

# RECURSION AND CALLBACKS

The first “advanced” technique we’ll see is recursion. *Recursion* is a method of solving a problem by reducing it to a simpler problem of the same type.

Unlike most of the techniques in this book, recursion is already well known and widely understood. But it will underlie several of the later techniques, and so we need to have a good understanding of its fine points.

## 1.1 DECIMAL TO BINARY CONVERSION

Until the release of Perl 5.6.0, there was no good way to generate a binary numeral in Perl. Starting in 5.6.0, you can use `sprintf("%b", $num)`, but before that the question of how to do this was Frequently Asked.

Any whole number has the form  $2k + b$ , where  $k$  is some smaller whole number and  $b$  is either 0 or 1.  $b$  is the final bit of the binary expansion. It’s easy to see whether this final bit is 0 or 1; just look to see whether the input number is even or odd. The rest of the number is  $2k$ , whose binary expansion is the same as that of  $k$ , but shifted left one place. For example, consider the number  $37 = 2 \cdot 18 + 1$ ; here  $k$  is 18 and  $b$  is 1, so the binary expansion of 37 (100101) is the same as that of 18 (10010), but with an extra 1 on the end.

How did I compute the expansion for 37? It is an odd number, so the final bit must be 1; the rest of the expansion will be the same as the expansion of 18. How can I compute the expansion of 18? 18 is even, so its final bit is 0, and the rest of the expansion is the same as the expansion of 9. What is the binary expansion for 9? 9 is odd, so its final bit is 1, and the rest of its binary expansion is

the same as the binary expansion of 4. We can continue in this way, until finally we ask about the binary expansion of 1, which of course is 1.

This procedure will work for any number. To compute the binary expansion of a number  $n$  we proceed as follows:

1. If  $n$  is 1, its binary expansion is 1, and we may ignore the rest of the procedure. Similarly, if  $n$  is 0, the expansion is simply 0. Otherwise:
2. Compute  $k$  and  $b$  so that  $n = 2k + b$  and  $b = 0$  or 1. To do this, simply divide  $n$  by 2;  $k$  is the quotient, and  $b$  is the remainder, 0 if  $n$  was even, and 1 if  $n$  was odd.
3. Compute the binary expansion of  $k$ , using this same method. Call the result  $E$ .
4. The binary expansion for  $n$  is  $Eb$ .

Let's build a function called `binary()` that calculates the expansion. Here is the preamble, and step 1:

```
CODE LIBRARY
binary
sub binary {
  my ($n) = @_ ;
  return $n if $n == 0 || $n == 1;
```

Here is step 2:

```
my $k = int($n/2);
my $b = $n % 2;
```

For the third step, we need to compute the binary expansion of  $k$ . How can we do that? It's easy, because we have a handy function for computing binary expansions, called `binary()` — or we will once we've finished writing it. We'll call `binary()` with  $k$  as its argument:

```
my $E = binary($k);
```

Now the final step is a string concatenation:

```
return $E . $b;
}
```

This works. For example, if you invoke `binary(37)`, you get the string 100101.

The essential technique here was to reduce the problem to a simpler case. We were supposed to find the binary expansion of a number  $n$ ; we discovered that this binary expansion was the concatenation of the binary expansion of a smaller number  $k$  and a single bit  $b$ . Then to solve the simpler case of the same problem, we used the function `binary()` in its own definition. When we invoke `binary()` with some number as an argument, it needs to compute `binary()` for a different, smaller argument, which in turn computes `binary()` for an even smaller argument. Eventually, the argument becomes 1, and `binary()` computes the trivial binary representation of 1 directly.

This final step, called the *base case* of the recursion, is important. If we don't consider it, our function might never terminate. If, in the definition of `binary()`, we had omitted the line:

```
return $n if $n == 0 || $n == 1;
```

then `binary()` would have computed forever, and would never have produced an answer for any argument.

## 1.2 FACTORIAL

Suppose you have a list of  $n$  different items. For concreteness, we'll suppose that these items are letters of the alphabet. How many different orders are there for such a list? Obviously, the answer depends on  $n$ , so it is a function of  $n$ . This function is called the *factorial function*. The factorial of  $n$  is the number of different orders for a list of  $n$  different items. Mathematicians usually write it as a postfix (!) mark, so that the factorial of  $n$  is  $n!$ . They also call the different orders *permutations*.

Let's compute some factorials. Evidently, there's only one way to order a list of one item, so  $1! = 1$ . There are two permutations of a list of two items: A-B and B-A, so  $2! = 2$ . A little pencil work will reveal that there are six permutations of three items:

```
C AB   C BA
A C B   B C A
AB C    BA C
```

How can we be sure we didn't omit anything from the list? It's not hard to come up with a method that constructs every possible ordering, and in Chapter 4 we will see a program to list them all. Here is one way to do it. We can make any list of three items by adding a new item to a list of two items. We have two choices

for the two-item list we start with: AB and BA. In each case, we have three choices about where to put the C: at the beginning, in the middle, or at the end. There are  $2 \cdot 3 = 6$  ways to make the choices together, and since each choice leads to a different list of three items, there must be six such lists. The preceding left column shows all the lists we got by inserting the C into AB, and the right column shows the lists we got by inserting the C into BA, so the display is complete.

Similarly, if we want to know how many permutations there are of four items, we can figure it out the same way. There are six different lists of three items, and there are four positions where we could insert the fourth item into each of the lists, for a total of  $6 \cdot 4 = 24$  total orders:

```
D ABC   D ACB   D BAC   D BCA   D CAB   D CBA
A D BC  A D CB  B D AC  B D CA  C D AB  C D BA
AB D C  AC D B  BA D C  BC D A  CA D B  CB D A
ABC D   ACB D  BAC D  BCA D  CAB D  CBA D
```

Now we'll write a function to compute, for any  $n$ , how many permutations there are of a list of  $n$  elements.

We've just seen that if we know the number of possible permutations of  $n - 1$  things, we can compute the number of permutations of  $n$  things. To make a list of  $n$  things, we take one of the  $(n - 1)!$  lists of  $n - 1$  things and insert the  $n$ th thing into one of the  $n$  available positions in the list. Therefore, the total number of permutations of  $n$  items is  $(n - 1)! \cdot n$ :

```
sub factorial {
  my ($n) = @_;
  return factorial($n-1) * $n;
}
```

Oops, this function is broken; it never produces a result for any input, because we left out the termination condition. To compute `factorial(2)`, it first tries to compute `factorial(1)`. To compute `factorial(1)`, it first tries to compute `factorial(0)`. To compute `factorial(0)`, it first tries to compute `factorial(-1)`. This process continues forever. We can fix it by telling the function explicitly what 0! is so that when it gets to 0 it doesn't need to make a recursive call:

**CODE LIBRARY**  
factorial

```
sub factorial {
  my ($n) = @_;
  return 1 if $n == 0;
  return factorial($n-1) * $n;
}
```

Now the function works properly.

It may not be immediately apparent why the factorial of 0 is 1. Let's return to the definition. `factorial($n)` is the number of different orders of a given list of `$n` elements. `factorial(2)` is 2, because there are two ways to order a list of two elements: ('A', 'B') and ('B', 'A'). `factorial(1)` is 1, because there is only one way to order a list of one element: ('A'). `factorial(0)` is 1, because there is only one way to order a list of zero elements: (). Sometimes people are tempted to argue that  $0!$  should be 0, but the example of () shows clearly that it isn't.

Getting the base case right is vitally important in recursive functions, because if you get it wrong, it will throw off all the other results from the function. If we were to erroneously replace `return 1` in the preceding function with `return 0`, it would no longer be a function for computing factorials; instead, it would be a function for computing zero.

