CHAPTER 2

Rules

In the last chapter, we wrote some rules to compile and link our word-counting program. Each of those rules defines a target, that is, a file to be updated. Each target file depends on a set of prerequisites, which are also files. When asked to update a target, make will execute the command script of the rule if any of the prerequisite files has been modified more recently than the target. Since the target of one rule can be referenced as a prerequisite in another rule, the set of targets and prerequisites form a chain or graph of *dependencies* (short for "dependency graph"). Building and processing this dependency graph to update the requested target is what make is all about.

Since rules are so important in make, there are a number of different kinds of rules. *Explicit rules*, like the ones in the previous chapter, indicate a specific target to be updated if it is out of date with respect to any of its prerequisites. This is the most common type of rule you will be writing. *Pattern rules* use wildcards instead of explicit filenames. This allows make to apply the rule any time a target file matching the pattern needs to updated. *Implicit rules* are either pattern rules or suffix rules found in the rules database built-in to make. Having a built-in database of rules makes writing *makefile*s easier since for many common tasks make already knows the file types, suffixes, and programs for updating targets. *Static pattern rules* are like regular pattern rules except they apply only to a specific list of target files.

GNU make can be used as a "drop in" replacement for many other versions of make and includes several features specifically for compatibility. *Suffix rules* were make's original means for writing general rules. GNU make includes support for suffix rules, but they are considered obsolete having been replaced by pattern rules that are clearer and more general.

Explicit Rules

Most rules you will write are explicit rules that specify particular files as targets and prerequisites. A rule can have more than one target. This means that each target has

the same set of prerequisites as the others. If the targets are out of date, the same set of actions will be performed to update each one. For instance:

```
vpath.o variable.o: make.h config.h getopt.h gettext.h dep.h
```

This indicates that both *vpath.o* and *variable.o* depend on the same set of C header files. This line has the same effect as:

```
vpath.o: make.h config.h getopt.h gettext.h dep.h
variable.o: make.h config.h getopt.h gettext.h dep.h
```

The two targets are handled independently. If either object file is out of date with respect to any of its prerequisites (that is, any header file has a newer modification time than the object file), make will update the object file by executing the commands associated with the rule.

A rule does not have to be defined "all at once." Each time make sees a target file it adds the target and prerequisites to the dependency graph. If a target has already been seen and exists in the graph, any additional prerequisites are appended to the target file entry in make's dependency graph. In the simple case, this is useful for breaking long lines naturally to improve the readability of the *makefile*:

```
vpath.o: vpath.c make.h config.h getopt.h gettext.h dep.h
vpath.o: filedef.h hash.h job.h commands.h variable.h vpath.h
```

In more complex cases, the prerequisite list can be composed of files that are managed very differently:

```
# Make sure lexer.c is created before vpath.c is compiled.
vpath.o: lexer.c
...
# Compile vpath.c with special flags.
vpath.o: vpath.c
        $(COMPILE.c) $(RULE_FLAGS) $(OUTPUT_OPTION) $<
...
# Include dependencies generated by a program.
include auto-generated-dependencies.d
```

The first rule says that the *vpath.o* target must be updated whenever *lexer.c* is updated (perhaps because generating *lexer.c* has other side effects). The rule also works to ensure that a prerequisite is always updated before the target is updated. (Notice the bidirectional nature of rules. In the "forward" direction the rule says that if the *lexer.c* has been updated, perform the action to update *vpath.o*. In the "backward" direction, the rule says that if we need to make or use *vpath.o* ensure that *lexer.c* is up to date first.) This rule might be placed near the rules for managing *lexer.c* so developers are reminded of this subtle relationship. Later, the compilation rule for *vpath.o* is placed among other compilation rules. The command for this rule uses three make variables. You'll be seeing a lot of these, but for now you just need to know that a variable is either a dollar sign followed by a single character or a dollar sign followed by a word in parentheses. (I will explain more later in this chapter and

a lot more in Chapter 3.) Finally, the *.o/.h* dependencies are included in the *makefile* from a separate file managed by an external program.

Wildcards

A *makefile* often contains long lists of files. To simplify this process make supports wildcards (also known as *globbing*). make's wildcards are identical to the Bourne shell's: ~, *, ?, [...], and [^...]. For instance, *.* expands to all the files containing a period. A question mark represents any single character, and [...] represents a *character class*. To select the "opposite" (negated) character class use [^...].

In addition, the tilde (~) character can be used to represent the current user's home directory. A tilde followed by a user name represents that user's home directory.

Wildcards are automatically expanded by make whenever a wildcard appears in a target, prerequisite, or command script context. In other contexts, wildcards can be expanded explicitly by calling a function. Wildcards can be very useful for creating more adaptable *makefile*s. For instance, instead of listing all the files in a program explicitly, you can use wildcards:[*]

```
prog: *.c
        $(CC) -o $@ $^
```

It is important to be careful with wildcards, however. It is easy to misuse them as the following example shows:

```
*.o: constants.h
```

The intent is clear: all object files depend on the header file *constants.h*, but consider how this expands on a clean directory without any object files:

```
: constants.h
```

This is a legal make expression and will not produce an error by itself, but it will also not provide the dependency the user wants. The proper way to implement this rule is to perform a wildcard on the source files (since they are always present) and transform that into a list of object files. We will cover this technique when we discuss make functions in Chapter 4.

Finally, it is worth noting that wildcard expansion is performed by make when the pattern appears as a target or prerequisite. However, when the pattern appears in a command, the expansion is performed by the subshell. This can occasionally be important because make will expand the wildcards immediately upon reading the *makefile*, but the shell will expand the wildcards in commands much later when the command is executed. When a lot of complex file manipulation is being done, the two wildcard expansions can be quite different.

[*] In more controlled environments using wildcards to select the files in a program is considered bad practice because a rogue source file might be accidentally linked into a program.

Phony Targets

Until now all targets and prerequisites have been files to be created or updated. This is typically the case, but it is often useful for a target to be just a label representing a command script. For instance, earlier we noted that a standard first target in many *makefile*s is called `all`. Targets that do not represent files are known as *phony targets*. Another standard phony target is `clean`:

```
clean:
        rm -f *.o lexer.c
```

Normally, phony targets will always be executed because the commands associated with the rule do not create the target name.

It is important to note that `make` cannot distinguish between a file target and phony target. If by chance the name of a phony target exists as a file, `make` will associate the file with the phony target name in its dependency graph. If, for example, the file *clean* happened to be created running `make clean` would yield the confusing message:

```
$ make clean
make: `clean' is up to date.
```

Since most phony targets do not have prerequisites, the `clean` target would always be considered up to date and would never execute.

To avoid this problem, GNU `make` includes a special target, `.PHONY`, to tell `make` that a target is not a real file. Any target can be declared phony by including it as a prerequisite of `.PHONY`:

```
.PHONY: clean
clean:
        rm -f *.o lexer.c
```

Now `make` will always execute the commands associated with `clean` even if a file named *clean* exists. In addition to marking a target as always out of date, specifying that a target is phony tells `make` that this file does not follow the normal rules for making a target file from a source file. Therefore, `make` can optimize its normal rule search to improve performance.

It rarely makes sense to use a phony target as a prerequisite of a real file since the phony is always out of date and will always cause the target file to be remade. However, it is often useful to give phony targets prerequisites. For instance, the `all` target is usually given the list of programs to be built:

```
.PHONY: all
all: bash bashbug
```

Here the `all` target creates the `bash` shell program and the `bashbug` error reporting tool.

