Learning Perl the Hard Way
# Contents

1 Arrays and Scalars 1

1.1 Echo .......................................................... 1
1.2 Errors ......................................................... 3
1.3 Subroutines .................................................. 4
1.4 Local variables .............................................. 4
1.5 Array elements .............................................. 4
1.6 Arrays and scalars ......................................... 5
1.7 List literals .................................................. 6
1.8 List assignment ............................................. 6
1.9 The `shift` operator ....................................... 7
1.10 File handles ............................................... 7
1.11 `cat` ......................................................... 8
1.12 `foreach` and `@` ......................................... 9
1.13 Exercises .................................................. 10

2 Regular expressions 11

2.1 Pattern matching ........................................... 11
2.2 Anchors ..................................................... 12
2.3 Quantifiers .................................................. 12
2.4 Alternation .................................................. 13
2.5 Capture sequences ........................................ 14
2.6 Minimal matching ......................................... 14
2.7 Extended patterns ......................................... 15
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8 Some operators</td>
<td>15</td>
</tr>
<tr>
<td>2.9 Prefix operators</td>
<td>16</td>
</tr>
<tr>
<td>2.10 Subroutine semantics</td>
<td>17</td>
</tr>
<tr>
<td>2.11 Exercises</td>
<td>18</td>
</tr>
<tr>
<td>3 Hashes</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Stack operators</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Queue operators</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Hashes</td>
<td>20</td>
</tr>
<tr>
<td>3.4 Frequency table</td>
<td>21</td>
</tr>
<tr>
<td>3.5 \texttt{sort}</td>
<td>23</td>
</tr>
<tr>
<td>3.6 Set membership</td>
<td>24</td>
</tr>
<tr>
<td>3.7 References to subroutines</td>
<td>24</td>
</tr>
<tr>
<td>3.8 Hashes as parameters</td>
<td>25</td>
</tr>
<tr>
<td>3.9 Markov generator</td>
<td>26</td>
</tr>
<tr>
<td>3.10 Random text</td>
<td>28</td>
</tr>
<tr>
<td>3.11 Exercises</td>
<td>29</td>
</tr>
<tr>
<td>4 Objects</td>
<td>31</td>
</tr>
<tr>
<td>4.1 Packages</td>
<td>31</td>
</tr>
<tr>
<td>4.2 The \texttt{bless} operator</td>
<td>32</td>
</tr>
<tr>
<td>4.3 Methods</td>
<td>32</td>
</tr>
<tr>
<td>4.4 Constructors</td>
<td>34</td>
</tr>
<tr>
<td>4.5 Printing objects</td>
<td>34</td>
</tr>
<tr>
<td>4.6 Heaps</td>
<td>35</td>
</tr>
<tr>
<td>4.7 Heap::add</td>
<td>35</td>
</tr>
<tr>
<td>4.8 Heap::remove</td>
<td>36</td>
</tr>
<tr>
<td>4.9 Trickle up</td>
<td>37</td>
</tr>
<tr>
<td>4.10 Trickle down</td>
<td>40</td>
</tr>
<tr>
<td>4.11 Exercises</td>
<td>42</td>
</tr>
</tbody>
</table>
## Modules

5.1 Variable-length codes ........................................ 43
5.2 The frequency table ........................................... 44
5.3 Modules .......................................................... 45
5.4 The Huffman Tree ............................................... 45
5.5 Inheritance ....................................................... 48
5.6 Building the Huffman tree ..................................... 48
5.7 Building the code table ........................................ 49
5.8 Decoding .......................................................... 50

## Callbacks and pipes

6.1 URIs ............................................................. 53
6.2 HTTP GET ......................................................... 54
6.3 Callbacks ........................................................ 55
6.4 Mirroring .......................................................... 55
6.5 Parsing ............................................................. 56
6.6 Absolute and relative URIs .................................... 58
6.7 Multiple processes .............................................. 58
6.8 Family planning .................................................. 59
6.9 Creating children ................................................ 59
6.10 Talking back to parents ....................................... 60
6.11 Exercises ........................................................ 61
Chapter 1

Arrays and Scalars

This chapter presents two of the built-in types, arrays and scalars. A scalar is a value that Perl treats as a single unit, like a number or a word. An array is an ordered collection of elements, where the elements are scalars.

This chapter describes the statements and operators you need to read command-line arguments, define and invoke subroutines, parse parameters, and read the contents of files. The chapter ends with a short program that demonstrates these features.

In addition, the chapter introduces an important concept in Perl: context.

1.1 Echo

The UNIX utility called echo takes any number of command-line arguments and prints them. Here is a perl program that does almost the same thing:

    print @ARGV;

The operand of the print operator is $ARGV. The “at” symbol indicates that $ARGV is an array variable; in fact, it is a built-in variable that refers to an array of strings that contains whatever command-line arguments are provided when the program executes.

There are several ways to execute a Perl program, but the most common is to put a “shebang” line at the beginning that tells the shell where to find the program called perl that compiles and executes Perl programs. On my system, I typed whereis perl and found it in /usr/bin, hence:
#!/usr/bin/perl
print @ARGV;
I put those lines in a file named echo.pl, because files that contain Perl pro-
grams usually have the extension pl. I used the command
$ chmod +ox echo.pl
to tell my system that echo.pl is an executable file, so now I can execute the
program like this:
$ ./echo.pl
Now would be a good time to put down the book and figure out how to execute
a Perl program on your system. When you get back, try something like this:
$ ./echo.pl command line arguments
commandlinearguments$
Sure enough, it prints the arguments you provide on the command line, although
there are no spaces between words and no newline at the end of the line (which
is why the $ prompt appears on the same line).
We can solve these problems using the double-quote operator and the
n sequence.
print "@ARGV\n";
It might be tempting to think that the argument here is a string, but it is
more accurate to say that it is an expression that, when evaluated, yields a
string. When Perl evaluates a double-quoted expression, it performs variable
interpolation and backslash interpolation.

Variable interpolation: When the name of a variable appears in double
quotes, it is replaced by the value of the variable.

Backslash interpolation: When a sequence beginning with a backslash (  ) appears in double quotes, it is replaced with the character specified by
the sequence.

In this case, the
n sequence is replaced with a single newline character.

Now when you run the program, it prints the arguments as they appear on the
command line.
$ ./echo.pl command line arguments
command line arguments
$
Since the output ends with a newline, the prompt appears at the beginning of
the next line. But why is Perl putting spaces between the words now? The
reason is:

The way a variable is evaluated depends on context!

In this case, the variable appears in double quotes, so it is evaluated in interpolative context. It is an array variable, and in interpolative context, the
elements of the array are joined using the separator specified by the built-in variable ".". The default value is a space.
1.2 Errors

What could possibly go wrong? Only three things:

**Compile-time error:** Perl compiles the entire program before it starts execution. If there is a syntax error anywhere in the program, the compiler prints an error message and stops without attempting to run the program.

**Run-time error:** If the program compiles successfully, it will start executing, but if anything goes wrong during execution, the run-time system prints an error message and stops the program.

**Semantic error:** In some cases, the program compiles and runs without any errors, but it doesn’t do what the programmer intended. Of course, only the programmer knows what was intended, so semantic errors are in the eye of the beholder.

To see an example of a compile-time error, try spelling `print` wrong. When you try to run the program, you should get a compiler message like this:

```plaintext
String found where operator expected at ./echo.pl line 3, near "prin "@ARGV\n"
(Do you need to predeclare prin?)
syntax error at ./echo.pl line 3, near "prin "@ARGV\n"
Execution of ./echo.pl aborted due to compilation errors.
```

The message includes a lot of information, but some of it is difficult to interpret, especially when you are not familiar with Perl. As you are experimenting with a new language, I suggest that you make deliberate errors in order to get familiar with the most common error messages.

As a second example, try misspelling the name of a variable. This program:

```plaintext
print "@ARG\n";
```

yields this output:

```plaintext
$ ./echo.pl command line arguments
```

Since there is no variable named `@ARG`, Perl gives it the default value, which is the empty list. In effect, Perl ignores what is almost certainly an error and tries to run the program anyway. This sort of behavior is occasionally helpful, but normally we would like the compiler to help us find errors, not obscure them. We can use the `strict` pragma to change the compiler’s behavior.

A pragma is a module that controls the behavior of Perl. To use the `strict` pragma, add the following line to your program:

```plaintext
use strict;
```

Now if you misspell the name of a variable, you get something like this:

```plaintext
Global symbol "@ARG" requires explicit package name.
```

Like many compiler messages, this one is misleading, but it contains hints about where the problem is, if nothing else.
1.3 Subroutines

If you have written programs longer than one hundred lines or so, I don’t need to tell you how important it is to organize programs into subroutines. But for some reason, many Perl programmers seem to be allergic to them.

Well, different authors will recommend different styles, but I tend to use a lot of subroutines. In fact, when I start a new project, I usually write a subroutine with the same name as the program, and start the program by invoking it.

```perl
sub echo {
    print "@_\n";
}
```

This program does the same thing as the previous one; it’s just more complicated.

All subroutine declarations start with `sub` followed by the name of the subroutine and the body. The body of the subroutine is a block of statements enclosed in squiggly-braces. In this case, the block contains a single statement.

The variable `@_` is a built-in variable that refers to the array of values the subroutine got as parameters.

1.4 Local variables

The keyword `my` creates a new local variable. The following subroutine creates a local variable named `params` and assigns a copy of the parameters to it.

```perl
sub echo {
    my @params = @_;    
    print "@params\n";
}
```

If you leave out the word `my`, Perl assumes that you are creating a global variable. If you are using the `strict` pragma, it will complain. Try it so you will know what the error message looks like.

1.5 Array elements

To access the elements of an array, use the bracket operator:

```perl
print "$params[0] $params[2]\n";
```

The numbers in brackets are indices. This statement prints the element of `@param` with the index 0 and the element with index 2. The dollar sign indicates that the elements of the array are scalar values.

A scalar is a simple value that is treated as a unit with no parts, as opposed to array values, which are composed of elements. There are three types of scalar
values: numbers, strings, and references. In this case, the elements of the array are strings.

To store a scalar value, you have to use a scalar variable.

```perl
my $word = $params[0];
print "$word\n";
```

The dollar sign at the beginning of $word indicates that it is a scalar variable.

Since the name of the array is @params, it is tempting to write something like

```perl
# the following statement is wrong
my $word = @params[0];
```

The first line of this example is a comment. Comments begin with the hash character (#) and end at the end of the line.

As the comment indicates, the second line of the example is not correct, but as usual Perl tries to execute it anyway. As it happens, the result is correct, so it would be easy to miss the error. Again, there is a pragma that modifies Perl's behavior so that it checks for things like this. If you add the following line to the program:

```perl
use warnings;
```

you get a warning like this:

```
Scalar value @params[0] better written as $params[0].
```

While you are learning Perl, it is a good idea to use strict and warnings to help you catch errors. Later, when you are working on bigger programs, it is a good idea to use strict and warnings to enforce good programming practice. In other words, you should always use them.

You can get more than one element at a time from an array by putting a list of indices in brackets. The following program creates an array variable named @words and assigns to it a new array that contains elements 0 and 2 from @params.

```perl
my @words = @params[0, 2];
print "@words\n";
```

The new array is called a slice.

### 1.6 Arrays and scalars

So far, we have seen two of Perl's built-in types, arrays and scalars. Array variables begin with @ and scalar variables begin with $. In many cases, expressions that yield arrays begin with @ and expressions that yield scalars begin with $. But not always. Remember:

> The way an expression is evaluated depends on context!
In an assignment statement, the left side determines the context. If the left side is a scalar, the right side is evaluated in **scalar context**. If the left side is an array, the right side is evaluated in **list context**.

