Parsing systems in functional programming languages

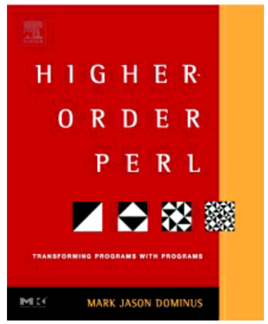
Mark Dominus

Plover Systems Co.

`mjd-perl-hop@plover.com`

Version 1.0

February, 2007



Warning

- Code examples in this talk will be in Perl

Warning

- This is about **functional** programming languages
- SML, Haskell, Perl, etc.
  - A lot of this stuff will be difficult or impossible in C, Java, etc.
- Too bad

Parsing

- *Parsing* is the process of taking an unstructured input
  - Such as a sequence of characters
- and turning it into a data structure
  - Such as a record
  - or an object
  - or a value
- For example, read a configuration file
  - build an object that represents the configuration

Every program parses

- This is a rudimentary parser:

```
while (read a line of input) {
  # do something with it
}
```

- The program must here convert an unstructured character stream into a sequence of lines
- As the input you're parsing becomes more complicated, the code becomes more elaborate
- At some point it may exceed your ability to keep up with ad-hoc mechanisms
- So we have parsing systems like `yacc` and `Parse::RecDescent`

Open vs. closed systems

- Some people like *closed* systems
  - The system should just do all the stuff you need it to
  - It should have a feature for every use-case
  - You should be able to use it without understanding what it is doing
  - Example: Microsoft Windows

Open vs. closed systems

- I prefer *open* systems
  - The system should provide modules for doing simple common things
  - The modules should be composable into specialized assemblages
  - It should be possible to assemble a solution for every use-case
  - It should be easy to build new modules
  - Example: Unix

Open vs. closed systems

- Benefit of open systems:
  - Flexible, powerful, unlimited
- Drawback:
  - Requires more understanding
- We're going to see an open one, `HOP::Parser`

Example: graphing program

- Suppose we want to read a web user's input
  - It will be a mathematical function, like

    ```
    (x^2 + 3*x)* sin(x * 2) + 14
    ```
- We will emit a web page with a graph of their function
- In Perl, there is an easy solution:
  - Use `eval` to turn the input string into compiled Perl code
- You could imagine something similar for almost any language:
  - Write out a source code file with a suitable function in it
  - Embed the user input in the appropriate place in the file
  - Compile the file and execute the resulting binary

Example: graphing program

- In Perl:

  ```
  my $function = eval $code;
  my $y = $function->($x);
  ```
- I don't need to explain all the things that can go wrong here, do I?
- Even if it could be made safe, it has some problems:

  ```
  (x^2 + 3*x)* sin(x * 2) + 14
  ```
- In Perl, `^` means bitwise exclusive or
  - Not exponentiation
- Alternative: implement an evaluator for expressions
  - Then we can give any notation any meaning we want

Grammars

```
atom &rarr; NUMBER | VAR | FUNCTION "(" expression ")"

factor &rarr; atom ("^" NUMBER | nothing)

term &rarr; factor ("*" term | nothing)

expression &rarr; "(" expression ")"
            | term ("+" expression | nothing)
```


Lexing

- First, our program must identify things like `NUMBER`
- Idea: preprocess the input
  - Turn it from a character string into a list of *tokens*
  - Each token is an atomic piece of input
  - Examples: `sin`, `x`, `+`, `12345`
- Humans do this when they read
  - First, turn the sequence of characters into a sequence of words
  - Then, try to understand the structure of the sentence based on meanings of words
- This is called *lexing*

Lexing

- I will omit the arcane but tedious details of building a lexer
- See for example *The Unix Programming Environment* by Kernighan and Pike
- We will assume that the lexer returns tokens like this:

```
1234            ["NUMBER", 1234]
sqrt            ["FUNCTION", "sqrt"]
x3              ["VAR", "x3"]
^               ["^"]
**              ["^"]
+               ["+"]
*               ["*"]
(               ["("]
)               [")"]
```

