コンパイラ実験

学生番号: 09B54923549
中嶋 空偉 (NAKAJIMA, Sorai)

2026 年 1 月 27 日

1　実験の概要と目的

1.1　本実験の概要

この実験では，独自に定義した言語をソース言語として，アセンブリ言語を目的言語として変換するコンパイラを作成する．コンパイラ作成の際には，字句解析を行うプログラムを生成するものとして flex，構文解析を行うプログラムを生成するものとして bison，ast とコード生成に関しては c 言語でプログラムを作成する．

1.2　本実験の目的

3 年間の総仕上げとしてある程度大きなプログラムを作成する．仕様を決め，プログラムを書き，データ構造を決め，アルゴリズムを考え，テストパターンを決め，デバッグするなどの一連の流れを経験する．

2　言語の定義

本コンパイラが受理する言語の文法定義を以下に示す．これは Bison の構文規則から C 言語アクションコードを取り除いたものである．

```
 1: program
 2:     : declarations statements
 3: ;
 4:
 5: declarations
 6:     : decl_statement declarations
 7:     | decl_statement
 8: ;
 9:
10: decl_statement
11:     : DEFINE IDENT SEMIC
12:     | ARRAY IDENT L_BRACKET NUMBER R_BRACKET SEMIC
13:     | ARRAY IDENT L_BRACKET NUMBER R_BRACKET L_BRACKET NUMBER R_BRACKET SEMIC
14: ;
15:
16: statements
17:     : statement statements
18:     | statement
19: ;
20:
21: statement
```

```
22:     : assignment_stmt
23:     | loop_stmt
24:     | cond_stmt
25:     | expression SEMIC
26: ;
27:
28: assignment_stmt
29:     : IDENT ASSIGN expression SEMIC
30:     | IDENT L_BRACKET expression R_BRACKET ASSIGN expression SEMIC
31:     | IDENT L_BRACKET expression R_BRACKET L_BRACKET expression R_BRACKET ASSIGN expression SEMIC
32: ;
33:
34: expression
35:     : expression add_op term
36:     | term
37: ;
38:
39: term
40:     : term mul_op factor
41:     | factor
42: ;
43:
44: factor
45:     : var
46:     | NUMBER
47:     | L_PAREN expression R_PAREN
48:     | IDENT L_BRACKET expression R_BRACKET
49:     | IDENT L_BRACKET expression R_BRACKET L_BRACKET expression R_BRACKET
50: ;
51:
52: add_op
53:     : PLUS
54:     | MINUS
55: ;
56:
57: mul_op
58:     : MUL
59:     | DIV
60: ;
61:
62: var
63:     : IDENT
64: ;
65:
66: loop_stmt
67:     : WHILE L_PAREN condition R_PAREN L_BRACE statements R_BRACE
68: ;
69:
70: cond_stmt
71:     : IF L_PAREN condition R_PAREN L_BRACE statements R_BRACE ELSE L_BRACE statements R_BRACE
72:     | IF L_PAREN condition R_PAREN L_BRACE statements R_BRACE
73: ;
74:
75: condition
76:     : expression cond_op expression
77: ;
78:
79: cond_op
80:     : EQ
81:     | NE
82:     | LE
83:     | GE
84:     | LT
85:     | GT
86: ;
```

3　受理されるプログラム例

本コンパイラで受理できるプログラムの例を示す.

3.1　基本演算と While ループ

まず，基本的な演算と繰り返し処理の例として，1 から 10 までの和を計算するプログラムを示す.

```
 1: define i;
 2: define sum;
 3: sum = 0;
 4: i = 1;
 5: while (i < 11) {
 6:   sum = sum + i;
 7:   i = i + 1;
 8: }
 9: sum;
```

このプログラムでは，変数 i を 1 から 10 まで 1 ずつ増やしながら，sum に加算していくことで和を求めている.

3.2　条件分岐 (If-Else)

次に，条件分岐の例として，2 つの数値の大小比較を行い，大きい方から小さい方を減算するプログラムを示す.

```
 1: define a;
 2: define b;
 3: a = 10;
 4: b = 5;
 5: if (a > b) {
 6:   a = a - b;
 7: } else {
 8:   a = a + b;
 9: }
10: a;
```

このプログラムでは，if-else 文を使用して条件によって実行する処理を分岐している．本コンパイラでは，==，!=，<=，>=，<，>の比較演算子が利用可能である.

