CHAPTER 12

Debugging Makefiles

Debugging *makefile*s is somewhat of a black art. Unfortunately, there is no such thing as a *makefile* debugger to examine how a particular rule is being evaluated or a variable expanded. Instead, most debugging is performed with simple print statements and by inspection of the *makefile*. GNU `make` provides some help with various built-in functions and command-line options.

One of the best ways to debug a *makefile* is to add debugging hooks and use defensive programming techniques that you can fall back on when things go awry. I'll present a few basic debugging techniques and defensive coding practices I've found most helpful.

Debugging Features of make

The `warning` function is very useful for debugging wayward *makefile*s. Because the `warning` function expands to the empty string, it can be placed anywhere in a *makefile*: at the top-level, in target or prerequisite lists, and in command scripts. This allows you to print the value of variables wherever it is most convenient to inspect them. For example:

```
$(warning A top-level warning)

FOO := $(warning Right-hand side of a simple variable)bar
BAZ = $(warning Right-hand side of a recursive variable)boo

$(warning A target)target: $(warning In a prerequisite list)makefile $(BAZ)
        $(warning In a command script)
        ls
$(BAZ):
```

yields the output:

```
$ make
makefile:1: A top-level warning
makefile:2: Right-hand side of a simple variable
makefile:5: A target
```

```
makefile:5: In a prerequisite list
makefile:5: Right-hand side of a recursive variable
makefile:8: Right-hand side of a recursive variable
makefile:6: In a command script
ls
makefile
```

Notice that the evaluation of the `warning` function follows the normal `make` algorithm for immediate and deferred evaluation. Although the assignment to `BAZ` contains a `warning`, the message does not print until `BAZ` is evaluated in the prerequisites list.

The ability to inject a `warning` call anywhere makes it an essential debugging tool.

Command-Line Options

There are three command-line options I find most useful for debugging: `--just-print` (`-n`), `--print-data-base` (`-p`), and `--warn-undefined-variables`.

--just-print

The first test I perform on a new *makefile* target is to invoke `make` with the `--just-print` (`-n`) option. This causes `make` to read the *makefile* and print every command it would normally execute to update the target but without executing them. As a convenience, GNU `make` will also echo commands marked with the silent modifier (`@`).

The option is supposed to suppress all command execution. While this may be true in one sense, practically speaking, you must take care. While `make` will not execute command scripts, it will evaluate `shell` function calls that occur within an immediate context. For instance:

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

$(objects) : $(sources)
```

As we've seen before, the purpose of the `_MKDIRS` simple variable is to trigger the creation of essential directories. When this is executed with `--just-print`, the shell command will be executed as usual when the *makefile* is read. Then `make` will echo (without executing) each compilation command required to update the `$(objects)` file list.

--print-data-base

The `--print-data-base` (`-p`) option is another one you'll use often. It executes the *makefile*, displaying the GNU copyright followed by the commands as they are run by `make`, then it will dump its internal database. The data is collected into groups of

values: variables, directories, implicit rules, pattern-specific variables, files (explicit rules), and the vpath search path:

```
# GNU Make 3.80
# Copyright (C) 2002  Free Software Foundation, Inc.
# This is free software; see the source for copying conditions.
# There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
# PARTICULAR PURPOSE.
normal command execution occurs here

# Make data base, printed on Thu Apr 29 20:58:13 2004

# Variables
…
# Directories
…
# Implicit Rules
…
# Pattern-specific variable values
…
# Files
…
# VPATH Search Paths
```

Let's examine these sections in more detail.

The variables section lists each variable along with a descriptive comment:

```
# automatic
<D = $(patsubst %/,%,$(dir $<))
# environment
EMACS_DIR = C:/usr/emacs-21.3.50.7
# default
CWEAVE = cweave
# makefile (from `../mp3_player/makefile', line 35)
CPPFLAGS = $(addprefix -I ,$(include_dirs))
# makefile (from `../ch07-separate-binaries/makefile', line 44)
RM := rm -f
# makefile (from `../mp3_player/makefile', line 14)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
        $(AR) $(ARFLAGS) $$@ $$^
endef
```

Automatic variables are not printed, but convenience variables derived from them like `$(<D)` are. The comment indicates the type of the variable as returned by the `origin` function (see the section "Less Important Miscellaneous Functions" in Chapter 4). If the variable is defined in a file, the filename and line number of the definition is given. Simple and recursive variables are distinguished by the

assignment operator. The value of a simple variable will be displayed as the evaluated form of the righthand side.

The next section, labeled Directories, is more useful to make developers than to make users. It lists the directories being examined by make, including SCCS and RCS subdirectories that might exist, but usually do not. For each directory, make displays implementation details, such as the device number, inode, and statistics on file pattern matches.

The Implicit Rules section follows. This contains all the built-in and user-defined pattern rules in make's database. Again, for those rules defined in a file, a comment indicates the file and line number:

