PEGs in a PEG

So I was reading Bryan Ford's thesis about parsing expression grammars and packrat parsers, and I thought it would be fun to implement them and see how easy they really were.

It turns out they're not that hard; this document contains a one-page PEG parser generator that generates PEG parsers in JavaScript, along with an explanation of how it works, and some example applications. If you've ever thought that writing a compiler was deep magic because parsing would take you way too long to understand, this should show you that writing a compiler can be simple! (At least, if you already know how to program.)

What Are PEGs?

A PEG is a formal language description which describes how to parse some language — like a regular expression, it describes the structure of some set of strings.

A Gentle Introduction by Example

Here's a simple PEG which describes simple arithmetic expressions with no operator precedence:

```
# in an example arithmetic parser:
sentence <- ('0' / '1' / '2' / '3' / '4' / '5' / '6' / '7' / '8' / '9')+
            ( ('+' / '-' / '*' / '×' / '/' / '÷') sentence / ).
```

This says that a `sentence` is one or more digits, followed by either an operator and another `sentence`, or nothing. The parentheses are used for grouping; apostrophes `''` are used for literal text; slashes `/` are used for choice ("try parsing this, and if it doesn't work out, try that"); a left arrow `<-` is used to attach a name (called a "nonterminal") to a parsing rule; and `x+` means "one or more of `x`".

(Typically, each of the strings that belongs to a language, such as a program in a programming language, is called a "sentence" of that language; thus my choice of that nonterminal name.)

So, to parse `2*30+4` as a `sentence`, first we try matching a `0` at the beginning, where there's a `2`; that doesn't work, so we try a `1`; that doesn't work, so we try a `2`. That does work, so then we try for repetition, looking for a second digit at the `*`. That doesn't work out (after ten tries), so we zoom along and look for a `+`. The `*` isn't a `+`, so after a couple of tries, we find out it's a `*`. Then we try parsing a nested `sentence` starting at the `3`. This time, we match the `3` after three tries, and then when we look for a second digit, we find a `0`; the third try fails, so we look for a `+`, and find it; then we look for a second nested `sentence`. We match a `4` after four tries, but we don't find another digit after it (because there isn't anything after it), so we try to find an operator after it,

which doesn't work, so we try to find nothing after it (the emptiness after the `/` after `sentence`) which works, and we're done.

Notice that this doesn't respect operator precedence (it gives `2*(30+4)` rather than `(2*30)+4`), and also associates to the right.

Here's an example PEG that handles operator precedence and parentheses, although not associativity:

```
# in an example arithmetic parser with precedence:
sentence   <- term ('+'/'-') sentence / term.
term       <- atom ('*' / '×' / '/' / '÷') term / atom.
atom       <- number / '(' sentence ')'.
number     <- ('0' / '1' / '2' / '3' / '4' / '5' / '6' / '7' / '8' / '9')+.
```

If we try to parse the same `2*30+4` with this grammar, we get down to `number` and parse the `2`, so `atom` succeeds with the `2`, and then `term` sucks up the `*` and then looks for an inner `term` at the `3`. Then `number` parses `30`, and the inner `term` looks for one of `*×/÷` after it, which doesn't work out since what's after it is a `+`, so it gives up on its first alternative and tries to parse just an `atom` starting at the `3`, rather than an `atom` followed by an operator and another term. Then `atom` sucks up the `30` just like before, and the inner `term` finishes, and then the outer `term` finishes, and it's up to `sentence` to deal with the `+4` bit, which it does in the predictable way.

It won't handle `40-1-1` as `(40-1)-1` as you might hope, though. If you try to rewrite `sentence` to handle this as `sentence ('+'/'-') term / term`, you run into trouble — the first thing `sentence` does is try to parse a `sentence`, so you get into an infinite loop. There are different ways to ameliorate this problem by enhancing the parser generator, but in general, you can always figure out a way to modify the grammar to remove this "left recursion"; it just makes it a little more complicated to handle the results of the parser.

(As an aside, most practical PEG systems let you abbreviate things like `('0' / '1' / '2' / '3' / '4' / '5' / '6' / '7' / '8' / '9')` as `[0-9]`, but the one in this document doesn't.)

That covers most of the stuff PEGs can do. A few things to notice:

1. They're a little more verbose than regular expressions but a lot more powerful at understanding structure. And, like with regexps, you can do a hell of a lot in a few lines of code.
2. The obvious implementation is pretty slow; it spends a lot of time re-parsing things it's already parsed and playing Cheese Shop with the next character. ("Have you got a 0?" "No." "How about a 1?" "No." …) It turns out there are ways to solve this, although I don't explore them in this document.
3. They have trouble with "left recursion", which is where the first thing in a "foo" (say, `sentence`) can be a smaller "foo".

There's one more big feature of PEGs: the ability to do negative lookahead, or negation. As an example, in C, a comment begins at a `/*` and continues until the next `*/`. But you can have `*` and `/` and even `/*` inside the comment, as long as there isn't a `*/`. Doing this in a regexp is a real pain and the result is unreadable. You end up with a regexp like `\/\*([^*]|\*[^/])*\*\/`, assuming you have to backslash your slashes. In a PEG, it looks like this:

```
# in the C comment example PEG:
comment <- '/*' (!'*/' char)* '*/'.
```

That is, to parse a comment, first parse a `/*`, then as long as the next thing isn't a `*/`, try to parse a `char`, and then parse a `*/`. (The `*` means "zero or more", just as `+` means "one or more".) You can write the same thing with Perl's enhanced regexp features: `qr|/\*(?:(?!\*/).)*\*/|`, and it's only slightly shorter, but I think it's not as clear.

You might think that in `!'*/' char`, the negation of `'*/'` somehow *modifies* `char`. But it doesn't, really; it just means that the parse fails at points in the input where `'*/'` can match, so `char` doesn't get a chance to match there. Instead, we backtrack from matching the `'*/'`, break out of the loop, and get a chance to match the `'*/'` on the outside.

You can use this magic PEG power for a variety of things that are traditionally painful. For example, most programming languages have keywords, which look like variables (or other identifiers) but are treated differently syntactically. In a PEG, you can write this:

```
# in the keyword example PEG:
keyword = ('if' / 'while' / 'for' / otherkeyword) !idchar.
identifier = !keyword idstartchar idchar*.
```

This first specifies that a `keyword` is one of the specified words as long as it's not followed by an `idchar`; then it specifies that when you're trying to parse an `identifier`, first try to parse a `keyword`, and if that succeeds, then parsing the `identifier` should fail; but if there's no `keyword`, go ahead and try to parse an `idstartchar` followed by zero or more `idchar`s.

Note that we throw away the results of trying to parse the `keyword` — we were only trying it in order to see if we shouldn't do something else.

If You've Taken a Compilers Class Lately

I thought I'd stick this section in for the benefit of folks who are all up on the theory. The rest of the document doesn't depend on it.

PEGs specify how to *parse* a language, by contrast with context-free grammars, which primarily describe how to *generate* sentences of a language. This difference makes it much easier to construct parsers for PEGs; they can be straightforwardly converted into simple recursive-descent parsers performing limited backtracking,

with each nonterminal becoming a parsing function. It also probably makes it much more difficult to prove properties of the language recognized by a PEG.

