CHAPTER 6

INFINITE STREAMS

There's a special interface that we can put on iterators that makes them easier to deal with in many cases. One drawback of the iterators we've seen so far is that they were difficult or impossible to rewind; once data came out of them, there was no easy way to put it back again. Later on, in Chapter 8, we will want to scan forward in an input stream, looking for a certain pattern; if we don't see it, we might want to rescan the same input, looking for a different pattern. This is inconvenient to do with the iterators of Chapter 4, but the variation in this chapter is just the thing, and we will use it extensively in Chapter 8.

What we need is a data structure more like an array or a list. We can make the iterators look like linked lists, and having done so we get another benefit: We can leverage the enormous amount of knowledge and technique that already exists for dealing with linked lists.

A linked list is a data structure common in most languages, but seldom used in Perl, because Perl's arrays usually serve as a good enough replacement. We'll take a moment to review linked lists.

6.1 LINKED LISTS

A *linked list* is made up of *nodes*; each node has two parts: a *head*, which contains some data, and a *tail*, which contains (a pointer to) another linked list node, or possibly an undefined value, indicating that the current node is the last one in the list:

Here's typical Perl code that uses arrays to represent nodes:

```
sub node {
  my ($h, $t) = @_;
  [$h, $t];
}

sub head {
  my ($ls) = @_;
  $ls->[0];
}

sub tail {
  my ($ls) = @_;
  $ls->[1];
}

sub set_head {
  my ($ls, $new_head) = @_;
  $ls->[0] = $new_head;
}


sub set_tail {
  my ($ls, $new_tail) = @_;
  $ls->[1] = $new_tail;
}
```

Linked lists are one of the data structures that's ubiquitous in all low-level programming. They hold a sequence of data, the way an array does, but unlike an

array they needn't be allocated all at once. To add a new data item to the front of a linked list, all that's needed is to allocate a new node, store the new data item in the head of the node, and store the address of the old first node into the tail of the new node; none of the data needs to be moved. This is what `node()` does:

```
$my_list = node($new_data, $my_list);
```

In contrast, inserting a new item at the start of an array requires all the array elements to be moved over one space to make room.

Similarly, it's easy to splice a data item into the middle of a linked list by tweaking the tail of the node immediately before it:

```
sub insert_after {
  my ($node, $new_data) = @_;
  my $new_node = node($new_data, tail($node));
  set_tail($node, $new_node);
}
```

To splice data into the middle of an array requires that all of the following elements in the array be copied to make room, and the entire array may need to be moved if there isn't any extra space at the end for the last item to move into.

Scanning a linked list takes about twice as long as scanning the corresponding array, since you spend as much time following the pointers as you do looking at the data; with the array, there are no pointers. The big advantage of the array over the list is that the array supports fast indexed access. You can get or set array element `$a[$n]` instantly, regardless of what `$n` is, but accessing the *n*th element of a list requires scanning the entire list starting from the head, taking time proportional to *n*.

6.2 LAZY LINKED LISTS

As you'll recall from Chapter 4, one of the primary reasons for using iterators is to represent lists that might be enormous, or even infinite. Using a linked list as an implementation of an iterator won't work if all the list nodes must be in memory at the same time.

The lazy computation version of the linked list has a series of nodes, just like a regular linked list. And it might end with an undefined value in the tail of the last node, just like a regular linked list. But the tail might instead be an object called a *promise*. The promise is just that: a promise to compute the rest of the

list, if necessary. We can represent it as an anonymous function, which, if called, will return the rest of the list nodes. We'll add code to the `tail()` function so that if it sees it's about to return a promise, it will collect on the promise and return the head node of the resulting list instead. Nobody accessing the list with the `head()` or `tail()` functions will be able to tell that anything strange is going on:

CODE LIBRARY
Stream.pm

```
package Stream;
use base Exporter;
@EXPORT_OK = qw(node head tail drop upto upfrom show promise
                filter transform merge list_to_stream cutsort
                iterate_function cut_loops);

%EXPORT_TAGS = ('all' => \@EXPORT_OK);

sub node {
  my ($h, $t) = @_;
  [$h, $t];
}

sub head {
  my ($s) = @_;
  $s->[0];
}

sub tail {
  my ($s) = @_;
  if (is_promise($s->[1])) {
    return $s->[1]->();
  }
  $s->[1];
}

sub is_promise {
  UNIVERSAL::isa($_[0], 'CODE');
}
```

The modified version of the `tail()` function checks to see if the tail is actually a promise; if so, it invokes the promise function to manufacture the real tail, and returns that. This is sometimes called *forcing* the promise.

If nobody ever tries to look at the promise, then so much the better. The code will never be invoked, and we'll never have to go to the trouble of computing the tail.

We'll call these trick lists *streams*.

As is often the case, the most convenient representation of an empty stream is an undefined value. If we do this, we won't need a special test to see if a stream is empty; a stream value will be true if and only if it's nonempty. This also means that we can create the last node in a stream by calling `node($value)`; the result is a stream node whose head contains `$value` and whose tail is undefined.

Finally, we'll introduce some syntactic sugar for making promises, as we did for making iterators:

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

6.2.1 A Trivial Stream: `upto()`

To see how this all works, let's look at a trivial stream. Recall the `upto()` function from Section 4.2.1: Given two numbers, *m* and *n*, it returned an iterator that would return all the numbers between *m* and *n*, inclusive. Here's the linked list version:

```
sub upto_list {
  my ($m, $n) = @_;
  return if $m > $n;
  node($m, upto_list($m+1, $n));
}
```

This might consume a large amount of memory if `$n` is much larger than `$m`. Here's the lazy-stream version:

```
sub upto {
  my ($m, $n) = @_;
  return if $m > $n;
  node($m, promise { upto($m+1, $n) } );
}
```

It's almost exactly the same. The only difference is that instead of immediately making a recursive call to construct the tail of the list, it defers the recursive call and manufactures a promise instead. The node it returns has the right value (`$m`) in the head, but the tail is an IOU. If someone looks at the tail, the `tail()` function sees the promise and invokes the anonymous promise function, which in turn invokes `upto($m+1, $n)`, which returns another stream

node. The new node's head is `$m+1` (which is what was wanted) and its tail is another IOU.

If we keep examining the successive tails of the list, we see node after node, as if they had all been constructed in advance. Eventually we get to the end of the list, and `$m` is larger than `$n`; in this case when the `tail()` function invokes the promise, the call to `upto()` returns an empty stream instead of another node.

If we want an *infinite* sequence of integers, it's even easier: Get rid of the code that terminates the stream:

```
sub upfrom {
  my ($m) = @_;
  node($m, promise { upfrom($m+1) } );
}
```

Let's return to `upto()`. Notice that although the `upto()` function was obtained by a trivial transformation from the recursive `upto_list()` function, it is not itself recursive; it returns immediately. A later call to `tail()` may call it again, but the new call will again return immediately. Streams are therefore another way of transforming recursive list functions into nonrecursive, iterative functions.

We could perform the transformation in reverse on `upfrom()` and come up with a recursive list version:

```
sub upfrom_list {
  my ($m) = @_;
  node($m, upfrom_list($m+1) );
}
```

This function does indeed compute an infinite list of integers, taking an infinite amount of time and memory to do so.

6.2.2 Utilities for Streams

The first function you need when you invent a new data structure is a diagnostic function that dumps out the contents of the data structure. Here's a stripped-down version:

```
sub show {
  my $s = shift;
  while ($s) {
```

```
    print head($s), $";
    $s = tail($s);
  }
  print $/;
}
```

If the stream `$s` is empty, the function exits, printing `$/`, normally a newline. If not, it prints the head value of `$s` followed by `$"` (normally a space), and then sets `$s` to its tail to repeat the process for the next node.

Since this prints every element of a stream, it's clearly not useful for infinite streams; the `while` loop will never end. So the version of `show()` we'll actually use will accept an optional parameter *n*, which limits the number of elements printed. If *n* is specified, `show()` will print only the first *n* elements:

```
sub show {
  my ($s, $n) = @_;
  while ($s && (! defined $n || $n-- > 0)) {
    print head($s), $";
    $s = tail($s);
  }
  print $/;
}
```

For example:

CODE LIBRARY show-example-1

```
use Stream 'upfrom', 'show';

show(upfrom(7), 10);
```

This prints:

```
7 8 9 10 11 12 13 14 15 16
```

We can omit the second argument of `show()`, in which case it will print all the elements of the stream. For an infinite stream like `upfrom(7)`, this takes a long time. For finite streams, there's no problem:

CODE LIBRARY show-example-2

```
use Stream 'upto', 'show';

show(upto(3,6));
```

The output:

```
3 4 5 6
```

The line `$s = tail($s)` in `show()` is a fairly common operation, so we'll introduce an abbreviation:

```
sub drop {
  my $h = head($_[0]);
  $_[0] = tail($_[0]);
  return $h;
}
```

Now we can call `drop($s)`, which is analogous to `pop` for arrays: It removes the first element from a stream, modifying the stream in place, and returns that element. `show()` becomes:

```
sub show {
  my ($s, $n) = @_;
  while ($s && (! defined $n || $n-- > 0)) {
    print drop($s), $";
  }
  print $/;
}
```

As with the iterators of Chapter 4, we'll want a few basic utilities such as versions of `map` and `grep` for streams. Once again, the analogue of `map` is simpler:

```
sub transform (&$) {
  my $f = shift;
  my $s = shift;
  return unless $s;
  node($f->(head($s)),
       promise { transform($f, tail($s)) });
}
```

This example is prototypical of functions that operate on streams, so you should examine it closely. It's called in a way that's similar to `map()`:

```
transform {...} $s;
```

For example,

```
my $evens = transform { $_[0] * 2 } upfrom(1);
```

generates an infinite stream of all positive even integers. Or rather, it generates the first node of such a stream, and a promise to generate more, should we try to examine the later elements.

The analog of `grep()` is only a little more complicated:

```
sub filter (&$) {
  my $f = shift;
  my $s = shift;
  until (! $s || $f->(head($s))) {
    drop($s);
  }
  return if ! $s;
  node(head($s),
       promise { filter($f, tail($s)) });
}
```

`filter()` scans the elements of `$s` until either it runs out of nodes (`! $s`) or the predicate function `$f` returns true (`$f->(head($s))`). In the former case, there are no matching elements, so it returns an empty stream; in the latter case, it returns a new stream whose head is the matching element it found and whose tail is a promise to filter the rest of the stream in the same way. It would probably be instructive to compare this with the `igrep()` function of Section 4.4.2.

Another utility that will be useful is one to iterate a function repeatedly. Given an initial value x and a function f, it produces the (infinite) stream containing x, $f(x), f(f(x)), \ldots$. We could write it this way:

```
sub iterate_function {
  my ($f, $x) = @_;
  node($x, promise { iterate_function($f, $f->($x)) });
}
```

But there's a more interesting and even simpler way to do it that we'll see in Section 6.6.1.

6.3 RECURSIVE STREAMS

The real power of streams arises from the fact that it's possible to define a stream in terms of itself. Let's consider the simplest possible example, a stream that

contains an infinite sequence of carrots. Following the `upfrom()` example of the previous section, we begin like this:

```
sub carrots {
  node('carrot', promise { carrots() });
}
my $carrots = carrots();
```

It's silly to define a function that we're going to call from only one place; we might as well do this:

```
my $carrots = node('carrot', promise { carrots() });
```

except that we now must eliminate the call to `carrots()` from inside the promise. But that's easy too, because the `carrots()` and `$carrots` will have the same value:

```
my $carrots = node('carrot', promise { $carrots });
```

This looks good, but it doesn't quite work, because of an oddity in the Perl semantics. The scope of the `my` variable `$carrots` doesn't begin until the *next* statement, and that means that the two mentions of `$carrots` refer to different variables. The declaration creates a new lexical variable, which is assigned to, but the `$carrots` on the right-hand side of the assignment is the *global* variable `$carrots`, not the same as the one we're creating. The line needs a tweak:

```
my $carrots;
$carrots = node('carrot', promise { $carrots });
```

We've now defined `$carrots` as a stream whose head contains `'carrot'` and whose tail is a promise to produce the rest of the stream — which is identical to the entire stream. And it does work:

```
show($carrots, 10);
```

The output:

```
carrot carrot carrot carrot carrot carrot carrot carrot carrot carrot
```

6.3.1 Memoizing Streams

Let's look at an example that's a little less trivial than the one with the carrots: We'll construct a stream of all powers of 2. We could follow the `upfrom()` pattern:

```
sub pow2_from {
  my $n = shift;
  node($n, promise {pow2_from($n*2)})
}
my $powers_of_2 = pow2_from(1);
```

but again, we can get rid of the special-purpose `pow2_from()` function in the same way that we did for the carrot stream:

```
my $powers_of_2;
$powers_of_2 =
  node(1, promise { transform {$_[0]*2} $powers_of_2 });
```

This says that the stream of powers of 2 begins with the element 1, and then follows with a copy of itself with every element doubled. The stream itself contains 1, 2, 4, 8, 16, 32, . . . ; the doubled version contains 2, 4, 8, 16, 32, 64, . . . ; and if you append a 1 to the beginning of the doubled stream, you get the original stream back. Unfortunately, a serious and subtle problem arises with this definition. It does produce the correct output:

```
show($powers_of_2, 10);
1 2 4 8 16 32 64 128 256 512
```

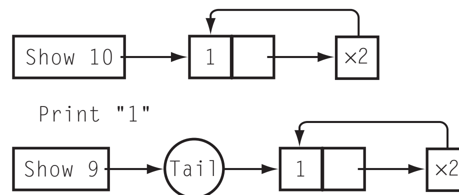
But if we instrument the definition, we can see that the transformation subroutine is being called too many times:

```
$powers_of_2 =
  node(1, promise {
            transform {
              warn "Doubling $_[0]\n";
              $_[0]*2
            } $powers_of_2
  });
```

The output is now:

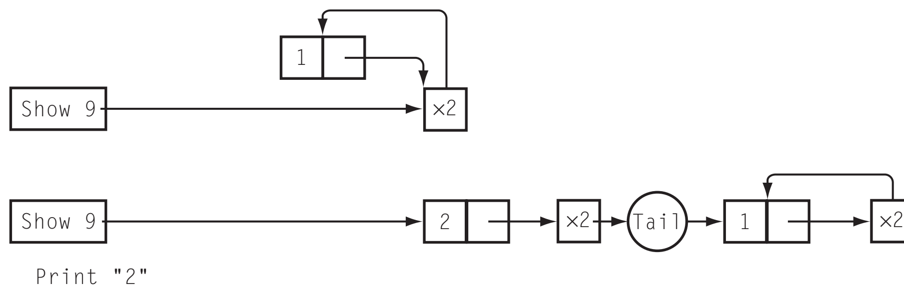
```
1 Doubling 1
2 Doubling 1
Doubling 2
4 Doubling 1
Doubling 2
Doubling 4
8 Doubling 1
Doubling 2
Doubling 4
Doubling 8
16 Doubling 1
...
```

The `show()` method starts by printing the head of the stream, which is 1. Then it goes to get the tail, using the `tail()` method:

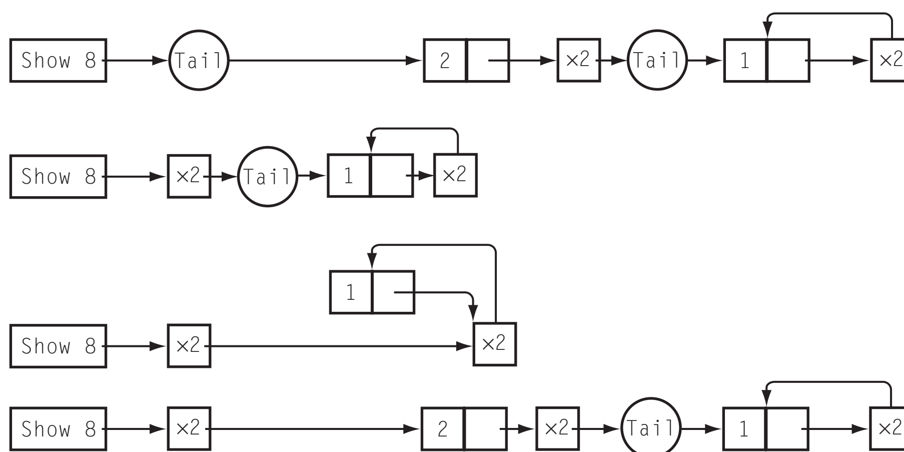



```
sub tail {
  my ($s) = @_;
  if (is_promise($s->[1])) {
    return $s->[1]->();
  }
  $s->[1];
}
```

Since the tail is a promise, this forces the promise, which calls `transform {...} $powers_of_2`. `transform()` gets the head of `$powers_of_2`, which is 1, and doubles it, yielding a stream whose head is 2 and whose tail is a promise to double the rest of the elements of `$powers_of_2`. This stream is the tail of `$powers_of_2`, and `show()` prints its head, which is 2.

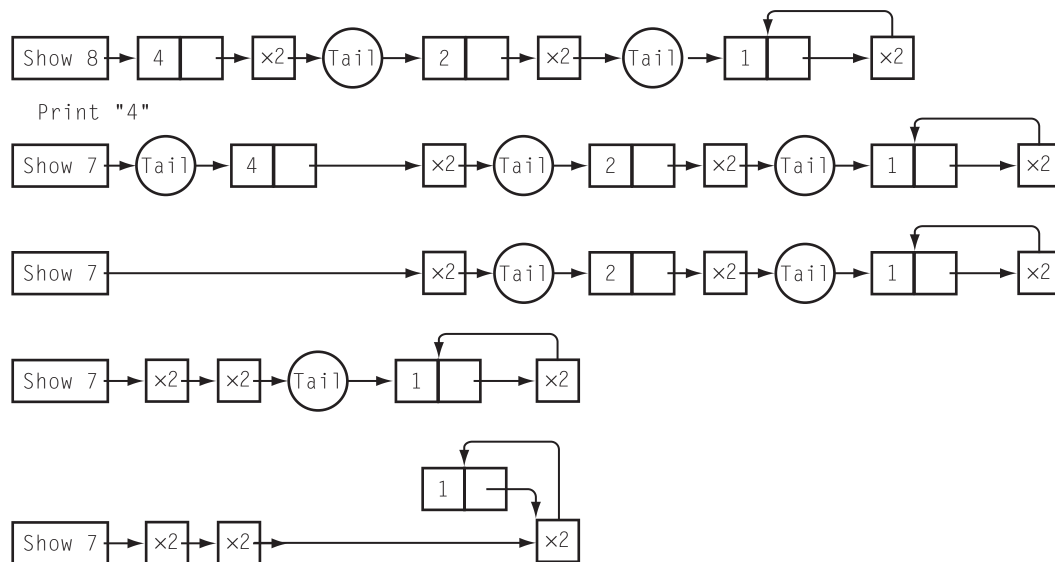


`show()` now wants to get the tail of the tail. It applies the `tail()` method to the tail stream. But the tail of the tail is a promise to double the tail of `$powers_of_2`. This promise is invoked, and the first thing it needs to do is compute the tail of `$powers_of_2`. This is the key problem, because computing the tail of `$powers_of_2` is something we've already done. Nevertheless, the promise is forced a second time, causing another invocation of `transform` and producing a stream whose head is 2 and whose tail is a promise to double the rest of the elements of `$powers_of_2`:



`transform` doubles the 2 and returns a new stream whose head is 4 and whose tail is (deep breath now) a promise to double the elements of the tail of a stream that was created by doubling the elements of the tail of `$powers_of_2`, which was itself created by doubling its own tail. `show()` prints the 4, but when it tries to get the tail of the new stream, it sets off a cascade of called promises, to get the

tail of the doubled stream, which itself needs to get the tail of another stream, which is the doubled version of the tail of the main stream:



Each element of the stream depends on calculating the tail of the original stream, and every time we look at a new element, we calculate the tail of `$powers_of_2`, including the act of doubling the first element. We're essentially computing each element from scratch by building it up from 1, and what we should be doing is building each element on the previous element. Our basic problem is that we're forcing the same promises over and over. But by now we have a convenient solution to problems that involve repeating the same work over and over: *memoization*. We should remember the result whenever we force a promise, and then if we need the same result again, instead of calling the promise function, we'll get it from the cache.

There's a really obvious, natural place to cache the result of a promise, too. Since we don't need the promise itself anymore, we can store the result in the tail of the stream — which was where it would have been if we hadn't deferred its computation in the first place.

The change to the code is simple:

```
sub tail {
  my ($s) = @_;
  if (is_promise($s->[1])) {
    $s->[1] = $s->[1]->();
```

```
  }
  $s->[1];
}
```

If the tail is a promise, we force the promise, and throw away the promise and replace it with the result, which is the real tail. Then we return the real tail. If we try to look at the tail again, the promise will be gone, and we'll see just the correct tail.

With this change, the $powers_of_2 stream is both correct *and* efficient. The instrumented version produces output that looks like this:

```
1 Doubling 1
2 Doubling 2
4 Doubling 4
8 Doubling 8
16 Doubling 16
32 Doubling 32
...
```

6.4 THE HAMMING PROBLEM

As an example of a problem that's easy to solve with streams, we'll turn to an old chestnut of computer science, Hamming's Problem.[1] Hamming's problem asks for a list of the numbers of the form $2^i 3^j 5^k$. The list begins as follows:

```
1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36 40 ...
```

It omits all multiples of 7, 11, 13, 17, and any other primes larger than 5.

The obvious method for generating this list is to try every number starting with 1. Suppose we want to learn whether the number *n* is on this list. If *n* is a multiple of 2, 3, or 5, then divide it by 2, 3, or 5 (respectively) until the result is no longer a multiple of 2, 3, or 5. If the final result is 1, then the original number *n* was a Hamming number. The code might look like this:

```
sub is_hamming {
  my $n = shift;
  $n/=2 while $n%2 == 0;
```

1 Named for Richard W. Hamming, who also invented Hamming codes.

```
  $n/=3 while $n%3 == 0;
  $n/=5 while $n%5 == 0;
  return $n == 1;
}


# Return the first $N hamming numbers
sub hamming {
  my $N = shift;
  my @hamming;
  my $t = 1;
  until (@hamming == $N) {
    push @hamming, $t if is_hamming($t);
    $t++;
  }
  @hamming;
}
```

Unfortunately, this is completely impractical. It starts off well enough. But the example Hamming numbers above are misleading — they're too close together. As you go further out in the list, the Hamming numbers get farther and farther apart. The 2999th Hamming number is 278,628,139,008. Nobody has time to test 278,628,139,008 numbers; even if they did, they would have to test 314,613,072 more before they found the 3000th Hamming number.

But there's a better way to solve the problem. There are four kinds of Hamming numbers: multiples of 2, multiples of 3, multiples of 5, and 1. And moreover, every Hamming number except 1 is either 2, 3, or 5 times some other Hamming number. Suppose we took the Hamming sequence and doubled it, tripled it, and quintupled it:

Hamming:	1	2	3	4	5	6	8	9	10	12	15	16	18	20	…
Doubled:	2	4	6	8	10	12	16	18	20	24	30	32	36	40	…
Tripled:	3	6	9	12	15	18	24	27	30	36	45	48	54	60	…
Quintupled:	5	10	15	20	25	30	40	45	50	60	75	80	90	100	…

and then merged the doubled, tripled, and quintupled sequences in order:

Merged: 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36 40 …

The result would be exactly the Hamming sequence, except for the 1 at the beginning. Except for the merging, this is similar to the way we constructed the sequence of powers of 2 earlier. To do it, we'll need a merging function:

```
sub merge {
  my ($S, $T) = @_;
  return $T unless $S;
  return $S unless $T;
  my ($s, $t) = (head($S), head($T));
  if ($s > $t) {
     node($t, promise {merge(     $S,  tail($T))});
   } elsif ($s < $t) {
     node($s, promise {merge(tail($S),      $T)});
   } else {
     node($s, promise {merge(tail($S), tail($T))});
   }
}
```

This function takes two streams of numbers, `$S` and `$T`, which are assumed to be in sorted order, and merges them into a single stream of numbers whose elements are also in sorted order. If either `$S` or `$T` is empty, the result of the merge is simply the other stream. (If both are empty, the result is therefore an empty stream.) If neither is empty, the function examines the head elements of `$S` and `$T` to decide which one should come out of the merged stream first. It then constructs a stream node whose head is the lesser of the two head elements, and whose tail is a promise to merge the rest of `$S` and `$T` in the same way. If the heads of `$S` and `$T` are the same number, the duplicate is eliminated in the output.

To avoid cluttering up our code with many calls to `transform()`, we'll build a utility that multiplies every element of a stream by a constant:

CODE LIBRARY
hamming.pl

```
use Stream qw(transform promise merge node show);

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

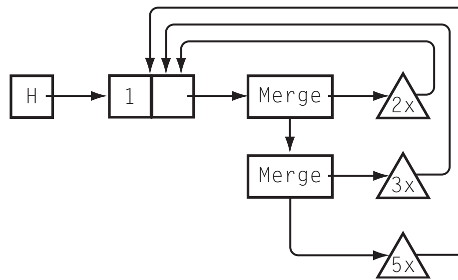
Now we can define a Hamming stream as a stream that begins with 1 and is otherwise identical to the merge of the doubled, tripled, and quintupled versions

of itself:

```
my $hamming;
$hamming = node(1,
                promise {
                  merge(scale($hamming, 2),
                  merge(scale($hamming, 3),
                        scale($hamming, 5),
                       ))
               }
              );


show($hamming, 3000);
```

This stream generates 3000 Hamming numbers (up to 278,942,752,080 = $2^4 \cdot 3^{20} \cdot 5$) in about 14 seconds. Its structure looks like this:



6.5 REGEX STRING GENERATION

Here's a question that comes up fairly often on IRC and in Perl-related newsgroups: Given a regex, how can one generate a list of all the strings matched by the regex? The problem can be rather difficult to solve. But the solution with streams is straightforward and compact.

There are a few complications that we should note first. In the presence of assertions and other oddities, there may not be any string that matches a given regex. For example, nothing matches the regex `/a\bz/`, because it requires the letters `a` and `z` to be adjacent, with a zero-length word boundary in between, and by definition, a word boundary does not occur between two adjacent letters. Similarly, `/a^b/` can't match any string, because the `b` must occur *at* the beginning of the string, but the `a` must occur *before* the beginning of the string.

Also, if our function is going to take a real regex as input, we have to worry about parsing regexes. We'll ignore this part of the problem until Chapter 8, where we'll build a parsing system that plugs into the string generator we'll develop here.

Most of the basic regex features can be reduced to combinations of a few primitive operators. These operators are concatenation, union, and `*`.[2] We'll review; if *A* and *B* are regexes, then:

- *AB*, the concatenation of *A* and *B*, is a regex that matches any string of the form *ab*, where *a* is a string that matches *A* and *b* is a string that matches *B*.
- *A* | *B*, the union of *A* and *B*, is a regex that matches any string *s* that matches *A* or *B*.
- *A** matches the empty string, or the concatenation of one or more strings that each individually match *A*.

With these operators, and the trivial regexes that match literal strings, we can build most of Perl's other regex operations. For example, `/A+/` is the same as `/AA*/`, and `/A?/` is the same as `/|A/`. Character classes are equivalent to unions; for example, `/[abc]/` and `/a|b|c/` are equivalent. Similarly `/./` is a union of 255 different characters (everything but the newline.)

`^` and `$` are easier to remove than to add, so we'll include them by default, so that all regexes are implicitly anchored at both ends. Our system will be able to generate the strings that match a regex only if the regex begins with `^` and ends with `$`. This is really no restriction at all, however. If we want to generate the strings that match `/A$/`, we can generate the strings that match `/^.*A$/` instead; these are exactly the same strings. Similarly the strings that match `/^A/` are the same as those that match `/^A.*$/` and the strings that match `/A/` are the same as those that match `/^.*A.*$/`. Every regex is therefore equivalent to one that begins with `^` and ends with `$`.

We'll represent a regex as a (possibly infinite) stream of the strings that it matches. The `Regex` class will import from `Stream`:

CODE LIBRARY
Regex.pm

