PART I
Basic Concepts

In Part I, we focus on the features of make, what they do, and how to use them properly. We begin with a brief introduction and overview that should be enough to get you started on your first *makefile*. The chapters then cover make rules, variables, functions, and finally command scripts.

When you are finished with Part I, you will have a fairly complete working knowledge of GNU make and have many advanced usages well in hand.

CHAPTER 1

How to Write a Simple Makefile

The mechanics of programming usually follow a fairly simple routine of editing source files, compiling the source into an executable form, and debugging the result. Although transforming the source into an executable is considered routine, if done incorrectly a programmer can waste immense amounts of time tracking down the problem. Most developers have experienced the frustration of modifying a function and running the new code only to find that their change did not fix the bug. Later they discover that they were never executing their modified function because of some procedural error such as failing to recompile the source, relink the executable, or rebuild a jar. Moreover, as the program's complexity grows these mundane tasks can become increasingly error-prone as different versions of the program are developed, perhaps for other platforms or other versions of support libraries, etc.

The `make` program is intended to automate the mundane aspects of transforming source code into an executable. The advantages of `make` over scripts is that you can specify the relationships between the elements of your program to `make`, and it knows through these relationships and timestamps exactly what steps need to be redone to produce the desired program each time. Using this information, `make` can also optimize the build process avoiding unnecessary steps.

GNU `make` (and other variants of `make`) do precisely this. `make` defines a language for describing the relationships between source code, intermediate files, and executables. It also provides features to manage alternate configurations, implement reusable libraries of specifications, and parameterize processes with user-defined macros. In short, `make` can be considered the center of the development process by providing a roadmap of an application's components and how they fit together.

The specification that `make` uses is generally saved in a file named *makefile*. Here is a *makefile* to build the traditional "Hello, World" program:

```
hello: hello.c
        gcc hello.c -o hello
```

To build the program execute `make` by typing:

```
$ make
```

at the command prompt of your favorite shell. This will cause the `make` program to read the *makefile* and build the first target it finds there:

```
$ make
gcc hello.c -o hello
```

If a target is included as a command-line argument, that target is updated. If no command-line targets are given, then the first target in the file is used, called the *default goal*.

Typically the default goal in most *makefile*s is to build a program. This usually involves many steps. Often the source code for the program is incomplete and the source must be generated using utilities such as `flex` or `bison`. Next the source is compiled into binary object files (*.o* files for C/C++, *.class* files for Java, etc.). Then, for C/C++, the object files are bound together by a linker (usually invoked through the compiler, `gcc`) to form an executable program.

Modifying any of the source files and reinvoking `make` will cause some, but usually not all, of these commands to be repeated so the source code changes are properly incorporated into the executable. The specification file, or *makefile*, describes the relationship between the source, intermediate, and executable program files so that `make` can perform the minimum amount of work necessary to update the executable.

So the principle value of `make` comes from its ability to perform the complex series of commands necessary to build an application and to optimize these operations when possible to reduce the time taken by the edit-compile-debug cycle. Furthermore, `make` is flexible enough to be used anywhere one kind of file depends on another from traditional programming in C/C++ to Java, $\TeX$, database management, and more.

Targets and Prerequisites

Essentially a *makefile* contains a set of rules used to build an application. The first rule seen by `make` is used as the *default rule*. A *rule* consists of three parts: the target, its prerequisites, and the command(s) to perform:

```
target: prereq1 prereq2
        commands
```

The *target* is the file or thing that must be made. The *prerequisites* or *dependents* are those files that must exist before the target can be successfully created. And the *commands* are those shell commands that will create the target from the prerequisites.

Here is a rule for compiling a C file, *foo.c*, into an object file, *foo.o*:

```
foo.o: foo.c foo.h
        gcc -c foo.c
```

The target file *foo.o* appears before the colon. The prerequisites *foo.c* and *foo.h* appear after the colon. The command script usually appears on the following lines and is preceded by a tab character.

When make is asked to evaluate a rule, it begins by finding the files indicated by the prerequisites and target. If any of the prerequisites has an associated rule, make attempts to update those first. Next, the target file is considered. If any prerequisite is newer than the target, the target is remade by executing the commands. Each command line is passed to the shell and is executed in its own subshell. If any of the commands generates an error, the building of the target is terminated and make exits. One file is considered newer than another if it has been modified more recently.

Here is a program to count the number of occurrences of the words "fee," "fie," "foe," and "fum" in its input. It uses a flex scanner driven by a simple main:

```
#include <stdio.h>

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

int main( int argc, char ** argv )
{
    yylex( );
    printf( "%d %d %d %d\n", fee_count, fie_count, foe_count, fum_count );
    exit( 0 );
}
```

The scanner is very simple:

```
        int fee_count = 0;
        int fie_count = 0;
        int foe_count = 0;
        int fum_count = 0;
%%
fee     fee_count++;
fie     fie_count++;
foe     foe_count++;
fum     fum_count++;
```

The *makefile* for this program is also quite simple:

```
count_words: count_words.o lexer.o -lfl
        gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
        gcc -c count_words.c

lexer.o: lexer.c
        gcc -c lexer.c

lexer.c: lexer.l
        flex -t lexer.l > lexer.c
```

When this *makefile* is executed for the first time, we see:

```
$ make
gcc -c count_words.c
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

We now have an executable program. Of course, real programs typically consist of more modules than this. Also, as you will see later, this *makefile* does not use most of the features of `make` so it's more verbose than necessary. Nevertheless, this is a functional and useful *makefile*. For instance, during the writing of this example, I executed the *makefile* several dozen times while experimenting with the program.

As you look at the *makefile* and sample execution, you may notice that the order in which commands are executed by `make` are nearly the opposite to the order they occur in the *makefile*. This *top-down* style is common in *makefile*s. Usually the most general form of target is specified first in the *makefile* and the details are left for later. The `make` program supports this style in many ways. Chief among these is `make`'s two-phase execution model and recursive variables. We will discuss these in great detail in later chapters.

Dependency Checking

How did `make` decide what to do? Let's go over the previous execution in more detail to find out.

First `make` notices that the command line contains no targets so it decides to make the default goal, *count_words*. It checks for prerequisites and sees three: *count_words.o*, *lexer.o*, and `-lfl`. `make` now considers how to build *count_words.o* and sees a rule for it. Again, it checks the prerequisites, notices that *count_words.c* has no rules but that the file exists, so `make` executes the commands to transform *count_words.c* into *count_words.o* by executing the command:

```
gcc -c count_words.c
```

This "chaining" of targets to prerequisites to targets to prerequisites is typical of how `make` analyzes a *makefile* to decide the commands to be performed.

The next prerequisite `make` considers is *lexer.o*. Again the chain of rules leads to *lexer.c* but this time the file does not exist. `make` finds the rule for generating *lexer.c* from *lexer.l* so it runs the `flex` program. Now that *lexer.c* exists it can run the `gcc` command.

Finally, `make` examines `-lfl`. The `-l` option to `gcc` indicates a system library that must be linked into the application. The actual library name indicated by "fl" is *libfl.a*. GNU `make` includes special support for this syntax. When a prerequisite of the form-`l<NAME>` is seen, `make` searches for a file of the form *libNAME.so*; if no match is found, it then searches for *libNAME.a*. Here `make` finds */usr/lib/libfl.a* and proceeds with the final action, linking.

Minimizing Rebuilds

When we run our program, we discover that aside from printing fees, fies, foes, and fums, it also prints text from the input file. This is not what we want. The problem is that we have forgotten some rules in our lexical analyzer and `flex` is passing this unrecognized text to its output. To solve this problem we simply add an "any character" rule and a newline rule:

```
        int fee_count = 0;
        int fie_count = 0;
        int foe_count = 0;
        int fum_count = 0;
%%
fee     fee_count++;
fie     fie_count++;
foe     foe_count++;
fum     fum_count++;
.
\n
```

After editing this file we need to rebuild the application to test our fix:

```
$ make
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

Notice this time the file *count_words.c* was not recompiled. When `make` analyzed the rule, it discovered that *count_words.o* existed and was newer than its prerequisite *count_words.c* so no action was necessary to bring the file up to date. While analyzing *lexer.c*, however, `make` saw that the prerequisite *lexer.l* was newer than its target *lexer.c* so `make` must update *lexer.c*. This, in turn, caused the update of *lexer.o* and then *count_words*. Now our word counting program is fixed:

```
$ count_words < lexer.l
3 3 3 3
```

Invoking make

The previous examples assume that:

- All the project source code and the `make` description file are stored in a single directory.
- The `make` description file is called *makefile*, *Makefile*, or *GNUMakefile*.
- The *makefile* resides in the user's current directory when executing the `make` command.

When make is invoked under these conditions, it automatically creates the first target it sees. To update a different target (or to update more than one target) include the target name on the command line:

```
$ make lexer.c
```

When make is executed, it will read the description file and identify the target to be updated. If the target or any of its prerequisite files are out of date (or missing) the shell commands in the rule's command script will be executed one at a time. After the commands are run make assumes the target is up to date and moves on to the next target or exits.

If the target you specify is already up to date, make will say so and immediately exit, doing nothing else:

```
$ make lexer.c
make: `lexer.c' is up to date.
```

If you specify a target that is not in the *makefile* and for which there is no implicit rule (discussed in Chapter 2), make will respond with:

```
$ make non-existent-target
make: *** No rule to make target `non-existent-target'.  Stop.
```

make has many command-line options. One of the most useful is `--just-print` (or `-n`) which tells make to display the commands it would execute for a particular target without actually executing them. This is particularly valuable while writing *makefile*s. It is also possible to set almost any *makefile* variable on the command line to override the default value or the value set in the *makefile*.

Basic Makefile Syntax

Now that you have a basic understanding of make you can almost write your own *makefile*s. Here we'll cover enough of the syntax and structure of a *makefile* for you to start using make.

*Makefile*s are usually structured top-down so that the most general target, often called `all`, is updated by default. More and more detailed targets follow with targets for program maintenance, such as a `clean` target to delete unwanted temporary files, coming last. As you can guess from these target names, targets do not have to be actual files, any name will do.

In the example above we saw a simplified form of a rule. The more complete (but still not quite complete) form of a rule is:

```
target1 target2 target3 : prerequisite1 prerequisite2
        command1
        command2
        command3
```

One or more targets appear to the left of the colon and zero or more prerequisites can appear to the right of the colon. If no prerequisites are listed to the right, then only the target(s) that do not exist are updated. The set of commands executed to update a target are sometimes called the *command script*, but most often just the *commands*.

Each command *must* begin with a tab character. This (obscure) syntax tells make that the characters that follow the tab are to be passed to a subshell for execution. If you accidentally insert a tab as the first character of a noncommand line, make will interpret the following text as a command under most circumstances. If you're lucky and your errant tab character is recognized as a syntax error you will receive the message:

```
$ make
Makefile:6: *** commands commence before first target.  Stop.
```

We'll discuss the complexities of the tab character in Chapter 2.

The comment character for make is the hash or pound sign, #. All text from the pound sign to the end of line is ignored. Comments can be indented and leading whitespace is ignored. The comment character # does not introduce a make comment in the text of commands. The entire line, including the # and subsequent characters, is passed to the shell for execution. How it is handled there depends on your shell.

Long lines can be continued using the standard Unix escape character backslash (\). It is common for commands to be continued in this way. It is also common for lists of prerequisites to be continued with backslash. Later we'll cover other ways of handling long prerequisite lists.

You now have enough background to write simple *makefile*s. Chapter 2 will cover rules in detail, followed by make variables in Chapter 3 and commands in Chapter 5. For now you should avoid the use of variables, macros, and multiline command sequences.

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 `make`s. 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.

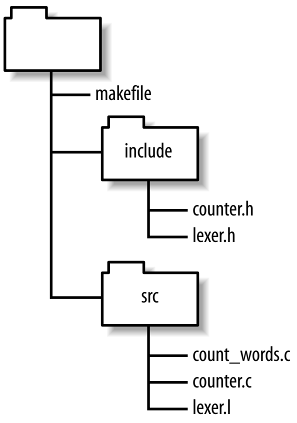

Figure 2-1. Example source tree layout

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 *makefile*s 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.

CHAPTER 3

Variables and Macros

We've been looking at *makefile* variables for a while now and we've seen many examples of how they're used in both the built-in and user-defined rules. But the examples we've seen have only scratched the surface. Variables and macros get much more complicated and give GNU `make` much of its incredible power.

Before we go any further, it is important to understand that `make` is sort of two languages in one. The first language describes dependency graphs consisting of targets and prerequisites. (This language was covered in Chapter 2.) The second language is a macro language for performing textual substitution. Other macro languages you may be familiar with are the C preprocessor, `m4`, $\TeX$, and macro assemblers. Like these other macro languages, `make` allows you to define a shorthand term for a longer sequence of characters and use the shorthand in your program. The macro processor will recognize your shorthand terms and replace them with their expanded form. Although it is easy to think of *makefile* variables as traditional programming language variables, there is a distinction between a macro "variable" and a "traditional" variable. A macro variable is expanded "in place" to yield a text string that may then be expanded further. This distinction will become more clear as we proceed.

A variable name can contain almost any characters including most punctuation. Even spaces are allowed, but if you value your sanity you should avoid them. The only characters actually disallowed in a variable name are :, #, and =.

Variables are case-sensitive, so `cc` and `CC` refer to different variables. To get the value of a variable, enclose the variable name in `$( )`. As a special case, single-letter variable names can omit the parentheses and simply use `$`*`letter`*. This is why the automatic variables can be written without the parentheses. As a general rule you should use the parenthetical form and avoid single letter variable names.

Variables can also be expanded using curly braces as in `${CC}` and you will often see this form, particularly in older *makefile*s. There is seldom an advantage to using one over the other, so just pick one and stick with it. Some people use curly braces for variable reference and parentheses for function call, similar to the way the shell uses them. Most modern *makefile*s use parentheses and that's what we'll use throughout this book.

Variables representing constants a user might want to customize on the command line or in the environment are written in all uppercase, by convention. Words are separated by underscores. Variables that appear only in the *makefile* are all lowercase with words separated by underscores. Finally, in this book, user-defined functions in variables and macros use lowercase words separated by dashes. Other naming conventions will be explained where they occur. (The following example uses features we haven't discussed yet. I'm using them to illustrate the variable naming conventions, don't be too concerned about the righthand side for now.)

```
# Some simple constants.
CC    := gcc
MKDIR := mkdir -p

# Internal variables.
sources = *.c
objects = $(subst .c,.o,$(sources))

# A function or two.
maybe-make-dir  = $(if $(wildcard $1),,$(MKDIR) $1)
assert-not-null = $(if $1,,$(error Illegal null value.))
```

The value of a variable consists of all the words to the right of the assignment symbol with leading space trimmed. Trailing spaces are not trimmed. This can occasionally cause trouble, for instance, if the trailing whitespace is included in the variable and subsequently used in a command script:

```
LIBRARY = libio.a # LIBRARY has a trailing space.
missing_file:
        touch $(LIBRARY)
        ls -l | grep '$(LIBRARY)'
```

The variable assignment contains a trailing space that is made more apparent by the comment (but a trailing space can also be present without a trailing comment). When this *makefile* is run, we get:

```
$ make
touch libio.a
ls -l | grep 'libio.a '
make: *** [missing_file] Error 1
```

Oops, the `grep` search string also included the trailing space and failed to find the file in `ls`'s output. We'll discuss whitespace issues in more detail later. For now, let's look more closely at variables.

What Variables Are Used For

In general it is a good idea to use variables to represent external programs. This allows users of the *makefile* to more easily adapt the *makefile* to their specific environment.

For instance, there are often several versions of awk on a system: awk, nawk, gawk. By creating a variable, AWK, to hold the name of the awk program you make it easier for other users of your *makefile*. Also, if security is an issue in your environment, a good practice is to access external programs with absolute paths to avoid problems with user's paths. Absolute paths also reduce the likelihood of issues if trojan horse versions of system programs have been installed somewhere in a user's path. Of course, absolute paths also make *makefile*s less portable to other systems. Your own requirements should guide your choice.