PEGs can parse some languages that context-free grammars can't, such as the language `anbncn`, that is, some number of `a`s, followed by the same number of `b`s, followed by the same number of `c`s. However, because PEGs can't handle ambiguity, and because there's a linear-time parsing algorithm for them, it is suspected that PEGs can't parse all languages context-free grammars can. `S → a S a | a S b | b S a | b S b | a` is a simple context-free language which Ford conjectured cannot be parsed with a PEG; it describes strings of odd numbers of `a`s and `b`s in which the middle letter is an `a`. PEGs can parse all languages that can be parsed with LL(k) or LR(k) parsers.

PEGs are more composable than LL(k) or LR(k) CFGs; because PEGs can't handle ambiguity, it's easy to predict the effect of adding new parsing rules to the grammar.

You can parse general CFGs with a backtracking approach like the PEG approach; the difference is that each nonterminal must be able to succeed multiple times on the same input with different possible parses, in case something that follows it fails. Definite clause grammars in Prolog are one example of this strategy. In PEGs, once a nonterminal succeeds at some position, it throws away its backtracking state, so it can only produce at most one result at that position. As a consequence, even though there are PEGs that take exponential time to parse (if implemented the naïve way) CFGs with exponential-time parsing (again, if implemented the naïve way, as with DCGs) are much more common.

(Allan Schiffman tells me that all you really need to do to make DCGs perform well is to put cuts in "the obvious places", e.g. between statements. I haven't tried it myself.)

A Minimal PEG Language

The expressions in PEGs minimally contain (using the TDPL notation in the thesis) negation `!`, ordered choice or alternation `/`, concatenation or sequencing (denoted by juxtaposition), terminal strings (written in single quotes `''`), and nonterminals (written as bare words `foo`). (We can leave out repetition `*` and `+`, because as shown below, we can synthesize them.)

Here's a relatively minimal grammar describing a notation for a grammar with these features, the same one I used in the "Gentle Introduction" section, written in terms of itself:

```
# in a minimal parsing expression grammar:
_               <- sp _ / .
sp              <- ' ' / '\n' / '\t'.
sentence        <- _ rule sentence / _ rule.
rule            <- name _ '<-'_ choice '.'_.
choice          <- sequence '/'_ choice / sequence.
```

```
sequence       <- term sequence / .
term           <- '!'_ term / '\'' stringcontents '\''_ / name _.
stringcontents <- stringchar stringcontents / .
stringchar     <- !'\\' !'\'' char / '\\' char.
name           <- namechar name / namechar.
namechar       <- !'!' !'\'' !sp !'<-' !'/' !'.' char.
```

This all depends on the primitive nonterminal `char`, which I'm assuming matches any character, for some definition of character.

The nonterminal `_` consumes any amount of whitespace. It's used everywhere we want to consume whitespace, generally at the lowest possible level of the grammar, with the exception of `name` (on the theory that the whitespace is not really part of the name.) (Even though it has a funny non-alphabetic name, the language doesn't treat it specially. I used to call it `s` but it was distracting.)

There are three cases of the pattern `group <- item group / .`, which means `group` is zero or more things that match `item`. Because PEGs are greedy and don't backtrack after returning, `group` will only ever parse the maximum possible number of `item` items. It's not possible for a parsing failure after the `group` to cause `group` to backtrack and return a smaller number of `item` objects, the way it could in a parser for a context-free grammar, although a parsing failure inside the last `item` will indeed do so. This allows us to get by without a separate scanner for this grammar! One minor variation of this pattern is found in `sentence` and `name`, which match *one* or more of their elements, not *zero* or more.

Note that the above grammar tells us how to parse the language, but doesn't tell us anything about its semantics. But it's nice and short.

Adding Grouping

The PEG language as written above is pretty weak. It doesn't have grouping or repetition, although they can be emulated with the use of extra productions, as in the `foos` pattern explained above.

We can add grouping by redefining `term` like this:

```
# in a slightly more powerful parsing expression grammar:
term           <- '!'_ term / '\'' stringcontents '\''_ / name _
                / '('_ choice ')'_.
```

This simplifies the grammar only slightly; we can rewrite `stringcontents` as follows:

```
stringcontents <- (!'\\' !'\'' char / '\\' char) stringcontents / .
```

A Diversion: Adding Repetition

Although it turns out not to be very useful for what I'll do next, adding the capability for repetition to the language makes it shorter and clearer.

```
# in a more powerful PEG:
sp       <- ' ' / '\n' / '\t'.
_        <- sp*.
sentence <- _ (name _ '<-'_ choice '.'_)+.
choice   <- term* ('/'_ term*)*.
term     <- ('!'_ term / string / name / '('_ choice ')')_ ('+' / '*' / )_.
string   <- '\'' (!'\\' !'\'' char / '\\' char)* '\''_.
meta     <- '!' / '\'' / '<-' / '/' / '.' / '+' / '*' / '(' / ')'.
name     <- (!meta !sp char)+.
```

That shrinks the grammar considerably, while significantly expanding the expressiveness of the grammar language it describes.

Adding Result Expressions

In theory, the grammar as written could be useful. It's expressive enough to describe the tree structure of a language, such as the PEG language defined above. So you could use it to parse some string into a syntax tree.

However, it would be even more useful to have a version of the grammar language that can include result expressions written in some programming language that compute useful things. For example, you could use such a system to write and maintain a working compiler from PEG grammars to some programming language, or from some other language.

A straightforward and readable way to do this is to label some parts of a sequence with names, and then to use those names in a result specification at the end of the sequence.

Here's an extension of the above grammar that allows for such names and result specifications:

```
# in a PEG describing results:
sp             <- ' ' / '\n' / '\t'.
_              <- sp _ / .
sentence        <- _ rule sentence / _ rule.
rule           <- name _ '<-'_ choice '.'_.
choice         <- sequence '/'_ choice / sequence.
sequence       <- term sequence / '->'_ expr / .
expr           <- '('_ exprcontents ')'_.
exprcontents   <- (!'(' !')' char / expr) exprcontents / .
term           <- name _ ':'_ term / '!'_ term / string / name _
                / '('_ choice ')'_.
string         <- '\'' stringcontents '\''_.
stringcontents <- !'\\' !'\'' char stringcontents
                / '\\' char stringcontents / .
meta           <- '!' / '\'' / '<-' / '/' / '.' / '(' / ')' / ':' / '->'.
name           <- namechar name / namechar.
```

```
namechar       <- !meta !sp char.
```

This adds the possibility that a term may be preceded by a colon and a name, and that a sequence may end with a `->` and a parenthesized expression.

This lets you write things like `n: expr` and `expr _ -> (print("got expr"))`. It doesn't place strong requirements on the embedded expression, so it can be in almost any language, but it does require that any parentheses inside of it be balanced. (If that's difficult in a certain case, due to embedded strings, maybe you can incorporate some commented-out parentheses to balance things.)

