平成26年度 3回生前期学生実験SW
最終レポート

松田貴大

学籍番号:1029-24-4015

提出日：平成26 年7 月4 日17:00
提出期限：平成26 年7 月25 日 17:00

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　ソースコード

パスは、~/2014sw/compiler/report3/compiler.yである。プログラムをレポートに載せると膨大になるので、やめておく。

3　対象文法の変更

言語仕様を拡張し、対象となる文法を変更したので、以下に記述する。

3.1　文法

```
program
    : external_declaration
    | program external_declaration
external_declaration
    : declaration
    | function_definition
declaration
    : DATATYPE declarator_list ';'
declarator_list
    : declarator
    | declarator_list ',' declarator
declarator
    : IDENTIFIER
function_definition
    : DATATYPE IDENTIFIER '(' parameter_type_list_opt ')'
      compound_statement
parameter_type_list
```

[1]http://i.loveruby.net/ja/projects/racc/

```
    : parameter_declaration
    | parameter_type_list ',' parameter_declaration
parameter_type_list_opt
    : parameter_type_list
    | /* none */
parameter_declaration
    : DATATYPE declarator
statement
    : ';'
    | 'continue' ';'
    | 'break' ';'
    | expression ';'
    | compound_statement
    | 'if' '(' expression ')' statement
    | 'if' '(' expression ')' statement 'else' statement
    | 'while' '(' expression ')' statement
    | 'for' '(' expression ';' expression ';' expression ')'
      statement
    | 'return' expression ';'
compound_statement
    : '{' declaration_list_opt statement_list_opt '}'
declaration_list
    : declaration
    | declaration_list declaration
declaration_list_opt
    : declaration_list
    | /* none */
statement_list
    : statement
    | statement_list statement
statement_list_opt
    : statement_list
    | /* none */
expression
    : assign_expr
```

```
    | expression ',' assign_expr
assign_expr
    : logical_OR_expr
    | IDENTIFIER '=' assign_expr
    | IDENTIFIER '+=' assign_expr
    | IDENTIFIER '-=' assign_expr
    | IDENTIFIER '*=' assign_expr
    | IDENTIFIER '/=' assign_expr
    | IDENTIFIER '%=' assign_expr
logical_OR_expr
    : logical_AND_expr
    | logical_OR_expr '||' logical_AND_expr
logical_AND_expr
    : or_expr
    | logical_AND_expr '&&' or_expr
or_expr
    : xor_expr
    | or_expr '|' xor_expr
xor_expr
    : and_expr
    | xor_expr '^' and_expr
and_expr
    : equality_expr
    | and_expr '&' equality_expr
equality_expr
    : relational_expr
    | equality_expr '==' relational_expr
    | equality_expr '!= relational_expr
relational_expr
    : add_expr
    | relational_expr '<' add_expr
    | relational_expr '>' add_expr
    | relational_expr '<=' add_expr
    | relational_expr '>=' add_expr
add_expr
```

```
    : mult_expr
    | add_expr '+' mult_expr
    | add_expr '-' mult_expr
mult_expr
    : unary_expr
    | mult_expr '*' unary_expr
    | mult_expr '/' unary_expr
    | mult_expr '%' unary_expr
unary_expr
    : posifix_expr
    | '-' unary_expr
posifix_expr
    : subprimary_expr
    | IDENTIFIER '(' argument_expression_list_opt ')'
subprimary_expr
    : primary_expr
    | primary_expr '++'
    | primary_expr '--'
primary_expr
    : IDENTIFIER
    | CONSTANT
    | '(' expression ')'
argument_expression_list
    : assign_expr
    | argument_expression_list ',' assign_expr
argument_expression_list_opt
    : argument_expression_list
    | /* none */
```

3.2 拡張点

具体的に、以下の仕様を加えた。

3.2.1 ビット演算子

AND、OR、XOR のビット演算をそれぞれ'&'、'|'、'^' を用いて導入した。意味解析は他の算術演算と同様に行うことが出来る。コード生成では、それぞれアセンブリ命令の AND、OR、XOR を用いることで対応出来る。

3.2.2 剰余演算子

剰余演算子'%' を導入した。意味解析は他の算術演算と同様に行うことが出来る。コード生成では、idiv を用いると edx に剰余が保存されるので、それを用いることで対応出来る。

