CHAPTER 5

Commands

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

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

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

Parsing Commands

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Continuing Long Commands

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

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

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

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

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

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

which generates the error:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Command Modifiers

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

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

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

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

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

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

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

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

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

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

Errors and Interrupts

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Deleting and preserving target files

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

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

Which Shell to Use

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

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

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

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

Empty Commands

An *empty command* is one that does nothing.

```
header.h: ;
```

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

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

Command Environment

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

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

```
MAKEFLAGS
MFLAGS
MAKELEVEL
```

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

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

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

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

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

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

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

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

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

Evaluating Commands

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

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

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

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

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

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

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

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

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

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

Command-Line Limits

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The file list is produced with the line:

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

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

```
comma := ,
```

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

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

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

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

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

Other patterns might use special characteristics of the filenames themselves.

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

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

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

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

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

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