associate the union member `nodetype` with the nonterminal symbols *expr* and `stat`.

```
%type   <nodetype>   expr   stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no a priori type. Similarly, reference to left context values (such as `$0`) leaves `yacc` with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between "<" and ">" immediately after the first `$`, as in the following example:

```
rule    :    aaa
          {
             $<intval>$ = 3;
          }
             bbb
          {
             fun( $<intval>2, $<other>0 );
          }
          ;
```

This syntax has little to recommend it, but the situation arises rarely. A sample specification is given in "Example: An Advanced Grammar." 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.

17. Example: a desk calculator

This section contains an example that gives the complete `yacc` applications for a small desk calculator. The calculator has 26 registers labeled `a` through `z` and accepts arithmetic expressions made up of the following operators:

Table 20-2. Arithmetic operators

Symbol	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus (Remainder)
&	Binary AND
|	Binary OR
=	Assignment

If an expression at the top level is an assignment, the value is printed. Otherwise, the expression is printed. As in the C language, an integer that begins with 0 (zero) is assumed to be octal. Otherwise, it is assumed to be decimal.

As an example of a `yacc` specification, the desk calculator does a reasonable job of showing how precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer is much simpler than what is necessary for most applications, and the output is produced immediately line by line.

Note the way that decimal and octal integers are read in by grammar rules. This job is probably better done by the lexical analyzer.

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

int regs[26];
int base;

%}
%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
```

```
%left '*' '/' '%'
%left UMINUS  /* precedence for unary minus */
%%           /* beginning of rule section */
list    :    /* empty */
        |    list stat '\n'
        |    list error '\n'
        {
             yyerror;
        }
        ;
stat    :     expr
        {
              printf( "%d\n", $1 );
        }
        |     LETTER '=' expr
        {
              regs[$1] = $3
        }
        ;
expr    :    '(' expr ')'
        {
              $$ = $2;
        }
        |     expr '+' expr
        {
              $$ = $1 + $3
        }
        |     expr '-' expr
        {
              $$ = $1 - $3
        }
        |     expr '*' expr
        {
              $$ = $1 * $3;
        }
        |     expr '/' expr
        {
              $$ = $1/$3;
        }
        |     exp '%' expr
        {
              $$ = $1 % $3
        }
        |     expr '&' expr
        {
              $$ = $1 & $3;
        }
        |     expr '|' expr
        {
              $$ = $1 | $3
        }
        |     '-' expr  %prec UMINUS
        {
              $$ = - $2;
        }
        |     LETTER
        {
              $$ = reg[$1];
        }
        |     number
        ;
number  :    DIGIT
        {
                $$ = $1; base = ($1==0) ? 8 : 10;
        }
        |    number DIGIT
        {
                $$ = base * $1 + $2
        }
        ;
%% /* start of program */
/*
 * lexical analysis routine
 * return LETTER for lowercase letter
 *    (i.e., yylval = 0 through 25)
 * returns DIGIT for digit
 *    (i.e., yylval = 0 through 9)
 * all other characters are returned immediately
 *
 */
yylex( )
{
  int c;
    while (c=getchar( )) == ' ') /* skip blanks */
        ;
        if( islower( c ))
        {
            yylval = c - 'a';
                return( LETTER );
        }
        if( isdigit( c ))
        {
            yylval = c - '0';
                return( DIGIT );
        }
  return( c );
}
```

18. Example: yacc input syntax

This section contains a description of the yacc input syntax as a yacc specification. Context dependencies, and so forth, are not considered. Ironically, the yacc input specification language is most naturally specified as an LR(2) grammar. The sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which just happens to have an action embedded in it.

As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, comments, and so on) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERs but never as part of C_IDENTIFIERs.

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

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

/* reserved words: */
/* %type -> TYPE,  %left -> LEFT,   etc. */
%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION
%token MARK           /* the %% mark */
%token LCURL          /* the %{ mark */
%token RCURL          /* the %} mark */

/* ASCII character literals stand for themselves */
%token spec

%%

spec    :   defs MARK rules tail
        ;
tail    :   MARK
        {
           In this action, eat up the rest of the file
        }
        |   /* empty: the second MARK is optional */
        ;
defs    :   /* empty */
        |   defs def
        ;
defs    :   START IDENTIFIER
        |   UNION
        {
           Copy union definition to output
        }
        |   LCURL
        {
           Copy C code to output file
           RCURL
        }
        |   ndefs rword tag nlist
        ;
rword   :   TOKEN
        |   LEFT
        |   RIGHT
        |   NONASSOC
        |   TYPE
        ;
tag     :   /* empty: union tag is optional */
        |   '<' IDENTIFIER '>'
        ;
nlist   :   nmno
        |   nlist nmno
        |   nlist ',' nmno
        ;
/* Note: literal illegal with %type */
nmno    :   IDENTIFIER
        |   IDENTIFIER NUMBER
        ;
/* rule section */

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

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

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

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

19. Example: an advanced grammar

This section gives an example of a grammar using some of the advanced features. It modifies the example from "Example: A Desk Calculator" to provide a desk calculator that does floating-point interval arithmetic.

The calculator understands floating-point constants, as well as the arithmetic operations +, −, *, /, unary −, and the letters a through z. The calculator also understands intervals written as is the following example, where X is less than or equal to Y:

```
(X,Y)
```

There are 26 interval valued variables A through Z that may also be used. The usage is similar to that in "Example: A Desk Calculator." That is, assignments return no value and print nothing while expressions print the floating or interval value.

Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, INTERVAL, by using typedef. The yacc value stack can also contain floating-point scalars and integers that are used to index into the arrays holding the variable values. The entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the actions call functions that return structures as well.

Note the use of YYERROR to handle error conditions: division by an interval containing 0 and an interval presented in the wrong order. The error-recovery mechanism of yacc is used to throw away the rest of the offending line. In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Scalars 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 following input lines:

```
2.5+(3.5-4.)
```

and

```
2.5 + ( 3.5,4 )
```

Notice that the `2.5` is to be used in an interval-value expression in the second example, but this fact is not known until the comma is read. By this time `2.5` is finished, and the parser cannot go back and change its mind.

More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator, one when the left operand is a scalar and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion is applied automatically.

Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file. In this way, the conflict is resolved in the direction of keeping scalar-valued expressions scalar valued until they are forced to become intervals. This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating-point constants. The C language library routine `atof` is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser and thence error recovery.

```
%{
```

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

typedef struct interval
{
   double lo, hi;
}  INTERVAL;

INTERVAL vmul(), vdiv( );

double atof();
double dreg[26];
INTERVAL vreg[26];

%}

%start line

%union
{
  int ival;
  double dval;
  INTERVAL vval;
}

%token <ival> DREG VREG /*indices into dreg, vreg */
%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 '=' '\n'
        {
          dreg[$1] = $3;
        }
        |  VREG '=' vexp '\n'
        {
          vreg[$1] = $3;
        }
        |  error '\n'
        {
          yyerrork;
        }
        ;

dexp    :  CONST
        |  DREG
        {
          $$ = dreg[$1]
        }
        |  dexp '+' dexp
        {
          $$ = $1 + $3
        }
        |  dexp '-' dexp
        {
          $$ = $1 - $3
        }
        |  dexp '*' dexp
        {
```

```
                  $$ = $1 * $3
                }
                |  dexp '/' dexp
                {
                  $$ = $1 / $3
                }
                |  '-' dexp    %prec UMINUS
                {
                  $$ =- $2
                }
                |  '(' dexp ')'
                {
                  $$ = $2
                }
                ;

vexpp           :  dexp
                {
                  $$.hi = $$.lo = $1;
                }
                |  '(' dexp ',' dexp ')'
                {
                  $$.lo = $2;
                  $$.hi = $4;
                  if( $$.lo > $$.hi )
                  {
                    printf( "interval out of order n" );
                    YYERROR;
                  }
                }
                |   VREG
                {
                  $$ = vreg[$1]
                }
                |  vexp '+' vexp
                {
                  $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo
                }
```

```
|  dexp '+' vexp
{
  $$.hi = $1 + $3.hi;
  $$.lo = $1 + $3.lo
}
|  vexp '=' vexp
{
  $$.hi = $1.hi - $3.lo;
  $$.lo = $1.lo - $3.hi
}
|  dvep '-' vdep
{
  $$.hi = $1 - $3.lo;
  $$.lo = $1 - $3.hi
}
|  vexp '*' vexp
{
  $$ = vmul( $1.lo,$.hi,$3 )
}
|  dexp '*' vexp
{
  $$ = vmul( $1, $1, $3 )
}
|  vexp '/' vexp
{
  if( dcheck( $3 ) ) YYERROR;
  $$ = vdiv( $1.lo, $1.hi, $3 )
}
|  dexp '/' vexp
{
  if( dcheck( $3 ) ) YYERROR;
  $$ = vdiv( $1.lo, $1.hi, $3 )
}
|  '-' vexp    %prec UMINUS
{
  $$.hi = -$2.lo;$$.lo =-$2.hi
}
|  '(' vexp ')'
}
```

```
                $$ = $2
              }
              ;
%%
/* buffer size for floating point number */
# define BSZ 50
/*
 *lexical analysis
 */

yylex( )
{
        register c;
        while ((c=getchar()) == ' ') /* skip blanks */
                      ;
        if(isupper(c))
        {
            yylval.ival = c - 'A'
            return(VREG);
        }
        if(islower(c))
        {
            yylval.ival = c - 'a',
            return(DREG);
        }
/*
 * gobble up digits, points, exponents
 */
        if(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)
                     /* causes syntax error */
                     return( '.' );
                  continue;
                }
                if(c == 'e')
                {
                  if( exp++ )
                     /* causes syntax error */
                     return( 'e' );
                  continue;
                }
                break;                  /* end of number */
            }
            *cp = '\0';
            if((cp - buff) >= BSZ)
                printf( "constant too long truncated\n");
            else
                /* push back last char read */
                ungetc(c, stdin);
            yylval.dval = atof(buf);
            return(CONST);
        }
        return(c);
}
/*
 * returns the smallest interval
 * between a, b, c and d
 */

INTERVAL hilo( a, b, c, d )
double a, b, c, d;
{
    INTERVAL v;
    if( a>b )
    {
        v.hi = a;
        v.lo = b;
    }
    else
    {
        v.hi = b;
        v.lo = a;
    }
    if( c>d )
    {
        if( c>v.hi )
           v.hi = c;
        if( d<v.lo )
           v.lo = d;
    }
    else
    {
        if( d>v.hi )
           v.hi = d;
        if( c<v.lo )
           v.lo = c;
    }
    return( v );
}
INTERVAL vmul( a, b, v )
double a, b;
INTERVAL v;
{
    return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
}

dcheck( v )
INTERVAL v;
{
    if( v.hi >=0.&& v.lo <=0. )
    {
       printf( "divisor internal contains 0.\n" );
       return( 1);
    }
    return( 0 );
}
```

```
INTERVAL vdiv( a, b, v )
double a, b;
INTERVAL v;
{
   return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
}
```

20. Backward compatibility

This section mentions synonyms and features that are supported for historical continuity but, for various reasons, are not encouraged.

- Literals may also be delimited by double quotes.
- Literals may be more that one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined just as if the literal did not have the quotes around it.

  Otherwise, it is difficult to find the value for such a literal.

  The use of multicharacter literals is likely to mislead those unfamiliar with `yacc`, because it suggests that `yacc` is doing a job that must actually be done by the lexical analyzer.
- Most places where (`%`) is legal, the backslash (\) may be used. In particular, \\ is the same as `%%`, `\left` the same as `%left`, and so on.
- There are a number of other synonyms:

`%<`	is the same as	`%left`
`%>`	is the same as	`%right`
`%binary`	is the same as	`%nonassoc`
`%2`	is the same as	`%nonassoc`
`%0`	is the same as	`%token`
`%term`	is the same as	`%token`
`%=`	is the same as	`%prec`

- Actions may also have the form

  ```
  ={ ... }
  ```

  and the braces can be dropped if the action is a single C language statement.

- C language code between `%{` and `%}` used to be permitted at the head of the rules section as well as in the declaration section.

Chapter 21
bc Reference

Contents

Tables

Chapter 21

bc Reference

1. bc: a basic calculator

bc is a specialized language and compiler for handling arbitrary-precision arithmetic. bc calls the dc calculator program to do any actual computations. In fact, bc was designed specifically to augment dc routines for manipulating infinitely large numbers, scaled up to 99 decimal places.

Because bc is based on a dynamic storage allocator, overflow does not occur until all available core storage is exhausted. bc has a complete control structure, and can be used either in immediate mode (direct immediate input/output to and from bc) or as an interactive processor for bc programs. Consequently, complex functions can be defined and saved in a file for later execution. A small library of predefined functions is also available, among which are the sine, cosine, arctangent, logarithmic, exponential, and Bessel functions of integer order.

bc contains scaling provisions that permit the use of decimal-point notation, as well as input and output in bases other than base 10. Numbers can be converted from decimal to octal simply by setting the output base to eight. The limit on the number of digits that can be manipulated depends only on the amount of core storage available.

While bc is not intended as a complete programming language, it can be used effectively to do a number of specific tasks, most notably the following:

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

2. Using bc

In this chapter we use the term "bc command" to refer to the command you type from the shell command line, and the term "bc program" to refer to the set of calculations to be performed by the bc command. These calculations can reside in a bc program file.

2.1 bc command syntax

The bc command has the following syntax:

```
bc [-c] [-l] [file]
```

The -c compile-only option directs bc to output what it would normally pass as input to dc. The output is instructive but complicated.

The -l (library) option calls bc's own set of math library functions:

Function syntax	Operation
s(*x*)	Sine
c(*x*)	Cosine
a(*x*)	Arctangent
l(*x*)	Natural logarithm
e(*x*)	Exponential
j(*n*,*x*)	Bessel function integer order

The library option initially sets the scale (number of available decimal places after the decimal point) to 20, but this can be reset using the scale function call. See the section "scale."

The *file* is an optional bc program file which bc can read calculations from.

2.2 Entering a program at the terminal

For the immediate evaluation of simple arithmetic expressions that do not involve standard bc library functions or do not require any user-defined functions, simply enter the bc program at the terminal. For example, to perform a simple operation, first invoke bc and then enter the calculation to be done:

```
bc
142857 + 285714
```

bc then responds immediately with the result

```
428571
```

2.3 Program files

For more complicated calculations, you may find it more efficient to define the functions or procedures in a program file. You would then pass the filename as an argument to the bc command:

bc *filename*

bc then reads and executes the contents of the named file before accepting further commands from the keyboard.

2.4 Exiting bc

To exit from bc, even when using a command file, you must issue a quit or an end-of-file character (see stty(1) in *A/UX Command Reference* for more information). Unless you use the syntax "bc < *filename*," bc will not exit when it reaches the end of the program file. If no quit statement is given, bc simply waits for further instructions, and your shell prompt is not returned.

To exit, you can either place a quit statement at the end of your file or enter quit or your end-of-file character directly when bc has completed the file. Your end-of-file character can still be used as an interrupt and terminate signal while the file is being processed.

The quit statement is not treated as an executable statement, and so cannot be used in a function definition or in an if, for or while statement.

3. Program syntax

The syntax of a bc program is very similar to that of a C language program. In general, statements and control structures are identical in bc and in C. A good example of this similarity is the manner in which a bc function is defined. The following program defines a function that computes the approximate value of the exponential function and prints the result for the first ten integers. The pieces of this example are discussed in individual sections below.

