CHAPTER 4

ITERATORS

4.1 INTRODUCTION

An *iterator* is an object interface to a list.

The object's member data consists of the list and some state information marking a "current position" in the list. The iterator supports one method, which we will call NEXTVAL. The NEXTVAL method returns the list element at the current position and updates the current position so that the next time NEXTVAL is called, the next list element will be returned.

Why would anyone want an object interface to a list? Why not just use a list? There are several reasons. The simplest is that the list might be enormous, so large that you do not want to have it in memory all at once. It is often possible to design iterators to generate list items as they're requested, so that only a small part of the list need ever be in memory at once.

4.1.1 Filehandles Are Iterators

Iterators are familiar to everyone who has ever programmed in Perl, because filehandles are iterators. When you open a file for reading, you get back a filehandle object:

```
open(FILEHANDLE, 'filename');
```

We'll look at filehandles first because they are a familiar example that exhibit all the advantages of iterators. A filehandle does represent a list, namely the list of lines from a file. The NEXTVAL operation is written in Perl as <FILEHANDLE>.

When you do <FILEHANDLE>, Perl returns the line at the current position and updates the current position so that the next time you do <FILEHANDLE> you get the next line.

Imagine an alternate universe in which the Perl open function yielded not a filehandle but instead, a list of lines:

```
@lines = open('filename');    # alternate universe interface
```

Almost any programmer asked to criticize this interface will complain that this would consume too much memory if the file is very large. One of the principal reasons for filehandles is that files can be so large and need to be represented in programs in some way other than as a possibly enormous list of lines.

Another problem with the imaginary iterator-less version is the following common pattern:

```
open(FILEHANDLE, 'filename');
while (<FILEHANDLE>) {
  last if /Plutonium/;
}
close FILEHANDLE;
# do something with $_;
```

This code opens a file and reads through it looking for a record that contains the word "Plutonium". When it finds the record, it exits the loop immediately, closes the file, and then does something with the record it just extracted. On average, it has to search only half of the file, because the plutonium will typically be somewhere near the middle; it might even get lucky and find it right at the beginning. In the worst case, the plutonium is at the end of the file, or is missing entirely, and the program has to read the whole file to discover that.

In the imaginary alternate universe with no filehandles, we get the worst case every time:

```
# alternate universe interface
@lines = open('filename');
for (@lines) {
  last if /Plutonium/;
}
# do something with $_;
```

Even if the plutonium is at the beginning of the file, the alternate universe `open()` still reads the entire file, a colossal waste of I/O and processor time.

Unix programmers, remembering that Perl's open function can also open a pipe to a command, will object even more strenuously:

```
@lines = open("yes |");      # alternate universe interface
```

Here Perl runs the Unix `yes` command and reads its output. But there's a terrible problem: the output of `yes` is infinite. The program will hang in an infinite loop at this line until all memory is exhausted, and then it will drop dead. The filehandle version works just fine.

4.1.2 Iterators Are Objects

The final advantage of an iterator over a plain array is that an iterator is an object, which means it can be shared among functions.

Consider a program that opens and reads a Windows `INI` file. Here's an example of an `INI` file:

```
[Display]
model=Samsui

[Capabilities]
supports_3D=y
power_save=n
```

The file is divided into sections, each of which is headed by a title like `[Display]` or `[Capabilities]`. Within each section are variable definitions such as `model=Samsui`. `model` is the name of a configuration variable and `Samsui` is its value.

A function to parse a single section of an `INI` file might look something like this:

```
sub parse_section {
  my $fh = shift;
  my $title = parse_section_title($fh);
  my %variables = parse_variables($fh);
  return [$title, \%variables];
}
```

The function gets a filehandle as its only argument. `parse_section()` passes the filehandle to the `parse_section_title()` function, which reads the first line and extracts and returns the title; then `parse_section()` passes the same filehandle to `parse_variables()`, which reads the rest of the section and returns a hash with the variable definitions. Unlike an array of lines, `$fh` keeps track of the current position in the `INI` file, so that `parse_section_title()` and `parse_variables()` don't read the same data. Instead, `parse_variables()` picks up wherever `parse_section_title` left off. The corresponding code with an array wouldn't work:

```
sub parse_section {
  my @lines = @_;
  my $title = parse_section_title(@lines);
  my %variables = parse_variables(@lines);
  return [$title, \%variables];
}
```

There would be no way for `parse_section_title()` to remove the section title line from `@lines`. (This is a rather contrived example, but illustrates the possible problem. Packaging up `@lines` as an object, even by doing something as simple as passing `\@lines` instead, solves the problem.)

4.1.3 Other Common Examples of Iterators

Like all good, simple ideas, iterators pop up all over. If you remember only one example, remember filehandles, because filehandles are ubiquitous. But Perl has several other examples of built-in iterators. We'll take a quick tour of the most important ones.

Dirhandles are analogous to filehandles. They are created with the `opendir` function, and encapsulate a list of directory entries that can be read with the `readdir` operator:

```
opendir D, "/tmp";
@entries = readdir D;
```

But `readdir` in scalar context functions as an iterator, reading one entry at a time from `D`:

```
opendir D, "/tmp";
while (my $entry = readdir D) {
```

```
  # Do something with $entry
}
```

The built-in `glob` operator is similar, producing one file at a time whose name matches a certain pattern:

```
while (my $file = glob("/tmp/*.[ch]")) {
  # Do something with $file
}
```

Perl hashes always have an iterator built in to iterate over the keys or values in the hash. The `keys` and `values` functions produce lists of keys and values, respectively. If the hash is very large, these lists will be large, so Perl also provides a function to operate the iterator directly, namely `each`:

```
while (my $key = each %hash) {
  # Do something with $key
}
```

Normally the Perl regex engine just checks to see if a string matches a pattern, and reports true or false. However, it's sometimes of interest what part of the target string matched. In list context, the `m//g` operator produces a list of all matching substrings:

```
@matches = ("12:34:56" =~ m/(\d+)/g);
```

Here `@matches` contains `("12", "34", "56")`. In scalar context, `m//g` becomes the `NEXTVAL` operation for an iterator inside the regex, producing a different match each time:

```
while ("12:34:56" =~ m/(\d+)/g) {
  # do something with $1
}
```

We will see this useful and little-known feature in more detail in Chapter 8.

Now we'll see how we can build our own iterators.

4.2 HOMEMADE ITERATORS

Our `dir_walk()` function from Chapter 1 took a directory name and a callback function and searched the directory recursively, calling the callback for

each file and directory under the original directory. Now let's see if we can structure `dir_walk()` as an iterator. If we did, then instead of searching the directory, `dir_walk()` would return an iterator object. This object would support a `NEXTVAL` operation, which would return a different file or directory name each time it was called.

First let's make sure that doing this is actually worthwhile. Suppose we had such an iterator. Could we still use it in callback style? Certainly. Suppose `make_iterator` were a function that constructed an iterator that would return the filenames from a directory tree. Then we would still be able to emulate the original `dir_walk()` like this:

```
sub dir_walk {
  my ($dir, $filefunc, $dirfunc, $user) = @_;
  my $iterator = make_iterator($dir);
  while (my $filename = NEXTVAL($iterator)) {
    if (-f $filename) { $filefunc->($filename, $user) }
    else              {  $dirfunc->($filename, $user) }
  }
}
```

Here I've written `NEXTVAL($iterator)` to represent the `NEXTVAL` operation. Since we don't know yet how the iterator is implemented, we don't know what the real syntax of the `NEXTVAL` operation will be.

This example shows that the iterator version is at least as powerful as the original callback version. However, if we could build it, the iterator version would have several advantages over the callback version. We would be able to stop part way through processing the directory tree, and then pick up later where we left off, and we would have a file-tree-walk object that we could pass around from one function to another.

We'll use a really excellent trick to build our iterator: the iterator will be a function. The `NEXTVAL` operation on the iterator will simply be to call the function. When we call the iterator function it will do some computing, figure out what the next filename is, and return it. This means that the `NEXTVAL($iterator)` in our example code is actually doing `$iterator->()`.

The iterator will need to retain some state information inside it, but we've already seen that Perl functions can do that. In Chapter 3, memoized functions were able to retain the cache hash between calls.

Before we get into the details of the `dir_walk()` iterator, let's try out the idea on a simpler example.

4.2.1 A Trivial Iterator: `upto()`

Here's a function called `upto()` that builds iterators, and which is mostly useful as an example. Given two numbers, *m* and *n*, it returns an iterator that will return all the numbers between *m* and *n*, inclusive:

CODE LIBRARY
upto

```
sub upto {
  my ($m, $n) = @_;
  return sub {
    return $m <= $n ? $m++ : undef;
  };
}
my $it = upto(3, 5);
```

This constructs an iterator object that will count from 3 up to 5 if we ask it to. The iterator object is just an anonymous subroutine that has captured the values of `$m` and `$n`.

The iterator is a subroutine, returned by the final `return sub { ... }` statement. Because the iterator is a subroutine, its NEXTVAL operation is simply invoking the subroutine. The subroutine runs and returns a value; this is the next value from the iterator. To get the next value ("kick the iterator") we simply do:

```
my $nextval = $it->();
```

This stores the number 3 into `$nextval`. If we do it again, it stores 4. If we do it a third time, it stores 5. Any calls after that will return `undef`.

To loop over the iterator's values:

```
while (defined(my $val = $it->())) {
  # now do something with $val, such as:
  print "$val\n";
}
```

This prints 3, 4, 5, and then quits the loop.

This may have a substantial memory savings over something like:

```
for my $val (1 .. 10000000) {
  # now do something with $val
}
```

which, until Perl 5.005, would generate a gigantic list of numbers before starting the iteration.

If you have a sweet tooth, you can put some syntactic sugar on your serial:

CODE LIBRARY
Iterator_Utils.pm

```
package Iterator_Utils;
use base Exporter;
@EXPORT_OK = qw(NEXTVAL Iterator
                append imap igrep
                iterate_function filehandle_iterator list_iterator);
%EXPORT_TAGS = ('all' => \@EXPORT_OK);
sub NEXTVAL { $_[0]->() }
```

Then in place of the preceding examples, we can use this:

```
my $nextval = NEXTVAL($it);
```

and this:

```
while (defined(my $val = NEXTVAL($it))) {
  # now do something with $val
}
```

We'll do this from now on.

The internal operation of the iterator is simple. When the subroutine is called, it returns the value of `$m` and increments `$m` for next time. Eventually, `$m` exceeds `$n`, and the subroutine returns an undefined value thereafter. When an iterator runs out of data this way, we say it has been *exhausted*. We'll adopt the convention that a call to an exhausted iterator returns an undefined value, and then see some alternatives to this starting in Section 4.5.

SYNTACTIC SUGAR FOR MANUFACTURING ITERATORS

From now on, instead of writing `return sub { ... }` in a function, we will write `return Iterator { ... }` to make it clear that an iterator is being constructed:

```
sub upto {
  my ($m, $n) = @_;
```

```
  return Iterator {
    return $m <= $n ? $m++ : undef;
  };
}
```

This bit of sugar is easy to accomplish:

```
sub Iterator (&) { return $_[0] }
```

when we write this:

```
Iterator { ... }
```

Perl behaves as though we had written:

```
Iterator(sub { ... })
```

instead. Once past the sugar, the `Iterator()` function itself is trivial. Since the iterator *is* the anonymous function, it returns the argument unchanged.

Using this `Iterator()` sugar may make the code a little easier to understand. It will also give us an opportunity to hang additional semantics on iterator construction if we want to, by adding features to the `Iterator()` function. We will see an example of this in Section 4.5.7.

4.2.2 `dir_walk()`

Now that we've seen a function that builds simple iterators, we can investigate a more useful one, which builds iterators that walk a directory tree and generate filenames one at a time:

CODE LIBRARY
dir-walk-iterator

```
# iterator version
sub dir_walk {
  my @queue = shift;
  return Iterator {
    while (@queue) {
      my $file = shift @queue;
      if (-d $file) {
        opendir my $dh, $file or next;
        my @newfiles = grep {$_ ne "." && $_ ne ".."} readdir $dh;
        push @queue, map "$file/$_", @newfiles;
```

```
            }
            return $file;
          } else {
            return;
          }
        };
      }
```

The pattern here is the same as in `upto()`. `dir_walk()` is a function that sets up some state variables for the iterator and then returns a closure that captures the state variables. When the closure is executed, it computes and returns the next filename, updating the state variables in the process.

The closure maintains a queue of the files and directories that it hasn't yet examined. Initially, the queue contains only the single top-level directory that the user asked it to search. Each time the iterator is invoked, it removes the item at the front of the queue. If this item is a plain file, the iterator returns it immediately; if the item is a directory, the iterator reads the directory and queues the directory's contents before returning the name of the directory.

After enough calls to the iterator, the queue will become empty. Once this happens, the iterator is exhausted, and further calls to the iterator will return `undef`. In this case, `undef` doesn't cause a semipredicate problem, because no valid filename is ever `undef`.

There is one subtle point to make here. The items in `@queue` must be full paths like `./src/perl/japh.pl`, not basenames like `japh.pl`, or else the `-d` operator won't work. A common error when using `-d` is to get the basenames back from `readdir` and test them with `-d` immediately. This doesn't work, because `-d`, like all file operators, interprets a bare filename as a request to look for that name in the *current* directory. In order to use `-d`, we have to track the directory names also.

The `map` function accomplishes this. When we read the filenames out of the directory named `$file` with `readdir`, we get only the basenames. The `map` appends the directory name to each basename before the result is put on the queue. The result is full paths that work properly with `-d`.

Even if we didn't need the full paths for use with `-d`, the user of the iterator probably needs them. It's not usually useful to be told that the program has located a file named `japh.pl` unless you also find out which directory it's in.

4.2.3 On Clever Inspirations

Although this works well, it has one big defect: it appears to have required cleverness. The original `dir_walk()` from Chapter 1 was reasonably

straightforward: process the current file, and if it happens to be a directory, make recursive calls to process its contents. The iterator version is not recursive; in place of recursion, it maintains a queue.

The problem that the queue is solving is that a recursive function maintains a lot of state on Perl's internal call stack. Here's the recursive function `dir_walk()` again:

```
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);
    }
  }
}
```

Each recursive call down in the `while` loop must save the values of `$top`, `$code`, `$DIR`, and `$file` on the call stack; when `dir_walk()` is re-entered, new instances of these variables are created. The values must be saved so that they can be restored when the recursive call returns; at this time, the new instances are destroyed.

When the `dir_walk()` function finally returns to its original caller, all of the state information that was held in `$top`, `$code`, `$DIR`, and `$file` has been lost. In order for the iterator to simulate a recursive function, it needs to be able to return to its caller without losing all that state information.

Recursion is essentially an automatic stack-management feature. When our function makes a recursive call, Perl takes care of saving the function's state information on its private, internal stack, and restoring it again as necessary. But here the automatic management isn't what we want; we need manual control over what is saved and restored, so recursion doesn't work. Instead, we replace the call stack with the `@queue` variable and do all our stack management manually, with `push` and `shift`.

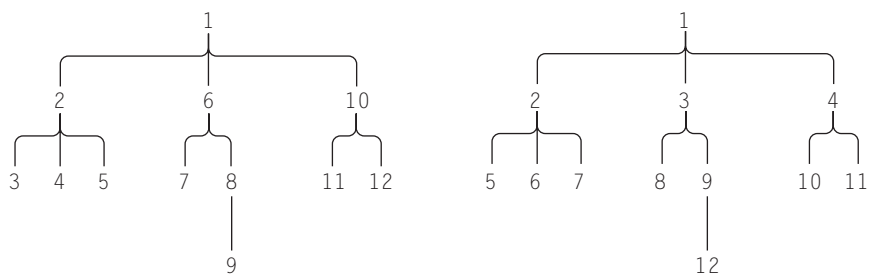


FIGURE 4.1 Depth-first traversal/breadth-first traversal.

The cost of the manual stack management is the trouble we have to go to. But the payoff, as do-it-yourselfers know, is flexibility. A recursive function for directory walking usually traverses the tree in depth-first order, visiting all the contents of each directory before moving on to the next directory. Sometimes we might prefer a breadth-first search, where all the files and directories at one level of the tree are visited before those lower down. Figure 4.1 illustrates both methods.