### 1.2.1 Why Private Variables Are Important

Let's spend a little while looking at what happens if we leave out the `my`. The following version of `factorial()` is identical to the previous version, except that it is missing the `my` declaration on `$n`:

```
sub factorial {
    ($n) = @_;
    return 1 if $n == 0;
    return factorial($n-1) * $n;
}
```

**CODE LIBRARY**  
factorial-broken

Now `$n` is a global variable, because all Perl variables are global unless they are declared with `my`. This means that even though several copies of `factorial()` might be executing simultaneously, they are all using the same global variable `$n`. What effect does this have on the function's behavior?

Let's consider what happens when we call `factorial(1)`. Initially, `$n` is set to 1, and the test on the second line fails, so the function makes a recursive call to `factorial(0)`. The invocation of `factorial(1)` waits around for the new function call to complete. When `factorial(0)` is entered, `$n` is set to 0. This time the test on the second line is true, and the function returns immediately, yielding 1.

The invocation of `factorial(1)` that was waiting for the answer to `factorial(0)` can now continue; the result from `factorial(0)` is 1. `factorial(1)` takes this 1, multiplies it by the value of `$n`, and returns the result. But `$n` is now 0, because `factorial(0)` set it to 0, so the result is  $1 \cdot 0 = 0$ . This is the final, incorrect return value of `factorial(1)`. It should have been 1, not 0.

Similarly, `factorial(2)` returns 0 instead of 2, `factorial(3)` returns 0 instead of 6, and so on.

In order to work properly, each invocation of `factorial()` needs to have its own private copy of `$n` that the other invocations won't interfere with, and that's exactly what `my` does. Each time `factorial()` is invoked, a new variable is created for that invocation to use as its `$n`.

Other languages that support recursive functions all have variables that work something like Perl's `my` variables, where a new one is created each time the function is invoked. For example, in C, variables declared inside functions have this behavior by default, unless declared otherwise. (In C, such variables are called *auto* variables, because they are automatically allocated and deallocated.) Using global variables or some other kind of storage that isn't allocated for each invocation of a function usually makes it impossible to call that function recursively; such a function is called *non-reentrant*. Non-reentrant functions were once quite common in the days when people used languages like Fortran (which didn't support recursion until 1990) and became less common as languages with private variables, such as C, became popular.

### 1.3 THE TOWER OF HANOI

Both our examples so far have not actually required recursion; they could both be rewritten as simple loops.

This sort of rewriting is always possible, because after all, the machine language in your computer probably doesn't support recursion, so in some sense it must be inessential. For the factorial function, the rewriting is easy, but this isn't always so. Here's an example. It's a puzzle that was first proposed by Edouard Lucas in 1883, called the Tower of Hanoi.

The puzzle has three pegs, called  $\mathcal{A}$ ,  $\mathcal{B}$ , and  $\mathcal{C}$ . On peg  $\mathcal{A}$  is a tower of disks of graduated sizes, with the largest on the bottom and the smallest on the top (see Figure 1.1).

The puzzle is to move the entire tower from  $\mathcal{A}$  to  $\mathcal{C}$ , subject to the following restrictions: you may move only one disk at a time, and no disk may ever rest atop a smaller disk. The number of disks varies depending on who is posing the problem, but it is traditionally 64. We will try to solve the problem in the general case, for  $n$  disks.

Let's consider the largest of the  $n$  disks, which is the one on the bottom. We'll call this disk "the Big Disk." The Big Disk starts on peg  $\mathcal{A}$ , and we want it to end on peg  $\mathcal{C}$ . If any other disks are on peg  $\mathcal{A}$ , they are on top of the Big Disk, so we will not be able to move it. If any other disks are on peg  $\mathcal{C}$ , we will not be able to move the Big Disk to  $\mathcal{C}$  because then it would be atop a smaller disk. So if

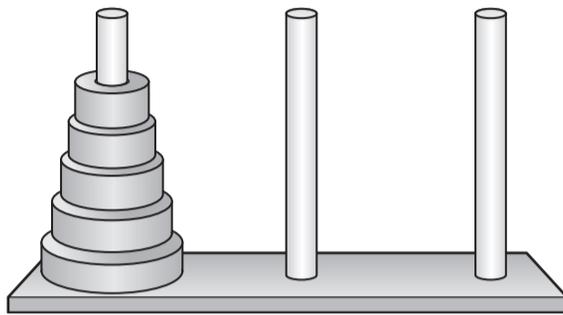


FIGURE 1.1 The initial configuration of the Tower of Hanoi.

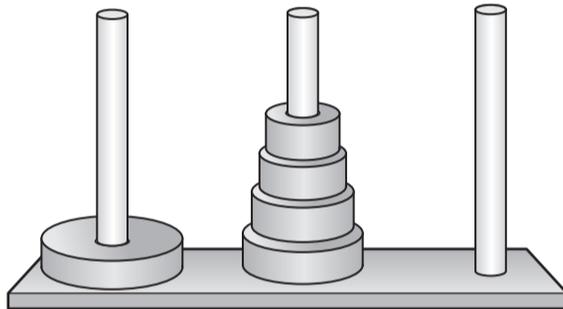


FIGURE 1.2 An intermediate stage of the Tower of Hanoi.

we want to move the Big Disk from  $\mathcal{A}$  to  $\mathcal{C}$ , all the other disks must be heaped up on peg  $\mathcal{B}$ , in size order, with the smallest one on top (see Figure 1.2).

This means that to solve this problem, we have a subgoal: we have to move the entire tower of disks, except for the Big Disk, from  $\mathcal{A}$  to  $\mathcal{B}$ . Only then we can transfer the Big Disk from  $\mathcal{A}$  to  $\mathcal{C}$ . After we've done that, we will be able to move the rest of the tower from  $\mathcal{B}$  to  $\mathcal{C}$ ; this is another subgoal.

Fortunately, when we move the smaller tower, we can ignore the Big Disk; it will never get in our way no matter where it is. This means that we can apply the same logic to moving the smaller tower. At the bottom of the smaller tower is a large disk; we will move the rest of the tower out of the way, move this bottom disk to the right place, and then move the rest of the smaller tower on top of it. How do we move the rest of the smaller tower? The same way.

The process bottoms out when we have to worry about moving a smaller tower that contains only one disk, which will be the smallest disk in the whole set. In that case our subgoals are trivial, and we just put the little disk wherever we need to. We know that there will never be anything on top of it (because that

would be illegal) and we know that we can always move it wherever we like; it's the smallest, so it is impossible to put it atop anything smaller.

Our strategy for moving the original tower looks like this:

To move a tower of  $n$  disks from the start peg to the end peg,

1. If the "tower" is actually only one disk high, just move it. Otherwise:
2. Move all the disks except for disk  $n$  (the Big Disk) from the start peg to the extra peg, using this method.
3. Move disk  $n$  (the Big Disk) from the start peg to the end peg.
4. Move all the other disks from the extra peg to the end peg, using this method.

It's easy to translate this into code:

**CODE LIBRARY**  
hanoi

```
# hanoi(N, start, end, extra)
# Solve Tower of Hanoi problem for a tower of N disks,
# of which the largest is disk #N. Move the entire tower from
# peg 'start' to peg 'end', using peg 'extra' as a work space
sub hanoi {
  my ($n, $start, $end, $extra) = @_;
  if ($n == 1) {
    print "Move disk #1 from $start to $end.\n"; # Step 1
  } else {
    hanoi($n-1, $start, $extra, $end); # Step 2
    print "Move disk #N from $start to $end.\n"; # Step 3
    hanoi($n-1, $extra, $end, $start); # Step 4
  }
}
```

This function prints a series of instructions for how to move the tower. For example, to ask it for instructions for moving a tower of three disks, we call it like this:

```
hanoi(3, 'A', 'C', 'B');
```

Its output is:

```
Move disk #1 from A to C.
Move disk #2 from A to B.
Move disk #1 from C to B.
Move disk #3 from A to C.
```

```

Move disk #1 from B to A.
Move disk #2 from B to C.
Move disk #1 from A to C.