3.2.3 複合代入演算子

複合代入演算子'+='、'-='、'*='、'/='、'%=' を導入した。たとえば、a+=5; という命令では、a に 5 が加算される。すなわち、a=a+5; と同じ演算となる。意味解析は、代入演算と算術演算のものを組み合わせることで行うことが出来る。コード生成では、代入演算と算術演算のコードを生成することで対応出来る。

3.2.4 インクリメント、デクリメント

インクリメント++、デクリメント--を追加した。たとえば、a++; という命令は、a+=1; と等しい。意味解析は、複合代入演算子のもので行うことが出来る。コード生成では、複合代入演算子のオペランドが 1 になっているものとすれば対応出来る。

3.2.5 for

ループ文 for を導入した。for(exp1;exp2;exp3) statement で、exp1;while(exp2){statement;exp3;}と同じ意味を表す。意味解析は、代入演算と while と算術演算のものを組み合わせることで行うことが出来る。コード生成では、代入演算と while と算術演算のコードを生成することで対応出来る。

3.2.6 continue、break

ループ脱出のための continue、break を導入した。continue は、その時点でループブロック内の statement を終了し、for 文の場合は exp3 を実行した後にループ条件のチェックに戻る。break は、その時点でループブロックを脱出する。意味解析については 4 節を参照のこと。コード生成については 5 節を参照のこと。

3.2.7 コメント

'//' 以降の 1 行および'/*A*/' の形式の A 部分をコメントとみなす処理を構文解析時に追加した。これは、以下のようなアルゴリズムで実現できる。

1. flag を 0 とする。
2. flag が 0 のとき、
   (a) '//' を読んだら flag を 1 とする。
   (b) '/*' を読んだら flag を 2 とする。
   (c) それ以外のときは通常の構文解析を行う。
3. flag が 1 のとき、
   (a) '\n' を読んだら flag を 0 とする。
   (b) それ以外のときは無視する。
4. flag が 2 のとき、
   (a) '*/' を読んだら flag を 0 とする。
   (b) それ以外のときは無視する。
5. 2. 以降を繰り返す。

4 意味解析 (課題 7)

意味解析では、構文解析によって生成された構文木に現れるシンボルが具体的にどのアドレスに存在するものか等を解析し、矛盾が無いかを解析すると共にシンボルの意味を確定させていく。

4.1 設計方針

意味解析時に、シンボルに付加できる意味が存在しなかったり曖昧であったりする場合や、continue と break がループブロック内に存在しない場合はエラーを出して構文木を破棄する[2]。また、シンボル宣言時に、異なる深さレベルの名前空間に既に同じ名前のシンボルが存在する場合は警告を表示する。

名前空間を保持するために、スタックを導入する。スタック内のデータは、シンボル名・深さレベル・種類・備考の情報を持つ。あるシンボルが宣言されると、スタックにシンボル情報が積まれる。ある深さレベルの意味解析が終了すると、終了した深さレベルを持つデータをスタックから取り出し、破棄する。スタック内のデータは、必ず深さレベルの小さい順に積まれているので、この方法で意味解析が可能である。

4.2 解析方法

以下では、構文木の各ノードからどのように意味解析を行っていくかを説明する。

4.2.1 宣言文

関数や変数の宣言文を表すノードは先頭要素が'int'から始まる配列である。具体的には、以下のようになっている[3]。

- 変数 … ["int", IDENTIFIER]
- 関数 … ["int", IDENTIFIER, parameters, statements]

宣言文は、配列の 2 番目の要素がシンボル名となっている。このシンボル名と同じ名前を持つデータがスタック内に存在するか確認する。データが存在する場合、現在の深さレベルとデータの深さレベルが等しい場合はエラーを出す。異なる場合は警告を出す。データが存在しない場合、シンボルが変数ならスタック状態からアドレスを確定させて構文木内の

[2]エラー行の表示を行いたかったが、生成規則のアクション部分で意味解析を行っていくためエラー行の確定が難しかったので、今回はエラー内容の表示のみを行う。

[3]以降、配列を Ruby 形式で記述する。

シンボルに情報を付加して備考とし、シンボルが関数なら引数の数を備考とした後、スタックにシンボルデータを積む。

なお、実験資料ではパラメータ宣言と変数宣言と関数宣言を分けて考えているが、特にその必要性は無いと判断したのでパラメータ宣言や関数宣言の意味解析も変数宣言に含んでいる[4]。

4.2.2 代入演算

代入演算を表すノードは先頭要素が'=' から始まる配列である。具体的には、以下のようになっている。

```
["=", IDENTIFIER, expression]
```