If an array is evaluated in scalar context, it yields the number of elements in the array. The following program

```perl
my $word = @params;
print "$word\n";
```

prints the number of parameters. I will leave it up to you to see what happens if you evaluate a scalar in a list context.

### 1.7 List literals

One way to assign a value to an array variable is to use a list literal. A list literal is an expression that yields a list value. Here is the standard list example.

```perl
my @list = (1, 2, 3);
print "@list\n";
```

Most of the time, you can pretend that lists and arrays are the same thing. There are some differences, but for now the only one we are likely to run into is this: when you evaluate a list in a scalar context, you get the last element of the list. The following program prints 3.

```perl
my $scalar = (1, 2, 3);
print "$scalar\n";
```

But when you assign a list to an array variable, the result is an array value. So the following program prints the length of the list, which is 3.

```perl
my @list = (1, 2, 3);
my $scalar = @list;
print "$scalar\n";
```

The difference is subtle.

### 1.8 List assignment

When a list of variables appears on the left side of an assignment, Perl performs list assignment. The right side is evaluated in list context, and then the first element of the result is assigned to the first variable, the second element to the second variable, and so on.

A common use of this feature is to assign values from a parameter list to local variables.

The following subroutine assigns the first parameter to \p1, the second to \p2, and a list of the remaining parameters to @params.
1.9 The shift operator

Another way to do the same thing (because in Perl there’s always another way to do the same thing) is to use the shift operator.

shift takes an array as an argument and does two things: it remove the first element of the list and returns the value it removed. Like many operators, shift has both a side effect (modifying the array) and a return value (the result of the operation).

The following subroutine is the same as the previous one:

```perl
sub echo {
    my $p1 = shift @_; 
    my $p2 = shift @_; 
    print "$p1 $p2 @_
";
}
```

If you invoke shift without an argument, it uses @_ by default. In this example, it is possible (and common) to omit the argument.

1.10 File handles

To read the contents of a file, you have to use the open operator to get a file handle, and then use the file handle to read lines.

The operand of open is a list of two terms: an arbitrary name for the file handle, and the name of the file you want to open. The name of the file I want to open is /usr/share/dict/words, which contains a long list of English words.

```perl
open FILE, " /usr/share/dict/words";
```

In this case, the identifier FILE is global. An alternative is to create a local variable that contains an indirect file handle.

```perl
open my $fh, " /usr/share/dict/words";
```

By convention, the name of a global variable is all capital, and the name of a local variable is lower case. In either case, we can use the angle operator to read a line from the file:

```perl
while (<FILE>) {
    print 

    close FILE;
```
my $first = <FILE>;
my $first = <$fh>;

To be more precise, I should say that in a scalar context, the angle operator reads one line. What do you think it does in a list context?

When we get to the end of the file, the angle operator returns undef, which is a special value Perl uses for undefined variables, and for unusual conditions like the end of a file. Inside a while loop, undef is considered a false truth value, so it is common to use the angle operator in a loop like this:

```perl
while (my $line = <FILE>) {
    print $line;
}
```

### 1.11 cat

The UNIX cat utility takes a list of file names as command-line arguments, and prints the contents of the files. Here is a Perl program that does pretty much the same thing.

```perl
use strict;
use warnings;

sub print_file {
    my $file = shift;
    open FILE, $file;
    while (my $line = <FILE>) {
        print $line;
    }
}

sub cat {
    while (my $file = shift) {
        print_file $file;
    }
}

cat @ARGV;
```

There are two subroutines here, `print_file` and `cat`. The last line of the program invokes `cat`, passing the command-line arguments as parameters.

cat uses the shift operator inside a while statement in order to iterate through the list of file names. When the list is empty, shift returns undef and the loop terminates.

Each time through the loop, cat invokes `print_file`, which opens the file and then uses a while loop to print the contents.
Notice that `cat` and `print_file` both have local variables named `$file`. Naturally, there is no conflict between local variables in different subroutines.

The definition of a subroutine has to appear before it is invoked. If you type in this program (and you should), try rearranging the order of the subroutines and see what error messages you get.

### 1.12 `foreach` and `@_`

In the previous section, I used the `shift` operator and a `while` loop to iterate through the parameter list. A more common way to do the same thing is to use a `foreach` statement.

```perl
# the loop from cat
foreach my $file (@_) {
    print_file $file;
}
```

When a `foreach` statement is executed, the expression in parentheses is evaluated once, in list context. Then the first element of the list is assigned to the named variable (`$file`) and the body of the loop is executed. The body of the loop is executed once for each element of the list, in order.

If you don’t provide a loop variable, Perl uses `$_` as a default. So we could write the same loop like this:

```perl
# the loop from cat
foreach ( @_) {
    print_file $_;
}
```

When the angle operator appears in a `while` loop, it also uses `$_` as a default loop variable, so we could write the loop in `print_file` like this:

```perl
# the loop from print_file
while (<FILE>) {
    print $_;
}
```

Using the default loop variable has one advantage and one disadvantage. The advantage is that many of the built-in operators use `$_` as a default parameter, so you can leave it out:

```perl
# the loop from print_file
while (<FILE>) {
    print;
}
```

The disadvantage is that `$_` is global, so changing it in one subroutine affects other parts of the program. For example, try printing the value of `$_` in `cat`, like this:
# the loop from cat
foreach (@_) {
    print_file $_;
    print $_;
}

After print_line executes, the value of$_ is undef, because that is the terminating condition of the loop in print_line.

In this example, it is probably better to use explicit, local loop variables. Why?
Because the name of the variable contains useful documentation. In cat, it is clear that we are iterating over a list of files, and in print_file it is clear that we are iterating over the lines of the file. Using the default loop variable is more concise, but it obscures the function of the program.

## 1.13 Exercises

**Exercise 1.1**  The glob operator takes a pattern as an argument and returns a list of all the files that match the given pattern. A common use of glob is to list the files in a directory:

```perl
my @files = glob "$dir/*";
```

The pattern $dir/* means “all the files in the directory whose name is stored in $dir”. See the documentation of glob for examples of other patterns.

Write a subroutine called print_dir that takes the name of a directory as a parameter and that prints the file in that directory, one per line.

**Exercise 1.2**  Modify the previous subroutine so that instead of printing the name of the file, it prints the contents of the file, using print_file.

**Exercise 1.3**  The operator -d tests whether a given file is a directory (as opposed to a plain file). The following example prints “directory!” if the variable $file contains the name of a directory.

```perl
if (-d $file) {
    print "directory!";
}
```

Modify cat.pl so that if any of the command line arguments are directories, it invokes print_dir to print the contents of the files in the directory.
Chapter 2

Regular expressions

2.1 Pattern matching

The pattern binding operator (=~) compares a string on the left to a pattern on the right and returns true if the string matches the pattern. For example, if the pattern is a sequence of characters, the the string matches if it contains the sequence.

```
if ($line =~ "abc") { print $line; }
```

In my dictionary, the only word that contains this pattern is “Babcock”.

More often, the pattern on the right side is a match pattern, which looks like this: m/abc/. The pattern between the slashes can be any regular expression, which means that in addition to simple characters, it can also contain metacharacters with special meanings. A common metacharacter is ., which looks like a period, but is actually a wild card that can match any character.

For example, the regular expression pa..u.e matches any string that contains the characters pa and then exactly two characters, and then u and then exactly one character, and then e. In my dictionary, four words fit the description: “departure”, “departures”, “pasture”, and “pastures”.

The following subroutine takes two parameters, a pattern and a file. It reads each line from the file and prints the ones that match the pattern. This sort of thing is very useful for cheating at crossword puzzles.

```
sub grep_file {
    my $pattern = shift;
    my $file = shift;
    open FILE, $file;

    while (my $line = <FILE>) {
        if ($line =~ m/$pattern/) { print $line }
    }
}
```
I called this subroutine `grep_file` after the UNIX utility `grep`, which does almost the same thing.

In passing, notice that the last statement in a block doesn’t need a semi-colon.

**Exercise 2.1** Write a program called `grep.pl` that takes a pattern and a list of files as command line arguments, and that traverses each file printing lines that match the pattern. Warning: in this case it is not a good idea to create a subroutine named `grep` because there is already a function named `grep`. Try it, so you will know what the error message looks like, then choose a different name.

## 2.2 Anchors

Although the previous program is useful for cheating at crossword puzzles, we can make it better with anchors. Anchors allow you to specify where in the line the pattern has to appear.

For example, imagine that the clue is “Grazing place,” and you have filled in the following letters: p, blank, blank, blank, u, blank, e. If you search the dictionary using the pattern `p...u.e`, you get 57 words, including the surprising “Winnipesaukee”.

You can narrow the search using the ^ metacharacter, which means that the pattern has to begin at the beginning of the line. Using the pattern `^p...u.e`, we narrow the search to only 38 words, including “procurements” and “protuberant”.

Again, we can narrow the search using the $ metacharacter, which means that the pattern has to end at the end of the line. With the pattern `^p...u.e$`, we get only 12 words, of which only one means anything like “Grazing place”. The rejects include “perjure” and “profuse”.

## 2.3 Quantifiers

A **quantifier** is a part of a regular expression that controls how many times a sequence must appear. For example, the quantifier `{2}` means that the pattern must appear twice. It is, however, a little tricky to use, because it applies to a part of a pattern called an **atom**.

A character in a pattern is an atom, and so is a sequence of characters in parentheses. So the pattern `ab{2}` matches any word with a `a` followed by two `b`s, but the pattern `(ba){2}` requires the sequence `ba` to be repeated twice, as in the capital of Swaziland, which is Mbabane. The pattern `(es.){3}` matches any word where the pattern `.es.` appears three times. There’s only one in my dictionary: “restlessness”.

The `?` quantifier specifies that an atom is optional; that is, it may appear 0 or 1 times. So the pattern `(un)?usual` matches both “usual” and “unusual”. 
2.4 Alternation

Similarly, the + quantifier means that an atom can appear one or more times, and the * quantifier means that an atom can appear any number of times, including 0.

So far, I have been talking about regular expressions in terms of pattern matching. But there is another way to think about them: a regular expression is a way to denote a set of strings. In the simplest example, the regular expression abc represents the set that contains one string: abc. With quantifiers, the sets are more interesting. For example, the regular expression a+ represents the set that contains a, aa, aaa, aaaa, and so on. It happens to be an infinite set, so it is convenient that we can represent it so concisely.

The expressions a+ and a* almost represent the same set. The difference is that a* also contains the empty string.

**Exercise 2.2** Write a regular expression that matches any word that starts with pre and ends in al; for example, “prejudicial” and “prenatal.”

2.4 Alternation

The | metacharacter is like the conjunction “or”; it means either the previous atom or the next atom. So the regular expression Nina|Pinta|Santa Maria represents a set containing three strings: the names of Columbus’s ships. Of the three, only Nina appears in my dictionary.

The expression ^|(un|in) matches any word that begins with either un or in.

If you find yourself conjoining a set of characters, like a|b|c|d|e, there is an easier way. The bracket metacharacters define a character class, which matches any single character in the set. So the expression ^[abcde] matches any word that starts with one of the letters in brackets, and ^[abce]+$ matches any word that contains only those characters, from start to finish, like “acceded”. What set of five letters do you think yields the most words? I don’t know the answer, but the best I found was [eastr], which matches 133 words. What set of five letters yields the longest word? Again, I don’t know the answer, but the best I could do was [nesit], which includes “intensities”.

Inside brackets, the hyphen metacharacter specifies a range of characters, so [1-5] matches the digits from 1 to 5, and [a-emnx-z] is equivalent to [abcdemnxyz].

Also inside brackets, the carot metacharacter negates the character class, so [^0-9] matches anything that is not a digit, and [^\-] matches anything that does not start with a hyphen.