```
%.c %.h: %.y
# commands to execute (from `../mp3_player/makefile', line 73):
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

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

%.o: %.c
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Examining this section is a great way to become familiar with the variety and structure of make's built-in rules. Of course, not all built-in rules are implemented as pattern rules. If you don't find the rule you're looking for, check in the Files section where the old-style suffix rules are listed.

The next section catalogs the pattern-specific variables defined in the *makefile*. Recall that pattern-specific variables are variable definitions whose scope is precisely the execution time of their associated pattern rule. For example, the pattern variable YYLEXFLAG, defined as:

```
%.c %.h: YYLEXFLAG := -d
%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h
```

would be displayed as:

```
# Pattern-specific variable values

%.c :
# makefile (from `Makefile', line 1)
# YYLEXFLAG := -d
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%
```

```
%.h :
# makefile (from `Makefile', line 1)
# YYLEXFLAG := -d
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%

# 2 pattern-specific variable values
```

The Files section follows and lists all the explicit and suffix rules that relate to specific files:

```
# Not a target:
.p.o:
#  Implicit rule search has not been done.
#  Modification time never checked.
#  File has not been updated.
#  commands to execute (built-in):
        $(COMPILE.p) $(OUTPUT_OPTION) $<

lib/ui/libui.a: lib/ui/ui.o
#  Implicit rule search has not been done.
#  Last modified 2004-04-01 22:04:09.515625
#  File has been updated.
#  Successfully updated.
#  commands to execute (from `../mp3_player/lib/ui/module.mk', line 3):
        ar rv $@ $^

lib/codec/codec.o: ../mp3_player/lib/codec/codec.c ../mp3_player/lib/codec/codec.c ..
/mp3_player/include/codec/codec.h
#  Implicit rule search has been done.
#  Implicit/static pattern stem: `lib/codec/codec'
#  Last modified 2004-04-01 22:04:08.40625
#  File has been updated.
#  Successfully updated.
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Intermediate files and suffix rules are labeled "Not a target"; the remainder are targets. Each file includes comments indicating how make has processed the rule. Files that are found through the normal vpath search have their resolved path displayed.

The last section is labeled VPATH Search Paths and lists the value of VPATH and all the vpath patterns.

For *makefile*s that make extensive use of user-defined functions and eval to create complex variables and rules, examining this output is often the only way to verify that macro expansion has generated the expected values.

--warn-undefined-variables

This option causes make to display a warning whenever an undefined variable is expanded. Since undefined variables expand to the empty string, it is common for typographical errors in variable names to go undetected for long periods. The problem

with this option, and why I use it only rarely, is that many built-in rules include undefined variables as hooks for user-defined values. So running make with this option will inevitably produce many warnings that are not errors and have no useful relationship to the user's *makefile*. For example:

```
$ make --warn-undefined-variables -n
makefile:35: warning: undefined variable MAKECMDGOALS
makefile:45: warning: undefined variable CFLAGS
makefile:45: warning: undefined variable TARGET_ARCH
...
makefile:35: warning: undefined variable MAKECMDGOALS
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
...
make: warning: undefined variable LDFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable LOADLIBES
make: warning: undefined variable LDLIBS
```

Nevertheless, this command can be extremely valuable on occasion in catching these kinds of errors.

The --debug Option

When you need to know how make analyzes your dependency graph, use the `--debug` option. This provides the most detailed information available other than by running a debugger. There are five debugging options and one modifier: `basic`, `verbose`, `implicit`, `jobs`, `all`, and `makefile`, respectively.

If the debugging option is specified as `--debug`, basic debugging is used. If the debugging option is given as `-d`, `all` is used. To select other combinations of options, use a comma separated list `--debug=`*`option1`*`,`*`option2`* where the option can be one of the following words (actually, make looks only at the first letter):

`basic`
: Basic debugging is the least detailed. When enabled, make prints each target that is found to be out-of-date and the status of the update action. Sample output looks like:

```
File all does not exist.
  File app/player/play_mp3 does not exist.
    File app/player/play_mp3.o does not exist.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

`verbose`
: This option sets the `basic` option and includes additional information about which files where parsed, prerequisites that did not need to be rebuilt, etc.:

```
File all does not exist.
 Considering target file app/player/play_mp3.
  File app/player/play_mp3 does not exist.
   Considering target file app/player/play_mp3.o.
    File app/player/play_mp3.o does not exist.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/include/player/play_mp3.h.
    Finished prerequisites of target file app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
   Pruning file app/player/play_mp3.o.
```

`implicit`

This option sets the `basic` option and includes additional information about implicit rule searches for each target:

```
File all does not exist.
  File app/player/play_mp3 does not exist.
  Looking for an implicit rule for app/player/play_mp3.
  Trying pattern rule with stem play_mp3.
  Trying implicit prerequisite app/player/play_mp3.o.
  Found an implicit rule for app/player/play_mp3.
    File app/player/play_mp3.o does not exist.
    Looking for an implicit rule for app/player/play_mp3.o.
    Trying pattern rule with stem play_mp3.
    Trying implicit prerequisite app/player/play_mp3.c.
    Found prerequisite app/player/play_mp3.c as VPATH ../mp3_player/app/player/
play_mp3.c
    Found an implicit rule for app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

`jobs`

This options prints the details of subprocesses invoked by `make`. It does not enable the `basic` option.

```
Got a SIGCHLD; 1 unreaped children.
gcc ... ../mp3_player/app/player/play_mp3.c
Putting child 0x10033800 (app/player/play_mp3.o) PID 576 on the chain.
Live child 0x10033800 (app/player/play_mp3.o) PID 576
Got a SIGCHLD; 1 unreaped children.
Reaping winning child 0x10033800 PID 576
Removing child 0x10033800 PID 576 from chain.
```

`all`

This enables all the previous options and is the default when using the `-d` option.

`makefile`

Normally, debugging information is not enabled until after the *makefile*s have been updated. This includes updating any included files, such as lists of dependencies. When you use this modifier, `make` will print the selected information

while rebuilding *makefile*s and include files. This option enables the `basic` option and is also enabled by the `all` option.

Writing Code for Debugging

As you can see, there aren't too many tools for debugging *makefile*s, just a few ways to dump `make`'s internal data structures and a couple of print statements. When it comes right down to it, it is up to you to write your *makefile*s in ways that either minimize the chance of errors or provide your own scaffolding to help debug them.

The suggestions in this section are laid out somewhat arbitrarily as coding practices, defensive coding, and debugging techniques. While specific items, such as checking the exit status of commands, could be placed in either the good coding practice section or the defensive coding section, the three categories reflect the proper bias. Focus on coding your *makefile*s well without cutting too many corners. Include plenty of defensive coding to protect the *makefile* against unexpected events and environmental conditions. Finally, when bugs do arise, use every trick you can find to squash them.

The "Keep It Simple" Principle (*http://www.catb.org/~esr/jargon/html/K/KISS-Principle.html*) is at the heart of all good design. As you've seen in previous chapters, *makefile*s can quickly become complex, even for mundane tasks, such as dependency generation. Fight the tendency to include more and more features in your build system. You'll fail, but not as badly as you would if you simply include every feature that occurs to you.

Good Coding Practices

In my experience, most programmers do not see writing *makefile*s as programming and, therefore, do not take the same care as they do when writing in C++ or Java. But the `make` language is a complete nonprocedural language. If the reliability and maintainability of your build system is important, write it with care and use the best coding practices you can.

One of the most important aspects of programming robust *makefile*s is to check the return status of commands. Of course, `make` will check simple commands automatically, but *makefile*s often include compound commands that can fail quietly:

```
do:
        cd i-dont-exist; \
        echo *.c
```

When run, this *makefile* does not terminate with an error status, although an error most definitely occurs:

```
$ make
cd i-dont-exist; \
echo *.c
```

```
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
*.c
```

Furthermore, the globbing expression fails to find any *.c* files, so it quietly returns the globbing expression. Oops. A better way to code this command script is to use the shell's features for checking and preventing errors:

```
SHELL = /bin/bash
do:
        cd i-dont-exist && \
        shopt -s nullglob &&
        echo *.c
```

Now the `cd` error is properly transmitted to `make`, the `echo` command never executes, and `make` terminates with an error status. In addition, setting the `nullglob` option of `bash` causes the globbing pattern to return the empty string if no files are found. (Of course, your particular application may prefer the globbing pattern.)