```

If we wanted a graphic display of moving disks instead of a simple printout of instructions, we could replace the `print` statements with something fancier. But we can make the software more flexible almost for free by parametrizing the output behavior. Instead of having a `print` statement hardwired in, `hanoi()` will accept an extra argument that is a function that will be called each time `hanoi()` wants to move a disk. This function will print an instruction, or update a graphical display, or do whatever else we want. The function will be passed the number of the disk, and the source and destination pegs. The code is almost exactly the same:

```

sub hanoi {
  my ($n, $start, $end, $extra, $move_disk) = @_;
  if ($n == 1) {
    $move_disk->(1, $start, $end);
  } else {
    hanoi($n-1, $start, $extra, $end, $move_disk);
    $move_disk->($n, $start, $end);
    hanoi($n-1, $extra, $end, $start, $move_disk);
  }
}

```

To get the behavior of the original version, we now invoke `hanoi()` like this:

```

sub print_instruction {
  my ($disk, $start, $end) = @_;
  print "Move disk #${disk} from $start to $end.\n";
}

hanoi(3, 'A', 'C', 'B', \&print_instruction);

```

The `\&print_instruction` expression generates a *code reference*, which is a scalar value that represents the function. You can store the code reference in a scalar variable just like any other scalar, or pass it as an argument just like any other scalar, and you can also use the reference to invoke the function that it represents. To do that, you write:

```

$code_reference->(arguments...);

```

This invokes the function with the specified arguments.<sup>1</sup> Code references are often referred to as *coderefs*.

The *coderef* argument to `hanoi()` is called a *callback*, because it is a function supplied by the caller of `hanoi()` that will be “called back” to when `hanoi()` needs help. We sometimes also say that the `$move_disk` argument of `hanoi()` is a *hook*, because it provides a place where additional functionality may easily be hung.

Now that we have a generic version of `hanoi()`, we can test the algorithm by passing in a `$move_disk` function that keeps track of where the disks are and checks to make sure we aren’t doing anything illegal:

**CODE LIBRARY**  
check-move

```
@position = ('', ('A') x 3); # Disks are all initially on peg A

sub check_move {
    my $i;
    my ($disk, $start, $end) = @_;
```

The `check_move()` function maintains an array, `@position`, that records the current position of every disk. Initially, every disk is on peg *A*. Here we assume that there are only three disks, so we set `$position[1]`, `$position[2]`, and `$position[3]` to “A”. `$position[0]` is a dummy element that is never used because there is no disk 0. Each time the main `hanoi()` function wants to move a disk, it calls `check_move()`.

```
    if ($disk < 1 || $disk > $#position) {
        die "Bad disk number $disk. Should be 1..$#position.\n";
    }
```

This is a trivial check to make sure that `hanoi()` doesn’t try to move a nonexistent disk.

```
    unless ($position[$disk] eq $start) {
        die "Tried to move disk $disk from $start, but it is on peg
            $position[$disk].\n";
    }
```

<sup>1</sup> This notation was introduced in Perl 5.004; users of 5.003 or earlier will have to use a much uglier notation instead: `&{$code_reference}(arguments...)`. When the `$code_reference` expression is a simple variable, as in the example, the curly braces may be omitted.

Here the function checks to make sure that `hanoi()` is not trying to move a disk from a peg where it does not reside. If the start peg does not match `check_move()`'s notion of the current position of the disk, the function signals an error.

```
for $i (1 .. $disk-1) {
  if ($position[$i] eq $start) {
    die "Can't move disk $disk from $start because $i is on top of it.\n";
  } elsif ($position[$i] eq $end) {
    die "Can't move disk $disk to $end because $i is already there.\n";
  }
}
```

This is the really interesting check. The function loops over all the disks that are smaller than the one `hanoi()` is trying to move, and makes sure that the smaller disks aren't in the way. The first `if` branch makes sure that each smaller disk is not on top of the one `hanoi()` wants to move, and the second branch makes sure that `hanoi()` is not trying to move the current disk onto the smaller disk.

```
print "Moving disk $disk from $start to $end.\n";
$position[$disk] = $end;
}
```

Finally, the function has determined that there is nothing wrong with the move, so it prints out a message as before, and adjusts the `@position` array to reflect the new position of the disk.

Running:

```
hanoi(3, 'A', 'C', 'B', \&check_move);
```

yields the same output as before, and no errors — `hanoi()` is not doing anything illegal.

This example demonstrates a valuable technique we'll see over and over again: by parametrizing some part of a function to call some other function instead of hardwiring the behavior, we can make it more flexible. This added flexibility will pay off when we want the function to do something a little different, such as performing an automatic self-check. Instead of having to clutter up the function with a lot of optional self-testing code, we can separate the testing part from the main algorithm. The algorithm remains as clear and simple as ever, and we can enable or disable the self-checking code at run time if we want to, by passing a different `coderef` argument.

## 1.4 HIERARCHICAL DATA

The examples we've seen give the flavor of what a recursive procedure looks like, but they miss an important point. In introducing the Tower of Hanoi problem, I said that recursion is useful when you want to solve a problem that can be reduced to simpler cases of the same problem. But it might not be clear that such problems are common.

Most recursive functions are built to deal with recursive data structures. A recursive data structure is one like a list, tree, or heap that is defined in terms of simpler instances of the same data structure. The most familiar example is probably a file system directory structure. A file is either:

- a plain file, which contains some data, or
- a directory, which contains a list of files

A file might be a directory, which contains a list of files, some of which might be directories, which in turn contain more lists of files, and so on. The most effective way of dealing with such a structure is with a recursive procedure. Conceptually, each call to such a procedure handles a single file. The file might be a plain file, or it might be a directory, in which case the procedure makes recursive calls to itself to handle any subfiles that the directory has. If the subfiles are themselves directories, the procedure will make more recursive calls.

Here's an example of a function that takes the name of a directory as its argument and computes the total size of all the files contained in it, and in its subdirectories, and in their subdirectories, and so on:

**CODE LIBRARY**  
total-size-broken

```
sub total_size {
    my ($top) = @_;
    my $total = -s $top;
```

When we first call the function, it's with an argument `$top`, which is the name of the file or directory we want to examine. The first thing the function does is use the Perl `-s` operator to find the size of this file or directory itself. This operator yields the size of the file, in bytes. If the file is a directory, it says how much space the directory itself takes up on the disk, apart from whatever files the directory may contain—the directory is a list of files, remember, and the list takes up some space too. If the top file is actually a directory, the function will add the sizes of its contents to a running total that it will keep in `$total`.

```
    return $total if -f $top;
    unless (opendir DIR, $top) {
```

```

warn "Couldn't open directory $top: $!; skipping.\n";
return $total;
}

```

The `-f` operator checks to see if the argument is a plain file; if so, the function can return the total immediately. Otherwise, it assumes that the top file is actually a directory, and tries to open it with `opendir()`. If the directory can't be opened, the function issues a warning message and returns the total so far, which includes the size of the directory itself, but not its contents.

```

my $file;
while ($file = readdir DIR) {
    next if $file eq '.' || $file eq '..';
    $total += total_size("$top/$file");
}

```

The next block, the `while` loop, is the heart of the function. It reads filenames from the directory one at a time, calls itself recursively on each one, and adds the result to the running total.

```

closedir DIR;
return $total;
}

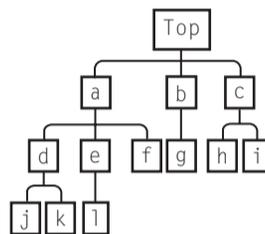
```

At the end of the loop, the function closes the directory and returns the total.

In the loop, the function skips over the names `.` and `..`, which are aliases for the directory itself and for its parent; if it didn't do this, it would never finish, because it would try to compute the total sizes of a lot of files with names like `././././././fred` and `dir/./dir/./dir/./dir/./dir/fred`.