A Metacircular Compiler-Compiler

So let's suppose that we want to use this result-expression facility to write a compiler for these grammars, producing a parser for the specified grammar in, say, JavaScript. We want to translate each parsing expression in the grammar language into an expression in the target language that parses the sub-language defined by that parsing expression. For example, we want to translate `choice <- sequence '/'_ choice / sequence.` into a recursive JavaScript function that parses expressions containing slash-separated `choice`s. Since it doesn't specify a result expression, it's sort of indeterminate what it should actually do, other than consume characters from the input stream until it finds something `choice` can't parse.

So now we have to figure out what the semantics are of each of the various actions.

I'm going to factor out the code generation parts into separate named blocks so that it's relatively easy to have the parser, say, generate code in some other language, or just an abstract syntax tree.

Whitespace

Whitespace is fairly easy: it is a no-op.

```
# in the metacircular compiler-compiler:
sp <- ' ' / '\n' / '\t'.
_  <- sp _ / .
```

Rules

Let's compile each rule into a JavaScript function that parses the language described by that rule, and the grammar as a whole into the collection of these functions plus whatever support code is needed. (Here I'm going to use double angle-brackets `<<>>` to name chunks of code that aren't given until later.)

```
rule    <- n: name _ '<-'_ body: choice '.'_ ->
               <<code to produce a function>>
           .
```

```
sentence <- _ r: rule g: sentence -> (r + "\n" + g)
          / _ r: rule -> (r + "\n"
                <<support code>>
            ).
```

The code to produce a function in JavaScript is quite straightforward:

```
# in code to produce a function:
(["function parse_", n, "(input, pos) {\n",
    <<function prologue>>
    body,
    <<function epilogue>>
 "}\n"].join(''))
```

So a grammar nonterminal named `term` will be compiled into a function called `parse_term`, whose body will be the value computed by `choice`, bracketed by some startup and cleanup code, and therefore `choice` needs to evaluate to a string of zero or more valid JavaScript statements.

These functions will need to do several things to implement the semantics of a PEG parser:

1. Advance the input position, starting from the input position the caller passed in, and in case of success, communicate the new input position to the caller.
2. Save the input position (and any other state) in order to backtrack when a sequence inside a choice fails, or after testing a negation condition. They may have to save several input positions at once in cases where there is nested alternation.
3. Compute the value given by the result expressions in the grammar and, in case of success, pass it back to the caller, along with the new input position.

In order to avoid global variables, we're passing in the input string (which doesn't change during a parse) and the current position in it as arguments to each parsing function.

To package the value computed along with the new input position, we'll return a JavaScript object with `val` and `pos` properties, like `{val: "foo", pos: 37}`. In case of failure, we'll just return `null`.

From here we'll mostly work bottom-up.

Names

Names are used in two contexts: at the top level of a rule, they define the name of the nonterminal, and in a term, they request a call to that nonterminal. In both cases, we basically just need the contents of the name.

```
# in the metacircular compiler-compiler:
```

```
meta     <- '!' / '\'' / '<-' / '/' / '.' / '(' / ')' / ':' / '->'.
name     <- c: namechar n: name -> (c + n) / namechar.
namechar <- !meta !sp char.
```

In this case, we presume that the value produced by `char` (and thus the value produced by `namechar`) is the character it consumed, and that in the absence of an explicit result expression, the result of the whole rule is that same character. This can be implemented, for example, by having a sequence return by default the value of the last term in it. (I'm not sure that's a good default, because it seems a little error-prone, but I'll try it.)

Nonterminals

A reference to a nonterminal is compiled as a call to its parsing function, passing in the current position.

```
# in the metacircular compiler-compiler:
term <- labeled / nonterminal / string / negation / parenthesized.
nonterminal <- n: name _ ->
                   <<code to parse another nonterminal>>
               .
```

Again, the JS implemntation of a subroutine call is quite simple:

```
# in code to parse another nonterminal:
(['  state = parse_', n, '(input, state.pos);\n'].join(''))
```

This means we need a variable `state` to store this returned value in, and it needs to be initialized with the position passed in by the caller.

```
# in function prologue:
'  var state = { pos: pos };\n',
```

What do we do with `state.val`? It depends on where the nonterminal is found. If it's preceded by a label, we want to store it in a variable under that name for later use, unless it fails. Let's have `term`, just like `choice`, return a string of zero or more valid JavaScript statements.

```
# in the metacircular compiler-compiler:
labeled <- label: name _ ':'_ value: term ->
               <<code to save a value in a variable>>
           .
```

We protect this with a conditional on `state` in case the parse has failed:

```
# in code to save a value in a variable:
([value, '  if (state) var ', label, ' = state.val;\n'].join(''))
```

(Ideally we would undo this saving if the nonterminal is in an alternative that fails and ends up being backtracked; but hopefully the result expressions of later alternatives will simply not use that variable.)

Now, if the nonterminal was the last thing in a parsing function, then we want to return the `state.val` it gave us as our own `state.val`, and additionally we want to return its `state.pos` as our `state.pos`; or, if it failed, it returned `null`, in which case we want to return `null`.

So at the end of the function, we can just return `state`:

```
# in function epilogue:
'  return state;\n',
```

Now we just need to ensure that all of the other expression types (sequence, terminal strings, ordered choice, negation, parenthesized) update `state` in a manner analogous to how calls to nonterminals update `state`.

While we're on the topic of nonterminals, we should probably define the one predefined nonterminal, `char`:

```
# in support code:
+ 'function parse_char(input, pos) {\n'
+ '  if (pos >= input.length) return null;\n'
+ '  return { pos: pos + 1, val: input.charAt(pos) };\n'
+ '}\n'
```

Sequence

Sequences are relatively simple. Given a sequence of two expressions `foo bar`, we first parse `foo` from the current position, and if that succeeded, we parse `bar` from the new position. If it fails, the sequence as a whole fails, and there is no current position.

This is one of the things that is easier to do if you don't try to write your grammar with features like `*`, since it treats sequences of arbitrary numbers of things as nested sequences of two items, the innermost of which is empty.

```
# in the bare grammar:
sequence <- term sequence / '->'_ expr / .
```

The case of an empty sequence doesn't update `state` at all. In the case of a non-empty sequence, we execute `foo`, and if `foo` doesn't set `state` to `null`, we execute `bar`.

```
# in the metacircular compiler-compiler:
sequence <- foo: term  bar: sequence ->
                  <<code to handle a sequence>>
              / result_expression / -> ('').
```

The `result_expression` case is one of the last things explained, so ignore it for now.

This will result in deeply nested if statements without proper indentation in the output when there is a long sequence, but that's probably okay:

```
# in code to handle a sequence:
([foo, '  if (state) {\n', bar, '  }\n'].join(''))
```

Terminal Strings

A "terminal" or literal string like `'->'` either matches some characters in the input or fails to do so. Rather than inserting code into every parsing function to compare parts of the input, making the parsing functions less readable, we'll factor this out into a single "literal" function:

```
# in support code:
+ 'function literal(input, pos, string) {\n'
+ '  if (input.substr(pos, string.length) === string) {\n'
+ '    return { pos: pos + string.length, val: string };\n'
+ '  } else return null;\n'
+ '}\n'
```

So then we just need to emit code to call this function and update `state` appropriately when we encounter a terminal string. As it happens, the translation from string syntax in the PEG language to string syntax in JavaScript is the null transformation. If we were compiling to some other language, such as C, this might pose some difficulty.

```
# in the metacircular compiler-compiler:
string <- '\'' s: stringcontents '\''_ ->
            <<code to match a literal string>>
          .
stringcontents <-   !'\\' !'\'' c: char  s: stringcontents -> (c + s)
                / b: '\\'       c: char  s: stringcontents -> (b + c + s)
                / -> ('').
```

So here's the function call:

```
# in code to match a literal string:
(["  state = literal(input, state.pos, '", s, "');\n"].join(''))
```

As we iterate through the characters or backslash-escapes inside the string, we convert them to strings — either by default, or explicitly by concatenating the backslash to the character that follows it. Then we call `literal` with the current position and it either returns `null` or gives us the new position and the value it matched as our new `state`.

