

How to do the most with the least

SAMPLE  
CHAPTER

# Minimal Perl

for UNIX and  
Linux People

Tim Maher



 MANNING



*Minimal Perl*  
by Tim Maher

**Chapter 8**

Copyright 2007 Manning Publications

# *brief contents*

---

## *Part 1 Minimal Perl: for UNIX and Linux Users 1*

- 1 Introducing Minimal Perl 3*
- 2 Perl essentials 16*
- 3 Perl as a (better) grep command 53*
- 4 Perl as a (better) sed command 89*
- 5 Perl as a (better) awk command 121*
- 6 Perl as a (better) find command 178*

## *Part 2 Minimal Perl: for UNIX and Linux Shell Programmers 203*

- 7 Built-in functions 205*
- 8 Scripting techniques 247*
- 9 List variables 295*
- 10 Looping facilities 330*
- 11 Subroutines and variable scoping 362*
- 12 Modules and the CPAN 388*



## CHAPTER 8

---

# *Scripting techniques*

- |                                                                       |                                                         |
|-----------------------------------------------------------------------|---------------------------------------------------------|
| 8.1 Exploiting script-oriented functions 248                          | 8.5 Interpolating command output into source code 269   |
| 8.2 Pre-processing arguments 256                                      | 8.6 Executing OS commands using <code>system</code> 275 |
| 8.3 Executing code conditionally with <code>if/else</code> 259        | 8.7 Evaluating code using <code>eval</code> 283         |
| 8.4 Wrangling strings with concatenation and repetition operators 265 | 8.8 Summary 292                                         |

Those nifty one-line commands you saw in part 1 are easy to type, and they're adequate for an impressively wide variety of common tasks. And gosh darn it, you might even say they're cute, if not downright *elegant*.

But sooner or later, you'll need to write programs that can validate their arguments, handle arguments that aren't filenames, capture and manipulate outputs from Unix commands, process inputs from interactive users, select particular branches of code to execute, or even compose and execute new Perl programs on the fly. This chapter teaches you the language features and programming techniques that are used to perform such tasks.

For instance, you'll learn how to write programs that accept arguments, and the benefits of doing so. As a case in point, the following `grep` commands can look for different patterns in different places—despite the fact that they're all running the same program—because `grep` accepts arguments:

```
grep 'SpamMeister@scumbags\.com' inbox
grep 'Mr\. Waldo Goodbar'           WhereCouldHeBe
grep 'Loofabob Circletrousers'     bikini_bottom_of_another_dimension
```

Arguments can also be used in emulating a familiar user interface. For example, we'll discuss a `grep`-like script called `perlgrep` that accepts the search pattern as its first argument:

```
perlgrep 'RE' filename
```

That's a more natural user interface for a grepper than this switch-oriented one we employed in section 3.13.2:

```
greperl -pattern='RE' filename
```

Another technique you'll learn is how to run Shell commands from within Perl programs, using the `system` function or Perl's version of the Shell's *command-substitution* facility. Although using these techniques reduces a script's OS portability, it's sometimes the best way—or the *only* way—to obtain certain kinds of vital information.

For example, what if you need to know if there is enough disk space in the current directory's partition to accommodate the needs of your program? Executing this Perl statement in a program running on Unix will help you make that determination, using *command interpolation*<sup>1</sup> to capture the output of the back-quoted `df` command:

```
$free_space=`df -k .`;
```

The `system` function also runs OS commands, but it works differently. For this reason, `system 'df -k .'` would *not* be an alternative way to obtain the same information. Accordingly, you'll learn where each technique should be used in preference to the other.

When you combine the scripting techniques you'll learn in this chapter with the built-in functions of the previous one—and the techniques for data storage and retrieval you'll learn in the next one—you'll be able to write scripts that are more robust, versatile, and advanced than those featured in part 1, as well as more OS-portable and efficient than Shell scripts.

We'll begin by discussing some special functions and variables that are primarily used in scripts.

## 8.1 EXPLOITING SCRIPT-ORIENTED FUNCTIONS

Certain built-in functions are used more commonly in scripts than in one-line commands like those you saw in part 1. These include Perl's `shift` and `exit` functions, which resemble their Shell namesakes, and `defined`, which is unique to Perl. We'll discuss the applications and benefits of these functions next, so you'll understand how they're used in the scripts that appear later in this chapter.

---

<sup>1</sup> Which is Perl's name for what the Shell calls *command substitution*.

### 8.1.1 Defining `defined`

To help you appreciate the value of `defined`, we'll first illustrate a common problem that it can solve. Consider these lines from `some-script`, which might appear at the top of a script that requires arguments:

```
$ARGV > 0 or warn "$0: No arguments!\n" and exit 255;
$pattern=$ARGV[0];
```

If the program reaches the assignment statement, we know for sure that at least one argument was provided. But to handle the case where that argument is present but *empty*, we might want to add this third statement:

```
$pattern or warn "$0: Bad first argument!\n" and exit 250;
```

This extra precaution detects potential slip-ups like the following one, which, through a combination of conscientious quoting and bad typing, winds up invoking the script with an empty first argument:

```
$ RE='helium'; # set the variable
$ some-script "$ER" hynerians # pass $RE's contents as arg1
some-script: Bad first argument!
```

But what does an expression of the form `$pattern` or *something* do? It determines whether *something* will be evaluated on the basis of the True/False value of `$pattern` (see section 2.4.2). The warn message is triggered in this case because the null string that gets assigned to `$pattern` is one of Perl's False values (the other is 0).

However, there's a complication. Some scripts may *want* to accept a False value—especially 0—as a legitimate argument. After all, with a `grep`-like script, shouldn't the user be allowed to look for records containing zeroes? This one disallows that:

```
$ some-script '0' luxans # pass 0 as pattern argument
some-script: Bad first argument!
```

Unlike most languages, Perl provides an elegant solution to this problem, by allowing you to conduct separate tests to identify undefined, empty, and False values. This gives you the ability to treat certain Falsely-valued expressions—such as those associated with missing or empty arguments—differently than others, such as an argument of 0.

False values can be sensed using logical operators (`$pattern` or *whatever*), and empty values can be identified using the string comparison operator (`$something` ne "" or `warn`; see table 5.11). The property of being *defined*, on the other hand, is determined using the `defined` function, which returns a True/False value according to the status of its argument.

What does it mean to be “defined”? Simply that the expression (usually a variable) has been assigned a value. If it hasn't—e.g., because it's accidentally being used before it's been initialized—a warning is triggered:

```
print $name;           # What's in a $name?
$name='Willy Nilly';
Use of uninitialized value in print ...
```

With this background in mind, let's look at a practical example that can benefit from the use of `defined`:

```
$tip=shift;           # tip amount is in argument
$tip or die "Usage: $0 tip_to_waiter_in_dollars\n";
```

That code must have been commissioned by the Waiters Union! It forces the diner to tip the waiter a non-zero amount, because an argument of 0 is False and therefore causes `die` to terminate the script.

Instead of asking the question “Does `$tip` have a True value?” the code should ask these two questions: “Does `$tip` have a value?” and if so, “Is its value non-empty?”

Here's an improved version that applies these tests and doesn't reject a null tip for a bad waiter (the improvements are in bold):

```
$tip=shift;
defined $tip and $tip ne "" or
    die "Usage: $0 tip_to_waiter_in_dollars (0 or more)\n";
# Now report the tip to the IRS
```

If `defined` returns False (because no argument was provided), the `or`-branch is executed, which terminates the program. If `defined` returns True, the next test is whether `$tip` contains something other than a null string. If it does—even if it's just the (False) number 0—the program lives on. On the other hand, if a null-string argument is provided (as demonstrated earlier), the program terminates—as it should.<sup>2</sup>

That, in a nutshell, is why we need Perl's `defined` function—so we can test whether an expression *has* a value, independently of whether that value is True or False. You'll find a more detailed explanation of how Perl treats undefined values in the next section, but feel free to skip it for now if you wish.

### **Using *defined* for keyboard input**

As shown earlier (in table 7.6), you can read input from the standard input channel by using the angle brackets of the input operator with `STDIN`. In such cases, a prompt is typically used to solicit the input, leading to a `printf`/variable-assignment sequence like this one:

```
printf 'Email resignation letter? YES to confirm, <^D> to exit: ';
$answer=<STDIN>;
```

In cases like this where you're dealing with a live user providing input from a keyboard, you have to be ready for these possibilities:

---

<sup>2</sup> The following additional test could be added before the `or` keyword, to ensure that a positive integer number was provided: `and $tip =~ /\^d+$/`.

- 1 The user types some characters, such as “maybe”, and then presses <ENTER>. Because `$answer` contains “maybe\n”, which isn’t a null string, its string-value is True (see section 2.4.2).
- 2 The user just presses <ENTER>. Because `$answer` contains “\n”, which isn’t a null string either, it’s also True.
- 3 The user presses <^D>, which signifies the “end of file” condition for typed input. In this special case, `$answer` receives nothing but is marked as “undefined”, which makes it a False string.
- 4 The user kills the program by pressing <^C> (or some other fatal-signal-generating character), which terminates the program immediately—thereby liberating the programmer from concerns about handling user input!

The programmer would identify the case at hand by conducting various tests on `$answer` and then provide an appropriate response.

One obvious approach would look something like this:

```
#!/usr/bin/perl -wl
printf 'Email resignation letter? YES to confirm, <^D> to exit: ';
$answer=<STDIN>;           # omitting chomp to simplify
$answer ne "YES\n" and    # This is line 6
    die "\n$0: Hasty resignation averted\n";
print 'Sending email';    # (emailing code unshown)
```

This technique works nicely for cases A and B, which result in something usable being stored in `$answer` (a newline or more).

But with case C, this output is produced:

```
Email resignation letter? YES to confirm, <^D> to exit: <^D>
Use of uninitialized value in string ne at line 6.
...
```

The good news is that the user still has time to reconsider her resignation, because the undefined value in `$answer` didn’t equate to “YES\n”, thereby causing the program to die.

But what about that warning message? That’s Perl’s way of telling you that one of the operands for the string inequality operator (`ne`) did not have a *defined* value—and you know that it can’t be complaining about “YES\n”, so it must be `$answer`. The program is allowed to continue, but Perl fudges in a null string as `$answer`’s value to allow the comparison to be performed. That’s why it issues a warning—so you’ll know it’s working with “best-guess” data rather than the real thing.

Here’s what’s happening behind the scenes. When Perl encounters an end-of-file immediately upon reading input, it returns a special value called “undefined” to signify that no usable value was obtained. Accordingly, when `$answer=<STDIN>` calls on Perl to assign the value returned by the input operator to `$answer`, Perl marks

`$answer` as undefined This signifies that the variable has been brought into existence, but not yet given a usable value.

The solution is to add an additional check using the `defined` function, like so:

```
(! defined $answer or $answer ne "YES\n" ) and
  die "\n$0: Hasty resignation averted\n";
```

This ensures that the program will die if `$answer` is undefined, and also that `$answer` won't be compared to "YES\n" unless it has a defined value. That last property circumvents the use of a fabricated value in the inequality comparison, and the "uninitialized value" warning that goes with it.

With this adjustment, if `$answer` is undefined, the program can terminate without a scary-looking warning disturbing the user.<sup>3</sup>

The rule for avoiding the accidental use of undefined values, and triggering the warnings they generate, is this:

*Always test a value that might be undefined, for being defined, before attempting to use that value.*

But there is an exception—copying a value, as in `$got_switch`, never triggers a warning—even when `$answer` is undefined. That's because moving undefined values around, as opposed to using them in significant ways, is considered a harmless activity.

### ***Tips on using `defined`***

The following statement attempts to set `$got_switch` to a True/False value, according to whether any (or all) of the script's switches was provided on the command line:

```
$got_switch=defined $debug or defined $verbose;      # WRONG!
```

Here's the warning it generates:

```
Useless use of defined operator in void context
```

That message arises because the assignment operator (`=`) has higher precedence than the logical `or`, causing the statement to be interpreted as if it had been typed like this:<sup>4</sup>

```
( $got_switch=defined $debug ) or defined $verbose;
```

Perl's warning tells the programmer that it was useless to include the `or defined` part, because there's no way for its result to be used anywhere (i.e., it's in a *void context*). As with other problems based on operator precedence, the fix is to add explicit parentheses to indicate which expressions need to be evaluated before others:

```
$got_switch=( defined $debug or defined $verbose ); # Right.
```

---

<sup>3</sup> Which might result in you being paged at 3 a.m.—prompting you to consider your *own* resignation!