This function has a gigantic bug, and in fact it doesn't work at all. The problem is that directory handles, like `DIR`, are global, and so our function is not reentrant. The function fails for essentially the same reason that the `my-less` version of `factorial()` failed. The first call goes ahead all right, but if `total_size()` calls itself recursively, the second invocation will open the same dirhandle `DIR`. Eventually, the second invocation will reach the end of its directory, close `DIR`, and return. When this happens, the first invocation will try to continue, find that `DIR` has been closed, and exit the `while` loop without having read all the filenames from the top directory. The second invocation will have the same problem if it makes any recursive calls itself.

The result is that the function, as written, looks down only the first branch of the directory tree. If the directory hierarchy has a structure like this:



then our function will go down the *top-a-d* path, see files *j* and *k*, and report the total size of *top + a + d + j + k*, without ever noticing *b, c, e, f, g, h, i, or l*.

To fix it, we need to make the directory handle `DIR` a private variable, like `$top` and `$total`. Instead of `opendir DIR, $top`, we'll use `opendir $dir, $top`, where `$dir` is a private variable. When the first argument to `opendir` is an undefined variable, `opendir` will create a new, anonymous dirhandle and store it into `$dir`.<sup>2</sup>

Instead of doing this:

```
opendir DIR, $somedir;
print (readdir DIR);
closedir DIR;
```

we can get the same effect by doing this instead:

```
my $dir;
opendir $dir, $somedir;
print (readdir $dir);
closedir $dir;
```

The big difference is that `DIR` is a global dirhandle, and can be read or closed by any other part of the program; the dirhandle in `$dir` is private, and can be read

<sup>2</sup> This feature was introduced in Perl 5.6.0. Users of earlier Perl versions will have to use the `IO::Handle` module to explicitly manufacture a dirhandle: `my $dir = IO::Handle->new; opendir $dir, $top; .`

or closed only by the function that creates it, or by some other function that is explicitly passed the value of `$dir`.

With this new technique, we can rewrite the `total_size()` function so that it works properly:

```
sub total_size {
    my ($top) = @_;
    my $total = -s $top;
    my $DIR;

    return $total if -f $top;
    unless (opendir $DIR, $top) {
        warn "Couldn't open directory $top: $!; skipping.\n";
        return $total;
    }

    my $file;
    while ($file = readdir $DIR) {
        next if $file eq '.' || $file eq '..';
        $total += total_size("$top/$file");
    }

    closedir $DIR;
    return $total;
}
```

**CODE LIBRARY**  
total-size

Actually, the `closedir` here is unnecessary, because `dirhandles` created with this method close automatically when the variables that contain them go out of scope. When `total_size()` returns, its private variables are destroyed, including `$DIR`, which contains the last reference to the `dirhandle` object we opened. Perl then destroys the `dirhandle` object, and in the process, closes the `dirhandle`. We will omit the explicit `closedir` in the future.

This function still has some problems: it doesn't handle symbolic links correctly, and if a file has two names in the same directory, it gets counted twice. Also, on Unix systems, the space actually taken up by a file on disk is usually different from the length reported by `-s`, because disk space is allocated in blocks of 1024 bytes at a time. But the function is good enough to be useful, and we might want to apply it to some other tasks as well. If we do decide to fix these problems, we will need to fix them only in this one place, instead of fixing the same problems in fifty slightly different directory-walking functions in fifty different applications.

## 1.5 APPLICATIONS AND VARIATIONS OF DIRECTORY WALKING

Having a function that walks a directory tree is useful, and we might like to use it for all sorts of things. For example, if we want to write a recursive file lister that works like the Unix `ls -R` command, we'll need to walk the directory tree. We might want our function to behave more like the Unix `du` command, which prints out the total size of every subdirectory, as well as the total for all the files it found. We might want our function to search for dangling symbolic links; that is, links that point to nonexistent files. A frequently asked question in the Perl newsgroups and IRC channels is how to walk a directory tree and rename each file or perform some other operation on each file.

We could write many different functions to do these tasks, each one a little different. But the core part of each one is the recursive directory walker, and we'd like to abstract that out so that we can use it as a tool. If we can separate the walker, we can put it in a library, and then anyone who needs a directory walker can use ours.

An important change of stance occurred in the last paragraph. Starting from here, and for most of the rest of the book, we are going to take a point of view that you may not have seen before: we are no longer interested in developing a complete program that we or someone else might use entirely. Instead, we are going to try to write our code so that it is useful to *another programmer* who might want to re-use it in another program. Instead of writing a program, we are now writing a library or module that will be used by other programs.

One direction that we could go from here would be to show how to write a *user interface* for the `total_size()` function, which might prompt the user for a directory name, or read a directory name from the command line or from a graphical widget, and then would display the result somehow. We are not going to do this. It is not hard to add code to prompt the user for a directory name or to read the command-line arguments. For the rest of this book, we are not going to be concerned with user interfaces; instead, we are going to look at *programmer interfaces*. The rest of the book will talk about "the user," but it's not the usual user. Instead, the user is another programmer who wants to use our code when writing their own programs. Instead of asking how we can make our entire program simple and convenient for an end-user, we will look at ways to make our functions and libraries simple and convenient for other programmers to use in their own programs.

There are two good reasons for doing this. One is that if our functions are well designed for easy re-use, we will be able to re-use them ourselves and save time and trouble. Instead of writing similar code over and over, we'll plug a

familiar directory-walking function into every program that needs one. When we improve the directory-walking function in one program, it will be automatically improved in all our other programs as well. Over time, we'll develop a toolkit of useful functions and libraries that will make us more productive, and we'll have more fun programming.

But more importantly, if our functions are well designed for re-use, other programmers will be able to use them, and they will get the same benefits that we do. And being useful to other people is the reason we're here in the first place.<sup>3</sup>

With that change of stance clearly in mind, let's go on. We had written a function, `total_size()`, which contained useful functionality: it walked a directory tree recursively. If we could cleanly separate the directory-walking part of the code from the total-size-computing part, then we might be able to re-use the directory-walking part in many other projects for many other purposes. How can we separate the two functionalities?

As in the Tower of Hanoi program, the key here is to pass an additional parameter to our function. The parameter will itself be a function that tells `total_size()` what we want it to do. The code will look like this:

```
sub dir_walk {
  my ($top, $code) = @_;
  my $DIR;

  $code->($top);

  if (-d $top) {
    my $file;
    unless (opendir $DIR, $top) {
      warn "Couldn't open directory $top: $!; skipping.\n";
      return;
    }
    while ($file = readdir $DIR) {
      next if $file eq '.' || $file eq '..';
      dir_walk("$top/$file", $code);
    }
  }
}
```

**CODE LIBRARY**  
dir-walk-simple

<sup>3</sup> Some people find this unpersuasive, so perhaps I should point out that if we make ourselves useful to other people, they will love and admire us, and they might even pay us more.

This function, which I've renamed `dir_walk()` to honor its new generality, gets two arguments. The first, `$top`, is the name of the file or directory that we want it to start searching in, as before. The second, `$code`, is new. It's a coderef that tells `dir_walk` what we want to do for each file or directory that we discover in the file tree. Each time `dir_walk()` discovers a new file or directory, it will invoke our code with the filename as the argument.

Now whenever we meet another programmer who asks us, "How do I do *X* for every file in a directory tree?" we can answer, "Use this `dir_walk()` function, and give it a reference to a function that does *X*." The `$code` argument is a callback.

For example, to get a program that prints out a list of all the files and directories below the current directory, we can use:

```
sub print_dir {
    print $_[0], "\n";
}

dir_walk('.', \&print_dir );
```

This prints out something like this:

```
.
./a
./a/a1
./a/a2
./b
./b/b1
./c
./c/c1
./c/c2
./c/c3
./c/d
./c/d/d1
./c/d/d2
```

(The current directory contains three subdirectories, named `a`, `b`, and `c`. Subdirectory `c` contains a sub-subdirectory, named `d`.)