Several character classes are predefined, and can be specified with backslash sequences like \d, which matches any digit. It is equivalent to [0-9]. Similarly \w matches any whitespace character (space, tab, newline, return, form feed), and \w matches a so-called “word character” (upper or lower case letter, digit, and, of course, underscore).
Exercise 2.3

- Find all the words that begin with a|b and end with a|b. The list should include “adverb” and “balalaika”.
- Find all the words that either start and end with a or start and end with b. The list should include “alfalfa” and “bathtub”, but not “absorb” or “bursa”.
- Find all the words that begin with un or in and have exactly 17 letters.
- Find all the words that begin with un or in or non and have more than 17 letters.

2.5 Capture sequences

In a regular expression, parentheses do double-duty. As we have already seen, they group a sequence of characters into an atom so that, for example, a quantifier can apply to a sequence rather than a single letter. In addition, they indicate a part of the matching string that should be captured; that is, stored for later use.

For example, the pattern `http:(.*)` matches any URL that begins with `http:`, but it also saves the rest of the URL in the variable named `$1`. The following fragment checks a line for a URL and then prints everything that appears after `http:`.

```perl
my $pattern = "http:(.*)";
if ($line =~ m/$pattern/) { print "$1\n" }
```

If we are also interested in URLs that use `ftp`, we could write something like this:

```perl
my $pattern = "(ftp|http):(.*)";
if ($line =~ m/$pattern/) { print "$1, $2\n" }
```

Since there are two sequences in parentheses, the match creates two variables, `$1` and `$2`. These variables are called backreferences, and the strings they refer to are captured strings.

Capture sequences can be nested. For example, the regular expression `((ftp|http):(.*))` creates three variables: `$1` corresponds the outermost capture sequence, which yields the entire matching string; `$2` and `$3` correspond to the two nested sequences.

2.6 Minimal matching

If we extend the previous example, we encounter a property of regular expressions that is often problematic: quantifiers are greedy. Let’s say we want to parse a URL like `http://www.gnu.org/philosophy/free-sw.html` and separate the machine name (`www.gnu.org`) from the file name (`philosophy/free-sw.html`). We might try something like this:
my $pattern = "(ftp|http)://(\.*?)/(\.*\n";
if ($line =~ m/$pattern/) { print "$1, $2, $3\n" }

But the result would be this:

The first quantifier (\.*?) performed a maximal match, grabbing not only the machine name, but also the first part of the file name. What we intended was a minimal match, which would stop at the first slash character.

We can change the behavior of the quantifiers by adding a question mark. The pattern (ftp|http)://(\.*?)/(\.*) does what we wanted. The quantifiers *, +, and ? are the same as *, +, and ?, except that they perform minimal matching.

### 2.7 Extended patterns

As regular expressions get longer, they get harder to read and debug. In the previous examples, I have tried to help by assigning the pattern to a variable and then using the variable inside the match operator m/>. But that only gets you so far.

An alternative is to use the extended pattern format, which looks like this:

```perl
if ($line =~ m{  
  (ftp|http) # protocol 
  ://    # machine name (minimal) 
  /      # file name 
  (...?  # machine name (minimal) 
}x
{ print "$1, $2, $3\n" }

The pattern begins with m{ and ends with }x. The x indicates extended format; it is one of several modifiers that can appear at the end of a regular expression.

The rest of the statement is standard, except that the arrangement of the statements and punctuation is unusual.

The most important features of the extended format are the use of whitespace and comments, both of which make the expression easier to read and debug.

### 2.8 Some operators

Perl provides a set of operators that might be best described as a superset of the C operators. The mathematical operators +, -, *, and / have their usual meanings, and % is the modulus operator. In addition, ** performs exponentiation.
The comparison operators $>, <$, $==$, $>=, <=$ and $!=$ perform numerical comparisons, but the operators $gt$, $lt$, $eq$, $ge$, $le$ and $ne$ perform string comparison. In both cases, Perl converts the operands to the appropriate types automatically. So the expression $10 lt 2$ performs string comparison even though both operands are numbers, and the result is true.

$<>$ is called the “spaceship” operator. Its value is 1 if the left operand is numerically bigger, -1 if the right operand is bigger, and 0 if they are equal.

There are two sets of logical operators: $&&$ is the same as $and$, and $||$ is the same as $or$. Actually, there is one difference. The textual operators have lower precedence than the corresponding symbolic operators.

## 2.9 Prefix operators

We have already used several prefix operators, including $print$, $shift$, and $open$. These operators are followed by a list of operands, usually separated by commas. The operands are evaluated in list context, and then “flattened” into a single list.

There is an alternative syntax for a prefix operator that makes it behave like a C function call. For example, the following pairs of statements are equivalent:

```perl
print $1, $2;
print($1, $2);
shift @_;  
shift(@_);
open FILE, $file;
open(FILE, $file);
```

In a sense, the parentheses are optional, but there is a little more to it than that, because the two syntaxes have different precedence. Normally that wouldn’t matter much, except that there is a common idiom for error-handling that looks like this:

```perl
open FILE, $file or die "couldn’t open $file\n";
```

The $die$ operator prints its operands and then ends the program. The $or$ operator performs short circuit evaluation, which means that it only evaluates as much of the expression as necessary, reading from right to left.

If the $open$ succeeds, it returns a true value, so the $or$ operator stops without executing $die$ (because true or $x$ is always true, no matter what $x$ is).

Since $or$ and $||$ are equivalent, you might assume that it would be equally correct to write

```perl
open FILE, $file || die "couldn’t open $file\n";
```

Unfortunately, because $||$ has higher priority than $or$, this expression computes $file || die "couldn’t open $file\n"$ first, which yields the value of $file$, so $die$ never executes, even if the file doesn’t exist.
2.10 Subroutine semantics

One way to avoid this problem is to use `or`. Another way is to use the function call syntax for `open`. The following works because function call syntax is evaluated in the order you would expect.

```
open(FILE, $file) || die "couldn't open $file\n";
```

While we are at it, I should mention that there are two special variables that can generate more helpful error messages.

```
die "$0: Couldn't open $file: $!\n"
```

`$0` contains the name of the program that is running, and `$!` contains a textual description of the most recent error message. This idiom is so common that it is a good idea to encapsulate it in a subroutine:

```
sub croak { die "$0: @_: $!\n" }
```

I borrowed the name `croak` from *Programming Perl*, by Wall, Christiansen and Orwant.

### 2.10 Subroutine semantics

In the previous chapter I said that the special name `@_` in a subroutine refers to the list of parameters. To make that statement more precise, I should say that the elements of the parameter list are **aliases** for the scalars provided as arguments. An alias is an alternative way to refer to a variable. In other words, `@_` can be used to access and modify variables that are used as arguments.

For example, `swap` takes two parameters and swaps their values:

```
sub swap {
    ($_[0], $_[1]) = ($_[1], $_[0]);
}
```

In a list assignment, the right side is evaluated before any of the assignments are performed, so there is no need for a temporary variable to perform the swap.

The following code tests `swap`:

```
my $one = 1;
my $two = 2;
swap($one, $two);
print "$one, $two\n",
```

Sure enough, the output is `2, 1`. Since `swap` attempts to modify its parameters, it is illegal to invoke it with constant values. The expression `swap(1,2)` yields:

```
Modification of a read-only value attempted in ./swap.pl
```

On the other hand, we can invoke it with a list:

```
my @list = (1, 2);
swap(@list);
print "@list\n";
```

When a list appears as an argument, it is “flattened”; that is, the elements of the list are added to the parameter list. So the following code does not swap two lists:
my @list1 = (1, 2);
my @list2 = (3, 4);
swap(@list1, @list2);
print "@list1 @list2\n";
Instead, swap gets a list of four scalars as parameters, and it swaps the first two. The output is 2 1 3 4.

2.11 Exercises

Exercise 2.4  In a regular expression, the backslash sequence \1 refers to the first (prior) capture sequence in the same expression. As you might guess, \2 refers to the second sequence, and so on.

Write a regular expression that matches all lines that begin and end with the same character.

Exercise 2.5
Chapter 3

Hashes

3.1 Stack operators

As a simple implementation of a stack, you can use the push and pop operators on an array. push adds an element to the end of an array; pop removes and returns the last element.

my @list = (1, 2);
push @list, 3;
At this point, @list contains 1 2 3.

my $elt = pop @list;
At this point, $elt contains 3 and @list is back to 1 2.

When we are using a list as a stack, the names push and pop are appropriate. For example, one use of a stack is to reverse the elements of a list. The following subroutine takes a list as a parameter and returns a new list with the same elements in reverse order.

sub rev {
    my @stack;
    foreach (@_) { push @stack, $_; } 

    my @list;
    while (my $elt = pop @stack) {
        push @list, $elt;
    }
    return @list;
}

The first loop traverses the parameter list and pops each element onto a local stack. The second loop pops the elements off stack and adds them to list, which is the return value.
Exercise 3.1  Perl also provides an operator named reverse that does almost the same thing as rev, except that it modifies the parameter list rather than creating a new one. Modify rev so that it works the same way.

The point of this exercise is just to demonstrate the stack operators. If you really had to write your own version of reverse, you would probably skip the stack and swap the elements in place.

```perl
sub rev3 {
    for (my $i = 0; $i < @_/2; $i++) {
        swap ($_[$i], $_[-$i-1]);
    }
    return @_;}
```

This subroutine demonstrates a for loop, which is similar to the same statement in C, including the increment operator ++.

It also takes advantage of negative indices, which count from the end of the array. So, when i=0, the expression -$i-1 is -1, which refers to the last element of the array.

3.2 Queue operators

We have already seen shift, which removes and returns the first element of a list. In the same way that push and pop implement a stack, push and shift implement a queue.

In addition, unshift adds a new element at the beginning of an array. shift and unshift are often used for parsing a stream of tokens.

3.3 Hashes

A hash is a collection of scalar values, like an array. The difference is that the elements of an array are ordered, and accessed using numbers called indices; the elements of a hash are unordered, and accessed using scalar values called keys.

Just as scalars are identified by the $ prefix, and arrays are identified by the @ prefix, hashes begin with a percent sign (%). Just as the index of an array appears in square brackets, the key of a hash appears in squiggly braces.

```perl
my %hash;
$hash{do} = "a deer, a female deer";
$hash{re} = "a drop of golden sun";
$hash{mi} = "what it's all about";
```

The first line creates a local hash named %hash. The next three lines assign values with the keys do, re and me. These keys are strings, so we could have put them in double quotes, but in the context of a hash key, Perl understands that they are strings.
3.4 Frequency table

Hashes are sometimes called associative arrays because they create an association between keys and values. In this example, the key do is associated with the string a deer, a female deer, and so on.

The keys operator returns a list of the keys in a hash. The expression keys %hash yields mi do re. Notice that the keys are in no particular order; it depends on how the hash is implemented, and might even change if you run the program again (although probably not).

Here is a loop that traverses the list of keys and prints the corresponding values.

```perl
foreach my $key (keys %hash) {
    print "$key => $hash{$key}\n";
}
```

The result of this loop looks like this:

```
mi => what it's all about
do => a deer, a female deer
re => a drop of golden sun
```

My use of the double arrow symbol (\) isn't a coincidence. The double arrow can also be used to assign a set of key-value pairs to a hash.

```perl
%hash = (  
    do => "a deer, a female deer",
    re => "a drop of golden sun",
    mi => "what it's all about",
);
```

Another way to traverse a hash is with the each operator. Each time each is called, it returns the next key-value pair from the hash as a two-element list. Internally, each keeps track of which pairs have already been traversed.

