CHAPTER 7

HIGHER-ORDER FUNCTIONS AND CURRYING

Our `memoize()` function of Chapter 3 was a "function factory," building stub functions that served as replacements for other functions. The technique of using functions to build other functions is extremely powerful. In this chapter, we'll look at a technique called *currying*, which transforms an ordinary function into a function factory for manufacturing more functions, and at other techniques for transforming one function into another.

A higher-order function is a function that operates on other functions instead of on data values. Some of these take data arguments and manufacture functions to order; others, like the `imap()` function of Chapter 4, transform one function into another one.

7.1 CURRYING

We have seen several times how to use callbacks to parametrize the behavior of a function so that it can serve many purposes. For example, in Section 1.5 we saw how a generic directory-walking function could be used to print a list of dangling symbolic links, to return a list of interesting files, or to copy an entire directory.

Callbacks are a way to make functions more general by supplying other functions to them as arguments. We saw how to write functions that used closures to generate other functions as return values. The *currying* technique we'll see combines closures and callbacks, turning an ordinary function into a factory that manufactures functions on demand.

Recall our `walk_html()` function from Chapter 1. Its arguments were an HTML tree and a pair of callbacks, one to handle plain text and one to handle tagged text. We found a way to use this to extract all the text that was enclosed in `<h1>` tags:

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

sub promote_if_h1tag {
  my $element = shift;
  if ($element->{_tag} eq 'h1') {
    return ['KEEPER', join '', map {$_->[1]} @_];
  } else {
    return @_;
  }
}

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

We then observed that it would make sense to abstract the `<h1>` out of `promote_if_h1tag()`, to make it more general:

```
sub promote_if {
  my $is_interesting = shift;
  my $element = shift;
  if ($is_interesting->($element->{_tag}) {
    return ['keeper', join '', map {$_->[1]} @_];
  } else {
```

```
    return @_;
  }
}

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

The second callback in `walk_html()` is rather peculiar. It's an anonymous function that we manufactured solely to call `promote_if()` with the right arguments. The previous version of the code was tidier. What we need is a way to get `promote_if()` to *manufacture* the `promote_if_h1tag()` function we need. This seems like it should be possible, because `promote_if()` already knows how to perform the task that we want `promote_if_h1tag()` to perform. All that we need to do is to have `promote_if()` wrap up that behavior into a new function:

CODE LIBRARY
promote_if_curr

```
sub promote_if {
  my $is_interesting = shift;
  return sub {
    my $element = shift;
    if ($is_interesting->($element->{_tag}) {
      return ['keeper', join '', map {$_->[1]} @_];
    } else {
      return @_;
    }
  }
}
```

Instead of accepting both arguments right away, `promote_if()` now gets the `$is_interesting` callback only, and manufactures a new function that, given an HTML element, promotes it if it's considered interesting. Making this change to `promote_if()`, to turn it from a function of two arguments into a function of one argument that returns a function of one argument, is called *currying* it, and the version of `promote_if()` immediately above is the *curried* version of `promote_if()`.[1]

1 Currying is so-named because it was popularized by Haskell B. Curry in 1930, although it had been discovered by Gottlob Frege in 1893 and rediscovered by Moses Schönfinkel in 1924.

The happy outcome is that the call to `walk_html()` is now much simpler:

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

Once you get used to the idea of currying, you start to see opportunities to do it all over. Recall our functions from Chapter 6 for adding and multiplying two streams together element-by-element: `add2()` and `mul2()`:

```
sub add2 {
  my ($s, $t) = @_;
  return unless $s && $t;
  node(head($s) + head($t),
       promise { add2(tail($s), tail($t)) });
}
sub mul2 {
  my ($s, $t) = @_;
  return unless $s && $t;
  node(head($s) * head($t),
       promise { mul2(tail($s), tail($t)) });
}
```

These functions are almost identical. We saw in Chapter 1 that two functions with similar code can often be combined into a single function that accepts a callback parameter. In this case, the callback, `$op`, specifies the operation to use to combine `head($s)` and `head($t)`:

```
sub combine2 {
  my ($s, $t, $op) = @_;
  return unless $s && $t;
  node($op->(head($s), head($t)),
       promise { combine2(tail($s), tail($t), $op) });

}
```

Now we can build `add2()` and `mul2()` from `combine2()`:

```
sub add2 { combine2(@_, sub { $_[0] + $_[1] }) }
sub mul2 { combine2(@_, sub { $_[0] * $_[1] }) }
```

Since a major use of `combine2()` is to manufacture such functions, it would be more convenient for `combine2()` to do what we wanted in the first place. We can turn `combine2()` into a factory that manufactures stream-combining functions by currying it:

CODE LIBRARY
combine2

```
sub combine2 {
   my $op = shift;
   return sub {
     my ($s, $t) = @_;
     return unless $s && $t;
     node($op->(head($s), head($t)),
          promise { combine2($op)->(tail($s), tail($t))});
   };
}
```

Now we have simply:

```
$add2 = combine2(sub { $_[0] + $_[1] });
$mul2 = combine2(sub { $_[0] * $_[1] });
```

This may also be fractionally more efficient, since we won't have to do an extra function call every time we call `add2()` or `mul2()`. `add2()` is the function to add the two streams, rather than a function that re-invokes `combine2()` in a way that adds two streams.

If we want these functions to stick around, we can give them names, as we just did; alternatively, we can use them anonymously:

```
my $catstrs = combine2(sub { "$_[0]$_[1]" })->($s, $t);
```

Instead of the `scale()` function we saw earlier, we might prefer this curried version:

```
sub scale {
    my $s = shift;
    return sub {
      my $c = shift;
      return if $c == 0;
      transform { $_[0] * $c } $s;
    }
  }
```

`scale()` is now a function factory. Instead of taking a stream and a number and returning a new stream, it takes a stream and manufactures a function that produces new streams. `$scale_s = scale($s)` returns a function for scaling `$s`; given a numeric argument, say `$n`, `$scale_s` produces a stream that has the elements of `$s` scaled by `$n`. For example, `$scale_s->(2)` returns a stream whose every element is twice `$s`'s, and `$scale_s->(3)` returns a stream whose every element is three times `$s`'s. If we're planning to scale the same stream by several different factors, it might make sense to have a single scale function to generate all the outputs.

Depending on how we're using it, we might have preferred to curry the function arguments in the other order:

CODE LIBRARY
scale

```
sub scale {
  my $c = shift;
  return sub {
    my $s = shift;
    transform { $_[0] * $c } $s;
  }
}
```

Now `scale()` is a factory for manufacturing scaling functions. `scale(2)` returns a function that takes any stream and doubles it; `scale(3)` returns a function that takes any stream and triples it. We could write `$double = scale(2)` and then use `$double->($s)` to double `$s`, or `scale(2)->($s)` to double `$s`.

If you don't like the extra arrows in `$double->($s)` you can get rid of them by using Perl's glob feature, as we did in Chapter 3:

```
*double = scale(2);
$s2 = double($s);
```

Similarly, in Chapter 6, we defined a `slope()` function that returned the slope of some other function at a particular point:

```
sub slope {
  my ($f, $x) = @_;
  my $e = 0.00000095367431640625;
  ($f->($x+$e) - $f->($x-$e)) / (2*$e);
}
```

We could make this more flexible by currying the `$x` argument:

CODE LIBRARY
slope0

```
sub slope {
  my $f = shift;
  my $e = 0.00000095367431640625;
  return sub {
    my $x = shift;
    ($f->($x+$e) - $f->($x-$e)) / (2*$e);
  };
}
```

`slope()` now takes a function and returns its derivative function! By evaluating the derivative function at a particular point, we compute the slope at that point.

If we like, we can use Perl's polymorphism to put both behaviors into the same function:

CODE LIBRARY
slope

```
sub slope {
  my $f = shift;
  my $e = 0.00000095367431640625;
  my $d = sub {
    my ($x) = shift;
    ($f->($x+$e) - $f->($x-$e)) / (2*$e);
  };
  return @_ ? $d->(shift) : $d;
}
```

Now we can call `slope($f, $x)` as before, to compute the slope of `$f` at the point `$x`, or we can call `slope($f)` and get back the derivative function of `$f`.

Currying can be a good habit to get into. Earlier, we wrote:

```
sub iterate_function {
  my ($f, $x) = @_;
  my $s;
  $s = node($x, promise { &transform($f, $s) });
}
```

But it's almost as easy to write it this way instead:

CODE LIBRARY
iterate_function

```
sub iterate_function {
  my $f = shift;
```

```
    return sub {
      my $x = shift;
      my $s;
      $s = node($x, promise { &transform($f, $s) });
    };
  }
```

It requires hardly any extra thought to do it this way, and the payoff is substantially increased functionality. We now have a function that manufactures stream-building functions to order. We could construct `upfrom()` as a special case of `iterate_function()`; for example:

```
*upfrom = iterate_function(sub { $_[0] + 1 });
```

Or similarly, our earlier example of `pow2_from()`:

```
*pow2_from = iterate_function(sub { $_[0] * 2 });
```

One final lexical point about currying: When currying a recursive function, it's often possible to get a small time and memory performance improvement by tightening up the recursion. For example, consider `combine2()` again:

```
sub combine2 {
  my $op = shift;
  return sub {
    my ($s, $t) = @_;
    return unless $s && $t;
    node($op->(head($s), head($t)),
         promise { combine2($op)->(tail($s), tail($t)) });
  };
}
```

`combine2($op)` will return the same result function every time. So we should be able to get a speed-up by caching its value and using the cached value in the promise instead of repeatedly calling `combine2($op)`. Moreover, `combine2($op)` is precisely the value that `combine2()` is about to return anyway. So we can change this to:

CODE LIBRARY
combine2-shorter