```
scale = 10
define e(x) {
     auto a,b,c,i,s
     a = 1
     b = 1
     s = 1
     for(i=1; 1==1; i++) {
          a = a*x
          b = b*i
          c = a/b
          if (c == 0) return(s)
          s = s+c
        }
   }
for(i=1; i<=10; i++) e(i)
```

3.1 Comments

The characters / and * introduce a comment that terminates with the characters * and /. Anything between the asterisks is ignored by the bc compiler.

3.2 Constants

Constants are primitive expressions and 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.

3.3 Keywords

The following are reserved as bc keywords, and cannot be used other than for their predefined purposes:

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

3.4 Identifiers

In bc, an identifier is a character, or sequence of characters, that names an expression. The identifier is the ''place'' where the value of that expression is stored. Therefore, identifiers are legal on the left side of an assignment statement.

bc has three kinds of identifiers:

- Simple identifiers
- Function calls
- Array, or subscripted, variables

All three types should be indicated with single lowercase letters. Identifier names do not conflict; a bc program may have a simple variable identifier named x, an array named x, and a function named x, all of which are separate and distinct.

3.5 Defining functions

Functions are specified by a single lowercase letter, followed immediately by a set of parentheses:

```
a()
```

Since function names are permitted to coincide with simple variable names, the parentheses indicate the difference between a function and a variable, and provide a means of passing arguments to the function. Twenty-six different defined functions are permitted in addition to the 26 variable names.

A function is defined in the following manner:

```
define a(x) {
            defining statements
            return
     }
```

The word define initiates the function definition; a(x) names the function and indicates that the function requires one argument; the left brace opens the body of the definition and must occur on the same line as the define keyword; return returns control to the calling function; and the right brace closes the definition. The body of the definition must contain one or more statements, and must begin and close with a left and right brace, respectively.

3.5.1 Function calls and function arguments

A function call consists of the function name followed by parentheses, which in turn should contain any required arguments to be passed to the function. Individual arguments should each be separated by commas. Functions with no arguments are called and defined using empty parentheses. If a function is called with the wrong number of arguments, the result is unpredictable.

All function arguments are passed by value, and as a result the values remain discrete, local to the called function. Therefore, changes made to the argument values within the called function do not alter the original parameters outside the function.

3.5.2 The `return` statement

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 following two forms:

```
return
return(x)
```

In the first case, the value returned from the function is 0; in the second, the value returned from the function is the expression in parentheses.

3.6 Automatic variables

Automatic variables are allocated space and initialized to 0 on entry to the function, and thrown away on return (exit). The values of any similarly named variables outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected.

It should be noted, however, that automatic variables in `bc` do not work exactly the same way as they do in the C language. On entry to a function, the old values of automatic variables or parameters named previously are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

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

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

There can be only one such `auto` statement in a function, and it must be the first statement in the definition.

The following is an example of a function definition that uses an automatic variable:

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

When called, the value of this function `a` is the product of its two arguments, `x` and `y`. Consequently, the input

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

would send the result, 21.98, to the standard output. Using this same function, the input

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

would send the result, 60, to the standard output.

3.7 Global variables

There are only two storage classes in `bc`: automatic variables and global variables. Unlike automatic variables, global variables retain their values between function calls, and are available to all functions. However, both types have initial values of 0.

3.8 Arrays or subscripted variables

An array, also referred to as a subscripted variable, is indicated with a single lowercase letter (the array name) followed by an expression in brackets (the subscript). For example,

`f[`*expression*`]`

The names of arrays can coincide with simple variable names or function names without conflicting. The subscript values must be greater than or equal to 0 and less than or equal to 2047; any fractional part of a subscript is discarded before use. Only one-dimensional arrays are permitted.

Subscripted variables may be used in expressions, function calls, and 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. For example,

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

When an array name is declared automatic, the entire contents of the array are copied for the use of the function and thrown away on exit from the function. Such array names, used with empty brackets and referring to whole arrays, cannot be used in any context other than that shown above.

3.9 Statements

A statement is any direct instruction. Statements can be grouped together by surrounding them with braces, as in the body of a function definition:

```
define a(x) {
        statement
        statement;  statement
        return
    }
```

When statements are grouped, each individual statement must end with a semicolon or a newline to distinguish it from the next. Except where altered by control statements (such as a `while` loop), execution of grouped statements is sequential.

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

The following is a basic dictionary of `bc` predefined statements:

"*string*"
: The quote statement prints the *string* contained within the quotes.

`break`
: The `break` statement causes termination of a `for` or `while` statement.

`auto` *identifier*[, *identifier*] ...
: The `auto` statement causes the values of one or more identifiers to be pushed down on the stack. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by

following the array name with empty brackets. The `auto` statement must be the first statement in a function definition.

`define` *function-name* `(`[*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 brackets.

`return`
`return (`*expression*`)`

The `return` statement causes the following:

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

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

`quit`

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

`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 0.

`scale (`*expression*`)`

The result is the number of available decimal places after the decimal point in the expression. The scale of the result is 0.

3.10 Assignment statements

`bc` assignment statements work in exactly the same manner as they do in the C programming language. The following table lists the assignment statement constructs:

Table 21-1. Assignment statements

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

Note: In some of these constructs, spaces are significant. There is an important difference between `x=-y` and `x= -y`. The first replaces `x` by `x-y` and the second replaces `x` by `-y`.

All assignment operators are interpreted from right to left. The variables in an assignment statement should have single lowercase letter names. Ordinary variables are used as internal storage registers to hold integer values, and have an initial value of 0. The statement

```
x=x+3
```

has the effect of increasing by three the value of the contents of register `x`. In this case, although the increase in value is performed, that value is not printed. To print the value of `x` after the assignment, either explicitly call `x`, as in the following:

```
x=x+3
x
```

or surround the assignment with parentheses. The latter instructs bc to treat the statement as the value of the result of the operation. The assignment can then be used anywhere an expression can be used. For example,

```
(x=x+3)
```

In this example, the value of x is incremented and the resulting value is printed.

The value of an assignment statement can be used even when it is not placed within parentheses. For example,

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

instructs bc to increment i before using it as a subscript and then assign the resulting value to x.

Since each variable register name must be a unique, single lowercase letter, there can be only 26.

3.11 Control statements

The if, while, and for control statements are available in bc to alter the flow within programs or to cause iteration. They can be used individually as a simple statement or grouped to form a compound statement. A compound statement consists of a collection of statements enclosed in braces, as in a function definition.

3.11.1 Relational operators

Unlike all other operators, the bc relational operators are valid only as the object of an if or while statement or inside a for statement. Similarly, all control structures rely at least in part on the evaluation of a relational statement or expression.

The following table illustrates the six relational operators and their definitions:

Table 21-2. Relational operators

Operator	Definition
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Note: Do not use = 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 = operator is an assignment operator.

3.11.2 The `if` statement

The `if` statement is a conditional statement that causes execution of its instruction if and only if the relation is true. Then control passes to the next statement in sequence. The following is the standard format for an `if` statement in `bc`:

```
if (relation)  statement
```

3.11.3 The `while` statement

`while` causes repeated execution of its instruction as long as the relation tests as true. The relation is tested before each execution of its range; if the result is true, the body of the `while` statement is executed, and the loop continues. If the relation is false, control passes to the next statement beyond the range of the `while` statement. The following is the standard format for the `while` statement in `bc`:

```
while (relation)  {
            statement
            statement
            ...
    }
```

3.11.4 The `for` statement

The typical use of a `for` statement is for controlled iteration. For example,

`for(`*expression1*`;` *relation*`;` *expression2*`)` *statements*

The `for` statement begins by executing *expression1*. Then the *relation* is tested. If the *relation* is true, the *statements* in the body of the `for` are executed. Then *expression2* is executed. The *relation* is then tested, and so forth until the relational test fails.

The following is an example (in immediate mode) of proper use of the `for` statement. In this example, the function returns the factorial of the integer given as input.

```
define f(n) {
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
f(5)
120
f(3)
6
```

3.12 Expressions

The simplest `bc` expression is a single digit. An expression can consist of any number of operators and operands provided they represent a value.

The following are important points to remember when using expressions in `bc`:

- Any term in an expression may be prefixed by a minus sign to indicate that it is a negative (the unary minus sign).
- The value of an expression is printed unless the main operator is an assignment.
- Division by 0 produces an error comment.

The following is a table of the operators that can be used in `bc` expressions, in order of precedence:

Table 21-3. Operators and their precedence

Operator	Function
`^`	Exponentiation

`*`
`%`
 	
`/`
	Multiplication
	Remaindering (integer result truncated toward 0)
	Division

`+` `-`	Addition Subtraction
`=`	Assignment

In the above table, operators with the same precedence are grouped together.

Contents of parentheses are evaluated before items outside the parentheses. Exponentiations are performed from right to left, while the other operations are performed from left to right.

> `a^b^c` and `a^(b^c)` are equivalent
>
> `a-b*c` is the same as `a-(b*c)`
>
> `a/b*c` is equivalent to `(a/b)*c` because the expression is evaluated from left to right.

Following are brief descriptions of the various types of expressions recognized by `bc`:

-expression	The result is the negative of the *expression.*
++expression	The *expression* is incremented by one. The result is the value of the *expression* after incrementing.
--expression	The *expression* is decremented by one. The result is the value of the *expression* after decrementing.
expression++	The *expression* is incremented by one. The result is the value of the *expression* before

incrementing.

expression--	The *expression* is decremented by one. The result is the value of the *expression* before decrementing.
expression^*expression*	The result is the first *expression* raised to the power of the second *expression.* The second *expression* must be an integer. If *a* is the scale of the left expression and *b* is the absolute value of the right expression, then the scale of the result is `min(`*a*`*`*b*`,max(scale,`*a*`))`
*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 `%` (modulus) operator produces the remainder of the division of the two *expressions*. More precisely, *a*%*b* has the same value as *a*–((*a*/*b*)**b*). The scale of the result is the sum of the scales of the quotient and the divisor.

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*.

3.13 Input and output bases: **`ibase`** and **`obase`**

`bc` possesses a scaling provision that enables it to work in bases other than decimal. In addition, input and output can be set to different bases, for automatic conversion from one base to another. `ibase` handles the conversion for input, and `obase` for output.

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

3.13.1 **`ibase`**

The setting for `ibase` determines the base used for interpreting input, and is initially set to 10 (decimal). To set `ibase` to another base, use the = assignment operator. For example, the following sets the input base to base 8:

```
ibase = 8
```

Assuming that the output base is set to decimal, with the `ibase` now set to octal, the input

```
11
```

would automatically produce the following output:

```
9
```

If at this point you want to change the input base back to decimal, you must compensate for the fact that input is now being interpreted as octal. So, in setting the new base, you must use the correct octal value:

```
ibase = 12
```

Because the `ibase` is still set to octal, it will interpret the 12 as an octal 10, and reset the base to decimal. Until reset again, `ibase` will then interpret all input in decimal.

For handling hexadecimal notation, the characters A through F are permitted in numbers (regardless of what base is in effect) and are interpreted as digits having values 10 through 15, respectively. The statement

```
ibase = A
```

changes the base to decimal regardless of the current input base.

`ibase` can handle base settings from 1 to 16. If larger or smaller settings are attempted, `ibase` disregards them. There is no error message to this effect, and the last valid setting remains intact.

3.13.2 obase

The setting for `obase` is used for interpreting the output base, and is initially set to 10 (decimal). Assuming that `ibase` is set to 10,

```
obase = 16
1000
```

produces the following output:

```
3E8
```

thus providing a simple decimal-to-hexadecimal conversion facility.

Very large output bases are permitted and are sometimes useful; for example, large numbers can be generated in groups of five digits by setting `obase` to 100000. Very large numbers are split across lines with 70 characters per line. To force the continuation of a line, end it with a backslash (\).

Decimal output conversion is practically instantaneous, but output of very large numbers (that is, more than 100 digits) with other bases is rather slow. Nondecimal output conversion of a 100-digit number takes about 3 seconds.

3.14 scale

The number of digits after the decimal point of a number is referred to as its scale. `bc` can handle numbers possessing up to 99 decimal places. The initial default setting for `scale` is 0. When the library option is invoked, however, the default is automatically set to 20. To set `scale` to a specific value, use the following statement:

```
scale = n
```

where *n* equals the new value of the `scale` setting. The contents of `scale` must be no greater than 99 and no less than its initial value of 0. However, appropriate scaling can be arranged when more than 99 fraction digits are required.

When two scaled numbers are combined by means of an arithmetic operation, the scale of the result is 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 (not rounding) is performed.

The value held in `scale` can be used in expressions just like other variables. The expression

```
scale = scale + 1
```

increases the value of `scale` by 1, and the statement

```
scale
```

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

It should be noted that, regardless of the `ibase` or `obase` settings, the `scale` setting is always interpreted in decimal base.

Chapter 22
dc Reference

Contents

Tables

Chapter 22

dc Reference

1. dc: a desk calculator

dc is an interactive desk calculator program for handling arbitrary-precision integer arithmetic. It has provisions for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The dc program works like a stacking calculator using reverse Polish notation. Ordinarily, dc operates on decimal integers; however, the input base, output base, and scale can be set according to user specifications. Because dc is based on a dynamic storage allocator, number size is limited only by available core storage.

dc can also be used in conjunction with bc, a high-level language and compiler designed specifically as a front-end for dc. Complex functions can be defined and saved in a file for later execution through bc. When a program is executed, bc compiles the input and automatically pipes it to the dc interpreter, which produces the final result. See ''bc Reference'' in this manual for more information.

2. Using dc

To begin using dc, simply type its name to the shell:

```
    dc
```

Anything you then enter will be interpreted as dc input, up to an end-of-file (CONTROL-d). You can also exit dc by using the q command, discussed later.

For very complex computations, you may find it more efficient to place the instructions into a file. You can then pass the filename as an argument to the dc command:

    dc *filename*

dc will read and execute the contents of the *filename* argument before accepting further commands from the keyboard.

dc operates like a stacking calculator using reverse Polish notation. Initially, the value of a number is pushed onto the stack. The top two values on the stack may then be added (+), subtracted (−), multiplied (*), divided (/), remaindered (%), or exponentiated (^), according to the current operator. The two entries are popped off the stack, and the result is pushed on the stack in their place.

Similarly, the top value on the stack may be duplicated, removed, stored in a register, and so forth. For the full list of operations, see below.

2.1 Command syntax

You can have any number of commands on a line. Blanks and newline characters are ignored, except when used to delineate numbers and in places where a register name is expected. Tabs are not allowed.

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). A negative number can be indicated by preceding a number with an underscore (_). Numbers may also contain decimal points.

To perform simple operations, you can use the following format:

```
24.2 56.2 + p
```

The p command instructs dc to print the result of the computation (in this case, an addition). Here is an example of a more complex problem, using a variety of commands:

```
[ la 1+ d sa * p la 10 >y ] sy
0 sa
ly x
```

This example prints the first ten values of the factorial function (that is, 1! through 10!). To fully understand how it does so, please see "Programming dc."

2.1.1 Operators

Following is a table of the operators that can be used in dc expressions:

Table 22-1. dc operators

Operator	Function
^	Exponentiation
*	Multiplication
%	Remaindering modulus (integer result truncated toward zero)
/	Division
+	Addition
-	Subtraction
v	Square root

2.1.2 Relational operators

dc allows the following relational operators (also referred to as testing commands):