<sup>4</sup> The Minimal Perl approach minimizes precedence problems, but they'll still crop up with logical operators now and then (see "Tips" at the end of section 2.4.5, appendix B, and `man perl op`).

In many cases, a Perl program ends up terminating by running out of statements to process. But in other cases, the programmer needs to force an earlier *exit*, which you'll learn how to do next.

### 8.1.2 Exiting with `exit`

As in the Shell, the `exit` command is used to terminate a script—but before doing so, it executes the `END` block, if there is one (like `AWK`). Table 8.1 compares the way the Shell and Perl versions of `exit` behave when they're invoked without an argument or with a numeric argument from 0 to 255.

**Table 8.1** The `exit` function

Shell	Perl	Explanation
<code>exit</code>	<code>exit;</code>	With no argument, the Shell's <code>exit</code> returns the latest value of its "\$?" variable to its parent process, to indicate the program's success or failure. Perl returns 0 by default, to indicate success. <sup>a</sup>
<code>exit 0</code>	<code>exit 0;</code>	The argument 0 signifies a successful run of the script to the parent.
<code>exit 1-255</code>	<code>exit 1-255;</code>	A number in the range 1–255 signifies a failed run of the script to the parent.

a. Because it's justifiably more *optimistic* than the Shell.

As indicated in the table, Perl's `exit` generally works like that of the Shell, except it uses 0 as the default exit value, rather than the exit value of the last command.

Although the languages agree that 0 signifies success, neither has established conventions concerning the meanings of other exit values—apart from them all indicating error conditions. This leaves you free to associate 1, for example, with a “required arguments missing” error, and 2 with an “invalid input format” error, if desired.

As discussed in section 2.4.4, Perl's `die` command provides an alternative to `exit` for terminating a program. It differs by printing an error message before exiting with the value of 255 (by default), as if you had executed `warn "message" and exit 255`. (But remember, in Minimal Perl we use the `warn` and `exit` combination rather than `die` in `BEGIN` blocks, to avoid the unsightly warning messages about aborted compilations that a `die` in `BEGIN` elicits.)

The following illustrates proper uses of the `exit` and `die` functions in a script that has a `BEGIN` block, as well as how to specify `die`'s exit value by setting the “\$!” variable,<sup>5</sup> to load the desired value into the parent shell's "\$?" variable:

---

<sup>5</sup> Later in this chapter, you'll learn how to use Perl's `if` construct, which is better than the logical `and` for making the setting of “\$!”, and the execution of `die`, *jointly* dependent on the success of the matching operator.

```

$ cat message_data
#! /usr/bin/perl -wnl

BEGIN {
    @ARGV == 1 or warn "Usage: $0 filename\n" and exit 1;
}
/^\#/ and $!=2 and die "$0: Comments not allowed in data file\n";
...

$ message_data
Usage: message_data filename
$ echo $?
1

$ message_data file # correct invocation; 0 is default exit value
$ echo $?
0

$ echo '# comment' | message_data - # "-" means read from STDIN
message_data: Comments not allowed in data file
$ echo $?
2

```

We'll look next at another important function shared by the Shell and Perl.

### 8.1.3 Shifting with `shift`

Both the Shell and Perl have a function called `shift`, which is used to manage command-line arguments. Its job is to shift argument values leftward relative to the storage locations that hold them, which has the side effect of discarding the original first argument.<sup>6</sup>

Figure 8.1 shows how `shift` affects the allocation of arguments to a Shell script's *positional parameter* variables, or to the indices of Perl's `@ARGV` array.

Shell				
Variable	\$1	\$2	\$3	
Original value	A	B	C	Before shift
New value	B	C		After shift

Perl				
Array index	0	1	2	
Original value	A	B	C	Before shift
New value	B	C		After shift

**Figure 8.1**  
Effect of `shift` in the Shell and Perl

<sup>6</sup> A common programming technique used with early UNIX shells was to process `$1` and then execute `shift`, and repeat that cycle until every argument had taken a turn as `$1`. It's discussed in section 10.2.1.

As the figure illustrates, after `shift` is executed in the Shell, the value initially stored in `$1` (A) gets discarded, the one in `$2` (B) gets relocated to `$1`, and the one in `$3` gets relocated to `$2`. The same migration of values across storage locations occurs in Perl, except the movement is from `$ARGV[1]` to `$ARGV[0]`, and so forth. Naturally, the affected Perl variables (`@ARGV` and `$#ARGV`) are updated automatically after `shift`, just as “`$*`”, “`$@`”, and “`$#`” are updated in the Shell.

Although Perl’s `shift` provides the same basic functionality as the Shell’s, it also provides two new features, at the expense of losing one standard Shell feature (see table 8.2). The new feature—shown in the table’s second row—is that Perl’s `shift` *returns* the value that’s removed from the array, so it can be saved for later access.

**Table 8.2 Using `shift` and `unshift` in the Shell and Perl**

Shell	Perl	Explanation
<code>shift</code>	<code>shift;</code>	<code>shift</code> removes the leftmost argument and moves any others one position leftward to fill the void.
N/A	<code>\$variable=shift;</code>	In Perl, the removed parameter is returned by <code>shift</code> , allowing it to be stored in a variable.
<code>shift 2</code>	<code>shift; shift;</code> OR <code>\$arg1=shift;</code> <code>\$arg2=shift;</code>	The Shell’s <code>shift</code> takes an optional numeric argument, indicating the number of values to be shifted away. That effect is achieved in Perl by invoking <code>shift</code> multiple times.
N/A	<code>shift @any_array;</code>	Perl’s <code>shift</code> takes an optional argument of an array name, which specifies the one it should modify instead of the default (normally <code>@ARGV</code> , but <code>@_</code> if within a subroutine).
N/A	<code>unshift @array1, @array2;</code>	Perl’s <code>unshift</code> reinitializes <code>@array1</code> to contain the contents of <code>@array2</code> before the initial contents of <code>@array1</code> . For example, if <code>@array1</code> in the example contained <code>(a,b)</code> and <code>@array2</code> contained <code>(1,2)</code> , <code>@array1</code> would end up with <code>(1,2,a,b)</code> .

That allows Perl programmers to write this simple statement:

```
$arg1=shift;    # save first arg's value, then remove it from @ARGV
```

where Shell programmers would have to write

```
arg1="$1"      # save first arg's value before it's lost forever!
shift         # now remove it from argument list
```

Another improvement is that Perl’s `shift` takes an optional argument that specifies the array to be shifted, which the Shell doesn’t support. However, by attaching this new interpretation to `shift`’s argument, Perl sacrificed the ability to recognize it as a numeric “amount of shifting” specification, which is the meaning `shift`’s argument has in the Shell.

Now that you've learned how to use `defined`, `shift`, and `exit` in Perl, we'll use these tools to improve on certain techniques you saw in part 1 and to demonstrate some of their other useful applications. We'll begin by discussing how they can be used in the pre-processing of script arguments.

## 8.2 PRE-PROCESSING ARGUMENTS

Many kinds of scripts need to pre-process their arguments before they can get on with their work. We'll cover some typical cases, such as extracting non-filename arguments, filtering out undesirable arguments, and generating arguments automatically.

### 8.2.1 Accommodating non-filename arguments with implicit loops

The `greperl` script of section 3.13.2 obtains its pattern argument from a command-line `switch`:

```
greperl -pattern='RE' filename
```

When this invocation format is used with a script having the `s` option on the shebang line, Perl automatically assigns `RE` to the script's `$pattern` variable and then discards the switch argument. This approach certainly makes switch-handling scripts easy to write!

But what if you want to provide a user interface that feels more *natural* to the users, based on the interface of the traditional `grep`?

```
grep 'RE' filename
```

The complication is that filter programs are most conveniently written using the `n` invocation option, which causes all command-line arguments (except switches) to be treated as filenames—including a `grep`-like script's pattern argument:

```
$ perlgrep.bad 'root' /etc/passwd # Hey! "root" is my RE!  
Can't open root: No such file or directory
```

Don't despair, because there's a simple way of fixing this program, based on an understanding of how the implicit loop works.

Specifically, the `n` option doesn't start treating arguments as filenames until the implicit input-reading loop starts running, and that doesn't occur until after the `BEGIN` block (if present) has finished executing. This means initial non-filename arguments can happily coexist with filenames in the argument list—on one condition:

*You must remove non-filename arguments from `@ARGV` in a `BEGIN` block, so they'll be gone by the time the input-reading loop starts executing.*

The following example illustrates the coding for this technique, which isn't difficult. In fact, all it takes to harvest the pattern argument is a single line; the rest is all error checking:

```

$ cat perlgrep
#! /usr/bin/perl -wnl

BEGIN {
    $Usage="Usage: $0 'RE' [file ...]";
    @ARGV > 0 or warn "$Usage\n" and exit 31; # 31 means no arg

    $pattern=shift; # Remove arg1 and load into $pattern
    defined $pattern and $pattern ne "" or
        warn "$Usage\n" and exit 27; # arg1 undefined, or empty
}
# Now -n loop takes input from files named in @ARGV, or from STDIN
/$pattern/ and print; # if match, print record

```

Here's a sample run, which shows that this script succeeds where its predecessor `perlgrep.bad` failed:

```

$ perlgrep 'root' /etc/passwd
root:x:0:0:root:/root:/bin/bash

```

The programmer even defined some custom exit codes (see section 8.1.2), which may come in handy sometime:

```

$ perlgrep "$EMPTY" /etc/passwd
Usage: perlgrep 'RE' [file ...]
$ echo $? # Show exit code
27

```

Once you understand how to code the requisite `shift` statement(s) in the `BEGIN` block, it's easy to write programs that allow initial non-filename arguments to precede filename arguments, which is necessary to emulate the user interface of many traditional Unix commands.

But don't get the idea that `perlgrep` is the final installment in our series of `grep`-like programs that are both educational and practical. Not by a long shot! There's an option-rich `preg` script lurking at the end of this chapter, waiting to impress you with its versatility.

We'll talk next about some other kinds of pre-processing, such as reordering and removing arguments.

## 8.2.2 Filtering arguments

The filter programs featured in part 1 employ Perl's AWKish `n` or `p` option, to handle filename arguments automatically. That's nice, but what if you want to exert some influence over that handling—such as processing files in alphanumeric order?

As indicated previously, you can do anything you want with a filter-script's arguments, so long as you do it in a `BEGIN` block. For example, this code is all that's needed to sort a script's arguments:

```
BEGIN {
    @ARGV=sort @ARGV;      # rearrange into sorted order
}
# Normal argument processing starts here
```

It's no secret that users can't always be trusted to provide the correct arguments to commands, so a script may want to remove inappropriate arguments.

Consider the following invocation of `change_file`, which was presented in chapter 4:

```
change_file -old='problems' -new='issues' *
```

The purpose of this script is to change occurrences of “problems” to “issues” in the text files whose names are presented as arguments. But of course, the “\*” metacharacter doesn't know that, so if any *non*-text files reside in the current directory, the script will process them as well. This could lead to trouble, because a binary file might happen to contain the bit sequence that corresponds to the word “problems”—or any other word, for that matter! Imagine the havoc that could ensue if the superuser were to accidentally modify the `ls` command's file—or, even worse, the Unix kernel's file—through such an error!

To help us sleep better, the following code silently removes non-text-file arguments, on the assumption that the user probably didn't realize they were included in the first place:

```
BEGIN {
    @ARGV=grep { -T } @ARGV; # retain only text-file arguments
}
# Normal argument processing starts here
```

`grep` selects the text-file (`-T`; see table 6.1) arguments from `@ARGV`, and then they're assigned as the new contents of that array. The resulting effect is as if the unacceptable arguments had never been there.

A more informational approach would be to report the filenames that were deleted. This can be accomplished by selecting them with `! -T` (which means “non-text files”), storing them in an array for later access, and then printing their names (if any):

```
BEGIN {
    @non_text=grep { ! -T } @ARGV; # select NON-text-file arguments
    @non_text and
        warn "$0: Omitting these non-text-files: @non_text\n";
    @ARGV=grep { -T } @ARGV;      # retain text-file arguments
}
# Normal argument processing starts here
```

But an ounce of prevention is still worth at least a pound of cure, so it's best to free the user from typing arguments wherever possible, as we'll discuss next.

