CHAPTER 8

PARSING

Parsing is the process of converting an unstructured input, such as a text file, into a data structure. Almost every program that reads input must perform parsing of some type. Long ago when I started writing this book, I was amazed to discover how underappreciated parsing processes are. One editor even worried that parsing might not be a useful application of Perl. But parsing is nearly universal, because almost every program must read an unstructured input, such as a text file, and turn it into a data structure, so that it can do some processing on it.

Programmers have become adept at designing data formats so that parsing is as simple as possible, and they may be unaware that they are doing it. But even Perl's `<HANDLE>` operator is a rudimentary parser, transforming a character stream into a record stream. For more complicated sorts of inputs, many programmers fall back on a lot of weird hackery. But writing parsers can be straightforward and elegant, even for complex inputs, once you have the right set of tools available.

8.1 LEXERS

Although it isn't absolutely required, parsing is almost always split into two smaller tasks. Consider the process of parsing an English sentence, such as "The bear devoured Joe's tent." Before we can consider the way the words go together, we have to *recognize* the words. At the beginning, all we have is a sequence of characters. We analyze the characters, observing that there are five words, and also a punctuation character, the period. We should probably consider the apostrophe to be part of the word `Joe's`, although an alternative is to consider `Joe's` to be a combination of the "words" `Joe` and `'s`. We'll probably throw away the whitespace, since that doesn't contribute to the meaning of the sentence except that it allows us to distinguish word boundaries.

A *token* is the smallest part of an input that has a definite meaning. In our English example, the characters `devoured` have a meaning, but `dev` and `oured`

don't. `devoured` is therefore a token; `dev` and `oured` aren't. (We could also argue that `devoured` can be further split, into `devour` and `ed`. But there's clearly a limit to how far the meaning can be divided; `d` and `e` have no meaning by themselves.)

Computer parsing processes are similar. Given an utterance such as `my $terminator = @_ ? shift:$/;`, the first thing we usually do is assemble up the unstructured characters of the input into meaningful tokens. Here, the tokens are `my`, `$terminator`, `=`, `@_`, `?`, `shift`, `:`, `$/`, and `;`. Alternatively, we could divide some of the tokens a little further; Perl's parser divides `$terminator` into `$` and `terminator`, and `@_` into `@` and `_`, for example. But again, there's a limit to how far we can divide things up without destroying the meaning. When analyzing the meaning of this statement, one simply doesn't consider the meaning of `min`.

This process of dividing the characters into tokens goes by several names: *tokenization*, *lexical analysis* or *lexing*, and *scanning*. All these words mean the same thing. The programs that do it are called *tokenizers*, *lexical analyzers*, *lexers,* or *scanners*.

One natural way to represent the lexing process is as an iterator of some type. The lexer can generate a sequence of tokens, potentially infinite, to be consumed by some later part of the parsing process.

8.1.1 Emulating the <> Operator

As a very simple example of a lexer, we'll build an iterator that takes a sequence of characters and turns it into a sequence of records. Here's a function that builds iterators that generate sequences of characters:

CODE LIBRARY
Lexer.pm

```
package Lexer;
use base "Exporter";
@EXPORT_OK = qw(make_charstream blocks records tokens iterator_to_stream
                make_lexer allinput);

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

sub make_charstream {
  my $fh = shift;
  return sub { return getc($fh) };
}

# For example:
my $stdin = make_charstream(\*STDIN);
```

If `$chars` is an iterator that generates a sequence of characters, then `records($chars)` turns it into an iterator that generates a sequence of records:

CODE LIBRARY
records

```
sub records {
  my $chars = shift;
  my $terminator = @_ ? shift : $/;
  my $buffer = "";
  sub {
    my $char;
    while (substr($buffer, -(length $terminator)) ne $terminator
           && defined($char = $chars->())
          ) {
      $buffer .= $char;
    }
    if ($buffer ne "") {
      my $line = $buffer;
      $buffer = "";
      return $line;
    } else {
      return;
    }
  }
}
```

In addition to the character generator, `records()` also gets an optional argument, a line-terminator string; if not supplied, it defaults to the current value of `$/`, which is the variable that controls the analogous behavior of the `<...>` operator. The iterator keeps a buffer with the characters seen so far but not returned, and appends each new character to the buffer as it appears. When the buffer is seen to end with the terminator string, it's emptied and the contents are returned to the caller.

With this formulation, we're in a position to do things that Perl doesn't have built-in already. For example, people often ask for the ability to set `$/` to a regex instead of to a plain string. This doesn't work; the "regex" is interpreted as a plain string anyway. The reason that Perl doesn't have this feature is that it's surprisingly tricky to implement. It looks easy. For example, we might try implementing it by changing `records()` as follows:

```
sub records {
  my $chars = shift;
  my $terminator = @_ ? shift : quotemeta($/);
```

```
    my $pattern = qr<(?:$terminator)$>;
    my $buffer = "";
    sub {
      my $char;
      while ($buffer !~ /$pattern/
             && defined($char = $chars->())
            ) {
        $buffer .= $char;
      }
      if ($buffer ne "") {
        my $line = $buffer;
        $buffer = "";
        return $line;
      } else {
        return;
      }
    }
}
```

If we're expecting inputs with sections that might be terminated either with "`---\n`" or with "`+++\n`", but we don't know which, we can say:

```
my $records = records($chars, qr/-{3}\n|\+{3}\n/);
```

Inside of `records()`, the terminator pattern is wrapped up into `qr/-{3}\n|\+{3}\n/`, which looks for the terminator pattern only at the end of the buffer.

There are two problems with this implementation. One problem is that it's not very efficient. Reading an input one character at a time into a buffer may be a good way to perform lexical analysis in C, but in Perl, it's slow. The more serious problem is that this implementation mishandles a number of regexes that come up in practice. The simplest example is the regex `/\n\n+/`, which says that each record is terminated by one or more following blank lines. Given this input:

```
a


b
```

(that is, "`a\n\n\nb\n`") there should be two records: "`a\n\n\n`" and "`b\n`". But this iterator gets a different answer; it produces "`a\n\n`" and "`\nb\n`". The problem here is that the terminator regex could be considered ambiguous. It says that a record is terminated by two or more newline characters. The iterator has decided

that the first record is terminated by exactly two newline characters, and that the third newline character is part of the *second* record. While this is arguably "correct", it probably isn't what was wanted. The problem occurs because the input is being read character-by-character; when the buffer contains "`a\n\n`", the terminator pattern succeeds, and the record is split, even though more reading would have generated a longer match.

The same bug causes a more serious problem in a different example. Suppose we're reading an email header and we'd like the iterator to generate logical fields instead of physical lines. Suppose the email header is as follows:

```
Delivered-To: mjd-filter-deliver2@plover.com
Received: from localhost [127.0.0.1] by plover.com
        with SpamAssassin (2.55 1.174.2.19-2003-05-19-exp);
        Mon, 11 Aug 2003 16:22:12 -0400
From: "Doris Bower" <yij447mrx@yahoo.com.hk>
To: webmaster@plover.com
Subject: LoseWeight Now with Pphentermine,Aadipex,Bontriil,PrescribedOnline,shipped
  to Your Door fltynzlfoybv kie
```

There are five fields here; the second one, with the `Received` tag, consists of three physical lines. Lines that begin with whitespace are continuations of the previous line. So if the records we want are email header fields, the terminator pattern is `/\n(?!\s)/`. That is, it's a newline that is not followed by a whitespace.

Note that `/\n[^\s]/` is not correct here, as this says that the following non-whitespace is actually part of the terminator, which it isn't. This would treat the `F` of `From` as the final character of the terminator of the `Received` field; the next field would then begin `rom: "Doris`. In contrast, the assertion `(?!\s)` behaves like part of the preceding `\n` symbol, constraining it to match only certain newline characters instead of all newline characters.

Plugging `/\n(?!\s)/` into `records()` doesn't work, however. The `Received` field is broken into three separate records anyway. What went wrong? Suppose the first `Received` line has been read in completely, including the newline character at the end. The iterator checks to see if the buffer matches `/(?:\n(?!\s))$/` — and it *does* match, because the buffer *does* end with a newline character, and the newline character is *not* followed by whitespace. So the iterator cuts off the line prematurely, without waiting to discover that this wasn't actually an appropriate newline.

We might try to fix this by changing the pattern to `/\n(?=\S)/`, which says that the fields are terminated by newline characters that *are* followed by *non*-whitespace. This does indeed prevent the `Received` field from being split prematurely, because when the first newline comes along, the pattern says that it must

be followed by non-whitespace, and it isn't followed by anything. But the pattern also prevents the field from being split in the correct place, and in fact the entire input comes out as one big field. This is because by the time the non-whitespace comes along, the pattern can no longer match at the end of the string, because it requires that the string end with a newline! So we need another approach.

Unfortunately, there doesn't seem to be any good way to solve this problem with the features currently in Perl.

The essential problem is that a pattern match can fail and backtrack for two essentially different reasons. Consider a pattern like `/(abcd)+/`, and the two target strings `"abcde"` and `"abcda"`. The regex matches the `"abcd"` part of both strings, but in different ways. When it matches the string `"abcde"`, it stops at the `e` because it sees the `e` and knows the match can't possibly continue. But when it matches the string `"abcda"` the match stops because the engine runs out of characters after the second `"a"`, and backtracks to the `"d"`.

For most applications, this distinction doesn't matter. But in *this* application, the second situation is quite different, because when the engine runs off the end of the string, we want it to try to read some more data, append it to the end of the target string, and continue. If the engine would tell us whether it reached the end of the string during the matching process, we could write code that would extend the target string and try the match again, but at present that feature is not available, and Perl doesn't give us any good way to distinguish the two situations.

The best we can do at present is to read the entire input into memory at once:

```
sub records {
  my $input = shift;
  my $terminator = @_ ? shift : quotemeta($/);
  my @records;
  my @newrecs = split /($terminator)/, $input;
  while (@newrecs > 2) {
    push @records, shift(@newrecs).shift(@newrecs);
  }
  push @records, @newrecs;
  return sub {
    return shift @records;
  }
}
```

There are a few complications here. We enclose the terminator in parentheses, to prevent `split` from discarding it. The `shift(@newrecs).shift(@newrecs)` expression reassembles a record with its terminator. The function does this only

when it is sure it has seen the beginning of the *next* record on the input, because it doesn't want to jump the gun and return an incomplete record or a record with an incomplete terminator; hence `@newrecs > 2` instead of `@newrecs >= 2`. When the input is exhausted, the input buffer contains the (possibly incomplete) final record, which is put onto the agenda.

8.1.2 Lexers More Generally

To write a lexer in a language like C, one typically writes a loop that reads the input, one character at a time, and which runs a state machine, returning a complete token to the caller when the state machine definition says to. Alternatively, we could use a program like `lex`, whose input is a definition of the tokens we want to recognize, and whose output is a state machine program in C.

In Perl, explicit character-by-character analysis of input is slow. But Perl has a special feature whose sole purpose is to analyze a string character-by-character and to run a specified state machine on it; the character loop is done internally in compiled C, so it's fast. This feature is regex matching. To match a string against a regex, Perl examines the string one character at a time and runs a state machine as it goes. The structure of the state machine is determined by the regex.

This suggests that regexes can act like lexers, and in fact Perl's regexes have been extended with a few features put in expressly to make them more effective for lexing.

As an example, let's consider a calculator program. The program will accept an input in the following form:

```
a = 12345679 * 6
b=a*9; c=0
print b
```

This will perform the indicated computations and print the result, 666666666. The first stage of processing this input is to tokenize it. Tokens are integer numerals; variable names, which have the form `/^[a-zA-Z_]\w*$/`; parentheses; the operators `+`, `-`, `*`, `/`, `**`, and `=`; and the special directive `print`. Also, newlines are significant, since they terminate expressions, while other whitespace is unimportant except insofar as it separates tokens that might otherwise be considered a single token. (For example, `printb` is a variable name, as opposed to `print b`, which isn't.)

The classic style for Perl lexers looks like this:

CODE LIBRARY
tokens-calc

```
sub tokens {
  my $target = shift;
```

```
  return sub {
    TOKEN: {
      return ['INTEGER', $1]    if $target =~ /\G (\d+)          /gcx;
      return ['PRINT']          if $target =~ /\G print \b       /gcx;
      return ['IDENTIFIER', $1] if $target =~ /\G ([A-Za-z_]\w*) /gcx;
      return ['OPERATOR', $1]   if $target =~ /\G (\*\*)         /gcx;
      return ['OPERATOR', $1]   if $target =~ /\G ([-+*\/=()])   /gcx;
      return ['TERMINATOR', $1] if $target =~ /\G (; \n* | \n+)  /gcx;
      redo TOKEN                if $target =~ /\G \s+            /gcx;
      return ['UNKNOWN', $1]    if $target =~ /\G (.)            /gcx;
      return;
    }
  };
}
```

There are a few obscure features here. Every Perl scalar may have associated with it a "current matching position," initially the leftmost end of the string. Whenever a scalar is matched against an expression with the `/g` flag, its current matching position is set to the position at which the regex left off matching. Moreover, if a scalar has a current matching position, then when it's matched against a pattern with `/g`, the search starts at the current matching position instead of at the leftmost end of the string, effectively ignoring everything to the left of the current matching position.

The current matching position of a scalar can be examined or set with the `pos()` built-in function. For example:

CODE LIBRARY
regex-g-demo

```
my $target = "123..45.6789...0";
while ($target =~ /(\d+)/g) {
  print "Saw '$1' ending at position ", pos($target), "\n";
}
```

The output is:

```
Saw '123' ending at position 3
Saw '45' ending at position 7
Saw '6789' ending at position 12
Saw '0' ending at position 16
```

In this example, the matching was able to skip past the dots in the string, just as `"carrot" =~ /r\w+/` is able to skip past the `c` and the `a`. The `\G` metacharacter anchors each match to occur *only* at the position that the previous match left off;

it's no longer allowed to skip characters at that position. This is what we want for our lexer, because we don't want the lexer to skip forward in the string looking for a numeral when there might be some other token that appears earlier; we want to process the string in strict left-to-right order, skipping nothing.

Our basic strategy is something like this:

```
if ($target =~ /\Gtoken1/g) {
  # do something with token1
} elsif ($target =~ /\Gtoken2/g) {
  # do something with token2
} elsif ...
```

The idea is that we'll search at the current position for something that looks like `token1`; if we don't find it, we'll look at that position for `token2`, and so on. Unfortunately, `/g` has a misfeature that prevents this from working: if the pattern match *fails* in a `/g` match, the current matching position is destroyed! By the time control reaches the `elsif` branch, the scalar has forgotten where the search is intended to occur.

We could work around this by using `pos` to save and restore the current matching position before and after every match, but Perl has another special feature that was introduced just so that we wouldn't have to do that: If the match operator has the `/c` flag as well as the `/g` flag, the misfeature is disabled. An unsuccessful match against a `/gc` pattern leaves the current matching position unchanged, instead of resetting it.

The iterator returned by `tokens()` uses this strategy. It captures the target string, and then, each time it's called, looks for some sort of token at the current matching position. If it finds one, it returns a value that represents the token; if not, it tries looking for a different kind of token.

The only exceptions to this rule are in the last three lines of the function. If the text at the current position is whitespace (but not newlines, which would have been taken care of by the `TERMINATOR` line) the function skips the whitespace and tries again. The following line (`UNKNOWN`) is a catchall for handling unrecognized characters. Alternatively, we might have written the function to throw an exception. Finally, the last line handles the case where the current position is at the very end of the string; as usual, it returns a false value to the caller to indicate that it has no more output.

On our sample input:

```
a = 12345679 * 6
b=a*9; c=0
print b
```

The output is:

```
[IDENTIFIER, "a"]
[OPERATOR, "="]
[INTEGER, "12345679"]
[OPERATOR, "*"]
[INTEGER, "6"]
[TERMINATOR, "\n"]
[IDENTIFIER, "b"]
[OPERATOR, "="]
[IDENTIFIER, "a"]
[OPERATOR, "*"]
[INTEGER, "9"]
[TERMINATOR, ";"]
[IDENTIFIER, "c"]
[OPERATOR, "="]
[INTEGER, "0"]
[TERMINATOR, "\n"]
[PRINT]
[IDENTIFIER, "b"]
[TERMINATOR, "\n"]
```

8.1.3 Chained Lexers

The lexer of the previous section is easy to read and to write, and it's efficient. It has one major drawback, however: the target string must be stored entirely in memory. As we saw, it's quite tricky to use regexes to tokenize an input that hasn't been completely read yet; we might have read some string that ends with `print`, and tokenized the `print` as a `print` operator, only to read another block and discover that it was actually the first five letters of an identifier `printmaking`. The lexer might similarly misparse `**` as `* *`.

A modified version of the iterator of Section 8.1.1 solves this problem. It gets three arguments. The first is an input iterator, as before. The input iterator will generate strings of input, perhaps a block at a time, or perhaps less or more. The second argument is a label, such as `IDENTIFIER` or `OPERATOR`, to include in the output tokens. The third argument is a regex that matches the tokens we want to find. The iterator's output will be a sequence of tokens and strings. When it sees something it recognizes, it converts it into a token, which is a value of the form:

```
[$label, $string]
```

and when it sees something it doesn't recognize, it returns it unchanged, as a plain string. So, for example, the lexer produced by `tokens($input, "NUMERAL", qr/\d+/)`, if given the same preceding sample input, will generate the following items:

```
"a = "
["NUMERAL", 12345679]
" * "
["NUMERAL", 6]
"\nb=a*"
["NUMERAL", 9]
"; c="
["NUMERAL", 0]
"\nprint b\n"
```

Once we have this, what do we do with it? We feed it as input to another lexer iterator, one which passes the `NUMERAL` tokens through unmodified, but examines the string portions for tokens that match its own regex. If we filter the input with a series of these iterators, we'll get an output stream that will contain the tokens we want, and also some plain strings that represent unrecognized portions of the input.

The code is complicated by the need to hold onto input while looking ahead to make sure that something coming up doesn't change the interpretation of the input we've seen already:

CODE LIBRARY
tokens

```
sub tokens {
  my ($input, $label, $pattern) = @_;
  my @tokens;
  my ($buf, $finished) = ("");
  my $split = sub { split /($pattern)/, $_[0] };
  my $maketoken = sub { [$label, $_[0] ]};
  sub {
    while (@tokens == 0 && ! $finished) {
      my $i = $input->();
      if (ref $i) {          # Input is a token
        my ($sep, $tok) = $split->($buf);
        $tok = $maketoken->($tok) if defined $tok;
        push @tokens, grep defined && $_ ne "", $sep, $tok, $i;
        $buf = "";
      } else {               # Input is an untokenized string
        $buf .= $i if defined $i; # Append new input to buffer
```

```
            my @newtoks = $split->($buf);
            while (@newtoks > 2
                   || @newtoks && ! defined $i) {
              # Buffer contains complete separator plus complete token
              # OR we've reached the end of the input
              push @tokens, shift(@newtoks);
              push @tokens, $maketoken->(shift @newtoks) if @newtoks;
            }
            # Reassemble remaining contents of buffer
            $buf = join "", @newtoks;
            $finished = 1 if ! defined $i;
            @tokens = grep $_ ne "", @tokens;
          }
        }
        return shift(@tokens);
      }
    }
```

The output agenda is `@tokens`. Tokens are put onto it under three circumstances:

1. When the current input contains so much text that it's clear the function has seen an entire token, plus at least one character of what follows it, then the token and any preceding non-token text are placed into `@tokens`. This occurs when `@newtoks > 2`.
2. When the current input is exhausted, a token is extracted from it, if possible, and then whatever was found is put into `@tokens`. This occurs when `$i`, the most recent input, is undefined.
3. When the most recent input is itself a token, passed up from some lower-level lexer, we know that the following characters aren't part of *this* token, so we can examine the current text in the same way as if it were at the end of the input. This occurs when `$i`, the most recent input, is a token; the test is `if (ref $i)`.

The input to `tokens()` is usually another iterator that was built by `tokens()`. To get the process started, we could supply an iterator that emits a string containing the data that is to be tokenized:

```
sub allinput {
  my $fh = shift;
  my @data;
  { local $/;
```

```
    $data[0] = <$fh>;
  }
  sub { return shift @data }
}
```

We can avoid the need to read the entire input into memory all at once, and use a base iterator that returns one block of data at a time from a filehandle:

```
sub blocks {
  my $fh = shift;
  my $blocksize = shift || 8192;
  sub {
    return unless read $fh, my($block), $blocksize;
    return $block;
  }
}
```

But if we use this, then we must be extremely careful that we don't exercise the problem we saw back in Section 8.1.1, in which a lexer might return a short token when it could have returned a longer one.