Though your first use of variables should be to hold simple constants, they can also store user-defined command sequences such as:*

```
DF  = df
AWK = awk
free-space := $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
```

for reporting on free disk space. Variables are used for both these purposes and more, as we will see.

Variable Types

There are two types of variables in make: simply expanded variables and recursively expanded variables. A *simply expanded* variable (or a simple variable) is defined using the := assignment operator:

```
MAKE_DEPEND := $(CC) -M
```

It is called "simply expanded" because its righthand side is expanded immediately upon reading the line from the *makefile*. Any make variable references in the righthand side are expanded and the resulting text saved as the value of the variable. This behavior is identical to most programming and scripting languages. For instance, the normal expansion of this variable would yield:

```
gcc -M
```

However, if CC above had not yet been set, then the value of the above assignment would be:

```
<space>-M
```

$(CC) is expanded to its value (which contains no characters), and collapses to nothing. It is not an error for a variable to have no definition. In fact, this is extremely useful. Most of the implicit commands include undefined variables that serve as place holders for user customizations. If the user does not customize a variable it

* The df command returns a list of each mounted filesystem and statistics on the filesystem's capacity and usage. With an argument, it prints statistics for the specified filesystem. The first line of the output is a list of column titles. This output is read by awk which examines the second line and ignores all others. Column four of df's output is the remaining free space in blocks.

collapses to nothing. Now notice the leading space. The righthand side is first parsed by make to yield the string $(CC) -M. When the variable reference is collapsed to nothing, make does not rescan the value and trim blanks. The blanks are left intact.

The second type of variable is called a recursively expanded variable. A *recursively expanded* variable (or a recursive variable) is defined using the = assignment operator:

```
MAKE_DEPEND = $(CC) -M
```

It is called "recursively expanded" because its righthand side is simply slurped up by make and stored as the value of the variable without evaluating or expanding it in any way. Instead, the expansion is performed when the variable is *used*. A better term for this variable might be *lazily expanded* variable, since the evaluation is deferred until it is actually used. One surprising effect of this style of expansion is that assignments can be performed "out of order":

```
MAKE_DEPEND = $(CC) -M
...
# Some time later
CC = gcc
```

Here the value of MAKE_DEPEND within a command script is gcc -M even though CC was undefined when MAKE_DEPEND was assigned.

In fact, recursive variables aren't really just a lazy assignment (at least not a normal lazy assignment). Each time the recursive variable is used, its righthand side is re-evaluated. For variables that are defined in terms of simple constants such as MAKE_DEPEND above, this distinction is pointless since all the variables on the righthand side are also simple constants. But imagine if a variable in the righthand side represented the execution of a program, say date. Each time the recursive variable was expanded the date program would be executed and each variable expansion would have a different value (assuming they were executed at least one second apart). At times this is very useful. At other times it is very annoying!

Other Types of Assignment

From previous examples we've seen two types of assignment: = for creating recursive variables and := for creating simple variables. There are two other assignment operators provided by make.

The ?= operator is called the *conditional variable assignment operator*. That's quite a mouth-full so we'll just call it conditional assignment. This operator will perform the requested variable assignment only if the variable does not yet have a value.

```
# Put all generated files in the directory $(PROJECT_DIR)/out.
OUTPUT_DIR ?= $(PROJECT_DIR)/out
```

Here we set the output directory variable, OUTPUT_DIR, only if it hasn't been set earlier. This feature interacts nicely with environment variables. We'll discuss this in the section "Where Variables Come From" later in this chapter.

The other assignment operator, +=, is usually referred to as *append*. As its name suggests, this operator appends text to a variable. This may seem unremarkable, but it is an important feature when recursive variables are used. Specifically, values on the righthand side of the assignment are appended to the variable *without changing the original values in the variable*. "Big deal, isn't that what append always does?" I hear you say. Yes, but hold on, this is a little tricky.

Appending to a simple variable is pretty obvious. The += operator might be implemented like this:

```
simple := $(simple) new stuff
```

Since the value in the simple variable has already undergone expansion, make can expand $(simple), append the text, and finish the assignment. But recursive variables pose a problem. An implementation like the following isn't allowed.

```
recursive = $(recursive) new stuff
```

This is an error because there's no good way for make to handle it. If make stores the current definition of recursive plus new stuff, make can't expand it again at runtime. Furthermore, attempting to expand a recursive variable containing a reference to itself yields an infinite loop.

```
$ make
makefile:2: *** Recursive variable `recursive' references itself (eventually).  Stop.
```

So, += was implemented specifically to allow adding text to a recursive variable and does the Right Thing™. This operator is particularly useful for collecting values into a variable incrementally.

Macros

Variables are fine for storing values as a single line of text, but what if we have a multiline value such as a command script we would like to execute in several places? For instance, the following sequence of commands might be used to create a Java archive (or *jar*) from Java class files:

```
echo Creating $@...
$(RM) $(TMP_JAR_DIR)
$(MKDIR) $(TMP_JAR_DIR)
$(CP) -r $^ $(TMP_JAR_DIR)
cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@ .
$(JAR) -ufm $@ $(MANIFEST)
$(RM) $(TMP_JAR_DIR)
```

At the beginning of long sequences such as this, I like to print a brief message. It can make reading make's output much easier. After the message, we collect our class files into a clean temporary directory. So we delete the temporary *jar* directory in case an old one is left lying about,* then we create a fresh temporary directory. Next we copy our prerequisite files (and all their subdirectories) into the temporary directory. Then we switch to our temporary directory and create the jar with the target filename. We add the manifest file to the jar and finally clean up. Clearly, we do not want to duplicate this sequence of commands in our *makefile* since that would be a maintenance problem in the future. We might consider packing all these commands into a recursive variable, but that is ugly to maintain and difficult to read when make echoes the command line (the whole sequence is echoed as one enormous line of text).

Instead, we can use a GNU make "canned sequence" as created by the define directive. The term "canned sequence" is a bit awkward, so we'll call this a *macro*. A macro is just another way of defining a variable in make, and one that can contain embedded newlines! The GNU make manual seems to use the words *variable* and *macro* interchangeably. In this book, we'll use the word *macro* specifically to mean variables defined using the define directive and *variable* only when assignment is used.

```
define create-jar
 @echo Creating $@...
 $(RM) $(TMP_JAR_DIR)
 $(MKDIR) $(TMP_JAR_DIR)
 $(CP) -r $^ $(TMP_JAR_DIR)
 cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@ .
 $(JAR) -ufm $@ $(MANIFEST)
 $(RM) $(TMP_JAR_DIR)
endef
```

The define directive is followed by the variable name and a newline. The body of the variable includes all the text up to the endef keyword, which must appear on a line by itself. A variable created with define is expanded pretty much like any other variable, except that when it is used in the context of a command script, each line of the macro has a tab prepended to the line. An example use is:

```
$(UI_JAR): $(UI_CLASSES)
        $(create-jar)
```

Notice we've added an @ character in front of our echo command. Command lines prefixed with an @ character are not echoed by make when the command is executed. When we run make, therefore, it doesn't print the echo command, just the output of that command. If the @ prefix is used within a macro, the prefix character applies to the individual lines on which it is used. However, if the prefix character is used on the macro reference, the entire macro body is hidden:

```
$(UI_JAR): $(UI_CLASSES)
        @$(create-jar)
```

* For best effect here, the RM variable should be defined to hold rm -rf. In fact, its default value is rm -f, safer but not quite as useful. Further, MKDIR should be defined as mkdir -p, and so on.

This displays only:

```
$ make
Creating ui.jar...
```

The use of `@` is covered in more detail in the section "Command Modifiers" in Chapter 5.

When Variables Are Expanded

In the previous sections, we began to get a taste of some of the subtleties of variable expansion. Results depend a lot on what was previously defined, and where. You could easily get results you don't want, even if `make` fails to find any error. So what are the rules for expanding variables? How does this really work?

When `make` runs, it performs its job in two phases. In the first phase, `make` reads the *makefile* and any included *makefile*s. At this time, variables and rules are loaded into `make`'s internal database and the dependency graph is created. In the second phase, `make` analyzes the dependency graph and determines the targets that need to be updated, then executes command scripts to perform the required updates.

When a recursive variable or `define` directive is processed by `make`, the lines in the variable or body of the macro are stored, including the newlines without being expanded. The very last newline of a macro definition is not stored as part of the macro. Otherwise, when the macro was expanded an extra newline would be read by `make`.

When a macro is expanded, the expanded text is then immediately scanned for further macro or variable references and those are expanded and so on, recursively. If the macro is expanded in the context of an action, each line of the macro is inserted with a leading tab character.

To summarize, here are the rules for when elements of a *makefile* are expanded:

- For variable assignments, the lefthand side of the assignment is always expanded immediately when `make` reads the line during its first phase.
- The righthand side of `=` and `?=` are deferred until they are used in the second phase.
- The righthand side of `:=` is expanded immediately.
- The righthand side of `+=` is expanded immediately if the lefthand side was originally defined as a simple variable. Otherwise, its evaluation is deferred.
- For macro definitions (those using `define`), the macro variable name is immediately expanded and the body of the macro is deferred until used.
- For rules, the targets and prerequisites are always immediately expanded while the commands are always deferred.

Table 3-1 summarizes what occurs when variables are expanded.

Table 3-1. Rules for immediate and deferred expansion

Definition	Expansion of a	Expansion of b
`a = b`	Immediate	Deferred
`a ?= b`	Immediate	Deferred
`a := b`	Immediate	Immediate
`a += b`	Immediate	Deferred or immediate
`define a` `b...` `b...` `b...` `endef`	Immediate	Deferred

As a general rule, always define variables and macros before they are used. In particular, it is required that a variable used in a target or prerequisite be defined before its use.

An example will make all this clearer. Suppose we reimplement our `free-space` macro. We'll go over the example a piece at a time, then put them all together at the end.

```
BIN    := /usr/bin
PRINTF := $(BIN)/printf
DF     := $(BIN)/df
AWK    := $(BIN)/awk
```

We define three variables to hold the names of the programs we use in our macro. To avoid code duplication we factor out the *bin* directory into a fourth variable. The four variable definitions are read and their righthand sides are immediately expanded because they are simple variables. Because `BIN` is defined before the others, its value can be plugged into their values.

Next, we define the `free-space` macro.

```
define free-space
  $(PRINTF) "Free disk space "
  $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
endef
```

The `define` directive is followed by a variable name that is immediately expanded. In this case, no expansion is necessary. The body of the macro is read and stored unexpanded.

Finally, we use our macro in a rule.

```
OUTPUT_DIR := /tmp

$(OUTPUT_DIR)/very_big_file:
        $(free-space)
```

When *$(OUTPUT_DIR)/very_big_file* is read, any variables used in the targets and prerequisites are immediately expanded. Here, `$(OUTPUT_DIR)` is expanded to */tmp* to form the */tmp/very_big_file* target. Next, the command script for this target is read. Command lines are recognized by the leading tab character and are read and stored, but not expanded.

Here is the entire example *makefile*. The order of elements in the file has been scrambled intentionally to illustrate make's evaluation algorithm.

```
OUTPUT_DIR := /tmp

$(OUTPUT_DIR)/very_big_file:
        $(free-space)

define free-space
  $(PRINTF) "Free disk space "
  $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
endef

BIN    := /usr/bin
PRINTF := $(BIN)/printf
DF     := $(BIN)/df
AWK    := $(BIN)/awk
```

Notice that although the order of lines in the *makefile* seems backward, it executes just fine. This is one of the surprising effects of recursive variables. It can be immensely useful and confusing at the same time. The reason this *makefile* works is that expansion of the command script and the body of the macro are deferred until they are actually used. Therefore, the relative order in which they occur is immaterial to the execution of the *makefile*.

In the second phase of processing, after the *makefile* is read, make identifies the targets, performs dependency analysis, and executes the actions for each rule. Here the only target, `$(OUTPUT_DIR)/very_big_file`, has no prerequisites, so make will simply execute the actions (assuming the file doesn't exist). The command is `$(free-space)`. So make expands this as if the programmer had written:

```
/tmp/very_big_file:
        /usr/bin/printf "Free disk space "
        /usr/bin/df . | /usr/bin/awk 'NR == 2 { print $$4 }'
```

Once all variables are expanded, it begins executing commands one at a time.

Let's look at the two parts of the makefile where the order is important. As explained earlier, the target *$(OUTPUT_DIR)/very_big_file* is expanded immediately. If the definition of the variable `OUTPUT_DIR` had followed the rule, the expansion of the target would have yielded */very_big_file*. Probably not what the user wanted. Similarly, if the definition of `BIN` had been moved after `AWK`, those three variables would have expanded to */printf*, */df*, and */awk* because the use of `:=` causes immediate evaluation of the righthand side of the assignment. However, in this case, we could avoid the problem for `PRINTF`, `DF`, and `AWK` by changing `:=` to `=`, making them recursive variables.

One last detail. Notice that changing the definitions of OUTPUT_DIR and BIN to recursive variables would not change the effect of the previous ordering problems. The important issue is that when *$(OUTPUT_DIR)/very_big_file* and the righthand sides of PRINTF, DF, and AWK are expanded, their expansion happens immediately, so the variables they refer to must be already defined.

Target- and Pattern-Specific Variables

Variables usually have only one value during the execution of a *makefile*. This is ensured by the two-phase nature of *makefile* processing. In phase one, the *makefile* is read, variables are assigned and expanded, and the dependency graph is built. In phase two, the dependency graph is analyzed and traversed. So when command scripts are being executed, all variable processing has already completed. But suppose we wanted to redefine a variable for just a single rule or pattern.

In this example, the particular file we are compiling needs an extra command-line option, -DUSE_NEW_MALLOC=1, that should not be provided to other compiles:

```
gui.o: gui.h
        $(COMPILE.c) -DUSE_NEW_MALLOC=1 $(OUTPUT_OPTION) $<
```

Here, we've solved the problem by duplicating the compilation command script and adding the new required option. This approach is unsatisfactory in several respects. First, we are duplicating code. If the rule ever changes or if we choose to replace the built-in rule with a custom pattern rule, this code would need to be updated and we might forget. Second, if many files require special treatment, the task of pasting in this code will quickly become very tedious and error-prone (imagine a hundred files like this).

To address this issue and others, make provides *target-specific variables*. These are variable definitions attached to a target that are valid only during the processing of that target and any of its prerequisites. We can rewrite our previous example using this feature like this:

```
gui.o: CPPFLAGS += -DUSE_NEW_MALLOC=1
gui.o: gui.h
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

The variable CPPFLAGS is built in to the default C compilation rule and is meant to contain options for the C preprocessor. By using the += form of assignment, we append our new option to any existing value already present. Now the compile command script can be removed entirely:

```
gui.o: CPPFLAGS += -DUSE_NEW_MALLOC=1
gui.o: gui.h
```

While the *gui.o* target is being processed, the value of CPPFLAGS will contain -DUSE_NEW_MALLOC=1 in addition to its original contents. When the *gui.o* target is finished, CPPFLAGS will revert to its original value.

The general syntax for target-specific variables is:

```
target...: variable = value
target...: variable := value
target...: variable += value
target...: variable ?= value
```

As you can see, all the various forms of assignment are valid for a target-specific variable. The variable does not need to exist before the assignment.

Furthermore, the variable assignment is not actually performed until the processing of the target begins. So the righthand side of the assignment can itself be a value set in another target-specific variable. The variable is valid during the processing of all prerequisites as well.

Where Variables Come From

So far, most variables have been defined explicitly in our own *makefile*s, but variables can have a more complex ancestry. For instance, we have seen that variables can be defined on the make command line. In fact, make variables can come from these sources:

File

Of course, variables can be defined in the *makefile* or a file included by the *makefile* (we'll cover the include directive shortly).

Command line

Variables can be defined or redefined directly from the make command line:

```
$ make CFLAGS=-g CPPFLAGS='-DBSD -DDEBUG'
```

A command-line argument containing an = is a variable assignment. Each variable assignment on the command line must be a single-shell argument. If the value of the variable (or heaven forbid, the variable itself) contains spaces, the argument must be surrounded by quotes or the spaces must be escaped.

An assignment of a variable on the command line overrides any value from the environment and any assignment in the *makefile*. Command-line assignments can set either simple or recursive variables by using := or =, respectively. It is possible using the override directive to allow a *makefile* assignment to be used instead of a command-line assignment.

