

平成26年度3回生前期学生実験SW 中間レポート2

松田貴大

学籍番号:1029-24-4015

提出日：平成26年6月12日 17:00

提出期限：平成26年6月27日 16:00

1 はじめに

使用言語は Ruby(2.1.0p0) で、パーサの作成には racc(1.4.11)¹を用いている。

2 課題6

2.1 ソースコード

パスは、~/2014sw/compiler/report2/compiler.y である。

```
1 # $Id: compiler.y
2 #
3 # TinyC compiler
4
5 class Tinc
6   prehigh
7     right DATATYPE
8     right IF
9     right ELSE
10    right WHILE
11    right RETURN
12    left LE
13    left GE
14    left EQUAL
15    left NOTEQUAL
16    left LOGICALAND
17    left LOGICALOR
18    left '*' '/' '%'
19    left '+' '-'
20    left '<' '>'
21    right '='
22    right '(' '{'
23    left ')' '}'
24  preclow
25
26 rule
27   program
28     : external_declaration
29     | program external_declaration
```

¹<http://i.loveruby.net/ja/projects/racc/>

```

30         {
31             result = val[0] + val[1]
32         }
33     external_declaration
34         : declaration
35         {
36             result = val[0]
37         }
38         | function_definition
39         {
40             result = [val[0]]
41         }
42     declaration
43         : DATATYPE declarator_list ';'
44         {
45             result = []
46             for i in val[1]
47                 result += [ [val[0], i] ]
48             end
49         }
50     declarator_list
51         : declarator
52         {
53             result = [val[0]]
54         }
55         | declarator_list ',' declarator
56         {
57             result += [val[2]]
58         }
59     declarator
60         : IDENTIFIER
61     function_definition
62         : DATATYPE IDENTIFIER '('
63         {
64         }
65         parameter_type_list_opt ')'
66         {
67         }
68         compound_statement
69         {
70             result = [ [val[0], val[1]], val[4], val[7] ]
71         }
72     parameter_type_list

```

```

73         : parameter_declaration
74         {
75             result = [val[0]]
76         }
77     | parameter_type_list ',' parameter_declaration
78     {
79         result = val[0] + [val[2]]
80     }
81 parameter_type_list_opt
82     : parameter_type_list
83     | /* none */
84     {
85         result = []
86     }
87 parameter_declaration
88     : DATATYPE declarator
89     {
90         result = [val[0], val[1]]
91     }
92 statement
93     : ';'
94     {
95         result = ''
96     }
97     | expression ';'
98     {
99         result = val[0]
100    }
101     | compound_statement
102     | IF '(' expression ')' statement
103     {
104         result = [['IF'] + val[2] + [val[4]]]
105     }
106     | IF '(' expression ')' statement ELSE statement
107     {
108         result = [['IF'] + val[2] + [val[4], val[6]]]
109     }
110     | WHILE '(' expression ')' statement
111     {
112         result = [['WHILE'] + val[2] + [val[4]]]
113     }
114     | RETURN expression ';'
115     {

```

```

116         result = [['RETURN'] + val[1]]
117     }
118     compound_statement
119     : '{'
120     {
121     }
122     declaration_list_opt statement_list_opt '}'
123     {
124         result = []
125         if val[2] != nil
126             result += val[2]
127         end
128         if val[3] != nil
129             result += val[3]
130         end
131     }
132     declaration_list
133     : declaration
134     | declaration_list declaration
135     {
136         result = [val[0], val[1]]
137     }
138     declaration_list_opt
139     : declaration_list
140     | /* none */
141     statement_list
142     : statement
143     | statement_list statement
144     {
145         result = val[0] + val[1]
146     }
147     statement_list_opt
148     : statement_list
149     | /* none */
150     expression
151     : assign_expr
152     {
153         result = [val[0]]
154     }
155     | expression ',' assign_expr
156     {
157         result = [val[0], val[2]]
158     }

