情報実験第 4 コンパイラ 課題 1

池田 光
学籍番号:14_0095
ログイン名:j40095
メールアドレス:ikeda.h.am@m.titech.ac.jp

2016 年 10 月 24 日

1　目的

以下をコンパイルできるように，コンパイラを拡張する．

1. int 型の大域変数
2. 代入式 (=式)
3. while 文
4. < 式
5. + 式
6. -式

これらを実装することで，簡単な計算を行うプログラムの実行が可能になるとともに，繰り返し処理をするプログラムの実行ができるようになる．

2　変更点

変更はすべて codegen.c ファイルにて行った．

2.1　共通の変更点

2.1.1　関数の宣言について

codegen_global 関数では,codegen_fnction_definition 関数が呼ばれ, この関数内ではスタックフレームを作成するアセンブリコードを生成した後, 構文木の各ノードの属性にあったアセンブリコードを生成するため, 各ノードの属性による条件分岐を行いアセンブリコードを生成する関数を呼ぶ visit_AST 関数が呼ばれる．そのため, 本課題において,visit_AST 関数内に代入式,while 文,< 式,+ 式,-式のアセンブリコードを生成するための条件分岐及び関数を加えるとともに, プログラムの先頭に新たに追加した関数の定義をおこなう. これにより, 各ノードの属性に対応したアセンブリコードを出力する関数を visit_AST 関数内で呼び出しが可能となる．以下に変更箇所のコードを示す．

ソースコード 1　visit_AST 関数内

```
1  else if (!strcmp (ast->ast_type, "AST_statement_while")) { //while 文
2    codegen_statement_while (ast);
3  } else if (!strcmp (ast->ast_type, "AST_expression_assign")) { //代入文 (=)
4    codegen_expression_assign (ast);
5  } else if (!strcmp (ast->ast_type, "AST_expression_less")) { //  <式
6    codegen_expression_less (ast);
7  } else if (!strcmp (ast->ast_type, "AST_expression_add")) { // + 式
8    codegen_expression_add (ast);
9  } else if (!strcmp (ast->ast_type, "AST_expression_sub")) { // -式
10   codegen_expression_sub (ast);
11 }
```

ソースコード 2　関数の宣言

```
1  static void codegen_statement_while(struct AST *ast);  //while 文用関数の宣言
2  static void codegen_expression_assign(struct AST *ast); //代入文 (=) 用関数の宣言
3  static void codegen_expression_less(struct AST *ast);   //<式用関数の宣言
4  static void codegen_expression_add(struct AST *ast);    //+ 式用関数の宣言
```

```
5 static void codegen_expression_sub(struct AST *ast);    //-式用関数の宣言
```

2.1.2 スタックフレームについて

Mac OS X のバイナリ形式は関数呼び出しの際, スタックポインタが 16 バイト境界を指すよう要求しているため, 関数呼び出し時に 16 バイト境界を指すように padding をいれる必要がある. 本実験では,codegen_expression_funcall 関数内における padding の計算において大域変数 frame_height を用いて計算をおこなっている. そのため, 各関数内においてスタックから値を取り出したとき (pop を行ったとき) は frame_heigh から 4 を減算し, スタックへ値を積んだとき (push を行うとき) は frame_height に 4 を加算する.

2.2 int 型の大域変数の実装

codegen_global 関数で変更を行った．この関数は global な関数の宣言を行っている関数で，変更前は構文木のなかの global 関数しかアセンブリコードを生成しなかったので，この中に，大域変数の宣言の記述を加えることで，大域変数についてもアセンブリコードを生成出来るようにした．具体的な方法については，global 関数の宣言と同様に，記号表から，属性が TYPE_KIND_PRIM であるものに関して大域変数と処理し，その変数名を取得し,

.comm _変数名,4,2

にて宣言をした．本課題では, 大域変数は int 型なので,4 バイト確保する．以下にコードを示す．

ソースコード 3　大域変数の宣言

```
1 sym = sym_table.global;
2 while (sym != NULL) {
3   if (sym->type->kind == TYPE_KIND_PRIM) {
4     emit_code(sym->ast,"\t.comm\t_%s,4,2\n",sym->name);
5   };
6   sym = sym->next;
7 }
```

2.3 代入式 (=) について

代入文のアセンブリコードの生成は codegen_expression_assign 関数にて行う.
はじめに, 代入文の全体の流れを示す. 右辺値の値をスタックにつみ, 左辺値つまり代入先のアドレスをスタックに積む. 代入先のアドレスを %eax に代入し, スタックトップの値を %ecx に代入した後,%ecx の値を %eax に代入する. ここで, 右辺値とは右辺式を処理した返り値であることで, 右辺式が計算式や代入式の場合も左辺値に代入することを可能にしている.
右辺値の値からスタックに積む必要があるので, 子ノードを逆順にたどる. このとき,0 番目の子であれば左辺値として処理し,1 番目であれば右辺値として処理する. これは, 記号表における属が TYPE_KIND_PRIM なので,codegen_expression_id 関数内の TYPE_KIND_PRIM 内で条件分岐により処理を行った. 親ノードの属性が=, つまり AST_expression_assign であり, 自分が 0 番目であれば左辺値であるので変数のアドレスをスタックに積み, それ以外であれば, 変数の値をスタックに積むアセンブリコードを生成するよう変更を行った.
以上により代入文の実装が出来た. コードを以下に示す.