```
# Use big-endian objects or the program crashes!
override LDFLAGS = -EB
```

Of course, you should ignore a user's explicit assignment request only under the most urgent circumstances (unless you just want to irritate your users).

Environment

All the variables from your environment are automatically defined as make variables when make starts. These variables have very low precedence, so assignments within the *makefile* or command-line arguments will override the value of an environment variable. You can cause environment variables to override *makefile* variables using the --environment-overrides (or -e) command-line option.

When make is invoked recursively, some variables from the parent make are passed through the environment to the child make. By default, only those variables that originally came from the environment are exported to the child's environment, but any variable can be exported to the environment by using the export directive:

```
export CLASSPATH := $(HOME)/classes:$(PROJECT)/classes
SHELLOPTS = -x
export SHELLOPTS
```

You can cause all variables to be exported with:

```
export
```

Note that make will export even those variables whose names contain invalid shell variable characters. For example:

```
export valid-variable-in-make = Neat!
show-vars:
        env | grep '^valid-'
        valid_variable_in_shell=Great
        invalid-variable-in-shell=Sorry


$ make
env | grep '^valid-'
valid-variable-in-make=Neat!
valid_variable_in_shell=Great
invalid-variable-in-shell=Sorry
/bin/sh: line 1: invalid-variable-in-shell=Sorry: command not found
make: *** [show-vars] Error 127
```

An "invalid" shell variable was created by exporting valid-variable-in-make. This variable is not accessible through normal shell syntax, only through trickery such as running grep over the environment. Nevertheless, this variable is inherited by any sub-make where it is valid and accessible. We will cover use of "recursive" make in Part II.

You can also prevent an environment variable from being exported to the subprocess:

```
unexport DISPLAY
```

The export and unexport directives work the same way their counterparts in sh work.

The conditional assignment operator interacts very nicely with environment variables. Suppose you have a default output directory set in your *makefile*, but you want users to be able to override the default easily. Conditional assignment is perfect for this situation:

```
# Assume the output directory $(PROJECT_DIR)/out.
OUTPUT_DIR ?= $(PROJECT_DIR)/out
```

Here the assignment is performed only if OUTPUT_DIR has never been set. We can get nearly the same effect more verbosely with:

```
ifndef OUTPUT_DIR
  # Assume the output directory $(PROJECT_DIR)/out.
  OUTPUT_DIR = $(PROJECT_DIR)/out
endif
```

The difference is that the conditional assignment operator will skip the assignment if the variable has been set in any way, even to the empty value, while the `ifdef` and `ifndef` operators test for a nonempty value. Thus, `OUTPUT_DIR=` is considered set by the conditional operator but not defined by `ifdef`.

It is important to note that excessive use of environment variables makes your *makefile*s much less portable, since other users are not likely to have the same set of environment variables. In fact, I rarely use this feature for precisely that reason.

Automatic

Finally, `make` creates automatic variables immediately before executing the command script of a rule.

Traditionally, environment variables are used to help manage the differences between developer machines. For instance, it is common to create a development environment (source code, compiled output tree, and tools) based on environment variables referenced in the *makefile*. The *makefile* would refer to one environment variable for the root of each tree. If the source file tree is referenced from a variable `PROJECT_SRC`, binary output files from `PROJECT_BIN`, and libraries from `PROJECT_LIB`, then developers are free to place these trees wherever is appropriate.

A potential problem with this approach (and with the use of environment variables in general) occurs when these "root" variables are not set. One solution is to provide default values in the *makefile* using the `$?` form of assignment:

```
PROJECT_SRC ?= /dev/$(USER)/src
PROJECT_BIN ?= $(patsubst %/src,%/bin,$(PROJECT_SRC))
PROJECT_LIB ?= /net/server/project/lib
```

By using these variables to access project components, you can create a development environment that is adaptable to varying machine layouts. (We will see more comprehensive examples of this in Part II.) Beware of overreliance on environment variables, however. Generally, a *makefile* should be able to run with a minimum of support from the developer's environment so be sure to provide reasonable defaults and check for the existence of critical components.

Conditional and include Processing

Parts of a *makefile* can be omitted or selected while the *makefile* is being read using *conditional processing* directives. The condition that controls the selection can have several forms such as "is defined" or "is equal to." For example:

```
# COMSPEC is defined only on Windows.
ifdef COMSPEC
  PATH_SEP := ;
  EXE_EXT  := .exe
else
  PATH_SEP := :
  EXE_EXT  :=
endif
```

This selects the first branch of the conditional if the variable `COMSPEC` is defined.

The basic syntax of the conditional directive is:

```
if-condition
  text if the condition is true
endif
```

or:

```
if-condition
  text if the condition is true
else
  text if the condition is false
endif
```

The *`if-condition`* can be one of:

```
ifdef  variable-name
ifndef variable-name
ifeq  test
ifneq test
```

The *`variable-name`* should not be surrounded by `$()` for the `ifdef`/`ifndef` test. Finally, the *`test`* can be expressed as either of:

```
"a" "b"
(a,b)
```

in which single or double quotes can be used interchangeably (but the quotes you use must match).

The conditional processing directives can be used within macro definitions and command scripts as well as at the top level of *makefile*s:

```
libGui.a: $(gui_objects)
        $(AR) $(ARFLAGS) $@ $<
    ifdef RANLIB
        $(RANLIB) $@
    endif
```

I like to indent my conditionals, but careless indentation can lead to errors. In the preceding lines, the conditional directives are indented four spaces while the enclosed commands have a leading tab. If the enclosed commands didn't begin with a tab, they would not be recognized as commands by `make`. If the conditional directives had a leading tab, they would be misidentified as commands and passed to the subshell.

The `ifeq` and `ifneq` conditionals test if their arguments are equal or not equal. Whitespace in conditional processing can be tricky to handle. For instance, when using the parenthesis form of the test, whitespace after the comma is ignored, but all other whitespace is significant:

```
ifeq (a, a)
  # These are equal
endif

ifeq ( b, b )
  # These are not equal - ' b' != 'b '
endif
```

Personally, I stick with the quoted forms of equality:

```
ifeq "a" "a"
  # These are equal
endif

ifeq 'b' 'b'
  # So are these
endif
```

Even so, it often occurs that a variable expansion contains unexpected whitespace. This can cause problems since the comparison includes all characters. To create more robust *makefile*s, use the `strip` function:

```
ifeq "$(strip $(OPTIONS)) "-d"
  COMPILATION_FLAGS += -DDEBUG
endif
```

The include Directive

We first saw the `include` directive in Chapter 2, in the section "Automatic Dependency Generation." Now let's go over it in more detail.

A *makefile* can include other files. This is most commonly done to place common `make` definitions in a `make` header file or to include automatically generated dependency information. The `include` directive is used like this:

```
include definitions.mk
```

The directive can be given any number of files and shell wildcards and `make` variables are also allowed.

include and Dependencies

When make encounters an include directive, it expands the wildcards and variable references, then tries to read the include file. If the file exists, we continue normally. If the file does not exist, however, make reports the problem and continues reading the rest of the *makefile*. When all reading is complete, make looks in the rules database for any rule to update the include files. If a match is found, make follows the normal process for updating a target. If any of the include files is updated by a rule, make then clears its internal database and rereads the entire *makefile*. If, after completing the process of reading, updating, and rereading, there are still include directives that have failed due to missing files, make terminates with an error status.

We can see this process in action with the following two-file example. We use the warning built-in function to print a simple message from make. (This and other functions are covered in detail in Chapter 4.) Here is the *makefile*:

```
# Simple makefile including a generated file.
include foo.mk
$(warning Finished include)

foo.mk: bar.mk
        m4 --define=FILENAME=$@ bar.mk > $@
```

and here is *bar.mk*, the source for the included file:

```
# bar.mk - Report when I am being read.
$(warning Reading FILENAME)
```

When run, we see:

```
$ make
Makefile:2: foo.mk: No such file or directory
Makefile:3: Finished include
m4 --define=FILENAME=foo.mk bar.mk > foo.mk
foo.mk:2: Reading foo.mk
Makefile:3: Finished include
make: `foo.mk' is up to date.
```

The first line shows that make cannot find the include file, but the second line shows that make keeps reading and executing the *makefile*. After completing the read, make discovers a rule to create the include file, *foo.mk*, and it does so. Then make starts the whole process again, this time without encountering any difficulty reading the include file.

Now is a good time to mention that make will also treat the *makefile* itself as a possible target. After the entire *makefile* has been read, make will look for a rule to remake the currently executing *makefile*. If it finds one, make will process the rule, then check if the *makefile* has been updated. If so, make will clear its internal state and reread the *makefile*, performing the whole analysis over again. Here is a silly example of an infinite loop based on this behavior:

```
.PHONY: dummy
makefile: dummy
        touch $@
```

When make executes this *makefile*, it sees that the *makefile* is out of date (because the .PHONY target, *dummy*, is out of date) so it executes the touch command, which updates the timestamp of the *makefile*. Then make rereads the file and discovers that the *makefile* is out of date.... Well, you get the idea.

Where does make look for included files? Clearly, if the argument to include is an absolute file reference, make reads that file. If the file reference is relative, make first looks in its current working directory. If make cannot find the file, it then proceeds to search through any directories you have specified on the command line using the --include-dir (or -I) option. After that, make searches a compiled search path similar to: */usr/local/include*, */usr/gnu/include*, */usr/include*. There may be slight variations of this path due to the way make was compiled.

If make cannot find the include file and it cannot create it using a rule, make exits with an error. If you want make to ignore include files it cannot load, add a leading dash to the include directive:

```
-include i-may-not-exist.mk
```

For compatibility with other makes, the word sinclude is an alias for -include.

Standard make Variables

In addition to automatic variables, make maintains variables revealing bits and pieces of its own state as well as variables for customizing built-in rules:

MAKE_VERSION
: This is the version number of GNU make. At the time of this writing, its value is 3.80, and the value in the CVS repository is 3.81rc1.

  The previous version of make, 3.79.1, did not support the eval and value functions (among other changes) and it is still very common. So when I write *makefile*s that require these features, I use this variable to test the version of make I'm running. We'll see an example of that in the section "Flow Control" in Chapter 4.

CURDIR
: This variable contains the current working directory (cwd) of the executing make process. This will be the same directory the make program was executed from (and it will be the same as the shell variable PWD), unless the --directory (-C) option is used. The --directory option instructs make to change to a different directory before searching for any *makefile*. The complete form of the option is

--directory=*directory-name* or -C *directory-name*. If --directory is used, CURDIR will contain the directory argument to --include-dir.

I typically invoke make from emacs while coding. For instance, my current project is in Java and uses a single *makefile* in a top-level directory (not necessarily the directory containing the code). In this case, using the --directory option allows me to invoke make from any directory in the source tree and still access the *makefile*. Within the *makefile*, all paths are relative to the *makefile* directory. Absolute paths are occasionally required and these are accessed using CURDIR.

MAKEFILE_LIST

This variable contains a list of each file make has read including the default *makefile* and *makefile*s specified on the command line or through include directives. Just before each file is read, the name is appended to the MAKEFILE_LIST variable. So a *makefile* can always determine its own name by examining the last word of the list.

MAKECMDGOALS

The MAKECMDGOALS variable contains a list of all the targets specified on the command line for the current execution of make. It does not include command-line options or variable assignments. For instance:

```
$ make -f- FOO=bar -k goal <<< 'goal:;# $(MAKECMDGOALS)'
# goal
```

The example uses the "trick" of telling make to read the *makefile* from the *stdin* with the -f- (or --file) option. The *stdin* is redirected from a command-line string using bash's *here string*, "<<<", syntax.[*] The *makefile* itself consists of the default goal goal, while the command script is given on the same line by separating the target from the command with a semicolon. The command script contains the single line:

```
# $(MAKECMDGOALS)
```

MAKECMDGOALS is typically used when a target requires special handling. The primary example is the "clean" target. When invoking "clean," make should not perform the usual dependency file generation triggered by include (discussed in the section "Automatic Dependency Generation" in Chapter 2). To prevent this use ifneq and MAKECMDGOALS:

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(subst .xml,.d,$(xml_src))
endif
```

[*] For those of you who want to run this type of example in another shell, use:

```
$ echo 'goal:;# $(MAKECMDGOALS)' | make -f- FOO=bar -k goal
```

`.VARIABLES`

This contains a list of the names of all the variables defined in *makefile*s read so far, with the exception of target-specific variables. The variable is read-only and any assignment to it is ignored.

```
list:
        @echo "$(.VARIABLES)" | tr ' ' '\015' | grep MAKEF
$ make
MAKEFLAGS
MAKEFILE_LIST
MAKEFILES
```

As you've seen, variables are also used to customize the implicit rules built in to `make`. The rules for C/C++ are typical of the form these variables take for all programming languages. Figure 3-1 shows the variables controlling translation from one file type to another.

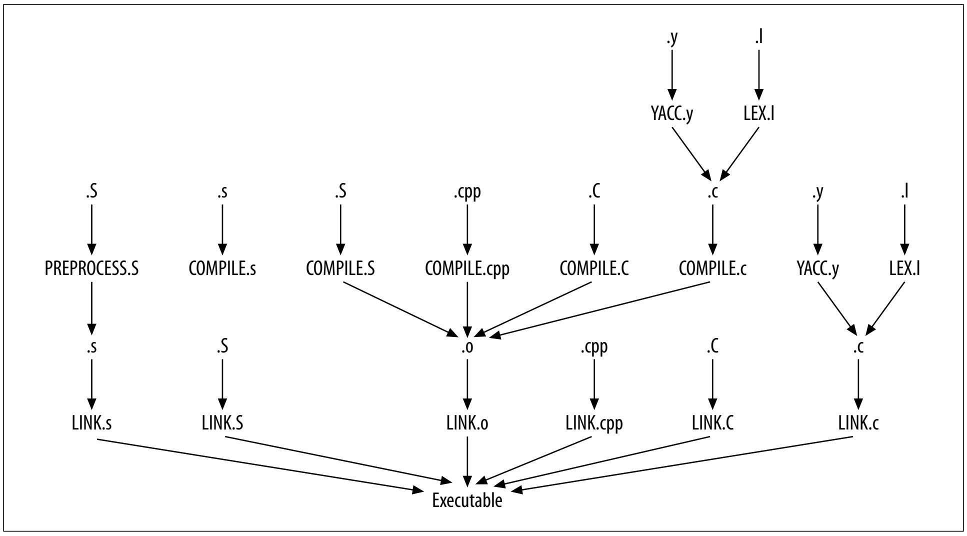

Figure 3-1. Variables for C/C++ compilation

The variables have the basic form: *`ACTION.suffix`*. The *`ACTION`* is `COMPILE` for creating an object file, `LINK` for creating an executable, or the "special" operations `PREPROCESS`, `YACC`, `LEX` for running the C preprocessor, yacc, or `lex`, respectively. The *`suffix`* indicates the source file type.

The standard "path" through these variables for, say, C++, uses two rules. First, compile C++ source files to object files. Then link the object files into an executable.

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

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

The first rule uses these variable definitions:

```
COMPILE.C     = $(COMPILE.cc)
COMPILE.cc    = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
CXX           = g++
OUTPUT_OPTION = -o $@
```

GNU make supports either of the suffixes *.C* or *.cc* for denoting C++ source files. The CXX variable indicates the C++ compiler to use and defaults to g++. The variables CXXFLAGS, CPPFLAGS, and TARGET_ARCH have no default value. They are intended for use by end-users to customize the build process. The three variables hold the C++ compiler flags, C preprocessor flags, and architecture-specific compilation options, respectively. The OUTPUT_OPTION contains the output file option.