Ordered Choice

Two of the remaining expression types (ordered choice, negation, but not terminal strings and parenthesized) can require backtracking. So we have to save a state and possibly restore that state.

Here's how ordered choice works; negation is fairly similar. In ordered choice, if the first alternative succeeds, we don't try the others; but if it fails, we restore

the previously saved state.

This is complicated somewhat by the fact that we might be inside a parenthesized expression, so there may be a stack of previously saved states, even inside the same function.

So on entry to the function, we create a stack:

```
# in function prologue:
'  var stack = [];\n',
```

The grammar entry treats N-way choices like `labeled / negation / string / nonterminal / parenthesized` as nested 2-way choices like `labeled / (negation / (string / (nonterminal / parenthesized)))`. This is a little bit needlessly inefficient, since we'll be using potentially four stack entries instead of one, but it will do for now.

```
# in the metacircular compiler-compiler:
choice <- a: sequence '/'_  b: choice ->
                  <<code to handle a choice>>
              / sequence.
```

Execution of `b` is conditional on failure of `a`; if `a` succeeds, we simply discard the state we saved before trying it.

```
# in code to handle a choice:
(['  stack.push(state);\n',
  a,
  '  if (!state) {\n',
  '    state = stack.pop();\n',
  b,
  '  } else stack.pop();\n'].join(''))
```

It's only safe to push `state` rather than a copy of `state` because we never mutate the existing `state`; we only make new `state` objects.

Negation

Negation is `!x`:

```
# in the metacircular compiler-compiler:
negation <- '!'_ t: term ->
                <<code to handle negation>>
            .
```

This is implemented by saving the parse state, trying to parse `x`, failing if parsing `x` succeeded, and otherwise proceeding from the saved parse state.

```
# in code to handle negation:
(['  stack.push(state);\n',
  t,
```

```
  '  if (state) {\n',
  '    stack.pop();\n',
  '    state = null;\n',
  '  } else state = stack.pop();\n'].join(''))
```

You can use a double negative like `!!'->'` to write a "zero-width positive lookahead assertion" in Perl lingo. That compiles into this:

```
# in the output of the compiler-compiler:
  stack.push(state);
  stack.push(state);
  state = literal(input, state.pos, '->');
  if (state) {
    stack.pop();
    state = null;
  } else state = stack.pop();
  if (state) {
    stack.pop();
    state = null;
  } else state = stack.pop();
```

The initial `state` is assumed to be non-`null`. So after the call to `literal`, `state` is non-`null` iff the next couple of characters were `->`. Then, after the first `if`, `state` is non-`null` iff the next couple of characters *weren't* `->`. Then, after the second `if`, it is again non-`null` iff the next couple of characters were `->`. And if it's non-`null`, it's the `state` you started with.

So that does the right thing, perhaps a bit verbosely.

Result Expressions

A result expression gives a JavaScript expression to evaluate to get the value that a sequence parses to. Normally, it uses variable bindings produced by labels. The value it returns may become the value of the term (if the sequence is inside parentheses) or the value returned by a whole parsing function.

```
# in the metacircular compiler-compiler:
result_expression <- '->'_ result: expr _ ->
                         <<code to handle result expressions>>
                     .
```

Note the `_` to discard whitespace.

Of course, this is conditional on the parser not being in a failed state:

```
# in code to handle result expressions:
(['  if (state) state.val = ', result, ';\n'].join(''))
```

The expression is delimited by parentheses `()`. The outermost pair of parentheses are kept, which simplifies the grammar and avoids tricky problems of operator

precedence when the result expression is copied into the output program in the `state.val =` context above.

```
# in the metacircular compiler-compiler:
expr         <- '('_ e: exprcontents ')' -> ('(' + e + ')').
exprcontents <- c: (!'(' !')' char / expr)  e: exprcontents -> (c + e)
              / -> ('').
```

`result_expression` discards whitespace after the expression rather than having the expression production do it itself in order to preserve whitespace after right parens consumed by recursive calls to the expression production.

Parenthesized Expressions

Parenthesized expressions don't need any real special handling; or, rather, the special handling consists of the `stack` variable everything uses to backtrack; the parentheses are only there to direct the parser how to parse `/` and `!` and so on.

```
parenthesized <- '('_ body: choice ')'_ -> (body).
```

Exporting

We need one more thing if our grammar is to be loadable as a CommonJS module by systems like node.js:

```
# in support code:
+ "if (typeof exports !== 'undefined')\n"
+ "    exports.parse_sentence = parse_sentence;\n"
```

This assumes that the grammar being processed has a production called `sentence`, which is the only thing that will be exported.

The Whole Metacircular Compiler-Compiler

Here's the whole thing, extracted from this document:

```
# in the output metacircular compiler-compiler:
sp <- ' ' / '\n' / '\t'.
_  <- sp _ / .
rule    <- n: name _ '<-'_ body: choice '.'_ ->
               (["function parse_", n, "(input, pos) {\n",
                   '  var state = { pos: pos };\n',
                   '  var stack = [];\n',
                   body,
                   '  return state;\n',
                "}\n"].join(''))
           .
sentence <- _ r: rule g: sentence -> (r + "\n" + g)
          / _ r: rule -> (r + "\n"
```

```
                + 'function parse_char(input, pos) {\n'
                + '  if (pos >= input.length) return null;\n'
                + '  return { pos: pos + 1, val: input.charAt(pos) };\n'
                + '}\n'
                + 'function literal(input, pos, string) {\n'
                + '  if (input.substr(pos, string.length) === string) {\n'
                + '    return { pos: pos + string.length, val: string };\n'
                + '  } else return null;\n'
                + '}\n'
                + "if (typeof exports !== 'undefined')\n"
                + "    exports.parse_sentence = parse_sentence;\n"
            ).
meta     <- '!' / '\'' / '<-' / '/' / '.' / '(' / ')' / ':' / '->'.
name     <- c: namechar n: name -> (c + n) / namechar.
namechar <- !meta !sp char.
term <- labeled / nonterminal / string / negation / parenthesized.
nonterminal <- n: name _ ->
                   (['  state = parse_', n, '(input, state.pos);\n'].join(''))
               .
labeled <- label: name _ ':'_ value: term ->
              ([value, '  if (state) var ', label, ' = state.val;\n'].join(''))
           .
sequence <- foo: term  bar: sequence ->
                   ([foo, '  if (state) {\n', bar, '  }\n'].join(''))
              / result_expression / -> ('').
string <- '\'' s: stringcontents '\''_ ->
             (["  state = literal(input, state.pos, '", s, "');\n"].join(''))
          .
stringcontents <-   !'\\' !'\'' c: char  s: stringcontents -> (c + s)
                / b: '\\'       c: char  s: stringcontents -> (b + c + s)
                / -> ('').
choice <- a: sequence '/'_  b: choice ->
                  (['  stack.push(state);\n',
                    a,
                    '  if (!state) {\n',
                    '    state = stack.pop();\n',
                    b,
                    '  } else stack.pop();\n'].join(''))
              / sequence.
negation <- '!'_ t: term ->
                (['  stack.push(state);\n',
                  t,
                  '  if (state) {\n',
                  '    stack.pop();\n',
                  '    state = null;\n',
                  '  } else state = stack.pop();\n'].join(''))
```

```
            .
result_expression <- '->'_ result: expr _ ->
                         ([' if (state) state.val = ', result, ';\n'].join(''))
                     .
expr         <- '('_ e: exprcontents ')' -> ('(' + e + ')').
exprcontents <- c: (!'(' !')' char / expr)  e: exprcontents -> (c + e)
              / -> ('').
parenthesized <- '('_ body: choice ')'_ -> (body).
```

