情報工学実験 C コンパイラ実験最終レポート

氏名: 寺岡 久騎 (TERAOKA, Hisaki)
学生番号: 09B22433

出題日: 2025 年 1 月 26 日
提出日: 2025 年 2 月 3 日
締切日: 2025 年 2 月 4 日

1 本実験の目的

本実験の目的は, これまで学んできた C 言語についての知識を再確認するとともに, それを使って,

- ソフトウェア全体の仕様の決定
- プログラムで利用するデータ構造，アルゴリズムの考案と実装
- 動作のテスト，デバッグの作業

これら一連の作業を通して大規模なプログラムの作成の経験をすること，そして lex，yacc といったプログラムジェネレータを使用したコンパイラプログラムの作成を通して，コード解析やファイルシステムに利用される木構造の取り扱い，ソースコードとアセンブリ言語との対応についての理解と習熟を深めることである.

今回作成したコンパイラプログラムの実行ファイルは，演習室の PC の `/home/users/ecs/09B22433/exp-c/compiler/lang` ディレクトリに `lang2` という名前で保存している.

2 作成した言語の定義

以下に作成した言語の定義として，文法規則を記述した yacc のルールを示す．言語は，最終課題 1–5 が実行できる実装となっている.

```
%union {
  struct node *np;
  char* sp;
  int   ival;
}

%type <np> program declarations decl_statement decl_part
           statements statement
           assignment_stmt assignment loop_stmt cond_stmt
           while_stmt if_stmt else_stmt elif_stmt for_stmt
           expressions expression
           condition array array_index term factor unary_factor
           comp_op unary_op bit_op var idents
```

```
14:
15: %type <ival> add_op mul_op
16:
17: %token DEFINE ASSIGN ARRAY_DEF
18:         L_BRACKET R_BRACKET L_PARAN R_PARAN L_BRACE R_BRACE
19:         SEMIC COMMA ADD SUB MUL DIV REM INCREM DECREM EQ NE LT GT LTE GTE
20:         AND OR XOR NOT L_SHIFT R_SHIFT
21:         FOR WHILE IF ELSE
22:
23: %token <sp> IDENT
24: %token <ival> NUMBER
25:
26: %define parse.error verbose
27:
28: %%
29:
30: program
31:   : declarations statements
32:   | declarations
33: ;
34:
35: declarations
36:   : decl_statement declarations
37:   | decl_statement
38: ;
39:
40: array_index
41:   : L_BRACKET expression R_BRACKET
42:   | L_BRACKET expression R_BRACKET L_BRACKET expression R_BRACKET
43: ;
44:
45: array
46:   : IDENT array_index
47: ;
48:
49: decl_part
50:   : DEFINE idents
51:   | ARRAY_DEF array
52: ;
53:
54: decl_statement
55:   : decl_part SEMIC
56: ;
57:
58: statements
59:   : statement statements
60:   | statement
61: ;
62:
63: statement
64:   : assignment_stmt
65:   | loop_stmt
66:   | cond_stmt
67: ;
68:
69: assignment
70:   : IDENT ASSIGN expression
71:   | array ASSIGN expression
72:   | unary_factor
73: ;
74:
75: assignment_stmt
76:   : assignment SEMIC
```

```
77: ;
78:
79: expressions
80:   : expression COMMA expressions
81:   | expression
82: ;
83:
84: expression
85:   : expression add_op term
86:   | term
87: ;
88:
89: term
90:   : term mul_op factor
91:   | term bit_op factor
92:   | factor
93: ;
94:
95: unary_factor
96:   : IDENT unary_op
97:   | unary_op IDENT
98: ;
99:
100: factor
101:   : var
102:   | L_PARAN expression R_PARAN
103:   | unary_factor
104:   | NOT IDENT
105: ;
106:
107: add_op
108:   : ADD
109:   | SUB
110: ;
111:
112: mul_op
113:   : MUL
114:   | DIV
115:   | REM
116:
117: unary_op
118:   : INCREM
119:   | DECREM
120: ;
121:
122: bit_op
123:   : AND
124:   | OR
125:   | XOR
126:   | L_SHIFT
127:   | R_SHIFT
128: ;
129:
130: var
131:   : IDENT
132:   | NUMBER
133:   | IDENT array_index
134: ;
135:
136: loop_stmt
137:   : while_stmt
138:   | for_stmt
139: ;
```

```
140:
141: while_stmt
142:   : WHILE L_PARAN condition R_PARAN L_BRACE statements R_BRACE
143: ;
144:
145: for_stmt
146:   : FOR L_PARAN assignment SEMIC condition SEMIC assignment R_PARAN
147:   L_BRACE statements R_BRACE
148: ;
149:
150: cond_stmt
151:   : if_stmt
152:   | elif_stmt
153: ;
154:
155: if_stmt
156:   : IF L_PARAN condition R_PARAN L_BRACE statements
157: ;
158:
159: else_stmt
160:   : ELSE L_BRACE statements R_BRACE
161: ;
162:
163: elif_stmt
164:   : if_stmt else_stmt
165: ;
166:
167: condition
168:   : expression comp_op expression
169: ;
170:
171: comp_op
172:   : EQ
173:   | NE
174:   | LT
175:   | GT
176:   | LTE
177:   | GTE
178: ;
179:
180: idents
181:   : IDENT COMMA idents
182:   | IDENT
183: ;
184:
185: %%
```

