Programming Utilities Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 805–4035–10
October 1998

Contents

Preface

The *Programming Utilities Guide* provides information for developers about the special built-in programming tools available in the SunOS system.

Who Should Use This Book

This guide is intended for application programmers who are using the Solaris 2.x system.

Before You Read This Book

Readers of this guide are expected to understand the Solaris 2.x operating system, programming, and networking.

How This Book Is Organized

This guide has several chapters, each discussing a unique topic. Each chapter describes a tool that can aid you in programming. These include:

Chapter 1

TNF collects trace information about a program's execution. The Trace Implementation Format lets you insert probe points into source code to collect data for analysis.

Chapter 2

lex generates programs to be used in simple lexical analysis of text. It is a tool that solves problems by recognizing different strings of characters.

Chapter 3

yacc generates language parsers. It imposes structure on computer input and turns it into a C language function that examines the input stream.

Chapter 4

make automatically maintains, updates, and regenerates related programs and files.

Chapter 5

SCCS (Source Code Control System) allows you to control access to shared files and to keep a history of changes made to a project.

Chapter 6

m4 macro language processor creates library archives and adds or extracts files

Appendix A

System V make describes a version of make(1) that is compatible with older versions of the tool.

Other tools of interest, documented more completely in the *SunOS Reference Manual*, are listed briefly here.

ar(1)	Creates and maintains portable libraries or archives
cpp(1)	Preprocesses C language directives
dis(1)	Disassembles object code for COFF
dump(1)	Dumps (displays) selected parts of an object file

`lorder`(1)	Finds an ordering relation for an object library or archive
`mcs`(1)	Manipulates the comments section of an ELF object file
`nm`(1)	Prints a name list of an object file
`size`(1)	Displays the size of an object file
`strip`(1)	Removes symbols and relocation bits from an object file
`tsort`(1)	Performs a topological sort
`unifdef`(1)	Resolves and removes `ifdef`'ed lines from C program source.

Related Books

- *SunOS Reference Manual*

Ordering Sun Documents

The SunDocs(SM) program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of *SunExpress™ On The Internet* at `http://www.sun.com/sunexpress`.

What Typographic Changes and Symbols Mean

Command names, C code, UNIX code, system calls, header files, data structures, declarations, short examples, file names, and path names are printed in listing (constant width) font.

User input is in listing font when by itself, or bold listing font when used in combination with computer output.

Items being emphasized, variable names, and parameters are printed in *italics*.

```
Screens are used to simulate what a user will see on a
video display screen or to show program source code.

Data structure contents and formats are also shown in screens.
```

Note - The Note sign is used to emphasize points of interest, to present parenthetical information, and to cite references to other documents and commands.

CHAPTER 1

Tracing Program Execution With the TNF Utilities

This chapter explains how to use the Trace Normal Form (TNF) utilities to collect data from your program.

The TNF utilities consist of a library and a group of programs that create, manipulate, and read Trace Normal Form binary files. Use the TNF utilities to:

- Trace C or C^{++} program execution
- Debug your C or C^{++} programs
- Gather performance data on C or C^{++} programs
- Trace kernel activity under application workloads

These utilities are equally useful to both the novice programmer who is tracking events in an executing program, and to the sophisticated programmer who is tracking the interaction among multiple threads and the kernel in a multithreaded program.

- "Defining Types of Users" on page 2
- "How TNF Works" on page 2
- "Kernel Tracing" on page 19
- "Advanced Topics" on page 31

Read the first part of this chapter for an overview of the TNF utilities. Detailed information follows the overview. Novice programmers are encouraged to stop reading when they've learned enough to accomplish their tasks. Sophisticated programmers should read the entire chapter.

Defining Types of Users

Aside from the differences between novice and experienced programmers, users can be defined by the way they use the TNF utilities.

Using Existing Probe Points

If you use the probe points that are shipped with your program or library, concentrate your reading on the `prex` and `tnfdump` sections. You can skip the instructions for inserting probes.

Debugging a Program

If you are debugging a program, read as far into the chapter as is useful for you. How much you read depends upon how complicated your tracing goals are.

Inserting Probe Points in a Library

If you are inserting probe points in a library that will be shipped, all the sections in this chapter are probably helpful. Also, be sure to give your customers information about your probe points.

Tracing Kernel Execution

A small number of TNF probes in the kernel record system calls, thread state transitions, page faults, swapping, and I/O. For the most part, you use the same procedures to trace the kernel as you use to trace a user-level process.

How TNF Works

The TNF utilities consist of the `libtnfprobe` library and the `prex(1)` and `tnfdump(1)` commands:

- `libtnfprobe`—A library that is linked to your program and generates trace records to the binary TNF trace file.

- prex—A utility that manipulates the probe points and the kinds of data they gather.
- tnfdump—A utility that converts the binary TNF trace file to an ASCII file.

You insert commands in your code at points *(probe points)* where you want to collect data. You start running your executable object with prex, which lets you manipulate the probe points. The information collected is written to a trace file in a format called Trace Normal Form (TNF). tnfdump converts the binary TNF trace file to an ASCII file.

You can insert probe points anywhere in C or C^{++} code, including .init() sections, .fini() sections, multithreaded code, shared objects, and shared objects opened by dlopen(3X).

You can also connect functions called probe functions to these probe points. Probe functions perform actions based on the information collected. (Currently, the only available probe function is a debug function that writes output to stderr.)

In a typical TNF session, you do the following:

1. Insert probe points in your programs. (Your source code might already have probe points built into it.)
2. Start your program with prex (this automatically preloads libtnfprobe). (Alternatively, you can link with libtnfprobe when you start your program. If you do this, you can attach prex to the running program. See "Attaching prex to a Running Process" on page 4.)
3. Manipulate the probe points through prex to gather the information that you want from the program.
4. Convert the binary information trace file produced by the program to an ASCII file with tnfdump. (See "Converting the Binary File to Readable Format" on page 18.)
5. Examine the information in the ASCII file.

Inserting Probe Points

The details of inserting probe points are covered fully in "Advanced Topics" on page 31.

"A Sample C Program" on page 14 shows the design and placement of some probe points. "A Sample prex Session" on page 15 and "Reading the Trace File" on page 18 show how probe points are manipulated and what information they provide.

Starting prex

You can either load your program with prex or attach prex to an already running process. The following two sections explain both procedures.

Starting Your Program With prex

When you start your program with `prex`, it preloads the `libtnfprobe` library into the target program, so the program does not have to be explicitly linked with `libtnfprobe`.

`prex` starts your program running, then stops the program before any user code is executed. Execution is stopped even before the `.init()` sections are executed, so you can place probe points in the `.init()` sections to trace initialization code.

In the simplest case, with an executable named `a.out` that takes no arguments and that contains probe points, the following command can be used:

```
$ prex a.out
```

Note - (`prex` uses the `$PATH` environment variable to find the executable.)

To specify options, use:

```
prex [-o outfilename] [-s kbytes_size]
   [-l sharedobjs] cmd [cmdargs...]
```

The `–o`, `–s`, and `–l` options are explained in "Command Line Options for prex" on page 5.

Attaching `prex` to a Running Process

If the program named `a.out` is running and the `ps(1)` command shows that `a.out` has a process ID (PID) of 2374, then the following command attaches `prex` to `a.out`:

```
$ prex -p 2374
```

If you are attaching `prex` to your already-running program, libtnfprobe must be linked to your program first. If you have not linked with libtnfprobe, `prex` exits with the error message that libtnfprobe is not linked to your program.

Linking With libtnfprobe

Linking with the library increases the size of your program by about 33 kilobytes (25 kilobytes shareable) and by the size of the trace file that is allocated (controllable through a command line option to `prex`). If you are never going to attach to a running program using `prex`, don't link with libtnfprobe.

However, if a program is not easy to restart and is a long-running program that you might someday want to gather information about, then *do* link with libtnfprobe. A window server is a good example of this—not the sort of program you want to stop and restart often—so you probably want to link with libtnfprobe before you start it running.

You can link with libtnfprobe in one of two ways:

If you are compiling your program, include –`ltnfprobe` on the `cc` line (if using –`lthread`, always put –`ltnfprobe` before –`lthread`).

```
$ cc -ltnfprobe -lthread -o cookie cookie.c
```

If your program is already compiled or you don't want to build your program with an explicit dependency on libtnfprobe, use the following command:

```
$ LD_PRELOAD=libtnfprobe.so.1 executable_object_name
```

See the *Linker and Libraries Guide* for an explanation of `LD_PRELOAD`.

Halting and Continuing `prex`

After you use `prex` to load the program, or attach `prex` to a running program, use Control-c to stop your program and pass control to `prex`.

When you see the `prex` prompt, `prex` is running and your program is halted.

```
prex>
```

Enter `prex` commands to control the probe points. For example, you can list probe points, turn on tracing, and debug your program.

To continue running your program, enter the `continue` command.

```
prex> continue
```

To stop your program and return to `prex`, press Control-c.

```
$ ^c
prex>
```

Table 1–1 lists the standard `prex` command line options.

Command Line Options for `prex`

TABLE 1–1 `prex` Command Line Options

Option	Definition
–o*trace_file*	The trace file contains information gathered by the `prex` command. The *trace_file* location is assumed to be relative to the directory from which you started `prex`. When no *trace_file* is specified, the default location is `$TMPDIR/trace-pid` where *pid* is the process ID of your program. If *$*`TMPDIR` is not set, `/tmp` is used. When a program is traced, the trace file that is set at the start (the default or the –o *name*) is used for the life of the program.
–l *libraries*	The *libraries* argument contains the names of libraries to preload for your programs. It follows the `LD_PRELOAD` (see ld(1)) rules on how libraries should be specified and where they are found. Enclose the space-separated list of library names in double quotes. When `prex` loads the program, the default library is libtnfprobe.so.1. You cannot use this option when attaching `prex` to an already-running process.
–s *size*	*size* is the size of the trace file in kilobytes (2^{10} bytes). The default size of a trace file is four megabytes. The minimum size that can be specified is 128 kilobytes.

Running `prex`

After you have `prex` connected to your program, use `prex` commands to change the parameters of the data to be collected. You can use the `prex list` command to see all the probes in your program. As your program runs, each time a probe point is executed, information is recorded in the trace file. You look at the collected information with `tnfdump( )`, described in "Reading the `tnfdump` File" on page 18.

You can attach and detach `prex` from your program repeatedly, creating multiple sessions. For example, in your first session you could start the target with `prex`, set up the probes, and type `quit resume.`

This will start `prex`, define the information you want to collect, quit `prex`, and start collecting information by resuming execution of your program. The commands to do this are shown in Table 1–2.

In a following session, you could attach `prex` to the same program, reconfigure the probes, and again type `continue`. You can do this any number of times.

Note - All traces are written to the trace file name specified in the first session —this file name cannot be changed.

TABLE 1–2 Basic prex Commands

Command	Result
% **prex a.out**	Attach prex to your program and start prex.
prex> **enable $all**	Enable all the probes.
prex> **quit resume**	Quit prex and resume execution of program.

Components of the prex Command

The programmer who has inserted the probe points in the source code has assigned various attributes and (optional) values to each probe point. For example, each probe point has a “name” attribute whose value is the name the programmer chose for it.

Look at Table 1–3 for a list of the predefined attributes and the meanings of their values. The prex command lets you select a group of probe points by matching on their attributes or values. You can then trace or debug the selected probe points.

Table 1–5 lists the prex commands and their actions.

In code Code Example 1–1, the TNF_PROBE macro defines the probe point named work_start. This probe point has two arguments whose values will be logged—state and message. Each time the probe named work_start is encountered, a time stamp and the values of the variables state and message are logged to the trace file.

CODE EXAMPLE 1–1 prex Attributes and Values

```
#include <tnf/probe.h>

int
work(int state, char *message)
{
   TNF_PROBE_2(work_start, "work_module work"
              "sunw%debug in function work",
              tnf_long, int_input, state,
              tnf_string, string_input, message);
  ...
  ...
}
```

For more information on TNF_PROBE macros, including this example, see “Using the TNF_PROBE Macros” on page 31.

Table 1–3 lists the predefined attributes.

Attributes

Attributes are characteristics by which you identify probe points in your code. Table 1–3 lists the predefined attributes. You can add more attributes with the TNF_PROBE macros. See “Using the TNF_PROBE Macros” on page 31.

TABLE 1–3 Predefined Attributes

Attribute	Characteristics	Value
enable	A probe point performs the action that it is set up for only if it is enabled. For example, even when the tracing state is set, tracing occurs only if the probe point has been enabled.	OFF (default)
file	The name of the file containing the probe point.	work.c
funcs	Shows the list of probe functions connected to this probe. Currently, only the debug function is available.	<no value> (default)
keys	The groups to which the probe point belongs. If any key in a probe point is enabled, then that probe point is enabled.	work_module work
line	The line number in the code on which the probe point occurs.	10
name	The name of the probe point.	work_start
object	The name of the shared object or executable that the probe is in. Useful for selecting all the probes in a particular module.	work
slots	The names of the probe point arguments (arg_name_n), see page 41.	int_input string_input
trace	When tracing for a probe point is on, a line is written to the trace file each time the probe point is executed in your program.	ON (default)

Grammar for Matching Probes

Select probes based on their attributes and values. The selection specification is called a *selector_list*. Each attribute or value can be described as one of the following:

- Identifier—a sequence of characters using letters, numbers, and the symbols _\. and % (underscore, backslash, period, and percent). Identifiers cannot begin with a number.
- Quoted string—a sequence of characters in single quotes, taken literally; useful when the string being matched is a reserved word. See Table 1–4 for a list of reserved words.
- Regular expressions—a sequence of characters enclosed in slashes (/ /); expanded for a match following the rules for `ed(1)`; when slashes occur in the regular expression, as in a path name, escape the slashes with backslashes: `/\/tmp\/`*filename*.

A *selector_list* consists of one or more *selector*= selector items. If the initial *selector*= is not provided, it defaults to `keys=`. For example, the enable command is specified as :

`enable` *selector_list*

An example of this command is:

```
enable name=/first/ file='sampleZ.c'
```

This enables all probe points with either the `name` attribute containing the value `first` (a regular expression match), or the `file` attribute equal to the value `sampleZ.c` (Note that the trace is a disjunction and not a conjunction.)

Use $*set_name* as a shorthand alias for a *selector_list* specification. In the following example, the set_name is `myprobes`.

```
create $myprobes name=/first/ file='sampleZ.c'
enable $myprobes
```

This does the same thing as the previous example. The set_name follows the identifier naming rules. The `$all` default set selects all probes in the program.

Reserved Words

Table 1–4 lists the reserved words. Enclose them in single quotes if you use them to select attributes or values.

TABLE 1–4 Reserved Words

`add`	`alloc`	`buffer`
`clear`	`connect`	`continue`
`create`	`dealloc`	`delete`
`disable`	`enable`	`fcns`
`filter`	`help`	`history`
`kill`	`ktrace`	`list`
`off`	`on`	`pfilter`
`probes`	`quit`	`resume`
`sets`	`source`	`suspend`
`trace`	`untrace`	`values`

For example:

```
enable 'trace'='on'
```

means enable all probes whose trace attribute has the value of ON. Both `trace` and `on` are reserved words, so they both have to be enclosed in single quotation marks.

TABLE 1–5 `prex` Commands

Command	Action
`clear` $*set_name* `clear` *selector_list*	Disconnect connected probe functions.
`connect &debug` $*set_name* `connect &debug` *selector_list*	Connect the debug function to the probe points. This does not enable the probe points. The debug function sends its output to `stderr` and not to the trace file.
`continue`	Resume execution of your program, leaving `prex` attached.
`create` $*set_name selector_list*	Create a set with probe points matching *selector_list*. Also creates an alias, $*set_name*, for the *selector_list*.

TABLE 1–5 `prex` Commands *(continued)*

Command	Action
`enable` $*set_name* `enable` *selector_list* `disable` $*set_name* `disable` *selector_list*	Control whether the probe points perform the action they are set up for. Probe points are disabled by default; `prex` does not turn on tracing. The cheapest way (in terms of execution time at the probe point) to stop a probe point from tracing is to use the `disable` command. The `enable` and `disable` commands are a master switch. If a probe point is not enabled, even if the probe point is connected to the `debug()` function and `trace` is on, no information is sent to `stderr` or to the trace file.
`help`	List all available `prex` commands.
`list` *attributes* `probes` *selector_list* `list` *attributes* `probes` $*set_name*	List whether specified probe points are enabled or disabled, whether they have tracing on or off, and what the connected probe function is. Attributes are selectors as described in “Attributes ” on page 8. For example, list name file probes $all lists only *name* and *file* values for matching probe points, while the command **`list probes $all`** lists all default attributes and their values (*name*, *enable*, *trace*, *file*, *line*, and *funcs*).
`list fcns`	List the defined functions (currently, only *&debug* is defined).
`list history`	List the control command history. The history of the commands used with `connect`, `clear`, `trace`, `untrace`, `enable`, and `disable` are executed whenever a new shared object is brought into your program through `dlopen()`. See “`dlopen()` and `dlclose()` and History” on page 39.
`list sets`	List the defined sets.
`list values` *attributes*	List the unique values associated with the specified attributes. For example, **`list values keys`** lists all the unique keys in the program.
`source` *filename*	Source a file of `prex` commands. *filename* is a quoted string.

TABLE 1–5 `prex` Commands *(continued)*

Command	Action
`trace` $*set_name* >`trace` *selector_list* `untrace` $*set_name* `untrace` *selector_list*	Control the tracing action of the probe points. `trace` and `untrace` determine whether a probe point generates a trace record when it is executed. Neither `trace` nor `untrace` enables the probe points. The default mode is that tracing is on. The `untrace` command is useful when you are interested in getting only debug output. If you are using this, your probe should be enabled with tracing off and debug on. Tracing disturbs your program less than the debug function does because `trace` writes to an `mmap`'d file while `debug` writes to `stderr`.
`quit`	Quit `prex`; if your program was loaded with `prex`, it will be killed; if your program was attached to `prex`, it will be resumed.
`quit kill`	Quit `prex` and kill your program.
`quit resume`	Quit `prex` and resume execution of your program.
`quit suspend`	Quit `prex` and leave your program suspended.

Tracing, Enabling, and Connecting

To trace at a probe point, the probe point must have trace on and it must be enabled. To debug at a probe point, the probe point must be connected to the debug function and must be enabled.

Table 1–6 shows which combinations of tracing, enabling, and connecting result in which actions.

TABLE 1–6 Tracing, Enabling, and Connecting Actions

Enabled or Disabled	Tracing State (On/Off)	Debug State (Connected/ Cleared)	Results In
Enabled	On	Connected	Tracing and Debugging
Enabled	On	Cleared	Tracing only

TABLE 1–6 Tracing, Enabling, and Connecting Actions *(continued)*

Enabled or Disabled	Tracing State (On/Off)	Debug State (Connected/ Cleared)	Results In
Enabled	Off	Connected	Debugging only
Enabled	Off	Cleared	Nothing
Disabled	On	Connected	Nothing
Disabled	On	Cleared	Nothing
Disabled	Off	Connected	Nothing
Disabled	Off	Cleared	Nothing

Controlling `prex` From a Script

Enter commands to `prex` either from the `prex` command line or from a file containing `prex` commands.

When you start `prex`, it searches for a file named `.prexrc` first in `$HOME/` and next in the directory from which you started `prex`. Commands are read from all files that are found. Therefore it is possible to have a `.prexrc` file in the current directory that overrides defaults that are set up by the `.prexrc` file in the home directory.

After reading any `.prexrc` files that are found, input is expected from the `prex` command line. To set up an experiment entirely using `.prexrc` files, the last statement in it can be `quit resume`, which quits `prex` but lets your program resume.

Also, when `prex` is running you can use the `source` *filename* command to specify a file from which `prex` reads commands. This file can have any name you like.

As with commands entered from the `prex` command line, all commands in the script should be in ASCII. The following rules apply:

- Terminate each command with the newline character.
- Continue a command onto the next line by ending the previous line with a backslash (`\`) character.
- Separate tokens by white space (one or more spaces or tabs).
- Start comments with a hash mark (#).

  While the command language is the same for `prex` commands entered from the command line and for commands from a script, commands that return output (such as `list probes...`) make little sense in a script because the output goes to `stdout`.

A Sample C Program

This program, `cookie.c`, asks for a cookie. The response must be entered in uppercase, or it is incorrect. You can also find the prime factor of a number. In the process of doing these two things, you can see how the trace function works.

To compile this program and get an executable called `cookie()`, use

```
$ cc -o cookie cookie.c
```

Five probe points are defined (and highlighted) in this program. They are named `start` (line 17), `inloop` (line 33), `factor_start` (line 60), `found_a_factor` (line 65), and `factor_end` (line 72). More information about these probe points is gathered and explained in "A Sample `prex` Session" on page 15.

CODE EXAMPLE 1–2 Code for `cookie.c()`

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <tnf/probe.h>

#define MAX_RESPONSE_SIZE 256

static void find_factor(int n);

int
main(int argc, char **argv)
{
    boolean_t shouldexit = B_FALSE;
    int  sum = 0, max_loop = 5;
    int  i;

    TNF_PROBE_0(start, "cookie main",
         "sunw%debug starting main");

    while (!shouldexit) {
       char response[MAX_RESPONSE_SIZE];
       int factor_input;

       (void) printf("give me a COOKIE! ");
       (void) scanf(‘‘%s’’, response);

       if (!strcmp(response, "COOKIE")) {
            (void) printf("thanks!\n");
            shouldexit = B_TRUE;
       }

       else if (!strcmp(response, "loop")) {
            for (i = 0; i < max_loop; i++) {
               TNF_PROBE_2(inloop, "cookie main
                            loop","sunw%debug in the loop",
```

```
                            tnf_long, loop_count,  i,
                            tnf_long, total_iterations,  sum);
                sum++;
            }
            max_loop += 2 ;
        }

      else if (!strcmp(response, "factor")) {
            (void) printf("number you want factored? ");
            (void) scanf("%d", &factor_input);
            find_factor(factor_input);
      }

      else {
            (void) printf("not a %s, ", response);
      }
    }
    return 0;

}   /* end main */
static void
find_factor(int n)
{
    int i;

    (void) printf("\tfactors of %d = ", n);
    TNF_PROBE_1(factor_start, "factor", "",
        tnf_long, input_number, n);

    for (i=2; i <= n; i++) {
        while (n % i == 0) {
           TNF_PROBE_2(found_a_factor, "cookie find_factor",
               "", tnf_long, searching_for, n,
               tnf_long, factor, i);
           (void) printf("%d ", i);
           n /= i;
        }
    }
    TNF_PROBE_0(factor_end, "factor", "");
    (void) printf("\n");
}
```

A Sample `prex` Session

This sample of a `prex` session is designed to show the different capabilities of `prex`. The data collected when cookie is run is shown in “Reading the `tnfdump` File” on page 18.

```
% prex cookie     /* prex is loading the executable cookie. */
Target process stopped
Type "continue" to resume the target, "help" for help ...
prex> list sets
$all      'keys'=/.*/    /* One set is defined---$all
                            (which is all the probes). */
prex> list fcns
&debug    tnf_probe_debug      /* The debug function is
                                  the only one available. */
prex> list probes $all
name=inloop enable=off trace=on file=cookie.c line=35 funcs=<no value>
```

```
name=factor_end enable=off trace=on file=cookie.c line=72 funcs=<no value>
name=factor_start enable=off trace=on file=cookie.c line=61 funcs=<no value>
name=found_a_factor enable=off trace=on file=cookie.c line=67 funcs=<no value>
name=start enable=off trace=on file=cookie.c line=17 funcs=<no value>
prex>/*  The line number shows the end of each of the five probes. */
prex> create $factor /factor/    /* Create a new set that
                            matches any probe whose "keys" */
prex>       /* attribute contains the string "factor". */
prex> list sets
$all      'keys'=/.*/
$factor   'keys'=/factor/
/* A new set named "factor"
                                   is created and now shows
                                   up in */
prex>
  /* the list of sets. */
prex> list probes $factor
/* This line tells you which probes matched the set
$factor.*/
name=factor_end enable=off trace=on file=cookie.c line=72 funcs=<no value>
name=factor_start enable=off trace=on file=cookie.c line=61 funcs=<no value>
name=found_a_factor enable=off trace=on file=cookie.c line=67 funcs=<no value>
prex> list probes $all          /* Check to see if any
                                 probes are enabled. */
name=inloop enable=off trace=on file=cookie.c line=35 funcs=<no value>
name=factor_end enable=off trace=on file=cookie.c line=72 funcs=<no value>
name=factor_start enable=off trace=on file=cookie.c line=61 funcs=<no value>
name=found_a_factor enable=off trace=on file=cookie.c line=67 funcs=<no value>
name=start enable=off trace=on file=cookie.c line=17 funcs=<no value>
prex>                  /* None are enabled,
                      but all have trace on. */

prex> enable $all             /* Enable all the probes.*/
prex> list probes $all        /* Check again to see if any
                              probes are enabled. */
name=inloop enable=on trace=on file=cookie.c line=35 funcs=<no value>
name=factor_end enable=on trace=on file=cookie.c line=72 funcs=<no value>
name=factor_start enable=on trace=on file=cookie.c line=61 funcs=<no value>
name=found_a_factor enable=on trace=on file=cookie.c line=67 funcs=<no value>
name=start enable=on trace=on file=cookie.c line=17 funcs=<no value
prex> list values name              /* Find out what the probe names are. */
name =
    factor_end
    factor_start
    found_a_factor
    inloop
    start
prex> list values /.*/          /* List all predefined
                              attributes with their values. */
enable =              /* Only unique attributes are listed.*/
    on
file =
    cookie.c
funcs =

keys =
    cookie
    factor
    find_factor
    loop
```

```
    main
line =
    17
    35
    61
    67
    72
name =
    factor_end
    factor_start
    found_a_factor
    inloop
    start
object =
    cookie
slots =
    factor
    input_number
    loop_count
    searching_for
    total_iterations
sunw%debug =           /* The user-defined macro, sunw%debug,
                       is also listed. */
    in     /* This macro is defined in line 17 of cookie.c. */
    loop
    main
    starting
    the
trace =
    on
prex> list values object
object =
    cookie
prex> connect &debug name=inloop
prex> list /.*/ probes $all         /* List all the
                           information about all the probes */
enable=on trace=on object=cookie funcs=<no value> name=inloop slots=loop_count
total_iterations keys=cookie main loop file=cookie.c line=35 sunw%debug=in the loop
enable=on trace=on object=cookie funcs=<no value> name=factor_end slots=<no value>
keys=factor file=cookie.c line=72
enable=on trace=on object=cookie funcs=<no value> name=factor_start slots=input_number
keys=factor file=cookie.c line=61
enable=on trace=on object=cookie funcs=<no value> name=found_a_factor slots=searching_for
factor keys=cookie find_factor file=cookie.c line=67
enable=on trace=on object=cookie funcs=<no value> name=start slots=<no value> keys=cookie
main file=cookie.c line=17 sunw%debug=starting main
prex> continue
give me a COOKIE! loop
 /* An example of loop counts*/
probe inloop; sunw%debug ‘‘in the loop’’; loop_count=0; total_iterations=0;
probe inloop; sunw%debug ‘‘in the loop’’; loop_count=1; total_iterations=1;
probe inloop; sunw%debug ‘‘in the loop’’; loop_count=2; total_iterations=2;
probe inloop; sunw%debug ‘‘in the loop’’; loop_count=3; total_iterations=3;
probe inloop; sunw%debug ‘‘in the loop’’; loop_count=4; total_iterations=4;
give me a COOKIE! factor
number you want factored? 25
 factors of 25 = 5 5
give me a COOKIE! factor
number you want factored? 43645729
 factors of 43645729 = 43645729
```

```
give me a COOKIE! ^C Target process stopped
Type ‘‘continue’’ to resume the target, ‘‘help’’ for help...
prex> continue
give me a COOKIE! biscuit
not a biscuit, give me a COOKIE!  cookie
not a cookie, give me a COOKIE! COOKIE
thanks!
prex: target process finished
```

Reading the Trace File

The binary trace file you create with `prex` contains information determined by the `prex` commands you chose (see “Running `prex`” on page 6).

By default, this file is created in `/$TMPDIR/trace-`*pid*, where *pid* is the process ID of the target program. If `$TMPDIR` is not set, the file is created in `/tmp/trace-`*pid*. You can override these default locations with the `-o` option to the `prex` command (see “Command Line Options for `prex` ” on page 5 for a full explanation).

When the trace file has been filled, newer events overwrite the older events. The default size of a trace file is four megabytes. This can be changed with the -s option for `prex`.

Note - After a program is tracing to a file, there is no way to clear the trace file or to give it a different name for the lifetime of that program. If you disconnect from the target and attach later with a different trace file name, the newer name is ignored.

Converting the Binary File to Readable Format

To convert the binary trace file to an ASCII file, use the `tnfdump` command and the name of the binary trace file. Because `tnfdump` output goes to `stdout` by default, you probably want to redirect it into a file.

$ **`tnfdump`** *filename* > *newfile*

The– `-r` option to `tnfdump` provides detailed (raw) TNF output. Reading this output requires an understanding of TNF that is beyond the scope of this chapter.

Reading the `tnfdump` File

The following table shows output from the `prex cookie` command described in “A Sample `prex` Session” on page 15. The output of the `tnfdump` file is very wide—open a very wide window to display it:

```
probe  tnf_name: ‘‘start’’ tnf_string: ‘‘keys cookie main;file cookie.c;line 17;sunw%debug
starting main’’
probe  tnf_name: ‘‘factor_start’’ tnf_string: ‘‘keys factor;file cookie.c;line 61;’’
probe  tnf_name: ‘‘found_a_factor’’ tnf_string: ‘‘keys cookie find_factor;file cookie.c;line
67;’’
```

```
probe  tnf_name: ''factor_end'' tnf_string: ''keys factor;file cookie.c;line 72;''
---------------- ---------------- ----- ----- ---------- --- ------------------------
------------------------
    Elapsed (ms)  Delta (ms)     PID  LWPID  TID  CPU Probe Name       Data / Description . . .
---------------- ---------------- ----- ----- ---------- --- ------------------------
------------------------
  0.000000 0.000000 5354 1 0 - start
 4551.625000 4551.625000 5354 1 0 - factor_start input_number: 25
 4571.278000 19.653000 5354 1 0 - found_a_factor searching_for: 25 factor: 5
 4571.543000 0.265000 5354 1 0 - found_a_factor searching_for: 5 factor: 5
 4571.732000 0.189000 5354 1 0 - factor_end
  23151.434000 18579.702000 5354 1 0 - factor_start input_number: 101247
  23151.509000 0.075000 5354 1 0 - found_a_factor searching_for: 101247 factor: 3
  23228.090000 76.581000 5354 1 0 - found_a_factor searching_for: 33749 factor: 33749
  23228.250000 0.160000 5354 1 0 - factor_end
  89041.868000 65813.618000 5354 1 0 - factor_start input_number: -1690908149
  89041.920000 0.052000 5354 1 0 - factor_end
 108271.852000 19229.932000 5354 1 0 - factor_start input_number: 43645729
 208857.756000  100585.904000 5354 1 0 - found_a_factor searching_for: 43645729 factor:
43645729
 208857.960000 0.204000 5354 1 0 - factor_end
 334511.548000  125653.588000 5354 1 0 - factor_start input_number: 12
 334511.618000 0.070000 5354 1 0 - found_a_factor searching_for: 12 factor: 2
 334511.689000 0.071000 5354 1 0 - found_a_factor searching_for: 6 factor: 2
 334511.750000 0.061000 5354 1 0 - found_a_factor searching_for: 3 factor: 3
 334511.808000 0.058000 5354 1 0 - factor_end
```

Looking at the `tnfdump` display, you can see how long it takes to find a factor by subtracting the `factor_start` time from the `factor_end` time. Factoring 43645729 took 208857.960000 - 108271.852000, or 100586.11 milliseconds. Factoring 12 took 334511.808000 - 334511.548000, or .260000 milliseconds.

Note - Results are reported with nanosecond precision for all hardware platforms. Accuracy, however, depends on the hardware platform used.

Kernel Tracing

Starting with the Solaris 2.5 release, the SunOS kernel has a small number of TNF probes built into it. The probes record kernel events such as system calls, thread state transitions, page faults, swapping, and I/O. You can use these probes to obtain detailed traces of kernel activity under your application workloads. The probes have negligible impact on the performance of the running kernel when the probes are disabled.

The method for tracing the kernel is similar to tracing a user-level process, although there are some differences. To understand the following discussion, you should have read the sections "Running `prex`" on page 6 and "Reading the Trace File" on page 18.You need super-user privileges to trace the kernel.

Use the prex utility to control kernel probes. The standard prex commands to list and manipulate probes are available to you, along with commands to set up and manage kernel tracing.

Kernel probes write trace records into a kernel trace buffer. You must copy the buffer into a TNF file for post-processing; use the tnfxtract utility for this.

You use the tnfdump utility to examine a kernel trace file. This is exactly the same as examining a user-level trace file.

The steps you typically follow to take a kernel trace are:

1. Become superuser (su).
2. Allocate a kernel trace buffer of the desired size (prex).
3. Select the probes you want to trace and enable (prex).
4. Turn kernel tracing on (prex).
5. Run your application.
6. Turn kernel tracing off (prex).
7. Extract the kernel trace buffer (tnfxtract).
8. Disable all probes (prex).
9. Deallocate the kernel trace buffer (prex).
10. Examine the trace file (tnfdump).

    A convenient way to follow these steps is to use two shell windows; run an interactive prex session in one, and run your application and tnfxtract in the other.You might find it simpler to use the ktrace shell script shown in " Shell Script for Kernel Tracing" on page 28.

Controlling Kernel Tracing (prex)

Start prex on the kernel with the –k flag (make sure you are root). After prex successfully attaches to the kernel, it prompts you for commands.

```
# prex -k
prex>
```

Note - Only one prex session can be attached to the kernel at any given time.

Buffer Allocation

The first step in taking a kernel trace is to allocate a kernel trace buffer. The trace buffer is circular, which means that newer data overwrites older data after the buffer fills up.

The default size for the buffer is 384 kilobytes, but you can override this when you allocate the buffer. When deciding on the buffer size, keep in mind that the buffer occupies both physical memory and kernel virtual memory, so it will impact the system you are tracing—the more RAM you have, the less the impact of measurement on your experiment, and the bigger the buffer you can allocate.

Use the buffer command in `prex` to allocate the buffer. For example, you could allocate a 512 kilobyte buffer as follows:

```
prex> buffer         # do we have a buffer?
No trace buffer allocated
prex> buffer alloc 512k# allocate 512KB buffer
Buffer of size 524288 bytes allocated
prex>
```

Note - The minimum buffer size is 128 kilobytes; `prex` allocates a buffer of this size if you request anything smaller.

Selecting and Enabling Probes

Use the standard `prex` list command to list kernel probes and their attributes. For example, to list the `name` and `keys` attributes of the probe named `pagein`:

```
prex> list name=pagein keys
name=pagein keys=vm pageio io
```

To list the `name` and `keys` attributes of all probes in the `io` group:

```
prex> list name keys probes io
name=biodone keys=io blockio
name=physio_start keys=io rawio
name=pagein keys=vm pageio io
name=pageout keys=vm pageio io
name=physio_end keys=io rawio
name=strategy keys=io blockio
prex>
```

The next step is to trace and enable the probes you need. For example, to select all probes whose keys specify `thread`, `vm`, and `io`, and prepare them for tracing:

```
prex> trace thread vm io     # attach trace function to probes
prex> enable thread vm io    # enable probes
prex>
```

Note - Unlike user-level probes, kernel probes are not automatically ‘traced’ (in `prex` terminology) when `prex` attaches to the kernel. You must explicitly issue the trace and enable commands as shown above.

At this point the relevant probes are ready for tracing, but kernel tracing has not been globally enabled so no trace records are being written.

Process Filtering

If you want to trace all system activity (for example, on a busy server), you can proceed to “Enabling and Disabling Tracing” on page 22. This is the most common use of kernel tracing, and usually the most informative.

However, you also have the option of restricting trace data generation to selected processes; this can reduce the amount of data you need to collect and analyze.

`prex` uses two abstractions to allow you to do process filtering.

- The process filter *set* is a list of process identifiers (PIDs) for those processes that you want enabled for tracing; no trace data will be written by (threads belonging to) processes that are not in the filter set. The default filter set is empty.
- The process filter *mode* is a global flag that selects whether process filtering is enabled or disabled in the kernel. The default is that process filtering is disabled, which means that all processes (and threads) write trace records. When it is enabled, only (threads belonging to) the processes in the filter set write trace records.

Use the `prex pfilter` command to control process filtering.

```
prex> pfilter                # are we filtering?
Process filtering is off
Process filter set is empty.
prex> pfilter add 408       # add PID 408 to filter set
prex> pfilter
Process filtering is off
Process filter set is {408}
prex> pfilter on            # enable process filter mode
prex>
```

Note - System threads (such as interrupt threads) are treated as belonging to process 0.

Enabling and Disabling Tracing

The final step in starting up tracing is to globally enable kernel tracing. When you do this, the probes you have enabled write records into the kernel trace buffer as they are encountered.

```
prex> ktrace          # check tracing status
Tracing is off
prex> ktrace on       # enable kernel tracing
prex>
```

To trace the kernel behavior of your application (perhaps with user-level tracing enabled as well), start it running at this time. When your application finishes, or when you decide you have enough trace data to sample its behavior, globally disable kernel tracing.

```
prex> ktrace off        # disable kernel tracing
prex>
```

The trace buffer is still present in the kernel. Copy it out into a TNF file using `tnfxtract`. The procedure that you follow is explained in the subsection " Extracting Kernel Trace Data (`tnfxtract`)" on page 23.

Resetting Kernel Tracing

After you have copied the trace buffer into a TNF file, reset kernel tracing by disabling all probes and deallocating the buffer. This is important because it brings the performance of the kernel back to where it was before you started tracing.

Use the standard `prex` commands to disable and untrace all probes.

```
prex> disable $all      # disable all probes
prex> untrace $all      # untrace all probes
prex>
```

Finally, use the `prex buffer` command to deallocate the trace buffer.

Caution - Do *not* deallocate the trace buffer until you have copied it out into a trace file. Otherwise, you will lose the trace data you collected for your experiment.

```
prex> buffer dealloc    # deallocate buffer
buffer deallocated
prex>
```

You can now quit `prex` and examine the trace data that you have collected.

```
prex> quit
#
```

Extracting Kernel Trace Data (`tnfxtract`)

Use the `tnfxtract` utility to make a copy (or snapshot) of the active kernel trace buffer into an external TNF trace file. You typically run the utility after tracing has been disabled, although you can also run it concurrently with tracing. The utility ensures that it reads and writes only consistent TNF data.

Specify a file name that `tnfxtract` can use to hold the extracted kernel trace data. This file will be overwritten and truncated to the size of the kernel trace buffer. For example, to extract it into a temporary file named `ktrace.tnf`:

```
# tnfxtract /tmp/ktrace.tnf
# ls -l /tmp/ktrace.tnf
-rw------- 1 root other 524288 Aug 15 16:00 /tmp/ktrace.tnf
#
```

You might find it convenient to change the permissions on the trace file so that it is world-readable. This makes it easier for you to run analysis tools over the file when you are not superuser.

Note - You can also use `tnfxtract` on a kernel crash dump; see `tnfxtract(1)` for details.

Examining Kernel Trace Data (`tnfdump`)

Use `tnfdump` to get a time-ordered ASCII dump of a kernel trace file on `stdout` (exactly as in user-level tracing). Because the trace output might be very large, you typically run it through a pager (such as `more`) or redirect it to an output file.

