コンパイラ理論 10
Racc その5 (if-then-else-end)

櫻井彰人

「Racc その4」はスキップ

練習問題としてはスキップ

練習問題4: if then else end

- Ruby の if then else end と if then end 相当の文が使えるようにしてください。
  - if 部分には数式を書き、その値が0ならばelse部分の値を、0以外であればthen部分の値を if 式の値とする
- 何となくできるようにするのは、簡単です。
  - 例えば、then部分、else部分には、"代入のない"式しか現れないとする
  - 無駄な計算をしてもよいとする
    - 結果には不要な式（例えば、条件が成立しているときのelse部分）の計算をしてもよいとする

例解

- 例えば exp の定義に次のようなものを付け加えればよい

```
| IF exp THEN exp END
  { if val[1] !=0 then result=val[3] else result=nil end}
| IF exp THEN exp ELSE exp END
  { if val[1] !=0 then result=val[3] else result=val[5] end}
```

```
a = 0
if a then 1 else 2 end
```
↓
= 2

```
a = 1
if a then 1 else 2 end
```
↓
= 1

プログラム作成上の注意

if, then, else, end といったキーワードが出てくる。
変数として認識しては困る。
予約語（つまり、システムが使用することを予約している語）とする。

```
def parse(str)

    # 色々

     when /¥A[a-zA-Z_]¥w*/
         word = $&
         @q.push [(RESERVED[word] || :IDENT),
                  RESERVED_V.key?(word) ? RESERVED_V[word] : word.intern ]
     when /¥A¥d+¥.¥d+/
       @q.push [:NUMBER, $&.to_f]

     # 色々

    end
```

```
RESERVED = {
  'if'    => :IF,
  'else'  => :ELSE,
  'while' => :WHILE,
  'then'  => :THEN,
  'do'    => :DO,
  'def'   => :DEF,
  'true'  => :TRUE,
  'false' => :FALSE,
  'nil'   => :NIL,
  'end'   => :END
}
```

```
RESERVED_V = {
  'true'  => true,
  'false' => false,
  'nil'   => nil
}
```

しかし、失敗？

- 代入式が書けるようにすると、結果が思わしくない

```
| IF exp THEN exp END
  { if val[1] !=0 then result=val[3] else result=nil end}
| IF exp THEN exp ELSE exp END
  { if val[1] !=0 then result=val[3] else result=val[5] end}
```

```
a=0
if a then b=1 else b=2 end
print( "b= ", b,"¥n")
```
↓
b= 2

```
a=1
if a then b=1 else b=2 end
print( "b= ", b,"¥n")
```
↓
b= 2

代入式を書けるようにする。
式として、代入式も認める

```
prechigh
  right '^'
  nonassoc UMINUS
  left '*' '/'
  left '+' '-'
  right '='
preclow
rule
 target: exp
    | assign
    | /* none */ { result = 0 }

 exp: exp '+' exp { result += val[2] }
    | exp '-' exp { result -= val[2] }

 # 色々

    | assign


 assign : IDENT '=' exp { result = do_assign( val[0], val[2] ) }

end
```

思わしくない理由

- "then" 部分も "else" 部分も、必ず、構文解析を行い
- 構文解析に成功すれば、意味解析（インタープリターでは「実行」）もしてしまう。
- if 全体の値は正しいのだが、副作用がある部分（代入！）はおかしくなる。
  - また、効率が悪い（計算する必要のない計算を行ってしまう）

```
| IF exp THEN exp END
  { if val[1] !=0 then result=val[3] else result=nil end}
| IF exp THEN exp ELSE exp END
  { if val[1] !=0 then result=val[3] else result=val[5] end}
```

副作用とは

プログラミングにおける副作用とは、ある機能がコンピュータの(論理的な)状態を変化させ、それ以降で得られる結果に影響を与えることをいう。代表的な例は変数への値の代入である。

例えば与えられた数字を二倍して返す機能"double"があるとする。この機能を右図で実現すれば次が成立する。
1. 同じ入力を与えれば必ず同じ結果が得られる
2. 他のいかなる機能の結果にも影響を与えない