```
package Regex;
use Stream ':all';
use base 'Exporter';
@EXPORT_OK = qw(literal union concat star plus charclass show
                matches);
```

2 The `*` operator is officially called the *closure operator*, and the set of strings that match `/A*/` is the *closure* of the set of those that match `/A/`. This has nothing to do with anonymous function closures.

FIGURE 6.1 The stream generated by `literal()`.

Literal regexes are trivial. The corresponding stream has only one element, as shown in Figure 6.1:

```
sub literal {
  my $string = shift;
  node($string, undef);
}
show(literal("foo"));
foo
```

Union is almost as easy. We have some streams, and we want to merge all their elements into a single stream. We can't append the streams beginning-to-end as we would with ordinary lists, because the streams might not have ends. Instead, we'll interleave the elements. Here's a demonstration function that mingles two streams this way:

```
sub mingle2 {
  my ($s, $t) = @_;
  return $t unless $s;
  return $s unless $t;
  node(head($s),
       node(head($t),
                promise { mingle2(tail($s),
                                  tail($t))
                                }
      ));
}
```

Later on it will be more convenient if we have a more general version that can mingle any number of streams:

```
sub union {
  my ($h, @s) = grep $_, @_;
  return unless $h;
  return $h unless @s;
  node(head($h),
       promise {
```

```
            union(@s, tail($h));
          });
    }
```

The function starts by throwing out any empty streams from the argument list. Empty streams won't contribute anything to the output, so we can discard them. If all the input streams are empty, `union()` returns an empty stream. If there is only one nonempty input stream, `union()` returns it unchanged. Otherwise, the function does the mingle: The first element of the first stream is at the head of the result, and the rest of the result is obtained by mingling the rest of the streams with the rest of the first stream. The key point here is that the function puts `tail($h)` at the *end* of the argument list in the recursive call, so that a different stream gets assigned to `$h` next time around. This will ensure that all the streams get cycled through the `$h` position in turn. The behavior is depicted in Figure 6.2. Here's a simple example:

```
# generate infinite stream ($k:1, $k:2, $k:3, ...)
sub constant {
  my $k = shift;
  my $i = shift || 1;
  my $s = node("$k:$i", promise { constant($k, $i+1) });
}

my $fish = constant('fish');
show($fish, 3);
fish:1 fish:2 fish:3

my $soup = union($fish, constant('dog'), constant('carrot'));

show($soup, 10);
fish:1 dog:1 carrot:1 fish:2 dog:2 carrot:2 fish:3 dog:3 carrot:3 fish:4
```

Now we'll do concatenation. If either of regexes S or T never matches anything, then ST also can't match anything. Otherwise, S is matched by some list of strings, and this list has a head `s` and a tail s_{tail}; similarly T is matched by some other list of strings with head `t` and tail t_{tail}. What strings are matched by ST? We can choose one string that matches S and one that matches T, and their concatenation is one of the strings that matches ST. Since we split each of the two lists into two parts, we have four choices for how to construct a string that matches ST:

1. `st` matches ST
2. `s` followed by any string from t_{tail} matches ST

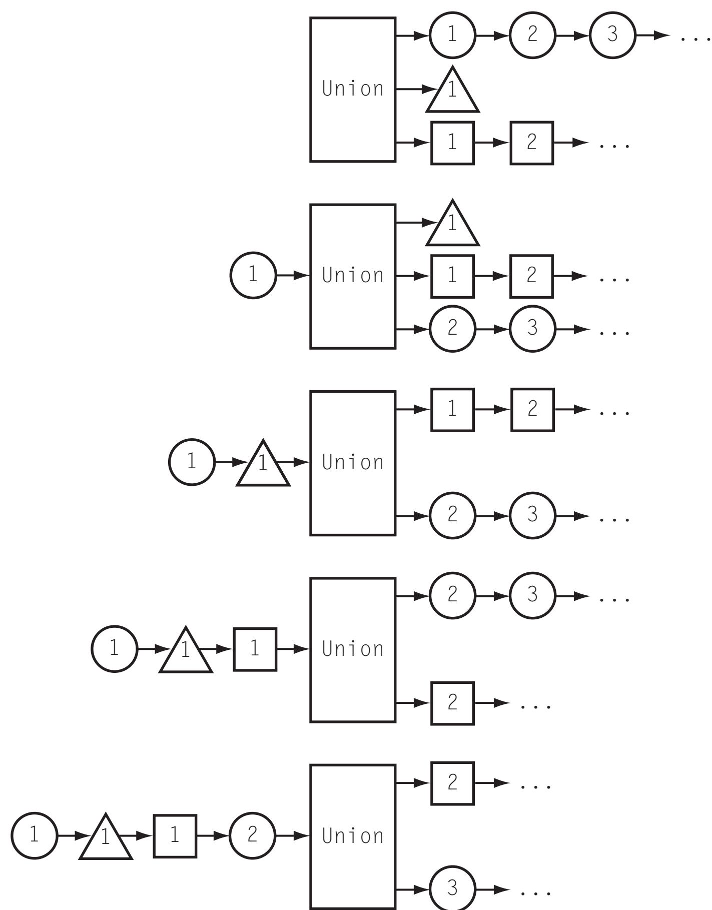


FIGURE 6.2 The behavior of `union()`.

3. Any string from s_{tail} followed by t matches ST
4. Any string from s_{tail} followed by any string from t_{tail} matches ST

Notationally, we write:

$$(s \mid s_{tail}).(t \mid t_{tail}) = s.t \mid s.t_{tail} \mid s_{tail}.t \mid s_{tail}.t_{tail}$$

The first of these contains only one string. The middle two are simple transformations of the tails of S and T. The last one is a recursive call to the `concat()` function itself. So the code is simple:

```
sub concat {
  my ($S, $T) = @_;
```

```
  return unless $S && $T;

  my ($s, $t) = (head($S), head($T));

  node("$s$t", promise {
    union(postcat(tail($S), $t),
           precat(tail($T), $s),
          concat(tail($S), tail($T)),
         )
  });
}
```

precat() and postcat() are simple utility functions that concatenate a string to the beginning or end of every element of a stream:

```
sub precat {
  my ($s, $c) = @_;
  transform {"$c$_[0]"} $s;
}

sub postcat {
  my ($s, $c) = @_;
  transform {"$_[0]$c"} $s;
}
```

An example:

```
# I'm /^(a|b)(c|d)$/
my $z = concat(union(literal("a"), literal("b")),
               union(literal("c"), literal("d")),
              );
show($z);
ac bc ad bd
```

The behavior of concat() is illustrated in Figure 6.3.

Now that we have concat(), the * operator is trivial, because of this simple identity:

```
s* = "" | ss*
```

That is, *s** is either the empty string or else something that matches *s* followed by something else that matches *s**. We want to generate *s**; let's call this result *r*.

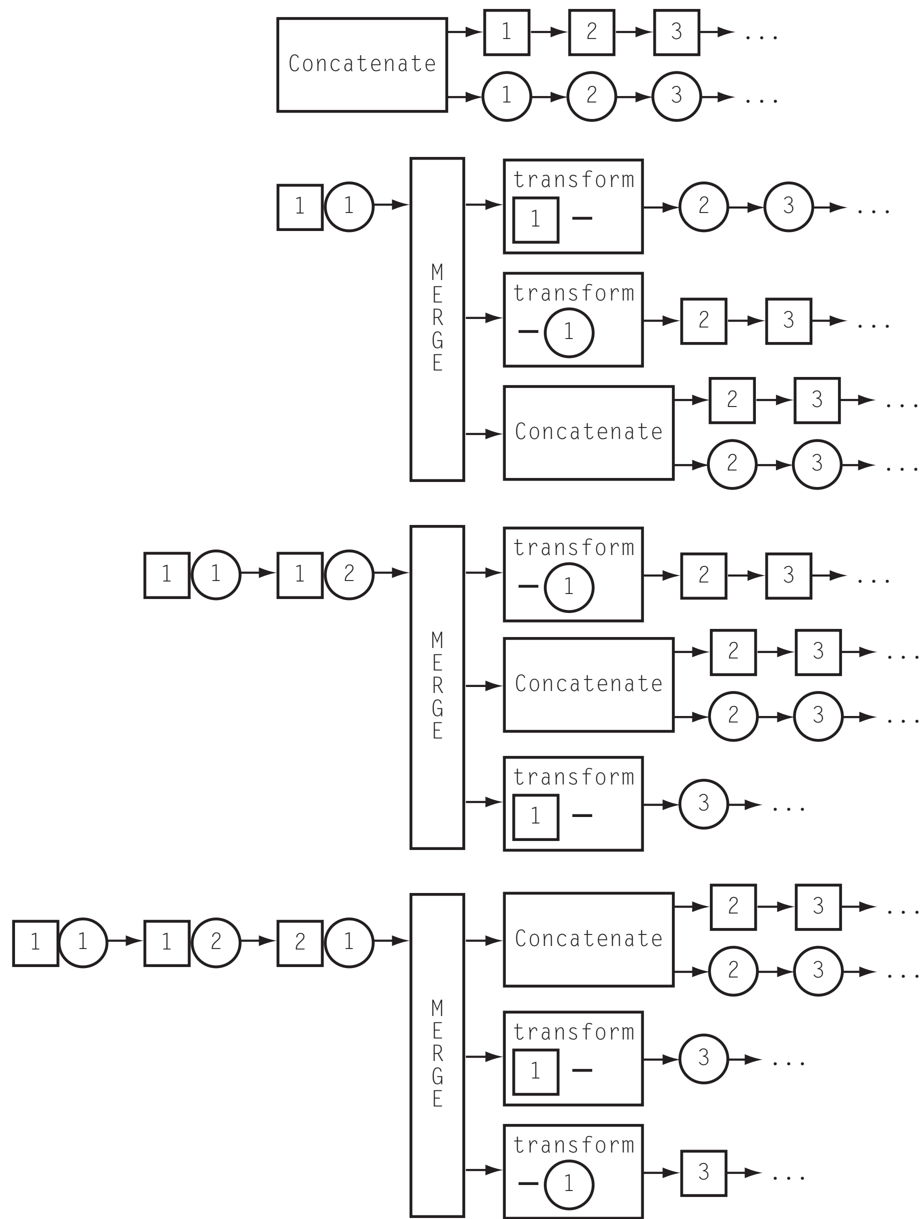


FIGURE 6.3 The behavior and internals of `concat()`.

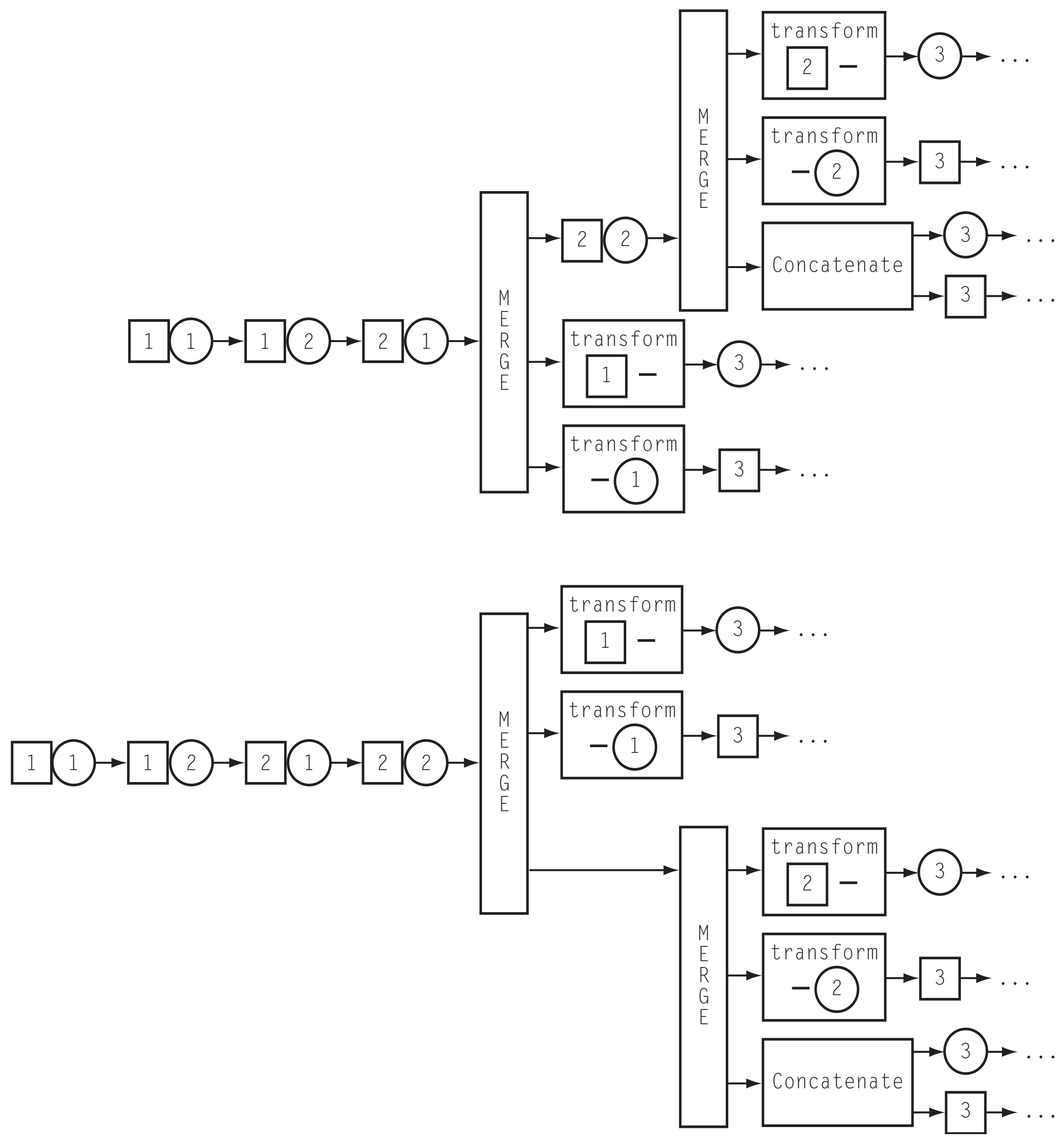


FIGURE 6.3 The behavior and internals of `concat()`, continued.

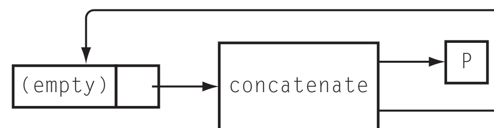


FIGURE 6.4 The behavior and internals of `star()`.

Then:

```
r = "" | sr
```

Now we can use the wonderful recursive definition capability of streams:

```
sub star {
  my $s = shift;
  my $r;
  $r = node("", promise { concat($s, $r) });
}
```

`$r`, the result, will be equal to the `*` of `$s`. It begins with the empty string, and the rest of `$r` is formed by concatenating something in `$s` with `$r` itself: Figure 6.4 shows how it works; here's an example:

```
# I'm /^(HONK)*$/
show(star(literal('HONK')), 6)
 HONK HONKHONK HONKHONKHONK HONKHONKHONKHONK HONKHONKHONKHONKHONK
```

The empty string is hiding at the beginning of that output line. Let's use a modified version of `show()` to make it visible:

```
sub show {
  my ($s, $n) = @_;
  while ($s && (! defined $n || $n-- > 0)) {
    print qq{"}, drop($s), qq{"\n};
  }
  print "\n";
}
```

Now the output is:

```
""
"HONK"
```

```
"HONKHONK"
"HONKHONKHONK"
"HONKHONKHONKHONK"
"HONKHONKHONKHONKHONK"
```

We can throw in a couple of extra utilities if we like:

```
# charclass('abc') = /^[abc]$/
sub charclass {
  my $class = shift;
  union(map literal($_), split(//, $class));
}

# plus($s) = /^s+$/
sub plus {
  my $s = shift;
  concat($s, star($s));
}
```

And now a demonstration:

```
use Regex qw(concat star literal show);
# I represent /^ab*$/
my $regex1 = concat(     literal("a"),
                    star(literal("b"))
                   );
show($regex1, 10);
```

The output is:

```
"a"
"ab"
"abb"
"abbb"
"abbbb"
"abbbbb"
"abbbbbb"
"abbbbbbb"
"abbbbbbbb"
"abbbbbbbbb"
```

Let's try something a little more interesting:

```
# I represent /^(aa|b)*$/
my $regex2 = star(union(literal("aa"),
                        literal("b"),
                       ));
show($regex2, 16);
```

The output is:

```
""
"aa"
"b"
"aaaa"
"baa"
"aab"
"bb"
"aaaaaa"
"baaaa"
"aabaa"
"bbaa"
"aaaab"
"baab"
"aabb"
"bbb"
"aaaaaaaa"
...
```

One last example:

```
# I represent /^(ab+|c)*$/
my $regex3 = star(union(concat(     literal("a"),
                               plus(literal("b"))),
                        literal("c")
                       ));
show($regex3, 20);
```

The output is:

```
""
"ab"
```

```
"c"
"abab"
"cab"
"abb"
"abc"
"abbab"
"abbb"
"ababab"
"cc"
"abbbb"
"abcab"
"abbc"
"abbbbb"
"ababb"
"abbbab"
"abbbbbb"
"ababc"
"cabab"
...
```

6.5.1 Generating Strings in Order

It's hard to be sure, from looking at this last output, that it really is generating all the strings that will match `/^(ab+|c)*/`. Will `cccc` really show up? Where's `cabb`? We might prefer the strings to come out in some order, say in order by length. It happens that this is also rather easy to do. Let's say that a stream of strings is "ordered" if no string comes out after a longer string has come out, and see what will be necessary to generate ordered streams.

The streams produced by `literal()` contain only one string, so those streams are already ordered, because one item can't be disordered.[3] `concat()`, it turns out, is already generating its elements in order as best it can. The business end is:

```
my ($s, $t) = (head($S), head($T));

node("$s$t", promise {
  union( precat(tail($T), $s),
```

3 Perhaps I should have included a longer explanation of this point, since I seem to be the only person in the world who is bothered by the phrase "Your call will be answered in the order it was received." It always seems to me that my call could not have an order.

```
                postcat(tail($S), $t),
                concat(tail($S), tail($T)),
               )
      });
```

Let's suppose that the inputs, `$S` and `$T`, are already ordered. In that case, `$s` is one of the shortest elements of `$S`, and `$t` is one of the shortest elements of `$T`. `$s$t` therefore can't be any longer than any other concatenation of elements from `$S` and `$T`, so it's all right that it will come out first. As long as the output of the `union()` call is ordered, the output of `concat()` will be too.

`union()` does need some rewriting. It currently cycles through its input streams in sequence. We need to modify it to find the input stream whose head element is shortest and to process that stream first:

```
sub union {
  my (@s) = grep $_, @_;
  return unless @s;
  return $s[0] if @s == 1;
  my $si = index_of_shortest(@s);
  node(head($s[$si]),
              promise {
                union(map $_ == $si ? tail($s[$_]) : $s[$_],
                          0 .. $#s);
              });
}
```

The first two `returns` correspond to the early `returns` in the original version of `union()`, handling the special cases of zero or one argument stream. If there's more than one argument stream, the function calls `index_of_shortest()`, which will examine the heads of the streams to find the shortest string. `index_of_shortest()` returns `$si`, the index number of the stream with the shortest head string. `union()` pulls off this string and puts it first in the output, then calls itself recursively to process the remaining data. `index_of_shortest()` is quite ordinary:

```
sub index_of_shortest {
  my @s = @_;
  my $minlen = length(head($s[0]));
  my $si = 0;
  for (1 .. $#s) {
    my $h = head($s[$_]);
    if (length($h) < $minlen) {
```

```
      $minlen = length($h);
      $si = $_;
    }
  }
  $si;
}
```

The last function to take care of is `star()`. But `star()`, it turns out, has taken care of itself:

```
sub star {
  my $s = shift;
  my $r;
  $r = node("", promise { concat($s, $r) });
}
```

The empty string, which comes out first, is certainly no longer than any other element in `$r`'s output. And since we already know that `concat` produces an ordered stream, we're finished.

That last example again:

```
# I represent /^(ab+|c)*$/
my $regex3 = star(union(concat(     literal("a"),
                               plus(literal("b"))),
                        literal("c")
                       ));
show($regex3, 30);
```

And the now-sorted output:

```
""
"c"
"ab"
"cc"
"abb"
"cab"
"ccc"
"abc"
"abbb"
"cabb"
"ccab"
```

```
"cccc"
"cabc"
"abbc"
"abab"
"abcc"
"abbbb"
"cabbb"
"ccabb"
"cccab"
"ccccc"
"ccabc"
"cabbc"
"cabab"
"cabcc"
"abbbc"
"ababb"
"abcab"
"abccc"
"ababc"
...
```

Aha, cccc and cabb *were* produced, after all.

6.5.2 Regex Matching

At this point we've built a system that can serve as a regex engine: Given a regex and a target string, it can decide whether the string matches the regex. A regex is a representation of a set of strings that supports operations like concatenation, union, and closure. Our regex-string-streams fit the bill. Here's a regex-matching engine:

```
sub matches {
  my ($string, $regex) = @_;
  while ($regex) {
    my $s = drop($regex);
    return 1 if $s eq $string;
    return 0 if length($s) > length($string);
  }
  return 0;
}
```

The `$regex` argument here is one of our regex streams. (After we attach the parser from Chapter 8, we'll be able to pass in a regex in standard Perl notation instead.) The function looks at the shortest string matched by the regex; if it's the target string, then we have a match. If it's longer than the target string, then the match fails, because every other string in the regex is also too long. Otherwise, the function throws away the head and repeats with the next string. If the regex runs out of strings, the match fails.

This is just an example; it should be emphasized that, in general, streams are *not* a good way to do regex matching. To determine whether a string matches `/^[ab]*$/`, this method generates all possible strings of `a`'s and `b`'s and checks each one to see if it is the target string. This is obviously a silly way to do it. The amount of time it takes is exponential in the length of the target string; an obviously better algorithm is to scan the target string left to right, checking each character to make sure it is an `a` or a `b`, which requires only linear time.

Nevertheless, in some ways this implementation of regex matching is actually *more* powerful than Perl's built-in matcher. For example, there's no convenient way to ask Perl if a string contains a balanced arrangement of parentheses. (Starting in 5.005, you can use the `(?{...})` operator, but it's nasty.[4]) But our "regexes" are just lists of strings, and the lists can contain whatever we want. If we want a regex that represents balanced arrangements of parentheses, all we need to do is construct a stream that contains the strings we want.

Let's say that we would like to match strings of `a`, `(`, and `)` in which the parentheses are balanced. That is, we'd like to match the following strings:

```
""
"a"
"aa"
"()"
"aaa"
"a()"
"()a"
"(a)"
"aaaa"
"aa()"
"a()a"
"()aa"
"a(a)"
"(a)a"
```

4 `/^(?{local$d=0})(?:\((?{$d++})|\)(?{$d--})(?(?{$d<0})(?!))|(?>[^()]*))*(?(?{$d!=0})(?!))$/.`

```
"(aa)"
"()()"
"(())"
"aaaaa"
...
```

Suppose *s* is a regex that matches the expressions that are legal between parentheses. Then a sequence of these expressions with properly balanced parentheses is one of the following:

- the empty string, or
- something that matches *s*, or
- (*b*), where *b* is some balanced string, or
- a balanced string followed by one of the above

Then we can almost read off the definition:

```
sub bal {
  my $contents = shift;
  my $bal;
  $bal = node("", promise {
    concat($bal,
           union($contents,
                 transform {"($_[0])"} $bal,
                )
          )
       });
}
```

And now the question "Does `$s` contain a balanced sequence of parentheses, `a`'s, and `b`'s" is answered by:

```
if (matches($s, bal(charclass('ab')))) {
  ...
}
```

6.5.3 Cutsorting

The regex-string generator suggests another problem that sometimes comes up. At present, it generates strings in order by length, but strings of the same length

come out in no particular order, as in the following column on the left. Suppose we want the strings of the same length to come out in sorted order, as in the column on the right:

```
""                              ""
"c"                             "c"
"ab"                            "ab"
"cc"                            "cc"
"abb"                           "abb"
"cab"                           "abc"
"ccc"                           "cab"
"abc"                           "ccc"
"abbb"                          "abab"
"cabb"                          "abbb"
"ccab"                          "abbc"
"cccc"                          "abcc"
"cabc"                          "cabb"
"abbc"                          "cabc"
"abab"                          "ccab"
"abcc"                          "cccc"
"abbbb"                         "ababb"
"cabbb"                         "ababc"
"ccabb"                         "abbab"
"cccab"                         "abbbb"
"ccccc"                         "abbbc"
"ccabc"                         "abbcc"
"cabbc"                         "abcab"
"cabab"                         "abccc"
"cabcc"                         "cabab"
"abbbc"                         "cabbb"
"ababb"                         "cabbc"
"abcab"                         "cabcc"
"abccc"                         "ccabb"
"ababc"                         "ccabc"
"abbab"                         "cccab"
"abbcc"                         "ccccc"
...                             ...
```

We should note first that although it's reasonable to ask for the strings sorted into groups by length and then lexicographically within each group, it's *not* reasonable to ask for *all* the strings to be sorted lexicographically. This is for two reasons.

First, even if we could do it, the result wouldn't be useful:

```
""
"ab"
"abab"
"ababab"
"abababab"
"ababababab"
"abababababab"
...
```

None of the strings that contains `c` would ever appear, because there would always be some other string that was lexicographically earlier that we had not yet emitted. But the second reason is that in general it's not possible to sort an infinite stream at all. In this example, the first string to be emitted is clearly the empty string. The second string out should be `"ab"`. But the sorting process can't know that. It can't emit the `"ab"` unless it's sure that no other string would come between `""` and `"ab"`. It doesn't know that the one-billionth string in the input won't be `"a"`. So it can never emit the `"ab"`, because no matter how long it waits to do so, there's always a possibility that `"a"` will be right around the corner.

But if we know in advance that the input stream will be sorted by string length, and that there will be only a finite number of strings of each length, then we certainly can sort each group of strings of the same length.

In general, the problem with sorting is that, given some string we want to emit, we don't know whether it's safe to emit it without examining the entire rest of the stream, which might be infinite. But suppose we could supply a function that would say whether or not it was safe. If the input stream is already sorted by length, then at the moment we see the first length–3 string in the input, we know it's safe to emit all the length–2 strings that we've seen already; the function could tell us this. The function effectively "cuts" the stream off, saying that enough of the input has been examined to determine the next part of the output, and that the cut-off part of the input doesn't matter yet.

This idea is the basis of *cutsorting*. The cutting function will get two arguments: the element we would like to emit, which should be the smallest one we have seen so far, and the current element of the input stream. The cutting function will return true if we have seen enough of the input stream to be sure that it's safe to emit the element we want to, and false if the rest of the input stream might contain an element that precedes the one we want the function to emit.

For sorting by length, the cutting function is trivial:

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

```
    # It's OK to emit item $a if the next item in the stream is $b
    length($a) < length($b);
  }
```

Since the cutsorter may need to emit several items at a time, we'll build a utility function for doing that:

```
sub list_to_stream {
  my $node = pop;
  while (@_) {
    $node = node(pop, $node);
  }
  $node;
}
```

`list_to_stream(h1, h2, ... t)` returns a stream that starts with `h1, h2, ...` and whose final tail is `t`. `t` may be another (possibly empty) stream or a promise. `list_to_stream(h, t)` is equivalent to `node(h, t)`.

The cutsorting function gets four arguments: `$s`, the stream to sort; `$cmp`, the sorting comparator (analogous to the comparator function of `sort()`); and `$cut`, the cutting test. It also gets an auxiliary argument `@pending` that we'll see in a moment:

```
sub insert (\@$$);

sub cutsort {
  my ($s, $cmp, $cut, @pending) = @_;
  my @emit;

  while ($s) {
    while (@pending && $cut->($pending[0], head($s))) {
      push @emit, shift @pending;
    }

    if (@emit) {
      return list_to_stream(@emit,
                            promise { cutsort($s, $cmp, $cut, @pending) });
    } else {
      insert(@pending, head($s), $cmp);
      $s = tail($s);
    }
  }
```