```
sub combine2 {
  my $op = shift;
```

```
  my $r;
  $r = sub {
    my ($s, $t) = @_;
    return unless $s && $t;
    node($op->(head($s), head($t)),
         promise { $r->(tail($s), tail($t)) });
  };
}
```

Now the promise no longer needs to call `combine2()`; we've cached the value that `combine2()` is about to return by storing it in `$r`, and the promise can call `$r` directly. The code is also easier to understand this way: Now the promise says explicitly that the function will be calling itself on the tails of the two streams.

These curried functions are examples of *higher-order functions*. Ordinary functions operate on values: You put some values in, and you get some other values out. Higher-order functions are functions that operate on other functions: You put some functions in, and you get some other functions out. For example, in `combine2()` we put in a function to operate on two scalars and we got out an analogous function to operate on two streams.

7.2 COMMON HIGHER-ORDER FUNCTIONS

Probably the two most fundamental higher-order functions for any list or other kind of sequence are analogs of `map()` and `grep()`. `map()` and `grep()` are higher-order functions because each of them takes an argument that is itself another function. We've already seen versions of `map()` and `grep()` for iterators and streams. Perl's standard `map()` and `grep()` each take a function and a list and return a new list; for example:

```
map { $_ * 2 } (1..5);           # returns 2, 4, 6, 8, 10
grep { $_ % 2 == 0 } (1..10);    # returns 2, 4, 6, 8, 10
```

Often it's more convenient to have curried versions of these functions:

CODE LIBRARY
cmap

```
sub cmap (&) {
  my $f = shift;
  my $r = sub {
```

```
    my @result;
    for (@_) {
      push @result, $f->($_);
    }
    @result;
  };
  return $r;
}
```

CODE LIBRARY
cgrep

```
sub cgrep (&) {
  my $f = shift;
  my $r = sub {
    my @result;
    for (@_) {
      push @result, $_ if $f->($_);
    }
    @result;
  };
  return $r;
}
```

These functions should be called like this:

```
$double = cmap { $_ * 2 };
$find_slashdot = cgrep { $_->{referer} =~ /slashdot/i };
```

After which `$double->(1..5)` returns (2, 4, 6, 8, 10) and `$find_slashdot ->(weblog())` returns the web log records that represent referrals from Slashdot.

It may be tempting to try to make `cmap()` and `cgrep()` polymorphic, as we did with `slope()` (I was tempted, anyway.):

```
sub cmap (&;@) {
  my $f = shift;
  my $r = sub {
    my @result;
    for (@_) {
      push @result, $f->($_);
    }
    @result;
  };
```

```
  return @_ ? $r->(@_) : $r;
}
```

Then we would also be able to use `cmap()` and `cgrep()` like regular `map()` and `grep()`:

```
@doubles = cmap { $_ * 2 } (1..5);
@evens = cgrep { $_ % 2 == 0 } (1..10);
```

Unfortunately, this apparently happy notation hides an evil surprise:

```
@doubles = cmap { $_ * 2 } @some_array;
```

If `@some_array` is empty, `@doubles` is assigned a reference to a doubling function.

7.2.1 Automatic Currying

We've written the same code several times to implement curried functions:

```
sub some_curried_function {
  my $first_arg = shift;
  my $r = sub {
    ...
  };
  return @_ ? $r->(@_) : $r;
}
```

(Possibly with the polymorphism trick omitted from the final line.)

As usual, once we recognize this pattern, we should see if it makes sense to abstract it into a function:

CODE LIBRARY
Curry.pm

```
package Curry;
use base 'Exporter';
@EXPORT = ('curry');
@EXPORT_OK = qw(curry_listfunc curry_n);

sub curry {
  my $f = shift;
  return sub {
    my $first_arg = shift;
    my $r = sub { $f->($first_arg, @_) };
```

```
    return @_ ? $r->(@_) : $r;
  };
}

sub curry_listfunc {
  my $f = shift;
  return sub {
    my $first_arg = shift;
    return sub { $f->($first_arg, @_) };
  };
}

1;
```

`curry()` takes any function and returns a curried version of that function. For example, consider the `imap()` function from Chapter 4:

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

`imap()` is analogous to `map()`, but operates on iterators rather than on lists. We might use it like this:

```
my $doubles_iterator = imap { $_[0] * 2 } $it;
```

If we end up doubling a lot of iterators, we have to repeat the `{$_[0] * 2}` part:

```
my $doubles_a = imap { $_[0] * 2 } $it_a;
my $doubles_b = imap { $_[0] * 2 } $it_b;
my $doubles_c = imap { $_[0] * 2 } $it_c;
```

We might wish we had a single, special-purpose function for doubling every element of an iterator, so that we could write instead:

```
my $doubles_a = double $it_a;
my $doubles_b = double $it_b;
my $doubles_c = double $it_c;
```

or even:

```
my ($doubles_a, $doubles_b, $doubles_c)
  = map double($_), $it_a, $it_b, $it_c;
```

If we had written `imap()` in a curried style, we could have done:

```
*double = imap { $_[0] * 2 };
```

but we didn't, so we can't. But that's no problem, because `curry()` will manufacture a curried version of `imap()` on the fly:

```
*double = curry(\&imap)->(sub { $_[0] * 2 });
```

Since the curried `imap()` function came in handy once, perhaps we should keep it around in case we need it again:

```
*c_imap = curry(\&imap);
```

Then to manufacture `double()` we do:

```
*double = c_imap(sub { $_[0] * 2 });
```

7.2.2 Prototypes

The only drawback of this approach is that we lose `imap()`'s pretty calling syntax, which is enabled by the `(&@)` prototype at compile time. We can get it back, although the results are somewhat peculiar. First, we modify `curry()` so that the function it manufactures has the appropriate prototype:

```
sub curry {
  my $f = shift;
  return sub (&;@) {
    my $first_arg = shift;
    my $r = sub { $f->($first_arg, @_) };
    return @_ ? $r->(@_) : $r;
  };
}
```

Then we `call curry()` at compile time instead of at run time:

```
BEGIN { *c_imap = curry (\&imap); }
```

Now we can say:

```
*double = c_imap { $_[0] * 2 };
```

and we can still use `c_imap()` in place of regular `imap()`:

```
$doubles_a = c_imap { $_[0] * 2 } $it_a;
```

PROTOTYPE PROBLEMS

The problem with this technique is that the prototype must be hardwired into `curry()`, so now it will generate *only* curried functions with the prototype `(&;@)`. This isn't a problem for functions like `c_imap()` or `c_grep()`, which would have had that prototype anyway. But that prototype is inappropriate for the curried version of the `scale()` function from Chapter 6. The uncurried version was:

```
sub scale {
  my ($s, $c) = @_;
  $s->transform(sub { $_[0]*$c });
}
```

`curry(\&scale)` returns a function that behaves like this:

```
sub {
    my $s = shift;
    my $r = sub { scale($s, @_) };
    return @_ ? $r->(@_) : $r;
}
```

The internals of this function are correct, and it will work just fine, as long as it *doesn't have* a `(&;@)` prototype. Such a prototype would be inappropriate, since the function is expecting to get one or two scalar arguments. The correct prototype would be `($;$)`. But if we did:

```
BEGIN { *c_scale = curry(\&scale) }
```

then the resulting `c_scale()` function wouldn't work, because it would have a `(&;@)` prototype when we expected to call it as though it had a `($;$)` prototype.

We want to call it in one of these two ways:

```
my $double = c_scale(2);
my $doubled_it = c_scale(2, $it);
```

but because `c_scale()` would have a prototype of `(&;@)`, these both would be syntax errors, yielding:

```
Type of arg 1 to main::c_scale must be block or sub {} (not
constant item)...
```

This isn't a show-stopper. This works:

```
*c_scale = curry(\&scale);
my $double = c_scale(2);
my $doubled_it = c_scale(2, $it);
```

Here the call to `c_scale()` is compiled, with no prototype, before `*c_scale` is assigned to; the call to `curry()` that sets up the bad prototype occurs too late to foul up our attempt to (correctly) call `c_scale()`.

But now we have a somewhat confusing situation. Our `curry()` function creates curried functions with `(&;@)` prototypes, and these prototypes may be inappropriate. But the prototypes are inoperative unless `curry()` is called in a `BEGIN` block. To add to the confusion, this doesn't work:

```
*c_scale = curry(\&scale);
my $double = eval 'c_scale(2)';
```

because, once again, the call to `c_scale()` has been compiled after the prototype was set up by `curry()`.

There isn't really any easy way to fix this. The obvious thing to do is to tell `curry()` what prototype we desire by supplying it with an optional parameter:

```
# Doesn't really work
sub curry {
  my $f = shift;
  my $PROTOTYPE = shift;
  return sub ($PROTOTYPE) {
     my $first_arg = shift;
     my $r = sub { $f->($first_arg, @_) };
     return @_ ? $r->(@_) : $r;
  };
}
```

Unfortunately, this is illegal; `($PROTOTYPE)` *does not* indicate that the desired prototype is stored in `$PROTOTYPE`. Perl 5.8.1 provides a `Scalar::Util::set_prototype` function to set the prototype of a particular function:

CODE LIBRARY
curry-set_proto

```
# Doesn't work before 5.8.1
use Scalar::Util 'set_prototype';

sub curry {
  my $f = shift;
  my $PROTOTYPE = shift;
  set_prototype(sub {
    my $first_arg = shift;
    my $r = sub { $f->($first_arg, @_) };
    return @_ ? $r->(@_) : $r;
  }, $PROTOTYPE);
}
```

If you don't have 5.8.1 yet, the only way to dynamically specify the prototype of a function is to use string `eval`:

CODE LIBRARY
curry_eval

```
sub curry {
  my $f = shift;
  my $PROTOTYPE = shift;
  $PROTOTYPE = "($PROTOTYPE)" if defined $PROTOTYPE;
  my $CODE = q{sub PROTOTYPE {
                my $first_arg = shift;
                my $r = sub { $f->($first_arg, @_) };
                return @_ ? $r->((@_) : $r;
               }};
  $CODE =~ s/PROTOTYPE/$PROTOTYPE/;
  eval $CODE;
}
```

7.2.3 More Currying

We can extend the idea of `curry()` and build a function that generates a generic curried version of another function:

```
sub curry_n {
  my $N = shift;
  my $f = shift;
```

```
    my $c;
    $c = sub {
      if (@_ >= $N) { $f->(@_) }
      else {
        my @a = @_;
        curry_n($N-@a, sub { $f->(@a, @_) });
      }
    };
  }
```

`curry_n()` takes two arguments: a number *N*, and a function *f*, which expects at least *N* arguments. The result is a new function, *c*, which does the same thing *f* does, but which accepts curried arguments. If *c* is called with *N* or more arguments, it just passes them on to *f* and returns the result. If there are fewer than *N* arguments, *c* generates a new function that remembers the arguments that were passed; if this new function is called with the remaining arguments, both old and new arguments are given to *f*. For example:

```
*add = curry_n(2, sub { $_[0] + $_[1] });
```

And now we can call:

```
add(2, 3);      # Returns 5
```

or:

```
*increment = add(1);
increment(8);   # return 9
```

or perhaps more realistically:

```
*csubstr = curry_n(3, sub { defined $_[3] ?
                                substr($_[0], $_[1], $_[2], $_[3]) :
                                substr($_[0], $_[1], $_[2]) });
```

Then we can use any of:

```
$target = "I like pie";

# Just like regular substr

$ss = csubstr($target, $start, $length);
csubstr($target, $start, $length, $replacement);
```

```
# Not just like regular substr

# This '$part' function gets two arguments: a start position
# and a length; it returns the appropriate part of $target.

$part = csubstr($target);
my $ss = $part->($start, $length);

# This function gets an argument N and returns that many characters
# from the beginning of $target.

$first_N_chars = csubstr($target, 0);
my $prefix_3 = $first_N_chars->(3);     # "I l"
my $prefix_7 = $first_N_chars->(7);     # "I like "
```

7.2.4 Yet More Currying

Many of the functions we saw earlier in the book would benefit from currying. For example, `dir_walk()` from Chapter 1:

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

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

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

Here we specify a top directory and two callback functions. But the callback functions are constant through any call to `dir_walk()`, and we might like to

specify them in advance, because we might know them well before we know which directories we want to search. The conversion is easy:

CODE LIBRARY
dir-walk-curried

```
sub dir_walk {
  unshift @_, undef if @_ < 3;
  my ($top, $filefunc, $dirfunc) = @_;

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

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

We can still call `dir_walk($top, $filefunc, $dirfunc)` and get the same result, or we can omit the `$top` argument (or pass `undef`) and get back a specialized file-walking function. As a minor added bonus, the recursive call will be fractionally more efficient because the callback arguments don't need to be explicitly passed.

7.3 reduce() AND combine()

The standard Perl `List::Util` module provides several commonly requested functions that are not built-in to Perl. These include `max()` and `min()` functions,

which respectively return the largest and smallest numbers in their argument lists, `maxstr()` and `minstr()`, which are the analogous functions for strings; and `sum()`, which returns the sum of the numbers in a list.

If we write sample code for these five functions, we'll see the similarity immediately:

```
sub max { my $max = shift;
          for (@_) { $max = $_ > $max ? $_ : $max }
          return $max;
        }

sub min { my $min = shift;
          for (@_) { $min = $_ < $min ? $_ : $min }
          return $min;
        }

sub maxstr { my $max = shift;
             for (@_) { $max = $_ gt $max ? $_ : $max }
             return $max;
           }

sub minstr { my $min = shift;
             for (@_) { $min = $_ lt $min ? $_ : $min }
             return $min;
           }

sub sum { my $sum = shift;
          for (@_) { $sum = $sum + $_ }
          return $sum;
        }
```

Generalizing this gives us the `reduce()` function that is also provided by `List::Util`:

```
sub reduce { my $code = shift;
             my $val = shift;
             for (@_) { $val = $code->($val, $_) }
             return $val;
           }
```

(`List::Util::reduce` is actually written in C for speed, but what it does is equivalent to this Perl code.) The idea is that the function will scan the list one element

at a time, accumulating a "total" of some sort. We provide a function (`$code`) that says how to compute the new total, given the old total (first argument) and the current element (second argument). If our goal is just to add up all the list elements, then we compute the total at each stage by adding the previous total to the current element:

```
reduce(sub { $_[0] + $_[1] }, @VALUES) == sum(@VALUES)
```

If our goal is to find the maximum element, then the "total" is actually the maximum so far. Then we compute the total at each stage by taking whichever of the current maximum and the current element is larger:

```
reduce(sub { $_[0] > $_[1] ? $_[0] : $_[1] }, @VALUES) == max(@VALUES)
```

The `reduce()` function provided by `List::Util` is easier to call than the preceding one. It places the total-so-far in `$a` and the current list element into `$b` before invoking the callback, so that we can write:

```
reduce(sub { $a + $b }, @VALUES)
reduce(sub { $a > $b ? $a : $b }, @VALUES)
```

We saw how to make this change back in Section 4.4, when we arranged to have `imap()`'s callback invoked with the current iterator value in `$_` in addition to `$_[0]`; this allowed it to have a more `map()`-like calling syntax. We can arrange `reduce()` similarly:

```
sub reduce (&@) {
  my $code = shift;
  my $val = shift;
  for (@_) {
    local ($a, $b) = ($val, $_);
    $val = $code->($val, $_)
  }
  return $val;
}
```

Here we're using the global variables `$a` and `$b` to pass the total and the current list element. Use of global variables normally causes a compile-time failure under `strict 'vars'`, but there is a special exemption for the variables `$a` and `$b`. The exemption is there to allow usages just like this one, and in particular to support the analogous feature of Perl's built-in `sort()` function. The `List::Util` version of `reduce()` already has this feature built in.