なお、このような性質を参照透過性という。

```
function double(x) {
    return 2*x
}
```

一方、状態を持つ機能"add1"を考える。このadd1は外側の変数eを1増加させて返すものとする。

```
function add1() {
    return e = e+1
}
```

このような機能は見えない所で状態を変化させてしまうために、参照透過性の一つ目の条件が成立せず、また他のeを利用する機能では、結果が変化する可能性があるので二つ目の条件も成立しない。このadd1 は副作用を持つ機能である。

（Wikipediaの記述を少し訂正）

回避方法はあるか？

- ある。action は途中に書ける。
  そこで、
  - 「ifの条件」の結果を（例えば）@goに記憶する
  - then部は,「ifの条件」の結果がtrueの時、@go=trueとし、else部は,「ifの条件」の結果がfalseの時、@go=trueとする
  - then部やelse部に書けるexp1では、@goがtrueの時のみ、計算を行う。

つまり

```
IF exp THEN exp ELSE exp END
   { if val[1] !=0 then result=val[3] else result=val[5] end}
```

⇩

```
IF exp {値を保存} THEN exp1 ELSE exp2 END
   { if val[1] !=0 then result=val[3] else result=val[5] end}
```

そして

```
exp1:
  exp1 '+' exp1 { if 条件がtrue then result += val[2] end}
   など

exp2:
  exp2 '+' exp2 { if 条件がfalse then result += val[2] end}
   など
```

つまり（その２）

```
IF exp THEN exp ELSE exp END
   { if val[1] !=0 then result=val[3] else result=val[5] end}
```

⇩

```
IF exp {値を@goに保存} THEN exp1 ELSE {@goを反転} exp1 END
   { if val[1] !=0 then result=val[3] else result=val[5] end}
```

そして

```
exp1:
  exp1 '+' exp1 { if @go==true then result += val[2] end}
   など
```

回避方法を試してみた

- だめだった
  - 規則記述の途中にあるアクション記述の処理がうまく行われていない。
- racc のバグか、未実装と思われる

- 対処方法は、ある。別の非終端記号を定義する。

つまり（その3）

```
IF exp THEN exp ELSE exp END
   { if val[1] !=0 then result=val[3] else result=val[5] end}
                              ⇩
IF exp {値を@goに保存} THEN exp1 ELSE {@goを反転} exp1 END
   { if val[1] !=0 then result=val[3] else result=val[5] end}
                              ⇩
IF expC THEN exp1 ELSElabel exp1 END
   { if val[1] !=0 then result=val[3] else result=val[5] end}

expC : exp  {値を@goに保存}
ELSElabel : ELSE  {@goを反転}
```

そして

```
exp1:
  exp1 '+' exp1 { if @go==true then result += val[2] end}
    など
```

出来上がり

```
def initialize
   @vtable={}
   @go=[]
end
```

```
exp: exp '+' exp { result += val[2] }

  # 色々

  | IF expC THEN exp1 END
      { if val[1] !=0 then result=val[3] else result=nil end}
  | IF expC THEN exp1 ELSEc exp1 END
      { if val[1] !=0 then result=val[3] else result=val[5] end}
  | assign

assign : IDENT '=' exp { result = do_assign( val[0], val[2] ) }

expC  : exp  { @go = result!=0 }
ELSEc : ELSE { @go = (not @go) }

exp1: exp1 '+' exp1 { if @go then result += val[2] else result = nil end }
    | exp1 '-' exp1 { if @go then result -= val[2] else result = nil end }
    | exp1 '*' exp1 { if @go then result *= val[2] else result = nil end }
    | exp1 '/' exp1 { if @go then result /= val[2] else result = nil end }
    | exp1 '^' exp1 { if @go then result **= val[2] else result = nil end }
    | '(' exp1 ')'  { if @go then result = val[1] else result = nil end }
    | '-' exp1  =UMINUS { if @go then result = -val[1] else result = nil end }
    | NUMBER
    | IDENT       { if @go then result = do_varref( val[0] ) else result = nil end }
    | assign1

assign1 : IDENT '=' exp1 { if @go then result = do_assign( val[0], val[2] ) else result = nil end}
```