```
# tnfdump /tmp/ktrace.tnf | more
```

Note - `tnfdump` accepts multiple TNF files as input; it produces a time-sorted output of all event records in its input files, so you can use it to combine multiple user-level trace files along with the kernel trace file.

Available Kernel Probes (`tnf_probes`)

The SunOS kernel probes supply information approximately at the level of statistics-based Solaris performance monitoring tools such as `vmstat`, `iostat`, and `sar`. However, they provide more detail — each probe records a high-resolution timestamp, a thread (LWP, thread, and process identifiers), a CPU, and individual probes identify the system resources (such as disks, files, CPUs, and so on) associated with an event.

You can use kernel tracing, along with user-level tracing, to correlate the events recorded by your application or library probes with the events recorded by the kernel probes. Thus you can get a detailed view of how your code uses kernel services, and how the demands your application places on system resources interact with the demands placed by other clients.

The text below summarizes the SunOS kernel probes; see the `tnf_probes(4)` manual page for detailed information on the event record fields.

Thread Probes

The thread_create probe traces kernel thread creation. It records the process identifier, the kernel thread identifier, and the kernel address of the start routine of the new thread.

The thread_exit probe records the termination of the current thread.

Probes that trace thread-state transitions are called microstate probes. They record a thread state and, optionally, a thread identifier. If a thread identifier is provided, the state change applies to the indicated thread. Otherwise, it applies to the writing thread.

Thread state values use the microstate constants defined in <sys/msacct.h>. The states are recorded as shown in Table 1–7.

TABLE 1–7 Thread Microstate Constants

State	Explanation
user	Running in user mode
system	Running in system mode
tfault	Initial state for user text fault
dfault	Initial state for user data fault
trap	Initial state for other trap
user_lock	Asleep, waiting for user-mode lock
sleep	Asleep for any other reason
wait_cpu	Waiting for a CPU (runnable)
stopped	Stopped (/proc, jobcontrol, lwp_stop)

Note - The kernel does not trace transitions between the system and user states that are implied by system calls. This is to reduce trace volume. You need to enable the system call probes to get this information; for convenience, they are automatically enabled for you when you enable probes with the thread key.

System Call Probes

System call entry and exit probes identify the system operations explicitly requested by user code.

The syscall_start probe marks the start of a system call, and records the system call number. The probe does not capture system call arguments, because this is fairly expensive. (Some of this information can be captured by interposing on the C library entry points at the user level.) The syscall_start probe also implicitly marks the current thread as entering the system state.

The syscall_end probe marks the end of a system call, and records the two return values of the call, as well as the errno value. The syscall_end probe also implicitly marks the current thread as entering the user state.

Note - System call implementation at this level can change from release to release. Do not rely on a consistent mapping of system calls to numbers.

VM Probes

The virtual memory subsystem (VM) probes provide information on page faults, page I/O, the page daemon, and the swapper.

Page Faults

Page fault probes relate virtual addresses with fault types and with files (vnodes).

The address_fault probe traces address-space faults; it records the faulting virtual address, the type of fault, and the desired access protection.

The fault type and access type values use the constants defined in <vm/seg_enum.h>. Fault types are invalid page (minor fault), protection fault, or software requests to lock and unlock pages for physical I/O (softlock and softunlock). Access types are read, write, execute and create.

The major_fault probe traces major page faults; it records the vnode and offset (which together identify a file system page) from which the fault should be resolved. This data can be correlated with the immediately preceding address_fault event for the current thread to obtain the faulting virtual address.

The anon_private probe traces copy-on-write faults.

The anon_zero probe traces zero-fill faults.

The page_unmap probe marks the dissociation between a physical page and a file system page (for example, when a page is renamed or destroyed).

Page I/O

The pagein probe traces the initiation of a pagein request; it records the vnode, offset, and size of the pagein.

The pageout probe traces the completion of a pageout request; it records the number of pages paged out, the number of pages freed, and the number of pages reclaimed after the pageout.

Page Daemon

Iterations of the page daemon (page stealer) are traced by two probes: pageout_scan_start and pageout_scan_end. The probes report the number of free pages needed before the scan, the number of pages examined during the scan, and the free page counts before and after the scan. Potentially, more pages are freed when pageout requests queued by the scan are completed.

Swapper

Three probes trace the activity of the swapper.

- The swapout_process probe traces the swapping out of a process address space; it records the process identifier and the total number of pages either freed or queued for output.
- The swapout_lwp probe traces the swapping out of an LWP's stack pages; it records the LWP's identity and the number of pages queued for output.
- The swapin_lwp probe traces the swapping in of an LWP's stack pages; it records the LWP's identity and the number of stack pages brought in.

Local I/O Probes

The strategy probe traces the initiation of local block device I/O by the kernel. It records the device number, logical block number, size, buffer pointer, and buffer flags associated with the transfer. The flag values are the buffer status flags as defined in <sys/buf.h>.

The biodone probe traces the completion of a buffered I/O transfer, that is, calls to the kernel biodone(9f) routine. It records the device number, logical block number, and buffer pointer associated with the transfer.

Physical (raw) I/O is traced by two probes in physio(9f): physio_start and physio_end. These probes record the device number, offset, size, and direction of the I/O transfer.

Other Probes

The `thread_queue` probe traces thread scheduling; it records the thread identifier of the scheduled thread, the CPU associated with the dispatch queue on which it is placed, the thread's dispatch priority, and the current number of runnable threads on the dispatch queue.

Shell Script for Kernel Tracing

CODE EXAMPLE 1–3 ktrace Script

```
#!/bin/sh
#
# ktrace
# Reset and enable kernel tracing
# Run command in background, if specified
# Sleep for duration, if specified
# Wait for command completion, timeout, or keyboard interrupt
# (Note: keyboard interrupt kills command)
# Disable kernel tracing
# Extract kernel trace buffer
# Reset kernel tracing, if specified
#

TMPDIR=${TMPDIR:-/tmp}

output_file=$TMPDIR/ktrace.tnf
buffer_size=512k
duration=
events=
command=
do_reset=y
child=
alarm=

# usage message
usage() {
 echo ‘’‘’
 echo $1
 echo ‘’
Usage: ktrace [-o <output_file>] # default /tmp/ktrace.tnf
  [-s <buffer_size>] # default 512k
  -e <events>  # kernel probes (keys) to enable
  [-r]   # don’t reset kernel tracing
     # default is to reset after command
  <cmd> | -t <seconds>

Eg,
 # ktrace -e ‘thread vm io’ -t 10
 # ktrace -e ‘thread’ -s 256k myapp ...
‘’
 exit 1
```

(continued)

(Continuation)

```
}
# failure message
fail() {
 while [ $# -gt 0 ]
 do
  echo $1
  shift
 done
 echo ‘‘ktrace failed’’
 exit 2
}

# Reset kernel tracing
ktrace_reset() {
 if [ $1 = ‘‘y’’ ]; then
  echo ‘‘Resetting kernel tracing’’
  prex -k >/dev/null 2>&1 <<-EOF
   ktrace off
   untrace /.*/
   disable /.*/
   buffer dealloc
   EOF
  test $? -ne 0 && fail ‘‘Could not reset kernel tracing’’ \
   ‘‘‘su root’ and retry’’
 fi
}

# Enable kernel tracing
ktrace_on() {
 echo ‘‘Enabling kernel tracing’’
 prex -k >/dev/null 2>&1 <<-EOF
  buffer alloc $buffer_size
  trace $events
  enable $events
  ktrace on
  EOF
 test $? -ne 0 && fail ‘‘Could not enable kernel tracing’’ \
  ‘‘Check syntax of ‘-e’ argument’’ \
  ‘‘Check buffer size is not too high’’
}

# Disable kernel tracing
ktrace_off() {
 prex -k >/dev/null 2>&1 <<-EOF
  ktrace off
  EOF
 test $? -ne 0 && fail ‘‘Could not disable kernel tracing’’
 echo ‘‘Kernel tracing disabled’’
}

# Extract kernel trace buffer
ktrace_xtract() {
 echo ‘‘Extracting kernel trace buffer’’
```

(continued)

```
 tnfxtract $output_file || fail ''Could not extract kernel trace
buffer''
 ls -l $output_file
}

# Run command, sleep for duration, or wait for command to complete
run_command() {
 trap 'interrupt' 0 1 2 3 15
 if [ ''$command'' ]; then
  $command &
  child=$!
  echo '''$command' pid is $child''
 fi

 if [ ''$duration'' ]; then
  sleep $duration &
  alarm=$!
  wait $alarm
  # XXX test -z ''$child'' || kill -15 $child
 else
  wait $child
 fi
 trap 0 1 2 3 15
}

# Keyboard interrupt
interrupt() {
 test -z ''$alarm'' || kill -15 $alarm
 test -z ''$child'' || kill -15 $child
}

# Parse options
while getopts o:s:t:e:r opt
do
 case $opt in
  o) output_file=''$OPTARG'';;
  s) buffer_size=''$OPTARG'';;
  t) duration=''$OPTARG'';;
  e) events=''$OPTARG'';;
  r) do_reset=''n'';;
  \?) usage;;
 esac
done

shift 'expr $OPTIND - 1'

# Get command to run
test $# -gt 0 && command=''$*''

# Check and normalize options
test -z ''$events'' && usage ''No kernel events specified''
test -z ''$command'' && test -z ''$duration'' && \
 usage ''No command or time duration specified''
```

(continued)

```
# Perform experiment
ktrace_reset y    # Reset kernel tracing
ktrace_on    # Enable kernel tracing
run_command    # Run command, wait, or sleep
ktrace_off    # Disable kernel tracing
ktrace_xtract    # Extract buffer
ktrace_reset $do_reset   # Reset kernel tracing again
exit 0
```

Advanced Topics

Inserting Probe Points

Insert probe points in your code to:

- Trace the values of variables in your program.
- Provide internal state information that is not available through the exported interface. This is useful for debugging or for performance analysis.

  For example, use probe points to show performance relevant information hidden in C++ private classes, or to show operational data such as the collision rate in a hash table. By placing a probe point in the hash table code, the probe point can write to the trace file each time a collision is detected.

  The interface for inserting probes is defined by the `TNF_PROBE` macros `TNF_PROBE_0` through `TNF_PROBE_5`. The numbers 0 through 5 are the number of variables being traced by the macro.

  With these macros, you can insert probe points anywhere in your code to get the values of variables and to trace program execution. The `libtnfprobe` library defines the standard scalar types (ints, longs, floats, and so on), but you can define more complex structures with the `TNF_DECLARE_RECORD` and `TNF_DEFINE_RECORD` macros. See “Defining User Types for Probe Points” on page 36.

Using the `TNF_PROBE` Macros

In the simplest case, `TNF_PROBE_0`, you give no argument types:

`TNF_PROBE_0 (`*name*`,` *keys*`,` *detail*`);`

The variables are:

- *name* — The name of the probe, following all the syntax guidelines for identifiers in ANSI C. The use of `name` declares it, so no separate declaration is necessary. This is a block scope declaration, so it does not affect the name space of the program.
- *keys* — A list of groups to which the probe belongs. The list is a string containing space-separated keywords and cannot contain a semicolon, an equal sign, or a single quotation mark.(;= ‘). When any of the groups are enabled, the probe point is enabled. `keys` cannot be a variable—it must be an in-line string.
- *detail* — Provides a way for you to define your own attributes and values. The detail string is made up of attribute-value pairs separated from each other by semicolons, although the value is optional. The first word (up to a space) is considered to be the attribute and the rest of the string (up to the semicolon) is considered to be the value. Spaces around the semicolon delimiter are allowed. Single quotation marks and the equal sign are not allowed in the detail statement.

  Prefix the attribute name with a vendor stock symbol followed by the % character to avoid name collisions. In the following example, four attributes are defined: `sunw%debug`, `comX%exception`, `comY%func_entry`, and `comY%color`. Since `prex` tokenizes the value on spaces, multiword values can be matched on any of the words, but not on the entire string. For example:

```
sunw%debug entering function A;
   comX%exception no file;
   comY%func_entry;
   comY%color red blue
```

Table 1–8 gives the values that are matched on for the command shown above:

TABLE 1–8 Examples of User-Defined Attributes

Attribute	**Value**	**Values `prex` matches on**
`sunw%debug`	`entering function A`	`entering` or `function` or `A`
`comX%exception`	`no file`	`no` or `file`
`comY%func_entry`		`/.*/` (regular expression)
`comY%color`	`red blue`	`red` or `blue`

libtnfprobe reserves all attribute names that are not prefixed by a vendor symbol (it reserves all attributes that do not have the % character in them). The code for cookie.c in “A Sample C Program” on page 14 contains the following use of TNF_PROBE_0:

```
TNF_PROBE_0(start, ‘‘cookie main’’, ‘‘sunw%debug starting main’’);
```

Note - Compiling with the preprocessor option -DNPROBE (see cc(1)), or with the preprocessor control statement #define NPROBE ahead of the #include <tnf/probe.h> statement, stops probe points as well as TNF type extension code from being compiled into the program.

TNF_PROBE_1 Through TNF_PROBE_5

The numbers 1 through 5 in the argument names are used here to illustrate the number of variables you give to the probe point. For example, the syntax for TNF_PROBE_1 is:

```
TNF_PROBE_1(name, keys, detail,
arg_type_1, arg_name_1, arg_value_1);
```

and the syntax for TNF_PROBE_5 is:

```
TNF_PROBE_5(name, keys, detail,
arg_type_1, arg_name_1, arg_value_1
arg_type_2, arg_name_2, arg_value_2
arg_type_3, arg_name_3, arg_value_3
arg_type_4, arg_name_4, arg_value_4
arg_type_5, arg_name_5, arg_value_5);
```

The arguments are:

- *arg_type_n*—The type of the *n*th argument. *n* is a number from 1 through 5. The predefined types are listed in Table 1–9. See “Defining User Types for Probe Points” on page 36 for information about defining your own types.
- *arg_name_n*—The name you give the *n*th argument. Follow the ANSI C rules for identifiers, and do not use quotation marks around the argument name. (Note that the slots attribute mentioned on “Attributes ” on page 8 contains a string version of this name.)
- *arg_value_n*—The expression that is evaluated to a value that is included in the trace file. A read access is done on any variables that are mentioned in *value_n*. In a multithreaded program, place locks around the TNF_PROBE_n macro if *value_n* contains data that should be read protected.

TABLE 1–9 Predefined Types

Type	Associated C Type and Semantics
tnf_long	int, long
tnf_ulong	unsigned int, unsigned long
tnf_longlong	long long (if implemented in compilation system)
tnf_ulonglong	unsigned long long (if implemented in compilation system)
tnf_float	float
tnf_double	double
tnf_string	char *
tnf_opaque	void *

For example, the cookie.c program on “A Sample C Program” on page 14 uses TNF_PROBE_2 as follows:

```
TNF_PROBE_2(inloop, "cookie main loop","sunw%debug in the loop",
    tnf_long, loop_count,   i,
    tnf_long, total_iterations,  sum);
```

Table 1–10 explains some of the macro definitions in cookie.c.

TABLE 1–10 TNF Macro Definitions in cookie.c

TNF_PROBE_0 (	A probe with no argument types
start,	The name of the probe
"cookie main",	The list of groups the probe belongs to cookie and main (the values of the keys attribute)
"sunw%debug starting main");	User-defined attribute=sunw%debug; value=starting main (used by debug probe function)

TABLE 1–10 TNF Macro Definitions in `cookie.c` *(continued)*

TNF_PROBE_2 (	A probe with two variables
inloop,	The name of the probe
"cookie main loop",	The keys - `cookie, main`, and `loop`
"sunw%debug in the loop",	Values for the debug probe function
tnf_long,	The type of the first variable
loop_count,	The name of the first variable (value of the `slots` attribute)
i,	The first variable
tnf_long,	The type of the second variable
total_iterations,	The name of the second variable (value of the `slots` attribute)
`sum`);	The second variable
"");	

Example—Timing Functions

In Code Example 1–4, probe points are placed at the entry and exit of a function to see how much time is spent in the function. The probe at the function entry also logs the arguments to the function.

When `prex` encounters a probe point at run time that is enabled for tracing, it writes a record to the trace file. Each probe point logs the time when it was encountered and also references a tag record containing information like the file name, line number, name, keys, and detail of the probe point. These tag records are written only once, and are never overwritten in the trace file.

The first probe point, `work_args`, also logs the value of the two arguments of the probe point (`state` and `message`).

CODE EXAMPLE 1–4 Probe Points at Entry and Exit of Function

```
#include <tnf/probe.h>

int
work(int state, char *message)
```

```
{
    TNF_PROBE_2(work_start, ''work_module work''
                ''sunw%debug in function work'',
                tnf_long, int_input, state,
                tnf_string, string_input, message);
   ...
    TNF_PROBE_0(work_end, ''work_module work'', '''');
}
```

Defining User Types for Probe Points

To trace a structure in your program, define a new type with the `TNF_DECLARE_RECORD` and `TNF_DEFINE_RECORD_n` macros. These are parts of a compile time interface for extending the types sent in to probe points.

```
TNF_DECLARE_RECORD(c_type, tnf_type);


TNF_DEFINE_RECORD_1(c_type, tnf_type,
                    tnf_member_type_1,
                    tnf_member_name_1)
TNF_DEFINE_RECORD_2(c_type, tnf_type,
                    tnf_member_type_1,
                    tnf_member_name_1,
                    tnf_member_type_2,
                    tnf_member_name_2)
TNF_DEFINE_RECORD_3(c_type, tnf_type,
                    tnf_member_type_1,
                    tnf_member_name_1,
                    tnf_member_type_2,
                    tnf_member_name_2,
                    tnf_member_type_3,
                    tnf_member_name_3)
TNF_DEFINE_RECORD_4(c_type, tnf_type,
                    tnf_member_type_1,
                    tnf_member_name_1,
                    tnf_member_type_2,
                    tnf_member_name_2,
                    tnf_member_type_3,
                    tnf_member_name_3,
                    tnf_member_type_4,
                    tnf_member_name_4)
TNF_DEFINE_RECORD_5(c_type, tnf_type,
                    tnf_member_type_1,
                    tnf_member_name_1,
                    tnf_member_type_2,
                    tnf_member_name_2,
                    tnf_member_type_3,
                    tnf_member_name_3,
                    tnf_member_type_4,
                    tnf_member_name_4,
                    tnf_member_type_5,
                    tnf_member_name_5)
```

Create only one `TNF_DECLARE_RECORD` and one `TNF_DEFINE_RECORD` for each new type you define. The `TNF_DECLARE_RECORD` should precede the `TNF_DEFINE_RECORD`. It can be in a header file that multiple source files share if those source files need to use the tnf_type being defined. The TNF_DEFINE_RECORD should appear in only one of the source files.

The TNF_DEFINE_RECORD macro interface defines a function as well as several data structures. Therefore, use this interface in a source file (.c file or .cc file) at file scope and not inside a function.

Note - Do not put a semicolon after the TNF_DEFINE_RECORD statement; it will generate a compiler warning.

The variables are:

- *c_type* — The template from which the new tnf_type is created. Not all the elements of the C struct need to be provided in the TNF type being defined. *c_type* must be a C struct type.
- *tnf_type* — The name given to the newly created type. This interface uses the name space prefixed by the tnf_type. So, if a new type called xxx_type is defined by a library, then the library should not use xxx_type as a prefix in any other symbols it defines.

  The policy on managing the type name space is the same as that for managing any other name space in a library: prefix any new TNF types by the unique prefix that the rest of the symbols in the library use. This prevents name space collisions when you link multiple libraries that define new TNF types.

  For example, if a library called libpalloc.so uses the prefix pal for all symbols it defines, then it should also use the prefix pal for all new TNF types being defined.
- *tnf_member_type_n* — The TNF type of the *n*th provided member of the C structure.
- *tnf_member_name_n* — The name of the *n*th provided member of the C structure.

Examples—Defining TNF Types

Code Example 1–5 shows how a new TNF type is defined and used in a probe.

Code Example 1–5 is assumed to be part of a fictitious library called libpalloc.so that uses the prefix pal for all its symbols.

CODE EXAMPLE 1–5 Defining a new TNF type

```
#include <tnf/probe.h>

typedef struct pal_header {
        long    size;
        char * descriptor;
        struct pal_header *next;
} pal_header_t;
```

```
TNF_DECLARE_RECORD(pal_header_t, pal_tnf_header);
TNF_DEFINE_RECORD_2(pal_header_t, pal_tnf_header,
                        tnf_long,   size,
                        tnf_string, descriptor)

/*
 * Note: name space prefixed by pal_tnf_header should not be
 * used by this client any more.
 */

void
pal_free(pal_header_t *header_p)
{
        int state;

        TNF_PROBE_2(pal_free_start, ''palloc pal_free'',
                ''sunw%debug entering pal_free'',
                tnf_long,       state_var, state,
                pal_tnf_header, header_var, header_p);
        . . .
}
```

It is possible to make a `tnf_type` definition recursive or mutually recursive, such as in a structure that uses the `next` field to point to itself (a linked list).

When such a structure is sent in to a `TNF_PROBE`, then the entire linked list is logged to the trace file (until the `next` field is `NULL`). But, when the list is circular, it results in an infinite loop. To break the recursion, either omit the `next` field from the `tnf_type`, or define the type of the `next` member as `tnf_opaque.`

Performance Issues

Don't place probe points in sections of code that are traversed frequently, as in a mutex lock that is used often.

Estimate about 30 words of working set memory (10 words data and 20 words text) for each probe and about 200ns for each disabled probe on a SPARCStation10. You can control the performance degradation of the application by controlling the number and placement of probes.

If you are shipping a library with probe points, it is important to run benchmarks to ensure that the performance is still at an acceptable level. Reduce the number of probes or change their positions to increase performance.

/proc

`dbx`, `truss`, and `prex` all use `/proc` to control the target process. `/proc` allows only one client to control a target process safely. Because of this, you cannot run programs like `dbx` and `prex` simultaneously on the same target program. If you try to run prex on a target while dbx or truss is running on the same target, prex displays the message “Cannot attach to target.”

You can, however, interleave prex and dbx execution by following these steps:

1. Start prex.
2. Set up the state of the probe points.
3. Give the command quit suspend.
4. Start dbx.
5. Attach to the suspended program.

   The target will not execute any code between prex and dbx.

   You can also suspend the target by sending it a SIGSTOP signal, then type “quit resume” to prex. If you do this, you should also send a SIGCONT signal after invoking dbx on the stopped process (or else dbx hangs).

dlopen() and dlclose() and History

Probes in shared objects that are brought in by a dlopen(3X) are automatically set up according to the prex command history. When a shared object is removed by a dlclose(3X), prex refreshes its understanding of the probes in the target program. This implies that there is more work to do for dlopen and dlclose, so they take slightly longer.

If you are not interested in this feature and don’t want dlopen and dlclose to be perturbed, detach prex from the target.

Signals

prex does not interfere with signals that are delivered directly to the target program. However, prex receives all terminal-generated signals, such as Control-c (SIGINT) and Control-z (SIGSTOP), and does not forward them to the target program.

Use the kill(1) command from a shell to signal the target program.

Failure of Event-Writing Operations

A few failure points, like system call failures, are possible when writing events to trace files. These failures result in a failure code being set in the target process. The target process continues normally (but no trace records are written).

Whenever a user types Control-c to prex to get to a prex prompt, prex checks the failure code in the target and informs the user if there was a tracing failure.

Target Executing a fork() or exec()

If your program does a fork(), any probes that the child encounters are logged to the same trace file. Events are annotated with a process ID, so it is possible to determine which process a particular event came from.

A thread in a multithreaded program doing a fork while the other threads are still running can cause a race conditon. For the trace file to stay uncorrupted, make sure that the other threads are quiescent when doing a fork, or else use fork1(2).

If the target program itself (not any children it might fork) does an exec(2), prex detaches from the target and exits. The user can reconnect prex with:

```
$ prex -p pid
```

CHAPTER **2**

Lexical Analysis

With the `lex(1)` software tool, you can solve problems from text processing, code enciphering, and compiler writing. In text processing, you might check the spelling of words for errors; in code enciphering, you might translate certain patterns of characters into others; and in compiler writing, you might determine what the tokens (smallest meaningful sequences of characters) are in the program to be compiled.

The task common to all these problems is lexical analysis: recognizing different strings of characters that satisfy certain characteristics. Hence the name lex. You don't have to use lex to handle problems of this kind. You could write programs in a standard language like C to handle them. In fact, what lex does is produce such C programs: lex is therefore called a program generator.

What lex offers you is typically a faster, easier way to create programs that perform these tasks. Its weakness is that it often produces C programs that are longer than necessary for the task at hand and that execute more slowly than they otherwise might. In many applications this is a minor consideration, and the advantages of using lex considerably outweigh it.

lex can also be used to collect statistical data on features of an input text, such as character count, word length, number of occurrences of a word, and so forth. The remaining sections of this chapter include:

- Generating a lexical analyzer program
- Writing lex source
- Translating lex source
- C++ Mangled Symbols
- Using lex and yacc Together
- Automaton
- Summary of Source Format

Internationalization

For information about using lex to develop applications in languages other than English, see `lex (1)`.

Generating a Lexical Analyzer Program

`lex` generates a C-language scanner from a source specification that you write. This specification contains a list of rules indicating sequences of characters — expressions — to be searched for in an input text, and the actions to take when an expression is found. To see how to write a lex specification see the section “Writing `lex` Source ” on page 44.

The C source code for the lexical analyzer is generated when you enter

```
$ lex lex.l
```

where `lex.l` is the file containing your lex specification. (The name `lex.l` is conventionally the favorite, but you can use whatever name you want. Keep in mind, though, that the `.l` suffix is a convention recognized by other system tools, `make` in particular.) The source code is written to an output file called `lex.yy.c` by default. That file contains the definition of a function called `yylex()` that returns 1 whenever an expression you have specified is found in the input text, 0 when end of file is encountered. Each call to `yylex()` parses one token (assuming a return); when `yylex()` is called again, it picks up where it left off.

Note that running `lex` on a specification that is spread across several files, as in the following example, produces one `lex.yy.c`:

```
$ lex lex1.l lex2.l lex3.l
```

Invoking `lex` with the `–t` option causes it to write its output to `stdout` rather than `lex.yy.c`, so that it can be redirected:

```
$ lex -t lex.l > lex.c
```

Options to `lex` must appear between the command name and the filename argument.

The lexical analyzer code stored in `lex.yy.c` (or the `.c` file to which it was redirected) must be compiled to generate the executable object program, or scanner, that performs the lexical analysis of an input text.

The lex library supplies a default main( ) that calls the function yylex( ), so you need not supply your own main( ). The library is accessed by invoking the –ll option to cc:

```
$ cc lex.yy.c -ll
```

Alternatively, you might want to write your own driver. The following is similar to the library version:

```
extern int yylex();

int yywrap()
{
     return(1);
}

main()
{
     while (yylex());
          ;
}
```

For more information about the function yywrap( ), see the “Writing lex Source ” on page 44 section. When your driver file is compiled with lex.yy.c, as in the following example, its main( ) will call yylex( ) at run time exactly as if the lex library had been loaded:

```
$ cc lex.yy.c driver.c
```

The resulting executable file reads stdin and writes its output to stdout. Figure 2–1 shows how lex works.

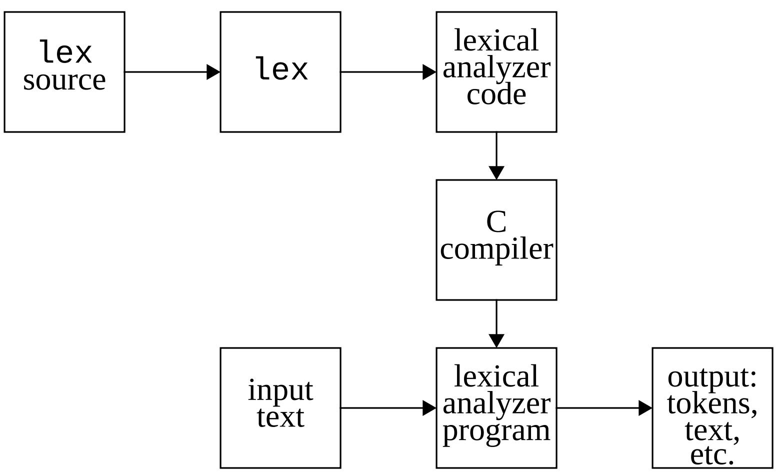


Figure 2–1 Creation and Use of a Lexical Analyzer with lex

Writing lex Source

lex source consists of at most three sections: definitions, rules, and user-defined routines. The rules section is mandatory. Sections for definitions and user routines are optional, but must appear in the indicated order if present:

```
definitions
%%
rules
%%
user routines
```

The Fundamentals of lex Rules

The mandatory rules section opens with the delimiter %%. If a routines section follows, another %% delimiter ends the rules section. The %% delimiters must be entered at the beginning of a line, that is, without leading blanks. If there is no second delimiter, the rules section is presumed to continue to the end of the program.

Lines in the rules section that begin with white space and that appear before the first rule are copied to the beginning of the function `yylex()`, immediately after the first brace. You might use this feature to declare local variables for `yylex()`.

Each rule specifies the pattern sought and the actions to take on finding it. The pattern specification must be entered at the beginning of a line. The scanner writes input that does not match a pattern directly to the output file. So the simplest lexical analyzer program is just the beginning rules delimiter, %%. It writes out the entire input to the output with no changes at all.

Regular Expressions

You specify the patterns you are interested in with a notation called a regular expression. A regular expression is formed by stringing together characters with or without operators. The simplest regular expressions are strings of text characters with no operators at all:

```
apple
orange
pluto
```

These three regular expressions match any occurrences of those character strings in an input text. To have the scanner remove every occurrence of `orange` from the input text, you could specify the rule

```
orange ;
```

Because you specified a null action on the right with the semicolon, the scanner does nothing but print the original input text with every occurrence of this regular expression removed, that is, without any occurrence of the string orange at all.

Operators

Unlike `orange`, most expressions cannot be specified so easily. The expression itself might be too long, or, more commonly, the class of desired expressions is too large; it might, in fact, be infinite.

Using `lex` operators — summarized in Table 2–1 — you can form regular expressions for any expression of a certain class. The `+` operator, for instance, means one or more occurrences of the preceding expression, the `?` means 0 or 1 occurrences of the preceding expression (which is equivalent to saying that the preceding expression is optional), and the `*` means 0 or more occurrences of the preceding expression. So `m+` is a regular expression that matches any string of `m`s:

```
mmm
m
mmmmm
```

and `7*` is a regular expression that matches any string of zero or more 7s:

```
77
77777

777
```

The empty third line matches because it has no 7s in it at all.

The `|` operator indicates alternation, so that `ab|cd` matches either `ab` or `cd`. The operators `{}` specify repetition, so that `a{1,5}` looks for 1 to 5 occurrences of `a`, and `A(B{1,4})` matches ABC, ABBC, ABBBC, and ABBBBC (notice the use of parentheses, `()`, as grouping symbols).

Brackets, `[]`, indicate any one character from the string of characters specified between the brackets. Thus, `[dgka]` matches a single `d`, `g`, `k`, or `a`. The characters between brackets must be adjacent, without spaces or punctuation.

The ^ operator, when it appears as the first character after the left bracket, indicates all characters in the standard set except those specified between the brackets. (Note that `|`, {,}} and ^ may serve other purposes as well.)

Ranges within a standard alphabetic or numeric order (A through Z, a through z, 0 through 9) are specified with a hyphen. [a-z], for instance, indicates any lowercase letter.

```
[A-Za-z0-9*&#]
```

This is a regular expression that matches any letter (whether upper or lowercase), any digit, an asterisk, an ampersand, or a #.

Given the following input text, the lexical analyzer with the previous specification in one of its rules will recognize *, &, r, and #, perform on each recognition whatever action the rule specifies (we have not indicated an action here), and print the rest of the text as it stands:

```
$$$$?? ????!!!*$$ $$$$$$&+====r~~# ((
```

To include the hyphen character in the class, have it appear as the first or last character in the brackets: [-A-Z] or [A-Z-].

The operators become especially powerful in combination. For example, the regular expression to recognize an identifier in many programming languages is:

```
[a-zA-Z][0-9a-zA-Z]*
```

An identifier in these languages is defined to be a letter followed by zero or more letters or digits, and that is just what the regular expression says. The first pair of brackets matches any letter. The second, if it were not followed by a *, would match any digit or letter.

The two pairs of brackets with their enclosed characters would then match any letter followed by a digit or a letter. But with the *, the example matches any letter followed by any number of letters or digits. In particular, it would recognize the following as identifiers:

```
e
not
idenTIFIER
pH
EngineNo99
R2D2
```

Note that it would not recognize the following as identifiers because not_idenTIFIER has an embedded underscore; 5times starts with a digit, not a letter; and $hello starts with a special character:

```
not_idenTIFIER
5times
$hello
```

A potential problem with operator characters is how to specify them as characters to look for in a search pattern. The previous example, for instance, does not recognize text with a * in it. lex solves the problem in one of two ways: an operator character preceded by a backslash, or characters (except backslash) enclosed in double quotation marks, are taken literally, that is, as part of the text to be searched for.

To use the backslash method to recognize, say, an * followed by any number of digits, you can use the pattern:

```
\*[1-9]*
```

To recognize a \ itself, we need two backslashes: \\. Similarly, "x*x" matches x*x, and "y\"z" matches y"z. Other lex operators are noted as they arise; see Table 1-1:

TABLE 2–1 lex Operators

<table>
<tr><th>Expression</th><th>Description</th></tr>
<tr><td>\x</td><td>x, if x is a lex operator</td></tr>
<tr><td>"xy"</td><td>xy, even if x or y is a lex operator (except \)</td></tr>
<tr><td>[xy]</td><td>x or y</td></tr>
<tr><td>[x-z]</td><td>x, y, or z</td></tr>
<tr><td>[^x]</td><td>Any character but x</td></tr>
<tr><td>.</td><td>Any character but newline</td></tr>
<tr><td>^x</td><td>x at the beginning of a line</td></tr>
<tr><td><y>x</td><td>x when lex is in start condition y</td></tr>
<tr><td>x$</td><td>x at the end of a line</td></tr>
<tr><td>x?</td><td>Optional x</td></tr>
<tr><td>x*</td><td>0, 1, 2, ... instances of x</td></tr>
<tr><td>x+</td><td>1, 2, 3, ... instances of x</td></tr>
<tr><td>x{m,n}</td><td>m through n occurrences of x</td></tr>
<tr><td>xx|yy</td><td>Either xx or yy</td></tr>
<tr><td>x |</td><td>The action on x is the action for the next rule</td></tr>
<tr><td>(x)</td><td>x</td></tr>
<tr><td>x/y</td><td>x but only if followed by y</td></tr>
<tr><td>{xx}</td><td>The translation of xx from the definitions section</td></tr>
</table>

Actions

After the scanner recognizes a string matching the regular expression at the start of a rule, it looks to the right of the rule for the action to be performed. You supply the actions.

Kinds of actions include recording the token type found and its value, if any; replacing one token with another; and counting the number of instances of a token or token type. You write these actions as program fragments in C.

An action can consist of as many statements as are needed. You might want to change the text in some way or print a message noting that the text has been found. So, to recognize the expression `Amelia Earhart` and to note such recognition, apply the rule:

```
"Amelia Earhart" printf("found Amelia");
```

To replace lengthy medical terms in a text with their equivalent acronyms, a rule such as the following would work:

```
Electroencephalogram printf("EEG");
```

To count the lines in a text, you recognize the ends of lines and increment a line counter.

`lex` uses the standard C escape sequences, including `\n` for newline. So, to count lines you might have the following syntax, where `lineno`, like other C variables, is declared in the “Definitions ” on page 53 section.