We can generalize `tokens()` a little bit. The code that manufactures the token value, `$maketoken`, is an anonymous function. We might as well let the user specify the function that performs this task, making it into a callback. Then `$label`, which isn't used anywhere else, merely becomes a user argument to this callback:

```
sub tokens {
  my ($input, $label, $pattern, $maketoken) = @_;
  $maketoken ||= sub { [ $_[1], $_[0] ] };
  my @tokens;
  my $buf = "";   # set to undef to when input is exhausted
  my $split = sub { split /($pattern)/, $_[0] };
  sub {
    while (@tokens == 0 && defined $buf) {
      my $i = $input->();
      if (ref $i) {
        my ($sep, $tok) = $split->($buf);
        $tok = $maketoken->($tok, $label) if defined $tok;
        push @tokens, grep defined && $_ ne "", $sep, $tok, $i;
        $buf = "";
        last;
```

```
            }

            $buf .= $i if defined $i;
            my @newtoks = $split->($buf);
            while (@newtoks > 2
                   || @newtoks && ! defined $i) {
              push @tokens, shift(@newtoks);
              push @tokens, $maketoken->(shift(@newtoks), $label)
                      if @newtoks;
            }
            $buf = join "", @newtoks;
            undef $buf if ! defined $i;
            @tokens = grep $_ ne "", @tokens;
          }
          return shift(@tokens);
        }
      }
```

We can call this version of `tokens` just as before, but now it's more flexible. Formerly, we could have written a whitespace recognizer that generated whitespace tokens:

```
tokens($input, "WHITESPACE", qr/\s+/)
```

but we would have had to discard these tokens later on, perhaps with `igrep`, since we weren't really interested in them. With the new formulation, we can write:

```
tokens($input, "WHITESPACE", qr/\s+/, sub { "" });
```

which represents a whitespace token as an empty string; the empty strings are removed from the output by the iterator itself, so the whitespace just disappears. (If this seems like too much of a trick, it's simple to change the code so that the `$maketoken` argument is expected to return a *list* of tokens to be inserted into the output, and then have the `WHITESPACE maketoken()` function return an empty list.)

Similarly, instead of a generic `OPERATOR` token, it might be convenient to include the operator type and its precedence in the token:

```
%optype = ('+'  => ['ARITHMETIC', 3],
           '-'  => ['ARITHMETIC', 3],
```

```
            '*'  => ['ARITHMETIC', 4],
            '/'  => ['ARITHMETIC', 4],
            '**' => ['ARITHMETIC', 5],
            'x'  => ['STRING'    , 4],
            '.'  => ['STRING'    , 3],
            ...
           );
tokens($input, \%optype, qr/\*\*|[-+*\/x.]|.../,
       sub { my $optype = $_[1];
             [ "OPERATOR", @{$optype->{$_[0]}}, $_[0] ]
           });
```

The tokens that come out of this iterator look like `["OPERATOR", "ARITHMETIC", 5, "**"]`.

Lexers are easy to write with the chained-lexer technique, but the resulting code is ugly:

```
my $lexer = tokens(
              tokens(
                tokens(
                  tokens(
                    tokens($input,
                           'TERMINATOR',
                           qr/;\n*|\n+/,
                          ),
                    'INTEGER',
                    qr/\b\d+\b/,
                  ),
                  'PRINT',
                  qr/\bprint\b/,
                ),
                'IDENTIFIER',
                qr|[A-Za-z_]\w*|,
              ),
              'OPERATOR',
              qr#\*\*|[-=+*/()]#,
             ),
             'WHITESPACE',
             qr/\s+/,
             sub { "" },  # discard
            );
```

But a spoonful of syntactic sugar makes the medicine go down:

```
sub make_lexer {
  my $lexer = shift;
  while (@_) {
    my $args = shift;
    $lexer = tokens($lexer, @$args);
  }
  $lexer;
}
```

Now we can build the lexer with a tidy, tabular piece of code:

```
my $lexer = make_lexer($input,
                       ['TERMINATOR', qr/;\n*|\n+/              ],
                       ['INTEGER',    qr/\b\d+\b/               ],
                       ['PRINT',      qr/\bprint\b/             ],
                       ['IDENTIFIER', qr|[A-Za-z_]\w*|          ],
                       ['OPERATOR',   qr#\*\*|[-=+*/()]#        ],
                       ['WHITESPACE', qr/\s+/,         sub { "" } ],
            );
```

Calls to `make_lexer()` can be chained the same way that calls to `tokens()` were.

We'll use this lexer later, so let's observe one feature that you might otherwise miss: A semicolon (possibly followed by newlines, but possibly alone) is lexed as a `TERMINATOR` token, the same as actual newline characters would be.

8.1.4 Peeking

Our lexers need one more feature to be complete. Consider a parser for some kind of expression that has two distinct forms; say a parser for numerals that might look either like `"123"` or like `"-123"`. These will be tokenized as `[INTEGER, "123"]` and as `[OPERATOR, "-"]`, `[INTEGER, "123"]`, respectively.

We would like to build the parser to understand both forms, and for reasons of maintenance and modularity we might like to build it from two separate parts, one that handles unsigned integers and one that handles negated integers. A master control will examine the next token in the input. If it's an `INTEGER`, the master control will invoke the sub-parser for unsigned integers; if it's an `OPERATOR` the master control will invoke the sub-parser for negated integers. If the next token is anything else, the master parser will signal an error.

The problem is that once the master control has eaten the token at the front of the input stream, the token is gone; the sub-parsers no longer have access to it, and the unsigned integer parser certainly needs it, because the token contains the value of the integer. We could fix this by passing the eaten token to the sub-parsers explicitly. However, this will entail complications in the parsers, which will now have to get their inputs from two separate sources.

A simpler approach is to allow the master parser to put the eaten token back into the input stream, or, alternatively, to allow it to examine the next token in an input *without* removing it from the stream. We'll use the second approach:

```
sub tokens {
  ...
  sub {
    while (@tokens == 0 && ! $finished) {
      ...
    }
    return $_[0] eq 'peek' ? $tokens[0] : shift(@tokens);
  }
}
```

Normally, we invoke a lexer as `$lexer->()`, which consumes and returns the next token in the input. With this change, we have the option of saying `$lexer->('peek')`, which returns the next token in the input *without* consuming it; the next call to `$lexer->()` will return the same token we just peeked at.

An alternative approach is to represent lexers as streams, in the sense of Chapter 6. Turning our generic function-style iterators into lazy streams is easy:

CODE LIBRARY
it2stream.pl

```
use Stream 'node';

sub iterator_to_stream {
  my $it = shift;
  my $v = $it->();
  return unless defined $v;
  node($v, sub { iterator_to_stream($it) });
}

1;
```

If we do this, the "peek" function is just `head()`, and the "read and discard" function is `drop()`.

8.2 PARSING IN GENERAL

We've finished with lexers, which transform unstructured character sequences into token sequences. Now we'll deal with parsers, which read sequences of tokens and integrate them into complex structures. The result of parsing is the *meaning* of the input, or the *value* of the input. For example, the result of parsing the input `3 + 4 * 5` might be the number 23, which is the meaning or the value of the expression. If the parser is part of a compiler, the value might be a sequence of machine language instructions for calculating $3 + 4 \cdot 5$.

8.2.1 Grammars

The key to parsing is the *grammar*, which describes the structure of a legal input. As a simple example, we'll consider a parser for a very limited set of arithmetic expressions, including only addition, multiplication, and parentheses. Here is the grammar:

```
expression → atom '+' expression
expression → atom '*' expression
expression → '(' expression ')'
expression → atom

atom → 'INT'
atom → 'VAR'
```

Items in quotes represent literal tokens; items not in quotes represent other parts of the grammar. Both kinds of items are called *symbols*. The first section is the definition of the *expression symbol*. It says that there are four alternative forms for an *expression*. An *expression* might be an atom followed by a `+` token followed by a complete expression, or it might be an atom followed by a `*` token followed by a complete expression, or it might be a complete expression enclosed in parentheses, or it might simply be an atom by itself. What's an atom? The second section defines the *atom symbol*: It's either an `INT` token or a `VAR` token. The four alternatives in the *expression* definition and the two alternatives in the *atom* definition are sometimes called *clauses* or *productions*. One of the symbols is usually considered more important than the others, and represents the entire structure that we're trying to parse; the other symbols usually represent various sub-structures or special cases. In this case the important symbol is *expression*. This more-important symbol is called the *start symbol*.

The grammar defines all the legal expressions, as follows. Start with the start symbol, *expression*. Apply a production by replacing the symbol on the left side of the production with the symbols on the right; for example, replace *expression* with (*expression*). Repeat until there is nothing left but literal tokens. For example,

```
expression
atom * expression                  # expression clause 2
INT  * expression                  # atom clause 1
INT  * ( expression              ) # expression clause 3
INT  * ( atom + expression       ) # expression clause 1
INT  * ( atom + atom * expression ) # expression clause 2
INT  * ( atom + VAR  * expression ) # atom clause 2
INT  * ( atom + VAR  * atom       ) # expression clause 4
INT  * ( VAR  + VAR  * atom       ) # atom clause 2
INT  * ( VAR  + VAR  * INT        ) # atom clause 1
```

This *derivation* shows that the sequence of tokens `INT * ( VAR + VAR * INT )` is a valid expression. The valid expressions are exactly those for which derivations exist. `INT INT INT` and `INT + + VAR` are not valid expressions, according to our grammar, because there are no derivations for them.

Now some jargon: The symbols that appear on the left side of productions, such as *expression* and *atom*, are called *nonterminal symbols* or just *nonterminals*, because a derivation is not complete as long as one of them appears in the result. Conversely, the literal tokens are called *terminal symbols* or just *terminals* because once they appear, they can't be replaced. A sequence of symbols is sometimes called a *sentential form*; if all the symbols are terminals, the sentential form is a *sentence*. So a derivation of a particular sentence consists of a sequence of sentential forms, each of which is obtained from the previous one by the application of one of the productions to one of the nonterminal symbols.

One of the primary jobs of parsing is to determine if a particular sequence of tokens is a sentence according to a given grammar, and, if so, which productions to apply in which order to produce it. In principle, this is simple. The space of all possible sentential forms has a tree structure, and we need only do a tree search on this tree, looking for the sentential form that corresponds to the input tokens. Figure 8.1 shows the top portion of the tree for the *expression* symbol in the example grammar.

The root node is the sentential form that consists only of the start symbol, *expression*. At each node, we obtain the child nodes by replacing the leftmost nonterminal symbol with the right-hand side of one if its productions. When there are no nonterminal symbols, the node is a leaf. A path from the root to

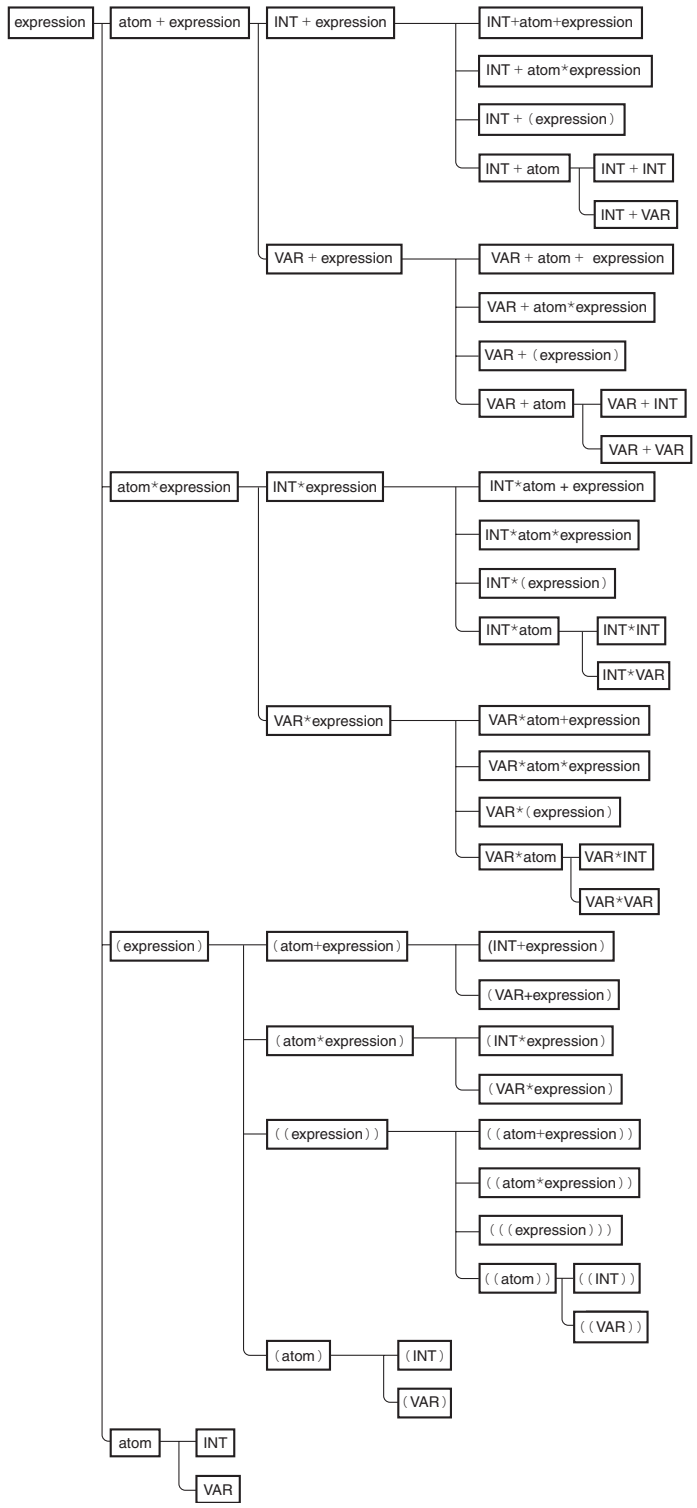


FIGURE 8.1 Part of the search space of all possible expressions.

a particular leaf gives a derivation of the sentence at that leaf. For example, by tracing backwards from the `INT+INT` leaf to the root, we find a derivation of the sentence `INT+INT`:

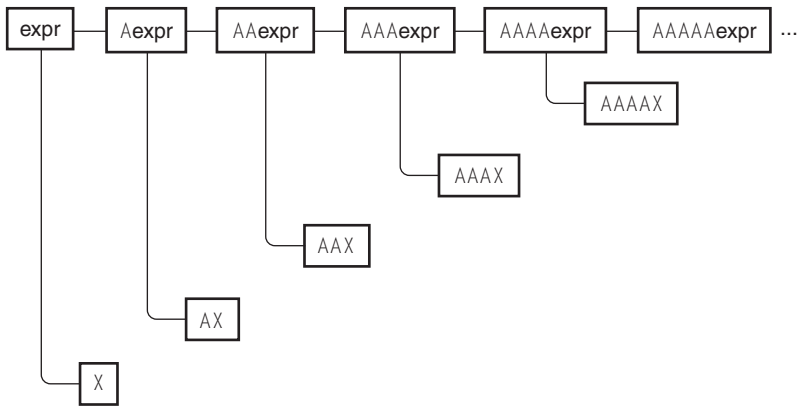
```
expression
atom  '+' expression          # expression clause 1
'INT' '+' expression          # atom clause 1
'INT' '+' atom                # expression clause 4
'INT' '+' 'INT'               # atom clause 1
```

Finding a derivation for a particular sentence, therefore, is equivalent to locating the sentence in the tree, and therefore to a tree search.

Breadth-first search will eventually find the derivation, if it exists, but, as is usual with breadth-first search, consumes a lot of memory, and so isn't used very often. Depth-first search is cheaper, but since the tree is typically infinite, depth-first search can go charging down one of the infinite branches, never to return. To illustrate the potential problem and its solution, here's a particularly simple example grammar:

```
expr → 'A' expr
expr → 'X'
```

An *expr* is either an `X` token or else an `A` token followed by a complete *expr*. Expressions (*expr*) in this grammar therefore consist of some number of `A`'s followed by an `X`. The derivation tree looks like this:



A naive depth-first search will go charging down the topmost branch, looking for a complete sentence, and never find one. It will never investigate any of the

other branches. But there's an obvious way to prune the search. Suppose the search is looking for AX. The search first examines the Aexpr node. Each node below this one will begin with A. The target sentence, AX, also begins with A, so the search continues. The next node is AAexpr. Every node below *this* one will begin with AA. The target sentence does not begin with AA, so the search can ignore this entire branch, moving over to the sibling branch, AX. The search then succeeds. The tree is infinite, but the search process can trim out large infinite portions of it.

For an analogous example with a more typical grammar, consider our original grammar example. Suppose the target sentence is simply VAR. The search proceeds down the first branch, *atom*+*expression*, to the first of its two sub-nodes, which is INT+*expression*. Each node below this one must begin with INT +, and the target sentence does not, so the search can skip all the lower nodes, moving instead to the VAR+*expression* branch. Each node below this one begins with VAR +, and the target sentence does not, so the search again skips all the lower nodes, moving instead to the *atom***expression* branch, which fails similarly. The third branch, (*expression*), fails even more quickly, since the search knows that all the lower nodes on that branch will begin with (, and the target sentence does not. Finally, the search moves to the fourth branch, *atom*, tries the first sub-node, INT, which fails, and then the second sub-node, VAR, which is the sentence that was being sought.

8.2.2 Parsing Grammars

One way to implement a parser is the same way we've done tree search in previous chapters. The search maintains an agenda of sentential forms that need to be investigated. Initially, the agenda contains only the sentential form consisting of just the start symbol. At each stage, the search pops a sentential form off the agenda list. If the sentential form is a sentence, the search succeeds if it is the correct sentence, and continues otherwise. If the sentential form is not a sentence, but begins with some terminal symbols, the search checks to make sure that the target sentence begins with the same terminals. If not, the sentential form is discarded, pruning the search. When there are no sentential forms remaining in the agenda, the search fails.

If the sentential form popped from the agenda is not discarded, the search locates the leftmost nonterminal symbol in the sentential form — let's say the sentential form is VAR + *expression*, so the leftmost nonterminal symbol is *expression*. The search looks up all the productions whose left side is this nonterminal symbol, *expression*. In our example, there are four of these. It adds a new item to the agenda for each one, obtaining the new item by replacing the nonterminal symbol in the

left side of the production with the sentential form on the right. In our example, the search adds the four items `VAR + ` *atom* ` + ` *expression*, `VAR + ` *atom* ` * ` *expression*, `VAR + ( ` *expression* ` )`, and `VAR + ` *atom* to the agenda. The process then repeats.

Here's sample code, using the `make_dfs_search()` function of Section 5.3. We will represent a grammar as a hash:

```
expression => [['INT', '+', 'expression'],
               ['INT', '*', 'expression'],
               ['(', 'expression', ')'],
               ['INT'],
              ]
```

Keys are nonterminal symbols; values are arrays of productions. To distinguish a terminal from a nonterminal symbol, we just look it up in the hash:

CODE LIBRARY
DFSParser.pm

```
require "make-dfs-search";

sub make_parser_for_grammar {
  my ($start, $grammar, $target) = @_;

  my $is_nonterminal = sub {
    my $symbol = shift;
    exists $grammar->{$symbol};
  };
```

When the search finds a sentential form, it scans the form to see if it matches the target sentence, stopping at the first nonterminal symbol. If the sentential form is too long or too short, or if one of its leading tokens doesn't match the corresponding token in the target sentence, then the search won't mention it:

```
  my $is_interesting = sub {
    my $sentential_form = shift;
    for my $i (0 .. $#$sentential_form) {
      return 1 if $is_nonterminal->($sentential_form->[$i]);
      return if $i > $#$target;
      return if $sentential_form->[$i] ne $target->[$i];
    }
    return @$sentential_form == @$target ;
  };
```

Given a sentential form, we find the children in the tree by locating the leftmost nonterminal symbol and replacing it with each of its productions from the grammar.

