

コンパイラ理論 7 Racc その2

櫻井彰人

Racc はパーサージェネレータ

- ◆ コンパイラコンパイラは、意味(アクション)もコンパイラコンパイラ記述言語で書ける(べき)。
- ◆ しかし、RaccはRubyで書き、yacclはCで書く。
- ◆ これは、Raccやyacclは(Bisonも...も)「パーサージェネレータ」であるということの意味する

Racc の練習問題1

- ◆ calc.y を改造して、べき乗ができるようにして下さい。べき乗演算子は、一文字の"^^"としましょう。

実行すると

```
$ ruby calc.y
type "Q" to quit.

? 2^3
= 8

?
? 2^3^4
= 2417851639229258349412352

? -2^(3^4)
= -2417851639229258349412352

? 4^2^3
= 65536

? -4^(2^3)
= -65536

?
```

分かりにくいところ

^ 行頭。文字列の先頭や改行文字の直後の位置にマッチします。
¥A 文字列先頭。^とは異なり改行の有無には影響しません。
¥s 空白文字。[¥n¥r¥f]と同じ
¥1, ¥2 ... ¥n 後方参照(back reference)。
後方参照 n 番目の括弧(正規表現 () グループ)にマッチした文字列にマッチします。
例: regexp("Hello to the the world", /(¥w+)¥1/)

べき乗の導入

- ◆ 2項演算子なので、他の2項演算子と同様に考えればよい
- ◆ 結合度(優先順位)は、*/より上位、単項演算子よりも上位
 - $-2^4 = -(2^4)$
- ◆ 右結合(right associative)である
 - $2^3^4 = 2^{(3^4)}$

```
class Calc
  precedence
    right '^'
    nonassoc UMI NUS
    left '*' '/'
    left '+ '-'
  precedence
  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 '^' exp { result **= val[2] }
    '(' exp ')' { result = val[1] }
    '-' exp =UMI NUS { result = -val[1] }
    NUMBER

```

必要です

補足

- ◆ Ruby のように、** でべき乗を表すには、scanner に少し変更が必要

```

class Calc
  prechigh
  right '***'
  nonassoc UMINUS
  left '*/'
  left '^'
  prec_low
  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 '^' exp { result **= val [2] }
  | '(' exp ')' { result = val [1] }
  | '-' exp =UMINUS { result = -val [1] }
  | NUMBER
end

```

```

case str
when /\A\s+$/
when /\A\d+$/
  @q.push [ :NUMBER, $&.to_i ]
when /\A\s*\s*/
  @q.push [ $&, $& ]
when /\A.|\n/o
  s = $&
  @q.push [ s, s ]
end

```

変数の導入

- ◆ 変数宣言なしとしよう
- ◆ 変数は、式の左辺(代入される)と右辺(引用される)に現れる。
- ◆ 同じものだが、左辺に現れるときと右辺に現れるときとは、扱いが全く異なる
 - 右辺で参照されたときは、その値が使われる
 - 左辺で参照されたときは、代入先である「場所」が使われる
 - インタープリタのときは、常識に従えば、まあ、大丈夫
- ◆ とりあえず、代入式を式(exp)とは別に定義する

変数名表

- ◆ 変数の型は考えない(宣言もない)
- ◆ 必要な表は、変数名と値の対応を記録する表だけ
 - 関数定義を許すには、この表を拡張する必要がある。変数名の表ではなく、識別子の表となる。
- ◆ Rubyにはハッシュ表がある。これは便利。

```

rule
target: exp
| assign
| /* 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 '^' exp { result **= val [2] }
| '(' exp ')' { result = val [1] }
| '-' exp =UMINUS { result = -val [1] }
| NUMBER
| IDENT { result = do_varref( val [0] ) }
assign : IDENT '=' exp { result = do_assign( val [0], val [2] ) }
end

```

変数に入っている値を変数表からもってくる。未定義のときは、exception発生予定

右辺の値を、変数表に入れる

Defaultのアクションは、{ result = result }, { result = val[0] } (ともに同語反復)である

```

---- inner
def initialize
  @vtable = {}
end
def do_assign( vname, val )
  @vtable[ vname ] = val
end
def do_varref( vname )
  @vtable[ vname ] ||
  raise( ParseError, "unknown variable #{vname}" )
end

```

innerの先頭に入れる

Hashにするつもりで初期化

Rubyのシンボル型のデータがくるはず

値がnilでも未定義になるがご勘弁

Exceptionを発生させる

```

---- footer
parser = Calc.new
puts
puts 'type "0" 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

```

Exception ParseErrorを受取る

Exceptionの内容

Raccの練習問題2

- ◆ 浮動小数点数が扱えるようにする。
- ◆ Ruby は変数に型がないため、今作っている電卓にも型がないため、数値表現のみを可能とすれば、よい
- ◆ scannerの変更ですむ

```
case str
  when /%A%+./
    when [ ]
      @q.push [ :NUMBER, [ ] ]
    when /%A%d+./
      @q.push [ :NUMBER, $&.to_i ]
```

ここに正規表現を入れる

ここに、文字列から不動小数点数への変換を入れる

Ruby 補足

- ◆ 配列: 要素の追加・削除
- ◆ push メソッドを使用して、配列に要素を追加することができる。

```
irb(main):001:0> a = [1,2,3,4]
=> [1, 2, 3, 4]
irb(main):002:0> a.push(10)
=> [1, 2, 3, 4, 10]
irb(main):003:0>
```

- ◆ pop メソッドを使用して、配列の最後の要素を取り出すことができる

```
irb(main):003:0> a.pop
=> 10
irb(main):004:0> a
=> [1, 2, 3, 4]
irb(main):005:0>
```

Ruby 補足: ハッシュ

- ◆ 配列では、インデックスを用いて要素を指定する。ハッシュでは、キーと呼ばれるものを用いて要素を指定する。

```
irb(main):001:0> h = Hash::new
=> {}
irb(main):002:0> h['apple'] = 150
=> 150
irb(main):003:0> h['banana'] = 200
=> 200
irb(main):004:0> h['lemon'] = 300
=> 300
irb(main):005:0> h
=> {"apple"=>150, "banana"=>200, "lemon"=>300}
irb(main):006:0> h['apple']
=> 150
irb(main):007:0>
```