情報実験第 4 コンパイラ 課題 2

池田光
学籍番号:14_00954
ログイン名:j40095
メールアドレス:ikeda.h.am@m.titech.ac.jp

2016 年 11 月 7 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 型の局所変数
2. int 型の関数引数
3. if 文
4. if-else 文
5. return
6. ==式
7. || 式
8. &&式
9. *式
10. /式

これらの実装により,int 型の関数を含んだプログラムや,if 文を含んだプログラム, 四則演算を含んだプログラム (加算, 減算は課題 1 にて実装済み), またはこれらを組み合わせたプログラムがコンパイル可能となる.

2 変更点

全ての変更は codegen.h にて行った.

2.1 visit_AST 関数について

課題 1 の考察にもあげたが, 右辺値と左辺値で取り扱い方が異なるため,1 つの visit_AST で扱うと分岐が煩雑になってしまうので, 右辺値用に visit_AST_right 関数, 左辺値用に visit_AST_left 関数を用意し, 特別な扱いが必要な左辺値のみ visit_AST_left 関数で扱い, その他を visit_AST_right で扱うことでこの問題の解決とした. また, この変更により, 子から親の属性を見る必要がなくなるため実行速度の向上が期待される.

2.2 共通の変更点

2.2.1 関数の宣言について

課題 1 と同様,visit_AST_right 関数内に代入式,if 文,if-else 文,return 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_right 関数内で呼び出しが可能となる. 以下に変更箇所を示す.

ソースコード 1 visit_AST_right 関数

```
1   else if (!strcmp (ast->ast_type, "AST_expression_eq")){   // == 式
2     codegen_expression_eq (ast);
3   } else if (!strcmp (ast->ast_type, "AST_expression_lor")){  // ||式
4     codegen_expression_lor (ast);
5   } else if (!strcmp (ast->ast_type, "AST_expression_land")){  // &&式
6     codegen_expression_land (ast);
```

```
} else if (!strcmp (ast->ast_type, "AST_expression_mul")){  // *式
  codegen_expression_mul (ast);
} else if (!strcmp (ast->ast_type, "AST_expression_div")){  // /式
  codegen_expression_div (ast);
} else if (!strcmp (ast->ast_type, "AST_statement_if")){   //if 文
  codegen_statement_if (ast);
} else if (!strcmp (ast->ast_type, "AST_statement_ifelse")){ // ifelse 文
  codegen_statement_ifelse (ast);
} else if (!strcmp (ast->ast_type, "AST_statement_return")){ // return 文
  codegen_statement_return (ast);
}
```

ソースコード 2　関数の宣言

```
static void codegen_expression_eq(struct AST *ast);     //==式用関数の宣言
static void codegen_expression_lor(struct AST *ast);     //||式用関数の宣言
static void codegen_expression_land(struct AST *ast);    //&&式用関数の宣言
static void codegen_expression_mul(struct AST *ast);    //*式用関数の宣言
static void codegen_expression_div(struct AST *ast);    ///式用関数の宣言
static void codegen_statement_if(struct AST *ast);     //if 文
static void codegen_statement_ifelse(struct AST *ast); //ifelse 文
static void codegen_statement_return(struct AST *ast); //return 文
```

2.2.2　スタックフレームについて

課題 1 のときと同様に, 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. この時, 関数に入ったときと関数から出るときで frame_height を同じにしなければならない. 詳細は後述する.

2.3　int 型の局所変数の実装

codegen_function_definition 関数にて定義を行う. AST.h 内 22 行目にて, 局所変数の合計サイズとして,total_local_size という変数が用意されているので, こちらを使用する. 具体的には,total_local_size が 0 でない場合は領域の確保が必要となるため, この変数分 %esp を減らすことで自動変数用の領域を確保するするアセンブリコードを出力するとともに, スタックフレームの高さをこの領域分増やすことで, 局所変数の定義が完了する.
以下に局所変数の定義部分のコードを示す.