ソースコード 4　TYPE_KIND_PRIM における変更点

```
case TYPE_KIND_PRIM:
  if((!strcmp (ast->parent->ast_type, "AST_expression_assign")) && (ast->nth == 0)){
      emit_code (ast, "\tpushl\t$_%s\t\t#左辺値をスタックに積む\n",id);
      frame_height +=4;
  }else{
    emit_code(ast,"\tpushl\t_%s\n",id);
    frame_height += 4;
  }
  break;
```

ソースコード 5　codegen_expression_assign 関数

```
static void
codegen_expression_assign(struct AST *ast)
{
  int i;
  for (i = ast->num_child-1; i>=0; i--){
    visit_AST(ast->child[i]);
  }
  emit_code(ast,"\tpopl\t%%eax\n");
  frame_height -=4;
  emit_code(ast,"\tmovl\t0(%%esp),%%ecx\n");
  emit_code(ast,"\tmovl\t%%ecx,0(%%eax)\n");
}
```

2.4　while 文

while 文のアセンブリコードの生成は codegen_statement_while 関数にて行う.
はじめに,while の戻り先となるラベルを出力したのち,0 番目の子ノードである式文 (条件文) を処理し, その返り値をスタックに積む. さらにこのスタックトップの値と即値 0 を比べる. ここで, 条件文の真偽を判断し while 文に入るか入らないかを決める. その後,while 文を抜ける場合のジャンプ先のラベルへのジャンプ命令,while 文の処理 (1 番目の子ノード),while 文の先頭へのジャンプ命令,while 文を抜けた場合のラベルの出力の順にアセンブリコードを出力する.
上記のように,while 文を処理するにあたりラベルが必要となるが, 一つのプログラム中に同一のラベルを複数作成することはできないので,global 変数 label にて処理を行う.(宣言は codegen.c/44 行目) while_label に現在の label を代入し,codegen_statement_while 関数内では while_label を用いる. また while 文では while ループの中と外の 2 つのラベルが必要なので,global 変数の label に 2 を加える. このような処理をラベルを使用する全ての条件文に行うことで, 同一のラベルを使用することを防ぐともに, 複数の条件文をしようしたプログラムに対応することが出来る.
以上により while 文の実装が完了した. この実装により, 繰り返し処理を含むプログラムのコンパイルが可能となる. コードを以下に示す.

ソースコード 6　codegen_statement_while 関数

```
static void
codegen_statement_while(struct AST *ast)
{
  int i;
  int while_label = label;
  label += 2; //whileroop の中と外用の 2 つを確保
  emit_code(ast,"L%d:\t\t\t\t#while の戻り先\n",while_label);
  visit_AST(ast->child[0]);
  emit_code(ast,"\tpopl\t%%eax\n");
  frame_height -= 4;
```

```
  emit_code(ast,"\tcmpl\t$0,%%eax\n");
  emit_code(ast,"\tje\tL%d\n",while_label+1);  //roop   out
  visit_AST(ast->child[1]);
  emit_code(ast,"\tjmp\tL%d\n",while_label);  //roop restart
  emit_code(ast,"L%d:\n",while_label+1);
}
```

2.5 ＜式

＜式のアセンブリコードの生成は codegen_expression_less 関数にて行う.
関数内でのアセンブリコード生成処理を順に示す. はじめに, 子ノードを左辺式, 右辺式と順に visit_AST 関数にて処理し, スタックから順に pop し %ecx,%eax に順に格納する.
次に %ecx と %eax を cmpl 命令により比較し,setl を用いて比較結果を %al に保存する.sete 命令は直前の比較演算結果が less の場合 %al を 1 に, そうでない場合 0 にする.
その後,%al の値を %eax に movzbl 命令を用いてコピーをし, スタックに積む. この movzbl 命令は 8bit レジスタ %al の値を 32bit レジスタ %eax にコピーする命令である.
以上により, 式と式の小なり関係の比較を実装することが出来た. この実装により, により while 文や if 文などの条件文などに用いられる＜式のコンパイルが可能となる. 以下にコードを示す.