代入演算は、配列の 2 番目の要素が代入する変数名となっている。この変数名を 4.2.4 節の方法で解析する。

4.2.3 continue、break

continue、break のノードは、それぞれ ['CONTINUE']、['BREAK'] である。以下のアルゴリズムで、continue・break が適切な位置で使用されているか解析出来る。

1. ループの深さを保持する変数 loop_depth を用意して、0 を代入する。
2. FOR、WHILE 文のループブロックに入るとき、loop_depth に 1 を足す。
3. ループブロックを抜けるとき、loop_depth から 1 を引く。
4. continue または break が現れたとき、loop_depth が 0 ならエラー。
5. 2. 以降を繰り返す。

[4]関数宣言は必ず深さレベル 0 で行われるため、二重宣言の際は必ずエラー検出が可能である。

4.2.4 変数参照

変数名と同じ名前を持つ変数データがスタック内に存在するか確認する。データが存在しない場合、エラーを出す。データが存在する場合、スタック上で最も近い位置に存在する同じ名前のデータを参照する変数として、構文木内の変数にアドレス情報を付加する。

4.2.5 関数呼び出し

関数呼び出しを表すノードは先頭要素が'FCALL'から始まる配列である。具体的には、以下のようになっている。

```
["FCALL", IDENTIFIER, parameter_list]
```

関数名と同じ名前を持つ関数データがスタック内に存在するか確認する。データが存在しない場合、未定義関数の呼び出しとして警告を出す。データが存在する場合、引数の数を照合して一致しない場合はエラーを出す。

4.3 実行例と結果

2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後に意味解析を行った結果を示す。

4.3.1 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+y+z;
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);
23              g();
24      }
```

結果

```
warning: Variable 'x' is already defined at level 0.
warning: Variable 'x' is already defined at level 1.
warning: Variable 'x' is already defined at level 2.
warning: Variable 'y' is already defined at level 1.
warning: Variable 'x' is already defined at level 3.
error: Parameter-length of function 'f' is 2.
```

4.3.2 例 2

```
1   int f(){
2       int i, n;
3       n = 3;
4       for(i=0; i<8; i++){
5           if(i == 5)  continue;
6           else if(i == 7){
7               while(n >= 0){
8                   if(n == 2)  break;
9                   n--;
10              }
11              break;
12          }
13      }
14      if(n == 0){
15          break;
16          continue;
17      }
18      return 0;
19  }
```

結果

```
error: break is NOT available here.
error: continue is NOT available here.
```

5　コード生成 (課題 8)

意味解析でエラーが発生しなかった場合、アドレス情報の付加された構文木を用いてコード生成を行っていく。

5.1　設計方針

命令・文の種類はノードの先頭要素で判断出来るので、それを用いて分岐処理を行い、コード生成を行う。

5.2　生成方法

以下では、コード生成方法のうち資料と異なるものについて述べる。

5.2.1　IF 文

IF 文を表すノードは以下のようになっている。

```
["IF", expression, statement1, statement2]
```

expression は真偽値を返す。真の場合は statement1、偽の場合は statement2 を実行する。すなわち、

```
if(expression)  statement1
else            statement2
```

に対応する。else が無い場合、statement2 は空の配列になる。

コード生成については資料と同じだが、ラベル名は以下の規則で付ける。

- else 対応部分 … 関数名_else 関数内で IF 文が出現した回数
- IF 文終了対応部分 … 関数名_if 関数内で IF 文が出現した回数

5.2.2 WHILE 文

コード生成については資料と同じだが、ラベル名は以下の規則で付ける。

- WHILE 文開始対応部分 … 関数名_whilestart 関数内で WHILE 文・FOR 文が出現した回数
- WHILE 文終了対応部分 … 関数名_whileend 関数内で WHILE 文・FOR 文が出現した回数

5.2.3 FOR 文

コード生成については資料と同じだが、ラベル名は以下の規則で付ける。

- FOR 文開始対応部分 … 関数名_whilestart 関数内で WHILE 文・FOR 文が出現した回数
- FOR 文ループ毎処理対応部分[5] … 関数名_whilemid 関数内で WHILE 文・FOR 文が出現した回数
- FOR 文終了対応部分 … 関数名_whileend 関数内で WHILE 文・FOR 文が出現した回数

5.2.4 continue、break

continue 時には、対応ループの mid ラベルにジャンプする。break 時には、対応ループの end ラベルにジャンプする。

5.3 最適化

コード生成後、以下の最適化を行う。生成したコードは、まず文字列として変数に保存する。その後、文字列を走査して以下の条件に当てはまる部分が存在すれば、最適化操作を行う。最適化後に新たな最適化可能部分が生成される可能性もあるため、最適化が行われなくなるまで走査を繰り返す。

[5] continue 時に使用する。

5.3.1 無意味な無条件ジャンプ

たとえば、以下のようなアセンブリコードでは jmp 命令は不必要であるので、削除する。