```perl
while ((my $key, my $value) = each %hash) {
    print "$key => $value\n";
}
```

Finally, the values operator returns a list of the values in a hash.

```perl
my @values = values %hash;
```

Of course, you can traverse the list of values, but there is no way to look up a value and get the corresponding key. In fact, there might be more than one key associated with a given value.

3.4 Frequency table

One use for a hash is to count the number of times a word in used in a document. To demonstrate this application, we will start with a copy of `grep.pl` from the previous chapter. It contains a subroutine that opens a file and traverses the lines. With a few small changes, it looks like this:
sub read_file {
    my $file = shift;
    open (FILE, $file) || croak "Couldn’t open $file";

    while (my $line = <FILE>) {
        read_line $line;
    }
}

As `read_file` gets each line from the file, it invokes `read_line` to process the line. `read_line` uses the `split` operator to break the line into words, then traverses the list of words.

sub read_line {
    our %hash;

    my @list = split " ", shift;
    foreach my $word (@list) {
        $hash{$word}++;
    }
}

The first parameter of `split` is a regular expression that is used to decide where to split the string. In this case, the expression is trivial; it’s the space character.

The first line of the subroutine creates the hash. The keyword `our` indicates that it is a global variable, so we will be able to access it from other subroutines.

The workhorse of this subroutine is the expression `$hash{$word}++`, which finds the value in the hash that corresponds to the given word and increases it. When a word appears for the first time, Perl magically does the right thing, making a new key-value pair and initializing the value to zero.

To print the results, we can write another subroutine that accesses the global hash.

sub print_hash {
    our %hash;

    my @list = keys %hash;
    print "@list\n";
}

Exercise 3.2

- Using the subroutines in this chapter, write a program that reads a text file and counts the number of times each word is used. Hint: use the `lc` operator to convert words to lower case, and the first argument of `split` to list the characters that should separate words. You can download the text of many out-of-copyright books from gutenberg.net.

- Modify the program so that it prints the number of unique words that appear in the book.
• Modify the program so that it prints the most commonly used word and the number of times it appears.

3.5 sort

The sort operator sorts the elements of a list, usually in the order determined by the string comparison operator. For example, the following version of print_hash prints the keys in alphabetical order:

```perl
sub print_hash {
    my @list = sort keys our %hash;
    print "@list\n";
}
```

If you want the list sorted in a different order, you can provide a special subroutine that compares two values and returns 1 if the first is greater (by whatever definition of “greater), -1 if the second is greater, and zero if they are equal.

For example, to sort the values numerically, we could provide this subroutine:

```perl
sub numerically { $a <=> $b }
```

Inside the subroutine, the special names `$a` and `$b` refer to the elements being compared. Now we can use `sort` like this:

```perl
my @list = sort numerically values our %hash;
```

The values from the hash are sorted from low to high. Unfortunately, this doesn’t help us find the most common words, because we can’t look up a value to get the associated word.

On the other hand, we can provide a comparison subroutine that compares keys by looking up their associated values:

```perl
sub byvalue {
    our %hash;
    $hash{$b} <=> $hash{$a};
}
```

And then sort the keys by value like this:

```perl
my @list = sort byvalue keys our %hash;
```

**Exercise 3.3** Modify the program from the previous section to print the 20 most common words in a file and their frequencies.

The most common word in *The Great Gatsby*, by F. Scott Fitzgerald, is “the”, which appears 2403 times, followed by “and”, which appears 1573. The most frequent non-boring word is “Gatsby”, which comes in 32nd on the list with 197 appearances.
3.6 Set membership

Hashes are frequently used to check whether an element is a member of a set. For example, we could read the list of words in /usr/share/dict/words and build a hash that contains an entry for each word.

The following subroutine takes a line from the dictionary and makes an entry for it in a hash.

```perl
sub dict_line {
    our %dict;
    my $word = lc shift;
    chomp $word;
    $dict{$word}++;
}
```

The first line declares the global hash named %dict. The second line gets the parameter and converts it to lower case. The third line uses chomp to remove the newline character from the end of the word. The last line makes the entry in the dictionary.

Now we can check whether a word is in the dictionary by checking whether a hash entry with the given key is defined. The defined operator tells whether an expression is defined.

```perl
if (!defined $dict{$word}) { print "*" }
```

When the body of an if statement is short, it is common to write it on a single line, and omit the semi-colon on the last statement in the block. It is also common to take advantage of the alternative syntax

```perl
print "*" if !defined $dict{$word};
```

which simplifies the punctuation a little.

Applying this analysis to The Great Gatsby yields some surprising lapses in my dictionary, like "coupe" and "yacht", and some surprising vocabulary in the book, like "pasquinade" (public ridicule of an individual) and "echolalia" (the involuntary repetition of sounds made by others).

3.7 References to subroutines

At this point we find ourselves traversing two files, a dictionary and a text, and performing different operations on the lines. Of course, we could copy the code that opens and traverses a file, but it might be better to generalize read_file so that it takes a second argument, which is a reference to the subroutine it should use to process each line.

```perl
sub read_file {
    my $file = shift;
    my $subref = shift || \&read_line;

    open (FILE, $file) || croak "Couldn’t open $file";
```
while (my $line = <FILE>) {
    $subref->($line);
}

The second line uses shift to read the second parameter. If the result is undefined (that is, if there is no second parameter), then it uses the default value \&read_line, which is a reference to a subroutine. Strictly speaking, \&read_line is the name of the subroutine, and the backslash makes a reference to it.

Notice that $subref is a scalar variable, which is a hint that references are scalar values.

Inside the loop we see the syntax for invoking a subroutine when we have a reference to it. The arrow operator (->) is one of several ways to dereference a reference; that is, use the value the reference refers to. The argument in parentheses is just an argument like the ones we’ve seen before.

Now to read the dictionary, we can invoke read_line with a second parameter.

read_file "/usr/share/dict/words", \&dict_line;

Again, the expression \&dict_line is a reference to the subroutine whose name is technically &dict_line.

Exercise 3.4  
Grab the text of your favorite book from gutenberg.net and make a list of the words in the book that aren’t in your dictionary.

3.8 Hashes as parameters

When a hash is passed as a parameter, it is converted to an alternating list of keys and values. For example, the following subroutine

sub print_hash {
    print "@_
";
}

produces the following abstruseness

mi what it’s all about do a deer, a female deer re a drop of golden sun

One solution is to convert the list back to a hash:

sub print_hash {
    my %hash = @_;  
    while ((my $key, my $value) = each %hash) {
        print "$key => $value\n";
    }
}

For the vast majority of applications, the performance of that solution would be just fine, but for a very large hash, it would be better to pass the hash by reference.
When we invoke `print_hash`, we pass a reference to the hash, which we create with the backslash operator.

```
print_hash \%hash;
```

Inside `print_hash`, we assign the reference to a scalar named `$hashref`, and then use the `%` prefix to dereference it; that is, to access the hash that `$hashref` refers to.

```
sub print_hash {
    my $hashref = shift;
    while ((my $key, my $value) = each %$hashref) {
        print "$key => $value\n";
    }
}
```

References can be syntactically awkward, but they are useful and versatile, so we will be seeing more of them.

### 3.9 Markov generator

To demonstrate some of the features we have been looking at, I am going to develop a program that reads a text and analyses the frequency of various word combinations, and then generates a new, random text that has the same frequencies. The result is usually entertainingly nonsensical, often bordering on parody.

For example, given the text of *The Great Gatsby*, the generator produces the following:

"Why CANDLES?" objected Daisy, frowning. She snapped them out to the garage, Wilson was so sick that he was in he answered, "That’s my affair," before he went there. A pause. "I don’t like mysteries," I answered. "And I think of you.” This included me. Mr. Sloane and the real snow, our snow, began to melt away until gradually I became aware now of a burglar blowing a safe.

Given the first three chapters of this book, it produces:

One way to refer to a variable appears in double quotes, so it would be easy to miss the error. Again, there is an array of strings that contains only those characters, from start to finish, like “acceded”. What set of characters, so matches anything that does not start with sub followed by “love” and “tender, love”, although there are no spaces between the words now?

...which probably makes as much sense as the original.

The first step is to analyze the text by looking at all the three-word combinations. For each two-word prefix, we would like to know all the words that might come next, and how often each occurs. For example, in Elvis’ immortal words
love me tender, love me true.

The prefix “love me” appears twice, followed by either “tender,” or “true.” Also, the prefix “me tender,” which is followed by “love” and “tender, love”, which is followed by “me”.

To complete the analysis, we can build a hash that maps from prefixes to the list of words that can follow.

```perl
read_file $file, \&triple_line;
```

The function `triple_line` just splits the line into words and invokes `triple_word`

```perl
sub triple_line {  
    my @list = split " ", shift;  
    foreach my $word (@list) {  
        triple_word $word;  
    }  
}
```

In `triple_word`, we store the most recent prefix in a global variable named `prefix`.

```perl
sub triple_word {  
    our @prefix;  
    my $word = shift;  
    if (@prefix == 2) {  
        print "@prefix => $word\n";  
        shift @prefix;  
    }  
    push @prefix, $word;  
}
```

When the length of the prefix is two, we can print an entry and remove a word from the beginning. Otherwise, we just add the word to the end. The output is

```
love me => tender,
me tender, => love
tender, love => me
love me => true.
```

What we need now is a hash that maps from each prefix to a list of words. There are several ways to do that. The simplest is to store the list of words as a string, which can be stored as a value in a hash because it is a scalar. But if the lists are long, we might get better performance by storing the words in an array. An array is not a legal value in a hash, but a reference to an array is.
Here is a version of `triple_word` that uses a hash to map from a prefix to an array reference.

```perl
sub triple_word {
    our (@prefix, %hash);
    my $word = shift;

    if (@prefix == 2) {
        my $key = join " ", @prefix;

