rparsec overview

Though rparsec API is quite easy to use, it still helps if one has an overview of how rparsec works and be wary of some gotchas. In this article I'll try to answer questions that might be fequently asked.

How do I start?

With rparsec, one creates Parser object in terms of the production rule of the grammar. Once a Parser object is created, it can be used as in:

```
parse_result = parser.parse "your code goes here"
```

Depending on your need, this parse_result can be either the calculation result (as in the calculator example), or an abstract syntax tree (as in the sql parser example).

So how do you create Parser object? Check out Parsers module. It has everything for you to start with.

You may very possibly want to include Parsers module into your parser module/class. (You can of course not include Parsers module and just use individual methods as in "Parsers.char(?a)", "Parsers.token(:word)" etc. It is just way more convenient if the repetition of "Parsers" can be avoided.)

How does rparsec work?

A Parser object accepts any input that wacks and quacks like a string or an array, with [] and *length* methods.

Each Parser object encapsulates a piece of parsing logic, that when executed, will read the current inputs and optionally adjusts the cursor.

Parsers#satisfies uses the associated predicate to check if the current input satisfies the requirement. And if it does, advances the cursor for one step.

Parsers#char, Parsers#among, Parsers#range are all derived from *satisfies*.

Parsers#string, Parsers#string_nocase matches the given string literal, case sensitively or insensitively.

Parsers#regexp uses a regular expression pattern to match the current input.

All parsers mentioned above are character level parsers. They check the current string input and are typically used in the lexing phase (if you choose to do lexing/parsing as two phases. It is not required to have two phases though.)

It is relatively more straight-forward to use the 1-phase approach, where all parsers work on the string level (as in the calculator example). However, when the grammar rule scales up and when performance is a concern, 2-phase approach should be used to speed things up and better isolate lexical/syntactical rules.

And suppose you want to do 2-phase parsing, lexcial analysis should result in tokens. The Parser#token method is responsible for converting the current string result to a token identified by the symbol specified. This token will then be recognized during the second phase (syntactical analysis) by the Parsers#token method.

Syntactic parsers (created by Parsers#token, Keywords#[] or Operators#[]) accept as input an array of tokens. This array of tokens are typically generated by some lexical parsers.

The api to chain lexical parser and syntactical parser together is Parser#nested where syntactical parser is nested within the lexical parser (that results in a token array).

For the simplest case,

```
word.token(:word).lexeme
```

will create a lexer that results in an array of word tokens. And

```
word.token(:word).lexeme.nested(syntax_parser)
```

chains a syntactical parser as nested within the lexer so that the lexer result is passed as input to the syntactic parser.

What are the step 1-2-3 in creating a typical parser?

Step 1: Lexer

A typical language contains keyword, operators and number literal, string literal and regular words.

Parsers#integer and Parsers#number can be used to scan integer/number; string literal is better handled by Parsers#regexp (use Parser#map to convert the matched string literal and to deal with quotes/escapes.)

Keyword is easily handled by Keywords helper class. Typically regular words can also be handled by this class.

Operator is handled by Operators class. Pass it all the operators you want to support, and it will take care of ambiguities such that "==" is parsed as "==", not two "=" operators.

Once all lexers are ready, combine them together as in:

```
operators = Operators.new(%w{+ - * / ++ --})
keywords = Keywords.case_sensitive(%w{if then else end})
num = number.token(:number)
lexer = operators.lexer | keywords.lexer | num
```

Use the *lexeme* method to skip whitespaces and comments and collect the tokens as an array:

```
lexeme = lexer.lexeme << eof
```

Note, we use "<< eof" to assert that after all tokens are read, eof directly follows (meaning, all inputs are consumed. "<<" operator reads as "followed by").

Step 2: atomic syntactic parsers

To start syntactic parser that accepts the token array created by the lexer, we need to first recognize the tokens we created.

Number and string literal will be recognized by Parsers#token(:number) and Parsers#token(:stringlit) (assuming :stringlit was used as the token identifier for string literal).

To recognize the "++" operator, we use

```
operators[:"++"]
```

To recognize "if" keyword, we use

```
keywords[:if]
```

Step 3: production rule

The last step is to build the parser following production rules. The alternative operator "|" and sequential operator ">>", "<<" are used most frequently to model production rules.

Parser#many combinator models the "kleen star" construct in BNF.

Parsers#map and Parsers#sequence are the two most used combinators to associate semantic actions to syntactic rules. For example:

```
p_a = parser_a.map {|a|create_a_expression a}
p_bcd = sequence(parser_b, parser_c, parser_d) do |b,c,d|
  create_bcd_expression b, c, d
end
```

When production rule involves recursion (as in almost any expression parser), the Parsers#lazy can be used to get around. The usage is as simple as:

```
expr = ... lazy{expr} ...
```

As in most recursive descent parsers, left-recursion is a nightmare. Beware not to write a rparsec parser like this:

```
expr = sequence(lazy{expr}, operators[:"+"], num)
```

It will fail with stack overflow!

A less obvious left-recursion is a production rule that looks like:

```
expr = sequence(operator[:"-"].many, lazy{expr})
```

As *many* can occur 0 times, we have a potential left recursion here.

I regret that I don't know enough to attack this problem. But rparsec does provide a nice work-around. Most of the time left recursion stems from left associative binary operator. And rparsec has operator precedence grammar support that can build the production rule for you. Just declare the operators and their precedence, associativity in an OperatorTable and let Expressions class does the work. (refer to the calculator example).

Operator table is probably more useful than you might expect it to be. It can even be used for senario not as obvious as the arithmetic or logical operators. Indeed, any operation that takes one or two participants and result in a new one may take advantage of operator table. The sql "union" and "union all" syntax, for example, can be modeled as a binary operator too.

Tips

Please see rparsec tips for tips and gotchas.

Created by benyu benyu
On Mon Oct 23 21:42:22 CDT 2006
Using TimTam