To get the recursive function to traverse the tree in breadth-first order or in any order other than depth-first is very difficult. But the iterator version accomplishes this easily. The previous iterator code traverses the directory in breadth-first order. If we replace `shift` with `pop`, `@queue` behaves as a stack, rather than a queue, and the iterator generates its output in depth-first order, exactly as the original recursive function did.

Replacing the recursion with the queue seems like a clever inspiration, but clever inspirations are usually in short supply. In Chapter 5, we'll see that any recursive function can be turned into an iterator in a formulaic way, so that we can save our clever inspirations for something else.

4.3 EXAMPLES

Let's see some possibly useful examples of iterators. We'll start with a replacement for `File::Find`, a variation on `dir_walk()`. It searches a directory hierarchy, looking for possibly interesting files:

CODE LIBRARY
interesting-files

```
sub interesting_files {
  my $is_interesting = shift;
```

```
  my @queue = @_;
  return Iterator {
    while (@queue) {
      my $file = shift @queue;
      if (-d $file) {
        opendir my $dh, $file or next;
        my @newfiles = grep {$_ ne "." && $_ ne ".."} readdir $dh;
        push @queue, map "$file/$_", @newfiles;
      }
      return $file if $is_interesting->($file);
    }
    return;
  };
}
```

Here we've made only a few changes. `interesting_files()` accepts a callback, `$is_interesting`, which will return true if its argument is the name of an "interesting" file. We'll also allow the user to specify more than one initial directory to search. This is trivial: We just take all the given directory names and load them into the initial queue.

The returned iterator is very similar. Instead of returning every file that it finds in the queue, the iterator returns only if the file is interesting, as determined by the callback. Otherwise, the iterator shifts another file off the queue and tries again. If the queue is exhausted before an interesting file is found, control leaves the `while` loop and the iterator returns `undef`. If the user calls the iterator again, the queue is still empty, so the iterator returns `undef` immediately.

To use this, we might write:

```
# Files are deemed to be interesting if they mention octopuses
sub contains_octopuses {
  my $file = shift;
  return unless -T $file && open my($fh), "<", $file;
  while (<$fh>) {
    return 1 if /octopus/i;
  }
  return;
}
my $octopus_file =
  interesting_files(\&contains_octopuses, 'uploads', 'downloads');
```

Now that we have the iterator, we can find all the interesting files:

```
while ($file = NEXTVAL($octopus_file)) {
  # do something with the file
}
```

Or perhaps we only want to know if there are any interesting files at all:

```
if (NEXTVAL($next_octopus)) {
  # yes, there is an interesting file
} else {
  # no, there isn't.
}
undef $next_octopus;
```

With a recursive function, we might have had trouble stopping the function when we found the interesting file; with the iterator, it's trivial, since it only searches as far as is necessary to find the first interesting file, and then leaves the rest of the hierarchy unsearched and waiting in the queue. When we `undef $next_octopus`, this saved state is discarded, and the memory used for storing it is freed.

4.3.1 Permutations

A *permutation* is a rearrangement of the items in a list. A frequently asked question in newsgroups is how to produce all the permutations of a certain list. For example, the permutations of the list `('red', 'yellow', 'blue')` are:

```
(   ['red', 'yellow', 'blue'],
    ['red', 'blue', 'yellow'],
    ['yellow', 'red', 'blue'],
    ['yellow', 'blue', 'red'],
    ['blue', 'red', 'yellow'],
    ['blue', 'yellow', 'red'],
)
```

It's not completely clear to me why this is useful. Last time it came up in the newsgroup, I asked the poster, and he explained that he was trying to generate

a name for a new product by assembling short phrases or syllables into different orders. Regardless of whether this is a good idea, it does seem to be something people want to do.

A beginner who tries to solve this problem may be completely puzzled. A programmer with more experience will immediately try to write a recursive function to generate the list, and will usually produce something that works. For example, here's the solution from the Perl Frequently Asked Questions List, written by Tom Christiansen and Nathan Torkington:

```
sub permute {
    my @items = @{ $_[0] };
    my @perms = @{ $_[1] };
    unless (@items) {
        print "@perms\n";
    } else {
        my(@newitems,@newperms,$i);
        foreach $i (0 .. $#items) {
            @newitems = @items;
            @newperms = @perms;
            unshift(@newperms, splice(@newitems, $i, 1));
            permute([@newitems], [@newperms]);
        }
    }
}
# sample call:
permute([qw(red yellow blue green)], []);
```

Items are removed from `@items` and placed onto the end of `@perms`. When all the items have been so placed, `@items` is empty and the resulting permutation, which is in `@perms`, is printed. (We should probably replace the `print` with a call to a callback.) The important part of this function is the `else` clause. In this clause, the function removes one of the unused items from the `@items` array, appends it to the end of the `@perms` array, and calls itself recursively to distribute the remaining items.

This solution works, but has a glaring problem. If you pass in a list of ten items, it doesn't return until it has printed all 3,628,800 permutations. This is likely to take a lot of time — twenty or thirty minutes on my computer. If we modify the function to generate a list of permutations, it's even worse. It returns a list of 3,628,800 items, each of which is an array of 10 items. This is likely to use up a substantial portion of your computer's real memory; if it does, your OS is likely to start thrashing while trying to compute the result, and it will take an

even longer time to finish. The function is inefficient to begin with, because it performs six array copies per call, a total of 24,227,478 copies in our preceding example above. The function is simply too slow to be practical except in trivial cases. And since we probably can't use all of the 3.6 million permutations anyway, most of the work is wasted.

This is the sort of problem that iterators were made to solve. We want to generate a list of permutations, but the list might be enormous. Rather than generating the entire list at once, as the FAQ solution does, we will use an iterator that generates the permutations one at a time.

To make an iterator for permutations requires either an insight, or techniques from later in the chapter. The insight-requiring version is interesting and instructive, so we'll look at it briefly before we move into the more generally useful versions that require less insight.

Regardless of the internals of `permute()`, here's how we'll be using it:

```
my $it = permute('A'..'D');

while (my @p = NEXTVAL($it)) {
  print "@p\n";
}
```

The function `permute()` constructs the iterator itself:

CODE LIBRARY
permute

```
sub permute {
  my @items = @_;
  my @pattern = (0) x @items;
  return Iterator {
    return unless @pattern;
    my @result = pattern_to_permutation(\@pattern, \@items);
    @pattern = increment_pattern(@pattern);
    return @result;
  };
}
```

Each permutation is represented by a "pattern" that says in what order to select elements from the original list. Suppose the original list is `('A', 'B', 'C', 'D')`. A pattern of `2 0 1 0` selects (and removes) item 2 from the original list, the `'C'`, leaving `('A', 'B', 'D')`; then item 0, `'A'`, from the remaining items; then item 1, the `'D'`, then item 0, the `'B'`; the result is the permutation `('C', 'A', 'D', 'B')`. This selection process is performed by

`pattern_to_permutation()`:

```
sub pattern_to_permutation {
  my $pattern = shift;
  my @items = @{shift()};
  my @r;
  for (@$pattern) {
    push @r, splice(@items, $_, 1);
  }
  @r;
}
```

The generation of the patterns is the interesting part. What patterns make sense? If there are four items in the original list, then the first element of the pattern must be a number between 0 and 3, the second element must be a number between 0 and 2, the third must be 0 or 1, and the last element must be 0. Each pattern corresponds to a different permutation; if we can generate all possible patterns, we can generate all possible permutations.

Generating all the patterns is performed by `increment_pattern()`. For this example, it generates the following patterns in the following order:

0 0 0 0	1 1 0 0	2 2 0 0
0 0 1 0	1 1 1 0	2 2 1 0
0 1 0 0	1 2 0 0	3 0 0 0
0 1 1 0	1 2 1 0	3 0 1 0
0 2 0 0	2 0 0 0	3 1 0 0
0 2 1 0	2 0 1 0	3 1 1 0
1 0 0 0	2 1 0 0	3 2 0 0
1 0 1 0	2 1 1 0	3 2 1 0

What is the pattern here? It turns out that getting from one pattern to the next is rather simple:

1. Scan the numbers in the pattern from right to left.
2. If you can legally increment the current number, do so, and halt.
3. Otherwise, change the current number to 0 and continue.
4. If you fall off the left end, then the sequence was the last one.

This algorithm should sound familiar, because you learned it a long time ago. It's exactly the same as the algorithm you use to *count*:

210397
210398

```
210399
210400
210401
......
```

To increment a numeral, scan the digits right to left. If you find a digit that you can legally increment (that is, a digit that is less than 9) then increment it, and stop; you are finished. Otherwise, change the digit to 0 and continue leftwards. If you fall off the left end, it's because every digit was 9, so that was the last number. (You can now extend the number by inferring and incrementing an unwritten 0 just past the left end.)

To count in base 2, the algorithm is again the same. Only the definition of "legal digit" changes: instead of "less than 10" it is "less than 2". To generate the permutation patterns, the algorithm is the same, except that this time "legal" means "the digit in the *n*th column from the right may not exceed *n*."

The elements of the permutation pattern are like the wheels of an imaginary odometer. But where each wheel on a real odometer is the same size, and carries numbers from 0 to 9 (or 0 to 1 on planets where the odometer reads out in base 2), each wheel in the permutation odometer is a different size. The last one just has a 0 on it; the next has just a 0 and a 1, and so on. But like a real odometer, each wheel turns one notch when the wheel to its right has completed a whole revolution.

The code to manage a regular odometer looks like this:

```
sub increment_odometer {
  my @odometer = @_;
  my $wheel = $#odometer;    # start at rightmost wheel


  until ($odometer[$wheel] < 9 || $wheel < 0) {
    $odometer[$wheel] = 0;
    $wheel--; # next wheel to the left
  }
  if ($wheel < 0) {
    return;   # fell off the left end; no more sequences
  } else {
    $odometer[$wheel]++;  # this wheel now turns one notch
    return @odometer;
  }
}
```

The code to produce the permutation patterns is almost exactly the same:

```
sub increment_pattern {
  my @odometer = @_;
  my $wheel = $#odometer;    # start at rightmost wheel

  until ($odometer[$wheel] < $#odometer-$wheel || $wheel < 0) {
    $odometer[$wheel] = 0;
    $wheel--; # next wheel to the left
  }
  if ($wheel < 0) {
    return;   # fell off the left end; no more sequences
  } else {
    $odometer[$wheel]++; # this wheel now turns one notch
    return @odometer;
  }
}
```

We can simplify the code with a little mathematical trickery. Just as we can predict in advance what positions the wheels of an odometer will hold after we've travelled 19,683 miles, even if it reads out in base 2, we can predict what positions the wheels of our pattern-odometer will hold the 19,683rd time we call it:

CODE LIBRARY
permute-n

```
sub n_to_pat {
  my @odometer;
  my ($n, $length) = @_;
  for my $i (1 .. $length) {
    unshift @odometer, $n % $i;
    $n = int($n/$i);
  }
  return $n ? () : @odometer;
}
```

`permute()` must change a little to match, since the state information is now a simple counter instead of an entire pattern:

```
sub permute {
  my @items = @_;
  my $n = 0;
  return Iterator {
```

```
    my @pattern = n_to_pat($n, scalar(@items));
    my @result = pattern_to_permutation(\@pattern, \@items);
    $n++;
    return @result;
  };
}
```

This last function is an example of a useful class of iterators that return $f(0), f(1), f(2), \ldots$ for some function f:

```
sub iterate_function {
  my $n = 0;
  my $f = shift;
  return Iterator {
    return $f->($n++);
  };
}
```

This is an iterator that generates values of a function for $n = 0, 1, 2 \ldots$. You might want many values of the function, or few; an iterator may be a more flexible way to get them than a simple loop, because it is a data structure.

The permutation iterators shown here do a lot of splicing. `pattern_to_permutation()` copies the original list of items and then dismantles it; every time an element is removed the other elements must be shifted down in memory to fill up the gap. With enough ingenuity, it's possible to avoid this, abandoning the idea of the patterns. Instead of starting over with a fresh list every time, in the original order, and then using the pattern to select items from it to make the new permutation, we can take the previous permutation and just apply whatever transformation is appropriate to turn it into the new one:

CODE LIBRARY
permute-flop

```
sub permute {
  my @items = @_;
  my $n = 0;
  return Iterator {
    $n++, return @items if $n==0;
    my $i;
    my $p = $n;
    for ($i=1; $i<=@items && $p%$i==0; $i++) {
      $p /= $i;
    }
    my $d = $p % $i;
```

```
    my $j = @items - $i;
    return if $j < 0;

    @items[$j+1..$#items] = reverse @items[$j+1..$#items];
    @items[$j,$j+$d] = @items[$j+$d,$j];

    $n++;
    return @items;
  };
}
```

The key piece of code here is the pair of slice assignments of `@items`. The insight behind this code is that at any given stage, we can ignore the first few items and concentrate only on the last few. Let's say we're rearranging just the last three items. We start with something like `... A B C D` and produce the various rearrangements of the last three items, ending with `... A D C B`.

At this point, the last three items are in backwards order. We need to put them back in forward order (this is the assignment with the `reverse`) and then switch the `A`, the next item over, with one of the three we just finished permuting. (This is the second assignment.) We need to do this three times, first switching `A` with `B`, then with `C`, and finally with `D`; after each switch, we run again through all possible permutations of the last three items. Of course, there are complications, since permuting the last three items involves applying the same process to the last *two* items, and is itself part of the process of permuting the last four items.

4.3.2 Genomic Sequence Generator

In 1999, I got email from a biologist at the University of Virginia. He was working on the Human Genome Project, dealing with DNA. DNA is organized as a sequence of base pairs, each of which is typically represented by the letter `A`, `C`, `G`, or `T`. The information carried in the chromosome of any organism can be recorded as a string of these four letters. A bacteriophage will have a few thousand of these symbols, and a human chromosome will have between 30 and 300 million. Much of the Human Genome Project involved data munging on these strings; Perl was invaluable for this munging. (For more details about this, see Lincoln Stein's widely-reprinted article "How Perl Saved the Human Genome Project."[1])

The biologist who wrote to me wanted a function that, given an input pattern like `"A(CGT)CGT"`, would produce the output list `('ACCGT', 'AGCGT',`

1 The Perl Journal, Vol 1, #2 (Summer 1996) pp. 5–9.

'ATCGT'). The (CGT) in the input is a wildcard that indicates that the second position may be filled by any one of the symbols C, G, or T. Similarly, an input of "A(CT)G(AC)" should yield the list ('ACGA', 'ATGA', 'ACGC', 'ATGC'). He had written a recursive function to generate the appropriate output list, but was concerned that he would run into memory limitations if he used it on long, ambiguous inputs, where the result would be a list of many thousands of strings.

An iterator is exactly the right solution here:

CODE LIBRARY
make-genes-1

```
sub make_genes {
  my $pat = shift;
  my @tokens = split /[()]/, $pat;
  for (my $i = 1; $i < @tokens; $i += 2) {
    $tokens[$i] = [0, split(//, $tokens[$i])];
  }
  my $FINISHED = 0;
  return Iterator {
    return if $FINISHED;
    my $finished_incrementing = 0;
    my $result = "";
    for my $token (@tokens) {
      if (ref $token eq "") {    # plain string
        $result .= $token;
      } else {                   # wildcard
        my ($n, @c) = @$token;
        $result .= $c[$n];
        unless ($finished_incrementing) {
          if ($n == $#c) { $token->[0] = 0 }
          else { $token->[0]++; $finished_incrementing = 1 }
        }
      }
    }
    $FINISHED = 1 unless $finished_incrementing;
    return $result;
  }
}
```

Here the input pattern "AA(CGT)CG(AT)" is represented by the following data structure, which is stored in @tokens:

```
[ "AA",
  [ 0, "C", "G", "T"],
```

```
  "CG",
  [ 0, "A", "T"],
]
```

The code to construct the data structure uses some tricks:

```
my @tokens = split /[()]/, $pat;
for (my $i = 1; $i < @tokens; $i += 2) {
  $tokens[$i] = [0,split(//, $tokens[$i])];
}
```

The peculiar-looking `split` pattern says that `$pat` should be split wherever there is an open- or a close- parenthesis character. The return value has the convenient property that the wildcard sections are always in the odd-numbered positions in the resulting list. For example, `"AA(CGT)CG(AT)"` is split into `("AA", "CGT", "CG", "AT")`. Even if the string begins with a delimiter, `split` will insert an empty string into the initial position of the result: `"(A)C"` is split into `("", "A", "C")`.