First we locate the leftmost nonterminal symbol:

```
my $children = sub {
  my $sentential_form = shift;
  my $leftmost_nonterminal;
  my @children;

  for my $i (0 .. $#$sentential_form) {
    if ($is_nonterminal->($sentential_form->[$i])) {
      $leftmost_nonterminal = $i;
      last;
```

If the sentential form is too long to match the target, we prune the tree at that point by reporting that it has no children:

```
    } else {
      return if $i > $#$target;
```

Similarly, if the initial tokens of the sentential form don't match the initial tokens of the target sentence, we prune:

```
      return if $target->[$i] ne $sentential_form->[$i];
    }
  }
```

If a node has no nonterminal symbols, it is a leaf and has no children:

```
  return unless defined $leftmost_nonterminal; # no nonterminal symbols
```

Having located the leftmost nonterminal symbol, we generate the child nodes by replacing the nonterminal with each of the possible productions for it:

```
  for my $production (@{$grammar->{$sentential_form->[$leftmost_nonterminal]}}) {
    my @child = @$sentential_form;
    splice @child, $leftmost_nonterminal, 1, @$production;
    push @children, \@child;
  }
  @children;
};
```

The parser itself uses `make_dfs_search()` (See Section 5.3) to do a DFS search of the space of sentential forms. The root node is the sentential form containing only the start symbol:

```
  return sub {
    make_dfs_search([$start], $children, $is_interesting);
  };
}

1;
```

To use this, we say something like:

```
my $parser = make_parser_for_grammar 'expression',
  {
   expression => [['INT', '+', 'expression'],
                  ['INT', '*', 'expression'],
                  ['(', 'expression', ')'],
                  ['INT'],
                 ],

  },
  ['(', 'INT', '*', '(', 'INT', '+', 'INT', ')', ')']
;
```

The target sentence here is `( INT * ( INT + INT ) )`.

```
my $parses = $parser->();

while (my $parse = $parses->()) {
  print "@$parse\n";
}
```

The output is:

```
expression
( expression )
( INT * expression )
( INT * ( expression ) )
( INT * ( INT + expression ) )
( INT * ( INT + INT ) )
```

which is indeed a derivation of the target sentence.

If we try parsing a non-sentence, say `INT * + INT`, we get:

```
expression
INT * expression
```

The target sentence never comes out, because the parser got stuck. It wanted to find a way to make `+ INT` into an expression, but it couldn't.

8.3 RECURSIVE-DESCENT PARSERS

A commonly used technique for parsing by DFS is to move the responsibility for the search back into the Perl function-call mechanism. A parser becomes a function that takes an input stream and examines its front for a certain pattern of tokens and values. If it likes what it sees, it returns two things: the value or meaning it assigns to the tokens, and the unused portion of the input. If the function doesn't like what it sees, it returns false. If a parser represents a compound expression, it may call sub-parsers to help it decide if the input is in the proper form; this is where the DFS arises. Parsers that call each other recursively to parse an input in this way are called *recursive-descent parsers*.

We'll develop a module, `Parser.pm`, with tools for manufacturing recursive-descent parsers. The module starts off with the usual declarations:

CODE LIBRARY
Parser.pm

```
package Parser;
use Stream ':all';
use base Exporter;
@EXPORT_OK = qw(parser nothing End_of_Input lookfor
                alternate concatenate star list_of
                operator T
                error action test);
%EXPORT_TAGS = ('all' => \@EXPORT_OK);

sub parser (&);   # Advance declaration - see below
```

8.3.1 Very Simple Parsers

The simplest parser of all consumes no input and always succeeds, yielding the value `undef`:

```
sub nothing {
  my $input = shift;
```

```
  return (undef, $input);
}
```

The next simplest just checks for the end of the input:

```
sub End_of_Input {
  my $input = shift;
  defined($input) ? () : (undef, undef);
}
```

If the input is undefined (that is, empty), then the function returns success: the value `undef`, and the remaining unread input, also `undef` (still empty). Otherwise, the function returns an empty list to indicate failure.

The next-simplest parsers look for single specific tokens. For example, a parser that succeeds if the next token is an `INT`:

```
sub INT {
  my $input = shift;
  return unless defined $input;
  my $next = head($input);
  return unless $next->[0] eq 'INT';
  my $token_value = $next->[1];
  return ($token_value, tail($input));
}
```

If the input is empty, it fails immediately. Otherwise, it examines the token at the head of the input to see if it is an `INT` token; if not, it fails. But if the next token is an `INT` token, the function returns the token's numeric value (stored in `$token->[1]`) and the remaining input. The value returned by this parser is a number, namely the value one would expect the integer token to represent.

Since we'll need many of these token-recognizing functions, we'll build them with a function factory:

```
sub lookfor {
  my $wanted = shift;
  my $value = shift || sub { $_[0][1] };
  my $u = shift;
  $wanted = [$wanted] unless ref $wanted;
  my $parser = parser {
    my $input = shift;
```

```
    return unless defined $input;
    my $next = head($input);
    for my $i (0 .. $#$wanted) {
      next unless defined $wanted->[$i];
      return unless $wanted->[$i] eq $next->[$i];
    }
    my $wanted_value = $value->($next, $u);
    return ($wanted_value, tail($input));
  };

  return $parser;
}
```

To generate a function that looks for an `[OP +]` token, we can call `lookfor(['OP', '+'])`. The generated parser will examine all the elements of `$wanted`, the argument, and will succeed if all the specified components in `$wanted` match the actual components of the next input token in `$next`. To generate the preceding `INT` function, we could call `lookfor(['INT'])`. `lookfor('INT')` is a shorthand for `lookfor(['INT'])`.

`lookfor()` gets an optional second argument, which is a callback function that turns the token into a value, and an optional third argument, which is a user parameter to the callback. The default callback extracts the second element from the token; with the lexers we've been using, this is always the literal text of the token.

The `parser()` function takes a block of code and builds a parser from it. Right now, it does nothing:

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

Later on, it will have some additional behavior.

8.3.2 Parser Operators

Now that we have some simple parsers, how can we put parsers together? The most obvious thing to do with two parsers is to call them in sequence on the input. Suppose we have the following grammar:

```
doorbell → DING DONG
```

We need a parser that looks for `DING`, and then if it finds `DING` it looks for `DONG`, and if it finds `DONG`, it succeeds. This is called the *concatenation* of the two components.

If we have a parser function that looks for `DING` and one that looks for `DONG`, here's a function to concatenate them into a function that looks for `doorbell`:

```
sub concatenate {
  my ($p1, $p2) = @_;
  my $parser = parser {
    my $input0 = shift;
    my ($v1, $input1) = $p1->($input0) or return;
    my ($v2, $input2) = $p2->($input1) or return;
    return ([$v1, $v2], $input2);
  }
}
```

The value returned by the concatenation of the two parsers is an array containing the values returned by the parsers individually. The parser for the `doorbell` symbol can be built by saying:

```
$doorbell = concatenate(lookfor('DING'),
                        lookfor('DONG'));
```

If given an input that begins with tokens `DING DONG ...`, this parser returns the value `['DING', 'DONG']` and the remaining tokens.

It's easy to generalize concatenation to more than two parsers:

```
sub concatenate {
  my @p = @_;
  return \&nothing if @p == 0;

  my $parser = parser {
    my $input = shift;
    my $v;
    my @values;
    for (@p) {
      ($v, $input) = $_->($input) or return;
      push @values, $v;
    }
   return (\@values, $input);
  }
}
```

The other important operation on parsers is *alternation*, which looks for an input in one of several alternative forms. For example, `alternate(lookfor('DING'),`

`lookfor('DONG')`) succeeds if the next token in the input is `DING` *or* `DONG` and fails otherwise. Here's a two-parser version of `alternate()`:

```
sub alternate {
  my ($p1, $p2) = @_;
  my $parser = parser {
    my $input = shift;
    my ($v, $newinput);
    if (($v, $newinput) = $p1->($input)} { return ($v, $newinput) }
    if (($v, $newinput) = $p2->($input)} { return ($v, $newinput) }
    return;
  };
}
```

The new parser tries running `$p1` on the input; if `$p1` succeeds, the new parser just returns whatever `$p1` returned, effectively behaving like `$p1`. If `$p1` fails, the new parser tries `$p2` in the same way, behaving like `$p2` instead. If they both fail, then the new parser indicates failure also.

It's even easier to generalize `alternate()`:

```
sub alternate {
  my @p = @_;
  return parser { return () } if @p == 0;
  return $p[0]                if @p == 1;
  my $parser = parser {
    my $input = shift;
    my ($v, $newinput);
    for (@p) {
      if (($v, $newinput) = $_->($input)) {
        return ($v, $newinput);
      }
    }
    return;
  };
}
```

The only fine point here is that if there are no alternatives at all (`@p == 0`), then `alternate()` returns a parser that never succeeds (`parser { return () }`).

8.3.3 Compound Operators

Having written functions to build parsers to handle alternatives and concatenations of other parsers, it's now easy to build more powerful operators for parsers.

For example, we can build an operator that's analogous to the regex * operator. If the parser `$P` matches a certain kind of input, say `DING DONG`, then `star($P)` will be a parser that matches an empty input, `DING DONG`, `DING DONG DING DONG`, `DING DONG DING DONG DING DONG`, and so on. The first cut is:

```
sub star {
  my $p = shift;
  my $p_star;
  $p_star = alternate(concatenate($p, $p_star),
                      \&nothing);
}
```

The idea here is to match an input of the form matched by `$p`, followed by something else (possibly empty) of the form `star($p)`, or, if we can't find anything acceptable to `$p`, match nothing.

This function doesn't quite work, because at the time `concatenate()` is called, its argument `$p_star` is still undefined. There are several tricks we can use to work around this. The simplest is:

```
sub star {
  my $p = shift;
  my $p_star;
  $p_star = alternate(concatenate($p, parser { $p_star->(@_) }),
                      \&nothing);
}
```

We've replaced `$p_star` on the right-hand side with a new parser that just calls `$p_star` on its arguments and returns the result. Behaviorally, the new parser is interchangeable with `$p_star` itself. But the `$p_star` on the right isn't evaluated until the new parser is called, by which time `$p_star` has the right value. This trick goes by the peculiar name of *eta-conversion*.

An alternative formulation would be:

```
sub star {
  my $p = shift;
  my $p_star;
  $p_star = parser { alternate(concatenate($p, $p_star),
                               \&nothing)->(@_) };
}
```

which works for essentially the same reason. But it will be less efficient than the previous version, because it will call `alternate()` and `concatenate()` each time it is called.

Using our parser operators, we can build larger operators whenever it's convenient. A common feature of programming languages is lists of various sorts. For example, Perl has list expressions:

```
@items = ($expression1, $expression2, $expression3);
```

and blocks:

```
$block = sub { $statement1; $statement2; $statement3 };
```

Here's a parser operator for parsing lists of elements separated by some kind of separator sequence:

```
sub list_of {
  my ($element, $separator) = @_;
  $separator = lookfor('COMMA') unless defined $separator;
  return concatenate($element,
                     star(concat($separator, $element));
}

1;
```

Now we can make a parser for lists of expressions:

```
$expression_list = list_of($expression);
```

or for lists of statements:

```
$statement_list = list_of($statement, lookfor('SEMICOLON'));
```

8.4 ARITHMETIC EXPRESSIONS

In Section 8.2, we saw a grammar for a simple subset of arithmetic expressions:

```
expression → atom '+' expression
expression → atom '*' expression
```

```
expression → '(' expression ')'
expression → atom
atom → 'INT'
atom → 'VAR'
```

Let's use an even simpler example, which has only numbers, and no variables:

```
expression → 'INT' '+' expression
expression → 'INT' '*' expression
expression → '(' expression ')'
expression → 'INT'
```

There's also an additional rule that represents an entire input:

```
entire_input → expression 'End_of_Input'
```

We will build a parser for this example, and later add the other required features, like subtraction. Transforming the grammar into a parser is simple. We build one parser function for each nonterminal symbol. We use `alternate()` when a symbol has several alternative definitions, and `concatenate()` when a definition is the concatenation of more than one token or symbol. When the grammar mentions that a symbol might contain a token, we call `lookfor()` to build a parser that looks for that token; when the grammar mentions that a symbol might contain an instance of some other nonterminal symbol, we invoke the parser for that symbol.

To transform the preceding grammar, we first transform the definition of `expression` into a parser for expressions:

```
my $expression;
$expression = alternate(concatenate(lookfor('INT'),
                                    lookfor(['OP', '+']),
                                    $expression),
                        concatenate(lookfor('INT'),
                                    lookfor(['OP', '*']),
                                    $expression),
                        concatenate(lookfor(['OP', '(']),
                                    $expression,
                                    lookfor(['OP', ')'])),
                        lookfor('INT'));
```

Again, this doesn't quite work. We can't use `$expression` in its own definition, because until we've defined it, it's `undef`. The eta-conversion trick works

just fine here:

CODE LIBRARY
expr-parser.pl

```
use Parser ':all';
use Lexer ':all';

my $expression;
my $Expression = parser { $expression->(@_) };
$expression = alternate(concatenate(lookfor('INT'),
                                    lookfor(['OP', '+']),
                                    $Expression),
                        concatenate(lookfor('INT'),
                                    lookfor(['OP', '*']),
                                    $Expression),
                        concatenate(lookfor(['OP', '(']),
                                    $Expression,
                                    lookfor(['OP', ')'])),
                        lookfor('INT'));
```

Defining a parser for `entire_input` is simple:

```
my $entire_input = concatenate($Expression, \&End_of_Input);
```

Suppose the input is the string `"2 * 3 + (4 * 5)"`, and we use the lexer of Section 8.1.3:

```
my @input = q[2 * 3 + (4 * 5)];
my $input = sub { return shift @input };

my $lexer = iterator_to_stream(
               make_lexer($input,
                       ['TERMINATOR', qr/;\n*|\n+/              ],
                       ['INT',        qr/\b\d+\b/               ],
                       ['PRINT',      qr/\bprint\b/             ],
                       ['IDENTIFIER', qr|[A-Za-z_]\w*|          ],
                       ['OP',         qr#\*\*|[-=+*/()]#        ],
                       ['WHITESPACE', qr/\s+/, sub { "" }       ],
               )
             );

if (my ($result, $remaining_input) = $entire_input->($lexer)) {
  use Data::Dumper;
```

```
  print Dumper($result), "\n";
} else {
  warn "Parse error.\n";
}
```

The parser does succeed, returning the following `$result`:

```
[
  [
    '2',
    '*',
    [
      '3',
      '+',
      [
        '(',
        [
          '4',
          '*',
          '5'
        ],
        ')'
      ]
    ]
  ],
  undef
]
```

Each of the three-element arrays was returned by `concatenate()`, which returned an array of the three values that it concatenated. The trailing `undef` was returned by the `End_of_Input()` parser and concatenated into the final result by the `concatenate()` in:

```
my $entire_input = concatenate($Expression, \&End_of_Input);
```

There are a couple of problems with this result. What we'd most like is to have the parser generate an abstract syntax tree (AST), as in Section 2.2, where each node is labeled with an operator, and has child nodes representing the operands of that operator. We'd like the parser to understand that operators have different precedences — that `2 * 3 + 4` should be parsed the same as `(2 * 3) + 4` but differently from `2 * (3 + 4)`. The parentheses shouldn't appear literally in the

output; they should affect only the results of the parse. The value we'd like to get out of the parser for the input `"2 * 3 + (4 * 5)"` is:

```
[ '+',
  [ '*', 2, 3 ],
  [ '*', 4, 5 ]
]
```

which says that the expression is adding two quantities: the product of 2 and 3, and the product of 4 and 5. Similarly, `2 * 3 + 4` and `(2 * 3) + 4` should produce:

```
[ '+', ['*', 2, 3], 4 ]
```

but `2 * (3 + 4)` should produce:

```
[ '*', 2, ['+', 3, 4] ]
```

We'll tackle the precedence issue first. There are a few ways to take care of this, but the quickest one is to make a small change to the grammar. We can think of an expression like `"2 * 3 + (4 * 5)"` as a sum of one or more *terms*, where each term is a product of one or more *factors*. If we rewrite the grammar to express this, the precedence will take care of itself:

```
entire_input → expression 'End_of_Input'

expression → term '+' expression | term

term → factor '*' term | factor

factor → 'INT' | '(' expression ')'
```

The notation:

```
a → b | c
```

is shorthand for the two rules:

```
a → b
a → c
```

so the grammar says that an expression is either a plain term, or a term plus a complete expression. A term is either a plain factor, or a factor times a complete term. Finally, a factor is either an integer token or else a complete expression enclosed in parentheses.

Translating this grammar into code gives us:

CODE LIBRARY
expr-parser-2.pl

```
use Parser ':all';
use Lexer ':all';
my ($expression, $term, $factor);
my $Expression = parser { $expression->(@_) };
my $Term       = parser { $term      ->(@_) };
my $Factor     = parser { $factor    ->(@_) };
$expression = alternate(concatenate($Term,
                                    lookfor(['OP', '+']),
                                    $Expression),
                        $Term);


$term       = alternate(concatenate($Factor,
                                    lookfor(['OP', '*']),
                                    $Term),
                        $Factor);

$factor     = alternate(lookfor('INT'),
                        concatenate(lookfor(['OP', '(']),
                                    $Expression,
                                    lookfor(['OP', ')'])),
                        );


$entire_input = concatenate($Expression, \&End_of_Input);
```

The output of the parser on `"2 * 3 + (4 * 5)"` is now:

```
[
  [
    [
      '2',
      '*',
      '3'
    ],
    '+',
```

```
      [
        '(',
        [
          '4',
          '*',
          '5'
        ],
        ')'
      ]
    ],
    undef
]
```

Not all the problems have been fixed, but the multiplication arguments have been clustered together, as we wanted; 2 is now associated with 3, and 4 with 5. Similarly, `"2 * 3 + 4"` produces:

```
[
  [
    [
      '2',
      '*',
      '3'
    ],
    '+',
    '4'
  ],
  undef
]
```

with the 2 and 3 correctly grouped; the previous grammar produced this incorrect parse instead:

```
[
  [
    '2',
    '*',
    [
      '3',
      '+',
      '4'
```

```
    ]
  ],
  undef
]
```

Now we'll fix the parsers so that they generate proper abstract syntax trees. This isn't hard. All the necessary information is available; we just need to arrange it correctly. In a parser definition like this one:

```
$term       = alternate(concatenate($Factor,
                                    lookfor(['OP', '*']),
                                    $Term),
                        $Factor);
```

the `concatenate()` operator assembles the values returned by its three arguments in the order they're listed. But for an abstract syntax tree, we want the operator first, not second. Similarly, in:

```
$factor     = alternate(lookfor('INT'),
                        concatenate(lookfor(['OP', '(']),
                                    $Expression,
                                    lookfor(['OP', ')'])
                        );
```

the `concatenate()` operator assembles all three values, but we're interested only in the middle one, not the parentheses themselves. The value of a factor of the form `'(' expression ')'` should be the same as the value of the inner expression.

The following function is a little bit like `map` for parsers generated by `concatenate()`. It takes a parser and a transformation function, and returns a new parser, which recognizes the same inputs, but whose return value is filtered through the transformation function. It is named `T`, which is short for "transform":

```
sub T {
  my ($parser, $transform) = @_;
  return parser {
    my $input = shift;
    if (my ($value, $newinput) = $parser->($input)) {
      $value = $transform->(@$value);
      return ($value, $newinput);
```

```
    } else {
      return;
    }
  };
}
```

For example, to get `$factor` to throw away the parentheses and return only the inner expression, we replace this:

```
$factor     = alternate(lookfor('INT'),
                        concatenate(lookfor(['OP', '(']),
                                    $Expression,
                                    lookfor(['OP', ')'])),
                        );
```

with this:

```
$factor     = alternate(lookfor('INT'),
                        T(
                          concatenate(lookfor(['OP', '(']),
                                      $Expression,
                                      lookfor(['OP', ')'])),
                          sub { $_[1] })
                       );
```

The three values accumulated by `concatenate()` are passed to the anonymous subroutine, which returns only the second one.