```
\n  lineno++;
```

Input is ignored when the C language null statement, a colon `;`, is specified. So the following rule causes blanks, tabs, and new-lines to be ignored:

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

The alternation operator **|** can also be used to indicate that the action for a rule is the action for the next rule. The previous example could have been written with the same result:

```
" " |
\t
\n ;
```

The scanner stores text that matches an expression in a character array called `yytext[]`. You can print or manipulate the contents of this array as you like. In fact, `lex` provides a macro called `ECHO` that is equivalent to `printf ("%s", yytext)`.

When your action consists of a long C statement, or two or more C statements, you might write it on several lines. To inform lex that the action is for one rule only, enclose the C code in braces.

For example, to count the total number of all digit strings in an input text, print the running total of the number of digit strings, and print out each one as soon as it is found, your `lex` code might be:

```
\+?[1-9]+      { digstrngcount++;
                  printf("%d",digstrngcount);
                  printf("%s", yytext); }
```

This specification matches digit strings whether or not they are preceded by a plus sign because the ? indicates that the preceding plus sign is optional. In addition, it catches negative digit strings because that portion following the minus sign matches the specification.

Advanced lex Features

You can process input text riddled with complicated patterns by using a suite of features provided by lex. These include rules that decide which specification is relevant when more than one seems so at first, functions that transform one matching pattern into another, and the use of definitions and subroutines.

Here is an example that draws together several of the points already covered:

```
%%
-[0-9]+      printf("negative integer");
\+?[0-9]+    printf("positive integer");
-0.[0-9]+    printf("negative fraction, no whole number part");
rail[ \t]+road   printf("railroad is one word");
crook        printf("Here’s a crook");
function     subprogcount++;
G[a-zA-Z]*   { printf("may have a G word here:%s", yytext);
                       Gstringcount++; }
```

The first three rules recognize negative integers, positive integers, and negative fractions between 0 and -1. The terminating + in each specification ensures that one or more digits compose the number in question.

Each of the following rules recognizes a specific pattern:

- The specification for railroad matches cases where one or more blanks intervene between the two syllables of the word. In the cases of railroad and crook, synonyms could have been printed rather than the messages.
- The rule recognizing a function increments a counter.

  This rule illustrates several points:

  - The braces specify an action sequence that extends over several lines.
  - The action uses the lex array yytext[], which stores the recognized character string.
  - The specification uses the * to indicate that zero or more letters can follow the G.

Some Special Features

Besides storing the matched input text in yytext[], the scanner automatically counts the number of characters in a match and stores it in the variable `yyleng`. You can use this variable to refer to any specific character just placed in the array `yytext[]`.

Remember that C language array indexes start with 0, so to print the third digit (if there is one) in a just-recognized integer, you might enter:

```
[1-9]+    {if (yyleng > 2)
             printf("%c", yytext[2]);
            }
```

`lex` follows a number of high-level rules to resolve ambiguities that might arise from the set of rules that you write. In the following lexical analyzer example, the ‘‘reserved word’’ end could match the second rule as well as the eighth, the one for identifiers:

```
begin       return(BEGIN);
end         return(END);
while       return(WHILE);
if          return(IF);
package     return(PACKAGE);
reverse     return(REVERSE);
loop        return(LOOP);
[a-zA-Z][a-zA-Z0-9]*  { tokval = put_in_tabl();
                         return(IDENTIFIER); }
[0-9]+   { tokval = put_in_tabl();
             return(INTEGER); }
\+       { tokval = PLUS;
            return(ARITHOP); }
\-       { tokval = MINUS;
            return(ARITHOP); }
>        { tokval = GREATER;
            return(RELOP); }
>=       { tokval = GREATEREQL;
            return(RELOP); }
```

`lex` follows the rule that, where there is a match with two or more rules in a specification, the first rule is the one whose action is executed. Placing the rule for end and the other reserved words before the rule for identifiers ensures that the reserved words are recognized.

Another potential problem arises from cases where one pattern you are searching for is the prefix of another. For instance, the last two rules in the lexical analyzer example above are designed to recognize > and >=.

`lex` follows the rule that it matches the longest character string possible and executes the rule for that string. If the text has the string >= at some point, the scanner recognizes the >= and acts accordingly, instead of stopping at the > and executing the > rule. This rule also distinguishes + from ++ in a C program.

When the analyzer must read characters beyond the string you are seeking, use trailing context. The classic example is the `DO` statement in FORTRAN. In the following DO( ) statement, the first 1 looks like the initial value of the index k until the first comma is read:

```
DO 50 k = 1 , 20, 1
```

Until then, this looks like the assignment statement:

```
DO50k = 1
```

Remember that FORTRAN ignores all blanks. Use the slash, /, to signify that what follows is trailing context, something not to be stored in yytext[], because the slash is not part of the pattern itself.

The rule to recognize the FORTRAN DO statement could be:

```
DO/([ ]*[0-9]+[ ]*[a-zA-Z0-9]+=[a-zA-Z0-9]+,) {
        printf("found DO");
        }
```

While different versions of FORTRAN limit the identifier size, here the index name, this rule simplifies the example by accepting an index name of any length. See the “Start Conditions ” on page 54 section for a discussion of a similar handling of prior context.

lex uses the $ symbol as an operator to mark a special trailing context — the end of a line. An example would be a rule to ignore all blanks and tabs at the end of a line:

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

The previous example could also be written as:

```
[ \t]+/\n ;
```

To match a pattern only when it starts a line or a file, use the ^ operator. Suppose a text-formatting program requires that you not start a line with a blank. You could check input to the program with the following rule:

```
^[ ]    printf("error: remove leading blank");
```

Notice the difference in meaning when the ^ operator appears inside the left bracket.

lex Routines

The following macros enable you to perform special actions.

- input() reads another character
- unput() puts a character back to be read again a moment later
- output() writes a character on an output device

One way to ignore all characters between two special characters, such as between a pair of double quotation marks, is to use input() like this:

```
\"      while (input() != '"');
```

After the first double quotation mark, the scanner reads all subsequent characters, and does not look for a match, until it reads the second double quotation mark. (See the further examples of `input()` and `unput(c)` usage in the "User Routines " on page 56 section.)

For special I/O needs that are not covered by these default macros, such as writing to several files, use standard I/O routines in C to rewrite the macro functions.

These routines, however, must be modified consistently. In particular, 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 `input()` and `unput(c)` must be maintained or the `lex` lookahead will not work.

If you do provide your own `input()`, `output(c)`, or `unput(c)`, write a `#undef input` and so on in your definitions section first:

```
#undef input
#undef output
  . . .
#define input() ...  etc.
more declarations
  . . .
```

Your new routines will replace the standard ones. See the "Definitions " on page 53 section for further details.

A `lex` library routine that you can redefine is `yywrap()`, which is called whenever the scanner reaches the end of file. If `yywrap()` returns 1, the scanner continues with normal wrapup on the end of input. To arrange for more input to arrive from a new source, redefine `yywrap()` to return 0 when more processing is required. The default `yywrap()` always returns 1.

Note that it is not possible to write a normal rule that recognizes end of file; the only access to that condition is through `yywrap()`. 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.

lex routines that let you handle sequences of characters to be processed in more than one way include yymore(), yyless(n), and REJECT. Recall that the text that matches a given specification is stored in the array yytext[]. In general, once the action is performed for the specification, the characters in yytext[] are overwritten with succeeding characters in the input stream to form the next match.

The function yymore(), by contrast, ensures that the succeeding characters recognized are appended to those already in yytext[]. This lets you do things sequentially, such as when one string of characters is significant and a longer one that includes the first is significant as well.

Consider a language that defines a string as a set of characters between double quotation marks and specifies that to include a double quotation mark in a string, it must be preceded by a backslash. The regular expression matching that is somewhat confusing, so it might be preferable to write:

```
\"[^"]* {
     if (yytext[yyleng-2] == '\\')
         yymore();
     else
         ... normal processing
     }
```

When faced with the string "abc\"def", the scanner first matches the characters "abc\. Then the call to yymore() causes the next part of the string "def to be tacked on the end. The double quotation mark terminating the string is picked up in the code labeled ‘‘normal processing.’’

With the function yyless(n) you can specify the number of matched characters on which an action is to be performed: only the first n characters of the expression are retained in yytext[]. Subsequent processing resumes at the nth + 1 character.

Suppose you are deciphering code, and working with only half the characters in a sequence that ends with a certain one, say an upper or lowercase Z. You could write:

```
[a-yA-Y]+[Zz] { yyless(yyleng/2);
                 ...  process first half of string
                 ...
                }
```

Finally, with the REJECT function, you can more easily process strings of characters even when they overlap or contain one another as parts. REJECT does this by immediately jumping to the next rule and its specification without changing the contents of yytext[]. To count the number of occurrences both of the regular expression snapdragon and of its subexpression dragon in an input text, the following works:

```
snapdragon    {countflowers++; REJECT;}
dragon         countmonsters++;
```

As an example of one pattern overlapping another, the following counts the number of occurrences of the expressions comedian and diana, even where the input text has sequences such as comediana:

```
comedian    {comiccount++; REJECT;}
diana        princesscount++;
```

The actions here can be considerably more complicated than incrementing a counter. In all cases, you declare the counters and other necessary variables in the definitions section at the beginning of the lex specification.

Definitions

The lex definitions section can contain any of several classes of items. The most critical are external definitions, preprocessor statements like #include, and abbreviations. For legal lex source this section is optional, but in most cases some of these items are necessary. Preprocessor statements and C source code appear between a line of the form %{ and one of the form %}.

All lines between these delimiters — including those that begin with white space — are copied to lex.yy.c immediately before the definition of yylex( ). (Lines in the definition section that are not enclosed by the delimiters are copied to the same place provided they begin with white space.)

The definitions section is where you usually place C definitions of objects accessed by actions in the rules section or by routines with external linkage.

For example, when using lex with yacc, which generates parsers that call a lexical analyzer, include the file y.tab.h, which can contain #defines for token names:

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

After the %} that ends your #includes and declarations, place your abbreviations for regular expressions in the rules section. The abbreviation appears on the left of the line and, separated by one or more spaces, its definition or translation appears on the right.

When you later use abbreviations in your rules, be sure to enclose them within braces. Abbreviations avoid repetition in writing your specifications and make them easier to read.

As an example, reconsider the lex source reviewed in the section " Advanced lex Features " on page 49. Using definitions simplifies later reference to digits, letters, and blanks.

This is especially true when the specifications appear several times:

```
D                  [0-9]
L                  [a-zA-Z]
B                  [ \t]+
%%
-{D}+              printf("negative integer");
\+?{D}+            printf("positive integer");
-0.{D}+            printf("negative fraction");
G{L}*              printf("may have a G word
                           here");
rail{B}road        printf("railroad is one word");
crook              printf("criminal");
 ...
 ...
```

Start Conditions

Start conditions provide greater sensitivity to prior context than is afforded by the ^ operator alone. You might want to apply different rules to an expression depending on a prior context that is more complex than the end of a line or the start of a file.

In this situation you could set a flag to mark the change in context that is the condition for the application of a rule, then write code to test the flag. Alternatively, you could define for lex the different ‘‘start conditions’’ under which it is to apply each rule.

Consider this problem:

- Copy the input to the output, except change the word `magic` to the word `first` on every line that begins with the letter `a`.
- Change `magic` to `second` on every line that begins with `b`.
- Change `magic` to `third` on every line that begins with `c`. Here is how the problem might be handled with a flag.

  Recall that `ECHO` is a `lex` macro equivalent to `printf("%s", yytext)`:

```
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;
        }
  }
```

To handle the same problem with start conditions, each start condition must be introduced to `lex` in the definitions section with a line, such as the following one, where the conditions can be named in any order:

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

The word `Start` can be abbreviated to `S` or `s`. The conditions are referenced at the head of a rule with `<>` brackets. So the following is a rule that is recognized only when the scanner is in start condition `name1`:

```
<name1>expression
```

To enter a start condition, execute the action following statement:

```
BEGIN name1;
```

The above statement changes the start condition to `name1`. To resume the normal state, use the following:

```
BEGIN 0;
```

This resets the initial condition of the scanner.

A rule can be active in several start conditions. For example, the following is a legal prefix:

```
<name1,name2,name3>
```

Any rule not beginning with the <> prefix operators is always active.

The example can be written with start conditions as follows:

```
%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");
```

User Routines

You can use your `lex` routines in the same ways you use routines in other programming languages. Action code used for several rules can be written once and called when needed. As with definitions, this simplifies program writing and reading.

The `put_in_tabl()` function, discussed in the "Using `lex` and `yacc` Together" on page 59 section, fits well in the user routines section of a `lex` specification.

Another reason to place a routine in this section is to highlight some code of interest or to simplify the rules section, even if the code is to be used for one rule only. As an example, consider the following routine to ignore comments in a language like C where comments occur between `/*` and `*/`:

```
%{
static skipcmnts();
%}
%%
"/*"       skipcmnts();
 ...
 ...    /* rest of rules */
%%
static
skipcmnts()
{
       for(;;)
       {
            while (input() != '*')
                  ;

            if (input() != '/')
                unput(yytext[yyleng-1])
            else return;
       }
}
```

There are three points of interest in this example.

- First, the `unput(c)` macro puts back the last character that was read to avoid missing the final / if the comment ends unusually with a `**/`.

  In this case, after the scanner reads an *, it finds that the next character is not the terminal / and it continues reading.

- Second, the expression `yytext[yyleng-1]` picks the last character read.
- Third, this routine assumes that the comments are not nested, as is the case with the C language.

C++ Mangled Symbols

If the function name is a C++ mangled symbol, `lex` prints its demangled format. All mangled C++ symbols are bracketed by [] following the demangled symbol. For regular mangled C++ function names (including member and non-member functions), the function prototype is used as its demangled format.

For example,

```
_ct_13Iostream_initFv
```

is printed as:

```
Iostream_init::Iostream_init()
```

C++ static constructors and destructors are demangled and printed in the following format:

```
static constructor function for
```

or

```
static destructor function for
```

For example,

```
_std_stream_in_c_Fv
```

is demangled as

```
static destructor function for _stream_in_c
```

For C++ virtual table symbols, its mangled name takes the following format:

```
_vtbl_class
```

```
_vtbl_root_class_derived_class
```

In the lex output, the demangled names for the virtual table symbols are printed as

```
virtual table for class
```

```
virtual table for class derived_class derived from root_class
```

For example, the demangled format of

```
_vtbl_7fstream
```

is

```
virtual table for fstreamH
```

And the demangled format of

```
_vtbl_3ios_18ostream_withassign
```

is

```
virtual table for class ostream_withassign derived from ios
```

Some C++ symbols are pointers to the virtual tables; their mangled names take the following format:

```
_ptbl_class_filename
```

```
_ptbl_root_class_derived_class_filename
```

In the lex output, the demangled names for these symbols are printed as:

```
pointer to virtual table for class in filename
```

```
pointer to virtual table for class derived class derived from
  root_class in filename
```

For example, the demangled format of

```
_ptbl_3ios_stream_fstream_c
```

is

```
pointer to the virtual table for ios in _stream_fstream_c
```

and the demangled format of

```
_ptbl_3ios_11fstreambase_stream_fstream_c
```

is

```
_stream_fstream_c
```

```
pointer to the virtual table for class fstreambase derived
 from ios in _stream_fstream_c
```

Using lex and yacc Together

If you work on a compiler project or develop a program to check the validity of an input language, you might want to use the system tool yacc (Chapter 3). yacc generates parsers, programs that analyze input to insure that it is syntactically correct.

lex and yacc often work well together for developing compilers.

As noted, a program uses the lex-generated scanner by repeatedly calling the function yylex(). This name is convenient because a yacc-generated parser calls its lexical analyzer with this name.

To use lex to create the lexical analyzer for a compiler, end each lex action with the statement return token, where token is a defined term with an integer value.

The integer value of the token returned indicates to the parser what the lexical analyzer has found. The parser, called yyparse() by yacc, then resumes control and makes another call to the lexical analyzer to get another token.

In a compiler, the different values of the token indicate what, if any, reserved word of the language has been found or whether an identifier, constant, arithmetic operator, or relational operator has been found. In the latter cases, the analyzer must also specify the exact value of the token: what the identifier is, whether the constant is, say, 9 or 888, whether the operator is + or *, and whether the relational operator is = or >.

Consider the following portion of lex source for a scanner that recognizes tokens in a "C-like" language:

TABLE 2–2 Sample lex Source Recognizing Tokens

```
begin       return(BEGIN);
end         return(END);
while       return(WHILE);
if          return(IF);
package      return(PACKAGE);
reverse      return(REVERSE);
loop        return(LOOP);
[a-zA-Z][a-zA-Z0-9]*
        { tokval = put_in_tabl();
              return(IDENTIFIER); }
[0-9]+     { tokval = put_in_tabl();
              return(INTEGER); }
\+        { tokval = PLUS;
              return(ARITHOP); }
\-        { tokval = MINUS;
              return(ARITHOP); }
>        { tokval = GREATER;
              return(RELOP); }
>=        { tokval = GREATEREQL;
              return(RELOP); }
```

The tokens returned, and the values assigned to tokval, are integers. Good programming style suggests using informative terms such as BEGIN, END, and WHILE, to signify the integers the parser understands, rather than using the integers themselves.

You establish the association by using #define statements in your C parser calling routine. For example:

```
#define BEGIN 1
#define END 2
 ...
#define PLUS 7
 ...
```

Then, to change the integer for some token type, change the #define statement in the parser rather than change every occurrence of the particular integer.

To use yacc to generate your parser, insert the following statement in the definitions section of your lex source:

```
#include "y.tab.h"
```

The file y.tab.h, which is created when yacc is invoked with the –d option, provides #define statements that associate token names such as BEGIN and END with the integers of significance to the generated parser.

To indicate the reserved words in Table 2–2, the returned integer values suffice. For the other token types, the integer value is stored in the variable tokval.

This variable is globally defined so that the parser and the lexical analyzer can access it. yacc provides the variable yylval for the same purpose.

Note that Table 2–2 shows two ways to assign a value to tokval.

- First, a function put_in_tabl( ) places the name and type of the identifier or constant in a symbol table so that the compiler can refer to it.

  More to the present point, put_in_tabl( ) assigns a type value to tokval so that the parser can use the information immediately to determine the syntactic correctness of the input text. The function put_in_tabl( ) is a routine that the compiler writer might place in the user routines section of the parser.

- Second, in the last few actions of the example, tokval is assigned a specific integer indicating which arithmetic or relational operator the scanner recognized.

  If the variable PLUS, for instance, is associated with the integer 7 by means of the #define statement above, then when a + is recognized, the action assigns to tokval the value 7, which indicates the +.

  The scanner indicates the general class of operator by the value it returns to the parser (that is, the integer signified by ARITHOP or RELOP).

  When using lex with yacc, either can be run first. The following command generates a parser in the file y.tab.c:

```
$ yacc d grammar.y
```

As noted, the –d option creates the file y.tab.h, which contains the #define statements that associate the yacc-assigned integer token values with the user-defined token names. Now you can invoke lex with the following command:

```
$ lex lex.l
```

You can then compile and link the output files with the command:

```
$ cc lex.yy.c y.tab.c -ly -ll
```

The yacc library is loaded with the –ly option before the lex library with the –ll option to insure that the supplied main( ) calls the yacc parser.

Also, to use yacc with CC, especially when routines like yyback( ), yywrap( ), and yylook( ) in .l files are to be extern C functions, the command line must include the following.

```
$ CC -D__EXTERN_C__ ... filename
```

Automaton

Recognition of expressions in an input text is performed by a deterministic finite automaton generated by lex. The –v option prints a small set of statistics describing the finite automaton. (For a detailed account of finite automata and their importance for lex, see the Aho, Sethi, and Ullman text, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.)

lex uses a table to represent its finite automaton. The maximum number of states that the finite automaton allows is set by default to 500. If your lex source has many rules or the rules are very complex, you can enlarge the default value by placing another entry in the definitions section of your lex source:

```
%n 700
```

This entry tells lex to make the table large enough to handle as many as 700 states. (The –v option indicates how large a number you should choose.)

To increase the maximum number of state transitions beyond 2000, the designated parameter is a:

```
%a 2800
```

Finally, see lex(1) for a list of all the options available with the lex command.

Summary of Source Format

The general form of a lex source file is:

```
definitions
%
rules
%
user routines
```

The definitions section contains any combination of:

- Definitions of abbreviations in the form:

  name space translation

- Included code in the form:

```
% {
  C code
  % }
```

- Start conditions in the form:

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

- 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.

Changes to internal array sizes could be represented as follows:

TABLE 2–3 Internal Array Sizes

`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 can be continued on succeeding lines by using braces to mark it.

The lex operator characters are:

```
" \ [] ^ - ? .  * | () $ / {} <> +
```

Important lex variables, functions, and macros are:

TABLE 2–4 `lex` Variables, Functions, and Macros

`yytext[]`	array of char
`yyleng`	int
`yylex()`	function
`yywrap()`	function

TABLE 2–4 `lex` Variables, Functions, and Macros *(continued)*

`yymore()`	function
`yyless(n)`	function
`REJECT`	macro
`ECHO`	macro
`input()`	macro
`unput(c)`	macro
`output(c)`	macro

CHAPTER 3

yacc — A Compiler Compiler

The yacc (yet another compiler compiler) utility provides a general tool for imposing structure on the input to a computer program. Before using yacc, you prepare a specification that includes:

- A set of rules to describe the elements of the input
- Code to be invoked when a rule is recognized
- Either a definition or declaration of a low-level scanner to examine the input

yacc then turns the specification into a C-language function that examines the input stream. This function, called a parser, works by calling the low-level scanner.

The scanner, called a lexical analyzer, picks up items from the input stream. The selected items are known as tokens. Tokens are compared to the input construct rules, called grammar rules.

When one of the rules is recognized, the code you have supplied for the rule is invoked. This code is called an action. Actions are fragments of C-language code. They can return values and use values returned by other actions.

The heart of the yacc specification is the collection of grammar rules. Each rule describes a construct 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 constructs of interest; presumably, month_name, day, and year are defined in greater detail elsewhere.

In the example, the comma is enclosed in single quotes. This means that the comma is to appear literally in the input. The colon and semicolon are punctuation in the rule and have no significance in evaluating the input. With proper definitions, the input:

```
July 4, 1776
```

might be matched by the rule.

The lexical analyzer is an important part of the parsing function. This user-supplied routine reads the input stream, recognizes the lower-level constructs, and communicates these as tokens to the parser. The lexical analyzer recognizes constructs of the input stream as terminal symbols; the parser recognizes constructs as nonterminal symbols. To avoid confusion, refer to terminal symbols as tokens.

There is considerable leeway in deciding whether to recognize constructs using the lexical analyzer or grammar rules. For example, the rules:

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;
...
month_name : 'D' 'e' 'c' ;
```

might be used in the above example. While the lexical analyzer only needs to recognize individual letters, such low-level rules tend to waste time and space and can 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 and the detailed rules are not needed.

Literal characters, such as a comma, must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the previous example the rule:

```
date : month '/' day '/' year ;
```

allowing:

```
7/4/1776
```

as a synonym for:

```
July 4, 1776
```

on input. In most cases, this new rule could be slipped into a working system with minimal effort and little danger of disrupting existing input.

The input being read might not conform to the specifications. With a left-to-right scan, input errors are detected as early as is theoretically possible. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data usually can be found quickly.

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 might be self-contradictory, or they might require a more powerful recognition mechanism than that available to yacc.

The former cases represent design errors; the latter cases often can 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 constructs 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 program development.

The remainder of this chapter describes the following subjects:

- Basic process of preparing a yacc specification
- Parser operation
- Handling ambiguities
- Handling operator precedences in arithmetic expressions
- Error detection and recovery
- The operating environment and special features of the parsers yacc produces
- Suggestions to improve the style and efficiency of the specifications
- Advanced topics

In addition, the chapter offers two examples and a summary of the yacc input syntax.

Internationalization

To use yacc in the development of applications in languages other than English, see yacc(1) for further information.

Basic Specifications

Names refer to either tokens or nonterminal symbols. yacc requires token names to be declared as such. While the lexical analyzer might be included as part of the specification file, it is perhaps more in keeping with modular design to keep it as a separate file. Like the lexical analyzer, other subroutines can be included as well.

Thus, every specification file theoretically consists of three sections: the declarations, (grammar) rules, and subroutines. The sections are separated by double percent signs (%%; the percent sign is generally used in yacc specifications as an escape character).

When all sections are used, a full specification file looks like:

```
declarations
%%
rules
%%
subroutines
```

The declarations and subroutines sections are optional. The smallest legal yacc specification might be:

```
%%
S:;
```

Blanks, tabs, and newlines are ignored, but they cannot appear in names or multicharacter reserved symbols. Comments can 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 form:

```
A: BODY ;
```

where A represents a nonterminal symbol, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are yacc punctuation.

Names can be of any length and can consist of letters, periods, underscores, and digits, although a digit cannot be the first character of a name. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule can 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. yacc recognizes all C language escape sequences. For a number of technical reasons, the null character should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule is dropped before a vertical bar.

Thus the grammar rules:

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to yacc as:

```
A      : B C D
       | E F
       | G
;
```

by using the vertical bar. It is not necessary that all grammar rules with the same left side appear together in the grammar rules section although it makes the input more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by:

```
epsilon : ;
```

The blank space following the colon is understood by yacc to be a nonterminal symbol named epsilon.

Names representing tokens must be declared. This is most simply done by writing:

```
$token name1 name2 name3
```

and so on in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the start symbol has particular importance. By default, the symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible and desirable to declare the start symbol explicitly in the declarations section using the %start keyword:

```
$start symbol
```

The end of the input to the parser is signaled by a special token, called the end-marker. The end-marker is represented by either a zero or a negative number.

If the tokens up to but not including the end-marker form a construct 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.

Actions

With each grammar rule, you can associate actions to be performed when the rule is recognized. Actions can return values and can 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 subroutines, and alter arrays and variables. An action is specified by one or more statements enclosed in { and }. For example, the following two examples are grammar rules with actions:

```
A    : ’(’ B ’)’
     {
         hello( 1, "abc" );
     }
```

and

```
XXX : YYY ZZZ
        {
            (void) printf("a message\n");
            flag = 25;
        }
```

The $ symbol is used to facilitate communication between the actions and the parser. The pseudo-variable $$ represents the value returned by the complete action.

For example, the action:

```
{ $$ = 1; }
```

returns the value of one; in fact, that's all it does.

To obtain the values returned by previous actions and the lexical analyzer, the action can use the pseudo-variables $1, $2, ... $n. These refer to the values returned by components 1 through *n* of the right side of a rule, with the components being numbered from left to right. If the rule is

```
A       : B C D ;
```

then $2 has the value returned by C, and $3 the value returned by D. The following rule provides a common example:

```
expr    : '(' expr ')' ;
```

You would expect the value returned by this rule to be the value of the expr within the parentheses. Since the first component of the action is the literal left parenthesis, the desired logical result can be indicated by:

```
expr      : '(' expr ')'
          {
              $$ = $2 ;
          }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the following form frequently need not have an explicit action:

```
A : B ;
```

In previous examples, all the actions came at the end of rules. Sometimes, it is desirable to get control before a rule is fully parsed. yacc permits an action to be written in the middle of a rule as well as at the end.

This action is assumed to return a value accessible through the usual $ mechanism by the actions to the right of it. In turn, it can access the values returned by the symbols to its left. Thus, in the rule below, the effect is to set x to 1 and y to the value returned by C:

```
A      : B
      {
          $$ = 1;
```

```
      }
C
      {
          x = $2;
          y = $3;
      }
      ;
```

Actions that do not terminate a rule are 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 by recognizing this added rule.

yacc treats the above example as if it had been written

```
$ACT      : /* empty */
          {
               $$ = 1;
          }
          ;
A         :  B $ACT C
          {
               x = $2;
               y = $3;
          }
          ;
```

where $ACT is an empty action.

In many applications, output is not a directl result of the actions. A data structure, such as a parse tree, is constructed in memory and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired.

For example, suppose there is a C-function node written so that the call:

```
node( L, n1, n2 )
```

creates a node with label L and descendants n1 and n2 and returns the index of the newly created node. Then a parse tree can be built by supplying actions such as in the following specification:

```
expr     : expr '+' expr
         {
             $$ = node( '+', $1, $3 );
         }
```

You can define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in %{ and%}.These declarations and definitions have global scope, so they are known to the action statements and can be made known to the lexical analyzer. For example:

```
%{ int variable = 0; %}
```

could be placed in the declarations section making variable accessible to all of the actions. You should avoid names beginning with yy because the yaccparser uses only such names. Note, too, that in the examples shown thus far, all the values are integers.

A discussion of values is found in the section “Advanced Topics ” on page 92. Finally, note that in the following case:

```
%{
      int i;
      printf("%}");
%}
```

yacc starts copying after %{ and stops copying when it encounters the first %}, the one in printf( ). In contrast, it would copy %{ in printf( ) if it encountered it there.

Lexical Analysis

You 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 a value is 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 can be chosen by yacc or the user. In either case, the #define mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically.

For example, suppose that the token name DIGIT has been defined in the declarations section of the yacc specification file. The relevant portion of the lexical analyzer might look like the following to return the appropriate token:

```
int yylex()
{
     extern int yylval;
     int c;
       ...
     c = getchar();
       ...
     switch (c)
     {
         ...
       case ’0’:
       case ’1’:
         ...
       case ’9’:
           yylval = c - ’0’;
           return (DIGIT);
         ...
      }
         ...
}
```

The intent is to return a token number of DIGIT and a value equal to the numerical value of the digit. You put the lexical analyzer code in the subroutines section and the declaration for DIGIT in the declarations section. Alternatively, you can put the lexical analyzer code in a separately compiled file, provided you:

- Invoke yacc with the –d option, which generates a file called y.tab.h that contains #define statements for the tokens.
- #include y.tab.h in the separately compiled lexical analyzer.

  This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is the use of any token names in the grammar that are reserved or significant in C language or the parser.

  For example, the use of token names if or while will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name error is reserved for error handling and should not be used naively.

  In the default situation, token numbers are chosen by yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

  If you prefer to assign the token numbers, the first appearance of the token name or literal in the declarations section must be followed immediately by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined this way are assigned default definitions by yacc. The potential for duplication exists here. Care must be taken to make sure that all token numbers are distinct.

  For historical reasons, the end-marker must have token number 0 or negative. You cannot redefine this token number. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token upon reaching the end of their input.

  As noted in Chapter 2, lexical analyzers produced by lex are designed to work in close harmony with yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. lex can be used to produce quite complicated lexical analyzers, but there remain some languages that do not fit any theoretical framework and whose lexical analyzers must be crafted by hand.

Parser Operation

Use yacc to turn the specification file into a C language procedure, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex and is not discussed here. The parser itself, though, is relatively simple and understanding its usage makes treatment of error recovery and ambiguities easier.

The parser produced by yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token, called the lookahead token. The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0 (the stack contains only state 0) and no lookahead token has been read.

The machine has only four actions available: shift, reduce, accept, and error. A step of the parser is done as follows:

1. Based on its current state, the parser decides if it needs a lookahead 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 lookahead token if needed, the parser decides on its next action and carries it out. This can result in states being pushed onto the stack or popped off of the stack and in the lookahead token being processed or left alone.

   The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there can be an action

```
IF shift 34
```

that says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The reduce action keeps the stack from growing without bounds. reduce actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right-hand side by the left-hand side. It might be necessary to consult the lookahead token to decide whether or not to reduce. In fact, the default action (represented by .) is often a reduce action.

reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. The following action refers to grammar rule 18:

```
.  reduce 18
```

However, the following 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-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.)

In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z* and no longer serve any useful purpose. After popping these states, a state is uncovered, which was the state the parser was in before beginning to process the rule.

Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of *A*. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a `goto` action. In particular, the `lookahead` token is cleared by a shift but is not affected by a `goto`. In any case, the uncovered state contains an entry such as the following, causing 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-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off the stacks. The uncovered state is in fact the current state.

The `reduce` action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions.

When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the `goto` action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables `$1`, `$2`, 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 `lookahead` 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 `lookahead` token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing.

The error recovery (as opposed to the detection of error) is discussed in the “Error Handling ” on page 85 section.

Consider the following as a `yacc` specification:

```
$token      DING DONG DELL
%%
rhyme     : sound place
          ;
sound     : DING DONG
          ;
place     : DELL
```

```
        ;
```

When yacc is invoked with the –v (verbose) option, a file called y.output is produced which describes the parser.

The y.output file corresponding to the above grammar (with some statistics stripped off the end) follows.

```
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 actions for each state are specified and there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen and what is yet to come in each rule. The following input can be used to track the operations of the parser:

```
DING DONG DELL
```

Initially, the current state is state 0.

The parser refers to the input to decide between the actions available in state 0, so the first token, `DING`, is read and becomes the `lookahead` token. The action in state 0 on `DING` is shift 3, state 3 is pushed onto the stack, and the `lookahead` token is cleared. State 3 becomes the current state. The next token, `DONG`, is read and becomes the `lookahead` token.

The action in state 3 on the token `DONG` is shift 6, state 6 is pushed onto the stack, and the `lookahead` is cleared. The stack now contains 0, 3, and 6. In state 6, without consulting the `lookahead`, the parser reduces by

```
sound : DING DONG
```

which is rule 2. 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),

```
sound goto 2
```

is obtained. State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token, `DELL`, must be read. The action is shift 5, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the `lookahead` token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered.

The `goto` in state 2 on place (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again.

- In state 0, there is a `goto` on `rhyme`, causing the parser to enter state 1.
- In state 1, the input is `read` 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.

Ambiguity and Conflicts

A set of grammar rules is ambiguous if some input string 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
```

Notice that this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is the following:

```
expr - expr - expr
```

the rule allows this input to be structured as either:

```
( expr - expr ) - expr
```

or as:

```
expr - ( expr - expr )
```

The first is called left association, the second right association.

`yacc` detects such ambiguities when it is attempting to build the parser. Given that the input is as follows, consider the problem that confronts the parser:

```
expr - expr - expr
```

When the parser has read the second `expr`, the input seen is:

```
expr - expr
```

It matches the right side of the grammar rule above. The parser could `reduce` the input by applying this rule. After applying the rule, the input is reduced to `expr` (the left side of the rule). The parser then reads the final part of the input (as represented below) 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 the following is seen:

```
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. Thus, having read the following, the parser can do one of two legal things, `shift` or `reduce`:

```
expr - expr
```

It has no way of deciding between them. This is called a `shift-reduce` conflict. It might also happen that the parser has a choice of two legal reductions. This is called a `reduce-reduce` conflict. 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.

yacc invokes two default disambiguating rules:

1. In a shift-reduce conflict, the default is to shift.
2. In a reduce-reduce conflict, the default is to reduce by the earlier grammar rule (in the yacc specification).

   Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce-reduce conflicts should be avoided when possible.

   Conflicts can 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 rules 1 and 2 above.

   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.

   This rewriting is somewhat unnatural and produces slower parsers. Thus, yacc produces 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 is called the simple if rule and the second the if-else rule.

These two rules form an ambiguous construction because input of the form:

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 )
{
     IF ( C2 )
         S1
}
ELSE
     S2
```

or:

```
IF ( C1 )
{
    IF ( C2 )
        S1
    ELSE
        S2
}
```

where the second interpretation is the one given in most programming languages having this construct; each `ELSE` is associated with the last preceding un-`ELSE`'d `IF`. In this example, consider the situation where the parser has seen the following 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:

```
IF ( C1 ) stat ELSE S2
```

by the `if-else` rule. This leads to the first of the above groupings of the input.

On the other hand, the `ELSE` may be shifted, S2 read, and then the right-hand portion of:

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the `if-else` rule to get:

```
IF ( C1 ) stat
```

which can be reduced by the simple if rule.

This leads to the second of the above groupings of the input, which is usually the one desired.

Once again, the parser can do two valid things — there is a `shift-reduce` conflict. The application of disambiguating rule 1 tells the parser to `shift` in this case, which leads to the desired grouping. This `shift-reduce` conflict arises only when there is a particular current input symbol, `ELSE`, and particular inputs, such as:

```
IF ( C1 ) IF ( C2 ) S1
```

have already been seen. In general, there can be many conflicts, and each one is 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 –v output. 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 underscore 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 possible things. If the input symbol is ELSE, it is possible to shift into state 45.

State 45 has, as part of its description, the line:

```
stat : IF ( cond ) stat ELSE_stat
```

because the ELSE has been shifted in this state. In state 23, the alternative action (specified by .) is to be done if the input symbol is not mentioned explicitly in the actions. In this case, if the input symbol is not ELSE, the parser reduces to:

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

by grammar rule 18.

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the y.output file, rule numbers are printed in parentheses after those rules that can be reduced. In most states, there is a reduce action possible, and reduce is the default command. If you encounter unexpected shift-reduce conflicts, look at the –v output to decide whether the default actions are appropriate.

Precedence

There is one common situation where the rules 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, with information about left or right associativity. Ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form:

```
expr : expr OP expr
```

and:

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts.

You specify as disambiguating rules 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 the 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. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. + and - are left associative and have lower precedence than * and /, which are also left associative. The keyword %right is used to describe right associative operators. The keyword %nonassoc is used to describe operators, like the operator .LT. in FORTRAN, that may not associate with themselves. That is, because:

```
A .LT.  B .LT.  C
```

is illegal in FORTRAN, .LT. would be described with the keyword %nonassoc in yacc.

As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-
%left '*' '/'

%%

expr    : expr '=' expr
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
         | expr '/' expr
        | NAME
        ;
```

might be used to structure the input

```
a = b = c * d - e - f * g
```

as follows

```
a = ( b = ( ((c * d) - e) - (f * g) ) )
```

in order to achieve the correct precedence of operators. When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary minus.

Unary minus cab 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. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules:

```
%left '+' '-'
%left '*' '/'

%%

expr     : expr '+' expr
         | expr '-' expr
         | expr '*' expr
         | expr '/' expr
         | '-' expr %prec '*'
         | NAME
         ;
```

might be used to give unary minus the same precedence as multiplication.

A token declared by `%left`, `%right`, and `%nonassoc` need not, but may, be declared by `%token` as well.

Precedences and associativities are used by `yacc` to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. Precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the final token or literal in the body of the rule. If the `%prec` construction is used, it overrides this default value. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a `reduce-reduce` or `shift-reduce` conflict, and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given in the preceding section are used, and the conflicts are reported.
4. If there is a `shift-reduce` conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the

conflict is resolved in favor of the action — `shift` or `reduce` — associated with the higher precedence. If precedences are equal, then associativity is used. Left associative implies `reduce`; right associative implies `shift`; 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 strictly until some experience has been gained. The `y.output` file is useful in deciding whether the parser is actually doing what was intended.

To illustrate further how you might use the precedence keywords to resolve a shift-reduce conflict, look at an example similar to the one described in the previous section. Consider the following C statement:

```
if (flag) if (anotherflag) x = 1;
    else x = 2;
```

The problem for the parser is whether the `else` goes with the first or the second `if`. C programmers will recognize that the `else` goes with the second `if`, contrary to what the misleading indentation suggests. The following `yacc` grammar for an `if-then-else` construct abstracts the problem. That is, the input *iises* will model these C statements.

```
%{
#include <stdio.h>
%}
%token SIMPLE IF ELSE
%%
S           ;  stmnt
            ;
stmnt       : SIMPLE
            | if_stmnt
            ;
if_stmnt    : IF stmnt
                 { printf("simple if\n");}
            | IF stmnt ELSE stmnt
                 { printf("if_then_else\n");}
            ;
%%
int
yylex() {
        int c;
        c=getchar();
        if (c= =EOF) return 0;
        else switch(c) {
            case 'i': return IF;
            case 's': return SIMPLE;
            case 'e': return ELSE;
            default: return c;
        }
}
```

When the specification is passed to yacc, however, we get the following message:

```
conflicts: 1 shift/reduce
```

The problem is that when yacc has read *iis* in trying to match i*ises*, it has two choices: recognize *is* as a statement (reduce) or read some more input (shift) and eventually recognize *ises* as a statement.

One way to resolve the problem is to invent a new token REDUCE, whose sole purpose is to give the correct precedence for the rules:

```
%{
#include <stdio.h>
%}
%token SIMPLE IF
%nonassoc REDUCE
%nonassoc ELSE
%%
S       : stmnt '\n'
        ;
stmnt   : SIMPLE
        | if_stmnt
        ;
if_stmnt    : IF stmnt %prec REDUCE
                { printf("simple if");}
            | IF stmnt ELSE stmnt
                { printf("if_then_else");}
            ;
%%
```

Since the precedence associated with the second form of if_stmnt is higher now, yacc tries to match that rule first, and no conflict is reported.

Actually, in this simple case, the new token is not needed:

```
%nonassoc IF
%nonassoc ELSE
```

would also work. Moreover, it is not really necessary to resolve the conflict in this way, because, as we have seen, yacc shifts by default in a shift-reduce conflict. Resolving conflicts is a good idea, though, in the sense that you should not see diagnostic messages for correct specifications.

Error Handling

Error handling contains many semantic problems. When an error is found, for example, it might 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 the token name error. This name can be used in grammar rules. In effect, it suggests where errors are expected and recovery might take place.

The parser pops its stack until it enters a state where the token error is legal. It then behaves as if the token error were the current lookahead token and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

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 deleted.

As an example, a rule of the form:

```
stat : error
```

means that on a syntax error the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and starts processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it might 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 can be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, and so forth. Error rules such as the above are very general but difficult to control.

Rules such as the following are somewhat easier:

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

Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule is reduced and any cleanup action associated with it is performed.

Another form of error rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example:

```
input     : error '\n'
           {
               (void) printf("Reenter last line: " );
           }
           input
           {
               $$ = $4;
           }
```

```
;
```

is one way to do this. There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is unacceptable.

For this reason, there is a mechanism that can force the parser to believe that error recovery has been accomplished. The statement:

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as:

```
input      : error '\n'
            {
                yyerrok;
                (void) 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 `lookahead` token must be cleared. The statement:

```
yyclearin ;
```

in an action has this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by `yylex()` is presumably the first token in a legal statement. The old illegal token must be discarded and the error state reset. A rule similar to:

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

could perform this.

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

The yacc Environment

You create a yacc parser with the command:

```
$ yacc grammar.y
```

where grammar.y is the file containing your yacc specification. (The .y suffix is a convention recognized by other operating system commands. It is not strictly necessary.) The output is a file of C-language subroutines called y.tab.c. The function produced by yacc is called yyparse( ), and is integer-valued.

When it is called, it repeatedly calls yylex( ), the lexical analyzer supplied by the user (see “Lexical Analysis ” on page 72), to obtain input tokens. Eventually, an error is detected, yyparse( ) returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, yyparse( ) returns the value 0.

You 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 routine called main( ) must be defined that eventually calls yyparse( ). In addition, a routine called yyerror( ) is needed to print a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using yacc, a library has been provided with default versions of main( ) and yyerror( ). The library is accessed by a –ly argument to the cc command. The source codes:

```
main()
{
     return (yyparse());
}
```

and:

```
# include <stdio.h>

yyerror(s)
char *s;
{
     (void) fprintf(stderr, "%s\n", s);
}
```

show the triviality of these default programs. The argument to yyerror( ) is a string containing an error message, usually the string syntax error. The average application wants to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the `lookahead` token number at the time the error was detected (a feature which gives better diagnostic)s. Since the `main( )` routine is probably supplied by the user (to read arguments, for instance), 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 outputs a verbose description of its actions, including a discussion of the input symbols read and the parser actions.

Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy-to-change, and clear specifications.

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. This is useful in debugging.
- Put grammar rules and actions on separate lines to make editing easier.
- Put all rules with the same left-hand side together. Put the left-hand side in only once and let all following rules begin with a vertical bar.
- Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be easily added.
- Indent rule bodies by one tab stop and action bodies by two tab stops.
- Put complicated actions into subroutines defined in separate files.

  Examples in this section are written following this style. The central problem is to make the rules visible through the maze of action code.

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:

```
list     : item
          | list ',' item
          ;
```

and:

```
seq      : item
         | seq item
         ;
```

frequently arise when writing specifications of sequences and lists. In each of these cases, the first rule is reduced for the first item only; and the second rule is reduced for the second and all succeeding items.

With right-recursive rules, such as:

```
seq     : item
        | item seq
        ;
```

the parser is somewhat larger; the items are seen and reduced from right to left. More seriously, an internal stack in the parser is in danger of overflowing if an extremely long sequence is read (although `yacc` can now process very large stacks). Thus, you should use left recursion wherever reasonable.

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

```
seq      : /* empty */
         | seq item
         ;
```

using an empty rule. Once again, the first rule is always reduced exactly once before the first item is read, and the second rule is reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if `yacc` is asked to decide which empty sequence it has seen when it hasn't seen enough to know.

C++ Mangled Symbols

The material for this section is an exact duplication of material found in the "C++ Mangled Symbols" on page 57 section of Chapter 2. Please substitute `yacc` when they refer to lex.

Lexical Tie-Ins

Some lexical decisions depend on context. For example, the lexical analyzer might normally delete blanks, but not within quoted strings, or names might be entered into a symbol table in declarations but not in expressions. One way of handling these situations is to create a global flag that is examined by the lexical analyzer and set by actions. For example,

```
%{
        int dflag;
%}
...  other declarations ...
%%
prog     : decls stats
         ;
decls    :      /* empty */
         {
             dflag = 1;
         }
         | decls declaration
         ;
stats    :     /* empty */
         {
             dflag = 0;
         }
         | stats statement
         ;

.  .  .  other rules .  .  .
```

specifies a program consisting of zero or more declarations followed by zero or more statements. The flag *dflag* is now 0 when reading statements and 1 when reading declarations, except for the first token in the first statement.

This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan. This approach represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit you to use words like `if`, which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of `yacc`.

It is difficult to pass information to the lexical analyzer, telling it this instance of `if` is a keyword and that instance is a variable. Using the information found in the previous section, “Lexical Tie-Ins ” on page 91 might prove useful here.

Advanced Topics

This part discusses a number of advanced features of yacc.

Simulating error and accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. The YYACCEPT macro causes yyparse() to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; yyerror() is called, and error recovery takes place.

These mechanisms can be used to simulate parsers with multiple end-markers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules

An action can refer to values returned by actions to the left of the current rule. The mechanism is the same as ordinary actions, $ followed by a digit.