```
<x   >x   =x   !<x   !>x   !=x
```

These cause the top two elements of the stack to be popped and compared. Register x is executed if the top two elements of the stack satisfy the stated relation. The exclamation point indicates negation.

2.2 dc command set

The following sections describe the dc commands in detail, categorized by subject. At the end of the categorized sections is a quick-reference list of all dc commands, with brief descriptions of each.

2.2.1 Input/output format and base

The input and output bases affect only the interpretation of numbers on input and output. They have no effect on internal arithmetic computations.

Large numbers are generated with 70 characters per line; a backslash (\) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 are used for decimal-octal or decimal-hexadecimal conversions.

2.2.2 Input conversion and base

Numbers are converted to their internal representation as they are read in to `dc`.

`_` Negative numbers are indicated by preceding the number with an underscore (_).

`i` 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 default for input base (`ibase`) is 10 (decimal) but may, for example, be changed to 8 or 16 for octal- or hexadecimal-to-decimal conversions.

`I` The `I` command pushes the value of the input base on the stack.

No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16. The hexadecimal digits A through F correspond to the numbers 10 through 15, regardless of input base.

2.2.3 Output commands

`p` The `p` command causes the top of the stack to be printed. It does not remove the top of the stack.

`f` The `f` command prints the contents of all of the stack registers.

`o` The `o` command is used to change the output base (`obase`). This command uses the top of the stack truncated to an integer as the base for all further output. The default output base is 10 (decimal).

`O` The `O` command pushes the value of the output base on the stack.

2.2.4 Scale

`dc` can accommodate scales up to 99 decimal places. The default scale is 0.

`k` The `k` command sets the scale to the number on the top of the stack, truncated to an integer.

`K` The `K` command can 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 rules governing how the scale of a result is resolved for the different operations are as follows:

Operator	Scale
^	The scale of the result is the sum of the scales of the two operands. If this exceeds the value of `scale` it is truncated to that value.
*	The scale of the result is the sum of the scales of the two operands. If this exceeds the value of `scale` it is truncated to that value.
%	The scale of the remainder is the maximum of the dividend scale and quotient scale, plus the divisor scale.
/	The scale of the result is the value of `scale`. You must specify a `scale` value for any scale to occur.
+	The scale of the result is the larger scale of the two operands.
–	The scale of the result is the larger scale of the two operands.
v	The scale of the result is given the scale of the operand or the value of `scale`, whichever is larger.

2.2.5 Stack commands

`c` The `c` command clears the stack.

`d` The `d` command pushes a duplicate of the top number onto the stack.

`z` The `z` command pushes the stack size onto the stack.

`X` The `X` command replaces the number on the top of the stack with its scale factor.

`Z` The `Z` command replaces the top of the stack with its length.

2.2.6 Subroutine definitions and calls

`[]` Enclosing a string in brackets pushes the ASCII string onto the stack.

`q` The `q` command quits or (when executing a string) pops the recursion level by two.

2.2.7 Internal registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands `s` and `l`:

`s`*x* The `s`*x* command pops the top of the stack and stores the result in register *x*. The *x* can be any character; even a blank or newline is considered a valid register name.

`l`*x* The `l`*x* command puts the contents of register *x* on the top of the stack. The *x* can be any character; even a blank or newline is considered a valid register name.

Note: The `l` command has no effect on the contents of register *x*. The `s` command, however, is destructive.

2.2.8 Pushdown registers and arrays

Note: The following commands are intended for use by a compiler, rather than for direct use by programmers.

`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* `S`*x* pushes the top value of the main stack onto the stack for the register *x*.

`L`*x* `L`*x* pops the stack for register *x* and puts the result on the main stack.

`s` and `l`
The `s` and `l` commands also work on registers, but not as pushdown stacks. The `l` command does not affect the top of the register stack, but `s` destroys what was there before.

The commands that work on arrays are `:` and `;`.

`:`*x* The `:`*x* command 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.

`;x` The `;x` command 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.

2.2.9 Miscellaneous commands

`!` The `!` command interprets the rest of the line as an A/UX system command and passes it to the operating system to execute.

`Q` The `Q` command uses the top of the stack as the number of levels of recursion to skip.

2.3 dc command quick reference

The following is a quick-reference list of `dc` command characters and their functions:

`[...]` Puts the bracketed character string on top of the stack.

`!` Interprets the rest of the line as an A/UX system command. Control returns to `dc` when the command terminates.

`?` Takes a line of input from the input source (usually the console) and executes it.

`c` Pops all values on the stack; the stack becomes empty.

`d` Duplicates the top value on the stack.

`f` Prints all values on the stack and in registers.

`i` and `I`
Pops the top value on the stack and uses it as the number radix for further input. The command `I` pushes the value of the input base on the stack.

`k` and `K`
Pops the top of the stack and uses that value as a scale factor that determines the maximum number of decimal places which are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. The `K` command can be used to push the value of `scale` on the stack.

`l`*x* and `L`*x*
The `l` command puts the contents of register *x* on top of the

stack. The initial value of a new register is treated as a zero by the command `l`, but treated as an error by the command `L`. The `L`*x* command pops the stack for register *x* and puts the result on the main stack.

`o` and `O`
: The top value on the stack is popped and used as the number radix for further output. The command `O` pushes the value of the output base on the stack.

`p`
: The top value on the stack is printed. The top value remains unchanged.

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

`s`*x* and `S`*x*
: The top of the main stack is popped and stored in a register named *x* (where *x* may be any character). The value of register *x* is pushed onto the stack. Register *x* is not altered. `S`*x* pushes the top value of the main stack onto the stack for the register *x*.

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

`x` and `X`
: The `x` command assumes the top of the stack is a string of `dc` commands, removes it from the stack, and executes it. The `X` command replaces the number on the top of the stack with its scale factor.

`z` and `Z`
: The value of the stack level is pushed onto the stack. The command `Z` replaces the top of the stack with its length.

3. Programming dc

By combining a few of the available constructs, such as the load, store, execute, and print commands (`l`, `s`, `x`, `p`), the `[ ]` construct to store strings, and the testing commands (relational operators), it is possible to program `dc`. For example, the following expressions instruct `dc` to print the numbers 0 through 9:

```
[ li p 1+ si li 10 >a ]sa
0 si
la x
```

Consider the first expression in this example:

```
[ li p 1+ si li 10 >a ]sa
```

This first instruction makes use of the `[ ]` construct for storing strings. The entire expression is stored as a character string on top of the stack. Reading from left to right, this character array holds the following commands:

- Load the contents of register `i` on top of the stack, and print it.

  Note: Using the print command does not remove the top of the stack.

- Add (+) 1 to the value found on top of the stack, and place the result on top of the stack.
- Store the value currently found on top of the stack in register `i`.
- Load the contents of register `i` on top of the stack, then load the number 10 onto the stack. Use the testing operator > on these top two stack elements to see if 10 is greater than the number that was loaded from register `i`. If 10 is greater, then execute register `a`. This is the ''control element'' in this example, because it will stop the processing of the expressions as soon as the value in register `i` is equal to 10.
- Store the character array in register `a`.

The second and third lines of the example contain the expressions

```
0 si
la x
```

- The `0 si` instruction clears register `i` by storing 0 in that register, thereby clobbering any previous value it may have had.

- The `la` and `x` instructions load the contents of register `a` on top of the stack and execute it.

Note: The size of numbers in `dc` is limited only by the size of available memory.

Chapter 23
m4 Reference

Contents

Tables

Chapter 23

m4 Reference

1. m4: a macro processor

The m4 macro processor is a general-purpose macro-processing utility. It can also be considered to be an interpreter for the m4 language. The #define statement in the C language is an example of the basic facility provided by any macro processor: the replacement of some text by some (other) text. For several reasons, m4 is a more powerful macro processor than the standard C preprocessor, cpp.

The basic operation of m4 is to read every alphanumeric token (string of letters and digits) in the input and to determine if the token is the name of a macro. The name of a macro is replaced by its defining text and the resulting string is pushed back onto the input to be rescanned.

Besides the straightforward replacement of one string of text by another, the m4 macro processor also provides the following features:

- Arguments to macros
- Arithmetic capabilities
- File manipulation
- Conditional macro expansion
- String and substring functions
- Recursive definitions

When a macro is called with arguments, the arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

The m4 macro processor accepts user-defined macros as well as its ''built-in'' macros. Both types of macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

2. Invoking `m4`

To run `m4`, give the command

```
   m4 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. The output may be redirected for subsequent processing, as follows:

```
   m4 files > outputfile
```

3. Defining macros

3.1 `define`

The primary built-in function of `m4` is `define`. This function is used to define new macros. The general form is

```
   define(name, replacement)
```

All subsequent occurrences of *name* are replaced by *replacement*. The *name* must be alphanumeric and must begin with a letter (the underscore (_) counts as a letter). The *replacement* is any text that contains balanced parentheses. An escaped RETURN or an embedded newline character allows a multi-line *replacement* to be specified.

The following is a typical example of the use of `define`, in which `N` is defined to be the string `100` and is then used in a later `if` statement:

```
   define(N, 100)
   if (i > N) echo "number too large"
```

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 this character, the macro call is assumed to have no arguments.

Macro calls have the following general form:

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

A macro name is recognized as such only if it appears surrounded by nonalphanumerics. In the following example, the variable NNN is absolutely unrelated to the defined macro N, even though the variable contains a lot of N's:

```
define(N, 100)
if (NNN > 100) echo "number too large"
```

Macros may be defined in terms of other macros. For example, the following defines both M and N to be 100. If N is redefined and subsequently changes, M retains the value of 100, not N.

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

The m4 macro processor expands macro names into their defining text as soon as possible. The string N is immediately replaced by 100. The string M is then defined to be 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).

3.2 Quoting

The more general solution to the problem of making sure the correct strings get substituted is to delay the expansion of the arguments of define by quoting them. The quoting characters initially recognized by m4 are the left and right single quotes, ` and '. 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 result of using quotes is to define M as the string N, not as 100.

The general rule is that m4 always strips off one level of single quotes whenever it evaluates something. *This is true even outside macros.*

If the word define itself is to appear in the output, the word must be quoted in the input as follows:

```
`define' = 1;
```

Another example of using quotes is to redefine a macro. 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, for instance, 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, however, because only names that begin with an alphanumeric character can be defined.

3.3 changequote

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
```

3.4 undefine

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')
```

Once removed, the definition cannot be reused.

3.5 ifdef

The built-in ifdef provides a way to determine if a macro is currently defined.

Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef('pdp11', 'define(wordsize,16)')
ifdef('u3b', 'define(wordsize,32)')
```

Remember to use the quotes.

The ifdef macro actually permits three arguments. If the first argument is defined, the value of ifdef is the second argument. If the first argument is not defined, the value of ifdef is the third argument. If there is no third argument, the value of ifdef is null.

If the name is undefined, the value of ifdef is then the third argument, as in

```
ifdef('unix', on UNIX, not on UNIX)
```

3.6 Arguments

User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its define), any occurrence of $*n*$ is replaced by the *n*th argument when the macro is actually used. Thus, the following macro, bump, generates code to increment its argument by 1:

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

The statement

```
bump(x)
```

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`) (see "Built-In Macro Summary" under `shift` for more information). 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 that 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, because no corresponding arguments were provided. Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus,

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

defines `a` to be `b   c`.

Arguments are separated by commas; however, when commas occur within parentheses, the argument is neither terminated nor separated. For example,

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

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

There are three other constructions that are useful in macro definitions:

```
$#
$*
$@
```

During macro replacement, the construction `$#` is replaced by the number of arguments. The `$*` construction is replaced by a list of the arguments separated by commas. The construction `$@` is like `$*` except that each argument is quoted (using the current quotes). See the section ''Recursive Definitions'' for examples of the first two constructions.

3.7 ifelse

Arbitrary conditional testing is performed via the built-in macro `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 to compare two strings and return `yes` or `no` if they are the same or different, as follows:

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

Note the quotes, which prevent evaluation of `ifelse` occurring too early. If the fourth argument is missing, it is treated as empty. Thus,

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

is *c* if *a* matches *b*, and null otherwise.

`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* is the same as 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 and the specified strings don't match, the result is null.

4. Arithmetic built-ins

The `m4` program provides three built-in functions for doing arithmetic on integers (only):

```
incr
decr
eval
```

The simplest are `incr`, which increments its numeric argument by 1, and `decr`, which 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 function called eval, which is capable of arbitrary arithmetic on integers. The operators in decreasing order of precedence are as follows:

Table 23-1. Arithmetic operators

Symbol	Meaning
+ -	Unary plus and minus
** ^	Exponentiation
* / %	Multiplication and division
+ -	Binary plus and minus
== != < <= > >=	Relational operators
!	Logical negation (NOT)
& &&	Logical multiplication (AND)
| ||	Logical addition (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 A/UX 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)')
```

First N is defined as 3; then M is defined as 0, since 2 is not equal to N+1. If M were defined as

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

then its defined value would be 1, because the result of the comparison would be true.

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.

5. I/O manipulation

5.1 `include` and `sinclude`

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, and so on.

A fatal error occurs if the file named by *filename* cannot be accessed. To get some control over this situation, you can use the alternate form, `sinclude`, or quote the filename. The built-in `sinclude` (silent include) says nothing and continues if the file named cannot be accessed.

5.2 `divert`, `undivert`, and `divnum`

The output of `m4` can be diverted to temporary files during processing, and the collected material can be generated upon command. The `m4` program 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 produced all at once at the end of processing with the diversions produced in ascending 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 following code, for example, throws away excess newlines.

```
divert(-1)
define(N, 100)
define(M, 200)
define(L, 300)
divert
```

Note: The newline character at the end of each `define` is passed to the output, as described in the following section.

The built-in macro `undivert`, with no arguments, brings back all diversions in numerical order. With arguments, `undivert` brings back the selected diversions in the order specified by the argument. `undivert` discards the diverted text. You can also discard text by using a diversion number which is not between 0 and 9, inclusive.

The value of `undivert` is *not* the diverted text but rather the number of the diversion to bring back into the text. Furthermore, the diverted material is not rescanned for macros.

As an example of the interaction between `divert`, `undivert`, and current diversion, consider the following code:

```
this is current diversion
divert(1)
this is diversion 1
divert(2)
this is diversion 2
divert(3)
this is diversion 3
divert
this is current diversion again
undivert
once again, current diversion
```

In the above trivial code there are three diversions between the two lines of current diversion code. The use of `divert` at the end of diversion 3 is needed to inform `m4` that what follows is not part of diversion 3. `undivert` with no arguments will insert at the current position all previous diversions, with no rescanning of any macros there may be there. The output of the above code is

```
this is current diversion

this is current diversion again

this is diversion 1

this is diversion 2

this is diversion 3

once again, current diversion
```

Note that the diverted text is not brought back again at the end of the output by the normal process; the diverted text has been discarded by the use of `undivert`. Another example can make this clearer:

```
this is main diversion
divert(1)
this is diversion 1
divert(2)
this is diversion 2
divert(3)
this is diversion 3
divert
this is main diversion again
undivert(3)
once again, main diversion
undivert(2)
```

The ouput for the above is

```
this is main diversion

this is main diversion again

this is diversion 3

once again, main diversion

this is diversion 2


this is diversion 1
```

As you can see, only diversion 1 is brought back by the normal process, because only diversion 1 has not been undiverted and therefore discarded. Note also that you can change the order of appearance of the diverted versions.

The built-in macro `divnum` returns the number of the currently active diversion. The current output stream is 0 during normal processing.

5.3 `dnl`

There is a built-in macro called `dnl` that deletes all characters that follow it, up to and including the next newline. 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 newline at the end of each line that is not part of the definition. The newline is copied into the output so that each `define` statement is followed by a blank line. If the built-in macro `dnl` is added to each of these lines, the newlines will disappear.

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

6. String manipulation

6.1 `len`

The built-in macro `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 parentheses and comma are counted along with `a` and `b`).

6.2 `substr`

The built-in macro `substr` can be used to produce substrings of strings. The input

```
substr(s, i, n)
```

returns the substring of *s* that starts at the *i*th position (origin 0) and is *n* characters long. If *n* is omitted, the rest of the string is returned. For example,

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

returns the following string:

```
ow is the time.
```

If *i* or *n* is out of range, various actions occur.

6.3 `index` and `translit`

The built-in macro `index` returns the index (position) in one string where the first character of another given string occurs, or –1 if it does not occur. If is written as

```
index(s1, s2)
```

where *s1* is the string to be searched and *s2* is the string to be searched for. As with `substr`, the origin for strings is 0.

The built-in macro `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

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

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So,

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

would delete vowels from *s*.

7. Printing

7.1 `errprint`

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

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

7.2 `dumpdef`

The built-in macro `dumpdef` is a debugging aid that dumps the current names and definitions of items named as arguments. If no arguments are given, then all current names and definitions are printed. Remember to quote the names.

8. Executing system commands

8.1 `syscmd` and `maketemp`

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

```
syscmd(date)
```

on the A/UX system runs the `date` command. Normally, `syscmd` would be used to create a file for a subsequent `include`.

To facilitate making unique filenames, the built-in macro `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.

9. Interactive use of `m4`

The input to `m4` may come from a file, the standard input, or both. Thus, it is possible to use `m4` interactively, by telling it to take its input from the standard input. There are several ways to do this. The simplest is to invoke `m4` as follows:

```
m4
```

At this point, `m4` will read from the standard input.

If you have an existing set of `m4` commands stored in a file, you may instruct `m4` to process those commands first by invoking it as

m4 *file* –

The minus sign is required here to instruct `m4` to read *file* and then the standard input. Alternatively, if you invoke `m4` using just the `m4` command with no arguments, you can tell `m4` to fetch the set of commands from *file* by typing the following line:

include(*file*)

The effect is the same in both cases.