The linking rule is a bit simpler:

```
LINK.o = $(CC) $(LDFLAGS) $(TARGET_ARCH)
CC     = gcc
```

This rule uses the C compiler to combine object files into an executable. The default for the C compiler is gcc. LDFLAGS and TARGET_ARCH have no default value. The LDFLAGS variable holds options for linking such as -L flags. The LOADLIBES and LDLIBS variables contain lists of libraries to link against. Two variables are included mostly for portability.

This was a quick tour through the make variables. There are more, but this gives you the flavor of how variables are integrated with rules. Another group of variables deals with T_EX and has its own set of rules. Recursive make is another feature supported by variables. We'll discuss this topic in Chapter 6.

CHAPTER 4
Functions

GNU make supports both built-in and user-defined functions. A function invocation looks much like a variable reference, but includes one or more parameters separated by commas. Most built-in functions expand to some value that is then assigned to a variable or passed to a subshell. A user-defined function is stored in a variable or macro and expects one or more parameters to be passed by the caller.

User-Defined Functions

Storing command sequences in variables opens the door to a wide range of applications. For instance, here's a nice little macro to kill a process:*

```
AWK  := awk
KILL := kill

# $(kill-acroread)
define kill-acroread
  @ ps -W |                                   \
  $(AWK) 'BEGIN      { FIELDWIDTHS = "9 47 100" }     \
          /AcroRd32/ {                          \
                       print "Killing " $$3;         \
                       system( "$(KILL) -f " $$1 )    \
                     }'
endef
```

* "Why would you want to do this in a *makefile*?" you ask. Well, on Windows, opening a file locks it against writing by other processes. While I was writing this book, the *PDF* file would often be locked by the Acrobat Reader and prevent my *makefile* from updating the *PDF*. So I added this command to several targets to terminate Acrobat Reader before attempting to update the locked file.

(This macro was written explicitly to use the Cygwin tools,* so the program name we search for and the options to ps and kill are not standard Unix.) To kill a process we pipe the output of ps to awk. The awk script looks for the Acrobat Reader by its Windows program name and kills the process if it is running. We use the FIELDWIDTHS feature to treat the program name and all its arguments as a single field. This correctly prints the complete program name and arguments even when it contains embedded blanks. Field references in awk are written as $1, $2, etc. These would be treated as make variables if we did not quote them in some way. We can tell make to pass the $*n* reference to awk instead of expanding it itself by escaping the dollar sign in $*n* with an additional dollar sign, $$*n*. make will see the double dollar sign, collapse it to a single dollar sign and pass it to the subshell.

Nice macro. And the define directive saves us from duplicating the code if we want to use it often. But it isn't perfect. What if we want to kill processes other than the Acrobat Reader? Do we have to define another macro and duplicate the script? No!

Variables and macros can be passed arguments so that each expansion can be different. The parameters of the macro are referenced within the body of the macro definition with $1, $2, etc. To parameterize our kill-acroread function, we only need to add a search parameter:

```
AWK         := awk
KILL        := kill
KILL_FLAGS  := -f
PS          := ps
PS_FLAGS    := -W
PS_FIELDS   := "9 47 100"

# $(call kill-program,awk-pattern)
define kill-program
  @ $(PS) $(PS_FLAGS) |                               \
  $(AWK) 'BEGIN { FIELDWIDTHS = $(PS_FIELDS) }        \
          /$1/  {                                     \
                  print "Killing " $$3;               \
                  system( "$(KILL) $(KILL_FLAGS) " $$1 ) \
                }'
endef
```

We've replaced the awk search pattern, /AcroRd32/, with a parameter reference, $1. Note the subtle distinction between the macro parameter, $1, and the awk field reference, $$1. It is very important to remember which program is the intended recipient for a variable reference. As long as we're improving the function, we have also

* The *Cygwin* tools are a port of many of the standard GNU and Linux programs to Windows. It includes the compiler suite, X11R6, ssh, and even inetd. The port relies on a compatibility library that implements Unix system calls in terms of Win32 API functions. It is an incredible feat of engineering and I highly recommend it. Download it from *http://www.cygwin.com*.

renamed it appropriately and replaced the Cygwin-specific, hardcoded values with variables. Now we have a reasonably portable macro for terminating processes.

So let's see it in action:

```
FOP        := org.apache.fop.apps.Fop
FOP_FLAGS  := -q
FOP_OUTPUT := > /dev/null
%.pdf: %.fo
        $(call kill-program,AcroRd32)
        $(JAVA) $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)
```

This pattern rule kills the *Acrobat* process, if one is running, and then converts an *fo* (Formatting Objects) file into a *pdf* file by invoking the Fop processor (*http://xml.apache.org/fop*). The syntax for expanding a variable or macro is:

```
$(call macro-name[, param1...])
```

`call` is a built-in `make` function that expands its first argument and replaces occurrences of `$1`, `$2`, etc., with the remaining arguments it is given. (In fact, it doesn't really "call" its macro argument at all in the sense of transfer of control, rather it performs a special kind of macro expansion.) The `macro-name` is the name of any macro or variable (remember that macros are just variables where embedded newlines are allowed). The macro or variable value doesn't even have to contain a `$n` reference, but then there isn't much point in using `call` at all. Arguments to the macro following `macro-name` are separated by commas.

Notice that the first argument to `call` is an unexpanded variable name (that is, it does not begin with a dollar sign). That is fairly unusual. Only one other built-in function, `origin`, accepts unexpanded variables. If you enclose the first argument to `call` in a dollar sign and parentheses, that argument is expanded as a variable and its value is passed to `call`.

There is very little in the way of argument checking with `call`. Any number of arguments can be given to `call`. If a macro references a parameter `$n` and there is no corresponding argument in the `call` instance, the variable collapses to nothing. If there are more arguments in the `call` instance than there are `$n` references, the extra arguments are never expanded in the macro.

If you invoke one macro from another, you should be aware of a somewhat strange behavior in `make` 3.80. The `call` function defines the arguments as normal `make` variables for the duration of the expansion. So if one macro invokes another, it is possible that the parent's arguments will be visible in the child macro's expansion:

```
define parent
  echo "parent has two parameters: $1, $2"
  $(call child,$1)
endef

define child
  echo "child has one parameter: $1"
  echo "but child can also see parent's second parameter: $2!"
endef

scoping_issue:
        @$(call parent,one,two)
```

When run, we see that the macro implementation has a scoping issue.

```
$ make
parent has two parameters: one, two
child has one parameter: one
but child can also see parent's second parameter: two!
```

This has been resolved in 3.81 so that `$2` in `child` collapses to nothing.

We'll spend a lot more time with user-defined functions throughout the rest of the book, but we need more background before we can get into the really fun stuff!

Built-in Functions

Once you start down the road of using `make` variables for more than just simple constants you'll find that you want to manipulate the variables and their contents in more and more complex ways. Well, you can. GNU `make` has a couple dozen built-in functions for working with variables and their contents. The functions fall into several broad categories: string manipulation, filename manipulation, flow control, user-defined functions, and some (important) miscellaneous functions.

But first, a little more about function syntax. All functions have the form:

```
$(function-name arg1[, argn])
```

The `$(` is followed by built-in function name and then followed by the arguments to the function. Leading whitespace is trimmed from the first argument, but all subsequent arguments include any leading (and, of course, embedded and following) whitespace.

Function arguments are separated by commas, so a function with one argument uses no commas, a function with two arguments uses one comma, etc. Many functions accept a single argument, treating it as a list of space-separated words. For these functions, the whitespace between words is treated as a single-word separator and is otherwise ignored.

I like whitespace. It makes the code more readable and easier to maintain. So I'll be using whitespace wherever I can "get away" with it. Sometimes, however, the whitespace in an argument list or variable definition can interfere with the proper functioning of the code. When this happens, you have little choice but to remove the problematic whitespace. We already saw one example earlier in the chapter where trailing whitespace was accidentally inserted into the search pattern of a `grep` command. As we proceed with more examples, we'll point out where whitespace issues arise.

Many `make` functions accept a *pattern* as an argument. This pattern uses the same syntax as the patterns used in pattern rules (see the section "Pattern Rules" in Chapter 2). A pattern contains a single `%` with leading or trailing characters (or both). The `%` character represents zero or more characters of any kind. To match a target string, the pattern must match the entire string, not just a subset of characters within the string. We'll illustrate this with an example shortly. The `%` character is optional in a pattern and is commonly omitted when appropriate.

String Functions

Most of `make`'s built-in functions manipulate text in one form or another, but certain functions are particularly strong at string manipulation, and these will be discussed here.

A common string operation in `make` is to select a set of files from a list. This is what `grep` is typically used for in shell scripts. In `make` we have the `filter`, `filter-out`, and `findstring` functions.

`$(filter pattern...,text)`

The `filter` function treats `text` as a sequence of space separated words and returns a list of those words matching `pattern`. For instance, to build an archive of user-interface code, we might want to select only the object files in the *ui* subdirectory. In the following example, we extract the filenames starting with *ui/* and ending in *.o* from a list of filenames. The `%` character matches any number of characters in between:

```
$(ui_library): $(filter ui/%.o,$(objects))
        $(AR) $(ARFLAGS) $@ $^
```

It is also possible for `filter` to accept multiple patterns, separated by spaces. As noted above, the pattern must match an entire word for the word to be included in the output list. So, for instance:

```
words := he the hen other the%
get-the:
        @echo he matches: $(filter he, $(words))
        @echo %he matches: $(filter %he, $(words))
        @echo he% matches: $(filter he%, $(words))
        @echo %he% matches: $(filter %he%, $(words))
```

When executed the *makefile* generates the output:

```
$ make
he matches: he
%he matches: he the
he% matches: he hen
%he% matches: the%
```

As you can see, the first pattern matches only the word `he`, because the pattern must match the entire word, not just a part of it. The other patterns match `he` plus words that contain `he` in the right position.

A pattern can contain only one `%`. If additional `%` characters are included in the pattern, all but the first are treated as literal characters.

It may seem odd that `filter` cannot match substrings within words or accept more than one wildcard character. You will find times when this functionality is sorely missed. However, you can implement something similar using looping and conditional testing. We'll show you how later.

`$(filter-out pattern...,text)`

The `filter-out` function does the opposite of `filter`, selecting every word that does not match the pattern. Here we select all files that are not C headers.

```
all_source := count_words.c counter.c lexer.l counter.h lexer.h
to_compile := $(filter-out %.h, $(all_source))
```

`$(findstring string,text)`

This function looks for `string` in `text`. If the string is found, the function returns `string`; otherwise, it returns nothing.

At first, this function might seem like the substring searching `grep` function we thought `filter` might be, but not so. First, and most important, this function returns just the search string, not the word it finds that contains the search string. Second, the search string cannot contain wildcard characters (putting it another way, `%` characters in the search string are matched literally).

This function is mostly useful in conjunction with the `if` function discussed later. There is, however, one situation where I've found `findstring` to be useful in its own right.

Suppose you have several trees with parallel structure such as reference source, sandbox source, debugging binary, and optimized binary. You'd like to be able to find out which tree you are in from your current directory (without the current relative path from the root). Here is some skeleton code to determine this:

```
find-tree:
        # PWD = $(PWD)
        # $(findstring /test/book/admin,$(PWD))
        # $(findstring /test/book/bin,$(PWD))
        # $(findstring /test/book/dblite_0.5,$(PWD))
        # $(findstring /test/book/examples,$(PWD))
        # $(findstring /test/book/out,$(PWD))
        # $(findstring /test/book/text,$(PWD))
```

(Each line begins with a tab and a shell comment character so each is "executed" in its own subshell just like other commands. The Bourne Again Shell, `bash`, and many other Bourne-like shells simply ignore these lines. This is a more convenient way to print out the expansion of simple `make` constructs than typing `@echo`. You can achieve almost the same effect using the more portable `:` shell operator, but the `:` operator performs redirections. Thus, a command line containing `> word` creates the file *word* as a side effect.) When run, it produces:

```
$ make
# PWD = /test/book/out/ch03-findstring-1
```

```
#
#
#
#
# /test/book/out
#
```

As you can see, each test against `$(PWD)` returns null until we test our parent directory. Then the parent directory itself is returned. As shown, the code is merely as a demonstration of `findstring`. This can be used to write a function returning the current tree's root directory.

There are two search and replace functions:

`$(subst search-string,replace-string,text)`

This is a simple, nonwildcard, search and replace. One of its most common uses is to replace one suffix with another in a list of filenames:

```
sources := count_words.c counter.c lexer.c
objects := $(subst .c,.o,$(sources))
```

This replaces all occurrences of "*.c*" with "*.o*" anywhere in `$(sources)`, or, more generally, all occurrences of the search string with the replacement string.

This example is a commonly found illustration of where spaces are significant in function call arguments. Note that there are no spaces after the commas. If we had instead written:

```
sources := count_words.c counter.c lexer.c
objects := $(subst .c, .o, $(sources))
```

(notice the space after each comma), the value of `$(objects)` would have been:

```
count_words .o counter .o lexer .o
```

Not at all what we want. The problem is that the space before the `.o` argument is part of the replacement text and was inserted into the output string. The space before the `.c` is fine because all whitespace before the first argument is stripped off by `make`. In fact, the space before `$(sources)` is probably benign as well since `$(objects)` will most likely be used as a simple command-line argument where leading spaces aren't a problem. However, I would never mix different spacing after commas in a function call even if it yields the correct results:

```
# Yech, the spacing in this call is too subtle.
objects := $(subst .c,.o, $(source))
```

Note that `subst` doesn't understand filenames or file suffixes, just strings of characters. If one of my source files contains a `.c` internally, that too will be substituted. For instance, the filename *car.cdr.c* would be transformed into *car.odr.o*. Probably not what we want.

In the section "Automatic Dependency Generation" in Chapter 2, we talked about dependency generation. The last example *makefile* of that section used `subst` like this:

```
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 subst function is used to transform the source file list into a dependency file list. Since the dependency files appear as an argument to include, they are considered prerequisites and are updated using the %.d rule.

$(patsubst search-pattern,replace-pattern,text)

This is the wildcard version of search and replace. As usual, the pattern can contain a single %. A percent in the replace-pattern is expanded with the matching text. It is important to remember that the search-pattern must match the entire value of text. For instance, the following will delete a trailing slash in text, not every slash in text:

```
strip-trailing-slash = $(patsubst %/,%,$(directory-path))
```

Substitution references are a portable way of performing the same substitution. The syntax of a substitution reference is:

```
$(variable:search=replace)
```

The *search* text can be a simple string; in which case, the string is replaced with *replace* whenever it occurs at the end of a word. That is, whenever it is followed by whitespace or the end of the variable value. In addition, *search* can contain a % representing a wildcard character; in which case, the search and replace follow the rules of patsubst. I find this syntax to be obscure and difficult to read in comparison to patsubst.

As we've seen, variables often contain lists of words. Here are functions to select words from a list, count the length of a list, etc. As with all make functions, words are separated by whitespace.

$(words text)

This returns the number of words in text.

```
CURRENT_PATH := $(subst /, ,$(HOME))
words:
        @echo My HOME path has $(words $(CURRENT_PATH)) directories.
```

This function has many uses, as we'll see shortly, but we need to cover a few more functions to use it effectively.

`$(word n,text)`

This returns the *n*th word in `text`. The first word is numbered 1. If *n* is larger than the number of words in `text`, the value of the function is empty.

```
version_list  := $(subst ., ,$(MAKE_VERSION))
minor_version := $(word 2, $(version_list))
```

The variable `MAKE_VERSION` is a built-in variable. (See the section "Standard make Variables" in Chapter 3.)

You can always get the last word in a list with:

```
current := $(word $(words $(MAKEFILE_LIST)), $(MAKEFILE_LIST))
```

This returns the name of the most recently read *makefile*.

`$(firstword text)`

This returns the first word in `text`. This is equivalent to `$(word 1,text)`.

```
version_list := $(subst ., ,$(MAKE_VERSION))
major_version := $(firstword $(version_list))
```

`$(wordlist start,end,text)`

This returns the words in `text` from `start` to `end`, inclusive. As with the `word` function, the first word is numbered 1. If `start` is greater than the number of words, the value is empty. If `start` is greater than `end`, the value is empty. If `end` is greater than the number of words, all words from `start` on are returned.