```
    return list_to_stream(@pending, undef);
  }
```

The idea of the cutsorter is to scan through the input stream, maintaining a buffer of items that have been seen so far but not yet emitted; this is `@pending`. `@pending` is kept in sorted order, so that if any element is ready to come out, it will be `$pending[0]`. The `while (@pending...)` loop checks to see if any elements can be emitted; if so, the emittable elements are transferred to `@emit`. If there are any such elements, they are emitted immediately: `cutsort()` returns a stream that begins with these elements and that ends with a promise to cutsort the remaining elements of `$s`. Any unemitted elements of `@pending` are passed along in the promise to be emitted later.

If no elements are ready for emission, the function discards the head element of the stream after inserting it into `@pending`. `insert()` takes care of inserting `head($s)` into the appropriate place in `@pending` so that `@pending` is always properly sorted.

If `$s` is exhausted, all items in `@pending` immediately become emittable, so the function calls `list_to_stream()` to build a finite stream that contains them and that ends with an empty tail.

Now if we'd like to generate strings in sorted order, we call `cutsort()` like this:

```
my $sorted =
  cutsort($regex3,
          sub { $_[0] cmp $_[1] },  # comparator
          \&cut_bylen               # cutting function
          );
```

The one piece of this that we haven't seen is `insert()`, which inserts an element into the appropriate place in a sorted array:

```
sub insert (\@$$) {
  my ($a, $e, $cmp) = @_;
  my ($lo, $hi) = (0, scalar(@$a));
  while ($lo < $hi) {
    my $med = int(($lo + $hi) / 2);
    my $d = $cmp->($a->[$med], $e);
    if ($d <= 0) {
      $lo = $med+1;
    } else {
      $hi = $med;
```

```
    }
  }
  splice(@$a, $lo, 0, $e);
}
```

This is straightforward, except possibly for the prototype. The prototype `(\@$$)` says that `insert()` will be called with three arguments: an array and two scalars, and that it will be passed a reference to the array argument instead of a list of its elements. It then performs a binary search on the array `@$a`, looking for the appropriate place to splice in the new element `$e`. A linear scan is simpler to write and to understand than the binary search, but it's not efficient enough for heavy-duty use.

At all times, `$lo` and `$hi` record the indices of elements of `@$a` that are known to satisfy `$a->[$lo]` ≤ `$e` < `$a->[$hi]`, where ≤ here represents the comparison defined by the `$cmp` function. Each time through the `while` loop, the function compares `$e` to the element of the array at the position halfway between `$lo` and `$hi`. Depending on the outcome of the comparison, the function now knows a new element `$a->[$med]` with `$a->[$med]` ≤ `$e` or with `$e` < `$a->[$med]`. We can then replace either `$hi` or `$lo` with `$med` while still preserving the condition `$a->[$lo]` ≤ `$e` < `$a->[$hi]`. When `$lo` and `$hi` are the same, the function has located the correct position for `$e` in the array, and uses `splice()` to insert `$e` in the appropriate place. For further discussion of binary search, see *Mastering Algorithms with Perl*, pp. 162–165.

LOG FILES

For a more practical example of the usefulness of cutsorting, consider a program to process a mail log file. The popular `qmail` mail system generates a log in the following format:

```
@400000003e382910351ebf4c new msg 706430
@400000003e3829103573e42c info msg 706430: bytes 2737 from <boehm5@email.com> qp 31064 uid 1001
@400000003e38291035d359ac starting delivery 190552: msg 706430 to local guitar-tpj-regex@plover.com
@400000003e38291035d3cedc status: local 1/5 remote 2/10
@400000003e3829113084e7f4 delivery 190552: success: did_0+1+0/qp_31067/
@400000003e38291130aa3aa4 status: local 1/5 remote 2/10
@400000003e3829120762c51c end msg 706430
```

The first field in each line is a time stamp in *tai64n* format. The rest of the line describes what the mail system is doing. `new msg` indicates that a new message has

been added to one of the delivery queues and includes the ID number of the new message. `info msg` records the sender of the new message. (A message always has exactly one sender, but may have any number of recipients.) `starting delivery` indicates that a delivery attempt is being started, the address of the intended recipient, and a unique delivery ID number. `delivery` indicates the outcome of the delivery attempt, which may be a successful delivery, or a temporary or permanent failure, and includes the delivery ID number. `end msg` indicates that delivery attempts to all the recipients of a message have ended in success or permanent failure, and that the message is being removed from the delivery queue. `status` lines indicate the total number of deliveries currently in progress.

This log format is complete and not too difficult to process, but it is difficult for humans to read quickly. We might like to generate summary reports in different formats; for example, we might like to reduce the life of the previous message to a single line:

```
706430 29/Jan/2003:14:18:30 29/Jan/2003:14:18:32 <boehm5@email.com> 1 1 0 0
```

This records the message ID number, the times at which the message was inserted into and removed from the queue, the sender, the total number of delivery attempts, and the number of attempts that were respectively successful, permanent failures, and temporary failures.

`qmail` writes its logs to a file called `current`; when `current` gets sufficiently large, it is renamed and a new `current` file is started. We'll build a stream that follows the `current` file, notices when a new `current` is started, and switches files when necessary. First we need a way to detect when a file's identity has changed. On Unix systems, a file's identity is captured by two numbers: the device number of the device on which it resides, and an *i-number* which is a per-device identification number. Both numbers can be obtained with the Perl `stat()` function:

CODE LIBRARY
logfile-process

```
sub _devino {
  my $f = shift;
  my ($dev, $ino) = stat($f);
  return unless defined $dev;
  "$dev;$ino";
}
```

The next function takes an open filehandle, a filename, and a device and i-number pair and returns the next record from the filehandle. If the handle is at the end of its file, the function checks to see if the filename now refers to a different file.

If so, the function opens the handle to the new file and continues; otherwise it waits and tries again:

```
sub _next_record {
  while (1) {
    my ($fh, $filename, $devino, $wait) = @_;
    $wait = 1 unless defined $wait;
    my $rec = <$fh>;
    return $rec if defined $rec;
    if (_devino($filename) eq $devino) {
      # File has not moved
      sleep $wait;
    } else {
      # $filename refers to a different file
      open $_[0], "<", $filename or return;
      $_[2] = _devino($_[0]);
    }
  }
}
```

Note that if `$fh` and `$devino` are initially unspecified, `_next_record` will initialize them when it is first called.

The next function takes a filename and returns a stream of records from the file, using `_next_record` to follow the file if it is replaced:

```
sub follow_file {
  my $filename = shift;
  my ($devino, $fh);
  tail(iterate_function(sub { _next_record($fh, $filename, $devino) }));
}

my $raw_mail_log = follow_file('/service/qmail/log/main/current');
```

Now we can write functions to transform this stream. For example, a quick-and-dirty function to convert tai64n format timestamps to Unix epoch format is:

```
sub tai64n_to_unix_time {
  my $rec = shift;
  return [undef, $rec] unless $rec =~ s/^\@([a-f0-9]{24})\s+//;
  [hex(substr($1, 8, 8)) + 10, $rec];
}
my $mail_log = &transform(\&tai64n_to_unix_time, $raw_mail_log);
```

Next is the function to analyze the log. Its input is a stream of log records from which the timestamps have been preprocessed by `tai64n_to_unix_time()`, and its output is a stream of hashes, each of which represents a single email message. The function gets two auxiliary arguments, `$msg` and `$del`, which are hashes that represent the current state of the delivery queue. The keys of `$del` are delivery ID numbers; each value is the ID number of the message with which the delivery is associated. The keys of `$msg` are message ID numbers; the values are structures that record information about the corresponding message, including the time it was placed in the queue, the sender, the total number of delivery attempts, and other information. A complete message structure looks like this:

```
{
   'id' => 706430,          # Message id number
   'bytes' => 2737,         # Message length
   'from' => '<boehm5@email.com>',  # Sender
   'deliveries' => [190552], # List of associated delivery ids

   'start' => 1043867776,  # Start time
   'end' => 1043867778,    # End time

   'success' => 1,         # Number of successful delivery attempts
   'failure' => 0,         # Number of permanently failed delivery attempts
   'deferral' => 0,        # Number of temporarily failed delivery attempts
   'total_deliveries' => 1,# Total number of delivery attempts
}
```

The stream produced by `digest_maillog()` is a sequence of these structures. To produce a structure, `digest_maillog()` scans the input records, adjusting `$msg` and `$del` as necessary, until it sees an `end msg` line; at that point it knows that it has complete information about a message, and it emits a single data item representing that message. If the input stream is exhausted, `digest_maillog()` terminates the output:

```
sub digest_maillog {
  my ($s, $msg, $del) = @_;
  for ($msg, $del) { $_ = {} unless $_ }
  while ($s) {
    my ($date, $rec) = @{drop($s)};
    next unless defined $date;
    if ($rec =~ /^new msg (\d+)/) {
      $msg->{$1} = {start => $date, id => $1,
```

```
                success => 0, failure => 0, deferral => 0};

    } elsif ($rec =~ /^info msg (\d+): bytes (\d+) from (<[^\>]*>)/) {
      next unless exists $msg->{$1};
      $msg->{$1}{bytes} = $2;
      $msg->{$1}{from} = $3;

    } elsif ($rec =~ /^starting delivery (\d+): msg (\d+)/) {
      next unless exists $msg->{$2};
      $del->{$1} = $2;
      push @{$msg->{$2}{deliveries}}, $1;

    } elsif ($rec =~ /^delivery (\d+): (success|failure|deferral)/) {
      next unless exists $del->{$1} && exists $msg->{$del->{$1}};
      $msg->{$del->{$1}}{$2}++;

    } elsif ($rec =~ /^end msg (\d+)/) {
      next unless exists $msg->{$1};
      my $m = delete $msg->{$1};
      $m->{total_deliveries} = @{$m->{deliveries}};
      for (@{$m->{deliveries}}) { delete $del->{$_} };
      $m->{end} = $date;
      return node($m, promise { digest_maillog($s, $msg, $del) });
    }
  }
  return;
}
```

Now we can generate reports by transforming the stream of message structures into a stream of log records:

```
use POSIX 'strftime';

sub format_digest {
  my $h = shift;
  join " ",
    $h->{id},
    strftime("%d/%b/%Y:%T", localtime($h->{start})),
    strftime("%d/%b/%Y:%T", localtime($h->{end})),
    $h->{from},
    $h->{total_deliveries},
```

```
              $h->{success},
              $h->{failure},
              $h->{deferral},
                ;
          }

          show(&transform(\&format_digest, digest_maillog($mail_log)));
```

Typical output looks like this:

```
...
707045 28/Jan/2003:12:10:03 28/Jan/2003:12:10:03 <Paulmc@371.net> 1 1 0 0
707292 28/Jan/2003:12:10:03 28/Jan/2003:12:10:06 <Paulmc@371.net> 1 1 0 0
707046 28/Jan/2003:12:10:06 28/Jan/2003:12:10:07 <Paulmc@371.net> 4 3 1 0
707293 28/Jan/2003:12:10:07 28/Jan/2003:12:10:07 <guido@odiug.zope.com> 1 1 0 0
707670 28/Jan/2003:12:10:06 28/Jan/2003:12:10:08 <spam-return-133409-@plover.com-@[]> 2 2 0 0
707045 28/Jan/2003:12:10:07 28/Jan/2003:12:10:11 <guido@odiug.zope.com> 1 1 0 0
707294 28/Jan/2003:12:10:11 28/Jan/2003:12:10:11 <guido@odiug.zope.com> 1 1 0 0
707047 28/Jan/2003:12:10:22 28/Jan/2003:12:10:23
        <ezmlm-return-10817-mjd-ezmlm=plover.com@list.cr.yp.to> 1 1 0 0
707048 28/Jan/2003:12:11:02 28/Jan/2003:12:11:02
        <perl5-porters-return-71265-mjd-p5p2=plover.com@perl.org> 1 1 0 0
707503 24/Jan/2003:11:29:49 28/Jan/2003:12:11:35
        <perl-qotw-discuss-return-1200-@plover.com-@[]> 388 322 2 64
707049 28/Jan/2003:12:11:35 28/Jan/2003:12:11:45 <> 1 1 0 0
707295 28/Jan/2003:12:11:41 28/Jan/2003:12:11:46
        <perl6-internals-return-14784-mjd-perl6-internals=plover.com@perl.org> 1 1 0 0
...
```

That was all a lot of work, and at this point it's probably not clear why the stream method has any advantage over the more usual method of reading the file one record at a time, tracking the same data structures, and printing output records as we go, something like this:

```
while (<LOG>) {
  # analyze current record
  # update $msg and $del
  if (/^end msg/) {
    print ...;
  }
}
```

One advantage was that we could encapsulate the follow-the-changing-file behavior inside its own stream. In a more conventionally structured program, the logic to track the moving file would probably have been threaded throughout the rest of the program. But we could also have accomplished this encapsulation by using a tied filehandle.

A bigger advantage of the stream approach comes if we want to reorder the output records. As written, the output stream contains message records in the order in which the messages were removed from the queue; that is, the output is sorted by the third field. Suppose we want to see the messages sorted by the second field, the time at which each message was first sent. In the preceding example output, notice the line for message 707503. Although the time at which it was removed from the queue (12:11:35 on 28 January) is in line with the surrounding messages, the time it was sent (11:29:49 on 24 January) is quite different. Most messages are delivered almost immediately, but this one took more than four days to complete. It represents a message that was sent to a mailing list with 324 subscribers. Two of the subscribers had full mailboxes, causing their mail systems to temporararily refuse new message for these subscribers. After four days, the mail system finally gave up and removed the message from the queue. Similarly, message 707670 arrived a second earlier but was delivered (to India) a second later than message 707293, which was delivered (locally) immediately after it arrived.

The ordinary procedural loop provides no good way to emit the log entries sorted in order by the date the messages were sent rather than by the date that delivery was completed. We can't simply use Perl's `sort()` function, since it works only on arrays, and we can't put the records into an array, because they extend into the indefinite future.

But in the stream-based solution, we can order the records with the cutsorting method, using the prefabricated cutsorting function we have already. There's an upper bound on how long messages can remain in the delivery queue; after four days any temporary delivery failures are demoted to permanent failures, and the message bounces. Suppose we have in hand the record for a message that was first queued on January 1 and that has been completely delivered. We can't emit it immediately, since the next item out of the stream might be the record for a message that was first queued on December 28 whose delivery didn't complete until January 2; this record should come out before the January 1 record because we're trying to sort the output by the start date rather than the end date. But we can tell the cutsorter that it's safe to emit the January 1 record once we see a January 5 record in the stream, because by January 5 any messages queued before January 1 will have been delivered one way or another:

```
my $QUEUE_LIFETIME = 4;      # Days
my $by_entry_date =
```

```
cutsort($mail_log,
        sub { $_[0]{start} <=> $_[1]{start} },
        sub { $_[1]{end} - $_[0]{end} >= $QUEUE_LIFETIME*86400 },
       );
```

The first anonymous function argument to `cutsort()` says how to order the elements of the output; we want them ordered by `{start}`, the date each message was placed into the queue. The second anonymous function argument is the cutting function; this function says that it's safe to emit a record *R* with a certain start date if the next record in the stream was for a message that was completed at least `$QUEUE_LIFETIME` days after *R*; any record that was queued before *R* would have to be removed less than `$QUEUE_LIFETIME` days later, and therefore there are no such records remaining in the stream. The output from `$by_entry_date` includes the records in the preceding sample, but in a different order:

```
...
707503 24/Jan/2003:11:29:49 28/Jan/2003:12:11:35 <perl-qotw-discuss-return-1200-@plover.com-@[]>
       388 322 2 64

... (many records omitted) ...