10. Recursive definitions

Since `m4` rescans any text that arises from the replacement of a macro by its defining text, it is possible to construct recursive macro definitions. That is, it is perfectly legal to define a macro in terms of itself. As with any well-constructed recursive definition, however, you must take care that the definition has a well-defined stopping point. Generally, this is easy to do with the `ifelse` command.

For instance, suppose that you need a macro that returns its last argument and discards the rest. You might write the following definition:

```
define(last,
`ifelse($#,1,$1,`last(shift($*))')')
```

When there are multiple arguments, `last` drops the first argument and then calls itself to look for the last argument in the remaining argument list. This definition is well behaved, because when there is only one argument, it alone is returned.

A more interesting example is the following definition of the factorial function:

```
define(fact,
`ifelse($1,1,1,`eval($1*fact(decr($1)))')')
```

If you give m4 the input

```
The factorial of 1 is fact(1).
The factorial of 2 is fact(2).
The factorial of 3 is fact(3).
The factorial of 4 is fact(4).
The factorial of 5 is fact(5).
The factorial of 6 is fact(6).
The factorial of 7 is fact(7).
The factorial of 8 is fact(8).
```

you get the following output:

```
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
```

Finally, you may want to define a recursive macro with two arguments. The standard power function will serve nicely:

```
define(pow,
`ifelse($2,1,$1,`eval($1*pow($1,decr($2)))')')
```

If you then give m4 the input

```
3 to power 1 is pow(3,1).
3 to power 2 is pow(3,2).
3 to power 3 is pow(3,3).
3 to power 4 is pow(3,4).
3 to power 5 is pow(3,5).
3 to power 6 is pow(3,6).
3 to power 7 is pow(3,7).
3 to power 8 is pow(3,8).
```

you get

```
3 to power 1 is 3.
3 to power 2 is 9.
3 to power 3 is 27.
3 to power 4 is 81.
3 to power 5 is 243.
3 to power 6 is 729.
3 to power 7 is 2187.
3 to power 8 is 6561.
```

11. Built-in macro summary

The following are m4 built-in macros:

`changecom`	Changes left and right comment markers from the default # and newline. With no arguments, the comment mechanism is disabled. Comment markers may be up to five characters long.
`changequote`	Changes quoting symbols to the first and second arguments. The symbols may be up to five characters long. With no arguments, this macro restores the original quote characters.
`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 (named 0 through 9).

`divnum`	Returns the number of the currently active diversion.
`dnl`	Reads and discards characters up to and including the next newline.
`dumpdef`	Dumps the current names and definitions of items named as arguments. With no arguments, definitions of all current macros are dumped.
`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 cannot 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.
`len`	Returns the number of characters that make up its argument.
`m4exit`	Causes immediate exit from `m4`.
`m4wrap`	Pushes the exit code back at final EOF.
`maketemp`	Facilitates making unique filenames.
`popdef`	Removes current definition of its argument(s), exposing any previous definitions.
`pushdef`	Defines new macros but saves any previous definition.
`shift`	Returns all arguments 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 A/UX system command given in the first argument.
`sysval`	Gives exit value of most recent system command.
`traceoff`	Turns the macro trace off.
`traceon`	Turns the macro trace on.
`translit`	Performs character transliteration.
`undefine`	Removes user-defined or built-in macro definitions.
`undivert`	Discards the diverted text.
`unix`	Null; indicates that the underlying system is derived from the UNIX operating system.

Chapter 24
curses Reference

Contents

Figures

Chapter 24

curses Reference

1. curses: terminal-independent screen I/O

The curses package is designed to provide a terminal-independent method of providing screen-oriented input and output. It includes facilities for taking input from the terminal, sending output to a terminal, creating and manipulating windows on the screen, and performing screen updates in an optimal fashion. A program using the curses routines and functions generally needs to know nothing about the capabilities of any particular terminal; these characteristics are determined at execution time and guide the program in taking input and producing output. Thus, programs using this package can interact with a large variety of terminals and terminal types.

This chapter is an introduction to the curses and terminfo packages for writing screen-oriented programs. This chapter documents each curses function and discusses several sample programs. The sample programs are at the end of this chapter.

We are also providing termcap for backward compatibility, but new programs should use terminfo.

2. Overview of curses usage

For curses to be able to produce the proper output, it has to know what kind of terminal you have. curses uses the standard A/UX system convention for this; the name of the terminal is stored in the environment variable TERM.

A program using curses always starts by calling initscr (see Figure 24-1). Other modes can then be set as needed by the program. Possible modes include cbreak and idlok(stdscr, TRUE). These modes will be explained later.

A curses program follows the framework shown in Figure 24-1.

Figure 24-1. Framework of a **curses** program

```
#include <curses.h>
main()
{
...
initscr();        /* Initialization */

cbreak();         /* Various optional mode settings */
nonl();
noecho();
...
while (!done) { /* Main body of program */
   ...
    /* Sample calls to draw on screen */
    move(row, col);
    addch(ch);
    printw("Formatted print with value %d\n", value);
   ...
}

endwin();          /* Clean up */
exit(0);

}
```

2.1 Output

During the execution of the program, output to the screen is done with routines such as

   `addch(`*ch*`)`

and

   `printw(`*fmt*`,` *args*`)`

which behave just like `putchar` and `printf` except that they go through `curses`. The cursor can be moved with the call

   `move(`*row*`,` *col*`)`

These routines generate output only to a data structure called a ''window,'' not to the actual screen. A window is a representation of a CRT screen, containing such things as an array of characters to be displayed on the screen, a cursor, a current set of video attributes, and various modes and options. Unless you use more than one of them, you don't need to worry about windows except to realize that a window is buffering your requests for output to the screen. For further information about windows, see the section "Multiple Windows" in this chapter.

To send all accumulated output, you must call

```
refresh()
```

Finally, before the program exits, it should call

```
endwin()
```

which restores all terminal settings and positions the cursor at the bottom of the screen.

See the sample program `scatter` at the end of this chapter. This program reads a file and displays it in a random order on the screen. Some programs assume all screens are 24 lines by 80 columns. It is important to understand that many are not. The variables

```
LINES
```

and

```
COLS
```

are defined by `initscr` with the current screen size. Programs should use them instead of assuming a 24 by 80 screen.

No output to the terminal actually happens until `refresh` is called. Instead, routines such as `move` and `addch` draw on a window data structure called `stdscr` (standard screen). `curses` always keeps track of what is on the physical screen, as well as what is in `stdscr`.

When `refresh` is called, `curses` compares the two screen images and sends a stream of characters to the terminal that will turn the current screen into what is desired. `curses` considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is desired. It usually produces as few characters as is possible. This function is called "cursor optimization" and is the source of the name of the `curses` package.

Note: Because of the hardware scrolling of terminals, writing to the lower right character position is impossible.

2.2 Input

`curses` functions are also provided for input from the keyboard. The primary function is

```
getch()
```

which is like `getchar` except that it goes through `curses`. This function waits for the user to type a character on the keyboard and then returns that character. Its use is recommended for programs using the options

```
cbreak()
```

or

```
noecho()
```

because several terminal- or system-dependent options become available that are not possible with `getchar`.

Options that you can use with `getch` include

```
keypad
```

which allows extra keys such as arrow keys, function keys, and other special keys that transmit escape sequences to be treated as just any other key. (The values returned for these keys are listed below; these values are over octal 400, so they should be stored in a variable larger than a `char`.)

The

```
nodelay
```

option causes the value –1 to be returned if there is no input waiting. Normally, `getch` waits until a character is typed.

Finally, the routine

`getstr(`*str*`)`

can be called, allowing input of an entire line, up to a newline. This routine handles echoing and the erase and kill characters of the user.

Examples of the use of these options are in later sample programs.

The following function keys might be returned by getch, if keypad has been enabled. Note that not all of these may be supported by a particular terminal/keyboard, because the key doesn't exist or the terminal is not transmitting a unique code when the key is pressed.

Name	Value	Key name
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	
KEY_HOME	0406	Home key (upward + left arrow)
KEY_BACKSPACE	0407	Backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 keys is reserved (only KF0 through KF10 are currently supported).
KEY_F(n)	(KEY_F0+(n))	Formula for fn
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert character or enter insert mode
KEY_EIC	0514	Exit insert character mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll one line forward
KEY_SR	0521	Scroll one line backward (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	Soft (partial) reset (unreliable)

Name	Value	Key name
KEY_RESET	0531	Reset or hard reset (unreliable)
KEY_PRINT	0532	Print or copy
KEY_LL	0533	Home down or bottom (lower left)

The following keys are not currently supported on the Macintosh™ II: KEY_BREAK, KEY_ENTER, KEY_SRESET, KEY_RESET, and KEY_PRINT.

See the sample program show at the end of this chapter for an example of the use of getch. The show program pages through a file, showing one full screen each time the user presses the space bar. By creating an input file for show made up of 24-line pages, each segment varying slightly from the previous page, nearly any exercise for curses can be created. Such input files are called "show scripts."

In the sample show program,

cbreak
: is called so that you can press the space bar without having to press RETURN.

noecho
: is called to prevent the space from echoing in the middle of a refresh, messing up the screen.

nonl
: is called to enable more screen optimization.

idlok
: is called to allow insert and delete line, because many show scripts are constructed to duplicate bugs caused by that feature.

clrtoeol
: clears from the cursor to the end of the line.

clrtobot
: clears from the cursor to the end of the screen.

2.3 Highlighting

The function `addch` always draws two things on a window. In addition to the character itself, it draws a set of "attributes" associated with the character. These attributes cover various forms of highlighting of the character. For example, the character can be put in reverse video, bold, or underline. You can think of the attributes as the color of the ink used to draw the character.

A window always has a set of current attributes associated with it. The current attributes are associated with each character as it is written to the window. The current attributes can be changed with a call to

`attrset(`*attrs*`)`

(Think of this as dipping the window's pen in a particular color of ink.) The names of the attributes are

```
A_STANDOUT
A_REVERSE
A_BOLD
A_DIM
A_INVIS
A_UNDERLINE
```

For example, to put the word "boldface" in bold, you might use the following code:

```
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
...
refresh();
```

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, `curses` will attempt to find a substitute attribute. If none is possible, the attribute is ignored.

The `A_STANDOUT` attribute is used to make text attract the attention of the user. The particular hardware attribute used for `A_STANDOUT` varies from terminal to terminal, and is chosen to be the most visually pleasing attribute the terminal has. `A_STANDOUT` is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or inverse video, but instead just need to highlight some text. For such applications, the `A_STANDOUT` attribute is recommended. Two convenient functions,

```
standout()
standend()
```

turn this attribute on and off.

Attributes can be turned on in combination. For example, to turn on blinking bold text, use

```
attrset(A_BLINK|A_BOLD)
```

Individual attributes can be turned on and off with `attron` and `attroff` without affecting other attributes.

For a sample program using attributes, see the `highlight` program at the end of this chapter. The `highlight` program takes a text file as input and allows embedded escape sequences to control attributes. In this sample program,

`\U`	Turns on underlining
`\B`	Turns on bold
`\N`	Restores normal text

Note the initial call to `scrollok`. This allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, `curses` automatically scrolls the terminal up a line and calls `refresh`.

The `highlight` program comes about as close to being a filter as is possible with `curses`. It is not a true filter, because `curses` must "take over" the CRT screen. To determine how to update the screen, it must know what is on the screen at all times. This requires `curses` to clear the screen in the first call to `refresh` and to know the cursor position and screen contents at all times.

2.4 Multiple windows

A window is a data structure representing all or part of the CRT screen. It has room for a two-dimensional array of characters, attributes for each character (a total of 16 bits per character: 7 for text and 9 for attributes), a cursor, a set of current attributes, and a number of flags.

`curses` provides a full screen window, called

```
stdscr
```

and a set of functions that use `stdscr`. Another window is provided called

```
curscr
```

representing the physical screen.

It is important to understand that a window is only a data structure. Use of more than one window does not imply use of more than one terminal, nor does it involve more than one process. A window is merely an object that can be copied to all or part of the terminal screen. The current implementation of `curses` does not allow windows that are bigger than the screen.

You can create additional windows with the function

`newwin(`*lines*`,` *cols*`,` *begin-row*`,` *begin-col*`)`

which returns a pointer to a newly created window. The window will be *lines* by *cols*, and the upper left corner of the window will be at screen position `(`*begin-row*`,` *begin-col*`)`.

All operations that affect `stdscr` have corresponding functions that affect an arbitrary named window. Generally, these functions have names formed by putting a `w` on the front of the `stdscr` function and adding the window name as the first parameter. Thus,

`waddch(`*mywin*`, c)`

would write the character `c` to window *mywin*. The `wrefresh` function is used to flush the contents of a window to the screen.

Windows are useful for maintaining several different screen images, among which you can alternate. Also, you can subdivide the screen into several windows, refreshing each of them as desired. When windows overlap, the contents of the screen will be copied from the more recently refreshed window.

In all cases, the non-`w` version of the function calls the `w` version of the function, using `stdscr` as the additional argument. Thus, a call to

```
addch(c)
```

results in a call to

```
waddch(stdscr, c)
```

The sample program `window` at the end of this chapter shows the use of multiple windows. The main display is kept in `stdscr`. When the user temporarily wants to put something else on the screen, a new window is created covering part of the screen. A call to `wrefresh` on that window causes the window to be written over `stdscr` on the screen. Calling `refresh` on `stdscr` causes the original window to be redrawn on the screen.

In the sample `window` program, note the calls to

```
touchwin
```

before an overlapping window is written out. These are necessary to defeat an optimization in `curses`. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call `touchwin` on the new window to get it completely written out.

For convenience, a set of move functions are also provided for most of the common functions, which result in a call to `move` before the other function. For example,

```
mvaddch(row,col,c)
```

is the same as

```
move(row, col); addch(c)
```

Combinations also exist, for example,

```
mvwaddch(row,col,win,c)
```

2.5 Multiple terminals

`curses` can produce output on more than one terminal at once. This is useful for single-process programs that access a common database, such as multiplayer games. Output to multiple terminals is a difficult business, and `curses` does not solve all the problems for the programmer. The program itself must determine the filename of each terminal line and what kind of terminal is on each of those lines.

The standard method (checking `$TERM` in the environment) does not work, because each process can examine only its own environment. Another problem that must be solved is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. Nonetheless, a program wishing to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security considerations would also make this inappropriate. However, for some applications, such as an interterminal communication program or a program that takes over unused TTY lines, it would be appropriate.)

A typical solution requires that the user logged in on each line run a program that notifies the master program that the user is interested in joining the master program, telling it the notification program's process ID, the name of the TTY line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program, and all programs exit.

`curses` handles multiple terminals by always having a "current terminal." All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary `curses` routines.

References to terminals have type `struct screen *`.

A new terminal is initialized by calling

`newterm(`*type*`,`*fd*`)`

`newterm` returns a screen reference to the terminal being set up; *type* is a character string, naming the kind of terminal being used; and *fd* is a `stdio` file descriptor to be used for input and output to the terminal. (If only output is needed, the file can be open for output only.)

This call replaces the normal call to `initscr`, which calls

```
newterm(getenv("TERM"),stdout)
```

To change the current terminal, call

```
   set_term(sp)
```

where *sp* is the screen reference to be made current. `set_term` returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with `newterm`. Options such as `cbreak` and `noecho` must be set separately for each terminal. The functions `endwin` and `refresh` must be called separately for each terminal. See Figure 24-2 for a typical scenario to send a message to each terminal.

Figure 24-2. Sending a message to several terminals

```
for (i=0; i<nterm; i++) {
     set_term(terms[i]);
     mvaddstr(0, 0, "Important message");
     refresh();
}
```

See the sample program `two` at the end of this chapter for a full illustration. The `two` program pages through a file, showing one page to the first terminal and the next page to the second terminal. It then waits for a space to be typed on either terminal, and shows the next page to the terminal typing the space. Each terminal has to be separately put into `nodelay` mode. It is necessary to busy-wait or call `sleep` (see `sleep`(3C) in *A/UX Programmer's Reference*) between each check for keyboard input, or use the multiplexor `select(2)`. This program sleeps for a second between checks.