```

```

159  assign_expr
160      : logical_OR_expr
161      | IDENTIFIER '=' assign_expr
162      {
163          result = ['=', val[0], val[2]]
164      }
165  logical_OR_expr
166      : logical_AND_expr
167      | logical_OR_expr LOGICALOR logical_AND_expr
168      {
169          result = ['||', val[0], val[2]]
170      }
171  logical_AND_expr
172      : equality_expr
173      | logical_AND_expr LOGICALAND equality_expr
174      {
175          result = ['&&', val[0], val[2]]
176      }
177  equality_expr
178      : relational_expr
179      | equality_expr EQUAL relational_expr
180      {
181          result = ['==', val[0], val[2]]
182      }
183      | equality_expr NOTEQUAL relational_expr
184      {
185          result = ['!=', val[0], val[2]]
186      }
187  relational_expr
188      : add_expr
189      | relational_expr '<' add_expr
190      {
191          result = ['<', val[0], val[2]]
192      }
193      | relational_expr '>' add_expr
194      {
195          result = ['>', val[0], val[2]]
196      }
197      | relational_expr LE add_expr
198      {
199          result = ['<=', val[0], val[2]]
200      }
201      | relational_expr GE add_expr

```

```

202         {
203             result = ['>=', val[0], val[2]]
204         }
205     add_expr
206     : mult_expr
207     | add_expr '+' mult_expr
208     {
209         result = ['+', val[0], val[2]]
210     }
211     | add_expr '-' mult_expr
212     {
213         result = ['-', val[0], val[2]]
214     }
215     mult_expr
216     : unary_expr
217     | mult_expr '*' unary_expr
218     {
219         result = ['*', val[0], val[2]]
220     }
221     | mult_expr '/' unary_expr
222     {
223         result = ['/', val[0], val[2]]
224     }
225     unary_expr
226     : postfix_expr
227     | '-' unary_expr
228     {
229         result = -(val[1].to_i).to_s
230     }
231     postfix_expr
232     : primary_expr
233     | IDENTIFIER '(' argument_expression_list_opt ')'
234     {
235         if val[2] == nil
236             result = ['FCALL', val[0], []]
237         else
238             result = ['FCALL', val[0]] + [val[2]]
239         end
240     }
241     primary_expr
242     : IDENTIFIER
243     {
244         result = val[0]

```

```

245         }
246         | CONSTANT
247         | '(' expression ')',
248         {
249             result = [val[1]]
250         }
251     argument_expression_list
252     : assign_expr
253     {
254         result = [val[0]]
255     }
256     | argument_expression_list ',', assign_expr
257     {
258         result = val[0] + [val[2]]
259     }
260     argument_expression_list_opt
261     : argument_expression_list
262     | /* none */
263 end
264
265 —— header
266 # $Id: calc.y,v 1.4 2005/11/20 13:29:32 aamine Exp $
267 —— inner
268
269 def parse(str)
270     @q = []
271     until str.empty?
272         case str
273         when /\A\s+/
274         when /\A\d+/
275             @q.push [:CONSTANT, $&.to_i]
276         when /\A(&&)/
277             @q.push [:LOGICALAND, $&]
278         when /\A(\|\|)/
279             @q.push [:LOGICALOR, $&]
280         when /\A(int)/
281             @q.push [:DATATYPE, $&]
282         when /\A(if)/
283             @q.push [:IF, $&]
284         when /\A(else)/
285             @q.push [:ELSE, $&]
286         when /\A(while)/
287             @q.push [:WHILE, $&]

```



```

288     when /\A(<=)/
289         @q.push [:LE, $&]
290     when /\A(>=)/
291         @q.push [:GE, $&]
292     when /\A(=)/
293         @q.push [:EQUAL, $&]
294     when /\A(!=)/
295         @q.push [:NOTEQUAL, $&]
296     when /\A(return)/
297         @q.push [:RETURN, $&]
298     when /\A[a-zA-Z]\w*/
299         @q.push [:IDENTIFIER, $&]
300     when /\A.\n/o
301         s = $&
302         @q.push [s, s]
303     end
304     str = $'
305 end
306 @q.push [false, '$end']
307 do_parse
308 end
309
310 def next_token
311     @q.shift
312 end
313
314 —— footer
315
316 parser = Tinyc.new
317
318 str = ''
319 while true
320     add = gets
321     if add == nil
322         break
323     else
324         str += add
325     end
326 end
327 if str != nil
328     str.chop!
329     begin

```

```

330     puts "success!!! \n result => \n#{parser.parse(str)}"
331   rescue ParseError
332     puts $!
333   end
334 end