707045 28/Jan/2003:12:10:03 28/Jan/2003:12:10:03 <Paulmc@371.net> 1 1 0 0
707292 28/Jan/2003:12:10:03 28/Jan/2003:12:10:06 <Paulmc@371.net> 1 1 0 0
707046 28/Jan/2003:12:10:06 28/Jan/2003:12:10:07 <Paulmc@371.net> 4 3 1 0
707670 28/Jan/2003:12:10:06 28/Jan/2003:12:10:08 <spam-return-133409-@plover.com-@[]> 2 2 0 0
707293 28/Jan/2003:12:10:07 28/Jan/2003:12:10:07 <guido@odiug.zope.com> 1 1 0 0
707045 28/Jan/2003:12:10:07 28/Jan/2003:12:10:11 <guido@odiug.zope.com> 1 1 0 0
...
```

Even on a finite segment of the log file, cutsorting offers advantages over a regular sort. To use regular sort, the program must first read the entire log file into memory. With cutsorting, the program can begin producing output after only `$QUEUE_LIFETIME` days worth of records have been read in.

6.6 THE NEWTON-RAPHSON METHOD

How does Perl's `sqrt()` function work? It probably uses some variation of the *Newton-Raphson method*. You may have spent a lot of time toiling in high school

to solve equations; if so, rejoice, because the Newton-Raphson method is a general technique for solving any equation whatsoever.[5]

Suppose we're trying to calculate `sqrt(2)`. This is a number, which, when multiplied by itself, will give 2. That is, it's a number x such that $x^2 = 2$, or, equivalently, such that $x^2 - 2 = 0$.

If you plot the graph of $y = x^2 - 2$ you get a parabola illustrated in Figure 6.5. Every point on the parabola has $x^2 - 2 = y$. Points on the x-axis have $y = 0$. Where the parabola crosses the x-axis, we have $x^2 - 2 = 0$, and so the x-coordinate of the crossing point is equal to $\sqrt{2}$. This value is the solution, or *root*, of the equation.

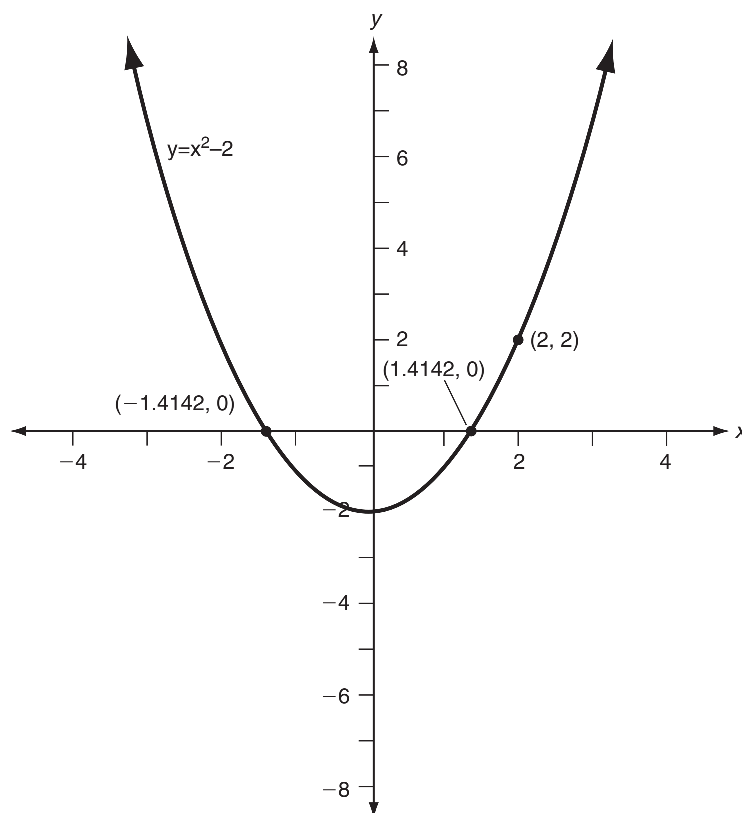


FIGURE 6.5 The parabola $y = x^2 - 2$.

5 Isaac Newton discovered and wrote about the method first, but his write-up wasn't published until 1736. Joseph Raphson discovered the technique independently and published it in 1671.

The Newton-Raphson method takes an approximation to the root and produces a closer approximation. We get the method started by guessing a root. Techniques for making a good first guess are an entire field of study themselves, but for the parabola in Figure 6.5, any guess except 0 will work. To show that the initial guess doesn't have to be particularly good, we'll guess that $\sqrt{2} = 2$. The method works by observing that a smooth curve, such as the parabola, can be approximated by a straight line, and constructs the tangent line to the curve at the current guess, which is $x = 2$. This is the straight line that touches the curve at that point, and that proceeds in the same direction that the curve was going. The curve will veer away from the straight line (because it's a curve) and eventually intersect the x-axis in a different place than the straight line does. But if the curve is reasonably well-behaved, it won't veer away too much, so the line's intersection point will be close to the curve's intersection point, and closer than the original guess.

The tangent line in this case happens to be the line $y = 4x - 6$. This line intersects the x-axis at $x = 1.5$, as shown in Figure 6.6. This value, 1.5, is our new guess for the value of $\sqrt{2}$. It is indeed more accurate than the original guess.

To get a better approximation, we repeat the process. The tangent line to the parabola at the point (1.5, 0.25) has the equation $y = 3x - 4.25$. This line intersects the x-axis at 1.41667, which is correct to two decimal places.

Now we'll see how these calculations were done. Let's suppose our initial guess is g, so we want to construct the tangent at $(g, g^2 - 2)$. If a line has slope m and passes through the point (p, q), its equation is $y - q = m(x - p)$. We'll see later how to figure out the slope of the tangent line without calculus, but in the meantime calculus tells us that the slope of the tangent line to the parabola at the point $(g, g^2 - 2)$ is $2g$, and therefore that the tangent line itself has the equation $(y - (g^2 - 2)) = 2g(x - g)$. We want to find the value of x for which this line intersects the x-axis; that is, we want to find the value of x for which y is 0. This gives us the equation $(0 - (g^2 - 2)) = 2g(x - g)$. Solving for x yields $x = g - (g^2 - 2)/2g = (g^2 + 2)/2g$. That is, if our initial guess is g, a better guess will be $(g^2 + 2)/2g$. A function that computes $\sqrt{2}$ is therefore:

CODE LIBRARY
Newton.pm

```
sub sqrt2 {
  my $g = 2; # Initial guess
  until (close_enough($g*$g, 2)) {
    $g = ($g*$g + 2) / (2*$g);
  }
  $g;
}
```

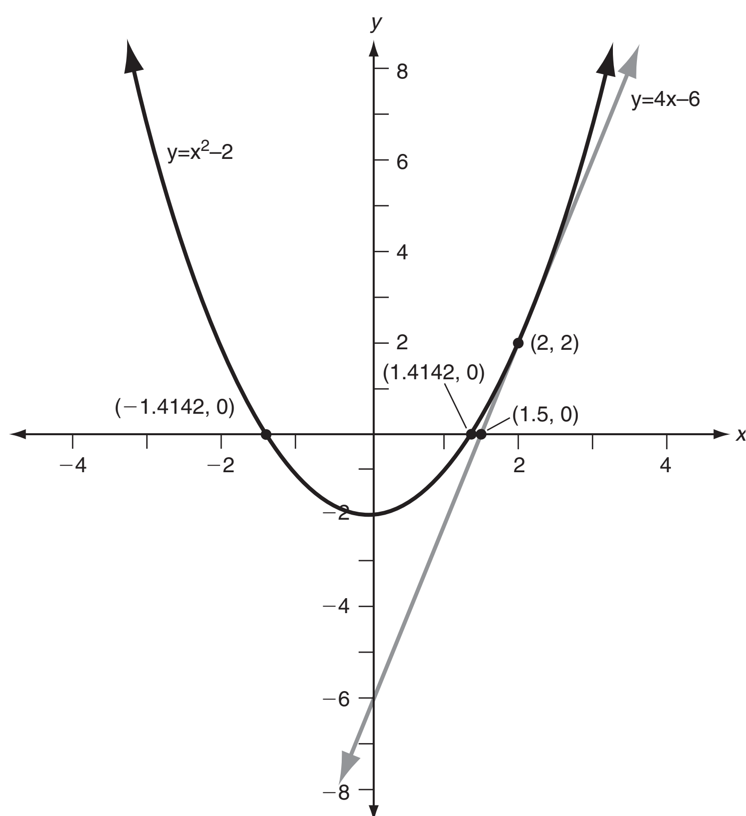


FIGURE 6.6 The parabola $y = x^2 - 2$.

```
sub close_enough {
  my ($a, $b) = @_;
  return abs($a - $b) < 1e-12;
}
```

This code rapidly produces a good approximation to $\sqrt{2}$, returning 1.414213562373095 after only five iterations. (This is correct to 15 decimal places.) To calculate the square root of a different number, we do the mathematics the same way, this time replacing the 2 with a variable *n*; the result is:

```
sub sqrtn {
  my $n = shift;
  my $g = $n;   # Initial guess
  until (close_enough($g*$g, $n)) {
    $g = ($g*$g + $n) / (2*$g);
```

```
  }
  $g;
}
```

6.6.1 Approximation Streams

But what does all this have to do with streams? One of the most useful and interesting uses for streams is to represent the results of an approximate calculation. Consider the following stream definition, which delivers the same sequence of approximations that the `sqrtn()` function would compute:

```
use Stream 'iterate_function';

sub sqrt_stream {
  my $n = shift;
  iterate_function (sub { my $g = shift;
                          ($g*$g + $n) / (2*$g);
                        },
                    $n);
}

1;
```

We saw `iterate_function()` back in Section 6.2.2. At the time, I promised a simpler and more interesting version. Here it is:

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

Recall that `iterate_function($f, $x)` produces the stream `$x`, `$f->($x)`, `$f->($f->($x))`,.... The preceding recursive version relies on the observation that the stream begins with `$x`, and the rest of the stream can be gotten by applying the function `$f` to each element in turn. The `&` on the call to `transform()` disables `transform()`'s prototype-derived special syntax. Without it, we'd have to write:

```
transform { $f->($_[0]) } $s
```

which would introduce an unnecessary additional function call.

6.6.2 Derivatives

The problem with the Newton-Raphson method as I described it in the previous section is that it requires someone to calculate the slope of the tangent line to the curve at any point. When we needed the slope at any point of the parabola $g^2 - 2$, I magically pulled out the formula $2g$. The function $2g$ that describes the slope at the tangent line at any point of the parabola $g^2 - 2$ is called the *derivative function* of the parabola; in general, for any function, the related function that describes the slope is called the *derivative*. Algebraic computation of derivative functions is the subject of the branch of mathematics called *differential calculus*.

Fortunately, though, you don't need to know differential calculus to apply the Newton-Raphson method. There's an easy way to compute the slope of a curve at any point. What we really want is the slope of the tangent line at a certain point. But if we pick two points that are close to the point we want, and compute the slope of the line between them, it won't be too different from the slope of the actual tangent line.

For example, suppose we want to find the slope of the parabola $y = x^2 - 2$ at the point (2, 2). We'll pick two points close to that and find the slope of the line that passes through them. Say we choose (2.001, 2.004001) and (1.999, 1.996001). The slope of the line through two points is the y difference divided by the x difference; in this case, $0.008/0.002 = 4$. And this does match the answer from calculus exactly. It won't always be an exact match, but it will always be close, because differential calculus uses exactly the same strategy, augmented with algebraic techniques to analyze what happens to the slope as the two points get closer and closer together.

It's not hard to write code that, given a function, calculates the slope at any point:

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

The value of `$e` that I chose is exactly 2^{-20}; I picked it because it was the power of 2 closest to one one-millionth. Powers of 2 work better than powers of 10 because they can be represented exactly; with a power of 10 you're introducing round-off error before you even begin. Smaller values of `$e` will give us more accurate answers, up to a point. The computer's floating-point numbers have only a fixed amount of accuracy, and as the numbers we deal with get smaller,

the round-off error will tend to dominate the answer. For the function `$f = sub { $_[0] * $_[0] - 2 }` and `$x = 2` our `slope()` function produces the correct answer (4) for values of `$e` down to 2^{-52}; at that point the round-off error takes over, and when `$e` is 2^{-54}, the calculated slope is 0 instead of 4. It's not hard to see what has happened: `$e` has become so small that when it's added to or subtracted from `$x`, and the result is rounded off to the computer's precision, the `$e` disappears entirely and we're left with exactly 2. So the calculated values of `$f->($x+$e)` and `$f->($x-$e)` are both exactly the same, and the `slope()` function returns 0.

Once we have this `slope()` function, it's easy to write a generic equation solver using the Newton-Raphson method:

```
# Return a stream of numbers $x that make $f->($x) close to 0
sub solve {
  my $f = shift;
  my $guess = shift || 1;
  iterate_function(sub { my $g = shift;
                         $g - $f->($g)/slope($f, $g);
                       },
                   $guess);
}
```

Now if we want to find $\sqrt{2}$, we do:

```
my $sqrt2 = solve(sub { $_[0] * $_[0] - 2 });

{ local $" = "\n";
  show($sqrt2, 10);
}
```

This produces the following output:

```
1
1.5
1.41666666666667
1.41421568627464
1.41421356237469
1.4142135623731
1.41421356237309
1.4142135623731
1.41421356237309
1.4142135623731
```

At this point the round-off error in the calculations has caused the values to alternate between 1.41421356237309 and 1.4142135623731.[6] The correct value is 1.41421356237309504880, so our little bit of code has produced an answer that is accurate to better than four parts per quadrillion.

If we want more accurate answers, we can use the standard Perl multi-precision floating-point library, `Math::BigFloat`. Doing so requires only a small change to the code:

```
use Math::BigFloat;

my $sqrt2 = solve(sub { $_[0] * $_[0] - 2 },
                  Math::BigFloat->new(2));
```

Using `Math::BigFloat` produces extremely accurate answers, but after only a few iterations, the numbers start coming out more and more slowly.

Because `Math::BigFloat` *never* rounds off, every multiplication of two numbers produces a number twice as long. The results increase in size exponentially, and so do the calculation times. The third iteration produces 1.4166666666666666666666666666666666666666667, which is an extremely precise but rather inaccurate answer. There's no point in retaining or calculating with all the 6's at the end, because we know they're wrong, but `Math::BigFloat` does it anyway, and the fourth iteration produces a result that has five accurate digits followed by 80 inaccurate digits.

One solution to this is to do more mathematics to produce an estimate of how many digits are accurate, and to round off the approximations to leave only the correct digits for later calculations. But this requires sophisticated technique. A simple solution is to improve the initial guess. Since Perl has a built-in square root function that is fast, we'll use it to generate our initial guess, which will already be accurate to about thirteen decimal places. Any work done by `Math::BigFloat` afterwards will only improve this:

```
my $sqrt2 = solve(sub { $_[0] * $_[0] - 2 },
                  Math::BigFloat->new(sqrt(2)));
```

The approximations still double in size at every step, and each one still takes twice as long as the previous one, but many more of the digits are correct, so the extra time spent isn't being wasted as it was before.

6 Actually they're alternating between 1.4142135623730949234300016933708 and 1.414213562373095145474621858739, but who's counting?

You may wait twice as long to get the answer, but you get an answer that has twice as many correct digits. The second element of the stream is 1.**4142135623730950488016887242**1836521533381246004441037, of which the first 28 digits after the decimal point are correct. The next element has 58 correct digits. In general, the Newton-Raphson method will double the number of correct digits at every step, so if you start with a reasonably good guess, you can get extremely accurate results very quickly.

6.6.3 The Tortoise and the Hare

The `$sqrt2` stream we built in the previous section is infinite, but after a certain point the approximations it produces won't get any more accurate because they'll be absorbed by the inherent error in the computer's floating-point numbers. The output of `$sqrt2` was:

```
1
1.5
1.41666666666667
1.41421568627464
1.41421356237469
1.4142135623731
1.41421356237309
1.4142135623731
1.41421356237309
...
```

`$sqrt2` is stuck in a loop. A process that was trying to use `$sqrt2` might decide that it needs more than 13 places of precision, and might search further and further down the stream, hoping for a better approximation that never arrives. It would be better if we could detect the loop in `$sqrt2` and cut off its tail.

The obvious way to detect a loop is to record every number that comes out of the stream and compare it to the items that came out before; if there is a repeat, then cut off the tail:

```
sub cut_loops {
  my $s = shift;
  return unless $s;
  my @previous_values = @_;
  for (@previous_values) {
    if (head($s) == $_) {
      return;
```

```
    }
  }
  node(head($s),
       promise { cut_loops(tail($s), head($s), @previous_values) });
}
```

cut_loops($s) constructs a stream that is the same as $s, but that stops at the point where the first loop begins. Unfortunately, it does this with a large time and memory cost. If the argument stream doesn't loop, the @previous_values array will get bigger and bigger and take longer and longer to search. There is a better method, sometimes called the *tortoise and hare algorithm*.

Imagine that each value in the stream is connected to the next value by an arrow. If the values form a loop, the arrows will too. Now imagine that a tortoise and a hare both start at the first value and proceed along the arrows. The tortoise crawls from one value to the next, following the arrows, but the hare travels twice as fast, leaping over every other value. If there is a loop, the hare will speed around the loop and catch up to the tortoise from behind. When this happens, you know that the hare has gone all the way around the loop once.[7] If there is no loop, the hare will vanish into the distance and will never meet the tortoise again:

```
sub cut_loops {
  my ($tortoise, $hare) = @_;
  return unless $tortoise;

  # The hare and tortoise start at the same place
  $hare = $tortoise unless defined $hare;

  # The hare moves two steps every time the tortoise moves one
  $hare = tail(tail($hare));

  # If the hare and the tortoise are in the same place, cut the loop
  return if head($tortoise) == head($hare);

  return node(head($tortoise),
              promise { cut_loops(tail($tortoise), $hare) });
}
```

7 It may not be obvious that the hare will necessarily catch the tortoise, but it is true. For details, see Donald Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*, exercise 3.1.6.

`show(cut_loops($sqrt2))` now generates:

```
1
1.5
1.41666666666667
1.41421568627464
1.41421356237469
1.4142135623731
```

and nothing else.

Notice that the entire loop didn't appear in the output. The loop consists of:

```
1.4142135623731
1.41421356237309
```

but we saw only the first of these. The tortoise and hare algorithm guarantees to cut the stream somewhere in the loop, *before* the values start to repeat; it might therefore place the cut sometime before all of the values in the loop have appeared. Sometimes this is acceptable behavior. If not, send the hare around the loop an extra time:

```
sub cut_loops2 {
  my ($tortoise, $hare, $n) = @_;
  return unless $tortoise;
  $hare = $tortoise unless defined $hare;

  $hare = tail(tail($hare));
  return if head($tortoise) == head($hare)
            && $n++;
  return node(head($tortoise),
              promise { cut_loops(tail($tortoise), $hare, $n) });
}
```

6.6.4 Finance

The square root of two is beloved by the mathematics geeks, but normal humans are motivated by other things, such as money. Let's suppose I am paying off a loan, say a mortgage. Initially I owe P dollars. (P is for "principal", which is the finance geeks' jargon word for it.) Each month, I pay *pmt* dollars, of which some goes to pay the interest and some goes to reduce the principal. When the principal reaches zero, I own the house.

For concreteness, let's say that the principal is \$100,000, the interest rate is 6% per year, or 0.5% per month, and the monthly payment is \$1,000. At the end of the first month, I've racked up \$500 in interest, so my \$1,000 payment reduces the principal to \$99,500. At the end of the second month, the interest is a little lower, only $\$99,500 \times 0.5\% = \$495.50$, so my payment reduces the principal by \$504.50, to \$98,995.50. Each month, my progress is a little faster. How long will it take me to pay off the mortgage at this rate?

First let's figure out how to calculate the amount owed at the end of any month. The first two months are easy:

Month	Amount owed
0	P

In the first month, we pay interest on the principal in the amount of $P \times .005$, bringing the total to $P \times 1.005$. But we also make a payment of *pmt* dollars, so that at the end of month 1, the amount owed is:

1	$P \cdot 1.005 - pmt$

The next month, we pay interest on the amount still owed. That amount is $P \times 1.005 - pmt$, so the interest is $(P \times 1.005 - pmt) \times .005$, and the total is $(P \times 1.005 - pmt) + (P \times 1.005 - pmt) \times 0.005$, or $(P \times 1.005^2 - pmt \times 1.005)$. Then we make another payment, bringing the total down to:

2	$P \cdot (1.005)^2 - pmt \cdot (1 + 1.005)$

The pattern continues in the third month:

3	$P \cdot (1.005)^3 - pmt \cdot (1 + 1.005 + (1.005)^2)$
4	$P \cdot (1.005)^4 - pmt \cdot (1 + 1.005 + (1.005)^2 + (1.005)^3)$

This pattern is simple enough that we can program it without much trouble:

```
sub owed {
  my ($P, $N, $pmt, $i) = @_;
  my $payment_factor = 0;
  for (0 .. $N-1) {
    $payment_factor += (1+$i) ** $_;
  }
  return $P * (1+$i)**$N - $pmt * $payment_factor;
}
```

It requires a little high school algebra to abbreviate the formula.[8] $1 + 1.005 + (1.005)^2 + \cdots + 1.005^{N-1}$ is equal to $(1.005^N - 1)/.005$, which is quicker to calculate:

4	$P \cdot (1.005)^4 - pmt \cdot ((1.005)^4 - 1)/0.005$
5	$P \cdot (1.005)^5 - pmt \cdot ((1.005)^5 - 1)/0.005$
6	$P \cdot (1.005)^6 - pmt \cdot ((1.005)^6 - 1)/0.005$

so the code gets simpler:

CODE LIBRARY
owed

```
sub owed {
  my ($P, $N, $pmt, $i) = @_;
  return $P * (1+$i)**$N - $pmt * ((1+$i)**$N - 1) / $i;
}
```

Now, the question that everyone with a mortgage wants answered: How long before my house is paid off?

We could try solving the equation $P \cdot (1 + i)^N - pmt \cdot \frac{(1+i)^N - 1}{i}$ for N, but doing that requires a lot of mathematical sophistication, much more than coming up with the formula in the first place.[9] It's much easier to hand the `owed()` function to `solve()` and let it find the answer:

```
sub owed_after_n_months {
  my $N = shift;
  owed(100_000, $N, 1_000, 0.005);
}

my $stream = cut_loops(solve(\&owed_after_n_months));
my $n;
$n = drop($stream) while $stream;
print "You will be paid off in only $n months!\n";
```

According to this, we'll be paid off in 138.9757 months, or eleven and a half years. This is plausible, since if there were no interest we would clearly have

8 It also requires a bit of a trick. Say $S = 1 + k + k^2 + \cdots + k^{n-1}$. Multiplying both sides by k gives $Sk = k + k^2 + \cdots + k^{n-1} + k^n$. These two equations are almost the same, and if we subtract one from the other almost everything cancels out, leaving only $Sk - S = k^n - 1$ and so $S = (k^n - 1)/(k - 1)$.

9 I'm afraid I am out of tricks.

the loan paid off in exactly 100 months. Indeed, after the 138th payment, the principal remains at $970.93, and a partial payment the following month finishes off the mortgage.

But we can ask more interesting questions. I want a thirty-year mortgage, and I can afford to pay $1,300 per month, or $15,600 per year. The bank is offering a 6.75% annual interest rate. How large a mortgage can I afford?

```
sub affordable_mortgage {
  my $mortgage = shift;
  owed($mortgage, 30, 15_600, 0.0675);
}

my $stream = cut_loops(solve(\&affordable_mortgage));
my $n;
$n = drop($stream) while $stream;
print "You can afford a \$$n mortgage.\n";
```

Apparently with a $1,300 payment I can pay off any mortgage up to $198,543.62 in 30 years.

6.7 POWER SERIES

We've seen that the Newton-Raphson method can be used to evaluate the `sqrt()` function. What about other built-in functions, such as `sin()` and `cos()`?

The Newton-Raphson method won't work here. To evaluate something like `sqrt(2)`, we needed to find a number x with $x^2 = 2$. Then we used the Newton-Raphson method, which required only simple arithmetic to approximate a solution. To evaluate something like `sin(2)`, we would need to find a number x with $\sin^{-1}(x) = 2$. This is at least as difficult as the original problem. x^2 is easy to compute; $\sin^{-1}(x)$ isn't.

To compute values of the so-called "transcendental functions" like `sin()` and `cos()`, the computer uses another strategy called *power series expansion*.[10]

A *power series* is an expression of the form:

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

10 These series are often called *Taylor series* or *Maclaurin series* after English mathematicians Brook Taylor and Colin Maclaurin who popularized them. The general technique for constructing these series was discovered much earlier by several people, including James Gregory and Johann Bernoulli.

for some numbers $a_0, a_1, a_2, \ldots$. Many common functions can be expressed as power series, and in particular, it turns out that for all x, $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \cdots$. (Here $a_0 = 0$, $a_1 = 1$, $a_2 = 0$, $a_3 = -1/3!$, etc.) The formula is most accurate for x close to 0, but if you carry it out to enough terms, it works for any x at all. The terms themselves get small rather quickly in this case, because the factorial function in the denominator increases more rapidly than the power of x in the numerator, particularly for small x. For example, $0.1 - (0.1)^3/3! + (0.1)^5/5! - (0.1)^7/7!$ is **.0998334166468 2**539683; the value of sin(0.1) is **.09983341664682**815230. When the computer wants to calculate the sine of some number, it plugs the number into the power series and calculates an approximation. The code to do this is simple:

CODE LIBRARY
sine

```
# Approximate sin(x) using the first n terms of the power series
sub approx_sin {
  my $n = shift;
  my $x = shift;
  my ($denom, $c, $num, $total) = (1, 1, $x, 0);
  while ($n--) {
    $total += $num / $denom;
    $num *= $x*$x * -1;
    $denom *= ($c+1) * ($c+2);
    $c += 2;
  }
  $total;
}
1;
```

At each step, `$num` holds the numerator of the current term and `$denom` holds the denominator. This is so simple that it's even easy in assembly language.

Similarly, $\cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + \cdots$.

Streams seem almost tailor-made for power series computations, because the power series itself is infinite, and with a stream representation we can look at as many terms as are necessary to get the accuracy we want. Once the terms become sufficiently small, we know that the rest of the stream won't make a significant contribution to the result.[11]

11 This shouldn't be obvious, since there are an infinite number of terms in the rest of the stream, and in general the infinite tail of a stream may make a significant contribution to the total. However, in a power series, the additional terms *do* get small so quickly that they can be disregarded, at least for sufficiently small values of x. For details, consult a textbook on numerical analysis or basic calculus.

We could build a `sin` function that, given a numeric argument, used the power series expansion to produce approximations to $\sin(x)$. But we can do better. We can use a stream to represent the entire power series itself, and then manipulate it as a single unit.

We will represent the power series $a_0 + a_1x + a_2x^2 + \cdots$ with a stream that contains $(a_0, a_1, a_2, \ldots)$. With this interpretation, we can build a function that evaluates a power series for a particular argument by substituting the argument into the series in place of x.

Since the nth terms of these power series depend in simple ways on n itself, we'll make a small utility function to generate such a series:

CODE LIBRARY
PowSeries.pm