        my $aref = $hash{$key};
        push @$aref, $word;
        $hash{$key} = $aref;
    }
    else {
        push @prefix, $word;
    }
}
```

In the `if` statement, the `join` operator concatenates the elements of `@prefix` into a string we can use as a key.

The next three lines do the real work. First we look up the key and get an array reference. To dereference the array, we add the `@` prefix to the scalar `$aref`, and use `push` to append the new word.

The third line is only necessary if we are making a new entry. If the key was already in the table, then modifying the array is all we have to do.

In passing, notice that the declarations `our` and `my` can apply to a list of variables.

**Exercise 3.5** This example is meant to demonstrate the use of array references, but if we don’t expect the arrays to be very long, it is much easier to use strings instead. Rewrite `triple_word` so that the values in the hash are strings. Hint: consider the `.=` operator.

### 3.10 Random text

Now that we have analyzed the original text, we would like to generate a random text that has the same combinations of words with the same frequency.

The first step is to choose a random prefix from the hash. The `rand` operator chooses a random (floating-point) number between 0 and its parameter (up to but not including the parameter. So the expression `rand @list` chooses a random index from a given list (because when the list is evaluated in a scalar context, it yields its length).

The subroutine `rand_elt` chooses and return a random element of the parameter list.
3.11 Exercises

sub rand_elt { $_[rand @_] }

Now choosing a random key from the hash is easy.

    my $prefix = rand_elt keys %hash;

Next we look up the prefix in the hash, split the string of words into a list, and
choose a random element.

    my $words = $hash{$prefix};
    my $word = rand_elt split " ", $words;

Finally, we can form the next prefix by splitting the old prefix, shifting the first
word, and appending the new word.

    my @triple = split " ", $prefix;
    shift @triple;
    $prefix = "@triple $word";

Exercise 3.6  Write a subroutine called random_text that generates a random text
by starting with a random key from the hash and then choosing random words from
the list of successors. Analyze your favorite text and generate a new text using the
hash you computed.

3.11 Exercises

Exercise 3.7  The text analysis implementation in this chapter is not very efficient
for large files, because as the suffix lists get longer, it gets more expensive to add
new words. A possible solution is to use a hash to hold the possible suffixes and
their frequencies. This implementation saves space, since each suffix is stored once,
and time, since the cost of adding an entry to a hash doesn’t depend on the number
of previous entries. One drawback of this approach is that it is harder to generate
random text.

- Change the implementation of the text analysis so that is uses a hash instead
  of an array of suffixes.
- Change the implementation of the random text generation so that it chooses
  words from a hash of suffixes. The probability of choosing a word should be
  proportional to the number of times it appears.

Exercise 3.8  If each value in a hash appears only once, the hash is invertible,
which means we can create a new hash with all the same key-value pairs, but where
the values become keys, and vice versa.

Write a subroutine that takes a hash as a parameter and returns its inverse. For
purposes of the exercise, you should traverse the elements of the hash, but you should
also be aware of the following very clever implementation:

sub invert_hash {
    my %hash = reverse @_;  
    return %hash;
}
When the hash is passed as a parameter, it is converted to an alternating list of keys and values. Reversed, it becomes a list of values and keys, and when assigned to a hash, it is converted back to a hash.

The `return` statement is optional. By default, the result of a subroutine is the value of the last expression.
Chapter 4

Objects

When we say that a language is object-oriented, we usually mean that it provides syntactic features that are intended specifically to support object-oriented programming. By that definition, Perl is not really an object-oriented language. On the other hand, there are many ways to write object-oriented programs in Perl, and many of the language's features are well-suited for it.

At the most basic level, an object is a reference. Starting there, this chapter will present the various features that are typically used to write object-oriented Perl.

4.1 Packages

Although you can think of any reference as an object, it might make you more comfortable to assign your objects to classes. In Perl, you create a class by defining a package and then using the bless operator to assign objects to the new class.

The package statement marks the beginning of a new package. All subsequent variable and subroutine declarations are added to the current package, until the end of the current file (or code block), or another package statement.

The following example declares a global variable named value and then starts a package named Fred. Inside the new package, we can create another variable with the same name without conflict, because the new package has its own namespace. The print statement refers to the variable in the current package, which has the value home:

```perl
our $value = "away";
package Fred;
our $value = "home";
print "$value\n";
```
Strictly speaking, what we have been calling “global” variables are really package variables. All variables belong to a package; if you don’t create your own package, the default name is main.

To refer to a variable in another package, you have to use a fully-qualified name, which includes the name of the package and the name of the variable. In previous example, we could access the first variable with the expression $main::value, which has the value away.

### 4.2 The bless operator

Any reference can be considered an object, but the most common kind of object is a reference to a hash. The keys of the hash are the instance variables of the object. So, the simplest way to create an object is to create a reference to a hash. We have already seen one way to do that, using the backslash operator.

```perl
my %hash;
my $hashref = \%hash;
```

Another way to do the same thing is to use squiggly braces.

```perl
my $nobody = { };  # Creates an empty hash
my $person = { name => "Allen B. Downey", webpage => "allendowney.com" };  # Creates a hash with two key-value pairs
```

The first example creates an empty hash and makes $nobody refer to it. The second example creates a hash with two key-value pairs (and a plug for the author!).

Now we just have to tell Perl that this object belongs to a particular class. The following example declares a Person package, creates an object, and assigns the object to the Person class.

```perl
package Person;
my $person = { name => "Allen B. Downey", webpage => "allendowney.com" };  # Creates a hash with two key-value pairs
bless $person, "Person";
```

Now, when we invoke a method on this object (which we’ll see shortly), Perl knows what package to find the method in.

### 4.3 Methods

A method is just a subroutine that takes an object as its first parameter. For example, name takes a Person as a parameter and returns the value of the instance variable name.

```perl
sub name {
    my $self = shift;
    my %hash = %$self;
```
4.3 Methods

Since a Person is a reference to a hash, we have to dereference it before we can look up the key name. The arrow operator provides a more convenient way to do both at the same time.

```perl
sub name {
    my $self = shift;
    return $self->{name};
}
```

name is an example of an accessor method, since it provides access to one of the instance variables.

To invoke this method, we can just pass a Person object as a parameter.

```perl
my $name = name $person;
```

Or we can use the arrow operator.

```perl
my $name = $person->name;
```

The latter might help maintain the illusion that you are using an object-oriented language.

It is often convenient to provide a single accessor that both sets and reads an instance variable.

```perl
sub name {
    my $self = shift;
    if (@_) { $self->{name} = shift; }
    return $self->{name};
}
```

Again, the first parameter is the object. If there is a second parameter, it is used to update name. Either way, the current value of name is returned.

And again, there are two ways to invoke this method. The object can appear explicitly as the first parameter, like this:

```perl
name $person, "Nella B. Yenwod";
```

Or we can use the arrow operator. Unfortunately, we can’t use the arrow operator with a list of parameters:

```perl
$person->name "Nella B. Yenwod";        # WRONG
```

But you can solve this problem with parentheses.

```perl
$person->name("Nella B. Yenwod");
```

In situations like this, Perl enthusiasts like to point out that “There’s more than one way to do it,” which they abbreviate TMTOWTDI. Perl detractors sometimes come back with FORHOTDW, which stands for, “For obscure reasons, half of them don’t work”.

4.4 Constructors

A constructor is a method, any method, that creates and returns a new object. By convention, constructors are often named `new`. Here is a `Person` constructor.

```perl
sub new {
    my $self = { @_; 
    bless $self, "Person";
}
```

The parameters are a list of keys and values that are used to initialize the hash. So you could invoke the constructor like this:

```perl
my $person = new(name => "Allen B. Downey");
```

But it is more common to invoke a constructor using the name of the class:

```perl
my $person = Person->new(name => "Allen B. Downey");
```

In that case, the class name is sent as the first parameter in the list, so we can use it as the second parameter for `bless`:

```perl
sub new {
    my $class = shift;
    my $self = { @_; 
    bless $self, $class;
}
```

This is a standard idiom for object constructors. Whenever you write a new class, you can start by copying this method.

4.5 Printing objects

For purposes of debugging, you will almost always want a method that prints the state of an object. Here is a simple version named `print`.

```perl
sub print {
    my $self = shift;
    while ((my $key, my $value) = each %$self) {
        print "$key => $value
    }
}
```

Because the argument is a reference to a hash, we have to use the expression `%$self` to dereference it.

This method just prints the keys and values from the hash. If some of the values are references, Perl prints the type of the reference and its unique identifier. You might want to add code to this method to print the contents of the object in a more informative way.
4.6 Heaps

To demonstrate a more interesting object, we're going to write an implementation of a heap. If you are not familiar with heaps, you should consult a data structures textbook, or How to Think Like a Computer Scientist: Java Version, which is available from thinkapjava.com.

In order to develop the code gradually, we’ll start with a heap of numbers. Here is the package declaration and constructor:

```perl
package Heap;

sub new {
    my $class = shift;
    my $self = {
        aref => [ "" ],
        next => 1
    };
    bless $self, $class;
}
```

The instance variables are a reference to an array (cleverly named `aref`) and a number named `next` that keeps track of the next available position in the array.

The expression `[ "" ]` creates a reference to an array with a single element, an empty string. We won’t use the first element of the array, so we are putting an empty string there as a place keeper. The initial value of `next` is the index of the first position we will use, 1.

As usual, it is a good idea to have a method that prints the state of an object. A method that prints the state of an object:

```perl
sub print {
    my $self = shift;
    my $next = $self->{next};
    my $aref = $self->{aref};
    print "array => @$aref\n";
    print "next => $next\n";
}
```

Since `$self->{aref}` is a reference to an array, we have to dereference it to print its contents. `$self->{next}` is a scalar, so all we have to do is print it.

4.7 Heap::add

The `add` method takes two parameters, the heap and the value we want to add it. It puts the value into the next available location in the array and then increments `next`.

```perl
sub add {
    my ($self, $value) = @_; 
    my $index = $self->{next};
    $self->{aref}[$index] = $value;
    $self->{next}++;
}
```
$_{index}$ is a local copy of the instance variable next. We can use its value as an index into aref, but in order to update next, we have to use the writable expression $self->{next}.

Personally, I am surprised that we can access the elements of the array using the bracket operator. After all, $self->{aref}$ is a reference to an array, not an array, and normally Perl doesn’t dereference automatically.

Consider the following examples:

my @array = ( 1, 2, 3 );
my $aref = [ 1, 2, 3 ];

The first line creates a list and assigns it to an array named @array. The second line creates a reference to an array and assigns it to a scalar named $aref. We can use square brackets to access the elements of @array, so the expression $array[1]$ has the value 2. But the expression $aref[1]$ is illegal. You will get something like

Global symbol "@aref" requires explicit package name

because Perl thinks you are asking for the second element of an array named @aref. The usual solution is to use the arrow operator, as in $aref->[1], which dereferences $aref and then gets element 1.

So, if we put an array reference in a hash, like this

my %hash;
$hash{aref} = $aref;

then how do we access its elements? You would think we should use the arrow operator again, as in $hash{aref}->[1]. Actually, that works, but it is not the most common form. It turns out that you can get the reference out of the hash and dereference it at the same time, as in $hash{aref}[1].

About now, you may be feeling a sharp pain near your temple. That’s normal. Ignore it.

We can test add and print by putting some random numbers into a heap:

my $heap = Heap->new;
for (my $i=0; $i<10; $i++) {
    $heap->add (rand 100);
}
$heap->print;

If you try it, you should see an array of ten random numbers between 0 and 100, and the value of next should be 10.

### 4.8 Heap::remove

Here is a version of remove that removes and returns the first element in the array, and moves the last element into the first position.
sub remove {
    my $self = shift;
    my $aref = $self->{aref};
    my $result = $aref->[1];
    $aref->[1] = pop @$aref;
    $self->{next}--;
    return $result
}

As usual, the first line gets the parameter. The second line makes a local copy of the array reference $aref. If the value of an instance variable is a reference, we can unpack it by assigning it to a local variable. Because it is a reference, we can use it for both reading and writing the array.

The next three lines (1) store the first element of the heap, which is the result we’ll return, (2) move the last element of the heap up to the front (for reasons that will become clear soon), and (3) decrement $next, which sets it to the index of the now-available location we just emptied.

We can test remove with the following loop:
for (1..10) {
    my $value = $heap->remove;
    print "$value\n";
}

The double-dot operator builds a list of the elements from 1 to 10. When the for statement is applied to a list, it behaves like a foreach statement. If you run this code, it will print the elements of the heap, but not in numerical order, yet (and not even in the order they were added).

A better alternative is to implement Heap::empty:
sub empty {
    my $self = shift;
    return $self->{next} == 1;
}

Then we can use an until statement to empty the heap:
until ($heap->empty) {
    my $value = $heap->remove;
    $value->print;
}

until is the same as while, except that it inverts the condition.

4.9 Trickle up

A fundamental idea in this implementation of a heap is that there is a relationship between the elements of the array and the nodes in a tree. This relationship is shown in the following figure.
Another way to say the same thing is that the nodes of the tree are stored in the array in **level order**, that is, all the nodes from one level, left to right, followed by all the nodes from the next level, and so on.

Because we skipped element 0 and started with element 1, it is easy to find the parent and children of a given node using index arithmetic. For example, for the node with index \(i\), the left child has the index \(2i\), the right child has index \(2i + 1\), and the parent has the index \([i/2]\).

In order for a tree to be considered a heap, it has to have the **heap property**:

> Every node must be greater than or equal to its children.

If this property holds, then the largest element in the heap must be at the root. As we add elements to the heap, we can maintain the heap property by putting the new element at the end and allowing it to “trickle up” into the right position.

Here is a more complete implementation of `Heap::add`:

```perl
sub add {
    my ($self, $value) = @_;
    my $i = $self->{next};
    $self->{aref}[$i] = $value;
    while ($i > 1) {
        my $parent = POSIX::floor($i/2);
        last if $self->compare($i, $parent) <= 0;
        $self->swap($i, $parent);
        $i = $parent;
    }
    $self->{next}++;
}
```

The index of the new element is \(i\). The **while loop** continues until \(i\) gets to the root (element 1). Inside the loop, we compute the parent of the current node. The `floor` function belongs to the module `POSIX`, which contains the standard POSIX 1003.1 identifiers, including lots of mathematical functions. Before you can use them, of course, you have to **use POSIX**;
The last statement exits the loop if the current node is less than or equal to its parent. The following statement

```perl
last if $self->compare($i, $parent) <= 0;
```

is equivalent to

```perl
if ($self->compare($i, $parent) <= 0) {
    last;
}
```

but it requires less punctuation, and some people think it is more readable because it puts the action before the condition.

The `compare` method takes two indices as parameters and compares the corresponding elements.

