実験レポート 3

坪井正太郎（101830245）

2021 年 2 月 6 日

1　課題 6

1.1　課題概要

PL-1 コンパイラを完成させるために、以下の機能を実装した。

- 定義時に手続きの仮引数を定義する
- 呼び出し時に手続きに引数を渡す
- 呼び出し時に手続きの引数の数をチェックする

1.2　実装

引数を扱うために、定義時の仮引数を v_args という構文要素で、呼び出し時の実引数を args という構文要素として定義した。また、引数の個数は上位の構文要素からはわからないので、gArity というグローバル変数によって管理した。(定義毎、呼び出し毎に 0 初期化する。)

1.2.1　仮引数の定義

仮引数を利用するためには、%0 から始まる引数を llvm の関数定義の中で alloc と store で仮引数の値を局所変数として持ってくる必要があり、その際レジスタ番号を消費する。この動作を実現するために、v_args の中で引数の数をカウントしておき、抜けたときに一括でコードを生成、消費したレジスタ番号を返すことでそれ以下のレジスタとの整合性をとる。これらは、llvm.h で void fundecl_add_arg(Factor *); int fundecl_add_arg_code(); として定義した関数を用いて実装した。

1.2.2　呼び出し時の操作、引数の数をチェック

手続きの呼び出しは文として、statement 内の構文要素 proc_call_statement として定義した。記号表に従って arity の正しさを検証し、args に逆順で入っている factor を取り出し、call 命令のコードに渡した。

arity のチェックは、v_args のときと同様に gArity を利用して記号との比較で実現した。

1.3 実行結果

1.3.1 pl1a.p の実行

pl1a.p を実行したところ、以下のような結果となった。

ソースコード 1　pl1a.p の実行コマンド、結果

```
1 $ ./parser pl1a.p
2 $ lli result.ll
3 $ 5
4   120
5 $ lli result.ll
6 $ 8
7   40320
```

1.3.2 エラーのあるプログラムの実行

pl1a.p の 9 行目の手続き呼び出しの際に、存在しない変数を引数として渡してコンパイルした。

ソースコード 2　ミスのある pl1a.p の実行結果

```
1 $ ./parser pl1a.p
2   not decleared yet
3   line: 9
4   )
5   too much procedure argments
6   line: 9
7   )
```

1.4 考察

構文要素をまたいで下位の要素がどれだけ連続しているか知る必要があるため、グローバル変数を使って実装した。また、今回の実装では変数周りのエラー (引数の数も) についてコンパイル時に yyerror を呼び出した。これまでの実装ではコンパイル時にエラー (記号表に存在するかなど) をチェックせず、セグフォや実行時エラーを出していたため、コードが書きやすくなった。

実行結果は、階乗を実行するプログラムが正しく動作した。また、エラー文のも正しいものが出力されていることが確認できた。

2 課題 7

2.1 課題概要

PL-2 コンパイラを完成させるために、以下の機能を実装した。

- 関数の定義

- 関数の戻り値の生成
- 定義時、呼び出し時の引数のチェック (手続きと一緒なので実装は省略)

2.2 実装

関数の定義は、LLVM において関数と手続きの区別がないことから構文要素 func_decl を追加し、その中で proc_decl と同じような操作を行った。

また、関数の戻り値は項として必ず利用され、文 (statement) として扱われることはないとした。そのため、関数呼び出しは新たな構文要素 func_call_factor を構文要素 factor の中で定義した。

2.2.1 関数の定義

関数の定義は、LLVM において関数と手続きの区別がないことから構文要素 func_decl を追加し、その中で proc_decl と同じような操作を行った。

2.2.2 関数の戻り値の生成

関数は戻り値を持つため、そのための領域を関数の先頭で alloca しておき、これをグローバル変数として保持した。変数名として、現在定義中の関数名と同じ名前のものが渡された場合には、保持しておいた戻り値領域を利用する。

2.3 実行結果

実行結果は以下のようになった。

2.3.1 pl2 の実行結果

ソースコード 3　pl2a.p, pl2b.p の実行結果

```
1 $ ./parser pl2a.p
2 $ lli result.ll
3 $ 5
4   120
5 $ ./parser pl2b.p
6 $ lli result.ll
7 $ 2
8 $ 4
9   16
```

