PhilippeSigaud / **Pegged**

PEG Basics

Edit New Page

Victor Widell edited this page on Mar 7, 2016 · 5 revisions

PEG Basics

Parsing Expressions Grammars (PEG) are a recent addition to the grammar and parsing world. They were proposed by Bryan Ford in 2004 (see Wikipedia).

Mostly, they describe a way to *read* inputs and destructure them into rules instead of explaining how to produce strings, as other grammars do. Bryan Ford created them with programming languages in mind and, as such, they are well suited to describe computer/machine grammars (DSL, JSON, the D grammar, even the PEG grammar itself).

Terminals:

Using the PEG formalism, the terminals (parsing expressions that do not depend on other expressions) are:

- `'xxxx'` or `"xxxx"`, that matches literals. For example, `'('` matches one opening parenthesis (no more, no less, no other character). `"abc"` matches the string "abc", `" "` matches one space, and so on. If the input characters are not those waited for, the expression fails and does not consume any input. You can use `'xxxx'` or `"xxxx"` interchangeably for any character or string, though it's more common to use ' with lone characters and " for strings.
- `eps` (short for 'epsilon') is an expression matching 'the empty char': it always succeeds and does not consume anything. You can also use the empty literals: `''` or `""`.
- `.` (dot) is an expression matching any character. It does *not* match the empty string.
- As a notation shortcut, regex-like char ranges are there : `[abc]` will match 'a' or 'b' or 'c'. If offered "abc" as input, it will match the first 'a' and then stop. `[a-z]` is a range of chars and will match any char in 'a', 'b', 'c', ..., 'z'. `[0-9]` will match any decimal digit. You can combine ranges in one expression, like this: `[a-zA-Z0-9_]` : any lowercase or uppercase English letter or any digit or an underscore.

Note that parsing expressions begin their parsing at the first character in the input. They do not 'skip' chars to find a match, as regular expressions do.

Non-Terminals:

Given the expressions `a`, `b`, `c`, ..., you can combine them using operators:

- `a b` (with spaces between them) is a 'sequence' expression, matching `a` (consuming input characters in the process) and then `b`. It succeeds if `a` and `b` succeed, and fails if any of them fails. A major difference between PEG and other grammar formalisms (like regexen) is that there is no backtracking in PEGs. If `b` fails, the entire expression fails, even if `a` could offer different matches. That means, for example, that `"abc" "def"` is equivalent to `"abcdef"`.

Of course, sequences can have as many elements as you wish: `a b c b a a` is a perfectly valid sequence.

- `a / b` is an 'ordered choice' expression: it tries to match `a`. If `a` matches, it succeeds and consumes the elements needed by `a`. If `a` fails, it tries `b`. In any case (success or failure), it returns `b`'s parsing result. It's 'ordered' in that it always tests sub-expressions in the order they were given. So `a / b / c` will *always* test first for `a`, then `b` and lastly `c` if the previous rules failed. That also means that in `a / b / a`, the second `a` is useless: it will be tested with exactly the same input as the first and will fail as well. This ordering makes PEGs non-ambiguous. A PEG *never* gives rise to an ambiguous grammar, unlike some other formalisms. That's why the '/' character was chosen to denote choice, instead of the more usual '|' (pipe) char. (Pipe is not used by PEGs).

This means that `'<' / '<='` will never match the second choice: if the input does not begin with '<', the choice will fail. And if it begins with '<', the second, longer, member will not be tested since the first choice already matched. As a consequence, choices should be ordered from the longer ones to the shorter ones if you want them to all have a chance to match: `"<<<=" / "<<<" / "<<" / ">>" / "<=" / ">=" / "<" / ">" / "="`, for example.

Choice has a lower priority than sequence, so `a b / c` is `(a b) / c`. As is commonly the case, parenthesis (`( )`) can be used to group expressions together. So if you meant '`a`, and then `b` or `c`', use `a (b / c)`.

