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.

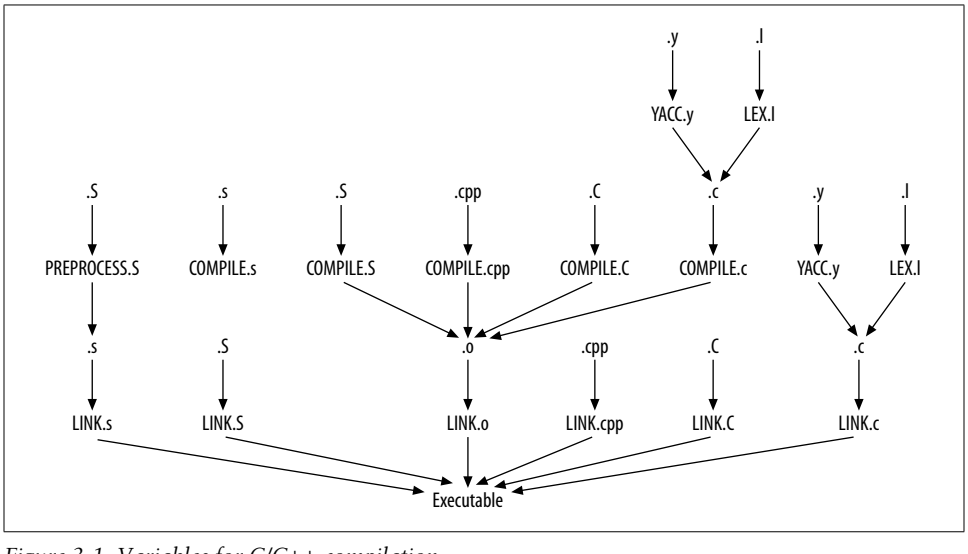

Figure 3-1. Variables for C/C++ compilation

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 TEX and has its own set of rules. Recursive make is another feature supported by variables. We'll discuss this topic in Chapter 6.