### 8.2.3 Generating arguments

It's senseless to require a user to painstakingly type in lots of filename arguments—which in turn burdens the programmer with screening out the invalid ones—in cases where the program could generate the appropriate arguments on its own.

For example, Uma, a professional icon-designer, needs to copy every regular file in her working directory to a CD before she leaves work. However, the subdirectories of that directory should *not* be archived. Accordingly, she uses the following code to generate the names of all the (non-hidden) regular files in the current directory that are readable by the current user (that permission is required for her to copy them):

```
BEGIN {
    # Simulate user supplying all suitable regular
    # filenames from current directory as arguments
    @ARGV=grep { -f and -r } <*>;
}
# Real work of script begins below
```

The `<*>` expression is a use of the globbing operator (see table 7.14) to generate an initial set of filenames, which are then filtered by `grep` for the desired attributes.

Other expressions commonly used to generate argument lists in Perl (and the Shell) are shown in section 9.3, which will give you additional ideas of what you could plug into a script's `BEGIN` block. You can't always automatically generate the desired arguments for every script, but for those cases where you can, you should keep these techniques in mind.

Next, you'll learn about an important control structure that's provided in every programming language. We've managed without it thus far, due to the ease of using Perl's logical operators in its place, but now you'll see how to arrange for conditional execution in a more general way.

## 8.3 Executing code conditionally with `if/else`

The logical `or` and logical `and` operators were adequate to our needs for controlling execution in part 1, where you saw many statements like this one:

```
$pattern or warn "Usage: $0 -pattern='RE' filename\n" and exit 255;
```

However, this technique of using the True/False value of a variable (`$pattern`) to conditionally execute two functions (`warn` and `exit`) has limitations. Most important, it doesn't deal well with cases where a True result should execute one set of statements and a False result a different set.

So now it's time to learn about more widely applicable techniques for controlling two-way and multi-way branching. Table 8.3 shows the Shell and Perl syntaxes for two-way branching using `if/else`, with layouts that are representative of current programming practices. The top panel shows the complete syntax, which includes branches for both the True ("then") and False (`else`) cases of the `condition`. In

**Table 8.3 The if/else construct**

Shell <sup>a</sup>	Perl
<pre>if    condition then  commands else  commands fi</pre>	<pre>if (condition) {     code; } else {     code; }</pre>
<pre>if cond; then cmds; else cmds; fi</pre>	<pre>if (cond) { code; } else { code; }</pre>

a. In the bottom panel, *cond* stands for *condition* and *cmds* stands for *commands*.

both languages, the `else` branch is optional, allowing that keyword and its associated components to be omitted. The table's bottom panel shows condensed forms of these control structures, which save space in cases where they'll fit on one line.

We'll examine a realistic programming example that uses `if/else` next, and compare it to its `and/or` alternative.

### 8.3.1 Employing `if/else` vs. `and/or`

Here's a code snippet that provides a default argument for a script when it's invoked without the required one, and terminates with an error message if too many arguments are supplied:

```
if (@ARGV == 0) {
    warn "$0: Using default argument\n";
    @ARGV=('King Tut');
}
else {
    if (@ARGV > 1) {          # nested if
        warn "Usage: $0 song_name\n";
        exit 255;
    }
}
```

For comparison, here's an equivalent chunk of code written using the logical `and/or` approach. It employs a style of indentation that emphasizes the dependency of each subsequent expression on the prior one:

```
@ARGV == 0 and
    warn "$0: Using default arguments\n" and
    @ARGV=('King Tut') or
    @ARGV > 1 and
        warn "Usage: $0 song_name\n" and
        exit 255;
```

This example illustrates the folly of stretching the utility of `and/or` beyond reasonable limits, which makes the code unnecessarily hard to read and maintain. Moreover,

matters would get even worse if you needed to parenthesize some groups of expressions in order to obtain the desired result.

The moral of this comparison is that branching specifications that go beyond the trivial cases are better handled with `if/else` than with `and/or`—which of course is why the language provides `if/else` as an alternative.

Perl permits additional `if/elses` to be included within `if` and `else` branches, which is called *nesting* (as depicted in the left side of table 8.4). However, in cases where tests are performed one after another to select one branch out of several for execution, readability can be enhanced and typing can be minimized by using the `elsif` contraction for “else { if” (see the table’s right column).

**Table 8.4** Nested `if/else` vs. `elsif`

<code>if/else</code> within <code>else</code>	<code>elsif</code> alternative
<pre>if ( A ) {     print 'A case'; } else { # this brace disappears --&gt;     if ( B ) {         print 'B case';     }     else {         print 'other case';     } } # this brace disappears --&gt;</pre>	<pre>if ( A ) {     print 'A case'; } elsif ( B ) {     print 'B case'; } else {     print 'other case'; }</pre>

Just remember that Perl’s keyword is `elsif`, not `elif`, as it is in the Shell.

Next, we’ll look at an example of a script that does lots of conditional branching, using both techniques.

### 8.3.2 Mixing branching techniques: The `cd_report` script

The purpose of `cd_report` is to let the user select and display input records that represent CDs by matching against the various fields within those records. Through use of the following command-line switches, the user can limit his regexes to match within various portions of a record, and request a report of the average rating for the group of selected CDs:

• <code>-search='RE'</code>	Search for <i>RE</i> anywhere in record
• <code>-a='RE'</code>	Search for <i>RE</i> in the Artist field
• <code>-t='RE'</code>	Search for <i>RE</i> in the Title field
• <code>-r</code>	Report average rating for selected CDs
• (default)	Print all records, under column headings

Let's try some sample runs:

```
$ cd_report rock      # prints whole file, below column-headings
TITLE                 ARTIST                 RATING
Dark Horse            George Harrison       3
Electric Ladyland    Jimi Hendrix          5
Dark Side of the Moon Pink Floyd            4
Tommy                 The Who               4
Weasels Ripped my Flesh Frank Zappa          2

Processed 5 CD records
```

That invocation printed the entire `rock` file, because by default all records are selected. This next run asks for a report of CDs that have the word “dark” in their Title field:

```
$ cd_report -t='\bdark\b' rock
TITLE                 ARTIST                 RATING
Dark Horse            George Harrison       3
Dark Side of the Moon Pink Floyd            4

Processed 5 CD records
```

As you can tell from what got matched and printed, the script ignores case differences.

The next invocation requests CDs having “hendrix” in the Artist field or “weasel” anywhere within the record, along with an average-rating report:

```
$ cd_report -a='hendrix' -search=weasel -r rock
TITLE                 ARTIST                 RATING
Electric Ladyland    Jimi Hendrix          5
Weasels Ripped my Flesh Frank Zappa          2

Average Rating for 2 CDs: 3.5

Processed 5 CD records
```

Now that I've piqued your interest, take a peek at the script, shown in listing 8.1. Notice its strategic use of the `if/else` and logical `and/or` facilities, to exploit the unique advantages of each. For example, `if/else` is used for selecting blocks of code for execution (e.g., Lines 18–20, 21–33), logical `and` is used for making matching operations conditional on the defined status of their associated switch variables (Lines 23–25), and logical `or` is used for terminating a series of tests (Lines 23–25) as soon as the True/False result is known.

Let's examine this script in greater detail. First, the shebang line includes the primary option cluster for “field processing with custom separators” (using tabs), plus the `s` option for switch processing (see table 2.9).

Then, the initialization on Line 6 tells the program how many tab-separated fields to expect to find in each input record, so it can issue warnings for improperly formatted ones. The next line sets `$sel_cds` to 0, because if Line 29 isn't executed, it would otherwise still be undefined by Line 38 and trigger a warning there.

### Listing 8.1 The cd\_report script

```
1  #!/usr/bin/perl -s -wnlaF'\t+'
2
3  our ( $search, $a, $t, $r );           # make switches optional
4
5  BEGIN {
6      $num_fields=3;                   # number of fields per line
7      $sel_cds=0; # so won't be undefined in END, if no selections
8
9      $options=( defined $r or defined $a or # any options?
10                 defined $t or defined $search );
11
12     print "TITLE\t\t\tARTIST\t\tRATING"; # print column headings
13 }
14
15 ##### BODY OF PROGRAM, EXECUTED FOR EACH LINE OF INPUT #####
16 ( $title, $artist, $rating )=@F; # load fields into variables
17 $fcount=@F; # get field-count for line
18 if ( $fcount != $num_fields ) { # line improperly formatted
19     warn "\n\tBad field count of $fcount on line #$.; skipping!";
20 }
21 else { # line properly formatted
22     $selected=( # T/F to indicate status of current record
23                 defined $t and $title =~ /$t/i or # match with title?
24                 defined $a and $artist =~ /$a/i or # match with artist?
25                 defined $search and /$search/i or # match with record?
26                 ! $options # without options, all records selected
27             );
28     if ( $selected ) { # the current CD was selected
29         $sel_cds++; # increment #CDs_selected
30         $sum_ratings+=$rating; # needed for -r option
31         print; # print the selected line
32     }
33 }
34 END {
35     $num_cds=$.; # maximum line number = #lines read
36     if ( $r and $sel_cds > 0 ) {
37         $ave_rating=$sum_ratings / $sel_cds;
38         print "\n\tAverage Rating for $sel_cds CDs: $ave_rating";
39     }
40     print "\nProcessed $num_cds CD records"; # report stats
41 }
```

Line 9 sets the variable `$options` to a True or False value to indicate whether the user supplied any switches.

The BEGIN block ends with Line 12, which prints column headings to label the upcoming output.

Line 16, the first one that's executed for each input record, loads its fields into suitably named variables. Then, the field count is loaded into `$fcount`, so it can be easily compared to the expected value in `$num_fields` and a warning can be issued on Line 19, if the record is improperly formatted.

If the “then” branch containing that warning is executed, the `else` branch comprising the rest of the program's body is skipped, causing the next line to be read and execution to continue from Line 16. But if the record is determined to have three fields on Line 18, the `else` branch on Line 21 is taken, and a series of conditional tests is conducted to see whether the current record should be selected for printing—as indicated by `$selected` being set to `True` (Line 22).

Let's look more closely at these tests. Line 23 senses whether the “search in the Title field” option was provided; if so, it employs the user-supplied pattern to test for a match with `$title`. If that fails, matches are next looked for in the `$artist` and `$_` variables—if requested by the user's switches. Because logical `ors` connect this series of “defined and *match*” clauses, the first `True` result (if any) circumvents the remaining tests. If no option was provided by the user, execution descends through the “defined and *match*” clauses and evaluates the `! $options` test on Line 26, which sets `$selected` to `True` to cause the current CD's record to be selected.

If the current record was selected, Line 29 increments the count of selected CDs, and its rating is added to the running total before the record is printed on Line 31.

The cycle of processing then resumes from Line 16 with the next record, until all input has been processed.

Because an average can't be computed until all the individual ratings have been totaled, that calculation must be relegated to the `END` block. Line 36 checks whether an average rating report was requested (via the `-r` switch); if that's so, and at least one CD was selected, the average rating is computed and printed.

As a final step, the script reports the number of records read. To enhance readability, the value of “\$.” is copied into a suitably named variable on Line 35 before its value is printed on Line 40.

### 8.3.3 Tips on using `if/else`

The most common mistake with the `if/else` construct is a syntax error: leaving out the closing right-hand brace that's needed to match the opening left-hand brace, or vice versa. In common parlance, this is called *not balancing* the curly braces (or having an *imbalance* of them). Users of the `vi` editor can get help with this problem by placing the cursor on a curly brace and pressing the `%` key, which causes the cursor to momentarily jump to the matching brace (if any).

Another common mistake beginners make is appending a semicolon after the final curly brace of `if/else`. That's somewhat gratifying to their teacher, because this reveals their awareness that semicolons are required terminators for Perl statements and critical elements of syntax. However, curly-brace delimited code blocks

are *constructs* that *encase* statements, rather than statements themselves, so they don't rate the semicolon treatment.

For help in spotting these syntax errors and others, try running your code through a beautifier. You can learn about and download the standard Perl beautifier from <http://perltidy.sourceforge.net>.<sup>7</sup>

As a final note, Perl, unlike some of its relatives, doesn't permit the omission of the curly braces in cases where only a single statement is associated with a condition:

```
if ( condition ) statement;           # WRONG!
if ( condition ) { statement; }       # {}s are mandatory in Perl
```

So get used to typing those curly braces—*without* terminating semicolons!

