コンパイラ理論 6
Racc の例

櫻井彰人

```
# $Id: calc.y,v 1.4 2005/11/20 13:29:32 aamine Exp $
#
# Very simple calculater.
class Calcp
  prechigh
    nonassoc UMINUS
    left       '*' '/'
    left       '+' '-'
  preclow
rule
  target: exp
        | /* none */ { result = 0 }

  exp: exp '+' exp { result += val[2] }
     |   exp '-' exp  { result -= val[2] }
     |   exp '*' exp { result *= val[2] }
     |   exp '/' exp  { result /= val[2] }
     |   '(' exp ')'    { result = val[1] }
     |   '-' NUMBER  =UMINUS { result = -val[1] }
     |   NUMBER
end
```

同位はエラー
演算子順位の定義
同位は左結合
開始記号（非終端記号）
この規則適用後に返す値
初期値は val[0] である（下述）
右辺2番目（0番目が先頭）に帰って来た値
このマイナスは UMINUSと同じ強さ

```
---- header
# $Id: calc.y,v 1.4 2005/11/20 13:29:32 aamine Exp $
---- inner
  def parse(str)
    @q = []
    until str.empty?
      case str
      when /¥A¥s+/
      when /¥A¥d+/
        @q.push [:NUMBER, $&.to_i]
      when /¥A.|¥n/o
        s = $&
        @q.push [s, s]
      end
      str = $'
    end
    @q.push [false, '$end']
    do_parse
  end

  def next_token
    @q.shift
  end
end
```

今回はなし（単なるコメント）
標準的作法
自分で呼ぶ関数なので自由に定義可能
インスタンス変数
行頭からの空白文字列
行頭からの数字列
約束事．パーズの開始
これが呼ばれる

^	行頭
$	行末
¥A	文字列先頭。
¥Z	文字列末尾。

¥d	は [0-9]
¥D	は [^0-9]
¥s	は [¥t¥r¥n¥f]
¥S	は [^ ¥t¥r¥n¥f]
¥w	は [A-Za-z0-9_]
¥W	は [^A-Za-z0-9_]

$&: 現在のスコープで最後に成功した正規表現のパターンマッチでマッチした文字列(Ruby 組み込み変数)
:で始まる識別子等: Rubyのシンボル
http://doc.ruby-lang.org/ja/1.9.3/class/Symbol.html
http://doc.ruby-lang.org/ja/2.1.0/class/Symbol.html
o: 一番最初に正規表現の評価が行われた時に一度だけ式展開(#{})を行う
.: 改行を除く任意の 1 文字にマッチ
$': 現在のスコープで最後に成功した正規表現のパターンマッチでマッチした部分より後ろの文字列

```
---- footer

parser = Calcp.new
puts
puts 'type "Q" to quit.'
puts
while true
  puts
  print '? '
  str = gets.chop!
  break if /q/i =~ str
  begin
    puts "= #{parser.parse(str)}"
  rescue ParseError
    puts $!
  end
end
```

これが実行されると思えばよい
定義は inner に書く
自分で定義したクラス
i: 正規表現はマッチ時に大文字小文字の区別を行わない
自分で定義した parse(…) を呼び出す
最近の例外に関する情報を表す Exception クラスのサブクラスのインスタンスです。raise によって設定され

演算子優先順位

あるトークン上でシフト・還元衝突がおこったとき、そのトークンに演算子優先順位が設定してあると衝突を解消できる場合があります。そのようなものとして特に有名なのは数式の演算子と if...else 構文です。

優先順位で解決できる文法は、うまく文法をくみかえてやれば、優先順位なしでも同じ効果を得ることができます。しかしたいていの場合は優先順位を設定して解決するほうが文法を簡単にできます。

シフト・還元衝突がおこったとき、Racc はまずその規則に順位が設定されているか調べます。規則の順位は、その規則で一番右にある終端トークンの優先順位です。
たとえば
target: TERM_A nonterm_a TERM_B nonterm_b
のような規則の順位はTERM_B の優先順位になります。
もしTERM_Bに 優先順位が設定されていなかったら、優先順位で衝突を解決することはできないと判断し、「Shift/Reduce conflict」を報告します。

演算子の優先順位はつぎのように書いて定義します。

```
prechigh
  nonassoc PLUSPLUS
  left       MULTI  DEVIDE
  left       PLUS MINUS
  right      '='
preclow
```

prechigh に近い行にあるほど優先順位の高いトークンです。上下をまるごとさかさまにして preclow...prechigh の順番に書くこともできます。left などは必ず行の最初になければいけません。
left right nonassoc はそれぞれ「結合性」を表します。結合性によって、同じ順位の演算子の規則が衝突した場合にシフト還元のどちらをとるかが決まります。
たとえば a - b - c が
(a - b) - c になるのが左結合 (left) です。四則演算は普通これです。一方
a - (b - c) になるのが右結合 (right) です。
代入を表すイコールは普通 右結合（right）です。
またこのように演算子が続くのがエラーとなる場合、非結合 (nonassoc) です。C 言語の ++ や単項のマイナスなどがこれにあたります。

ところで、説明したとおり、通常は
還元する規則の最後のトークンが順位を決めるのですが、
ある規則に限ってそのトークンとは違う順位にしたいことも あります。

例えば符号反転のマイナスは引き算のマイナスより順位を高くしないといけません。
このような場合 yacc では %prec を使います。
racc ではイコール記号を使って同じことができます。

```
prechigh
  nonassoc UMINUS    # これが最上位
  left          '*' '/'
  left          '+' '-'
preclow
(略)
exp: exp '*' exp
       | exp '-' exp
       | '-' exp       = UMINUS  # ここだけ順位を変えたい
```

このように記述すると、'-' exp の規則の順位が UMINUS の順位になります。
こうすることで符号反転の '-' は '*' よりも順位が高くなるので、意図どおりになります。