3.3　配列の使用

最後に，配列の例として，3 行 3 列の 2 次元配列の対角成分の和を計算するプログラムを示す.

```
1: array arr[3][3];
2: define sum;
3: define i;
4: arr[0][0] = 1;
5: arr[0][1] = 2;
6: arr[0][2] = 3;
7: arr[1][0] = 4;
```

```
 8: arr[1][1] = 5;
 9: arr[1][2] = 6;
10: arr[2][0] = 7;
11: arr[2][1] = 8;
12: arr[2][2] = 9;
13: sum = 0;
14: i = 0;
15: while (i < 3) {
16:   sum = sum + arr[i][i];
17:   i = i + 1;
18: }
19: sum;
```

このプログラムでは，2 次元配列を宣言し，対角成分（arr[0][0]，arr[1][1]，arr[2][2]）を while ループを用いて順に足し合わせている．1 次元配列と 2 次元配列の両方がサポートされている．

4 コード生成の実装

本章では，コンパイラの核心部分であるコード生成の実装詳細について述べる．本実験では，MIPS アーキテクチャ上で動作するアセンブリコードを出力することを目的とした．実装にあたり，レジスタ割り当ての複雑さを回避し，実装の確実性を優先するため，演算の途中経過を逐一スタックに退避する「スタックマシン」方式を採用した．

4.1 算術式のコード生成

算術式の計算においては，以下のルールに従ってコード生成を行った．

- 演算の結果は常に $v0 レジスタに格納する．
- 演算の対象となるもう一方の値は，スタックから $v1 レジスタに復帰させる．
- 演算終了後，必要に応じて結果を再度スタックにプッシュ（退避）する．

具体的には，PLUS_AST（加算）などの二項演算において，以下の手順で処理を行っている．

1. 右辺の子ノードを再帰的に評価し，結果（$v0）をスタックにプッシュ（退避）する．
2. 左辺の子ノードを再帰的に評価し，結果を $v0 に得る．
3. スタックから退避していた右辺の値を $v1 にポップ（復帰）させる．
4. add $v0, $v0, $v1 命令を発行する．

特に工夫した点として，右辺から先に評価する順序を採用したことが挙げられる．これにより，減算（sub）のような非可換な演算においても，スタックから取り出した値（右辺）が自然に第 2 オペランド（引く数）となり，sub $v0, $v0(**左**), $v1(**右**) という直感的な命令生成が可能となった．

4.2 制御構文の翻訳

while 文や if 文の実装では，ジャンプ先のラベル生成が課題となった．単純な固定ラベルではネスト構造に対応できないため，while_loop_count などの静的カウンタ変数を導入した．これにより，$WHILE_LOOP_START_0，$WHILE_LOOP_START_1 のようにユニークなラベルを生成し，多重ループや複雑な

分岐構造でも正しく動作するように実装した.

また，条件判定においては C 言語の「0 ならば偽，それ以外は真」という仕様に基づき，条件式の結果が 0（偽）の場合に処理をスキップ（分岐）させるため，MIPS の `beq $v0, $zero, LABEL` 命令を用いて表現した．さらに，MIPS 特有の遅延分岐（Delay Slot）に対しては，分岐命令の直後に必ず `nop` を挿入することで，意図しない命令実行を防ぐ安全な実装とした.

4.3 変数と配列のメモリ管理

変数のアクセスには，関数実行中に変動するスタックポインタ（`$sp`）ではなく，フレームポインタ（`$fp`）を基準としたオフセット方式（例：`lw $v0, -4($fp)`）を採用した.

特に 2 次元配列（`array[i][j]`）の実装では，行優先（Row-Major）配置を前提とし，シンボルテーブルに配列の列数（cols）を記録する設計とした．アクセス時には，$\texttt{address} = \texttt{base} + (i \times \texttt{cols} + j) \times 4$ という計算式をアセンブリ命令として展開し，動的なインデックスに対しても正しいアドレス計算を実現している.

5 特に工夫した点

5.1 2 パス処理によるメモリ領域管理

本コンパイラの実装において最も工夫し，苦労した点は変数のメモリ管理である．MIPS の関数呼び出し規約では，関数の先頭で必要なスタック領域を一括して確保することが望ましい．しかし，C 言語では関数の途中でも変数宣言が可能であるため，単純な 1 パスのコンパイラでは，コード生成開始時に正確なフレームサイズを知ることができないという問題に直面した.