```perl
sub compare {
    my ($self, $i, $j) = @_; 
    my $x = $self->{aref}[$i]; 
    my $y = $self->{aref}[$j]; 
    if (!defined $x) {
        if (!defined $y) {
            return 0;
        } else {
            return -1;
        }
    } else {
        return $x <=> $y;
    }
}
```

If both elements exist, then we use the spaceship operator to compare them. The rest of the method deals with the cases where one or more of the elements are undefined.

The `swap` method takes two indices as parameters and swaps the corresponding elements.

```perl
sub swap {
    my ($self, $i, $j) = @_; 
    my $aref = $self->{aref}; 
    ($aref->[key($i)], $aref->[key($j)]) = ($aref->[key($j)], $aref->[key($i)]);
}
```

The first line gets the parameters, and the second unpacks `aref`. The third line uses list assignment to swap elements.

At this point, if you run the test code again, the first element you remove should be the largest. That’s because `add` establishes the heap property. But `remove` doesn’t maintain it.
4.10 Trickle down

It is often useful to think of a tree as a root node with two subtrees, as shown in this figure:

The value at the root is $r$. The value at the root of Subtree A is $a$ and the value at the root of Subtree B is $b$.

If the tree is a heap, then $r$ is the largest value in the heap and $a$ and $b$ are the largest values in their respective subtrees.

Once we remove $r$, we have to restore the heap property. One way to do that is to remove the last element of the array, which we’ll call $c$, and put it at the root.

Of course, the chances are that the last value is not the highest, so putting it at the root breaks the heap property. Fortunately it is easy to restore. We know that the largest value in the heap is either $a$ or $b$. Therefore we can select whichever is larger and swap it with the value at the root.

Arbitrarily, let’s say that $b$ is larger. Since we know it is the highest value left in the heap, we can put it at the root and put $c$ at the top of Subtree B. Now the situation looks like this:
Again, c is the value we copied from the last element in the array and b is the highest value left in the heap. Since we haven’t changed Subtree A at all, we know that it is still a heap. The only problem is that we don’t know if Subtree B is a heap, since we just stuck a (probably low) value at its root.

So we haven’t completely solved the problem, but we have taken a step in the right direction, and the result is a smaller version of the same problem. All we have to do now is reheapify Subtree B.

Here is an updated version of Heap::remove

```perl
sub remove {
    my $self = shift;
    my $aref = $self->{aref};
    my $result = $aref->[1];
    $aref->[1] = pop @$aref;
    $self->{next}--;
    $self->reheapify(1);      # reheapify the tree
    return $result
}
```

The only difference is that we invoke reheapify to restore the heap property. Here is the implementation.

```perl
sub reheapify {
    my ($self, $i) = @_;
    my $left = 2 * $i;
    my $right = 2 * $i + 1;
    my $winleft = $self->compare($i, $left) >= 0;
    my $winright = $self->compare($i, $right) >= 0;
    return if $winleft and $winright;
    if ($self->compare ($left, $right) > 0) {
        $self->swap($i, $left);
        $self->reheapify($left);
    } else {
        $self->swap($i, $right);
        $self->reheapify($right);
    }
}
```

The first three lines get the parameters (as usual) and compute the indices of the children.

The next two lines compare the root node to each of its children and store the results. If the root is greater than either of its children, we can return immediately. Otherwise we have to find the larger of the children and swap.

If we swap with the left child, we have to invoke reheapify on the left subtree,
and vice versa. But either way, we only make one recursive invocation, so the number of recursions is just the height of the tree.

4.11 Exercises

Exercise 4.1 This implementation demonstrates the basic mechanisms of a heap, but it is not all that useful because it can only store scalars. We would like to be able to put any kind of object into a heap, as long as we have a way to compare objects.

- Write a new package called Thing. Write a constructor and a method that prints the instance variables of a Thing object.
- Modify the test code so that, instead of adding random numbers to the heap, it adds Thing objects with an instance variable named priority. For now, the priority of the Things should be a random number, but in general the priority of an object in a heap is the value that determines the order objects are removed.
- Write an accessor method for Thing named priority. You will not be surprised to hear that it should return the object’s priority.
- Modify the test code so that when it removed objects from the heap, it invokes print on them.
- At this point, you should be able to add and remove objects from the heap, but they are probably not coming out in the right order because compare doesn’t know how to compare Things. Modify compare so that instead of comparing references to objects (whatever that means), it invokes the method priority on each object and compares the results.

If everything goes according to plan, your heap implementation should be polymorphic in the sense that it can hold many kind of objects as elements, and it will work correctly as long as the objects have an instance variable named priority.

Save your implementation in a file named heap.pl; we will need it in the next chapter.
Chapter 5

Modules

In this chapter, we develop an implementation of a Huffman code, which is interesting enough in its own right, but it also serves as a demonstration of several Perl features. First, we will transform the heap program from the previous chapter into a full-fledged Perl module. Next we will define two new classes, one of which inherits from the other. Finally, as an example of a linked data structure, we will implement a binary tree.

5.1 Variable-length codes

If you are familiar with Morse code, you know that it is a system for encoding the letters of the alphabet as a series of dots and dashes. For example, the famous signal \ldots\ldots represents the letters SOS, which comprise an internationally-recognized call for help. This table shows the rest of the codes:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.-</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>-...</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>-.-.</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>-..</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>.</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>..-.</td>
<td>6</td>
</tr>
<tr>
<td>G</td>
<td>--.</td>
<td>7</td>
</tr>
<tr>
<td>H</td>
<td>----</td>
<td>8</td>
</tr>
<tr>
<td>I</td>
<td>-----</td>
<td>9</td>
</tr>
<tr>
<td>J</td>
<td>-----</td>
<td>0</td>
</tr>
<tr>
<td>K</td>
<td>-..-</td>
<td>/</td>
</tr>
<tr>
<td>L</td>
<td>-.--</td>
<td>+</td>
</tr>
<tr>
<td>M</td>
<td>--.</td>
<td>=</td>
</tr>
<tr>
<td>N</td>
<td>-.</td>
<td>\</td>
</tr>
<tr>
<td>O</td>
<td>---</td>
<td>,</td>
</tr>
<tr>
<td>P</td>
<td>.--.</td>
<td>?</td>
</tr>
<tr>
<td>Q</td>
<td>--.-</td>
<td>(</td>
</tr>
<tr>
<td>R</td>
<td>.--</td>
<td>)</td>
</tr>
<tr>
<td>S</td>
<td>...-</td>
<td>&quot;</td>
</tr>
<tr>
<td>T</td>
<td>-....</td>
<td>'</td>
</tr>
<tr>
<td>U</td>
<td>----.</td>
<td>-</td>
</tr>
<tr>
<td>V</td>
<td>-----</td>
<td>‘</td>
</tr>
<tr>
<td>W</td>
<td>-----</td>
<td>:</td>
</tr>
<tr>
<td>X</td>
<td>----.</td>
<td>;</td>
</tr>
<tr>
<td>Y</td>
<td>-.--</td>
<td>$</td>
</tr>
<tr>
<td>Z</td>
<td>--.</td>
<td>\</td>
</tr>
</tbody>
</table>

Notice that some codes are longer than others. By design, the most common letters have the shortest codes. Since there are a limited number of short codes, that means that less common letters and symbols have longer codes. A typical
message will have more short codes than long ones, which minimizes the average transmission time per letter.

Codes like this are called variable-length codes. In this chapter, we will look at an algorithm for generating a variable-length code called a **Huffman code**. It is an interesting algorithm in its own right, but it also makes a useful exercise because its implementation uses a variety of data structures.

Here is an outline of the next few sections:

- First, we will use a sample of English text to generate a table of characters and their frequencies.
- The heart of a Huffman code is the Huffman tree. We will use the frequency table to build the Huffman tree, and then use the tree to encode and decode sequences.
- Finally, we will traverse the Huffman tree and built a code table, which contains the sequence of dots and dashes for each letter.

## 5.2 The frequency table

Since the goal is to give short codes to common letters, we have to know how often each letter occurs. In Edgar Allen Poe’s short story “The Gold Bug,” one of the characters uses letter frequencies to crack a cypher. He explains,

> “Now, in English, the letter which most frequently occurs is e. Afterwards, the succession runs thus: a o i d h n r s t u y c f g l m w b k p q x z. E however predominates so remarkably that an individual sentence of any length is rarely seen, in which it is not the prevailing character.”

So our first mission is to see whether Poe got it right. To check, I chose as my sample the text of “The Gold Bug” itself, which I downloaded from [gutenberg.net](http://gutenberg.net).

**Exercise 5.1** Write a program that counts the number of times each letter appears in a sample text. Download the text of your favorite public-domain short story, and analyze the frequency of the letters.

You can traverse the characters in a string by using `split` with an empty string as a separator. The result is a list of characters. The following example splits a string and prints one character per line.