```
package PowSeries;
use base 'Exporter';
@EXPORT_OK = qw(add2 mul2 partial_sums powers_of term_values
                evaluate derivative multiply recip divide
                $sin $cos $exp $log_ $tan);
use Stream ':all';

sub tabulate {
  my $f = shift;
  &transform($f, upfrom(0));
}
```

Given a function f, this produces the infinite stream $f(0), f(1), f(2), \ldots$. Now we can define `sin()` and `cos()`:

```
my @fact = (1);
sub factorial {
  my $n = shift;
  return $fact[$n] if defined $fact[$n];
  $fact[$n] = $n * factorial($n-1);
}


$sin = tabulate(sub { my $N = shift;
                      return 0 if $N % 2 == 0;
                      my $sign = int($N/2) % 2 ? -1 : 1;
                      $sign/factorial($N)
                    });


$cos = tabulate(sub { my $N = shift;
                      return 0 if $N % 2 != 0;
```

```
                                my $sign = int($N/2) % 2 ? -1 : 1;
                                $sign/factorial($N)
                            });
```

`$sin` is now a stream that begins $(0, 1, 0, -0.16667, 0, 0.00833, 0, \ldots)$; `$cos` begins $(1, 0, -0.5, 0, 0.041667, \ldots)$.

Before we evaluate these functions, we'll build a few utilities for performing arithmetic on power series. First is `add2()`, which adds together the elements of two streams, element-by-element:

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

`add2($s, $t)` corresponds to the addition of two power series. (Multiplication of power series is more complicated, as we will see later.) Similarly, `scale($s, $c)`, which we've seen before, corresponds to the multiplication of the power series `$s` by the constant `$c`.

`mul2()`, which multiplies streams element-by-element, is similar to `add2()`:

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

We will also need a utility function for summing up a series. Given a stream $(a_0, a_1, a_2, \ldots)$, it should produce the stream $(a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots)$ of successive partial sums of elements of the first stream. This function is similar to several others we've already defined:

```
sub partial_sums {
  my $s = shift;
  my $r;
  $r = node(head($s), promise { add2($r, tail($s)) });
}
```

One of the eventual goals of all this machinery is to compute sines and cosines. To do that, we will need to evaluate the partial sums of a power series for a particular value of x. This function takes a number x and produces the stream $(1, x, x^2, x^3, \ldots)$:

```
sub powers_of {
  my $x = shift;
  iterate_function(sub {$_[0] * $x}, 1);
}
```

When we multiply this stream element-wise by the stream of coefficients that represents a power series, the result is a stream of the terms of the power series evaluated at a point x:

```
sub term_values {
  my ($s, $x) = @_;
  mul2($s, powers_of($x));
}
```

Given a power series stream `$s` $= (a_0, a_1, a_2, \ldots)$, and a value `$x`, `term_values()` produces the stream $(a_0, a_1x, a_2x^2, \ldots)$.

Finally, `evaluate()` takes a function, as represented by a power series, and evaluates it at a particular value of x:

```
sub evaluate {
  my ($s, $x) = @_;
  partial_sums(term_values($s, $x));
}
```

And lo and behold, all our work pays off:

```
my $pi = 3.1415926535897932;
show(evaluate($cos, $pi/6), 20);
```

producing the following approximations to $\cos(\pi/6)$:

1
1
0.862922161095981
0.862922161095981
0.866053883415747

0.866053883415747
0.866025264100571
0.866025264100571
0.866025404210352
0.866025404210352
0.866025403783554
0.866025403783554
0.866025403784440
0.866025403784440
0.866025403784439
0.866025403784439
0.866025403784439
0.866025403784439
0.866025403784439
0.866025403784439

This is correct. (The answer happens to be exactly $\sqrt{3}/2$.)

We can even work it in reverse to calculate π:

```
# Get the n'th term from a stream
sub nth {
  my $s = shift;
  my $n = shift;
  return $n == 0 ? head($s) : nth(tail($s), $n-1);
}

# Calculate the approximate cosine of x
sub cosine {
  my $x = shift;
  nth(evaluate($cos, $x), 20);
}
```

If we know that $\cos(\pi/6) = \sqrt{3}/2$, then to find π we need only solve the equation $\cos(x/6) = \sqrt{3}/2$, or equivalently, $\cos^2(x/6) = 3/4$:

```
sub is_zero_when_x_is_pi {
  my $x = shift;
  my $c = cosine($x/6);
  $c * $c - 3/4;
}

show(solve(\&is_zero_when_x_is_pi), 20);
```

And the output from this is:

```
1
5.07974473179368
3.19922525384188
3.14190177620487
3.14159266278343
3.14159265358979
3.14159265358979
...
```

which is correct. The initial guess of 1, you will recall, is the default for `solve()`. Had we explicitly specified a better guess, such as 3, the process would have converged more quickly; had we specified a much larger guess, like 10, the results would have converged to a different solution, such as 11π.

6.7.1 Derivatives

We used `slope()` to calculate the slope of the curve $\cos^2(x/6) - 3/4$ at various points; recall that `slope()` calculates an approximation of the slope by picking two points close together on the curve and calculating the slope of the line between them. If we had known the derivative function of $\cos^2(x/6) - 3/4$, we could have plugged it in directly. But calculating a derivative function requires differential calculus.

However, if you know a power series for a function, calculating its derivative is trivial. If the power series for the function is $a_0 + a_1x + a_2x^2 + \cdots$, the power series for the derivative is $a_1 + 2a_2x + 3a_3x^2 + \cdots$. That is, it's simply:

```
sub derivative {
  my $s = shift;
  mul2(upfrom(1), tail($s));
}
```

If we do:

```
show(derivative($sin), 20);
```

we get exactly the same output as for:

```
show($cos, 20);
```

demonstrating that the cosine function is the derivative of the sine function.

6.7.2 Other Functions

Many other common functions can be calculated with the power series method. For example, Perl's built-in `exp()` function is:

```
$exp = tabulate(sub { my $N = shift; 1/factorial($N) });
```

The hyperbolic functions `sinh()` and `cosh()` are like `sin()` and `cos()` except without the extra `$sign` factor in the terms. Perl's built-in `log()` function is almost:

```
$log_ = tabulate(sub { my $N = shift;
                       $N==0 ? 0 : (-1)**$N/-$N });
```

This actually calculates $\log(x + 1)$; to get $\log(x)$, subtract 1 from x before plugging it in. (Unlike the others, it works only for x between -1 and 1.) The power series method we've been using won't work for an unmodified `log()` function, because it approximates every function's behavior close to 0, and $\log(0)$ is undefined.

The tangent function is more complicated. One way to compute $\tan(x)$ is by computing $\sin(x)/\cos(x)$. We'll see another way in the next section.

6.7.3 Symbolic Computation

As one final variation on power series computations, we'll forget about the numbers themselves and deal with the series as single units that can be manipulated algebraically. We've already seen hints of this earlier. If `$f` and `$g` are streams that represent the power series for functions $f(x)$ and $g(x)$, then `add2($f,$g)` is the power series for the function $f(x) + g(x)$, `scale($f,$c)` is the power series for the function $c \cdot f(x)$, and `derivative($f)` is the power series for the function $f'(x)$, the derivative of f.

Multiplying and dividing power series is more complex. In fact, it's not immediately clear how to divide one infinite power series by another. Or even, for that matter, how to multiply them. `mul2()` is *not* what we want here, because algebra tells us that $(a_0 + a_1x + \cdots) \times (b_0 + b_1x + \cdots) = a_0b_0 + (a_0b_1 + a_1b_0)x + \cdots$, and `mul2()` would give us $a_0b_0 + a_1b_1x + \cdots$ instead.

Our regex-string generator comes to the rescue: Power series multiplication is formally almost identical to regex concatenation. First note that if `$S` represents some power series, say $a_0 + a_1x + a_2x^2 + \cdots$ then `tail($S)` represents

$a_1 + a_2x + a_3x^2 + \cdots$. Then:

$$\begin{aligned} S &= a_0 \quad + a_1x + a_2x^2 + a_3x^3 + \cdots \\ &= a_0 \quad + x \cdot (a_1 + a_2x + a_3x^2 + \cdots) \\ &= \text{head}(S) + x \cdot \text{tail}(S) \end{aligned}$$

Now we want to multiply two series, `$S` and `$T`:

$$\begin{aligned} S &= \text{head}(S) \quad + x\ \text{tail}(S) \\ T &= \text{head}(T) \quad + x\ \text{tail}(T) \end{aligned}$$

$$\begin{aligned} S \cdot T &= \text{head}(S)\text{head}(T) + x\ \text{head}(S)\text{tail}(T) \\ &\qquad + x\ \text{head}(T)\text{tail}(S) + x^2\text{tail}(T)\text{tail}(S) \\ &= \text{head}(S)\text{head}(T) + x\ (\text{head}(S)\text{tail}(T) + \text{head}(T)\text{tail}(S) \\ &\qquad + x\ (\text{tail}(T)\text{tail}(S))) \end{aligned}$$

The first term of the result, `head(S) * head(T)`, is simply the product of two numbers. The rest of the terms can be found by summing three series. The first is `head(S) * tail(T)`, which is the tail of T scaled by `head(S)`, or `scale(tail($T), head($S))`. The second is `head(T) * tail(S)`, which is similar. The last term, `x * tail(S) * tail(T)`, is the product of two power series and can be computed with a recursive call; the extra multiplication by x just inserts a 0 at the front of the stream, since $x \cdot (a_0 + a_1x + a_2x^2 + \cdots) = 0 + a_0x + a_1x^2 + a_2x^3 + \cdots$.

Here is the code:

```
sub multiply {
  my ($S, $T) = @_;
  my ($s, $t) = (head($S), head($T));
  node($s*$t,
       promise { add2(scale(tail($T), $s),
                  add2(scale(tail($S), $t),
                       node(0,
                        promise {multiply(tail($S), tail($T))}),
                      ))
                }
       );
}
```

For power series, we can get a more efficient implementation by optimizing `scale()` slightly:

```
sub scale {
  my ($s, $c) = @_;
  return    if $c == 0;
  return $s if $c == 1;
  transform { $_[0]*$c } $s;
}
```

To test this, we can try out the identity $\sin^2(x) + \cos^2(x) = 1$:

```
my $one = add2(multiply($cos, $cos), multiply($sin, $sin));
show($one, 20);

1 0 0 0 0 0 0 0 4.33680868994202e-19 0 0 0 0 0 0 0 0 0 6.46234853557053e-27 0
```

Exactly 1, as predicted, except for two insignificant round-off errors.

We might like to make `multiply()` a little cleaner and faster by replacing the two calls to `add2()` with a single call to a function that can add together any number of series:

```
sub sum {
  my @s = grep $_, @_;
  my $total = 0;
  $total += head($_) for @s;
  node($total,
       promise { sum(map tail($_), @s) }
      );
}
```

`sum()` first discards any empty streams from its arguments, since they won't contribute to the sum anyway. It then adds up the heads to get the head of the result and returns a new stream with the sum at its head; the tail promises to add up the tails similarly. With this new function, `multiply()` becomes:

```
sub multiply {
  my ($S, $T) = @_;
  my ($s, $t) = (head($S), head($T));
  node($s*$t,
```

```
        promise { sum(scale(tail($T), $s),
                      scale(tail($S), $t),
                      node(0,
                        promise {multiply(tail($S), tail($T))}),
                     )
                }
        );
}
```

The next step is to calculate the reciprocal of a power series. If `$s` is the power series for a function $f(x)$, then the reciprocal series `$r` is the series for the function $1/f(x)$. To get this requires a little bit of algebraic ingenuity. Let's suppose that the first term of `$s` is 1. (If it's not, we can scale `$s` appropriately, and then scale the result back when we're done.)

$$
\begin{aligned}
r &= 1/f(x) \\
r &= 1/s \\
r &= 1/(1 + \text{tail}(s)) \\
r \cdot (1 + \text{tail}(s)) &= 1 \\
r + r \cdot \text{tail}(s) &= 1 \\
r &= 1 - r \cdot \text{tail}(s)
\end{aligned}
$$

And now, amazingly, we're done. We now know that the first term of `$r` must be 1, and we can compute the rest of the terms recursively by using our trick of defining the `$r` stream in terms of itself:

```
# Works only if head($s) = 1
sub recip {
  my ($s) = shift;
  my $r;
  $r = node(1,
            promise { scale(multiply($r, tail($s)), -1) });
}
```

The heavy lifting is done; dividing power series is now a one-liner:

```
# Works only if head($t) = 1
sub divide {
  my ($s, $t) = @_;
```

```
  multiply($s, recip($t));
}

$tan = divide($sin, $cos);
show($tan, 10);
```

0 1
0 0.333333333333333
0 0.133333333333333
0 0.053968253968254
0 0.0218694885361552

My *Engineering Mathematics Handbook*[12] says that the coefficients are 0, 1, 0, 1/3, 0, 2/15, 0, 17/315, 0, 62/2835, . . . , so it looks as though the program is working properly. If we would like the program to generate the fractions instead of decimal approximations, we should download the `Math::BigRat` module from CPAN and use it to initialize the `factorial()` function that is the basis of `$sin` and `$cos`.

`Math::BigRat` values are infectious: if you combine one with an ordinary number, the result is another `Math::BigRat` object. Since the `@fact` table is initialized with a `Math::BigRat`, its other elements will be constructed as `Math::BigRats` also; since the return values of `fact()` are `Math::BigRats`, the elements of `$sin` and `$cos` will be too; and since these are used in the computation of `$tan`, the end result will be `Math::BigRat` objects. Changing one line in the source code causes a ripple effect that propagates all the way to the final result:

```
my @fact = (Math::BigRat->new(1));

sub factorial {
  my $n = shift;
  return $fact[$n] if defined $fact[$n];
  $fact[$n] = $n * factorial($n-1);
}
```

The output is now:

```
0 1 0 1/3 0 2/15 0 17/315 0 62/2835
```

12 Jan J. Tuma, McGraw-Hill, 1970.