Phony targets can also be thought of as shell scripts embedded in a *makefile*. Making a phony target a prerequisite of another target will invoke the phony target script

before making the actual target. Suppose we are tight on disk space and before executing a disk-intensive task we want to display available disk space. We could write:

```
.PHONY: make-documentation
make-documentation:
        df -k . | awk 'NR == 2 { printf( "%d available\n", $$4 ) }'
        javadoc ...
```

The problem here is that we may end up specifying the `df` and `awk` commands many times under different targets, which is a maintenance problem since we'll have to change every instance if we encounter a `df` on another system with a different format. Instead, we can place the `df` line in its own phony target:

```
.PHONY: make-documentation
make-documentation: df
        javadoc ...

.PHONY: df
df:
        df -k . | awk 'NR == 2 { printf( "%d available\n", $$4 ) }'
```

We can cause `make` to invoke our `df` target before generating the documentation by making `df` a prerequisite of `make-documentation`. This works well because `make-documentation` is also a phony target. Now I can easily reuse `df` in other targets.

There are a number of other good uses for phony targets.

The output of `make` can be confusing to read and debug. There are several reasons for this: *makefile*s are written top-down but the commands are executed by `make` bottom-up; also, there is no indication which rule is currently being evaluated. The output of `make` can be made much easier to read if major targets are commented in the `make` output. Phony targets are a useful way to accomplish this. Here is an example taken from the `bash` *makefile*:

```
$(Program): build_msg $(OBJECTS) $(BUILTINS_DEP) $(LIBDEP)
        $(RM) $@
        $(CC) $(LDFLAGS) -o $(Program) $(OBJECTS) $(LIBS)
        ls -l $(Program)
        size $(Program)

.PHONY: build_msg
build_msg:
        @printf "#\n# Building $(Program)\n#\n"
```

Because the `printf` is in a phony target, the message is printed immediately before any prerequisites are updated. If the build message were instead placed as the first command of the `$(Program)` command script, then it would be executed after all compilation and dependency generation. It is important to note that because phony targets are always out of date, the phony `build_msg` target causes `$(Program)` to be regenerated even when it is not out of date. In this case, it seems a reasonable choice since most of the computation is performed while compiling the object files so only the final link will always be performed.

Phony targets can also be used to improve the "user interface" of a *makefile*. Often targets are complex strings containing directory path elements, additional filename components (such as version numbers) and standard suffixes. This can make specifying a target filename on the command line a challenge. The problem can be avoided by adding a simple phony target whose prerequisite is the actual target file.

By convention there are a set of more or less standard phony targets that many *makefile*s include. Table 2-1 lists these standard targets.

Table 2-1. Standard phony targets

Target	Function
`all`	Perform all tasks to build the application
`install`	Create an installation of the application from the compiled binaries
`clean`	Delete the binary files generated from sources
`distclean`	Delete all the generated files that were not in the original source distribution
`TAGS`	Create a tags table for use by editors
`info`	Create GNU info files from their Texinfo sources
`check`	Run any tests associated with this application

The target `TAGS` is not really a phony since the output of the `ctags` and `etags` programs is a file named *TAGS*. It is included here because it is the only standard non-phony target we know of.

Empty Targets

Empty targets are similar to phony targets in that the target itself is used as a device to leverage the capabilities of `make`. Phony targets are always out of date, so they always execute and they always cause their *dependent* (the target associated with the prerequisite) to be remade. But suppose we have some command, with no output file, that needs to be performed only occasionally and we don't want our dependents updated? For this, we can make a rule whose target is an empty file (sometimes referred to as a cookie):

```
prog: size prog.o
        $(CC) $(LDFLAGS) -o $@ $^

size: prog.o
        size $^
        touch size
```

Notice that the `size` rule uses `touch` to create an empty file named *size* after it completes. This empty file is used for its timestamp so that `make` will execute the `size` rule only when *prog.o* has been updated. Furthermore, the `size` prerequisite to *prog* will not force an update of *prog* unless its object file is also newer.

Empty files are particularly useful when combined with the automatic variable `$?`. We discuss automatic variables in the section "Automatic Variables," but a preview of this variable won't hurt. Within the command script part of a rule, `make` defines the variable `$?` to be the set of prerequisites that are newer than the target. Here is a rule to print all the files that have changed since the last time `make print` was executed:

```
print: *.[hc]
        lpr $?
        touch $@
```

Generally, empty files can be used to mark the last time a particular event has taken place.

Variables

Let's look at some of the variables we have been using in our examples. The simplest ones have the syntax:

```
$(variable-name)
```

This indicates that we want to *expand* the variable whose name is *`variable-name`*. Variables can contain almost any text, and variable names can contain most characters including punctuation. The variable containing the C compile command is `COMPILE.c`, for example. In general, a variable name must be surrounded by `$( )` to be recognized by `make`. As a special case, a single character variable name does not require the parentheses.

A *makefile* will typically define many variables, but there are also many special variables defined automatically by `make`. Some can be set by the user to control `make`'s behavior while others are set by `make` to communicate with the user's *makefile*.

Automatic Variables

Automatic variables are set by `make` after a rule is matched. They provide access to elements from the target and prerequisite lists so you don't have to explicitly specify any filenames. They are very useful for avoiding code duplication, but are critical when defining more general pattern rules (discussed later).

There are six "core" automatic variables:

`$@` The filename representing the target.

`$%` The filename element of an archive member specification.

`$<` The filename of the first prerequisite.

`$?` The names of all prerequisites that are newer than the target, separated by spaces.

`$^` The filenames of all the prerequisites, separated by spaces. This list has duplicate filenames removed since for most uses, such as compiling, copying, etc., duplicates are not wanted.

`$+` Similar to `$^`, this is the names of all the prerequisites separated by spaces, except that `$+` includes duplicates. This variable was created for specific situations such as arguments to linkers where duplicate values have meaning.

`$*` The stem of the target filename. A stem is typically a filename without its suffix. (We'll discuss how stems are computed later in the section "Pattern Rules.") Its use outside of pattern rules is discouraged.

In addition, each of the above variables has two variants for compatibility with other `makes`. One variant returns only the directory portion of the value. This is indicated by appending a "D" to the symbol, `$(@D)`, `$(<D)`, etc. The other variant returns only the file portion of the value. This is indicated by appending an F to the symbol, `$(@F)`, `$(<F)`, etc. Note that these variant names are more than one character long and so must be enclosed in parentheses. GNU `make` provides a more readable alternative with the `dir` and `notdir` functions. We will discuss functions in Chapter 4.

Automatic variables are set by `make` after a rule has been matched with its target and prerequisites so the variables are only available in the command script of a rule.

Here is our *makefile* with explicit filenames replaced by the appropriate automatic variable.

```
count_words: count_words.o counter.o lexer.o -lfl
        gcc $^ -o $@

count_words.o: count_words.c
        gcc -c $<

counter.o: counter.c
        gcc -c $<

lexer.o: lexer.c
        gcc -c $<

lexer.c: lexer.l
        flex -t $< > $@
```

Finding Files with VPATH and vpath

Our examples so far have been simple enough that the *makefile* and sources all lived in a single directory. Real world programs are more complex (when's the last time you worked on a single directory project?). Let's refactor our example and create a more realistic file layout. We can modify our word counting program by refactoring `main` into a function called `counter`.

```
#include <lexer.h>
#include <counter.h>
```

```
void counter( int counts[4] )
{
    while ( yylex() )
        ;

    counts[0] = fee_count;
    counts[1] = fie_count;
    counts[2] = foe_count;
    counts[3] = fum_count;
}
```

A reusable library function should have a declaration in a header file, so let's create *counter.h* containing our declaration:

```
#ifdef COUNTER_H_
#define COUNTER_H_

extern void
counter( int counts[4] );

#endif
```

We can also place the declarations for our *lexer.l* symbols in *lexer.h*:

```
#ifndef LEXER_H_
#define LEXER_H_

extern int fee_count, fie_count, foe_count, fum_count;
extern int yylex( void );

#endif
```

In a traditional source tree layout the header files are placed in an *include* directory and the source is placed in a *src* directory. We'll do this and put our *makefile* in the parent directory. Our example program now has the layout shown in Figure 2-1.