```
sent     : adj noun verb adj noun
          {
              look at the sentence ...
          }
          ;
adj       : THE
          {
              $$ = THE;
          }
          | YOUNG
          {
              $$ = YOUNG;
          }
             ...
              ;
noun      : DOG
          {
              $$ = DOG;
          }
          | CRONE
          {
              if ( $0 = = YOUNG )
              {
                  (void) printf( "what?\n" );
              }
              $$ = CRONE;
          }
          ;
...
```

In this case, the digit can be 0 or negative. In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Notice, however, that this is only possible when a great deal is known about what might precede the symbol noun in the input. 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.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. yacc can also support values of other types including structures. In addition, yacc keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type checked. The yacc value stack is declared to be a union of the various types of values desired. You declare the union and associate union member names with 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.

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

To declare the union, you include:

```
%union
{
     body of union
}
```

in the declaration section. This declares the yacc value stack and the external variables *yylval* and *yyval* to have type equal to this union. If yacc was invoked with the –d option, the union declaration is copied into the y.tab.h file as YYSTYPE.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction:

```
<name>
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed.

Thus, saying

```
%left <optype> ’+’ ’-’
```

causes any reference to values returned by these two tokens to be tagged with the union member name optype. Another keyword, `%type`, is used to associate union member names with nonterminals. You could use the rule

```
%type <nodetype> expr stat
```

to associate the union member `nodetype` with the nonterminal symbols `expr` and `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 *`$0`*) leaves `yacc` with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between < and > immediately after the first `$`. The example below:

```
rule       : aaa
           {
                $<intval>$ = 3;
           }
           bbb
           {
                fun( $<intval>2, $<other>0 );
           }
           ;
```

shows this usage. This syntax has little to recommend it, but the situation arises occasionally.

The facilities in this subsection are not triggered until they are used. In particular, the use of `%type` turns 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.

yacc Input Syntax

This section has a description of the `yacc` input syntax as a `yacc` specification. Context dependencies and so forth are not considered. Although `yacc` accepts an `LALR(1)` grammar, the `yacc` input specification language is specified as an `LR(2)` grammar; the difficulty arises 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 figures out 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 `IDENTIFIER`s but never as part of `C_IDENTIFIER`s.

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

        /* basic entries */
%token IDENTIFIER    /* includes identifiers and literals */

%token C_IDENTIFIER   /* identifier (but not literal) */
                       /* followed by a : */

%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 t

%%

spec     : defs MARK rules tail
           ;
tail     : MARK
          {
             In this action,read in the rest of the file
          }
         |       /* empty: the second MARK is optional */
         ;
defs     :         /* empty */
         | defs def
         ;
def      : START IDENTIFIER
         | UNION
         {
             Copy union definition to output
          }
          | LCURL
         {
            Copy C code to output file
         }
         RCURL
           | rword tag nlist
           ;
rword    : TOKEN
         | LEFT
         | RIGHT
         | NONASSOC
          | TYPE
         ;
tag    :        /* empty: union tag is optional */
         | ’<’ IDENTIFIER ’>’
         ;
nlist    : nmno
         | nlist nmno
```

```
        | nlist ',' nmno
        ;
nmno    : IDENTIFIER /* Note: literal illegal with % type */
         | IDENTIFIER NUMBER  /* Note: illegal with % type */
         ;

                  /* rule section */
rules   : C_IDENTIFIER rbody prec
        | rules rule
        ;
rule    : C_IDENTIFIER rbody prec
        | '|' rbody prec
        ;
rbody   :        /* empty */
        | rbody IDENTIFIER
        | rbody act
        ;
act  : '{'
        {
            Copy action translate $$ etc.
        }
        '}'
        ;
prec    :        /* empty */
         | PREC IDENTIFIER
        | PREC IDENTIFIER act
         | prec ';'
        ;
```

Examples

A Simple Example

The following code sample shows the complete yacc applications for a small desk calculator. The calculator has 26 registers labeled a through z and accepts arithmetic expressions made up of the operators +, -, *, /, %, &, |, and the assignment operators.

If an expression at the top level is an assignment, only the assignment is made; otherwise, the expression is printed. As in the C language, an integer that begins with 0 is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a yacc specification, the desk calculator shows how precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer is much simpler than for most applications, and the output is produced immediately, line by line.

Note the way that decimal and octal integers are read by grammar rules. This job can also be performed by the lexical analyzer.

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

int regs[26];
int base;
%}

%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' %'
%left UMINUS        /* supplies precedence for unary minus */

%%              /* beginning of rules section */

list      :        /* empty */
          | list stat '\n'
          | list error '\n'
          {
              yyerrok;
          }
          ;
stat      : expr
          {
              (void) 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;
```

(continued)

```
        }
          | 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;
          }
          ;

%%          /* beginning of subroutines section */
int yylex( )  /* lexical analysis routine */
{          /* return LETTER for lowercase letter, */
              /* yylval = 0 through 25 returns DIGIT */
               /* for digit, yylval = 0 through 9 */
              /* all other characters are returned immediately */
   int c;                  /*skip blanks*/
   while ((c = getchar()) = = ' ')
       ;
                /* c is now nonblank */

   if (islower(c)) {
      yylval = c - 'a';
      return (LETTER);
   }
   if (isdigit(c)) {
      yylval = c - '0';
      return (DIGIT);
   }
   return (c);
}
```

An Advanced Example

This section gives an example of a grammar using some of the advanced features. The desk calculator in Example 1 is modified to provide a desk calculator that performs floating point interval arithmetic. The calculator understands floating point constants, and the arithmetic operations +, -, *, /, and unary -. It uses the registers a through z. Moreover, it understands intervals written

```
(X,Y)
```

where X is less than or equal to Y. There are 26 interval-valued variables *A* through *Z* that can also be used. The usage is similar to that in Example 1; assignments return no value and print nothing while expressions print the (floating or interval) value.

This example explores a number of features of `yacc` and C. 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 means of `typedef`.

The `yacc` value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that the entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the actions call functions that return structures as well.

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 a syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Notice that scalar value can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through `yacc`: 18 shift-reduce and 26 reduce-reduce.

The problem can be seen by looking at the two input lines:

```
2.5 + (3.5 - 4.)
```

and:

```
2.5 + (3.5, 4)
```

Notice that the 2.5 is to be used in an interval value expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and do something else. 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 can 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 instructive. 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. The following code sample is a `yacc` specification.

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

typedef struct interval {
      double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv();
double atof();
double dreg[26];
INTERVAL vreg[26];

%}

%start lines

%union {
       int ival;
        double dval;
       INTERVAL vval;
}

%token <ival> DREG VREG   /* indices into dreg, vreg arrays */
%token <dval> CONST       /* floating point constant */
%type <dval> dexp         /* expression */
%type <vval> vexp         /* interval expression */

         /* precedence information about the operators */
```

(continued)

(Continuation)

```
%left '+' '/-'
%left '*' '/'

%%        /* beginning of rules section */

lines     : /* empty */
            | lines line
          ;
line     : dexp '\n'
            {
              (void)printf("%15.8f\n", $1);
            }
            | vexp '\n'
            {
             (void)printf("(%15.8f, %15.8f)\n", $1.lo, $1.hi);
            }
            | DREG '=' dexp '\n'
            {
             dreg[$1] = $3;
            }
            | VREG '=' vexp '\n'
            {
            vreg[$1] = $3;
            }
            | error '\n'
            {
            yyerrok;
            }
            ;
dexp      : CONST
            | DREG
            {
            $$ = dreg[$1];
            }
            | dexp '+' dexp
            {
              $$ = $1 + $3;
            }
            | dexp '-' dexp
            {
             $$ = $1 - $3;
            }
            | dexp '*' dexp
            {
             $$ = $1 * $3;
            }
            | dexp '/' dexp
            {
             $$ = $1 / $3;
            }
            | '-' dexp
            {
```

(continued)

```
              $$ = -$2;
            }
            | '(' dexp ')'
            {
             $$ = $2;
            }
            ;
vexp      : dexp
            {
           $$.hi = $$.lo = $1;
            }
            | '(' dexp ',' dexp ')'
            {
             $$.lo = $2;
              $$.hi = $4;
               if($$.lo > $$.hi) {
                (void) printf("interval out of order\n");
               YYERROR;
            }
            }
            | VREG
            {
              $$ = vreg[$1];
            }
            | vexp '+' vexp
            {
              $$.hi = $1.hi + $3.hi;
              $$.lo = $1.lo + $3.lo;
            }
            | dexp '+' vexp
            {
              $$.hi = $1 + $3.hi;
              $$.lo = $1 + $3.lo;
            }
            | vexp '-' vexp
            {
              $$.hi = $1.hi - $3.lo;
              $$.lo = $1.lo - $3.hi;
            }
            | dexp '-' vexp
            {
              $$.hi = $1 - $3.lo;
              $$.lo = $1 - $3.hi;
            }
            | vexp '*' vexp
            {
              $$ = vmul($1.lo, $1.hi, $3);
            }
            | dexp '*' vexp
            {
              $$ = vmul($1, $1, $3);
            }
            | vexp '/' vexp
```

(continued)

```
            {
              if (dcheck($3)) YYERROR;
              $$ = vdiv($1.lo, $1.hi, $3);
            }
            | dexp '/' vexp
            {
              if (dcheck($3)) YYERROR;
              $$ = vdiv($1, $1, $3);
            }
            | '-' vexp
            {
              $$.hi = -$2.lo;
              $$.lo = -$2.hi;
            }
            | '(' vexp ')'
            {
              $$ = $2;
            }
            ;

%%              /* beginning of subroutines section */

# define BSZ 50   /* buffer size for floating point number */

/* lexical analysis */

int yylex()
{
        register int c;
                     /* skip over blanks */
        while ((c=getchar()) = = ' ')
                ;
        if (isupper(c)) {
             yylval.ival = c - 'A';
              return(VREG);
        }
        if (islower(c)) {
              yylval.ival = c - 'a';
             return(DREG);
        }

                     /* digits, points, exponents */
        if (isdigit(c) || c = = '.') {
             char buf[BSZ + 1], *cp = buf;
             int dot = 0, exp = 0;
               for (;(cp - buf) < BSZ; ++cp, c = getchar()) {
                *cp = c;
                if (isdigit(c))
                continue;
                if (c = = '.') {
                    if (dot++ || exp)
                    return('.'); /* will cause syntax error */
                    continue;
```

(continued)

(Continuation)

```
                }
                if (c = = 'e') {
                    if (exp++)
                    return('e'); /* will cause syntax error */
                        continue;
                }
                /* end of number */
                break;
             }
         *cp = '\0';
         if (cp - buf >= BSZ)
            (void)printf("constant too long -- truncated\n");
           else
            ungetc(c, stdin); /* push back last char read */
            yylval.dval = atof(buf);
         return(CONST);
      }
      return(c);
}

INTERVAL
hilo(a, b, c, d)
double a, b, c, d;
   /* returns the smallest interval containing a, b, c, and d
      used by vmul, vdiv routines */

     INTERVAL v;

     if (a > b) {
        v.hi = a;
        v.lo = b;
     }
     else {
        v.hi = b;
        v.lo = a;
     }
     if (c > d) {
        if (c > v.hi)
           v.hi = c;
           if (d < v.lo)
              v.lo = d;
     }
     else {
        if (d > v.hi)
           v.hi = d;
           if (c < v.lo)
              v.lo = c;:
     }
     return(v);
}

INTERVAL
vmul(a, b, v)
```

(continued)

```
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.) {
         (void) printf("divisor interval contains 0.\n");
         return(1);
      }
return(0);
}

INTERVAL
vdiv(a, b, v)
double a, b;
INTERVAL v;
{
       return(hilo(a / v.hi, a / v.lo, b / v.hi, b / v.lo));
}
```

CHAPTER 4

make Utility

This chapter describes the make utility, which includes:

- Hidden dependency checking
- Command dependency checking
- Pattern-matching rules
- Automatic retrieval of SCCS files

This version of the make utility runs successfully with makefiles written for previous versions of make. Makefiles that rely on enhancements might not be compatible with other versions of this utility (see Appendix A, "System V make" for more information on previous versions of make). Refer to "make Enhancements Summary " on page 168 for a complete summary of enhancements and compatibility issues.

make streamlines the process of generating and maintaining object files and executable programs. It helps compile programs consistently and eliminates unnecessary recompilation of modules that are unaffected by source code changes.

make provides features that simplify compilations. You can also use it to automate any complicated or repetitive task that is not interactive. You can use make to update and maintain object libraries, to run test suites, and to install files onto a file system or tape. In conjunction with SCCS, you can use make to ensure that a large software project is built from the desired versions in an entire hierarchy of source files.

make reads a file that you create, called a *makefile*, which contains information about what files to build and how to build them. After you write and test the makefile, you can forget about the processing details; make takes care of them.

Dependency Checking: `make` vs. Shell Scripts

While it is possible to use a shell script to assure consistency in trivial cases, scripts to build software projects are often inadequate. On the one hand, you don't want to wait for a simple minded script to compile every single program or object module when only one of them has changed. On the other hand, having to edit the script for each iteration can defeat the goal of consistency. Although it is possible to write a script of sufficient complexity to recompile only those modules that require it, `make` does this job better.

`make` allows you to write a simple, structured listing of what to build and how to build it. It uses the mechanism of *dependency checking* to compare each module with the source or intermediate files it derives from. `make` only rebuilds a module if one or more of these prerequisite files, called *dependency files*, has changed since the module was last built.

To determine whether a derived file is out of date with respect to its sources, `make` compares the modification time of the (existing) module with that of its dependency file. If the module is missing, or if it is older than the dependency file, `make` considers it to be out of date, and issues the commands necessary to rebuild it. A module can be treated as out of date if the commands used to build it have changed.

Because `make` does a complete dependency scan, changes to a source file are consistently propagated through any number of intermediate files or processing steps. This lets you specify a hierarchy of steps in a top-to-bottom fashion.

You can think of a `makefile` as a recipe. `make` reads the recipe, decides which steps need to be performed, and executes only those steps that are required to produce the finished module. Each file to build, or step to perform, is called a *target*. The `makefile` entry for a target contains its name, a list of targets on which it depends, and a list of commands for building it.

The list of commands is called a *rule*. `make` treats dependencies as prerequisite targets, and updates them (if necessary) before processing its current target. The rule for a target need not always produce a file, but if it does, the file for which the target is named is referred to as the *target file*. Each file from which a target is derived (for example, the file the target depends on) is called a *dependency file*.

If the rule for a target produces no file by that name, `make` performs the rule and considers the target to be up-to-date for the remainder of the run.

`make` assumes that only *it* will make changes to files being processed during the current run. If a source file is changed by another process while `make` is running, the files it produces might be in an inconsistent state.

Writing a Simple Makefile

The basic format for a makefile target entry is:

TABLE 4–1 Makefile Target Entry Format

```
target . . . : [ dependency . . . ]
        [ command ]
         . . .
```

In the first line, the list of target names is terminated by a colon. This, in turn, is followed by the dependency list, if there is one. If several targets are listed, this indicates that each such target is to be built independently using the rule supplied.

Subsequent lines that start with a TAB are taken as the command lines that comprise the target rule. A common error is to use SPACE characters instead of the leading TAB.

Lines that start with a # are treated as comments up until the next (unescaped) NEWLINE and do not terminate the target entry. The target entry is terminated by the next non-empty line that begins with a character other than TAB or #, or by the end of the file.

A trivial makefile might consist of just one target shown in the following figure:

TABLE 4–2 A Trivial Makefile

```
test:
   ls test
   touch test
```

When you run make with no arguments, it searches first for a file named makefile, or, if there is no file by that name, Makefile. If either of these files is under SCCS control, make checks the makefile against its history file. If it is out of date, make extracts the latest version.

If make finds a makefile, it begins the dependency check with the first target entry in that file. Otherwise, you must list the targets to build as arguments on the command line. make displays each command it runs while building its targets.

```
$ make
ls test
test not found
touch test
ls test
test
```

Because the file test was not present (and therefore out of date), make performed the rule in its target entry. If you run make a second time, it issues a message indicating that the target is now up to date and skips the rule:

```
$ make
'test' is up to date.
```

Note - make invokes a Bourne shell to process a command line if that line contains any shell metacharacters, such as a semicolon (;) redirection symbols (<, >, >>, |), substitution symbols (*, ?, [], $, =), or quotes, escapes or comments (", ', ', \, #, etc.:), If a shell is not required to parse the command line, make exec()'s the command directly.

Line breaks within a rule are significant in that each command line is performed by a separate process or shell.

This means that a rule such as:

```
test:
        cd /tmp
        pwd
```

behaves differently than you might expect, as shown below.

```
$ make test
cd /tmp
pwd
/usr/tutorial/waite/arcana/minor/pentangles
```

You can use semicolons to specify a sequence of commands to perform in a single shell invocation:

```
test:
cd /tmp ; pwd
```

Or, you can continue the input line onto the next line in the makefile by escaping the NEWLINE with a backslash (\). The escaped NEWLINE is treated as white space by make.

The backslash must be the last character on the line. The semicolon is required by the shell.

```
test:
        cd /tmp ; \
        pwd
```

Basic Use of Implicit Rules

When no rule is given for a specified target, `make` attempts to use an *implicit rule* to build it. When `make` finds a rule for the class of files the target belongs to, it applies the rule listed in the implicit rule target entry.

In addition to any `makefile`(s) that you supply, `make` reads in the default `makefile`, `/usr/share/lib/make/make.rules`, which contains the target entries for a number of implicit rules, along with other information.

Note - Implicit rules were hand-coded in earlier versions of `make`.

There are two types of implicit rules: *Suffix* and *Pattern-matching*. Suffix rules specify a set of commands for building a file with one suffix from another file with the same base name but a different suffix. Pattern-matching rules select a rule based on a target and dependency that match respective wild-card patterns. The implicit rules provided by default are suffix rules.

In some cases, the use of suffix rules can eliminate the need for writing a `makefile` entirely. For instance, to build an object file named `functions.o` from a single C source file named `functions.c`, you could use the command:

```
$ make functions.o
cc -c functions.c -o functions.o
```

This would work equally well for building the object file `nonesuch.o` from the source file `nonesuch.c`.

To build an executable file named functions (with a null suffix) from `functions.c`, you need only type the command:

```
$ make functions
cc -o functions functions.c
```

The rule for building a `.o` file from a `.c` file is called the `.c.o` (pronounced "dot-see-dot-oh") suffix rule. The rule for building an executable program from a `.c` file is called the `.c` rule. The complete set of default suffix rules is listed in Table 4–8.

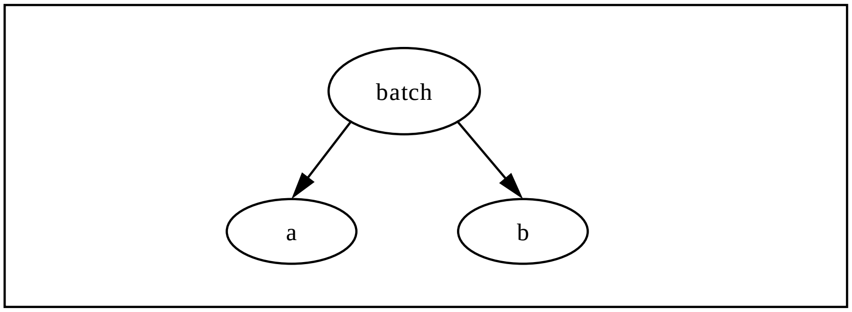
Processing Dependencies

After `make` begins, it processes targets as it encounters them in its depth-first dependency scan. For example, with the following `makefile`:

```
batch: a b
	touch batch
b:
	touch b
a:
	touch a
```

```
c:
     echo "you won't see me"
```

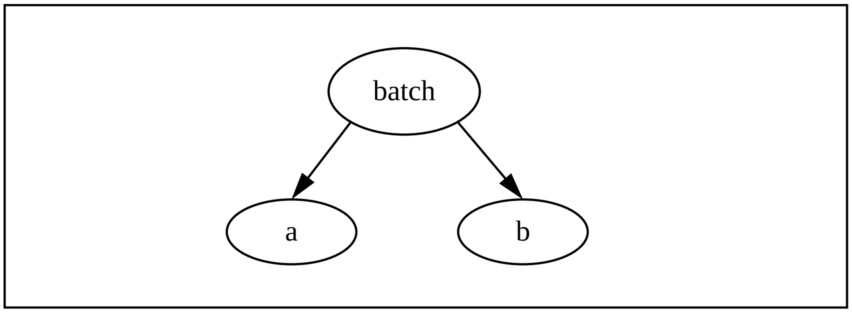
make starts with the target batch. Since batch has some dependencies that have not been checked, namely a and b, make defers batch until after it has checked a and b against any dependencies they might have.



Since a has no dependencies, make processes it; if the file is not present, make performs the rule in its target entry.

```
$ make
touch a
...
```

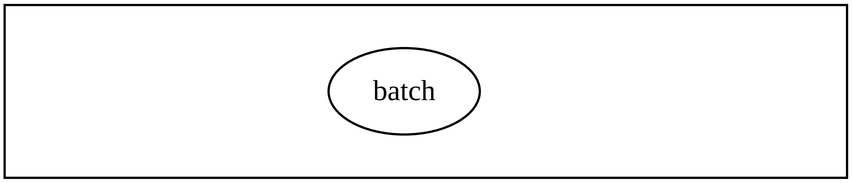
Next, make works its way back up to the parent target batch. Since there is still an unchecked dependency b, make descends to b and checks it.



b also has no dependencies, so make performs its rule:

```
...
touch b
...
```

Finally, now that all of the dependencies for batch have been checked and built (if needed), make checks batch.



Since it rebuilt at least one of the dependencies for `batch`, `make` assumes that `batch` is out of date and rebuilds it; if `a` or `b` had not been built in the current `make` run, but were present in the directory and newer than `batch`, `make`'s time stamp comparison would also result in `batch` being rebuilt:

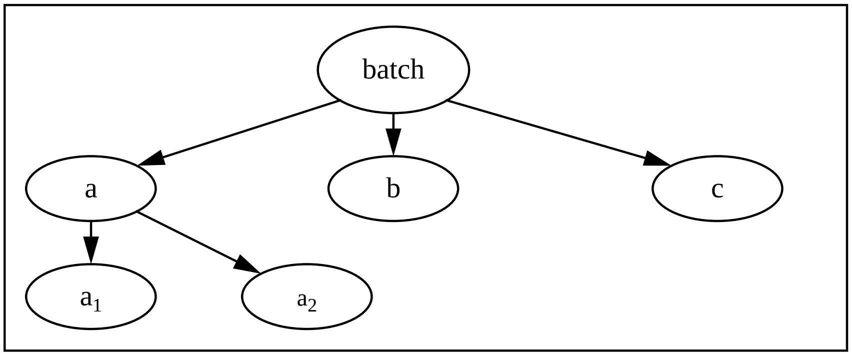
```
...
touch batch
```

Target entries that are not encountered in a dependency scan are not processed. Although there is a target entry for `c` in the `makefile`, `make` does not encounter it while performing the dependency scan for `batch`, so its rule is not performed. You can select an alternate starting target like `c` by entering it as an argument to the `make` command.

In the next example, the `batch` target produces no file. Instead, it is used as a label to group a set of targets.

```
batch: a b c
a: a1 a2
      touch a
b:
      touch b
c:
      touch c
a1:
      touch a1
a2:
      touch a2
```

In this case, the targets are checked and processed, as shown in the following diagram:



Essentially, `make` then:

1. Checks batch for dependencies, notices that there are three, and so defers it.
2. Checks a, the first dependency, and notices that it has two dependencies of its own. Continuing in like fashion, make:
   a. Checks a1, and if necessary, rebuilds it.
   b. Checks a2, and if necessary, rebuilds it.
3. Determines whether to build a.
4. Checks b and rebuilds it if necessary.
5. Checks and rebuilds c if necessary.
6. After traversing its dependency tree, make checks and processes the topmost target, batch. If batch contains a rule, make performs that rule. Here batch has no rule, therefore make performs no action, but notes that batch has been rebuilt; any targets depending on batch would also be rebuilt.

Null Rules

If a target entry contains no rule, make attempts to select an implicit rule to build it. If make cannot find an appropriate implicit rule and there is no SCCS history from which to retrieve it, make concludes that the target has no corresponding file, and regards the missing rule as a null rule.

Note - You can use a dependency with a null rule to force the target rule to be executed. The conventional name for such is FORCE.

With this makefile:

```
haste: FORCE
        echo "haste makes waste"
FORCE:
```

make performs the rule for making haste, even if a file by that name is up to date:

```
$ touch haste
$ make haste
echo "haste makes waste"
haste makes waste
```

Special Targets

make has several built-in *special targets* that perform special functions. For example, the .PRECIOUS special target directs make to preserve library files when make is interrupted.

Special targets:

- Begin with a period (.)
- Have no dependencies
- Can appear anywhere in a makefile

Table 4–3 includes a list of special targets.

Unknown Targets

If a target is named either on the command line or in a dependency list, and it

- Is not a file present in the working directory
- Has no target or dependency entry
- Does not belong to a class of files for which an implicit rule is defined
- Has no SCCS history file, and
- There is no rule specified for the .DEFAULT special target

Then `make` stops processing and issues an error message.

```
$ make believe
make: Fatal error: Don't know how to make target 'believe'.
```

Note - However, if the `–k` option is in effect, `make` continues with the other targets that do not depend on the one in which the error occurred.

Duplicate Targets

Targets can appear more than once in a `makefile`. For example,

```
foo:  dep_1
foo:  dep_2
foo:
     touch foo
```

is the same as

```
foo:  dep_1 dep_2
      touch foo
```

However, many people feel that it's preferable to have a target appear only once, for ease of reading.

Reserved `make` Words

The words in the following table are reserved by `make`:

TABLE 4–3 Reserved make Words

.BUILT_LAST_MAKE_RUN	.DEFAULT	.DERIVED_SRC
.DONE	.IGNORE	.INIT
.KEEP_STATE	.MAKE_VERSION	.NO_PARALLEL
.PRECIOUS	.RECURSIVE	.SCCS_GET
.SILENT	.SUFFIXES	.WAIT
FORCE	HOST_ARCH	HOST_MACH
KEEP_STATE	MAKE	MAKEFLAGS
MFLAGS	TARGET_ARCH	TARGET_MACH
VERSION_1.0	VIRTUAL_ROOT	VPATH

Running Commands Silently

You can inhibit the display of a command line within a rule by inserting an @ as the first character on that line. For example, the following target:

```
quiet:
        @echo you only see me once
```

produces:

```
$ make quiet
         you only see me once
```

If you want to inhibit the display of commands during a particular make run, you can use the –s option. If you want to inhibit the display of all command lines in every run, add the special target .SILENT to your makefile.

```
.SILENT:
quiet:
         echo you only see me once
```

Special-function targets begin with a dot (.). Target names that begin with a dot are never used as the starting target, unless specifically requested as an argument on the command line. make normally issues an error message and stops when a command returns a nonzero exit code. For example, if you have the target:

```
rmxyz:
     rm xyz
```

and there is no file named xyz, make halts after rm returns its exit status.

```
$ ls xyz
xyz not found
$ make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1
make: Fatal error: Command failed for target ‘rmxyz’
```

Note - If - and @ are the first two such characters, both take effect.

To continue processing regardless of the command exit code, use a dash character (-) as the first non-TAB character:

```
rmxyz:
        -rm xyz
```

In this case you get a warning message indicating the exit code make received:

```
$make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1 (ignored)
```

Note - Unless you are testing a makefile, it is usually a bad idea to ignore non-zero error codes on a global basis

---

Although it is generally ill-advised to do so, you can cause make to ignore error codes entirely with the –i option. You can also cause make to ignore exit codes when processing a given makefile, by including the .IGNORE special target, though this too should be avoided.

If you are processing a list of targets, and you want make to continue with the next target on the list rather than stopping entirely after encountering a non-zero return code, use the –k option.

Automatic Retrieval of SCCS Files

When source files are named in the dependency list, make treats them like any other target. Because the source file is presumed to be present in the directory, there is no need to add an entry for it to the makefile.

When a target has no dependencies, but is present in the directory, make assumes that file is up to date. If, however, a source file is under SCCS control, make does some additional checking to ensure that the source file is up to date. If the file is missing, or if the history file is newer, make automatically issues the following command to retrieve the most recent version:

```
sccs get -s filename -Gfilename
```

Note - With other versions of make, automatic sccs retrieval was a feature only of certain implicit rules. Also, unlike earlier versions, make only looks for history (s.) files in the sccs directory; history files in the current working directory are ignored.

However, if the source file is writable by anyone, make does not retrieve a new version.

```
$ ls SCCS/*
SCCS/s.functions.c
$ rm -f functions.c
$ make functions
sccs get -s functions.c -Gfunctions.c
cc -o functions functions.c
```

make checks the time stamp of the retrieved version against the time stamp of the history file. It does *not* check to see if the version present in the directory is the most recently checked-in version. So, if someone had done a *get by date* (sccs get -c), make would not discover this fact, and you might unwittingly build an older version of the program or object file. To be absolutely sure that you are compiling the latest version, you can precede make with an sccs get SCCS' or an sccs clean command.

Suppressing SCCS Retrieval

The command for retrieving SCCS files is specified in the rule for the .SCCS_GET special target in the default makefile. To suppress automatic retrieval, simply add an entry for this target with an empty rule to your makefile:

```
# Suppress sccs retrieval.
        .SCCS_GET:
```

Passing Parameters: Simple make Macros

The make macro substitution comes in handy when you want to pass parameters to command lines within a makefile. Suppose that you want to compile an optimized version of the program program, using the cc –O option. You can lend this sort of flexibility to your makefile by adding a *macro reference*, such as the following example, to the target for functions:

```
functions: functions.c
        cc $(CFLAGS) -o functions functions.c
```

The macro reference acts as a placeholder for a value that you define, either in the `makefile` itself, or as an argument to the `make` command. If you then supply `make` with a *definition* for the CFLAGS macro, `make` replaces its references with the value you have defined.

```
$rm functions
$ make functions "CFLAGS= -O"
cc -O -o functions functions.c
```

Note - There is a reference to the CFLAGS macro in both the `.c` and the `.c.o` implicit rules.

Note - The command-line definition must be a single argument, hence the quotes in this example.

If a macro is undefined, `make` expands its references to an empty string.

You can also include macro definitions in the `makefile` itself. A typical use is to set `CFLAGS` to `-O`, so that `make` produces optimized object code by default:

```
CFLAGS= -O
functions: functions.c
        cc $(CFLAGS) -o functions functions.c
```

A macro definition supplied as a command line argument to `make` overrides other definitions in the `makefile`. Conditionally defined macros are an exception to this.

For instance, to compile functions for debugging with `dbx` or `dbxtool`, you can define the value of `CFLAGS` to be `-g` on the command line:

```
$ rm functions
$ make CFLAGS=-g
cc -g -o functions functions.c
```

To compile a profiling variant for use with `gprof`, supply both `-O` and `-pg` in the value for `CFLAGS`.

A macro reference must include parentheses when the name of the macro is longer than one character. If the macro name is only one character, the parentheses can be omitted. You can use curly braces, { and }, instead of parentheses. For example, ‘`$X`’, ‘`$(X)`’, and ‘`${X}`’ are equivalent.

.KEEP_STATE and Command Dependency Checking

In addition to the normal dependency checking, you can use the special target .KEEP_STATE to activate *command dependency* checking. When activated, make not only checks each target file against its dependency files, it compares each command line in the rule with those it ran the last time the target was built. This information is stored in the .make.state file in the current directory (see “State File” on page 120).

With the makefile:

```
CFLAGS= -O
.KEEP_STATE:

functions: functions.c
         cc -o functions functions.c
```

the following commands work as shown:

```
$ make
cc -O -o functions functions.c
$ make CFLAGS=-g
cc -g -o functions functions.c
$ make "CFLAGS= -O -pg"
cc -O -pg -o functions functions.c
```

This ensures that make compiles a program with the options you want, even if a different variant is present and otherwise up to date.

The first make run with .KEEP_STATE in effect recompiles all targets in order to gather and record the necessary information. The KEEP_STATE variable, when imported from the environment, has the same effect as the .KEEP_STATE target.

Suppressing or Forcing Command Dependency Checking for Selected Lines

To suppress command dependency checking for a given command line, insert a question mark as the first character after the TAB.

Command dependency checking is automatically suppressed for lines containing the dynamic macro $?. This macro stands for the list of dependencies that are newer than the current target, and can be expected to differ between any two make runs.

To force make to perform command dependency checking on a line containing this macro, prefix the command line with a ! character (following the TAB).

State File

When .KEEP_STATE is in effect, make writes out a state file named .make.state, in the current directory. This file lists all targets that have ever been processed while

.KEEP_STATE has been in effect, along with the rules to build them, in makefile format. In order to ensure that this state file is maintained consistently, after you have added .KEEP_STATE to a makefile, it is recommended that you leave it in effect.

Note - Since this target is ignored in earlier versions of make, it does not introduce any compatibility problems. Other versions treat it as a superfluous target that no targets depend on, with an empty rule and no dependencies of its own. Because it starts with a dot, it is not used as the starting target.

.KEEP_STATE and Hidden Dependencies

When a C source file contains #include directives for interpolating headers, the target depends just as much on those headers as it does on the sources that include them. Because such headers might not be listed explicitly as sources in the compilation command line, they are called *hidden dependencies*. When .KEEP_STATE is in effect, make receives a report from the various compilers and compilation preprocessors indicating which hidden dependency files were interpolated for each target.

It adds this information to the dependency list in the state file. In subsequent runs, these additional dependencies are processed just like regular dependencies. This feature automatically maintains the hidden dependency list for each target; it ensures that the dependency list for each target is always accurate and up to date. It also eliminates the need for the complicated schemes found in some earlier makefiles to generate complete dependency lists.

A slight inconvenience can arise the first time make processes a target with hidden dependencies, because there is as yet no record of them in the state file. If a header is missing, and make has no record of it, make does not know that it needs to retrieve it from SCCS before compiling the target.

Even though there is an SCCS history file, the current version won't be retrieved because it doesn't yet appear in a dependency list or the state file. When the C preprocessor attempts to interpolate the header, it won't find it; the compilation fails.

Supposing that a #include directive for interpolating the header hidden.h is added to functions.c, and that the file hidden.h is somehow removed before the subsequent make run. The results would be:

```
$ rm -f hidden.h
$ make functions
cc -O -o functions functions.c
functions.c: 2: Can't find include file hidden.h
make: Fatal error: Command failed for target 'functions'
```

A simple workaround might be to make sure that the new header is extant before you run make. Or, if the compilation should fail (and assuming the header is under SCCS), you could manually retrieve it from SCCS:

```
$ sccs get hidden.h
1.1
10 lines
$ make functions
cc -O -o functions functions.c
```

In all future cases, should the header turn up missing, make will know to build or retrieve it for you because it will be listed in the state file as a hidden dependency.

Note that with hidden dependency checking, the $? macro includes the names of hidden dependency files. This might cause unexpected behavior in existing makefiles that rely on $?.

.INIT and Hidden Dependencies

The problem with both of these approaches is that the first make in the local directory might fail due to a random condition in some other (include) directory. This might entail forcing someone to monitor a (first) build. To avoid this, you can use the .INIT target to retrieve known hidden dependencies files from SCCS. .INIT is a special target that, along with its dependencies, is built at the start of the make run. To be sure that hidden.h is present, you could add the following line to your makefile

```
.INIT:     hidden.h
```

Displaying Information About a make Run

Running make with the –n option displays the commands make is to perform, without executing them. This comes in handy when verifying that the macros in a makefile are expanded as expected. With the following makefile:

```
CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
        $(LINK.c) -o functions main.o data.o
```

make –n displays:

```
$ make -n
cc -O -c main.c
cc -O -c data.c
cc -O -o functions main.o data.o
```

Note - There is an exception however. make executes any command line containing a reference to the MAKE macro (such as $(MAKE) or ${MAKE}), regardless of –n. It is a bad idea to include a line such as the following in your makefile:
$(MAKE) ; rm -f *

Note - Setting an environment variable named MAKEFLAGS can lead to complications, since make adds its value to the list of options. To prevent puzzling surprises, avoid setting this variable.

make has some other options that you can use to keep abreast of what it's doing and why:

–d
: Displays the criteria by which make determines that a target is be out-of-date. Unlike –n, it *does* process targets, as shown in the following example. This option also displays the value imported from the environment (null by default) for the MAKEFLAGS macro, which is described in detail in a later section.

```
$ make -d
MAKEFLAGS value:
    Building main.o using suffix rule for .c.o because
it is out of date relative to main.c
cc -O -c main.c
    Building functions because it is out of date
relative to main.o
    Building data.o using suffix rule for .c.o because
it is out of date relative to data.c
cc -O -c data.c
    Building functions because it is out of date
relative to data.o
cc -O -o functions main.o data.o
```

–dd
: This option displays all dependencies make checks, including any hidden dependencies, in vast detail.

–D
: Displays the text of the makefile as it is read.

–DD
: Displays the makefile and the default makefile, the state file, and hidden dependency reports for the current make run.

–f ***makefile***
: make uses the named *makefile* (instead of *makefile* or *Makefile*).

Note - Several –f options indicate the concatenation of the named makefiles.

–K***makestatefile*** If *makestatefile* is a directory, make writes the KEEP_STATE information into a .make.state file in that directory. If *makestatefile* is a file, make will write the KEEP_STATE information into the *makestatefile*

–p Displays the complete set of macro definitions and target entries.

–P Displays the complete dependency tree for the default target or the specified target.

An option that can be used to shortcut make processing is the –t option. When run with –t, make does not perform the rule for building a target. Instead it uses touch to alter the modification time for each target that it encounters in the dependency scan. It also updates the state file to reflect what it built. This often creates more problems than it solves, and it is recommended that you exercise caution if you do use it. Note that if there is no file corresponding to a target entry, touch creates it.

Note - Due to its potentially troublesome side effects, it is recommended that you not use the –t (touch) option for make.

The following is one example of how *not* to use make –t. Suppose you have a target named clean that performed housekeeping in the directory by removing target files produced by make:

```
clean:
        rm functions main.o data.o
```

Note - clean is the conventional name for a target that removes derived files. It is useful when you want to start a build from scratch

---

If you give the nonsensical command:

```
$ make -t clean
touch clean
$ make clean
'clean' is up to date
```

you then have to remove the file clean before your housekeeping target can work again.

–q Invokes the question mode, and returns a zero or non-zero status code, depending on whether or not the target file is up-to-date.

–r Suppresses reading in of the default makefile, /usr/share/lib/make/make.rules.

–S | Undoes the effect of the –K option by stopping processing when a non-zero exit status is returned by a command.

Using make to Compile Programs

In previous examples you have seen how to compile a simple C program from a single source file, using both explicit target entries and implicit rules. Most C programs, however, are compiled from several source files. Many include library routines, either from one of the standard system libraries or from a user-supplied library.

Although it might be easier to recompile and link a single-source program using a single cc command, it is usually more convenient to compile programs with multiple sources in stages—first, by compiling each source file into a separate object (.o) file, and then by linking the object files to form an executable (a.out) file. This method requires more disk space, but subsequent (repetitive) recompilations need to be performed only on those object files for which the sources have changed, which saves time.

Simple Makefile Example

The following makefile is not elegant, but it does the job.

TABLE 4–4 Simple Makefile for Compiling C Sources: Everything Explicit

```
# Simple makefile for compiling a program from
# two C source files.