If we curry the `reduce()` function, we can use it to *manufacture* functions like `sum()` and `max()`:

```
BEGIN {
  *reduce = curry(\&List::Util::reduce);
  *sum = reduce { $a + $b };
  *max = reduce { $a > $b ? $a : $b };
}
```

This version of `reduce()` isn't quite as general as it could be. All the functions manufactured by `reduce()` have one thing in common: Given an empty list of arguments, they always return `undef`. For `max()` and `min()` this may be appropriate, but for `sum()` it's wrong; the sum of an empty list should be taken to be 0. (The `sum()` function provided by `List::Util` also has this defect.) This small defect masks a larger one: When the argument list is nonempty, our version of `reduce()` assumes that the total should be initialized to the first data item. This happens to work for `sum()` and `max()`, but it isn't appropriate for all functions. `reduce` can be made much more general if we drop this assumption. As a trivial example, suppose we want a function to produce the length of a list. This is *almost* what we want:

```
reduce { $a + 1 };
```

But it only produces the correct length when given a list whose first element is 1, since otherwise `$val` is incorrectly initialized. A more general version of `reduce()` accepts an explicit parameter to say what value should be returned for an empty list:

```
sub reduce (&$@) {
  my $code = shift;
  my $val = shift;
  for (@_) {
    local ($a, $b) = ($val, $_);
    $val = $code->($val, $_)
  }
  return $val;
}
```

A version with optional currying is:

CODE LIBRARY
reduce

```
sub reduce (&;$@) {
  my $code = shift;
```

```
  my $f = sub {
    my $base_val = shift;
    my $g = sub {
      my $val = $base_val;
      for (@_) {
        local ($a, $b) = ($val, $_);
        $val = $code->($val, $_);
      }
      return $val;
    };
    @_ ? $g->(@_) : $g;
  };
  @_ ? $f->(@_) : $f;
}
```

The list-length function is now:

```
*listlength = reduce { $a + 1 } 0;
```

where the `0` here is the correct result for an empty list. Similarly,

```
*product = reduce { $a * $b } 1;
```

is a function that multiplies all the elements in a list of numbers. We can even use `reduce()` to compute both at the same time:

```
*length_and_product = reduce { [$a->[0]+1, $a->[1]*$b] } [0, 1];
```

This makes only one pass over the list to compute both the length and the product. For an empty list, the result is `[0, 1]`, and for a list with one element *x*, the result is `[1, x]`. `List::Util::reduce()` can manufacture only functions that return `undef` for the empty list, and that return the first list element for a single-element list. The `length_and_product()` function can't be generated by `List::Util::reduce()` because it doesn't have these properties.

A properly general version of `reduce()` gets an additional argument that says that the function should return when given an empty list as its argument. In the programming literature, the properly general version of `reduce()` is more typically called `fold()`:

```
sub fold {
  my $f = shift;
```

```
    my $fold;
    $fold = sub {
      my $x = shift;
      sub {
        return $x unless @_;
        my $first = shift;
        $fold->($f->($x, $first), @_)
      }
    }
  }
```

Eliminating the recursion yields:

CODE LIBRARY
fold

```
sub fold {
  my $f = shift;
  sub {
    my $x = shift;
    sub {
      my $r = $x;
      while (@_) {
        $r = $f->($r, shift());
      }
      return $r;
    }
  }
}
```

7.3.1 Boolean Operators

In Section 4.3 we saw a system that would search backwards through a log file looking for records that matched a simple query. To extend this into a useful database system, we need to be able to combine simple queries into more complex ones.

Let's suppose that `$a` and `$b` are iterators that will produce data items that match queries *A* and *B*, respectively. How can we manufacture an iterator that matches the query *A* `||` *B*?

One way we could do this is to interleave the elements of `$a` and `$b`:

CODE LIBRARY
interleave

```
sub interleave {
  my ($a, $b) = @_;
```

```
  return sub {
    my $next = $a->();
    unless (defined $next) {
      $a = $b;
      $next = $a->();
    }
    ($a, $b) = ($b, $a);
    $next;
  }
}
```

But this has the drawback that if the record sets produced by `$a` and `$b` happen to overlap, the interleaved outputs will include some records more than once.

We can do better if we suppose that the records will be produced in some sort of canonical order. This assumption isn't unreasonable. Typically, a database will have a natural order dictated by the physical layout of the information on the disk and will always produce records in this natural order, at least until the data is modified. For example, our program for searching the web log file always produces matching records in the order they appear in the file. Even DBM files, which don't appear to keep records in any particular order, have a natural order; this is the order in which the records will be generated by the `each()` function.