2.1 定義した言語で受理されるプログラム

2.1.1 全体の構造

作成した言語では，プログラム全体はまず

- 変数宣言部
- 処理文集合

からなる仕様となっており，使用される変数はプログラムの冒頭で全て宣言され，その後に変数が使用される様々な処理文の集合が記述される．従って，文集合中に変数宣言は行われない．

2.1.2 変数の宣言

プログラムの変数宣言部では，4 バイトの整数値を格納できる通常の変数と，この整数値を複数格納できる配列 (1 次元，2 次元) が記述される．yacc の規則から，プログラムにおいて変数の宣言として以下の記述ができる．ここで，**<識別子名>**は変数名であり，先頭がアルファベットの英数字列である．

整数変数: `define` **<識別子名>**`;`
整数変数 (複数): `define` **<識別子名>**`, ...;`
配列 (1 次元): `array` **<識別子名>**`[`**自然数**`];`
配列 (2 次元): `array` **<識別子名>**`[`**自然数**`][`**自然数**`];`

2.1.3 処理文

文集合では，C 言語と同様の

- 代入文
- ループ文 (`while`，`for`)
- 条件分岐文 (`if`，`if-else`)
- 算術式
- 条件式

による処理が記述できる．式については，条件式では比較演算子が，そして算術式では算術演算子が利用できる．以下，実装した各種演算子についての概説を行う．

■算術演算子　以下，それぞれ実装した算術式における演算子を示す．`a` 及び `b` は被演算子であり，整数の即値，変数，算術式 (の演算結果) が対象となる．また，演算対象とその演算結果は 4 バイトの符号付き整数である．

- `a + b`： `a` と `b` の値を加算する
- `a - b`： `a` の値から `b` の値を減する
- `a * b`： `a` と `b` の値を乗算する
- `a / b`： `a` の値を `b` で除する
- `a % b`： `a` の値を `b` の値で割った時の剰余を計算する
- `a++`・`(a--)`： `a` の値を 1 だけ増加 (減少) する．ただし，この処理は `a` を被演算子や対象とする代入・算術処理の後に行われる．
- `++a`・`(--a)`： `a` の値を 1 だけ増加 (減少) する．ただし，この処理は `a` を被演算子や対象とする代入・算術処理の前に行われる．

