実験レポート 2

坪井正太郎（101830245）

2021 年 1 月 21 日

1　はじめに

レポート内に記載したソースコードは、LLVM 命令部分のみが違いロジックが同じコードの省略など、意味が分かる範囲で修正が行われている。

1.1　コード規約

ケース

- 型名
  - UpperCamelCase
- 構造体名、メンバ名、変数名
  - lowerCamelCase
- 関数名
  - snake_case

変数のプレフィックス

- グローバル変数:g
  - gVar
- ローカル変数:t
  - tVar
- 仮引数:a
  - aVar

コメント

型、関数の説明はヘッダにのみ記載、c ファイルには実装に関するコメントのみ。プライベート関数は明記して実装にコメント。

2 課題 4

2.1 課題概要

四則演算を行う pl0 プログラムをコンパイルして、実行可能な LLVM IR を生成するプログラムを実装する。

2.2 実装

factor.h と、llvm.h を実装し、parser.y の適切な場所で呼び出した。

2.2.1 factor

factor では、計算途中の変数、定数を記録するための型、関数を定義した。

ソースコード 1 factor.h

```
1  #pragma once
2
3  #include "../symtab/symtab.h"
4
5  // +--------------------------+
6  // | 計算とか遷移のための機能 |
7  // +--------------------------+
8
9  // 変数もしくは定数の型
10 // ラベルとかコメントにも使う
11 typedef struct {
12   char *vname; // 変数の場合の変数名 ラベルはコメントにつけたい名前
13   int val; // 整数の場合はその値,変数の場合は割り当てたレジスタ番号
14   Scope type; // 実態が何なのか
15 } Factor;
16
17 // 変数もしくは定数のためのスタック
```

```
18 typedef struct {
19   Factor *element[100]; // スタック(最大要素数は 100まで)
20   unsigned int top; // スタックのトップの位置
21 } Factorstack;
22
23 // 諸々初期化
24 void fstack_init();
25
26 // スタックに積む
27 // 積むときも返してくれるので､積みつつ保持しておきたいときに便利
28 Factor *factor_push(char *, int, Scope);
29
30 // スタックから出す
31 Factor *factor_pop();
```

各関数は factor.c に実装した。プライベートとして実装にのみ定義したグローバル変数をスタックとして、ポインタ操作で push pop を行っているだけなので、省略する。

2.2.2 llvm

llvm.h では LLVM IR と関数の内部表現、その列に対する操作を定義した。

LLVM IR の内部表現は LLVMcode で定義、命令の種別に加えて、オペランドや返り値は全て Factor へのポインタとして持つ線形リストとして実装した。

それぞれの関数はコード列を持つ線形リストとしている。

ソースコード 2　llvm.h

```
1   /* LLVM 命令名の定義 */
2   typedef enum {Alloca, Global, Load, Store, Add, Sub, Mul, Sdiv,}
        LLVMcommand;
3
4   // 1つのLLVM 命令と次の命令へのリンク
5   // 基本的にFactor でオペランドを定義している
6   typedef struct llvmcode {
7     LLVMcommand command;
8     union {
9       struct {
10        Factor *retval;
11      } alloca;
```

```
!!global は alloca と同じ
      struct {
        Factor *arg1;
        Factor *retval;
      } load;
!!store は load と同じ
      struct {
        Factor *arg1;
        Factor *arg2;
        Factor *retval;
      } add;
!!あとの四則演算はadd と同じ
    } args;
    // 次の命令へのポインタ
    struct llvmcode *next;
  } LLVMcode;

  // 初期化処理
  void code_init();

  // 命令とオペランド、格納先とかを指定して、llvm 命令を作る
  LLVMcode *code_create(LLVMcommand, Factor *, Factor *, Factor *);

  // ↑↓を分けたのは、制御文とかでとりあえず作るけど追加したくないときを
      想定

  // スタックに 1つのllvm 命令を追加する
  void code_add(LLVMcode *);

  // 全てのコードをファイルに出力する
  // 最後に呼び出す
  void print_LLVM_code();

  // ~~~~~~~~~~ここから関数の表現定義~~~~~~~~~~

  /* LLVM の関数定義 */
  typedef struct fundecl {
    char fname[256]; // 関数名
```

```
49      unsigned arity; // 引数個数
50      Factor args[10]; // 引数名
51      LLVMcode *codes; // 命令列の線形リストへのポインタ
52      struct fundecl *next; // 次の関数定義へのポインタ
53    } Fundecl;
54
55    // 初期化処理
56    void fundecl_init();
57
58    // 関数を追加して､その関数に対するコード追加を開始する
59    void fundecl_add(char *, unsigned);
```