- Notice how the lexer can recognize both `^` and `**` and eliminate the distinction
  - This saves work in the parser
- Also notice that `**` is lexed as a power operator, not as two multiplication signs
- We will imagine that our lexer scans the entire input immediately
  - Returns a linked list of all tokens

Recursive-descent parsing

- Each grammar rule has a corresponding function
  - The job of the function `expression()` is to parse an expression
  - If it succeeds, it returns a data structure representing the expression
  - If not, it returns a failure indication
- Suppose you have a rule like this:

```
expression &rarr; "(" expression ")"
             | term ("+" expression | nothing)
```

- You will have functions called `expression()` and `term()`
- `expression()` gets the token list as an argument
- It looks to see if the next token is `(`
  - If so, it calls itself recursively, and then looks for the `)`
- Otherwise it calls `term()` to look for a term
  - If `term()` fails, `expression()` does too
  - Otherwise it looks to see if there's a `+` sign and another expression

Recursive-descent parsing

- The description on the previous slide sounds complicated
- But there are only a few fundamental operations:
  - Look for a certain token
  - Look for either of *x* or *y*
  - Look for *x* followed by *y*
  - Look for nothing
- A `HOP::Parser` parser will be a function that takes a token list
  - It examines some tokens
  - If it likes what it sees, it constructs a value
  - Then it returns the value and a list of the remaining tokens
  - Otherwise, it returns `undef` (Perl "null" value)

Basic parsers

```
expression &rarr; "(" expression ")"
            | term ("+" expression | nothing)
```

- The simplest parser is the one that corresponds to `nothing`
- It consumes no tokens and always succeeds:

```
sub nothing {
  my $tokens = shift;
  return (undef, $tokens);
}
```

- This *parser function* gets a token list
  - It examines the tokens
  - Returns a value and a new token list
- The `undef` here is a dummy value
  - The new token list is the same as the old one

Token parsers

- The next simplest parser looks for a particular token:

```
sub lookfor_PLUS {
  my $tokens = shift;
  my $tok = first($tokens);
  if ($tok->type eq "+") {
    return ("+", rest($tokens));
  } else {
    return;           # failure
  }
}

sub lookfor_NUMBER {
  my $tokens = shift;
  my $tok = first($tokens);
  if ($tok->type eq "NUMBER") {
    return ($tok->value, rest($tokens));
  } else {
    return;           # failure
  }
}
```

- Note that the "value" returned by `lookfor_NUMBER` is the value of the number token it finds

Token parsers

- In functional languages, we needn't write write 9 similar `lookfor` functions
- Instead, we can have another function build them as required:

```
sub lookfor {
  my $target = shift;
  my $parser =
    sub {
      my $tokens = shift;
      my $tok = first($tokens);
      if ($tok->type eq $target) {
        return ($tok->value, rest($tokens));
      } else {
       return;           # failure
      }
    };
  return $parser;
}
```

- Now instead of `lookfor_PLUS` we just use `lookfor("+")`
- Instead of `lookfor_NUMBER` we just use `lookfor("NUMBER")`

Concatenation

- Let's pretend for a bit that `atom` has only this one rule:

```
atom &rarr; "FUNC" "(" expression ")"
```

- We could write `atom()` like this:

```
sub atom {
  my $t1 = shift;
  my ($expr, $t2, $t3, $t4, $t5);

  if (   ($funcname, $t2) = lookfor("FUNC")->($t1)
      && (undef,     $t3) = lookfor("(")->($t2)
      && ($expr,     $t4) = expression($t3)
      && (undef,     $t5) = lookfor(")")->($t4)) {
        my $val = something involving $funcname and $expr;
        return ($val, $t5);
  } else {
        return;  # failure
  }
}
```

- Most of our parser functions would look something like this
- So instead we'll write a function that assembles small parsers into big ones
- Given parser functions *A*, *B*, etc.:

```
conc(A, B, ...)
```

- Will return a parser function that looks for A, then B, etc.