ソースコード 3　局所変数の定義

```
% if(ast->u.func.total_local_size != 0){
%   emit_code (ast,"\tsubl\t$%d,%%esp\t#自動変数用\n",ast->u.func.total_local_size);
%   frame_height += ast->u.func.total_local_size;
% }
```

局所変数は symbol.h 内 15 行目において, 名前空間が NS_Local であることがわかる. したがって, 左辺値用と右辺値用の codegen_expression_id_left 関数と codegen_expression_id_right 関数のそれぞれにおいて名前空間のける分岐の NS_Local 部にて変更を加える.

codegen_expression_id_right についてはスタックに値をつむだけでよい. このとき, 局所変数がベースポインタからどれくらい離れいるかを Symbol 構造体の offset を用いて計算する. 局所変数は int 型なので,4 バイトずつ積まれし,push を行うアセンブリコードを出力する.
以下に変更点を示す.

ソースコード 4　codegen_expression_id_right 関数内における局所変数

```
1  case NS_LOCAL:  //局所変数
2    emit_code (ast,"\tpushl\t%d(%%ebp)\t\t#%s の内容をスタックに積む\n",symbol->offset*(-1)-4,id);
3    frame_height += 4;
4    break;
```

codegen_expression_id_left については, 左辺値なのでアドレスをスタックに積む必要がある. よって,leal 命令によって使用する局所変数が格納されているアドレスを %eax に格納し, その後,%eax の値をスタックに積む. この時,codegen_expression_id_right 関数と同様に offset を用いて局所変数とベースポインターとの距離を計る. 以上の過程のアセンブリコードを出力する.

以下に変更点を示す.

ソースコード 5　codegen_expression_id_left 関数内における局所変数

```
1  case NS_LOCAL:  //局所変数
2    emit_code (ast,"\tleal\t%d(%%ebp),%%eax\t#左辺値\n",symbol->offset*(-1)-4);
3    emit_code (ast,"\tpushl\t%%eax\n");
4    frame_height += 4;
5    break;
```

2.4　int 型の関数引数の実装

関数呼び出し時において,16 バイト境界条件を満たすように padding を計算し積む必要があるが, これは codegen_expression_funcall 関数内にて実装されている. この関数では必要な padding を積んだのち, 子ノード に対し visit_AST_right 関数を呼ぶことで引数の値を積み関数を呼び出している. 最後に引数と padding を捨て関数を終わる. したがって, 子ノードに対して visit_AST_right 関数が呼ばれ, 関数引数に対する属性は AST_expression_id なので codegen_expression_id_right 関数と codegen_expression_id_left 関数内にて変更を行う. さらに, 関数引数は symbol.h 内 16 行目において, 名前空間が NS_ARG であることがわかる. したがって,codegen_expression_id_right 関数と codegen_expression_id_left 関数のそれぞれにおいて名前空間における分岐の NS_ARG 部にて変更を行う.

2 つともおおよその変更は局所変数の際と同じであるが, 関数引数は int 型であるので, ベースポインターから戻り番地を挟んで 4 バイトずつ足された方向に増えるので関数引数とベースポインタとの距離を計算する部分のみ局所変数と異なる.

以下に変更点を示す.

ソースコード 6　codegen_expression_id_right 関数内における関数引数

```
1  case NS_ARG:    //引数
2    emit_code (ast,"\tpushl\t%d(%%ebp)\t\t#%s の内容をスタックに積む\n",symbol->offset+8,id);
3    frame_height += 4;
4    break;
```

ソースコード 7　codegen_expression_id_left 関数内における関数引数

```
1  case NS_ARG:    //引数
2    emit_code (ast, "\tleal \t%d(%%ebp), %%eax\t#左辺値\n",symbol->offset + 8);
3    emit_code (ast, "\tpushl \t%%eax\n");
4    frame_height += 4;
5    break;
```

2.5 if 文について

if 文のアセンブリコードの生成は codegen_statement_if 関数にて行う.