The following code processes only the wildcard parts of the resulting `@tokens` list:

```
for (my $i = 1; $i < @tokens; $i += 2) {
  $tokens[$i] = [0,split(//, $tokens[$i])];
}
```

The odd-numbered elements of `("AA", "CGT", "CG", "AT")` are transformed by this into `("AA", [ 0, "C", "G", "T"], "CG", [ 0, "A", "T"])`. The iterator then captures this list, which is stored in `@tokens`. Elements of this list that are plain strings correspond to the non-wildcard parts of the input pattern, and are inserted into the output verbatim. Elements that are arrays correspond to the wildcard parts of the input pattern and indicate choice points.

The internal structure of the iterator is similar to the structure of the permutation generator. When it's run, it scans the token string, one token at a time. During the scan, it does two things: It accumulates an output string, and it adjusts the numeric parts of the wildcard tokens. Tokens are handled differently depending on whether they are plain strings (`ref $token eq ""`) or wildcards. Plain strings are just copied directly to the result.

Wildcard handling is a little more interesting. The wildcard token is first decomposed into its component parts:

```
my ($n, @c) = @$token;
```

`$n` says which element of `@c` should be chosen next:

```
$result .= $c[$n];
```

Then the iterator may need to adjust `$n` to have a different value so that a different element of `@c` will be chosen next time. In the permutation-pattern generator, we scanned from right to left, resetting wheels to zero until we found one small enough to be incremented. Here we're scanning from left to right, but the principle is the same. `$finished_incrementing` is a flag that tells the iterator whether it has been able to increment one of the digits, after which it doesn't need to adjust any of the others:

```
unless ($finished_incrementing) {
  if ($n == $#c) { $token->[0] = 0 }
  else { $token->[0]++; $finished_incrementing = 1 }
}
```

The function can increment the value in a wildcard token if it would still index a valid element of `@c` afterwards. Otherwise, the value is reset to zero and the iterator keeps looking. This is analogous to the way we used `increment_pattern()` earlier to cycle through all possible permutation patterns; here we use the same sort of odometer technique to cycle through all possible selections of the wildcards.

When we have cycled through all the possible choices, the numbers in the wildcard tokens all have their maximum possible values; we can recognize this condition because we will have scanned all of them without finding one we could increment, and so `$finished_incrementing` will still be false after the scan. The iterator sets the `$FINISHED` flag so that it doesn't start over again from the beginning; thereafter, the iterator returns immediately, without generating a string:

```
$FINISHED = 1 unless $finished_incrementing;
```

There's nothing in this iterator that treats `A`, `C`, `T`, and `G` specially, so we can use it as a generic string generator:

```
my $it = make_genes('(abc)(de)-(12)');
print "$s\n" while $s = NEXTVAL($it);
```

The output looks like this:

```
ad-1
bd-1
```

```
cd-1
ae-1
be-1
ce-1
ad-2
bd-2
cd-2
ae-2
be-2
ce-2
```

Biologists don't usually use `(ACT)` to indicate a choice of `A`, `C`, or `T`; they typically use the single letter `H`. I don't know if the biologist who asked me this question was trying to avoid confusing me with unnecessary detail, or if he really did want to handle patterns like `(ACT)`. But supposing that we want to handle the standard abbreviations, a simple preprocessor will take care of it:

CODE LIBRARY
make-genes-2

```
%n_expand = qw(N ACGT
               B CGT D AGT H ACT V ACG
               K GT M AC R AG S CG W AT Y CT);
sub make_dna_sequences {
  my $pat = shift;
  for my $abbrev (keys %n_expand) {
    $pat =~ s/$abbrev/($n_expand{$abbrev})/g;
  }
  return make_genes($pat);
}
```

4.3.3 Filehandle Iterators

Now we'll see how to turn an ordinary Perl filehandle into a synthetic closure-based iterator. Why would we want to this? Because in the rest of the chapter we'll develop many tools for composing and manipulating iterators, and these tools apply just as well to Perl filehandles as long as we use the following little wrapper:

```
sub filehandle_iterator {
  my $fh = shift;
  return Iterator { <$fh> };
}
```

We can now use:

```
my $it = filehandle_iterator(*STDIN);
while (defined(my $line = NEXTVAL($it))) {
  # do something with $line
}
```

4.3.4 A Flat-File Database

Now let's do a real application. We'll develop a small flat-file database. A *flat-file database* is one that stores the data in a plain text file, with one record per line.

Our database will have a format something like this:

CODE LIBRARY
db.txt

```
LASTNAME:FIRSTNAME:CITY:STATE:OWES
Adler:David:New York:NY:157.00
Ashton:Elaine:Boston:MA:0.00
Dominus:Mark:Philadelphia:PA:0.00
Orwant:Jon:Cambridge:MA:26.30
Schwern:Michael:New York:NY:149658.23
Wall:Larry:Mountain View:CA:-372.14
```

The first line is a header, sometimes called a *schema*, that defines the names of the fields; the later lines are data records. Each record has the same number of data fields, separated by colons. This sample of the data shows only six records, but the file might contain thousands of records. For large files, the iterator approach is especially important. A flat-file database must be searched entirely for every query, and this is slow. By using an iterator approach, we will allow programs to produce useful results before the entire file has been scanned.

We'll develop the database as an object-oriented class, `FlatDB`. The `FlatDB` class will support a `new` method that takes a data filename and returns a database handle object:

CODE LIBRARY
FlatDB.pm

```
package FlatDB;
my $FIELDSEP = qr/:/;

sub new {
  my $class = shift;
  my $file = shift;
  open my $fh, "<", $file or return;
  chomp(my $schema = <$fh>);
```

```
  my @field = split $FIELDSEP, $schema;
  my %fieldnum = map { uc $field[$_] => $_ } (0..$#field);
  bless { FH => $fh, FIELDS => \@field, FIELDNUM => \%fieldnum,
          FIELDSEP => $FIELDSEP } => $class;
}
```

The database handle object contains a number of items that might be useful, in addition to the open data filehandle itself. For our sample database, the contents of the database handle object look like this:

```
{
  FH => (the handle),
  FIELDS => ['LASTNAME', 'FIRSTNAME', 'CITY', 'STATE', 'OWES'],
  FIELDNUM => { CITY => 2,
                FIRSTNAME => 1,
                LASTNAME => 0,
                OWES => 4,
                STATE => 3,
              },
  FIELDSEP => qr/:/,
}
```

The database handle object will support a `query` method that takes a field name and a value and returns all the records that have the specified value in the field. But we don't want `query` to simply read all the records in the data file and return a list of matching records, because that might be very expensive. Instead, `query` will return an iterator that will return matching records one at a time:

```
# usage: $dbh->query(fieldname, value)
# returns all records for which (fieldname) matches (value)
use Fcntl ':seek';
sub query {
  my $self = shift;
  my ($field, $value) = @_;
  my $fieldnum = $self->{FIELDNUM}{uc $field};
  return unless defined $fieldnum;
  my $fh = $self->{FH};
  seek $fh, 0, SEEK_SET;
  <$fh>;                # discard schema line

  return Iterator {
    local $_;
    while (<$fh>) {
      chomp;
```

```
            my @fields = split $self->{FIELDSEP}, $_, -1;
            my $fieldval = $fields[$fieldnum];
            return $_ if $fieldval eq $value;
          }
          return;
        };
      }
```

`query` first looks in the `FIELDNUM` hash to ascertain two things. First, is the requested field name actually a field in the database, and second, if so, what number column is it? The result is stored in `$fieldnum`; if the field name is invalid, `query` returns `undef` to indicate an error. Otherwise, the function seeks the filehandle back to the beginning of the data to begin the search, using the `seek` function.

`seek()` has a rather strange interface, inherited from the original design of Unix in the 1970s: `seek($fh, $position, $whence)` positions the filehandle so that the next read or write will occur at byte position `$position`. The `$whence` argument is actually the integer 0, 1, or 2, but mnemonic names for these values are provided by the standard Perl `Fcntl` module. If `$whence` is the constant `SEEK_SET`, `$position` is interpreted as a number of bytes forward from the beginning of the file. Here we use `seek($fh, 0, SEEK_SET)`, which positions the handle at the beginning of the file, so that the following `<$fh>` reads and discards the schema line.

The `query` function then returns the iterator, which captures the values of `$self`, `$fh`, `$fieldnum`, and `$value`.

The iterator is quite simple. When it's invoked, it starts reading data lines from the database. It splits up each record into fields, and compares the appropriate field value (in `$fields[$fieldnum]`) with the desired value (in `$value`). If there's a match, it returns the current record immediately; if not, it tries the next record. When it reaches the end of the file, the `while` loop exits and the function returns an undefined result to indicate failure.

The iterator is planning to change the value of `$_` in the while loop. Since `$_` is a global variable, this means that the function calling the iterator might get a nasty surprise:

```
$_ = 'I love you';
NEXTVAL($q);
print $_;
```

We don't want the invocation of `$q` to change the value of `$_`. To prevent this, the iterator uses `local $_`. This saves the old value of `$_` on entry to the iterator, and

arranges for the old value to be automatically restored when the iterator returns. With this `local` line, it is safe for the iterator to use `$_` any way it wants to. You should probably take this precaution in any function that uses `$_`.

A simple demonstration:

```
use FlatDB;
my $dbh = FlatDB->new('db.txt') or die $!;

my $q = $dbh->query('STATE', 'NY');
while (my $rec = NEXTVAL($q)) {
  print $rec;
}
```

The output is:

```
Adler:David:New York:NY:157.00
Schwern:Michael:New York:NY:149658.23
```

Many obvious variations are possible. We might support different kinds of queries, which return a list of the fields, or a list of just some of the fields. Or instead of passing a field–value pair, we might pass a callback function that will be called with each record and returns true if the record is interesting:

```
use FlatDB;
my $dbh = FlatDB->new('db.txt') or die $!;

my $q = $dbh->callbackquery(sub { my %F=@_; $F{STATE} eq 'NY'});
while (my $rec = NEXTVAL($q)) {
  print $rec;
}

# Output as before
```

With `callbackquery` we can ask for a list of the people who owe more than $10, which was impossible with `->query`:

```
my $q = $dbh->callbackquery(sub { my %F=@_; $F{OWES} > 10 });
```

Similarly, we can now use Perl's full regex capabilities in queries:

```
my $q = $dbh->callbackquery(sub { my %F=@_; $F{FIRSTNAME} =~ /^M/ });
```

This callback approach is much more flexible than hardwiring every possible comparison type into the iterator code, and it's easy to support:

```
use Fcntl ':seek';
sub callbackquery {
  my $self = shift;
  my $is_interesting = shift;
  my $fh = $self->{FH};
  seek $fh, 0, SEEK_SET;
  <$fh>;                # discard header line

  return Iterator {
    local $_;
    while (<$fh>) {
      chomp;
      my %F;
      my @fieldnames = @{$self->{FIELDS}};
      my @fields = split $self->{FIELDSEP};
      for (0 .. $#fieldnames) {
        $F{$fieldnames[$_]} = $fields[$_];
      }
      return $_ if $is_interesting->(%F);
    }
    return;
  }
}
```

The only major change here is in the iterator itself, mostly to set up the `%F` hash that is passed to the callback. I originally had a hash slice assignment instead of the `for` loop:

```
@F{@{$self->{FIELDS}}} = split $self->{FIELDSEP};
```

The punctuation made my eyes glaze over, so I used the loop instead.

IMPROVED DATABASE

The database code we've just seen has one terrible drawback: All of the iterators share a single filehandle, and this means that only one iterator can be active at any time. Consider this example:

```
use FlatDB;
my $dbh = FlatDB->new('db.txt') or die $!;
```

```
my $q1 = $dbh->query('STATE', 'MA');
my $q2 = $dbh->query('STATE', 'NY');
for (1..2) {
  print NEXTVAL($q1), NEXTVAL($q2);
}
```

We'd like this to print both NY records and both MA records, but it doesn't; it produces only one of each:

```
Ashton:Elaine:Boston:MA:0.00
Schwern:Michael:New York:NY:149658.23
```

What goes wrong? We would like $q1 to generate records 2 and 4, and $q2 to generate records 1 and 5. The sequence of events is shown in Figure 4.2. $q1 executes the first time, and searches through the database looking for an MA record. In doing so, it skips over record 1 (David Adler) and then locates record 2 (Elaine Ashton), which it returns. The filehandle is now positioned at the beginning of the third record. When we invoke $q2, this is where the search continues. $q2 won't find record 1, because the handle is already positioned past record 1. Instead, the iterator skips the next two records, until it finds record 5 (Michael Schwern), which it returns. The filehandle is now positioned just before record 6 (Larry Wall). When $q1 executes the second time, it skips record 6, reaches the end of the file, and returns undef. All further calls to both iterators produce nothing but undef because the filehandle is stuck at the end of the file. Although some commercial databases (such as Sybase) have this same deficiency, we can do better, and we will.

The obvious solution is to have a separate filehandle for each iterator. But open filehandles are a limited resource, and a program might have many active iterators at any time, so we'll adopt a different solution. Each iterator will remember the position in the file at which its last search left off, and when it is invoked, it will reset the handle to that position and continue. This allows several iterators to share the same filehandle without getting confused.

We need to make only a few changes to query to support this:

```
# usage: $dbh->query(fieldname, value)
# returns all records for which (fieldname) matches (value)
use Fcntl ':seek';
sub query {
  my $self = shift;
  my ($field, $value) = @_;
  my $fieldnum = $self->{FIELDNUM}{uc $field};
```

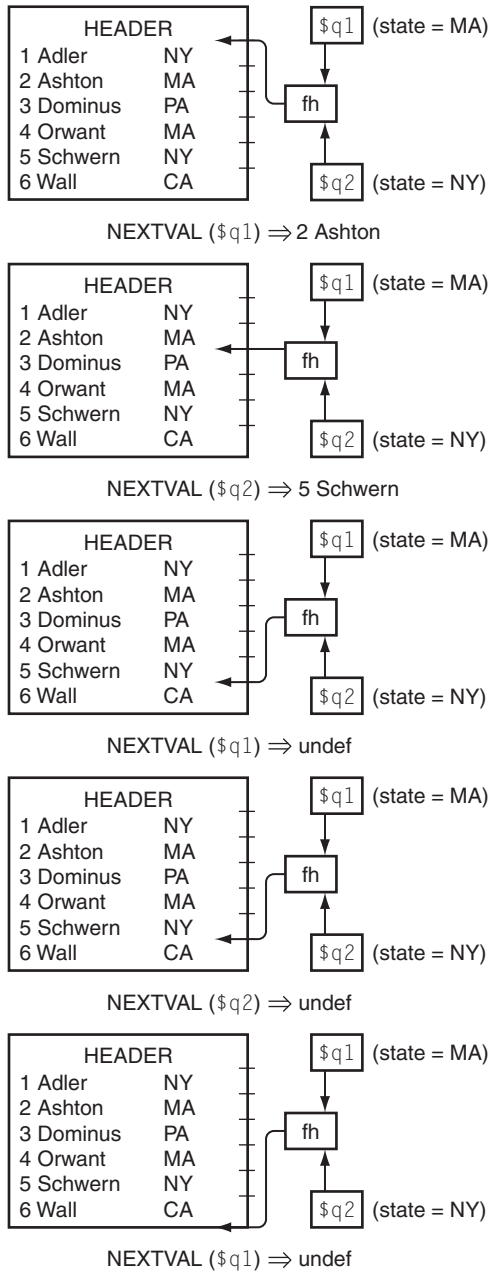


FIGURE 4.2 Interference between two query handles.

```
  return unless defined $fieldnum;
  my $fh = $self->{FH};
  seek $fh, 0, SEEK_SET;
  <$fh>;                # discard header line
  my $position = tell $fh;

  return Iterator {
    local $_;
    seek $fh, $position, SEEK_SET;
    while (<$fh>) {
      chomp;
      $position = tell $fh;
      my @fields = split $self->{FIELDSEP};
      my $fieldval = $fields[$fieldnum];
      return $_ if $fieldval eq $value;
    }
    return;
  };
}

# callbackquery with bug fix
use Fcntl ':seek';
sub callbackquery {
  my $self = shift;
  my $is_interesting = shift;
  my $fh = $self->{FH};
  seek $fh, 0, SEEK_SET;
  <$fh>;                # discard header line
  my $position = tell $fh;

  return Iterator {
    local $_;
    seek $fh, $position, SEEK_SET;
    while (<$fh>) {
      $position = tell $fh;
      my %F;
      my @fieldnames = @{$self->{FIELDS}};
      my @fields = split $self->{FIELDSEP};
      for (0 .. $#fieldnames) {
        $F{$fieldnames[$_]} = $fields[$_];
      }
      return $_ if $is_interesting->(%F);
```

```
        }
        return;
      };
    }

    1;
```

The iterators here capture one additional value, `$position`, which records the current position of the filehandle in the file; initially this position is at the start of the first data record. This position is supplied by the Perl `tell` operator, which returns the filehandle's current position; if this position is later used with `seek $fh, $position, SEEK_SET`, the filehandle will be set back to that position. This is precisely what the iterators do whenever they are invoked. Regardless of what other functions have used the filehandle in the meantime, or where they have left it, the first thing the iterators do is to seek the filehandle back to the current position using the `seek` operator. Each time an iterator reads a record, it updates its notion of the current position, again using `tell`, so its `seek` in a future invocation will skip the record that was just read.