`print_dir` is so simple that it's a shame to have to waste time thinking of a name for it. It would be convenient if we could simply write the function without having to write a name for it, analogous to the way we can write:

```
$weekly_pay = 40 * $hourly_pay;
```

without having to name the `40` or store it in a variable. Perl does provide a syntax for this:

```
dir_walk('.', sub { print $_[0], "\n" } );
```

The `sub { ... }` introduces an *anonymous function*; that is, a function with no name. The value of the `sub { ... }` construction is a coderef that can be used to call the function. We can store this coderef in a scalar variable or pass it as an argument to a function like any other reference. This one line does the same thing as our more verbose version with the named `print_dir` function.

If we want the function to print out sizes along with filenames, we need only make a small change to the coderef argument:

```
dir_walk('.', sub { printf "%6d %s\n", -s $_[0], $_[0] } );
```

```
4096 .
4096 ./a
 261 ./a/a1
 171 ./a/a2
4096 ./b
 348 ./b/b1
4096 ./c
 658 ./c/c1
 479 ./c/c2
 889 ./c/c3
4096 ./c/d
 568 ./c/d/d1
 889 ./c/d/d2
```

If we want the function to locate dangling symbolic links, it's just as easy:

```
dir_walk('.', sub { print $_[0], "\n" if -l $_[0] && ! -e $_[0] });
```

`-l` tests the current file to see if it's a symbolic link, and `-e` tests to see if the file that the link points at exists.

But my promises fall a little short. There's no simple way to get the new `dir_walk()` function to aggregate the sizes of all the files it sees. `$code` is invoked for only one file at a time, so it never gets a chance to aggregate. If the aggregation is sufficiently simple, we can accomplish it with a variable defined outside the callback:

```
my $TOTAL = 0;
```

```
dir_walk('.', sub { $TOTAL += -s $_[0] });
print "Total size is $TOTAL.\n";
```

There are two drawbacks to this approach. One is that the callback function must reside in the scope of the `$TOTAL` variable, as must any code that plans to use `$TOTAL`. Often this isn't a problem, as in this case, but if the callback were a complicated function in a library somewhere, it might present difficulties. We'll see a solution to this problem in Section 2.1.

The other drawback is that it works well only when the aggregation is extremely simple, as it is here. Suppose instead of accumulating a single total size, we wanted to build a hash structure of filenames and sizes, like this one:

```
{
  'a' => {
    'a1' => '261',
    'a2' => '171'
  },
  'b' => {
    'b1' => '348'
  },
  'c' => {
    'c1' => '658',
    'c2' => '479',
    'c3' => '889',
    'd' => {
      'd1' => '568',
      'd2' => '889'
    }
  }
}
```

Here the keys are file and directory names. The value for a filename is the size of the file, and the value for a directory name is a hash with keys and values that represent the contents of the directory. It may not be clear how we could adapt the simple `$TOTAL`-aggregating callback to produce a complex structure like this one.

Our `dir_walk` function is not general enough. We need it to perform some computation involving the files it examines, such as computing their total size, and to return the result of this computation to its caller. The caller might be the main program, or it might be another invocation of `dir_walk()`, which can then use the value it receives as part of the computation it is performing for *its* caller.

How can `dir_walk()` know how to perform the computation? In `total_size()`, the addition computation was hardwired into the function. We would like `dir_walk()` to be more generally useful.

What we need is to supply two functions: one for plain files and one for directories. `dir_walk()` will call the plain-file function when it needs to compute its result for a plain file, and it will call the directory function when it needs to compute its result for a directory. `dir_walk()` won't know anything about how to do these computations itself; all it knows is that it should delegate the actual computing to these two functions.

Each of the two functions will get a filename argument, and will compute the value of interest, such as the size, for the file named by its argument. Since a directory is a list of files, the directory function will also receive a list of the values that were computed for each of its members; it may need these values when it computes the value for the entire directory. The directory function will know how to aggregate these values to produce a new value for the entire directory.

With this change, we'll be able to do our `total_size` operation. The plain-file function will simply return the size of the file it's asked to look at. The directory function will get a directory name and a list of the sizes of each file that it contains, add them all up, and return the result. The generic framework function looks like this:

```
sub dir_walk {
  my ($stop, $filefunc, $dirfunc) = @_;
  my $DIR;

  if (-d $stop) {
    my $file;
    unless (opendir $DIR, $stop) {
      warn "Couldn't open directory $code: $!; skipping.\n";
      return;
    }

    my @results;
    while ($file = readdir $DIR) {
      next if $file eq '.' || $file eq '..';
      push @results, dir_walk("$stop/$file", $filefunc, $dirfunc);
    }
    return $dirfunc->($stop, @results);
  } else {
    return $filefunc->($stop);
  }
}
```

**CODE LIBRARY**  
dir-walk-cb

To compute the total size of the current directory, we will use this:

```

sub file_size { -s $_[0] }

sub dir_size {
    my $dir = shift;
    my $total = -s $dir;
    my $n;
    for $n (@_) { $total += $n }
    return $total;
}

$total_size = dir_walk('.', \&file_size, \&dir_size);

```

The `file_size()` function says how to compute the size of a plain file, given its name, and the `dir_size()` function says how to compute the size of a directory, given the directory name and the sizes of its contents.

If we want the program to print out the size of every subdirectory, the way the `du` command does, we add one line:

```

sub file_size { -s $_[0] }

sub dir_size {
    my $dir = shift;
    my $total = -s $dir;
    my $n;
    for $n (@_) { $total += $n }
    printf "%6d %s\n", $total, $dir;
    return $total;
}

$total_size = dir_walk('.', \&file_size, \&dir_size);

```

This produces an output like this:

```

4528 ./a
4444 ./b
5553 ./c/d
11675 ./c
24743 .

```

To get the function to produce the hash structure we saw earlier, we can supply the following pair of callbacks:

```
sub file {
    my $file = shift;
    [short($file), -s $file];
}

sub short {
    my $path = shift;
    $path =~ s{.*/}{};
    $path;
}
```

**CODE LIBRARY**  
dir-walk-sizehash

The file callback returns an array with the abbreviated name of the file (no full path) and the file size. The aggregation is, as before, performed in the directory callback:

```
sub dir {
    my ($dir, @subdirs) = @_;
    my %new_hash;
    for (@subdirs) {
        my ($subdir_name, $subdir_structure) = @$_;
        $new_hash{$subdir_name} = $subdir_structure;
    }
    return [short($dir), \%new_hash];
}
```

The directory callback gets the name of the current directory, and a list of name–value pairs that correspond to the subfiles and subdirectories. It merges these pairs into a hash, and returns a new pair with the short name of the current directory and the newly constructed hash for the current directory.

The simpler functions that we wrote before are still easy. Here’s the recursive file lister. We use the same function for files and for directories:

```
sub print_filename { print $_[0], "\n" }
dir_walk('.', \&print_filename, \&print_filename);
```

Here’s the dangling symbolic link detector:

```
sub dangles {
    my $file = shift;
```

```

    print "$file\n" if -l $file && ! -e $file;
}
dir_walk('.', \&dangles, sub {});

```

We know that a directory can't possibly be a dangling symbolic link, so our directory function is the *null function* that returns immediately without doing anything. If we had wanted, we could have avoided this oddity, and its associated function-call overhead, as follows:

**CODE LIBRARY**  
dir-walk-cb-def

```

sub dir_walk {
    my ($top, $filefunc, $dirfunc) = @_;
    my $DIR;
    if (-d $top) {
        my $file;
        unless (opendir $DIR, $top) {
            warn "Couldn't open directory $top: $!; skipping.\n";
            return;
        }

        my @results;
        while ($file = readdir $DIR) {
            next if $file eq '.' || $file eq '..';
            push@results, dir_walk("$top/$file", $filefunc, $dirfunc);
        }
        return $dirfunc ? $dirfunc->($top, @results) : ();
    } else {
        return $filefunc ? $filefunc->($top): ();
    }
}

```

This allows us to write `dir_walk('.', \&dangles)` instead of `dir_walk('.', \&dangles, sub {})`.