Similarly, to get `$term` to assemble the operator and operands in that order, we use:

```
$term       = alternate(
                        T(
                          concatenate($Factor,
                                      lookfor(['OP', '*']),
                                      $Term),
                          sub { [ $_[1], $_[0], $_[2] ]}),
                        $Factor);
```

For the term `3 * 4` the three arguments to the anonymous subroutine will be `(3, '*', 4)` and the return value will be `['*', 3, 4]`. We should make a similar change to the definition of `$expression`, and we can eliminate the spurious

trailing `undef` by changing `$entire_input` to:

```
my $entire_input = T(concatenate($Expression, \&End_of_Input),
                     sub { $_[0] }
                    );
```

With these changes, the output of the parser, given the input `"2 * 3 + (4 * 5)"`, is perfect:

```
[
  '+',
  [
    '*',
    '2',
    '3'
  ],
  [
    '*',
    '4',
    '5'
  ]
]
```

`"2 * 3 + 4"` and `"(2 * 3) + 4"` both come back as:

```
[
  '+',
  [
    '*',
    '2',
    '3'
  ],
  '4'
]
```

but `"2 * (3 + 4)"`, which is different, comes back as:

```
[
  '*',
  '2',
  [
    '+',
```

```
      '3',
      '4'
    ]
  ]
```

8.4.1 A Calculator

By adjusting the transformation functions, we can turn our parser into a calculator instead of an abstract-syntax-tree-maker. The value returned by each parser is an abstract syntax tree for some part of the expression. We need to change the parsers to return numeric values instead of AST values. Only two changes are necessary:

```
$expression = alternate(T(concatenate($Term,
                                      lookfor(['OP', '+']),
                                      $Expression),
                          sub { $_[0] + $_[2] }),
                        $Term);
$term       = alternate(T(concatenate($Factor,
                                      lookfor(['OP', '*']),
                                      $Term),
                          sub { $_[0] * $_[2] }),
                        $Factor);
```

The values returned by the parsers are no longer arrays; now they're numbers. When the parser sees an expression like `term + expression`, instead of building an abstract-syntax-tree node out of the values of the term and the expression, it simply adds the values numerically and returns the sum. Similarly, when computing the value of a term, it just does numeric multiplication on the constituents of the term.

The output for `"2 * (3 + 4)"` is now just the number 14, and the output for `"2 * 3 + 4"` and `"(2 * 3) + 4"` is 10. The parser returns failure when given an input with mismatched or unbalanced parentheses, or any other syntactically incorrect input, such as an input with two consecutive operators or numbers.

8.4.2 Left Recursion

Let's add subtraction and division to the calculator. This requires only small changes to the grammar:

```
entire_input → expression 'End_of_Input'
expression → term '+' expression
```

```
expression → term '-' expression
expression → term
term → factor '*' term
term → factor '/' term
term → factor
factor → 'INT' | '(' expression ')'
```

In the code, the definition of `$expression` changes from this:

```
$expression = alternate(concatenate($Term,
                                    lookfor(['OP', '+']),
                                    $Expression),
                        $Term);
```

to this:

```
$expression = alternate(concatenate($Term,
                                    lookfor(['OP', '+']),
                                    $Expression),
                        concatenate($Term,
                                    lookfor(['OP', '-']),
                                    $Expression),
                        $Term);
```

and `$term` changes similarly. We can get the calculator to calculate numeric values by supplying transformation functions like these:

```
$expression = alternate(T(concatenate($Term,
                                      lookfor(['OP', '+']),
                                      $Expression),
                          sub { $_[0] + $_[2] }),
                        T(concatenate($Term,
                                      lookfor(['OP', '-']),
                                      $Expression),
                          sub { $_[0] - $_[2] }),
                        $Term);
```

But now there's a problem: If we ask for the value of `"8 - 4"`, we get the right answer. But if we ask for the value of `"8 - 4 - 3"`, we get 7; the correct answer is 1. What's gone wrong?

If we return to the AST version of the program, we can see the problem. `"8 - 4 - 3"` is parsed as:

```
[ '-', 8, [ '-', 4, 3 ]]
```

which is the same as the parse of `"8 - (4 - 3)"`. But `"8 - 4 - 3"` is conventionally understood to mean `"(8 - 4) - 3"`, which parses as:

```
[ '-', [ '-', 8, 4], 3 ]
```

We say that subtraction *associates from left to right* because, in the absence of parentheses, multiple subtractions are performed from left to right.

The essential problem is that there are two different ways to understand an expression like `"8 - 4 - 3"`. We can parse it as an expression (`"8 - 4"`) minus a term (`"3"`) or as a term (`"8"`) minus an expression (`"4 - 3"`). The convention says that it should be the former, but our grammar rule:

```
expression → term '-' expression
```

says it's the latter. The problem didn't arise for addition; the values of the two parses were numerically the same because addition is associative, which means that we get the same answers whether we consider it to associate from left to right or from right to left.

It might seem that we could fix this by reversing the order of the expression and term symbols in the grammar, as follows:

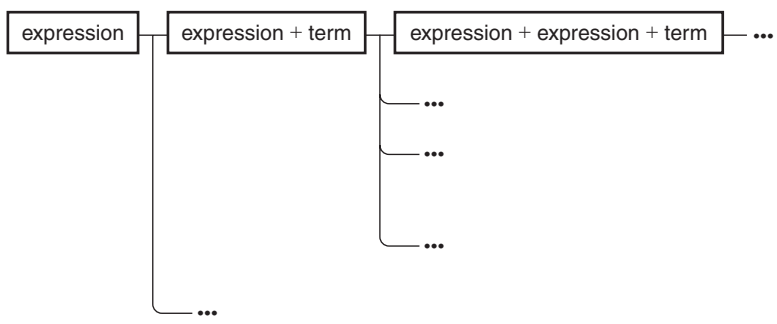
```
expression → expression '+' term
expression → expression '-' term
expression → term
```

Then the corresponding parser definition in the code becomes:

```
$expression = alternate(T(concatenate($Expression,
                                      lookfor(['OP', '+']),
                                      $Term),
                            sub { $_[0] + $_[2] }),
                        T(concatenate($Expression,
                                      lookfor(['OP', '-']),
                                      $Term),
                            sub { $_[0] - $_[2] }),
                        $Term);
```

Unfortunately, if we do this, the parser no longer works. The first thing that the `$expression` parser does is to look for another expression, so it recurses forever. Returning to the sentential form trees of Section 8.2.1, we see that this

is because there's a branch that looks like this:



The topmost branch is infinite, so we run the risk that a depth-first search on it will never terminate. We'd been dealing with this risk by pruning the search at each node whose initial terminal symbols failed to match the target sentence. But here, none of the nodes in the topmost branch have any initial terminal symbols, so the pruning doesn't work, and the search *is* infinite.

A grammar rule of the form:

```
symbol → symbol stuff...
```

is called *left-recursive*; recursive-descent parsers hang forever whenever they meet left-recursive grammar rules. In general, left-recursion can be quite complicated; consider:

```
A → B stuff
B → C stuff
C → A stuff
```

Our example is much simpler; the recursion loop involves only one symbol instead of three. Fortunately, there's a technique for eliminating simple loops from a grammar. If we have a set of rules like this:

```
symbol → symbol A | symbol B | X | Y
```

We can transform them into equivalent rules that aren't left-recursive. Let's consider what a *symbol* must begin with. Clearly, it must be an `X` or a `Y`. *symbol* might

expand to something more complicated that also begins with a *symbol*, but in that case we just have to ask the same question again. So a *symbol* must be an `X` or a `Y` followed by a tail of `A`'s and `B`'s. The equivalent grammar is:

```
symbol → X symbol_tail | Y symbol_tail

symbol_tail → A symbol_tail | B symbol_tail | (nothing)
```

For example, our left-recursive subtraction and addition rule:

```
expression → expression '+' term
expression → expression '-' term
expression → term
```

becomes:

```
expression → term expression_tail

expression_tail → '+' term expression_tail
expression_tail → '-' term expression_tail
expression_tail → (nothing)
```

expression_tail here represents the "rest of an expression." A full expression is a single term followed by the rest of an expression; the rest of the expression is either empty, or else a + or - followed by another term followed by the rest of the rest of the expression. If we apply a similar transformation to the *term* rule of the grammar, we get:

```
entire_input → expression 'End_of_Input'
expression → term expression_tail
expression_tail → '+' term expression_tail
expression_tail → '-' term expression_tail
expression_tail → (nothing)

term → factor term_tail
term_tail → '*' factor term_tail
term_tail → '/' factor term_tail
term_tail → (nothing)

factor → 'INT'
factor → '(' expression ')'
```

Now, however, we have another problem. Before, we had rules like:

```
expression → term '+' expression
```

and it was obvious how to compute the value of an expression, given the values of the subterm and the subexpression: Just add them. But what value should we assign to an *expression_tail?* It represents an incomplete expression, such as `"+ 3 - 4"`. When we're parsing an expression like `"1 + 3 - 4"`, the `"1"` will become the *term* part at the beginning of the *expression* rule, and the `"+ 3 - 4"` will become the *expression_tail* part at the end of the *expression* rule. The value of the *expression* can't be calculated until we finish parsing the *expression_tail* and assign a value to it, because the value of the *expression_tail* will be passed to the *expression* rule for the computation of *its* value. But the value of an *expression_tail* can't be a number, because it's an incomplete expression; the parser won't know what number it is until it finished parsing the complete expression.

The natural way to represent the result of an incomplete computation such as an *expression_tail* is as a function. An expression tail is like an expression with a blank space at the beginning:

```
_____ + 3 - 4
```

which can be represented as the function:

```
sub { $_[0] + 3 - 4 }
```

When the missing parts of the expression become known, they can be passed as arguments to the function, which will compute the final result. The value of an *expression_tail* will be a function, which, given the missing part of the expression, computes the final result. The action rule for the complete expression will get a *term* value and an *expression_tail* function, and will pass the value to the function to compute the final result.

As an example, let's consider `"5 + 7"`. Let's also disregard multiplication and division for a little while; our grammar will be:

```
expression → term expression_tail
expression_tail → '+' term expression_tail
                | '-' term expression_tail
                | (nothing)
term → 'INT' | '(' expression ')'
```

If `"5 + 7"` is to be an *expression*, it must be a *term* followed by an *expression_tail*. The *term* is clearly `"5"`, and has a value of 5.

The *expression_tail* matches the first alternative, and is a "+" followed by a *term* followed by another *expression_tail*. The *term* here is clearly `"7"`, and the following `expression_tail` is empty. The main *expression_tail* represents the partial input `"+ 7"`, and so its value should be a function that adds 7 to its argument: `sub { $_[0] + 7 }`.

The main *expression* now gets two values, which are 5 and `sub { $_[0]+ 7 }`. It passes the number to the function; the result is the value of the expression, 12.

To program *expression*, we write:

```
$expression = T(concatenate($Term, $Expression_Tail),
                sub { $_[1]->($_[0]) }
               );
```

It gets the value of the term (`$_[0]`), which is a number, and the value of the partial expression *expression_tail* (`$_[1]`), which is a function. It passes the term value to the expression tail function, and returns the result.

The code for *expression_tail* is a little more complicated. Let's consider just the case for the rule:

```
expression_tail → '+' term expression_tail
```

The parser code looks like this:

```
$expression_tail = alternate(T(concatenate(lookfor(['OP', '+']),
                                           $Term,
                                           $Expression_Tail),
                               sub { ... }),
```

Suppose the input has this form:

```
LEFT + term RIGHT
```

Say that `$term_value` is the number value returned by the `$Term` parser for the *term* part of the input, and `$rest` is the function value returned by the `$expression_tail` parser for the *RIGHT* part of the input. The argument to `T()` is a function factory that wants to build a new function that represents the + *term RIGHT* part of the input.

The arguments to the function factory are the `$term_value` and the `$RIGHT` function (and also the "+" token):

```
sub {
  my ($plus_token, $term_value, $RIGHT) = @_;
```

The job of this function factory is to build another function that given the value of *LEFT*, will return the value of the entire expression. The function we want to build is:

```
    return sub { $RIGHT->($_[0] + $term_value) }
}
```

The code for the minus rule is almost the same:

```
T(concatenate(lookfor(['OP', '-']),
                      $Term,
                      $Expression_Tail),
  sub {
    my ($plus_token, $term_value, RIGHT) = @_;
    return sub { $RIGHT->($_[0] - $term_value) }
  }),
```

The code for the "nothing" rule is simple. The "rest" of the expression contains nothing at all. We want a function, which, given the value of the left part of the expression, returns the value of the entire expression. But if the "rest" of the expression is empty, then the value of the left part of the expression *is* the value of the entire expression. So the function we want is just the function that returns its argument unchanged:

```
T(\&nothing, sub { sub { $_[0] } })
```

The code for the entire expression parser is:

```
$expression = T(concatenate($Term, $Expression_Tail),
                sub { $_[1]->($_[0]) }
               );
$expression_tail = alternate(T(concatenate(lookfor(['OP', '+']),
                                           $Term,
                                           $Expression_Tail),
                               sub { my ($op, $tv, $rest) = @_;
                                     sub { $rest->($_[0] + $tv) }
                                   }),
                             T(concatenate(lookfor(['OP', '-']),
                                           $Term,
                                           $Expression_Tail),
                               sub { my ($op, $tv, $rest) = @_;
```

```
                sub { $rest->($_[0] - $tv) }
              }),
       T(\&nothing, sub { sub { $_[0] } })
      );
```

With this structure, the parser works perfectly for all arithmetic expressions. To make it generate ASTs instead of numbers, we use:

```
$expression = T(concatenate($Term, $Expression_Tail),
                sub { $_[1]->($_[0]) }
               );
$expression_tail = alternate(T(concatenate(lookfor(['OP', '+']),
                                           $Term,
                                           $Expression_Tail),
                               sub { my ($op, $tv, $rest) = @_;
                                     sub { [ '+', $tv, $rest->($_[0]) ] }
                                   }),
                             T(concatenate(lookfor(['OP', '-']),
                                           $Term,
                                           $Expression_Tail),
                               sub { my ($op, $tv, $rest) = @_;
                                     sub { [ '-', $tv, $rest->($_[0]) ] }
                                   }),
                             T(\&nothing, sub { sub { $_[0] } })
                            );
```

8.4.3 A Variation on `star()`

We can use the `T()` function to improve our definition for `star()`. Formerly, the parser `star(lookfor('INT'))` would have returned a value like `[1, [2, [3, [4, undef]]]]`, because at each stage it's using `concatenate()` to join the values returned by `$p`, its argument, and `$p_star`, the recursive call to itself. We can use `T()` to adjust the value that comes out of `concatenate()` so that the result is `[1, 2, 3, 4]` instead:

```
sub star {
  my $p = shift;
  my $p_star;
  $p_star = alternate(T(concatenate($p, parser { $p_star->(@_) }),
                        sub { my ($first, $rest) = @_;
```

```
                    defined $rest ? [$first, @$rest] : [$first]
                  }),
            \&nothing);
}
```

The function we give to `T()` is responsible for appending the single value returned by `$p` to the (possibly empty) list of values returned by `$p_star` to generate a new list.

Given an input that's unacceptable to `$p`, `star($p)` succeeds, consuming none of the tokens, and returns an undefined value, behaving essentially like `nothing()`. We might prefer for it to return an empty array. In that case, we should use this version:

```
sub null_list {
  my $input = shift;
  return ([], $input);
}

sub star {
  my $p = shift;
  my $p_star;
  $p_star = alternate(T(concatenate($p, parser { $p_star->(@_) }),
                        sub { my ($first, $rest) = @_;
                              [$first, @$rest];
                            }),
                      \&null_list);
}
```

`null_list()` is like `nothing()`: It never consumes any input, and it always succeeds. But unlike `nothing()`, it returns an empty array instead of an undefined value. With this change, we no longer need the special-casery in the transformation function.

Having done this, we can simplify our elimination of left recursion. Instead of converting this:

```
A → A b | c
```

to this:

```
A → c A_tail
A_tail → b A_tail | (nothing)
```

we can convert it to this:

```
A → c star(b)
```

Similarly, this:

```
symbol → symbol A | symbol B
symbol → X | Y
```

becomes this:

```
symbol → (X | Y) star(A | B)
```

With this transformation, our original grammar for expressions:

```
expression → expression '+' term | expression '-' term | term
term → term '*' factor | term '/' factor | factor
factor → 'INT' | '(' expression ')'
```

is transformed to:

```
expression → term star('+' term | '-' term)
term → factor star('*' factor | '/' factor)
factor → 'INT' | '(' expression ')'
```

which is much easier to understand than the version that we saw before, with the artificial *expression_tail* and *term_tail* symbols.

The basic code for the *expression* parser now looks like this:

```
$expression = concatenate($Term,
                          star(alternate(concatenate(lookfor(['OP', '+']),
                                                     $Term),
                                         concatenate(lookfor(['OP', '-']),
                                                     $Term))));
```

To get this to compute the right values, we'll have the + *term* and - *term* parts return functions that, given the value of the left part of an expression, compute and return the value of the entire expression:

```
concatenate(lookfor(['OP', '+']), $Term)
```

becomes:

```
T(concatenate(lookfor(['OP', '+']), $Term),
  sub { my $term_value = $_[1];
        return sub { $_[0] + $term_value }
      });
```

The code for the - *term* part looks the same, with subtraction substituted for addition.

The parser generated by the `star()` call will return a list of these addition or subtraction functions. The outermost `concatenate()` call will produce a parser that expects the value of the leftmost *term* and the list of expression-computing functions returned by the `star()` parser. If the *expression* is `4+3-5`, the arguments to the parser will be the number 4 and the function list (`sub { $_[0] + 3 }`, `sub { $_[0] - 5 }`):

```
4           + 3                   - 5

|            |                     |

4    sub { $_[0] + 3 }      sub { $_[0] - 5 }
```

The outermost parser must compute the complete value of the expression. It does this by passing the leftmost *term* value to the first addition or subtraction function, passing the result to the second addition or subtraction function, and so on, like this:

```
sub { my ($total, $funcs) = @_;
      for my $f (@$funcs) {
        $total = $f->($total);
      }
      $total;
    }
```

The complete parser for expressions is :

```
$expression =
   T(concatenate($Term, star(alternate(T(concatenate(lookfor(['OP', '+']),
                                                     $Term),
                                         sub {
                                           my $term_value = $_[1];
```

```
                                        sub { $_[0] + $term_value };
                                    }),
                                T(concatenate(lookfor(['OP', '-']),
                                              $Term),
                                    sub {
                                      my $term_value = $_[1];
                                      sub { $_[0] - $term_value };
                                    }),
                                )))
    sub { my ($total, $funcs) = @_;
          for my $f (@$funcs) {
            $total = $f->($total);
          }
          $total;
        }
);
```

The corresponding change to the *term* parser is identical, with multiplication and division substituted for addition and subtraction.

8.4.4 Generic-Operator Parsers

We've already used this operator parser pattern twice, once in *expression* and once in *term*. Since operators are common, we might expect to use the same pattern in the future. We should try to abstract it into a generic function.