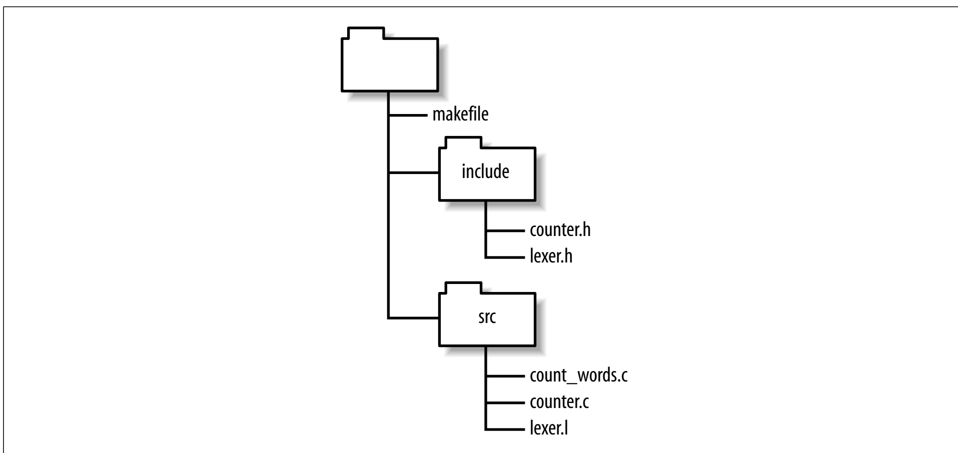

Figure 2-1. Example source tree layout

Since our source files now include header files, these new dependencies should be recorded in our *makefile* so that when a header file is modified, the corresponding object file will be updated.

```
count_words: count_words.o counter.o lexer.o -lfl
        gcc $^ -o $@

count_words.o: count_words.c include/counter.h
        gcc -c $<

counter.o: counter.c include/counter.h include/lexer.h
        gcc -c $<

lexer.o: lexer.c include/lexer.h
        gcc -c $<

lexer.c: lexer.l
        flex -t $< > $@
```

Now when we run our *makefile*, we get:

```
$ make
make: *** No rule to make target `count_words.c', needed by `count_words.o'.  Stop.
```

Oops, what happened? The *makefile* is trying to update *count_words.c*, but that's a source file! Let's "play `make`." Our first prerequisite is *count_words.o*. We see the file is missing and look for a rule to create it. The explicit rule for creating *count_words.o* references *count_words.c*. But why can't `make` find the source file? Because the source file is in the *src* directory not the current directory. Unless you direct it otherwise, `make` will look in the current directory for its targets and prerequisites. How do we get `make` to look in the *src* directory for source files? Or, more generally, how do we tell `make` where our source code is?

You can tell `make` to look in different directories for its source files using the `VPATH` and `vpath` features. To fix our immediate problem, we can add a `VPATH` assignment to the *makefile*:

```
VPATH = src
```

This indicates that `make` should look in the directory *src* if the files it wants are not in the current directory. Now when we run our *makefile*, we get:

```
$ make
gcc -c src/count_words.c -o count_words.o
src/count_words.c:2:21: counter.h: No such file or directory
make: *** [count_words.o] Error 1
```

Notice that `make` now successfully tries to compile the first file, filling in correctly the relative path to the source. This is another reason to use automatic variables: `make` cannot use the appropriate path to the source if you hardcode the filename. Unfortunately, the compilation dies because `gcc` can't find the include file. We can fix this

latest problem by "customizing" the implicit compilation rule with the appropriate -I option:

```
CPPFLAGS = -I include
```

Now the build succeeds:

```
$ make
gcc -I include -c src/count_words.c -o count_words.o
gcc -I include -c src/counter.c -o counter.o
flex -t src/lexer.l > lexer.c
gcc -I include -c lexer.c -o lexer.o
gcc count_words.o counter.o lexer.o /lib/libfl.a -o count_words
```

The VPATH variable consists of a list of directories to search when make needs a file. The list will be searched for targets as well as prerequisites, but not for files mentioned in command scripts. The list of directories can be separated by spaces or colons on Unix and separated by spaces or semicolons on Windows. I prefer to use spaces since that works on all systems and we can avoid the whole colon/semicolon imbroglio. Also, the directories are easier to read when separated by spaces.

The VPATH variable is good because it solved our searching problem above, but it is a rather large hammer. make will search each directory for *any* file it needs. If a file of the same name exists in multiple places in the VPATH list, make grabs the first one. Sometimes this can be a problem.

The vpath directive is a more precise way to achieve our goals. The syntax of this directive is:

```
vpath pattern directory-list
```

So our previous VPATH use can be rewritten as:

```
vpath %.c src
vpath %.h include
```

Now we've told make that it should search for *.c* files in the *src* directory and we've also added a line to search for *.h* files in the *include* directory (so we can remove the *include/* from our header file prerequisites). In more complex applications, this control can save a lot of headache and debugging time.

Here we used vpath to handle the problem of finding source that is distributed among several directories. There is a related but different problem of how to build an application so that the object files are written into a "binary tree" while the source files live in a separate "source tree." Proper use of vpath can also help to solve this new problem, but the task quickly becomes complex and vpath alone is not sufficient. We'll discuss this problem in detail in later sections.

Pattern Rules

The *makefile* examples we've been looking at are a bit verbose. For a small program of a dozen files or less we may not care, but for programs with hundreds or thousands of files, specifying each target, prerequisite, and command script becomes unworkable. Furthermore, the commands themselves represent duplicate code in our *makefile*. If the commands contain a bug or ever change, we would have to update all these rules. This can be a major maintenance problem and source of bugs.

Many programs that read one file type and output another conform to standard conventions. For instance, all C compilers assume that files that have a *.c* suffix contain C source code and that the object filename can be derived by replacing the *.c* suffix with *.o* (or *.obj* for some Windows compilers). In the previous chapter, we noticed that `flex` input files use the *.l* suffix and that `flex` generates *.c* files.

These conventions allow `make` to simplify rule creation by recognizing common filename patterns and providing built-in rules for processing them. For example, by using these built-in rules our 17-line *makefile* can be reduced to:

```
VPATH    = src include
CPPFLAGS = -I include

count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

The built-in rules are all instances of pattern rules. A *pattern rule* looks like the normal rules you have already seen except the *stem* of the file (the portion before the suffix) is represented by a `%` character. This *makefile* works because of three built-in rules. The first specifies how to compile a *.o* file from a *.c* file:

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

The second specifies how to make a *.c* file from a *.l* file:

```
%.c: %.l
        @$(RM) $@
        $(LEX.l) $< > $@
```

Finally, there is a special rule to generate a file with no suffix (always an executable) from a *.c* file:

```
%: %.c
        $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

We'll go into the details of this syntax in a bit, but first let's go over `make`'s output carefully and see how `make` applies these built-in rules.