Having just discussed an important flow-control structure that's highly conventional—which is an unusual occurrence in a Perl book—we will regain our Perlistic footing by looking next at some valuable yet *unconventional* operators for string manipulation.

## 8.4 WRANGLING STRINGS WITH CONCATENATION AND REPETITION OPERATORS

Table 8.5 shows some handy operators for strings that we haven't discussed yet. The *concatenation operator* joins together the strings on its left and right. It comes in handy when you need to assemble a longer string from shorter ones, or easily reorder the components of a string (as you'll see shortly).

The *repetition operator* duplicates the specified string the indicated number of times. It can save you a lot of work when, for example, you want to generate a row of dashes across the screen—without typing every one of them.

The concatenation operator doesn't get much use in Minimal Perl, for two reasons. First, our routine use of the `l` option eliminates the most common need for it

**Table 8.5** String operators for concatenation and repetition

Name	Symbol	Example	Result	Explanation
Concatenation operator	.	<code>\$ab='A' . 'B';</code>	AB	The concatenation operator joins together ( <i>concatenates</i> ) the strings on its left and right sides. When used in its compound form with the assignment operator ( <code>.</code> ), it causes the string on the right to be appended to the one on the left.
		<code>\$abc=\$ab . 'C';</code>	ABC	
		<code>\$abc='A';</code>	A	
		<code>\$abc.='B';</code>	AB	
		<code>\$abc.='C';</code>	ABC	
Repetition operator	x	<code>\$dashes='-' x 4;</code>	----	The repetition operator causes the string on its left to be repeated the number of times indicated on its right.
		<code>\$spaces=' ' x 2;</code>	□□	

<sup>7</sup> To learn about the *first* Perl beautifier, see [http://TeachMePerl.com/perl\\_beautifier.html](http://TeachMePerl.com/perl_beautifier.html).

that others have. Second, in other cases where this operator is commonly used, it's often simpler to use quotes to get the same result.

For example, consider this code sample, in which `random_kind_of_chow` is an imaginary user-defined function that returns a “chow” type (“mein”, “fun”, “Purina”, etc.):

```
$kind=random_kind_of_chow;
$order="large chow $kind";           # e.g., "large chow mein"
```

That last statement, which uses double quotes to join the words together, is easier to read and type than this equivalent concatenation-based alternative:

```
$order='large ' . 'chow ' . $kind;
```

But you can't call functions from within quotes, so the concatenation approach is used in cases like this one, where the words returned by two functions need to be joined with an intervening space:

```
$order=random_preparation . ' ' . random_food; # flambéed Vegemite?
```

On the other hand, concatenation using the compound version (`.=`) of the concatenation operator<sup>8</sup> is preferred over quoting for lines that would otherwise be inconveniently long.

For instance, this long assignment statement

```
$good_fast_things='cars computers action delivery recovery ]
reimbursement replies';
```

is less manageable than this equivalent pair of shorter ones:

```
$good_fast_things='cars computers action delivery';
$good_fast_things.=' recovery reimbursement replies';
```

The syntax used in that last statement

```
$var.=' new stuff';
```

appends ' *new stuff*' to the end of the variable's existing contents.

The compound form of the concatenation operator is sometimes also used with short strings, in applications where it may later be necessary to independently change, conditionally select, or reorder them. For instance, here's a case where the tail end of a message needs to be conditionally selected, to optimally tailor the description of a product for different groups of shoppers:

```
$sale_item='ONE HOUR SALE on: ';
if ($funky_web_site) {
    $sale_item.=' pre-weathered raw-hemp "gangsta" boxers';
}
else { # for posh sites
    $sale_item.=' hand-rubbed organic natural-fiber underpants';
}
```

---

<sup>8</sup> See the last panel of table 5.12 for more information.

This use of the concatenation operator is also helpful for aggregating strings that become available at different times during execution, as you'll see next.

#### 8.4.1 Enhancing the `most_recent_file` script

Remember the `most_recent_file` script, which provides a robust replacement for `find | xargs ls -lrd` when sorting large numbers of filenames?<sup>9</sup> It suffers from the limitation of showing only a single filename as the “most recent,” when others are tied with it for that status.

This shortcoming is easily overcome. Specifically, all that's required to enhance `most_recent_file` to handle ties properly is to take its original code

```
if ($mtime > $newest) {      # If current file is newest yet seen,
    $newest=$mtime;         # remember file's modification time, and
    $name=$_;               # remember file's name
}
```

and add to it the following `elsif` clause, which arranges for each filename having the same modification time to be *appended* to the `$name` variable (after a newline for separation), using the compound-assignment form of the concatenation operator:

```
elsif ($mtime == $newest) { # If current file ties newest yet seen
    $name.=$_; # append new tied filename after existing one(s)
}
```

Next we'll look at a code snippet that, when used as intended, will annoy law-abiding Netizens with its deceitful claims and awful shpeling mistakes. Its redeeming qualities are that it illustrates some important points about the relative precedence of the concatenation and repetition operators, and the code-maintenance advantages of using the concatenation operator.

#### 8.4.2 Using concatenation and repetition operators together

Here's a code snippet that uses both the repetition and concatenation operators in their simple forms, as well as the concatenation operator in its compound assignment form:

```
$pitch=($greedy_border='$' x 68 . "\n"); # initializes both variables

$pitch.="\t\t You con belief me, because I am lawyers. \n";
$pitch.="\t\tYou can reely MAKE MONEY FA\T with our system!\n";
$pitch.= $greedy_border;

print $pitch;
```

In the first statement, because the string repetition operator (`x`) has higher precedence than the concatenation operator, the `$` symbol gets repeated 68 times before

---

<sup>9</sup> See listing 6.1.



But this mistaken variation *overwrites* the first string with the second one:

```
$Usage="Usage: $0 [-f] [-i] [-l] [-v] [-n] [-d]";  
$Usage=" [-p|-c] [-m] [-s] [-r] 'RE' [file...]\n"; # WRONG!
```

So when you're using this coding technique and you find that the earlier portions of the built-up string have mysteriously disappeared, here's how to fix the problem. Locate the assignment statement that loads what appears at the beginning of the incomplete string (in this case, " [-p|-c] ..."), and change its "=" to the required ".=".

Next, we'll discuss an especially useful programming feature that Perl inherited from the Shell, which allows the output of OS commands to be manipulated within Perl programs.

## 8.5 INTERPOLATING COMMAND OUTPUT INTO SOURCE CODE

The Shell inherited a wonderful feature from the venerable MULTICS OS that it calls *command substitution*. It allows the output of a command to be captured and inserted into its surrounding command line, as if the programmer had typed that output there in the first place. In a sense, it's a special form of output redirection, with the current command line being the target of the redirection.

Let's say you needed a Shell script to report the current year every time it's run. One way to implement this would be to hard-wire the (currently) current year in an `echo` command, like so:

```
echo 'The year is 2006' # Output: The year is 2006
```

But to prevent the frenetic refrain of your beeper from rudely awakening you the next time January rolls around, you'd be better off writing that line as follows:

```
echo "The year is `date +%Y`"
```

Here's how it works. The back-quotes (or *grave accents*) and the string they enclose constitute a command-substitution request. It's job is to write `date`'s output over itself, making this the command that's ultimately executed:

```
echo "The year is 2006" # `date ...` replaced by its own output
```

The benefit is that a script that derives the year through command substitution always knows the current year—allowing its maintainer to sleep through the night.

Perl also provides this valuable service, but under the slightly different name of *command interpolation*. Table 8.6 shows the syntax for typical uses of this facility in the Shell and Perl.<sup>12</sup>

---

<sup>12</sup> As indicated in the left column of the table, the Bash and Korn shells simultaneously support an alternative to the back-quote syntax for command substitution, of the form `$(command)`.

**Table 8.6 Command substitution/interpolation in the Shell and Perl**

Shell <sup>a</sup>	Perl	Explanation
<code>var=`cmd`</code> OR <code>var=\$(cmd)</code>	<code>\$var=`cmd`</code>	The <code>cmd</code> is processed for variable substitutions as if it were in double quotes, and then it's executed, with the output being assigned in its entirety to the variable. <code>cmd</code> 's exit value is stored in the "\$?" variable.
<code>array=(`cmd`)</code> OR <code>array=\$(cmd)</code>	<code>@array=`cmd`</code>	<code>cmd</code> 's output is processed as described above, and then "words" (for the Shell) or \$/ separated records (for Perl) are assigned to the array.
<code>cmd2 `cmd`</code> OR <code>cmd2 \$(cmd)</code>	<code>function `cmd`</code> OR <code>function scalar `cmd`</code>	<code>cmd</code> is processed, and then, in the Shell case, the individual words of the output are supplied to <code>cmd2</code> as arguments. In Perl's list context, each record of the output is submitted to <code>function</code> as a separate argument, whereas in scalar context, all output is presented as a single argument.
<code>" `cmd` "</code> OR <code>"\$(cmd)"</code>	<code>`cmd`</code>	In the Shell, double quotes are needed to protect <code>cmd</code> 's output from further processing. In Perl, that protection is always provided, and double quotes aren't allowed around command interpolations. The Shell examples yield all of <code>cmd</code> 's output as one line, whereas the Perl example yields a list of \$/ separated records.

a. `cmd` and `cmd2` represent OS commands, `var/$var` and `array/@array` Shell/Perl variable names, and `function` a Perl function name.

When a Unix shell processes a command substitution, a shell of the same type (Bash, C-shell, etc.) interprets the command. In contrast, with Perl, an OS-designated command interpreter (`/bin/sh` on Unix) is used.

As indicated in the third row of the table, when command substitution (or interpolation) is used to provide arguments to another command (or function), the arguments are constructed differently in the two languages. The Shell normally presents each *word* separately, but it will use the entire output string as a single argument if the command substitution is double quoted. Perl, on the other hand, presents each *record* as a separate argument in list context, or all records as a single argument in scalar context.

Another difference is that the Shell automatically strips off the trailing newline from the command's output, and Perl doesn't. To make Perl act like the Shell, you can assign the output to a variable and then `chomp` it (see section 7.2.4).

Because of these differences, the corresponding Shell and Perl examples shown in table 8.6 don't behave in exactly the same way. However, Perl can generally be trusted

to give you what you want by default—and anything else you may need, with a little more coaxing.<sup>13</sup>

The major differences in the results provided by the languages are, as usual, due to the Shell's propensity for doing additional post-processing of the results of substitutions (as discussed earlier). We'll discuss this issue in greater depth as we examine some sample programs in upcoming sections.

The command we'll discuss next is held in high esteem by Shell programmers, because it makes output sent to terminal-type devices look a lot fancier—and, consequently, makes those writing the associated scripts seem a lot cleverer!

### 8.5.1 Using the `tput` command

The Unix utility called `tput` can play an important role in Shell scripts designed to run on computer terminals or their emulated equivalents (e.g., an `xterm` or `dtterm`). For instance, `tput` can render a script's error messages in reverse video, or make a prompt blink to encourage the user to supply the requested information.

Through use of command interpolation, Perl programmers writing scripts for Unix systems can also use this valuable tool.<sup>14</sup>

The top panel of table 8.7 lists the most commonly used options for the `tput` command. For your convenience, the ones that work on the widest variety of terminals (and emulators) are listed nearest the top of each of the table's panels.

**Table 8.7 Controlling and interrogating screen displays using `tput` options**

Display mode	Enabling option	Disabling option
standout	<code>smso</code>	<code>rmso</code>
underline	<code>smul</code>	<code>rmul</code>
bold	<code>bold</code>	<code>sgr0</code>
dim	<code>dim</code>	<code>sgr0</code>
blink	<code>blink</code>	<code>sgr0</code>
Terminal information	Option	Explanation
columns	<code>cols</code>	Reports number of columns.
lines	<code>lines</code>	Reports number of lines.

<sup>13</sup> I was put off by these disparities when I first sat down to learn Perl, but now I can't imagine how I ever put up with the Shell, and I'm pleased as punch with Perl.

<sup>14</sup> There's a Perl module (`Term:Cap`) that bypasses the `tput` command to access the Unix terminal information database directly, but it's much easier to run the Unix command via command interpolation than to use the module.