Supposing that `$a` and `$b` will produce records in the same order, we can perform an "or" operation as follows:

CODE LIBRARY
Iterator_Logic.pm

```
package Iterator_Logic;
use base 'Exporter';
@EXPORT = qw(i_or_ i_or i_and_ i_and i_without_ i_without);

sub i_or_ {
  my ($cmp, $a, $b) = @_;
  my ($av, $bv) = ($a->(), $b->());
  return sub {
    my $rv;
    if (! defined $av && ! defined $bv) { return }
    elsif (! defined $av) { $rv = $bv; $bv = $b->() }
    elsif (! defined $bv) { $rv = $av; $av = $a->() }
    else {
      my $d = $cmp->($av, $bv);
      if    ($d < 0) { $rv = $av; $av = $a->() }
      elsif ($d > 0) { $rv = $bv; $bv = $b->() }
      else           { $rv = $av; $av = $a->(); $bv = $b->() }
    }
```

```
    return $rv;
  }
}

use Curry;
BEGIN { *i_or = curry(\&i_or_) }
```

`i_or_()` gets a comparator function, `$cmp`, which defines the canonical order, and two iterators, `$a` and `$b`. It returns a new iterator that returns the next record from either `$a` or `$b` in the canonical order. If `$a` and `$b` both produce the same record, the duplicate is discarded. It begins by kicking `$a` and `$b` to obtain the next record from each. If either is exhausted, it returns the record from the other; if both are exhausted, it returns `undef` to indicate that there are no more records. `$rv` holds the record that is to be the return value.

If both input iterators produce records, the new iterator compares the records to see which should come out first. If the comparator returns zero, it means the two records are the same, and only one of them should be emitted. `$rv` is assigned one of the two records, as appropriate, and then one or both of the iterators is kicked to produce new records for the next call.

The logic is very similar to the `merge()` function of Section 6.4. In fact, `merge()` is the stream analog of the "or" operator.

`i_or()` is a curried version of `i_or_()`, called like this:

```
BEGIN { *numeric_or = i_or { $_[0] <=> $_[1] };
        *alphabetic_or = i_or { $_[0] cmp $_[1] };
      }

     $event_times =  numeric_or($access_request_times,
                     numeric_or($report_request_times,
                                $server_start_times));
```

"And" is similar:

```
sub i_and_ {
  my ($cmp, $a, $b) = @_;
  my ($av, $bv) = ($a->(), $b->());
  return sub {
    my $d;
    until (! defined $av || ! defined $bv ||
           ($d = $cmp->($av, $bv)) == 0) {
      if ($d < 0) { $av = $a->() }
      else        { $bv = $b->() }
    }
```

```
    return unless defined $av && defined $bv;
    my $rv = $av;
    ($av, $bv) = ($a->(), $b->());
    return $rv;
  }
}

BEGIN { *i_and = curry \&i_and_ }
```

7.4 DATABASES

In Section 4.3 we saw the beginnings of a database system that would manufacture an iterator containing the results of a simple query. To open the database we did:

```
my $dbh = FlatDB->new($datafile);
```

and then to perform a query,

```
$dbh->query($fieldname, $value);
```

or:

```
$dbh->callbackquery(sub { ... });
```

which selects the records for which the subroutine returns true.

Let's extend this system to handle compound queries. Eventually, we'll want the system to support calls like this:

```
$dbh->select("STATE = 'NY' |
              OWES > 100 & STATE = 'MA'");
```

This will require parsing of the query string, which we'll see in detail in Chapter 8. In the meantime, we'll build the internals that are required to support such queries.

The internals for simple queries like `"STATE = 'NY'"` are already done, since that's exactly what the `$dbh->query('STATE', 'NY')` does. We can assume that other simple queries are covered by similar simple functions, or perhaps by calls to `callbackquery()`. What we need now are ways to combine simple queries into compound queries.

The `i_and()` and `i_or()` functions we saw earlier will do what we want, if we modify them suitably. The main thing we need to arrange is to define a canonical

order for records produced by one of the simple query iterators. In particular, we need some way for the `i_and()` and `i_or()` operators to recognize that their two argument iterators have generated the same output record.

The natural way to do this is to tag each record with a unique ID number as it comes out of the query. Two different records will have different ID numbers. For flat-file databases, there's a natural record ID number already to hand: the record number of the record in the file. We'll need to adjust the `query()` function so that the iterators it returns will generate record numbers. When we last saw the `query()` function, it returned each record as a single string; this is a good opportunity to have it return a more structured piece of data:

CODE LIBRARY
FlatDB_Compose.pm

```
package FlatDB_Compose;
use base 'FlatDB';
use base 'Exporter';
@EXPORT_OK = qw(query_or query_and query_not query_without);
use Iterator_Logic;

# 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 $fh = $self->{FH};
seek $fh, 0, 0;
<$fh>;                # discard header line
my $position = tell $fh;
my $recno = 0;

return sub {
  local $_;
  seek $fh, $position, 0;
  while (<$fh>) {
    chomp;
    $recno++;
    $position = tell $fh;
    my @fields = split $self->{FIELDSEP};
    my $fieldval = $fields[$fieldnum];
    return [$recno, @fields] if $fieldval eq $value;
```

```
    }
    return;
  };
}
```

It might be tempting to try to use Perl's built-in `$.` variable here instead of having each iterator carry its own synthetic `$recno`, but that's a bad idea. We took some pains to make sure that a single database filehandle could be shared among more than one query. However, the information for `$.` is stored inside the filehandle; since we don't want the current record number to be shared among queries, we need to store it in the query object (which is private) rather than in the filehandle (which isn't). An alternative to maintaining a special `$recno` variable would be to use `$position` as a record identifier, since it's already lying around, and since it has the necessary properties of being different for different records and of increasing as the query proceeds through the file.