When we run `make` on our two-line *makefile*, the output is:

```
$ make
gcc  -I include  -c -o count_words.o src/count_words.c
```

```
gcc  -I include  -c -o counter.o src/counter.c
flex  -t src/lexer.l > lexer.c
gcc  -I include  -c -o lexer.o lexer.c
gcc   count_words.o counter.o lexer.o /lib/libfl.a   -o count_words
rm lexer.c
```

First, make reads the *makefile* and sets the default goal to *count_words* since there are no command-line targets specified. Looking at the default goal, make identifies four prerequisites: *count_words.o* (this prerequisite is missing from the *makefile*, but is provided by the implicit rule), *counter.o*, *lexer.o*, and -lfl. It then tries to update each prerequisite in turn.

When make examines the first prerequisite, *count_words.o*, make finds no explicit rule for it but discovers the implicit rule. Looking in the local directory, make cannot find the source, so it begins searching the VPATH and finds a matching source file in *src*. Since *src/count_words.c* has no prerequisites, make is free to update *count_words.o* so it runs the commands for the implicit rule. *counter.o* is similar. When make considers *lexer.o*, it cannot find a corresponding source file (even in *src*) so it assumes this (nonexistent source) is an intermediate file and looks for a way to make *lexer.c* from some other source file. It discovers a rule to create a *.c* file from a *.l* file and notices that *lexer.l* exists. There is no action required to update *lexer.l*, so it moves on to the command for updating *lexer.c*, which yields the flex command line. Next make updates the object file from the C source. Using sequences of rules like this to update a target is called *rule chaining*.

Next, make examines the library specification -lfl. It searches the standard library directories for the system and discovers */lib/libfl.a*.

Now make has all the prerequisites for updating *count_words*, so it executes the final gcc command. Lastly, make realizes it created an intermediate file that is not necessary to keep so it cleans it up.

As you can see, using rules in *makefile*s allows you to omit a lot of detail. Rules can have complex interactions that yield very powerful behaviors. In particular, having a built-in database of common rules makes many types of *makefile* specifications very simple.

The built-in rules can be customized by changing the values of the variables in the command scripts. A typical rule has a host of variables, beginning with the program to execute, and including variables to set major groupings of command-line options, such as the output file, optimization, debugging, etc. You can look at make's default set of rules (and variables) by running make --print-data-base.

The Patterns

The percent character in a pattern rule is roughly equivalent to * in a Unix shell. It represents any number of any characters. The percent character can be placed

anywhere within the pattern but can occur only once. Here are some valid uses of percent:

```
%,v
s%.o
wrapper_%
```

Characters other than percent match literally within a filename. A pattern can contain a prefix or a suffix or both. When make searches for a pattern rule to apply, it first looks for a matching pattern rule target. The pattern rule target must start with the prefix and end with the suffix (if they exist). If a match is found, the characters between the prefix and suffix are taken as the stem of the name. Next make looks at the prerequisites of the pattern rule by substituting the stem into the prerequisite pattern. If the resulting filename exists or can be made by applying another rule, a match is made and the rule is applied. The stem word must contain at least one character.

It is also possible to have a pattern containing only a percent character. The most common use of this pattern is to build a Unix executable program. For instance, here are several pattern rules GNU make includes for building programs:

```
%: %.mod
        $(COMPILE.mod) -o $@ -e $@ $^

%: %.cpp
        $(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@

%: %.sh
        cat $< >$@
        chmod a+x $@
```

These patterns will be used to generate an executable from a Modula source file, a preprocessed C source file, and a Bourne shell script, respectively. We will see many more implicit rules in the section "The Implicit Rules Database."

Static Pattern Rules

A static pattern rule is one that applies only to a specific list of targets.

```
$(OBJECTS): %.o: %c
        $(CC) -c $(CFLAGS) $< -o $@
```

The only difference between this rule and an ordinary pattern rule is the initial `$(OBJECTS):` specification. This limits the rule to the files listed in the `$(OBJECTS)` variable.

This is very similar to a pattern rule. Each object file in `$(OBJECTS)` is matched against the pattern `%.o` and its stem is extracted. The stem is then substituted into the pattern `%.c` to yield the target's prerequisite. If the target pattern does not exist, make issues a warning.

Use static pattern rules whenever it is easier to list the target files explicitly than to identify them by a suffix or other pattern.

Suffix Rules

Suffix rules are the original (and obsolete) way of defining implicit rules. Because other versions of make may not support GNU make's pattern rule syntax, you will still see suffix rules in *makefiles* intended for a wide distribution so it is important to be able to read and understand the syntax. So, although compiling GNU make for the target system is the preferred method for *makefile* portability, you may still need to write suffix rules in rare circumstances.

Suffix rules consist of one or two suffixes concatenated and used as a target:

```
.c.o:
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

This is a little confusing because the prerequisite suffix comes first and the target suffix second. This rule matches the same set of targets and prerequisites as:

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

The suffix rule forms the stem of the file by removing the target suffix. It forms the prerequisite by replacing the target suffix with the prerequisite suffix. The suffix rule is recognized by make only if the two suffixes are in a list of known suffixes.

The above suffix rule is known as a double-suffix rule since it contains two suffixes. There are also single-suffix rules. As you might imagine a single-suffix rule contains only one suffix, the suffix of the source file. These rules are used to create executables since Unix executables do not have a suffix:

```
.p:
        $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

This rule produces an executable image from a Pascal source file. This is completely analogous to the pattern rule:

```
%: %.p
        $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

The known suffix list is the strangest part of the syntax. A special target, .SUFFIXES, is used to set the list of known suffixes. Here is the first part of the default .SUFFIXES definition:

```
.SUFFIXES: .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l
```

You can add your own suffixes by simply adding a .SUFFIXES rule to your *makefile*:

```
.SUFFIXES: .pdf .fo .html .xml
```

If you want to delete all the known suffixes (because they are interfering with your special suffixes) simply specify no prerequisites:

```
.SUFFIXES:
```

You can also use the command-line option --no-builtin-rules (or -r).

We will not use this old syntax in the rest of this book because GNU make's pattern rules are clearer and more general.

The Implicit Rules Database

GNU make 3.80 has about 90 built-in implicit rules. An implicit rule is either a pattern rule or a suffix rule (which we will discuss briefly later). There are built-in pattern rules for C, C++, Pascal, FORTRAN, ratfor, Modula, Texinfo, TEX (including Tangle and Weave), Emacs Lisp, RCS, and SCCS. In addition, there are rules for supporting programs for these languages, such as cpp, as, yacc, lex, tangle, weave and dvi tools.

If you are using any of these tools, you'll probably find most of what you need in the built-in rules. If you're using some unsupported languages such as Java or XML, you will have to write rules of your own. But don't worry, you typically need only a few rules to support a language and they are quite easy to write.

To examine the rules database built into make, use the --print-data-base command-line option (-p for short). This will generate about a thousand lines of output. After version and copyright information, make prints its variable definitions each one preceded by a comment indicating the "origin" of the definition. For instance, variables can be environment variables, default values, automatic variables, etc. After the variables, come the rules. The actual format used by GNU make is:

```
%: %.C
#  commands to execute (built-in):
        $(LINK.C) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

For rules defined by the *makefile*, the comment will include the file and line where the rule was defined:

```
%.html: %.xml
#  commands to execute (from `Makefile', line 168):
        $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<
```

Working with Implicit Rules

The built-in implicit rules are applied whenever a target is being considered and there is no explicit rule to update it. So using an implicit rule is easy: simply do not specify a command script when adding your target to the *makefile*. This causes make to search its built-in database to satisfy the target. Usually this does just what you want, but in rare cases your development environment can cause problems. For instance, suppose you have a mixed language environment consisting of Lisp and C source code. If the file *editor.l* and *editor.c* both exist in the same directory (say one is a low-level implementation accessed by the other) make will believe that the Lisp file is really a flex file (recall flex files use the *.l* suffix) and that the C source is the output

of the flex command. If *editor.o* is a target and *editor.l* is newer than *editor.c*, make will attempt to "update" the C file with the output of flex overwriting your source code. Gack.