.KEEP_STATE

functions: main.o data.o
      cc -O -o functions main.o data.o
main.o: main.c
      cc -O -c main.c
data.o: data.c
      cc -O -c data.c
clean:
      rm functions main.o data.o
```

In this example, make produces the object files main.o and data.o, and the executable file functions:

```
$ make
cc -o functions main.o data.o
cc -O -c main.c
cc -O -c data.c
```

Using make's Predefined Macros

The next example performs exactly the same function, but demonstrates the use of make's predefined macros for the indicated compilation commands. Using predefined macros eliminates the need to edit makefiles when the underlying compilation environment changes. Macros also provide access to the CFLAGS macro (and other FLAGS macros) for supplying compiler options from the command line. Predefined macros are also used extensively within make's implicit rules. The predefined macros in the following makefile are listed below.[1] They are generally useful for compiling C programs.

COMPILE.C

The cc command line; composed of the values of CC, CFLAGS, and CPPFLAGS, as follows, along with the –c option.

```
COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) -c
```

The root of the macro name, COMPILE, is a convention used to indicate that the macro stands for a compilation command line (to generate an object, or .o file). The .c suffix is a mnemonic device to indicate that the command line applies to .c (C source) files.

Note - Macro names that end in the string FLAGS pass options to a related compiler-command macro. It is good practice to use these macros for consistency and portability. It is also good practice to note the desired default values for them in the makefile. The complete list of all predefined macros is shown in Table 4–9.

LINK.c

The basic cc command line to link object files, such as COMPILE.c, but without the –c option and with a reference to the LDFLAGS macro:

```
LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)
```

CC

The value cc. (You can redefine the value to be the path name of an alternate C compiler.)

CFLAGS

Options for the cc command; none by default.

1. Predefined macros are used more extensively than in earlier versions of make. Not all of the predefined macros shown here are available with earlier versions.

CPPFLAGS	Options for `cpp`; none by default.
LDFLAGS	Options for the link editor, `ld`; none by default.

TABLE 4–5 Makefile for Compiling C Sources Using Predefined Macros

```
# Makefile for compiling two C sources
CFLAGS= -O
.KEEP_STATE:

functions: main.o data.o
	$(LINK.c) -o functions main.o data.o
main.o: main.c
	$(COMPILE.c) main.c
data.o: data.c
	$(COMPILE.c) data.c
clean:
	rm functions main.o data.o
```

Using Implicit Rules to Simplify a Makefile: Suffix Rules

Since the command lines for compiling `main.o` and `data.o` from their `.c` files are now functionally equivalent to the `.c.o` suffix rule, their target entries are redundant; `make` performs the same compilation whether they appear in the `makefile` or not. This next version of the `makefile` eliminates them, relying on the `.c.o` rule to compile the individual object files.

TABLE 4–6 Makefile for Compiling C Sources Using Suffix Rules

```
# Makefile for a program from two C sources
# using suffix rules.
CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
	$(LINK.c) -o functions main.o data.o
clean:
	rm functions main.o data.o
```

Note - A complete list of suffix rules appears in Table 4–8.

As make processes the dependencies main.o and data.o, it finds no target entries for them. It checks for an appropriate implicit rule to apply. In this case, make selects the .c.o rule for building a .o file from a dependency file that has the same base name and a .c suffix.

Note - make uses the order of appearance in the suffixes list to determine which dependency file and suffix rule to use. For instance, if there are both main.c and main.s files in the directory, make uses the .c.o rule, since .c is ahead of .s in the list.

First, make scans its suffixes list to see if the suffix for the target file appears. In the case of main.o, .o appears in the list. Next, make checks for a suffix rule to build it with, and a dependency file to build it from. The dependency file has the same base name as the target, but a different suffix. In this case, while checking the .c.o rule, make finds a dependency file named main.c, so it uses that rule.

The suffixes list is a special-function target named .SUFFIXES. The various suffixes are included in the definition for the SUFFIXES macro; the dependency list for .SUFFIXES is given as a reference to this macro:

TABLE 4–7 The Standard Suffixes List

```
SUFFIXES= .o .c .c~ .cc .cc~ .C .C~ .y .y~ .l .l~ .s .s~ .sh .sh~ .S .S~ .ln
\ .h .h~ .f .f~ .F .F~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ \ .cps
.cps~ .Y .Y~ .L .L~ .SUFFIXES: $(SUFFIXES)
```

The following example shows a makefile for compiling a whole set of executable programs, each having just one source file. Each executable is to be built from a source file that has the same basename, and the .c suffix appended. For instance demo_1 is built from demo_1.c.

Note - Like clean, all is a target name used by convention. It builds "all" the targets in its dependency list. Normally, make and make all are usually equivalent.

```
# Makefile for a set of C programs, one source
# per program.  The source file names have ".c"
# appended.
CFLAGS= -O
.KEEP_STATE:

all: demo_1 demo_2 demo_3 demo_4 demo_5
```

In this case, make does not find a suffix match for any of the targets (through demo_5). So, it treats each as if it had a null suffix. It then searches for a suffix rule and dependency file with a valid suffix. In the case of demo_2, it would find a file named demo_2.c. Since there is a target entry for a .c rule, along with a corresponding .c file, make uses that rule to build demo_2 from demo_2.c.

To prevent ambiguity when a target with a null suffix has an explicit dependency, make does not build it using a suffix rule. This makefile

```
program: zap
zap:
```

produces no output:

```
$ make program
$
```

When to Use Explicit Target Entries vs. Implicit Rules

Whenever you build a target from multiple dependency files, you must provide make with an explicit target entry that contains a rule for doing so. When building a target from a single dependency file, it is often convenient to use an implicit rule.

As the previous examples show, make readily compiles a single source file into a corresponding object file or executable. However, it has no built-in knowledge about how to link a list of object files into an executable program. Also, make compiles only those object files that it encounters in its dependency scan. It needs a starting point—a target for which each object file in the list (and ultimately, each source file) is a dependency.

So, for a target built from multiple dependency files, make needs an explicit rule that provides a collating order, along with a dependency list that accounts for its dependency files. If each of those dependency files is built from just one source, you can rely on implicit rules for them.

Implicit Rules and Dynamic Macros

make maintains a set of macros dynamically, on a target-by-target basis. These macros are used quite extensively, especially in the definitions of implicit rules. It is important to understand what they mean.

Note - Because they are not explicitly defined in a makefile, the convention is to document dynamic macros with the $-sign prefix attached (in other words, by showing the macro reference).

They are:

$@	The name of the current target.
$?	The list of dependencies newer than the target.

`$<`	The name of the dependency file, as if selected by `make` for use with an implicit rule.
`$*`	The base name of the current target (the target name stripped of its suffix).
`$%`	For libraries, the name of the member being processed. See “Building Object Libraries ” on page 139 for more information.

Implicit rules use these dynamic macros in order to supply the name of a target or dependency file to a command line within the rule itself. For instance, in the `.c.o` rule, shown in the next example:

```
.c.o:
        $(COMPILE.c) $< $(OUTPUT_OPTION)
```

`$<` is replaced by the name of the dependency file (in this case the `.c` file) for the current target.

Note - The macro `OUTPUT_OPTION` has an empty value by default. While similar to `CFLAGS` in function, it is provided as a separate macro intended for passing an argument to the `–o` compiler option to force compiler output to a given file name.

In the `.c` rule:

```
.c:
        $(LINK.c) $< -o $@
```

$@ is replaced with the name of the current target.

Because values for both the `$<` and `$*` macros depend upon the order of suffixes in the suffixes list, you might get surprising results when you use them in an explicit target entry. See “Suffix Replacement in Macro References ” on page 142 for a strictly deterministic method for deriving a file name from a related file name.

Dynamic Macro Modifiers

Dynamic macros can be modified by including `F` and `D` in the reference. If the target being processed is in the form of a pathname, `$(@F)` indicates the file name part, while `$(@D)` indicates the directory part. If there are no `/` characters in the target name, then `$(@D)` is assigned the dot character (`.`) as its value. For example, with the target named `/tmp/test`, `$(@D)` has the value `/tmp`; `$(@F)` has the value `test`.

Dynamic Macros and the Dependency List: Delayed Macro References

Dynamic macros are assigned while processing any and all targets. They can be used within the target rule as is, or in the dependency list by prepending an additional $ character to the reference. A reference beginning with $$ is called a *delayed* reference to a macro. For instance, the entry:

```
x.o y.o z.o: $$@.BAK
         cp $@.BAK $@
```

could be used to derive `x.o` from `x.o.BAK`, and so forth for `y.o` and `z.o`.

Dependency List Read Twice

This technique works because `make` reads the dependency list twice, once as part of its initial reading of the entire `makefile`, and again as it processes target dependencies. In each pass through the list, it performs macro expansion. Since the dynamic macros aren't defined in the initial reading, unless references to them are delayed until the second pass, they are expanded to null strings.

The string `$$` is a reference to the predefined macro '`$`'. This macro, conveniently enough, has the value '`$`'; when `make` resolves it in the initial reading, the string `$$@` is resolved to `$@`. In dependency scan, when the resulting `$@` macro reference has a value dynamically assigned to it, `make` resolves the reference to that value.

Notice that `make` only evaluates the target-name portion of a target entry in the first pass. A delayed macro reference as a target name produces incorrect results. The `makefile`:

```
NONE= none
all: $(NONE)

$$(NONE):
         @: this target's name isn't 'none'
```

produces the following results.

```
$ make
make: Fatal error: Don't know how to make target 'none'
```

Rules Evaluated Once

`make` evaluates the rule portion of a target entry only once per application of that command, at the time that the rule is executed. Here again, a delayed reference to a `make` macro produces incorrect results.

No Transitive Closure for Suffix Rules

There is no transitive closure for suffix rules. If you had a suffix rule for building, say, a .Y file from a .X file, and another for building a .Z file from a .Y file, make would not combine their rules to build a .Z file from a .X file. You must specify the intermediate steps as targets, although their entries can have null rules:

```
trans.Z:
trans.Y:
```

In this example trans.Z is built from trans.Y if it exists. Without the appearance of trans.Y as a target entry, make might fail with a "don't know how to build" error, since there would be no dependency file to use. The target entry for trans.Y guarantees that make will attempt to build it when it is out of date or missing. Since no rule is supplied in the makefile, make will use the appropriate implicit rule, which in this case would be the .X.Y rule. If trans.X exists (or can be retrieved from SCCS), make rebuilds both trans.Y and trans.Z as needed.

Adding Suffix Rules

Although make supplies you with a number of useful suffix rules, you can also add new ones of your own. However, pattern-matching rules are to be preferred when adding new implicit rules (see "Pattern-Matching Rules: An Alternative to Suffix Rules " on page 133. Unless you need to write implicit rules that are compatible with earlier versions of make, you can skip the remainder of this section, which describes the traditional method of adding implicit rules to makefiles. (The procedure for adding implicit rules is given here for compatibility with previous versions of make.)

Adding a *suffix rule* is a two-step process. First, you must add the suffixes of both target and dependency file to the suffixes list by providing them as dependencies to the .SUFFIXES special target. Because dependency lists accumulate, you can add suffixes to the list by adding another entry for this target, for example:

```
.SUFFIXES:  .ms .tr
```

Second, you must add a target entry for the suffix rule:

```
ms.tr:
        troff -t -ms $< > $@
```

A makefile with these entries can be used to format document source files containing ms macros (.ms files) into troff output files (.tr files):

```
$ make doc.tr
troff -t -ms doc.ms > doc.tr
```

Entries in the suffixes list are contained in the SUFFIXES macro. To insert suffixes at the head of the list, first clear its value by supplying an entry for the .SUFFIXES target that has no dependencies. This is an exception to the rule that dependency lists accumulate. You can clear a previous definition for this target by supplying a target entry with no dependencies and no rule like this:

```
.SUFFIXES:
```

You can then add another entry containing the new suffixes, followed by a reference to the SUFFIXES macro, as shown below.

```
.SUFFIXES:
.SUFFIXES: .ms .tr $(SUFFIXES)
```

Pattern-Matching Rules: An Alternative to Suffix Rules

A *pattern-matching rule* is similar to an implicit rule in function. Pattern-matching rules are easier to write, and more powerful, because you can specify a relationship between a target and a dependency based on prefixes (including path names) and suffixes, or both. A pattern-matching rule is a target entry of the form:

```
tp%ts: dp%ds
     rule
```

where `tp` and `ts` are the optional prefix and suffix in the target name, `dp` and `ds` are the (optional) prefix and suffix in the dependency name, and `%` is a wild card that stands for a base name common to both.

Note - `make` checks for pattern-matching rules ahead of suffix rules. While this allows you to override the standard implicit rules, it is not recommended.

If there is no rule for building a target, `make` searches for a pattern-matching rule, *before* checking for a suffix rule. If `make` can use a pattern-matching rule, it does so.

If the target entry for a pattern-matching rule contains no rule, `make` processes the target file as if it had an explicit target entry with no rule; `make` therefore searches for a suffix rule, attempts to retrieve a version of the target file from SCCS, and finally, treats the target as having a null rule (flagging that target as updated in the current run).

A pattern-matching rule for formatting a `troff` source file into a `troff` output file looks like:

```
%.tr: %.ms
     troff -t -ms $< > $@
```

make's Default Suffix Rules and Predefined Macros

The following tables show the standard set of suffix rules and predefined macros supplied to `make` in the default `makefile`, `/usr/share/lib/make/make.rules`.

TABLE 4–8 Standard Suffix Rules

Use	Suffix Rule Name	Command Line(s)
Assembly Files	`.s.o`	`$(COMPILE.s) -o $@ $<`
	`.s`	`$(COMPILE.s) -o $@ $<`
		`$(COMPILE.s) -o $% $<`
	`.s.a`	`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
	`.S.o`	`$(COMPILE.S) -o $@ $<`
		`$(COMPILE.S) -o $% $`
	`.S.a`	`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
	`.c`	`$(LINK.c) -o $@ $< $(LDLIBS)`
	`.c.ln`	`$(LINT.c) $(OUTPUT_OPTION) -i $<`
	`.c.o`	`$(COMPILE.c) $(OUTPUT_OPTION) $<`
C Files (.c Rules)	`.c.a`	`$(COMPILE.c) -o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
	`.cc`	`$(LINK.cc) -o $@ $< $(LDLIBS)`
	`.cc.o`	`$(COMPILE.cc) $(OUTPUT_OPTION) $<`
C++ Files	`.cc.a`	`$(COMPILE.cc) -o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
	`.C`	`$(LINK.C) -o $@ $< $(LDFLAGS) $*.c`
	`.C.o`	`$(COMPILE.C) $<`
C++ Files (SVr4 style)	`.C.a`	`$(COMPILE.C) $<`
		`$(AR) $(ARFLAGS) $@ $*.o`
		`$(RM) -f $*.o`

TABLE 4–8 Standard Suffix Rules *(continued)*

Use	Suffix Rule Name	Command Line(s)
FORTRAN 77 Files	.cc.o	$(LINK.f) -o $@ $< $(LDLIBS)
	.cc.a	$(COMPILE.f) $(OUTPUT_OPTION) $<
		$(COMPILE.f) -o $% $<
		$(AR) $(ARFLAGS) $@ $%
		$(RM) $%
	.F	$(LINK.F) -o $@ $< $(LDLIBS)
	.F.o	$(COMPILE.F) $(OUTPUT_OPTION) $<
	.F.a	$(COMPILE.F) -o $% $<
		$(AR) $(ARFLAGS) $@ $%
		$(RM) $%
lex Files	.l	$(RM) $*.c
		$(LEX.l) $< > $*.c
		$(LINK.c) -o $@ $*.c $(LDLIBS)
		$(RM) $*.c
	.l.c	$(RM) $@
		$(LEX.l) $< > $@
	.l.ln	$(RM) $*.c
		$(LEX.l) $< > $*.c
		$(LINT.c) -o $@ -i $*.c
		$(RM) $*.c
	.l.o	$(RM) $*.c
		$(LEX.l) $< > $*.c
		$(COMPILE.c) -o $@ $*.c
		$(RM) $*.c
	.L.C	$(LEX) $(LFLAGS) $<
	.L.o	$(LEX)(LFLAGS) $<
		$(COMPILE.C) lex.yy.c
	.L.o	rm -f lex.yy.c
		mv lex.yy.o $@

TABLE 4–8 Standard Suffix Rules *(continued)*

Use	Suffix Rule Name	Command Line(s)
	`.mod`	`$(COMPILE.mod) -o $@ -e $@ $<`
Modula 2 Files	`.mod.o`	`$(COMPILE.mod) -o $@ $<`
	`.def.sym`	`$(COMPILE.def) -o $@ $<`
NeWS	`.cps.h`	`$(CPS) $(CPSFLAGS) $*.cps`
Pascal Files	`.p`	`$(LINK.p) -o $@ $< $(LDLIBS)`
	`.p.o`	`$(COMPILE.p) $(OUTPUT_OPTION) $<`
	`.r`	`$(LINK.r) -o $@ $< $(LDLIBS)`
	`.r.o`	`$(COMPILE.r) $(OUTPUT_OPTION) $<`
Ratfor Files	`.r.a`	`$(COMPILE.r) -o $% $<`
		`$(AR) $(ARFLAGS) $@ $%`
		`$(RM) $%`
	`.sh`	`$(RM) $@`
Shell Scripts		`cat $< >$@`
		`chmod +x $@`
	`.y`	`$(YACC.y) $<`
		`$(LINK.c) -o $@ y.tab.c $(LDLIBS)`
		`$(RM) y.tab.c`
	`.y.c`	`$(YACC.y) $<`
		`mv y.tab.c $@`
yacc Files (.yc Rules)	`.y.ln`	`$(YACC.y) $<`
		`$(LINT.c) -o $@ -i y.tab.c`
		`$(RM) y.tab.c`
	`.y.o`	`$(YACC.y) $<`
		`$(COMPILE.c) -o $@ y.tab.c`
		`$(RM) y.tab.c`

TABLE 4–8 Standard Suffix Rules *(continued)*

Use	Suffix Rule Name	Command Line(s)
yacc Files (SVr4)	.Y.C	$(YACC) $(YFLAGS) $< mv y.tab.c $@
	.Y.o	$(YACC) $(YFLAGS) $< $(COMPILE.c) y.tab.c rm -f y.tab.c mv y.tab.o $@

TABLE 4–9 Predefined and Dynamic Macros

Use	Macro	Default Value
Library Archive	AR	ar
	ARFLAGS	rv
Assembler Commands	AS	as
	ASFLAGS	
	COMPILE.s	$(AS) $(ASFLAGS)
	COMPILE.S	$(CC) $(ASFLAGS) $(CPPFLAGS) -target -c
C Compiler Commands	CC	cc
	CFLAGS	
	CPPFLAGS	
	COMPILE.c	$(CC) $(CFLAGS) $(CPPFLAGS) -c
	LINK.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)
C++ Compiler Commands[1]	CCC	CC
	CCFLAGS	
	COMPILE.cc	$(CCC) $(CCFLAGS) $(CPPFLAGS) -c
	LINK.cc	$(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)

TABLE 4–9 Predefined and Dynamic Macros *(continued)*

Use	Macro	Default Value
C++ SVr4 Compiler Commands	(C++C)	CC
	(C++FLAGS)	-O
	COMPILE.C	$(C++C) $(C++FLAGS) $(CPPFLAGS) -c
	LINK.C	$(C++C) $(C++FLAGS) $(CPPFLAGS) $(LDFLAGS) -target
FORTRAN 77 Compiler Commands	FC in SVr4	f77
	FFLAGS	
	COMPILE.f	$(FC) $(FFLAGS) -c
	LINK.f	$(FC) $(FFLAGS) $(LDFLAGS)
	COMPILE.F	$(FC) $(FFLAGS) $(CPPFLAGS) -c
	LINK.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(LDFLAGS)
Link Editor Command	LD	ld
	LDFLAGS	
lex Command	LEX	lex
	LFLAGS	
	LEX.l	$(LEX) $(LFLAGS) -t
lint Command	LINT	lint
	LINTFLAGS	
	LINT.c	$(LINT) $(LINTFLAGS) $(CPPFLAGS)
Modula 2 Commands	M2C	m2c
	M2FLAGS	
	MODFLAGS	
	DEFFLAGS	
	COMPILE.def	$(M2C) $(M2FLAGS) $(DEFFLAGS)
	COMPILE.mod	$(M2C) $(M2FLAGS) $(MODFLAGS)
NeWS	CPS	cps
	CPSFLAGS	
Pascal Compiler Commands	PC	pc
	PFLAGS	
	COMPILE.p	$(PC) $(PFLAGS) $(CPPFLAGS) -c
	LINK.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(LDFLAGS)

TABLE 4–9 Predefined and Dynamic Macros *(continued)*

Use	Macro	Default Value
Ratfor Compilation Commands	RFLAGS	
	COMPILE.r	$(FC) $(FFLAGS) $(RFLAGS) -c
	LINK.r	$(FC) $(FFLAGS) $(RFLAGS) $(LDFLAGS)
rm Command	RM	rm -f
yacc Command	YACC	yacc
	YFLAGS	
	YACC.y	$(YACC) $(YFLAGS)
Suffixes List	SUFFIXES	.o .c .c~ .cc .cc~ .C .C~ .y .y~ .l .l~ .s .s~ .sh .sh~ .S .S~ .ln .h .h~ .f .f~ .F .F~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .cps .cps~ .Y .Y~ .L .L~
SCCS get Command	.SCCS_GET	sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@ -G$@
	SCCSGETFLAGS	-s

1. For backward compatibility, the C++ macros have alternate forms. For C++C, you can instead use CCC; instead of C++FLAGS, you can use CCFLAGS; for COMPILE.C, you can use COMPILE.cc; and LINK.cc can be substituted for LINK.C. Note that these alternate forms will disappear for future releases.

Building Object Libraries

Libraries, Members, and Symbols

An object library is a set of object files contained in an ar library archive (see ar(1) and lorder(1) in the *SunOS reference Manual* for details about library archive files.) Various languages make use of object libraries to store compiled functions of general utility, such as those in the C library.

ar reads in a set of one or more files to create a library. Each member contains the text of one file, preceded by a header. The member header contains information from the file directory entry, including the modification time. This allows make to treat the library member as a separate entity for dependency checking.

When you compile a program that uses functions from an object library (specifying the proper library either by filename, or with the –l option to cc), the link editor selects and links with the library member that contains a needed symbol.

You can use ar to generate a symbol table for a library of object files. ld requires this table in order to provide random access to symbols within the library—to locate and link object files in which functions are defined. You can also use lorder and tsort ahead of time to put members in calling order within the library. (See ar(1) and lorder(1) for details.) For very large libraries, it is a good idea to do both.

Library Members and Dependency Checking

make recognizes a target or dependency of the form:

lib.a(member . . .)

as a reference to a library member, or a space-separated list of members.

Note - Earlier versions of make recognize this notation. However, only the first item in a parenthesized list of members is processed.

In this version of make, all members in a parenthesized list are processed. For example, the following target entry indicates that the library named librpn.a is built from members named stacks.o and fifos.o. The pattern-matching rule indicates that each member depends on a corresponding object file, and that object file is built from its corresponding source file using an implicit rule.

```
librpn.a:   librpn.a (stacks.o fifos.o)
             ar rv $@ $?

             $@
librpn.a (%.o): %.o
             @true
```

When used with library-member notation, the dynamic macro $? contains the list of files that are newer than their corresponding members:

```
$ make
cc -c stacks.c
cc -c fifos.c
ar rv librpn.a stacks.o fifos.o
a - stacks.o
a - fifos.o
```

Libraries and the $% Dynamic Macro

The $% dynamic macro is provided specifically for use with libraries. When a library member is the target, the member name is assigned to the $% macro. For instance, given the target libx.a(demo.o) the value of $% would be demo.o.

.PRECIOUS: Preserving Libraries Against Removal due to Interrupts

Normally, if you interrupt make in the middle of a target, the target file is removed. For individual files this is a good thing, otherwise incomplete files with brand new modification times might be left in the directory. For libraries that consist of several members, the story is different. It is often better to leave the library intact, even if one of the members is still out-of-date. This is especially true for large libraries, especially since a subsequent make run picks up where the previous one left off—by processing the object file or member whose processing was interrupted.

.PRECIOUS is a special target that is used to indicate which files should be preserved against removal on interrupts; make does not remove targets that are listed as its dependencies. If you add the line:

```
.PRECIOUS:  librpn.a
```

to the makefile shown above, run make, and interrupt the processing of librpn.a, the library is preserved.

Using make to Maintain Libraries and Programs

In previous sections you learned how make can help compile simple programs and build simple libraries. This section describes some of make's more advanced features for maintaining complex programs and libraries.

More about Macros

Macro definitions can appear on any line in a makefile; they can be used to abbreviate long target lists or expressions, or as shorthand to replace long strings that would otherwise have to be repeated.

You can even use macros to derive lists of object files from a list of source files. Macro names are allocated as the makefile is read in; the value a particular macro reference takes depends upon the most recent value assigned.

Note - Macro evaluation is more complicated than described here. Refer to ...

With the exception of conditional and dynamic macros, make assigns values in the order the definitions appear.

Embedded Macro References

Macro references can be embedded within other references (not supported in previous versions of make)..

```
$(CPPFLAGS$(TARGET_ARCH))
```

Note - The += assignment appends the indicated string to any previous value for the macro.

In which case they are expanded from innermost to outermost. With the following definitions, make will supply the correct symbol definition for (for example) a Sun-4 system.

```
CPPFLAGS-sun4 = -DSUN4
CPPFLAGS += $(CPPFLAGS-$(TARGET_ARCH))
```

Suffix Replacement in Macro References

make provides a mechanism for replacing suffixes of words that occur in the value of the referred-to macro. Although conventional suffixes start with dots, a suffix can consist of any string of characters. A reference of the form:

```
$(macro:old-suffix=new-suffix)
```

is a *suffix replacement* macro reference. You can use such a reference to express the list of object files in terms of the list of sources:

```
OBJECTS= $(SOURCES:.c=.o)
```

In this case, make replaces all occurrences of the .c suffix in words within the value with the .o suffix. The substitution is not applied to words that do not end in the suffix given. The following makefile:

```
SOURCES= main.c data.c moon
OBJECTS= $(SOURCES:.c=.o)

all:
        @echo $(OBJECTS)
```

offers a simple illustration:

```
$ make
main.o data.o moon
```

Using lint with make

For easier debugging and maintenance of your C programs use the lint tool. lint also checks for C constructs that are not considered portable across machine architectures. It can be a real help in writing portable C programs.

lint, the C program verifier, is an important tool for forestalling the kinds of bugs that are most difficult and tedious to track down. These include uninitialized pointers, parameter-count mismatches in function calls, and non-portable uses of C constructs. As with the clean target, lint is a target name used by convention; it is usually a good practice to include it in makefiles that build C programs. lint produces output files that have been preprocessed through cpp and its own first (parsing) pass. These files characteristically end in the .ln suffix and can also be derived from the list of sources through suffix replacement (this might not be true for older versions of lint):

```
LINTFILES= $(SOURCES:.c=.ln)
```

A target entry for the lint target might appear as:

```
lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)
$(LINTFILES):
        $(LINT.c) $@ -i
```

There is an implicit rule for building each .ln file from its corresponding .c file, so there is no need for target entries for the .ln files. As sources change, the .ln files are updated whenever you run

```
make lint
```

Since the LINT.c predefined macro includes a reference to the LINTFLAGS macro, it is a good idea to specify the lint options to use by default (none in this case). Since lint entails the use of cpp, it is a good idea to use CPPFLAGS, rather than CFLAGS for compilation preprocessing options (such as –I). The LINT.c macro does not include a reference to CFLAGS.

Also, when you run make clean, you will want to get rid of any .ln files produced by this target. It is not difficult to add another such macro reference to a clean target.

Linking with System-Supplied Libraries

The next example shows a makefile that compiles a program that uses the curses and termlib library packages for screen-oriented cursor motion.

TABLE 4–10 Makefile for a C Program with System-Supplied Libraries

```
# Makefile for a C program with curses
# and termlib.

CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o
      $(LINK.c) -o $@ main.o data.o -lcurses
      -ltermlib
lint: main.ln data.ln
      $(LINT.c) main.ln data.ln
main.ln data.ln:
      $(LINT.c) $@ -i
clean:
      rm -f functions main.o data.o main.ln \
      data.ln
```

Because the link editor resolves undefined symbols as they are encountered, it is normally a good idea to place library references at the end of the list of files to link.

This `makefile` produces:

```
$ make
cc -O -c main.c
cc -O -c data.c
cc -O -o functions main.o data.o -lcurses -ltermlib
```

Compiling Programs for Debugging and Profiling

Compiling programs for debugging or profiling introduces a new twist to the procedure and to the `makefile`. These variants are produced from the same source code, but are built with different options to the C compiler. Use the `cc` `–g` option to produce object code that is suitable for debugging. The `cc` options that produce code for profiling are `–O` and `–pg`.

Because the compilation procedure is the same otherwise, you *could* give `make` a definition for `CFLAGS` on the command line. Since this definition overrides the definition in the `makefile`, and `.KEEP_STATE` assures any command lines affected by the change are performed, the following `make` command produces the results as presented in this example:

```
$ make "CFLAGS= -O -pg"
cc -O -pg -c main.c
cc -O -pg -c data.c
cc -O -pg -o functions main.o data.o -lcurses -ltermlib
```

Of course, you might not want to memorize these options or type a complicated command like this, especially when you can put this information in the `makefile`. What is needed is a way to tell `make` how to produce a debugging or profiling variant, and some instructions in the `makefile` that tell it how. One way to do this might be to add two new target entries, one named `debug`, and the other named `profile`, with the proper compiler options hard-coded into the command line.

A better way would be to add these targets, but rather than hard-coding their rules, include instructions to alter the definition of `CFLAGS`, depending upon which target it starts with. Then, by requiring each one to depend on the existing target for functions, `make` could use its rule, along with the specified options.

Instead of saying:

```
make "CFLAGS= -g"
```

to compile a variant for debugging, you could say:

```
make debug
```

The question is, how do you tell `make` that you want a macro defined one way for one target (and its dependencies), and another way for a different target?

Conditional Macro Definitions

A conditional macro definition is a line of the form:

```
target-list := macro = value
```

which assigns the given *value* to the indicated *macro* while `make` is processing the target named *target-list* and its dependencies.

Note - Each word in *target-list* can contain one `%` pattern; `make` must know which targets the definition applies to, so you cannot use a conditional macro definition to alter a target name.

The following lines give `CFLAGS` an appropriate value for processing each program variant.

```
debug := CFLAGS= -g
profile := CFLAGS= -pg -O
```

Notice that when you use a reference to a conditional macro in the dependency list, that reference must be delayed (by prepending a second `$`). Otherwise, `make` expands the reference before the correct value has been assigned. When it encounters a (possibly) incorrect reference of this sort, `make` issues a warning.

Compiling Debugging and Profiling Variants

The following `makefile` produces optimized, debugging, or profiling variants of a C program, depending on which target you specify (the default is the optimized variant). Command dependency checking guarantees that the program and its object files will be recompiled whenever you switch between variants.

TABLE 4–11 Makefile for a C Program with Alternate Debugging and Profiling Variants

```
# Makefile for a C program with alternate
# debugging and profiling variants.

CFLAGS= -O

.KEEP_STATE:

all debug profile: functions

debug := CFLAGS = -g
profile := CFLAGS = -pg -O

functions: main.o data.o
	$(LINK.c) -o $@ main.o data.o -lcurses
	-ltermlib
lint: main.ln data.ln
	$(LINT.c) main.ln data.ln
clean:
	rm -f functions main.o data.o main.ln data.ln
```

The first target entry specifies three targets, starting with `all`.

Note - Debugging and profiling variants are not normally considered part of a finished program.

`all` traditionally appears as the first target in `makefiles` with alternate starting targets (or those that process a list of targets). Its dependencies are "all" targets that go into the final build, whatever that might be. In this case, the final variant is optimized. The target entry also indicates that `debug` and `profile` depend on functions (the value of `$(PROGRAM)`).

The next two lines contain conditional macro definitions for `CFLAGS`.

Next comes the target entry for `functions`. When `functions` is a dependency for `debug`, it is compiled with the `–g` option.

The next example applies a similar technique to maintaining a C object library.

TABLE 4–12 Makefile for a C Library with Alternate Variants

```
# Makefile for a C library with alternate
# variants.

CFLAGS= -O

.KEEP_STATE
.PRECIOUS:  libpkg.a

all debug profile: libpkg.a
debug := CFLAGS= -g
profile := CFLAGS= -pg -O

libpkg.a: libpkg.a(calc.o map.o draw.o)
        ar rv $@ $?
        libpkg.a(%.o): %.o
        @true
lint: calc.ln map.ln draw.ln
        $(LINT.c) calc.ln map.ln draw.ln
clean:
        rm -f libpkg.a calc.o map.o draw.o \
        calc.ln map.ln draw.ln
```

Maintaining Separate Program and Library Variants

The previous two examples are adequate when development, debugging, and profiling are performed in distinct phases. However, they suffer from the drawback that all object files are recompiled whenever you switch between variants, which can result in unnecessary delays. The next two examples illustrate how all three variants can be maintained as separate entities.

To avoid the confusion that might result from having three variants of each object file in the same directory, you can place the debugging and profiling object files and executables in subdirectories. However, this requires a technique for adding the name of the subdirectory as a prefix to each entry in the list of object files.

Pattern-Replacement Macro References

A pattern-replacement macro reference is similar in form and function to a suffix replacement reference. You can use a pattern-replacement reference to add or alter a prefix, suffix, or both, to matching words in the value of a macro.

Note - As with pattern-matching rules, pattern-replacement macro references are not available in earlier versions of `make`.

A pattern-replacement reference takes the form:

```
$(macro:p%s =np%ns)
```

where *p* is the existing prefix to replace (if any), *s* is the existing suffix to replace (if any), *np* and *ns* are the new prefix and new suffix, and `%` is a wild card. The pattern replacement is applied to all words in the value that match '*p%s*. For instance:

```
SOURCES= old_main.c old_data.c moon
OBJECTS= $(SOURCES:old_%.c=new_%.o)
all:
        @echo $(OBJECTS)
```

produces:

```
$ make
new_main.o new_data.o moon
```

You can use any number of `%` wild cards in the right-hand (replacement) side of the = sign, as needed. The following replacement:

```
...
OBJECTS= $(SOURCES:old_%.c=%/%.o)
```

would produce:

```
main/main.o data/data.o moon
```

Note, however, that pattern-replacement macro references should not appear in the dependency line of the target entry for a pattern-matching rule. This produces a conflict, since `make` cannot tell whether the wild card applies to the macro, or to the target (or dependency) itself. With the `makefile`:

```
OBJECT= .o

x:
x.Z:
        @echo correct
%: %$(OBJECT:%o=%Z)
```

it seems as if `make` should attempt to build `x` from `x.Z`. However, the pattern-matching rule is not recognized; `make` cannot determine which of the `%` characters in the dependency line to use in the pattern-matching rule.

Makefile for a Program with Separate Variants

The following example shows a `makefile` for a C program with separately maintained variants. First, the `.INIT` special target creates the `debug_dir` and profile_dir subdirectories (if they do not already exist), which will contain the debugging and profiling object files and executables.

Note - make performs the rule in the .INIT target just after the makefile is read.

The variant executables are made to depend on the object files listed in the VARIANTS.o macro. This macro is given the value of OBJECTS by default; later on it is reassigned using a conditional macro definition, at which time either the debug_dir/ or profile_dir/ prefix is added. Executables in the subdirectories depend on the object files that are built in those same subdirectories.

Next, pattern-matching rules are added to indicate that the object files in both subdirectories depend upon source (.c) files in the working directory. This is the key step needed to allow all three variants to be built and maintained from a single set of source files.

Finally, the clean target has been updated to recursively remove the debug_dir and profile_dir subdirectories and their contents, which should be regarded as temporary. This is in keeping with the custom that derived files are to be built in the same directory as their sources, since the subdirectories for the variants are considered temporary.

TABLE 4–13 Sample Makefile for Separate Debugging and Profiling Program Variants

```
# Simple makefile for maintaining separate
# debugging and profiling program variants.

CFLAGS= -O

SOURCES= main.c rest.c
OBJECTS= $(SOURCES:%.c=$(VARIANT)/%.o)
VARIANT= .

functions profile debug: $$(OBJECTS)
 $(LINK.c) -o $(VARIANT)/$@ $(OBJECTS)

debug := VARIANT = debug_dir
debug := CFLAGS = -g
profile := VARIANT = profile_dir
profile := CFLAGS = -O -pg

.KEEP_STATE:
.INIT:  profile_dir debug_dir
profile_dir debug_dir:
 test -d $@ || mkdir $@
 $$(VARIANT)/%.o: %.c
 $(COMPILE.c) $< -o $@
clean:
 rm -r profile_dir debug_dir $(OBJECTS) functions
```

Makefile for a Library with Separate Variants

The modifications for separate library variants are quite similar:

```
# Makefile for maintaining separate library variants.

CFLAGS= -O

SOURCES= main.c rest.c
LIBRARY= lib.a
LSOURCES= fnc.c

OBJECTS= $(SOURCES:%.c=$(VARIANT)/%.o)
VLIBRARY= $(LIBRARY:%.a=$(VARIANT)/%.a)
LOBJECTS= $(LSOURCES:%.c=$(VARIANT)/%.o)
VARIANT= .

program profile debug: $$(OBJECTS) $$(VLIBRARY)
          $(LINK.c) -o $(VARIANT)/$@ $<

lib.a debug_dir/lib.a profile_dir/lib.a: $$(LOBJECTS)
          ar rv $@ $?

$$(VLIBRARY)($$(VARIANT)%.o): $$(VARIANT)%.o
         @true
profile := VARIANT = profile_dir
profile := CFLAGS = -O -pg

debug := VARIANT = debug_dir
debug := CFLAGS = -g

.KEEP_STATE:
profile_dir debug_dir:
         test -d $@ || mkdir $@
$$(VARIANT)/%.o: %.c
         $(COMPILE.c) $< -o $@
```

While an interesting and useful compilation technique, this method for maintaining separate variants is a bit complicated. For the sake of clarity, it is omitted from subsequent examples.

Maintaining a Directory of Header Files

The `makefile` for maintaining an include directory of headers is quite simple. Since headers consist of plain text, all that is needed is a target, `all`, that lists them as dependencies. Automatic SCCS retrieval takes care of the rest. If you use a macro for the list of headers, this same list can be used in other target entries.

```
# Makefile for maintaining an include directory.