ソースコード 7　codegen_expression_less 関数

```
static void
codegen_expression_less (struct AST *ast)
{
  int i;
  for (i=0; i < ast->num_child; i++){
    visit_AST (ast->child[i]);
  }
  emit_code(ast,"\tpopl\t%%ecx\n");
  frame_height -= 4;
  emit_code(ast,"\tpopl\t%%eax\n");
  frame_height -= 4;
  emit_code(ast,"\tcmpl\t%%ecx,%%eax\n");
  emit_code(ast,"\tsetl\t%%al\t\t# <式\n");
  emit_code(ast,"\tmovzbl\t%%al,%%eax\n");
  emit_code(ast,"\tpushl\t%%eax\n");
  frame_height += 4;
}
```

2.6 ＋式

＋式のアセンブリコードの生成は codegen_expression_add 関数にて行う.
この関数内でのアセンブリコード生成処理を順に示す. はじめに, 左辺式と右辺式を順に visit_AST 関数にて処理をし, それぞれの値をスタックに積んだ後, スタックから順に pop し %ecx,%eax に格納する. 次に,addl 命令を用いて %ecx と %eax の足し算を行い, 結果が格納された %eax をスタックに積む.
以上により, 足し算についてコンパイルすることが可能となった. 以下にコードを示す.

ソースコード 8　codegen_expression_add 関数

```
static void codegen_expression_add (struct AST *ast){
  int i;
  for (i=0; i < ast->num_child; i++){
    visit_AST (ast->child[i]);
  }
```

```
  emit_code(ast,"\tpopl\t%%ecx\n");
  frame_height -= 4;
  emit_code(ast,"\tpopl\t%%eax\n");
  frame_height -= 4;
  emit_code(ast,"\taddl\t%%ecx,%%eax\t# 足し算\n");
  emit_code(ast,"\tpushl\t%%eax\n");
}
```

2.7 -式

-式のアセンブリコードの生成は codegen_expression_sub 関数にて行う.
この関数内でのアセンブリコード生成処理を順に示す. はじめに, 左辺式と右辺式を順に visit_AST 関数にて処理をし, それぞれの値をスタックに積んだ後, スタックから順に pop し %ecx,%eax に格納する. 次に,subl 命令を用いて %ecx と %eax の引き算を行い, 結果が格納された %eax をスタックに積む.
以上により, 引き算についてコンパイルすることが可能となった. 以下にコードを示す.

ソースコード 9　codegen_expression_sub 関数

```
static void codegen_expression_sub (struct AST *ast){
  int i;
  for (i=0; i < ast->num_child; i++){
    visit_AST (ast->child[i]);
  }
  emit_code(ast,"\tpopl\t%%ecx\n");
  frame_height -= 4;
  emit_code(ast,"\tpopl\t%%eax\n");
  frame_height -= 4;
  emit_code(ast,"\tsubl\t%%ecx,%%eax\t# 引き算\n");
  emit_code(ast,"\tpushl\t%%eax\n");
}
```

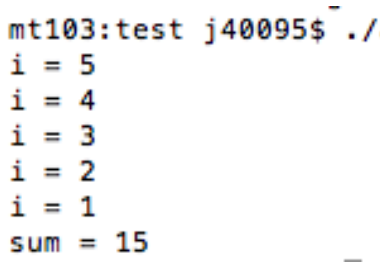
3 例プログラムとその実行結果

3.1 例プログラム

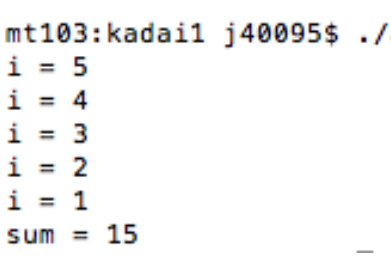
```
int printf ();
int i;
int sum;

int main()
{
  i = 5;
  sum = 0;
  while (0 < i) {
    printf ("i = %d\n", i);
    sum = sum + i;
    i = i - 1;
  }
  printf ("sum = %d\n", sum);
}
```

3.2 実行結果

以下の通り,gcc によるコンパイル結果と, 本課題によるコンパイル結果は一致した.



gcc におけるコンパイル結果



本課題によるコンパイル結果

図 1　実行結果

4　評価

本課題のすべての拡張について変更を加え, 例プログラムの結果も正しいものとなったので, 課題を十分達成出来たと考える.

5　考察

本課題では, 変数について大域変数のみの拡張であったため,codegen_expression_id 内において条件分岐を用いて左辺値と右辺値を別に処理をしたが, 今後, ローカル変数やポインタを実装する際, すべての場合で条件分岐をすると煩雑になるため, 右辺値用と左辺値用で関数を別で考えたほうが今後の拡張にむけて展望があると感じた.

2 項の小なり比較を行う setl 命令を調べている際に,2 項のイコール比較を行う sete 命令,2 項の大なり比較を行う setg 命令についても知識を得られたので, 今後の拡張のために記憶をしておきたいと思う.

6　まとめ

コードの理解に時間がかかったが, 前期のコンパイラ構成の授業と合わせ, 本課題によってコンパイラの動きをより理解できた.

MieruCompiler により, どこのコードがどのアセンブリを生成しているのかが視覚的に分かりやすかったので課題をこなすにおいてとても助かったが, その分理解が後回しになってしまっていた所があった. 以降の課題については, まず自分の頭で考え, 実行して正しく挙動しなかったときに MieruCompiler を使用したいと思う.