To work around this particular problem you can delete the two rules concerning flex from the built-in rule base like this:

```
%.o: %.l
%.c: %.l
```

A pattern with no command script will remove the rule from make's database. In practice, situations such as this are very rare. However, it is important to remember the built-in rules database contains rules that will interact with your own *makefile*s in ways you may not have anticipated.

We have seen several examples of how make will "chain" rules together while trying to update a target. This can lead to some complexity, which we'll examine here. When make considers how to update a target, it searches the implicit rules for a target pattern that matches the target in hand. For each target pattern that matches the target file, make will look for an existing matching prerequisite. That is, after matching the target pattern, make immediately looks for the prerequisite "source" file. If the prerequisite is found, the rule is used. For some target patterns, there are many possible source files. For instance, a *.o* file can be made from *.c*, *.cc*, *.cpp*, *.p*, *.f*, *.r*, *.s*, and *.mod* files. But what if the source is not found after searching all possible rules? In this case, make will search the rules again, this time assuming that the matching source file should be considered as a new target for updating. By performing this search recursively, make can find a "chain" of rules that allows updating a target. We saw this in our *lexer.o* example. make was able to update the *lexer.o* target from *lexer.l* even though the intermediate *.c* file was missing by invoking the *.l* to *.c* rule, then the *.c* to *.o* rule.

One of the more impressive sequences that make can produce automatically from its database is shown here. First, we setup our experiment by creating an empty yacc source file and registering with RCS using ci (that is, we want a version-controlled yacc source file):

```
$ touch foo.y
$ ci foo.y
foo.y,v  <--  foo.y
.
initial revision: 1.1
done
```

Now, we ask make how it would create the executable *foo*. The --just-print (or -n) option tells make to report what actions it would perform without actually running them. Notice that we have no *makefile* and no "source" code, only an RCS file:

```
$ make -n foo
co  foo.y,v foo.y
foo.y,v  -->  foo.y
```

```
revision 1.1
done
bison -y  foo.y
mv -f y.tab.c foo.c
gcc    -c -o foo.o foo.c
gcc   foo.o   -o foo
rm foo.c foo.o foo.y
```

Following the chain of implicit rules and prerequisites, make determined it could create the executable, *foo*, if it had the object file *foo.o*. It could create *foo.o* if it had the C source file *foo.c*. It could create *foo.c* if it had the yacc source file *foo.y*. Finally, it realized it could create *foo.y* by checking out the file from the RCS file *foo.y,v*, which it actually has. Once make has formulated this plan, it executes it by checking out *foo.y* with co, transforming it into *foo.c* with bison, compiling it into *foo.o* with gcc, and linking it to form *foo* again with gcc. All this from the implicit rules database. Pretty cool.

The files generated by chaining rules are called *intermediate* files and are treated specially by make. First, since intermediate files do not occur in targets (otherwise they would not be intermediate), make will never simply update an intermediate file. Second, because make creates intermediate files itself as a side effect of updating a target, make will delete the intermediates before exiting. You can see this in the last line of the example.

Rule Structure

The built-in rules have a standard structure intended to make them easily customizable. Let's go over the structure briefly then talk about customization. Here is the (by now familiar) rule for updating an object file from its C source:

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

The customization of this rule is controlled entirely by the set of variables it uses. We see two variables here, but COMPILE.c in particular is defined in terms of several other variables:

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
CC = gcc
OUTPUT_OPTION = -o $@
```

The C compiler itself can be changed by altering the value of the CC variable. The other variables are used for setting compilation options (CFLAGS), preprocessor options (CPPFLAGS), and architecture-specific options (TARGET_ARCH).

The variables in a built-in rule are intended to make customizing the rule as easy as possible. For that reason, it is important to be very careful when setting these variables in your *makefile*. If you set these variables in a naive way, you destroy the end user's ability to customize them. For instance, given this assignment in a *makefile*:

```
CPPFLAGS = -I project/include
```

If the user wanted to add a CPP define to the command line, they would normally invoke make like:

```
$ make CPPFLAGS=-DDEBUG
```

But in so doing they would accidentally remove the -I option that is (presumably) required for compiling. Variables set on the command line override all other assignments to the variable. (See the section "Where Variables Come From" in Chapter 3 for more details on command-line assignments). So, setting CPPFLAGS inappropriately in the *makefile* "broke" a customization feature that most users would expect to work. Instead of using simple assignment, consider redefining the compilation variable to include your own variables:

```
COMPILE.c = $(CC) $(CFLAGS) $(INCLUDES) $(CPPFLAGS) $(TARGET_ARCH) -c
INCLUDES = -I project/include
```

Or you can use append-style assignment, which is discussed in the section "Other Types of Assignment" in Chapter 3.

Implicit Rules for Source Control

make knows about two source code control systems, RCS and SCCS, and supports their use with built-in implicit rules. Unfortunately, it seems the state of the art in source code control and modern software engineering have left make behind. I've never found a use for the source control support in make, nor have I seen it used in other production software. I do not recommend the use of this feature. There are a number of reasons for this.

First, the source control tools supported by make, RCS and SCCS, although valuable and venerable tools in their day, have largely been supplanted by CVS, the Concurrent Version System, or proprietary tools. In fact, CVS uses RCS to manage individual files internally. However, using RCS directly proved to be a considerable problem when a project spanned more than one directory or more than one developer. CVS, in particular, was implemented to fill the gaps in RCS's functionality in precisely these areas. Support for CVS has never been added to make, which is probably a good thing.*

It is now well recognized that the life cycle of software becomes complex. Applications rarely move smoothly from one release to the next. More typically, one or more distinct releases of an application are being used in the field (and require bug fix support), while one or more versions are in active development. CVS provides powerful features to help manage these parallel versions of the software. But it also means that a developer must be very aware of the specific version of the code she is working on. Having the *makefile* automatically check out source during a compilation begs the

* CVS is, in turn, becoming supplanted by newer tools. While it is currently the most ubiquitous source control system, subversion (*http://subversion.tigris.org*) looks to be the new wave.

question of what source is being checked out and whether the newly checked out source is compatible with the source already existing in the developer's working directories. In many production environments, developers are working on three or more distinct versions of the same application in a single day. Keeping this complexity in check is hard enough without having software quietly updating your source tree for you.

Also, one of the more powerful features of CVS is that it allows access to remote repositories. In most production environments, the CVS repository (the database of controlled files) is not located on the developer's own machine, but on a server. Although network access is now quite fast (particularly on a local area network) it is not a good idea to have `make` probing the network server in search of source files. The performance impact would be disastrous.

So, although it is possible to use the built-in implicit rules to interface more or less cleanly with RCS and SCCS, there are no rules to access CVS for gathering source files or *makefile*. Nor do I think it makes much sense to do so. On the other hand, it is quite reasonable to use CVS in *makefile*s. For instance, to ensure that the current source is properly checked in, that the release number information is managed properly, or that test results are correct. These are uses of CVS by *makefile* authors rather than issues of CVS integration with `make`.

A Simple Help Command

Large *makefile*s can have many targets that are difficult for users to remember. One way to reduce this problem is to make the default target a brief help command. However, maintaining the help text by hand is always a problem. To avoid this, you can gather the available commands directly from `make`'s rules database. The following target will present a sorted four column listing of the available `make` targets:

```
# help - The default goal
.PHONY: help
help:
	$(MAKE) --print-data-base --question |              \
	$(AWK) '/^[^.%][-A-Za-z0-9_]*:/                     \
	      { print substr($$1, 1, length($$1)-1) }' |    \
	$(SORT) |                                            \
	$(PR) --omit-pagination --width=80 --columns=4
```

The command script consists of a single pipeline. The `make` rule database is dumped using the `--print-data-base` command. Using the `--question` option prevents `make` from running any actual commands. The database is then passed through a simple `awk` filter that grabs every line representing a target that does not begin with percent or period (pattern rules and suffix rules, respectively) and discards extra information on the line. Finally, the target list is sorted and printed in a simple four-column listing.

Another approach to the same command (my first attempt), used the awk command on the *makefile* itself. This required special handling for included *makefile*s (covered in the section “The include Directive” in Chapter 3) and could not handle generated rules at all. The version presented here handles all that automatically by allowing make to process these elements and report the resulting rule set.

Special Targets

A *special target* is a built-in phony target used to change make’s default behavior. For instance, .PHONY, a special target, which we’ve already seen, declares that its prerequisite does not refer to an actual file and should always be considered out of date. The .PHONY target is the most common special target you will see, but there are others as well.

These special targets follow the syntax of normal targets, that is *target*: *prerequisite*, but the *target* is not a file or even a normal phony. They are really more like directives for modifying make’s internal algorithms.

There are twelve special targets. They fall into three categories: as we’ve just said many are used to alter the behavior of make when updating a target, another set act simply as global flags to make and ignore their targets, finally the .SUFFIXES special target is used when specifying old-fashioned suffix rules (discussed in the section “Suffix Rules” earlier in this chapter).

The most useful target modifiers (aside from .PHONY) are:

.INTERMEDIATE

Prerequisites of this special target are treated as intermediate files. If make creates the file while updating another target, the file will be deleted automatically when make exits. If the file already exists when make considers updating the file, the file will not be deleted.

This can be very useful when building custom rule chains. For instance, most Java tools accept Windows-like file lists. Creating rules to build the file lists and marking their output files as intermediate allows make to clean up many temporary files.

.SECONDARY

Prerequisites of this special target are treated as intermediate files but are never automatically deleted. The most common use of .SECONDARY is to mark object files stored in libraries. Normally these object files will be deleted as soon as they are added to an archive. Sometimes it is more convenient during development to keep these object files, but still use the make support for updating archives.

.PRECIOUS

When make is interrupted during execution, it may delete the target file it is updating if the file was modified since make started. This is so make doesn’t leave

a partially constructed (possibly corrupt) file laying around in the build tree. There are times when you don't want this behavior, particularly if the file is large and computationally expensive to create. If you mark the file as precious, make will never delete the file if interrupted.

Use of .PRECIOUS is relatively rare, but when it is needed it is often a life saver. Note that make will not perform an automatic delete if the commands of a rule generate an error. It does so only when interrupted by a signal.

.DELETE_ON_ERROR
: This is sort of the opposite of .PRECIOUS. Marking a target as .DELETE_ON_ERROR says that make *should* delete the target if any of the commands associated with the rule generates an error. make normally only deletes the target if it is interrupted by a signal.

The other special targets will be covered later when their use is more relevant. We'll discuss .EXPORT_ALL_VARIABLES in Chapter 3 and the targets relating to parallel execution in Chapter 10.

Automatic Dependency Generation

When we refactored our word counting program to use header files, a thorny little problem snuck up on us. We added the dependency between the object files and the C header files to the *makefile* by hand. It was easy to do in this case, but in normal programs (not toy examples) this can be tedious and error-prone. In fact, in most programs it is virtually impossible because most header files include other header files forming a complex tree. For instance, on my system, the single header file *stdio.h* (the most commonly referenced header file in C) expands to include 15 other header files. Resolving these relationships by hand is a hopeless task. But failing to recompile files can lead to hours of debugging headaches or worse, bugs in the resulting program. So what do we do?

Well, computers are pretty good at searching and pattern matching. Let's use a program to identify the relationships between files and maybe even have this program write out these dependencies in *makefile* syntax. As you have probably guessed, such a program already exists—at least for C/C++. There is a option to gcc and many other C/C++ compilers that will read the source and write *makefile* dependencies. For instance, here is how I found the dependencies for *stdio.h*:

```
$ echo "#include <stdio.h>" > stdio.c
$ gcc -M stdio.c
stdio.o: stdio.c /usr/include/stdio.h /usr/include/_ansi.h \
  /usr/include/newlib.h /usr/include/sys/config.h \
  /usr/include/machine/ieeefp.h /usr/include/cygwin/config.h \
  /usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/stddef.h \
  /usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/stdarg.h \
  /usr/include/sys/reent.h /usr/include/sys/_types.h \
  /usr/include/sys/types.h /usr/include/machine/types.h \
```

```
/usr/include/sys/features.h /usr/include/cygwin/types.h \
/usr/include/sys/sysmacros.h /usr/include/stdint.h \
/usr/include/sys/stdio.h
```

"Fine." I hear you cry, "Now I need to run gcc and use an editor to paste the results of -M into my *makefile*s. What a pain." And you'd be right if this was the whole answer. There are two traditional methods for including automatically generated dependencies into *makefile*s. The first and oldest is to add a line such as:

```
# Automatically generated dependencies follow - Do Not Edit
```

to the end of the *makefile* and then write a shell script to update these generated lines. This is certainly better than updating them by hand, but it's also very ugly. The second method is to add an include directive to the make program. By now most versions of make have the include directive and GNU make most certainly does.

So, the trick is to write a *makefile* target whose action runs gcc over all your source with the -M option, saves the results in a dependency file, and then re-runs make including the generated dependency file in the *makefile* so it can trigger the updates we need. Before GNU make, this is exactly what was done and the rule looked like:

```
depend: count_words.c lexer.c counter.c
        $(CC) -M $(CPPFLAGS) $^ > $@

include depend
```

Before running make to build the program, you would first execute make depend to generate the dependencies. This was good as far as it went, but often people would add or remove dependencies from their source without regenerating the *depend* file. Then source wouldn't get recompiled and the whole mess started again.

GNU make solved this last niggling problem with a cool feature and a simple algorithm. First, the algorithm. If we generated each source file's dependencies into its own dependency file with, say, a *.d* suffix and added the *.d* file itself as a target to this dependency rule, then make could know that the *.d* needed to be updated (along with the object file) when the source file changed:

```
counter.o counter.d: src/counter.c include/counter.h include/lexer.h
```

Generating this rule can be accomplished with a pattern rule and a (fairly ugly) command script (this is taken directly from the GNU make manual):*

* This is an impressive little command script, but I think it requires some explanation. First, we use the C compiler with the -M option to create a temporary file containing the dependencies for this target. The temporary filename is created from the target, `$@`, with a unique numeric suffix added, `$$$$`. In the shell, the variable `$$` returns the process number of the currently running shell. Since process numbers are unique, this produces a unique filename. We then use sed to add the *.d* file as a target to the rule. The sed expression consists of a search part, `\($*\)\.o[ :]*`, and a replacement part, `\1.o $@ :`, separated by commas. The search expression begins with the target stem, `$*`, enclosed in a regular expression (RE) group, `\(\)`, followed by the file suffix, `\.o`. After the target filename, there come zero or more spaces or colons, `[ :]*`. The replacement portion restores the original target by referencing the first RE group and appending the suffix, `\1.o`, then adding the dependency file target, `$@`.