In general, we have a grammar where *symbol* can be expanded as *subpart* `OP` *subpart* `OP` ... *subpart*, where `OP` is left-associative. The parser we just saw is an example, with *symbol*, *subpart*, and `OP` being replaced by *expression*, *term*, and `+`, respectively. We'll write a function, `operator()`, which, given a parser for the subpart, a parser that recognizes the operator, and a callback function that implements the operator (`sub { $_[0] + $_[1] }` in the case of addition), generates a parser for sequences of *subpart* combined with `OP`. In general we'll want multiple operators of the same precedence, but to begin with let's assume there's only one operator at a time. Here's an example of what we want to produce: the parser for *expression*, with only addition, and not subtraction:

```
$expression =
   T(concatenate($Term, star(T(concatenate(lookfor(['OP', '+']),
                                                   $Term),
     sub {
```

```
    my $term_value = $_[1];
    sub { $_[0] + $term_value };
  }),
)),
    sub { my ($total, $funcs) = @_;
          for my $f (@$funcs) {
            $total = $f->($total);
          }
          $total;
        }
  );
```

The outline of our `operator()` function is:

```
sub operator {
  my ($subpart, $op, $opfunc) = @_;
  # Build and return parser like the preceding one
}
```

To construct the `operator()` function, we start by copying the *expression* parser into the body of `sub operator`. Then we systematically remove all the *expression*-specific code and replace each removed bit with the corresponding argument:

CODE LIBRARY
operator-singleop

```
sub operator {
  my ($subpart, $op, $opfunc) = @_;

  # Build and return parser like the earlier one
  T(concatenate($subpart, star(T(concatenate($op,
                                             $subpart),
                                 sub {
                                   my $subpart_value = $_[1];
                                   sub { $opfunc->($_[0], $subpart_value) };
                                 }),
                               )),
     sub { my ($total, $funcs) = @_;
           for my $f (@$funcs) {
             $total = $f->($total);
           }
           $total;
         }
    );

}
```

This does work; `operator($Term, lookfor(['op', '+']), sub { $_[0] + $_[1] })` does generate a parser for sums of terms. We now need to extend `operator()` to handle multiple operators of the same precedence. The argument to `star()` will be an alternation of several sections, rather than a single section. The argument to `operator()` itself may contain several `$opses` and `$opfuncs`. We'll call it like this:

```
operator($Term, [lookfor(['OP', '+']), sub { $_[0] + $_[1] }],
                [lookfor(['OP', '-']), sub { $_[0] - $_[1] }]);

sub operator {
  my ($subpart, @ops) = @_;
  my (@alternatives);
```

First we generate the alternatives that we'll give to `star()`:

```
  for my $operator (@ops) {
    my ($op, $opfunc) = @$operator;
    push @alternatives,
      T(concatenate($op,
                    $subpart),
                  sub {
                    my $subpart_value = $_[1];
                    sub { $opfunc->($_[0], $subpart_value) }
                  });
  }
```

Then we build the parser from the alternatives:

```
my $result =
  T(concatenate($subpart,
                star(alternate(@alternatives))),
    sub { my ($total, $funcs) = @_;
          for my $f (@$funcs) {
            $total = $f->($total);
          }
          $total;
        });
}
```

This is a lot of hairy code, but the payoff is excellent. We can stick the hairy `operator()` function into our parser-generating library and forget about it.

The code to generate the parsers for *term* and *expression* becomes short and transparent:

```
$expression =
  operator($Term,   [lookfor(['OP', '+']), sub { $_[0] + $_[1] }],
                    [lookfor(['OP', '-']), sub { $_[0] - $_[1] }]);

$term =
  operator($Factor, [lookfor(['OP', '*']), sub { $_[0] * $_[1] }],
                    [lookfor(['OP', '/']), sub { $_[0] / $_[1] }]);
```

Now that we have `operator()`, it's easy to imagine the next step: a parser generator function whose argument is an entire operator precedence table, and that, given a table like this:

```
[
  [['OP', '*'], ['OP', '/']],  # highest precedence
  [['OP', '+'], ['OP', '-']],  # lower precedence
]
```

does the work of the two calls to `operator()`, or more if we ask it to.

8.4.5 Debugging

Debugging programs containing complex nested functions can be difficult. The Perl interactive debugger isn't very helpful. If `$z` contains a reference to an anonymous function, the debugger won't give us much useful information about it:

```
  DB<119> x $z
0  CODE(0x849aca0)
   -> &main::__ANON__[arith15.pl:169] in arith15.pl:159-169
```

Just the file and the line numbers in the file. It could easily display the code, but it doesn't; with some extensions it could display the values of the subroutine's lexical variables, but again it doesn't. Internally, the debugger is a tremendous mess, and these improvements probably aren't forthcoming.[1] So we have to fall back on other techniques.

1 Someone looking for a fun project to garner fame and renown in the Perl world would do well to consider replacing the debugger.

The first problem with the debugger's display of anonymous functions is that it's hard to tell them apart. `CODE(0x849aca0)` doesn't communicate anything intelligible. The easy way to fix this is to have a hash that maps anonymous functions to names or descriptions:

```
$N{$z} = "main parser";
```

Now if we're in the middle of the program and we see a mysterious anonymous function, we can ask for:

```
print $N{$mystery};
```

and get something like:

```
Third alternative of 'expression' symbol
```

A variation on this technique makes the functions into blessed objects with an overloaded stringification operator. The stringification operator simply returns the appropriate element of `%N`.

Another variation that may not be for everyone, but that I've sometimes used when the functions had little natural significance, is to tie the `%N` hash and have it *invent* a name when asked for the name of a function that hasn't already had a name assigned:

```
open NAMES, "<", $namefile or die ...;


sub STORE {
  my ($self, $key, $name) = @_;
  $self->{$key} = $name;
}


sub FETCH {
  my ($self, $key) = @_:
  if (exists $self->{$key}) { return $self->{$key} }
  chomp(my $name = <NAMES>);
  $self->{$key} = $name;
  warn "Function had no name; I'll call it '$name'.\n";
  return $name;
}
```

Then you fill up `$namefile` with twenty or thirty random but evocative nouns, such as:

Máximo Perez
The Train
Luis Melián Lafineur
Olimar
Brimstone
Clubs
The Whale
Gas
The Cauldron
Napoleon
Agustín de Vedia
Nine
The Negro Timoteo
The Flesh Blanket

Even meaningless names can be helpful. They don't tell you what the function does, but they give your memory a peg to hang an association on, and you're likely to recognize a function you've seen before when it comes up again. If you don't like the name that was automatically assigned, you can easily replace it by assigning a different value to `%N`.

From here it's a small step to having the functions receive their names at their time of manufacture; for example:

```
my $CON = 'A';
sub concatenate {
  my $id;
  if (ref $_[0]) { $id = "Unnamed concatenation $CON"; $CON++ }
  else {           $id = shift }

  my @p = @_;
  return \&nothing if @p == 0;
  return $p[0]  if @p == 1;

  my $parser = parser {
    my $input = shift;
    my $v;
    my @values;
    for (@p) {
      ($v, $input) = $_->($input) or return;
```

```
        push @values, $v;
      }
     return values;
    };
    $N{$parser} = $id;
    return $parser;
}
```

Now `concatenate()` gets an optional initial argument, which is a string that will be used as the name of the parser it generates; for example:

```
$factor      = alternate(lookfor('INT'),
                         T(concatenate("factor : '(' expr ')'",
                                       lookfor(['OP', '(']),
                                       $Expression,
                                       lookfor(['OP', ')'])),
                           sub { $_[1] })
                        );
```

We can of course do the same thing to `alternate()`:

```
$factor      = alternate("factor symbol parser",
                         lookfor('INT'),
                         T(concatenate("factor : '(' expr ')'",
                                       lookfor(['OP', '(']),
                                       $Expression,
                                       lookfor(['OP', ')'])),
                           sub { $_[1] })
                        );
```

Since `$id` is a lexical variable, it can be captured and used inside the parser itself:

CODE LIBRARY
Parser::Debug.pm

```
package Parser::Debug;
use base 'Exporter';
use Parser ':all';
@EXPORT_OK = @Parser::EXPORT_OK;
%EXPORT_TAGS = %Parser::EXPORT_TAGS;

my $CON = 'A';
sub concatenate {
  my $id;
```

```
if (ref $_[0]) { $id = "Unnamed concatenation $CON"; $CON++ }
else {           $id = shift }

my @p = @_
return \&n ll if @p == 0;
return $p[ ]  if @p == 1;

my $parser = parser {
  my $input = shift;
  debug "Looking for $id\n";
  my $v;
  my @values;
  my ($q, $np) = (0, scalar @p);
  for (@p) {
    $q++;
    unless (($v, $input) = $_->($input)) {
      debug "Failed concatenated component   $q/$np\n";
      return;
    }
    debug "Matched concatenated component   $q/$np\n";
    push @values, $v;
   }
   debug "Finished matching $id\n";
   return \@values;
  };
  $N{$parser} = $id;
  return $parser;
}
```

With a suitable definition of `debug()`, this will generate output like:

```
Looking for factor : '(' expr ')'
Matched concatenated component 1/3
Matched concatenated component 2/3
Matched concatenated component 3/3
Finished matching factor : '(' expr ')'
```

or:

```
Looking for factor : '(' expr ')'
Failed concatenated component 1/3
```

The easiest thing to do in `debug()` is just to print out the message. But we can do a little better:

```
sub debug ($) {
  return unless $DEBUG || $ENV{DEBUG};
  my $msg = shift;
  my $i = 0;
  $i++ while caller($i);
  $I = "| " x ($i-2);
  print $I, $msg;
}
```

The `while caller` loop computes the depth to which function calls have been nested. `$I` is an indentation string that is used to indent the debugging message accordingly. Now the output while parsing an expression like `"8 - 3"` begins like this:

```
| Looking for Input: expression EOI
| (Next token is INT 8)
| | | | Looking for expression : term star('+'term | '-' term)
| | | | (Next token is INT 8)
| | | | | | | Looking for term: factor star('*'factor | '/' factor)
| | | | | | | (Next token is INT 8)
| | | | | | | | | Looking for factor: INT | '(' expression ')'
| | | | | | | | | (Next token is INT 8)
| | | | | | | | | Trying alternative 1/2
| | | | | | | | | | Looking for token [INT]
| | | | | | | | | | Next token is [INT 8]
| | | | | | | | | | Token matched
| | | | | | | | | Matched alternative 1/2
| | | | | | | Matched concatenated component 1/2
...
```

Capturing `$id` inside the generated parser, as we've done, causes a problem. If we want to change the name of a parser after we've constructed it, we can't, because the name is captured in an inaccessible lexical variable inside the parser. We can change the name in the `%N` hash, but when the parser prints debug messages, it will still use the old name. One way to fix this is to have the debug messages refer to `%N`; another way is to provide each parser with a method for changing its name. The `%N` tactic is easier by far. Using a global variable like this should make you uncomfortable, and when you do it, the first question

you should ask is "What if there are several different parsers in one program? Will their overlapping uses of `%N` collide?" In this case, though, there's no problem. Each anonymous function in the program resides at a different address, which means that the `CODE(0x436c1d)` strings will all be unique. As long as the parsers don't tamper with hash elements they don't understand, all will be well.

Why would we want to change the name of a parser? We'll see in a minute. Before that, let's note that the names we've been using are formulaic, which means that the next step is to get the parser construction functions to generate the names automatically. We begin by naming the basic parsers:

```
$N{\&End_of_Input} = 'EOI';
$N{\&nothing} = '(nothing)';
$N{$Expression} = 'expression';
$N{$Term} = 'term';
$N{$Factor} = 'factor';
```

Then we fix up `concatenate()` and `alternate()` to generate names from the names of their arguments:

```
sub concatenate {
  my @p = @_;
  return \&nothing if @p == 0;
  return $p[0]  if @p == 1;

  my $id = "@N{@p}";

  my $p = parser {
    ...
  };
  $N{$p} = $id;
  return $p;
}

sub alternate {
  my @p = @_;
  return parser { return () } if @p == 0;
  return $p[0]                if @p == 1;
  my $id = join " | ", @N{@p};
```

```
  my $p = parser {
    ...
  };
  $N{$p} = "($id)";
  return $p;
}
```

Similarly, the description of the parser produced by `T()` is the same as the description of its argument:

```
sub T {
  my ($parser, $transform) = @_;
  my $p = parser {
    ...
  };
  $N{$p} = $N{$parser};
  return $p;
}
```

Now we change `lookfor()` to name its parsers after the token they're looking for:

```
sub lookfor {
  my $wanted = shift;
  ...
  $N{$parser} = "[@$wanted]";
  return $parser;
}
```

Finally, we change `star()`:

```
sub star {
  my $p = shift;
  my ($p_star, $conc);
  $p_star = alternate(T($conc = concatenate($p, parser { $p_star->(@_) }),
                        sub { my ($first, $rest) = @_;
                              [$first, @$rest];
                            }),
                      \&null_list);
  $N{$p_star} = "star($N{$p})";
  $N{$conc} = "$N{$p} $N{$p_star}";
  return $p_star;
}
```

This is why we needed to be able to change the names of parsers. The arguments to `concatenate()` are the argument `$p`, whose description, let's say, is `P`, and the eta-converted version of `$p_star`, which didn't have a description. The parser that would come out of `concatenate()` therefore would have a name something like `"P "`, and the parser that would come out of `alternate()` would be named something like `P | (nothing)`, neither of which is very helpful. After the parsers are generated, we change the name of the alternation, `$p_star` itself, to `star(P)`, and the name of the concatenation, temporarily stored in `$conc`, to `P star(P)`.

The output, including automatically generated names, now looks like this (the input was `"8 - 3"`):

```
| Looking for expression EOI
| (Next token is INT 8)
| | | | Looking for term star(([OP +] term | [OP -] term))
| | | | (Next token is INT 8)
| | | | | | | Looking for factor star(([OP *] factor | [OP /] factor))
| | | | | | | (Next token is INT 8)
| | | | | | | | | Looking for ([INT] | [OP (] expression [OP )])
| | | | | | | | | (Next token is INT 8)
| | | | | | | | | Trying alternative 1/2
| | | | | | | | | | Looking for token [INT]
| | | | | | | | | | Next token is [INT 8]
| | | | | | | | | | Token matched
| | | | | | | | | Matched alternative 1/2
| | | | | | | Matched concatenated component 1/2
| | | | | | | | Looking for star(([OP *] factor | [OP /] factor))
| | | | | | | | (Next token is OP -)
| | | | | | | | Trying alternative 1/2
```

... looking for multiplicative factors ...

```
| | | | | | | | Failed alternative 1/2
| | | | | | | | Trying alternative 2/2
| | | | | | | | | | Looking for nothing
| | | | | | | | | | (Next token is OP -)
| | | | | | | | Matched alternative 2/2
| | | | | | | Matched concatenated component 2/2
| | | | | | | Finished matching factor star(([OP *] factor | [OP /] factor))
| | | | Matched concatenated component 1/2
| | | | | Looking for star(([OP +] term | [OP -] term))
| | | | | (Next token is OP -)
| | | | | Trying alternative 1/2
  ...
```

When parsing goes wrong, examination of this debugging output is usually enough to figure out where the problem lies.

8.4.6 The Finished Calculator

Let's finish the calculator now. It starts with the lexer that we saw before:

CODE LIBRARY
calculator

```
use Parser ':all';
use Lexer ':all';

my $input = allinput(\*STDIN);
my $lexer = iterator_to_stream(
                make_lexer($input,
                        ['TERMINATOR', qr/;\n*|\n+/          ],
                        ['INT',        qr/\b\d+\b/           ],
                        ['PRINT',      qr/\bprint\b/         ],
                        ['IDENTIFIER', qr|[A-Za-z_]\w*|      ],
                        ['OP',         qr#\*\*|[-=+*/()]#    ],
                        ['WHITESPACE', qr/\s+/, sub { "" }   ],
                )
               );
```

The complete grammar for the calculator is:

```
program → star(statement) 'End_of_Input';

statement → 'PRINT' expression 'TERMINATOR'
          |  'IDENTIFIER' '=' expression 'TERMINATOR'

expression → term star('+' term | '-' term)

term → factor star('*' factor | '/' factor)

factor → base ('**' factor | (nothing))

base → 'INT' | 'IDENTIFIER' | '(' expression ')'
```

There are two new grammar rules at the top: *program*, which represents the entire input, and *statement*, which represents a single statement, either a variable assignment or a request to print a result.[2]

2 In most modern languages, including Perl and C, statements may have a simpler structure, typically not much different than an expression. For example, in Perl, `print $x` and `$x = $y` are both

The grammar for expressions is a little more complicated also. The *factor* symbol no longer represents an atomic expression. Instead, it contains an optional exponentiation operation. Filling the atomic role formerly played by *factor* is a new symbol, *base*, which as before can be a number or a parenthesized expression, and now can also be a variable name.

```
my %VAR;

my ($base, $expression, $factor, $program, $statement, $term);
$Base       = parser { $base->(@_) };
$Expression = parser { $expression->(@_) };
$Factor     = parser { $factor->(@_) };
$Program    = parser { $program->(@_) };
$Statement  = parser { $statement->(@_) };
$Term       = parser { $term->(@_) };

$program = concatenate(star($Statement), \&End_of_Input);

$statement = alternate(T(concatenate(lookfor('PRINT'),
                                     $Expression,
                                     lookfor('TERMINATOR')),
                         sub { print ">> $_[1]\n" }),
                       T(concatenate(lookfor('IDENTIFIER'),
                                     lookfor(['OP', '=']),
                                     $Expression,
                                     lookfor('TERMINATOR')
                                    ),
                         sub { $VAR{$_[0]} = $_[2] }),
                      );
```

When the parser recognizes a complete `print` statement, it prints out the value of the expression. When it recognizes a complete assignment statement, it stores the value of the expression in a hash, `%VAR`, with the variable's name as a key. (We can prepopulate `%VAR` with the values of useful constants, such as π.)

The parsers for *expression* and *term* are exactly as before:

```
$expression =
  operator($Term,  [lookfor(['OP', '+']), sub { $_[0] + $_[1] }],
                   [lookfor(['OP', '-']), sub { $_[0] - $_[1] }]);
```

---

expressions, the former returning true or false to indicate success or failure of printing, and the latter returning the value of `$y`. If we do this, we get the opportunity to do things like `$result = print $x` and `$x = $y = $z`, which the calculator won't allow. It would have been both simpler and more useful to write the calculator this way, and I introduced the special *statement* forms solely for variety.

```
$term =
  operator($Factor, [lookfor(['OP', '*']), sub { $_[0] * $_[1] }],
                    [lookfor(['OP', '/']), sub { $_[0] / $_[1] }]);
```

Factors are a little different than in earlier examples, because they may now contain ** operators:

```
$factor = T(concatenate($Base,
                        alternate(T(concatenate(lookfor(['OP', '**']),
                                                $Factor),
                                    sub { $_[1] }),
                                  T(\&nothing, sub { 1 }))),
            sub { $_[0] ** $_[1] });
```

For an expression like `3 ** 4`, we assign a value of 4 to the `** 4` part; the final value computation assigns a value of 81 to the entire `3 ** 4` expression. A missing exponent has a value of 1, so that `3` gets the same value as `3 ** 1`. We haven't used `operator()` here because `operator()` generates parsers for left-associative operators, and ** is right associative. `2**2**3` means `2**(2**3)` = 256, not `(2**2)**3` = 64.

```
$base      = alternate(lookfor('INT'),
                       lookfor('IDENTIFIER',
                               sub { $VAR{$_[0][1]} || 0 }),
                       T(concatenate(lookfor(['OP', '(']),
                                     $Expression,
                                     lookfor(['OP', ')'])),
                         sub { $_[1] })
                      );
```

The parser for *base* is just like the old parser for *term*, except with an extra clause for handling identifiers. To recover the value of an identifier, we look up the name of the identifier in the `%VAR` hash. Undefined variables behave like the number 0. Alternatively, we could have the calculator issue a warning message for undefined variables.

The calculator is complete; all we need is to invoke the parser:

```
$program->($lexer);
```

Given the following input:

```
a = 12345679 * 6
b=a*9; c=0
print b
```

it produces the correct output:

```
>> 666666666
```

8.4.7 Error Diagnosis and Recovery

Although the calculator works fine on correct inputs, it fails unpleasantly on erroneous input. If we delete the semicolon from the previous example, yielding:

```
a = 12345679 * 6
b=a*9 c=0
print b
```

then the parser simply fails after the assignment to `a`, returning an undefined value. No error message is generated.

ERROR-RECOVERY PARSERS

One easy way to put some error handling into the parser is to build a special-purpose sub-parser whose job is to recover from errors. If a statement goes awry, control will pass to the error-recovery parser. The error-recovery parser will try to resynchronize the parser with the input by discarding tokens until it gets to the end of the bad statement, and then restarting the parser from the new position:

```
$statement = alternate(T(concatenate(lookfor('PRINT'),
                                     $Expression,
                                     lookfor('TERMINATOR')),
                         sub { print ">> $_[1]\n" }),
                       T(concatenate(lookfor('IDENTIFIER'),
                                     lookfor(['OP', '=']),
                                     $Expression,
                                     lookfor('TERMINATOR')
                                    ),
                         sub { $VAR{$_[0]} = $_[2] }),
                        error(lookfor('TERMINATOR'), $Statement),
                      );
```

The `error()` call here generates a new parser, which is a third alternative form for *statement*. If, when trying to parse a statement, the upcoming input fails to

match either of the first two forms, the error-recovery parser returned by `error()` will be invoked.

`error()`'s first argument is another parser whose job is to identify a good place in the input to restart the parsing process. The error-recovery parser generated by `error()` will discard one token at a time until it reaches a point in the input that is acceptable to `error()`'s first argument. In this case, a good place to restart is immediately following a `TERMINATOR` token, because that is where a new statement is likely to begin. When the error-recovery parser is ready to continue, it invokes another parser to continue the job on the remaining input stream; this other parser is `error()`'s second argument. In this case, once the error-recovery parser reaches a newline, it invokes `$Statement` to start looking for another statement.