実装は以下のように行った。

命令、関数の追加は、線形リストに対するポインタを外部からは参照できない大域変数として定義し、それに対してポインタの付替えをすることで実現した。

内部表現を出力する関数は、print_code であり、対応する文字列としてファイルに書き込んだ。全体としては、関数列→コード列の順に線形リストを追い、都度出力した。

ソースコード 3　llvm.c

```
1   void print_code(LLVMcode *aCode);
2
3   // 命令列の前後のアドレスを保持するポインタ
4   LLVMcode *gCodehd, *gCodetl;
5
6   // 関数定義の線形リストの前後アドレスを保持するポインタ
7   Fundecl *gDeclhd, *gDecltl;
8
9   FILE *gFile; // 出力先
10
11 !!単純な線形リスト関係の関数は省略
12
13  LLVMcode *code_create(LLVMcommand aCommand, Factor *aArg1, Factor *
        aArg2,
14                        Factor *aRetval) {
15    LLVMcode *tCode = malloc(sizeof(LLVMcode));
16    tCode->command = aCommand;
17    switch (aCommand) {
18      case Add:
```

```
19          tCode->args.add.arg1 = aArg1;
20          tCode->args.add.arg2 = aArg2;
21          tCode->args.add.retval = aRetval;
22          break;
23  !!ほかもunion 型に合うように代入するだけ
24        default:
25          break;
26      }
27      return tCode;
28    }
29
30    void code_add(LLVMcode *aCode) {
31      // aCode の内容は変更される可能性がある
32
33      if (gCodetl == NULL && gDecltl == NULL)
34        fprintf(stderr, "[ERROR] unexpected error\n");
35
36      aCode->next = NULL;
37      if (gCodetl == NULL) // 解析中の関数の最初の命令の場合
38        gCodetl = gCodehd = gDecltl->codes = aCode;
39      else // 解析中の関数の命令列に 1つ以上命令が存在する場合
40        gCodetl = gCodetl->next = aCode;
41    }
42
43    // プライベート関数
44    // local とか、global とか、条件によってその Factor を示す表記が違う
45    void factor_encode(Factor *aFactor, char *aArg) {
46    !!%1や@n のように変換を行う
47    }
48
49    // プライベート関数
50    // 1命令単位で実際のLLVM コードを出力する
51    void print_code(LLVMcode *aCode) {
52      // Factor をいい感じの str 表現にしたいので、エンコしてから利用する
53      char tArg1[256], tArg2[256], tRetval[256];
54
55      switch (aCode->command) {
56        case Add:
```

```
57          factor_encode(aCode->args.add.arg1, tArg1);
58          factor_encode(aCode->args.add.arg2, tArg2);
59          factor_encode(aCode->args.add.retval, tRetval);
60          fprintf(gFile, "\t%s = add nsw i32 %s, %s\n", tRetval, tArg1,
                 tArg2);
61          break;
62  !!LLVM に変換して出力するだけなので省略
63        default:
64          break;
65      }
66    }
67
68    void print_LLVM_code() {
69  !!単純な 2重ループ
70    };
```

2.2.3 parser.y

まず、大域変数の定義は、最初の関数列を使用して行った。構文要素 id_list で、現在のスコープが GLOBAL_VAR と LOCAL_VAR の場合で分けてコードを追加した。大域変数の場合、outblock 前で__GlobalDecl という架空の関数を作られ、その中に global 命令列が追加されていく。

main となる手続きは subprog_decl_part が読み切られてから関数列に追加した。

四則演算を実行するために必要な命令は次のような操作で生成するようにした。

- factor の生成
  定数か変数名が読まれたら、記号表から検索して、factor_push した。特に変数は Load 命令を追加し、その戻り値になる factor を push した。
- 代入文
  計算が終了した時点で、2 回 pop すると値→代入先の順番に出てくる。これを渡してストア命令を追加した。
- 単項演算
  + 単項演算子については、何も操作を行わっていないが、-単項演算子は定数ゼロに対する減算として定義した。実際に、clang では +-どちらもこれと同じ命令を生

成することを確認した。

- 2 項演算
  factor は 2 回 pop すると右オペランド→左オペランドの順番に出てくる。これをそれぞれ第 2、第 1 引数に渡して命令を追加した。
  四則演算それぞれは命令種が違うだけで、その他の操作は同じである。

factor 生成のとき、レジスタの割付も行った。基本的に、1 つの命令で 1 つのレジスタを使用するために code_create でインクリメントした。手続きごとにレジスタ番号はリセットされ、gRegnum という大域変数で定義した。