Concatenation

```
sub conc {
  my @p = @_;
  my $parser = sub {
    my $tokens = shift;
    my @results;
    for my $p (@p) {
      my ($result, $t_new) = $p->($tokens)
        or return;   # failure
      push @results, $result;
      $tokens = $t_new;
    }

    # all parsers succeeded
    return (\@results, $tokens);
  };
  return $parser;
}
```

- With this definition, `atom` simply becomes:

```
$atom = conc(lookfor("FUNC"),
             lookfor("("),
             $expression,
             lookfor(")"),
            );
```

Concatenation

- Similarly, the rule

```
expression &rarr; "(" expression ")"
```

- Translates to:

```
$expression = conc(lookfor("("),
                   $expression,
                   lookfor(")"),
                  );
```

- Oops, no, not quite
- In better functional languages, this is no problem
- Even in Perl, this is fixable—but I don’t have time to fix it

Alternation

- Atoms come in three varieties, not just one:

```
atom &rarr; NUMBER | VAR | function "(" expression ")"
```

- So we need the `atom` parser to try these three different things
- It fails only if the upcoming tokens match none of them
- Something like this:

```
sub atom {
  my $in = shift;
  my ($result, $out);
  my $alt3 = conc(lookfor("FUNC"),
                  lookfor("("), $Expression,  lookfor(")"),
                 );

  if (     ($result, $out) = lookfor("NUMBER")->($in)) {
    return ($result, $out);
  } elsif (($result, $out) = lookfor("VAR")->($in)) {
    return ($result, $out);
  } elsif (($result, $out) = $alt3->($in)) {
    return ($result, $out);
  } else {
    return;
  }
}
```

- But again, we’d have to write a lot of code that was very similar
- So instead we’ll write a function that assembles small parsers into big ones
- Given parser functions *A*, *B*, etc.:

```
alt(A, B, ...)
```

- Will return a parser function that looks for A or for B, etc.

Alternation

```
sub alt {
  my @p = @_;
  my $parser = sub {
    my $in = shift;
    for my $p (@p) {
      if (my ($result, $out) = $p->($in)) {
        return ($result, $out);
      }
    }
    return;   # failure
  };
  return $parser;
}
```


Parsers

- With this definition, a complete definition of `atom()` is:

```
$atom = alt(lookfor("NUMBER"),
            lookfor("VAR"),
            conc(lookfor("FUNC"),
                 lookfor("("),
                 $Expression,
                 lookfor(")"),
                ));
```

- Similarly, here's `factor()`:

```
#  factor &rarr; atom ("^" NUMBER | nothing)

$factor = conc($Atom, alt(conc(lookfor("^"),
                               lookfor("NUMBER")),
                          \&nothing));
```

- Here's `term()`:

```
#  term &rarr; factor ("*" term | nothing)

$term = conc($Factor, alt(conc(lookfor("*"), $Term),
                          \&nothing));
```

Parsers

- Here's `expression()`:

```
# expression &rarr; "(" expression ")"
#              | term ("+" expression | nothing)

$expression = alt(conc(lookfor("(")),
                       $Expression,
                       lookfor(")"),

                  conc($Term,
                       alt(conc(lookfor("+"), $Expression),
                           \&nothing));
```

- This doesn't look great, but:

1. When you consider how much it's doing, it's amazingly brief, and
2. We can use operator overloading and rewrite it as:

```
$expression = L("(") - $Expression - L(")")
            | $Term - (L("+") - $Expression | $nothing);
```


Overloading

```
# expression &rarr; "(" expression ")"
#              | term ("+" expression | nothing)

$expression = L("(") - $Expression - L(")")
            | $Term - (L("+") - $Expression | $nothing);
```