```
    jmp L1
L1:
```

このような無意味な無条件ジャンプは、return 文生成時によく現れる。

5.3.2 無意味なラベル

コード生成時には、IF 文に else が存在しない場合でも else ラベルを生成する。すなわち、IF 文に else が存在しない場合は以下のようなコードが生成される。

```
    je f_else0
;   ### 条件式が真 ###
    jmp f_if0
f_else0:
f_if0:
```

これは、以下のようなコードに書き換えられる。

```
    je f_if0
;   ### 条件式が真 ###
    jmp f_if0
f_if0:
```

このように、不必要なラベルを消去することが出来る。

5.3.3 実行されない命令

無条件ジャンプからラベルまでの処理は、実行されることはない。

```
    jmp L1
    add eax, ebx
    sub ebx, eax
L:
```

このようなコードでは、加算命令 add と減算命令 sub は実行されることはない。よって、この部分を削除して

```
    jmp L1
L:
```

と書き換えられる。

5.3.4 関数定義時の esp 操作

まだ作ってないよ。

5.4 実行例と結果

2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後に意味解析を行い、最適化の後コード生成を行った結果を示す。なお、実行するプログラムは共通して以下のものである。

```
1 #include <stdio.h>
2 main(){
3   int i;
4   for(i=0; i<10; i++)  printf("%d\n", f(i));
5 }
```

5.5 例1

```
1 // n の階乗を計算する
2 int f(int n){
3     if(n <= 0)  return 1;
4     else        return n * f(n-1);
5 }
```

出力アセンブリコード

```
    GLOBAL  f
f:
    push    ebp
```

```
    mov ebp, esp
    sub esp, 0
    mov dword eax, 0
    push    eax
    mov eax, [ebp+8]
    pop ebx
    cmp eax, ebx
    setle   al
    movzx   eax, al
    cmp eax, 0
    je  f_else0
    mov eax, 1
    jmp f_ret
f_else0:
    mov dword eax, 1
    push    eax
    mov eax, [ebp+8]
    pop ebx
    sub eax, ebx
    push    eax
    call    f
    pop ebx
    push    eax
    mov eax, [ebp+8]
    pop ebx
    imul    eax, ebx
    jmp f_ret
f_if0:
f_ret:
    mov esp, ebp
    pop ebp
    ret
```

結果

```
1
```

```
1
2
6
24
120
720
5040
40320
362880
```

5.6 例2

```
1  /*
2   1 から n までの和を求める
3  */
4  int f(int n){
5      int i, ans;
6      ans = 0;
7      for(i=1; i<=n; i++){
8          ans += i;
9      }
10     return ans;
11 }
```

出力アセンブリコード

```
    GLOBAL  f
f:
    push    ebp
    mov ebp, esp
    sub esp, 8
    mov dword [ebp-8], 0
    mov dword [ebp-4], 1
f_whilestart0:
    mov eax, [ebp+8]
    push    eax
    mov eax, [ebp-4]
    pop ebx
```

```
    cmp eax, ebx
    setle   al
    movzx   eax, al
    cmp eax, 0
    je  f_whileend0
    mov eax, [ebp-4]
    push    eax
    mov eax, [ebp-8]
    pop ebx
    add eax, ebx
    mov [ebp-8], eax
f_whilemid0:
    mov dword eax, 1
    push    eax
    mov eax, [ebp-4]
    pop ebx
    add eax, ebx
    mov [ebp-4], eax
    jmp f_whilestart0
f_whileend0:
    mov eax, [ebp-8]
f_ret:
    mov esp, ebp
    pop ebp
    ret
```

結果

```
0
1
3
6
10
15
21
28
```

36
45

6　感想

コード生成部分に非常に時間がかかった。バグが発生した場合、アセンブリコードのどの部分がおかしいか特定するのが非常に難しかった。拡張は文法規則から変更するのでいろいろと考えるべきことが多かったが、楽しかった。最適化は、様々な手法が考えられるが、結局キリが無いような気がした。今回の実験を通して、コンパイラが行っていることを理解できた。