The `two` program is just a simple example of two-terminal `curses`. It does not handle notification, as described above; instead it requires the name and type of the second terminal on the command line. As written, the command

```
   sleep 100000
```

must be typed on the second terminal to put it to sleep while the program runs, and the first user must have both read and write permission on the second terminal.

2.6 Low-level `terminfo` usage

Some programs need to use lower-level primitives than those offered by `curses`. For such programs, the `terminfo`-level interface is offered.

The `terminfo`-level interface does not manage your CRT screen, but rather gives you access to strings and capabilities that you can use to manipulate the terminal. `curses` takes care of all the glitches and misfeatures present in physical terminals, but at the `terminfo` level you must deal with them yourself. Whenever possible, the higher-level `curses` routines should be used. This will make your program more portable to other A/UX systems and to a wider class of terminals. Also, it cannot be guaranteed that this part of the interface will not change or will be upwardly compatible with previous releases.

There are two circumstances in which you should use `terminfo`. The first is when you are writing a special-purpose tool that sends a special-purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. The second situation is when you are writing a filter. A typical filter does one transformation on the input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use `terminfo`.

A program written at the `terminfo` level uses the framework shown in Figure 24-3.

Figure 24-3. `terminfo`-level framework

```
#include <curses.h>
#include <term.h>
   ...
   setupterm(0, 1, 0);
   ...
   putp(clear_screen);
   ...
   reset_shell_mode();
exit(0);
```

Initialization is done by calling `setupterm`.

Passing the values 0, 1, 0 invokes reasonable defaults. If `setupterm` cannot figure out what kind of terminal you are on, it prints an error message and exits. Your program should call `reset_shell_mode` before it exits. Global variables with names like `clear_screen` and `cursor_address` are initialized by the call to `setupterm`. They can be produced using `putp` or `tputs` (which allows the programmer more control). These strings should not be directly sent to the terminal using `printf`, because they contain padding information. A program that directly generates strings will fail on terminals that require padding or that use the xon/xoff flow-control protocol.

In the `terminfo` level, the higher-level routines described previously are not available. It is up to the programmer to generate whatever is needed. For a list of capabilities and a description of what they do, see `terminfo`(4).

The `termhl` sample program at the end of this chapter shows a simple use of `terminfo`. It is a version of the `highlight` sample program that uses `terminfo` instead of `curses`. This version can be used as a filter. The strings to enter bold and underline mode, and to turn off all attributes, are used.

This program is more complex than it has to be in order to illustrate some properties of `terminfo`. The routine `vidattr` could have been used instead of directly generating

```
enter_bold_mode
enter_underline_mode
exit_attribute_mode
```

In fact, the program would be more robust if it did so, since there are several ways to change video attribute modes. However, this program was written to illustrate typical use of `terminfo`.

The function

`tputs` *(cap, affcnt, outc)*

applies padding information. Some capabilities contain strings like `$<20>`. This means to pad for 20 milliseconds. `tputs` generates enough pad characters to delay for the appropriate time. The first parameter is the string capability to be generated. The second is the number of lines affected by the capability. Some capabilities may require padding that depends on the number of lines affected. For example, `insert_line` may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention, *affcnt* is 1 if no lines are affected. For safety, the value 1 is used rather than 0 (*affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0). The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always just calls `putchar`. For these programs, the routine `putp(`*cap*`)` is a convenient abbreviation. The `termhl` sample program could be simplified by using `putp`.

Note also in this example the special check for the capability `underline_char`. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. The `termhl` program keeps track of the current mode, and if the current character is supposed to be underlined, will output `underline_char` if necessary.

Low-level details such as this are precisely why the `curses` level is recommended over the `terminfo` level. `curses` takes care of terminals with different methods of underlining and other CRT functions. Programs at the `terminfo` level must handle such details themselves.

2.7 A larger example

For a final example, see the `editor` sample program at the end of this chapter.

The `editor` program illustrates how to use `curses` to write a screen editor patterned after the `vi` editor. This editor keeps the buffer in `stdscr` to keep the program simple; a real screen editor would keep a separate data structure. Many simplifications have been made here. No provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth noting. The routine to write out the file illustrates the use of the `mvinch` function, which returns the character in a window at a given position. The data structure used here does not have a provision for keeping track of the number of characters in a line, or the number of lines in the file, so trailing blanks are eliminated when the file is written out.

The program uses these built-in `curses` functions:

```
insch
delch
insertln
deleteln
```

These functions behave much as the similar functions on intelligent terminals behave, inserting and deleting a character or a line.

The command interpreter accepts not only ASCII characters, but also special keys. (Some editors are "modeless," using nonprinting characters for commands. This is largely a matter of taste; the point being made here is that both arrow keys and ordinary ASCII characters should be handled.)

In the `editor` sample program, note the call to `mvaddstr` in the input routine. `addstr` is roughly like the C `fputs` function, which writes out a string of characters. Like `fputs`, `addstr` does not add a trailing newline. It is the same as a series of calls to `addch` using the characters in the string. `mvaddstr` is the `mv` version of `addstr`, which moves to the given location in the window before writing.

The CONTROL-l command illustrates a feature that most programs using `curses` should add. Often some program beyond the control of `curses` has written something to the screen, or some line noise has messed up the screen beyond what `curses` can keep track of. In this case, the user would type CONTROL-l, causing the screen to be cleared and redrawn. This is done with the call to

```
clearok(curscr)
```

which sets a flag causing the next `refresh` to first clear the screen. Then `refresh` is called to force the redraw.

Note also the call to

```
flash()
```

which flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within earshot of the user. The routine

```
   beep()
```

can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to `beep` will flash the screen.)

Another important point is that the input command is terminated by CONTROL-d, not ESCAPE. It is very tempting to use ESCAPE as a command, because ESCAPE is one of the few special keys that is available on most keyboards. (RETURN and BREAK are among the others.) However, using ESCAPE as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with ESCAPE (''escape sequences'') to control the terminal, and have special keys that send escape sequences to the computer. If the computer recognizes an ESCAPE coming from the terminal, it cannot determine for sure whether the user pressed the ESCAPE key, or whether a special key was pressed. `curses` handles the ambiguity by waiting for up to 1 second. If another character is received during this second, and if that character might be the beginning of a special key, more input is read (waiting for up to 1 second for each character) until either (1) a full special key is read, (2) 1 second passes, or (3) a character is received that could not have been generated by a special key.

While this strategy works most of the time, it is not foolproof. It is possible for the user to press ESCAPE, then to type another key quickly, which causes `curses` to think a special key has been pressed. Also, there is a 1-second pause until the escape can be passed to the user program, resulting in slower response to the ESCAPE key.

Many existing programs use ESCAPE as a fundamental command, so it cannot be changed without infuriating a large class of users. Such programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a timeout solution. The message is clear: When designing your program, avoid the ESCAPE key.

3. List of curses routines

This section describes all the routines available to the programmer in the curses package. The routines are organized by function. For an alphabetical list, see curses(3X).

3.1 Structure

All programs using curses should include the file <curses.h>. This file defines several curses functions as macros, and defines several global variables and the datatype WINDOW. References to windows are always of type WINDOW *.

curses also defines certain windows as constants:

stdscr	The standard screen, used as a default to routines expecting a window
curscr	The current screen, used only for certain low-level operations like clearing and redrawing a garbaged screen

Integer variables are declared, containing the size of the screen.

LINES	Number of lines on the screen
COLS	Number of columns on the screen

Boolean constants are defined as follows with values 1 and 0:

```
#define TRUE(1)
#define FALSE(0)
#define ERR(-1)
#define OK(0)
```

Additional constants are values returned from most curses functions:

ERR	Returned if there was some error, such as moving the cursor outside a window
OK	Returned if the function could be properly completed

The include file

```
<curses.h>
```

automatically includes <stdio.h> and an appropriate TTY driver interface file, currently either <sgtty.h> or <termio.h>.

Note: The driver interface `<sgtty.h>` is a TTY driver interface used in other versions of the UNIX system.

Including `<stdio.h>` again is harmless but wasteful; including `<sgtty.h>` again usually results in a fatal error.

A program using `curses` should include the loader option

```
-lcurses
```

in the makefile. This is true both for the `terminfo` level and the `curses` level.

The compilation flag

```
-DMINICURSES
```

can be included if you restrict your program to a small subset of `curses` concerned primarily with screen output and optimization. The routines possible with `mini-curses` are listed in the section ''Mini-`curses`'' below.

3.2 Initialization

The following functions are called when initializing a program.

`initscr()`
: The first function called should always be `initscr`. This determines the terminal type and initializes `curses` data structures. `initscr` also arranges that the first call to `refresh` clears the screen.

`endwin()`
: A program should always call `endwin` before exiting. This function restores TTY modes, moves the cursor to the lower-left corner of the screen, resets the terminal into the proper nonvisual mode, and tears down all appropriate data structures.

`newterm(`*type*`,`*fd*`)`
: A program that generates output to more than one terminal should use `newterm` instead of `initscr`. `newterm` should be called once for each terminal. It returns a variable of type `SCREEN *` which should be saved as a reference to that terminal. The arguments are the type of the terminal (a string)

and a `stdio` file descriptor `(FILE *)` for output to the terminal. The file descriptor should be open for both reading and writing if input from the terminal is desired. The program should also call `endwin` for each terminal being used (see `set_term` below). If an error occurs, the value `NULL` is returned.

`set_term(`*new*`)`

This function is used to switch to a different terminal. The screen reference *new* becomes the new current terminal. The previous terminal is returned by the function. All other calls affect only the current terminal.

`longname()`

This function returns a pointer to a static area containing a verbose description of the current terminal. It is defined only after a call to `initscr`, `newterm`, or `setupterm`.

3.3 Option setting

The functions described here set options within `curses`. In each case, *win* is the window affected, and *bf* is a Boolean flag with value `TRUE` or `FALSE` (indicating whether to enable or disable the option). All options are initially `FALSE`. It is not necessary to turn these options off before calling `endwin`.

`clearok(`*win*`, `*bf*`)`

If set, the next call to `wrefresh` with this window clears the screen and redraws the entire screen. If *win* is `curscr`, the next call to `wrefresh` with any window causes the screen to be cleared. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

`idlok(`*win*`, `*bf*`)`

If this feature is enabled, `curses` considers using the hardware insert/delete line feature of terminals so equipped. If disabled, `curses` never uses this feature. The insert/delete character feature is always considered. Enable this option only if your application needs insert/delete line, for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it is not really needed. If insert/delete line cannot be used, `curses` redraws the changed portions of all lines that do not match the desired line.

`keypad`(*win*, *bf*)

This option enables the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and `getch` will return a single value representing the function key. If `keypad` is disabled, `curses` does not treat function keys specially. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option turns on the terminal keypad.

`leaveok`(*win*, *bf*)

Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, because it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

`meta`(*win*, *bf*)

If enabled, characters returned by `getch` are transmitted with all 8 bits, instead of stripping the highest bit. The value `OK` is returned if the request succeeded; the value `ERR` is returned if the terminal or system is not capable of 8-bit input.

`meta` mode is useful for extending the nontext command set in applications where the terminal has a meta shift key. `curses` takes whatever measures are necessary to arrange for 8-bit input. On other versions of UNIX systems, `raw` mode is used. On A/UX systems, the character size is set to 8, parity checking disabled, and stripping of the 8th bit turned off.

Note that 8-bit input is a fragile mode. Many programs and networks pass only 7 bits. If any link in the chain from the terminal to the application program strips the 8th bit, 8-bit input is impossible.

`nodelay`(*win*, *bf*)

This option causes `getch` to be a nonblocking call. If no input is ready, `getch` returns `-1`. If disabled, `getch` hangs until a key is pressed.

`intrflush`(*win*, *bf*)
: If this option is enabled when an interrupt key is pressed on the keyboard (interrupt, quit, suspend), all output in the TTY driver queue is flushed, giving the effect of faster response to the interrupt but causing `curses` to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default is for the option to be enabled. This option depends on support in the underlying teletype driver.

`typeahead`(*fd*)
: Sets the file descriptor for typeahead check. *fd* should be an integer returned from `open` or `fileno`. Setting `typeahead` to -1 disables typeahead check. By default, file descriptor 0 (`stdin`) is used. `typeahead` is checked independently for each screen, and for multiple interactive terminals it should probably be set to the appropriate input for each screen. A call to `typeahead` always affects only the current screen.

`scrollok`(*win*, *bf*)
: This option controls what happens when the cursor of a window is moved off the edge of the window, either from a newline on the bottom line or because the last character of the last line was typed. If disabled, the cursor is left on the bottom line. If enabled, `wrefresh` is called on the window, and then the physical terminal and window are scrolled up one line. Note that to get the physical scrolling effect on the terminal, it is also necessary to call `idlok`.

`setscrreg`(*t*, *b*)
`wsetscrreg`(*win*, *t*, *bf*)
: These two functions allow the user to set a software scrolling region in a window *win* or `stdscr`. *t* and *b* are the line numbers of the top and bottom margins of the scrolling region. (Line 0 is the top line of the window.) If this option and `scrollok` are enabled, an attempt to move off the bottom margin line causes all lines in the scrolling region to scroll up one line. Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the VT100. Only the text of the window is scrolled. If `idlok` is enabled and the terminal

has either a scrolling region or insert/delete line capability, they will probably be used by the output routines.

3.4 Terminal mode setting

The functions described here are used to set modes in the TTY driver. The initial mode usually depends on the setting when the program is called; the initial modes documented here represent the normal situation.

`cbreak()`
`nocbreak()`

These two functions put the terminal into and out of `CBREAK` mode. In this mode, characters typed by the user are immediately available to the program. When out of this mode, the teletype driver buffers characters typed until newline is typed. Interrupt and flow-control characters are unaffected by this mode. Initially the terminal is not in `CBREAK` mode. Most interactive programs using `curses` will set this mode.

`echo()`
`noecho()`

These functions control whether characters typed by the user are echoed as typed. Initially, characters typed are echoed by the teletype driver. Authors of many interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing.

`nl()`
`nonl()`

These functions control whether newline is translated into carriage return and line feed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations, `curses` is able to make better use of the line feed capability, resulting in faster cursor motion.

`raw()`
`noraw()`

The terminal is placed into or out of `raw` mode. `raw` mode is similar to `cbreak` mode in that characters typed are immediately passed through to the user program. The differences are that in `raw` mode, the interrupt, quit, and suspend characters are passed through uninterpreted instead of generating a signal. `raw` mode also causes 8-bit input and output. The behavior of the BREAK key may be different on different systems.

`resetty()`
`savetty()`
These functions save and restore the state of the TTY modes. `savetty` saves the current state in a buffer; `resetty` restores the state to what it was at the last call to `savetty`.

3.5 Window manipulation

`newwin(`*num-lines*`,` *num-cols*`,` *beg-row*`,` *beg-col*`)`
Creates a new window with the given number of lines and columns. The upper-left corner of the window is at line *beg-row* column *beg-col*. If either *num-lines* or *num-cols* is 0, they are defaulted to `LINES`-*beg-row* and `COLS`-*beg-col*. A new full-screen window is created by calling `newwin(0,0,0,0)`.

`newpad(`*num-lines*`,` *num-cols*`)`
Creates a new `pad` data structure. A pad is like a window, except that it is not restricted by the screen size and is not associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (for example, from scrolling or echoing of input) do not occur. It is not legal to call `refresh` with a pad as an argument; the routines `prefresh` or `pnoutrefresh` should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

`subwin(`*orig*`,` *num-lines*`,` *num-cols*`,` *begy*`,` *begx*`)`
Creates a new window with the given number of lines and columns. The window is at position (*begy*, *begx*) on the screen. (It is relative to the screen, not *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window affect both windows. When using this function, often it will be necessary to call `touchwin` before calling `wrefresh`.

`delwin(`*win*`)`

Deletes the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

`mvwin(`*win*`,`*br*`,`*bc*`)`

Moves the window so that the upper-left corner is at position (*br*, *bc*). If the move would cause the window to be off the screen, it is an error and the window is not moved.

`touchwin(`*win*`)`

Throws away all optimization information about which parts of the window have been touched, by pretending the entire window has been drawn on. This is sometimes necessary when using overlapping windows, because a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change.