With this change, our two-iterators-at-once example works perfectly:

```
Ashton:Elaine:Boston:MA:0.00
Adler:David:New York:NY:157.00
Orwant:Jon:Cambridge:MA:26.30
Schwern:Michael:New York:NY:149658.23
```

4.3.5 Searching Databases Backwards

Perhaps the most common occurrence of a flat-file database is a process log file. Anyone who runs a web server knows that the server can churn out megabytes of log information every day. These logs are essentially flat databases. Each line represents a request for a web page, and includes fields that describe the source of the request, the page requested, the date and time of the request, and the outcome of the request. A sample follows:

```
208.190.220.160 - - [04/Aug/2001:08:14:29 -0400] "GET /~mjd/pictures/new.gif HTTP/1.1"
  200 95 "http://perl.plover.com/" "Mozilla/5.0 (Macintosh; U; PPC; en-US; rv:0.9.2)
  Gecko/20010629"
195.3.19.207 - - [04/Aug/2001:13:39:11 -0400] "GET /pics/small-sigils.gif HTTP/1.1" 200 1586
  "http://perl.plover.com/" "Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0; DigExt)"
```

```
192.94.94.33 - - [07/Aug/2001:12:06:34 -0400] "GET /yak/Identity/slide005.html HTTP/1.0"
  200 821 "http://perl.plover.com/yak/Identity/slide004.html" "Mozilla/4.6 [en]
  (X11; I; SunOS 5.8 sun4u)"
199.93.193.10 - - [13/Aug/2001:13:04:39 -0400] "GET /yak/dirty/miller_glenn_r.jpg HTTP/1.0"
  200 4376 "http://perl.plover.com/yak/dirty/slide009.html" "Mozilla/4.77 [en] (X11; U;
  SunOS 5.6 sun4u)"
216.175.77.248 - - [15/Aug/2001:14:25:20 -0400] "GET /yak/handson/examples/wordsort.pl
  HTTP/1.0" 200 125 "http://perl.plover.com:80/yak/handson/examples/" "Wget/1.5.3"
194.39.218.254 - - [16/Aug/2001:07:44:02 -0400] "GET /pics/medium-sigils.gif HTTP/1.0" 304 -
  "http://perl.plover.com/local.html" "Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)"
210.239.93.70 - msdw [22/Aug/2001:01:29:28 -0400] "GET /class/msdw-tokyo/ HTTP/1.0" 401 469
  "http://perl.plover.com/class/" "Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 4.0)"
151.204.38.119 - - [25/Aug/2001:13:48:01 -0400] "GET /yak/path/hanoi06.gif HTTP/1.0" 200 239
  "http://perl.plover.com/yak/path/" "Mozilla/4.77 [en] (WinNT; U)"
```

One of the common tasks of system administrators is to search through the log files looking for certain matching records; for example, the last time a certain user visited, or the last time a certain page was fetched. In fact, Perl itself first rose to prominence as a tool for helping system administrators answer exactly these sorts of questions. A typical query will look something like this:

```
perl -ane 'print $F[10] if $F[6] =~ m{/book/$}' access-log
```

The `-n` option implies a loop; Perl will automatically read the input file `access-log` line by line and execute the indicated program once for each line. The `-a` option implies that each line will be automatically `split` into the special `@F` array. The `-e` option introduces the program, which uses the `@F` array that was set up by `-a`. In these log files, field #6 is the path of the page that is being requested, and field #10 is the URL of the "referring page," which is the one that contained a link to the page that is being requested. This query will yield the URLs of pages that may contain links to the page that talks about the book you are now reading.

If what you want is all such records, this works very well. But more often, you are more interested in recent activity than in old activity. The preceding `perl` command example produces the records in chronological order, with the oldest ones first, because that's the order in which they appear in the file. That means that to get to the part of interest, you have to wait until the entire file has been read, analyzed, and printed. If the file is large, this will take a long time.

One solution to this problem is to store the records in reverse order, with the most recent ones first. Unfortunately, under most operating systems, this is

impossible. Unix, for example, supports appending records only to the end of a file, not to the beginning.

Instead, we'll build an iterator that can read a file backwards, starting with the most recent records. If we plug this iterator into our existing query system, we'll get a flat-file database query system that produces the most recent records first, with no additional effort.

A QUERY PACKAGE THAT TRANSFORMS ITERATORS

There is one minor technical problem that we have to solve before we can proceed. As written, the `FlatDB` constructor wants a filename, not another iterator. A few changes are necessary to build a version that accepts an arbitrary iterator:

```
package FlatDB::Iterator;
my $FIELDSEP = qr/\s+/;

sub new {
  my $class = shift;
  my $it = shift;
  my @field = @_;
  my %fieldnum = map { uc $field[$_] => $_ } (0..$#field);
  bless { FH => $it, FIELDS => \@field, FIELDNUM => \%fieldnum,
          FIELDSEP => $FIELDSEP } => $class;
}
```

For the original `FlatDB` package, we assumed that the data file itself would begin with a schema line. HTTP log files don't have a schema line, so here we've assumed that the field names will be passed to the constructor as arguments. The calling sequence for `FlatDB::Iterator::new` is:

```
FlatDB::Iterator->new(
  $iterator,
  qw(address rfc931 username datetime tz method page protocol
     status bytes referrer agent)
);
```

The `qw(...)` list specifies the names of the fields in the data that will be produced by `$iterator`.

`query` requires only trivial changes. The code to skip the descriptor records goes away, and the code to fetch the next record changes from:

```
while (<$fh>) {
```

to:

```
while (defined ($_ = NEXTVAL($it))) {
```

A more subtle change is that we must get rid of the `seek $fh, 0, SEEK_SET` line, because there's no analogous operation for iterators. (We'll see in Chapter 6 how to build iterators that overcome this drawback.) This means that each database object can be used only for one query. After that, we must throw it away, because there's no way to rewind and reread the data:

```
# usage: $dbh->query(fieldname, value)
# returns all records for which (fieldname) matches (value)
sub query {
  my $self = shift;
  my ($field, $value) = @_;
  my $fieldnum = $self->{FIELDNUM}{uc $field};
  return unless defined $fieldnum;
  my $it = $self->{FH};
  # seek $fh, 0, SEEK_SET;
  # <$fh>;                # discard header line

  return Iterator {
    local $_;
    while (defined ($_ = NEXTVAL($it))) {
      my @fields = split $self->{FIELDSEP};
      my $fieldval = $fields[$fieldnum];
      return $_ if $fieldval eq $value;
    }
    return;
  };
}
```

It's similarly easy to write the amended version of `callbackquery`.

If `$it` were an iterator that produced the lines from a log file in reverse order, we could use:

```
my $qit =
  FlatDB::Iterator->new($it, @FIELDNAMES)->query($field, $value);
```

And `$qit` would be an iterator that would generate the specified records from the file, one at a time, starting with the most recent.

AN ITERATOR THAT READS FILES BACKWARDS

Building an iterator that reads a file backwards is more an exercise in systems programming than anything else. We'll take the easy way out and use the Unix `tac` program as our base. The `tac` program reads a file and emits its lines in reverse order:

```
sub readbackwards {
  my $file = shift;
  open my($fh), "|-", "tac", $file
    or return;
  return Iterator { return scalar(<$fh>) };
}
```

If `tac` isn't available, we can use the `File::ReadBackwards` module from CPAN instead:

```
use File::ReadBackwards;
sub readbackwards {
  my $file = shift;
  my $rbw = File::ReadBackwards->new($file)
    or return;
  return Iterator { return $rbw->readline };
}
```

PUTTING IT TOGETHER

We can now search a log file backwards:

```
my @fields = qw(address rfc931 username datetime tz method
                page protocol status bytes referrer agent);

my $logfile = readbackwards("/usr/local/apache/logs/access-log")
my $db = FlatDB::Iterator->new($logfile, @fields);
my $q = $db->callbackquery(sub {my %F=@_; $F{PAGE}=~ m{/book/$}});
while (1) {
  for (1..10) {
```

```
    print NEXTVAL($q);
  }
  print "q to quit; CR to continue\n";
  chomp(my $resp = <STDIN>);
  last if $resp =~ /q/i;
}
```

This program starts up fast and immediately produces the most recent few records, without reading through the entire file first. It uses little memory, even when there are many matching records.

We had to do some extra work to read a file backwards in the first place, but once we had done that, we could plug the iterator directly into our existing query system.

4.3.6 Random Number Generation

Perl comes with a built-in random number generator. The random numbers are not truly random; they're what's called *pseudo-random*, which means they're generated by a mechanical process that yields repeatable results. Perl typically uses the `rand`, `random`, or `drand48` function provided by the local C library. A typical random number generator function looks something like this:

```
my $seed = 1;

sub Rand {
  $seed = (27*$seed+11111) & 0x7fff;
  return $seed;
}
```

This example has poor randomness properties. For example, its output alternates between odd and even numbers. Don't use it in any software that needs truly random numbers.

The `Rand()` function takes no arguments and returns a new "random" number. The random number generator has an internal value, called the *seed*. Each time it is invoked, it performs a transformation on the seed, saves the new seed, and returns the new seed. Since the output for `Rand` depends only on the seed, we can think of it as generating a single sequence of numbers:

```
11138
16925
```

```
9334
985
4938
13365
11518
27185
24210
9421
...
```

We can see that the sequence produced by `Rand` must eventually repeat. Because the output is always an integer less than 32,768, if we call `Rand` 32,769 times, two of the calls must have returned the same value, let's say *v*. This means that the seed was *v* both times. But since the output of `Rand` depends only on the value of the seed, the outputs that follow the second appearance of *v* must be identical to those that followed the first appearance; this shows that the sequence repeats after no more than 32,768 calls. It might, of course, repeat much sooner than that. The length of the repeated portion is called the *period* of the generator. The sample random number generator shown here has a period of only 16384. Changing the `27` to `29` will increase the period to 32768. (The design of random number generators is a topic of some complexity. The interested reader is referred to *The Art of Computer Programming*, Volume II,[2] for more information about this.)

Since the output of the generator depends only on the seed, and the seed is initialized to 1, this random number generator will generate the same sequence of numbers, starting with 1, 11138, 16925, . . . each time the program is run.

Sometimes this is desirable. Suppose the program crashes. You might like to rerun it under the debugger to see what went wrong. But if the program's behavior depended on a sequence of random numbers, it will be important to be able to reproduce the same sequence of numbers, or else the debugging run may not do the same thing and may not reveal the problem.

Nevertheless, when you ask for random numbers, you usually want them to be different every time. For this reason, random number generators come with an auxiliary function for initializing the seed:

```
sub SRand {
  $seed = shift;
}
```

2 The Art of Computer Programming, Volume II: Seminumerical Algorithms, Donald E. Knuth, Addison–Wesley.

To get unpredictable random numbers, we call the `SRand()` function once, at the beginning of the program, with an argument that will vary from run to run, such as the current time or process ID number:

```
SRand($$);
```

The random generator will start at a different place in the sequence each time the program is run. If the program saves the seed in a file, the seed can be re-used later during a debugging run to force the generator to produce the same sequence of random numbers a second time.

This design is very common; C libraries come with paired sets of functions called `rand` and `srand`, or `random` and `srandom`, or `drand48` and `srand48`, which are analogous to `Rand()` and `SRand()`. The Perl built-in `rand` and `srand` functions work the same way, and are usually backed by one or another of the C function pairs.

This interface has several problems, however. One is that it's not clear who has responsibility for seeding the random generator. Suppose you have the following program:

```
use CGI::Push;
my $seed = shift || $$ ;
srand($seed);
open LOG, "> $logfile" or die ... ;
print LOG "Random seed: $seed\n";

do_push(...);
```

Normally, the program is run with no command-line argument, and the random number generator is seeded with the process ID as usual. The seed is then saved to the debugging log. If the program fails, it can be rerun, and the same seed can be supplied as a command-line argument.

However, there's a problem. The `CGI::Push` module also needs to generate random numbers, and it makes its own call to `srand` when `do_push()` is called. This will overwrite the seed that the main program wanted to use.

Imagine the problems this could cause. Suppose the main program had saved its seed to a file for use in a later debugging session, and then the program did indeed crash. You start the debugger, and tell the program to re-use the same seed. And it does, up until the call to `do_push`, which re-seeds the generator, using its own seeding policy, which knows nothing of your debugging strategy. After the call to `do_push`, all the random numbers produced by `rand` are unpredictable again. The more separate modules the program uses, the more likely they are

to fight over the random number seed like drunken fraternity brothers fighting over the remote control.

A related problem concerns the generator itself. Recall that our example generator generates a sequence of 16,384 numbers before repeating, but with careful choice of the constants, we can improve it to get the maximum possible period of 32,768. Now consider the following program:

```
use Foo;

while (<>) {
  my $random = Rand();
  # do something with $random
  foo();
}
```

Unbeknownst to the author of this program, the `foo` function, imported from the `Foo` module, also generates a random number using `Rand`. This means that the pool of 32,768 random numbers is split between the main program and `foo`, with the main program getting the first, third, fifth, seventh random numbers, and so on, and the `foo` function getting the second, fourth, six, eighth, and so on. Since the main program is seeing only half of the pool of numbers, the period of the sequence it sees is only half as big. Whatever is done with `$random`, it will repeat every 16,384 lines. Even though we were at some pains to make the pool of random numbers as large as possible, our efforts were foiled, because both sources of random data were drawing from the same well of entropy.

The underlying problem here is that the random number generator is a single global resource, and the seed is a global variable. Global variables almost always have this kind of allocation problem. Iterators provide a solution. It's easy to convert the `Rand` function to an iterator:

CODE LIBRARY
rng-iterator.pl

```
sub make_rand {
  my $seed = shift || (time & 0x7fff);
  return Iterator {
    $seed = (29*$seed+11111) & 0x7fff;
    return $seed;
  }
}
```

Calling `make_rand()` returns an iterator that generates a different random number each time it is called. The optional argument to `make_rand()` specifies

the seed; if omitted, it is derived from the current time of day. Revisiting the last example:

```
use Foo;
my $rng = make_rand();

while (<>) {
  my $random = NEXTVAL($rng);
  # do something with $random
  foo();
}
```

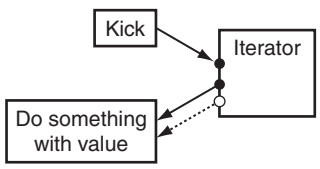
The main program now has its own private random number generator, represented by `$rng`. This generator is not available to `foo`. `foo` can allocate its own random number generator, which is completely independent of `$rng`. Each generator is seeded separately, so there is no question about who bears responsibility for the initial seeding. Each part of the program is responsible for seeding its own generators at the time they are created.

If it is desirable for two parts of the program to share a generator for some reason, they can do that simply by sharing the iterator object.

4.4 FILTERS AND TRANSFORMS

Because iterators are objects, we can write functions to operate on them. What might be useful? Since iterators encapsulate lists, we should expect that the same sort of functions that are useful for lists will also be useful for iterators. Two of Perl's most useful list functions are `grep` and `map`. `grep` filters a list, returning a new list of all the elements that possess some property. `map` transforms a list, applying an operation to each element, and returning a new list. Both of these are useful operations for iterators.