FILES.h= calc.h map.h draw.h

all: $(FILES.h)

clean:
```

```
        rm -f $(FILES.h)
```

Compiling and Linking with Your Own Libraries

When preparing your own library packages, it makes sense to treat each library as an entity that is separate from its header(s) and the programs that use it. Separating programs, libraries, and headers into distinct directories often makes it easier to prepare `makefiles` for each type of module. Also, it clarifies the structure of a software project.

Note - It is not a good idea to have things pop up all over the file system as a result of running `make`.

A courteous and necessary convention of `makefiles` is that they only build files in the working directory, or in temporary subdirectories. Unless you are using `make` specifically to install files into a specific directory on an agreed-upon file system, it is regarded as very poor form for a `makefile` to produce output in another directory.

Building programs that rely on libraries in other directories adds several new wrinkles to the `makefile`. Up until now, everything needed has been in the directory, or else in one of the standard directories that are presumed to be stable. This is not true for user-supplied libraries that are part of a project under development.

Because these libraries aren't built automatically (there is no equivalent to hidden dependency checking for them), you must supply target entries for them. On the one hand, you need to ensure the libraries you link with are up to date.

On the other, you need to observe the convention that a `makefile` should only maintain files in the local directory. In addition, the `makefile` should not contain information duplicated in another.

Nested `make` Commands

The solution is to use a nested `make` command, running in the directory the library resides in, to rebuild it (according to the target entry in the `makefile` there)

Note - The `MAKE` macro, which is set to the value ''make'' by default, overrides the `–n` option. Any command line in which it is referred to is executed, even though `–n` might be in effect. Since this macro is used to invoke `make`, and since the `make` it invokes inherits `–n` option from the special `MAKEFLAGS` macro, make can trace a hierarchy of nested make commands with the `–n` option.

```
# First cut entry for target in another directory.

../lib/libpkg.a:
        cd ../lib ; $(MAKE) libpkg.a
```

The library is specified with a path name relative to the current directory. In general, it is better to use relative path names. If the project is moved to a new root directory or machine, so long as its structure remains the same relative to that new root directory, all the target entries will still point to the proper files.

Within the nested `make` command line, the dynamic macro modifiers F and D come in handy, as does the `MAKE` predefined macro. If the target being processed is in the form of a pathname, $(@F) indicates the filename part, while $(@D) indicates the directory part. If there are no / characters in the target name, then $(@D) is assigned the dot character (.) as its value.

The target entry can be rewritten as:

```
# Second cut.

../lib/libpkg.a:
        cd $(@D); $(MAKE) $(@F)
```

Forcing a Nested `make` Command to Run

Because it has no dependencies, this target runs only when the file named `../lib/libpkg.a` is missing. If the file is a library archive protected by `.PRECIOUS`, this could be a rare occurrence. The current `make` invocation neither knows nor cares about what that file depends on, nor should it. It is the nested invocation that decides whether and how to rebuild that file.

After all, just because a file is present in the file system does not mean it is up-to-date. This means that you have to force the nested `make` to run, regardless of the presence of the file, by making it depend on another target with a null rule (and no extant file):

TABLE 4–14 Target Entry for a Nested make Command

```
# Reliable target entry for a
# nested make command.

../lib/libpkg.a: FORCE
      cd $(@D); $(MAKE) $(@F)
FORCE:
```

In this way, `make` reliably changes to the correct directory `../lib` and builds `libpkg.a` if necessary, using instructions from the `makefile` found in that directory. These lines are produced by the nested `make` run:

```
$ make ../lib/libpkg.a
cd ../lib; make libpkg.a
make libpkg.a
'libpkg.a' is up to date.
```

The following makefile uses a nested make command to process local libraries that a program depends on.

TABLE 4–15 Makefile for C Program with User-Supplied Libraries

```
# Makefile for a C program with user-supplied
# libraries and nested make commands.

CFLAGS= -O

.KEEP_STATE:

functions: main.o data.o ../lib/libpkg.a
      $(LINK.c) -o $@ main.o data.o
      ../lib/libpkg.a -lcurses -ltermlib
      ../lib/libpkg.a:  FORCE
      cd $(@D); $(MAKE) $(@F)
FORCE:

lint: main.ln data.ln
      $(LINT.c) main.ln data.ln
clean:
      rm -f functions main.o data.o main.ln data.ln
```

When ../lib/libpkg.a is up to date, this makefile produces:

```
$ make
cc -O -c main.c
cc -O -c data.c
cd ../lib; make libpkg.a
'libpkg.a' is up to date.
cc -O -o functions main.o data.o ../lib/libpkg.a -lcurses -l   termlib
```

The MAKEFLAGS Macro

Like the MAKE macro, MAKEFLAGS is also a special case.

Note - Do not define MAKEFLAGS in your makefiles.

MAKEFLAGS contains flags (that is, single-character options) for the make command. Unlike other FLAGS macros, the MAKEFLAGS value is a concatenation of flags, without a leading '-'. For instance the string eiknp would be a recognized value for MAKEFLAGS, while -f x.mk or macro=value would not.

If the MAKEFLAGS environment variable is set, make runs with the combination of flags given on the command line and contained in that variable.

The value of MAKEFLAGS is always exported, whether set in the environment or not, and the options it contains are passed to any nested make commands (whether invoked by `$(MAKE)`, `make`, or `/usr/bin/make`). This insures that nested `make` commands are always passed the options which the parent `make` was invoked.

Passing Parameters to Nested `make` Commands

With the exception of `MAKEFLAGS`, `make` imports variables from the environment and treats them as if they were defined macros. In turn, `make` propagates those environment variables and their values to commands it invokes, including nested `make` commands.

Note - The `SHELL` environment variable is neither imported nor exported to this version of `make`.

Macros can be defined as command-line arguments, as well as the `makefile`. This can lead to name-value conflicts when a macro is defined in more than one place, and `make` has a fairly complicated precedence rule for resolving them.

First, conditional macro definitions always take effect within the targets (and their dependencies) for which they are defined.

If `make` is invoked with a macro-definition argument, that definition takes precedence over definitions given either within the `makefile`, or imported from the environment. (This does not necessarily hold true for nested `make` commands, however.) Otherwise, if you define (or redefine) a macro within the `makefile`, the most recent definition applies. The latest definition normally overrides the environment.

Lastly, if the macro is defined in the default file and nowhere else, that value is used.

With nested `make` commands, definitions made in the `makefile` normally override the environment, but only for the `makefile` in which each definition occurs; the value of the corresponding environment variable is propagated regardless.

Command-line definitions override both environment and `makefile` definitions, but only in the `make` run for which they are supplied. Although values from the command line are propagated to nested `make` commands, they are overridden both by definitions in the nested `makefiles`, and by environment variables imported by the nested `make` commands.

The `–e` option behaves more consistently. The environment overrides macro definitions made in any `makefile`, and command-line definitions are always used ahead of definitions in the `makefile` and the environment. One drawback to `–e` is that it introduces a situation in which information that is *not contained in the makefile* can be critical to the success or failure of a build.

To avoid these complications, when you want to pass a specific value to an entire hierarchy of `make` commands, run `make –e` in a subshell with the environment set properly (in the C shell):

```
% (unsetenv MAKEFLAGS LDFLAGS; setenv CFLAGS -g; make -e)
```

If you want to test the cases yourself, you can use the following `makefiles` to illustrate the various cases.

```
# top.mk

MACRO= "Correct but unexpected."

top:
	@echo "------------------------------ top"
	echo $(MACRO)
	@echo "------------------------------"
	$(MAKE) -f nested.mk
	@echo "------------------------------ clean"
clean:
	rm nested


# nested.mk

MACRO=nested

nested:
	@echo "------------------------------ nested"
	touch nested
	echo $(MACRO)
	$(MAKE) -f top.mk
	$(MAKE) -f top.mk clean
```

The following is a summary of macro assignment orders:

TABLE 4–16 Summary of Macro Assignment Order

Without -e	With -e in effect
top-level `make` commands:	
Conditional definitions	Conditional definitions
Make command line	Make command line
Latest makefile definition	Environment value
Environment value	Latest makefile definition
Predefined value, if any	Predefined value, if any
nested `make` commands:	
Conditional definitions	Conditional definitions
Make command line	Make command line

TABLE 4–16 Summary of Macro Assignment Order *(continued)*

Latest makefile definition	Parent make cmd. line
Environment variable	Environment value
Predefined value, if any	Latest makefile definition
Parent make cmd. line	Predefined value, if any

Compiling Other Source Files

Compiling and Linking a C Program with Assembly Language Routines

The `makefile` in the next example maintains a program with C source files linked with assembly language routines. There are two varieties of assembly source files: those that do not contain `cpp` preprocessor directives, and those that do.

By convention, assembly source files without preprocessor directives have the `.s` suffix. Assembly sources that require preprocessing have the `.S` suffix.

Note - ASFLAGS passes options for as to the .s.o and .S.o implicit rules.

Assembly sources are assembled to form object files in a fashion similar to that used to compile C sources. The object files can then be linked into a C program. `make` has implicit rules for transforming `.s` and `.S` files into object files, so a target entry for a C program with assembly routines need only specify how to link the object files. You can use the familiar `cc` command to link object files produced by the assembler:

TABLE 4–17 Summary of Macro Assignment Order

```
CFLAGS= -O
ASFLAGS= -O

.KEEP_STATE:

driver: c_driver.o s_routines.o S_routines.o
      cc -o driver c_driver.o s_routines.o
      S_routines.o
```

Note that the .S files are processed using the cc command, which invokes the C preprocessor cpp, and invokes the assembler.

Compiling lex and yacc Sources

lex and yacc produce C source files as output. Source files for lex end in the suffix .l, while those for yacc end in .y. When used separately, the compilation process for each is similar to that used to produce programs from C sources alone.

There are implicit rules for compiling the lex or yacc sources into .c files; from there, the files are further processed with the implicit rules for compiling object files from C sources. When these source files contain no #include statements, there is no need to keep the .c file, which in this simple case serves as an intermediate file. In this case you could use .l.o rule, or the .y.o rule, to produce the object files, and remove the (derived) .c files.

For example, the makefile:

```
CFLAGS= -O
.KEEP_STATE:

all: scanner parser
scanner: scanner.o
parser: parser.o
```

produces the result shown below.

```
$ make -n
rm -f scanner.c
lex -t scanner.l > scanner.c
cc -O -c -o scanner.o scanner.c
rm -f scanner.c
yacc parser.y
cc -O -c -o parser.o y.tab.c
rm -f y.tab.c
```

Things become more complicated when you use lex and yacc in combination. In order for the object files to work together properly, the C code from lex must include a header produced by yacc. It might be necessary to recompile the C source file produced by lex when the yacc source file changes. In this case, it is better to retain the intermediate (.c) files produced by lex, as well as the additional .h file yacc provides, to avoid running lex whenever the yacc source changes.

Note - yacc produces output files named y.tab.c and y.tab.h. If you want the output files to have the same basename as the source file, you must rename them.

The following makefile maintains a program built from a lex source, a yacc source, and a C source file.

```
CFLAGS= -O
.KEEP_STATE:
```

```
a2z: c_functions.o scanner.o parser.o
        cc -o $@ c_functions.o scanner.o parser.o
scanner.c:

parser.c + parser.h: parser.y
        yacc -d parser.y
        mv y.tab.c parser.c
        mv y.tab.h parser.h
```

Because there is no transitive closure for implicit rules, you must supply a target entry for `scanner.c`. This entry bridges the gap between the `.l.c` implicit rule and the `.c.o` implicit rule, so that the dependency list for scanner.o extends to `scanner.l`. Since there is no rule in the target entry, `scanner.c` is built using the `.l.c` implicit rule.

The next target entry describes how to produce the `yacc` intermediate files. Because there is no implicit rule for producing both the header and the C source file using `yacc –d`, a target entry must be supplied that includes a rule for doing so.

Specifying Target Groups with the + Sign

In the target entry for `parser.c` and `parser.h`, the `+` sign separating the target names indicates that the entry is for a *target group*. A target group is a set of files, all of which are produced when the rule is performed. Taken as a group, the set of files comprises the target. Without the `+` sign, each item listed would comprise a separate target. With a target group, `make` checks the modification dates separately against each target file, but performs the target's rule only once, if necessary, per `make` run.

Maintaining Shell Scripts with `make` and SCCS

Although a shell script is a plain text file, it must have execute permission to run. Since `SCCS` removes execute permission for files under its control, it is convenient to make a distinction between a shell script and its "source" under `SCCS`. `make` has an implicit rule for deriving a script from its source. The suffix for a shell script source file is `.sh`. Even though the contents of the script and the `.sh` file are the same, the script has execute permissions, while the `.sh` file does not. `make`'s implicit rule for scripts "derives" the script from its source file, making a copy of the `.sh` file (retrieving it first, if necessary) and changing the mode of the resulting script file to allow execution. For example:

```
$ file script.sh
script.sh:    ascii text
$ make script
cat script.sh > script
chmod +x script
$ file script
script:    commands text
```

Running Tests with make

Shell scripts are often helpful for running tests and performing other routine tasks that are either interactive or don't require make's dependency checking. Test suites, in particular, often entail providing a program with specific, repeatable input that a program might expect to receive from a terminal.

In the case of a library, a set of programs that exercise its various functions can be written in C, and then executed in a specific order, with specific inputs from a script. In the case of a utility program, there can be a set of benchmark programs that exercise and time its functions. In each of these cases, the commands to run each test can be incorporated into a shell script for repeatability and easy maintenance.

After you have developed a test script that suits your needs, including a target to run it is easy. Although make's dependency checking might not be needed within the script itself, you *can* use it to make sure that the program or library is updated before running those tests.

In the following target entry for running tests, test depends on lib.a. If the library is out of date, make rebuilds it and proceeds with the test. This insures that you always test with an up-to-date version:

```
#This is the library we're testing
LIBRARY= lib.a

test: $(LIBRARY) testscript
         set -x ; testscript > /tmp/test.\$\$

testscript: testscript.sh test_1 test_2 test_3

#rules for building the library
$(LIBRARY):
         @ echo Building $(LIBRARY)
        (library-building rules here)

#test_1 ... test_3 exercise various library functions
test_1 test_2 test_3: $$@.c $(LIBRARY)
         $(LINK.c) -o $@ $<
```

test also depends on testscript, which in turn depends on the three test programs.

This ensures that they too are up-to-date before make initiates the test procedure. lib.a is built according to its target entry in the makefile; testscript is built using the .sh implicit rule; and the test programs are built using the rule in the last target entry, assuming that there is just one source file for each test program. (The .c implicit rule doesn't apply to these programs because they must link with the proper libraries in addition to their .c files).

Escaped References to a Shell Variable

The string `\$\$` in the rule for test illustrates how to escape the dollar-sign from interpretation by `make`. `make` passes each `$` to the shell, which expands the `$$` to its process ID. This technique allows each test to write to a unique temporary filename. The `set -x` command forces the shell to display the commands it runs on the terminal, which allows you to see the actual file name containing the results of the specific test.

Shell Command Substitutions

You can supply shell command substitutions within a rule as in the following example:

```
do:
        @echo 'cat Listfile'
```

You can even place the backquoted expression in a macro:

```
DO= 'cat Listfile'
do:
        @echo $(DO)
```

However, you can only use this form of command substitution within a rule.

Command-Replacement Macro References

If you supply a shell command as the definition of a macro:

```
COMMAND= cat Listfile
```

you can use a *command-replacement macro reference* to instruct `make` to replace the reference with the output of the command in the macro's value. This form of command substitution can occur anywhere within a `makefile`:

```
COMMAND= cat Listfile
$(COMMAND:sh): $$(@:=.c)
```

This example imports a list of targets from another file and indicates that each target depends on a corresponding `.c` file.

As with shell command substitution, a command replacement reference evaluates to the standard output of the command. `NEWLINE` characters are converted to `SPACE` characters. The command is performed whenever the reference is encountered. The command's standard error is ignored. However, if the command returns a non-zero exit status, `make` halts with an error.

A workaround for this is to append the `true` command to the command line:

```
COMMAND = cat Listfile ; true
```

Command-Replacement Macro Assignment

A macro assignment of the form

```
cmd_macro:sh = command
```

assigns the standard output of the indicated *command* to *cmd_macro*; for instance:

```
COMMAND:sh = cat Listfile
$(COMMAND): $$(@:=.c)
```

is equivalent to the previous example. However, with the assignment form, the command is only performed once per `make` run. Again, only the standard output is used, `NEWLINE` characters are converted to `SPACE` characters, and a non-zero exit status halts `make` with an error.

Alternate forms of command-replacement macro assignments are:

```
macro:sh += command
```

Append command output to the value of `macro`.

```
target := macro:sh = command
```

Conditionally define `macro` to be the output of `command` when processing `target` and its dependencies.

```
target := macro:sh += command
```

Conditionally append the output of *command* to the value of *macro* when processing *target* and its dependencies.

Maintaining Software Projects

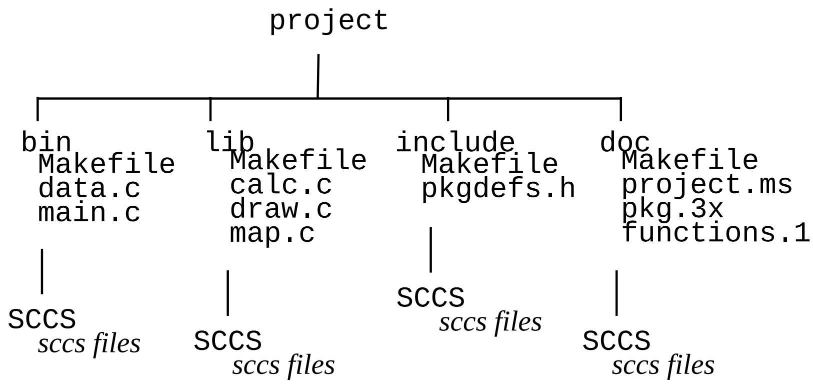
`make` is especially useful when a software project consists of a system of programs and libraries. By taking advantage of nested `make` commands, you can use it to maintain object files, executables, and libraries in a whole hierarchy of directories. You can use `make` in conjunction with `SCCS` to ensure that sources are maintained in a controlled manner, and that programs built from them are consistent. You can provide other programmers with duplicates of the directory hierarchy for simultaneous development and testing if you want (although there are trade-offs to consider).

You can use `make` to build the entire project and install final copies of various modules onto another file system for integration and distribution.

Organizing a Project for Ease of Maintenance

As mentioned earlier, one good way to organize a project is to segregate each major piece into its own directory. A project broken out this way usually resides within a single file system or directory hierarchy. Header files could reside in one subdirectory, libraries in another, and programs in still another. Documentation, such as reference pages, can also be kept on hand in another subdirectory.

Suppose that a project is composed of one executable program, one library that you supply, a set of headers for the library routines, and some documentation, as in the following diagram.



The `makefiles` in each subdirectory can be borrowed from examples in earlier sections, but something more is needed to manage the project as a whole. A carefully structured `makefile` in the root directory, the *root makefile* for the project, provides target entries for managing the project as a single entity.

As a project grows, the need for consistent, easy-to-use `makefiles` also grows. Macros and target names should have the same meanings no matter which `makefile` you are reading. Conditional macro definitions and compilation options for output variants should be consistent across the entire project.

Where feasible, a *template* approach to writing `makefiles` makes sense. With a template, you track how the project is built. All you have to do to add a new type of module is to make a new directory for it, copy an appropriate `makefile` into that directory, and edit a few lines. You also need to add the new module to the list of things to build in the root `makefile`.

Conventions for macro and target names, such as those used in the default `makefile`, should be instituted and observed throughout the project. Mnemonic names mean that although you might not remember the exact function of a target or value of a macro, you will know the type of function or value it represents by the name and that's usually valuable when deciphering a `makefile` also.

Using `include` Makefiles

One method of simplifying `makefiles`, while providing a consistent compilation environment, is to use the `make`:

```
include filename
```

This directive reads in the contents of a named `makefile`; if the named file is not present, `make` checks for a file by that name in `/etc/default`.

For instance, there is no need to duplicate the pattern-matching rule for processing troff sources in each `makefile`, when you can include its target entry, as shown below.

```
SOURCES= doc.ms spec.ms
...
clean: $(SOURCES)
include ../pm.rules.mk
```

Here, `make` reads in the contents of the `../pm.rules.mk` file:

```
# pm.rules.mk
#
# Simple "include" makefile for pattern-matching
# rules.

%.tr: %.ms
          troff -t -ms $< > $@
%.nr: %.ms
          nroff -ms $< > $@
```

Installing Finished Programs and Libraries

When a program is ready to be released for outside testing or general use, you can use `make` to install it. Adding a new target and new macro definition to do so is not difficult:

```
DESTDIR= /proto/project/bin

install: functions
         -mkdir $(DESTDIR)
         cp functions $(DESTDIR)
```

A similar target entry can be used for installing a library or a set of headers.

Building the Entire Project

Occasionally you should take a snapshot of the sources and the object files that they produce. Building an entire project involves invoking `make` successively in each subdirectory to build and install each module. The following example shows how to use nested `make` commands to build a simple project.

Assume your project is located in two different subdirectories, `bin` and `lib`, and that in both subdirectories you want `make` to debug, test, and install the project.

First, in the projects main, or root, directory, you put a `makefile` such as this:

```
# Root makefile for a project.

TARGETS= debug test install
SUBDIRS= bin lib

all: $(TARGETS)
$(TARGETS):
	@for i in $(SUBDIRS) ; \
	do \
	      cd $$i ; \
	      echo "Current directory:  $$i" ;\
	      $(MAKE) $@ ; \
	      cd .. ; \
	done
```

Then, in each subdirectory (in this case, `bin`) you place a `makefile` of this general form:

```
#Sample makefile in subdirectory
debug:
	@echo "   Building debug target"
	@echo
test:
	@echo "   Building test target"
	@echo
install:
	@echo "   Building install target"
	@echo
```

When you type `make` (in the base directory), you get the following output:

```
$ make
Current directory:  bin
          Building debugging target

Current directory:  lib
          Building debugging target

Current directory:  bin
          Building testing target

Current directory:  lib
          Building testing target

Current directory:  bin
          Building install target

Current directory:  lib
          Building install target
$
```

Maintaining Directory Hierarchies with the Recursive Makefiles

If you extend your project hierarchy to include more layers, chances are that not only will the `makefile` in each intermediate directory have to produce target files, but it will also have to invoke nested `make` commands for subdirectories of its own.

Files in the current directory can sometimes depend on files in subdirectories, and their target entries need to depend on their counterparts in the subdirectories.

The nested `make` command for each subdirectory should run before the command in the local directory. One way to ensure that the commands run in the proper order is to make a separate entry for the nested part and another for the local part. If you add these new targets to the dependency list for the original target, its action will encompass them both.

Maintaining Recursive Targets

Targets that encompass equivalent actions in both the local directory and in subdirectories are referred to as *recursive* targets.

Note - Strictly speaking, any target that calls `make` with its name as an argument, is recursive. However, here the term is reserved for the narrower case of targets that have both nested and local actions. Targets that have only nested actions are referred to as “nested” targets.

A `makefile` with recursive targets is referred to as a *recursive* `makefile`.

In the case of `all`, the nested dependencies are `NESTED_TARGETS`; the local dependencies, `LOCAL_TARGETS`:

```
NESTED_TARGETS=  debug test install
SUBDIRS= bin lib
LOCAL_TARGETS= functions

all: $(NESTED_TARGETS) $(LOCAL_TARGETS)

$(NESTED_TARGETS):
         @ for i in $(SUBDIRS) ; \
         do \
               echo "Current directory:  $$i" ;\
               cd $$i ; \
               $(MAKE) $@ ; \
               cd .. ; \
         done

$(LOCAL_TARGETS):
         @ echo "Building $@ in local directory."
       (local directory commands)
```

The nested make must also be recursive, unless it is at the bottom of the hierarchy. In the makefile for a leaf directory (one with no subdirectories), you build only local targets.

Maintaining a Large Library as a Hierarchy of Subsidiaries

When maintaining a very large library, it is sometimes easier to break it up into smaller, subsidiary libraries, and use make to combine them into a complete package. Although you cannot combine libraries directly with ar, you can extract the member files from each subsidiary library, then archive those files in another step, as shown in the following example:

```
$ ar xv libx.a
x - x1.o
x - x2.o
x - x3.o
$ ar xv liby.a
x - y1.o
x - y2.o
$ ar rv libz.a *.o
a - x1.o
a - x2.o
a - x3.o
a - y1.o
a - y2.o
ar: creating libz.a
```

A subsidiary library is maintained using a makefile in its own directory, along with the (object) files it is built from. The makefile for the complete library typically makes a symbolic link to each subsidiary archive, extracts their contents into a temporary subdirectory, and archives the resulting files to form the complete package.

The next example updates the subsidiary libraries, creates a temporary directory in which to put extracted the files, and extracts them. It uses the * (shell) wild card within that temporary directory to generate the collated list of files. While filename wild cards are generally frowned upon, this use of the wild card is acceptable because a new directory is created whenever the target is built. This guarantees that it contains only files extracted during the *current* make run.

Note - In general, use of shell filename wild cards is considered to be bad form in a makefile. If you do use them, you need to take steps to insure that it excludes spurious files by isolating affected files in a temporary subdirectory

---

The example relies on a naming convention for directories. The name of the directory is taken from the basename of the library it contains. For instance, if `libx.a` is a subsidiary library, the directory that contains it is named `libx`.

It makes use of suffix replacements in dynamic-macro references to derive the directory name for each specific subdirectory. (You can verify that this is necessary.) It uses a shell for loop to successively extract each library and a shell command substitution to collate the object files into proper sequence for linking (using `lorder` and `tsort`) as it archives them into the package. Finally, it removes the temporary directory and its contents.

```
# Makefile for collating a library from subsidiaries.

CFLAGS= -O

.KEEP_STATE:
.PRECIOUS:  libz.a

all: lib.a

libz.a: libx.a liby.a
         -rm -rf tmp
         -mkdir tmp
         set -x ; for i in libx.a liby.a ; \
                do ( cd tmp ; ar x ../$$i ) ; done
         ( cd tmp ; rm -f *_*_.SYMDEF ; ar cr ../$@ ‘lorder * | tsort‘ )
         -rm -rf tmp libx.a liby.a

libx.a liby.a: FORCE
         -cd $(@:.a=) ; $(MAKE) $@
         -ln -s $(@:.a=)/$@ $@
FORCE:
```

For the sake of clarity, this example omits support for alternate variants, as well as the targets for `clean`, `install`, and `test` (does not apply since the source files are in the subdirectories).

The `rm -f *_*_.SYMDEF` command embedded in the collating line prevents a symbol table in a subsidiary (produced by running ar on that library) from being archived in this library.

Because the nested `make` commands build the subsidiary libraries before the current library is processed, you can extend this `makefile` to account for libraries built from both subsidiaries and object files in the current directory. You need to add the list of object files to the dependency list for the library and a command to copy them into the temporary subdirectory for collation with object files extracted from subsidiary libraries.

```
# Makefile for collating a library from subsidiaries and local objects.

CFLAGS= -O

.KEEP_STATE:
.PRECIOUS:  libz.
```

```
OBJECTS= map.o calc.o draw.o

all: libz.a

libz.a: libx.a liby.a $(OBJECTS)
          -rm -rf tmp
          -mkdir tmp
          -cp $(OBJECTS) tmp
          set -x ; for i in libx.a liby.a ; \
                do ( cd tmp ; ar x ../$$i ) ; done
          ( cd tmp ; rm -f *_*_.SYMDEF ; ar cr ../$@ \
                'lorder * | tsort' )
          -rm -rf tmp lix.a liby.a

libx.a liby.a: FORCE
          -cd $(@:.a=) ; $(MAKE) $@
          -ln -s $(@:.a=)/$@ $@
FORCE:
```

Reporting Hidden Dependencies to `make`

You might need to write a command for processing hidden dependencies. For instance, you might need to trace document source files that are included in a troff document by way of `.so` requests. When `.KEEP_STATE` is in effect, `make` sets the environment variable `SUNPRO_DEPENDENCIES` to the value:

```
SUNPRO_DEPENDENCIES='report-file target'
```

After the command has terminated, `make` checks to see if the file has been created, and if it has, `make` reads it and writes reported dependencies to `.make.state` in the form:

```
target:dependency ...
```

where `target` is the same as in the environment variable.

`make` Enhancements Summary

The following summarizes additional new features to `make`.

Default Makefile

`make`'s implicit rules and macro definitions are no longer hard-coded within the program itself. They are now contained in the default makefile `/usr/share/lib/make/make.rules`. `make` reads this file automatically unless there is a file in the local directory named `make.rules`. When you use a local make.rules file, you must add a directive to include the standard make.rules file to get the standard implicit rules and predefined macros.

The State File .make.state

make also reads a state file, .make.state, in the directory. When the special-function target .KEEP_STATE is used in the makefile, make writes out a cumulative report for each target containing a list of hidden dependencies (as reported by compilation processors such as cpp) and the most recent rule used to build each target. The state file is very similar in format to an ordinary makefile.

Hidden-Dependency Checking

When activated by the presence of the .KEEP_STATE target, make uses information reported from cc, cpp, f77, ld, make, pc and other compilation commands and performs a dependency check against any header files (or in some cases, libraries) that are incorporated into the target file. These “hidden” dependency files do not appear in the dependency list, and often do not reside in the local directory.

Command-Dependency Checking

When .KEEP_STATE is in effect, if any command line used to build a target changes between make runs (either as a result of editing the makefile or because of a different macro expansion), the target is treated as if it were out of date; make rebuilds it (even if it is newer than the files it depends on).

Automatic Retrieval of SCCS Files

This section discusses the rule for the automatic retrieval of files under sccs.

Tilde Rules Superseded

This version of make automatically runs sccs get, as appropriate, when there is no rule to build a target file. A tilde appended to a suffix in the suffixes list indicates that sccs extraction is appropriate for the dependency file. make no longer supports tilde suffix rules that include commands to extract current versions of sccs files.

To inhibit or alter the procedure for automatic extraction of the current sccs version, redefine the .SCCS_GET special-function target. An empty rule for this target entirely inhibits automatic extraction.

Pattern-Matching Rules

Pattern-matching rules have been added to simplify the process of adding new implicit rules of your own design. A target entry of the form:

```
tp%ts : dp%ds
      rule
```

defines a pattern-matching rule for building a target from a related dependency file. `tp` is the target prefix; `ts`, its suffix.`dp` is the dependency prefix; `ds`, its suffix. The `%` symbol is a wild card that matches a contiguous string of zero or more characters appearing in both the target and the dependency file name. For example, the following target entry defines a pattern-matching rule for building a `troff` output file, with a name ending in `.tr` from a file that uses the –`ms` macro package ending in `.ms`:

```
%.tr: %ms
troff -t -ms $< > $@
```

With this entry in the `makefile`, the command:

```
make doc.tr
```

produces:

```
$ make doc.tr
troff -t -ms doc.ms > doc.tr
```

Using that same entry, if there is a file named `doc2.ms`, the command:

```
make doc2.tr
```

produces:

```
$ make doc2.tr
troff -t -ms doc2.ms > doc2.tr
```

An explicit target entry overrides any pattern-matching rule that might apply to a target. Pattern-matching rules, in turn, normally override implicit rules. An exception to this is when the pattern-matching rule has no commands in the rule portion of its target entry. In this case, `make` continues the search for a rule to build the target, and uses as its dependency the file that matched the (dependency) pattern.

Pattern-Replacement Macro References

As with suffix rules and pattern-matching rules, pattern-replacement macro references have been added to provide a more general method for altering the values of words in a specific macro reference than that already provided by suffix replacement in macro references. A pattern-replacement macro reference takes the form:

```
$ (macro :p %s =np %ns )
```

where p is an existing prefix (if any), s is an existing suffix (if any), np and ns are the new prefix and suffix, and % is a wild card character matching a string of zero or more characters within a word.

The prefix and suffix replacements are applied to all words in the macro value that match the existing pattern. Among other things, this feature is useful for prefixing the name of a subdirectory to each item in a list of files. For instance, the following makefile:

```
SOURCES= x.c y.c z.c
SUBFILES.o= $(SOURCES:%.c=subdir/%.o)

all:
        @echo $(SUBFILES.o)
```

produces:

```
$ make
subdir/x.o subdir/y.o subdir/z.o
```

You can use any number of % wild cards in the right-hand (replacement) side of the = sign, as needed. The following replacement:

```
...
NEW_OBJS= $(SOURCES:%.c=%/%.o)
```

would produce:

```
...
x/x.o y/y.o z/z.o
```

Pattern-replacement macro references should not appear on the dependency line of a pattern-matching rule's target entry. This produces unexpected results. With the makefile:

```
OBJECT= .o

x:
%: %.$(OBJECT:%o=%Z)
        cp $< $@
```

it appears that make should attempt to build a target named x from a file named x.Z. However, the pattern-matching rule is not recognized; make cannot determine which of the % characters in the dependency line apply to the pattern-matching rule and that apply to the macro reference.

Consequently, the target entry for x.Z is never reached. To avoid problems like this, you can use an intermediate macro on another line:

```
OBJECT= .o
ZMAC= $(OBJECT:%o=%Z)

x:
%: %$(ZMAC)
        cp $< $@
```

New Options

The new options are:

`–d`	Displays dependency-check results for each target processed. Displays all dependencies that are newer, or indicates that the target was built as the result of a command dependency.
`–dd`	Performs the same function as `–d` in earlier versions of `make`. Displays a great deal of output about all details of the `make` run, including internal states, and so forth.
`–D`	Displays the text of the `makefile` as it is read.
`–DD`	Displays the text of the `makefile` and of the default `makefile` being used.
`–p`	Prints macro definitions and target entries.
`–P`	Reports all dependencies for targets without rebuilding them.

Support for C++ and Modula-2

This version of `make` contains predefined macros for compiling C++ programs. It also contains predefined macros and implicit rules for compiling Modula-2.

Naming Scheme for Predefined Macros

The naming scheme for predefined macros has been rationalized, and the implicit rules have been rewritten to reflect the new scheme. The macros and implicit rules are upward compatible with existing `makefiles`.

Some examples include the macros for standard compilations commands:

```
LINK.c
```

which is a standard `cc` command line for producing executable files.

```
COMPILE.c
```

which is a standard `cc` command line for producing object files.

New Special-Purpose Targets

.KEEP_STATE	When included in a makefile, this target enables hidden dependency and command-dependency checking. In addition, make updates the state file .make.state after each run.

Note - The .KEEP_STATE target should not be removed if it has been used in a make run.

.INIT **and** .DONE	These targets can be used to supply commands to perform at the beginning and end of each make run.
.FAILED	The commands supplied are performed when make fails.
.PARALLEL	These can be used to indicate which targets are to be processed in parallel, and which are to be processed in serial fashion.
.SCCS_GET	This target contains the rule for extracting current versions of files from sccs history files.
.WAIT	When this target appears in the dependency list, make waits until the dependencies that precede it are finished before processing those that follow, even when processing is parallel.

New Implicit lint Rule

Implicit rules have been added to support incremental verification with lint.

Macro Processing Changes

A macro value can now be of virtually any length. Whereas in earlier versions only trailing white space was stripped from a macro value, this version strips off both leading and trailing white space characters.

Macros: Definition, Substitution, and Suffix Replacement

New Append Operator

`+=` — This is the new append operator that appends a SPACE followed by a word or words, onto the existing value of the macro.

Conditional Macro Definitions

`:=` — This is the conditional macro definitions operator that indicates a conditional (targetwise) macro definition. A makefile entry of the form:

```
target := macro = value
```

indicates that macro takes the indicated value while processing target and its dependencies.

Patterns in Conditional Macros

make recognizes the % wild card pattern in the target portion of a conditional macro definition. For instance:

```
profile_% := CFLAGS += -pg
```

would modify the CFLAGS macro for all targets having the 'profile_' prefix. Pattern replacements can be used within the value of a conditional definition. For instance:

```
profile_% := OBJECTS = $(SOURCES:%.c=profile_%.o)
```

applies the profile_ prefix and .o suffix to the basename of every .c file in the SOURCES list (value).

Suffix Replacement Precedence

Substring replacement now takes place following expansion of the macro being referenced. Previous versions of make applied the substitution first, with results that were counterintuitive.

Nested Macro References

make now expands inner references before parsing the outer reference. A nested reference as in this example:

```
CFLAGS-g = -I../include
  OPTION = -g
  $(CFLAGS$(OPTION))
```

now yields the value -I../include , rather than a null value, as it would have in previous versions.

Cross-Compilation Macros

The predefined macros HOST_ARCH and TARGET_ARCH are available for use in cross-compilations. By default, the *arch* macros are set to the value returned by the arch command.

Shell Command Output in Macros

A definition of the form:

```
MACRO :sh = command
```

sets the value of MACRO to the standard output of the indicated command, NEWLINE characters being replaced with SPACE characters. The command is performed just once, when the definition is read. Standard error output is ignored, and make halts with an error if the command returns a non-zero exit status.

A macro reference of the form:

```
$(MACRO :sh)
```

expands to the output of the command line stored in the value of MACRO, whenever the reference is evaluated. NEWLINE characters are replaced with SPACE characters, standard error output is ignored, and make halts with an error if the command returns a non-zero exit status.

Improved ar Library Support

make automatically updates an ar-format library member from a file having the same name as the member. Also, make now supports lists of members as dependency names of the form:

```
lib.a: lib.a(member member ...)
```

Target Groups

It is now possible to specify that a rule produces a set of target files. A + sign between target names in the target entry indicates that the named targets constitute a group. The target group rule is performed once, at most, in a make invocation.

Incompatibilities with Previous Versions

This section briefly discusses the following:

- The –d Option
- Dynamic Macros
- Tilde Rules
- Target Names

The –d Option

The –d option now reports the reason why a target is considered out of date.

Dynamic Macros

Although the dynamic macros `$<` and `$*` were documented as being assigned only for implicit rules and the `.DEFAULT` target, in some cases they actually were assigned for explicit target entries. The assignment action is now documented properly.

The actual value assigned to each of these macros is derived by the same procedure used within implicit rules (this rule has not changed). You can receive unexpected results when you use them in explicit target entries.

Even if you supply explicit dependencies, `make` does not use them to derive values for these macros. Instead, it searches for an appropriate implicit rule and dependency file. For instance, if you have the explicit target entry:

```
test: test.f
        @echo $<
```

and the files: `test.c` and `test.f`, you might expect that `$<` would be assigned the value `test.f`. This is *not* the case. It is assigned `test.c`, because `.c` is ahead of `.f` in the suffixes list:

```
$ make test
test.c
```

For explicit entries, it is best to use a strictly deterministic method for deriving a dependency name using macro references and suffix replacements. For example, you could use `$@.f` instead of `$<` to derive the dependency name. To derive the base name of a `.o` target file, you could use the suffix replacement macro reference: `$(@:.o=)` instead of `$*`.

When hidden dependency checking is in effect, the $? dynamic macro value includes the names of hidden dependencies, such as header files. This can lead to failed compilations when using a target entry such as:

```
x: x.c
        $(LINK.c) -o $@ $?
```

and the file x.c #include's header files. The workaround is to replace '$?' with '$@.<'.

Tilde Rules

Tilde rules are not supported. This version of make does not support tilde suffix rules for version retrieval under SCCS. This might create problems when older makefiles redefine tilde rules to perform special steps when version retrieval under SCCS is required.

Target Names

Target names beginning with ./ are treated as local filenames.

When make encounters a target name beginning with './', it strips those leading characters. For instance, the target named:

```
./filename
```

is interpreted as if it were written:

```
filename
```

This can result in endless loop conditions when used in a recursive target. To avoid this, rewrite the target relative to '..', the parent directory:

```
../dir/filename
```

CHAPTER 5

SCCS Source Code Control System

Coordinating write access to source files is important when changes can be made by several people. Maintaining a record of updates allows you to determine when and why changes were made.

The Source Code Control System (SCCS) allows you to control write access to source files, and to monitor changes made to those files. SCCS allows only one user at a time to update a file, and records all changes in a *history* file.

SCCS allows you to:

- Retrieve copies of any version of the file from the SCCS history.
- Check out and lock a version of the file for editing, so that only you can make changes to it. SCCS prevents one user from unwittingly "clobbering" changes made by another.
- Check in your updates to the file. When you check in a file, you can also supply comments that summarize your changes.
- Back out changes made to your checked-out copy.
- Inquire about the availability of a file for editing.
- Inquire about differences between selected versions.
- Display the *version log* summarizing the changes checked in so far.

The `sccs` Command

The Source Code Control System is composed of the `sccs(1)` command, which is a front end for the utility programs in the `/usr/ccs/bin` directory. The SCCS utility programs are listed in Table 5–2.

The `sccs create` Command

The `sccs create` command places your file under SCCS control. It creates a new history file, and uses the complete text of your source file as the initial version. By default, the history file resides in the SCCS subdirectory.