As a final example, let's use `dir_walk()` in a slightly different way, to manufacture a list of all the plain files in a file tree, without printing anything:

```

@all_plain_files =
    dir_walk('.', sub { $_[0] }, sub { shift; return @_ });

```

The file function returns the name of the file it's invoked on. The directory function throws away the directory name and returns the list of the files it contains. What if a directory contains no files at all? Then it returns an empty list to

`dir_walk()`, and this empty list will be merged into the result list for the other directories at the same level.

## 1.6 FUNCTIONAL VERSUS OBJECT-ORIENTED PROGRAMMING

Now let's back up a moment and look at what we did. We had a useful function, `total_size()`, which contained code for walking a directory structure that was going to be useful in other applications. So we made `total_size()` more general by pulling out all the parts that related to the computation of sizes, and replacing them with calls to arbitrary user-specified functions. The result was `dir_walk()`. Now, for any program that needs to walk a directory structure and do something, `dir_walk()` handles the walking part, and the argument functions handle the “do something” part. By passing the appropriate pair of functions to `dir_walk()`, we can make it do whatever we want it to. We've gained flexibility and the chance to re-use the `dir_walk()` code by factoring out the useful part and parametrizing it with two functional arguments. This is the heart of the functional style of programming.

Object-oriented (OO) programming style gets a lot more press these days. The goals of the OO style are the same as those of the functional style: we want to increase the re-usability of software components by separating them into generally useful parts.

In an OO system, we could have transformed `total_size()` analogously, but the result would have looked different. We would have made `total_size()` into an abstract base class of directory-walking objects, and these objects would have had a method, `dir_walk()`, which in turn would make calls to two undefined virtual methods called `file` and `directory`. (In C++ jargon, these are called *pure virtual methods*.) Such a class wouldn't have been useful by itself, because the `file` and `directory` methods would be missing. To use the class, you would create a subclass that defined the `file` and `directory` methods, and then create objects in the subclass. These objects would all inherit the same `dir_walk` method.

In this case, I think the functional style leads to a lighter-weight solution that is easier to use, and that keeps the parameter functions close to the places they are used instead of stuck off in a class file. But the important point is that although the styles are different, the decomposition of the original function into useful components has exactly the same structure. Where the functional style uses functional arguments, the object-oriented style uses pure virtual methods. Although the rest of this book is about the functional style of programming, many

of the techniques will be directly applicable to object-oriented programming styles also.

## 1.7 HTML

I promised that recursion was useful for operating on hierarchically defined data structures, and I used the file system as an example. But it's a slightly peculiar example of a data structure, since we normally think of data structures as being in memory, not on the disk.

What gave rise to the tree structure in the file system was the presence of directories, each of which contains a list of other files. Any domain that has items that include lists of other items will contain tree structures. An excellent example is HTML data.

HTML data is a sequence of elements and plain text. Each element has some content, which is a sequence of more elements and more plain text. This is a recursive description, analogous to the description of the file system, and the structure of an HTML document is analogous to the structure of the file system.

Elements are tagged with a *start tag*, which looks like this:

```
<font>
```

and a corresponding *end tag*, like this:

```
</font>
```

The start tag may have a set of *attribute–value pairs*, in which case it might look something like this instead:

```
<font size=3 color="red">
```

The end tag is the same in any case. It never has any attribute–value pairs.

In between the start and end tags can be any sequence of HTML text, including more elements, and also plain text. Here's a simple example of an HTML document:

```
<h1>What Junior Said Next</h1>

<p>But I don't <font size=3 color="red">want</font>
to go to bed now!</p>
```

This document's structure is shown in Figure 1.3.

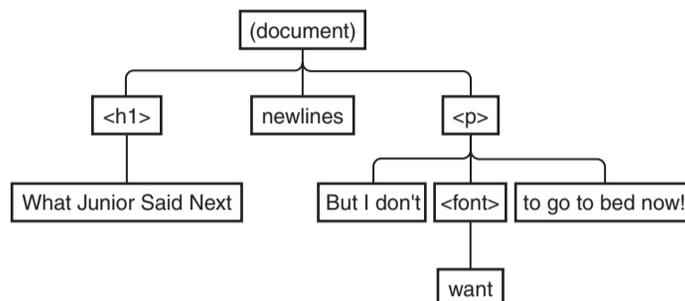


FIGURE 1.3 An HTML document.

The main document has three components: the `<h1>` element, with its contents; the `<p>` element, with its contents; and the blank space in between. The `<p>` element, in turn, has three components: the untagged text before the `<font>` element; the `<font>` element, with its contents; and the untagged text after the `<font>` element. The `<h1>` element has one component, which is the untagged text `What Junior Said Next`.

In Chapter 8, we'll see how to build parsers for languages like HTML. In the meantime, we'll look at a semi-standard module, `HTML::TreeBuilder`, which converts an HTML document into a tree structure.

Let's suppose that the HTML data is already in a variable, say `$html`. The following code uses `HTML::TreeBuilder` to transform the text into an explicit tree structure:

```

use HTML::TreeBuilder;
my $tree = HTML::TreeBuilder->new;
$tree->ignore_ignorable_whitespace(0);
$tree->parse($html);
$tree->eof();

```

The `ignore_ignorable_whitespace()` method tells `HTML::TreeBuilder` that it's not allowed to discard certain whitespace, such as the newlines after the `<h1>` element, that are normally ignorable.

Now `$tree` represents the tree structure. It's a tree of hashes; each hash is a node in the tree and represents one element. Each hash has a `_tag` key whose value is its tag name, and a `_content` key whose value is a list of the element's contents, in order; each item in the `_content` list is either a string, representing tagless text, or another hash, representing another element. If the tag also has attribute-value pairs, they're stored in the hash directly, with attributes as hash keys and the corresponding values as hash values.

So for example, the tree node that corresponds to the `<font>` element in the example looks like this:

```
{ _tag => "font",
  _content => [ "want" ],
  color => "red",
  size => 3,
}
```

The tree node that corresponds to the `<p>` element contains the `<font>` node, and looks like this:

```
{ _tag => "p",
  _content => [ "But I don't ",
               { _tag => "font",
                 _content => [ "want" ],
                 color => "red",
                 size => 3,
               },
               " to go to bed now!",
             ],
}
```

It's not hard to build a function that walks one of these HTML trees and “untags” all the text, stripping out the tags. For each item in a `_content` list, we can recognize it as an element with the `ref()` function, which will yield true for elements (which are hash references) and false for plain strings:

**CODE LIBRARY**  
untag-html

```
sub untag_html {
  my ($html) = @_;
  return $html unless ref $html; # It's a plain string

  my $text = '';
  for my $item (@{$html->{_content}}) {
    $text .= untag_html($item);
  }

  return $text;
}
```

The function checks to see if the HTML item passed in is a plain string, and if so the function returns it immediately. If it's not a plain string, the function

assumes that it is a tree node, as described above, and iterates over its content, recursively converting each item to plain text, accumulating the resulting strings, and returning the result. For our example, this is:

```
What Junior Said Next But I don't want to go to bed now!
```

Sean Burke, the author of `HTML::TreeBuilder`, tells me that accessing the internals of the `HTML::TreeBuilder` objects this way is naughty, because he might change them in the future. Robust programs should use the accessor methods that the module provides. In these examples, we will continue to access the internals directly.

We can learn from `dir_walk()` and make this function more useful by separating it into two parts: the part that processes an HTML tree, and the part that deals with the specific task of assembling plain text:

```
sub walk_html {
    my ($html, $textfunc, $elementfunc) = @_;
    return $textfunc->($html) unless ref $html; # It's a plain string

    my @results;
    for my $item (@{$html->{_content}}) {
        push @results, walk_html($item, $textfunc, $elementfunc);
    }
    return $elementfunc->($html, @results);
}
```

**CODE LIBRARY**  
walk-html

This function has exactly the same structure as `dir_walk()`. It gets two auxiliary functions as arguments: a `$textfunc` that computes some value of interest for a plain text string, and an `$elementfunc` that computes the corresponding value for an element, given the element and the values for the items in its content. `$textfunc` is analogous to the `$filefunc` from `dir_walk()`, and `$elementfunc` is analogous to the `$dirfunc`.