In the diagrams that follow, iterators will be represented as follows:



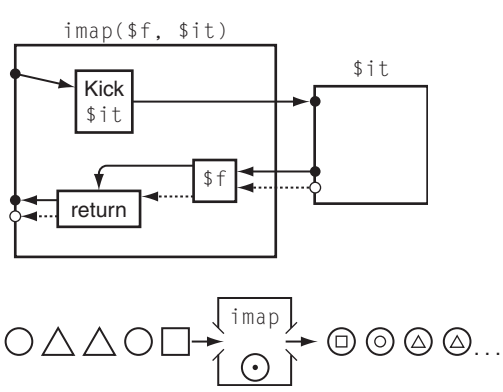
The boxes represent actions, as in a regular flow chart. When an iterator is kicked, it emits a value, which is represented by the dotted line coming out of the left-hand side. Solid arrows represent the flow of control, dotted arrows the flow of data.

4.4.1 imap()

We'll see the iterator version of map first because it's simpler:

```
sub imap {
  my ($transform, $it) = @_;
  return Iterator {
    my $next = NEXTVAL($it);
    return unless defined $next;
    return $transform->($next);
  }
}
```

imap() takes two arguments: a callback function and an iterator. It returns a new iterator whose output is the same as that of the original iterator, but with every element transformed by the callback function:



For example, suppose we wanted a random number generator that behaved more like Perl's built-in generator, returning a fraction between 0 and 1 instead of an integer between 0 and 32767. We could rewrite make_rand(), but there's no need if we have imap():

```
my $rng = imap(sub { $_[0] / 37268 }, make_rand());
```

make_rand() constructs an iterator that generates a random integer, as before. We pass the iterator to imap(), which returns a different iterator, which is stored in $rng. When we invoke $rng, it calls the original iterator, which

returns an integer; this is stored in `$next` and passed to `$transform`, which divides the integer by 32768 and returns the result. The first few outputs from `$rng` are:

```
0.298915960072985
0.174170870451862
0.0735751851454331
0.673312224965118
0.480626811205324
0.168267682730493
0.781635719652249
0.104996243425995
0.705269936674895
0.528147472362348
...
```

The syntax for `imap()` is a little cumbersome. Since it's analogous to `map`, it would be nice if it had the same syntax. Fortunately, Perl allows this. The first step is to add a prototype to `imap()`:

```
sub imap (&$) {
  my ($transform, $it) = @_;
  return Iterator {
    my $next = NEXTVAL($it);
    return unless defined $next;
    return $transform->($next);
  }
}
```

The `(&$)` tells Perl that `imap()` will get exactly two arguments, that the first will be a code reference (`&` symbolizes subroutines) and the second will be a scalar (`$` symbolizes scalars). When we announce to Perl that a function's first argument will be a code reference, the announcement triggers a change in the Perl parser to allow the word `sub` to be omitted from before the first argument and the comma to be omitted after — just as with `map` and `grep`. We can now write:

```
my $rng = imap { $_[0] / 37268 } make_rand();
```

The `$` in the prototype will ensure that `make_rand()` will be called in scalar context and will produce a single scalar result; normally, it would be called in list context and might produce many scalars.

The only difference between this syntax and `map`'s is that we had to use `$_[0]` in the code block instead of `$_`. If we are willing to commit more trickery, we can use `$_` instead of `$_[0]` in the code reference, just as with `map`:

```
sub imap (&$) {
  my ($transform, $it) = @_;
  return Iterator {
     local $_ = NEXTVAL($it);
     return unless defined $_;
     return $transform->();
  }
}
```

Instead of storing the output of the underlying iterator into a private `$next` variable, we store it into `$_`. Then we needn't pass it explicitly to `$transform`; `$transform` can see the value anyway, because `$_` is global. As usual, we use `local` to save the old value of `$_` before we overwrite it. We can now write:

```
my $rng = imap { $_ / 37268 } make_rand();
```

which has exactly the same syntax as `map`.

4.4.2 `igrep()`

The trickery is the same for `igrep()`, and only the control flow is different:

```
sub igrep (&$) {
  my ($is_interesting, $it) = @_;
  return Iterator {
    local $_;
    while (defined ($_ = NEXTVAL($it))) {
      return $_ if $is_interesting->();
    }
    return;
  }
}
```

The iterator returned by `igrep()` kicks the underlying iterator repeatedly until it starts returning `undef` (at which point `igrep()` gives up and also returns `undef`) or it returns an interesting item, as judged by the `$is_interesting` callback (see Figure 4.3). When it finds an interesting item, it returns it.

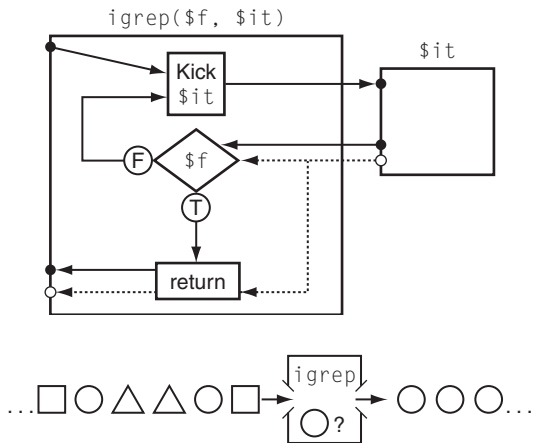


FIGURE 4.3 igrep().

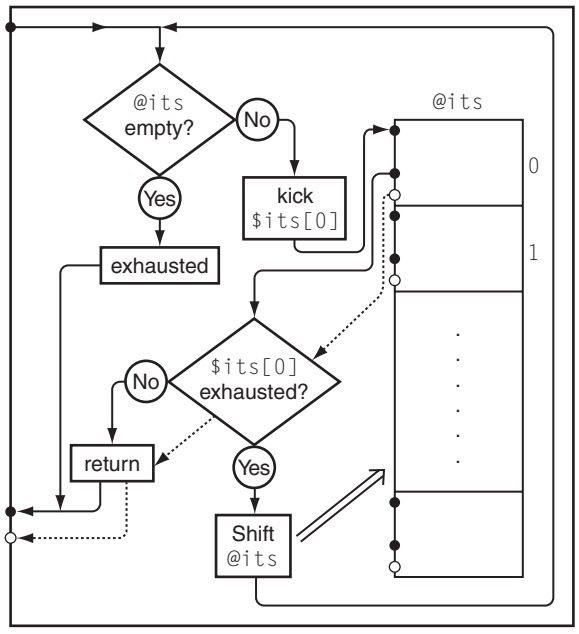
Now that we have igrep(), we no longer need interesting_files(), which searched a directory tree and returned the interesting files. Instead, we can get the same effect by filtering the output of dir_walk():

```
# instead of          my $next_octopus =
#    interesting_files(\&contains_octopuses,'uploads','downloads');

my $next_octopus = igrep { contains_octopuses($_) }
                         dir_walk('uploads', 'downloads');

while ($file = NEXTVAL($next_octopus)) {
  # do something with the file
}
```

4.4.3 list_iterator()

Sometimes it's convenient to have a way to turn a list into an iterator:

```
sub list_iterator {
  my @items = @_;
  return Iterator {
    return shift @items;
  };
}
```



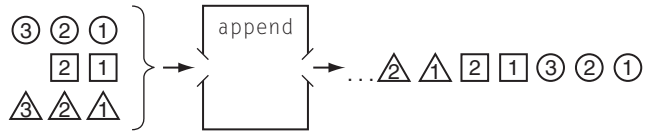


FIGURE 4.4 append().

list_iterator('fish', 'dog', 'carrot') produces an iterator that generates 'fish', then 'dog', then 'carrot', and then an infinite sequence of undefined values.

4.4.4 append()

map and grep aren't the only important operations on lists. Some of the most important operations don't even have names, because they're so common. One of these is the append() operation (see Figure 4.4), which attaches two or more lists together head-to-tail to yield a single list.

```
sub append {
  my @its = @_;
  return Iterator {
    while (@its) {
```

```
        my $val = NEXTVAL($its[0]);
        return $val if defined $val;
        shift @its;  # Discard exhausted iterator
      }
      return;
    };
}
```

We call `append()` with zero or more iterators as arguments. It returns a new iterator that produces all the items from the first iterator, followed by all the items from the second iterator, and so on. For example, `append(upto(1,3), upto(5,8), upto(9,11))` returns an iterator that yields the values 1, 2, 3, 5, 6, 7, 8, 9, 10, 11 in order, and undefined values thereafter.

The `while` loop invokes the first iterator in the argument list; if it yields an undefined value, the first iterator is exhausted, so it is discarded (by the `shift`) and the next argument is tried. This continues until a nonempty iterator is found or the argument list is itself exhausted. Then the loop exits and the iterator returns the value from the nonempty iterator, if there was one, or `undef` if not.

4.5 THE SEMIPREDICATE PROBLEM

So far, our iterators have all indicated exhaustion by returning the undefined value. For the examples we've seen, this is perfectly adequate. An undefined value can never be confused with any number, any file path, any permutation of a list, or any string. But in general, an iterator might generate data that includes the undefined value. For example, consider an iterator whose job is to generate values from certain columns of a SQL database. SQL databases have a special `NULL` value that is different from every number and every string. It's natural to represent this `NULL` value in Perl with `undef`, and in fact the Perl `DBI` module does this. But if the database field can potentially contain any string value or `NULL`, then the iterator cannot use `undef` to indicate end-of-data as well as `NULL`.

Most of our iterator utility functions, such as `imap()`, will become confused if the iterator generates `undef` as a valid data value. If the iterator returns `undef` to indicate that the database contained `NULL`, the utility function will erroneously conclude that the iterator has been exhausted.

You may recall from Chapter 3 that this is called the *semipredicate problem*. Our iterators are semipredicates because they return `undef` to indicate exhaustion, and a data value otherwise. The difficulty occurs when we need `undef` to sometimes be understood as a data value instead of as a flag indicating exhaustion.

4.5.1 Avoiding the Problem

There are several ways around this. One is simply to declare that no iterator is ever allowed to return an undefined value; if `undef` is a legal return, the iterator must be restructured to return its data in some other format. Suppose we have an iterator that might return any scalar value, including `undef`:

```
sub make_iterator {
  ...
  return Iterator {
    my $return_value;
    ...
    if (exhausted) {
      return;
    } else {
      return $return_value;
    }
  }
}
```

This won't work because the caller will not be able to distinguish an iterator returning `undef` to indicate exhaustion from one that happens to be returning an undefined value of `$return_value`. We can restructure this iterator to be unambiguous:

```
# array reference version
sub make_iterator {
  ...
  return Iterator {
    my $return_value;
    ...
    if (exhausted) {
      return;
    } else {
      return [$return_value];
    }
  }
}
```

The iterator now always returns an array reference, except that it returns `undef` when it is exhausted. When `$return_value` is undefined, the iterator returns `[undef]`, which is easily distinguished from `undef` because it *is* defined. In fact, the caller doesn't even need to use `defined`, because the exhaustion indicator is false, while all other possible returns, including `[undef]`, are true.

Often this simple solution imposes no extra costs. The Perl `DBI` module uses this strategy in some cases. The `$sth->fetchrow_arrayref` method returns a row of data from the result of a database query using an undefined return to indicate that there are no more rows available.

Each row may contain undefined values, which represent the database's `NULL` entries, and if a row contains only one field, it may be a single `NULL`. But `fetchrow_arrayref` always returns the row data as an array reference, so a row with a single `NULL` is easily distinguished from the `undef` that indicates that no more rows are available:

```
while (my $row = $sth->fetchrow_arrayref) {
  # do something with this $row
}
# no more rows
```

A related solution is to require that the iterator be used only in list context:

```
# list-context-only version
sub make_iterator {
  ...
  return Iterator {
    my $return_value;
    ...
    if (exhausted) {
      return ();
    } else {
      return $return_value;
    }
  }
}
```

Now a successful call to the iterator *always* returns a value, and when the iterator is exhausted it stops returning values. We can use it like this:

```
while (($val) = NEXTVAL($iterator)) {
  # do something with $val
}
# iterator is exhausted
```

The value of a list assignment when used as the condition of a `while` or `if` statement is the number of values returned on the right-hand side of the assignment. Even if the iterator returns a list such as `(undef)` or `(0)`, the condition tested by

`while` will be 1, which is true. When the iterator is exhausted, it will return an empty list, the condition will evaluate to 0, and the loop will exit.

4.5.2 Alternative `undef`s

Sometimes we don't want to avoid the semipredicate problem by returning lists or arrayrefs, or we can't. For `DBI`, the technique is natural, because the data in a row of a database is naturally represented as an array. But if our iterator returns only single scalars, it may be inconvenient to wrap up every value as an array reference just to unwrap it again in the caller. There are several strategies for dealing with the problem instead of avoiding it this way.

The first strategy is that when we construct the iterator, we could supply a special value that we know will never be returned normally. This value could be captured in the iterator, something like this:

```
sub make_iterator {
  my (..., $final_value, ...) = @_;
  return Iterator {
    ...
    if (exhausted) { return $final_value }
    ...
  }
}
```

However, this would be annoying, since we'd have to inform functions like `imap()` what the special final value was for each iterator they needed to process. It's possible to construct a single value that will work for every iterator in the entire program, so that the iterators and the functions that use them can all assume it.

To construct such a value, we use a trick borrowed from the C language. In C, many functions return pointers; for example, the memory allocator (`malloc`) returns a pointer to a block of unused memory, and the `fopen` function returns a pointer to C's version of a filehandle object. C has a special pointer value, called the *null pointer*, which these functions return when there is an error. The null pointer's only useful property is that it compares unequal to any valid pointer.

Occasionally a C programmer wants to write a function that can return pointers, and indicate two sorts of errors with two different special values. The null pointer will serve for one of these two, but what will be the other one? In C there's an easy solution: use `malloc` to allocate a byte and return a

pointer to it; this will serve as the alternative special value. No other valid pointer will ever point to that address, because `malloc` has reserved it.

We can do an analogous thing in Perl. We will invent a new value that can't possibly be confused with any legal data value, including `undef`. We will use this "alternative undef" in place of the real `undef`. How can we do that? It's simple:

```
$EXHAUSTED = [];
```

`$EXHAUSTED` is now guaranteed to be distinct from any other value our program will ever generate. If a value is not an arrayref, we can distinguish it from `$EXHAUSTED`, which is an arrayref. If a value is an arrayref, then it must refer to a different array than `$EXHAUSTED` does, unless it was copied from `$EXHAUSTED` itself. That is,

```
\@a == $EXHAUSTED
```

is guaranteed to be false for all arrays `@a`. Similarly, `[...] == $EXHAUSTED` is guaranteed to be false, because `[...]` generates a new, fresh array, which is different from the one in `$EXHAUSTED`.

We can now write functions, analogous to `undef` and `defined`, to generate and detect special values:

```
{
  my $EXHAUSTED = [];

  # like 'undef()'
  sub special () { return $EXHAUSTED }

  # like 'not defined()'
  sub is_special ($) {
    my $arg = shift;
    ref($arg) && $arg == $EXHAUSTED;
  }
}
```

Having done this, we can build an iterator that uses `$EXHAUSTED` to indicate end-of-data:

```
sub dbi_query_iterator {
  my ($sth, @params) = @_;
```

```
    $sth->execute(@params) or return ;
    return Iterator {
      my $row;
      if ($sth && $row = $sth->fetchrow_arrayref()) {
        return $row->[0];
      } else { # exhausted
        undef $sth;
        return special();
      }
    }
  }
```

`$sth` is a Perl `DBI` object that represents a SQL statement. `dbi_query_iterator()` takes this object and yields an iterator that will produce the results of the query, one at a time. It asks the database to execute the query by calling `$sth->execute`; if this fails it returns failure. Otherwise, the iterator uses the `DBI` method `fetchrow_arrayref` to fetch the next row of data from the database; it extracts the first item from the row, if there was one, and returns it. This item might be `undef`, which indicates a `NULL` database value.

When there are no more rows, `fetchrow_arrayref` returns `undef`. The iterator discards the private copy of `$sth` and returns the special value. Since the special value is distinguishable from any other scalar value, the caller receives an unambiguous indication that no more data is forthcoming. On future calls, the iterator continues to return the special value.