That's 66 lines of code, constituting a compiler that can compile itself into JavaScript, if you have a way to execute it.

XXX: a couple of lines are over 80 chars; fix this!

Bootstrapping to JavaScript

But, to actually execute this compiler-compiler, you need a version already running, so you can compile the compiler-compiler to JavaScript.

Hand-compiling: a blind alley

I started by trying to compile it by hand, using YASnippet, but after not very long, I gave up on that approach. Here are the hand-compiled versions of `sp <- ' ' / '\n' / '\t'.` and `_  <- sp _ / .`

```
# in the hand-compiled metacircular compiler-compiler:
function parse_sp(input, pos) {
  var state = { pos: pos };
  var stack = [];
  stack.push(state);
  state = literal(input, state.pos, ' ');
  if (!state) {
    state = stack.pop();
  stack.push(state);
  state = literal(input, state.pos, '\n');
  if (!state) {
    state = stack.pop();
  state = literal(input, state.pos, '\t');
  } else {
    stack.pop();
  }
  } else {
    stack.pop();
  }
  return state;
}
function parse__(input, pos) {
  var state = { pos: pos };
```

```
  var stack = [];
  stack.push(state);
  state = parse_sp(input, state.pos);
  if (state) {
  state = parse__(input, state.pos);
  }
  if (!state) {
    state = stack.pop();
  } else {
    stack.pop();
  }
  return state;
}
```

After thus inflating two lines of grammar into 35 lines of JavaScript, I knew I needed a better way. At that rate, the whole thing would be about 1200 lines. That's too much to debug, even if YASnippet makes it relatively easy to type, unless there's no easier way.

But there is.

A Bunch of Functions

So, instead, I'm writing one function for each interesting recognition rule from the grammar, returning the same result expressions that the parsing function will. Then I can construct a sort of abstract syntax tree of the grammar out of calls to these functions, and it will only be a little larger than the grammar itself.

For example, the first rule `sp <- ' ' / '\n' / '\t'.` will become:

```
# in the ASTs made of function calls:
var sp_rule = rule('sp', choice(string(' '), choice(string('\\n'),
                                                    string('\\t'))));
```

This is a bit of a cheat; the innermost choice really parses as `choice(sequence(string('\\n'), ''), sequence(string('\\t'), ''))` but I'm hoping that doesn't matter for now.

Then at the end I can combine all of the variables into a grammar.

First I need the functions, though.

I'm omitting `sp` (likewise `_`, `meta`) because they don't produce interesting values.

```
# in the bunch-of-functions version:
function rule(n, body) {
  return (["function parse_", n, "(input, pos) {\n",
                  '  var state = { pos: pos };\n',
                  '  var stack = [];\n',
```

```
                body,
                '  return state;\n',
              "}\n"].join(''));
}

function sentence2(r, g) {
  return (r + "\n" + g);
}

function sentence1(r) {
  return (r + "\n"
              <<support code>>
          );
}
```

I'm omitting `name` (likewise `expr`, `inner`, `exprcontents`, `stringcontents`) because it just copies a character string from the input into the output. I can do that myself. And I'm omitting `term` because it just returns one of its children's values.

```
function nonterminal(n) {
  return ['  state = parse_', n, '(input, state.pos);\n'].join('');
}
function labeled(label, value) {
  return [value, '  if (state) var ', label, ' = state.val;\n'].join('');
}
function sequence(foo, bar) {
  return [foo, '  if (state) {\n', bar, '  }\n'].join('');
}
function string(s) {
  return ["  state = literal(input, state.pos, '", s, "');\n"].join('');
}
function choice(a, b) {
  return [
     '  stack.push(state);\n',
     a,
     '  if (!state) {\n',
     '    state = stack.pop();\n',
     b,
     '  } else {\n',
     '    stack.pop();\n', // discard unnecessary saved state
     '  }\n'].join('');
}
function negation(t) {
  return [
     '  stack.push(state);\n',
     t,
```

```
      '  if (state) {\n',
      '    stack.pop();\n',
      '    state = null;\n',
      '  } else {\n',
      '    state = stack.pop();\n',
      '  }\n'].join('');
}
function result_expression(result) {
  return ['  state.val = ', result, ';\n'].join('');
}
```

We'll also need the support code from the `sentence` rule, except for the exporting of `parse_sentence`.

```
function parse_char(input, pos) {
  if (pos >= input.length) return null;
  return { pos: pos + 1, val: input.charAt(pos) };
}
function literal(input, pos, string) {
  if (input.substr(pos, string.length) === string) {
    return { pos: pos + string.length, val: string };
  } else return null;
}
```

Then, after all those functions are defined, we can call them to build up the ASTs.

```
<<the ASTs made of function calls>>
```

The rule for `_` is quite straightforward:

```
# in the ASTs made of function calls:
var __rule = rule('_',
                  choice(sequence(nonterminal('sp'), nonterminal('_')),
                         ''));
```

The rule for `rule` contains a rather long sequence, which will be treated as a deeply nested bunch of two-element sequences. But it's hard to read and write it that way, so I'm going to define a helper function `nseq` to make a sequence of an arbitrary number of sequence elements.

```
function nseq() {
  var rv = arguments[arguments.length-1];
  for (var ii = arguments.length-2; ii >= 0; ii--)
    rv = sequence(arguments[ii], rv);
  return rv;
}
```

This will fail (returning `null`) if we call it with no arguments, so let's be sure not do that. Now we can define the rule for `rule`:

```
var rule_rule = rule('rule',
    nseq(labeled('n', nonterminal('name')), nonterminal('_'),
         string('<-'), nonterminal('_'),
         labeled('body', nonterminal('choice')),
         string('.'), nonterminal('_'),
         result_expression(
            "[\"function parse_\", n, \"(input, pos) {\\n\",\n" +
            "                      '  var state = { pos: pos };\\n',\n" +
            "                      '  var stack = [];\\n',\n" +
            "                     body, \n" +
            "                      '  return state;\\n',\n" +
            "                   \"}\\n\"].join('')")));
```

`rule_rule` is clearly pretty verbose; it's 12 lines, and the corresponding `rule` function is 8 lines, for a total of 20 lines for the "hand-compiled" version of the original 7-line `rule` rule. That's a manageable expansion factor of about 3×.