2.3.2 関数を手続きとして呼び出したときの実行結果

pl2a.p の 12 行目に手続きとして fact(n); を挿入し、実行した。

ソースコード 4　関数を手続きとして呼び出す

```
1 $ ./parser pl2a.p
2   not decleared as procedure
```

```
3   line: 12
4   )
```

2.4 考察

今回の実装では、関数は文として使われないという仕様を決めた。実際に、実行結果では関数が手続きと同様に呼び出されたときにエラーを出力している。

また、通常の pl2 の実行結果から、階乗を行うプログラム (a) と、2 つの数のべき乗を求めるプログラム (b) が正しく動作していることが確認できる。

3 課題 7

3.1 課題概要

PL-3 コンパイラを完成させるために、以下の機能を実装した。

- 配列の定義
- 配列の参照
- 配列への代入

3.2 実装

記号表に配列用の新たな種別を追加し、Factor 構造体にサイズを持たせるための新たなメンバを追加した。これにより、通常の命令生成と同じように alloca や global に渡すことができる。

3.2.1 配列の定義

id_list の内部を新たな構文要素 id_decl で入れ替え、その中で変数と配列の定義を行った。表現するデータ構造はほぼ一緒で、メンバの内容だけが違うので、変数の定義とほぼ同様の処理である。

配列のサイズは、最大インデックス-オフセット +1 で算出した。

3.2.2 配列の参照、代入

var_name 中に、配列の参照部分を作成し、その中で getelementptr 命令を呼び出した。getelementptr で 1 次元の配列の要素のポインタを取り出すとき、getelementptr inbounds [size x i32], [size x i32]* array, i32 0, i32 index という呼び出しを行う。このような命令を生成するコマンドを llvm.h 内で宣言した。

3.3 プログラムの実行結果

3.3.1 pl3 の実行結果

pl3 を実行した結果、以下のようになった。

ソースコード 5　pl3.p の実行結果

```
1 $ ./parser pl3a.p
```

```
$ lli result.ll
$ 5
$ 1
$ 2
$ 3
$ 4
$ 5
  5
  4
  3
  2
  1

$ ./parser pl3a.p
$ lli result.ll
$ 20
  2
  3
  5
  7
  11
  13
  17
  19
```

3.4 考察

getelementptr の 1 つ目のオペランドに指定されている i32 0 は、配列それ自体を指すオフセットである。また、getelementptr に inbounds キーワードを指定することで、範囲外へのアクセスに対して poison value を返す。

pl3 の実行では、入力した数列を逆順に出力するプログラム (a) と、入力以下の素数を出力するプログラム (b) が正しく動作していることが確認できた。

4 課題 9

4.1 課題概要

以下の機能を実装した。

- forward 文による、先行宣言

4.2 実装

var_decl_part のあとに forward_decl_part を追加した。その中で、記号表に対する関数 (手続き) 名の追加を行った。また、引数の数も記号表に登録する必要があるので、f_args の中でグローバル変数 gArity をインクリメントした。

今回の実装では、宣言されたプログラムが定義されていない場合、エラーを出力する必要があり、そのためにグローバル変数として gUndefined を用意した。また、関数として宣言されて手続きとして定義された場合 (逆も) にもエラーを出力した。

4.3 pl4 の実行

pl4-test.p, pl4a.p を実行した結果、以下のようになった。

ソースコード 6　pl4 の実行結果

```
1  $ ./parser pl4-test.p
2  $ lli result.ll
3  $ 6
4    8
5  $ 100
6
7  $ ./parser pl4a.p
8  $ lli result.ll
9  $ 100
10   5050
```

4.4 考察

llvm IR ではグローバルに定義された関数はどの順序でも呼び出すことができる。したがって、記号表への追加のみでコード生成のプログラムに変更はなかった。

また、入力が 10 以下のときにフィボナッチ数列を出力するプログラム (pl4-test) と入力までの数の和を出力するプログラム (pl4a) が正しく動作していることが確認できた。