■比較演算子　比較式において，その式の結果は条件が成立する場合は 1，成立しない場合は 0 となる (詳細は第 3.2 節に示す)．以下，それぞれ実装した演算子を示す．`a` 及び `b` は被演算子であり，整数の即値，変数，算術式 (の演算結果) が対象となる．

- `a == b`： `a` と `b` の値が一致しているを比較する

- a != b： a と b の値を一致していないかを比較する
- a < b： a の値より b の値が大きいかを比較する
- a > b： a の値が b の値より大きいかを比較する
- a <= b： a の値が b の値以下であるかを比較する
- a >= b： a の値が b の値以上であるかを比較する

2.1.4 受理されるプログラムの例

以上の定義した言語の使用から，実際に作成したコンパイラに受理されるプログラムの例を示す．ただし，プログラム自体は意味のある処理を行うものではない．

```
 1: define i;
 2: define j, k;
 3: array  m1[10];
 4: araay  m2[3][3];
 5: i = 1;
 6: j = i + 1;
 7: k = i * j;
 8:
 9: while (i <= 100) {
10:   i = i + 2;
11:   if (i % 2 == 1) {
12:     j = i / j - 1;
13:   }
14:   else {
15:     --k;
16:   }
17: }
18:
19: for (i = 1; k != 10; k++) {
20:   if (i > j) {
21:     j = k - i--;
22:   }
23: }
```

3 コード生成の概要

作成したコンパイラプログラムでは，以下の一連の流れで処理を行う．

1. 入力された文字列について，lex ファイルで定義した文字列でトークンに該当するものをトークンへ置換，そして全体をトークン列に置換する
2. プログラムのトークン列に対して yacc で記述した文法規則に従って還元を行い，この時に生成規則のアクション部に記述した C 言語の処理により，各トークンをノードとし，プログラム全体のトークンからコード生成に必要な要素を残して抽象構文木を作成する．
3. 構成された抽象構文木をたどり，ノードの構成などからプログラム実行に必要な処理を出力対象のファイルへ maps で実行可能なアセンブリ言語の命令を記述する．

入力されたプログラムのコード生成 (ファイルの作成) は，上記の項目 3 の抽象構文木の解析・命令の記述，及びこの前後に行う以下の処理により行われる．

- (項目 3 実行前) 抽象構文木からプログラムで使用する変数の記号表の作成

- (項目 3 実行前) maps プログラム実行に必要なテキストセグメント宣言等の初期化処理の記述
- (項目 3 実行後) maps プログラム正常終了に必要な処理，データセグメントの記述

maps で実行可能なコードの生成について，上記の抽象構文木を用いた解析に基づく命令の記述と前後処理を踏まえて，以下の項目に分けて解説を行う．

- メモリ領域の使用方法
- maps の汎用レジスタの使用方法
- 算術式のコード生成の方法

3.1 メモリの使用方法

プログラムにおいて，メモリ領域は主に用途により

- 実行する命令が配置されるテキストセグメント
- プログラムで使用される変数が格納されるデータセグメント
- 算術式等の計算結果の一時保存に使用するスタック領域

に分けられる．以下，上記の用途によるメモリの利用について概説を行う．

3.1.1 テキストセグメントの利用

テキストセグメントについて，maps でプログラムを正常終了するための初期化処理をコード生成の直前に，以下の処理を記述する

- グローバルポインタと後述するスタックポインタの値の設定
- テキストセグメントの配置を制御するアセンブラ命令 (`.text`)
- グローバルポインター・スタックポインタのレジスタへの格納
- プログラム終了前に行う処理

そして，その直後にソースコードで実行されるテキストセグメントの配置を制御するアセンブラ命令を記述する．このセグメントのアドレスは上記の初期化処理の命令を考慮し，`0x00001000` として設定した．即ち，ソースコードから生成した命令が配置されるのは，メモリにおいて `0x00001000` から後述するデータセグメントまでの領域としている．