そこで本実装では，コード生成関数 `generate_code` において，以下の 2 段階（2 パス）の処理を行う設計とした.

1. **第 1 パス（宣言解析）**：AST 全体を一度走査（`find_declarations`）し，全ての変数宣言を検出してシンボルテーブルに登録する．同時に各変数のスタックフレーム上のオフセットアドレスを決定し，必要な変数の総サイズを確定させる．これにより第 2 パスでのアドレス計算処理が不要となる.
2. **第 2 パス（コード生成）**：確定したサイズ情報を元にスタック確保命令（`addiu $sp, $sp, -size`）を出力し，その後に実際の AST 走査を行う.

最初は変数が見つかるたびにスタックを拡張する方法も考えたが，オフセット計算が複雑になりバグの原因となったため，この「静的領域を先に確定させる」方式に切り替えた．結果として，メモリ管理が非常にシンプルかつ堅牢になり，バグを大幅に減らすことができた.

5.2 コンパイラのソースプログラム

本コンパイラのソースコードは以下のディレクトリに配置されている.

```
c-compiler/
├── src/
│   ├── codegen.c        # コード生成
│   ├── program.y        # Bison 構文規則
│   ├── program.l        # Flex 字句規則
│   ├── symbol_table.c   # シンボルテーブル
```

```
│   └── ...
└── Makefile
```

6 最終課題のプログラムと実行結果

6.1 1 から 10 までの数の和

6.1.1 プログラム

```
1: define i;
2: define sum;
3: sum = 0;
4: i = 1;
5: while(i < 11) {
6:   sum = sum + i;
7:   i = i + 1;
8: }
```

6.1.2 実行結果

シミュレータによる実行結果（主要部分）を以下に示す.

```
*** total inst. count:          347
*** total cycle count:         1421
***               IPC:   0.244 (inst/cycle)
***               CPI:   4.095 (cycle/inst)

7ffffe0: 0000000b 00000000 00000000 00000037
7fffff0: 0000000b 00000000 00000018 00000000
```

メモリダンプより，sum はアドレス 0x7fffffec に格納され，値は **0x37**（10 進数で **55**）であることが確認できる．これは $1+2+\cdots+10=55$ という正しい計算結果と一致している.

6.2 5 の階乗

6.2.1 プログラム

```
1: define i;
2: define fact;
3:
4: fact = 1;
5: i = 1;
6: while(i < 6) {
7:   fact = fact * i;
8:   i = i + 1;
9: }
10: fact;
```

6.2.2 実行結果

シミュレータによる実行結果（主要部分）を以下に示す.

```
*** total inst. count:          197
*** total cycle count:          806
***               IPC:   0.244 (inst/cycle)
***               CPI:   4.091 (cycle/inst)
```

```
7fffffe0: 00000006 00000000 00000000 00000078
7ffffff0: 00000006 00000000 00000018 00000000
```

メモリダンプより，fact はアドレス 0x7fffffec に格納され，値は **0x78**（10 進数で **120**）であることが確認できる．これは $1 \times 2 \times 3 \times 4 \times 5 = 120$ という正しい計算結果と一致している．

6.3 FizzBuzz

6.3.1 プログラム

```
 1: define fizz;
 2: define buzz;
 3: define fizzbuzz;
 4: define others;
 5: define i;
 6: fizz = 0;
 7: buzz = 0;
 8: fizzbuzz = 0;
 9: others = 0;
10: i = 1;
11: while(i < 31) {
12:   if((i / 15) * 15 == i) {
13:     fizzbuzz = fizzbuzz + 1;
14:   } else {
15:     if((i / 3) * 3 == i) {
16:       fizz = fizz + 1;
17:     } else {
18:       if((i / 5) * 5 == i) {
19:         buzz = buzz + 1;
20:       } else {
21:         others = others + 1;
22:       }
23:     }
24:   }
25:   i = i + 1;
26: }
27: others;
```

6.3.2 実行結果

シミュレータによる実行結果（主要部分）を以下に示す．

```
*** total inst. count:         3077
*** total cycle count:        12577
***               IPC:   0.245 (inst/cycle)
***               CPI:   4.087 (cycle/inst)

7fffffc0: 00000000 00000000 00000000 0000000f
7fffffd0: 0000000f 0000001f 00000000 00000000
7fffffe0: 0000001f 00000010 00000002 00000004
7ffffff0: 00000008 00000000 00000018 00000000
```