```
$ make
cd i-dont-exist && \
echo *.c
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
make: *** [do] Error 1
```

Another good coding practice is formatting your code for maximum readability. Most *makefile*s I see are poorly formatted and, consequently, difficult to read. Which do you find easier to read?

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

or:

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

If you're like most people, you'll find the first more difficult to parse, the semicolons harder to find, and the number of statements more difficult to count. These are not trivial concerns. A significant percentage of the syntax errors you will encounter in command scripts will be due to missing semicolons, backslashes, or other separators, such as pipe and logical operators.

Also, note that not all missing separators will generate an error. For instance, neither of the following errors will produce a shell syntax error:

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

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

Formatting commands for readability will make these kinds of errors easier to catch. When formatting user-defined functions, indent the code. Occasionally, the extra spaces in the resulting macro expansion cause problems. If so, wrap the formatting in a `strip` function call. When formatting long lists of values, separate each value on its own line. Add a comment before each target, give a brief explanation, and document the parameter list.

The next good coding practice is the liberal use of variables to hold common values. As in any program, the unrestrained use of literal values creates code duplication and leads to maintenance problems and bugs. Another great advantage of variables is that you can get `make` to display them for debugging purposes during execution. I show a nice command line interface in the section "Debugging Techniques," later in this chapter.

Defensive Coding

Defensive code is code that can execute only if one of your assumptions or expectations is wrong — an `if` test that is never true, an `assert` function that never fails, or tracing code. Of course, the value of this code that never executes is that occasionally (usually when you least expect it), it does run and produce a warning or error, or you choose to enable tracing code to allow you to view the inner workings of `make`.

You've already seen most of this code in other contexts, but for convenience it is repeated here.

Validation checking is a great example of defensive code. This code sample verifies that the currently executing version of `make` is 3.80:

```
NEED_VERSION := 3.80
$(if $(filter $(NEED_VERSION),$(MAKE_VERSION)),,              \
  $(error You must be running make version $(NEED_VERSION).))
```

For Java applications, it is useful to include a check for files in the `CLASSPATH`.

Validation code can also simply ensure that something is true. The directory creation code from the previous section is of this nature.

Another great defensive coding technique is to use the `assert` functions defined in the section "Flow Control" in Chapter 4. Here are several versions:

```
# $(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
```

I find sprinkling `assert` calls around the *makefile* to be a cheap and effective way of detecting missing and misspelled parameters as well as violations of other assumptions.

In Chapter 4, we wrote a pair of functions to trace the expansion of user-defined functions:

```
# $(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)'))
```

You can add these macro calls to your own functions and leave them disabled until they are required for debugging. To enable them, set `debug_trace` to any nonempty value:

```
$ make debug_trace=1
```

As noted in Chapter 4, these trace macros have a number of problems of their own but can still be useful in tracking down bugs.

The final defensive programming technique is simply to make disabling the `@` command modifier easy by using it through a `make` variable, rather than literally:

```
QUIET := @
…
target:
        $(QUIET) some command
```

Using this technique, you can see the execution of the silent command by redefining `QUIET` on the command line:

```
$ make QUIET=
```

Debugging Techniques

This section discusses general debugging techniques and issues. Ultimately, debugging is a grab-bag of whatever works for your situation. These techniques have worked for me, and I've come to rely on them to debug even the simplest *makefile* problems. Maybe they'll help you, too.

One of the very annoying bugs in 3.80 is that when `make` reports problems in *makefile*s and includes a line number, I usually find that the line number is wrong. I haven't investigated whether the problem is due to include files, multiline variable

assignments, or user-defined macros, but there it is. Usually the line number make reports is larger than the actual line number. In complex *makefile*s, I've had the line number be off by as much as 20 lines.

Often the easiest way to see the value of a make variable is to print it during the execution of a target. Although adding print statements using warning is simple, the extra effort of adding a generic debug target for printing variables can save lots of time in the long run. Here is a sample debug target:

