実験レポート 2

坪井正太郎（101830245）

2021 年 1 月 22 日

1　はじめに

1.1　ソースコードの省略について

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や実装が単純な関数の省略など、意味が分かる範囲で省略が行われている。省略されている部分は、「!!」で始まるコメントが入っている。

また、関数のシグネチャがわからなくなる省略は行っていない。

1.2　全体の実装で気をつけた部分

parser.y では複雑な制御構文、malloc は使用しないようにした。また、構造体は基本的には関数経由で定義するような実装にした。基本操作のみに縛ることで、parser.y でのコーディングが楽になった。

1.3　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.h 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、その線形リスト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定義した。

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;
12  !!global は alloca と同じ
13      struct {
14        Factor *arg1;
15        Factor *retval;
16      } load;
17  !!store は load と同じ
18      struct {
19        Factor *arg1;
20        Factor *arg2;
21        Factor *retval;
22      } add;
23  !!あとの四則演算はadd と同じ
24    } args;
25    // 次の命令へのポインタ
26    struct llvmcode *next;
27  } LLVMcode;
28
29  // 初期化処理
30  void code_init();
31
32  // 命令とオペランド、格納先とかを指定して、llvm 命令を作る
33  LLVMcode *code_create(LLVMcommand, Factor *, Factor *, Factor *);
34
35  // ↑↓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
36
37  // スタックに 1つのllvm 命令を追加する
38  void code_add(LLVMcode *);
```

```
39
40  // 全てのコードをファイルに出力する
41  // 最後に呼び出す
42  void print_LLVM_code();
43
44  // ~~~~~~~~~~ここから関数の表現定義~~~~~~~~~~
45
46  /* LLVM の関数定義 */
47  typedef struct fundecl {
48    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

parser.y では構文要素の間に中括弧で囲んだ C のコードを書くことで LLVM IR を生成した。

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。特に大域変数の場合、outblock 前で__GlobalDecl という架空の関数を作り、その中に global 命令列を追加した。

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。SSA では基本的に 1 つの命令で 1 つのレジスタを使用するため、code_create にレジスタ番号を渡す際に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、現在の番号は gRegnum という大域変数で管理した。

また、変数の宣言と使用についても実装を行った。宣言は id_list 内で、スコープを管理する gScope の値によって処理を変え、global または alloca 命令を追加した。使用時は factor の中の var_name で記号表からの検索後に、ロード命令で検索した記号列から tRetval へとロードし、スタックに積んだ。

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。

2.4 考察

四則演算のコンパイルは、各項を逆ポーランド記法と同じ順でスタックに積み下ろしすることで、実現させた。

また、PL-0 では文は式として扱われない。つまり、全ての expression は assign_statement でリデュースされるし、逆に statement が expression の中でリデュースされることもない。これは、要素数を管理しなくても、文の終わりで必ず factor を積んだスタックは空になるということであると考えた。この特徴は、課題 5 の制御構文をコンパイルするプログラムに有用であった。

出力された LLVM IR を見てみると、四則演算命令については基本的に同じような命令文だが、sdiv にのみ nsw という節がないことが分かる。LLVM Language Reference Manual[1] によると、nsw は'No Signed Wrapper' フラグであり、nsw が指定されると、オーバーフローする演算に対して、オーバーフローした値ではなく poison value という値を返す。(実装は調べていないため、どこかにフラグを立てて伝播させている可能性もある) つまり、nsw は「未定義動作の検知のために、正規の値で返さないラッパーを有効にする」というフラグだということが分かった。

poison value によって、LLVM 上で未定義動作に基づいた何らかの最適化を行うことができる。[2]

以上のことから、sdiv 演算はオーバーフローの可能性がないので nsw のフラグを持っていないと考えた。しかし、sdiv もゼロ除算によって未定義動作となるが、poison value を返すわけではなく、そのようなフラグも無かった。(丸め検知フラグのみ存在) 今回の考察、調査ではその理由は分からなかったので、未定義動作による最適化の手法と適用範囲については今後の課題としたい。

3 課題 5

3.1 課題概要

PL-0 をコンパイルするために、以下のような機能をコンパイラに追加した。

- 比較演算
- if,while,for などの制御文
- 手続きの呼び出し
- read,write 手続き

本来であれば、変数の扱いは課題 5 の内容だが、課題 4 で既に実装、説明してあるのでその部分は省略する。

3.2 実装

3.2.1 比較演算

llvm.h で新たな命令 icmp を定義した。icmp 命令は比較演算の種類を指定する必要があるので、Cmptype 列挙体型を用意してコード生成時に引数として渡すこととした。それに伴って、parser.y で code_create 命令を呼び出している部分を変更した。