## Highlighting trailing whitespaces with `tput`

People who do a lot of grepping in their jobs have two things in common: They're fastidious about properly quoting `grep`'s pattern argument (otherwise they'd wind up unemployed), and they *hate* text files that have stray whitespace characters at their ends. You'll see how `tput` can help them in a moment. But first, why do they view files having dangling whitespaces with contempt? Because such files thwart attempts that would otherwise be successful to match patterns at the ends of their lines:

```
grep 'The end!$' naughty_file # Hope there's no dangling space/tab!
```

Because the `$` metacharacter anchors the match to the end of the line, there's no provision for extra space or tab characters to be present there. For this reason, the lack of any matches could mean either that no line ends with "The end!" or that the lines that do visibly end with that string have invisible whitespace(s) afterwards.

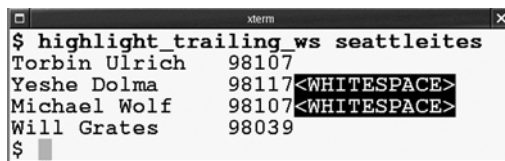
Figure 8.2 shows how `tput` can help with a simple script that makes the presence of dangling whitespace characters excruciatingly clear. It uses "standout" mode to draw the user's attention to the lines that need to be pruned, to make them safe for grepping.

Listing 8.2 presents the script. As with many of the `sed`-like scripts covered in chapter 4, this one uses the `p` option to automatically print the input lines after the substitution operator processes them.

### Listing 8.2 The `highlight_trailing_ws` script

```
1  #!/usr/bin/perl -wpl
2
3  BEGIN {
4      $ON = `tput smso`; # start mode "standout"
5      $OFF = `tput rmso`; # remove mode "standout"
6  }
7  # Show "<WHITESPACE>" in reverse video, to attract eyeballs
8  s/[<SPACE>\t]+$/$ON<WHITESPACE>$OFF/g;
```

The script works by replacing trailing sequences of spaces and/or tabs with the string "`<WHITESPACE>`", which is rendered in standout mode (usually *reverse video*) for additional emphasis.<sup>15</sup> Once the presence of dangling whitespace has been revealed by



```
xterm
$ highlight_trailing_ws seattleites
Torbin Ulrich 98107
Yeshe Dolma 98117<WHITESPACE>
Michael Wolf 98107<WHITESPACE>
Will Grates 98039
$
```

**Figure 8.2**  
Output from the  
`highlight_trailing_ws` script

<sup>15</sup> You might think it sufficient to highlight the offending whitespace characters themselves, rather than an inserted word, but reverse video mode doesn't affect the display of tabs on most terminals.

this tool, the “data hygiene” team could give some refresher training to the “data entry” team and have them correct the offending lines.

This is a good example of using `tput` to draw the user’s attention to important information on the screen, and I’m sure you’ll find other places to use it in your own programming.

Command interpolation is used to solve many other pesky problems in the IT workplace. In the next section, you’ll see how it can be used to write a `grep`-like script that handles directory arguments sensibly, by searching for matches in the files within them.

## 8.5.2 Grepping recursively: The `rgrep` script

As mentioned in chapter 6, a *recursive grep*, which automatically descends into sub-directories to search the files within them, can be a useful tool. Although the GNU `grep` provides this capability through an invocation option, a Perl-based grepper has several intrinsic advantages, as discussed in section 3.2. What’s more, writing a script that provides recursive grepping will allow us to demonstrate some additional features of Perl that are worth knowing.

For starters, let’s observe a sample run of the `rgrep` script, whose code we’ll examine shortly. In the situation depicted, the Linux superuser was having trouble with a floppy disk, and knew that some file(s) in the `/var/log` directory would contain error reports—but he wasn’t sure which ones:

```
$ rgrep '\bfloppy\b' /var/log          # output edited for fit
/var/log/warn:
kernel: floppy0: data CRC error: track 1, head 1, sector 14
/var/log/messages:
kernel: I/O error, dev 02:00 (floppy)
```

These reports, which were extracted from the indicated files under the user-specified directory, indicate that the diskette was not capable of correctly storing data in certain sectors.<sup>16</sup> The script can be examined in listing 8.3.

### Listing 8.3 The `rgrep` script

```
1  #! /usr/bin/perl -wnl
2
3  BEGIN {
4      $Usage="Usage: $0 'pattern' dir1 [dir2 ...]";
5      @ARGV >= 2 or warn "$Usage\n" and exit 255;
6
7      $pattern=shift;    # preserve pattern argument
8
9      # `@ARGV` treated like "@ARGV"; elements space-separated
10     @files=grep { chomp; -r and -T } # <-- find feeds files
11     `find @ARGV -follow -type f -print`;
```

---

<sup>16</sup> Which is one reason this venerable but unreliable storage technology has become nearly obsolete.

```

12     @files or warn "$0: No files to search\n" and exit 1;
13     @ARGV=@files; # search for $pattern within these files
14 }
15 # Because it's very likely that we'll search more than one file,
16 # prepend filename to each matching line with printf
17
18 /$pattern/ and printf "$ARGV: " and print;

```

Because this script requires a pattern argument and at least one directory argument, the argument count is checked in Line 5 to determine if a warning and early termination are in order. Then, Line 7 shifts the pattern argument out of the array, leaving only directory names within it.

The `find` command on Line 11 appears within the back quotes of command interpolation, but these quotes are treated like double quotes as far as variable interpolations are concerned. The result is that `@ARGV` is turned into a series of space-separated directory names, allowing the Shell to see each as a separate argument to `find`, as desired. The `-follow` option of `find` ensures that arguments that are symbolic links (such as `/bin` on modern UNIX systems) will be followed to their targets (such as `/usr/bin`), allowing the actual files to be processed. The result is the conversion of the user-specified directories into a list of the regular files that reside within them (or their sub-directories), and the presentation of that list to `grep` as its argument list.

In Line 10, `grep` filters out the filenames emitted by `find` that are not readable text files.<sup>17</sup> But before applying the `-T` test to `$_`, which holds each filename in turn, `chomp` is employed to remove the newline that `find` appends to each filename.

Line 12 ensures that there's at least one searchable filename before proceeding, to avoid surprising the user by defaulting to `STDIN` for input—which would be highly unexpected behavior for a program that takes directory arguments!

Finally, Line 18 attempts the pattern match, and on success, it prints the name of the file—because multiple files will usually be searched—along with the matching line.

Although this script is useful and educational, you won't be seeing it again. That's because it will be assimilated by a grander, more versatile Perl grepper, later in this chapter.

### 8.5.3 Tips on using command interpolation

Perl's command-interpolation mechanism is different in some fundamental ways from the Shell's command substitution. For one thing, the Shell's version works within double quotes, allowing literal characters and variables to be mixed within the back-quoted command:

```

$ echo "Testing: `tput smul`Shell"
Testing: Shell

```

<sup>17</sup> The `-T` operator has to read the file to characterize its contents, so it doesn't return True unless the file is readable—making `-r` redundant. Accordingly, we won't show `-r` with `-T` from here on.

In contrast, Perl treats back-quotes within double quotes as literal characters, requiring individual components to be separately quoted:

```
print 'Testing: ', `tput smul`, 'Perl';
Testing: Perl
```

Another difference is that what's tested for a back-quoted command in conditional context is the True/False value of its *output* in Perl, but of the command's *exit value* in the Shell:

**Shell**

```
o=`command` || echo 'message' >&2 # warns if command's $? False
```

**Perl**

```
$o=`command` or warn "message\n"; # warns if output in $o False
```

You can arrange for Perl to do what the Shell does, but because the languages have opposite definitions of True and False, this involves complementing *command's* exit value. With this in mind, here's the Perl counterpart for the previous Shell example:

```
$o=`command` ; ! $? or warn 'message'; # warns if $? False
```

And here's the same thing written as an if:

```
if ($o=`command` ; ! $?) { warn 'message'; } # warns if $? False
```

As mentioned earlier, Perl has a simpler processing model than the Shell for quoted strings, which has the benefit of making the final result easier to predict.<sup>18</sup> One conspicuous side-effect of that tradeoff is Perl's inability to allow command interpolation requests to be nested within double quotes—but that's a compromise worth making.

Next, we'll talk about the `system` function, because no matter how richly endowed with built-in resources your programming language may be, you'll still want to run OS commands from it now and again.

## 8.6 EXECUTING OS COMMANDS USING *system*

In cases where you want to use an OS command in a way that doesn't involve capturing its output within the Perl program—such as simply displaying its output on the screen—the `system` function is the tool of choice. Table 8.8 shows sample invocations of `system`, which is used to submit a command to the OS-dependent command interpreter (`/bin/sh` on Unix) for execution.

---

<sup>18</sup> See [http://TeachMePerl.com/DQs\\_in\\_shell\\_vs\\_perl.html](http://TeachMePerl.com/DQs_in_shell_vs_perl.html) for further details.

**Table 8.8 The system function**

Example	Explanation
<code>system 'command(s)';</code>	<code>command(s)</code> in single quotes are submitted without modification for execution.
<code>system "command(s)";</code>	<code>command(s)</code> in double quotes are subjected to variable interpolation before being executed. In some cases, single quotes may be required around command arguments to prevent the Shell from modifying them.
<code>system 'command(s)'; ! \$? or warn 'failed';</code>	Just as “function or warn” reports the failure of a Perl function, “! \$? or warn” reports a failed command run by <code>system</code> . The “!” converts the Unix True/False value to a Perl-compatible one.

As indicated in the table, it’s important to carefully quote the command presented as `system`’s argument, because

- special characters within the command may otherwise cause Perl syntax errors;
- judicious use of single quotes, double quotes, and/or backslashes may be required to have the command reach the Shell in the proper form.

Let’s say you want to do a long listing on a filename that resides in a Perl variable. For safety, the filename should appear in single quotes at the Shell level, so if it contains whitespace characters, it won’t be interpreted as multiple filenames.

The appropriate invocation of `system` for this case is

```
system "ls -l '$filename'"; # filename contains: ruby tuesday.mp3
```

which arranges for the Shell to see this:

```
ls -l 'ruby tuesday.mp3'
```

The double quotes around `system`’s argument allow the `$filename` variable to be expanded by Perl, while ensuring that the single quotes surrounding it are treated as literal characters. When the Shell scans the resulting string, the (now unquoted) single quotes disable word-splitting on any embedded whitespace, as desired.<sup>19</sup>

As shown in the last row of table 8.8, when you need to test whether a `system`-launched command has succeeded or failed, there is a complication—on Unix, the value returned by `system` (and simultaneously placed in “\$?”) is based on the Shell’s definitions of True and False, which are the opposite of Perl’s.

The recommended workaround is to *complement* that return value using the “!” operator and then write your branching instructions in the normal manner. For example:

```
system "grep 'stuff' 'file'";  
! $? or warn "Sorry, no stuff\n";
```

<sup>19</sup> For a more detailed treatment of the art of multi-level quoting, see <http://TeachMeUnix.com/quoting.html>.

Next, we'll look at two programs that use `system` to format “news flashes” on the screen.

### 8.6.1 Generating reports

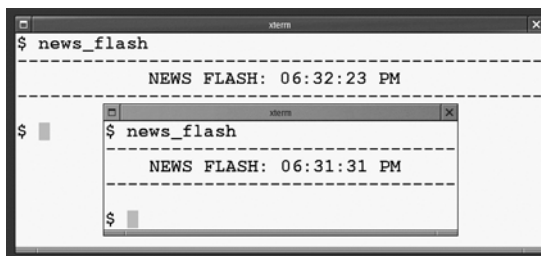
Instead of returning a string that adjusts the terminal's display mode, `tput`'s `lines` and `cols` options (see table 8.7, bottom panel) return the number of lines and columns on the terminal. The latter option is useful for drawing borders across the screen, among other things. These options also work with many terminal emulators (such as `xterms`) to report their current dimensions.

Figure 8.3 shows two sample runs of the `news_flash` script, which uses `tput` to print a heading across the width of the user's terminal. As you can see, the heading is centered within each of the differently sized windows, and the dashed lines occupy each window's full width.

Listing 8.4 shows the script.