```perl
my @list = split "", $line;
foreach (@list) {
    print "$_\n";
}
```
5.3 Modules

We have already seen how to use an existing module with the use statement. Now we will see how to create a new module.

In an object-oriented program, a module is usually a file that defines a package. Starting with the heap implementation from the previous chapter, we can create a module in just a few steps:

1. Rename the file from heap.pl, which indicates that it is a program, to heap.pm, which indicates that it is a module.

2. Remove or comment out the test code. When a program uses a module, Perl executes the module at compile time. You can put executable code in a module if you want to, but most of the time modules only contain subroutines and variable declarations.

3. The shebang line at the beginning of the file is unnecessary, since we will never execute a module from the command line.

4. The last expression in the module should be a “true” value, which indicates that the module executed successfully. By convention, the last statement is 1;

Exercise 5.2 Convert heap.pl from the previous chapter into a module named heap.pm, and add the statement use heap; to huffman.pl. Make sure you can compile and run the program, even if it doesn’t use anything from the heap module yet.

Exercise 5.3 Write a class called Pair that represents a letter-frequency pair. Pair objects should contain instance variables named letter and frequency, and provide methods with the same names that access them. In addition, Pair should provide a method named priority that returns the frequency.

Now sort the letter-frequency pairs from the frequency table by traversing the set of keys, building Pair objects, adding the Pairs to a heap, removing the Pairs from the heap, and printing them in descending order by frequency.

How good was Poe’s guess about the most frequent letters?

5.4 The Huffman Tree

The next step is to assemble the Huffman tree. Each node of the tree contains a letter and its frequency, and pointers to the left and right nodes.

To assemble the Huffman tree, we start by creating a set of singleton trees, one for each entry in the frequency table. Then we build the tree from the bottom up, starting with the least-frequent letters and iteratively joining subtrees until we have a single tree that contains all the letters.

Here is the algorithm in more detail.
1. For each entry in the frequency table, create a singleton Huffman tree and add it to a heap. When we remove a tree from the heap, we get the one with the lowest frequency.

2. Remove two trees from the heap and join them by creating a new parent node that refers to the removed nodes. The frequency of the parent node is the sum of the frequencies of the children.

3. If the heap is empty, we are done. Otherwise, put the new tree into the heap and go to Step 2.

An example will make this clearer. To keep things manageable, we will use a sample text that only contains the letters **adnrst**:

Eastern Tennessee anteaters ensnare and eat red ants, detest ant antennae (a tart taste) and dread Antarean anteater-eaters. Rare Andean deer eat tender sea reeds, aster seeds and rats’ ears. Dessert? Rats’ asses.

This eloquent treatise on the eating habits of various fauna yields the following frequency table:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>40</td>
</tr>
<tr>
<td>a</td>
<td>32</td>
</tr>
<tr>
<td>t</td>
<td>24</td>
</tr>
<tr>
<td>s</td>
<td>22</td>
</tr>
<tr>
<td>n</td>
<td>20</td>
</tr>
<tr>
<td>r</td>
<td>19</td>
</tr>
<tr>
<td>d</td>
<td>13</td>
</tr>
</tbody>
</table>

So after Step 1, the heap looks like this:

```
  40
 e
 32
 a
t
 24
 s
 22
 n
 20
 r
 19
 d
 13
```

In Step 2, we remove the two trees with the lowest frequency (r and d) and join them by creating a parent node with frequency 32. The letter value for the internal nodes is irrelevant, so it is omitted from the figures. When we put the new tree back in the heap, the result looks like this:

```
  32
 e
 40
 a
 32
t
 24
 s
 22
 n
 20
 r
 19
 d
 13
```

Now we repeat the process, combining s and n:
After the next iteration, we have the following collection of trees. By the way, a collection of trees is called a \textit{forest}.

![Huffman Tree Diagram]

After two more iterations, there is only one tree left:

![Huffman Tree Diagram]

This is the Huffman tree for the sample text. Actually, it is not the only one, because each time we join two trees, we choose arbitrarily which goes on the left and which on the right, and when there is a tie in the heap, the choice is arbitrary. So there may be many possible trees for a given sample.

So how do we get a code from the Huffman tree? The code for each letter is determined by the path from the root of the tree to the leaf that contains the letter. For example, the path from the root to \texttt{s} is left-right-left. If we represent a left with \texttt{.} and a right with \texttt{-} (another arbitrary choice) we get the following code table.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>-.</td>
</tr>
<tr>
<td>a</td>
<td>--</td>
</tr>
<tr>
<td>t</td>
<td>..-</td>
</tr>
<tr>
<td>s</td>
<td>.-.</td>
</tr>
<tr>
<td>n</td>
<td>.--</td>
</tr>
<tr>
<td>r</td>
<td>....</td>
</tr>
<tr>
<td>d</td>
<td>...-</td>
</tr>
</tbody>
</table>

Notice that we have achieved the goal: the most frequent letters have the shortest codes.

\textbf{Exercise 5.4} By hand, figure out a Huffman tree for the following frequency table:

\begin{verbatim}
e 93
s 71
r 57
t 53
n 49
i 44
d 43
\end{verbatim}
5.5 Inheritance

One way to implement HuffTree is to extend Pair from Exercise 5.3. A new class specifies its parent class (or classes) by declaring an array named @ISA, which is pronounced “is a”, because it is meant to indicate that the new class “is a” subset of the parent class.

```perl
package HuffTree;
our @ISA = "Pair";
```

**Exercise 5.5** Write methods named `left` and `right` that access the instance variables with the same names. These variables will contain references to other HuffTrees, making it possible to assemble nodes into a tree.

The HuffTree class inherits `new`, `letter`, `frequency` and `priority` from Pair. As you would expect, you can override an inherited method if you need a different implementation. In this case, when we remove a HuffTree from the heap, we want to get the one with the lowest frequency, so we have to override `priority` like this:

```perl
sub priority {
    my $self = shift;
    return -$self->{frequency};
}
```

The highest frequency gets the lowest priority.

5.6 Building the Huffman tree

Given the frequency table, we can build the Huffman tree in two steps. First, we traverse the frequency table and build a singleton tree for each entry, and add the singletons to a heap:

```perl
my $heap = Heap->new;
my $tree;
while ((my $key, my $value) = each %freqtab) {
    print "$key\t$value
";
    my $tree = HuffTree->new( letter => $key,
                              frequency => $value );
    $heap->add($tree);
}
```

The second step is to assemble the tree using the algorithm in Section 5.4.

```perl
while (1) {
    my $left = $heap->remove;
    my $right = $heap->remove;
```
my $freq = $left->frequency + $right->frequency;

$tree = HuffTree->new( frequency => $freq );
$tree->left($left);
$tree->right($right);

last if $heap->empty;
$heap->add($tree);
}

For the internal nodes of the tree, we set the frequency but we leave letter undefined.

When the loop exits, all the trees have been assembled into one, and $tree refers to the root node.

Exercise 5.6 Assemble this code into a method named hufftree that reads the package variable named %freqtab, builds a HuffTree, and assigns it to a package variable named $hufftree.

5.7 Building the code table

The code for a given letter is determined by the path from the root of the Huffman tree to the leaf node that contains the letter. To encode a letter, we might have to search the entire tree.

This process is much more efficient if we traverse the tree once, compute all the codes, and record them in a code table.

As we traverse the tree, we want to keep track of the path we are on. At first, that might seem hard, but there is a natural way to perform this computation recursively. Here is the key observation: if the path from the root to a given node is represented by a string of dots and dashes called path, then the path to the left child of the node is $path . "." and the path to the right child is $path . "-" (the dot operator performs string concatenation).

Here is an implementation of this algorithm:

sub codetab {
    our %codetab;
    my ($tree, $path) = @_;

    if (defined ($tree->letter)) {
        $codetab{$tree->letter} = $path;
        return;
    }
    codetab($tree->left, "$path.");
    codetab($tree->right, "$path-");}


If the value of letter is defined, that means we have reached a leaf node, so we can make an entry in the code table and return. Otherwise, we make recursive calls to traverse the left and right subtrees.

Initially, we invoke codetab with the root of the tree and an empty string as parameters:

    codetab $hufftree, "";

Exercise 5.7

a. Type in the codetab method and test it with your Huffman tree.
b. Write a subroutine called print_codetab that prints the entries in the code table. You should see that common letter have short codes and rare letters have long codes. Compare your Huffman code with Morse code. Which has longer codes?
c. Write a subroutine called encode that traverses a string, looks up each character in the code table, and returns the encoding of the string.

5.8 Decoding

When we receive an encoded message, we use the Huffman tree to decode it. Here is the algorithm:

1. Start at the root of the HuffTree.
2. If the next symbol is -, go to the left child; otherwise, go to the right child.
3. If you are at a leaf node, get the letter from the node and append it to the result. Go back to the root.
4. Go to Step 2.

Consider the code "..--.--..", as an example. Starting at the top of the tree, we go left-left-right and get to the letter t. Then we start at the root again, go right-left and get to the letter e. Back to the top, then right-right, and we get the letter a. If the code is well-formed, we should be at a leaf node when the code ends. In this case the message is my beverage of choice, tea.

Exercise 5.8 Use the example HuffTree to decode the following words:

a. "..--.--.--.."
b. "..--.--.--.--.--.
   c. "...--.--.--.--.
   d. "--.--.--.--

Notice that until you start decoding, you can’t tell how many letters there are or where the boundaries fall.
Exercise 5.9

a. Write a method called `decode` that takes a string of dots and dashes and that uses the Huffman tree to decode the string and return the result.

b. Test your method by using `encode` to encode a string and `decode` to decode it.

Exercise 5.10 The implementation I have sketched in this chapter is partly object-oriented, since it uses `Pair` and `HuffTree` objects, but the Huffman package is just a collection of subroutines, not a class. Since it uses package variables for `%freqtab`, `$hufftree` and `%codetab`, a program that uses this package can only have one Huffman code at a time.

To make this design more versatile, rewrite the program to define a `Huffman` class that contains `$freqtab`, `$hufftree` and `$codetab` as instance variables. Notice that, in this implementation, the frequency and code tables have to be references to hashes.

What are the pros and cons of the two implementations?
Chapter 6

Callbacks and pipes

In this chapter, we will develop two programs. One, named `get.pl`, uses HTTP to download a file from a Web site. The other, named `mirror.pl`, takes a URI as a parameter and uses `get.pl` to download it. As `get.pl` discovers unvisited links, `mirror.pl` creates additional processes to download them.

These programs demonstrate the use of callbacks, which provide a flexible way to interact with existing code in modules, and pipes, which provide a simple form of inter-process communication. Along the way, we will play with some standard (and some less standard) Perl modules like `URI`, `HTTP::Request`, `HTML::Parser`, and `IO::Select`.

6.1 URIs

One of the most powerful features of Perl is its extensive library of modules. Many of these modules are available from the Comprehensive Perl Archive Network (CPAN), conveniently located at `cpan.org`.

Some modules are more standard than others. Some Perl Core Modules can be considered part of the language. Other modules, like the ones we will use in this chapter, are included in the vast majority of Perl installations.

Different modules offer different interfaces. The ones we will see in this chapter tend to be object-oriented. They define a new class of objects and the methods that operate on them. Others, like `POSIX`, provide a set of subroutines.

To get warmed up, we’ll start with a simple class called `URI`, which stands for **Uniform Resource Identifier**. A URI is pretty much the same thing as a URL, which identifies a resource in the World Wide Web (see Section 2.6).

Here is a program that takes a URL as a string, creates a new URI object, and converts it to canonical form:
use URI;

sub main {
    my $uris = shift || "http://www.slashdot.com";
    my $uri = URI->new($uris)->canonical;
    $uris = $uri->as_string;
    print "$uris\n";
}

main @ARGV;

The first line of main gets the parameter from the list of command-line arguments. If the list is empty, it uses the default value http://www.slashdot.com. The second line creates a URI object, and then invokes canonical, which returns the original object if it is already in canonical form, or a new object if, for example, the original uses non-standard capitalization or a redundant port number. Here is an example:

$ ./uri.pl HTTP://www.Slashdot.org:80/Index.html
http://www.slashdot.org/Index.html

canonical converts HTTP to http, and makes the machine name all lower case. But the file name is case sensitive, so it is unchanged. The port specifier :80 is unnecessary because 80 is the standard port for the web server.

URI also provides methods that extract the various parts of a URI:

    my $scheme = $uri->scheme;
    my $authority = $uri->authority;
    my $path = $uri->path;

The scheme is usually the protocol name, like http or ftp. The authority is usually the name of the machine that provides this resource. The path is the complete file name (including its parent directories).

### 6.2 HTTP GET

In order to retrieve a file from a web server, a client has to issue an HTTP GET request and wait for a response. The World-Wide Web Library for Perl (LWP) provides a set of objects and methods that make it almost trivial to implement a web client.

The three components we will use are UserAgent, Request and Response.

use LWP::UserAgent;
use HTTP::Request;
use HTTP::Response;

Request objects represent the messages sent from client to server; Response objects represent the replies. A UserAgent provides the capability to send and receive these messages. Here is the simplest form of an HTTP GET request.
my $ua = LWP::UserAgent->new;
my $request = HTTP::Request->new(GET => $uri);
my $response = $ua->request($request);
print $response->content;

The first line simply creates a UserAgent. The second line create the Request object, specifying the URI and the fact that this is a GET request (as opposed to HEAD or POST, for example).

To get the file, we tell the UserAgent to process the request. The result is a Response object that contains the content of the file along with the header. The last line extracts and prints the content.

For more information about all three classes (and more) see the LWP documentation at http://search.cpan.org/dist/libwww-perl/.

### 6.3 Callbacks

In the previous section, we used the request method in the UserAgent class to download an entire file and then write the contents in a local file. But often the performance of wide-area networks is slow (compared to local processing) and unpredictable. It would be more efficient to start writing the file as soon as the first packet arrives, so that we can interleave network arrivals with disk accesses.

The request method can accept a second argument, which is a reference to a subroutine. As packets of data arrive, request invokes the given function, passing in chunks of the file. A subroutine that is passed as a parameter and then invoked by the receiver is called a callback. Here is an example:

```perl
sub callback {
    my $chunk = shift;
    print FILE $chunk;
}
```

Now we can invoke request like this:

```perl
open FILE, "">", $file or croak "Couldn’t open $file";
my $request = HTTP::Request->new(GET => $uri);
my $response = $ua->request($request, &callback);
```

**Exercise 6.1** Assemble the code in this section into a program named get.pl that downloads a file using a callback subroutine to write chunks of data as they arrive. Add a line to callback to print the length of each chunk. Do you see a pattern? Can you figure out what determines the sizes of the chunks?

### 6.4 Mirroring

The next step for this program is to turn it into a tool that mirrors a web page. That is, we would like to make a local file (or directory) that has the same name and contents as a file (or directory) on a remote site.
To do that, we’ll write a few lines of code to create a directory, if necessary, and then open the file. Given the path part of the URI, we find the parent directory using the `dirname` method from the `File::Basename` package.