So, on to `sentence`. I've played fast and loose with leading whitespace here, in order to retain some modicum of readability.

```
var sentence_rule = rule('sentence',
    choice(
        nseq(nonterminal('_'),
             labeled('r', nonterminal('rule')),
             labeled('g', nonterminal('sentence')),
             result_expression('r + "\\n" + g')),
        nseq(nonterminal('_'),
             labeled('r', nonterminal('rule')),
             result_expression('r + "\\n"\n' +
        "+ 'function parse_char(input, pos) {\\n'\n" +
        "+ '  if (pos >= input.length) return null;\\n'\n" +
        "+ '  return { pos: pos + 1, val: input.charAt(pos) };\\n'\n" +
        "+ '}\\n'\n" +
        "+ 'function literal(input, pos, string) {\\n'\n" +
        "+ '  if (input.substr(pos, string.length) === string) {\\n'\n" +
        "+ '    return { pos: pos + string.length, val: string };\\n'\n" +
        "+ '  } else return null;\\n'\n" +
        "+ '}\\n'\n" +
        "+ 'if (typeof exports !== "+'"undefined"'+") {\\n'\n" +
        "+ '    exports.parse_sentence = parse_sentence;\\n'\n" +
        "+ '}\\n'\n"))));
```

The quoting of the support code is kind of confusing; the original is one long string, containing a bunch of `\n` newlines, broken up into lines for readability, joined by the `+` operator. This version is also one long string, containing the lines of the original long string, also broken up into lines for readability, joined by the `+` operator. So there are two levels of quoting. The inner level has the `+` on the left and uses single quotes `''`, and the outer level has the `+` on the right

and uses double quotes "".

The next rule is `meta`, and it has a lot of `choice`s. So we define something like `nseq`, but for `choice`s.

```
function nchoice() {
  var rv = arguments[arguments.length-1];
  for (var ii = arguments.length-2; ii >= 0; ii--)
    rv = choice(arguments[ii], rv);
  return rv;
}

var meta_rule = rule('meta',
  nchoice(string('!'), string('\\\''), string('<-'), string('/'),
          string('.'), string('('),    string(')'),  string(':'),
          string('->')));
```

The next few rules are straightforward translations from the grammar.

```
var name_rule = rule('name',
    choice(nseq(labeled('c', nonterminal('namechar')),
                labeled('n', nonterminal('name')),
                result_expression('c + n')),
           nonterminal('namechar')));
var namechar_rule = rule('namechar',
    nseq(negation(nonterminal('meta')),
         negation(nonterminal('sp')), nonterminal('char')));
var term_rule = rule('term',
    nchoice(nonterminal('labeled'), nonterminal('nonterminal'),
            nonterminal('string'),  nonterminal('negation'),
            nonterminal('parenthesized')));
var nonterminal_rule = rule('nonterminal',
    nseq(labeled('n', nonterminal('name')), nonterminal('_'),
         result_expression("['  state = parse_', n, " +
                           "'(input, state.pos);\\n'].join('')")));
var labeled_rule = rule('labeled',
    nseq(labeled('label', nonterminal('name')), nonterminal('_'),
         string(':'), nonterminal('_'),
         labeled('value', nonterminal('term')),
         result_expression("[value, '  if (state) var ', " +
                           "label, ' = state.val;\\n'].join('')")));
var sequence_rule = rule('sequence',
    nchoice(nseq(labeled('foo', nonterminal('term')),
                 labeled('bar', nonterminal('sequence')),
                 result_expression("[foo, '  if (state) {\\n', " +
                                   "bar, '  }\\n'].join('')")),
            nonterminal('result_expression'),
            sequence(result_expression("''"))));
```

That's 29 lines, transliterating 12 lines from the grammar, and now the transliteration is halfway done.

```
var string_rule = rule('string',
    nseq(string("\\'"), labeled('s', nonterminal('stringcontents')),
         string("\\'"), nonterminal('_'),
         result_expression('["  state = literal(input, state.pos, ' +
                           '\'", s, "\');\\n"].join(\'\')')));
var stringcontents_rule = rule('stringcontents',
    nchoice(nseq(negation(string("\\\\")), negation(string("\\'")),
                 labeled('c', nonterminal('char')),
                 labeled('s', nonterminal('stringcontents')),
                 result_expression('c + s')),
            nseq(labeled('b', string("\\\\")),
                 labeled('c', nonterminal('char')),
                 labeled('s', nonterminal('stringcontents')),
                 result_expression('b + c + s')),
            result_expression("''")));
```

For `choice` I'm omitting not only whitespace but also a comment.

```
var choice_rule = rule('choice',
    choice(nseq(labeled('a', nonterminal('sequence')),
                string('/'), nonterminal('_'),
                labeled('b', nonterminal('choice')),
                result_expression(
                    "['  stack.push(state);\\n',\n" +
                    " a,\n" +
                    " '  if (!state) {\\n',\n" +
                    " '    state = stack.pop();\\n',\n" +
                    " b,\n" +
                    " '  } else {\\n',\n" +
                    " '    stack.pop();\\n',\n" +
                    " '  }\\n'].join('')")),
           nonterminal('sequence')));
var negation_rule = rule('negation',
    nseq(string('!'), nonterminal('_'), labeled('t', nonterminal('term')),
         result_expression(
                    "['  stack.push(state);\\n',\n" +
                    " t,\n" +
                    " '  if (state) {\\n',\n" +
                    " '    stack.pop();\\n',\n" +
                    " '    state = null;\\n',\n" +
                    " '  } else {\\n',\n" +
                    " '    state = stack.pop();\\n',\n" +
                    " '  }\\n'].join('')")));
var result_expression_rule = rule('result_expression',
```

```
    nseq(string('->'), nonterminal('_'),
         labeled('result', nonterminal('expr')),
         result_expression("['  if (state) state.val = ', " +
                           "result, ';\\n'].join('')")));
var expr_rule = rule('expr',
    nseq(string('('), nonterminal('_'),
         labeled('e', nonterminal('exprcontents')),
         string(')'), nonterminal('_'),
         result_expression('e')));
var inner_rule = rule('inner',
    nseq(string('('), nonterminal('_'),
         labeled('e', nonterminal('exprcontents')),
         string(')'),
         result_expression("'(' + e + ')'")));
var exprcontents_rule = rule('exprcontents',
    choice(
        nseq(labeled('c',
                     choice(nseq(negation(string('(')),
                                 negation(string(')')),
                                 nonterminal('char')),
                            nonterminal('inner'))),
             labeled('e', nonterminal('exprcontents')),
             result_expression('c + e')),
        result_expression("''")));
var parenthesized_rule = rule('parenthesized',
    nseq(string('('), nonterminal('_'),
         labeled('body', nonterminal('choice')),
         string(')'), nonterminal('_'),
         result_expression('body')));
```

So that's all the rules. Now we just need to assemble them into a sentence, using a technique similar to `nseq` and `nchoice`.

```
function nsentence() {
  var rv = sentence1(arguments[arguments.length-1]);
  for (var ii = arguments.length-2; ii >= 0; ii--)
    rv = sentence2(arguments[ii], rv);
  return rv;
}

var all_rules = nsentence(sp_rule, __rule, rule_rule, sentence_rule,
                          meta_rule, name_rule, namechar_rule, term_rule,
                          nonterminal_rule, labeled_rule, sequence_rule,
                          string_rule, stringcontents_rule, choice_rule,
                          negation_rule, result_expression_rule, expr_rule,
                          inner_rule, exprcontents_rule, parenthesized_rule);
```

Now the variable `all_rules` has a working parser in it in JavaScript.

