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.