- This looks almost exactly like the grammar rule we're implementing
    - But it's actually Perl code, not a limited sub-language
- We can do similar tricks in SML or Haskell
- I'll use this notation from now on

Parsers

- So far we've done a bunch of work to build a parser system
- It has some modular, interchangeable parts
    - We can use these to manufacture all kinds of parsers
- Our system is only getting started

Optional items

- Many rules are naturally expressed in terms of "optional" items
- Instead of:

```
term &rarr; factor ("*" term | nothing)
```

- We might want to say something like:

```
term &rarr; factor optional("*" term)
```

- We can define `optional` quite easily:

```
sub optional {
  my $p = shift;
  return alt($p, $nothing);
}
```

- Now this:

```
$term = $Factor - (L("*") - $Term | $nothing);
```

- Becomes this:

```
$term = $Factor - optional(L("*") - $Term);
```

`repeat`

- Many rules are naturally expressed in terms of "repeated" items
- For example, we might write

```
# term &rarr; factor repeat( "*" factor )
$term = $Factor - repeat(L("*") - $Factor);
```

- It’s not hard to express `repeat` with what we have already:

```
# repeat($p) is:

$p - repeat($p)  |  $nothing
```

- But we can wrap this up as a function:

```
sub repeat {
  my $p = shift;
  my $repeat_p;
  my $do_repeat_p = sub { $repeat_p->(@_) };  # proxy
  $repeat_p = alt(conc($p, $do_repeat_p), $nothing);
  return $repeat_p;
}
```


Lists

- Comma-separated expression lists are common in programming languages
- Similarly semicolon-separated statement blocks
- Or ...

```
sub list_of {
  my ($item, $separator) = @_;
  $separator = lookfor("COMMA") unless defined $separator;
  conc($item, repeat($separator, $item), optional($separator));
}
```

- Now comma-separated lists:

```
$list = conc(lookfor("("),
             list_of($Expression),
             lookfor(")"));
```

- Semicolon-separated statement blocks:

```
$block = conc(lookfor("{"),
              list_of($Statement, lookfor(";")),
              lookfor("}"));
```

Operators

- Parsing arithmetic-type expressions is not too uncommon
- A useful utility is an `operator` function:

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

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

- This little bit of code writes a function that parses an input like `2 + 3 * 4` and calculates the result (14)
- For technical reasons, getting `-` and `/` to work requires some tricks
  - The complications are encapsulated inside of `operator`
  - We don't have to worry about them

New tools

- We've built all this up just by gluing together a very few basic tools:

```
lookfor()
conc()
alt()
```

- But the tools themselves are simple
  - only about 25 lines of code, total
- If we need some new tool, we can build it
- For example, "look for *A*, but only if it doesn't also look like *B*":

```
sub this_but_not_that {
  my ($A, $B) = @_;
  my $parser = sub {
    my $in = shift;
    my ($res, $out) = $A->($in)
        or return;
    if ($B->($in)) { return; }
    return ($res, $out);
  };
  return $parser;
}
```

New tools

- Or "do what *A* does, but transform its result value somehow":

```
sub transform {
  my ($A, $transform) = @_;
  my $parser = sub {
    my $in = shift;
    my ($res, $out) = $A->($in)
        or return;
    return ($transform->($res), $out);
  };
}
```


New tools

- Or "do what *A* does, but only if the result satisfies some condition":

```
sub side_condition {
  my ($A, $condition) = @_;
  my $parser = sub {
    my $in = shift;
    my ($res, $out) = $A->($in)
        or return;
    unless ($condition->($res)) { return; }
    return ($res, $out);
  };
}
```

New tools

- In my book *Higher-Order Perl*, I put the same tools to work parsing very different sorts of input
- Example: Take an outline:

```
. Languages
  . Functional
    . Haskell
  . Imperative
    . C
    . Fortran
  . OO
    . C++
    . Smalltalk
    . Simula
```