```
# $(call uid_gid, user-name)
uid_gid = $(wordlist 3, 4, \
            $(subst :, ,   \
              $(shell grep "^$1:" /etc/passwd)))
```

Important Miscellaneous Functions

Before we push on to functions for managing filenames, let's introduce two very useful functions: `sort` and `shell`.

`$(sort list)`

The `sort` function sorts its `list` argument and removes duplicates. The resulting list contains all the unique words in lexicographic order, each separated by a single space. In addition, `sort` strips leading and trailing blanks.

```
$ make -f- <<< 'x:;@echo =$(sort    d  b s   d      t   )='
=b d s t=
```

The `sort` function is, of course, implemented directly by `make`, so it does not support any of the options of the `sort` program. The function operates on its argument, typically a variable or the return value of another `make` function.

`$(shell command)`

The `shell` function accepts a single argument that is expanded (like all arguments) and passed to a subshell for execution. The standard output of the command is then read and returned as the value of the function. Sequences of newlines in the output are collapsed to a single space. Any trailing newline is deleted. The standard error is not returned, nor is any program exit status.

```
stdout := $(shell echo normal message)
stderr := $(shell echo error message 1>&2)
shell-value:
        # $(stdout)
        # $(stderr)
```

As you can see, messages to *stderr* are sent to the terminal as usual and so are not included in the output of the `shell` function:

```
$ make
error message
# normal message
#
```

Here is a loop to create a set of directories:

```
REQUIRED_DIRS = ...
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); \
             do                               \
               [[ -d $$d ]] || mkdir -p $$d;  \
             done)
```

Often, a *makefile* is easier to implement if essential output directories can be guaranteed to exist before any command scripts are executed. This variable creates the necessary directories by using a `bash` shell "for" loop to ensure that a set of directories exists. The double square brackets are `bash` test syntax similar to the `test` program except that word splitting and pathname expansion are not performed. Therefore if the variable contains a filename with embedded spaces, the test still works correctly (and without quoting). By placing this `make` variable assignment early in the *makefile*, we ensure it is executed before command scripts or other variables use the output directories. The actual value of `_MKDIRS` is irrelevant and `_MKDIRS` itself would never be used.

Since the `shell` function can be used to invoke any external program, you should be careful how you use it. In particular, you should consider the distinction between simple variables and recursive variables.

```
START_TIME   := $(shell date)
CURRENT_TIME =  $(shell date)
```

The `START_TIME` variable causes the `date` command to execute once when the variable is defined. The `CURRENT_TIME` variable will reexecute `date` each time the variable is used in the *makefile*.

Our toolbox is now full enough to write some fairly interesting functions. Here is a function for testing whether a value contains duplicates:

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter              \
                   $(words $1)         \
                   $(words $(sort $1))))
```

We count the words in the list and the unique list, then "compare" the two numbers. There are no make functions that understand numbers, only strings. To compare two numbers, we must compare them as strings. The easiest way to do that is with filter. We search for one number in the other number. The has-duplicates function will be non-null if there are duplicates.

Here is a simple way to generate a filename with a timestamp:

```
RELEASE_TAR := mpwm-$(shell date +%F).tar.gz
```

This produces:

```
mpwm-2003-11-11.tar.gz
```

We could produce the same filename and have date do more of the work with:

```
RELEASE_TAR := $(shell date +mpwm-%F.tar.gz)
```

The next function can be used to convert relative paths (possibly from a *com* directory) into a fully qualified Java class name:

```
# $(call file-to-class-name, file-name)
file-to-class-name := $(subst /,.,$(patsubst %.java,%,$1))
```

This particular pattern can be accomplished with two substs as well:

```
# $(call file-to-class-name, file-name)
file-to-class-name := $(subst /,.,$(subst .java,,$1))
```

We can then use this function to invoke the Java class like this:

```
CALIBRATE_ELEVATOR := com/wonka/CalibrateElevator.java
calibrate:
        $(JAVA) $(call file-to-class-name,$(CALIBRATE_ELEVATOR))
```

If there are more parent directory components in $(sources) above com, they can be removed with the following function by passing the root of the directory tree as the first argument:[*]

```
# $(call file-to-class-name, root-dir, file-name)
file-to-class-name := $(subst /,.,         \
                        $(subst .java,,    \
                          $(subst $1/,,$2)))
```

When reading functions such as this, it is typically easiest to try to understand them inside out. Beginning at the inner-most subst, the function removes the string $1/, then removes the string *.java*, and finally converts all slashes to periods.

[*] In Java, it is suggested that all classes be declared within a package containing the developer's complete Internet domain name, reversed. Also, the directory structure typically mirrors the package structure. Therefore, many source trees look like *root-dir*/com/*company-name*/*dir*.

Filename Functions

Makefile writers spend a lot of time handling files. So it isn't surprising there are a lot of make functions to help with this task.

$wildcard pattern...)

Wildcards were covered in Chapter 2, in the context of targets, prerequisites, and command scripts. But what if we want this functionality in another context, say a variable definition? With the shell function, we could simply use the subshell to expand the pattern, but that would be terribly slow if we needed to do this very often. Instead, we can use the wildcard function:

```
sources := $(wildcard *.c *.h)
```

The wildcard function accepts a list of patterns and performs expansion on each one.[*] If a pattern does not match any files, the empty string is returned. As with wildcard expansion in targets and prerequisites, the normal shell globbing characters are supported: ~, *, ?, [...], and [^...].

Another use of wildcard is to test for the existence of a file in conditionals. When used in conjunction with the if function (described shortly) you often see wildcard function calls whose argument contains no wildcard characters at all. For instance,

```
dot-emacs-exists := $(wildcard ~/.emacs)
```

will return the empty string if the user's home directory does not contain a *.emacs* file.

$(dir list...)

The dir function returns the directory portion of each word in list. Here is an expression to return every subdirectory that contains C files:

```
source-dirs := $(sort                       \
                 $(dir                      \
                   $(shell find . -name '*.c')))
```

The find returns all the source files, then the dir function strips off the file portion leaving the directory, and the sort removes duplicate directories. Notice that this variable definition uses a simple variable to avoid reexecuting the find each time the variable is used (since we assume source files will not spontaneously appear and disappear during the execution of the *makefile*). Here's a function implementation that requires a recursive variable:

```
# $(call source-dirs, dir-list)
source-dirs = $(sort                          \
                $(dir                         \
                  $(shell find $1 -name '*.c'))))
```

[*] The make 3.80 manual fails to mention that more than one pattern is allowed.

This version accepts a space-separated directory list to search as its first parameter. The first arguments to find are one or more directories to search. The end of the directory list is recognized by the first dash argument. (A find feature I didn't know about for several decades!)

`$(notdir name...)`

The notdir function returns the filename portion of a file path. Here is an expression to return the Java class name from a Java source file:

```
# $(call get-java-class-name, file-name)
get-java-class-name = $(notdir $(subst .java,,$1))
```

There are many instances where dir and notdir can be used together to produce the desired output. For instance, suppose a custom shell script must be executed in the same directory as the output file it generates.

```
$(OUT)/myfile.out: $(SRC)/source1.in $(SRC)/source2.in
        cd $(dir $@); \
        generate-myfile $^ > $(notdir $@)
```

The automatic variable, $@, representing the target, can be decomposed to yield the target directory and file as separate values. In fact, if OUT is an absolute path, it isn't necessary to use the notdir function here, but doing so will make the output more readable.

In command scripts, another way to decompose a filename is through the use of $(@D) and $(@F) as mentioned in the section "Automatic Variables" in Chapter 2.

Here are functions for adding and removing file suffixes, etc.

`$(suffix name...)`

The suffix function returns the suffix of each word in its argument. Here is a function to test whether all the words in a list have the same suffix:

```
# $(call same-suffix, file-list)
same-suffix = $(filter 1 $(words $(sort $(suffix $1))))
```

A more common use of the suffix function is within conditionals in conjunction with findstring.

`$(basename name...)`

The basename function is the complement of suffix. It returns the filename without its suffix. Any leading path components remain intact after the basename call. Here are the earlier file-to-class-name and get-java-class-name functions rewritten with basename:

```
# $(call file-to-class-name, root-directory, file-name)
file-to-class-name  := $(subst /,.,          \
                         $(basename          \
                           $(subst $1/,,$2)))
# $(call get-java-class-name, file-name)
get-java-class-name =  $(notdir $(basename $1))
```

$(addsuffix suffix,name...)

The addsuffix function appends the given suffix text to each word in name. The suffix text can be anything. Here is a function to find all the files in the PATH that match an expression:

```
# $(call find-program, filter-pattern)
find-program = $(filter $1,                        \
                 $(wildcard                        \
                   $(addsuffix /*,                 \
                     $(sort                        \
                       $(subst :, ,                \
                         $(subst ::,:.:,           \
                           $(patsubst :%,.:%,      \
                             $(patsubst %:,%:.,$(PATH)))))))))
find:
        @echo $(words $(call find-program, %))
```

The inner-most three substitutions account for a special case in shell syntax. An empty path component is taken to mean the current directory. To normalize this special syntax we search for an empty trailing path component, an empty leading path component, and an empty interior path component, in that order. Any matching components are replaced with ".". Next, the path separator is replaced with a space to create separate words. The sort function is used to remove repeated path components. Then the globbing suffix /* is appended to each word and wildcard is invoked to expand the globbing expressions. Finally, the desired patterns are extracted by filter.

Although this may seem like an extremely slow function to run (and it may well be on many systems), on my 1.9 GHz P4 with 512 MB this function executes in 0.20 seconds and finds 4,335 programs. This performance can be improved by moving the $1 argument inside the call to wildcard. The following version eliminates the call to filter and changes addsuffix to use the caller's argument.

```
# $(call find-program,wildcard-pattern)
find-program = $(wildcard                          \
                 $(addsuffix /$1,                  \
                   $(sort                          \
                     $(subst :, ,                  \
                       $(subst ::,:.:,             \
                         $(patsubst :%,.:%,        \
                           $(patsubst %:,%:.,$(PATH))))))))
find:
        @echo $(words $(call find-program,*))
```

This version runs in 0.17 seconds. It runs faster because wildcard no longer returns every file only to make the function discard them later with filter. A similar example occurs in the GNU make manual. Notice also that the first version uses filter-style globbing patterns (using % only) while the second version uses wildcard-style globbing patterns (~, *, ?, [...], and [^...]).

`$(addprefix prefix,name...)`

The `addprefix` function is the complement of `addsuffix`. Here is an expression to test whether a set of files exists and is nonempty:

```
# $(call valid-files, file-list)
valid-files = test -s . $(addprefix -a -s ,$1)
```

This function is different from most of the previous examples in that it is intended to be executed in a command script. It uses the shell's `test` program with the `-s` option ("true if the file exists and is not empty") to perform the test. Since the `test` command requires a `-a` (and) option between multiple filenames, `addprefix` prepends the `-a` before each filename. The first file used to start the "and" chain is dot, which always yields true.

`$(join prefix-list,suffix-list)`

The `join` function is the complement of `dir` and `notdir`. It accepts two lists and concatenates the first element from `prefix-list` with the first element from `suffix-list`, then the second element from `prefix-list` with the second element from `suffix-list` and so on. It can be used to reconstruct lists decomposed with `dir` and `notdir`.

Flow Control

Because many of the functions we have seen so far are implemented to perform their operations on lists, they work well even without a looping construct. But without a true looping operator and conditional processing of some kind the `make` macro language would be very limited, indeed. Fortunately, `make` provides both of these language features. I have also thrown into this section the fatal `error` function, clearly a very extreme form of flow control!

`$(if condition,then-part,else-part)`

The `if` function (not to be confused with the conditional directives `ifeq`, `ifneq`, `ifdef`, and `ifndef` discussed in Chapter 3) selects one of two macro expansions depending on the "value" of the conditional expression. The `condition` is true if its expansion contains any characters (even space). In this case, the `then-part` is expanded. Otherwise, if the expansion of `condition` is empty, it is false and the `else-part` is expanded.*

Here is an easy way to test whether the *makefile* is running on Windows. Look for the `COMSPEC` environment variable defined only on Windows:

```
PATH_SEP := $(if $(COMSPEC),;,:)
```

* In Chapter 3, I made a distinction between macro languages and other programming languages. Macro languages work by transforming source text into output text through defining and expanding macros. This distinction becomes clearer as we see how the `if` function works.

make evaluates the condition by first removing leading and trailing whitespace, then expanding the expression. If the expansion yields any characters (including whitespace), the expression is true. Now PATH_SEP contains the proper character to use in paths, whether the *makefile* is running on Windows or Unix.

In the last chapter, we mentioned checking the version of make if you use some of the newest features (like eval). The if and filter functions are often used together to test the value of a string:

```
$(if $(filter $(MAKE_VERSION),3.80),,\
  $(error This makefile requires GNU make version 3.80.))
```

Now, as subsequent versions of make are released, the expression can be extended with more acceptable versions:

```
$(if $(filter $(MAKE_VERSION),3.80 3.81 3.90 3.92),,\
  $(error This makefile requires one of GNU make version ….))
```

This technique has the disadvantage that the code must be updated when a new version of make is installed. But that doesn't happen very often. (For instance, 3.80 has been the release version since October 2002.) The above test can be added to a *makefile* as a top-level expression since the if collapses to nothing if true and error terminates the make otherwise.

$(error *text*)

The error function is used for printing fatal error messages. After the function prints its message, make terminates with an exit status of 2. The output is prefixed with the name of the current *makefile*, the current line number, and the message text. Here is an implementation of the common assert programming construct for make:

```
# $(call assert,condition,message)
define assert
  $(if $1,,$(error Assertion failed: $2))
endef
# $(call assert-file-exists,wildcard-pattern)
define assert-file-exists
  $(call assert,$(wildcard $1),$1 does not exist)
endef
# $(call assert-not-null,make-variable)
define assert-not-null
  $(call assert,$($1),The variable "$1" is null)
endef
error-exit:
        $(call assert-not-null,NON_EXISTENT)
```

The first function, assert, just tests its first argument and prints the user's error message if it is empty. The second function builds on the first and tests that a wildcard pattern yields an existing file. Note that the argument can include any number of globbing patterns.

The third function is a very useful assert that relies on *computed variables*. A make variable can contain anything, including the name of another make variable. But if a variable contains the name of another variable how can you access the value of that other variable? Well, very simply by expanding the variable twice:

```
NO_SPACE_MSG := No space left on device.
NO_FILE_MSG  := File not found.
…;
STATUS_MSG   := NO_SPACE_MSG
$(error $($(STATUS_MSG)))
```

This example is slightly contrived to keep it simple, but here `STATUS_MSG` is set to one of several error messages by storing the error message variable name. When it comes time to print the message, `STATUS_MSG` is first expanded to access the error message variable name, `$(STATUS_MSG)`, then expanded again to access the message text, `$($(STATUS_MSG))`. In our `assert-not-null` function we assume the argument to the function is the *name* of a `make` variable. We first expand the argument, `$1`, to access the variable name, then expand again, `$($1)`, to determine if it has a value. If it is null, then we have the variable name right in `$1` to use in the error message.

```
$ make
Makefile:14: *** Assertion failed: The variable "NON_EXISTENT" is null.  Stop.
```

There is also a `warning` function (see the section "Less Important Miscellaneous Functions" later in this chapter) that prints a message in the same format as `error`, but does not terminate `make`.

`$(foreach variable,list,body)`

The `foreach` function provides a way to expand text repeatedly while substituting different values into each expansion. Notice that this is different from executing a function repeatedly with different arguments (although it can do that, too). For example:

```
letters := $(foreach letter,a b c d,$(letter))
show-words:
        # letters has $(words $(letters)) words: '$(letters)'
$ make
# letters has 4 words: 'a b c d'
```

When this `foreach` is executed, it sets the loop control variable, `letter`, to each value in `a b c d` and expands the body of the loop, `$(letter)`, once for each value. The expanded text is accumulated with a space separating each expansion.