はじめに,0 番目の子ノードについて visit_AST_right 関数を呼び出すことで, 条件式のアセンブリコードを出力する. 条件式の計算結果と 0 を比べることで, 条件式の真偽を判定する. 次に, 条件式が偽であった場合のジャンプ命令を出力し, ジャンプしなかった (条件式が真であった) 場合の if 文内の処理 (1 番目の子ノード) を visit_AST_right 関数を呼ぶことでアセンブリコードを出力する. 最後に, 条件式が偽であった場合のジャンプ先のラベルを出力する. 以上により,if 文のアセンブリコードを出力することが出来,if 文を含むプログラムのコンパイルが可能となる.

ラベルに関しては,if 文の中と外を分ける必要があり, 既存のラベル以外に一つ必要であり,if 文内で label_if に現在の label を代入し,label をインクリメントすることで条件を満たす.
以下に変更点を示す.

ソースコード 8　codegen_statement_if 関数

```
% static void
% codegen_statement_if (struct AST *ast)
% {
%   visit_AST_right(ast->child[0]);
%   emit_code(ast, "\tpopl\t%%eax\n");
%   frame_height -= 4;
%   emit_code(ast, "\tcmpl\t$0,%%eax\n");
%   int label_if = label;
%   label += 1;
%   emit_code(ast,"\tje\tL%d\n",label_if);
%   visit_AST_right(ast->child[1]);
%   emit_code(ast,"L%d:\n",label_if);
% }
```

2.6 if-else 文について

if-else 文のアセンブリコードの生成は codegen_statement_ifelse 関数で行う.

はじめに,0 番目の子ノードについて visit_AST_right 関数を呼び出すことで, 条件式のアセンブリコードを出力する. 条件式の計算結果と 0 を比べることで, 条件式の真偽を判定し, 条件式が偽であった場合のラベルへのジャンプ命令を出力する. 子ノードの 1 番目について visit_AST_right 関数を呼び出すことで if 文の中の処理に対するアセンブリコードを出力する. 次に,if-else 文の出口への無条件ジャンプ命令を出力する. 条件式が偽であった場合のラベルを出力し,2 番目の子ノードについて visit_AST_right 関数を呼び出すことで else 文内の処理に対するアセンブリコードを出力する. 最後に if-else 文のを出た時のラベルに対するアセンブリコードを出力する. 以上により,if-else 文のアセンブリコードを出力することが出来,if-else 文を含むプログラムのコンパイルが可能となる.

ラベルに関しては,if 文内・else 文内・if-else の出口の 3 つ必要だが, 既存のラベル以外に 2 つ必要であり,if-else 文内で label_ifelse に現在の label を代入し,label に 2 を加えることで条件を満たす.
以下に変更点を示す.

ソースコード 9　codegen_statement_ifelse

```
% static void
% codegen_statement_ifelse (struct AST *ast)
% {
```

```
4   %    visit_AST_right(ast->child[0]);
5   %    emit_code(ast,"\tpopl\t%%eax\n");
6   %    frame_height -= 4;
7   %    emit_code(ast,"\tcmpl\t$0,%%eax\n");
8   %    int label_ifelse = label;
9   %    label += 2;
10  %    emit_code(ast,"\tje\tL%d\n",label_ifelse);
11  %    visit_AST_right(ast->child[1]);
12  %    emit_code(ast,"\tjmp\tL%d\n",label_ifelse+1);
13  %    emit_code(ast,"L%d:\t\t\t\t\t#else のとき\n",label_ifelse);
14  %    visit_AST_right(ast->child[2]);
15  %    emit_code(ast,"L%d:\t\t\t\t\t#ifelse の出口\n",label_ifelse+1);
16  % }
```

2.7 return 文について

return 文のアセンブリコード生成は codegen_statement_return 関数にて行う.

return 文では返り値を伴うものと伴わないものがあるので,0 番目の子ノードの属性によって場合分けをする. 属性が AST_expression_opt_single のとき, 返り値を伴うので,visit_AST_right 関数にて子ノードのアセンブリコードを出力し,%eax を pop する.