- Read it in, preserving the structure:

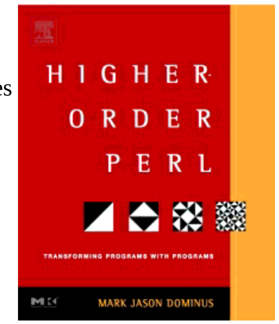
```
[ "Languages",
  [ "Functional", [ "Haskell" ]],
  [ "Imperative", [ "C", "Fortran" ] ],
  [ "OO", [ "C++", "Smalltalk", "Simula" ]]]
```

- The same set of tools does many different jobs

Warnings

- I had to leave out a lot of crucial details
- Recursive descent parsers need *backtracking*
  - I completely ignored this important issue
- The operator overloading is not as simple as I pretended
- Etc.
- But I don't think I misled you too badly

Higher-Order Perl



- I wrote a book about functional programming techniques in Perl
  - It was published in 2005 by Morgan Kaufmann
- It's a really good book (http://hop.perl.plover.com/reviews.html)
- Chapter 8, on parsing, is 90 pages long
  - I had to leave out a lot of good stuff for this talk

    ```
    http://hop.perl.plover.com/
    ```

- Eventually it will be available online
  - Meantime, source code is at:

    ```
    http://hop.perl.plover.com/Examples/Chap8/
    ```


Thank You!

- Any questions?

Bonus slides

- I prepared 90 minutes' worth of material for this 60-minute talk
- Here is the stuff I cut out to make room

Lexing

- Lexing is mostly a matter of simple pattern matching
- We build a scanner that works its way through the input string a character at a time
- It executes a state machine
- When the state machine indicates that a complete token has been read, the lexer returns the token
- In C, we can also use the program `lex` to generate the state machine
- In Perl, we usually use regular expressions

Recursive-descent parsing

- Idea: each grammar rule becomes a function
- A parser function gets the current token list as an argument
  - It can examine the tokens at the head of the list
  - It can pass all or part of the list to another parser
  - If it likes what it sees, it returns a success value
  - In this case, it informs its caller of how many tokens it consumed from the input
    - Probably by returning a suitable suffix of the original list

Labeled blocks

- Lately my big project has been a constraint-oriented drawing system called `linogram`
- The input language contains constructions like:

```
constraints { ... }
```

- And:

```
define square extends rectangle { ... }
```

- So I use an even higher-level parser constructor:

```
sub labeled_block {
  my ($header, $item, $separator) = @_;
  $separator = lookfor(";") unless defined $separator;
  conc($header,
       lookfor("{"),
       list_of($item, $separator),
       lookfor("}"));
}
```

- And define really complex parsers with it:

```
$constraint_block =
  labeled_block(L("CONSTRAINTS"), $constraint);

$definition =
  labeled_block($Definition_header, $declaration);
```

Open systems again

- Sorry to keep harping on this, but I think it's important

  1. By providing a few interchangeable parts, we enable not only powerful parsers

     - But ways to build *tools* to build *even more powerful* parsers

  2. Since the tools themselves are simple, it's easy to make new ones

     - A small amount of effort put into new tools pays off big

`repeat`

- Many rules are naturally expressed in terms of "repeated" items

- For example, we might write

```
# term &rarr; factor repeat( "*" factor )
$term = $Factor - repeat(L("*") - $Factor);
```

- It's not hard to express `repeat` with what we have already:

```
# repeat($p) is:

$p - repeat($p)  |  $nothing
```

- But we can wrap this up as a function:

```
sub repeat {
  my $p = shift;
  my $repeat_p;
  my $do_repeat_p = sub { $repeat_p->(@_) };  # proxy
  $repeat_p = alt(conc($p, $do_repeat_p), $nothing);
  return $repeat_p;
}
```