To get a usable `parse_sentence` function, we need to `eval` that script:

```
eval(all_rules);
```

And then we can export the function:

```
if (typeof exports !== 'undefined') exports.parse_sentence = parse_sentence;
```

The Output Parser in JavaScript

I used to include here the contents of in `all_rules` after a couple of iterations. It's ten pages long (660 lines), and the compile takes about 3–5 seconds on my machine, although it's under 100ms on modern computers. However, I decided that it was too much to want to include it here; this document is for reading. If you `git clone` it, it's in `output.js`.

Cross-Compiling to Lua

It was a lot of trouble getting the short compiler-compiler above to an actually runnable state; I had to write and debug, basically, two copies of the same code. It would have been much easier if I'd already happened to have such a compiler-compiler around that I could use to compile my grammar with.

Well, for the program I'm using to extract the code from this document, which I call HandAxeWeb, I would like to have such a compiler-compiler to generate code in Lua.

So I'm going to define a "version 2" of the compiler-compiler which, instead of generating JS code, generates Lua code. (It is still written in JS, though.)

First, instead of producing JS functions for rules, we produce Lua functions for rules:

```
# in code to produce a function v2:
(['function parse_',n,'(input, pos)\n',
  <<function prologue>>
  body,
  <<function epilogue>>
  'end\n'].join(''))
```

Invoking nonterminals needs no change; JS and Lua syntax overlap here. But local variable declaration and finite maps look different:

```
# in function prologue v2:
'  local state = { pos = pos }\n',
```

We have to declare variables outside their conditional; Lua's scoping rules here change the semantics somewhat because unless you declare the variables at the top of the function you can't write a rule like `x <- (bar y: foo / baz y:`

quux) -> (y) and have it work because the inner y variables are declared in an inner block in Lua, while in JS they automatically belong to the whole function.

```
# in code to save a value in a variable v2:
([value,
  '  local ',label,'\n',
  '  if state then ',label,' = state.val end\n'].join(''))
```

The parse_char and literal functions are a bit different; remember, Lua numbers character positions in strings from 1, and the second argument to its string.sub is not a length but an ending index:

```
# in support code v2:
+ 'function parse_char(input, pos)\n'
+ '  if pos > #input then return nil end\n'
+ '  return { pos = pos + 1, \n'
+ '           val = string.sub(input, pos, pos) }\n'
+ 'end\n'
+ 'function literal(input, pos, needle)\n'
+ '  if string.sub(input, pos, pos + #needle - 1)\n'
+ '     === needle then\n'
+ '     return { pos = pos + #needle, val = needle }\n'
+ '  else return nil end\n'
+ 'end\n'
```

The code to invoke literal doesn't actually need to change.

Sequence-handling differs only in minor bits of syntax:

```
# in code to handle a sequence v2:
([foo, '  if state then\n', bar, '  end\n'].join(''))
```

Initializing the stack is a little different:

```
# in function prologue v2:
'  local stack = {}\n',
```

Ordered choice looks quite similar to JS:

```
# in code to handle a choice v2:
(['  table.insert(stack, state)\n',
  a,
  '  if not state then\n',
  '    state = table.remove(stack)\n',
  b,
  '  else\n',
  '    table.remove(stack)\n',
  '  end\n'].join(''))
```

Negation too:

```
# in code to handle negation v2:
```

```
([' table.insert(stack, state)\n',
  t,
  '  if state then\n',
  '    table.remove(stack)\n',
  '    state = nil\n',
  '  else\n',
  '    state = table.remove(stack)\n',
  '  end\n'].join(''))
```

Result expressions too:

```
# in code to handle result expressions v2:
(['  if state then state.val = ',result,' end\n'].join(''))
```

And that is sufficient to be able to generate compilers in Lua from grammars whose result expressions are in Lua. Unfortunately, it's still not good enough to generate a metacircular compiler-compiler in Lua from the grammar given here, because that grammar is written in JS, even though it generates Lua code.

It would be relatively straightforward to make the modification needed to the grammar quite minor: all the result expressions merely concatenate a bunch of strings, and if they did so by calling a function, you'd only need to redefine that function in the two target languages; in JS, something like `Array.prototype.slice.apply(arguments).join('')` and in Lua, something like `table.concat({...})`.

But this is sort of unnecessary. Really, we just need to be able to compile our parsers using node.js.

TODO

- memoization
- performance measurement: it takes minimally 252ms to compile itself on my netbook, wallclock, under whatever version of Node I'm using. That's pretty pessimal; it's about 11 or 12 kilobytes per second, close to a hundred thousand clock cycles per byte. Follow sets may offer a way to improve that by probably an order of magnitude.
- re-add repetition `+` and `*` (in a later version)
- factor out loopbody? like,
  loopbody <- term: body -> (loop body code).
  zero_or_more <- loopbody: body -> (body). one_or_more <- loopbody: body -> (body + 'if …').
- how about removing `()` grouping? It leaves "a PEG describing results" (and "the metacircular compiler-compiler") one line shorter and one line longer, but perhaps it could simplify backtracking by eliminating the explicit stack? Because then each parsing function would only need to contain one level of backtracking for `/` and one for `!` — oh, well, hmm, `!` might be tricky if we want to support positive lookahead too. Probably

better to leave the stack in.
- Rewrite the Lua handaxeweb to use a PEG parser.
- maybe: rewrite the Lua handaxeweb to be written in JS with Node? The whole Lua story ("in a later version, this program switched to generating Lua grammars and lost the ability to compile itself") kind of stinks. And writing 39 to 48 lines of code to "port" a 66-line program also seems kind of silly, like it may not justify the abstraction overhead that permits it.
- maybe: reorganize this document, putting bootstrap.js first? Not sure.
- maybe: write a Markdown parser?
- move Makefile and pegcompile.js into this document?

Profiling results

XXX these are rough notes that should be cleaned up

I profiled this thing compiling itself in Arora.

It contains 2939 characters, but makes 32370 calls to `literal`, which is about 25% of its CPU time, I think (the profile output is a little hard to interpret; some of the numbers are over 100%, probably due to recursion, and 85% is attributed merely to "program"). `parse_meta` takes more than a third of the CPU time, largely by virtue of calling `literal` several times. It also makes 41903 calls each to `push` and `pop`.

That means it's testing about 11 literals per character, and backtracking 14 times. I could be wrong but I don't think much of this would be improved by memoizing; computing follow sets is likely to make a bigger difference by avoiding the majority of that backtracking.