メモリダンプより，各カウンタの値を確認できる．fizzbuzz（3 と 5 両方で割り切れる）は **0x02**（2 個），fizz（3 で割り切れる）は **0x08**（8 個），buzz（5 で割り切れる）は **0x04**（4 個），others（いずれでも割り切れない）は **0x10**（16 個）となっていることが読み取れる．プログラムの最後で others の値 **0x10**（16 個）が出力されており，$2 + 8 + 4 + 16 = 30$ となり，ループ回数と一致しているため正しい実行結果であると言える．

6.4 エラトステネス

6.4.1 プログラム

```
 1: define N;
 2: define i;
 3: define j;
 4: define k;
 5: array a[1001];
 6: N = 1000;
 7: i = 1;
 8: while (i <= N) {
 9:   a[i] = 1;
10:   i = i + 1;
11: }
12: i = 2;
13: while( i <= N/2) {
14:   j = 2;
15:   while(j <= N/i){
16:     k = i * j;
17:     a[k] = 0;
18:     j = j + 1;
19:   }
20:   i = i + 1;
21: }
```

6.4.2 実行結果

シミュレータによる実行結果（主要部分）を以下に示す.

```
*** total inst. count:       335792
*** total cycle count:      1389229
***               IPC:   0.242 (inst/cycle)
***               CPI:   4.137 (cycle/inst)

7ffff030: 00000000 000001f4 00000000 00000000
7ffff040: 00000000 00000001 00000001 00000001
7ffff050: 00000000 00000001 00000000 00000001
7ffff060: 00000000 00000000 00000000 00000001
7ffff070: 00000000 00000001 00000000 00000000
7ffff080: 00000000 00000001 00000000 00000001
7ffff090: 00000000 00000000 00000000 00000001
7ffff0b0: 00000000 00000001 00000000 00000001
```

メモリダンプより，配列 a の素数フラグを確認できる．最初の行には N = **0x1f4**（1000）が格納されている．配列 a[1] 〜 a[16] の値を見ると，1（素数フラグ）の位置は a[2]，a[3]，a[5]，a[7]，a[11]，a[13] であり，これは実際に素数である 2, 3, 5, 7, 11, 13 に対応していることが確認できる．エラトステネスの篩アルゴリズムが正しく実装されていることが分かる.

6.5 二次元配列

6.5.1 プログラム

```
 1: array matrix1[2][2];
 2: array matrix2[2][2];
 3: array matrix3[2][2];
 4: define i;
 5: define j;
```

```
 6: define k;
 7: matrix1[0][0] = 1;
 8: matrix1[0][1] = 2;
 9: matrix1[1][0] = 3;
10: matrix1[1][1] = 4;
11: matrix2[0][0] = 5;
12: matrix2[0][1] = 6;
13: matrix2[1][0] = 7;
14: matrix2[1][1] = 8;
15: i=0;
16: while(i<2){
17:     j=0;
18:     while(j<2){
19:         matrix3[i][j] = 0;
20:         j = j+1;
21:     }
22:     i = i+1;
23: }
24: i=0;
25: while(i<2){
26:     j=0;
27:     while(j<2){
28:         k=0;
29:         while(k<2){
30:             matrix3[i][j] = matrix3[i][j] + matrix1[i][k] * matrix2[k][j];
31:             k = k+1;
32:         }
33:         j = j+1;
34:     }
35:     i = i +1;
36: }
```

6.5.2 実行結果

シミュレータによる実行結果（主要部分）を以下に示す.

```
*** total inst. count:         1365
*** total cycle count:         5594
***               IPC:   0.244 (inst/cycle)
***               CPI:   4.098 (cycle/inst)

7fffffa0: 00000000 00000000 00000000 00000002
7fffffb0: 00000000 00000000 00000002 00000002
7fffffc0: 00000002 00000013 00000016 0000002b
7fffffd0: 00000032 00000005 00000006 00000007
7fffffe0: 00000008 00000001 00000002 00000003
7ffffff0: 00000004 00000000 00000018 00000000
```

メモリダンプより，行列 matrix3 の計算結果を確認できる．matrix3[0][0] = **0x13** (19)，matrix3[0][1] = **0x16** (22)，matrix3[1][0] = **0x2b** (43)，matrix3[1][1] = **0x32** (50) となっている．これらは以下の行列積の計算結果と正しく一致している．