最後に, 関数から抜け出すための無条件ジャンプ命令を出力する. 以上により,return 文に対するアセンブリコードの生成が可能となる.

以下に変更点を示す.

ソースコード 10　codegen_statement_return 関数

```
1   static void
2   codegen_statement_return (struct AST *ast)
3   {
4     if(!strcmp(ast->child[0]->ast_type,"AST_expression_opt_single")){
5       visit_AST_right(ast->child[0]);
6       emit_code(ast,"\tpopl\t%%eax\n");
7       frame_height -= 4;
8     }
9     emit_code(ast,"\tjmp\tL.XCC.RE.%s\t#return\n",funcname);
10  }
```

2.8 ==式について

==式のアセンブリコードの生成は codegen_expression_eq 関数で行う.

はじめに, 子ノードを左辺式, 右辺式と順に visit_AST_right 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,sete 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同じ場合 %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. 最後に %eax の値をスタックに積む. 以上のアセンブリコードを出力することで,if 文の条件などで==式を利用したプログラムがコンパイル可能となる.

以下に変更点を示す.

ソースコード 11　codegen_expression_eq 関数

```
1   static void
2   codegen_expression_eq (struct AST *ast)
3   {
```

```
    int i;
    for (i=0; i < ast->num_child; i++){
      visit_AST_right(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,"\tsete\t%%al\t\t# ==式\n");
    emit_code(ast,"\tmovzbl\t%%al,%%eax\n");
    emit_code(ast,"\tpushl\t%%eax\n");
    frame_height += 4;
  }
```

2.9 || 式について

|| 式のアセンブリコードの生成は codegen_expression_lor 関数で行う.

はじめに,0 番目の子ノードを visit_AST_right 関数にてアセンブリコードを出力するとともに左辺値をスタックに積む. この値と 0 とを cmpl 命令にて比較し, 真偽判定を行う. 真であった場合, 右辺値にかかわらず条件式は真になるので, ジャンプ命令を行いスタックに 1 を積む. 偽の場合,1 番目の子ノードを visit_AST_right 関数にてアセンブリコードを出力するとともに右辺値のをスタックに積む. この値と 0 とを cmple 命令にて比較し, 真偽を判定する. 真であった場合, 左辺値が真であったときと同じラベルにジャンプし,1 をスタックに積む. 偽の場合, 左辺値も右辺値も偽となるので, 新しいラベルにジャンプし,0 をスタックに積む.

以上のアセンブリコードを出力することで,or 演算を含むプログラムのコンパイルが可能となる.

|| 式において label を 2 つ使っているので,label に 2 を加える. また, 関数を呼び出したときと終了時ではスタックフレームの高さが同じではないといけないこと, 静的にみると push を 2 回おこなっているので,frame_hight を 8 加える必要があるように見えるが, 実際はいずれかの push が行われるので frame_height は 4 を加えるだけでよい. 以下に変更点を示す.

ソースコード 12　codegen_expression_lor 関数

```
  static void
  codegen_expression_lor (struct AST *ast)
  {
    int label_lor = label;
    label += 2;
    visit_AST_right(ast->child[0]);
    emit_code(ast,"\tpopl\t%%eax\n");
    frame_height -= 4;
    emit_code(ast,"\tcmpl\t$0,%%eax\n");
    emit_code(ast,"\tjne\tL%d\n",label_lor);
    visit_AST_right(ast->child[1]);
    emit_code(ast,"\tpopl\t%%eax\n");
    frame_height -= 4;
    emit_code(ast, "\tcmpl \t$0, %%eax\n");
    emit_code(ast, "\tjne \tL%d\n",label_lor);
    emit_code(ast, "\tpushl \t$0\n");
    // frame_height +=4;
    emit_code(ast, "\tjmp \tL%d\n",label_lor + 1);
    emit_code(ast, "L%d:\t\t\t\t\t# || in\n",label_lor);
    emit_code(ast, "\tpushl \t$1\n");
    frame_height += 4;
    emit_code(ast, "L%d:\t\t\t\t\t# || out\n",label_lor + 1);
  }
```

2.10 &&式について

&&式のアセンブリコードの生成は codegen_expression_land 関数で行う.