```

2.2 設計方針

実験資料とは少し異なる仕様の構文木を生成する。たとえば、実験資料の例(課題7のもの)を見ると、以下のような構文木が生成されている²。

TinyC プログラム

```

int x;
int f(int x, int y)
{
    int x;
    {
        int x, y;
        x+y;
        {
            int x, z;
            x+y+z;
        }
    }
    {
        int w;
        x+y+w;
    }
    x+y;
}
int g(int y)
{
    int z;

```

²以下では、構文木を Ruby 形式の配列で記述する。

```

    f(x, y);
    g(z);
}

```

構文木

```

[int x]
[[int f] [[int x][int y]]
[
  [int x]
  [
    [int x y]
    [+ x y]
    [
      [int x z]
      [+ [+ x y] z]
    ]
  ]
  [
    [int w]
    [+ [+ x y] w]
  ]
  [+ x y]
]]
[[int g] [[int y]]
[
  [int z]
  [FCALL f x y]
  [FCALL g z]
]]

```

今回採用した仕様では、同じ TinyC プログラムは以下のような構文木を生み出す³。

構文木

³実際には改行やインデントは行われない。

```
[
  ["int" "x"]
  [["int" "f"] [ ["int" "x"] ["int" "y"] ]
  [
    ["int" "x"]
    ["int" "x"]
    ["int" "y"]
    ["+" "x" "y"]
    ["int" "x"]
    ["int" "z"]
    ["+" ["+" "x" "y"] "z"]
    ["int" "w"]
    ["+" ["+" "x" "y"] "w"]
    ["+" "x" "y"]
  ]
  [["int" "g"] [ ["int" "y"] ]
  [
    ["int" "z"]
    ["FCALL" "f" ["x" "y"] ]
    ["FCALL" "g" ["z"] ]
  ]
]]
]
```

以下では、資料の仕様から変更した点について述べる。

全体で1つの配列 大域で複数の変数や関数が宣言されている場合、全体を1つの配列にまとめるようにした。これは、後にこのプログラム全体を処理する際に、処理を楽にするためである。

複数変数宣言

TinyC プログラム

```
int x, y;
```

以上のように、データ型宣言の後に複数変数を宣言した場合、資料では以下のように構文木を生成している。

構文木

```
[int x y]
```

この設計だと、変数宣言時に配列の長さが可変 (2 以上) になってしまい、処理がやや面倒になる。そのため、複数変数宣言されている場合、以下のように別々の配列を生成するようにした。

構文木

```
["int" "x"]  
["int" "y"]
```

不必要な括弧 資料では、スコープを制限するための中括弧 ('{' や '}') によって、構文木に括弧が生成される。しかし、`racc` では構文木の生成と同時にスコープの深さを調べることが出来る (すなわち、構文木生成と同時に意味解析が完全に可能である) ので、これらの括弧は不必要となる。よって、不必要な括弧は省略するようにした⁴。

関数呼び出し時の引数

TinyC プログラム

```
f(x, y);
```

以上のように、関数呼び出しの際に引数が複数ある場合、資料では以下のように構文木を生成している。

構文木

```
[FCALL f x y]
```

この設計だと、関数呼び出し時に配列の長さが可変 (2 以上) になってしまい、処理がやや面倒になる。そのため、どのような場合でも引数全体を配列で包み、関数呼び出し部分の配列の長さは 3 になるようにした。

⁴もちろん、`if` や `while` などでは範囲を括弧で囲む。

構文木

```
["FCALL" "f" ["x" "y"]]
```

同様に、以下のようなコードは、以下のような構文木になる。

TinyC プログラム

```
g(int x);  
h();
```

構文木

```
["FCALL" "g" ["x"]]  
["FCALL" "h" []]
```

2.3 文法規則のアクション部分について

2.4 演算子の優先順位が正しく反映される理由

2.5 工夫点

節 2.2 で述べたように、実験資料とは異なる構造の構文木を生成している。

2.6 実行例と結果

2.1 節で述べたソースコードから生成されたパーサと構文木生成プログラムを使い、いくつかの TinyC プログラムの構文を解析した結果を示す。

2.6.1 例 1

```
1 int x;  
2 int f(int x, int y)  
3 {  
4     int x;  
5     {  
6         int x, y;
```

```

7           x+y;
8           {
9               int x, z;
10              x = x*y-z/x;
11          }
12      }
13      {
14          int w;
15          x+y+w;
16      }
17      x+y;
18  }
19  int g()
20  {
21      int z;
22      f(x, y);
23      g();
24  }

```

結果

```

success!!!
result => int

```

3 感想