Now we can write our `untagger` like this:

```
walk_html($tree, sub { $_[0] },
          sub { shift; join '', @_ });
```

The `$textfunc` argument is a function that returns its argument unchanged. The `$elementfunc` argument is a function that throws away the element itself, then concatenates the texts that were computed for its contents, and returns the concatenation. The output is identical to that of `untag_html()`.

Suppose we want a document summarizer that prints out the text that is inside of `<h1>` tags and throws away everything else:

```
sub print_if_h1tag {
    my $element = shift;
    my $text = join ' ', @_;
    print $text if $element->{_tag} eq 'h1';
    return $text;
}
walk_html($tree, sub { $_[0] }, \&print_if_h1tag);
```

This is essentially the same as `untag_html()`, except that when the element function sees that it is processing an `<h1>` element, it prints out the untagged text.

If we want the function to *return* the header text instead of printing it out, we have to get a little trickier. Consider an example like this:

```
<h1>Junior</h1>
Is a naughty boy.
```

We would like to throw away the text `Is a naughty boy`, so that it doesn't appear in the result. But to `walk_html()`, it is just another plain text item, which looks exactly the same as `Junior`, which we *don't* want to throw away. It might seem that we should simply throw away everything that appears inside a non-header tag, but that doesn't work:

```
<h1>The story of <b>Junior</b></h1>
```

We mustn't throw away `Junior` here, just because he's inside a `<b>` tag, because that `<b>` tag is itself inside an `<h1>` tag, and we want to keep it.

We could solve this problem by passing information about the current tag context from each invocation of `walk_html()` to the next, but it turns out to be simpler to pass information back the other way. Each text in the file is either a “keeper,” because we know it's inside an `<h1>` element, or a “maybe,” because we don't. Whenever we process an `<h1>` element, we'll promote all the “maybes” that it contains to “keepers.” At the end, we'll print the keepers and throw away the maybes:

**CODE LIBRARY**  
extract-headers

```
@tagged_texts = walk_html($tree, sub { ['MAYBE', $_[0]] },
                                \&promote_if_h1tag);

sub promote_if_h1tag {
```

```

my $element = shift;
if ($element->{_tag} eq 'h1') {
    return ['KEEPER', join '', map {$_->[1]} @_];
} else {
    return @_;
}
}

```

The return value from `walk_html()` will be a list of labeled text items. Each text item is an anonymous array whose first element is either `MAYBE` or `KEEPER`, and whose second item is a string. The plain text function simply labels its argument as a `MAYBE`. For the string `Junior`, it returns the labeled item `['MAYBE', 'Junior']`; for the string `Is a naughty boy.`, it returns `['MAYBE', 'Is a naughty boy.']`.

The element function is more interesting. It gets an element and a list of labeled text items. If the element represents an `<h1>` tag, the function extracts all the texts from its other arguments, joins them together, and labels the result as a `KEEPER`. If the element is some other kind, the function returns its tagged texts unchanged. These texts will be inserted into the list of labeled texts that are passed to the element function call for the element that is one level up; compare this with the final example of `dir_walk()` in Section 1.5, which returned a list of filenames in a similar way.

Since the final return value from `walk_html()` is a list of labeled texts, we need to filter them and throw away the ones that are still marked `MAYBE`. This final pass is unavoidable. Since the function treats an untagged text item differently at the top level than it does when it is embedded inside an `<h1>` tag, there must be some part of the process that understands when something is at the top level. `walk_html()` can't do that because it does the same thing at every level. So we must build one final function to handle the top-level processing:

```

sub extract_headers {
    my $tree = shift;
    my @tagged_texts = walk_html($tree, sub { ['MAYBE', $_[0]] },
                                       \&promote_if_h1tag);
    my @keepers = grep { $_->[0] eq 'KEEPER' } @tagged_texts;
    my @keeper_text = map { $_->[1] } @keepers;
    my $header_text = join '', @keeper_text;
    return $header_text;
}

```

Or we could write it more compactly:

```

sub extract_headers {
    my $tree = shift;

```



we can use this:

```
my @tagged_texts = walk_html($tree,
    sub { ['maybe', $_[0]] },
    sub { promote_if(
        sub { $_[0] eq 'h1'},
        $_[0])
    });
```

We'll see a tidier way to do this in Chapter 7.

## 1.8 WHEN RECURSION BLOWS UP

Sometimes a problem appears to be naturally recursive, and then the recursive solution is grossly inefficient. A very simple example arises when you want to compute Fibonacci numbers. This is a rather unrealistic example, but it has the benefit of being very simple. We'll see a more practical example of the same thing in Section 3.7.

### 1.8.1 Fibonacci Numbers

*Fibonacci numbers* are named for Leonardo of Pisa, whose nickname was Fibonacci, who discussed them in the 13th century in connection with a mathematical problem about rabbits. Initially, you have one pair of baby rabbits. Baby rabbits grow to adults in one month, and the following month they produce a new pair of baby rabbits, making two pairs:

Month	Pairs of baby rabbits	Pairs of adult rabbits	Total pairs
1	1	0	1
2	0	1	1
3	1	1	2

The following month, the baby rabbits grow up and the adults produce a new pair of babies:

4	1	2	3
---	---	---	---

The month after that, the babies grow up, and the two pairs of adults each produce a new pair of babies:

5	2	3	5
---	---	---	---

Assuming no rabbits die, and rabbit production continues, how many pairs of rabbits are there in each month?

Let  $A(n)$  be the number of pairs of adults alive in month  $n$  and  $B(n)$  be the number of pairs of babies alive in month  $n$ . The total number of pairs of rabbits alive in month  $n$ , which we'll call  $T(n)$ , is therefore  $A(n) + B(n)$ :

$$T(n) = A(n) + B(n)$$

It's not hard to see that the number of baby rabbits in one month is equal to the number of adult rabbits the previous month, because each pair of adults gives birth to one pair of babies. In symbols, this is  $B(n) = A(n - 1)$ . Substituting into our formula, we have:

$$T(n) = A(n) + A(n - 1)$$

Each month the number of adult rabbits is equal to the total number of rabbits from the previous month, because the babies from the previous month grow up and the adults from the previous month are still alive. In symbols, this is  $A(n) = T(n - 1)$ . Substituting into the previous equation, we get:

$$T(n) = T(n - 1) + T(n - 2)$$

So the total number of rabbits in month  $n$  is the sum of the number of rabbits in months  $n - 1$  and  $n - 2$ . Armed with this formula, we can write down the function to compute the Fibonacci numbers:

**CODE LIBRARY**  
fib

```
# Compute the number of pairs of rabbits alive in month n
sub fib {
  my ($month) = @_;
  if ($month < 2) { 1 }
  else {
    fib($month-1) + fib($month-2);
  }
}
```

This is perfectly straightforward, but it has a problem: except for small arguments, it takes forever.<sup>4</sup> If you ask for `fib(25)`, for example, it needs to make recursive calls to compute `fib(24)` and `fib(23)`. But the call to `fib(24)` *also* makes a recursive call to `fib(23)`, as well as another to compute `fib(22)`. Both calls to `fib(23)` will *also* call `fib(22)`, for a total of three times. It turns out that `fib(21)` is computed 5 times, `fib(20)` is computed 8 times, and `fib(19)` is computed 13 times.