#### Listing 8.4 The `news_flash` script

```
1  #!/usr/bin/perl -wl
2
3  $width=(tput cols or 80); # supply a reasonable default
4  $line='- ' x $width;      # make a line the width of screen
5
6  $heading='NEWS FLASH:';
7  $heading.=' ' . `date +%X`; # append date's formatted output
8  chomp $heading;          # remove date-added newline
9
10 # Calculate offset from left, to center the string
11 $heading_length=length $heading;
12 $offset=($width - $heading_length) / 2;
13
14 # Offset may have decimal component, but Perl will
15 # convert to integer automatically for use with "x" operator
16 $padding=' ' x $offset; # generate spaces for calculated offset
17
18 print "$line";          # dashed line
19 print "$padding$heading"; # the centered heading
20 print "$line\n";       # dashed line
```



**Figure 8.3**  
Output from the `news_flash` script

```

$ news_flash2 peace
-----
NEWS FLASH: 06:48:58 PM
-----
This just in -- To the surprise of all, PEACE has broken
out between the warring factions. Startled politicians have
convened a summit in Geneva to discuss ways of coping with
this unexpected development.
$
$ news_flash2 peace
-----
NEWS FLASH: 06:50:21 PM
-----
This just in -- To the surprise of all, PEACE has
broken out between the warring factions. Startled
politicians have convened a summit in Geneva
to discuss ways of coping with this unexpected
development.
$

```

**Figure 8.4** Output from the `news_flash2` script

The script uses command interpolation rather than the implicit loop to obtain the information it needs, so it doesn't need the `n` option or its associated `BEGIN` block. It works by constructing a dashed line that's the width of the screen (Lines 3–4), building a heading string (Lines 6–8), determining its length (Line 11), calculating the offset needed to center it (Line 12), and then printing the left-padded heading string (Line 19) between dashed lines (Lines 18, 20).

By being more creative in the use of `tput`, and with a little help from `system`, we can make the heading look even fancier. Figure 8.4 shows an enhanced version, which uses both reverse video and underlining to decorate the heading, and formats the text of a news article to fit within the screen width.<sup>20</sup>

The script is shown in listing 8.5.

As indicated in the shebang line, the script supports a command-line switch (via `-s`) called `-debug`, which is declared on Line 3. It checks for a first argument that's the name of a text file in Line 6, and it issues a "Usage:" message and `die`s if it doesn't get one.

**Listing 8.5** The `news_flash2` script

```

1  #! /usr/bin/perl -s -wl
2
3  our ($debug);           # make switch optional
4
5  $file=shift;           # get filename of news article
6  if (! defined $file or ! -T $file) {
7      die "Usage: $0 filename\n";
8  }
9

```

<sup>20</sup> The `Text::Autoformat` module could reformat the string, but our emphasis in this section is on demonstrating the use of Shell-based processing options rather than pure-Perl ones.

```

10 # Get the display control sequences
11 $REV=( `tput smso` or ""); # use null string by default
12 $NO_REV=( `tput rmso` or "");
13 $UL=( `tput smul` or "");
14 $NO_UL=( `tput rmul` or "");
15
16 # Get the terminal's width
17 $width=( `tput cols` or 80); # supply standard default
18 chomp $width; # remove tput's newline
19 $line='- ' x $width; # make a line the width of screen
20
21 $heading='NEWS FLASH: '; # store heading string
22 $date=`date +%X`; # store date string
23 chomp $date; # remove date's newline
24
25 # Calculate needed offset from left, to center the string
26 $msg_length=length "$heading $date";
27 $offset=($width - $msg_length) / 2;
28
29 # Offset may have decimal component, but Perl will
30 # convert to integer automatically for use with "x" operator
31 $padding=' ' x $offset; # generate spaces for calculated offset
32

```

Then the script sets some variables for controlling the user's display, taking into account the possibility that `tput` might not succeed in obtaining the requested display-control string for the user's terminal. Specifically, for each display attribute, `tput`'s return value is tested for being `False` (to detect the "undefined" value), in which case a null string is assigned to the variable instead.<sup>21</sup> This allows those variables to be used without triggering any warnings about uninitialized values, with null-strings standing-in for any requested (but unavailable) display-control strings.

The parentheses are needed in those assignments (Lines 11–14) because the assignment operator has higher precedence than the logical `or`. In consequence, `tput`'s output would be assigned to each variable directly without them, leaving the `or ""` portions just dangling there uselessly.

In the case of the `$width` variable, we can do better than fudging in a null string for its value if `tput cols` returns `False`, so it's set in Line 17 to the standard width of a terminal.

The two parts of the heading line are stored in variables on Lines 21 and 22. They're kept separate so that the different display-control sequences can later be inserted around them (Lines 34–35). After the usual calculations are performed to center the heading string, it's printed between the dashed lines generated on Lines 33 and 36.

---

<sup>21</sup> There's no need in this case to check `tput`'s output for being merely `defined` as opposed to `True`, because the number 0 could never be a display-control string anyway. But here's how the `defined`-based coding would look:

```
$REV=`tput smso`; defined $REV or $REV="";
```

```

33 print $line;                                # dashed line
34 print $padding, $REV, $heading, $NO_REV, " ",
35     $UL, $date, $NO_UL;                    # the heading
36 print $line;                                # dashed line
37
38 # Assemble command in string
39 $command="fmt -$width '$file'"; # e.g., "fmt -62 Reuters.txt"
40
41 $debug and warn "Command is:\n\t$command\n\n" and
42     $command="set -x; $command"; # enable Shell execution trace
43
44 system $command;                            # format to fit on screen
45
46 # show error if necessary
47 ! $? or warn "$0: This command failed: $command\n";

```

The next step (Line 39) is to construct the Shell command that reformats the contents of `$file` to fit within the terminal's width, using the Unix `fmt` command. As you'll see in an upcoming example, storing the command in a variable is better for debugging purposes than passing the command as a direct argument to `system`. Note that `$file` is placed between Shell-level single quotes, to guard against the possibility that it may contain characters that are special to the Shell.

The command is executed on Line 44. If it fails, a warning is issued on Line 47.

The `news_flash2` script's use of `system` to run `fmt file` is appropriate, because it lets the command's output flow to the screen. However, if a script needs to repeatedly reenact reverse-video type, it's more economical to run `tput smso` once using command interpolation and save its output for later reuse, than to repeatedly run `system 'tput smso'`.

In the next section, we'll first discuss some general techniques for debugging Shell commands issued by Perl programs, and then you'll learn how to debug an actual problem that once afflicted `news_flash2`.

## 8.6.2 Tips on using `system`

The first and most important tip on successfully using `system`, as mentioned before, is to make sure you provide the necessary quotes at the Shell level to allow your command to be interpreted correctly. But no matter how hard you try, you may mess that up, so an even more important tip is to write your script with ease of debugging in mind.

In listing 8.5, you may have wondered why the `system` command for `news_flash2` was coded (Lines 39, 44) as

```

$command="fmt -$width '$file'";
...
system $command;

```

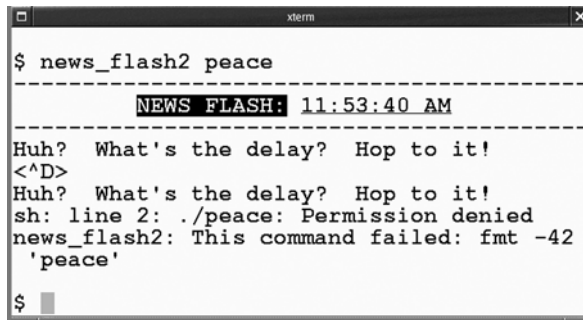
rather than more directly as

```
system "fmt -s$width '$file'";
```

Using a variable to hold `system`'s command has much to recommend it, because it facilitates

- printing the text of the command for inspection before running it (Line 41);
- showing the text of the command in a diagnostic message (Line 47);
- conditionally enabling the Shell's execution-trace mode for debugging purposes (Lines 41–42).

As a case in point, when I first tested this program, one of the lines currently visible in listing 8.5 had gone missing (as in all such cases, I blamed `vi`—*bad editor!*). This caused the script to print the heading, and then just *stall* (or *hang*), as shown in Figure 8.5.



```
$ news_flash2 peace
-----
NEWS FLASH: 11:53:40 AM
-----
Huh? What's the delay? Hop to it!
<^D>
Huh? What's the delay? Hop to it!
sh: line 2: ./peace: Permission denied
news_flash2: This command failed: fmt -42
'peace'
$
```

**Figure 8.5**  
Strange behavior of the  
`news_flash2` script

I've been goofing up commands on Unix systems for decades—so I know that when a program appears to be stalled, it's not taking a siesta, but waiting patiently for the user to type some input. So, I typed the “Huh?” line shown in figure 8.5 and pressed `<^D>`. That same line was immediately sent back to the screen, followed by a message indicating that the permissions were incorrect for the file `peace`. So I checked to see if that file was readable by me, and indeed it was!

Examining the output more carefully, I took comfort in seeing that the last line on the screen confirmed that the command, `fmt -42 peace`, was indeed the one I intended—even though the filename `peace` had wrapped around at the screen boundary (more on this in a moment).

Because I had written the script with debugging in mind, my next step was to run it again using the `-debug` switch, as shown in figure 8.6.

The first thing that caught my eye was the strange formatting of the output under “Command is:”. Checking the listing (Line 41), I confirmed that there was supposed to be a newline-tab combination before the command appeared, but there definitely shouldn't have been a newline between the `-42` option to `fmt` and that command's

```

xterm
$ news_flash2 -debug peace
-----
NEWS FLASH: 11:55:20 AM
-----
Command is:
  fmt -42
'peace'
+ fmt -42
What treachery doth this portend?
<^D>
What treachery doth this portend?
+ peace
sh: line 2: ./peace: Permission denied
news_flash2: This command failed: set -x;
fmt -42
'peace'
$

```

**Figure 8.6**  
Debugging the  
`news_flash2` script

intended filename argument. The Shell sees that as two separate commands, rather than one with a final argument of 'peace'.

The Shell's *execution-trace mode*,<sup>22</sup> which was enabled by the `-debug`-based code prepending `set -x; to $command` (Line 42), showed that the (“+”-prefixed) command being executed was just `fmt -42`. And that’s the problem: In the absence of the intended filename argument, `fmt` was waiting for input from STDIN! After I signaled the end of input with `<^D>`, the Shell tried to run a second command whose name was `peace`, while reading from “line 2” of its input.

Finally—making the script’s defect even more apparent—the “command failed” warning hit the screen-edge differently this time, allowing me to clearly see that the filename argument (`peace`) had a newline before it within `$command`.

The reported “permission problem” was now also clear, signifying that `peace` wasn’t *executable*, as any self-respecting command (which it isn’t) would be. I felt a momentary urge to set its execute bit (impulsive attempts for quick fixes can be *so* appealing), but I decided it would be wiser to deal with the pesky newline between `$width` and the filename instead, which was the real culprit.

The remedy is to insert the `chomp $width` statement shown on Line 18 of listing 8.5, which wasn’t there during my initial testing. Doing so removes the trailing newline from the variable if `trput`’s output initializes it, without doing any harm to the newline-free value of 80 if that is used as the initializer instead.

You can find other tips on using `system`, including techniques for increasing security by preventing `/bin/sh` from interpreting its arguments, and for recovering

---

<sup>22</sup> Execution-trace mode shows the (potentially modified) text of the original command after it’s been subjected to nearly a dozen stages of processing, allowing the programmer to see the actual command that’s about to be executed. Perl’s processing model is much simpler, obviating the need for any similar mechanism.

the actual Shell exit codes from the values that are encoded in “\$?”, by running `perldoc -f system`.

One of the most useful and powerful services that a script can provide is to compile and execute programs that it constructs on the fly—or that the user provides—while it is already running. We’ll discuss Perl’s support for this service next.

## 8.7 EVALUATING CODE USING `eval`

Like the Shell, Perl has a code-evaluation facility. It’s called `eval`, and its job is to compile and execute (i.e., “evaluate”) Perl code that becomes available during a program’s execution.

That might sound like a description of the `nexpr_p` script we discussed in chapter 5, which evaluates an expression formed from arguments supplied by the user, but there’s a big difference.

To jog your memory, here’s `nexpr_p`:

```
#!/bin/sh
perl -wl -e "print $*;" # nexpr_p '2 * 21' --> print 2 * 21
```

It uses the Shell to construct a Perl program, which Perl runs in the usual way. As an alternative, the Perl program could have been designed to accept the specification for the desired program as an argument and to run it on its own. That’s where the judicious use of `eval` would be required.

Why is `eval` needed in such cases? Because programs contain special keywords and symbols that are only recognized during the program’s compilation phase, which has completed by the time the program starts running. Accordingly, the benefit of `eval` is that it lets your running program handle code that wasn’t present during that program’s initial compilation.