The caller can use the iterator this way:

```
until (is_special(my $value = NEXTVAL($iterator))) {
  # do something with $value
}
# no more data
```

It might seem that we could simplify the definition of `is_special()` by eliminating the test for `ref($arg)`:

```
# MIGHT NOT ALWAYS WORK
sub is_special {
  my $arg = shift;
  $arg == $EXHAUSTED;
}
```

But this isn't so. If `$arg` were an integer, and we were very unlucky, the test might yield true! Comparing an integer to a reference with == actually compares

the integer to the machine address at which the referenced data is stored, and these two numbers might match. Similarly, using plain `eq` without the additional `ref` test wouldn't be enough, because `$arg` might happen to be the string `ARRAY(0x436c1d)` and might happen to match the stringized version of the reference exactly. So we need to check `$arg` for referencehood before using == or `eq`. This kind of failure is extremely unlikely, but if it *did* happen it could be very difficult to reproduce, and it might take us weeks to track down the problem, so it's better to be on the safe side.

4.5.3 Rewriting Utilities

If we switch from using an undefined value to using a special value, it looks like all the code that uses iterators, including functions like `imap()`, will have to be rewritten, because we've changed the interface specification. For example, `imap()` becomes:

```
sub special_imap {
  my ($transform, $it) = @_;
  return Iterator {
    my $next = NEXTVAL($it);
    return special() if is_special($next);
    return $transform->($next);
  }
}
```

It might seem that we need to pick an interface and then stick with it, but we don't necessarily. Suppose we write all our utilities to use the special-value interface; our `imap()` is actually the `special_imap()` above.

Now we want to use some iterator, say `$uit`, that uses the `undef` convention instead of the special-value convention. Do we have to re-implement `imap()` and our other utilities to deal with the new `undef` iterators? No, we don't:

```
sub undef_to_special {
  my $it = shift;
  return Iterator {
    my $val = NEXTVAL($it);
    return defined($val) ? $val : special() ;
  }
}
```

We can't pass `$uit` directly to `imap()`, but we can pass `undef_to_special($uit)` instead, and it will do what we want. `undef_to_special()` takes an `undef`-style

iterator and turns it into a special-value-style iterator. It is like a mask that an iterator can wear to pretend that its interface is something else. Any `undef`-style iterator can put on the `undef_to_special()` mask and pretend to be an special-value-style iterator.

We could also write a similar `special_to_undef()` mask function to convert the other way. Of course, it wouldn't work correctly on iterators that might return undefined values.

4.5.4 Iterators That Return Multiple Values

The "special value" solution to the semipredicate problem works adequately, but has the disadvantage that the `special()` and `is_special()` functions may have to be exported everywhere in the program. (And also the possible disadvantage that it may be peculiar.) Since functions in Perl can return multiple values, and an iterator is just a function, a more straightforward solution may be to have the iterator return two values at a time; the second value will indicate whether the iterator is exhausted:

```
sub dbi_query_iterator {
  my ($sth, @params) = @_;
  $sth->execute(@params) or return ;
  return Iterator {
    my $row;
    if ($sth && $row = $sth->fetchrow_arrayref()) {
      return ($row->[0], 1);
    } else { # exhausted
      if ($sth) { $sth->finish; undef $sth; }
      return (undef, 0);
    }
  }
}
```

To use this, the caller writes something like:

```
while (my ($val, $continue) = NEXTVAL($iterator)) {
   last unless $continue;
   # do something with $val...
}
# now it is empty
```

4.5.5 Explicit Exhaustion Function

The iterator knows when it is exhausted, and it will tell us if we ask it. But we haven't provided any way to do that; all we can do is ask it for the next value with the NEXTVAL operator. We would like to be able to ask the iterator two types of questions: "Are you empty?" and if not, "Since you're not empty, what is the next item?" There's an obvious hook to hang this expansion on: Since the iterator is simply a function, we will pass it an argument to tell it which question we want answered. To preserve compatibility (and to optimize the common case) we'll leave the iterator's behavior the same when it is called without arguments; calling it with no arguments will continue to ask an iterator to return its next value. But we'll add new semantics: if we pass the iterator the string `"exhausted?"`, it will return a true or false value indicating whether or not it is empty. With this new functionality added, our `dbi_query_iterator()` becomes:

```
sub dbi_query_iterator {
  my ($sth, @params) = @_;
  $sth->execute(@params) or return ;
  my $row = $sth->fetchrow_arrayref();
  return Iterator {
    my $action = shift() || 'nextval';
    if ($action eq 'exhausted?') {
       return ! defined $row;
    } elsif ($action eq 'nextval') {
       my $oldrow = $row;
       $row = $sth->fetchrow_arrayref;
       return $oldrow->[0];
    }
  }
}
```

The iterator now returns `undef` either when the rows are exhausted or when the value in the row happens to be NULL, and the caller can't tell which. But that doesn't matter, because the caller of this iterator isn't looking for `undef` to know when to stop reading the iterator. Instead, the caller is doing this:

```
until ($iterator->('exhausted?')) {
  my $val = NEXTVAL($iterator);
  ...
}
# now it is empty
```

We can provide syntactic sugar for `'exhausted?'` that is analogous to `NEXTVAL`:

```
sub EXHAUSTED {
  $_[0]->('exhausted?');
}
```

This loop then becomes:

```
until (EXHAUSTED($iterator)) {
  my $val = NEXTVAL($iterator);
  ...
}
# now it is empty
```

Or, if you don't like `until`, we could define the obvious `MORE` function, and write:

```
while (MORE($iterator)) {
  my $val = NEXTVAL($iterator);
  ...
}
# now it is empty
```

A mask function allows iterators in the old, `undef`-returning style to support `EXHAUSTED` queries:

```
sub undef_to_exhausted {
  my $it = shift;
  my $val = NEXTVAL($it);
  return Iterator {
    my $action = shift || 'nextval';
    if ($action eq 'nextval') {
      my $oldval = $val;
      $val = NEXTVAL($it);
      return $oldval;
    } elsif ($action eq 'exhausted?') {
      return not defined $val;
    }
  }
}
```

If our versions of utilities such as `imap()` are set up to support the `NEXTVAL`/`EXHAUSTED` interface, we can still use the old-style iterators with them, by wrapping them in an `undef_to_exhausted()` mask. Similarly, the utilities produce

iterators with the NEXTVAL/EXHAUSTED interface, so if we want to use one in the old undef style (and we know that's safe) we can build a mask function that goes the other way:

```
sub exhausted_to_undef {
  my $it = shift;
  return Iterator {
    if (EXHAUSTED($it)) { return }
    else                { return NEXTVAL($it) }
  }
}
```

It's better if all our iterators conform to the same interface style, of course, but the mask functions show that they don't have to, and that if we make the wrong choice early on and have to switch to a different system later on, we can do that, or if we want to use the simple style for most iterators and save the more complicated two-operation interface for a few cases, we can do that also.

Interfacing different sorts of iterators is something we've also been doing implicitly through the entire chapter. In Section 4.3 we saw `filehandle_iterator()`, which is essentially a mask function: it converts one kind of iterator (a filehandle) into another (our synthetic, function-based iterators). If we needed to, we could go in the other direction and write a mask function that would wrap up one of our iterators as a filehandle, using Perl's tied filehandle interface. We will see how to do this in Section 4.6.3.

Similarly, the various versions of `dbi_query_iterator()` were also mask functions, converting from DBI's statement handles to function-based iterators. We could go in the other direction here if we had to, probably by building an object class that obeyed the DBI statement handle interface, but implementing our own versions of `->fetchrow_arrayref` and the like.

4.5.6 Four-Operation Iterators

As long as we're on the topic of iterators that support two kinds of queries (one for exhaustion and one for the next value), we might as well see this idea in its full generality. The C-style `for` loop has a very general model of iteration:

```
for (initialize; test; update) {
  action;
}
```

Occasionally you may need an equally general iterator:

```
for ($it->('start'); not $it->('exhausted?'); $it->('next')) {
  # do something with $it->('value');
}
```

`exhausted?` here is as in the previous section. The `next` operation doesn't return anything; it just tells the iterator to forget the current value and to get ready to deliver the next value. `value` tells the iterator to return the current value; if we make two calls to `$it->('value')` without `$it->('next')` in between, we'll get the same value both times.

`start` initializes the iterator. An explicit `start` call simplifies the code in some of the iterators we've seen already. For example, `dbi_query_iterator()` had to do `fetchrow_arrayref()` in two places, once to initialize itself, and once after every `NEXTVAL`:

```
sub dbi_query_iterator {
  my ($sth, @params) = @_;
  $sth->execute(@params) or return;
  my $row = $sth->fetchrow_arrayref();
  return Iterator {
    my $action = shift() || 'nextval';
    if ($action eq 'exhausted?') {
       return ! defined $row;
    } elsif ($action eq 'nextval') {
       my $oldrow = $row;
       $row = $sth->fetchrow_arrayref;
       return $oldrow;
    }
  }
}
```

Here's the four-operation version of the same function. Although it does more, the code is almost the same length:

```
sub dbi_query_iterator {
  my ($sth, @params) = @_;
  $sth->execute(@params) or return ;
  my $row;
  return Iterator {
    my $action = shift();
```

```
    if ($action eq 'exhausted?') {
       return ! defined $row;
    } elsif ($action eq 'value') {
       return $row;
    } elsif ($action eq 'next' || $action eq 'start'){
       $row = $sth->fetchrow_arrayref;
    } else {
       die "Unknown iterator operation '$action'";
    }
  }
}
```

We can still support the old NEXTVAL operation if we want to:

```
sub dbi_query_iterator {
  my ($sth, @params) = @_;
  $sth->execute(@params) or return ;
  my $row;
  return Iterator {
    my $action = shift() || 'nextval';
    if ($action eq 'exhausted?') {
       return ! defined $row;
    } elsif ($action eq 'value') {
       return $row;
    } elsif ($action eq 'next'|| $action eq 'start'
          || $action eq 'nextval') {
       return $row = $sth->fetchrow_arrayref;
    } else {
       die "Unknown iterator operation '$action'";
    }
  }
}
```

Here calling `start` twice has a possibly surprising effect: `start` and `next` are identical! Sometimes it's useful to have `start` mean that the iterator should start over at the beginning, forcing it to go back to the beginning of its notional list. The cost of this is that the iterator has to remember a list of all the values it has ever produced, in case someone tells it to start over. It also complicates the programming. While this is occasionally useful enough to be worth the extra costs, it's usually simpler to declare that calling `start` twice on the same iterator is erroneous.

4.5.7 Iterator Methods

People can be funny about syntax, and Perl programmers are even more obsessed with syntax than most people. When Larry Wall described the syntax of Perl 6 for the first time, people were up in arms because he was replacing the `->` operator with `.` and the `.` operator with `_`. Even though there is only a little difference between:

```
$it->start
```

and:

```
$it->('start')
```

people love the first one and hate the second one. It's easy to make the first syntax available though. The first syntax is an object method call, so we need to make our iterators into Perl objects. We do that with a small change to the `Iterator()` function:

```
sub Iterator (&) {
  my $it = shift;
  bless $it => 'Iter';
}
```

Now we can add whatever methods we want for iterators:

```
sub Iter::start       { $_[0]->('start')      }
sub Iter::exhausted   { $_[0]->('exhausted?') }
sub Iter::next        { $_[0]->('next')       }
sub Iter::value       { $_[0]->('value')      }
```

The prototypical loop, which looked like this:

```
for ($it->('start'); not $it->('exhausted?'); $it->('next')) {
  # do something with $it->('value');
}
```

can now be written like this:

```
for ($it->start; not $it->exhausted; $it->next) {
  # do something with $it->value;
}
```

4.6 ALTERNATIVE INTERFACES TO ITERATORS

We've already seen that there are two ways to get the next value from any of these iterators. We can use:

```
$next_value = NEXTVAL($iterator);
```

or, equivalently, we can write:

```
$next_value = $iterator->();
```

which is just what `NEXTVAL` is doing behind the scenes.

So much for syntax; semantics is more interesting. Since an iterator is just a function, we aren't limited to iterators that return scalar values.

4.6.1 Using `foreach` to Loop Over More Than One Array

An occasional question is how to loop over two arrays simultaneously. Perl provides `foreach`, and the equivalent `for`, which are convenient when you want to loop over a single array:

```
for $element (@a) {
  # do something with $element
}
```

But suppose you want to write a function that compares two arrays element by element and reports whether they are the same? (The obvious notation, `@x == @y`, returns true whenever the two arrays have the same length.) You'd like to loop over pairs of corresponding elements, but there's no way to do that. The only obvious way out is to loop over the array indices instead:

```
sub equal_arrays (\@\@) {
  my ($x, $y) = @_;
  return unless @$x == @$y;     # arrays are the same length?
  for my $i (0 .. $#$x) {
    return unless $x->[$i] eq $y->[$i];   # mismatched elements
  }
  return 1;                     # arrays are equal
}
```

To call this function, we write:

```
if (equal_arrays(@x, @y)) { ... }
```

The (\@\@) prototype tells Perl that the two argument arrays should be passed by reference, instead of being flattened into a single list of array elements. Inside the function, the two references are stored into $x and $y. The @$x == @$y test makes sure that the two arrays have equal lengths before examining the elements.

An alternative approach is to build an iterator that can be used whenever this sort of loop is required:

```
sub equal_arrays (\@\@) {
  my ($x, $y) = @_
  return unless @$x == @$y;
  my $xy = each_array(@_);
  while (my ($xe, $ye) = NEXTVAL($xy)) {
    return unless $xe eq $ye;
  }
  return 1;
}
```

The following iterator, invented by Eric Roode, does the trick:

```
sub each_array {
  my @arrays = @_;
  my $cur_elt = 0;
  my $max_size = 0;

  # Get the length of the longest input array
  for (@arrays) {
    $max_size = @$_ if @$_ > $max_size;
  }

  return Iterator {
    $cur_elt = 0, return () if $cur_elt >= $max_size;
    my $i = $cur_elt++;
    return map $_->[$i], @arrays;
  };
}
```

The caller of this function passes in references to one or more arrays, which are stored into @arrays. The iterator captures this variable, as well as $cur_elt, which records its current position. Each time the iterator is invoked, it gathers one element from each array and returns the list of elements; it also increments $cur_elt for next time. When $cur_elt is larger than the last index of the

largest array, the iterator is exhausted and returns the empty list instead, resetting `$cur_elt` so that the iterator can be used again.

Another place where this function might be useful is in generating HTML forms. Suppose `@labels` contains a list of user-visible labels for radio buttons, say `('Alaska', 'Alabama', ... 'Yukon Territory')`, and `@values` contains the internal tags that will be sent back by the browser when the user selects each button, say `('AK', 'AL', ..., 'YT')`.

Probably these arrays should have been structured as an array of arrays, `(['Alaska', 'AK'], ['Alabama', 'AL'], ...)`, but that's not always convenient or even possible. (It's tempting to say that the structure should have been a hash, but that's a mistake for several reasons: the hash loses the order of the data; also, neither labels nor values are required to be distinct.)

With the multiple-list iterator, we can write things like:

```
my $buttons = each_array(\@labels, \@values);
...

while (my ($label, $value) = NEXTVAL($buttons)) {
  print HTML qq{<input type=radio value="$value"> $label<br>\n};
}
```

The version Eric Roode uses has a clever twist:

```
sub each_array (\@;\@\@\@\@\@\@\@\@\@\@\@\@\@\@\@\@\@\@\@\@\@\@) {
  my @arrays    = @_;
  ...
}
```

The prototype here says that the arguments will be arrays, and will be passed implicitly by reference, and that there will be at least one. The `;` character separates the required arguments from the optional ones. Because of the prototype, Eric can leave off the backslashes in calls to `each_array()`, and write:

```
$ea = each_array(@a, @b, @c, @d);
```

While this is slightly more convenient for his common case, it forecloses the possibility of passing literal array references or array references contained in scalar variables, so that:

```
$aref = \@a;
$ea = each_array($aref, [1,2,3]);
```

becomes illegal; it must be written in this somewhat bizarre form:

```
$ea = each_array(@$aref, @{[1,2,3]});
```

Besides, a more interesting use for the argument space is coming up.