具体的な生成は、parser.y で以下のように行った。

ソースコード 6　condition

```
1  condition
2  : expression EQ expression
3  {
4          // 四則演算と大体一緒
5          Factor *tArg2 = factor_pop();
6          Factor *tArg1 = factor_pop();
7          Factor *tRetval = factor_push("", gRegnum++, LOCAL_VAR);
8
9          code_add(code_create(Icmp, tArg1, tArg2, tRetval, EQUAL));
10 }
11 !!その他の比較演算も同様に定義。
```

3.2.2 if,while,for などの制御文

まず、ラベルを管理しやすくするために、symtab で定義した Scope を変更し、factor でラベルを扱えるようにした。LLVM には Label 命令はないが、llvm.h,llvm.c を変更し、コード生成時には命令として Label を指定し、ラベルとして factor を渡すことで生成させた。

これによって、バックパッチの実現が factor のスタックを通して行える。以下に示すように、後から番号を付けたいラベルを push しておき、for 文の内部の文が読まれてから pop して*Factor.val に適切な値を代入する。br 系命令のラベルと、バックパッチされる label 文は同じポインタを指しているので、レジスタ割当が未確定のラベルに対しても分岐できるようになる。

ソースコード 7　バックパッチの方法

```
1  hoge_statement
2    : hoge HUGA hoge
3    {
4      Factor *undef_label = factor_push("label_name", 0, LABEL);
5      // pop しない!
6
7      code_add(code_create(BrCond, undef_label, hoge, huga_condition,
           0));
8    }
9     statement // この中でもレジスタ番号は消費される
10   {
11     Factor *back_patch_label = factor_pop();
12     back_patch_label->val = gRegnum++;
13
14     code_add(code_create(BrUncond, back_patch_label, NULL, NULL, 0));
15     code_add(code_create(Label, back_patch_label, NULL, NULL, 0));
16   }
17   ;
```

各制御構文のうち、for に関してのみ parser.y での生成箇所と、生成されたコードについての説明を行う。

clang を使うことで確かめた LLVM IR は以下のような命令列であった。これに加えて、expression TO expression で push された 2 つの factor を pop し、(1)〜(2) の間で

使用する。

ソースコード 8　for 文の LLVM IR

```
1   カウンタ変数に初期値を代入する
2 LABEL (1)
3   カウンタ変数と上限値の比較
4   上限以下の場合 (2)、上限より大きい場合 (4)にジャンプする
5 LABEL (2)
6   for 文の中で行う操作
7 LABEL (3)
8   カウンタ変数をインクリメントする
9   (1)にジャンプする
10 LABEL (4)
```

実際の実装は以下のようになった。カウンタレジスタは statement の前後で同じ値を使うために、statement の前で push しておき、後ろから pop できるようにした。ラベルについては説明したように、push のみ行い pop したときに実際のレジスタ番号を割り当てた。これらの操作は、1 つの statement で factor のスタックが変化しないことが保証されているため、可能である。また、各制御文の中でも factor のスタックを文の前後で保存するように操作した。

ソースコード 9　for 文の生成箇所

```
1  for_statement
2          : FOR IDENT ASSIGN expression TO expression DO
3          {
4                  Row *tRow = symtab_lookup($2);
5                  Factor *tTo = factor_pop(); // tFrom から tTo まで
6                  Factor *tFrom = factor_pop();
7
8                  Factor *tCnt = factor_push(tRow->name, tRow->regnum,
                       tRow->type);
9                  // カウンタ（インクリメントしていく変数）
10                 //pop せずにとっておく
11
12                 code_add(code_create(Store, tFrom, tCnt, NULL, 0));
13
14                 Factor *tLoop = factor_push("for.loop", gRegnum++,
                       LABEL);
```

```
15              // ループで戻ってくる場所、あとからpop して br 命令を書く
16
17              code_add(code_create(BrUncond, tLoop, NULL, NULL, 0));
18              code_add(code_create(Label, tLoop, NULL, NULL, 0));
19
20              factor_push("", gRegnum++, LOCAL_VAR);
21              Factor *tCntLocal = factor_pop();
22              code_add(code_create(Load, tCnt, NULL, tCntLocal, 0));
                    // カウンタをレジスタに持ってくる
23
24              factor_push("", gRegnum++, LOCAL_VAR);
25              Factor *tCond = factor_pop(); // 条件を用意する
26
27              code_add(code_create(Icmp, tCntLocal, tTo, tCond, SLE
                    ));
28
29              factor_push("for.do", gRegnum++, LABEL);
30              Factor *tDo = factor_pop(); // 条件でブレークしないとき
                    にここに飛ぶ
31
32              Factor *tBreak = factor_push("for.break", 0, LABEL);
33              // バックパッチであとから値入れたいのでpush したままにする。
34
35              code_add(code_create(BrCond, tDo, tBreak, tCond, 0));
36              code_add(code_create(Label, tDo, NULL, NULL, 0));
37      }
38        statement
39      {
40              // この順番で出てくる
41              Factor *tBreak = factor_pop();
42              Factor *tLoop = factor_pop();
43              Factor *tCnt = factor_pop();
44
45              // cnt インクリメント部ここから
46              factor_push("for.increment", gRegnum++, LABEL);
47              Factor *tInc = factor_pop();
48              code_add(code_create(BrUncond, tInc, NULL, NULL, 0));
49              code_add(code_create(Label, tInc, NULL, NULL, 0));
```

```
50
51                  factor_push("", gRegnum++, LOCAL_VAR);
52                  Factor *tCntLocal = factor_pop();
53
54                  code_add(code_create(Load, tCnt, NULL, tCntLocal, 0));
                        // カウンタをレジスタに持ってくる
55
56                  factor_push("", 1, CONSTANT);
57                  Factor *tOne = factor_pop();
58
59                  factor_push("", gRegnum++, LOCAL_VAR);
60                  Factor *tRetval = factor_pop();
61
62                  code_add(code_create(Add, tCntLocal, tOne, tRetval,
                        0));
63
64                  code_add(code_create(Store, tRetval, tCnt, NULL, 0));
65                  // インクリメント部ここまで
66
67                  code_add(code_create(BrUncond, tLoop, NULL, NULL, 0));
68
69                  tBreak->val = gRegnum++; // バックパッチ
70                  code_add(code_create(Label, tBreak, NULL, NULL, 0));
71          }
72          ;
```