```
%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$;                  \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
        rm -f $@.$$$$
```

Now, for the cool feature. make will treat any file named in an include directive as a target to be updated. So, when we mention the *.d* files we want to include, make will automatically try to create these files as it reads the *makefile*. Here is our *makefile* with the addition of automatic dependency generation:

```
VPATH    = src include
CPPFLAGS = -I include

SOURCES  = count_words.c \
           lexer.c       \
           counter.c

count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h

include $(subst .c,.d,$(SOURCES))

%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$;                      \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@;     \
        rm -f $@.$$$$
```

The include directive should always be placed after the hand-written dependencies so that the default goal is not hijacked by some dependency file. The include directive takes a list of files (whose names can include wildcards). Here we use a make function, subst, to transform the list of source files into a list of dependency filenames. (We'll discuss subst in detail in the section "String Functions" in Chapter 4.) For now, just note that this use replaces the string *.c* with *.d* in each of the words in $(SOURCES).

When we run this *makefile* with the --just-print option, we get:

```
$ make --just-print
Makefile:13: count_words.d: No such file or directory
Makefile:13: lexer.d: No such file or directory
Makefile:13: counter.d: No such file or directory
gcc -M -I include src/counter.c > counter.d.$$;                \
sed 's,\(counter\)\.o[ :]*,\1.o counter.d : ,g' < counter.d.$$ >
counter.d; \
rm -f counter.d.$$
flex  -t src/lexer.l > lexer.c
gcc -M -I include lexer.c > lexer.d.$$;                \
sed 's,\(lexer\)\.o[ :]*,\1.o lexer.d : ,g' < lexer.d.$$ > lexer.d;
\
rm -f lexer.d.$$
gcc -M -I include src/count_words.c > count_words.d.$$;
```

```
\
sed 's,\(count_words\)\.o[ :]*,\1.o count_words.d : ,g' < count_words.d.
$$
count_words.d;     \
rm -f count_words.d.$$
rm lexer.c
gcc  -I include  -c -o count_words.o src/count_words.c
gcc  -I include  -c -o counter.o src/counter.c
gcc  -I include  -c -o lexer.o lexer.c
gcc   count_words.o counter.o lexer.o /lib/libfl.a   -o count_words
```

At first the response by `make` is a little alarming—it looks like a `make` error message. But not to worry, this is just a warning. `make` looks for the include files and doesn't find them, so it issues the `No such file or directory` warning before searching for a rule to create these files. This warning can be suppressed by preceding the `include` directive with a hyphen (-). The lines following the warnings show `make` invoking `gcc` with the `-M` option, then running the `sed` command. Notice that `make` must invoke `flex` to create *lexer.c*, then it deletes the temporary *lexer.c* before beginning to satisfy the default goal.

This gives you a taste of automatic dependency generation. There's lots more to say, such as how do you generate dependencies for other languages or build tree layouts. We'll return to this topic in more depth in Part II of this book.

Managing Libraries

An *archive* library, usually called simply a library or archive, is a special type of file containing other files called *members*. Archives are used to group related object files into more manageable units. For example, the C standard library *libc.a* contains low-level C functions. Libraries are very common so `make` has special support for creating, maintaining, and referencing them. Archives are created and modified with the `ar` program.

Let's look at an example. We can modify our word counting program by refactoring the reusable parts into a reusable library. Our library will consist of the two files *counter.o* and *lexer.o*. The `ar` command to create this library is:

```
$ ar rv libcounter.a counter.o lexer.o
a - counter.o
a - lexer.o
```

The options `rv` indicate that we want to *r*eplace members of the archive with the object files listed and that `ar` should *v*erbosely echo its actions. We can use the replace option even though the archive doesn't exist. The first argument after the options is the archive name followed by a list of object files. (Some versions of `ar` also require the "c" option, for *c*reate, if the archive does not exist but GNU `ar` does not.)

The two lines following the ar command are its verbose output indicating the object files were added.

Using the replace option to ar allows us to create or update an archive incrementally:

```
$ ar rv libcounter.a counter.o
r - counter.o
$ ar rv libcounter.a lexer.o
r - lexer.o
```

Here ar echoed each action with "r" to indicate the file was replaced in the archive.

A library can be linked into an executable in several ways. The most straightforward way is to simply list the library file on the command line. The compiler or linker will use the file suffix to determine the type of a particular file on the command line and do the Right Thing™:

```
cc count_words.o libcounter.a /lib/libfl.a -o count_words
```

Here cc will recognize the two files *libcounter.a* and */lib/libfl.a* as libraries and search them for undefined symbols. The other way to reference libraries on the command line is with the -l option:

```
cc count_words.o -lcounter -lfl -o count_words
```

As you can see, with this option you omit the prefix and suffix of the library filename. The -l option makes the command line more compact and easier to read, but it has a far more useful function. When cc sees the -l option it *searches* for the library in the system's standard library directories. This relieves the programmer from having to know the precise location of a library and makes the command line more portable. Also, on systems that support shared libraries (libraries with the extension *.so* on Unix systems), the linker will search for a shared library first, before searching for an archive library. This allows programs to benefit from shared libraries without specifically requiring them. This is the default behavior of GNU's linker/compiler. Older linker/compilers may not perform this optimization.

The search path used by the compiler can be changed by adding -L options indicating the directories to search and in what order. These directories are added before the system libraries and are used for all -l options on the command line. In fact, the last example fails to link because the current directory is not in cc's library search path. We can fix this error by adding the current directory like this:

```
cc count_words.o -L. -lcounter -lfl -o count_words
```

Libraries add a bit of complication to the process of building a program. How can make help to simplify the situation? GNU make includes special features to support both the creation of libraries and their use in linking programs. Let's see how they work.

Creating and Updating Libraries

Within a *makefile*, a library file is specified with its name just like any other file. A simple rule to create our library is:

```
libcounter.a: counter.o lexer.o
        $(AR) $(ARFLAGS) $@ $^
```

This uses the built-in definition for the `ar` program in `AR` and the standard options `rv` in `ARFLAGS`. The archive output file is automatically set in `$@` and the prerequisites are set in `$^`.

Now, if you make *libcounter.a* a prerequisite of *count_words* `make` will update our library before linking the executable. Notice one small irritation, however. All members of the archive are replaced even if they have not been modified. This is a waste of time and we can do better:

```
libcounter.a: counter.o lexer.o
        $(AR) $(ARFLGS) $@ $?
```

If you use `$?` instead of `$^`, `make` will pass only those objects files that are newer than the target to `ar`.

Can we do better still? Maybe, but maybe not. `make` has support for updating individual files within an archive, executing one `ar` command for each object file member, but before we go delving into those details there are several points worth noting about this style of building libraries. One of the primary goals of `make` is to use the processor efficiently by updating only those files that are out of date. Unfortunately, the style of invoking `ar` once for each out-of-date member quickly bogs down. If the archive contains more than a few dozen files, the expense of invoking `ar` for each update begins to outweigh the "elegance" factor of using the syntax we are about to introduce. By using the simple method above and invoking `ar` in an explicit rule, we can execute `ar` once for all files and save ourselves many fork/exec calls. In addition, on many systems using the `r` to `ar` is very inefficient. On my 1.9 GHz Pentium 4, building a large archive from scratch (with 14,216 members totaling 55 MB) takes 4 minutes 24 seconds. However, updating a single object file with `ar r` on the resulting archive takes 28 seconds. So building the archive from scratch is faster if I need to replace more than 10 files (out of 14,216!). In such situations it is probably more prudent to perform a single update of the archive with all modified object files using the `$?` automatic variable. For smaller libraries and faster processors there is no performance reason to prefer the simple approach above to the more elegant one below. In those situations, using the special library support that follows is a fine approach.

In GNU `make`, a member of an archive can be referenced using the notation:

```
libgraphics.a(bitblt.o): bitblt.o
        $(AR) $(ARFLAGS) $@ $<
```

Here the library name is *libgraphics.a* and the member name is *bitblt.o* (for *bit block transfer*). The syntax *libname*.a(*module*.o) refers to the module contained within the

library. The prerequisite for this target is simply the object file itself and the command adds the object file to the archive. The automatic variable `$<` is used in the command to get only the first prerequisite. In fact, there is a built-in pattern rule that does exactly this.

When we put this all together, our *makefile* looks like this:

```
VPATH    = src include
CPPFLAGS = -I include

count_words: libcounter.a /lib/libfl.a

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)

libcounter.a(lexer.o): lexer.o
        $(AR) $(ARFLAGS) $@ $<

libcounter.a(counter.o): counter.o
        $(AR) $(ARFLAGS) $@ $<

count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

When executed, `make` produces this output:

```
$ make
gcc  -I include  -c -o count_words.o src/count_words.c
flex  -t src/lexer.l> lexer.c
gcc  -I include  -c -o lexer.o lexer.c
ar rv libcounter.a lexer.o
ar: creating libcounter.a
a - lexer.o
gcc  -I include  -c -o counter.o src/counter.c
ar rv libcounter.a counter.o
a - counter.o
gcc   count_words.o libcounter.a /lib/libfl.a   -o count_words
rm lexer.c
```

Notice the archive updating rule. The automatic variable `$@` is expanded to the library name even though the target in the *makefile* is *libcounter.a(lexer.o)*.

Finally, it should be mentioned that an archive library contains an index of the symbols it contains. Newer archive programs such as GNU `ar` manage this index automatically when a new module is added to the archive. However, many older versions of `ar` do not. To create or update the index of an archive another program `ranlib` is used. On these systems, the built-in implicit rule for updating archives is insufficient. For these systems, a rule such as:

```
libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
        $(RANLIB) $@
```

must be used. Or if you choose to use the alternate approach for large archives:

```
libcounter.a: counter.o lexer.o
        $(RM) $@
        $(AR) $(ARFLGS) $@ $^
        $(RANLIB) $@
```

Of course, this syntax for managing the members of an archive can be used with the built-in implicit rules as well. GNU make comes with a built-in rule for updating an archive. When we use this rule, our *makefile* becomes:

```
VPATH    = src include
CPPFLAGS = -I include

count_words: libcounter.a -lfl
libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

Using Libraries as Prerequisites

When libraries appear as prerequisites, they can be referenced using either a standard filename or with the -l syntax. When filename syntax is used:

```
xpong: $(OBJECTS) /lib/X11/libX11.a /lib/X11/libXaw.a
        $(LINK) $^ -o $@
```

the linker will simply read the library files listed on the command line and process them normally. When the -l syntax is used, the prerequisites aren't proper files at all:

```
xpong: $(OBJECTS) -lX11 -lXaw
        $(LINK) $^ -o $@
```

When the -l form is used in a prerequisite, make will search for the library (preferring a shared library) and substitute its value, as an absolute path, into the $^ and $? variables. One great advantage of the second form is that it allows you to use the search and shared library preference feature even when the system's linker cannot perform these duties. Another advantage is that you can customize make's search path so it can find your application's libraries as well as system libraries. In this case, the first form would ignore the shared library and use the archive library since that is what was specified on the link line. In the second form, make knows that shared libraries are preferred and will search first for a shared version of *X11* before settling for the archive version. The pattern for recognizing libraries from the -l format is stored in .LIBPATTERNS and can be customized for other library filename formats.

Unfortunately, there is a small wrinkle. If a *makefile* specifies a library file target, it cannot use the -l option for that file in a prerequisite. For instance, the following *makefile*:

```
count_words: count_words.o -lcounter -lfl
        $(CC) $^ -o $@

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
```

fails with the error:

```
No rule to make target `-lcounter', needed by `count_words'
```

It appears that this error occurs because make does not expand -lcounter to *libcounter.a* and search for a target, but instead does a straight library search. So for libraries built within the *makefile*, the filename form must be used.

Getting complex programs to link without error can be somewhat of a black art. The linker will search libraries in the order in which they are listed on the command line. So if library *A* includes an undefined symbol, say open, that is defined in library *B*, the link command line must list *A* before *B* (that is, *A* requires *B*). Otherwise, when the linker reads *A* and sees the undefined symbol open, it's too late to go back to *B*. The linker doesn't ever go back. As you can see, the order of libraries on the command line is of fundamental importance.

When the prerequisites of a target are saved in the $^ and $? variables, their order is preserved. So using $^ as in the previous example expands to the same files in the same order as the prerequisites list. This is true even when the prerequisites are split across multiple rules. In that case, the prerequisites of each rule are appended to the target prerequisite list in the order they are seen.

A closely related problem is mutual reference between libraries, often referred to as *circular references* or *circularities*. Suppose a change is made and library *B* now references a symbol defined in library *A*. We know *A* must come before *B*, but now *B* must come before *A*. Hmm, a problem. The solution is to reference *A* both before and *after* *B*: -l*A* -l*B* -l*A*. In large, complex programs, libraries often need to be repeated in this way, sometimes more than twice.

This situation poses a minor problem for make because the automatic variables normally discard duplicates. For example, suppose we need to repeat a library prerequisite to satisfy a library circularity:

```
xpong: xpong.o libui.a libdynamics.a libui.a -lX11
        $(CC) $^ -o $@
```

This prerequisite list will be processed into the following link command:

```
gcc xpong.o libui.a libdynamics.a /usr/lib/X11R6/libX11.a -o xpong
```

Oops. To overcome this behavior of `$^` an additional variable is available in `make`, `$+`. This variable is identical to `$^` with the exception that duplicate prerequisites are preserved. Using `$+`:

```
xpong: xpong.o libui.a libdynamics.a libui.a -lX11
        $(CC) $+ -o $@
```

This prerequisite list will be processed into the following link command:

```
gcc xpong.o libui.a libdynamics.a libui.a /usr/lib/X11R6/libX11.a -o xpong
```

Double-Colon Rules

Double-colon rules are an obscure feature that allows the same target to be updated with different commands depending on which set of prerequisites are newer than the target. Normally, when a target appears more than once all the prerequisites are appended in a long list with only one command script to perform the update. With double-colon rules, however, each occurrence of the target is considered a completely separate entity and is handled individually. This means that for a particular target, all the rules must be of the same type, either they are all double-colon rules or all single-colon rules.

Realistic, useful examples of this feature are difficult to come by (which is why it is an obscure feature), but here is an artificial example:

```
file-list:: generate-list-script
        chmod +x $<
        generate-list-script $(files) > file-list

file-list:: $(files)
        generate-list-script $(files) > file-list
```

We can regenerate the *file-list* target two ways. If the generating script has been updated, we make the script executable, then run it. If the source files have changed, we simply run the script. Although a bit far-fetched, this gives you a feel for how the feature might be used.

We've covered most of the features of `make` rules and, along with variables and commands, this is the essence of `make`. We've focused largely on the specific syntax and behavior of the features without going much into how to apply them in more complex situations. That is the subject of Part II. For now, we will continue our discussion with variables and then commands.