Examples of tokens that require `eval` for recognition are keywords, operators, function names, matching and substitution operators, backslashes, quotes, commas, and semicolons. Table 8.9 shows the syntax for `eval` in both the Shell and Perl.

**Table 8.9** The `eval` function in the Shell and Perl

Shell	Perl
<code>eval 'command'</code> <code>error=\$?</code> <code>(( \$error &gt; 0 )) &amp;&amp;</code> <code>echo "failed: \$error" &gt;&amp;2</code>	<code>eval 'stuff'; # sets \$@</code> <code>\$@ ne "" and</code> <code>warn "failed: \$@";</code>
N/A	<code>eval; # evaluates code in \$_</code>

A similarity is that `eval`’s argument is shown in quotes for both languages in table 8.9, because proper quoting—with single quotes, double quotes, and/or backslashes—is often required for success.

Some differences are that the Shell provides an integer exit code for the `eval`’d command, whereas Perl provides a null string or a diagnostic message in the “\$@”

variable. Also, there's no need to make a copy of Perl's "\$@" to avoid losing access to it, because it's not overwritten in Perl by every subsequently evaluated expression—as "\$?" is in the Shell by every subsequently executed command. Another difference is that only Perl allows the invocation of `eval` without an argument (as shown in the table's bottom panel), in which case it defaults to using `$_`.

We'll look next at a simple yet surprisingly powerful application of `eval` that can be used to good advantage by every JAPH. It's based on a script I developed for one of our training courses (like most of the examples in this book).

### 8.7.1 Using a Perl shell: The `psh` script

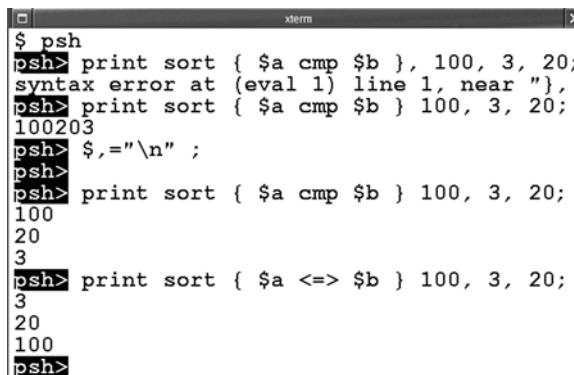
The `psh` script is a dramatic example of how easy it can be to write useful scripts with `eval` that could never work without its help. `psh`, a *Perl shell*, prompts the user for a Perl statement, compiles and executes that statement, returns any errors that it generated, and then continues the cycle until `<^D>` or `exit` is entered. Its major benefit is that it lets you quickly try some Perl code without writing a script first to do so.

Some of `psh`'s applications are as follows:

- Rapidly developing a prototype of a small program
- Determining the proper syntax for a particular language feature
- Helping teachers demonstrate the error messages associated with certain mistakes

A sample `psh` session is shown in Figure 8.7.

The student whose `psh` session is depicted in the figure was learning how to code the printing of a sorted list of numbers. The syntax error that Perl reported was caused by the understandable tendency to place a comma after `sort`'s code-block argument (see table 7.10). She corrected that mistake in the next line, only to find that the numbers were being squished together on output, which reminded her to set the "\$," variable to a suitable separator. But this adjustment revealed that `sort`'s arguments were being treated as strings rather than as numbers, leading to the replacement of the string comparison operator by its numeric counterpart, `<=>` (see table 5.11).



```
$ psh
psh> print sort { $a cmp $b }, 100, 3, 20;
syntax error at (eval 1) line 1, near \",'
psh> print sort { $a cmp $b } 100, 3, 20;
100203
psh> $, = "\n" ;
psh>
psh> print sort { $a cmp $b } 100, 3, 20;
100
20
3
psh> print sort { $a <=> $b } 100, 3, 20;
3
20
100
psh>
```

**Figure 8.7**  
The Perl shell as a learning tool

Voilà! The student figured out how to print a sorted list of numbers without engaging in the cycle of repeatedly editing a file, saving it, and submitting it for execution. That makes `psh` a valuable tool for explorations of this type.<sup>23</sup>

Listing 8.6 shows the `psh` script. As with many sophisticated and powerful Perl programs, there's almost nothing to it!

**Listing 8.6** The `psh` script

```
1  #!/usr/bin/perl -wnl
2
3  BEGIN {
4      $ON=`tput smso`;
5      $OFF=`tput rmso`;
6      $prompt="{ON}psh>$OFF ";
7      printf $prompt;          # print initial prompt
8  }
9
10 eval;    # uses $_ as argument, loaded by -n loop
11
12 $? ne "" and warn $?;      # if eval produced error, show it
13
14 printf $prompt;          # print prompt for next input
15
16 END {
17     # If user pressed <^D> to the prompt, which leaves $_
18     # undefined, we need to print a newline so the shell's
19     # prompt will start on a fresh line.
20
21     ! defined and print "";  # -l appends newline after ""
22 }
```

In Lines 4–5, the back quotes of command interpolation are used around each `tput` command to capture its output for assignment to a variable. Then those variables are used to render `psh`'s prompt in reverse video (Line 6), to make it stand out (see figure 8.7).

Line 7 issues the first prompt. Then, for each line of input the user provides, `eval` compiles and executes it (Line 10). If necessary, a diagnostic message is printed on Line 12.

The prompt for the next input is then issued (Line 14), and the cycle of reading and evaluating input continues until `<^D>`—or an input consisting of `exit`—requests termination. For the `exit` case, the user-supplied `<ENTER>` serves to position the

---

<sup>23</sup> In contrast, the somewhat similar Perl debugger (invoked via `-d`) has some very non-Perlish properties, which—unlike `psh`—may require you to stop thinking Perlishly when you use it! For this reason, I prefer to debug most problems using `print` statements for information gathering (as Larry does)—with an occasional dash of `psh`.

Shell's upcoming prompt on a fresh line of the terminal, but with `<^D>`, the program must supply a newline on its own. That's handled on Line 21, where the programmer saved some typing by exploiting the fact that `$_` is the default argument for `defined`.

And now, without further ado, it's time to reveal our much-ballyhooed comprehensive Perl grepper—which sports a name worthy of an *obstetrician's pet ferret*.

## 8.7.2 Appreciating a multi-faceted Perl grepper: The `preg` script

There's no getting around the fact that `grep` is one of the most important and popular Unix utilities, despite its limitations. That's why we've discussed so many Perl programs that behave like `grep` or one of its cousins, `fgrep` and `egrep`.

But do we really need to carry all of `greperl`, `text_grep`, `perlgrep`, and `rgrep` around in our toolkits?<sup>24</sup> That's too much of a good thing. Wouldn't it be better to have a single script that could provide the services of any of those specialized versions through options? It sure would, but there's a reason you haven't seen that script thus far—`eval` is needed to make it work.

`preg`, short for “**P**erlish **r**elative of **e**nhanced **g**rep”, is a veritable “Swiss Army knife” of Perl greppers compared to what you've seen thus far. It supports the following options, each of which enables a special kind of `grep`-like (or *grep-eclipsing*) functionality:

---

<code>-f:</code>	<b>f</b> grep style; disable metacharacters in the pattern.
<code>-i:</code>	<b>I</b> gnore case differences.
<code>-l:</code>	<b>L</b> ist filenames that have matches (not their matching records).
<code>-v:</code>	Only show records that don't contain matches.
<code>-n:</code>	Prepend record <b>n</b> umbers to the records that are shown.
<code>-d:</code>	<b>D</b> isplay matches within their records using screen's standout mode.
<code>-p:</code>	<b>P</b> aragraph mode; use blank lines as record separators.
<code>-c='S':</code>	<b>C</b> ustom delimiter mode; use string <i>S</i> as a record separator.
<code>-m:</code>	<b>M</b> ulti-line mode; <code>^</code> and <code>\$</code> match ends of lines, not ends of record (used with <code>-p</code> or <code>-c</code> ).
<code>-s:</code>	<b>S</b> ingle-line mode; <code>"</code> <code>.</code> matches newline.
<code>-r:</code>	<b>R</b> ecursive search; descend into arguments that are directories.

---

Here are some sample runs, matching against a short version of the UNIX “fortune cookie” file, to give you an idea of `preg`'s versatility:

---

<sup>24</sup> Respectively covered in sections 3.13.2, 6.4.1, 8.2.1, and 8.5.2.

```

$ preg -l      'eve'      fortunes # just list filename
fortunes

$ preg        'eve'      fortunes # show matching record
Eagles may soar, but weasels never get sucked into jet engines.

$ preg -d     'eve'      fortunes # display match in record
Eagles may soar, but weasels never get sucked into jet engines.

$ preg -p -d  'for.*the' fortunes # paragraph mode; no match

$ preg -p -d -s 'for.*the' fortunes # now . can match newline
The problem's solution is trivial and is left as an exercise for
the reader.

$ preg -p -d -m '^t\w+'  fortunes # display initial t-words
The solution of this problem is trivial and left as an exercise for
the reader.

$ preg -r -i '\bWaldo\b|\bGodot\b' testing # search directories
testing/waiting_place:Mr. Godot ain't here yet!
testing/hiding_place:waldo is virtually invisible.

```

The output of the last command indicates that Waldo and also Godot (no relation to <http://godot.com>) were found in the indicated files that reside under the directory `testing`.<sup>25</sup>

Now it's time to ogle the code! The script is shown in listing 8.7.

#### Listing 8.7 The `preg` script

```

1  #!/usr/bin/perl -s -wnl
2  our ($f, $i, $l, $v, $n, $d, $p, $m, $s, $r); # switch vars
3
4  BEGIN {
5      $Usage="Usage: $0 [-f] [-i] [-l] [-v] [-n] [-d]";
6      $Usage.=" [-p|-c] [-m] [-s] [-r] 'RE' [file...]\n";
7
8      # Must at least have pattern argument
9      @ARGV > 0 or warn "$Usage" and exit 255;
10
11     # Can't have mutually-exclusive switches
12     defined $p and defined $c and
13     warn "$Usage\n\tCan't have -p and -c\n" and exit 1;
14
15     $X='g'; # set modifier to perform all substitutions
16     $ON=$OFF=""; # by default, don't highlight matches
17
18     if ($d) { # for match-displaying with -d
19         $ON=(`tput smso` or ""); $OFF=(`tput rmso` or "");

```

<sup>25</sup> See sections 3.2.3 and 3.3.1 for information on what various `grep` commands do when given directory arguments.

```

20     };
21
22     $p and $/= "";      # paragraph mode
23     $c and $/= $c;     # custom record separator mode
24     $i and $X.='i';    # ignore case; add to modifiers in $X
25     $m and $X.='m';    # multi-line mode
26     $s and $X.='s';    # single-line mode
27
28     $pattern=shift @ARGV;      # remaining args are filenames
29     $f and $pattern='\Q' . $pattern . '\E'; # "quote" metachars
30
31     $r and @ARGV=grep { chomp; -T }
32     `find @ARGV -follow -type f -print`;
33     $multifiles=@ARGV > 1;    # controls "filenames:match" format
34
35     $matcher="s/$pattern/$ON\${&&$OFF}/$X";
36     $v and $matcher="! $matcher"; # complement match result
37 }
38 ##### BODY OF PROGRAM, EXECUTED FOR EACH LINE OF INPUT #####
39 $found_match=eval $matcher;    # run sub-op, to try for match
40 if ( $@ ne "" ) {              # show eval's error
41     warn "\n$0: Eval failed for pattern: '$matcher'\n\n";
42     die "Perl says:\n$@\n";
43 }
44 elsif ( $found_match ) {
45     if ($!) { print $ARGV; close ARGV; }      # print filename\n
46     elsif ($multifiles) { printf "$ARGV:"; } # print filename:
47     if (! $!){                             # don't show match if listing filenames
48         $n and printf "$.:"; # prepend line number to record
49         print;                # show selected record
50         $p and print "";      # separate paragraphs by blank line
51     }
52 }

```

The shebang line includes the `s` option for automatic switch processing, which is used heavily in this script—so it begins by calling `our` on all the variables corresponding to its optional switches.

In the `BEGIN` block, the `$Usage` variable is constructed in two steps to keep the lines short, with the help of the compound-assignment version of the concatenation operator.

Like `grep`, this script requires at least a “pattern” argument, so it issues a warning and exits on Line 9 if it’s missing. In similar fashion, Line 12 checks for an attempt to simultaneously use mutually exclusive switches, and `exits` the script if appropriate.