It's not clear what the best behavior is when the function is passed arrays of different lengths, say `(1,2,3)` and `('A','B','C','D')`. The preceding version returns four pairs: `(1, 'A')`, `(2, 'B')`, `(3, 'C')`, and `(undef, 'D')`. It might be preferable in some circumstances to have the iterator become exhausted at the end of the shortest input array, instead of the longest. To get this behavior, just replace the maximum computation with a minimum. We can also provide a version of `each_array` that has either behavior, depending on an optional argument:

```
sub each_array {
  my @arrays    = @_;
  my $stop_type = ref $arrays[0] ? 'maximum' : shift @arrays;
  my $stop_size = @{$arrays[0]};
  my $cur_elt   = 0;

  # Get the length of the longest (or shortest) input array
  if ($stop_type eq 'maximum') {
    for (@arrays) {
      $stop_size = @$_ if @$_ > $stop_size;
    }
  } elsif ($stop_type eq 'minimum') {
    for (@arrays) {
      $stop_size = @$_ if @$_ < $stop_size;
    }
  } else {
    croak "each_array: unknown stopping behavior '$stop_type'";
  }

  return Iterator {
    return ()  if $cur_elt >= $stop_size;
    my $i = $cur_elt++;
    return map $_->[$i], @arrays;
  };
}
```

If the first argument is not an array ref, we shift it off into `$stop_type`, which otherwise defaults to `"maximum"`. Only `"maximum"` and `"minimum"` are supported

here. As usual, we can gain flexibility and eliminate the repeated code by allowing a callback argument to specify the method for selecting the stopping point:

```
sub each_array {
  my @arrays    = @_;
  my $stop_func = UNIVERSAL::isa($arrays[0], 'ARRAY') ? 'maximum' : shift @arrays;
  my $stop_size  = @{$arrays[0]};
  my %stop_funcs = ('maximum'=>
                      sub { $_[0] > $_[1] ? $_[0] : $_[1] },
                    'minimum'=>
                      sub { $_[0] < $_[1] ? $_[0] : $_[1] },
                   );

  unless (ref $stop_func eq 'CODE') {
    $stop_func = $stop_funcs{$stop_func}
      or croak "each_array: unknown stopping behavior '$stop_func'";
  }

  # Get the length of the longest (or shortest) input array
  for (@arrays) {
    $stop_size = &$stop_func($stop_size, scalar @$_);
  }

  my $cur_elt   = 0;
  return Iterator {
    return ()  if $cur_elt >= $stop_size;
    my $i = $cur_elt++;
    return map $_->[$i], @arrays;
  };
}
```

each_array now has several calling conventions. The basic one we've seen already:

```
my $each = each_array(\@x, \@y, ...);
```

This builds an iterator that produces one list for each element in the longest input array. The first argument to each_array may also be a callback function that is used to generate the array limit:

```
sub max { $_[0] > $_[1] ? $_[0] : $_[1] };
my $each = each_array(\&max, \@x, \@y, ...);
```

The callback function is given two arguments: the current stopping size and the size of one of the arrays. It returns a new choice of stopping size. In the preceding example, it returns the maximum. The third way of calling `each_array` is to pass a string key that symbolizes a commonly chosen function:

```
my $each = each_array('maximum', \@x, \@y, ...);
```

This selects a canned maximum function from the table `%stop_funcs`.

We can get a different behavior by supplying a more interesting function:

```
my $all_equal = sub {
    if ($_[0] == $_[1]) { return $_[0] }
    croak "each_array: Not every array has length $_[0]";
};

my $each = each_array($all_equal, \@x, \@y, ...);
```

Here `each_array` croaks unless every input array has the same length. If this behavior turns out to be frequently needed, we can add the `$all_equal` function to the `%stop_funcs` table and support:

```
my $each = each_array('all_equal', \@x, \@y, ...);
```

without breaking backward compatibility.

4.6.2 An Iterator with an `each`-Like Interface

Every Perl hash contains an iterator component, which is accessed by `each()`. Each call to `each()` produces another key from the hash, and, in list context, the corresponding value.

Following this model, we can make a transformation function, analogous to `imap()`, that may produce a more useful result:

```
sub eachlike (&$) {
  my ($transform, $it) = @_;
  return Iterator {
    local $_ = NEXTVAL($it);
    return unless defined $_;
    my $value = $transform->();
    return wantarray ? ($_, $value) : $value;
  }
}
```

`eachlike()` transforms an iterator by applying a function to every element of the iterator. In scalar context, the iterators produced by `eachlike()` behave exactly the same as those produced by `imap()`. But in list context, an `eachlike()` iterator returns two values: the original, unmodified value, and the transformed value. For example:

```
my $n = eachlike { $_ * 2 } upto(3,5);
```

This loop will print the values 6, 8, 10, just as if we had used `imap()`:

```
while (defined(my $q = NEXTVAL($n))) {
  print "$q\n";
}
```

This loop will print `3 6`, `4 8`, `5 10`:

```
while (my @q = NEXTVAL($n)) {
  print "@q\n";
}
```

Our implementation of `dir_walk()` took a callback argument; the callback function was applied to each filename in the directory tree, and the iterator returned the resulting values. This complicated the implementation of `dir_walk()`. `eachlike()` renders this complication entirely unnecessary. It's quite enough for `dir_walk()` to return plain filenames, because in place of:

```
my $it = dir_walk($DIR, sub { ... } );
```

We can now use:

```
my $it = eachlike { ... } dir_walk($DIR);
```

In scalar context, `$it` will behave as if it had been generated by the former, more complicated version of `dir_walk()`. But in addition, it can also be used like this:

```
while (my ($filename, $value) = NEXTVAL($it)) {
  # do something with the filename or the value or both
}
```

For example, print out all the dangling symbolic links in a directory:

```
my $it = eachlike { -l && ! -e } dir_walk($DIR);
while (my ($filename, $badlink) = NEXTVAL($it)) {
  print "$filename is a dangling link" if $badlink;
}
```

4.6.3 Tied Variable Interfaces

The ordinary interface to an iterator may be unfamiliar. By using Perl's `tie` feature, which allows us to associate any semantics we want with a perl variable, we can make an iterator look like an ordinary scalar. Or rather, it will look like one of Perl's magical scalars, such as `$!` or `$.`, which might contain different values depending on when it's examined.

SUMMARY OF `tie`

In Perl, scalar variables may be *tied*. This means that access to the variable is mediated by a Perl object. The `tie` function associates the variable with a particular object, which we say is *tied to* the variable.

Suppose the scalar `$s` is tied to the object `$o`.

When you write this:	Perl actually does this instead:
`print $s;`	`print $o->FETCH();`
`$r = $s;`	`$r = $o->FETCH();`
`$s = $value;`	`$o->STORE($value);`

If anyone tries to read from or write to `$s`, then instead of doing whatever it would usually do, Perl makes a method call on `$o` instead. An attempt to read `$s` turns into `$o->FETCH()`. The return value of the `FETCH` method is reported as being the value stored in `$s`.

Similarly, an attempt to do `$s = $value` turns into `$o->STORE($value)`. It is the responsibility of the `STORE` method to store the value somewhere so that it can be retrieved by a later call to `FETCH`.

To tie a scalar variable in Perl, we use the built-in `tie` operator:

```
tie $scalar, 'Package', ARGUMENTS...
```

This makes a method call to `Package->TIESCALAR(`*`ARGUMENTS`*`...)`. `TIESCALAR` must be an object constructor. The object that it returns is the one that will be associated with `$scalar` and which will receive subsequent `FETCH` and `STORE` messages.

Here's a (silly) example:

```
package CIA;
sub TIESCALAR {
  my $package = shift;
```

```
  my $self = {};
  bless $self => $package;
}
sub STORE { }
sub FETCH { "<<Access forbidden>>" }
```

This is an implementation of an extra-secret tied scalar. If we tie a scalar into the `CIA` package, all data stored into it becomes inaccessible:

```
tie $secret, 'CIA';
```

This creates the association between `$secret` and the object constructed by `CIA::TIESCALAR`. Accesses to `$secret` will turn into `FETCH` and `STORE` methods on the object:

```
$secret = 'atomic ray';
```

Instead of storing `'atomic ray'` in the usual way, `STORE` is invoked. It's passed the object and new value as arguments, but it just throws them away, without storing the data anywhere. That's OK, because if you later ask the scalar what it contains:

```
print "The secret weapon is '$secret'.\n"
```

The output is:

```
The secret weapon is '<<Access forbidden>>'.
```

which is pleasantly mysterious, and has the side benefit of defending national security.

TIED SCALARS

We can use this feature to associate a tied scalar variable with an iterator. When the value of the scalar variable is examined, Perl gets control behind the scenes, kicks the iterator, and reports the next iterator value as the current value of the variable.

```
package Iterator2Scalar;

sub TIESCALAR {
  my ($package, $it) = @_;
```

```
  my $self = { It => $it };
  bless $self => $package;
}

sub FETCH {
  my ($self) = @_;
  NEXTVAL($self->{It});
}

sub STORE {
  require Carp;
  Carp::croak("Iterator is read-only");
}
```

Now we can use:

```
tie $nextfile, 'Iterator2Scalar', dir_walk($DIR);

while ($filename = $nextfile) {
  # do something with $filename
}
```

TIED FILEHANDLES

A tied scalar interface to an iterator produces a magical variable that encapsulates the iterator. This may be an intuitive interface in some cases, but it may also be peculiar. The user may be surprised to find that the variable's value changes spontaneously, or that the variable can't be assigned to.

Since Perl 5.004, filehandles have also been tie-able. A filehandle is often the most natural interface for a synthetic iterator. Since filehandles *are* examples of iterators, nobody will get any surprises when our filehandle behaves like an iterator or displays iterator-like limitations. People won't be surprised if a filehandle returns a different value each time it's read. They won't be surprised that they can't assign a value to it. They won't be surprised when a filehandle refuses to be rewound back to the beginning, since even ordinary filehandles don't always support that. They might be surprised by the handle's failure to support the `getc` operator, but more likely they won't even notice that it's missing.

The interface to tied filehandles is similar to that for tied scalars. Since filehandles aren't first-class variables in Perl, the caller passes an entire glob, and the `tie` operator ties the filehandle part of the glob:

```
tie *IT, 'Iterator2Handle', $iterator;
```

This calls the `Iterator2Handle::TIEHANDLE` constructor method, which is analogous to `TIESCALAR`:

```
package Iterator2Handle;

sub TIEHANDLE {
  my ($package, $iterator) = @_;
  my $self = { IT => $iterator };
  bless $self => $package;
}
```

When the user tries to read from the handle using the usual `<IT>` notation, Perl will call the `READLINE` method on the tied object:

```
sub READLINE {
  my $self = shift;
  return NEXTVAL($self->{IT});
}
```

To use the iterator, the user now does:

```
$some_value = <IT>;

# ... or ...

while ($nextval = <IT>) {
  # do something with $nextval
}
```

They can even use Perl's shortcut for reading a filehandle in a `while` loop:

```
while (<IT>) {
  # do something with $_
}
```

4.7 AN EXTENDED EXAMPLE: WEB SPIDERS

We'll now use the tools provided to build a replacement for Ave Wrigley's `WWW::SimpleRobot` module, which traverses a web site, invoking a callback for

each page. `SimpleRobot` provides two callback hooks, one of which is called for documents that can't be retrieved, the other for documents that can. It also supports options to specify whether the traversal will be breadth- or depth-first, and a regex that the URLs of the retrieved documents must match.

Our robot, which we'll call `Grasshopper`, will be at once simpler and more flexible. The robot will be embedded in an iterator, and the iterator will simply return URLs. If the user wants a callback invoked for each URL, they can add one with `imap`.

Our basic tools will be the `LWP::Simple` and `HTML::LinkExtor` modules. `LWP::Simple` provides a simple interface for building web clients. `HTML::LinkExtor` parses an HTML page and returns a list of all the links found on the page. Here is the first cut:

```
use HTML::LinkExtor;
use LWP::Simple;

sub traverse {
  my @queue = @_;
  my %seen;

  return Iterator {
    while (@queue) {
      my $url = shift @queue;
      $url =~ s/#.*$//;
      next if $seen{$url}++;

      my ($content_type) = head($url);
      if ($content_type =~ m{^text/html\b}) {
        my $html = get($url);
        push @queue, get_links($url, $html);
      }
      return $url;
    }
    return;     # exhausted
  }
}
```

The pattern here should be familiar; it's the same as the pattern we followed for `walk_tree`. The iterator maintains a queue of unvisited URLs, initialized with the list of URLs the user requests that it visit. Whenever the iterator is invoked, it gets the first item in the queue, reads it for more URLs, adds these URLs to the

end of the queue, and returns the URL. When the queue is empty, the iterator is exhausted.

Use of the iterator structure was essential. A simple recursive formulation just doesn't work. Recursive searches always do DFS, so a recursive robot will follow the first link on the first page to arrive at the second page, then follow the first link on the second page, then the first link on the third page, and so on, never returning to any page to process the other links there until it hits a dead end. Dead ends on the web are unusual, so the recursive robot goes wandering off into cyberspace and never comes back. Breadth-first search is almost always preferable for this application, but with a simple recursive function there's no way to get BFS.

There are a few web-specific features in the code. Some URLs may contain an anchor component, indicated by a `#` sign followed by the anchor name; this doesn't identify a document, so we discard it. (The specification for URLs guarantees that any other `#` sign in the URL will be represented as `%23`, so this substitution should be safe.) The iterator maintains a hash, `%seen`, which records whether or not it has visited a URL already; if it has, then the URL is skipped. It uses the `head` function, supplied by `LWP::Simple`, to find out whether the URL represents an HTML document, and if not it doesn't bother searching that document for links. In theory, `head` is supposed to indicate to the server that we do not want the entire document content; we want only meta-information, such as its content-type and length. In practice, the world is full of defective servers (and working servers running defective CGI programs) that send the entire document anyway. We've done the best we can here; if the server is going to send the document even though we asked it not to, there's nothing we can do about that.

If the document does turn out to be HTML, we use the `get` function, also supplied by `LWP::Simple`, to retrieve the content, which we then pass into `get_links`. `get_links` will parse the HTML and return a list of all the link URLs found in the document. Here it is:

```
sub get_links {
  my ($base, $html) = @_;
  my @links;
  my $more_links = sub {
    my ($tag, %attrs) = @_;
    push @links, values %attrs;
  };

  HTML::LinkExtor->new($more_links, $base)->parse($html);
  return @links;
}
```

The structure of `get_links` is a little peculiar, because `HTML::LinkExtor` uses an unfortunate interface. The `new` call constructs an `HTML::LinkExtor` object. We give the constructor a callback function, `$more_links`, and a base URL, `$base`. The callback function is invoked whenever the object locates an HTML element that contains a link. The link URLs themselves will be transformed into absolute URLs, interpreted relative to `$base`.

Parsing the document is triggered by the `->parse` method. We pass the HTML document as the argument. The object parses the document, invoking the callback each time it finds an HTML element that contains links. When we're done parsing the document, the object is no longer needed, so we never store it anywhere; we create it just long enough to call a single method on it.

The callback we supply to the `HTML::LinkExtor` object is called once for each link-bearing HTML element. The arguments are the tag of the element, and a sequence of attribute–value pairs for its link-bearing attributes. For example, if the element is:

```
<img border=0 src="pics/medium-sigils.gif" height=71 width=151
 align=right lowsrc="pics/small-sigils.gif" alt="$@%">
```

The arguments to the callback will be:

```
('IMG', 'SRC' => '.../pics/medium-sigils.gif',
        'LOWSRC' => '.../pics/small-sigils.gif', )
```

Our callback discards the tag, extracts the URLs, and pushes them into the `@links` array. When the parse is complete, we return `@links`.