5 課題 10

5.1 課題概要

最適化の機能として、以下の機能を追加した。

- コンパイル時の定数計算
- 除算の高速化

5.2 実装

optimize/optimize.h で最適化用のプログラムを実装した。まず、基本ブロックごとの最適化を行うために、関数列→コード列→基本ブロックの順に分解する関数の中で、基本ブロックの開始と終了ポインタをとる関数として最適化関数を定義した。

最適化された命令列を生成するために llvm.h 内で任意の文字列をコードとして埋め込むことの出来る疑似命令 Free を実装した。通常変数名となる部分に命令文字列を格納することで、その文字列を展開する。

また、llvm IR ではレジスタが番号でつけられていると、番号の間が飛んだ場合にエラーが発生する。そのため、全てのレジスタ番号にプレフィックスをつけることでレジスタ名による指定に切り替えた。

5.2.1 コンパイル時の定数計算

四則演算命令のうち、オペランドが 2 つとも定数である場合、そのコードを削除、格納先のレジスタを定数に書き換え、値として命令の演算結果を入力した。連続して行うことで、計算可能なコード列については全て置き換えることができる。

5.2.2 除算の高速化

除算の高速化として「2 のべき乗による割り算のシフト化」「3 で割る処理を乗算で実行」の 2 種類を実装した。

5.2.3 2 のべき乗による割り算のシフト化

2 のべき乗による割り算を、右シフト命令として置き換えた。今回のプログラムは全て符号付き整数なので、符号拡張を行う ashr 命令によって右シフトを実行した。

また、負の数を割る際の丸めを正しく実行するために、負の数を割る場合にのみ符号ビットをあらかじめ加算した。この処理は、31 ビット分 ashr で右シフトすることで 0 または-1 を取り出すことで実現した。

5.2.4 3 で割る処理を乗算で実行

2 のべき乗を 3 で割った巨大な整数を掛け合わせることで、べき乗分右シフトすると 3 で割った数を出力できる。今回は 2 の 32 乗を 3 で割り、上に切り上げた数 1431655766 を使った。2 で割ったときと同様に、負の数の丸め分を補うために、1 を加算する処理も行った。この処理は、符号ビットを結果から減算することで実装した。

また、2 の 32 乗を掛けた数は明らかに i32 には収まらない。そのため、型変換拡張命令 sext と縮小命令 trunc を前後につけた。

5.3 テストプログラムの実行

以下のようなプログラムを用意して、コンパイル結果を確認、time コマンドでベンチマークをとった。

ソースコード 7　テストプログラム

```
1 program bench;
2   var x, n, nn;
3   begin
```

```
4      for n := 1 to 1000000 do
5       for nn := 1 to 5000 do
6         begin
7             x := n div (2 + 1);
8             x := n div 2;
9         end
10 end.
```

5.3.1 コンパイル結果

ソースコード 7 をコンパイルした結果、以下のプログラムが生成された。for.do ブロック内において、定数のコンパイル時計算 (2+1 → 3) と、2 と 3 の割り算の最適化が行われていることが確認できる。

ソースコード 8　コンパイル結果

```
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 *, ...)
5
6  @x = common global i32 0, align 4
7  @n = common global i32 0, align 4
8  @nn = common global i32 0, align 4
9
10 define i32 @main(){
11   store i32 1, i32* @n, align 4
12   br label %o1
13
14 ; <for.loop>:
15 o1:
16   %o2 = load i32, i32* @n, align 4
17   %o3 = icmp sle i32 %o2, 1000000
18   br i1 %o3, label %o4, label %o21
19
20 ; <for.do>:
21 o4:
22   store i32 1, i32* @nn, align 4
23   br label %o5
24
25 ; <for.loop>:
26 o5:
27   %o6 = load i32, i32* @nn, align 4
28   %o7 = icmp sle i32 %o6, 5000
29   br i1 %o7, label %o8, label %o17
30
31 ; <for.do>:
32 o8:
```

```
33    %o9 = load i32, i32* @n, align 4
34
35    ;[OPTIMIZED]
36    ;div 3 -> sift hogehoge
37    %o9.1 = sext i32 %o9 to i64
38    %o9.2 = ashr i64 %o9.1, 31
39    %o9.3 = mul i64 %o9.1, 1431655766
40    %o9.4 = ashr i64 %o9.3, 32
41    %o9.5 = sub i64 %o9.4, %o9.2
42    %o11 = trunc i64 %o9.5 to i32
43
44    store i32 %o11, i32* @x, align 4
45    %o12 = load i32, i32* @n, align 4
46
47    ;[OPTIMIZED]
48    ;div 2^n -> sift-R n
49    %o12.1 = ashr i32 %o12, 31
50    %o12.2 = add i32 %o12, %o12.1
51    %o13 = ashr i32 %o12.2, 1
52
53    store i32 %o13, i32* @x, align 4
54    br label %o14
55
56  ; <for.increment>:
57  o14:
58    %o15 = load i32, i32* @nn, align 4
59    %o16 = add nsw i32 %o15, 1
60    store i32 %o16, i32* @nn, align 4
61    br label %o5
62
63  ; <for.break>:
64  o17:
65    br label %o18
66
67  ; <for.increment>:
68  o18:
69    %o19 = load i32, i32* @n, align 4
70    %o20 = add nsw i32 %o19, 1
71    store i32 %o20, i32* @n, align 4
72    br label %o1
73
74  ; <for.break>:
75  o21:
76    ret i32 0
77  }
```