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1\times5+2\times7 & 1\times6+2\times8 \\ 3\times5+4\times7 & 3\times6+4\times8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

7　考察

本実験におけるコード生成の実装と性能，そしてその工夫点について，定量的なデータと実験を用いて以下の通り考察する．

7.1　スタックマシン方式の評価と課題

7.1.1　メモリアクセス頻度の分析

本コンパイラは，演算の途中経過や変数の値を逐一スタックに退避する「スタックマシン」方式を採用している．この方式のオーバーヘッドを評価するため，「1 から 10 までの和」を計算するプログラム（`sum.c`）の生成コード（`output/sum.s`）を分析した．

■データ分析　総命令数 70 行（静的ステップ数）のうち，`sw`（ストア）および `lw`（ロード）命令は合計 19 行であり，全命令の約 **27%** がメモリアクセス命令で占められていることが判明した．一般的な RISC アーキテクチャ（MIPS 含む）において，メモリアクセスはレジスタ間演算に比べてレイテンシが大きく，パイプラインハザードの原因にもなりやすい．命令数の約 3 割がメモリアクセスであるという事実は，実行性能において看過できないボトルネックとなっている．

■実験による考察　この高いメモリアクセス頻度は，主に（1）レジスタ割り当ての複雑さを回避する設計，(2）演算のたびにスタックへのアクセスを強制する実装，の 2 つに起因していると考えられる．特にループ内の変数アクセス（例：`lw \$v0, -12(\$fp)` → 計算 → `sw \$v0, -16(\$fp)`）が毎回繰り返されているため，ループ全体の実行時間に支配的な影響を与えているのが見逃せない．

7.2　レジスタ割り当てによる最適化の実験

スタックマシン方式の課題を実証するため，`codegen.c` を一時的に改造し，特定の変数（`i` と `sum`）を強制的にレジスタ（`$t0`, `$t1`）に割り当てる「ハードコード最適化」実験を行った．この最適化を行ったコンパイラで `sum.c` を再コンパイルした結果，表 1 に示す改善が見られた．

表 1　スタックマシン方式とレジスタ割り当て（手動最適化）の性能比較

評価項目	オリジナル (Stack)	最適化実験版 (Reg)	改善率
実行サイクル数	347 cycles	305 cycles	約 12% 減
`sw` 命令実行回数	19 回	5 回	約 74% 減
実行結果	55 (正解)	55 (正解)	-

■分析

- 変数の読み書きをレジスタ化しただけで，ストア命令の実行回数は劇的に減少し，プログラム全体の実行ステップ数も 1 割以上削減された．
- 今回は演算の途中経過（プッシュ/ポップ）までは最適化していないため，もしそれらも含めてフルにレジスタを利用すれば，さらに大幅な性能向上が見込める．
- この実験より，実用的なコンパイラにおいては，スタックマシン方式から脱却し，グラフ彩色アルゴ

リズム等を用いた適切なレジスタ割り当て（Register Allocation）を実装することが不可欠であると結論付けられる．

7.3 AST 設計と拡張性

7.3.1 2 次元配列の実装と AST

本実験において，2 次元配列（`array[i][j]`）の実装がスムーズに進んだ要因は，構文解析（Parser）段階での AST 設計にある．`ARRAY_DECL_STATEMENT_AST` ノードに配列の次元情報（行数・列数）を保持させる設計としたことで，コード生成フェーズにおいてシンボルテーブルに必要な情報（`cols`）を確実に渡すことができた．もし AST に必要な情報が含まれていなければ，コード生成中に情報を補完する複雑な処理（バックパッチなど）が必要となり，実装難易度は格段に上がっていたはずである．「後のフェーズで必要となる情報は，全て AST に残す」という原則の重要性を再認識した．

7.4 MIPS アーキテクチャへの適応

7.4.1 遅延分岐スロットの対処

MIPS 特有の遅延分岐（Delay Slot）に対し，本実装では一律 `nop` を挿入する安全策をとった．`FizzBuzz` プログラムの実行結果（3,077 命令）を見ると，その相当数が `nop` であると推測される．商用コンパイラでは，このスロットに「分岐の直前の命令」や「分岐先でも分岐しなくても無害な命令」を移動させる「遅延スロット・スケジューリング」が行われる．本コンパイラでも，単純な `nop` 挿入ではなく，直前の計算命令をスロットに移動させる最適化を実装すれば，命令数をさらに数 %〜10% 程度削減できる余地があると考えられる．