Now we need to manufacture versions of `i_and()` and `i_or()` that use the record ID numbers when deciding what to pass along. Because these functions are curried, we don't need to rewrite any code to do this:

```
BEGIN { *query_or  =  i_or(sub { $_[0][0] <=> $_[1][0] });
        *query_and = i_and(sub { $_[0][0] <=> $_[1][0] });
      }
BEGIN { *query_without = i_without(sub { $_[0][0] <=> $_[1][0] }); }
```

The comparator function says that arguments `$_[0]` and `$_[1]` will be arrays of record data, and that we should compare the first element of each, which is the record number, to decide which data should come out first and to decide record identity.

Here's a similarly modified version of `callbackquery()`:

```
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;
  my $recno = 0;

  return sub {
    local $_;
```

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

1;
```

In Chapter 8, we'll build a parser that, given this query:

```
"STATE = 'NY' | OWES > 100 & STATE = 'MA'"
```

makes this call:

```
query_or($dbh->query('STATE', 'NY'),
         query_and($dbh->callbackquery(sub { my %F = @_; $F{OWES} > 100 }),
                   $dbh->query('STATE', 'MA')
                  ))
```

and returns the resulting iterator. In the meantime, we can manufacture the iterator manually.

The one important logical connective that's still missing is "not," which is a little bit peculiar, logically, because its meaning is tied to the original database. If `$q` is a query for all the people in a database who are male, then `query_not($q)` should produce all the people from the database who are female. But the `query_not` function can't do that without visiting the original database to find the female persons. Unlike the outputs of `query_and()` and `query_or()`, the output of `query_not()` is not a selection of the inputs.

One way around this is for each query to capture a reference back to the original database that it's a query on. An alternative is to specify the database explicitly, as `$dbh->query_not($q)`. Then we can implement a more

general operator on queries, the so-called *set difference operator*, also known as *without*:

```
# $a but not $b
sub i_without_ {
  my ($cmp, $a, $b) = @_;
  my ($av, $bv) = ($a->(), $b->());
  return sub {
    while (defined $av) {
      my $d;
      while (defined $bv && ($d = $cmp->($av, $bv)) > 0) {
        $bv = $b->();
      }
      if ( ! defined $bv || $d < 0 ) {
        my $rv = $av; $av = $a->(); return $rv;
      } else {
        $bv = $b->();
        $av = $a->();
      }
    }
    return;
  }
}

BEGIN {
  *i_without = curry \&i_without_;
  *query_without =
    i_without(sub { my ($a,$b) = @_; $a->[0] <=> $b->[0] });
}

1;
```

If `$a` and `$b` are iterators on the same database, `query_without($a, $b)` is an iterator that produces every record that appears in `$a` but *not* in `$b`. This is useful on its own, and it also gives us a base for "not", which becomes something like this:

```
sub query_not {
  my $self = shift;
  my $q = shift;
  query_without($self->all, $q);
}
```

`$self->all` is a database method that performs a trivial query that disgorges all the records in the database. We could implement it specially, or, less efficiently, we could simply use:

```
sub all {
  $_[0]->callbackquery(sub { 1 });
}
1;
```

A possibly amusing note is that once we have `query_without()`, we no longer need `query_and()`, since (*a* and *b*) is the same as (*a* without (*a* without *b*)).

7.4.1 Operator Overloading

Perl provides a feature called *operator overloading* that lets us write complicated query expressions more conveniently. Operator overloading allows us to redefine Perl's built-in operator symbols to have whatever meaning we like when they are applied to our objects. Enabling the feature is simple. First we make a small change to methods such as `query()` so that they return iterators that are blessed into package `FlatDB`:

CODE LIBRARY
FlatDB_Ovl.pm

```
package FlatDB_Ovl;
BEGIN {
  for my $f (qw(and or without)) {
    *{"query_$f"} = \&{"FlatDB_Compose::query_$f"};
  }
}
use base 'FlatDB_Compose';

sub query {
  $self = shift;
  my $q = $self->SUPER::query(@_);
  bless $q => __PACKAGE__;
}

sub callbackquery {
  $self = shift;
  my $q = $self->SUPER::callbackquery(@_);
  bless $q => __PACKAGE__;
}

1;
```

Then we add:

```
use overload '|' => \&query_or,
             '&' => \&query_and,
             '-' => \&query_without,
             'fallback' => 1;
```

at the top of `FlatDB.pm`. From then on, any time a `FlatDB` object participates in an `|` or `&` operation, the specified function will be invoked instead.

Now, given the following simple queries:

```
my ($ny, $debtor, $ma) =
        ($dbh->query('STATE', 'NY'),
         $dbh->callbackquery(sub { my %F = @_; $F{OWES} > 100 }),
         $dbh->query('STATE', 'MA')
        );
```

we'll be able to replace this:

```
my $interesting = query_or($ny, query_and($debtor, $ma))
```

with this:

```
my $interesting = $ny | $debtor & $ma;
```

The operators are still Perl's built-in operators, and so they obey the usual precedence and associativity rules. In particular, `&` has higher precedence than `|`.