The code for `error()` is fairly straightforward. First we'll see it without debugging clutter:

```
sub error {
  my ($checker, $continuation) = @_;
  my $p;
  $p = parser {
    my $input = shift;

    while (defined($input)) {
      if (my (undef, $result) = $checker->($input)) {
        $input = $result;
        last;
      } else {
        drop($input);
      }
    }

    return unless defined $input;

    return $continuation->($input);
  };
  $N{$p} = "errhandler($N{$continuation} -> $N{$checker})";
  return $p;
}
```

The essential line is `drop($input)`, which discards a token from the input. This is done until either the `$checker` parser (`lookfor('TERMINATOR')` in the example) succeeds, or the input is exhausted. Afterward, the error-handler parser continues by invoking the continuation (`$Statement` in the example) on the remaining input.

Here's the version with debugging messages:

```
sub error {
  my ($checker, $continuation) = @_;
  my $p;
  $p = parser {
    my $input = shift;
    debug "Error in $N{$continuation}\n";
    debug "Discarding up to $N{$checker}\n";
    my @discarded;
    while (defined($input)) {
      my $h = head($input);
      if (my (undef, $result) = $checker->($input)) {
        debug "Discarding $N{$checker}\n";
        push @discarded, $N{$checker};
        $input = $result;
        last;
      } else {
        debug "Discarding token [@$h]\n";
        push @discarded, $h->[1];
        drop($input);
      }
    }
    warn "Erroneous input: ignoring '@discarded'\n" if @discarded;
    return unless defined $input;
    debug "Continuing with $N{$continuation} after error recovery\n";
    $continuation->($input);
  };
  $N{$p} = "errhandler($N{$continuation} -> $N{$checker})";
  return $p;
}
```

On our erroneous input, the calculator program performs the assignment to a as instructed on the first line, and then says:

```
Erroneous input: ignoring 'b = a * 9 c = 0
'
>> 0
```

The `>> 0` is the result of printing out the value of `b` in the final line; `b` was unset and defaulted to 0 because of the syntax error in the previous line.

EXCEPTIONS

Another convenient way to deal with errors is to change the structure of the parsers. Instead of returning `undef` to indicate a failed parse, they will throw an exception. The exception will include information about what the parser would have accepted and what it saw instead. Error-handler parsers will catch the exceptions, issue error messages, resynchronize, and restart.

First, a brief review of the semantics of exceptions in Perl. Perl's exception-handling mechanism is unfortunately named `eval`:

```
my $result = eval { ... };
```

Code in the `eval` is run in exactly the same way as any other code, returning the same result into `$result`, except that if it throws an exception, the exception will be caught by the `eval` instead of terminating the program. If the code inside the `eval` throws an exception, `$result` becomes undefined.

Exceptions are thrown with the `die` function. They may be arbitrary data objects. After an exception is caught, the value thrown is placed into the special variable `$@`.

To rewrite the `End_of_Input()` parser in this style, we say:

CODE LIBRARY
Parser::Except.pm

```
sub End_of_Input {
  my $input = shift;
  return (undef, undef) unless defined($input);
  die ["End of input", $input];
}
```

If there is no more input, the parser returns a value as before. If there is more input, the parser fails by calling `die`. The `die` value includes a string describing what was being sought (`"End of Input"`) and what was found instead (`$input`).

Here's `lookfor()`:

```
sub lookfor {
  my $wanted = shift;
  my $value = shift || sub { $_[0][1] };
  my $u = shift;
  $wanted = [$wanted] unless ref $wanted;

  my $parser = parser {
    my $input = shift;
    unless (defined $input) {
```

```
      die ['TOKEN', $input, $wanted];
    }

    my $next = head($input);
    for my $i (0 .. $#wanted) {
      next unless defined $wanted->[$i];
      unless ($wanted->[$i] eq $next->[$i]) {
        die ['TOKEN', $input, $wanted];
      }
    }
    my $wanted_value = $value->($next, $u);
    return ($wanted_value, tail($input));
  };

  $N{$parser} = "[@$wanted]";
  return $parser;
}
```

If the `lookfor()`-generated parser sees the token it wants, it returns the same value as the old version. If it sees end-of-input or the wrong token, it throws an exception. As before, the exception value includes a tag indicating what kind of thing was being sought (a `TOKEN`), and what was found instead (`$input`). Here it also includes auxiliary information indicating what token was sought.

`nothing()` requires no changes because it never fails. `concatenate()` requires no changes, although we have an opportunity to make it simpler. It no longer needs to check to see if its sub-parsers have succeeded and terminate prematurely if one hasn't. It can assume that they all will succeed, because if one doesn't, it will throw an exception that will terminate `concatenate()` prematurely anyway.

`alternate()` is the interesting one. When a sub-parser succeeds, it stops and returns the value, as before. When a sub-parser fails, the `alternate()` parser needs to catch the exception so that it can try the next sub-parser. It installs the exception in an array `@failures`; if all the sub-parsers fail, `@failures` will contain the list of reasons why, and `alternate()` can throw an exception that includes this information:

```
sub alternate {
  my @p = @_;
  return parser { return () } if @p == 0;
  return $p[0]                if @p == 1;

  my $p;
  $p = parser {
```

```
        my $input = shift;
        my ($v, $newinput);
        my @failures;

        for (@p) {
          eval { ($v, $newinput) = $_->($input) };
          if ($@) {
            die unless ref $@;
            push @failures, $@;
          } else {
            return ($v, $newinput);
          }
        }
        die ['ALT', $input, \@failures];
      };
      $N{$p} = "(" . join(" | ", map $N{$_}, @p) . ")";
      return $p;
    }
```

The `die unless ref $@` line is there to propagate any exception that has to do with a programming error, such as division by zero. If we didn't propagate these, then they would get absorbed into the `@failures` array, and might be thrown away.

Finally, we need a function to actually catch exceptions and issue a report. Here's a simple one:

```
sub error {
  my ($try) = @_;
  my $p;
  $p = parser {
    my $input = shift;
    my @result = eval { $try->($input) };
    if ($@) {
      display_failures($@) if ref $@;
      die;
    }
    return @result;
  };
}
```

Its argument `$try` is a parser; `error()` returns a new parser that tries `$try`. If `$try` succeeds, `error()` returns its value. If `$try` fails, `error()` issues an error report

with `display_failures()` and then calls `die` to propagate the same exception up to *its* caller.

Here's a rather elaborate implementation of `display_failures()`:

```
sub display_failures {
  my ($fail, $depth) = @_;
  $depth ||= 0;
  my $I = " " x $depth;
  my ($type, $position, $data) = @$fail;
  my $pos_desc = "";

  while (length($pos_desc) < 40) {
    if ($position) {
      my $h = head($position);
      $pos_desc .= "[@$h] ";
    } else {
      $pos_desc .= "End of input ";
      last;
    }
    $position = tail($position);
  }
  chop $pos_desc;
  $pos_desc .= "..." if defined $position;

  if ($type eq 'TOKEN') {
    print $I, "Wanted [@$data] instead of '$pos_desc'\n";
  } elsif ($type eq 'End of input') {
    print $I, "Wanted EOI instead of '$pos_desc'\n";
  } elsif ($type eq 'ALT') {
    print $I, ($depth ? "Or any" : "Any"), " of the following:\n";
    for (@$data) {
      display_failures($_, $depth+1);
    }
  }
}
```

`display_failures()` is expecting to get at least one argument, which is the exception object, and has the form `[$type, $position, $data]`, where `$type` is just an identifying string, `$position` is the position in the input stream at which the error occurred, and the `$data` is optional and has a form that depends on the `$type`.

The middle section analyzes the input tokens that appear starting at `$position` and builds up `$pos_desc`, a string describing them. The end section examines `$type` and prints an appropriate message. The case for `ALT` is the interesting one; in this case `$data` is an array of all the failure exceptions from the sub-parsers of the alternation. `display_failures` prints an appropriate header and calls itself recursively to display the sub-failures.

The top-level call of the main parser on the entire input needs to be protected by an `eval` block, so that the program has a chance to recover from an uncaught exception:

```
my ($val, $rest) = eval { $program->($lexer) };
if ($@) {
  display_failures($@);
}
```

or we could just use:

```
my ($val, $rest) = error($program)->($lexer);
```

Let's consider the example `"a=3; b+7; c=5;"` and see what comes out:

```
Wanted EOI instead of '[IDENTIFIER b] [OP +] [INT 7] [TERMINATOR ;]...'
```

The parser is reporting that it was unhappy with the "`b+7;`" part of the input; it would have preferred to see the input end right after the "`a=3;`" part.

We can get better reporting by using the `error()` function to insert error reporting at appropriate places inside the parser. For example, if we change:

```
$statement = alternate(...);
```

to:

```
$statement = error(alternate(...));
```

then all erroneous statements will be diagnosed. Now the output includes:

```
Any of the following:
  Wanted [PRINT] instead of '[IDENTIFIER b] [OP +] [INT 7] [TERMINATOR ;]...'
  Wanted [OP =] instead of '[OP +] [INT 7] [TERMINATOR ;] [IDENTIFIER c]...'
```

It wanted to see `"print"` after `"a=3;"`, or else `"="` instead of `"+"` after the `"b"`.

This parser aborts at the first sign of trouble. We could add error recovery to the reporting behavior of `error()` and get the parser to continue and perhaps diagnose more errors.

8.4.8 Big Numbers

As a final improvement, we'll change the calculator to support arbitrarily large numbers. This is trivial; we add:

```
use Math::BigFloat;
```

to the top of the program, and change the part of the parser that assigns a value to an `INT` token:

```
lookfor('INT')
```

becomes:

```
lookfor('INT', sub { Math::BigFloat->new($_[0][1]) });
```

which passes the token's string representation to `Math::BigFloat`, which constructs a big number object for it. `Math::BigFloat` overloads the normal arithmetic operators to work on arbitrary-precision numbers, so we don't need to change anything else. It would be almost as easy to get the calculator to support complex numbers; we would use `Math::Complex`, and add a line to the lexer to properly interpret constants that matched `/\d+i/`.

8.5 PARSING REGEXES

As an example that's probably more practical than the calculator, we'll implement a parser for a subset of Perl's regular expressions. In Chapter 6 we saw a program that generated a (possibly infinite) stream of all the strings matched by a certain regex. But it wasn't convenient to use this program. To get a list of the strings matched by `/(a|b)*c+/`, for example, we had to write the following code:

```
my $z = concat(star(union(literal("a"), literal("b"))),
               plus(literal("c")),
              );
```

What we'd like is to be able to put in a regex in the usual notation and get out the same stream. Our parsing technology will do this. We'll build a parser that can analyze the structure of a regex. As it determines the structure of the regex, it will call the appropriate stream functions to manufacture the stream of matching strings.

First, the lexer. Regexes contain the following operators: `+ * ? ( ) |`. Other than this, they contain atomic expressions, or "atoms," such as `w`, `\r`, and `\x0d`. They also contain other items such as character classes, non-capturing parentheses, lookahead items, and embedded Perl code; we'll ignore these because they're not particularly instructive.

Of these lexical types, the atoms are all syntactically equivalent; any valid regex that contains a `w` is still valid if the `w` is replaced by `\r` or by `\x0d`. Similarly, the quantifiers `+`, `*`, and `?` are all syntactically equivalent. This suggests the following lexer:

CODE LIBRARY
regex-parser

```
use Lexer ':all';
use Stream 'node';

my ($regex, $alternative, $atom, $qatom);

sub regex_to_stream {
  my @input = @_;
  my $input = sub { shift @input };

  my $lexer = iterator_to_stream(
          make_lexer($input,
                     ['ATOM',       qr/\\x[0-9a-fA-F]{0,2} # hex escape
                                       |\\\d+              # octal escape
                                       |\\.                # other \
                                       /x,               ],
                     ['PAREN',      qr/[()]/,           ],
                     ['QUANT',      qr/[*+?]/           ],
                     ['BAR',        qr/[|]/,            ],
                     ['ATOM',       qr/./,              ], # other char
                    )
          );

    my ($result) =  $regex->($lexer);
    return $result;
  }
```

Regexes are similar in structure to arithmetic expressions, only with fewer different operators. The lowest-precedence operator is the vertical bar, `|`. A regex is a series of alternatives separated by vertical bars. Each alternative is a (possibly empty) sequence of (possibly quantified) atoms. For example, in the regex `/(a|b)*c+/`, there is a single alternative, consisting of two quantified atoms: `(a|b)*` and `c+`. The parentheses around `a|b` group the contents into a single atomic expression. `a|b`, of course, contains two alternatives, each with one unquantified atom.

The grammar is:

```
regex → alternative 'BAR' regex  |  alternative
alternative → qatom alternative | (nothing)
qatom → atom ('QUANT' | (nothing))
atom → 'ATOM' | '(' regex ')'
```

As before, we define eta-conversions of the parsers so that the parsers can be mututally recursive:

```
use Parser ':all';

my $Regex       = parser { $regex      ->(@_) };
my $Alternative = parser { $alternative->(@_) };
my $Atom        = parser { $atom       ->(@_) };
my $QAtom       = parser { $qatom      ->(@_) };
```

Building the basic parser from the grammar is straightforward:

```
# regex -> alternative 'BAR' regex  |  alternative
$regex = alternate(concatenate($Alternative,
                               lookfor('BAR'),
                               $Regex),
                   $Alternative);
```

This is fine if we want to generate ASTs for regexes. But what we really want is to generate streams of strings. If we assume that the values returned for `$Alternative` and `$Regex` are streams, it's easy to generate the result for the entire regex. We use the `union()` function from Section 6.5.1:

```
use Regex;

# regex -> alternative 'BAR' regex  |  alternative
$regex = alternate(T(concatenate($Alternative,
```

```
                                lookfor('BAR'),
                                $Regex),
                    sub { Regex::union($_[0], $_[2])}),
                $Alternative);
```

If the regex consists of a single alternative, then the list of strings it matches is the same as the list of strings matched by the single alternative, so nothing extra needs to be done.

Similarly, the `concat()` function from Section 6.5.1 takes the streams that list the strings matched by two regexes and returns the stream of strings that are matched by the concatenation of two regexes, so it's just what we need to generate the value of a single alternative:

```
# alternative -> qatom alternative | (nothing)
$alternative = alternate(T(concatenate($QAtom, $Alternative),
                           \&Regex::concat),
                         T(\&nothing, sub { Regex::literal("") }));
```

If the alternative is empty, it matches only the empty string. The call `Regex::literal("")` returns a stream that contains the empty string and nothing else.

```
my %quant;
# qatom -> atom ('QUANT' | (nothing))
$qatom  = T(concatenate($Atom,
                        alternate(lookfor('QUANT'),
                                  \&nothing),
                       ),
            sub { my ($at, $q) = @_;
                  defined $q ? $quant{$q}->($at) : $at });

%quant = ('*' => \&Regex::star,
          '+' => \&Regex::plus,
          '?' => \&Regex::query,
         );
```

For quantified atoms, we get a stream that represents the list of strings matched by the atom, to which we apply the appropriate quantifier. There might not be a quantifier, in which case the value of the second element of the concatenation is undefined, and we return the value of the atom unchanged. Note that `%quant` is nothing more than a dispatch table.

We saw `star()` and `plus()` back in Chapter 6, but not `query()`, which is trivial:

```
sub query {
  my $s = shift;
  union(literal(""), $s);
}
```

It matches anything its argument matches, and also the empty string.

```
# atom -> 'ATOM' | '(' regex ')'
$atom = alternate(lookfor("ATOM", sub { Regex::literal($_[0][1]) }),
                  T(concatenate(lookfor(["PAREN", "("]),
                                $Regex,
                                lookfor(["PAREN", ")"]),
                               ),
                    sub { $_[1] }),
                 );
```

Finally, we're down to atoms. If the atom is indeed a single atomic token, then the list of strings contains the value of the token itself; for example, the atomic regex `w` matches the string `'w'` and nothing else. If the atom is actually a complete regex in parentheses, we call the regex parser recursively and return the value it returns, throwing away the parentheses, just as we did in the arithmetic expression parser.

Calling `Regex::literal()` is a little too simple. When the atom is `\x0d`, it does *not* match the string `'\x0d'`; it matches the single-character string that contains only a carriage return. We could fix this by passing the token to a string-interpreting function before passing the result to `Regex::literal()`.

Some atoms are more difficult to handle. This parser treats `\b`, a word-boundary assertion, as an atom. It is atomic, but it certainly doesn't match the string `'\b'`. To handle this properly, we'd have to introduce a new kind of value in our string streams; the new values would denote strings with boundary requirements at the front, the back, or both. When the `concat()` operator tried to put together two strings with boundary requirements at the ends at which they were being joined, it would check the requirements for compatibility. If the requirements were incompatible, `concat()` would skip that pair of strings and move on. The same scheme could handle lookahead and lookbehind assertions, although the lexer would have to be extended to understand the notations.

Still, for all its limitations, the string generator performs adequately for a first cut. If we give it the input `(a|b)+(c|d*)`, it cheerfully returns an infinite stream that begins:

```
a       bbb     abba
b       add     abbb
aa      aac     baaa
ab      abc     baab
ba      bac     baba
bb      bbc     babb
ac      bdd     bbaa
ad      aad     bbab
bc      abd     bbba
bd      bad     bbbb
aaa     bbd     addd
aab     aaaa    aaac
aba     aaab    aabc
abb     aaba    abac
baa     aabb    abbc
bab     abaa    baac
bba     abab    ...
```

8.6 OUTLINES

The regex and calculator examples have many similarities. Here's an example that is quite different. We'll write a program to read in topic outlines and infer a tree structure from the indentation. A typical input looks like this:

```
* Parsing
  * Lexers
    * Emulating the <> operator
    * Lexers more generally
    * Chained Lexers
    * Peeking
  * Parsing in General
    * Grammars
    * Parsing Grammars
  * Recursive-Descent Parsers
    * Very Simple Parsers
    * Parser Operators
    * Compound Operators
```

```
* Arithmetic Expressions
  * A Calculator
  * Left Recursion
  * A Variation on 'star'
  * Generic-Operator Parsers
  * Debugging
  * The Finished Calculator
  * Error Diagnosis and Recovery
    * Error-Recovery Parsers
    * Exceptions
  * Big Numbers
* Parsing Regexes
* Outlines
* Database-Query Parsing
  * The Lexer
  * The Parser
* Backtracking Parsers
  * Continuations
  * Parse Streams
* Overloading
```

To keep the problem manageable, we'll make a few simplifying assumptions. Each item occupies exactly one line, and begins with a "bullet" character. The first line is the root of the tree, and will be flush with the left margin; each sub-item will be indented two spaces farther right than its parent.

All the lexers we've seen so far have discarded whitespace. The lexer for this problem mustn't do that because the whitespace is significant. But the input has a very simple lexical structure: each line is an "item," and is separated from the next item by newlines. So the lexer is simply:

CODE LIBRARY
outline-parser

```
use Lexer ':all';
use Stream 'node';

my ($tree, $subtree);
sub outline_to_array {
  my @input = @_;
  my $input = sub { shift @input };

  my $lexer = iterator_to_stream(
          make_lexer($input,
                     ['ITEM',        qr/^.*$/m ],
```

```
                        ['NEWLINE',     qr/\n+/    , sub { "" } ],
                       )
            );

    my ($result) = $tree->($lexer);
    return $result;
}
```

The grammar is almost as simple. A tree has a root item, followed by zero or more subtrees. The subtrees must be indented farther to the right than the root item. The grammar will look something like this:

```
input → tree 'End_of_input'
tree → 'ITEM' star(subtree)
subtree → tree
```

This isn't exactly right, because it doesn't take into account the indentations. When the parser sees the root item of a new tree, it needs to record that item's indentation; when it tries to parse a subtree, it should succeed if the next item is indented farther right than the previous root, and fail if it isn't. Consider this simple example:

```
* A
  * B
    * C
  * D
```

Item C here is a sub-item of B, because it's indented farther to the right. At this point, the parser will be looking for sub-items of C. The next item is D. D is not to the right of C, so it is not a sub-item of C. Moreover, it is not to the right of B, so it is not a sub-item of B. But D is to the right of A, so D is a sub-item of A.