5.3.2 ベンチマーク結果

最適化の前後で 10 回づつ計測し、time コマンドのうち user の CPU 時間をとった。

表 1　ベンチマーク結果

	最適化前 (s)	最適化後 (s)
1 回目	9.047	9.052
2 回目	9.028	9.123
3 回目	9.102	9.074
4 回目	9.020	9.068
5 回目	9.021	9.154
6 回目	9.009	9.102
7 回目	9.023	9.124
8 回目	9.026	9.071
9 回目	8.968	9.043
10 回目	9.050	9.448
平均	9.294	9.126

5.4 考察

今回の最適化では、定数の計算に関わる部分に関してのみ基本的な最適化を行った。一般に、プロセッサレベルでは除算はその他の四則演算より大幅に遅いことから、その部分について特に重点的に最適化を行った。

5.4.1 3 の除算

割られる数を q、結果を p とすると、以下の等式が成り立つ。

$$p = q\left(\frac{2^{32}}{3}\right)\left(\frac{1}{2^{32}}\right)$$

ここで、$\frac{2^{32}}{3} = 1431655765.33$ であり、小数点以下については $\frac{1}{2^{32}}$ を掛けたときに必ず切り捨てられる。そのため、q,p を整数に制限すると

$$p = q\,(1431655766)\left(\frac{1}{2^{32}}\right)$$

としてもよい。負の数の表現を考慮して結果には符号ビットを足す必要があるが、3 の除算についてはこれによって掛け算と右シフトで表現できる。

さらに、同様の計算を他の数に対しても適用できるため、ある程度小さな数については、3 以外の除算に対しても同じ最適化ができる。これによって、2 のべき乗との組み合わせで様々な数の除算が乗算に変更できる。

5.4.2 ベンチマークに対する考察

ベンチマークをとったところ、最適化後のほうが平均で 0.17 秒 (2%) ほど遅い結果となった。これは、i32 ←→ i64 の型変換によるオーバーヘッドの影響であると予想し、追実験を行った。コンパイル結果のソース

コード 8 のうち、型変換に関する 37,42 行目を削除し、実行にエラーが起きないように削除されたレジスタ部分を定数で書き換えて再度 10 回実行した。

結果は以下のようになり、0.313 秒 (3.3%) ほど早くなった。予想通り、型変換のオーバーヘッドが原因であることが分かった。

表 2　追実験結果

	最適化前 (s)	型変換なし最適化 (s)
1 回目	9.047	8.968
2 回目	9.028	9.052
3 回目	9.102	8.992
4 回目	9.020	9.017
5 回目	9.021	9.019
6 回目	9.009	9.051
7 回目	9.023	9.043
8 回目	9.026	8.900
9 回目	8.968	8.846
10 回目	9.050	8.928
平均	9.294	8.981

このことから、3 の除算最適化は LLVM IR においては効果が打ち消され、レジスタの幅が利用する型ごとに決定される LLVM との相性が悪いと考えた。逆にレジスタが、利用する整数型の倍以上の幅を持っている場合や、整数型のうち、下位半分のビットしか使わない場合には一定の効果があると言える。