ソースコード 4　parser.y

```
1  !!省略
2      program
3              :PROGRAM IDENT SEMICOLON
4              {
5                      gScope = GLOBAL_VAR;
6                      gRegnum = 1;
7
8                      symtab_push($2, 0, gScope);
9
10                     fundecl_add("__GlobalDecl", 0);
11             }
12               outblock PERIOD
13             {
14                     print_LLVM_code();
15             }
16             ;
17
18     outblock
19             : var_decl_part subprog_decl_part
20             {
21                     fundecl_add("main", 0);
22                     gRegnum = 1; // 手続きごとにレジスタ番号はリセットさ
                           れる
23                     factor_push("Func Retval", gRegnum++, LOCAL_VAR);
24                     Factor *tFunRet = factor_pop();
25                     code_add(code_create(Alloca, NULL, NULL, tFunRet));
```

```
                        // 戻り値を先に定義
26              }
27               statement
28              ;
29  !!省略
30      assignment_statement
31              : IDENT ASSIGN expression
32              {
33                      Row *tRow = symtab_lookup($1);
34                      factor_push(tRow->name, tRow->regnum, tRow->type);
35                      Factor *tArg2 = factor_pop();
36                      Factor *tArg1 = factor_pop();
37
38                      code_add(code_create(Store, tArg1, tArg2, NULL));
39              }
40              ;
41  !!省略
42      expression
43              : term
44              | PLUS term
45              | MINUS term
46              {
47                      // 単項演算はゼロからの足し引きで表現。
48                      Factor *tArg2 = factor_pop();
49                      factor_push("", 0, CONSTANT);
50                      Factor *tArg1 = factor_pop();
51                      Factor *tRetval = factor_push("", gRegnum++,
                            LOCAL_VAR);
52                      code_add(code_create(Sub, tArg1, tArg2, tRetval));
53              }
54              | expression PLUS expression
55              {
56                      // 四則演算は、全部これと一緒
57
58                      // オペランドをポップする（順番に注意）
59                      Factor *tArg2 = factor_pop();
60                      Factor *tArg1 = factor_pop();
61
```

```
62                      // 代入先として局所変数を用意、
                            pop しないことで、次のオペランドとしてもそのまま使える
63                      Factor *tRetval = factor_push("", gRegnum++,
                            LOCAL_VAR);
64
65                      // Factor からのコード生成と追加を同時に行う
66                      code_add(code_create(Add, tArg1, tArg2, tRetval));
67              }
68              | expression MINUS expression
69              ;
70  !!省略
71      factor
72              : var_name
73              | NUMBER
74              {
75                      factor_push("const", $1, CONSTANT);
76              }
77              | LPAREN expression RPAREN
78              ;
79
80      var_name
81              : IDENT
82              {
83                      Row* tRow = symtab_lookup($1);
84                      factor_push(tRow->name, tRow->regnum, tRow->type);
85
86                      Factor *tArg1 = factor_pop();
87                      Factor *tRetval = factor_push("", gRegnum++,
                            LOCAL_VAR);
88
89                      code_add(code_create(Load, tArg1, NULL, tRetval));
90              }
91              ;
92  !!省略
93      id_list
94              : IDENT
95              {
96                      // 大域変数と局所変数の場合で処理分けた
```

```
97                      // 局所だと、レジスタ番号をSSA で管理する必要がある
98                      LLVMcommand tCommand;
99                      if(gScope == GLOBAL_VAR){
100                             tCommand = Global;
101                             symtab_push($1, 0, gScope);
102                     } else{
103                             tCommand = Alloca;
104                             symtab_push($1, gRegnum++, gScope);
105                     }
106
107                     Row *tRow = symtab_lookup($1);
108
109                     factor_push(tRow->name, tRow->regnum, gScope);
110                     Factor *tRetval = factor_pop();
111
112                     code_add(code_create(tCommand, NULL, NULL, tRetval
                            ));
113             }
114             | id_list COMMA IDENT
115 !!↑と同じ
116             ;
117 !!省略
```

2.3 ex1.p のコンパイルと実行

以下の操作で ex1.p を実行した。

ソースコード 5　ex.p の実行

```
1  $ make
2  $ ./parser ex1.p
3  $ lli result.ll
```

result.ll の返り値を %4 に変更することで、最後に計算された z の値を確認した。ステータスコードとして 32 が返り、正しく実行できていることが確認できた。