`overlay(`*win1*`,`*win2*`)`
`overwrite(`*win1*`,`*win2*`)`

These functions overlay *win1* on top of *win2*; that is, all text in *win1* is copied into *win2*. The difference is that `overlay` is nondestructive (blanks are not copied) and `overwrite` is destructive.

3.6 Causing output to the terminal

`refresh()`
`wrefresh(`*win*`)`

These functions must be called to get any output on the terminal, as other routines merely manipulate data structures. `wrefresh` copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. `refresh` is the same, using `stdscr` as a default screen. Unless `leaveok` has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

`doupdate()`
`wnoutrefresh(`*win*`)`

These two functions allow multiple updates with more efficiency than `wrefresh`. To use them, it is important to understand how `curses` works. In addition to all the window structures, `curses` keeps two data structures representing the terminal screen: a ''physical'' screen, describing what is actually on the screen, and a ''virtual'' screen, describing what the programmer *wants* to have on the screen. `wrefresh` works by first copying the named window to the virtual screen `(wnoutrefresh)`, and then calling the routine to update the screen `(doupdate)`. If the programmer wishes to produce several windows at once, a series of calls to `wrefresh` will result in alternating calls to `wnoutrefresh` and `doupdate`, causing several bursts of output to the screen. By calling `wnoutrefresh` for each window, it is then possible to call `doupdate` once, resulting in only one burst of output, with probably fewer total characters transmitted.

`prefresh` *(pad, pminrow, pmincol*
sminrow, smincol
smaxrow, smaxcol)
`pnoutrefresh` *(pad, pminrow, pmincol*
sminrow, smincol
smaxrow, smaxcol)

These routines are analogous to `wrefresh` and `wnoutrefresh` except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper-left corner, in the pad, of the rectangle to be displayed. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower-right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, because the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures.

3.7 Writing on window structures

The routines described here are used to ''draw'' text on windows. In all cases, a missing *win* is taken to be `stdscr`. *y* and *x* are the row and column, respectively. The upper-left corner is always (0,0), not (1,1). The `mv` functions imply a call to `move` before the call to the other function.

3.7.1 Moving the cursor

`move` (*y*, *x*)
`wmove` (*win*, *y*, *x*)

The cursor associated with the window is moved to the given location. This does not move the physical cursor of the terminal until `refresh` is called. The position specified is relative to the upper-left corner of the window. The position specified is relative to the screen, not to the individual window. Thus, if you have a window which is not in the upper-left corner of the screen, moving to the upper-left corner of the window would require the screen coordinates of that corner of the window rather than `(0,0)` to be passed to `move`.

3.7.2 Writing one character

`addch` (*ch*)
`waddch` (*win*, *ch*)
`mvaddch` (*y*, *x*, *ch*)
`mvwaddch` (*win*, *y*, *x*, *ch*)

The character *ch* is put in the window at the current cursor position of the window. If *ch* is a tab, newline, or backspace, the cursor is moved appropriately in the window. If *ch* is a different control character, it is drawn in the ^X notation. The position of the window cursor is advanced. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if `scrollok` is enabled, the scrolling region is scrolled up one line.

The *ch* parameter is actually an integer, not a character. Video attributes can be combined with a character by ORing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another with `inch` and `addch`.)

3.7.3 Writing a string

`addstr` (*str*)
`waddstr` (*win*, *str*)
`mvaddstr` (*y*, *x*, *str*)
`mvwaddstr` (*win*, *y*, *x*, *str*)

These functions write all the characters of the null terminated character string *str* on the given window. They are identical to a series of calls to `addch`.

3.7.4 Clearing areas of the screen

`erase()`
`werase(`*win*`)`

These functions copy blanks to every position in the window.

`clear()`
`wclear(`*win*`)`

These functions are like `erase` and `werase` but they also call `clearok`, arranging that the screen will be cleared on the next call to `refresh` for that window.

`clrtobot()`
`wclrtobot(`*win*`)`

All lines below the cursor in this window are erased. Also, the current line to the right of the cursor is erased.

`clrtoeol()`
`wclrtoeol(`*win*`)`

The current line to the right of the cursor is erased.

3.7.5 Inserting and deleting text

`delch()`
`wdelch(`*win*`)`
`mvdelch(`*y*`,`*x*`)`
`mvwdelch(`*win*`,`*y*`,`*x*`)`

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position. This does not imply use of the hardware delete-character feature.

`deleteln()`
`wdeleteln(`*win*`)`

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. This does not imply use of the hardware delete-line feature.

`insch`(*c*)
`winsch`(*win*, *c*)
`mvinsch`(*y*, *x*, *c*)
`mvwinsch`(*win*, *y*, *x*, *c*)

The character *c* is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. This does not imply use of the hardware insert-character feature.

`insertln()`
`winsertln`(*win*)

A blank line is inserted above the current line. The bottom line is lost. This does not imply use of the hardware insert-line feature.

3.7.6 Formatted output

`printw`(*fmt*, *args*)
`wprintw`(*win*, *fmt*, *args*)
`mvprintw`(*y*, *x*, *fmt*, *args*)
`mvwprintw`(*win*, *y*, *x*, *fmt*, *args*)

These functions correspond to `printf`. The characters that would be produced by `printf` are instead produced using `waddch` on the given window.

3.7.7 Miscellaneous

`box`(*win*, *vert*, *hor*)

A box is drawn around the edge of the window. *vert* and *hor* are the characters with which the box is to be drawn.

`scroll`(*win*)

The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is `stdscr` and the scrolling region is the entire window, the physical screen is scrolled at the same time.

3.8 Input from a window

`getyx`(*win*, *y*, *x*)

The cursor position of the window is placed in the two integer variables *y* and *x*. Since this is a macro, no `&` is necessary for *x* or *y*.

`inch()`
`winch(`*win*`)`
`mvinch(`*y*`,`*x*`)`
`mvwinch(`*win*`,`*y*`,`*x*`)`

The character at the current position in the named window is returned. If any attributes are set for that position, their values will be ORed into the value returned. The predefined constants `A_ATTRIBUTES` and `A_CHARTEXT` can be used with the `&` operator to extract the character or attributes alone. For example:

```
#include <curses.h>
...
      char c;
      ...
      c = inch() & A_CHARTEXT;
      ...
```

3.9 Input from the terminal

`getch()`
`wgetch(`*win*`)`
`mvgetch(`*y*`,`*x*`)`
`mvwgetch(`*win*`,`*y*`,`*x*`)`

A character is read from the terminal associated with the window. In `nodelay` mode, if there is no input waiting, the value −1 is returned. In `delay` mode, the program hangs until the system passes text through to the program. Depending on the setting of `cbreak` , this is after one character, or after the first newline.

If `keypad` mode is enabled, and a function key is pressed, the code for that function key is returned instead of the raw characters. Possible function keys are defined with integers beginning with 0401, whose names begin with `KEY_`. These are listed in the section "Input" above. If a character is received that could be the beginning of a function key (such as ESCAPE), `curses` sets a 1-second timer. If the remainder of the sequence does not come in within 1 second, the character is passed through; otherwise the function key value is returned. For this reason, on many terminals, there will be a 1-second delay after a user presses the ESCAPE key. (Using the ESCAPE key for a single character function is discouraged.)

`getstr`(*str*)
`wgetstr`(*win*, *str*)
`mvgetstr`(*y*, *x*, *str*)
`mvwgetstr`(*win*, *y*, *x*, *str*)

A series of calls to `getch` is made, until a newline is received. The resulting value is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted.

`scanw`(*fmt*, *args*)
`wscanw`(*win*, *fmt*, *args*)
`mvscanw`(*y*, *x*, *fmt*, *args*)
`mvwscanw`(*win*, *y*, *x*, *fmt*, *args*)

This function corresponds to `scanf`. `wgetstr` is called on the window, and the resulting line is used as input for the scan.

3.10 Video attributes

`attroff`(*at*)
`wattroff`(*win*, *attrs*)
`attron`(*at*)
`wattron`(*win*, *attrs*)
`attrset`(*at*)
`wattrset`(*win*, *attrs*)
`standout`()
`standend`()
`wstandout`(*win*)
`wstandend`(*win*)

These functions set the current attributes of the named window. These attributes can be any combination of `A_STANDOUT`, `A_REVERSE`, `A_BOLD`, `A_DIM`, `A_BLINK`, and `A_UNDERLINE`. These constants are defined in `<curses.h>` and can be combined with the C language OR operator (|).

The current attributes of a window are applied to all characters that are written into the window with `waddch`. Attributes are a property of the character and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they are displayed as the graphic rendition of characters put on the screen.

```
attrset(at)
```

sets the current attributes of the given window to *at*.

```
attroff(at)
```

turns off the named attributes without affecting any other attributes.

```
attron(at)
```

turns on the named attributes without affecting any others.

```
standout
```

is the same as `attron(A_STANDOUT)`.

```
standend
```

is the same as `attrset(0)`; that is, it turns off all attributes.

3.11 Bells and flashing lights

`beep()`
`flash()`

These functions are used to signal the user. `beep` sounds the audible alarm on the terminal, if possible, and, if not, flashes the screen (visible bell), if that is possible. `flash` flashes the screen, and, if that is not possible, sounds the audible signal. If neither signal is possible, nothing happens. Nearly all terminals have an audible signal (a bell or beep) but only some can flash the screen.

3.12 Portability functions

The functions described here do not directly involve terminal-dependent character output but tend to be needed by programs that use `curses`. Unfortunately, their implementation varies from one version of UNIX to another. They have been included here to enhance the portability of programs using `curses`.

`baudrate()`

`baudrate` returns the output speed of the terminal. The number returned is the integer baud rate, for example, 9600, rather than a table index such as `B9600`.

`erasechar()`

The erase character chosen by the user is returned. This is the character typed by the user to erase the character just typed.

`killchar()`

The line-kill character chosen by the user is returned. This is the character typed by the user to abort the entire line being typed.

`flushinp()`

This function throws away any typeahead that has been typed by the user and has not yet been read by the program.

3.13 Delays

The functions described here are highly unportable, but are often needed by programs that use `curses`, especially real-time response programs. Some of these functions require a particular operating system or a modification to the operating system to work. In all cases, the routine compiles and returns an error status if the requested action is not possible. It is recommended that you avoid use of these functions if possible.

`draino(`*ms*`)`

The program is suspended until the output queue has drained enough to complete in *ms* additional milliseconds. Thus,

```
draino(50)
```

at 1200 baud would pause until there are no more than six characters in the output queue, because it would take 50 milliseconds to output the additional six characters. The purpose of this routine is to keep the program (and thus the keyboard) from getting ahead of the screen. If the operating system does not support the `ioctls` needed to implement `draino`, the value `ERR` is returned; otherwise, `OK` is returned.

`napms(`*ms*`)`

This function suspends the program for *ms* milliseconds. It is similar to `sleep` except with higher resolution. The resolution actually provided varies with the facilities available in the operating system, and often a change to the operating system is necessary to produce good results. If resolution of at least .1 second is not possible, the routine rounds to the next higher second, calls `sleep`, and returns `ERR`. Otherwise, the value `OK` is returned. Often the resolution provided is 1/60th second.

3.14 Lower-level functions

The functions described here are provided for programs not needing the screen optimization capabilities of `curses`. Programs are discouraged from working at this level, because they must handle various glitches in certain terminals. However, a program can be smaller if it only brings in the low-level routines.

3.14.1 Cursor motion

`mvcur` (*oldrow*, *oldcol*, *newrow*, *newcol*)

This routine optimally moves the cursor from (*oldrow, oldcol*) to (*newrow, newcol*). The user program is expected to keep track of the current cursor position. Note that unless a full screen image is kept, `curses` will have to make pessimistic assumptions, sometimes resulting in less than optimal cursor motion. For example, moving the cursor a few spaces to the right can be done by transmitting the characters being moved over, but if `curses` does not have access to the screen image, it does not know what these characters are.

3.14.2 `terminfo` level

These routines are called by low-level programs that need access to specific capabilities of `terminfo`. A program working at this level should include both `<curses.h>` and `<term.h>`, in that order. After a call to `setupterm`, the capabilities will be available with macro names defined in `<term.h>`. See `terminfo`(4) for a detailed description of the capabilities.

Boolean-valued capabilities will have the value 1 if the capability is present, 0 if it is not. Numeric capabilities have the value −1 if the capability is missing, and have a value at least 0 if it is present. String capabilities (both those with and those without parameters) have the value `NULL` if the capability is missing, and otherwise have type

```
char *
```

and point to a character string containing the capability. The special character codes involving the `\` and `^` characters (such as `\r` for RETURN, or `^A` for CONTROL-a) are translated into the appropriate ASCII characters. Padding information (of the form `$<time>`) and parameter information (beginning with `%`) are left uninterpreted at this stage. The routine `tputs` interprets padding information, and `tparm` interprets parameter information.

If the program needs to handle only one terminal, the definition `-DSINGLE` can be passed to the C compiler, resulting in static references to capabilities instead of dynamic references. This can result in smaller code, but prevents use of more than one terminal at a time. Very few programs use more than one terminal, so almost all programs can use this flag.

`setupterm(`*term*`,`*filenum*`,` *errret*`)`

This routine is called to initialize a terminal. *term* is the character string representing the name of the terminal being used. *filenum* is the A/UX file descriptor of the terminal being used for output. *errret* is a pointer to an integer, in which a success or failure indication is returned. The values returned can be 1 (all is well), 0 (no such terminal), or −1 (some problem locating the `terminfo` data base).

The value of *term* can be given as 0, which causes the value of `TERM` in the environment to be used. The *errret* pointer can also be given as 0, meaning no error code is wanted. If *errret* is defaulted, and something goes wrong, `setupterm` prints an appropriate error message and exits, rather than returning. Thus, a simple program can call `setupterm(0,1,0)` and not worry about initialization errors.

If the variable `TERMINFO` is set in the environment to a pathname, `setupterm` checks for a compiled `terminfo` description of the terminal under that path, before checking `/usr/lib/terminfo`. Otherwise, only `/usr/lib/terminfo` is checked.

`setupterm` checks the TTY driver mode bits, using *filenum*, and changes any that might prevent the correct operation of other low-level routines. Currently, the mode that expands tabs into spaces is disabled, because the tab character is sometimes used for different functions by different terminals. (Some terminals use it to move right one space. Others use it to address the cursor to row or column 9.) If the system is expanding tabs, `setupterm` removes the definition of the `tab` and `backtab` functions, making the assumption that because the user is not using hardware tabs, they may not be properly set in the terminal. Other system-dependent changes, such as disabling a virtual terminal driver, may be made here.

As a side effect, `setupterm` initializes the global variable `ttytype`, which is an array of characters, to the value of the list of names for the terminal. This list comes from the beginning of the `terminfo` description.

After the call to `setupterm`, the global variable `cur_term` is set to point to the current structure of terminal capabilities. By calling `setupterm` for each terminal, and saving and restoring `cur_term`, it is possible for a program to use two or more terminals at once.

The mode that turns newlines into ''carriage return-line feed'' on output is not disabled. Programs that use `cursor_down` or `scroll_forward` should avoid these capabilities if their value is line feed unless they disable this mode. `setupterm` calls `reset_prog_mode` after any changes it makes.

`def_prog_mode()`
`def_shell_mode()`
`reset_prog_mode()`
`reset_shell_mode()`

These routines can be used to change the TTY modes between the two states: shell (the mode they were in before the program was started) and program (the mode needed by the program). `def_prog_mode` saves the current terminal mode as program mode. `setupterm` and `initscr` call `def_shell_mode` automatically. `reset_prog_mode` puts the terminal into program mode, and `reset_shell_mode` puts the terminal into normal mode. A typical calling sequence is for a program to call `initscr` (or `setupterm` if a `terminfo`-level program), then to set the desired program mode by calling routines such as `cbreak` and `noecho`, and then to call `def_prog_mode` to save the current state. Before a shell escape or CONTROL-z suspension, the program should call `reset_shell_mode`, to restore normal mode for the shell. Then, when the program resumes, it should call `reset_prog_mode`. Also, all programs must call `reset_shell_mode` before they exit. (The higher level routine `endwin` automatically calls `reset_shell_mode`.)