Here is a function to test if a set of variables is set:

```
VARIABLE_LIST := SOURCES OBJECTS HOME
$(foreach i,$(VARIABLE_LIST), \
  $(if $($i),,               \
    $(shell echo $i has no value > /dev/stderr)))
```

(The pseudo file */dev/stderr* in the `shell` function requires setting `SHELL` to `bash`.) This loop sets `i` to each word of `VARIABLE_LIST`. The test expression inside the `if` first evaluates `$i` to get the variable name, then evaluates this again in a computed expression `$($i)` to see if it is non-null. If the expression has a value, the *then* part does nothing; otherwise, the *else* part prints a warning. Note that if we omit the redirection from the echo, the output of the `shell` command will be substituted into the *makefile*, yielding a syntax error. As shown, the entire `foreach` loop expands to nothing.

As promised earlier, here is a function that gathers all the words that contain a substring from a list:

```
# $(call grep-string, search-string, word-list)
define grep-string
$(strip                                 \
  $(foreach w, $2,                       \
    $(if $(findstring $1, $w),           \
      $w)))
endef
words := count_words.c counter.c lexer.l lexer.h counter.h
find-words:
        @echo $(call grep-string,un,$(words))
```

Unfortunately, this function does not accept patterns, but it does find simple substrings:

```
$ make
count_words.c counter.c counter.h
```

Style note concerning variables and parentheses

As noted earlier, parentheses are not required for `make` variables of one character. For instance, all of the basic automatic variables are one character. Automatic variables are universally written without parentheses even in the GNU `make` manual. However, the `make` manual uses parentheses for virtually all other variables, even single character variables, and strongly urges users to follow suit. This highlights the special nature of `make` variables since almost all other programs that have "dollar variables" (such as shells, `perl`, `awk`, `yacc`, etc.) don't require parentheses. One of the more common `make` programming errors is forgetting parentheses. Here is a common use of `foreach` containing the error:

```
INCLUDE_DIRS := …
INCLUDES := $(foreach i,$INCLUDE_DIRS,-I $i)
# INCLUDES now has the value "-I NCLUDE_DIRS"
```

However, I find that reading macros can be much easier through the judicious use of single-character variables and omitting unnecessary parentheses. For instance, I think the `has-duplicates` function is easier to read without full parentheses:

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter             \
                   $(words $1)         \
                   $(words $(sort $1))))
```

versus:

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter               \
                   $(words $(1))         \
                   $(words $(sort $(1)))))
```

However, the `kill-program` function might be more readable with full parentheses since it would help distinguish `make` variables from shell variables or variables used in other programs:

```
define kill-program
  @ $(PS) $(PS_FLAGS) |                             \
  $(AWK) 'BEGIN { FIELDWIDTHS = $(PS_FIELDS) }      \
          /$(1)/{                                   \
                  print "Killing " $$3;             \
                  system( "$(KILL) $(KILLFLAGS) " $$1 ) \
                }'
endef
```

The search string contains the first parameter to the macro, `$(1)`. `$$3` and `$$1` refer to `awk` variables.

I use single-character variables and omit the parentheses only when it seems to make the code more readable. I typically do this for the parameters to macros and the control variable in `foreach` loops. You should follow a style that suits your situation. If you have any doubts about the maintainability of your *makefile*s, follow the `make` manual's suggestion and use full parentheses. Remember, the `make` program is all about easing the problems associated with maintaining software. If you keep that in mind as you write your *makefile*s, you will most likely stay clear of trouble.

Less Important Miscellaneous Functions

Finally, we have some miscellaneous (but important) string functions. Although minor in comparison with `foreach` or `call`, you'll find yourself using these very often.

`$(strip text)`

The `strip` function removes all leading and trailing whitespace from `text` and replaces all internal whitespace with a single space. A common use for this function is to clean up variables used in conditional expressions.

I most often use this function to remove unwanted whitespace from variable and macro definitions I've formatted across multiple lines. But it can also be a good idea to wrap the function parameters `$1`, `$2`, etc., with `strip` if the function is sensitive to leading blanks. Often programmers unaware of the subtleties of `make` will add a space after commas in a `call` argument list.

`$(origin variable)`

The `origin` function returns a string describing the origin of a variable. This can be very useful in deciding how to use the value of a variable. For instance, you might want to ignore the value of a variable if it came from the environment, but not if it was set from the command line. For a more concrete example, here is a new assert function that tests if a variable is defined:

```
# $(call assert-defined,variable-name)
define assert-defined
  $(call assert,                    \
    $(filter-out undefined,$(origin $1)), \
    '$1' is undefined)
endef
```

The possible return values of `origin` are:

`undefined`
: The variable has never been defined.

`default`
: The variable's definition came from `make`'s built-in database. If you alter the value of a built-in variable, `origin` returns the origin of the most recent definition.

`environment`
: The variable's definition came from the environment (and the `--environment-overrides` option is *not* turned on).

`environment override`
: The variable's definition came from the environment (and the `--environment-overrides` option *is* turned on).

`file`
: The variable's definition came from the *makefile*.

`command line`
: The variable's definition came from the command line.

`override`
: The variable's definition came from an `override` directive.

`automatic`
: The variable is an automatic variable defined by `make`.

`$(warning text)`
: The `warning` function is similar to the `error` function except that it does not cause `make` to exit. Like the `error` function, the output is prefixed with the name of the current *makefile* and the current line number followed by the message text. The `warning` function expands to the empty string so it can be used almost anywhere.

```
    $(if $(wildcard $(JAVAC)),,                         \
      $(warning The java compiler variable, JAVAC ($(JAVAC)), \
                is not properly set.))
```

Advanced User-Defined Functions

We'll spend a lot of time writing macro functions. Unfortunately, there aren't many features in `make` for helping to debug them. Let's begin by trying to write a simple debugging trace function to help us out.

As we've mentioned, `call` will bind each of its parameters to the numbered variables `$1`, `$2`, etc. Any number of arguments can be given to `call`. As a special case, the name of the currently executing function (i.e., the variable name) is accessible through `$0`. Using this information, we can write a pair of debugging functions for tracing through macro expansion:

```
# $(debug-enter)
debug-enter = $(if $(debug_trace),\
                $(warning Entering $0($(echo-args))))

# $(debug-leave)
debug-leave = $(if $(debug_trace),$(warning Leaving $0))

comma := ,
echo-args   = $(subst ' ','$(comma) ',\
                $(foreach a,1 2 3 4 5 6 7 8 9,'$($a)'))
```

If we want to watch how functions `a` and `b` are invoked, we can use these trace functions like this:

```
debug_trace = 1

define a
  $(debug-enter)
  @echo $1 $2 $3
  $(debug-leave)
endef

define b
  $(debug-enter)
  $(call a,$1,$2,hi)
  $(debug-leave)
endef

trace-macro:
        $(call b,5,$(MAKE))
```

By placing `debug-enter` and `debug-leave` variables at the start and end of your functions, you can trace the expansions of your own functions. These functions are far from perfect. The `echo-args` function will echo only the first nine arguments and, worse, it cannot determine the number of actual arguments in the call (of course, neither can `make`!). Nevertheless, I've used these macros "as is" in my own debugging. When executed, the *makefile* generates this trace output:

```
$ make
makefile:14: Entering b( '5', 'make', '', '', '', '', '', '', '')
makefile:14: Entering a( '5', 'make', 'hi', '', '', '', '', '', '')
makefile:14: Leaving a
makefile:14: Leaving b
5 make hi
```

As a friend said to me recently, "I never thought of `make` as a programming language before." GNU `make` isn't your grandmother's `make`!

eval and value

The `eval` function is completely different from the rest of the built-in functions. Its purpose is to feed text directly to the `make` parser. For instance,

```
$(eval sources := foo.c bar.c)
```

The argument to `eval` is first scanned for variables and expanded (as all arguments to all functions are), then the text is parsed and evaluated as if it had come from an input file. This example is so simple you might be wondering why you would bother with this function. Let's try a more interesting example. Suppose you have a *makefile* to compile a dozen programs and you want to define several variables for each program, say `sources`, `headers`, and `objects`. Instead of repeating these variable assignments over and over with each set of variables:

```
ls_sources := ls.c glob.c
ls_headers := ls.h glob.h
ls_objects := ls.o glob.o
…
```

We might try to define a macro to do the job:

```
# $(call program-variables, variable-prefix, file-list)
define program-variables
  $1_sources = $(filter %.c,$2)
  $1_headers = $(filter %.h,$2)
  $1_objects = $(subst .c,.o,$(filter %.c,$2))
endef

$(call program-variables, ls, ls.c ls.h glob.c glob.h)

show-variables:
        # $(ls_sources)
        # $(ls_headers)
        # $(ls_objects)
```

The `program-variables` macro accepts two arguments: a prefix for the three variables and a file list from which the macro selects files to set in each variable. But, when we try to use this macro, we get the error:

```
$ make
Makefile:7: *** missing separator.  Stop.
```

This doesn't work as expected because of the way the `make` parser works. A macro (at the top parsing level) that expands to multiple lines is illegal and results in syntax errors. In this case, the parser believes this line is a rule or part of a command script but is missing a separator token. Quite a confusing error message. The `eval` function was introduced to handle this issue. If we change our `call` line to:

```
$(eval $(call program-variables, ls, ls.c ls.h glob.c glob.h))
```

we get what we expect:

```
$ make
# ls.c glob.c
# ls.h glob.h
# ls.o glob.o
```

Using `eval` resolves the parsing issue because `eval` handles the multiline macro expansion and itself expands to zero lines.

Now we have a macro that defines three variables very concisely. Notice how the assignments in the macro compose variable names from a prefix passed in to the function and a fixed suffix, `$1_sources`. These aren't precisely computed variables as described previously, but they have much the same flavor.

Continuing this example, we realize we can also include our rules in the macro:

```
# $(call program-variables,variable-prefix,file-list)
define program-variables
  $1_sources = $(filter %.c,$2)
  $1_headers = $(filter %.h,$2)
  $1_objects = $(subst .c,.o,$(filter %.c,$2))

  $($1_objects): $($1_headers)
endef

ls: $(ls_objects)

$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
```

Notice how these two versions of `program-variables` illustrate a problem with spaces in function arguments. In the previous version, the simple uses of the two function parameters were immune to leading spaces on the arguments. That is, the code behaved the same regardless of any leading spaces in `$1` or `$2`. The new version, however, introduced the computed variables `$($1_objects)` and `$($1_headers)`. Now adding a leading space to the first argument to our function (`ls`) causes the computed variable to begin with a leading space, which expands to nothing because no variable we've defined begins with a leading space. This can be quite an insidious problem to diagnose.

When we run this *makefile*, we discover that somehow the *.h* prerequisites are being ignored by `make`. To diagnose this problem, we examine `make`'s internal database by running `make` with its `--print-data-base` option and we see something strange:

```
$ make --print-database | grep ^ls
ls_headers = ls.h glob.h
ls_sources = ls.c glob.c
ls_objects = ls.o glob.o
ls.c:
ls.o: ls.c
ls: ls.o
```

The *.h* prerequisites for *ls.o* are missing! There is something wrong with the rule using computed variables.

When make parses the eval function call, it first expands the user-defined function, program-variables. The first line of the macro expands to:

```
ls_sources = ls.c glob.c
```

Notice that each line of the macro is expanded immediately as expected. The other variable assignments are handled similarly. Then we get to the rule:

```
$($1_objects): $($1_headers)
```

The computed variables first have their variable name expanded:

```
$(ls_objects): $(ls_headers)
```

Then the outer variable expansion is performed, yielding:

```
:
```

Wait! Where did our variables go? The answer is that the previous three assignment statements were expanded *but not evaluated* by make. Let's keep going to see how this works. Once the call to program-variables has been expanded, make sees something like:

```
$(eval   ls_sources = ls.c glob.c
ls_headers = ls.h glob.h
ls_objects = ls.o glob.o

:)
```

The eval function then executes and defines the three variables. So, the answer is that the variables in the rule are being expanded before they have actually been defined.

We can resolve this problem by explicitly deferring the expansion of the computed variables until the three variables are defined. We can do this by quoting the dollar signs in front of the computed variables:

```
$$($1_objects): $$($1_headers)
```

This time the make database shows the prerequisites we expect:

```
$ make -p | grep ^ls
ls_headers = ls.h glob.h
ls_sources = ls.c glob.c
ls_objects = ls.o glob.o
ls.c:
ls.o: ls.c ls.h glob.h
ls: ls.o
```

To summarize, the argument to eval is expanded *twice*: once when when make prepares the argument list for eval, and once again by eval.

We resolved the last problem by deferring evaluation of the computed variables. Another way of handling the problem is to force early evaluation of the variable assignments by wrapping each one with eval:

```
# $(call program-variables,variable-prefix,file-list)
define program-variables
```

```
  $(eval $1_sources = $(filter %.c,$2))
  $(eval $1_headers = $(filter %.h,$2))
  $(eval $1_objects = $(subst .c,.o,$(filter %.c,$2)))

  $($1_objects): $($1_headers)
endef

ls: $(ls_objects)

$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
```

By wrapping the variable assignments in their own eval calls, we cause them to be internalized by make while the program-variables macro is being expanded. They are then available for use within the macro immediately.

As we enhance our *makefile*, we realize we have another rule we can add to our macro. The program itself depends on its objects. So, to finish our parameterized *makefile*, we add a top-level *all* target and need a variable to hold all the programs our *makefile* can manage:

```
#$(call program-variables,variable-prefix,file-list)
define program-variables
  $(eval $1_sources = $(filter %.c,$2))
  $(eval $1_headers = $(filter %.h,$2))
  $(eval $1_objects = $(subst .c,.o,$(filter %.c,$2)))

  programs += $1

  $1: $($1_objects)

  $($1_objects): $($1_headers)
endef

# Place all target here, so it is the default goal.
all:

$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
$(eval $(call program-variables,cp,...))
$(eval $(call program-variables,mv,...))
$(eval $(call program-variables,ln,...))
$(eval $(call program-variables,rm,...))

# Place the programs prerequisite here where it is defined.
all: $(programs)
```

Notice the placement of the all target and its prerequisite. The programs variable is not properly defined until after the five eval calls, but we would like to place the all target first in the *makefile* so all is the default goal. We can satisfy all our constrains by putting all first and adding the prerequisites later.

The program-variables function had problems because some variables were evaluated too early. make actually offers a value function to help address this situation. The value function returns the value of its variable argument *unexpanded*. This unexpanded value can then be passed to eval for processing. By returning an unexpanded value, we can avoid the problem of having to quote some of the variable references in our macros.

Unfortunately, this function cannot be used with the program-variables macro. That's because value is an all-or-nothing function. If used, value will not expand *any* of the variables in the macro. Furthermore, value doesn't accept parameters (and wouldn't do anything with them if it did) so our program name and file list parameters wouldn't be expanded.

Because of these limitations, you won't see value used very often in this book.

Hooking Functions

User-defined functions are just variables holding text. The call function will expand $1, $2, etc. references in the variable text if they exist. If the function doesn't contain any of these variable references, call doesn't care. In fact, if the variable doesn't contain any text, call doesn't care. No error or warning occurs. This can be very frustrating if you happen to misspell a function name. But it can also be very useful.

Functions are all about reusable code. The more often you reuse a function, the more worthwhile it is to write it well. Functions can be made more reusable by adding *hooks* to them. A *hook* is a function reference that can be redefined by a user to perform their own custom tasks during a standard operation.

Suppose you are building many libraries in your *makefile*. On some systems, you'd like to run ranlib and on others you might want to run chmod. Rather than writing explicit commands for these operations, you might choose to write a function and add a hook:

```
# $(call build-library, object-files)
define build-library
  $(AR) $(ARFLAGS) $@ $1
  $(call build-library-hook,$@)
endef
```

To use the hook, define the function build-library-hook:

```
$(foo_lib): build-library-hook = $(RANLIB) $1
$(foo_lib): $(foo_objects)
        $(call build-library,$^)

$(bar_lib): build-library-hook = $(CHMOD) 444 $1
$(bar_lib): $(bar_objects)
        $(call build-library,$^)
```

Passing Parameters

A function can get its data from four "sources": parameters passed in using call, global variables, automatic variables, and target-specific variables. Of these, relying on parameters is the most modular choice, since their use insulates the function from any changes to global data, but sometimes that isn't the most important criteria.