4.7.1 Pursuing Only Interesting Links

The first cut of `Grasshopper` is almost useful. The one thing it's missing is a way to tell the robot not to pursue links to other web sites. We'll do that by inserting a callback filter function into `get_links()`. After `get_links()` extracts a list of links, it will invoke the user-supplied callback on this list. The callback will filter out the links that it doesn't want the robot to pursue, and return a list of the interesting links. The call to `traverse` will now look like this:

```
traverse($is_interesting_link, $url1, ...);
```

We need to make some minor changes to `traverse()`:

```
# Version with 'interesting links' callback
sub traverse {
  my $interesting_links = sub { @_ };
  $interesting_links = shift if ref $_[0] eq 'CODE';

  ...
        push @queue, $interesting_links->(get_links($url, $html));
  ...
}
```

Now we can ask it to traverse a single web site:

```
my $top = 'http://perl.plover.com/';
my $interesting = sub { grep /^\Q$top/o, @_ };

my $urls = traverse($interesting, $top);
```

This is already reasonably useful. Here's a program that copies every reachable file on a site:

```
use File::Basename;
while (my $url = NEXTVAL($urls)) {
  my $file = $url;
  $file =~ s/^\Q$top//o;
  my $dir = dirname($file);
  system('mkdir', '-p', $dir) == 0 or next;
  open F, ">", $file or next;
  print F get($url);
}
```

Here's a program to check whether any internal links on the site are bad:

```
while (my $url = NEXTVAL($urls)) {
  print "Bad link to: $url" unless head($url);
}
```

This last example exposes two obvious weaknesses in the current design. We can find out that `$url` is bad, but the iterator never tells us what page that bad URL appeared on, so we can't do anything about it. And the way we find out that `$url` is bad is rather silly. The iterator itself has just finished doing a `head` operation on this very URL, so we're repeating work that was just done a moment ago. The second of these is easier to repair. Since the information is available anyway,

we'll just have the iterator return it. In scalar context, it will return a URL; in list context, it will return a URL, a hash of header information, and the content, if available:

```
sub traverse {
      ...
      my (%head, $html);
      @head{qw(TYPE LENGTH LAST_MODIFIED EXPIRES SERVER)} = head($url);
      if ($head{TYPE} =~ m{^text/html\b}) {
        $html = get($url);
        push @queue, $interesting_links->(get_links($url,$html));
      }
      return wantarray ? ($url, \%head, $html) : $url;
      ...
}
```

The bad link detector now becomes:

```
while (my ($url, $head) = NEXTVAL($urls)) {
  print "Bad link to: $url\n" unless $head->{TYPE};
}
```

The site copier is:

```
use File::Basename;
while (my ($url, $head, $content) = NEXTVAL($urls)) {
  next unless $head->{TYPE};
  my $file = $url;
  $file =~ s/^\Q$top//o;
  my $dir = dirname($file);
  system('mkdir', '-p', $dir) == 0 or next;
  open F, ">", $file or next;
  $content = get($url) unless defined $content;
  print F $content;
}
```

4.7.2 Referring URLs

Including the referring URL is a little trickier, because by the time a URL shows up at the front of the queue, we've long since forgotten where we saw it. The solution

is to record the referring URLs in the queue also. Queue members will now be pairs of URLs. We will make the queue into an array of references to two-element arrays:

```
[ URL to investigate,
  URL of the page where we saw it (the 'referrer') ]
```

The `traverse` function is now:

```
sub traverse {
  my $interesting_links = sub { shift; @_ };
  $interesting_links = shift if ref $_[0] eq 'CODE';
  my @queue = map [$_, 'supplied by user'], @_;
  my %seen;

  return Iterator {
    while (@queue) {
      my ($url, $referrer) = @{shift @queue};
      $url =~ s/#.*$//;
      next if $seen{$url}++;

      my (%head, $html);
      @head{qw(TYPE LENGTH LAST_MODIFIED EXPIRES SERVER)} = head($url);
      if ($head{TYPE} =~ m{^text/html\b}) {
        my $html = get($url);
        push @queue,
          map [$_, $url],
            $interesting_links->($url, get_links($url, $html));
      }
      return wantarray ? ($url, \%head, $referrer, $html) : $url;
    }
    return;     #exhausted
  }
}
```

Instead of just copying the original URL list into the queue, we now annotate each one with a fake referrer. When we shift an item off the queue, we dismantle it into a URL and a referrer. The URL is treated as before. The referrer is passed to the `$interesting_links` callback, and each interesting link in the resulting list is annotated with its own referrer, the current URL, before being put into the queue. In list context, we return the referrer of each URL along with the other information about the document.

Our bad link detector is now :

```
my $top = 'http://perl.plover.com/'
my $interesting = sub { shift; grep /^\Q$top/o, @_ };

my $urls = traverse($interesting, $top);

while (my ($url, $head, $referrer) = NEXTVAL($urls)) {
  next if $head->{TYPE};
  print "Page '$referrer' has a bad link to '$url'\n";
}
```

Or, using `igrep` in the natural way:

```
my $top = 'http://perl.plover.com/';
my $interesting = sub { shift; grep /^\Q$top/o, @_ };

my $urls = igrep_l { not $_[1]{TYPE} } traverse($interesting, $top);

while (my ($url, $head, $referrer) = NEXTVAL($urls)) {
  print "Page '$referrer' has a bad link to '$url'\n";
}
```

The `igrep_l` here is a variation on `igrep` that filters a sequence of list values instead of a sequence of scalar values:

```
sub igrep_l (&$) {
  my ($is_interesting, $it) = @_;
  return Iterator {
    while (my @vals = NEXTVAL($it)) {
      return @vals if $is_interesting->(@vals);
    }
    return;
  }
}
```

Returning to the web spider, we might write:

```
while (my ($url, $head, $referrer) = NEXTVAL($urls)) {
  print "Page '$referrer' has a bad link to '$url'\n";
  print "Edit now? ";
```

```
  my $resp = <>;
  if ($resp =~ /^y/i) {
    system $ENV{EDITOR}, url_to_filename($referrer);
  } elsif ($resp =~ /^q/i) {
    last;
  }
}
```

Note that if the user enters `quit` to exit the loop and go on with the rest of the program, this *doesn't* foreclose the possibility that sometime later, they might continue from where they left off.

We now have a library good enough to check for bad offsite links as well as bad intrasite links:

```
my $top = 'http://perl.plover.com/';
my $interesting = sub { my $ref = shift;
                        $ref =~ /^\Q$top/o ? @_ : () };

my $urls = igrep_l { not $_[1]{TYPE} } traverse($interesting, $top);

while (my ($url, $head, $referrer) = NEXTVAL($urls)) {
  ...
}
```

The only thing that has changed is the `$interesting` callback that determines which links are worth pursuing. Formerly, links were worth pursuing if they *pointed to* a `http://perl.plover.com/` page. Now they're worth pursuing as long as they're *referred to* by some `http://perl.plover.com/` page. The checker will investigate pages at other sites, but it won't investigate the links on the pages at those sites.

This works well, but there's a more interesting solution available. If we think about how we're using the queue, we can see that the queue itself could be an iterator! We kick it periodically to produce another item for consideration by `traverse()`'s iterator, and then apply various transformations (`s/#.*$//`) and filters (`next if $seen{$url}++`) to the result. This is only going to get more complicated, so we'll probably get a win if we can leverage the tools we've developed for dealing with such structures:

```
sub traverse {
  my $interesting_link;
  $interesting_link = shift if ref $_[0] eq 'CODE';
```

```
my @queue = map [$_, 'supplied by user'], @_;
my %seen;
my $q_it = igrep { ! $seen{$_->[0]}++ }
             imap { $_->[0] =~ s/#.*$//; $_}
               Iterator { return shift(@queue) };

if ($interesting_link) {
  $q_it = igrep {$interesting_link->(@$_)} $q_it;
}

return imap {
    my ($url, $referrer) = @$_;
    my (%head, $html);

    @head{qw(TYPE LENGTH LAST_MODIFIED EXPIRES SERVER)} = head($url);
    if ($head{TYPE} =~ m{^text/html\b}) {
      $html = get($url);
      push @queue,
        map [$_, $url],
          get_links($url, $html);
    }
    return wantarray ? ($url, \%head, $referrer, $html) : $url;
  } $q_it;
}
```

The innermost iterator, the one that actually accesses the queue, shifts the first item off, as before, and returns it. Applications of `imap` and `igrep` trim fragment anchors off the URL and filter out URLs that have been seen already. Inside of the callbacks, `$_->[0]` is the URL and `$_->[1]` is the referrer. `$q_it` is the main queue iterator. `NEXTVAL($q_it)` will return the next URL/referrer pair that `traverse` should process.

If the user has supplied an `$interesting_link` function, we insert it into the queue iterator `$q_it`, where it will discard uninteresting links. If not, we ignore it completely, rather than inserting the identity function as a placeholder. Another change here is that because `$interesting_link` is filtering the output of the queue iterator, it processes only one URL at a time, rather than an entire list.

The `$interesting_link` function will receive the same implicit `$_` that the other segments of `$q_it` do, but for convenience we also pass the URL and referrer via the usual `@_` mechanism. Our earlier examples:

```
# Do not pursue links to other sites
my $interesting = sub { shift; grep /^\Q$top/o, @_ };
```

```
# Do not pursue links found on other sites
my $interesting = sub { my $ref = shift;
                        $ref =~ /^\Q$top/o ? @_ : () };
```

now become:

```
# Do not pursue links to other sites
my $interesting = sub { $_[0] =~ /^\Q$top/o };

# Do not pursue links found on other sites
my $interesting = sub { $_[1] =~ /^\Q$top/o };
```

The main `while(@queue) { ... shift @queue .. }` control is replaced with a call to `imap`, which maps the `head`, `get`, and queue updating behavior over `$q_it`.

Note that although we've added some code to support the new style, we've also deleted corresponding old code, so that both versions of the function are about the same length. This is to be expected, since the two functions are doing the same thing.

4.7.3 robots.txt

Let's add one more feature, one not supported by `WWW::SimpleRobot`. Some sites don't want to be walked by robots, or want to warn robots away from certain portions of their web space. For example, `/finance/admin/reports/` might actually be a CGI program, and asking for the document at `/finance/admin/reports/2000/12/24/08.html` would actually execute the program, which would compile the appropriate report and return it. Rather than storing 87,000 reports on the disk, on the off-chance that someone might want one, they are generated on demand. This is a good strategy when normal usage patterns are to request only a few reports per day.

A web robot that blunders into this part of the HTML space can waste a lot of network bandwidth and processing time on both ends of the connection, requesting thousands of reports. In the worst case, the report space might be infinite, and the robot will never get out.

To prevent this sort of accident, many sites advertise lists of the parts of their web space that robots should stay away from. Each site stores its robot policy in a file named`/robots.txt`. Good robots respect the policy laid out in this file.[3]

3 There are, unfortunately, very few good robots.

Here is a segment of `http://www.pathfinder.com/robots.txt`:

```
# Welcome to Pathfinder's robots.txt
#
...
#
# --------------------------

User-agent: *
Disallow: /cgi-bin/
Disallow: /event.ng/
Disallow: /money/money101/
Disallow: /offers/cp/
Disallow: /FoodWine/images/
# Disallow: /FoodWine/trecipes/
Disallow: /FoodWine/aspen/
...

User-agent: Mozilla
Disallow: /cgi-bin/Money/netc/story.cgi

User-agent: MSIECrawler
Disallow: /
```

Blank lines and lines beginning with `#` signs are comments and are ignored. `User-agent: *` marks the beginning of a section that applies to all robots. The `Disallow` lines are requests that robots not retrieve any documents whose URLs have any of the indicated prefixes. The sections at the bottom labelled `Mozilla` and `MSIECrawler` apply only to those browsers; other browsers can ignore them.

The Perl module `WWW::RobotRules` parses these files and returns an object that can be queried about the status of any URL:

```
my $rules = WWW::RobotRules->('Grasshopper/1.0');
```

`Grasshopper/1.0` is the name of our robot. This instructs the `WWW::RobotRules` object to pay attention to directives addressed to `Grasshopper/1.0`, and to ignore those addressed to `Mozilla`, `MSIECrawler`, and other browsers.

We add a set of rules to the object with the `->parse` method. It has two arguments: the contents of the `robots.txt` file, and the URL at which we found it. We can call `->parse` multiple times to add rules files for different sites.

To query the object about a URL, we use `$rules->allowed($url)`. This returns true if the rules allow us to visit the URL, false otherwise.

We will use `igrep()` to add a filter to the queue iterator `$q_it`. The filter will check each URL against the currently known set of robot rules and will discard it unless the rules allow it. Additionally, if the URL appears to refer to a site that hasn't been visited yet, the filter will attempt to load the `robots.txt` file from that site and add it to the current set of rules.

The filter callback will be manufactured by the following function:

```
use WWW::RobotRules;
use URI::URL;

sub make_robot_filter {
  my $agent = shift;
  my %seen_site;
  my $rules = WWW::RobotRules->new($agent);
  return sub {
    my $url = url(shift());
    return 1 unless $url->scheme eq 'http';
    unless ($seen_site{$url->netloc}++) {
      my $robots = $url->clone;
      $robots->path('/robots.txt');
      $robots->frag(undef);
      $rules->parse($robots, get($robots));
    }
    $rules->allowed($url)
  };
}
```

We can't simply use a single, named function, because the robot filter function needs to be able to capture private versions of the variables `$rules` and `%seen_site`, and named functions don't capture properly. We could have embedded the robot filter closure as a private function inside of `traverse()`, but I felt that `traverse()` was getting a little too long.

We're using the `URI::URL` module here, which provides convenience methods for parsing and constructing URLs. In the URL `http://perl.plover.com/perl.html#search`, `http` is the *scheme*, `perl.plover.com` is the *netloc*, `/perl.html` is the *path*, and `#search` is the *fragment*. `scheme`, `netloc`, `path`, and `frag` methods retrieve or set these sections of a URL. The `clone` method copies a URL object and returns a new object.

URLs for schemes other than `http` are always allowed by the filter, because other schemes don't have any mechanisms analogous to `robots.txt`. You could

make an argument that we should filter out `mailto` URLs and the like, but that would be more appropriately done by a different filter; this one is only about enforcing `robots.txt` rules.

If the URL is from a new site, as recorded in the private `%seen_site` hash, the filter constructs the URL for the `robots.txt` file and attempts to retrieve and parse it. It then consults the rules to decide whether the original URL will be discarded.

```
my $ROBOT_NAME = 'Grasshopper/1.0';

sub traverse {
  my $interesting_link;
  $interesting_link = shift if ref $_[0] eq 'CODE';
  my @queue = map [$_, 'supplied by user'], @_;
  my %seen;
  my $robot_filter = make_robot_filter($ROBOT_NAME);
  my $q_it = igrep {  ! $seen{$_->[0]}++ && $robot_filter->($_->[0]) }
              imap { $_->[0] =~ s/#.*$//; $_ }
                Iterator { return shift(@queue) };

  ...
}
```

4.7.4 Summary

The are only two major features of `WWW::SimpleRobot` that we've omitted. One is depth-first instead of breadth-first searching, which we've already seen is trivial to support; we just change `shift` to `pop` to turn the queue into a stack. With our iterator-structured queue, this is as simple as replacing:

```
                    Iterator { return shift(@queue) };
```

with:

```
      $depth_first ?  Iterator { return   pop(@queue) }
                   :  Iterator { return shift(@queue) };
```

The other feature is the `depth` feature, which allows the user to tell `WWW::SimpleRobot` how far to pursue chains of links. If `depth` is 5, then the robot will visit all the pages that are reachable by a path of five or fewer links, but no pages that can be reached only by paths of six or more links.

With a sufficiently ingenious `$interesting_links` callback, we can emulate this feature in the current system. But we might want to add it to the `traverse()` function for convenience. This is also only a small change: add the link depth of each URL to the queue items. It will then be passed automatically to the `$interesting_links` callback, which can cut off deep searches by saying:

```
return unless $_->[2] < $max_depth;
```

These are the missing features. On the other hand, `Grasshopper` supports `robots.txt`, a major benefit. It also has the feature that it can be incorporated into a larger program as an auxiliary component. `WWW::SimpleRobot` will tend to take over the behavior of any program it's part of, because once you call the `WWW::SimpleRobot::traverse` function, you won't get control back until it has traversed the entire site, which could be a very long time. `Grasshopper` never takes control for longer than it takes to retrieve one page (plus possibly the `robots.txt` file for a new site), and if the program wants to do something else afterwards, it can pick up where it left off.

I don't want to make too much of the operational differences between these two modules. They both have serious defects stemming from the design of `LWP::Simple`. But I think there's one other difference that's worth pointing out: `Grasshopper` requires less than half as much code; one-third if you don't count the code required to support `robots.txt` handling, which `WWW::SimpleRobot` doesn't do.

Where did this benefit come from? The queue structure itself didn't gain us much, because `WWW::SimpleRobot` is using the same queue technique that we are. The object-oriented style of `WWW::SimpleRobot` imposes some overhead; with the functional approach there are no classes to declare. Some of the extra code in `WWW::SimpleRobot` is to support diagnostics, which shouldn't count, because I omitted diagnostics from the iterator module. (On the other hand, a 49-line module probably doesn't need many diagnostics.)

Probably the greatest contributor to overhead is option checking. With the functional approach, the module hardly supports any options directly. `WWW::SimpleRobot` has all its options on the inside. To support a new option, we have to attach it inside the module. `Grasshopper` is a module that has been turned inside out, all of its useful hooks are exposed to the caller, who can apply whatever options they want afterwards via functions like `igrep`.