はじめに,0 番目の子ノードを visit_AST_right 関数にてアセンブリコードを出力するとともに左辺値をスタックに積む. この値と 0 とを cmpl 命令にて比較し, 真偽判定を行う. 偽であった場合, 右辺値にかかわらず条件式は偽になるので, ジャンプ命令を行いスタックに 0 を積む. 真の場合,1 番目の子ノードを visit_AST_right 関数にてアセンブリコードを出力するとともに右辺値のをスタックに積む. この値と 0 とを cmple 命令にて比較し, 真偽を判定する. 偽であった場合, 左辺値が偽であったときと同じラベルにジャンプし,0 をスタックに積む. 真の場合, 左辺値も右辺値も真となるので, 新しいラベルにジャンプし,1 をスタックに積む.

以上のアセンブリコードを出力することで,and 演算を含むプログラムのコンパイルが可能となる.

&&式において label を 2 つ使っているので,label に 2 を加える. また, 関数を呼び出したときと終了時ではスタックフレームの高さが同じではないといけないこと, 静的にみると push を 2 回おこなっているので,frame_hight を 8 加える必要があるように見えるが, 実際はいずれかの push が行われるので frame_height は 4 を加えるだけでよい. 以下に変更点を示す.

ソースコード 13　codegen_expression_land

```
1  static void
2  codegen_expression_land (struct AST *ast)
3  {
4    int label_land = label;
5    label += 2;   //条件内と外の 2 つ
6    visit_AST_right(ast->child[0]);
7    emit_code(ast,"\tpopl\t%%eax\n");
8    frame_height -= 4;
9    emit_code(ast,"\tcmpl\t$1,%%eax\n");
10   emit_code(ast,"\tjne\tL%d\n",label_land);
11   visit_AST_right(ast->child[1]);
12   emit_code(ast,"\tpopl\t%%eax\n");
13   frame_height -= 4;
14   emit_code(ast,"\tcmpl\t$1,%%eax\n");
15   emit_code(ast,"\tjne\tL%d\n",label_land);
16   emit_code(ast,"\tpushl\t$1\n");
17   // frame_height += 4;
18   emit_code(ast,"\tjmp\tL%d\n",label_land+1);
19   emit_code(ast,"L%d:\t\t\t\t\t# && in\n",label_land);
20   emit_code(ast,"\tpushl\t$0\n");
21   frame_height += 4;
22   emit_code(ast,"L%d:\t\t\t\t\t# && out\n",label_land+1);
23 }
```

2.11 *式について

*式のアセンブリコードの生成は codegen_expression_mul 関数にて行う.

はじめに, 左辺式と右辺式を順に visit_AST_right 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,imull 命令を用いて %ecx と %eax の掛け算を行い, 結果が格納された %eax をスタックに積む.

以上のアセンブリコードを出力することで, 掛け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 14　codegen_expression_mul

```
1   static void
2   codegen_expression_mul(struct AST *ast)
3   {
4     int i;
5     for(i=0; i < ast->num_child; i++){
6       visit_AST_right(ast->child[i]);
7     }
8     emit_code(ast,"\tpopl\t%%ecx\n");
9     frame_height -= 4;
10    emit_code(ast,"\tpopl\t%%eax\n");
11    frame_height -= 4;
12    emit_code(ast,"\timull\t%%ecx,%%eax\t# 掛け算\n");
13    emit_code(ast,"\tpushl\t%%eax\n");
14    frame_height += 4;
15  }
```

2.12　/式について

/式のアセンブリコードの生成は codegen_expression_div 関数にて行う.

はじめに, 左辺式と右辺式を順に visit_AST_right 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,cltd 命令により符号拡張した後,idivl 令を用いて %ecx と %eax の掛け算を行い, 結果が格納された %eax をスタックに積む.