```
$ sccs create program.c
program.c:
1.1
87 lines
```

The output from SCCS tells you the name of the *created* file, its version number (1.1), and the count of lines.

To prevent the accidental loss or damage to an original, `sccs create` makes a second link to it, prefixing the new filename with a comma (referred to as the *comma-file*.) When the history file has been initialized successfully, SCCS retrieves a new, read-only version. After you have verified the version against its comma-file, you can remove that file.

```
$ cmp ,program.c program.c
(no output means that the files match exactly
$ rm ,program.c
```

Do not try to edit the read-only version that SCCS retrieves. Before you can edit the file, you must check it out using the `sccs edit` command described below.

To distinguish the history file from a current version, SCCS uses the 's.' prefix. Owing to this prefix, the history file is often referred to as the s.file (*s-dot-file*). For historical reasons, it may also be referred to as the *SCCS-file*.

The format of an SCCS history file is described in `sccsfile(4)`.

Basic `sccs` Subcommands

The following `sccs` subcommands perform the basic version-control functions. They are summarized here, and, except for create, are described in detail under "`sccs` Subcommands " on page 182.

`create` — Initialize the history file and first version, as described above.

`edit` — Check out a writable version (for editing). SCCS retrieves a writable copy with you as the owner, and places a lock on the history file so that no one else can check in changes.

`delta` — Check in your changes. This is the complement to the `sccs edit` operation. Before recording your changes, SCCS prompts for a comment, which it then stores in the history file version log.

`get` — Retrieve a read-only copy of the file from the s.file. By default, this is the most recent version. While the retrieved version can be used as a source file for compilation, formatting, or display, it is *not* intended to be edited or changed in any way. (Attempting to bend the rules by changing permissions of a read-only version can result in your changes being lost.)

If you give a directory as a filename argument, `sccs` attempts to perform the subcommand on each `s.file` in that directory. Thus, the command:

```
sccs get SCCS
```

retrieves a read-only version for every `s.file` in the SCCS subdirectory.

`prt` — Display the version log, including comments associated with each version.

Deltas and Versions

When you check in a version, SCCS records only the line-by-line differences between the text you check in and the previous version. This set of differences is known as a *delta*. The version that is retrieved by an edit or get is constructed from the accumulated deltas checked in so far.

The terms "delta" and "version" are often used synonymously. However, their meanings aren't exactly the same; it is possible to retrieve a version that omits selected deltas (see "Excluding Deltas from a Retrieved Version " on page 191).

SIDs

An SCCS delta ID, or SID, is the number used to represent a specific delta. This is a two-part number, with the parts separated by a dot (.). The SID of the initial delta is 1.1 by default. The first part of the SID is referred to as the *release* number, and the second, the *level* number. When you check in a delta, the level number is incremented automatically. The release number can be incremented as needed. SCCS also recognizes two additional fields for *branch* deltas (described under "Branches " on page 196).

Strictly speaking, an SID refers directly to a delta. However, it is often used to indicate the version constructed from a delta and its predecessors.

ID Keywords

SCCS recognizes and expands certain keywords in a source file, which you can use to include version-dependent information (such as the SID) into the text of the checked-in version. When the file is checked out for editing, ID keywords take the following form:

```
%C%
```

where *C* is a capital letter. When you check in the file, SCCS replaces the keywords with the information they stand for. For example, %I% expands to the SID of the current version.

You would typically include ID keywords either in a comment or in a string definition. If you do not include at least one ID keyword in your source file, SCCS issues the diagnostic:

```
No Id Keywords (cm7)
```

For more information about ID keywords, refer to "Incorporating Version-Dependent Information by Using ID Keywords " on page 186.

sccs Subcommands

Checking Files In and Out

The following subcommands are useful when retrieving versions or checking in changes.

Checking Out a File for Editing: sccs edit

To edit a source file, you must check it out first using sccs edit. (The sccs edit command is equivalent to using the –c option to sccs get.)

SCCS responds with the delta ID of the version just retrieved, and the delta ID it assigna when you check in your changes.

```
$ sccs edit program.c
1.1
new delta 1.2
87
```

You can then edit it using a text editor. If a writable copy of the file is present, `sccs edit` issues an error message; it does not overwrite the file if anyone has write access to it.

Checking in a New Version: `sccs delta`

Having first checked out your file and completed your edits, you can check in the changes using `sccs delta`.

Checking a file in is also referred to as *making a delta*. Before checking in your updates, SCCS prompts you for comments. These typically include a brief summary of your changes.

```
$ sccs delta program.c
comments?
```

Comments should be meaningful, since you may return to the file one day.

You can extend the comment to an additional input line by preceding the NEWLINE with a backslash:

```
$ sccs delta program.c
comments? corrected typo in widget(), \
null pointer in n_crunch()
1.2
5 inserted
3 deleted
84 unchanged
```

SCCS responds by noting the SID of the new version, and the numbers of lines inserted, deleted and unchanged. Changed lines count as lines deleted and inserted. SCCS removes the working copy. You can retrieve a read-only version using `sccs get`.

Think ahead before checking in a version. Creating deltas after each minor edit can become excessive. On the other hand, leaving files checked out for so long that you forget about them can inconvenience others.

It is important to check in all changed files before compiling or installing a module for general use. A good technique is to:

- Edit the files you need.
- Make all necessary changes and tests.
- Compile and debug the files until you are satisfied.
- Check them in, retrieve read-only copies with get.
- Recompile the module.

Retrieving a Version: sccs get

To get the most recent version of a file, use the command:

```
sccs get filename
```

For example:

```
$ sccs get program.c
1.2
86
```

retrieves program.c, and reports the version number and the number of lines retrieved. The retrieved copy of program.c has permissions set to read-only.

Do not change this copy of the file, since SCCS does *not* create a new delta unless the file has been checked out. If you force changes into the retrieved copy, you may lose them the next time someone performs an sccs get or an sccs edit on the file.

Reviewing Pending Changes: sccs diffs

Changes made to a checked-out version, which are not yet checked in, are said to be *pending*. When editing a file, you can find out what your pending changes are using sccs diffs. The diffs subcommand uses diff(1) to compare your working copy with the most recently checked-in version.

```
$ sccs diffs program.c
------ program.c ------
37c37
<         if (((cmd_p - cmd) + 1) == l_lim) {
---
>         if (((cmd_p - cmd) - 1) == l_lim) {
```

Most of the options to diff can be used. To invoke the –c option to diff, use the –C argument to sccs diffs.

Deleting Pending Changes: sccs unedit

sccs unedit backs out pending changes. This comes in handy if you damage the file while editing it and want to start over. unedit removes the checked-out version, unlocks the history file, and retrieves a read-only copy of the most recent version checked in. After using unedit, it is as if you had not checked out the file at all. To resume editing, use sccs edit to check the file out again. (See also "Repairing a Writable Copy: sccs get -k -G " on page 186.)

Combining delta and get: sccs delget

sccs delget combines the actions of delta and get. It checks in your changes and then retrieves a read-only copy of the new version. However, if SCCS encounters an error during the `delta`, it does not perform the `get`. When processing a list of file names, `delget` applies all the deltas it can, and if errors occur, omits all of the get actions.

Combining `delta` and `edit`: `sccs deledit`

`sccs deledit` performs a delta followed by an `edit`. You can use this to check in a version and immediately resume editing.

Retrieving a Version by SID: `sccs get -r`

The `–r` option allows you to specify the `SID` to retrieve:

```
$ sccs get -r1.1 program.c
1.1
87
```

Retrieving a Version by Date and Time: `sccs get -c`

In some cases, you do not know the `SID` of the delta you want, but you do know the date on (or before) it was checked in. You can retrieve the latest version checked in before a given date and time using the `–c` option and a date-time argument of the form:

```
-cyy [mm [dd [hh [mm [ss ]]]]]
```

For example:

```
$ sccs get -c880722120000 program.c
1.2
86
```

retrieves whatever version was current as of July 22, 1988 at 12:00 noon. Trailing fields can be omitted (defaulting to their highest legal value), and punctuation can be inserted in the obvious places; for example, the above line could be written as:

```
sccs get -c"88/07/22 12:00:00" program.c
```

Note - Year 2000 issue. SCCS continues to use a two-digit year representation in the date format. Sun has adopted the proposed specification (XCU5) from the X/Open group that states that values of "yy" from 69 through 99 are to be interpreted as 1969 through 1999 respectively, and values of "yy" from 00 through 68 are to be interpreted as 2000 through 2068 respectively.

Repairing a Writable Copy: `sccs get -k -G`

Without checking out a new version, `sccs get -k -Gfilename` retrieves a writable copy of the text, and places it in the file specified by '–`G`'. This can be useful when you want to replace or repair a damaged working copy using `diff` and your editor.

Incorporating Version-Dependent Information by Using ID Keywords

As mentioned above, `SCCS` allows you to include version-dependent information in a checked-in version through the use of *ID keywords*. These keywords, which you insert in the file, are automatically replaced by the corresponding information when you check in your changes. `SCCS ID` keywords take the form:

```
%C%
```

where *C* is an uppercase letter.

For instance, `%I%` expands to the `SID` of the most recent delta. `%W%` includes the filename, the `SID`, and the unique string `@(#)` in the file. This string is searched for by the `what` command in both text and binary files (allowing you to see from which source versions a file or program was built). The `%G%` keyword expands to the date of the latest delta. Other `ID` keywords and the strings they expand to are listed in Table 5–1.

Note - Defining a string in this way allows version information to be compiled into the C object file. If you use this technique to put `ID` keywords into header (`.h`) files, use a different variable in each header file. This prevents errors from attempts to redefine the (static) variables.

To include version dependent information in a C program, use a line such as:

```
static char SccsId[ ] = "%W%\t%G%";
```

If the file were named `program.c`, this line would expand to the following when version 1.2 is retrieved:

```
static char SccsId[ ] = "@(#)program.c 1.2 08/29/80";
```

Since the string is defined in the compiled program, this technique allows you to include source-file information within the compiled program, which the `what` command can report:

```
$ cd /usr/ucb
$ what sccs
sccs
sccs.c 1.13 88/02/08 SMI
```

For shell and similar scripts, you can include ID keywords within comments:

```
# %W%  %G%
.  .  .
```

If you check in a version containing *expanded* keywords, the version-dependent information will no longer be updated. To alert you to this situation, SCCS gives you the warning:

```
No Id Keywords (cm7)
```

when a `get`, `edit`, or `create` finds no ID keywords.

Making Inquiries

The following subcommands are useful for inquiring about the status of a file or its history.

Seeing Which Version Has Been Retrieved: The `what` Command

Since SCCS allows you (or others) to retrieve any version in the file history, there is no guarantee that a working copy present in the directory reflects the version you want. The `what` command scans files for SCCS ID keywords. It also scans binary files for keywords, allowing you to see from which source versions a program was compiled.

```
$ what program.c program
program.c:
      program.c 1.1 88/07/05 SMI;
program:
      program.c 1.1 88/07/05 SMI;
```

In this case, the file contains a working copy of version 1.1.

Determining the Most Recent Version: sccs get -g

To see the SID of the latest delta, you can use `sccs get -g`:

```
$ sccs get -g program.c
1.2
```

In this case, the most recent delta is 1.2. Since this is more recent than the version reflected by `what` in the example above, you would probably want to use `get` for the new version.

Determining Who Has a File Checked Out: `sccs info`

To find out what files are being edited, type:

```
sccs info
```

This subcommand displays a list of all the files being edited, along with other information, such as the name of the user who checked out the file. Similarly, you can use

```
sccs check
```

which silently returns a non-zero exit status if anything is being edited. This can be used within a `makefile` to force `make(1S)` to halt if it should find that a source file is checked out.

If you know that all the files you have checked out are ready to be checked in, you can use the following to process them all:

```
sccs delta 'sccs tell -u'
```

`tell` lists only the names of files being edited, one per line. With the –`u` option, `tell` reports only those files checked out to you. If you supply a username as an argument to –`u`, `sccs tell` reports only the files checked out to that user.

Displaying Delta Comments: `sccs prt`

`sccs prt` produces a listing of the version log, also referred to as the *delta table*, which includes the `SID`, time and date of creation, and the name of the user who checked in each version, along with the number of lines inserted, deleted, and unchanged, and the commentary:

```
$ sccs prt program.c
D 1.2  80/08/29 12:35:31      pers  2 1 00005/00003/00084
corrected typo in widget(),
null pointer in n_crunch()

D 1.1  79/02/05 00:19:31      zeno  1 0 00087/00000/00000
date and time created 80/06/10 00:19:31 by zeno
```

To display only the most recent entry, use the -y option.

Updating a Delta Comment: `sccs cdc`

If you forget to include something important in a comment, you can add the missing information using:

```
sccs cdc -r sid
```

The delta must be the most recent (or the most recent in its branch, see "Branches " on page 196). Also, you must either be the user who checked the delta in, or you must own and have permission to write on both the history file and the SCCS subdirectory. When you use cdc, SCCS prompts for your comments and inserts the new comment you supply:

```
$ sccs cdc -r1.2 program.c
comments? also taught get_in() to handle control chars
```

The new commentary, as displayed by prt, looks like this:

```
$ sccs prt program.c
D 1.2  80/08/29 12:35:31       pers  2 1 00005/00003/00084
also taught get_in() to handle control chars
*** CHANGED *** 88/08/02 14:54:45 pers
corrected typo in widget(),
null pointer in n_crunch()

D 1.1  79/02/05 00:19:31       zeno  1 0 00087/00000/00000
date and time created 80/06/10 00:19:31 by zeno
```

Comparing Checked-In Versions: sccs sccsdiff

To compare two checked-in versions, use the following to see the differences between delta 1.1 and delta 1.2.

```
$ sccs sccsdiff -r1.1 -r1.2 program.c
```

Displaying the Entire History: sccs get -m -p

If you want to see a listing of all changes made to the file and the delta in which each was made, you can use the –m and –p options to get:

```
$ sccs get -m -p program.c
1.2
1.2 #define L_LEN 256
1.1
1.1 #include <stdio.h>
1.1
.  .  .
84
```

To find out what lines are associated with a particular delta, you can pipe the output through grep(1V):

```
$ sccs get -m -p program.c | grep ’^1.2’
```

You can also use –p by itself to send the retrieved version to the standard output, rather than to the file.

Creating Reports: sccs prs -d

You can use the prs subcommand with the –d *dataspec* option to derive reports about files under SCCS control. The *dataspec* argument offers a rich set of *data keywords* that correspond to portions of the history file. Data keywords take the form:

```
:X :
```

and are listed in Table 5–3. There is no limit to the number of times a data keyword can appear in the *dataspec* argument. A valid *dataspec* argument is a (quoted) string consisting of text and data keywords. `prs` replaces each recognized keyword with the appropriate value from the history file.

The format of a data keyword value is either simple, in which case the expanded value is a simple string, or multiline, in which case the expansion includes RETURN characters.

A TAB is specified by ‘`\t`’ and a RETURN by ‘`\n`’.

Here are some examples:

```
$ sccs prs -d"Users and/or user IDs for :F: are:\n:UN:" program.c
Users and/or user IDs for s.program.c are:
zeno
pers
$ sccs prs -d"Newest delta for :M:: :I:.  Created :D: by :P:." -r program.c
Newest delta for program.c: 1.3.  Created 88/07/22 by zeno.
```

Deleting Committed Changes

Replacing a Delta: sccs fix

Occasionally, a delta is checked in that contains small bugs, such as typographical errors, that need correcting but that do not require entries in the file audit trail. Or, perhaps the comment for a delta is incomplete or in error, even when the text is correct. In either case, you can make additional updates and replace the version log entry for the most recent delta using `sccs fix`:

```
$ sccs fix -r 1.2 program.c
```

This checks out version 1.2 of `program.c.` When you check the file back in, the current changes replaces delta 1.2 in the history file, and SCCS prompts for a (new) comment. You must supply an SID with `–r`. Also, the delta that is specified must be a leaf (most recent) delta.

Although the previously-checked-in delta 1.2 is effectively deleted, SCCS retains a record of it, marked as deleted, in the history file.

Before using `sccs fix` it is a good idea to make a copy of the current version.

Removing a Delta: sccs rmdel

To remove all traces of the most recent delta, you can use the `rmdel` subcommand. You must specify the SID using `–r`. In most cases, using `fix` is preferable to `rmdel`, since fix preserves a record of “deleted” delta, while rmdel does not (refer to sccs rmdel(1) for more information).

Reverting to an Earlier Version

To retrieve a writable copy of an earlier version, use get -k. This can be useful when you need to back track past several deltas.

To use an earlier delta as the basis for creating a new one:

1. **Check out the file as you normally would (using sccs edit).**

2. **Retrieve a writable copy of an earlier “good” version (giving it a different file name) using get –k:**

```
sccs get -k -r sid -Goldname filename
```

   The –G*oldname filename* option specifies the name of the newly retrieved version.

3. **Replace the current version with the older “good” version:**

   mv *oldname filename*

4. **Check the file back in.**

   In some cases, it may be simpler just to exclude certain deltas. Or refer to “Branches ” on page 196 for information on how to use SCCS to manage divergent sets of updates to a file.

Excluding Deltas from a Retrieved Version

Suppose that the changes that were made in delta 1.3 aren’t applicable to the next version, 1.4. When you retrieve the file for editing, you can use the -x option to *exclude* delta 1.3 from the working copy:

```
$ sccs edit -x1.3 program.c
```

When you check in delta 1.5, that delta will include the changes made in delta 1.4, but not those from delta 1.3. In fact, you can exclude a list of deltas by supplying a comma-separated list to -x, or a range of deltas, separated with a dash. For example, if you want to exclude 1.3 and 1.4, you could use:

```
$ sccs edit -x1.3,1.4 program.c
```

or

```
$ sccs edit -x1.3-1.4 program.c
```

In this example SCCS excludes the range of deltas from 1.3 to the current highest delta in release 1:

```
$ sccs edit -x 1.3-1 program.c
```

In certain cases when using –x there will be conflicts between versions; for example, it may be necessary to both include and delete a particular line. If this happens, SCCS displays a message listing the range of lines affected. Examine these lines carefully to see if the version SCCS derived is correct.

Since each delta (in the sense of "a set of changes") can be excluded at will, it is most useful to include a related set of changes within each delta.

Combining Versions: sccs comb

The comb subcommand generates a Bourne shell script that, when run, constructs a new history file in which selected deltas are combined or eliminated. This can be useful when disk space is at a premium.

Note - In combining several deltas, the comb-generated script destroys a portion of the file's version log, including comments.

The –p*sid* option indicates the oldest delta to preserve in the reconstruction. Another option,

-c *sid-list*

allows you to specify a list of deltas to include. *sid-list* is a comma-separated list; you can specify a range between two SIDs by separating them with a dash ('-') in the list. –p and –c are mutually exclusive. The –o option attempts to minimize the number of deltas in the reconstruction.

The –s option produces a script that compares the size of the reconstruction with that of the original. The comparison is given as a percentage of the original the reconstruction would occupy, based on the number of blocks in each.

Note - When using comb, it is a good idea to keep a copy of the original history file on hand. While comb is intended to save disk space, it does not always work. In some cases, it is possible that the resulting history file might be larger than the original.

If no options are specified, comb preserves the minimum number of ancestors needed to preserve the changes made so far.

Version Control for Binary Files

Although SCCS is typically used for source files containing ASCII text, this version of SCCS allows you to apply version control to *binary* files as well (files that contain NULL or control characters, or do not end with a NEWLINE. The binary files are encoded into an ASCII representation when checked in; working copies are decoded when retrieved. (See uuencode(1C) for details.)

You can use SCCS to track changes to files such as icons, raster images, and screen fonts.

You can use `sccs create -b` to force SCCS to treat a file as a binary file. When you use create or delta for a binary file, you get the warning message:

```
Not a text file (ad31)
```

You may also get the message:

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

These messages can be safely ignored. Otherwise, everything proceeds as expected:

```
$ sccs create special.font

special.font:
Not a text file (ad31)
No id keywords (cm7)
1.1
20
No id keywords (cm7)
$ sccs get special.font
1.1
20
$ file special.font SCCS/s.special.font
special.font:    vfont definition
SCCS/s.special.font: sccs
```

Note - Use SCCS to control the updates to source files, and make to compile objects consistently

---

Because the encoded representation of a binary file can vary significantly between versions, history files for binary sources can grow at a much faster rate than those for ASCII sources. However, using the same version control system for all source files makes dealing with them much easier.

Maintaining Source Directories

When using SCCS, it is the history files, and not the working copies, that are the real source files.

Duplicate Source Directories

If you are working on a project and want to create a duplicate set of sources for some private testing or debugging, you can create a symbolic link to the SCCS subdirectory in your private working directory:

```
$ cd /private/working/cmd.dir
$ ln -s /usr/src/cmd/SCCS SCCS
```

This enables you to retrieve a private (duplicate) set of working copies, of the source files using:

```
sccs get SCCS
```

While working in the duplicate directory, you can also check files in and out—just as you could if you were in the original directory.

SCCS and make

SCCS is often used with make(1S) to maintain a software project. make provides for automatic retrieval of source files. (Other versions of make provide special rules that accomplish the same purpose.) It is also possible to retrieve earlier versions of all the source files, and to use make to rebuild earlier versions of the project:

```
$ mkdir old.release ; cd old.release
$ ln -s ../SCCS SCCS
$ sccs get -c"87/10/01" SCCS
SCCS/s.Makefile:
1.3
47
. . .
$ make
. . .
```

As a general rule, no one should check in source files while a build is in progress. When a project is about to be released, all files should be checked in before a build. This ensures that the sources for a released project are stable.

Keeping SIDs Consistent Across Files

With some care, it is possible to keep the SIDs consistent across sources composed of multiple files. The trick here is to edit all the files at once. The changes can then be made to whatever files are necessary. Check in all the files (even those not changed). This can be done fairly easily by specifying the SCCS subdirectory as the file name argument to both edit and delta:

```
$ sccs edit SCCS
. . .
$ sccs delta SCCS
```

With the delta subcommand, you are prompted for comments only once; the comment is applied to all files being checked in. To determine which files have changed, you can compare the “lines added, deleted, unchanged” fields in each file delta table.

Starting a New Release

To create a new release of a program, specify the release number you want to create when you check out the file for editing, using the -r*n* option to edit; *n* is the new release number:

```
$ sccs edit -r 2 program.c
```

In this case, when you use delta with the new version, it will be the first level delta in release 2, with SID 2.1. To change the release number for all SCCS files in the directory, use:

```
$ sccs edit -r 2 SCCS
```

Temporary Files Used by SCCS

When SCCS modifies an `s.file` (that is, a history file), it does so by writing to a temporary copy called an `x.file`. When the update is complete, SCCS uses the `x.file` to overwrite the old `s.file`. This ensures that the history file is not damaged when processing terminates abnormally. The `x.file` is created in the same directory as the history file, is given the same permissions, and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, subcommands that update the history create a lock file, called a `z.file`, which contains the PID of the process performing the update. Once the update has completed, the `z.file` is removed. The `z file` is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user.

Branches

You can think of the deltas applied to anSCCS file as the nodes of a tree; the root is the initial version of the file. The root delta (node) is number '1.1' by default, and successor deltas (nodes) are named 1.2, 1.3, and so forth. As noted earlier, these first two parts of the SID are the release and level numbers. The naming of a successor to a delta proceeds by incrementing the level number. You have also seen how to check out a new release when a major change to the file is made. The new release number applies to all successor deltas as well, unless you specify a new level in a prior release.

Thus, the evolution of a particular file may be represented in the following figure:

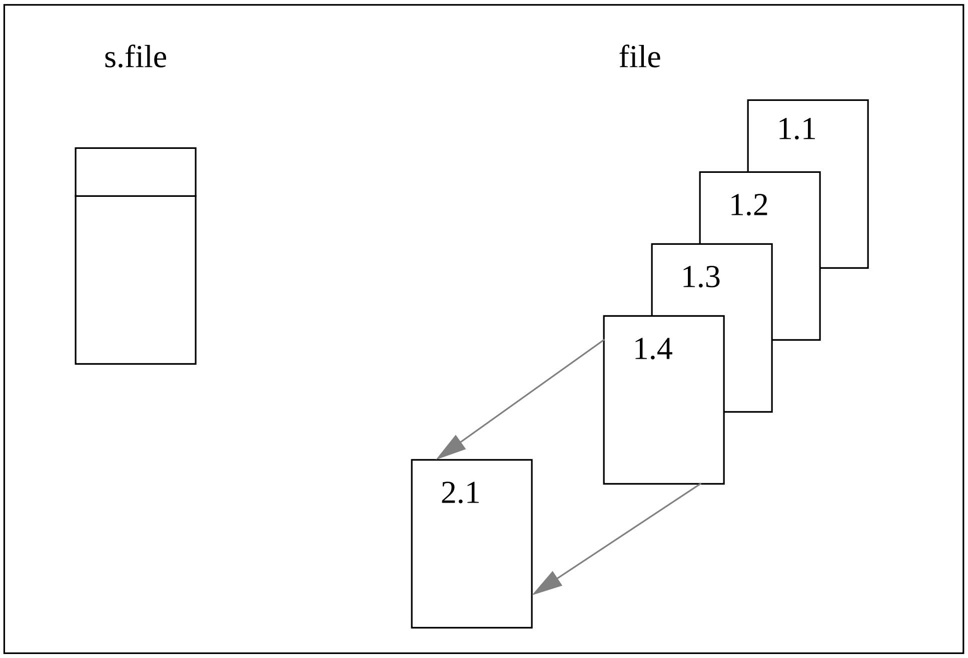


Figure 5–1 Evolution of an SCCS File

This structure is called the trunk of the SCCS delta tree. It represents the normal sequential development of an SCCS file; changes that are part of any given delta depend upon all the preceding deltas.

However, situations can arise when it is convenient to create an alternate branch on the tree. For instance, consider a program that is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 might already have some deltas.

Assume that a user reports a problem in version 1.3 which cannot wait until release 2 to be corrected. The changes necessary to correct the problem will have to be applied as a delta to version 1.3. This requires the creation of a new version, but one that is independent of the work being done for release 2. The new delta thus occupies a node on a new branch of the tree.

The `SID` for a branch delta consists of four parts: the release and level numbers, and the `branch` and `sequence` numbers:

```
release.level.branch.sequence
```

The `branch` number is assigned to each branch that is a descendant of a particular trunk delta; the first such branch is 1, the next one 2, and so on. The `sequence` number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.1 identifies the first delta of the first branch derived from delta 1.3, as shown in the following figure.

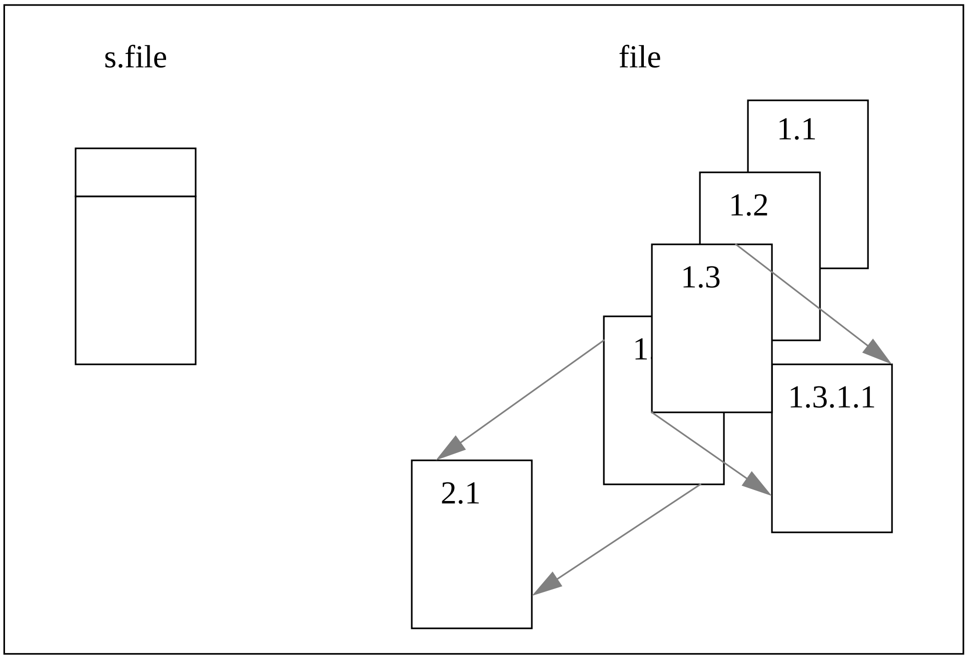


Figure 5–2 Tree Structure with Branch Deltas

The concept of branching might be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated. The first two components of the name of a branch delta are always those of the ancestral trunk delta.

The branch component is assigned in the order of creation on the branch, independent of its location relative to the trunk. Thus, a branch delta can always be identified as such from its name, and while the trunk delta can be identified from the branch delta name, it is *not* possible to determine the entire path leading from the trunk delta to the branch delta.

For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.*n*. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.*n*.

The only information that may be derived from the name of delta 1.3.2.2 is that it is the second chronological delta on the second chronological branch whose trunk ancestor is delta 1.3.

In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

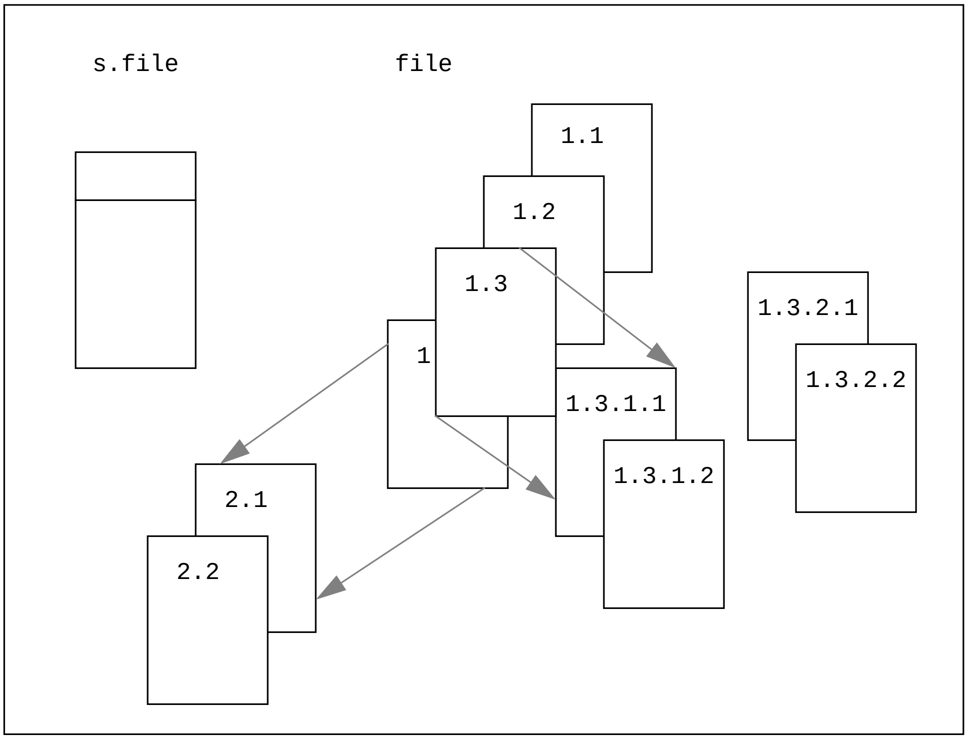


Figure 5–3 Extending the Branching Concept

Branch deltas allow the generation of arbitrarily complex tree structures. It is best to keep the use of branches to a minimum.

Using Branches

You can use branches when you need to keep track of an alternate version developed in parallel, such as for bug fixes or experimental purposes. Before you can create a branch, however, you must enable the “branch” flag in the history file using the sccs admin command, as follows:

```
$ sccs admin -f b program.c
```

The –fb option sets the b (branch) flag in the history file.

Creating a Branch Delta

To create a branch from delta 1.3 for program.c, you would use the sccs edit subcommand shown in the following figure:

```
$ sccs edit -r 1.3 -b program.c
```

When you check in your edited version, the branch delta contains SID 1.3.1.1. Subsequent deltas made from this branch are numbered 1.3.1.2, and so on.

Retrieving Versions from Branch Deltas

Branch deltas usually are not included in the version retrieved by get. To retrieve a branch version (the version associated with a branch delta), you must specifically request it with the –r option. If you omit the sequence number, as in the next example, SCCS retrieves the highest delta in the branch:

```
$ sccs get -r 1.3.1 program.c
1.3.1.1
87
```

Merging a Branch Back into the Main Trunk

At some point, perhaps when you’ve finished with the experiment, you might want to introduce the experimental features into production. But in the meantime, work may have progressed on the production version, in which case there may be incompatibilities between the branch version and the latest trunk version.

To help you sort this situation out, the –i option to sccs edit allows you to specify a list of deltas to include when you check out the file. If any of the changes that were included result in conflicts, edit issues a warning message. A conflict can arise if a line must be deleted to satisfy one delta, but inserted to satisfy another. While it is up to you to resolve each conflict, knowing where they are is a big help.

Administering SCCS Files

By convention, history files and all temporary SCCS files reside in the SCCS subdirectory. In addition to the standard file protection mechanisms, SCCS allows certain releases to be frozen, and access to releases to be restricted to certain users (see `sccs admin(1)` for details). History files normally have permissions set to 444 (read-only for everyone), to prevent modification by utilities other than SCCS. In general, it is not a good idea to edit the history files.

A history file should have just one link. SCCS utilities update the history file by writing out a modified copy (`x.file`), then renaming the copy.

Interpreting Error Messages: `sccs help`

The `help` subcommand displays information about SCCS error messages and utilities.

`help` normally expects either the name of an SCCS utility, or the code (in parentheses) from an SCCS error message. If you supply no argument, help prompts for one. The directory `/usr/ccs/lib/help` contains files with the text of the various messages `help` displays.

Altering History File Defaults: `sccs admin`

A number of parameters that can be set using the `admin` command. The most interesting of these are flags. Flags can be added by using the `–f` option.

For example, the following command sets the d flag to the value 1:

```
$ sccs admin -f d1 program.c
```

This flag can be deleted by using:

```
$ sccs admin -d d program.c
```

The most useful flags are:

`b` — Allow branches to be made using the `–b` option to `sccs edit` (see "Branches " on page 196).

`d`***SID*** — Default SID to be used on an `sccs get` or `sccs edit` . If this is just a release number it constrains the version to a particular release only.

`i` — Give a fatal error if there are no `ID` keywords in a file. This prevents a version from being checked in when the `ID` keywords are missing or expanded by mistake.

`y` — The value of this flag replaces the `%Y% ID` keyword.

–t*file*	Stores descriptive text from *file* in the s.file. This descriptive text might be the documentation or a design and implementation document. Using the –t option ensures that if the s.file is passed on to someone else, the documentation goes along with it. If *file* is omitted, the descriptive text is deleted. To see the descriptive text, use prt -t.

The sccs admin command can be used safely any number of times on files. A current version need not be retrieved for admin to work.

Validating the History File

You can use the val subcommand to check certain assertions about a history file. val always checks for the following conditions:

- Corrupted history file.
- The history file can't be opened for reading
- The file is not in SCCS history.

If you use the –r option, val checks to see if the indicated SID exists.

Restoring the History File

In particularly bad circumstances, the history file itself can become corrupted, usually by someone editing it. Since the file contains a checksum, you will get errors every time you read a corrupted file. To correct the checksum, use:

```
$ sccs admin -z program.c
```

Note - When SCCS says that the history file is corrupted, it may indicate serious damage beyond an incorrect checksum. Be careful to safeguard your current changes before attempting to correct a history file.

Reference Tables

TABLE 5–1 SCCS ID Keywords

Keyword	Expands to
%Z%	@(#) (search string for the what command)
%M%	The current module (file) name
%I%	The highest SID applied
%W%	shorthand for: %Z%%M% tab %I%
%G%>	The date of the delta corresponding to the %I% keyword
%R%	The current release number.
%Y%	The value of the –t flag (set by sccs admin).

TABLE 5–2 SCCS Utility Commands

SCCS Utility Programs	
Command	**Refer to:**
admin	sccs-admin(1)
cdc	sccs-cdc(1)
comb	sccs-comb(1)
delta	sccs-delta(1)
get	sccs-get(1)
help	sccs-help(1)
prs	sccs-prs(1)
prt	sccs-prt(1)
rmdel	sccs-rmdel(1)
sact	sccs-sact(1)
sccsdiff	sccs-sccsdiff(1)
unget	sccs-unget(1)

TABLE 5–2 SCCS Utility Commands *(continued)*

SCCS Utility Programs	
Command	**Refer to:**
val	sccs-val(1)
what	what(1)

TABLE 5–3 Data Keywords for prs -d

Keyword	Data Item	File Section	Value	Format
:Dt:	Delta information	Delta Table	:Dt: = :DT: :I: :D: :T: :P: :DS: :DP:	Single line
:DL:	Delta line statistics	Delta Table	:Li:/:Ld:/:Lu:	Single line
:Li:	Lines inserted by Delta	Delta Table	*nnnnn*	Single line
:Ld:	Lines deleted by Delta	Delta Table	*nnnnn*	Single line
:Lu:	Lines unchanged by Delta	Delta Table	*nnnnn*	Single line
:DT:	Delta type	Delta Table	D or R	Single line
:I:	SCCS ID string (SID)	Delta Table	:Rf3:.:Lf3:.:Bf3:.:S:	Single line
:R:	Release number	Delta Table	*nnnn*	Single line
:L:	Level number	Delta Table	*nnnn*	Single line
:B:	Branch number	Delta Table	*nnnn*	Single line
:S:	Sequence number	Delta Table	*nnnn*	Single line
:D:	Date Delta created	Delta Table	:Dy:/:Dm:/:Dd:	Single line
:Dy:	Year Delta created	Delta Table	*nn*	Single line
:Dm:	Month Delta created	Delta Table	*nn*	Single line
:Dd:	Day Delta created	Delta Table	*nn*	Single line
:T:	Time Delta created	Delta Table	:Th:::Tm:::Ts:	Single line

TABLE 5–3 Data Keywords for prs -d *(continued)*

Keyword	Data Item	File Section	Value	Format
:Th:	Hour Delta created	Delta Table	*nn*	Single line
:Tm:	Minutes Delta created	Delta Table	*nn*	Single line
:Ts:	Seconds Delta created	Delta Table	*nn*	Single line
:P:	Programmer who created Delta	Delta Table	*logname*	Single line
:DS:	Delta sequence number	Delta Table	*nnnn*	Single line
:DP:	Predecessor Delta seq-no.	Delta Table	*nnnn*	Single line
:DI:	Sequence number of deltas included, excluded, ignored	Delta Table	:Dn:/:Dx:/:Dg:	Single line
:Dn:	Deltas included (seq #)	Delta Table	:DS: :DS: . . .	Single line
:Dx:	Deltas excluded (seq #)	Delta Table	:DS: :DS: . . .	Single line
:Dg:	Deltas ignored (seq #)	Delta Table	:DS: :DS: . . .	Single line
:MR:	MR numbers for delta	Delta Table	*text*	Multiple line
:C:	Comments for delta	Delta Table	*text*	Multiple line
:UN:	User names	User Names	*text*	Multiple line
:FL:	Flag list	Flags	*text*	Multiple line
:Y:	Module type flag	Flags	*text*	Multiple line
:MF:	MR validation flag	Flags	yes or no	Multiple line
:MP:	MR validation pgm name	Flags	*text*	Single line
:KF:	Keyword error/warning flag	Flags	yes or no	Single line
:BF:	Branch flag	Flags	yes or no	Single line
:J:	Joint edit flag	Flags	yes or no	Single line
:LK:	Locked releases	Flags	:R: . . .	Single line
:Q:	User defined keyword	Flags	*text*	Single line

TABLE 5–3 Data Keywords for prs -d *(continued)*

Keyword	Data Item	File Section	Value	Format
:M:	Module name	Flags	*text*	Single line
:FB:	Floor boundary	Flags	:R:	Single line
:CB:	Ceiling boundary	Flags	:R:	Single line
:Ds:	Default SID	Flags	:I:	Single line
:ND:	Null delta flag	Flags	yes or no	Single line
:FD:	File descriptive text	Comments	*text*	Multiple line
:BD:	Body	Body	*text*	Multiple line
:GB:	Gotten body	Body	*text*	Multiple line
:W:	A form of what(1) string	N/A	:Z::M:\t:I:	Single line
:A:	A form of what(1) string	N/A	:Z::Y: :M: :I::Z:	Single line
:Z:	what(1) string delimiter	N/A	@(#)	Single line
:F:	SCCS file name	N/A	*text*	Single line
:PN:	SCCS file path name	N/A	*text*	Single line

CHAPTER 6

m4 Macro Processor

Overview

m4 is a general-purpose macro processor that can be used to preprocess C and assembly language programs. Besides the straightforward replacement of one string of text by another, m4 enables you to perform

- Integer arithmetic
- File inclusion
- Conditional macro expansion
- String and substring manipulation

You can use built-in macros to perform these tasks or you can define your own macros. Built-in and user-defined macros work exactly the same way except that some of the built-in macros have side effects on the state of the process.

The basic operation of m4 is to read every alphanumeric token (string of letters and digits) and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is replaced onto the input to be rescanned. Macros can be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

Macro calls have the general form:

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

If a macro name is not immediately followed by a left parenthesis, it is assumed to have no arguments. Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses that appear in the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is returned to the input stream and rescanned. This is explained in the following paragraphs.

You invoke `m4` with a command of the form

```
$ m4 file file file
```

Each argument file is processed in order. If there are no arguments or if an argument is a hyphen, the standard input is read. If you are eventually going to compile the `m4` output, use a command like this:

```
$ m4 file1.m4 > file1.c
```

You can use the –`D` option to define a macro on the `m4` command line. Suppose you have two similar versions of a program. You might have a single `m4` input file capable of generating the two output files. That is, `file1.m4` could contain lines such as:

```
if(VER, 1, do_something)
if(VER, 2, do_something)
```

Your `makefile` for the program might look like this:

```
file1.1.c : file1.m4
m4 -DVER=1 file1.m4 > file1.1.c
...
file1.2.c : file1.m4
m4 -DVER=2 file1.m4 > file1.2.c
...
```

You can use the –`U` option to ‘‘undefine’’ `VER`. If `file1.m4` contains

```
if(VER, 1, do_something)
if(VER, 2, do_something)
ifndef(VER, do_something)
```

then your `makefile` would contain

```
file0.0.c : file1.m4
m4 -UVER file1.m4 > file1.0.c
...
file1.1.c : file1.m4
m4 -DVER=1 file1.m4 > file1.1.c
...
file1.2.c : file1.m4
m4 -DVER=2 file1.m4 > file1.2.c
...
```

m4 Macros

Defining Macros

The primary built-in m4 macro is define( ), which is used to define new macros. The following input:

```
define(
name, stuff)
```

causes the string name to be defined as stuff. All subsequent occurrences of name are replaced by stuff.The defined string must be alphanumeric and must begin with a letter (an underscore is considered to be a letter). The defining string is any text that contains balanced parentheses; it may stretch over multiple lines.

As a typical example

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

defines N to be 100 and uses the *symbolic constant* N in a later if statement.

As noted, the left parenthesis must immediately follow the word define to signal that define( ) has arguments. If the macro name is not immediately followed by a left parenthesis, it is assumed to have no arguments. In the previous example, then, N is a macro with no arguments.

A macro name is only recognized as such if it appears surrounded by non-alphanumeric characters. In the following example, the variable *NNN* is unrelated to the defined macro N, even though the variable contains Ns.

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

m4 expands macro names into their defining text as soon as possible. So

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

defines M to be 100 because the string N is immediately replaced by 100 as the arguments of define(M, N) are collected. To put this another way, if N is redefined, M keeps the value 100.

There are two ways to avoid this result. The first, which is specific to the situation described here, is to change the order of the definitions:

```
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 always the value of N at that time. The M is replaced by N which is replaced by 100.

