PEGParser Documentation

Release 0.0.0

Abraham Schneider

July 04, 2014

Contents

PEGParser is a PEG parser for Julia with packrat capabilties. It borrows largely from Pyparsing, Parsimonous, as well as boost::spirit.

Contents:

CHAPTER 1

Indices and tables

- *genindex*
- *modindex*
- *search*

1.1 Defining a grammar

Grammars can be created by:

```
@grammar <name>
    start = ...
    rule1 = ...
    rule2 = ...
end
```

The following rules are supported:

- Terminals: Strings and characters
- Or: `a | b`
- And: `a + b`
- Grouping: `(a + b + c) | (d + e)`
- Optional: `?a`
- One or more: `+(a)`
- Zero or more: `*(a)`
- Regular expressions: `r"[a-zA-Z]"`
- Lists: `list(a, delim)`

1.2 Using the grammar

Once a grammar is defined, you can parse a string input with:

```
(ast, pos, error) = parse(grammar, input)
```

The variable `ast` contains the abstract syntax tree (AST) from the parse, `pos` contains the last position in the parse, and `error` contains any error that results from the parse.

1.3 Transforming the AST

Once the AST is obtained the result can be transformed into a final form. The function `transform` takes the AST and a function to apply to each node. Additionally, an optional list of nodes can be supplied:

```
transform(fn, ast, ignore=[:sym1, :sym2, ..., :symN])
```

Transform relies on multi-dispatch to choose the correct function to call based on symbol type:

```
transformFn(node, cvalues, ::MatchRule{:sym}) = ...
```

where `node` is the current node, `cvalues` contains the values of the child nodes, and `::MatchRule{:sym}` causes the function to be called if the function is of the type `:sym`.

1.4 Example 1

Suppose you want a parser that takes the input `[text]` and converts it to `<b>text</b>`:

```
@grammar markup begin
    start = bold_text
    bold_text = bold_open + text +  bold_close

    text = r"[a-zA-Z]"
    bold_open = '['
    bold_close = ']'
end

tohtml(node, cvalues, ::MatchRule{:bold_open}) = "<b>"
tohtml(node, cvalues, ::MatchRule{:bold_close}) = "</b>"
tohtml(node, cvalues, ::MatchRule{:text}) = node.value
tohtml(node, cvalues, ::MatchRule{:bold_text}) = join(cvalues)

(ast, pos, error) = parse(markup, "[test]")
result = transform(tohtml, ast)
println(result) # "<b>test</b>"
```

1.5 Example 2

Another example is a simple calculator:

```
@grammar calc begin
  start = expr
  number = r"([0-9]+)"
  expr = (term + op1 + expr) | term
  term = (factor + op2 + term) | factor
  factor = number | pfactor
  pfactor = lparen + expr + rparen
  op1 = '+' | '-'
  op2 = '*' | '/'
  lparen = "("
```

```
  rparen = ")"
end

# A ::MatchRule{:default} can be specified and will be used for anything that isn't
# explicitely defined and is not on the ignore list
evaluate(node, cvalues, ::MatchRule{:number}) = float(node.value)
evaluate(node, cvalues, ::MatchRule{:expr}) =
    length(children) == 1 ? children : eval(Expr(:call, cvalues[2], cvalues[1], cvalues[3]))
evaluate(node, cvalues, ::MatchRule{:factor}) = cvalues
evaluate(node, cvalues, ::MatchRule{:pfactor}) = cvalues
evaluate(node, cvalues, ::MatchRule{:term}) =
    length(children) == 1 ? children : eval(Expr(:call, cvalues[2], cvalues[1], cvalues[3]))
evaluate(node, cvalues, ::MatchRule{:op1}) = symbol(node.value)
evaluate(node, cvalues, ::MatchRule{:op2}) = symbol(node.value)

(node, pos, error) = parse(grammar, "5*(42+3+6+10+2)")

# Note: the ignore list -- these will produce no output when encountered.
result = transform(math, node, ignore=[:lparen, :rparen])

println(result) # 315.0
```