The grammar needs to be extended to take the indentations into account:

```
input → tree 'End_of_input'
tree → 'ITEM' <<record indentation of root node>> star(subtree)
subtree → <<check indentation of next item>> tree
```

Here, `<<record indentation of root node>>` is a special version of the null parser. It consumes no input, always succeeds, and returns a meaningless value. But it also examines the item that was just parsed and makes a note of how far it was indented. Similarly, `<<check indentation of next item>>` also consumes

no input and returns a meaningless value. But it also examines the head of the input token stream to make sure that the upcoming item is indented to the right of the current root node. If so, it succeeds, allowing the *subtree* parser to proceed to the real business of parsing the subtree, which it does by calling the *tree* parser recursively. If not, it fails, causing the *subtree* parser to fail also.

Here's the main parser:

```
use Parser ':all';
use Stream 'head';
my $Tree    = parser { $tree->(@_) };
my $Subtree = parser { $subtree->(@_) };

my $LEVEL = 0;
$tree = concatenate(lookfor('ITEM', sub { trim($_[0][1]) }),
                    action(sub { $LEVEL++ }),
                    star($Subtree),
                    action(sub { $LEVEL-- }));

my $BULLET = '[#*ox.+-]\s+';
sub trim {
  my $s = shift;
  $s =~ s/^ *//;
  $s =~ s/^$BULLET//o;
  return $s;
}
```

The item's string value is passed to `trim()`, which just trims off the leading whitespace and bullet, if any. The next item in the concatenation is an `action()` item, which performs the indicated action, incrementing the current indentation level. Then follow zero or more subtrees, and when the parser has finished parsing the subtrees, it invokes another action to put the indentation level back the way it was.

This is `action()`:

```
sub action {
  my $action = shift;
  return parser {
    my $input = shift;
    $action->($input);
    return (undef, $input);
  };
}
```

It takes an action argument and generates a parser that invokes the action, consumes no tokens, and always succeeds.

As it stands, the value returned by *tree* is a list of four items, of which two are the meaningless `undef`s returned by the `action()` parsers. As usual, we'll use `T()` to adjust the return value to the tree structure that we want.

The tree structure we'll construct will be an array. The first element of the array will be the root node, and the other elements will be the subtrees, in order. The tree for this example is `["A", ["B", ["C"]], ["D"]]`:

```
* A
  * B
    * C
  * D
```

The main tree has two subtrees, one rooted at B and one at D. The B subtree has a sub-subtree, with root C.

`concatenate()` will pass `T()` four arguments, as noted before; two will be `undef`. The other two will be `$_[0]`, the string value returned by `lookfor()`, and `$_[2]`, an array of the subtrees. To assemble these into a tree structure, we just need to make them into a single array:

```
$tree = T(concatenate(lookfor('ITEM', sub { trim($_[0][1]) }),
                      action(sub { $LEVEL++; }),
                      star($Subtree),
                      action(sub { $LEVEL--; })),
          sub { [ $_[0], @{$_[2]} ] });
```

The other half of the parser is easier in some ways, more complicated in others:

```
$subtree = T(concatenate(test(sub {
                                my $input = shift;
                                return unless $input;
                                my $next = head($input);
                                return unless $next->[0] eq 'ITEM';
                                return level_of($next->[1]) >= $LEVEL;
                              }),
                         $Tree,
                        ),
             sub { $_[1] });

sub level_of {
```

```
  my ($space) = $_[0] =~ /^( *)/;
  return length($space)/2;
}
```

The parser built by `test()` is like the one built by `action()`, except that it doesn't always succeed. Instead, it looks at the value returned by the action, and succeeds only if the value is true:

```
sub test {
  my $action = shift;
  return parser {
    my $input = shift;
    my $result = $action->($input);
    return $result ? (undef, $input) : ();
  };
}
```

The action here makes sure there is an item coming up, and extracts the text from it. It then uses the `level_of()` utility function to figure out the nesting depth of the item's text, and returns success if it is far enough to the right, failure if not. If the action succeeds, the *subtree* parser then invokes `$Tree` to parse the subtree. Note that the item examined by the action is *not* removed from the input stream. If `$Tree` returns successfully, `T()` throws away the meaningless value returned by the action, and returns the tree structure from `$Tree` as the value of the subtree.

This parser is complete. Now we'll make it a little more clever, by having it detect when the entire input is indented (and ignore the uniform indentation if so) and having it handle different amounts of indentation at each level. These two inputs should parse the same:

```
* A
  * B
    * C
      * D
  * E
    * F
    * G
  * H
    * I
  * J
```

```
* A
      * B
                * C
                   * D
      * E
              * F
              * G
      * H
         * I
      * J
```

Now that indentations vary, the action that records the current indentation level will have to actually examine the current item; simply incrementing a counter is

no longer sufficient. We didn't give `action()` parsers any good way to examine values that have already been parsed, so the easiest way to get what we want is to attach the action to the `lookfor('ITEM')` parser:

CODE LIBRARY
outline-parser-2

```
my @LEVEL;
$tree = T(concatenate(T(lookfor('ITEM', sub { $_[0] }),
                        sub {
                          my $s = $_[1];
                          push @LEVEL, level_of($s);
                          return trim($s);
                        }),
                      star($Subtree),
                      action(sub { pop @LEVEL })),
          sub { [ $_[0], @{$_[1]} ]},
         );
```

The task of looking up the level of an item and recording it has been delegated to the `lookfor('ITEM')` parser, since it has easy access to the item's text. We use `T()` to hang an action on the `lookfor()` parser. But since `T()` was designed for use with `concatenate()`, which returns a list of values, we need to override the normal return value of `lookfor()` to deliver the entire token instead of just the string data as usual. The string is extracted from the token and assigned to `$s`, and its level is computed and pushed on a stack. The parser needs to record the levels in a stack because when it finishes parsing the current tree, it will need to recall the level of the parent root. Let's return to the small example:

```
* A
  * B
    * C
  * D
```

While parsing the C tree, the parser will remember that the current root, B, is indented two spaces. When it sees item D, also indented two spaces, it will know that D is not a sub-item of B. This will conclude the parsing of the B subtree, and the action function will pop the stack. The top stack item will then be 0, the indentation of A. Since D is to the right of this, D will be parsed as a subtree of A.

The *subtree* parser is almost identical to the previous version:

```
$subtree = T(concatenate(test(sub {
                                  my $input = shift;
                                  return unless $input;
```

```
                    my $next = head($input);
                    return unless $next->[0] eq 'ITEM';
                    return level_of($next->[1]) > $LEVEL[-1];
                  }),
              $Tree,),
        sub { $_[1] });
```

The only difference is that the action compares the level of the current item with the top of the stack rather than with a scalar counter.

`level_of()` is a little more complicated, because the entire outline might be indented, and this has to be accounted for:

```
my $PREFIX;
sub level_of {
  my $count = 0;
  my $s = shift;
  if (! defined $PREFIX) {
    ($PREFIX) = $s =~ /^(\s*)/;
  }
  $s =~ s/^$PREFIX//o
    or die "Item '$s' wasn't indented the same as the previous items.\n";
  my ($indent) = $s =~ /^(\s*)/;
  my $level = length($indent);
  return $level;
}
```

`$PREFIX` is a string of whitespace indicating the indentation of the very first item. All subsequent indentations are figured relative to this. The `$PREFIX` is trimmed off each item before its indentation is figured; if an item doesn't begin with the same `$PREFIX` as the others, the parser dies.

Here's the example tree again, and the result of parsing it:

```
* A                          [ 'A',
    * B                       [ 'B', [ 'C', [ 'D' ] ] ],
        * C                   [ 'E', [ 'F' ], [ 'G' ] ],
         * D                  [ 'H', [ 'I' ] ],
    * E                       [ 'J' ]
      * F                    ]
      * G
    * H
     * I
    * J
```

8.7 DATABASE-QUERY PARSING

In Chapter 4, we saw a simple database system, `FlatDB`, which supported the combination of simpler queries into more complex ones:

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

I promised that we would attach this to a parser so that we could simply write something like:

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

and get the same result. The lexer will turn this example into the following tokens:

```
[FIELD, 'STATE']
[OP, '=']
[STRING, 'NY']
[OR]
[FIELD, 'OWES']
[OP, '>']
[NUMBER, 100]
[AND]
[FIELD, 'STATE']
[OP, '=']
[STRING, 'MA']
```

8.7.1 The Lexer

Here's the lexer for our database query language. It has a few novel features:

CODE LIBRARY
dqp.pl

```
use Lexer ':all';

sub lex_input {
```

```
my @input = @_;
my $input = sub { shift @input };

my $lexer = iterator_to_stream(
      make_lexer($input,
                 ['STRING', qr/' (?: \\. | [^'] )*  '
                             |" (?: \\. | [^"] )*  " /sx,

                     sub { my $s = shift;
                           $s =~  s/.//; $s =~ s/.$//;
                           $s =~  s/\\(.)/$1/g;
                           ['STRING', $s] }                ],

                 ['FIELD',  qr/[A-Z]+/                     ],
                 ['AND',    qr/&/                          ],
                 ['OR',     qr/\|/                         ],

                 ['OP',     qr/[!<>=]=|[<>=]/,
                      sub { $_[0] =~ s/^=$/==/;
                           [ 'OP', $_[0] ] }               ],
                           ['LPAREN', qr/[(]/                      ],
                           ['RPAREN', qr/[)]/                      ],
                           ['NUMBER', qr/\d+ (?:\.\d*)? | \.\d+/x  ],
                           ['SPACE',  qr/\s+/, sub { "" } ],
                          )
          );
}
```

Most of this is old hat. The exception is the first rule, which recognizes string constants, which have a fairly complicated lexical structure. First, we'll look at the regex, which is in two parts, one for single-quoted strings and one for double-quoted strings. The two kinds of strings will have identical behavior in this language; we're including them both only to suit the preferences of users for one or the other. The obvious regex to match a single-quoted string is `/' .* '/x`, but a little thought shows this isn't correct, since it will match `'O'Reilly'`, which is syntactically incorrect. Really, the characters inside the quotes are forbidden to be quotes themselves, so we want `/' [^']* '/x`. But now there's no way to write a string that contains a single-quote character, so we introduce backslash escapes. If a backslash appears inside a single-quoted string, the following character is

accepted unconditionally, whether or not it's a quote. The pattern becomes `/'(?: \\. | [^'])* '/sx`. (The `/s` tells the regex engine that it is acceptable for the . character to match a newline; normally it won't.) The corresponding pattern for double-quoted strings is similar.

The default method for building a token from a quoted string isn't exactly what we want, because the input `'O\'Reilly'` represents the string `O'Reilly`, not `O\'Reilly`. So the `STRING` rule makes use of a feature we haven't used much yet, which is the option to specify a token-manufacturing function to replace the default:

```
sub { my $s = shift;
      $s =~ s/.//; $s =~ s/.$//;
      $s =~ s/\\(.)/$1/g;
      ['STRING', $s] }
```

The token-manufacturing function trims off the leading and trailing quotation marks, and then expands the backslash escapes. If we wanted to support more backslash escapes, such as `\t` for a tab character, this would be the place to do it:

```
sub { my $s = shift;
      $s =~ s/.//; $s =~ s/.$//;
      $s =~ s/\\t/\t/g;
      $s =~ s/\\(.)/$1/g;
      ['STRING', $s] }
```

Note that once a `STRING` token comes out of the lexer, the parser won't be able to tell whether it was originally a single-quoted string or a double-quoted string. Since the two kinds of strings have the same semantics, that's just what we want; there's no reason for the parser to concern itself with this irrelevant distinction.

This kind of behavior is typical of lexers. The lexer's job is to convert the incoming sequence of characters into meaningful tokens, and if two incoming sequences have the same meaning, they should turn into the same tokens. The Perl lexer does the same thing; by the time the parser gets hold of the input:

```
$a = "O'Reilly";
```

it can't tell that the input wasn't originally:

```
$a = 'O\'Reilly';
```

instead, or:

```
$a = qq{\x4f\47\x52\x65\151\x6c\x6c\171};
```

for that matter. Similarly, the Perl lexer absorbs the two keywords `for` and `foreach`, which are identical, and grinds both of them down to identical `OPERATOR(FOR)` tokens. By the time the parser gets the input, it doesn't know which one you wrote. We saw another example of this earlier, when the lexer for our calculator treated `/;\n*/` and `/\n+/` as the same kind of token.

The only other lexer rule that's a little different from those we've seen before is `OP`, the rule for matching operators. An operator is one of the following:

```
!= <= >= ==
   <  >  =
```

== and = are considered the same; we're including both forms as a convenience for the user. The token-manufacturing function transforms = to ==, again eliminating a distinction that is of no concern to the parser.

8.7.2 The Parser

The grammar for the query language is simple, and doesn't contain anything new:

```
complex_query → term star('OR' complex_query)

term → simple_query star('AND' term)

simple_query → 'FIELD' 'OP' 'STRING'
simple_query → 'FIELD' 'OP' 'NUMBER'
simple_query → 'LPAREN' complex_query 'RPAREN'
```

The code for *complex_query* and *term* is completely handled by the `operator()` function we wrote in Section 8.4:

```
use Parser ':all';
use FlatDB_Composable qw(query_or query_and);

my ($cquery, $squery, $term);
my $CQuery = parser { $cquery->(@_) };
my $SQuery = parser { $squery->(@_) };
my $Term = parser { $term->(@_) };
```

```
use FlatDB;

$cquery = operator($Term, [lookfor('OR'), \&query_or]);
$term = operator($SQuery, [lookfor('AND'), \&query_and]);
```

The code for `simple_query()` is more interesting, because it is the main interface with the flat database library.

Recall that there are three productions for *simple_query*:

```
simple_query → 'FIELD' 'OP' 'STRING'
simple_query → 'FIELD' 'OP' 'NUMBER'
simple_query → 'LPAREN' complex_query 'RPAREN'
```

The third of these is just like several other parsers that we've seen before:

```
# This needs to be up here so that the following $squery
# definition can see $parser_dbh
my $parser_dbh;
sub set_parser_dbh { $parser_dbh = shift }
sub     parser_dbh { $parser_dbh }

$squery = alternate(
            T(concatenate(lookfor('LPAREN'),
                          $CQuery,
                          lookfor('RPAREN'),
                         ),
              sub { $_[1] }),
```

The first two are separate productions because they have different meanings. `"STATE = 'NY'"` is quite different from `"AMOUNT = 0"`, because the `query()` method can be used for the first query but not for the second — recall that `query()` always uses `eq` to detect matches. Because the `FIELD OP NUMBER` production must always use `callbackquery()`, never `query()`, it's a little simpler and we'll see it first:

```
            T(concatenate(lookfor('FIELD'),
                          lookfor('OP'),
                          lookfor('NUMBER')),
              sub {
                my ($field, $op, $val) = @_;
                my $cmp_code = 'sub { $_[0] OP $_[1] }';
```

```
    $cmp_code =~ s/OP/$op/;
    my $cmp = eval($cmp_code) or die;
    my $cb = sub { my %F = @_;
                   $cmp->($F{$field}, $val)};
    $parser_dbh->callbackquery($cb);
  }),
```

The callback for the callback query is `$cb`. `$cb` sets up a hash `%F` that maps field names to values, indexes `%F` to get the value of the appropriate field, and compares the field value with the specified number `$val`. When `$op` is `==`, the comparison needs to be `$F{$field} == $val`; when `$op` is `<=`, the comparison needs to be `$F{$field} <= $val`, and so on. There are essentially two ways to do this. One way is to select a function that performs the comparison from a dispatch table:

```
%compare = ('==' => sub { my %F = @_; $F{$field} == $val },
            '<=' => sub { my %F = @_; $F{$field} <= $val },
            '<'  => sub { my %F = @_; $F{$field} <  $val },
            '>=' => sub { my %F = @_; $F{$field} >= $val },
            '>'  => sub { my %F = @_; $F{$field} >  $val },
            '!=' => sub { my %F = @_; $F{$field} != $val },
           );
my $cb = $compare{$op};
```

This technique requires the construction of six anonymous functions, from which we select one to use and discard the other five. The other technique is to generate the desired code by textual substitution at run time; this is what the preceding implementation does. The code for a comparison function is manufactured by replacing `OP` in `sub { $_[0] OP $_[1] }` with the actual value of `$op`, and then using `eval` to compile the resulting string. `$cb` then calls the resulting function to actually compare the values. With the `eval` approach we run the risk of accidentally generating syntactically incorrect code; building Perl code is always fraught with peril because Perl's syntax is so irregular. I chose the `eval` approach because the peril seemed small in this case, because the code was smaller, and because I didn't like the idea of manufacturing six functions on every call just to throw five of them away.[3]

3 One of the technical reviewers ridiculed me extensively for this decision, since the performance difference is negligible. But it wasn't for performance reasons that I disliked the idea of manufacturing six times as many functions as I needed. It was because I'm compulsive.

The `$parser_dbh` variable in the final line is a bit of a puzzle. `callbackquery()` is a method, that is called on a database handle object. In order to perform a `callbackquery()`, the parser needs to know which database to query.

The cleanest way to accomplish this might be to provide each parser with a user-parameter argument, which it would then pass to its sub-parsers. Then we would supply the desired database handle to the top-level parser, and it would percolate down to the `T()` function in `simple_query()`, which would use it as the target object of the `callbackquery()` method. Unfortunately, this would require redesigning all our parser-generation functions to accommodate a new argument format. At present, the argument convention for `alternate()` and `concatenate()` is utterly simple and straightforward: You just pass in the parsers you want to alternate or concatenate. If a user-parameter argument were allowed, it would have to be distinguished specially somehow. The `alternate()` and `concatenate()` functions wouldn't be able to recognize a user argument just by looking at it, the way they recognized the debugging labels of Section 8.4, because a user argument might have any value at all.

There's an alternative that is preferable in this case. In Chapter 2, we saw how to use user parameters to avoid having to communicate with callbacks via global variables. Now we're going to turn around and use a global variable anyway. But our "global" variable, `$parser_dbh`, won't be truly global. It will actually be a lexical variable whose scope includes the parser-callback functions. To use the parser, the caller will first set `$parser_dbh`, then invoke the top-level parser. To avoid requiring that the caller also be in the scope of `$parser_dbh`, we'll provide a setter function that has access to it:

```
my $parser_dbh;
sub set_parser_dbh { $parser_dbh = shift }
sub     parser_dbh { $parser_dbh }

$squery = alternate(...
                ... $parser_dbh->callbackquery($cb) ...
          );
```

The big drawback of *this* approach is that the parser can't change databases in the middle of a parse without saving `$parser_dbh` before and restoring it after. For our examples, this isn't important, but it means that we can't easily support queries on more than one database at a time.

With all that out of the way, here's the third production for *simple_query*:

```
T(concatenate(lookfor('FIELD'),
              lookfor('OP'),
```

```
            lookfor('STRING')),
  sub {
    if ($_[1] eq '==') {
      $parser_dbh->query($_[0], $_[2]);
    } else {
      my ($field, $op, $val) = @_;
      my $cmp_code = 'sub { $_[0] OP $_[1] }';
      $cmp_code =~ s/OP/$string_version{$op}/;
      my $cmp = eval($cmp_code) or die;
      my $cb = sub { my %F = @_;
                     $cmp->($F{$field}, $val)};
      $parser_dbh->callbackquery($cb);
    }
  }),
);
```

There are only two differences between this and the production for numeric comparison. First, there's a special case for when `$op` is ==. In this case, instead of using the slower and more general `callbackquery()` to perform the query, we use the simpler and faster `query()`, which is hardwired to do a string-equality test. Second, in the other cases, when we do use `callbackquery()`, instead of replacing `OP` with `$op` in the comparison function for the callback, we must replace `OP` with the string version of `$op` — if `$op` is <=, the string version is `le`, and so on. String versions are provided by a simple hash table:

```
my %string_version = ('>' => 'gt', '>=' => 'ge', '==' => 'eq',
                      '<' => 'lt', '<=' => 'le', '!=' => 'ne');
```

We need one last function to serve as an entry point from programs that want to use our parser:

```
package FlatDB::Parser;
use base FlatDB_Composable;

sub parse_query {
  my $self = shift;
  my $query = shift;
  my $lexer = main::lex_input($query);
  my $old_parser_dbh = main::parser_dbh();
  main::set_parser_dbh($self);
  my ($result) = $cquery->($lexer);
```

```
    main::set_parser_dbh($old_parser_dbh);
    return $result;
}
```

Our parser for database queries is now complete.