3.1.2 データセグメントの利用

データセグメントについては，メモリ領域の `0x10004000` 領域以降のアドレスを利用している．実際に生成するコードではこのアドレスを保持した汎用レジスタを利用と，先述の変数の記号表で保持している変数のデータセグメントの先頭からのオフセットを利用して，この領域に各変数を格納する．変数の記号表は，ソースコードに宣言された順にサイズとオフセットをそれぞれ計算しており，この際始めに宣言された変数はデータセグメントの先頭アドレスを以降の変数は直前に宣言された変数のオフセットを格納する領域のアドレスとしているため，データセグメントは，先頭アドレスから順に変数を並んで格納するように使用される．

以下，例として通常の変数 (4 バイト) を 3 つ使用する際のデータセグメントにあたるメモリ領域の使用状態を示す．順に `a`，`b`，`c` と宣言したとする．

- 0x10004000 – 0x10004003： a (4Byte)
- 0x10004004 – 0x10004007： b (4Byte)
- 0x10004008 – 0x1000400b： c (4Byte)

3.1.3 スタック領域の利用

プログラムの算術式等の処理で一時的なデータの保存に利用するスタック領域の先頭アドレスは 0x7ffffffc としている．先述の 2 つの領域とは異なり，データをスタックに積む処理を行う場合はこのアドレスよりも小さいアドレスへ変数のデータを格納する．

以下，例として通常の変数 (4 バイト) を 3 を一時的にスタックへ積む処理を行った際のメモリ領域の使用状態を示す．順に変数として a，b，c を順に格納したとする．

- 0x7ffffff4 – 0x7ffffff7： c (4Byte)
- 0x7ffffff8 – 0x7ffffffb： b (4Byte)
- 0x7ffffffc – 0x7fffffff： a (4Byte)

仮に c がスタックから取り出すことに該当する処理が行われた場合，次にスタックへ積まれるデータは c のデータがあった領域に格納される．

3.2 レジスタの使用方法

プログラムコードでは様々な処理で一時的にデータを保持する必要があるため，mips で使用できる汎用レジスタを利用した．レジスタについてはコンパイラプログラムのコード生成の処理の複雑化を避けるために，その一部に固定の役割を与えて使用した．表 1 に使用した汎用レジスタとその役割・用途について示す．

レジスタ	役割・用途
$t0	データセグメントの先頭アドレス (0x10004000) を保持する
$t1	比較演算の演算結果の値の格納先
$t2	算術式・条件式の第 1 オペランド (左側) の値の格納先
$t3	算術式・条件式の第 2 オペランド (右側) の値の格納先
$v0	算術式の計算結果の一時的な保持

表 1: 汎用レジスタの役割・用途

$t0 レジスタはプログラム実行時の始めにデータセグメントの先頭アドレスを格納し，以降はデータセグメントの領域に格納されている変数に対してロードやストアを行う際のアドレスとして利用する．そのため，プログラムの処理で値を変更しない．

$t1 レジスタは条件式の演算結果の格納先としている．これにより，ループ文や分岐処理文においてジャンプ命令・比較演算のコードの生成がこれレジスタを必ず参照するようにすることで条件式の生成処理に依存せず，コンパイラプログラム全体の処理の拡張・修正の実装が容易に行えるようにした．

$t2 及び$t3 は演算処理においてオペランドの値の格納先として利用した．これにより，オペランドに更に算術式が連なる複雑な式の再帰的な生成処理において，使用するレジスタの削減と算術式関

連のプログラムの複雑化を避けた.

3.3 算術式のコード生成の方法

算術式は，抽象構文木において演算子を親ノードとし，演算対象となる 2 つのオペランドのノードが子ノードとなる 2 分木の構造となっている．そのため，演算処理のコードの生成にあたってはオペランドの値を第 3.2 節に示したレジスタに格納した上で，それを対象として各計算の命令を記述する．計算結果は`$v0` に格納し，その値をスタックへプッシュする．

演算子の対象となるオペランドのノードには以下のいずれかが該当する.