- PEGs use the three regex-standard postfix operators `?` (option, aka zero-or-one), `*` (aka Kleene star, zero-or-more) and `+` (one-or-more). They act on the preceding expression: `a?` will match 0 or 1 `a`, whatever the lone `a` expression matches. `"abc"*` matches zero-or-more "abc" (note the string literal here), and `(a b)+` will match one or more `(a b)` sequence. In PEGs these operators are greedy: they consume as many input characters as they can. This means that `a* a` will always fail: as long as there is something an `a` can consume, the `*` operator will make it so. Once there is no input, the `a*` transmits the input to the second sequence member, which, being also an `a`, will fail.

- PEGs also use predicates, expressions that succeed or fail but do not consume any input. `&` is the positive lookahead predicate: `&a` succeeds if `a` succeeds and fails if `a` fails. In both cases, `&a` does not consume any input. `!a` is the negative lookahead predicate: it succeeds when `a` fails and inversely fails if `a` succeeds, never consuming any input. So `!'_' .` is a parsing expression that matches any char but the underscore (`'_'`). Let's explain this: if the first char is an underscore, `!'_'` fails and the entire `!'_' .` sequence fails, not consuming any input. If the first char is not an underscore, `!'_'` succeeds and the sequence tests its second member, namely `.` that matches any char, thus succeeding globally.

Defining Rules

Rules are named parsing expressions, that can be used in other parsing expression, thereby allowing direct or indirect recursion. Rules are defined by a name (any C-like identifier will do), a leftward arrow, written like this: `<-` and a parsing expression.

So: `as <- 'a'*` is a rule named `as` that matches zero or more 'a' chars. `abSequence <- ('a' / 'b')*` is a rule matching any sequence of zero or more chars, with 'a' or 'b' as elements. It will parse 'aaaa', but also 'ababbaab' and so on.

Here is a rule matching C-like identifiers: `Identifier <- [a-zA-Z_] [a-zA-Z0-9_]*`

And here is a rule matching end-of-input: `endOfInput <- !.`. Indeed, it fails if there is any char left in the input, and succeeds otherwise).

Rules can refer to other rules:

```
lowerCase  <- [a-z]
upperCase  <- [A-Z]
```

```
digit      <- [0-9]
underscore <- '_'
identifier <- (lowerCase / upperCase / underscore) (lowerCase / upperCase /
underscore / digit)*
```

This defines five rules, the last one using the first four rules in its definition.

Rule can call themselves, recursively:

```
Parens <- '(' Parens ')' / identifier
```

is a rule matching any identifier, optionally enclosed inside matching parenthesis. It will match "abc", but also "(abc)", "((abc))", but not "((abc)". Given "(abc))" it matches "(abc)", letting the last ')' unconsumed in the input. This is one place where PEGs shine compared to regular expressions which cannot parse recursive inputs like this.

By the way, if you want to force a parsing expression to match the entire input, just put the previously seen `endOfInput` at the end of its definition. `Parens <- ('(' Parens ')' / Identifier) endOfInput` will fail on "(abc))".

And yes, for those of you knowing grammar formalism, that means PEGs are prey to infinite recursion for left-recursive rules: `A <- A 'a'` really means 'to parse an `A`, first parse an `A`, and then...', which is nonsensical as far as PEGs are concerned.

An Example Grammar

Here is a grammar for standard arithmetic expressions:

```
Arithmetic:
    Expr     <- Factor AddExpr*
    AddExpr  <- ('+'/'-') Factor
    Factor   <- Primary MulExpr*
    MulExpr  <- ('*'/'/') Primary
    Primary  <- '(' Expr ')'
              / Number
              / Variable
              / '-' Primary

    Number   <- [0-9]+
    Variable <- identifier
```

This is the most complicated grammar we've seen so far: the first five rules are mutually recursive non-terminals: they call one another, as deeply as necessary to parse the input. `Number` and `Variable` are the only terminals in this grammar, representing ... numbers or variables.

Operator precedence is there implicitly by the fact that an expression is seen as constituted by '+'- or '-'-separated factors, themselves containing '*' or '/'. This way, "2 * 3 + 1" is correctly parsed as "(2 * 3)" (a factor) and "1" (another factor, somewhat extreme), separated by a '+'.

Next

That's all for PEGs themselves. Let's enter **Pegged** now. Go to Declaring a Grammar for the next lesson.

Pegged Tutorial

+ Add a custom footer