Normal mode is stored in

```
cur_term->Ottyb,
```

and program mode is in

```
cur_term->Nttyb
```

These structures are both of type `SGTTYB` (which varies depending on the system). Currently the possible types are

```
struct sgttyb
```

(on some other systems) and

```
struct termio
```

(on this version of the A/UX system). `def_prog_mode` should be called to save the current state in `Nttyb`.

`vidputs(`*newmode*`, `*putc*`)`
: `newmode` is any combination of attributes, defined in `<curses.h>`. `putc` is a `putchar`-like function. The proper string to `put` the terminal in the given video mode is generated. The previous mode is remembered by this routine. The result characters are passed through `putc`.

`vidattr(`*newmode*`)`
: The proper string to `put` the terminal in the given video mode is output to `stdout`.

`tparm(`*instring*`,`*p1*`,`*p2*`,`*p3*`,`*p4*`,`*p5*`,`*p6*`,`*p7*`,`*p8*`,`*p9*`)`
: `tparm` is used to instantiate a parameterized string. The character string returned has the given parameters applied, and is suitable for `tputs`. Up to nine parameters can be passed, in

addition to the parameterized string.

`tputs` (*cp*, *affcnt*, *outc*)
: A string capability, possibly containing padding information, is processed. Enough padding characters to delay for the specified time replace the padding specification, and the resulting string is passed, one character at a time, to the routine *outc*, which should expect one character parameter. (This routine often just calls `putchar`.) *cp* is the capability string. *affcnt* is the number of units affected by the capability, which varies with the particular capability. (For example, the *affcnt* for `insert_line` is the number of lines below the inserted line on the screen, that is, the number of lines that will have to be moved by the terminal.) *affcnt* is used by the padding information of some terminals as a multiplication factor. If the capability does not have a factor, the value 1 should be passed.

`putp` (*str*)
: This is a convenient function to output a capability with no *affcnt*. The string is output to `putchar` with an *affcnt* of 1. It can be used in simple applications that do not need to process the output of `tputs`.

`delay_output` (*ms*)
: A delay is inserted into the output stream for the given number of milliseconds. The current implementation inserts sufficient pad characters for the delay. This should not be used in place of a high-resolution sleep, but rather for delay effects in the output. Due to buffering in the system, it is unlikely that this call will result in the process actually sleeping. Because large numbers of pad characters can be generated, it is recommended that *ms* not exceed 500.

4. Operation details

These paragraphs describe many of the details of how the `curses` and `terminfo` package operates.

4.1 Insert and delete line and character

The algorithm used by `curses` takes into account insert and delete line and character functions, if available, in the terminal. Calling the routine

```
   idlok(stdscr, TRUE);
```

enables insert/delete line. By default, `curses` does not use insert/delete line. This was not done for performance reasons, because there is no speed penalty involved. Rather, experience has shown that some programs do not need this facility, and that if `curses` uses insert/delete line, the result on the screen can be visually annoying. Many simple programs using `curses` do not need this, so the default is to avoid insert/delete line. Insert/delete character is always considered.

4.2 Additional terminals

`curses` will work even if absolute cursor addressing is not possible, as long as the cursor can be moved from any location to any other location. It considers local motions, parameterized motions, home, and carriage return.

`curses` is aimed at full-duplex, alphanumeric, video terminals. No attempt is made to handle half-duplex, synchronous, hard copy, or bit-mapped terminals. Bit-mapped terminals can be handled by programming the bit-mapped terminal to emulate an ordinary alphanumeric terminal. This does not take advantage of the bit-map capabilities, but it is the fundamental nature of `curses` to deal with alphanumeric terminals.

The `curses` package handles terminals with the ''magic cookie glitch'' in their video attributes. The term ''magic cookie'' means that a change in video attributes is implemented by storing a magic cookie in a location on the screen. This cookie takes up a space, preventing an exact implementation of what the programmer wanted. `curses` takes the extra space into account, and moves part of the line to the right, as necessary. Advantage is taken of existing spaces, but in some cases, this unavoidably results in losing text from the right edge of the screen.

4.3 Multiple terminals

Some applications need to display text on more than one terminal, controlled by the same process. Even if the terminals are of different types, `curses` can handle this.

All information about the current terminal is kept in a global variable

```
struct screen *SP;
```

Although the screen structure is hidden from the user, the C compiler will accept declarations of variables that are pointers. The user program should declare one screen pointer variable for each terminal it wishes to handle. The routine

```
struct screen *newterm(type,fd)
```

sets up a new terminal of the given terminal type, which does output on file descriptor *fd*. A call to `initscr` is essentially

```
newterm(getenv("TERM"),stdout)
```

A program wishing to use more than one terminal should use `newterm` for each terminal and save the value returned as a reference to that terminal.

To switch to a different terminal, call

```
set_term(term)
```

The old value of `SP` is returned. The programmer should not assign directly to `SP` because certain other global variables must also be changed.

All `curses` routines always affect the current terminal. To handle several terminals, switch to each one in turn with `set_term`, and then access it. Each terminal must be set up with `newterm`, and closed down with `endwin`.

4.4 Video attributes

Video attributes can be displayed in any combination on terminals with this capability. They are treated as an extension of the standout capability, which is still present.

Each character position on the screen has 16 bits of information associated with it. Seven of these bits are the character to be displayed, leaving separate bits for nine video attributes. These bits are used for standout, underline, reverse video, blink, dim, bold, blank, protect, and alternate character set. Standout is taken to be whatever highlighting works best on the terminal, and should be used by any program that does not need specific or combined attributes. Underlining, reverse video, blink, dim, and bold are the usual video attributes. Blank means that the character is displayed as a space, for security reasons. Protected and alternate character set depend on the particular terminal. The use of these last three bits is subject to change and not recommended. Note also that not all terminals implement all attributes—in particular, no current terminal implements both dim and bold.

The routines to use these attributes include

`attrset(`*attrs*`)`	`wattrset(`*win*`,` *attrs*`)`
`attron(`*attrs*`)`	`wattron(`*win*`,` *attrs*`)`
`attroff(`*attrs*`)`	`wattroff(`*win*`,` *attrs*`)`
`standout()`	`wstandout(`*win*`)`
`standend()`	`wstandend(`*win*`)`

Attributes, if given, can be any combination of

```
A_STANDOUT
A_UNDERLINE
A_REVERSE
A_BLINK
A_DIM
A_BOLD
A_INVIS
A_PROTECT
A_ALTCHARSET
```

These constants, defined in `curses.h`, can be combined with the C language OR operator (|) to get multiple attributes.

`attrset(`*attrs*`)`	Sets the current attributes to the given *attrs*
`attron(`*attrs*`)`	Turns on the given *attrs* in addition to any attributes that are already on
`attroff(`*attrs*`)`	Turns off the given *attrs*, without affecting any others
`standout()` `standend()`	Equivalent to `attron(A_STANDOUT)` `attrset(A_NORMAL)`

If the particular terminal does not have the particular attribute or combination requested, `curses` will attempt to use some other attribute in its place. If the terminal has no highlighting at all, all attributes will be ignored.

4.5 Special keys

Many terminals have special keys, such as arrow keys, keys to erase the screen or insert or delete text, and keys intended for user functions. The particular sequences these terminals send differ from terminal to terminal. `curses` allows the programmer to handle these keys.

A program using special keys should turn on the keypad by calling

```
keypad(stdscr, TRUE)
```

at initialization. This causes special characters to be passed through to the program by the function `getch`. These keys have constants that are listed in the section on "Input" above. They have values starting at 0401, so they should not be stored in a `char` variable, as significant bits will be lost.

A program using special keys should avoid using the ESCAPE key, because most sequences start with escape, creating an ambiguity. `curses` will set a 1-second alarm to deal with this ambiguity, which will cause delayed response to the ESCAPE key. It is a good idea to avoid escape in any case, since there is eventually pressure for nearly *any* screen-oriented program to accept arrow-key input.

4.6 Scrolling region

There is a programmer-accessible scrolling region. Normally, the scrolling region is set to the entire window, but the calls

```
setscrreg(top,bot)
wsetscrreg(win,top,bot)
```

set the scrolling region for `stdscr` or the given window to any combination of top and bottom margins. When scrolling past the bottom margin of the scrolling region, the lines in the region move up one line, destroying the top line of the region. If scrolling has been enabled with `scrollok`, scrolling takes place only within that window. Note that the scrolling region is a software feature, and only causes a window data structure to scroll. This may or may not translate to use of the hardware scrolling-region feature of a terminal or of insert/delete line; some ''intelligent'' terminals perform these operations rather than being controlled directly by the software.

4.7 Mini-curses

`curses` copies from the current window to an internal screen image for every call to `refresh`. If the programmer is interested only in screen output optimization and does not want the windowing or input functions, an interface to the lower-level routines is available. This will make the program somewhat smaller and faster. The interface is a subset of full `curses`, so that conversion between the levels is not necessary to switch from mini-`curses` to full `curses`.

The following functions of `curses` and `terminfo` are available to the user of mini-`curses`:

addch(*ch*)	addstr(*str*)	attroff(*attrs*)
attron(*attrs*)	ttrset(*at*)	clear()
erase()	initscr	move(*y*,*x*)
mvaddch(*y*,*x*,*ch*)	mvaddstr(*y*,*x*,*str*)	newterm
refresh()	standend()	standout()

The following functions of `curses` and `terminfo` are not available to the user of mini-`curses`:

box	clrtobot	clrtoeol
delch	deleteln	delwin
getch	getstrs	inch
insch	insertln	longname
makenew	mvdelch	mvgetch
mvgetstr	mvinch	mvinsch
mvprintw	mvscanw	mvwaddch
mvwaddstr	mvwdelch	mvwgetch
mvwgetstr	mvwin	mvwinch
mvwinsch	mvwprintw	mvwscanw
newwin	overlay	overwrite
printw	putp	scanw
scroll	setscrreg	subwin
touchwin	vidattr	waddch
waddstr	wclear	wclrtobot
wclrtoeol	wdelch	wdeleteln

werase	wgetch	wgetstr
winsch	winsertln	wmove
wprintw	wrefresh	wscanw
wsetscrreg		

The subset mainly requires the programmer to avoid use of more than the one-window stdscr. Thus, all functions beginning with w are generally undefined. Certain high-level functions that are convenient but not essential are also not available, including printw and scanw. Also, the input routine getch cannot be used with mini-curses. Features implemented at a low level, such as use of hardware insert/delete line and video attributes, are available in both versions. Also, mode setting routines such as crmode and noecho are allowed.

To access mini-curses, add –DMINICURSES to the CFLAGS in the makefile. If routines are requested that are not in the subset, the loader will print error messages such as

```
Undefined:
m_getch
m_waddch
```

to tell you that the routines getch and waddch were used but are not available in the subset. Because the preprocessor is involved in the implementation of mini-curses, the entire program must be recompiled when changing from one version to the other.

4.8 TTY mode functions

In addition to the save/restore routines savetty and resetty, standard routines are available for going into and out of normal TTY mode. These routines are resetterm, which puts the terminal back in the mode it was in when curses was started; fixterm, which undoes the effects of resetterm, that is, restores the "current curses mode"; and saveterm, which saves the current state to be used by fixterm. endwin automatically calls resetterm, and the routine to handle CONTROL-z (on other systems that have process control) also uses resetterm and fixterm. Programmers should use these routines before and after shell escapes, and also if they write their own routine to handle CONTROL-z. These routines are also available at the terminfo level.

4.9 Typeahead check

If the user types something during an update, the update stops, pending a future update. This is useful when the user hits several keys, each of which causes a good deal of output. For example, in a screen editor, if the user presses the ''forward screen'' key, which draws the next screenful of text, several times rapidly, rather than drawing several screens of text, the updates are cut short, and only the last screenful is actually displayed. This feature is automatic and cannot be disabled.

4.10 `getstr`

No matter what the setting of the stty *echo* is, strings typed in here are echoed at the current cursor location. The user's *erase* and *kill* characters are understood and handled. This makes it unnecessary for an interactive program to deal with erase, kill, and echoing when the user is typing a line of text.

4.11 `longname`

The `longname` function does not need any arguments. It returns a pointer to a static area containing the actual long name of the terminal.

4.12 `nodelay` mode

The call

```
   nodelay(stdscr, TRUE)
```

puts the terminal in `nodelay` mode. While in this mode, any call to `getch` returns –1 if there is nothing waiting to be read immediately. This is useful for writing programs requiring ''real-time'' behavior, where the users watch action on the screen and press a key when they want something to happen. For example, the cursor can be moving across the screen, in real time. When it reaches a certain point, the user can press an arrow key to change direction at that point.

4.13 Portability

Several useful routines are provided to improve portability. The implementation of these routines is different from system to system, and the differences can be isolated from the user program by including them in `curses`.

`erasechar()`
: Returns the character that erases one character.

`killchar()`
: Returns the character that kills the entire input line.

`baudrate()`
: Returns the current baud rate as an integer. (For example, at 9600 baud, the integer 9600 is returned, not the value `B9600` from `<sgtty.h>`.)

`flushinp()`
: Causes all typeahead to be thrown away.

5. Example program: `scatter`

```
/*
 * scatter: this program takes the first
 * screenful of lines from the standard
 * input and displays them on the
 * VDU screen, in a random manner.
 */

#include <curses.h>

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];      /* Screen array */

main()
{
   register int row,col;
   register char c;
   int char_count=0;
   long t;
   char buf[BUFSIZ];

   initscr();
   for(row=0;row<MAXLINES;row++)
      for(col=0;col<MAXCOLS;col++)
         s[row][col]=' ';

   row = 0;
   /* Read screen in */
   while( (c=getchar()) != EOF && row < LINES ) {
      if(c != '\n') {
         /* Place char in screen array */
         s[row][col++] = c;
         if(c != ' ')
            char_count++;
      } else {
          col=0;
          row++;
      }
   }

   time(&t);     /* Seed random number generator */
   srand((int)(t&0177777L));

   while(char_count) {
      row=rand() % LINES;
      col=(rand()>>2) % COLS;
      if(s[row][col] != ' ')
      {
         move(row, col);
         addch(s[row][col]);
         s[row][col]=EOF;
         char_count--;
         refresh();
      }
   }
   endwin();
   exit(0);
}
```

6. Example program: show

```
/*
 * The show program pages through
 * a file, showing one full screen each
 * time the user presses the space bar
 */
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
   FILE *fp;
   char linebuf[BUFSIZ];
   int line;
   void done(), perror(), exit();

   if(argc != 2)
   {
      fprintf(stderr,"usage: %s file\n", argv[0]);
      exit(1);
   }
   if((fp=fopen(argv[1],"r")) == NULL)
   {
      perror(argv[1]);
      exit(2);
   }
   signal(SIGINT, done);

   initscr();
   noecho();
   cbreak();
   nonl();
   idlok(stdscr, TRUE);

   while(1)
   {
      move(0,0);
      for(line=0; line<LINES; line++)
      {
      if(fgets(linebuf, sizeof linebuf, fp) == NULL)
```

```
            {
               clrtobot();
               done();
            }
            move(line, 0);
            printw("%s", linebuf);
            }
            refresh();
            if(getch() == 'q')
            done();
        }
}


void
done()
{
    move(LINES-1, 0);
    clrtoeol();
    refresh();
    endwin();
    exit(0);
}
```

7. Example program: highlight

```
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */
#include <curses.h>

main(argc, argv)
char **argv;
{
   FILE *fp;
   int c, c2;

   if (argc != 2) {
      fprintf(stderr, "Usage: highlight file\n");
      exit(1);
   }

   fp = fopen(argv[1], "r");
   if (fp == NULL) {
      perror(argv[1]);
      exit(2);
   }

   initscr();
   scrollok(stdscr, TRUE);

   for (;;) {
      c = getc(fp);
      if (c == EOF)
         break;
      if (c == '\\') {
         c2 = getc(fp);
      switch (c2) {
      case 'B':
          attrset(A_BOLD);
          continue;
      case 'U':
          attrset(A_UNDERLINE);
          continue;
```

```
        case 'N':
            attrset(0);
            continue;
        }
        addch(c);
        addch(c2);
        }
        else
        addch(c);
    }
    fclose(fp);
    refresh();
    endwin();
    exit(0);
}
```