Suppose we have several projects using a common set of make functions. Each project might be identified by a variable prefix, say PROJECT1_, and critical variables for the project all use the prefix with cross-project suffixes. The earlier example, PROJECT_SRC, might look like PROJECT1_SRC, PROJECT1_BIN, and PROJECT1_LIB. Rather than write a function that requires these three variables we could instead use computed variables and pass a single argument, the prefix:

```
# $(call process-xml,project-prefix,file-name)
define process-xml
  $($1_LIB)/xmlto -o $($1_BIN)/xml/$2 $($1_SRC)/xml/$2
endef
```

Another approach to passing arguments uses target-specific variables. This is particularly useful when most invocations use a standard value but a few require special processing. Target-specific variables also provide flexibility when the rule is defined in an include file, but invoked from a *makefile* where the variable is defined.

```
release: MAKING_RELEASE = 1
release: libraries executables
…
$(foo_lib):
        $(call build-library,$^)
…
# $(call build-library, file-list)
define build-library
  $(AR) $(ARFLAGS) $@          \
    $(if $(MAKING_RELEASE),     \
      $(filter-out debug/%,$1), \
      $1)
endef
```

This code sets a target-specific variable to indicate when a release build is being executed. In that case, the library-building function will filter out any debugging modules from the libraries.

CHAPTER 5

Commands

We've already covered many of the basic elements of `make` commands, but just to make sure we're all on the same page, let's review a little.

Commands are essentially one-line shell scripts. In effect, `make` grabs each line and passes it to a subshell for execution. In fact, `make` can optimize this (relatively) expensive fork/exec algorithm if it can guarantee that omitting the shell will not change the behavior of the program. It checks this by scanning each command line for shell special characters, such as wildcard characters and i/o redirection. If none are found, `make` directly executes the command without passing it to a subshell.

By default, */bin/sh* is used for the shell. This shell is controlled by the `make` variable `SHELL` but it is not inherited from the environment. When `make` starts, it imports all the variables from the user's environment as `make` variables, except `SHELL`. This is because the user's choice of shell should not cause a *makefile* (possibly included in some downloaded software package) to fail. If a user really wants to change the default shell used by `make`, he can set the `SHELL` variable explicitly in the *makefile*. We will discuss this issue in the section "Which Shell to Use" later in this chapter.

Parsing Commands

Following a `make` target, lines whose first character is a tab are assumed to be commands (unless the previous line was continued with a backslash). GNU `make` tries to be as smart as possible when handling tabs in other contexts. For instance, when there is no possible ambiguity, comments, variable assignments, and include directives may all use a tab as their first character. If `make` reads a command line that does not immediately follow a target, an error message is displayed:

```
makefile:20: *** commands commence before first target.  Stop.
```

The wording of this message is a bit odd because it often occurs in the middle of a *makefile* long after the "first" target was specified, but we can now understand it without too much trouble. A better wording for this message might be, "encountered a command outside the context of a target."

When the parser sees a command in a legal context, it switches to "command parsing" mode, building the script one line at a time. It stops appending to the script when it encounters a line that cannot possibly be part of the command script. There the script ends. The following may appear in a command script:

- Lines beginning with a tab character are commands that will be executed by a subshell. Even lines that would normally be interpreted as `make` constructs (e.g., `ifdef`, comments, `include` directives) are treated as commands while in "command parsing" mode.
- Blank lines are ignored. They are not "executed" by a subshell.
- Lines beginning with a `#`, possibly with leading spaces (not tabs!), are *makefile* comments and are ignored.
- Conditional processing directives, such as `ifdef` and `ifeq`, are recognized and processed normally within command scripts.

Built-in `make` functions terminate command parsing mode unless preceded by a tab character. This means they must expand to valid shell commands or to nothing. The functions `warning` and `eval` expand to no characters.

The fact that blank lines and `make` comments are allowed in command scripts can be surprising at first. The following lines show how it is carried out:

```
long-command:
        @echo Line 2: A blank line follows

        @echo Line 4: A shell comment follows
        # A shell comment (leading tab)
        @echo Line 6: A make comment follows
# A make comment, at the beginning of a line
        @echo Line 8: Indented make comments follow
  # A make comment, indented with leading spaces
        # Another make comment, indented with leading spaces
        @echo Line 11: A conditional follows
    ifdef COMSPEC
        @echo Running Windows
    endif
        @echo Line 15: A warning "command" follows
        $(warning A warning)
        @echo Line 17: An eval "command" follows
        $(eval $(shell echo Shell echo 1>&2))
```

Notice that lines 5 and 10 appear identical, but are quite different. Line 5 is a shell comment, indicated by a leading tab, while line 10 is a `make` comment indented eight spaces. Obviously, we do not recommend formatting `make` comments this way (unless you intend entering an obfuscated *makefile* contest). As you can see in the following output, make comments are not executed and are not echoed to the output even though they occur within the context of a command script:

```
$ make
makefile:2: A warning
Shell echo
Line 2: A blank line follows
Line 4: A shell comment follows
# A shell comment (leading tab)
Line 6: A make comment follows
Line 8: Indented make comments follow
Line 11: A conditional follows
Running Windows
Line 15: A warning command follows
Line 17: An eval command follows
```

The output of the warning and eval functions appears to be out of order, but don't worry, it isn't. (We'll discuss the order of evaluation later this chapter in the section "Evaluating Commands.") The fact that command scripts can contain any number of blank lines and comments can be a frustrating source of errors. Suppose you accidentally introduce a line with a leading tab. If a previous target (with or without commands) exists and you have only comments or blank lines intervening, make will treat your accidental tabbed line as a command associated with the preceding target. As you've seen, this is perfectly legal and will not generate a warning or error unless the same target has a rule somewhere else in the *makefile* (or one of its include files).

If you're lucky, your *makefile* will include a nonblank, noncomment between your accidental tabbed line and the previous command script. In that case, you'll get the "commands commence before first target" message.

Now is a good time to briefly mention software tools. I think everyone agrees, now, that using a leading tab to indicate a command line was an unfortunate decision, but it's a little late to change. Using a modern, syntax-aware editor can help head off potential problems by visibly marking dubious constructs. GNU emacs has a very nice mode for editing *makefile*s. This mode performs syntax highlighting and looks for simple syntactic errors, such as spaces after continuation lines and mixing leading spaces and tabs. I'll talk more about using emacs and make later on.

Continuing Long Commands

Since each command is executed in its own shell (or at least *appears* to be), sequences of shell commands that need to be run together must be handled specially. For instance, suppose I need to generate a file containing a list of files. The Java compiler accepts such a file for compiling many source files. I might write a command script like this:

```
.INTERMEDIATE: file_list
file_list:
        for d in logic ui
        do
          echo $d/*.java
        done > $@
```

By now it should be clear that this won't work. It generates the error:

```
$ make
for d in logic ui
/bin/sh: -c: line 2: syntax error: unexpected end of file
make: *** [file_list] Error 2
```

Our first fix is to add continuation characters to each line:

```
.INTERMEDIATE: file_list
file_list:
        for d in logic ui       \
        do                      \
          echo $d/*.java        \
        done > $@
```

which generates the error:

```
$ make
for d in logic ui       \
do                      \
  echo /*.java  \
done > file_list
/bin/sh: -c: line 1: syntax error near unexpected token `>'
/bin/sh: -c: line 1: `for d in logic ui  do                      echo /*.java
make: *** [file_list] Error 2
```

What happened? Two problems. First, the reference to the loop control variable, d, needs to be escaped. Second, since the for loop is passed to the subshell as a single line, we must add semicolon separators after the file list and for-loop statement:

```
.INTERMEDIATE: file_list
file_list:
        for d in logic ui;      \
        do                      \
          echo $$d/*.java;      \
        done > $@
```

Now we get the file we expect. The target is declared .INTERMEDIATE so that make will delete this temporary target after the compile is complete.

In a more realistic example, the list of directories would be stored in a make variable. If we are sure that the number of files is relatively small, we can perform this same operation without a for loop by using make functions:

```
.INTERMEDIATE: file_list
file_list:
        echo $(addsuffix /*.java,$(COMPILATION_DIRS)) > $@
```

But the for-loop version is less likely to run up against command-line length issues if we expect the list of directories to grow with time.

Another common problem in make command scripts is how to switch directories. Again, it should be clear that a simple command script like:

```
TAGS:
        cd src
        ctags --recurse
```

will not execute the ctags program in the *src* subdirectory. To get the effect we want, we must either place both commands on a single line or escape the newline with a backslash (and separate the commands with a semicolon):

```
TAGS:
        cd src;         \
        ctags --recurse
```

An even better version would check the status of the cd before executing the ctags program:

```
TAGS:
        cd src &&         \
        ctags --recurse
```

Notice that in some circumstances omitting the semicolon might not produce a make or shell error:

```
disk-free = echo "Checking free disk space..." \
            df . | awk '{ print $$4 }'
```

This example prints a simple message followed by the number of free blocks on the current device. Or does it? We have accidentally omitted the semicolon after the echo command, so we never actually run the df program. Instead, we echo:

```
Checking free disk space... df .
```

into awk which dutifully prints the fourth field, space....

It might have occurred to you to use the define directive, which is intended for creating multiline command sequences, rather than continuation lines. Unfortunately, this isn't quite the same problem. When a multiline macro is expanded, each line is inserted into the command script with a leading tab and make treats each line independently. The lines of the macro are not executed in a single subshell. So you will need to pay attention to command-line continuation in macros as well.

Command Modifiers

A command can be modified by several prefixes. We've already seen the "silent" prefix, @, used many times before. The complete list of prefixes, along with some gory details, are:

@ Do not echo the command. For historical compatibility, you can make your target a prerequisite of the special target .SILENT if you want all of its commands to be hidden. Using @ is preferred, however, because it can be applied to individual commands within a command script. If you want to apply this modifier to all targets (although it is hard to imagine why), you can use the `--silent` (or `-s`) option.

Hiding commands can make the output of `make` easier on the eyes, but it can also make debugging the commands more difficult. If you find yourself removing the `@` modifiers and restoring them frequently, you might create a variable, say `QUIET`, containing the `@` modifier and use that on commands:

```
QUIET = @
hairy_script:
        $(QUIET) complex script …
```

Then, if you need to see the complex script as `make` runs it, just reset the `QUIET` variable from the command line:

```
$ make QUIET= hairy_script
complex script …
```

`-` The dash prefix indicates that errors in the command should be ignored by `make`. By default, when `make` executes a command, it examines the exit status of the program or pipeline, and if a nonzero (failure) exit status is returned, `make` terminates execution of the remainder of the command script and exits. This modifier directs `make` to ignore the exit status of the modified line and continue as if no error occurred. We'll discuss this topic in more depth in the next section.

For historical compatibility, you can ignore errors in any part of a command script by making the target a prerequisite of the `.IGNORE` special target. If you want to ignore all errors in the entire *makefile,* you can use the `--ignore-errors` (or `-i`) option. Again, this doesn't seem too useful.

`+` The plus modifier tells `make` to execute the command even if the `--just-print` (or `-n`) command-line option is given to `make`. It is used when writing recursive *makefile*s. We'll discuss this topic in more detail in the section "Recursive make" in Chapter 6.

Any or all of these modifiers are allowed on a single line. Obviously, the modifiers are stripped before the commands are executed.

Errors and Interrupts

Every command that `make` executes returns a status code. A status of zero indicates that the command succeeded. A status of nonzero indicates some kind of failure. Some programs use the return status code to indicate something more meaningful than simply "error." For instance, `grep` returns 0 (success) if a match is found, 1 if no match is found, and 2 if some kind of error occurred.

Normally, when a program fails (i.e., returns a nonzero exit status), `make` stops executing commands and exits with an error status. Sometimes you want `make` to continue, trying to complete as many targets as possible. For instance, you might want to compile as many files as possible to see all the compilation errors in a single run. You can do this with the `--keep-going` (or `-k`) option.

Although the - modifier causes make to ignore errors in individual commands, I try to avoid its use whenever possible. This is because it complicates automated error processing and is visually jarring.

When make ignores an error it prints a warning along with the name of the target in square brackets. For example, here is the output when rm tries to delete a nonexistent file:

```
rm non-existent-file
rm: cannot remove `non-existent-file': No such file or directory
make: [clean] Error 1 (ignored)
```

Some commands, like rm, have options that suppress their error exit status. The -f option will force rm to return success while also suppressing error messages. Using such options is better than depending on a preceding dash.

Occasionally, you want a command to fail and would like to get an error if the program succeeds. For these situations, you should be able to simply negate the exit status of the program:

```
# Verify there are no debug statements left in the code.
.PHONY: no_debug_printf
no_debug_printf: $(sources)
        ! grep --line-number '"debug:' $^
```

Unfortunately, there is a bug in make 3.80 that prevents this straightforward use. make does not recognize the ! character as requiring shell processing and executes the command line itself, resulting in an error. In this case, a simple work around is to add a shell special character as a clue to make:

```
# Verify there are no debug statement left in the code
.PHONY: no_debug_printf
no_debug_printf: $(sources)
        ! grep --line-number '"debug:' $^ < /dev/null
```

Another common source of unexpected command errors is using the shell's if construct without an else.

```
$(config): $(config_template)
        if [ ! -d $(dir $@) ];     \
        then                       \
          $(MKDIR) $(dir $@);      \
        fi
        $(M4) $^ > $@
```

The first command tests if the output directory exists and calls mkdir to create it if it does not. Unfortunately, if the directory does exist, the if command returns a failure exit status (the exit status of the test), which terminates the script. One solution is to add an else clause:

```
$(config): $(config_template)
        if [ ! -d $(dir $@) ];     \
        then                       \
          $(MKDIR) $(dir $@);      \
        else                    \
          true;                 \
        fi
        $(M4) $^ > $@
```

In the shell, the colon (`:`) is a no-op command that always returns true, and can be used instead of `true`. An alternative implementation that works well here is:

```
$(config): $(config_template)
        [[ -d $(dir $@) ]] || $(MKDIR) $(dir $@)
        $(M4) $^ > $@
```

Now the first statement is true when the directory exists or when the `mkdir` succeeds. Another alternative is to use `mkdir -p`. This allows `mkdir` to succeed even when the directory already exists. All these implementations execute something in a subshell even when the directory exists. By using `wildcard`, we can omit the execution entirely if the directory is present.

```
# $(call make-dir, directory)
make-dir = $(if $(wildcard $1),,$(MKDIR) -p $1)

$(config): $(config_template)
        $(call make-dir, $(dir $@))
        $(M4) $^ > $@
```

Because each command is executed in its own shell, it is common to have multiline commands with each component separated by semicolons. Be aware that errors within these scripts may not terminate the script:

```
target:
        rm rm-fails; echo But the next command executes anyway
```

It is best to minimize the length of command scripts and give `make` a chance to manage exit status and termination for you. For instance:

```
path-fixup = -e "s;[a-zA-Z:/]*/src/;$(SOURCE_DIR)/;g" \
             -e "s;[a-zA-Z:/]*/bin/;$(OUTPUT_DIR)/;g"

# A good version.
define fix-project-paths
  sed $(path-fixup) $1 > $2.fixed && \
  mv $2.fixed $2
endef

# A better version.
define fix-project-paths
  sed $(path-fixup) $1 > $2.fixed
  mv $2.fixed $2
endef
```

This macro transforms DOS-style paths (with forward slashes) into destination paths for a particular source and output tree. The macro accepts two filenames, the input and output files. It is careful to overwrite the output file only if the `sed` command completes correctly. The "good" version does this by connecting the `sed` and `mv` with

`&&` so they execute in a single shell. The "better" version executes them as two separate commands, letting `make` terminate the script if the `sed` fails. The "better" version is no more expensive (the `mv` doesn't need a shell and is executed directly), is easier to read, and provides more information when errors occur (because `make` will indicate which command failed).