if のネスト

if 文の中に、if文を書けるようにするには、どうしたらよいだろうか？
（先ほどのraccプログラムでは、できない。exp1の定義に if文 が入っていないからである。
exp の定義のように、exp1の定義に if文 の定義を入れればよいかというと、話はそう簡単ではない。アクション部分（インタープリター部分、意味解析部分）が思うように動かなくなる。
理由は簡単。@go という一つの変数が、ネストした内側のif文でも、外側のif文でも使われているからである。
回避も（ほぼ）簡単。スタックに入れればよいからである。
@go をスタックにする。そして、値を決める if文 の条件式に伴うアクション部分の最後で、もともとの@goをスタック先頭に入れれば（pushすれば）よい。
そして、if文の最後を示すendの認識後に、スタックをpopすればよい。

なお、上記の場合、exp1の評価を行う（計算を行う）のは、@goスタックの要素がすべてtrueのときだけである。一つでもfalseであれば、その場所のより外側のどこかの if部分 の条件が、自分が実行してはいけない値になっているからである。

プログラム

```
def aand( a )
   a.inject{|x,y| x and y}
end
```

```
expC  : exp  { @go.push( result!=0 ) }
ELSEc : ELSE { @go.push( not( @go.pop ) ) }
ENDc  : END  { @go.pop }

exp1: exp1 '+' exp1 { if aand(@go) then result += val[2] else result = nil end }
    | exp1 '-' exp1 { if aand(@go) then result -= val[2] else result = nil end }
    | exp1 '*' exp1 { if aand(@go) then result *= val[2] else result = nil end }
    | exp1 '/' exp1 { if aand(@go) then result /= val[2] else result = nil end }
    | exp1 '^' exp1 { if aand(@go) then result **= val[2] else result = nil end }
    | '(' exp1 ')'  { if aand(@go) then result = val[1] else result = nil end }
    | '-' exp1  =UMINUS { if aand(@go) then result = -val[1] else result = nil end }
    | NUMBER
    | IDENT       { if aand(@go) then result = do_varref( val[0] ) else result = nil end }
    | IF expC THEN exp1 ENDc
                { if val[1] !=0 then result=val[3] else result=nil end}
    | IF expC THEN exp1 ELSEc exp1 ENDc
                { if val[1] !=0 then result=val[3] else result=val[5] end}
    | assign

assign : IDENT '=' exp1 { if aand(@go) then result = do_assign( val[0], val[2] ) else result = nil end}
```

構文規則を短く

ことの是非は別にして、exp と exp1 は一緒にすることができる。

```
exp: exp '+' exp { if aand(@go) then result += val[2] else result = nil end }
    | exp '-' exp { if aand(@go) then result -= val[2] else result = nil end }
    | exp '*' exp { if aand(@go) then result *= val[2] else result = nil end }
    | exp '/' exp { if aand(@go) then result /= val[2] else result = nil end }
    | exp '^' exp { if aand(@go) then result **= val[2] else result = nil end }
    | '(' exp ')'  { if aand(@go) then result = val[1] else result = nil end }
    | '-' exp  =UMINUS { if aand(@go) then result = -val[1] else result = nil end }
    | NUMBER
    | IDENT       { if aand(@go) then result = do_varref( val[0] ) else result = nil end }
    | IF expC THEN exp ENDc
                { if val[1] !=0 then result=val[3] else result=nil end}
    | IF expC THEN exp ELSEc exp ENDc
                { if val[1] !=0 then result=val[3] else result=val[5] end}
    | assign

assign : IDENT '=' exp { if aand(@go) then result = do_assign( val[0], val[2] ) else result = nil end}
```

```
def initialize
   @vtable={}
   @go=[true]
end
```

繰り返し（ループ）

- 次の大きな機能としては、繰り返し（ループ）がある。
- この機能は、プログラムの字面上では同じもの（プログラムの一部）を繰り返し実行する。
- 今実装をしている calc.y では、構文解析の意味解析の一つとして、実行を行っている。
- 用いている構文解析方法であるraccは、バックトラックを行わないため、ある文字列（繰り返しの中身。while ループの中身）を繰り返して解析することはない。
  すなわち、「繰り返し」を構文解析の一環として行うことはできない。
- では、どうするか？　次回の楽しみ。