8. Example program: window

```
/*
 * This program shows the use of multiple windows.
 * The main display is kept in stdscr.
 * When the user temporarily wants to put
 * something else on the screen,
 * a new window is created covering
 * part of the screen.
 */
#include <curses.h>

WINDOW *cmdwin;

main()
{
   int i, c;
   char buf[120];

   initscr();
   nonl();
   noecho();
   cbreak();

   /* top 3 lines */
   cmdwin = newwin(3, COLS, 0, 0);
   for (i=0; i<LINES; i++)
      mvprintw(i, 0, "This is line %d of stdscr", i);

   for (;;) {
      refresh();
      c = getch();
      switch (c) {
      case 'c':      /* Enter command from keyboard */
         werase(cmdwin);
         wprintw(cmdwin, "Enter command:");
         wmove(cmdwin, 2, 0);
         for (i=0; i<COLS; i++)
         waddch(cmdwin, '-');
         wmove(cmdwin, 1, 0);
         touchwin(cmdwin);
         wrefresh(cmdwin);
         wgetstr(cmdwin, buf);
```

```
                touchwin(stdscr);
                /*
                * The command is now in buf.
                * It should be processed here.
                */
                break;
               case 'q':
                endwin();
                exit(0);
               }
        }
}
```

9. Example program: two

```
/*
 * The two program pages through a file,
 * showing one page to the first terminal and
 * the next page to the second terminal
 * It then waits for a space to be typed on
 * either terminal, and shows the next
 * page to the terminal typing the space.
 */
#include <curses.h>
#include <signal.h>

struct screen *me, *you;
struct screen *set_term();

FILE *fp, *fpyou;
char linebuf[512];

main(argc, argv)
char **argv;
{
   int done();
   int c;

   if (argc != 4) {
      fprintf(stderr,
      "Usage: two otherrtty othertttytype inputfile\n");
      exit(1);
   }

   fp = fopen(argv[3], "r");
   fpyou = fopen(argv[1], "w+");
   signal(SIGINT, done);
   /* die gracefully */

   me = newterm(getenv("TERM"), stdout);
   /* initialize my tty */
   you = newterm(argv[2], fpyou);
   /* Initialize his terminal */

   /* Set modes for my terminal */
   set_term(me);
```

```
        noecho();                  /* turn off tty echo */
        cbreak();                  /* enter cbreak mode */
        nonl();                    /* Allow linefeed */
        nodelay(stdscr,TRUE);      /* No hang on input */

        /* Set modes for other terminal */;
        set_term(you)
        noecho();
        cbreak();
        nonl();
        nodelay(stdscr,TRUE);

        /* Dump first screenful on my terminal */
        dump_page(me);

        /* Dump second screenful on his terminal */
        dump_page(you);

        /* for each screenful */
        for (;;) {
            set_term(me);
            c = getch();
            /* wait for user to read it */)
            if (c == 'q'
            done();
            if (c == ' ')
            dump_page(me);

            set_term(you);
            c = getch();
            /* wait for user to read it */
            if (c == 'q')
            done();
            if (c == ' ')
            dump_page(you);
            sleep(1);
        }
    }

    dump_page(term)
    struct screen *term;
    {
```

```
    int line;

    set_term(term);
    move(0, 0);
    for (line=0; line<LINES-1; line++) {
       if (fgets(linebuf, sizeof linebuf, fd) == NULL)
       {
       clrtobot();
       done();
       }
       mvprintw(line, 0, "%s", linebuf);
    }
    standout();
    mvprintw(LINES-1, 0, "--More--");
    standend();
    refresh();                    /* sync screen */
}

/*
 * Clean up and exit.
 */
done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES-1,0);              /* to lower left corner */
    clrtoeol();                   /* clear bottom line */
    refresh();                    /* flush out everything */
    endwin();                     /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES-1,0);              /* to lower left corner */
    clrtoeol();                   /* clear bottom line */
    refresh();                    /* flush out everything */
    endwin();                     /* curses cleanup */

    exit(0);
}
```

10. Example program: `termhl`

```
/*
 * A terminfo-level version of highlight.
 */
#include <curses.h>
#include <term.h>

int ulmode = 0;          /* Currently underlining */

main(argc, argv)
char **argv;
{
   FILE *fp;
   int c, c2;
   int outch();

   if (argc > 2) {
      fprintf(stderr, "Usage: termhl [file]\n");
      exit(1);
   }

   if (argc == 2) {
      fp = fopen(argv[1], "r");
      if (fp == NULL) {
      perror(argv[1]);
      exit(2);
      }
   } else {
      fp = stdin;
   }

   setupterm(0, 1, 0);

   for (;;) {
      c = getc(fp);
      if (c == EOF)
      break;
      if (c == '\\') {
      c2 = getc(fp);
      switch (c2) {
      case 'B':
         tputs(enter_bold_mode, 1, outch);
```

```
            continue;
        case 'U':
            tputs(enter_underline_mode, 1, outch);
            ulmode = 1;
            continue;
        case 'N':
            tputs(exit_attribute_mode, 1, outch);
            ulmode = 0;
            continue;
        }
        putch(c);
        putch(c2);
        }
        else
        putch(c);
    }
    fclose(fp);
    fflush(stdout);
    resetterm();
    exit(0);
}

/*
 * This function is like putchar,
 * but it checks for underlining.
 */
putch(c)
int c;
{
    outch(c);
    if (ulmode && underline_char) {
        outch('\b');
        tputs(underline_char, 1, outch);
    }
}

/*
 * Outchar is a function version
 * of putchar that can be passed to
 * tputs as a routine to call.
 */
outch(c)
```

```
int c;
{
   putchar(c);
}
```

11. Example program: **editor**

```
/*
 * editor: A screen-oriented editor.  The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr itself to simplify
 * the program.
 */

#include <curses.h>

#define CTRL(c) ('c' & 037)

main(argc, argv)
int argc;
char **argv;
{
   int i, n, l;
   int c;
   FILE *fp;

   if (argc != 2) {
      fprintf(stderr, "Usage: edit file\n");
      exit(1);
   }

   fp = fopen(argv[1], "r");
   if (fp == NULL) {
      perror(argv[1]);
      exit(2);
   }

   initscr();
   cbreak();
   nonl();
   noecho();
   idlok(stdscr, TRUE);
   keypad(stdscr, TRUE);

   /* Read in the file */
   while ((c = getc(fp)) != EOF)
      addch(c);
   fclose(fp);
   move(0,0);
   refresh();
   edit();

   /* Write out the file */
   fp = fopen(argv[1], "w");
   for (l=0; l<23; l++) {
      n = len(l);
      for (i=0; i<n; i++)
      putc(mvinch(l, i), fp);
      putc('\n', fp);
   }
   fclose(fp);

   endwin();
   exit(0);
}

len(lineno)
int lineno;
{
   int linelen = COLS-1;

   while (linelen >=0
         && mvinch(lineno, linelen) == ' ')
      linelen--;
   return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
   int c;

   for (;;) {
      move(row, col);
      refresh();
      c = getch();
      switch (c) {        /* Editor commands */
```

```
/* hjkl and arrow keys: move cursor */
/* in direction indicated */
case 'h':
case KEY_LEFT:
if (col > 0)
   col--;
break;

case 'j':
case KEY_DOWN:
if (row < LINES-1)
   row++;
break;

case 'k':
case KEY_UP:
if (row > 0)
   row--;
break;

case 'l':
case KEY_RIGHT:
if (col < COLS-1)
   col++;
break;

/* i: enter input mode */
case KEY_IC:
case 'i':
input();
break;

/* x: delete current character */
case KEY_DC:
case 'x':
delch();
break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
move(++row, col=0);
```

```
        insertln();
        input();
        break;

        /* d: delete current line */
        case KEY_DL:
        case 'd':
        deleteln();
        break;

        /* ^L: redraw screen */
        case KEY_CLEAR:
        case CTRL(L):
        clearok(curscr);
        refresh();
        break;

        /* w: write and quit */
        case 'w':
        return;

        /* q: quit without writing */
        case 'q':
        endwin();
        exit(1);
        default:
        flash();
        break;
        }
    }
}

/*
 * Insert mode: accept characters and insert them.
 * End with ^D or EIC
 */
input()
{
    int c;

    standout();
    mvaddstr(LINES-1, COLS-20, "INPUT MODE");
    standend();
    move(row, col);
    refresh();
    for (;;) {
        c = getch();
        if (c == CTRL(D) || c == KEY_EIC)
        break;
        insch(c);
        move(row, ++col);
        refresh();
    }
    move(LINES-1, COLS-20);
    clrtoeol();
    move(row, col);
    refresh();
}
```

Chapter 25
Other Programming Tools

Contents

Chapter 25

Other Programming Tools

1. Overview

This chapter provides a brief introduction to some of the other programming tools available on the A/UX system. Some of these commands group together naturally. For example, if you are creating an archive library, you probably will want to familiarize yourself with each of `ar`, `lorder`, and `tsort`. If you need to identify unfamiliar binary files, you have a choice between using `strings` and using `od` (with the `-c` option) to isolate the printable portions of these files (see the section on `od` for both commands).

2. Maintaining portable archives and libraries: `ar`

You may use the archive command `ar` to combine several files into one archive. An **archive** consists of a collection of files, plus a table of contents. They are used mainly as libraries to be searched by the link editor `ld`. A **library** (or library archive) is an archive that contains object files (plus a table of contents). Putting together your own library allows you to use locally produced functions (instead of limiting you to the functions supplied in standard libraries).

`ar` also provides the facility to append files to and delete files from the archive. Because the order of files is so important to `ld`'s efficient operation, you can also move files around within the archive, as well as extract them, print them, and produce a table of contents. See `ar`(1) in *A/UX Command Reference* for more information.

3. Beautifying C programs: `cb`

`cb` is used to improve the legibility and structure of your own or someone else's C code. It reads C programs either from its arguments or from the standard input and writes them on the standard output with spacing and indentation that displays the structure of the code. See `cb`(1) in *A/UX Command Reference* for more information.

4. Generating a C flowgraph: `cflow`

`cflow` generates a C flowgraph. A C flowgraph gives an idea of the following:

- How the program is put together
- The program's flow of control
- How subroutines are called (that is, by which other routines and in which order)

This flowgraph shows the order in which routines are called *graphically,* by level of indentation. The graph is built of external references, which include globals and function calls. See `cflow`(1) in *A/UX Command Reference* for more information.

5. A C language preprocessor: `cpp`

You can use `cpp`, the C preprocessor, as a simple programming language that takes less time to compile than more complex languages. It strips comments, expands macros into their definitions, allows files to be read in (via `#includes`), and provides a facility for conditional command execution. This means that you can intersperse text with comments. Comments will be stripped; commands will be executed.

Normally, `cpp` is invoked automatically as (the first) part of the `cc` command.

You can use `m4`, instead of `cpp`, if you need a macro facility. `m4` is generally much more powerful than `cpp` as a macro processor. (For instance, `m4` allows recursive macro substitutions, while `cpp` does not.)

`cpp` is useful for

- Stripping comments
- Standardizing included definitions among many files for one project
- Debugging (certain commands executed if in this mode, others if not)
- Minimizing file space, combining many files into one

One of the most useful applications of `cpp` is as a debugging and program control tool. Any statement included after an `#ifdef` *definition* is executed only if the *definition* has actually been defined previously by means of a `#define` statement (or a `-D`*definition* in the command line). If not, and if there is an `#else` present, the statements between it and the `#endif` are executed. Otherwise, control is resumed at the level of the statement immediately following `#endif`. See `cpp`(1) in *A/UX Command Reference* for more information.

6. Finding a function definition quickly: `ctags`

Programs can rapidly accumulate a large number of functions, either in one source file or scattered across many files. `ctags` goes through the file(s) given as its argument(s) and creates a new file, called `tags`. Each line in the file `tags` contains the following:

- The name of one function
- Where that function is located
- A scanning pattern that can be used to find the above

Unless `ctags` is used with either the `-a` (append) or the `-u` (update) option, a new `tags` file is created each time it is invoked.

Once the `tags` file is created, it can be accessed (thanks to the scanning pattern in the last field of each line) from `vi` (also from `ex`) by typing

`:ta` *function-name*

This causes the named function to appear on the editor's screen.

`ctags` may be used on Fortran and Pascal sources as well as C programs. See `ctags`(1) in *A/UX Command Reference*.

7. Comparing source files

A/UX includes a number of programs that compare files, including

`bdiff`	Used similarly to `diff`; its purpose is to allow processing of files that are too large for `diff`.
`diff`	A differential file comparator. It tells what lines differ in two files.

`diff3`	A three-way differential file comparator, which works only on files less than or equal to 64K bytes. It compares three versions of a file and publishes disagreeing ranges of text, flagged with special codes.
`diffmk`	Marks the differences between files. It compares two versions of a file and creates a third file that includes ''change mark'' commands for the `nroff` and `troff` formatters.
`diffdir`	Compares the differences in two directories of files.
`comm`	Selects or rejects lines common to two sorted files.

8. Finding files: `find`

`find` is a powerful utility that performs a depth-first recursive search for files of a given characteristic such as name, group, owner name, time of last modification or access, and so on.

See `find`(1) in *A/UX Command Reference* for more information.

9. Printing the symbol table for a COFF file: `nm`

`nm` writes the symbol table for a COFF file to standard output. This is useful for debugging. `nm` lists each symbol and its value, along with the location at which it is stored in memory. See `nm`(1) in *A/UX Command Reference* for more information.

10. Obtaining an octal dump of a file: `od`

`od` provides a means for examining binary files (usually unreadable on A/UX systems). If you need to know the function and procedure of some file available only in binary, you can try the `od` command with various options to discover what the file contains. The options correspond to available formats for interpreting either bytes, characters, or words. If no options are specified, a true octal dump is obtained, as words are interpreted in octal.

See `od`(1) in *A/UX Command Reference* for more information.

You can also use the `strings` program to write the printable ASCII strings in a binary file onto standard output. This is useful for identifying unknown binary files. See `strings(1)` in *A/UX Command Reference* for more information.

11. Displaying profile data: `prof`

`prof` displays profile data on the running of a program to aid in its optimization. For each function or global, it gives the percentage of time spent executing it, the number of times it was called, and the time (in milliseconds) per call. You must compile your program with a special option to enable profiling (see `cc`(1) in *A/UX Command Reference* for more details). See `prof`(1) in *A/UX Command Reference* for more information.

12. Printing the section sizes of COFF files: `size`

The `size` command produces size information for common object format files. See `size`(1) in *A/UX Command Reference* for more information.

13. Finding the version number of a file: `version`

`version` is useful for determining which version of a program you are running. `version` takes a list of files and reports the version number for each. If the file is not a binary, it reports that. If no version number is associated with the file, it reports that. `version` also reports the object file format of each file, that is, either `Coff object file format`, or `Old a.out object file format`.

The user may associate a version number with a file by defining a string constant at the top of the source code, such as

```
char *_Version_ = \
   "(c) Copyright 1986\
    Standard Software Version V.2.1"
```

See `version`(1) in *A/UX Command Reference* for more details.

14. Sharing strings from C programs: `xstr`

The object of using `xstr` is to share one copy of a string among several files. If you need to modify the string throughout your program, you can modify it once instead of doing global searches through all your modules. If you have, in two different files,

```
char *ptr1 = "blah";
char *ptr2 = "blah";
```

`xstr` combines this into one string, in its `strings` file, and replaces occurrences of the string in the original files with a pointer to this string. This allows for shared constant strings among several files, or possibly among several users.

In practice, use of `xstr` can save memory space. After making the `xstr` array read only, you can arrange to have multiple users share these strings, thereby saving even more memory space. See `xstr`(1) in *A/UX Command Reference* for more information.

Appendix A
Additional Reading

Introduction to Compiler Construction with UNIX
Axel T. Schreiner, H. George Friedman, Jr.
Prentice-Hall, 1985
{lex and yacc, practice}

Compilers: Principles, Techniques, and Tools
Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
Addison-Wesley, 1986
{lex and yacc, theory}

The UNIX Programming Environment
Brian W. Kernighan, Rob Pike
Prentice-Hall, 1984

1027045479