- 整数の即値のノード
- 変数のノード (通常の変数・配列の 1 要素)
- 算術式の演算子のノード

オペランドとして更に算術式が連なっている場合があり，この場合ではより深い側の演算は計算の優先順序が高いため，正しい計算順序でコードの生成を行うには，木のより深い側の演算から行う必要がある．これと先述の算術式に関する値の格納に使用するレジスタの扱いを考慮すると，算術式のコード生成の処理は以下の流れで行われるよう実装した.

1. 親にあたる演算子のノードの 2 つの子ノードが算術演算子であるかを確認する
   - 子ノードが算術演算子の場合，1 の処理からはじまる一連の処理を，このノードを親ノードとして再帰的に呼び出す (第 1 オペランド，第 2 オペランドの順でこの処理を行う)
2. 第 2 オペランドのノードが算術演算子であるかを確認し，該当するかどうかで以下処理を行う
   算術演算子の場合: 演算結果がスタックに積まれているため，そのデータをスタックからポップして`$t3` レジスタへロードする
   算術演算子でない場合: 該当の数値や変数の値を`$t3` レジスタに格納する
3. 第 1 オペランドに対して 2 の処理を行う (格納先のレジスタは`$t2` となる)
4. オペランドの値が格納された`$t2` レジスタ，`$t3` レジスタを利用して演算子に対応した命令を記述する．ここで計算結果の格納先は`$v0` レジスタとする
5. 計算結果が格納された`$v0` レジスタの値をスタックへプッシュする

以上の処理により，算術式に対してオペランドに算術式が続くような形であっても，再帰的な処理によって深さ優先探索の要領で優先順序が高い計算を実行する命令が先に記述され，計算結果がスタックに保存された状態になり，その後にこの計算結果を取り出して使用する親側の算術式が記述される．これによって算術演算が連なる式であっても正しい順序で計算が行われるコードが生成される.

4 工夫した点について

コンパイラプログラムの作成において，生成されるコードにおける最適化・実行サイクル数の削減に関する以下の工夫・改良を行った.

- 式におけるオペランドのデータの遅延ロードを考慮した `nop` 命令の削減
- 算術式・条件式におけるオペランドの種類による最適なデータ取得処理の選択

4.1 オペランドのロード処理時 nop 命令の削減

算術式・条件式においては，第 3.3 節で示したように演算対象である 2 つのオペランドのデータをレジスタへ格納を行う．対象となるデータの内，変数 (配列の要素) とさらに連なる算術式の結果をレジスタに格納する場合，ロード命令 (lw) を使用する．maps においてはこのロードが完了してレジスタにデータが格納されるには 1 サイクルの遅延がかかり，直後にそのデータを用いる処理が正常に動作しないため，その対策として通常は以下の手順の様にロード命令直後に nop 命令を追加で記述する必要がある．

1. 第 2 オペランドのデータのロード ($t3 レジスタ)
2. nop 命令 (遅延ロード対策)
3. 第 1 オペランドのデータのロード ($t2 レジスタ)
4. nop 命令 (遅延ロード対策)

今回の実装では，オペランドのデータのロードは必ず第 2 オペランドのデータを格納する処理の後に第 1 オペランドのデータを格納する処理が記述される．そのため，第 2 オペランドの処理の後に対象のレジスタ ($t3) を使用する処理が記述されないため，第ロード命令によりレジスタにデータを格納する場合，直後に nop 命令が不要となるため，以下の処理手順の様にオペランドのデータ格納時に対象が第 2 オペランドである場合は nop 命令を記述しないようにすることで，サイクル数の削減を行った．

1. 第 2 オペランドのデータのロード ($t3 レジスタ)
2. 第 1 オペランドのデータのロード ($t2 レジスタ)
3. nop 命令 (遅延ロード対策)