Line 15 initializes the variable that holds the match modifiers with the `g` (for `/g`) that is the script’s default, in preparation for other modifiers being appended to it later.

If the match-displaying switch was chosen, the screen-highlighting variables initialized to null strings in Line 16 are overwritten in Line 19 with the appropriate terminal escape sequences (see section 8.5.1 for details on `tput`).

Lines 22–26 check for switches, and set their associated variables as needed.

Line 28 extracts the pattern argument from `@ARGV`, leaving only filenames as its contents (or nothing at all, to obtain input from `STDIN`).

If the `-f` (fixed-string, like `fgrep`) switch was chosen, the next line places `$pattern` between `\Q` and `\E` metacharacters, to render any metacharacters within it as literal. The concatenation operator is a good choice here, because the double-quoted alternative is more error prone, requiring doubled-backslashes to get single ones into the variable:

```
$pattern="\Q$pattern\E"; # stores: \Q ... \E
```

If the “recursive” switch was used, Line 32 uses `find`<sup>26</sup> to convert any arguments that are directories into a list of the regular files residing within or below them, while preserving any arguments that are regular files. Then `grep` extracts the text-files from that list (Line 31) and stores the results back in `@ARGV`.

For example, given initial arguments of

```
/home/plankton
```

and

```
/tmp/neptunes_crown.txt
```

the final contents of `@ARGV` might become

```
/home/plankton/todo_list.doc
```

and

```
/home/plankton/world_domination_plan.stw
```

along with the originally specified

```
/tmp/neptunes_crown.txt
```

Line 33 sets a variable to `True` or `False` according to the number of filenames to be searched, so that `grep`’s functionality of prepending filenames to matched records can be conditionally provided in Line 46.

Line 35 assembles the text of the expression that will do the matching. A substitution operator is used, because matching-plus-replacement is required to support the “display matches” feature of the `-d` switch, and it can also handle the other cases.

---

<sup>26</sup> The use of command interpolation to provide arguments to a script, as in `script `find ...``, is subject in extreme cases to buffer-overflow problems requiring an `xargs`-style remedy (as discussed in section 6.5). However, in this case, the output of `find` is delivered to the already-running process through other channels, so these concerns don’t apply.

For example, with `-d`, the substitution operator replaces the match by the values of `$ON` and `$OFF` to highlight the match; without `-d`, it replaces the match by itself (`$&`), because `$ON` and `$OFF` have null values in that case. Either way, the substitution operator returns a True/False value on Line 39 to indicate the success or failure of the substitution (and, by extension, the match), and that's used to control the reporting of results (on Line 44).

Back in Line 35, the backslashing of the `$&` variable's dollar symbol delays the variable's expansion until the matching text has been stored within `$&`, which happens during the `eval` on Line 39. In contrast, the values of the `$pattern`, `$ON`, `$OFF`, and `$X` variables are immediately placed into the string on Line 35, because they're already available and won't change during execution. Also notice that the contents of `$X` are appended after the substitution operator's closing delimiter, which is how the script communicates the user's choices for match modifiers to that operator.

Line 36 prepends the complementing operator (`!`) to the contents of `$matcher`, if the (grep-like) `-v` switch for displaying non-matching lines was supplied.

And that's the end of the `BEGIN` block! Now all that remains to be done is—the real work of the program.

On Line 39, our old friend `eval` is used to evaluate the string representing a substitution operator that's stored in `$matcher`. This is required because it includes tokens that can only be recognized as having special meanings at compile-time. The substitution operator returns the number of substitutions it performed, which equals the number of matches it found, and `eval` assigns that value to `$found_match`. This allows the code block following `elsif ($found_match)` on Line 44 to be executed only if a match is found.

But first, Line 40 checks whether `eval` had problems with the `$matcher` code. If it did, the offending code is displayed, along with Perl's diagnostic message. This situation shouldn't arise as long as the user provides a syntactically correct pattern and the script assembles the substitution operator properly, but to make sure the diagnostic message doesn't go unnoticed, `die` terminates the script after printing the message found in `$@`.

If a match was found, the conditional statements starting on Line 44 are evaluated next; if not, the next input record is fetched, and the processing cycle continues from Line 39.

On Line 45, if the “filename listings” switch was provided, the filename (only) for the matched record is printed. As with `grep`'s `-l` option, the filename should only be printed once, no matter how many matches it might contain. This means there's no point in looking for additional matches after finding this first one, so the current filehandle is closed, which triggers the opening of the next file named in `@ARGV` (if any; see section 3.8).

Line 46 checks whether multiple files are to be searched; if so, it prints the matched record's filename followed by a colon (just as `grep` does), with no following newline (thanks to `printf`).

Line 47 ensures that “filename listing” mode isn’t in effect, because if it is, all the output for the current match is already on the screen. Then, if the user asked for line numbers, the current record’s number is printed using `printf`. The matched record itself is then printed, on the same physical line to which the `printf` on Line 46 may have already made a contribution.

If paragraph mode is in effect, Line 50 prints a blank line to provide separation between this record and the next.

It may all sound rather complicated when scrutinized at this level, but don’t lose sight of how straightforward the processing is in the simplest situation, when there are no switches and only one input file. In that case, after finding a True result on Line 44, Line 49 prints the record. Then, the next input record is fetched, and the cycle continues from Line 39.

I use this multi-faceted, Perlshly enhanced grepper all the time, and I trust you’ll find it as useful as I do. But you need to be aware of a few gotchas, which are covered next.

### **Tips on using `preg`**

Consider this attempt to run `preg` with an improperly constructed pattern argument, and the associated diagnostic message:

```
$ ps auxw | preg '?' # syntax error; ? needs \  
preg: Eval failed; sub-op is: 's/?/$&/g'  
Perl says:  
Quantifier follows nothing in regex; marked by   
  <-- HERE in m/? <-- HERE / at (eval 1) line 1, <> line 1.
```

What’s happening? Well, when `eval` fails while compiling and/or running the code it was given, `preg` displays the text of the substitution operator (sub-op) it constructed, because that’s where the error has got to be. `preg` also shows Perl’s diagnostic message, and, as you can see, the `<--HERE` pointer makes it abundantly clear that the quantifier metacharacter “?” caused the trouble. Why? Because when it appears in the substitution operator’s regex field, it’s supposed to be preceded by the element whose quantity it’s specifying—but it wasn’t.<sup>27</sup>

To fix the problem, you can either backslash the “?” to remove its special meaning as a quantifier or use the `-f` switch, which enables `fgrep`-like automatic quoting of all metacharacters:

```
$ ps auxw | preg -f '?' # show terminal-less processes; edited  
root    1  0.0  0.0  448   64  2  S   Dec08   0:04  init  
root   411  0.0  0.0 1356  220  2  S   Dec08   0:09  /sbin/syslogd  
bin    586  0.0  0.0 1292    0  2  SW  Dec08   0:00  /sbin/portmap  
root   795  0.0  0.0 1416  128  2  S   Dec08   0:00  /usr/sbin/cron  
...
```

---

<sup>27</sup> For example, `x?` would allow 0 or 1 occurrences of `x`; see table 3.9 for details.

The command shows reports on processes that are not attached to terminals, as indicated by the “?” character in the seventh field.

One more word of caution on using `preg`: If your pattern includes a slash character, you’ll have to backslash it, even when using the `-f` switch, because `preg` uses that character as the delimiter for the substitution operator itself:

```
preg -f 'TCP/IP?'          # WRONG!
preg -f 'TCP\|IP?'        # Right.
```

Now you’re ready to use this powerful and OS-portable grepper, in place of: `grep`, `fgrep`, `egrep`, `rgrep`, `text_grep`, `greperl`, and `perlgrep`<sup>28</sup>!

## 8.8 SUMMARY

Perl provides a variety of tools that are principally used in the kinds of programs that are sufficiently large to be worth storing in a file, which we call *scripts*.

You saw how `defined` is used to test for the mere existence of a value, independently of its True/False status. As a general rule, you should always use `defined` to ensure that a variable that might be unset actually has a value, before you attempt to use that value. We reviewed examples showing how `defined` is used with other operators in validating arguments (e.g., as readable text files) and in proofreading interactive input from users (e.g., as a confirming “Yes”).

We discussed how non-filenames—such as a *pattern*—can coexist with filenames in a script’s argument list, as long as the non-filenames are removed from `@ARGV` by `shift` in a `BEGIN` block. An advantage of Perl’s `shift` over the Shell’s is that it returns the value being removed from the array, making it easy to preserve that value for later access.

You also learned how to *pre-process* script arguments in various ways, such as using `grep` to filter out the ones that have the wrong properties, and using `sort` to reorder them. The *generation* of filename arguments, by using the globbing operator in a `BEGIN` block, was also illustrated.

Perl’s `exit` does the same basic job as the Shell’s, but it differs by returning the “success” code of 0 by default.<sup>29</sup>

Perl’s `die` function, which is like `warn 'message'` coupled with `exit 255`, provides an alternative to `exit` for terminating a script. In cases where a custom exit code is desired, you can set “\$!” to the required number before calling `die`, to override the default of 255.

---

<sup>28</sup> You can download this script, along with the others featured in this book, at this web site: <http://manning.com/maher>.

<sup>29</sup> This is appropriate because, IMHO, Perl—as a better-designed, easier-to-use, harder-to-abuse, and more DWIMerrific language than the Shell—has more reason to be optimistic about its scripts completing successfully than the Shell does!

We showed the limitations of the logical `and/or` for controlling non-trivial conditional execution requirements, and how to control multi-way branching in such cases by using `if/else` instead. The `cd_report` script demonstrated the use of `if/else` to handle a program's large-scale branching needs, while using `and/or` to construct compound tests, so that the unique benefits of each facility could be realized.

The string-concatenation operator comes in handy for joining short strings into longer ones, which can make programs easier to read and maintain. It's also used to store newly acquired data after existing data, by appending to the contents of a variable. We demonstrated that technique in an upgrade to `most_recent_file`, which allows it to properly handle files whose modification times are tied.

You learned about a special benefit of the compound-assignment form of the concatenation operator—it allows the statements that were used to build up the contents of a variable to be easily reordered, if desired, to change the order in which the substrings are loaded. This capability was exploited in the example that promises to help web surfers MAKE MONEY FAST.

You can easily live without the string-repetition operator, but only if you're willing to do a lot of unnecessary typing to do so. It specializes in replicating strings, and you saw it used to construct a string consisting of a series of dashes, which was displayed across the width of a terminal.

Command interpolation and `system` are valuable tools that let Perl run commands provided by the host OS. You saw how judicious use of these tools obviates the need to re-invent the functionality of existing OS resources, at the expense of reducing a program's portability to other OSs. Command interpolation is used when a Perl script needs to capture an OS-command's output so it can be manipulated in some way. In contrast, `system` is used when it's sufficient merely to run the command without any access to its output.

A Unix command that's especially useful for scripts running on terminals is `tput`, which—by making it easy to control display modes such as reverse video, underline, and bold—can really spruce up an otherwise drab display of characters on the screen. `tput` can also report the terminal's current height and width, which is valuable information for scripts running in re-sizable windows. This capability was used by `news_flash` to draw full-width lines across windows having uncertain dimensions.

During our examination of `news_flash2`, we discussed the Shell's execution-trace mode, which shows exactly what each command looks like before it's executed. You saw how enabling this facility in a Perl script can help debug mysterious problems with commands run by `system`.

Like its Shell namesake, Perl's `eval` function is a powerful utility for compiling and executing code that's acquired or manufactured during a script's own execution. Through use of `eval`, programs can be endowed with advanced capabilities—as illustrated by `psh`, the Perl shell, and `preg`, a multi-faceted Perl grepper. Keep `psh` in mind during your further adventures with Perl, because it will come in handy when

you want to quickly try out some Perl code—*without* the bother of first creating a script to do so.

### Directions for further study

To obtain information about specific Perl functions covered in this chapter, such as `defined`, `exit`, `shift`, or `system`, you can either browse through the output of `man perlfunc` (not recommended) or use `perldoc` to zero in on the documentation for each specific function:

- `perldoc -f function-name # coverage of "function-name"`

For additional information on other topics covered here, you may wish to access these online resources:

- `man perlop # operators, and command interpolation`
- `man perlsyn # basic syntax, including if/else`
- `man tput # command that retrieves terminal information`
- `man Term::ANSIColor # module for coloring terminal text`