以上のアセンブリコードを出力することで, 割り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 15　codegen_expression_div

```
1   static void
2   codegen_expression_div (struct AST *ast)
3   {
4     int i;
5     for(i=0; i < ast->num_child; i++){
6       visit_AST_right(ast->child[i]);
7     }
8     emit_code(ast,"\tpopl\t%%ecx\n");
9     frame_height -= 4;
10    emit_code(ast,"\tpopl\t%%eax\n");
11    frame_height -= 4;
12    emit_code(ast,"\tcltd\n");
13    emit_code(ast,"\tidivl\t%%ecx,%%eax\t# 割り算\n");
14    emit_code(ast,"\tpushl\t%%eax\n");
15    frame_height += 4;
16  }
```

3　例プログラムと実行結果

3.1　例プログラムと自作プログラム

ソースコード 16　例プログラム

```
1  int printf ();
2
3  int mrn (int n)
4  {
5      if (n < 0 || n == 0)
6          return 0;
7      else
```

```
        return 10 * mrn (n - 1) + n;
}

int main ()
{
  int i;

  i = 0;
  while (i < 11) {
    printf ("mrn(%d) = %d\n", i, mrn(i));
    i = i+1;
  }
}
```

ソースコード 17　自作プログラム

```
int printf ();

int mrn (int n)
{
    if (0 < n && n < 10)
        return 120/n;
    else
        return n;
}

int main ()
{
  int i;

  i = 0;
  while (i < 15) {
    printf ("mrn(%d) = %d\n", i, mrn(i));
    i = i+2;
  }
}
```

3.2　例プログラムの実行結果

```
mt104:test j40095$ gcc kadai2.c
kadai2.c:1: warning: conflicting types for built-in function 'printf'
mt104:test j40095$ ./a.out
mrn(0) = 0
mrn(1) = 1
mrn(2) = 12
mrn(3) = 123
mrn(4) = 1234
mrn(5) = 12345
mrn(6) = 123456
mrn(7) = 1234567
mrn(8) = 12345678
mrn(9) = 123456789
mrn(10) = 1234567900
```

図 1　gcc での実行結果

図 2　xcc での実行結果

3.3 自作プログラムの実行結果

```
mt104:test j40095$ gcc kadai2-2.c
kadai2-2.c:1: warning: conflicting
mt104:test j40095$ ./a.out
mrn(0) = 0
mrn(2) = 60
mrn(4) = 30
mrn(6) = 20
mrn(8) = 15
mrn(10) = 10
mrn(12) = 12
mrn(14) = 14
```

図 3　gcc での実行結果

```
mt104:kadai2 j40095$ gcc -m32 kadai2-2.s
ld: warning: PIE disabled. Absolute addres
in _main from /var/folders/jr/21c2lh8s01g
 -mdynamic-no-pic or link with -Wl,-no_pi
mt104:kadai2 j40095$ ./a.out
mrn(0) = 0
mrn(2) = 60
mrn(4) = 30
mrn(6) = 20
mrn(8) = 15
mrn(10) = 10
mrn(12) = 12
mrn(14) = 14
```

図 4　xcc での実行結果

4　評価

すべての課題を実装し, 実行結果も正しいものとなったが,MieruCompiler に頼ってしまった部分が多かったのが残念であった.

5　考察

or 演算子と and 演算子において, 実際の動きを考えず push が 2 回されたので 2 回 frame_height を増やしてしまい, 余計な padding を積んでしまっていた. 前回の課題と今回の課題において,push や pop を行った際,frame_height をその都度変えていたが, 必要なときはその都度変更をし, その他の場合は, 関数の最後にまとめて行うことで, 上記のエラーは起こらなくなると考えた.

6　まとめ

今回の課題で, 局所変数や関数引数の処理を行ったことで，スタックフレームの使われ方や構造をより深く理解することが出来た. 原理と実際の挙動を考えてしっかりコードを書くことを忘れずに行っていく必要があると改めて強く感じた.