4.2 算術式・条件式におけるデータの取得時の最適なデータ取得処理の選択

算術式のオペランドとして算術式が連なるような長い式では，子ノードにあたる式の計算結果を親にあたる演算で用いるため，一時的にその結果を保存しておく必要がある．この場合，オペランドが算術式かそうでないかに関わらず，データをスタックへプッシュし，演算の際に 2 つのオペランドをポップする処理を，抽象構文木の演算子のノードから子ノードをたどって再帰的に行うことでオペランドのデータあるいは計算結果を利用して演算処理を行うことができる．しかし，この処理では本来レジスタに格納するだけで使用できる整数の即値と変数のデータに対してスタックにプッシュしてポップする操作を行っており，余分にサイクル数がかかっていると考えられる．

そのため，単に再帰的処理の中で全てのデータに対してスタックを使用するのではなく，算術式にあたる構文木に対して，より深い優先すべき計算から順に算術演算子に該当するノードを対象としてコード生成を行い，オペランドが算術式かそうでないかでスタックからデータを取り出すか，単にレジスタに値を格納するかを分岐させることで，常に必要なデータの格納処理を選択する処理を実現し，サイクル数の削減を図った．

4.3 最適化による実行サイクル数の変化

今回のコンパイラプログラムでは，以上 2 つの最適化の工夫を施した上で実装した．この工夫により生成した最終課題のコードのサイクル数ついて，

- オペランドを種類に依らず必ずスタックへプッシュし，都度ポップする
- レジスタへのロードの直後に必ず nop 命令を記述する

という最適化を施さない処理で生成した場合と比較した結果を表 2 に示す．

いずれのプログラムでも約 40% 程実行サイクル数が削減された．課題の内，変数の宣言と値の初期化処理を除いた処理において，算術式に即値を用いる処理の割合が比較的高い課題 1，2，3 ではやや削減率が高いことからも，スタックを用いずにレジスタへへ直接データを格納する処理にしたことによるサイクル数の減少の効果が大きいことが考えられる．

課題	最適化なし	最適化あり	サイクル数の削減率 (%)
課題 1	2325	1231	47.06
課題 2	1245	681	45.31
課題 3	15341	8269	46.10
課題 4	2125655	1212350	42.97
課題 5	7582	4450	41.31

表 2: 最適化によるプログラムの合計実行サイクル数

5 最終課題のプログラム及び実行結果

定義した言語及びそれを対象としたコンパイラプログラムにより実行が可能な課題として，以下の最終課題に取り組んだ．

課題 1 1 から 10 までの数の和の計算
課題 2 5 の階乗の計算
課題 3 FizzBuzz 問題
課題 4 エラトステネスのふるいを用いた素数の探索
課題 5 2 つの 2×2 行列の積の計算

これら最終課題を解くにあたって記述したプログラムのコードと，それを作成したコンパイラにより maps アセンブリ言語へ変換したコードについて，

- シミュレーション上での実行によるサイクル数・命令数の出力結果
- コンパイル時に出力する，プログラムで使用した変数の情報
- 実行後の変数に関するメモリ領域の出力結果

を示す．

5.1 課題 1 1 から 10 までの数の和の計算

■ソースコード

```
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: }
```

5.1.1 実行・出力結果

実行の結果，合計 1231 サイクル (ステップ) で処理が終了した．プログラムでは，sum は和を格納する変数として，変数 i は加算する数として使用される．i は while 文中で 1 ずつ加算され，値が 11 となった時点で処理が終了する．実行後の変数のメモリから，i は $b_{(16)} = 11$，sum が $37_{(16)} = 55$ となっており，正しく 1 から 10 までの和が計算されたことが確認できた．

■実行時の命令・サイクル数

```
*** multicycle statistics ***
*** total inst. count:          303
*** total cycle count:         1231
***               IPC:   0.246 (inst/cycle)
***               CPI:   4.061 (cycle/inst)
```