Note that this is a different issue than the common problem with `cd`:

```
TAGS:
        cd src && \
        ctags --recurse
```

In this case, the two statements must be executed within the same subshell. Therefore, the commands must be separated by some kind of statement connector, such as `;` or `&&`.

Deleting and preserving target files

If an error occurs, `make` assumes that the target cannot be remade. Any other targets that have the current target as a prerequisite also cannot be remade, so `make` will not attempt them nor execute any part of their command scripts. If the `--keep-going` (or `-k`) option is used, the next goal will be attempted; otherwise, `make` exits. If the current target is a file, it may be corrupt if the command exits before finishing its work. Unfortunately, for reasons of historical compatibility, `make` will leave this potentially corrupt file on disk. Because the file's timestamp has been updated, subsequent executions of `make` may not update the file with correct data. You can avoid this problem and cause `make` to delete these questionable files when an error occurs by making the target file a prerequisite of `.DELETE_ON_ERROR`. If `.DELETE_ON_ERROR` is used with no prerequisites, errors in any target file build will cause `make` to delete the target.

A complementary problem occurs when `make` is interrupted by a signal, such as a Ctrl-C. In this case, `make` deletes the current target file if the file has been modified. Sometimes deleting the file is the wrong thing to do. Perhaps the file is very expensive to create and partial contents are better than none, or perhaps the file must exist for other parts of the build to proceed. In these cases, you can protect the file by making it a prerequisite of the special target `.PRECIOUS`.

Which Shell to Use

When `make` needs to pass a command line to a subshell, it uses */bin/sh*. You can change the shell by setting the `make` variable `SHELL`. Think carefully before doing this. Usually, the purpose of using `make` is to provide a tool for a community of developers to build a system from its source components. It is quite easy to create a *makefile* that fails in this goal by using tools that are not available or assumptions that are not true for other developers in the community. It is considered very bad form to use any shell other than */bin/sh* in any widely distributed application (one distributed via anonymous ftp or open cvs). We'll discuss portability in more detail in Chapter 7.

There is another context for using `make`, however. Often, in closed development environments, the developers are working on a limited set of machines and operating systems with an approved group of developers. In fact, this is the environment I've most often found myself in. In this situation, it can make perfect sense to customize the environment `make` is expected to run under. Developers are instructed in how to set up their environment to work properly with the build and life goes on.

In environments such as this, I prefer to make some portability sacrifices "up front." I believe this can make the entire development process go much more smoothly. One such sacrifice is to explicitly set the `SHELL` variable to */usr/bin/bash*. The `bash` shell is a portable, POSIX-compliant shell (and, therefore, a superset of `sh`) and is the standard shell on GNU/Linux. Many portability problems in *makefile*s are due to using nonportable constructs in command scripts. This can be solved by explicitly using one standard shell rather than writing to the portable subset of `sh`. Paul Smith, the maintainer of GNU `make`, has a web page "Paul's Rules of Makefiles" (*http://make.paulandlesley.org/rules.html*) on which he states, "Don't hassle with writing portable makefiles, use a portable *make* instead!" I would also say, "Where possible, don't hassle with writing portable command scripts, use a portable shell (bash) instead." The `bash` shell runs on most operating systems including virtually all variants of Unix, Windows, BeOS, Amiga, and OS/2.

For the remainder of this book, I will note when a command script uses bash-specific features.

Empty Commands

An *empty command* is one that does nothing.

```
header.h: ;
```

Recall that the prerequisites list for a target can be followed by a semicolon and the command. Here a semicolon with nothing after it indicates that there are no commands. You could instead follow the target with a line containing only a tab, but that would be impossible to read. Empty commands are most often used to prevent a pattern rule from matching the target and executing commands you don't want.

Note that in other versions of `make`, empty targets are sometimes used as phony targets. In GNU `make`, use the `.PHONY` special target instead; it's safer and clearer.

Command Environment

Commands executed by make inherit their processing environment from make itself. This environment includes the current working directory, file descriptors, and the environment variables passed by make.

When a subshell is created, make adds a few variables to the environment:

```
MAKEFLAGS
MFLAGS
MAKELEVEL
```

The MAKEFLAGS variable includes the command-line options passed to make. The MFLAGS variable mirrors MAKEFLAGS and exists for historical reasons. The MAKELEVEL variable indicates the number of nested make invocations. That is, when make recursively invokes make, the MAKELEVEL variable increases by one. Subprocesses of a single parent make will have a MAKELEVEL of one. These variables are typically used for managing recursive make. We'll discuss them in the section "Recursive make" in Chapter 6.

Of course, the user can add whatever variables they like to the subprocess environment with the use of the export directive.

The current working directory for an executed command is the working directory of the parent make. This is typically the same as the directory the make program was executed from, but can be changed with the the --directory=<replaceable>directory</replaceable> (or -C) command-line option. Note that simply specifying a different *makefile* using --file does not change the current directory, only the *makefile* read.

Each subprocess make spawns inherits the three standard file descriptors: *stdin*, *stdout*, and *stderr*. This is not particularly noteworthy except to observe that it is possible for a command script to read its *stdin*. This is "reasonable" and works. Once the script completes its read, the remaining commands are executed as expected. But *makefile*s are generally expected to run without this kind of interaction. Users often expect to be able to start a make and "walk away" from the process, returning later to examine the results. Of course, reading the *stdin* will also tend to interact poorly with cron-based automated builds.

A common error in *makefile*s is to read the *stdin* accidentally:

```
$(DATA_FILE): $(RAW_DATA)
        grep pattern $(RAW_DATA_FILES) > $@
```

Here the input file to grep is specified with a variable (misspelled in this example). If the variable expands to nothing, the grep is left to read the *stdin* with no prompt or indication of why the make is "hanging." A simple way around this issue is to always include */dev/null* on the command line as an additional "file":

```
$(DATA_FILE): $(RAW_DATA)
        grep pattern $(RAW_DATA_FILES) /dev/null > $@
```

This grep command will never attempt to read *stdin*. Of course, debugging the *makefile* is also appropriate!

Evaluating Commands

Command script processing occurs in four steps: read the code, expand variables, evaluate make expressions, and execute commands. Let's see how these steps apply to a complex command script. Consider this (somewhat contrived) *makefile*. An application is linked, then optionally stripped of symbols and compressed using the upx executable packer:

```
# $(call strip-program, file)
define strip-program
  strip $1
endef

complex_script:
        $(CC) $^ -o $@
    ifdef STRIP
        $(call strip-program, $@)
    endif
        $(if $(PACK), upx --best $@)
        $(warning Final size: $(shell ls -s $@))
```

The evaluation of command scripts is deferred until they are executed, but ifdef directives are processed immediately wherever they occur. Therefore, make reads the command script, ignoring the content and storing each line until it gets to the line ifdef STRIP. It evaluates the test and, if STRIP is not defined, make reads and discards all the text up to and including the closing endif. make then continues reading and storing the rest of the script.

When a command script is to be executed, make first scans the script for make constructs that need to be expanded or evaluated. When macros are expanded, a leading tab is prepended to each line. Expanding and evaluating before any commands are executed can lead to an unexpected execution order if you aren't prepared for it. In our example, the last line of the script is wrong. The shell and warning commands are executed *before* linking the application. Therefore, the ls command will be executed before the file it is examining has been updated. This explains the "out of order" output seen earlier in the section "Parsing Commands."

Also, notice that the ifdef STRIP line is evaluated while reading the file, but the $(if...) line is evaluated immediately before the commands for complex_script are executed. Using the if function is more flexible since there are more opportunities to control when the variable is defined, but it is not very well suited for managing large blocks of text.

As this example shows, it is important to always attend to what program is evaluating an expression (e.g., make or the shell) and when the evaluation is performed:

```
$(LINK.c) $(shell find $(if $(ALL),$(wildcard core ext*),core) -name '*.o')
```

This convoluted command script attempts to link a set of object files. The sequence of evaluation and the program performing the operation (in parentheses) is:

1. Expand `$ALL` (`make`).
2. Evaluate `if` (`make`).
3. Evaluate the `wildcard`, assuming `ALL` is not empty (`make`).
4. Evaluate the `shell` (`make`).
5. Execute the `find` (`sh`).
6. After completing the expansion and evaluation of the `make` constructs, execute the link command (`sh`).

Command-Line Limits

When working with large projects, you occasionally bump up against limitations in the length of commands `make` tries to execute. Command-line limits vary widely with the operating system. Red Hat 9 GNU/Linux appears to have a limit of about 128K characters, while Windows XP has a limit of 32K. The error message generated also varies. On Windows using the Cygwin port, the message is:

```
C:\usr\cygwin\bin\bash: /usr/bin/ls: Invalid argument
```

when `ls` is given too long an argument list. On Red Hat 9 the message is:

```
/bin/ls: argument list too long
```

Even 32K sounds like a lot of data for a command line, but when your project contains 3,000 files in 100 subdirectories and you want to manipulate them all, this limit can be constraining.

There are two basic ways to get yourself into this mess: expand some basic value using shell tools, or use `make` itself to set a variable to a very long value. For example, suppose we want to compile all our source files in a single command line:

```
compile_all:
        $(JAVAC) $(wildcard $(addsuffix /*.java,$(source_dirs)))
```

The `make` variable `source_dirs` may contain only a couple hundred words, but after appending the wildcard for Java files and expanding it using `wildcard`, this list can easily exceed the command-line limit of the system. By the way, `make` has no built-in limits to constrain us. So long as there is virtual memory available, `make` will allow any amount of data you care to create.

When you find yourself in this situation, it can feel like the old Adventure game, "You are in a twisty maze of passages all alike." For instance, you might try to solve the above using `xargs`, since `xargs` will manage long command lines by parceling out arguments up to the system-specific length:

```
compile_all:
        echo $(wildcard $(addsuffix /*.java,$(source_dirs))) | \
        xargs $(JAVAC)
```

Unfortunately, we've just moved the command-line limit problem from the `javac` command line to the `echo` command line. Similarly, we cannot use `echo` or `printf` to write the data to a file (assuming the compiler can read the file list from a file).

No, the way to handle this situation is to avoid creating the file list all at once in the first place. Instead, use the shell to glob one directory at a time:

```
compile_all:
        for d in $(source_dirs); \
        do                       \
            $(JAVAC) $$d/*.java; \
        done
```

We could also pipe the file list to `xargs` to perform the task with fewer executions:

```
compile_all:
        for d in $(source_dirs); \
        do                       \
            echo $$d/*.java;     \
        done |                   \
        xargs $(JAVAC)
```

Sadly, neither of these command scripts handle errors during compilation properly. A better approach would be to save the full file list and feed it to the compiler, if the compiler supports reading its arguments from a file. Java compilers support this feature:

```
compile_all: $(FILE_LIST)
        $(JAVA) @$<

.INTERMEDIATE: $(FILE_LIST)
$(FILE_LIST):
        for d in $(source_dirs); \
        do                       \
            echo $$d/*.java;     \
        done > $@
```

Notice the subtle error in the `for` loop. If any of the directories does not contain a Java file, the string `*.java` will be included in the file list and the Java compiler will generate a "File not found" error. We can make `bash` collapse empty globbing patterns by setting the `nullglob` option.

```
compile_all: $(FILE_LIST)
        $(JAVA) @$<

.INTERMEDIATE: $(FILE_LIST)
$(FILE_LIST):
        shopt -s nullglob;       \
        for d in $(source_dirs); \
        do                       \
            echo $$d/*.java;     \
        done > $@
```

Many projects have to make lists of files. Here is a macro containing a `bash` script producing file lists. The first argument is the root directory to change to. All the files in the list will be relative to this root directory. The second argument is a list of directories to search for matching files. The third and fourth arguments are optional and represent file suffixes.

```
# $(call collect-names, root-dir, dir-list, suffix1-opt, suffix2-opt)
define collect-names
  echo Making $@ from directory list...
  cd $1;                                                  \
  shopt -s nullglob;                                      \
  for f in $(foreach file,$2,'$(file)'); do               \
    files=( $$f$(if $3,/*.{$3$(if $4,$(comma)$4)}) );     \
    if (( $${#files[@]} > 0 ));                           \
    then                                                  \
      printf '"%s"\n' $${files[@]};                       \
    else :; fi;                                           \
  done
endef
```

Here is a pattern rule for creating a list of image files:

```
%.images:
        @$(call collect-names,$(SOURCE_DIR),$^,gif,jpeg) > $@
```

The macro execution is hidden because the script is long and there is seldom a reason to cut and paste this code. The directory list is provided in the prerequisites. After changing to the root directory, the script enables null globbing. The rest is a `for` loop to process each directory we want to search. The file search expression is a list of words passed in parameter `$2`. The script protects words in the file list with single quotes because they may contain shell-special characters. In particular, filenames in languages like Java can contain dollar signs:

```
for f in $(foreach file,$2,'$(file)'); do
```

We search a directory by filling the `files` array with the result of globbing. If the `files` array contains any elements, we use `printf` to write each word followed by a newline. Using the array allows the macro to properly handle paths with embedded spaces. This is also the reason `printf` surrounds the filename with double quotes.

The file list is produced with the line:

```
files=( $$f$(if $3,/*.{$3$(if $4,$(comma)$4)}) );
```

The `$$f` is the directory or file argument to the macro. The following expression is a `make` `if` testing whether the third argument is nonempty. This is how you can implement optional arguments. If the third argument is empty, it is assumed the fourth is as well. In this case, the file passed by the user should be included in the file list as is. This allows the macro to build lists of arbitrary files for which wildcard patterns are inappropriate. If the third argument is provided, the `if` appends `/*.{$3}` to the root file. If the fourth argument is provided, it appends `,$4` after the `$3`. Notice the subterfuge we must use to insert a comma into the wildcard pattern. By placing a comma in a `make` variable we can sneak it past the parser, otherwise, the comma would be interpreted as separating the *then* part from the *else* part of the if. The definition of comma is straightforward:

```
comma := ,
```

All the preceding for loops also suffer from the command-line length limit, since they use wildcard expansion. The difference is that the wildcard is expanded with the contents of a single directory, which is far less likely to exceed the limits.

What do we do if a make variable contains our long file list? Well, then we are in real trouble. There are only two ways I've found to pass a very long make variable to a subshell. The first approach is to pass only a subset of the variable contents to any one subshell invocation by filtering the contents.

```
compile_all:
        $(JAVAC) $(wordlist 1, 499, $(all-source-files))
        $(JAVAC) $(wordlist 500, 999, $(all-source-files))
        $(JAVAC) $(wordlist 1000, 1499, $(all-source-files))
```

The filter function can be used as well, but that can be more uncertain since the number of files selected will depend on the distribution within the pattern space chosen. Here we choose a pattern based on the alphabet:

```
compile_all:
        $(JAVAC) $(filter a%, $(all-source-files))
        $(JAVAC) $(filter b%, $(all-source-files))
```

Other patterns might use special characteristics of the filenames themselves.

Notice that it is difficult to automate this further. We could try to wrap the alphabet approach in a foreach loop:

```
compile_all:
        $(foreach l,a b c d e ...,                          \
          $(if $(filter $l%, $(all-source-files)),          \
            $(JAVAC) $(filter $l%, $(all-source-files));))
```

but this doesn't work. make expands this into a single line of text, thus compounding the line-length problem. We can instead use eval:

```
compile_all:
        $(foreach l,a b c d e ...,                  \
          $(if $(filter $l%, $(all-source-files)),  \
            $(eval                                  \
              $(shell                               \
                $(JAVAC) $(filter $l%, $(all-source-files));))))
```

This works because eval will execute the shell command immediately, expanding to nothing. So the foreach loop expands to nothing. The problem is that error reporting is meaningless in this context, so compilation errors will not be transmitted to make correctly.

The wordlist approach is worse. Due to make's limited numerical capabilities, there is no way to enclose the wordlist technique in a loop. In general, there are very few satisfying ways to deal with immense file lists.