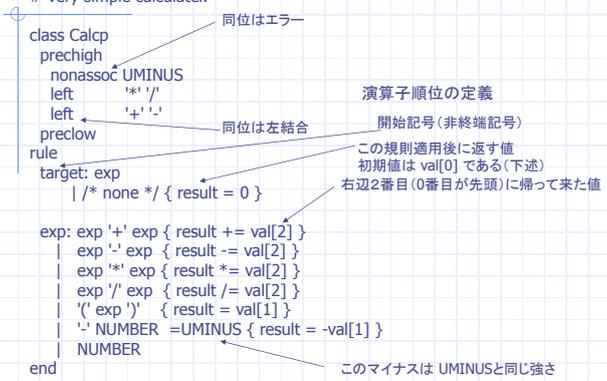


# コンパイラ理論 6 Racc の例

櫻井彰人

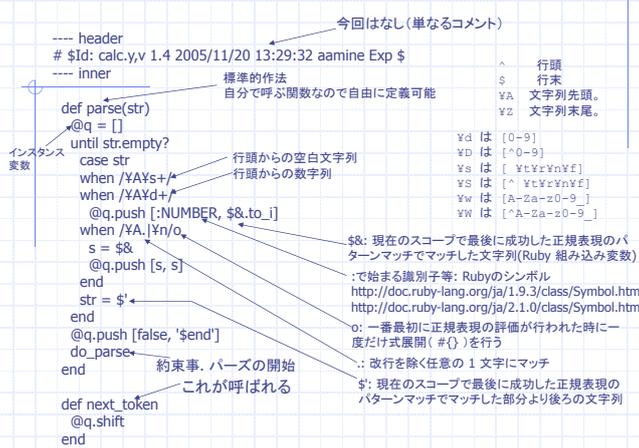
```
# $Id: calc.y,v 1.4 2005/11/20 13:29:32 aamine Exp $
#
# Very simple calculator.

class Calc
  prehigh
  nonassoc UMINUS
  left '+' '/'
  left '-' '*'
  prelow
  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
```

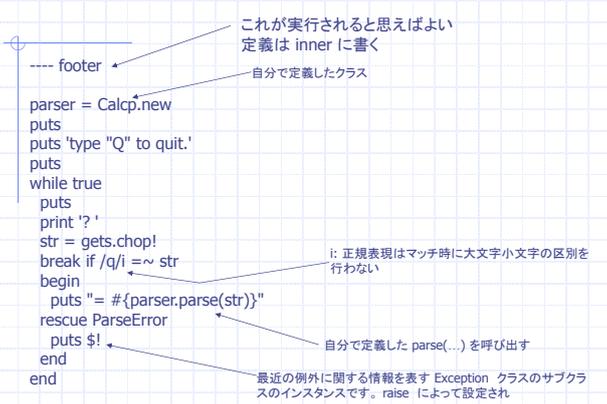


```
---- header
# $Id: calc.y,v 1.4 2005/11/20 13:29:32 aamine Exp $
---- inner

def parse(str)
  @q = []
  until str.empty?
  case str
  when /%A%+*/
  when /%A%+*/
  @q.push [:NUMBER, $&.to_i]
  when /%A.%/
  s = $&
  @q.push [s, s]
  end
  str = $'
  @q.push [false, 'end']
  do_parse
  end
  def next_token
  @q.shift
  end
```



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



## 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。

```
prehigh
nonassoc PLUSPLUS
left MULTI DEVICE
left PLUS MINUS
right '='
prelow
```

prehigh 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 prelow...prehigh 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と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。