All this computing and recomputing has a heavy price. On my small computer, it takes about four seconds to compute `fib(25)`; it makes 242,785 recursive calls while doing so. It takes about 6.5 seconds to compute `fib(26)`, and makes 392,835 recursive calls, and about 10.5 seconds to make the 635,621 recursive calls for `fib(27)`. It takes as long to compute `fib(27)` as to compute `fib(25)` and `fib(26)` put together, and so the running time of the function increases rapidly, more than doubling every time the argument increases by 2.<sup>5</sup>

The running time blows up really fast, and it's all caused by our repeated computation of things that we already computed. Recursive functions occasionally have this problem, but there's an easy solution for it, which we'll see in Chapter 3.

### 1.8.2 Partitioning

Fibonacci numbers are rather abstruse, and it's hard to find simple realistic examples of programs that need to compute them.

Here's a somewhat more realistic example. We have some valuable items, which we'll call "treasures," and we want to divide them evenly between two people. We know the value of each item, and we would like to ensure that both people get collections of items whose total value is the same. Or, to recast the problem in a more mundane light: we know the weight of each of the various groceries you bought today, and since you're going to carry them home with one bag in each hand, you want to distribute the weight evenly.

To convince yourself that this can be a tricky problem, try dividing up a set of ten items that have these dollar values:

\$9, \$12, \$14, \$17, \$23, \$32, \$34, \$40, \$42, and \$49

<sup>4</sup> One of the technical reviewers objected that this was an exaggeration, and it is. But I estimate that calculating `fib(100)` by this method would take about 2,241,937 billion billion years, which is close enough.

<sup>5</sup> In fact, each increase of 2 in the argument increases the running time by a factor of about 2.62.

Since the total value of the items is \$272, each person will have to receive items totalling \$136. Then try:

\$9, \$12, \$14, \$17, \$23, \$32, \$34, \$40, \$38, and \$49

Here I replaced the \$42 item with a \$38 item, so each person will have to receive items totalling \$134.

This problem is called the *partition problem*. We'll generalize the problem a little: instead of trying to divide a list of treasures into two equal parts, we'll try to find some share of the treasures whose total value is a given target amount. Finding an even division of the treasures is the same as finding a share whose value is half of the total value of all the treasures; then the other share is the rest of the treasures, whose total value is the same.

If there is no share of treasures that totals the target amount, our function will return undef:

**CODE LIBRARY**  
find-share

```
sub find_share {
    my ($target, $treasures) = @_;
    return [] if $target == 0;
    return   if $target < 0 || @$treasures == 0;
```

We take care of some trivial cases first. If the target amount is exactly zero, then it's easy to produce a list of treasures that total the target amount: the empty list is sure to have value zero, so we return that right away.

If the target amount is less than zero, we can't possibly hit it, because treasures are assumed to have positive value. In this case no solution can be found and the function can immediately return failure. If there are no treasures, we know we can't make the target, since we already know the target is larger than zero; we fail immediately.

Otherwise, the target amount is positive, and we will have to do some real work:

```
    my ($first, @rest) = @$treasures;
    my $solution = find_share($target-$first, \@rest);
    return [$first, @$solution] if $solution;
    return      find_share($target      , \@rest);
}
```

Here we copy the list of treasures, and then remove the first treasure from the list. This is because we're going to consider the simpler problem of how to divide up the treasures without the first treasure. There are two possible divisions: either this first treasure is in the share we're computing, or it isn't. If it is, then we

have to find a subset of the rest of the treasures whose total value is `$target - $first`. If it isn't, then we have to find a subset of the rest of the treasures whose total value is `$target`. The rest of the code makes recursive calls to `find_share` to investigate these two cases. If the first one works out, the function returns a solution that includes the first treasure; if the second one works out, it returns a solution that omits the first treasure; if neither works out, it returns `undef`.

Here's a trace of a sample run. We'll call `find_share(5, [1, 2, 4, 8])`:

Share so far	Total so far	Target	Remaining treasures
	0	5	1 2 4 8

None of the trivial cases apply—the target is neither negative nor zero, and the remaining treasure list is not empty—so the function tries allocating the first item, 1, to the share; it then looks for some set of the remaining items that can be made to add up to 4:

1	1	4	2 4 8
---	---	---	-------

The function will continue investigating this situation until it is forced to give up.

The function then allocates the first remaining item, 2, toward the share of 4, and makes a recursive call to find some set of the last 2 elements that add up to 2:

1 2	3	2	4 8
-----	---	---	-----

Let's call this "situation *a*." The function will continue investigating this situation until it concludes that situation *a* is hopeless. It tries allocating the 4 to the share, but that overshoots the target total:

1 2 4	7	-2	8
-------	---	----	---

so it backs up and tries continuing from situation *a* *without* allocating the 4 to the share:

1 2	3	2	8
-----	---	---	---

The share is still wanting, so the function allocates the next item, 8, to the share, which obviously overshoots:

1 2 8	11	-6	
-------	----	----	--

Here we have `$target < 0`, so the function fails, and tries omitting 8 instead. This doesn't work either, as it leaves the share short by 2 of the target, with no

items left to allocate:

Share so far	Total so far	Target	Remaining treasures
1 2	3	2	

This is the `if (@$treasures == 0) { return undef }` case.

The function has tried every possible way of making situation *a* work; they all failed. It concludes that allocating both 1 and 2 to the share doesn't work, and backs up and tries omitting 2 instead:

1	1	4	4 8
---	---	---	-----

It now tries allocating 4 to the share:

1 4	5	0	8
-----	---	---	---

Now the function has `$target == 0`, so it returns success. The allocated treasures are 1 and 4, which add up to the target 5.

The idea of ignoring the first treasure and looking for a solution among the remaining treasures, thus reducing the problem to a simpler case, is natural. A solution without recursion would probably end up duplicating the underlying machinery of the recursive solution, and simulating the behavior of the function-call stack manually.

Now solving the partition problem is easy; it's a call to `find_share()`, which finds the first share, and then some extra work to compute the elements of the original array that are not included in the first share:

**CODE LIBRARY**  
partition

```
sub partition {
    my $total = 0;
    my $share_2;
    for my $treasure (@_) {
        $total += $treasure;
    }

    my $share_1 = find_share($total/2, @_);
    return unless defined $share_1;
}
```

First the function computes the total value of all the treasures. Then it asks `find_share()` to compute a subset of the original treasures whose total value is exactly half. If `find_share()` returns an undefined value, there was no equal division, so `partition()` returns failure immediately. Otherwise, it will set about

computing the list of treasures that are *not* in `$share_1`, and this will be the second share:

```

my %in_share_1;
for my $treasure (@$share_1) {
    ++$in_share_1{$treasure};
}

for my $treasure (@_) {
    if ($in_share_1{$treasure}) {
        --$in_share_1{$treasure};
    } else {
        push @$share_2, $treasure;
    }
}

```

The function uses a hash to count up the number of occurrences of each value in the first share, and then looks at the original list of treasures one at a time. If it saw that a treasure was in the first share, it checks it off; otherwise, it put the treasure into the list of treasures that make up share 2.

```

    return ($share_1, $share_2);
}

```

When it's done, it returns the two lists of treasures.

There's a lot of code here, but it mostly has to do with splitting up a list of numbers. The key line is the call to `find_share()`, which actually computes the solution; this is `$share_1`. The rest of the code is all about producing a list of treasures that *aren't* in `$share_1`; this is `$share_2`.

The `find_share` function, however, has a problem: it takes much too long to run, especially if there is no solution. It has essentially the same problem as `fib` did: it repeats the same work over and over. For example, suppose it is trying to find a division of 1 2 3 4 5 6 7 with target sum 14. It might be investigating shares that contain 1 and 3, and then look to see if it can make 5 6 7 hit the target sum of 10. It can't, so it will look for other solutions. Later on, it might investigate shares that contain 4, and again look to see if it can make 5 6 7 hit the target sum of 10. This is a waste of time; `find_share` should remember that 5 6 7 cannot hit a target sum of 10 from the first time it investigated that.

We will see in Chapter 3 how to fix this.

— |

| —

— |

| —