Quoting

The more general solution is to delay the expansion of the arguments of define() by quoting them. Any text surrounded by left and right single quotes is not expanded immediately, but has the quotes stripped off as the arguments are collected. The value of the quoted string is the string stripped of the quotes.

Therefore, the following defines M as the string N, not 100.

```
define(N, 100)
define(M, ‘N’)
```

The general rule is that m4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If the word define is to appear in the output, the word must be quoted in the input:

```
‘define’ = 1;
```

It is usually best to quote the arguments of a macro to ensure that what you are assigning to the macro name actually gets assigned. To redefine N, for example, you delay its evaluation by quoting:

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

Otherwise the N in the second definition is immediately replaced by 100.

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

The effect is the same as saying:

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

Note that this statement will be ignored by m4 since only things that look like names can be defined.

If left and right single quotes are not convenient, the quote characters can be changed with the built-in macro changequote():

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

In this example the macro makes the “quote” characters the left and right brackets instead of the left and right single quotes. The quote symbols can be up to five characters long. The original characters can be restored by using `changequote()` without arguments:

```
changequote
```

`undefine()` removes the definition of a macro or built-in macro:

```
undefine('N')
```

Here the macro removes the definition of N. Be sure to quote the argument to `undefine()`. Built-ins can be removed with `undefine()` as well:

```
undefine('define')
```

Note that after a built-in is removed or redefined, its original definition cannot be reused. Macros can be renamed with `defn()`. Suppose you want the built-in `define()` to be called `XYZ()`. You specify

```
define(XYZ, defn('define'))
undefine('define')
```

After this, `XYZ()` takes on the original meaning of `define()`. So

```
XYZ(A, 100)
```

defines A to be 100.

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)')
```

The `ifdef()` macro 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:

```
ifdef('unix', on UNIX, not on UNIX)
```

If there is no third argument, the value of `ifdef()` is null.

Arguments

So far you have been given information about the simplest form of macro processing, that is, replacing one string with another (fixed) string. Macros can also be defined so that different invocations have different results. In the replacement text for a macro (the second argument of its `define()`), any occurrence of `$n` is replaced by the nth argument when the macro is actually used. So the macro `bump()`, defined as

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

is equivalent to `x = x + 1` for `bump(x)`.

A macro can have as many arguments as you want, but only the first nine are accessible individually, $1 through $9. $0 refers to the macro name itself. As noted, arguments that are not supplied are replaced by null strings, so a macro can be defined that concatenates its arguments:

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

That is, `cat(x, y, z)` is equivalent to `xyz`. Arguments $4 through $9 are null since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained, so

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

defines `a` to be `b c`.

Arguments are separated by commas. A comma "protected" by parentheses does not terminate an argument. The following example has two arguments, `a` and `(b,c)`. You can specify a comma or parenthesis as an argument by quoting it:

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

In the following example, `$(**` is replaced by a list of the arguments given to the macro in a subsequent invocation. The listed arguments are separated by commas. So

```
define(a, 1)
define(b, 2)
define(star, '$(**')
star(a, b)
```

gives the result `1,2`. So does

```
star('a', 'b')
```

because `m4` strips the quotes from `a` and `b` as it collects the arguments of `star( )`, then expands `a` and `b` when it evaluates `star( )`.

`$@` is identical to `$(**` except that each argument in the subsequent invocation is quoted. That is,

```
define(a, 1)
define(b, 2)
define(at, '$@')
at('a', 'b')
```

gives the result `a,b` because the quotes are put back on the arguments when `at( )` is evaluated.

`$#` is replaced by the number of arguments in the subsequent invocation. So

```
define(sharp, '$#')
sharp(1, 2, 3)
```

gives the result `3`,

```
sharp()
```

gives the result `1`, and

```
sharp
```

gives the result `0`.

The built-in `shift()` returns all but its first argument. The other arguments are quoted and returned to the input with commas between. The simplest case

```
shift(1, 2, 3)
```

gives `2,3`. As with `$@`, you can delay the expansion of the arguments by quoting them, so

```
define(a, 100)
define(b, 200)
shift('a', 'b')
```

gives the result `b` because the quotes are put back on the arguments when `shift()` is evaluated.

Arithmetic Built-Ins

`m4` provides three built-in macros for doing integer arithmetic. `incr()` increments its numeric argument by 1. `decr()` decrements by 1. So, to handle the common programming situation in which a variable is to be defined as "one more than N," you would use:

```
define(N, 100)
define(N1, 'incr(N)')
```

That is, `N1` is defined as one more than the current value of `N`.

The more general mechanism for arithmetic is a built-in macro called `eval()`, which is capable of arbitrary arithmetic on integers. Its operators, in decreasing order of precedence, are

```
+ - (unary)
(**(**
(** / %
+ -
== != < <= > >=
! ~
&
| ^
&&
||
```

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.

As a simple example, you can define M to be `2(**(**N+1` with

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

Then the sequence

```
define(N, 3)
M(2)
```

gives 9 as the result.

File Inclusion

A new file can be included in the input at any time with the built-in macro `include()`:

```
include(filename)
```

inserts the contents of filename in place of the macro and its argument. The value of `include()` (its 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 in `include()` cannot be accessed. To get some control over this situation, the alternate form `sinclude()` ("silent include") can be used. This built-in says nothing and continues if the file named cannot be accessed.

Diversions

`m4` output can be diverted to temporary files during processing, and the collected material can be output on command. `m4` maintains nine of these diversions, numbered 1 through 9. If the built-in macro `divert(n)` is used, all subsequent output is appended to a temporary file referred to as n. Diverting to this file is stopped by the `divert()` or `divert(0)` macros, which resume the normal output process.

Diverted text is normally placed at the end of processing in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The built-in `undivert()` brings back all diversions in numerical order; `undivert()` with arguments brings back the selected diversions in the order given. Undiverting discards the diverted text (as does diverting) into a diversion whose number is not between 0 and 9, inclusive.

The value of `undivert()` is not the diverted text. Furthermore, the diverted material is not rescanned for macros. The built-in `divnum()` returns the number of the currently active diversion. The current output stream is 0 during normal processing.

System Commands

Any program can be run by using the `syscmd()` built-in. The following example invokes the operating system date command. Normally, `syscmd()` would be used to create a file for a subsequent `include()`.

```
syscmd(date)
```

To make it easy to name files uniquely, the built-in `maketemp()` replaces a string of *XXXXX* in the argument with the `process ID` of the current process.

Conditional Testing

Arbitrary conditional testing is performed with the built-in `ifelse()`. In its simplest form

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

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

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

Notice the quotes, which prevent evaluation of `ifelse()` from occurring too early. If the final argument is omitted, the result is null, so

```
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 branched decision capability. In the input

```
ifelse(a,b,c,d,e,f,g)
```

if the string `a` matches the string `b`, the result is `c`. Otherwise, if `d` is the same as `e`, the result is `f`. Otherwise, the result is `g`.

String Manipulation

The `len()` macro returns the length of the string (number of characters) in its argument. So

```
len(abcdef)
```

is `6`, and

```
len((a,b))
```

is `5`.

The `substr()` macro can be used to produce substrings of strings. So

```
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. When you input the following example:

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

it returns the following string:

```
ow is the time
```

If `i` or `n` are out of range, various things happen.

The `index(s1,s2)` macro returns the index (position) in `s1` where the string `s2` occurs, `-1` if it does not occur. As with `substr()`, the origin for strings is 0.

`translit()` performs character transliteration [character substitution] and has the general form

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

that modifies `s` by replacing any character in `f` by the corresponding character in `t`.

Using the following input:

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

replaces the vowels by the corresponding digits. If `t` is shorter than `f`, characters that do not have an entry in `t` are deleted. As a limiting case, if `t` is not present at all, characters from `f` are deleted from `s`.

Therefore, the following would delete vowels from *s*:

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

The macro `dnl()` deletes all characters that follow it, up to and including the next newline. It is useful mainly for removing empty lines that otherwise would clutter `m4` output. The following input, for example, results in a newline at the end of each line that is not part of the definition:

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

So the new-line is copied into the output where it might not be wanted. When you add dnl() to each of these lines, the newlines disappear. Another method of achieving the same result is to type:

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

Printing

The built-in macro `errprint()` writes its arguments on the standard error file. An example would be

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

`dumpdef()` is a debugging aid that dumps the current names and definitions of items specified as arguments. If no arguments are given, then all current names and definitions are printed.

Summary of Built-In m4 Macros

TABLE 6–1 Summary of Built-In m4 Macros

Built-In m4 Macros	Description
`changequote(L, R)`	Change left quote to `L`, right quote to `R`.
`changecom`	Change left and right comment markers from the default # and newline.
`decr`	Return the value of the argument decremented by 1.
`define(name, stuff)`	Define `name` as `stuff`.
`defn('name')`	Return the quoted definition of the argument(s)
`divert(number)`	Divert output to stream `number`
`divnum`	Return number of currently active diversions

TABLE 6–1 Summary of Built-In m4 Macros *(continued)*

Built-In m4 Macros	Description
`dnl`	Delete up to and including newline
`dumpdef('name', 'name', . . .)`	Dump specified definitions.
`errprint(s, s, . . .)`	Write arguments `s` to standard error.
`eval(numeric expression)`	Evaluate `numeric expression`.
`ifdef('name', true string, false string)`	Return `true string` if name is defined, `false string` if name is not defined.
`ifelse(a, b, c, d)`	If `a` and `b` are equal, return `c`, else return `d`.
`include(file)`	Include contents of file.
`incr(number)`	Increment number by 1.
`index(s1, s2)`	Return position in `s1` where `s2` occurs, or -1 if `s2` does not work.
`len(string)`	Return length of `string`.
`maketemp(. . .XXXXX. . .)`	Make a temporary file.
`m4 exit`	Cause immediate `exit` from `m4`.
`m4 wrap`	Argument 1 will be returned to the input stream at final `EOF`.
`popdef`	Remove current definition of argument(s).
`pushdef`	Save any previous definition (similar to `define( )`)
`shift`	Return all but first argument(s)
`sinclude(file)`	Include contents of file — ignore and continue if file not found
`substr(string, position, number)`	Return substring of `string` starting at `position` and `number` characters long
`syscmd(command)`	Run command in the system
`sysval`	Return code from the last call to `syscmd( )`
`traceoff`	Turn off trace globally and for any macros specified

TABLE 6–1 Summary of Built-In `m4` Macros *(continued)*

Built-In m4 Macros	Description
`traceon`	Turn on tracing for all macros, or with arguments, turn on tracing for named macros
`translit(string, from, to)`	Transliterate characters in `string` from the set specified by `from` to the set specified by `to`
`undefine('name')`	Remove name from the list of definitions
`undivert(number, number,. . .)`	Append diversion number to the current diversion

APPENDIX **A**

System V make

Note - This version of make (/usr/ccs/lib/svr4.make) is included for those users who have makefiles that rely on the older, System V version of make. However, it is recommended that you use the default version of make (/usr/ccs/bin/make) where possible. See also Chapter 4, on Chapter 4.

To use this version of make, set the system variable USE_SVR4_MAKE:

```
$ USE_SVR4_MAKE='''' ; export USE_SVR4_MAKE  (Bourne Shell)
% setenv USE_SVR4_MAKE  (C shell)
```

For more information on this version of make, see also the **sysV-make***(1)* man page.

The trend toward increased modularity of programs means that the project manager may have to work with a large assortment of individual files. A wide range of generation procedures may be needed to turn the assortment of individual files into the final executable product.

make provides a method for maintaining up-to-date versions of programs that consist of a number of files that can be generated in a variety of ways.

An individual programmer can easily forget:

- File-to-file dependencies
- Files that were modified and the impact that has on other files
- The exact sequence of operations needed to generate a new version of the program

In a description file, make keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that constitiute a program, the make command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

The basic operation of make is to:

- Find the target in the description file

- Ensure that all the files on which the target depends, the files needed to generate the target, exist and are up to date
- (Re)create the target file if any of the generators have been modified more recently than the target

The description file that holds the information on interfile dependencies and command sequences is conventionally called `makefile`, `Makefile`, `s.makefile`, or `s.Makefile`. If this naming convention is followed, the simple command `make` is usually sufficient to regenerate the target regardless of the number of files edited since the last `make`. In most cases, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, rather than entering all the commands to regenerate the target, entering the `make` command ensures that the regeneration is done in the prescribed way.

Basic Features

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

The `make` program operates using three sources of information:

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

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

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

The following specification describes the relationships and operations:

```
prog : x.o y.o z.o
       cc x.o y.o z.o -lm -o prog
x.o y.o : defs.h H
```

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

```
$ make
```

would perform the operations needed to regenerate prog after any changes had been made to any of the four source files x.c, y.c, z.c, or defs.h. In the previous example, the first line states that prog depends on three .o files. After these object files are current, the second line describes how to load them to create prog. The third line states that x.o and y.o depend on the file defs.h. From the file system, make discovers that there are three .c files corresponding to the needed .o files and that they use built-in rules on how to generate an object from a C source file (that is, issue a cc c command).

If make did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

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

If none of the source or object files have changed since the last time prog was made, and all of the files are current, the command make announces this fact and stops. If, however, the defs.h file has been edited, x.c and y.c (but not z.c) are recompiled; and then prog is created from the new x.o and y.o files, and the existing z.o file. If only the file y.c had changed, only it is recompiled; but it is still necessary to reload prog. If no target name is given on the make command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

```
$ make x.o
```

would regenerate x.o if x.c or defs.h had changed.

A method often useful to programmers is to include rules with mnemonic names and commands that do not actually produce a file with that same name. These entries can take advantage of make's ability to generate files and substitute macros (for information about macros, see "Description Files and Substitutions " on page 225.) Thus, an entry save might be included to copy a certain set of files, or an entry clean might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

A simple macro mechanism for substitution in dependency lines and command strings is used by make. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by the symbol = followed by what the macro stands for. A macro is invoked by preceding the name by the symbol $. Macro names longer than one character must be enclosed in parentheses. The following are valid macro invocations:

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

The last two are equivalent.

Four special macros `$*`, `$@`, `$?`, and `$<` change values during the execution of the command. (These four macros are described under “Description Files and Substitutions ” on page 225.) The following fragment shows assignment and use of some macros:

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

The command

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

loads the three objects with both the `lex` (–`ll`) and the `math` (–`lm`

) libraries, because macro definitions on the command line override definitions in the description file. (In operating system commands, arguments with embedded blanks must be quoted.)

As an example of the use of `make`, a description file that might be used to maintain the `make` command itself is given. The code for `make` is spread over a number of C language source files and has a `yacc` grammar. The description file contains the following:

```
# Description file for the make command

FILES = Makefile defs.h main.c doname.c misc.c \
        files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o dosys.o gram.o
LIBES =
LINT = lint -p
CFLAGS = -O
LP = lp

make: $(OBJECTS)
        $(CC) $(CFLAGS) -o make $(OBJECTS) $(LIBES)
        @size make

$(OBJECTS): defs.h

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

```
install:
        make
        @size make /usr/bin/make
        cp make /usr/bin/make && rm make

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

        # print files that are out-of-date
        # with respect to "print" file.

print: $(FILES)
        pr $? | $(LP)
        touch print
```

The make program prints each command before issuing it.

The following output results from entering the command make in a directory containing only the source and description files:

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

The last line results from the size make command. The printing of the command line itself was suppressed by the symbol @ in the description file.

Description Files and Substitutions

The following section explains the most commonly used elements of the description file.

Comments

The # symbol marks the beginning of a command, and all characters on the same line after it are ignored. Blank lines and lines beginning with # are ignored.

Continuation Lines

If a noncomment line is too long, the line can be continued by using the symbol \, which must be the last character on the line. If the last character of a line is \, then it, the new line, and all following blanks and tabs are replaced by a single blank.

Macro Definitions

A macro definition is an identifier followed by the symbol =. The identifier must not be preceded by a colon (:) or a tab. The name (string of letters and digits) to the left of the = (trailing blanks and tabs are stripped) is assigned the string of characters following the =+ (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

The last definition assigns `LIBES` the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in `make`'s own rules. (See “Internal Rules ” on page 239.)

General Form

The general form of an entry in a description file is:

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands]
[# ...] [ \t commands] [# ...]
 .  .  .
```

Items inside brackets can be omitted and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as * and ? are expanded when the commands are evaluated. Commands can appear either after a semicolon on a dependency line or on lines beginning with a tab (denoted above as `\t`) immediately following a dependency line. A command is any string of characters not including #, except when # is in quotes.

Dependency Information

A dependency line can have either a single or a double colon. A target name can appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the more common single colon case, a command sequence can be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule can be invoked. In the double colon case, a command sequence can be associated with more than one dependency line. If the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule can also be executed. The double-colon form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in the "Archive Libraries " on page 231 section.)

Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (–s option of the make command) or if the command line in the description file begins with an @ sign. make normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the –i flag has been specified on the make command line, if the fake target name .IGNORE appears in the description file, or if the command string in the description file begins with a hyphen (-). If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (cd and shell control commands, for instance) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The $@ macro is set to the full target name of the current target. The $@ macro is evaluated only for explicitly named dependencies. The $? macro is set to the string of names that were found to be younger than the target. The $? macro is evaluated when explicit rules from the makefile are evaluated. If the command was generated by an implicit rule, the $< macro is the name of the related file that caused the action; and the $* macro is the prefix shared by the current and the dependent filenames. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used. If there is no such name, make prints a message and stops.

In addition, a description file can also use the following related macros: $(@D), $(@F), $(*D), $(*F), $(<D), and $(<F).

Extensions of $*, $@, and $<

The internally generated macros $*, $@, and $< are useful generic terms for current targets and out-of-date relatives. To this list is added the following related macros: $(@D), $(@F), $(*D), $(*F), $(<D), and $(<F). The D refers to the directory part of the single character macro. The F refers to the filename part of the single character macro. These additions are useful when building hierarchical makefiles. They

allow access to directory names for purposes of using the cd command of the shell. Thus, a command can be:

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

Output Translations

The values of macros are replaced when evaluated. The general form, where brackets indicate that the enclosed sequence is optional, is as follows:

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

The parentheses are optional if there is no substitution specification and the macro name is a single character. If a substitution sequence is present, the value of the macro is considered to be a sequence of ''words'' separated by sequences of blanks, tabs, and new-line characters. Then, for each such word that ends with string1, string1 is replaced with string2 (or no characters if string2 is not present).

This particular substitution capability was chosen because make is sensitive to suffixes. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script that can handle all the C language programs (that is, files ending in .c). The following fragment optimizes the executions of make for maintaining an archive library:

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

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added to offer more general use of the wealth of information that make generates.

Recursive Makefiles

Another feature of make concerns the environment and recursive invocations. If the sequence $(MAKE) appears anywhere in a shell command line, the line is executed even if the –n flag is set. Since the –n flag is exported across invocations of make (through the MAKEFLAGS variable), the only thing that is executed is the make command itself. This feature is useful when a hierarchy of makefiles describes a set of software subsystems. For testing purposes, make –n can be executed and everything that would have been done is printed, including output from lower-level invocations of make.

Suffixes and Transformation Rules

make uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the –r flag is used on the make command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name .SUFFIXES. make searches for a file with any of the suffixes on the list. If it finds one, make transforms it into a file with another suffix. Transformation rule names are the concatenation of the before and after suffixes. The name of the rule to transform a .r file to a .o file is thus .r.o. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule .r.o is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro $< is the full name of the dependent that caused the action.

The order of the suffix list is significant because the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for .SUFFIXES in the description file. The dependencies are added to the usual list.

A .SUFFIXES line without any dependencies deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

Implicit Rules

make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list (in order) is as follows:

.o	Object file
.c	C source file
.c~	SCCS C source file
.y	yacc C source grammar
.y~	SCCS yacc C source grammar
.l	lex C source grammar
.l~	SCCS lex C source grammar
.s	Assembler source file
.s~	SCCS assembler source file

.sh	Shell file
.sh~	SCCS shel file
.h	Header file
.h~	SCCS header file
.f	FORTRAN source file
.f~	SCCS FORTRAN source file
.C	C++ source file
.C~	SCCS C++ source file
.Y	yacc C++ source grammar
.Y~	SCCS yacc C++ source grammar
.L	lex C++ source grammar
.L~	SCCS lex C++ source grammar

Figure A–1 summarizes the default transformation paths. If two paths connect a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

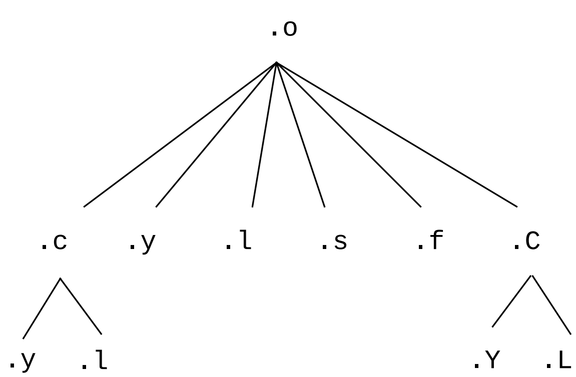


Figure A–1 Summary of Default Transformation Path

If the file `x.o` is needed and an `x.c` is found in the description or directory, the `x.o` file would be compiled. If there is also an `x.l`, that source file is run through `lex` before compiling the result. However, if there is no `x.c` but there is an `x.l`, `make` discards the intermediate C language file and uses the direct link, as shown in Figure A–1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, C++C, F77, YACC, and LEX. The following command causes the newcc command to be used instead of the usual C language compiler.

```
$ make CC=newcc
```

The macros CFLAGS, YFLAGS, LFLAGS, ASFLAGS, FFLAGS, and C++FLAGS can be set to cause these commands to be issued with optional flags. Thus

```
$ make CFLAGS=-g
```

causes the cc command to include debugging information.

Archive Libraries

The make program has an interface to archive libraries. A user can name a member of a library in the following manner:

```
projlib(object.o)
```

or

```
projlib((entry_pt))
```

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

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

```
projlib:: projlib(pfile1.o)
         $(CC) -c $(CFLAGS) pfile1.c
         $(AR) $(ARFLAGS) projlib pfile1.o
         rm pfile1.o
projlib:: projlib(pfile2.o)
         $(CC) -c $(CFLAGS) pfile2.c
         $(AR) $(ARFLAGS) projlib pfile2.o
         rm pfile2.o
```

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

The make command also gives the user access to a rule for building libraries. The handle for the rule is the .a suffix. Thus, a .c.a rule is the rule for compiling a C language source file, adding it to the library, and removing the .o file. Similarly, the .y.a, the .s.a, and the .l.a rules rebuild yacc, assembler, and lex files. The archive rules defined internally are .c.a, .c~.a, .f.a, .f~.a, and .s~.a. (The tilde (~) syntax will be described shortly.) The user can define other needed rules in the description file.

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

```
projlib: projlib(pfile1.o) projlib(pfile2.o)
        @echo projlib up-to-date.
```

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

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

Thus, the `$@` macro is the `.a` target (`projlib`); the `$<` and `$*` macros are set to the out-of-date C language file, and the filename minus the suffix, (`pfile1.c` and `pfile1`). The `$<` macro (in the preceding rule) could have been changed to `$*.c`.

It is useful to go into some detail about exactly what `make` does when it executes the construction

```
projlib: projlib(pfile1.o)
        @echo projlib up-to-date
```

Assume the object in the library is out of date with respect to `pfile1.c`. Also, there is no `pfile1.o` file.

1. `make projlib`.
2. Before using `make projlib`, check each dependent of `projlib`.
3. `projlib(pfile1.o)` is a dependent of `projlib` and needs to be generated.
4. Before generating `projlib(pfile1.o)`, check each dependent of `projlib(pfile1.o)`. (There are none.)
5. Use internal rules to try to create `projlib(pfile1.o)`. (There is no explicit rule.) Note that `projlib(pfile1.o)` has a parenthesis in the name to identify the target suffix as `.a`. This is the key. There is no explicit `.a` at the end of the `projlib` library name. The parenthesis implies the `.a` suffix. In this sense, the `.a` is hard-wired into `make`.
6. Breakup the name `projlib(pfile1.o)` into `projlib` and `pfile1.o`. Define two macros, `$@` (`projlib`) and `$*` (`pfile1`).
7. Look for a rule `.X.a` and a file `$*.X`. The first `.X` (in the `.SUFFIXES` list) that fulfills these conditions is `.c so` the rule is `.c.a`, and the file is `pfile1.c`. Set `$<` to be `pfile1.c` and execute the rule. In fact, `make` must then compile `pfile1.c`.
8. The library has been updated. Execute the command associated with the `projlib:` dependency, namely

```
@echo projlib up-to-date
```

To enable `pfile1.o` to have dependencies, the following syntax is required:

```
projlib(pfile1.o): $(INCDIR)/stdio.h pfile1.c
```

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 parenthesis.

Source Code Control System (SCCS) Filenames

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

To allow `make` easy access to the prefix `s.`, the symbol ~ is used as an identifier of SCCS files. Hence, `.c~.o` refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the internal rule is
`$(GET) $(GFLAGS) $< $(CC) $(CFLAGS) -c $*.c rm -f $*.c` .

Thus, ~ 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 symbol.

The following SCCS suffixes are internally defined:

.c~	.sh~	.C~
.y~	.h~	.Y~
.l~	.f~	.L~
.s~		

The following rules involving SCCS transformations are internally defined:

.c~:	.s~.s:	.c~:
.c~.c:	.s~.a:	.C~.C:
.c~.a:	.s~.o:	.C~.a:
.c~.o:	.sh~:	.C~.o:
.y~.c:	.sh~.sh:	.Y~.C:
.y~.o:	.h~.h:	.Y~.o:
.y~.y:	.f~:	.Y~.Y:
.l~.c	.f~.f:	.L~.C:
.l~.o:	.f~.a:	.L~.o:

```
.l~.l:              .f~.o:              .L~.L:
.s,:
```

Obviously, the user can define other rules and suffixes that can prove useful. The ~ provides a handle on the SCCS filename format so that this is possible.

The Null Suffix

Many programs consist of a single source file. make handles this case by the null suffix rule. To maintain the operating system program cat, a rule in the makefile of the following form is needed:

```
$(CC) -o $@ $(CFLAGS) $(LDFLAGS) $<
```

In fact, this .c: rule is internally defined so no makefile is necessary at all. The user only needs to enter $ **make cat dd echo date** (these are all operating system single-file programs) and all four C language source files are passed through the above shell command line associated with the .c: rule. The internally defined single suffix rules are:

```
.c:                 .sh:                .f,:
.c,:                .sh,:               .C:
.s:                 .f:                 .C,:
.s,:
```

Others can be added in the makefile by the user.

Included Files

The make program has a capability similar to the #include directive of the C preprocessor. 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 rest of the line is assumed to be a filename, which the current invocation of make reads. Macros can be used in filenames. The file descriptors are stacked for reading include files so that no more than 16 levels of nested includes are supported.

SCCS Makefiles

Makefiles under SCCS control are accessible to make. That is, if make is typed and only a file named s.makefile or s.Makefile exists, make performs a get on the file, then reads and removes the file.

Dynamic-Dependency Parameters

A dynamic-dependency parameter has meaning only on the dependency line in a makefile. The $$@ refers to the current ‘‘thing’’ to the left of the : symbol (which is $@). Also the form $$(@F) exists, which allows access to the file part of $@. Thus, in the following example:

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

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

```
CMDS = cat dd echo date cmp comm chown

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

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

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

```
INCDIR = /usr/include

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

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

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

Command Usage

The make Command

The make command takes macro definitions, options, description filenames, and target filenames as arguments in the form:

```
$ make [ options ] [ macro definitions and targets ]
```

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

First, all macro definition arguments (arguments with embedded = symbols) are analyzed and the assignments made. Command line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The permissible options are as follows:

–i	Ignore error codes returned by invoked commands. This mode is entered if the fake target name .IGNORE appears in the description file.
–s	Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name .SILENT appears in the description file.
–r	Do not use the built-in rules.
–n	No execute mode. Print commands, but do not execute them. Even lines beginning with an @ sign are printed.
–t	Touch the target files (causing them to come up to date) rather than issue the usual commands.
–q	Question. The make command returns a zero or nonzero status code depending on whether the target file is or is not up to date.
–p	Print the complete set of macro definitions and target descriptions.
–k	Abandon work on the current entry if something goes wrong, but continue on other branches that do not depend on the current entry.

–e — Environment variables override assignments within makefiles.

–f — Description filename. The next argument is assumed to be the name of a description file. A file name of - denotes the standard input. If there are no –f arguments, the file named makefile, Makefile, s.makefile, or s.Makefile in the current directory is read. The contents of the description files override the built-in rules if they are present. The following two fake target names are evaluated in the same manner as flags:

.DEFAULT — If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists.

.PRECIOUS — Dependents on this target are not removed when Quit or Interrupt is pressed.

Finally, the remaining arguments are assumed to be the names of targets to be made and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with the symbol . is made.

Environment Variables

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

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

1. Read the MAKEFLAGS environment variable. If it is not present or null, the internal make variable *MAKEFLAGS* is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the –f, –p, and –r flags.)
2. Read the internal list of macro definitions.
3. Read the environment. The environment variables are treated as macro definitions and marked as exported (in the shell sense).

4. Read the makefile(s). The assignments in the makefile(s) override the environment. This order is chosen so that when a makefile is read and executed, you know what to expect. That is, you get what is seen unless the –e flag is used. The –e is the input flag argument, which tells make to have the environment override the makefile assignments. Thus, if make –e is entered, the variables in the environment override the definitions in the makefile. Also MAKEFLAGS overrides the environment if assigned. This is useful for further invocations of make from the current makefile.

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

1. Internal definitions
2. Environment
3. makefile(s)
4. Command line

   The –e flag has the effect of rearranging the order to:

1. Internal definitions
2. makefile(s)
3. Environment
4. Command line

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

Suggestions and Warnings

The most common difficulties arise from make's specific meaning of dependency. If file x.c has the following line:

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

then the object file x.o depends on defs.h; the source file x.c does not. If defs.h is changed, nothing is done to the file x.c while file x.o must be recreated.

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

```
$ make -n
```

orders make to print out the commands that make would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (adding a comment to an include file, for example), the –t (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, make updates the modification times on the affected file. Thus, the command

```
$ make -ts
```

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

Internal Rules

The standard set of internal rules used by make are reproduced below.

Suffixes recognized by make are:

.o	.c	.c~	.y	.y~	.l	.l~	.s	.s~	.sh	.sh~
.h	.h~	.f	.f~	.C	.C~	.Y	.Y~	.L	.L~	

The following are predefined macros:

```
AR=ar ARFLAGS=-rv AS=as ASFLAGS= BUILD=build CC=cc CFLAGS= -0 C++C=CC
C++FLAGS= -0 F77=f77 FFLAGS= -0 GET=get GFLAGS= LEX=lex LFLAGS= LD=ld
LDFLAGS= MAKE=make MAKEFLAGS= YACC=yacc YFLAGS= $=$
```

Special Rules

This section covers special make rules with either single or double suffixes.

```
markfile.o : markfile
     A=@; echo "static char _sccsid[]=\042‘grep $$A’(#)’ \
     markfile‘\042;" > markfile.c
     $(CC) -c markfile.c
     rm -f markfile.c
```

Single-Suffix Rules

The following are single-suffix rules:

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


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

```
.s:
  $(AS) $(AFLAGS) -o $@ $<


.s~:
  $(GET) $(GFLAGS) $<
  $(AS) $(AFLAGS) -o $@ $*.s
  rm -f $*.s


.sh:
  cp $< $@; chmod 0777 $@


.sh~:
 $(GET) $(GFLAGS) $<
  cp $*.sh $*; chmod 0777 $@
  rm -f $*.sh


.f:
  $(F77) $(FFLAGS) $(LDFLAGS) -o $@ $<


.f~:
  $(GET) $(GFLAGS) $<
  $(F77) $(FFLAGS) -o $@ $(LDFLAGS) $*.f
  rm -f $*.f


.C~:
  $(GET) $(GFLAGS) $<
  $(C++C) $(C++FLAGS) -o $@ $(LDFLAGS) $*.C
  rm -f $*.C


.C:
  $(C++C) $(C++FLAGS) -o $@ $(LDFLAGS) $<
```

Double-Suffix Rules

The following are double-suffix rules:

.c~.c	.y~.y	.l~.l	.s~.s	.sh~.sh
.h~.h	.f~.f	.C~.C	.Y~.Y	.L~.L

```
$(GET) $(GFLAGS) $<


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

```
c.a~:
  $(GET) $(GFLAGS) $<
  $(CC) -c $(CFLAGS) $*.c
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.[co]


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


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


.y.c:


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


.y~.c:


 $(GET) $(GFLAGS) $<
  $(YACC) $(YFLAGS) $*.y
  mv y.tab.c $*.c
  rm -f $*.y


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


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


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


.l~.c:
  $(GET) $(GFLAGS) $<
  $(LEX) $(LFLAGS) $*.l
  mv lex.yy.c $@
  rm -f $*.l
```

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

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

.s.a:
  $(AS) $(ASFLAGS) -o $*.o $*.s
  $(AR) $(ARFLAGS) $@ $*.o

.s~.a:
  $(GET) $(GFLAGS) $<
  $(AS) $(ASFLAGS) -o $*.o $*.s
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.[so]

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

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

.f.a:
  $(F77) $(FFLAGS) -c $*.f
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.o

.f~.a:
  $(GET) $(GFLAGS) $<
  $(F77) $(FFLAGS) -c $*.f
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.[fo]

.f.o:
  $(F77) $(FFLAGS) -c $*.f

.f~.o:
  $(GET) $(GFLAGS) $<
  $(F77) $(FFLAGS) -c $*.f
  rm -f $*.f
```

```
.C.a:
  $(C++C) $(C++FLAGS) -c $<
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.o


.C~.a:
  $(GET) $(GFLAGS) $<
  $(C++C) $(C++FLAGS) -c $*.C
  $(AR) $(ARFLAGS) $@ $*.o
  rm -f $*.[Co]


.C.o:
  $(C++C) $(C++FLAGS) -c $<


.C~.o:
  $(GET) $(GFLAGS) $<
  $(C++C) $(C++FLAGS) -c $*.C
  rm -f $*.C


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


.Y~.C:
  $(GET) $(GFLAGS) $<
  $(YACC) $(YFLAGS) $*.Y
  mv y.tab.c $*.C
  rm -f $*.Y


.Y.o
  $(YACC) $(YFLAGS) $<
  $(C++C) $(C++FLAGS) -c y.tab.c
  rm -f y.tab.c
  mv y.tab.o $@


.Y~.o:
  $(GET) $(GFLAGS) $<
  $(YACC) $(YFLAGS) $*.Y
  $(C++C) $(C++FLAGS) -c y.tab.c
  rm -f y.tab.c $*.Y
  mv y.tab.o $*.o


.L.C:
  $(LEX) $(LFLAGS) $<
  mv lex.yy.c $@


.L~.C:
  $(GET) $(GFLAGS) $<
  $(LEX) $(LFLAGS) $*.L
  mv lex.yy.c $@
  rm -f $*.L
```

```
.L.o:
	$(LEX) $(LFLAGS) $<
	$(C++C) $(C++FLAGS) -c lex.yy.c
	rm -f lex.yy.c
	mv lex.yy.o $@


.L~.o:
	$(GET) $(GFLAGS) $<
	$(LEX) $(LFLAGS) $*.L
	$(C++C) $(C++FLAGS) -c lex.yy.c
	rm -f lex.yy.c $*.L
	mv lex.yy.o $@
```

Index

G

H

I

K

L

M

T

U

V

W

X

Y

Z