8.8 BACKTRACKING PARSERS

The parser technique we've seen so far has a serious problem: It doesn't always work. Here's a contrived but very simple example:

```
S → A B | B c c
A → a a | a
B → a b c | a b
```

Now consider the sentence `a a b c`. *S* will try the first alternative, *A B*. *A* will try its first alternative, `a a`, which will succeed. Then *B* will try both of its alternatives, each of which will fail, because they begin with `a` and the remaining input is `b c`. *B* will report failure, and so *A B* will fail. *S* will then try its second alternative, *B* `c c`, which will also fail, because the input doesn't end with `c c`.

But the grammar *does* generate the string `a a b c`, by the derivation:

```
S
A B             # S clause 1
a B             # A clause 2
a a b c         # B clause 1
```

Why didn't the parser find the answer? The problem is that after the first alternative for *A* succeeded, the rest of the parse failed. The parser should have tried the second alternative for *A*, since it would have led to success. But that's not how our `alternate()` function works. Once it commits to an alternative, it's too late for it to come back and try something else. The chosen alternative looked good in the short term. `alternate()` had no way to find out that its choice turned out bad in the long run, and that the parser called after it failed.

We've seen a number of useful parser examples already, and this problem didn't come up; often it doesn't. But what do we need to do if it does come up?

The `alternate()` parser can't assume success just because one or another of its alternatives succeeds. It would need to find out if the following parsing succeeded too. If so, then fine; if not, it would need to try another alternative.

8.8.1 Continuations

How can `alternate()` find out if the rest of the parse succeeded? It would need to invoke a parser for the entire rest of the input. We'll pass it one in its argument list.

We'll make each parser responsible for completing the entire parse. Parsers formerly handled a bit of input and then returned. Now they'll get an extra argument, called a *continuation*, for parsing the rest of the input after the bit that they've handled themselves. A parser will look for whatever it normally looks for, and if it is successful, it will invoke the continuation on the rest of the input, returning success if and only if the continuation succeeds.

In the preceding example, *B* is the continuation of *A*. *A* will try the first alternative, `a a`, which will succeed. *A* will then try its continuation, *B*, on the remaining input, `b c`; this will fail. So *A* will know that even though `a a` looked good, it ultimately fails, and is incorrect. *A* will then try the second alternative, `a`, which will also succeed, so *A* will try the continuation again, this time on the remaining input `a b c`. *B* will succeed this time, so *A* will report success, having selected the second alternative rather than the first.

With this structure, parsers no longer need to return the unused portion of the input. Instead, they'll pass the unused portion to the continuation. Formerly, parsers got one argument, which was the input stream, and returned two results, which were the calculated value and the remaining input stream. Parsers now get two arguments, which are the input stream and the continuation, and return one result, which is the calculated value.

Here's the rewrite of `alternate()` to handle continuation arguments:

CODE LIBRARY
alternate-cont

```
sub alternate {
  my @p = @_;
  return parser { return () } if @p == 0;
  return $p[0]                if @p == 1;

  my $p;
  $p = parser {
    my ($input, $continuation) = @_;
    for (@p) {
      if (my ($v) = $_->($input, $continuation)) {
        return $v;
      }
    }
    return;  # Failure
  };
  $N{$p} = "(" . join(" | ", map $N{$_}, @p) . ")";
  return $p;
}
```

`$p` tries the alternatives (the elements of `@p`) in order. When one leads to a complete successful parse, the resulting value is returned. If an alternative fails, `$p` tries the next alternative.

Note that `$p` doesn't have to invoke the continuation itself. It just passes the continuation to the chosen alternative; the chosen alternative will take care of invoking the continuation if it succeeds.

Who actually invokes the continuation? Parsers generated by `lookfor()` invoke the continuation:

CODE LIBRARY
lookfor-cont

```
sub lookfor {
  my $wanted = shift;
  my $value = shift || sub { $_[0] };
  my $u = shift;
  $wanted = [$wanted] unless ref $wanted;
  my $parser = parser {
    my ($input, $continuation) = @_;
    return unless defined $input;

    my $next = head($input);
    for my $i (0 .. $#$wanted) {
      next unless defined $wanted->[$i];
      return unless $wanted->[$i] eq $next->[$i];
    }
    my $wanted_value = $value->($next, $u);

    # Try continuation
    if (my ($v) = $continuation->(tail($input))) {
      return $wanted_value;
    } else {
      return;
    }
  };

  $N{$parser} = "[@$wanted]";
  return $parser;
}
```

The process of checking the next token to see if it's what was expected is exactly the same, and, as before, if this fails, then the parser fails and returns false. But if it succeeds, the `lookfor()` parser doesn't immediately return success. Instead, it

invokes the continuation to try to parse the rest of the input. It returns success only if the continuation succeeds as well.

The null parser is still simple. It does nothing, passing the buck to the continuation. If the continuation succeeds, `nothing()` is happy; if not, `nothing()` reports the failure to its caller:

CODE LIBRARY
nothing-continuation

```
sub nothing {
  my ($input, $continuation) = @_;
  return $continuation->($input);
}
```

The `End_of_Input()` parser doesn't change at all. It doesn't even get a continuation, because there's nothing left to do after `End_of_Input()` is finished. It just checks to make sure that the input is exhausted, succeeding if it is.

`concatenate()` is a little trickier. Suppose we have:

```
S -> Z blah de blah
Z -> A B
```

Here *Z* is the concatenation of *A* and *B*; the continuation of *Z* is a parser for `blah de blah`. What does *Z* do?

Clearly, the first thing it needs to do is to invoke the parser for *A*. But what is the continuation for *A*? It's *B*, followed by `blah de blah`. So the continuation for *A* is a parser that invokes *B* with `blah de blah` as *its* continuation. The code may make this clearer:

CODE LIBRARY
concatenate2-cont

```
sub concatenate2 {
  my ($A, $B) = @_;
  my $p;
  $p = parser {
    my ($input, $continuation) = @_;
    my ($aval, $bval);
    my $BC = parser {
      my ($newinput) = @_;
      return unless ($bval) = $B->($newinput, $continuation);
    };
    $N{$BC} = "$N{$B} $N{$continuation}";
    if (($aval) = $A->($input, $BC)) {
      return ([$aval, $bval]);
    } else {
      return;
```

```
    }
  };
  $N{$p} = "$N{$A} $N{$B}";
  return $p;
}
```

The parser for *A B* gets a continuation, which is a parser for `blah de blah`. It builds a new parser, `$BC`, which invokes *B*, telling *B* that it is followed by `blah de blah`. It then invokes *A*, giving it the continuation `$BC`. If both *A* and *B* succeed, it packages the two values into an array and returns the array.

This concatenates only two parsers. To concatenate more than two, we call `concatenate2()` repeatedly:

CODE LIBRARY
concatenate-cont

```
sub concatenate {
  my (@p) = @_;
  return \&nothing if @p == 0;
  my $p0 = shift @p;

  return concatenate2($p0, concatenate(@p));
}
```

`T()` doesn't change significantly. The parser it builds expects a continuation argument, which it passes along:

CODE LIBRARY
T-continuation

```
sub T {
  my ($parser, $transform) = @_;
  my $p = sub {
    my ($input, $continuation) = @_;
    if (my $v = $parser->($input, $continuation)) {
      $v = $transform->(@$v);
      return $v;
    } else {
      return;
    }
  };
  $N{$p} = $N{$parser};
  return $p;
}
```

There's one final detail: Where does the first continuation come from? This is almost the simplest part of the whole operation. Say the top-level symbol is *S*.

Like all parsers, the parser *S* expects two arguments: the upcoming input, and the continuation parser. The upcoming input, of course, is the entire input. The continuation of *S* is simply the `End_of_Input()` parser.

With these changes, the parser from the beginning of the section succeeds:

```
...
$S = alternate(concatenate($A, $B),
               concatenate($B, lookfor('c'), lookfor('c')));

my $results = $S->($input, \&End_of_Input);
```

`$results` is assigned a value representing the correct parse, $S \rightarrow A\ B \rightarrow$ `a` $B \rightarrow$ `a a b c`.

8.8.2 Parse Streams

We introduced continuations to solve the problems caused by `alternate()`, not really knowing whether an alternative has succeeded until the following parsers reached the end of the input. Another way to solve the problem is to change the parsers so that instead of returning a single result from the first parse they find, they return *all* the results from *all* possible parses. Since this would be a big waste of time in the event that the caller cared about only the first parse, we would want to do it in a lazy fashion, returning a stream of possible parses.

This solves the problem too, because in this model, `alternate()` returns a stream that is the lazy merge of the streams returned by its arguments; if any of these streams is empty, it is ignored. `concatenate()` is responsible for combining two streams of parses into a stream of concatenations; if either input is empty, so is the output. If the first argument to `alternate()` succeeds but some later-concatenated part of the parsing process fails, that later part will return an empty stream, the concatenation with the successful alternative will be empty, and `alternate()` will effectively skip that alternative.

Before we see the changes to the parser constructors themselves, here are some utility functions we'll need. `single()` manufactures a stream of length 1, with specified head and empty tail:

```
sub single {
  my $v = shift;
  node($v, undef);
}
```

We saw `union()` before; it gets a list of streams, and produces a single stream formed by appending the arguments end-to-end. `sunion()` is similar, except that its argument is a stream of streams rather than a list of streams:

```
sub sunion {
  my ($s) = @_;
  my $cur_stream;
  while ($s && ! $cur_stream) {
    $cur_stream = head($s);
    $s = tail ($s);
  }
  return undef unless $cur_stream;

  return node(head($cur_stream),
              promise { sunion(node(tail($cur_stream), $s)) }
             );
}
```

Note that this is unsuitable if parsers might produce an infinite stream of possible parses; if so, we'd need to use a different version of `sunion()` that mingled the argument streams instead of appending them end-to-end.

`lookfor()` is the same as before, except that it returns an empty stream on failure and a singleton stream on success:

```
sub lookfor {
  ...
  my $parser = parser {
    ...
    for my $i (0 .. $#$wanted) {
      next unless defined $wanted->[$i];
      unless ($wanted->[$i] eq $next->[$i]) {
        return undef;
      }
    }
    my $wanted_value = $value->($next, $u);
    return single([$wanted_value, tail($input)]);
  };

  $N{$parser} = "[@$wanted]";
  return $parser;
}
```

Similarly, `End_of_input()` and `nothing()` return singleton streams when they succeed:

```
sub End_of_Input {
  my $input = shift;
  defined($input) ? () : single(["EOI", undef]);
}

sub nothing {
  my $input = shift;
  return single([undef, $input]);
}
```

`alternate()` becomes much simpler; it's merely a call to `union()` to join together the streams returned by its argument parsers:

```
sub alternate {
  my @p = @_;
  return parser { return undef } if @p == 0;
  return $p[0]                   if @p == 1;
  my $p;
  $p = parser {
    my $input = shift;
    union(map $_->($input), @p);
  };
  $N{$p} = "(" . join(" | ", map $N{$_}, @p) . ")";
  return $p;
}
```

`concatenate()`, however, is trickier. To concatenate parsers *S* and *T*, we must first call *S* on the main input, returning a stream of `[$svalue, $input1]` pairs. We then call *T* on each of the `$input1` values, producing a stream of `[$tvalue, $input2]` pairs. The parser produced by `concatenate()` then produces a stream of `[[$svalue, $tvalue], $input2]` pairs for each `$tvalue` and its corresponding `$svalue`:

```
sub concatenate2 {
  my ($S, $T) = @_;
  my $p;
```

```
  $p = parser {
    my $input = shift;
    my $sparses = $S->($input);
    sunion(transform {
      my ($sv, $input1) = @{$_[0]};
      my $tparses = $T->($input1);
      transform {
        my ($tv, $input2) = @{$_[0]};
        [[$sv, $tv], $input2];
      } $tparses;
    } $sparses);
  };
  $N{$p} = "@N{$S, $T}";
  return $p;
}
```

This concatenates only two parsers; to concatenate more than two, we repeat the process as necessary:

```
sub concatenate {
  my @p = @_;
  return \&null if @p == 0;
  my $p = shift @p;
  return $p     if @p == 0;

  my $result = concatenate2($p, concatenate(@p));
  $N{$result} = "@N{$p, @p}";
  return $result;
}
```

Finally, `T` needs to be changed to apply its transformation function to each of the values returned by its argument parser:

```
sub T {
  my ($parser, $transform) = @_;
  my $p = parser {
    my $input = shift;
    transform {
      my ($v, $input1) = @{$_[0]};
      [$transform->($v), $input1];
```

```
    } $parser->($input);
  };
  $N{$p} = $N{$parser};
  return $p;
}
```

8.9 OVERLOADING

The recursive-descent parsing system works well, and it's easy to compose small parsers into bigger ones and to capture common patterns in larger parser-generating functions, such as `operator()` of Section 8.4.4. But the notation looks awful. We can clean this up a little bit by using Perl's operator-overloading feature, introduced in Chapter 7.

The most frequently-used parser operators have been `concatenate()`, `alternate()`, and `T()`. We'll overload three operators to invoke these functions, so that what we once wrote as:

```
$factor     = alternate(lookfor('INT'),
                        T(
                          concatenate(lookfor(['OP', '(']),
                                      $Expression,
                                      lookfor(['OP', ')'])),
                          sub { $_[1] })
                       );
```

will become:

```
$factor = lookfor('INT')
        | lookfor(['OP', '(']) - $Expression - lookfor(['OP', ')'])
                >> sub { $_[1] }
        ;
```

which looks almost exactly like the grammar rules it represents. The - is used for concatenation, the | is of course used for alternation, and the `>>` replaces `T()`, because it visually suggests that the value from the parser on the left is being passed into the function on the right. The precedences of these three operators just happen to have the order we need: - has higher precedence than both | and `>>`, so `a | b - c >> d` means `a | ((b - c) >> d)`, which is almost always what we want.

The remaining syntactic clutter is mostly the calls to `lookfor()`. If we want to trim this as well, it's easy: just give `lookfor()` a shorter name:[4]

```
sub _ { @_ = [@_]; goto &lookfor }

$factor = _("INT")
        | _('OP', '(') - $Expression - _('OP', ')')
                >> sub { $_[1] }
        ;
```

To pull this off, we must engage the Perl overloading features by turning parsers into objects blessed into a class that defines the operators. We'll change the formerly trivial `parser()` function to do that:

```
package Parser;

sub parser (&) { bless $_[0] => __PACKAGE__ }
```

We have the `Parser` class overload the relevant operators:

```
use overload '-'  => \&concatenate2,
             '|'  => \&alternate2,
             '>>' => \&T,
             '""' => \&overload::StrVal,
             ;
```

I've used `concatenate2()` and `alternate2()` here just in case I someday want to have a place to hang some extra semantics on the overloaded operators. In the meantime, these binary functions just call their more general counterparts:

```
sub concatenate2 {
  my ($A, $B) = @_;
  concatenate($A, $B);
}

sub alternate2 {
  my ($A, $B) = @_;
  alternate($A, $B);
}
```

4 It would be nice to get rid of the parentheses as well, but we can't, because `_ $x - $y` means `_($x - $y)`, rather than `_($x) - $y` as we'd like.

The function associated with the "" (*stringification*) pseudo-operator is invoked by Perl when a `Parser` object needs to be converted to a string. This occurs in debugging messages and also whenever we use a parser as a key to the debugging names in the `%N` hash. The function `\&overload::StrVal` restores the default behavior of converting the `Parser` objects to strings of the form `Parser=CODE(0x83bb36c)`. An alternative would be to associate the stringification operator with a function that looks up the parser's name in the `%N` hash:

```
use overload '""' => \&parser_name;

sub parser_name {
  my $parser = shift;
  my $key = overload::StrVal($parser);
  exists($N{$key}) ? $N{$key} : $key;
}
```

If we did this, we would never need to refer explicitly to `%N` in debugging messages; just printing out a parser as if it were a string would print the name assigned to it from the `%N` hash.

There are a few technical problems associated with the change. The assignment:

```
$parser = $A - $B - $C
```

is not exactly equivalent to:

```
$parser = concatenate($A, $B, $C) ;
```

but rather to:

```
$parser = concatenate(concatenate($A, $B), $C) ;
```

If the values returned by `$A`, `$B`, and `$C` are `a`, `b`, and `c`, the first parser would have returned `[a, b, c]`, but the overloaded version will return `[[a, b], c]`. It might seem that we could fix this by having `concatenate()` flatten its first argument. But a little thought shows that this won't work, because `$A - $B` would try to flatten `a`, which would be an error. And `concatenate()` can't have a policy of flattening its first argument whenever the first argument is an array reference, because `$A` might actually return a legitimate array reference value.

Probably the best way to handle this is to leave `concatenate()` mostly alone and let it return values like `[[[a, b], c], d]`. Almost all compound values

returned by `concatenate()` are passed through `T()` anyway, so we'll let `T()` take care of undoing the nesting of the values. But we need some way to signal to `T()` the difference between a value returned by `$A - $B - $C`, where `[[a, b], c]` needs to be flattened to `[a, b, c]`, and the same value returned by `$A - $C`, where `$A` happened to return an array reference `[a, b]`, which doesn't need to be flattened. A solution is to tag the pairs returned by `concatenate` so that `T()` can recognize them. Instead of:

```
for (@p) {
  ($v, $input) = $_->($input);
  push @values, $v;
}
return (\@values, $input);
```

we have:

```
for (@p) {
  ($v, $input) = $_->($input);
  push @values, $v;
}
return (bless(\@values => 'Pair'), $input);
```

We must also add code to `T()` to recognize and flatten lists of these special `Pair` objects:

```
my ($value, $newinput) = $parser->($input);
# Flatten nested lists returned by concatenate()
my @values;
while (ref($value) eq 'Pair') {
  unshift @values, $value->[1];
  $value = $value->[0];
}
unshift @values, $value;
$value = $transform->(@values);
return ($value, $newinput);
```

Only a few other changes need to be made. Expressions like:

```
T(\&nothing, sub { 1 })
```

don't translate as they should. The corresponding expression,

```
\&nothing >> sub { 1 }
```

doesn't work, because `\&nothing` is not an overloaded `Parser` object. (It hasn't been blessed.) The solution is simple; instead of using `\&nothing` directly, provide a blessed version:

```
bless($nothing = \&nothing, "Parser");
$N{$nothing} = "(nothing)";
```

and then use `$nothing` instead of `\&nothing`:

```
$nothing >> sub { 1 }
```

With the new syntax, the parser for the final calculator example of Section 8.4.6 becomes:

```
$program = star($Statement) - $Parser::End_of_Input;

$statement = _("PRINT") - $Expression - _("TERMINATOR")
                  >> sub { print ">> $_[1]\n" }
           | _("IDENTIFIER") - _('OP', '=')
                             - $Expression - _("TERMINATOR")
                  >> sub { $VAR{$_[0]} = $_[2] };

$expression = $Term - star(_('OP', '+') - $Term
                               >> sub { my $term = $_[1];
                                        sub { $_[0] + $term } }
                          |
                           _('OP', '-') - $Term
                               >> sub { my $term = $_[1];
                                        sub { $_[0] - $term } }
                          )
  >> sub { my ($first, $rest) = @_;
           for my $f (@$rest) {
             $first = $f->($first);
           }

           $first;
         };


$term = $Factor - star(_('OP', '*') - $Factor
```

```
                                    >> sub { my $factor = $_[1];
                                             sub { $_[0] * $factor } }
                                  |
                                   _('OP', '/') - $Factor
                                    >> sub { my $factor = $_[1];
                                             sub { $_[0] / $factor } }
                                  )
          >> sub { my ($first, $rest) = @_;
                   for my $f (@$rest) {
                     $first = $f->($first);
                   }
                   $first;
                 };


$factor = $Base - (
                    _('OP', '**') - $Factor >> sub { $_[1] }
                   |
                    $Parser::null >> sub { 1 }
                   )
  >> sub { $_[0] ** $_[1] };
$base = _("INT")
      | lookfor('IDENTIFIER', sub { $VAR{$_[0][1]} })
      | _('OP', '(') - $Expression - _('OP', ')')
                >> sub { $_[1] }
;
```

The overloaded version is substantially less bulky and much easier to read.