`parse_char` is called 4219 times, mostly from `parse_exprcontents`.

Building up the output tree with `string.join` takes only about 0.6% of its time.

I suspect that current WebKit has a much better profiler.

Firebug agrees on most things (23.87% in `literal`), but it has the interesting result that actually 17% of the time is in `compile`, which was called only once and does little more than call `eval`. So apparently the time to generate the output JS was only about 4x the time needed for SpiderMonkey to compile it!

Other Interesting PEGs

Here's some nifty stuff you can do with the one-page parser generator described above.

CSV files

Ierusalemschy gives this grammar for parsing Excel-style CSV files:

```
# in the LPEG notation with captures:
```

```
record     <- (<field> (',' <field>)*)->{} (%nl / !.)
field      <- <escaped> / <nonescaped>
nonescaped <- { [^,"%nl]* }
escaped    <- '"' {~ ([^"] / '""'->'"')* ~} '"'
```

The `{}` capture pieces of text and `{~ ~}` capture and replace them. `*` is for repetition, `%nl` is `'\n'`, `""` are equivalent to `''`, `.` is our `char`, `[abc]` is a character class equivalent to `( 'a' / 'b' / 'c' )`, and `->{}` means "make a list of the results". In the notation I've used for PEGs here, without repetition features, this looks like this:

```
# in csv.peg:
sentence           <- d: (f: field ',' r: sentence -> ([f].concat(r))
                    / f: field               -> ([f])) ('\n' / !char)
                   -> (d).
field              <- escaped / nonescaped.
normal_char        <- !',' !'"' !'\n' char.
nonescaped         <- c: normal_char s: nonescaped -> (c + s) / normal_char.
escaped_inner_char <- !'"' char / '""' -> ('"').
escaped_inner      <- c: escaped_inner_char s: escaped_inner -> (c + s)
                    / escaped_inner_char.
escaped            <- '"' s: escaped_inner '"' -> (s).
```

That's 2½ times as big, which is unreasonable. If we have `*` repetition that makes JavaScript Arrays, we can write it with only a bit more ugliness than in LPEG:

```
# in csvstar.peg:
sentence   <- h: field t: (',' field)* ('\n' / !char) -> ([h].concat(t)).
field      <- escaped / nonescaped.
nonescaped <- s: (!',' !'"' !'\n' char)* -> (s.join('')).
escaped    <- '"' s: (!'"' char / '""' -> ('"'))* '"' -> (s.join('')).
```

ichbins

[Darius Bacon's ichbins] ichbins is an inspiring small Lisp compiler; it can compile itself to C with full run-time type-checking, even though it's only a bit over six pages of code. Its recursive-descent parser is a model of clarity, as recursive-descent parsers go:

```
# in the parser in ichbins.scm:
(define (read)
  (read-dispatch (skip-blanks (read-char))))

(define (skip-blanks c)
  (cond ((memq? c whitespace-chars) (skip-blanks (read-char)))
        ('t c)))
```

```
(define whitespace-chars (cons linefeed "   "))
(define non-symbol-chars "\"\\(')")

(define eof-object '("eof"))

(define (read-dispatch c)
  (cond ((eq? c 'f) eof-object)
        ((eq? c \\) (read-char-literal (read-char)))
        ((eq? c \") (read-string (read-char)))
        ((eq? c \() (read-list))
        ((eq? c \') (cons 'quote (cons (read) '())))
        ((eq? c \)) (error "Unbalanced parentheses"))
        ('t (intern (cons c (read-symbol (peek-char)))))))

(define (read-char-literal c)
  (cond ((eq? c 'f) (error "EOF in character literal"))
        ('t c)))

(define (read-string c)
  (cond ((eq? c 'f) (error "Unterminated string literal"))
        ((eq? c \") '())
        ((eq? c \\) (cons (read-char) (read-string (read-char))))
        ('t (cons c (read-string (read-char))))))

(define (read-symbol c)
  (cond ((memq? c whitespace-chars) '())
        ((memq? c non-symbol-chars) '())
        ('t (read-char) (cons c (read-symbol (peek-char))))))

(define (read-list)
  (read-list-dispatch (skip-blanks (read-char))))

(define (read-list-dispatch c)
  (cond ((eq? c 'f) (error "Unterminated list"))
        ((eq? c \)) '())
        ('t (cons (read-dispatch c) (read-list)))))
```

But with a language suited for parsing, we can do better. Here's a PEG simply describing the same grammar as the above:

```
# in ichbins.peg:
whitespace <- '\n' / ' ' / '\t'.
_          <- whitespace _ / .
non-symbol <- '"' / '\\' / '(' / '\'' / ')'.
sentence   <- _ sexp.
sexp       <- '\\' char / '"' string / '(' list / '\'' read / symbol.
string     <- '"' / (!'\\' char / '\\' char) string.
```

```
symbol     <- !whitespace !non-symbol char / .
list       <- ')' / read list.
```

Instead of 33 lines of code, we have 8. Note that I've followed the kind of weird structure of the original parser: the closing parenthesis is considered part of the list contents, and the closing quote is considered part of the string contents. This simplifies the grammar slightly, and eliminates nearly all non-tail calls (except inside of `list`, and to `_`, and in distinguishing character categories) but I think it makes it a little less clear.

In 16 lines, we can get a real parser that returns a parse of the code, in this case as a JSON string:

```
# in ichbins-parser.peg:
sentence   <- _ s: sexp       -> (JSON.stringify(s, null, 4)).

sexp       <- '('_ list
            / '"' string
            / s: symbol       -> ({symbol: s})
            / '\''_ s: sexp   -> ([{symbol: 'quote'}, s])
            / '\\' c: char _  -> ({char: c}).

list       <- ')'_            -> ([])
            / a: sexp b: list -> ([a].concat(b)).
string     <- '"'_            -> ('')
            / a: (!'\\' char / '\\' b: char -> ('\\' + b))
              t: string       -> (a + t).
symbol     <- a: symchar b: symtail -> (a + b).
symtail    <- symbol / _               -> ('').

_          <- whitespace _ / .
whitespace <- '\n' / ' ' / '\t'.
symchar    <- !( whitespace /'"' / '\\' / '(' / '\'' / ')' ) char.
```

Thanks

Thanks to D. Val Schorre for inventing META-II, of which this is a refinement, in 1964 or a bit before; to Bob M. McClure for inventing TMG, the TransMoGrifier, also in 1964, and to Doug McIlroy for maintaining it afterwards, which not only carried META-II forward, but also helped Thompson write B which became C; to Romuald Ireneus 'Scibor-Marchocki, who apparently ported TMG to TMGL; to Bryan Ford for resurrecting TMG's parsing schema and enhancing it into the form of parsing expression grammars, in 2002; to Alan Kay for bringing META-II back to public attention; to Alessandro Warth and Yoshiki Ohshima for developing OMeta and showing that PEGs can be extended to a wide variety of non-parsing tasks.

To [Aristotle Pagaltzis] (http://plasmasturm.org/) for innumerable improvements

to the readability and correctness of this document.

To Andy Isaacson, Allan Schiffman, [Chris Hibbert] (http://pancrit.org/) for further suggestions for the readability and content of this document.