```
debug:
        $(for v,$(V), \
          $(warning $v = $($v)))
```

To use it, just set the list of variables to print on the command line, and include the debug target:

```
$ make V="USERNAME SHELL" debug
makefile:2: USERNAME = Owner
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
```

If you want to get really tricky, you can use the MAKECMDGOALS variable to avoid the assignment to the variable V:

```
debug:
        $(for v,$(V) $(MAKECMDGOALS), \
          $(if $(filter debug,$v),,$(warning $v = $($v))))
```

Now you can print variables by simply listing them on the command line. I don't recommend this technique, though, because you'll also get confusing make warnings indicating it doesn't know how to update the variables (since they are listed as targets):

```
$ make debug PATH SHELL
makefile:2: USERNAME = Owner
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
make: *** No rule to make target USERNAME.  Stop.
```

In Chapter 10, I briefly mentioned using a debugging shell to help understand some of the activities make performs behind the scenes. While make echos commands in command scripts before they are executed, it does not echo the commands executed in shell functions. Often these commands are subtle and complex, particularly since they may be executed immediately or in a deferred fashion, if they occur in a recursive variable assignment. One way to see these commands execute is to request that the subshell enable debug printing:

```
DATE := $(shell date +%F)
OUTPUT_DIR = out-$(DATE)

make-directories := $(shell [ -d $(OUTPUT_DIR) ] || mkdir -p $(OUTPUT_DIR))

all: ;
```

When run with sh's debugging option, we see:

```
$ make SHELL="sh -x"
+ date +%F
+ '[' -d out-2004-05-11 ']'
+ mkdir -p out-2004-05-11
```

This even provides additional debugging information beyond make warning statements, since with this option the shell also displays the value of variables and expressions.

Many of the examples in this book are written as deeply nested expressions, such as this one that checks the PATH variable on a Windows/Cygwin system:

```
$(if $(findstring /bin/,                                  \
       $(firstword                                        \
         $(wildcard                                       \
           $(addsuffix /sort$(if $(COMSPEC),.exe),        \
             $(subst :, ,$(PATH))))),,                    \
  $(error Your PATH is wrong, c:/usr/cygwin/bin should    \
    precede c:/WINDOWS/system32))
```

There is no good way to debug these expressions. One reasonable approach is to unroll them, and print each subexpression:

```
$(warning $(subst :, ,$(PATH)))
$(warning /sort$(if $(COMSPEC),.exe))
$(warning $(addsuffix /sort$(if $(COMSPEC),.exe),     \
             $(subst :, ,$(PATH))))
$(warning $(wildcard                                   \
            $(addsuffix /sort$(if $(COMSPEC),.exe),    \
              $(subst :, ,$(PATH)))))
```

Although a bit tedious, without a real debugger, this is the best way (and sometimes the only way) to determine the value of various subexpressions.

Common Error Messages

The 3.81 GNU make manual includes an excellent section listing make error messages and their causes. We review a few of the most common ones here. Some of the issues described are not strictly make errors, such as syntax errors in command scripts, but are nonetheless common problems for developers. For a complete list of make errors, see the make manual.

Error messages printed by make have a standard format:

```
makefile:n: *** message.  Stop.
```

or:

```
make:n: *** message.  Stop.
```

The *makefile* part is the name of the *makefile* or include file in which the error occurred. The next part is the line number where the error occurred, followed by three asterisks and, finally, the error message.

Note that it is make's job to run other programs and that, if errors occur, it is very likely that problems in your *makefile* will manifest themselves as errors in these other programs. For instance, shell errors may result from badly formed command scripts, or compiler errors from incorrect command-line arguments. Figuring out what program produced the error message is your first task in solving the problem. Fortunately, make's error messages are fairly self-evident.

Syntax Errors

These are usually typographical errors: missing parentheses, using spaces instead of tabs, etc.

One of the most common errors for new make users is omitting parentheses around variable names:

```
foo:
        for f in $SOURCES; \
        do                 \
           …               \
        done
```

This will likely result in make expanding $S to nothing, and the shell executing the loop only once with f having a value of OURCES. Depending on what you do with f, you may get a nice shell error message like:

```
OURCES: No such file or directory
```

but you might just as easily get no message at all. Remember to surround your make variables with parentheses.

missing separator

The message:

```
makefile:2:missing separator. Stop.
```

or:

```
makefile:2:missing separator (did you mean TAB instead of 8 spaces?).  Stop.
```

usually means you have a command script that is using spaces instead of tabs.

The more literal interpretation is that make was looking for a make separator such as :, =, or a tab, and didn't find one. Instead, it found something it didn't understand.

commands commence before first target

The tab character strikes again!

This error message was first covered in the section "Parsing Commands" in Chapter 5. This error seems to appear most often in the middle of *makefile*s when a line outside of a command script begins with a tab character. make does its best to disambiguate this situation, but if the line cannot be identified as a variable assignment, conditional expression, or multiline macro definition, make considers it a misplaced command.

unterminated variable reference

This is a simple but common error. It means you failed to close a variable reference or function call with the proper number of parentheses. With deeply nested function calls and variable references, make files can begin to look like Lisp! A good editor that does parenthesis matching, such as Emacs, is the surest way to avoid these types of errors.

Errors in Command Scripts

There are three common types of errors in command scripts: a missing semicolon in multiline commands, an incomplete or incorrect path variable, or a command that simply encounters a problem when run.

We discussed missing semicolons in the section "Good Coding Practices," so we won't elaborate further here.

The classic error message:

```
bash: foo: command not found
```

is displayed when the shell cannot find the command foo. That is, the shell has searched each directory in the PATH variable for the executable and found no match. To correct this error, you must update your PATH variable, usually in your *.profile* (Bourne shell), *.bashrc* (bash), or *.cshrc* (C shell). Of course, it is also possible to set the PATH variable in the *makefile* itself, and export the PATH variable from make.

Finally, when a shell command fails, it terminates with a nonzero exit status. In this case, make reports the failure with the message:

```
$ make
touch /foo/bar
touch: creating /foo/bar: No such file or directory
make: *** [all] Error 1
```

Here the failing command is touch, which prints its own error message explaining the failure. The next line is make's summary of the error. The failing *makefile* target is shown in square brackets followed by the exit value of the failing program. Sometimes make will print a more verbose message if the program exits due to a signal, rather than simply a nonzero exit status.

Note also that commands executed silently with the @ modifier can also fail. In these cases, the error message presented may appear as if from nowhere.

In either of these cases, the error originates with the program make is running, rather than make itself.

No Rule to Make Target

This message has two forms:

```
make: *** No rule to make target XXX. Stop.
```

and:

```
make: *** No rule to make target XXX, needed by YYY. Stop.
```

It means that make decided the file *XXX* needed to be updated, but make could not find any rule to perform the job. make will search all the implicit and explicit rules in its database before giving up and printing the message.

There are three possible reasons for this error:

- Your *makefile* is missing a rule required to update the file. In this case, you will have to add the rule describing how to build the target.
- There is a typo in the *makefile*. Either make is looking for the wrong file or the rule to update the file specifies the wrong file. Typos can be hard to find in *makefile*s due to the use of make variables. Sometimes the only way to really be sure of the value of a complex filename is to print it out either by printing the variable directly or examining make's internal database.
- The file should exist but make cannot find it either, because it is missing or because make doesn't know where to look. Of course, sometimes make is absolutely correct. The file is missing—perhaps you forgot to check it out of CVS. More often, make simply can't find the file, because the source is placed somewhere else. Sometimes the source is in a separate source tree, or maybe the file is generated by another program and the generated file is in the binary tree.

Overriding Commands for Target

make allows only one command script for a target (except for double-colon rules, which are rarely used). If a target is given more than one command script, make prints the warning:

```
makefile:5: warning: overriding commands for target foo
```

It may also display the warning:

```
makefile:2: warning: ignoring old commands for target foo
```

The first warning indicates the line at which the second command script is found, while the second warning indicates the location of the original command script that is being overridden.

In complex *makefile*s, targets are often specified many times, each adding its own prerequisites. One of these targets usually includes a command script, but during development or debugging it is easy to add another command script without realizing you are actually overriding an existing set of commands.

For example, we might define a generic target in an include file:

```
# Create a jar file.
$(jar_file):
        $(JAR) $(JARFLAGS) -f $@ $^
```

and allow several separate *makefile*s to add their own prerequisites. Then in a *makefile* we could write:

```
# Set the target for creating the jar and add prerequisites
jar_file = parser.jar
$(jar_file): $(class_files)
```

If we were to inadvertently add a command script to this *makefile* text, `make` would produce the overriding warning.