3.2.3 手続きの呼び出し

LLVM IR での手続き呼び出しは call 命令を使用する。llvm.c, llvm.h には call の定義と生成ルールを実装した。

parser.y では以下のようにロード命令に近い形で生成した。

ソースコード 10　手続きの呼び出し生成部分

```
1 proc_call_name
2 : IDENT
3 {
4         Row *tRow = symtab_lookup($1);
5         factor_push(tRow->name, tRow->regnum, tRow->type);
```

```
6           Factor *tProc = factor_pop();
7
8           factor_push("", gRegnum++, LOCAL_VAR);//関数の戻り血
9           Factor *tRetval = factor_pop();
10          code_add(code_create(Call, tProc, NULL, tRetval, 0));
11  }
12  ;
```

3.2.4 read,write 手続き

clang を利用して scanf,printf がどのように表現されるか確かめたところ、それぞれの関数の宣言と、フォーマット文字列が定義されていた事がわかった。取り出したものは以下のようになる。

@.str.w は整数に置換される部分 (%d)、改行コード (0A)、終端文字 (00) で、@.str.r は整数部と終端文字で構成され、それぞれ printf,scanf で使用する。呼び出し時はソースコード 12 で示すように、call 命令を使用して引数に出力する整数または、読み込まれる変数を渡す。

ソースコード 11　read,write のための宣言

```
1 @.str.w = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
2 @.str.r = private unnamed_addr constant [3 x i8] c"%d\00", align 1
3 declare i32 @printf(i8*, ...)
4 declare i32 @__isoc99_scanf(i8 *, ...)
```

ソースコード 12　read,write の呼び出し

```
1   call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
        [4 x i8]* @.str.w, i64 0, i64 0), i32「レジスタ」)
2   call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x
        i8], [3 x i8]* @.str.r, i64 0, i64 0), i32*「変数」)
```

3.3 pl0a.p〜pl0d.p のコンパイルと実行

以下のコマンドでそれぞれのソースに対してコンパイルを行い、実行した。出力されない結果は ret 文を書き換えてステータスコードを参照することで確かめた。

pl0a.p の結果は n=0,sum=55 となり、正しい結果が確認できた。また、b,c,d はそれぞれ、入力以下の素数を大きなものから出力するプログラム、同様に小さなものから出力

するプログラム、入力値の階乗を出力するプログラムであることが確認でき、入出力が正しいことが確認できた。

ソースコード 13　pl0a.p〜pl0d.p のコンパイルと実行コマンド

```
$ make
$ ./parser pl0a.p
$ lli result.ll
$ ./parser pl0b.p
$ lli result.ll
  10
  7
  5
  3
  2
$ ./parser pl0c.p
$ lli result.ll
  10
  2
  3
  5
  7
$ ./parser pl0d.p
$ lli result.ll
  10
  3628800
```

3.4　考察

課題 4 の考察で触れたように、factor のスタックは statement を抜けるときに必ず入ったときと同じ状態になる。課題 5 の実装では、factor にラベルを記憶できるようにしたことで、その特徴をバックパッチに活用することができた。これによって、コード量を削減することができ、可読性が向上したと考える。

参考文献

[1] LLVM Language Reference Manual #binary-operations, https://llvm.org/docs/LangRef.html#binary-operations, (2020/1/20 閲覧)

[2] LLVM Language Reference Manual #poison-values, https://llvm.org/docs/LangRef.html#poisonvalues, (2020/1/20 閲覧)