```perl
my $path = $uri->path;
my $dir = $uri->authority . File::Basename::dirname $path;
```

The next step is to create the directory.

```perl
my $res = system "mkdir -p $dir";
if ($res != 0) { croak "Couldn’t create directory $dir" }
```

Annoyingly, the return values from many UNIX commands do not follow the Perl convention. In Perl, any true value indicates success and the false value indicates an error. For most UNIX commands, the value 0 indicates success, and all other values are error codes.

The last step is to assemble the file name and open the file.

```perl
my $file = $uri->authority . $path;
open FILE, "">", $file or croak "Couldn’t open $file"
```

We are using the `authority` part of the URI as the top-level directory, so that all the pages we get from a given site go into the same directory.

**Exercise 6.2** Modify `get.pl` so that when it downloads a file, it stores it in a local directory and file that mirror the original site.

**Exercise 6.3** There is a detail of HTTP that makes this part of the program a little tricky. If a URI identifies a directory (rather than a file), the server looks for a file in the directory named `index.html` and sends it. Unfortunately, there is no sure way to know, looking at a URI, whether it identifies a directory.

One way to handle this problem is to check whether the path part of a URI ends in `.html`. If not, the program could append `/index.html` to the path. But there are a number of ways this solution could fail. As an exercise, you might want to investigate better ways to handle this problem.

### 6.5 Parsing

As we read the contents of a Web page, we might also like to extract the links it contains. The simplest way to do that is to look for patterns that match the `<a>` tag. But there is a module called `HTML::Parser` that is even easier (once you figure out how to use it), and gives us another chance to use callbacks.

Before we start parsing, we have to create a `Parser` object.
our $p = HTML::Parser->new(
    start_h => \&start_tag, "tagname, attr"
);  
The parameters we send to the constructor are, as usual, a list of instance variables and their values. In this case, the instance variable is start_h, which specifies a handler that will be invoked when the parser sees a start tag. A handler is a kind of callback that is invoked to handle a particular kind of event. The Parser uses handlers for several other kinds of events, including end_h, which handles end tags, and text_h which handles plain text that is not part of a tag.

The value of start_h is a list that contains a reference to the callback, which is cleverly named start_tag, and a string that specifies what information should be passed as parameters to the callback. In this case, we want the name of the tag and a hash that contains the attributes that appear in the tag.

For example, the following HTML generates a text link.

```html
<a href="http://allendowney.com">my web page</a>
```

When this code is parsed, it generates three events. The first is a start tag with tagname a and a hash of attributes that contains one entry, href => http://allendowney.com. The next event is a text event with the text my web page. The last event is an end tag with tagname a (and no attributes).

Here is a version of start_tag that waits for <a> links and prints the URI they refer to.

```perl
sub start_tag {
    my ($tagname, $attr) = @_;  
    if ($tagname eq "a") {
        my $href = $attr->{href};  
        print "$href\n";
    }
}
```

The Parser object is incremental, which means that you can invoke parse repeatedly, passing any size chunk of HTML code. This capability works well with the code from the previous section that stores the incoming file in chunks. We can modify callback so that it stores each chunk in a file and passes it to the Parser.

```perl
sub callback {
    my $line = shift;  
    our $p->parse($line);  
    print FILE $line;
}
```

Callbacks are a versatile way to integrate a module with your program. The biggest drawback is that the flow of execution in the program can get complicated. In this case, when a chunk of a file arrives, the UserAgent invokes callback, which passes the chunk to the Parser. If the chunk contains a start tag, the Parser invokes start_tag.
6.6 Absolute and relative URIs

The URIs we have seen so far have been absolute, which means that they include the scheme, authority, and path. The references that appear in `<a>` tags can be relative, which means that they include only a path.

For example, `http://allendowney.com/index.html` is the absolute URI of a file. If that file contains a relative URI as a reference, we assume that the reference is relative to the parent of the current file. For example, a reference to `projects.html` would be understood to refer to `http://allendowney.com/projects.html`. The parent of the current file is called the base URI.

The URI class provides methods for converting from relative to absolute URIs, and back. The `abs` method converts a relative URI to an absolute URI, given the base URI as a parameter. Here is an example.

```perl
my $base = URI->new("http://allendowney.com/index.html");
my $rel = URI->new("projects.html");
my $abs = $rel->abs($base);
print "$abs\n";
```

The output of this code fragment is `http://allendowney.com/projects.html`.

Exercise 6.4 Modify `start_tag` so that whenever it find a relative URI, it converts it to an absolute URI using the URI of the current page as a base.

Exercise 6.5 Modify `start_tag` so that it only prints “internal” links; that is, links to URIs that have the same authority as the base URI.

6.7 Multiple processes

At this point we have a program that downloads a given URI, stores the result in an appropriately named file, and prints the URIs of any links it finds. We will use this program as a component of a robot, which is a program that traverses a Web site by downloading files and following links until it gets all the files (or at least all the files that can be reached by following links from the initial URI).

For several reasons, it is useful to make programs like this multithreaded, which means that there are several threads of execution running concurrently. The reasons are:

- Networks are slow compared to local processing, so it is often faster to run several network connections at the same time.

- Network performance is highly variable. When one connection is slow or stalled, another might be making progress.

Of course, if you have multiple connections to the same site, they compete with each other for resources in the network and on the server. So there is a point where increasing the number of connections is no longer useful.
6.8 Family planning

In the next couple of sections, we will develop a parent program that creates child processes to download files. The parent program will start with a single URI, and create a single child to download it. As the child discovers unvisited links, the parent will create additional children to download them. The role of the parent is to control the number of children that run concurrently and to maintain a list of URIs that are waiting to be downloaded.

Here is the main loop of the program.

```perl
sub main {
    our @list = @_;  
    our %seen;  
    our $procs = 0;  
    our $s = IO::Select->new();  

    while (1) {
        while (@list > 0 && $procs < 6) {
            get shift @list;
        }  

        my @ready = $s->can_read;  

        foreach my $fh (@ready) {
            process $fh;
        }  

        last if (@list == 0 && $procs == 0);
    }
}
```

@list is the list of URIs waiting to be downloaded. Initially we get the list from the command-line arguments. %seen is a hash that contains the URIs we have already downloaded. Before we add a URI to @list, we make sure is is not already in %seen. $procs is the number of child processes currently running.

$s is an IO::Select object, which provides a convenient interface to the select system call. Right in the middle of the main loop, we invoke can_read on this object to get a list of the file handles that have data ready to read; that is, the list of child processes that have found a link.

The main loop contains two inner loops. The first controls the number of child processes; the second collects links from the children and adds them to the list.

The main loop exits when @list is empty and there are no child processes.

6.9 Creating children

Each time through the main loop, we check @list and create up to 6 child processes. For each element in the list, we invoke get.
sub get {
    our (%seen, $procs, $s);
    my $uris = shift;

    open my $fh, "./get.pl $uris |" or croak "can't fork";
    $seen{$uris}++;
    $procs++;
    $s->add($fh);
}

We have used the `open` operator before to open files. Here, we are using `open` to create a pipe. A pipe is a child process that communicates with the parent process through a file handle. In this case, the child process generates output using the `print` operator, and the parent reads it through the file handle.

If the `open` operator succeeds, we update `%seen` and `$procs`, and then add the new file handle, `$fh`, to the list of file handles being monitored by the `Select` object. Back in the main loop, when we invoke `can_read`, it tells us which of the file handles are ready to be read.

### 6.10 Talking back to parents

`can_read` doesn't complete until one or more file handles are ready. When it completes, we traverse the list of file handles and invoke `process` for each one.

sub process {
    our (@list, %seen, $procs, $s);
    my $fh = shift;
    my $line = <$fh>;
    chomp $line;

    if ($line eq "done") {
        close $fh;
        $s->remove($fh);
        $procs--;
    } else {
        push @list, $line unless defined $seen{$line};
    }
}

`process` uses the angle operator to read a single line from the given file handle, and `chomp` to remove the newline character.

Sometimes it is not a good idea to use the angle operator with `select`, but in this case, we know that whenever a child process produces data, it produces an entire line. So if a file handle is ready to be read, we know that we can read an entire line. Otherwise, the angle operator might block waiting for a child to produce the rest of a line.
process handles two cases. If the line we read is done, then we know the child
process is done and we can close the file handle, remove it from the list of file
handles in the Select object, and decrement the number of processes.

If the line is a link discovered by a child process, we add it to the list of URIs
waiting to be downloaded, unless it is already in %seen.

Exercise 6.6 Assemble the code from the previous sections into a program named
mirror.pl and test it by downloading files from a friendly Web site. You should
use some caution, because the program as written is very aggressive. Downloading
multiple files at the same time from the same site imposes a lot of load on the server.
Robots are supposed to be more polite than than. For more information, see the
documentation of LWP::RobotUA, which is a variation of UserAgent that imposes delays
between downloads.

6.11 Exercises

Exercise 6.7 After you have made a local copy of a Web site, you might want to
update it periodically. Of course, you could just download all the files (and any new
ones) again, but it would be more efficient to download only the new files and the files
that have been modified.

Modify the mirror program so that, before downloading a file, it checks whether you
already have a local copy. If so, then it should get the modification date of the local
file and issue a conditional request that will only download the file again if it has been
modified since the last time you downloaded it.

Hints: see the stat command and the File::Stat modules for two ways to get the
modification time of a local file. See the documentation of HTTP::Headers to see how
to make a conditional request.

Exercise 6.8 The mirror program in this chapter doesn’t handle redirection.
If an HTML request is redirected, the URI of the file that is provided won’t have
the same name as the URI that was requested. For example, if you ask for

Ideally, the mirror program should invoke base on the Response object and use the
reply to determine the file name. Unfortunately, if we use the callback form of
UserAgent::request, we don’t get the Response object until the transfer is complete.

As a challenge, think of several ways you could work around this problem and, if you
are feeling ambitious, implement the one you think is best.

Exercise 6.9 The mirror program in this chapter doesn’t handle fragments, which
are URIs that refer to a part of a Web page. See the documentation of URI for more
information and think about how you might fix the program.