■使用した変数の情報

```
# ------ [ Symbols ] ------
#         symbol_0         size: 4         offset:     0(0)         [i]
#         symbol_1         size: 4         offset:   0x4(4)         [sum]
```

■実行後の変数メモリ

```
# tag=256 index=4
10004000: 0000000b 00000037 00000000 00000000
```

5.2 課題 2 5 の階乗の計算

■ソースコード

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

```
 8: }
```

5.2.1 実行・出力結果

実行の結果，合計 681 サイクル (ステップ) で処理が終了した．プログラムでは，fact は階乗の値を格納する変数として，変数 i は乗算する数として使用される．i は while 文中で 1 ずつ加算され，値が 6 となった時点で処理が終了する．実行後の変数のメモリから，i は 6，sum が $78_{(16)} = 120$ となっており，1 から 5 が乗算され，正しく階乗の計算が行われたことが確認できた．

■実行時の命令・サイクル数

```
*** multicycle statistics ***
*** total inst. count:          168
*** total cycle count:          681
***               IPC:   0.247 (inst/cycle)
***               CPI:   4.052 (cycle/inst)
```

■使用した変数の情報

```
# ------ [ Symbols ] ------
#         symbol_0         size: 4         offset:     0(0)         [i]
#         symbol_1         size: 4         offset:   0x4(4)         [fact]
```

■実行後の変数メモリ

```
# tag=256 index=4
10004000: 00000006 00000078 00000000 00000000
```

5.3 課題 3 FizzBuzz 問題

本課題は，基本言語仕様を拡張して実装した剰余演算子を用いて記述した．

■ソースコード

```
 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 == 0){
13:     fizzbuzz = fizzbuzz + 1;
14:   } else {
15:     if (i % 3 == 0){
16:       fizz = fizz + 1;
17:     } else {
18:       if (i % 5 == 0){
```

```
19:         buzz = buzz + 1;
20:       } else {
21:         others = others + 1;
22:       }
23:     }
24:   }
25:   i = i + 1;
26: }
```

5.3.1 実行・出力結果

実行の結果，合計 8269 サイクル (ステップ) で処理が終了した．プログラムにおいて FizzBuzz 問題の対象は 1 から 30 であり，15 の倍数は 2 個存在するため，FizzBuzz の該当数は 2 となる．このことから，Fizz は $30/3-2$ で 8，Buzz は $30/5-2$ で 4 となる．実行後の変数メモリでは，`Fizz` が 8，`Buzz` が 4，`FizzBuzz` が 2，となっており，対象の数として `while` 文の処理で 1 加算される `i` が $\mathtt{1f}_{(16)} = 31$ となっており，正しく処理が行われたことが確認できた．

■実行時の命令・サイクル数

```
*** multicycle statistics ***
*** total inst. count:         2092
*** total cycle count:         8269
***               IPC:   0.253 (inst/cycle)
***               CPI:   3.953 (cycle/inst)
```

■使用した変数の情報

```
# ------ [ Symbols ] ------
#         symbol_0         size: 4         offset:     0(0)         [fizz]
#         symbol_1         size: 4         offset:   0x4(4)         [buzz]
#         symbol_2         size: 4         offset:   0x8(8)         [fizzbuzz]
#         symbol_3         size: 4         offset:   0xc(12)         [others]
#         symbol_4         size: 4         offset:  0x10(16)         [i]
```

■実行後の変数メモリ

```
# tag=256 index=4
10004000: 00000008 00000004 00000002 00000010
10004010: 0000001f 00000000 00000000 00000000
```

5.4 課題 4 エラトステネスのふるいを用いた素数の探索

本課題では，1 から 1000 までの数を対象となっている．

■ソースコード

```
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: }
```

5.4.1 実行・出力結果

このプログラムでは，配列の a の素数に該当する数の要素が 1，そうでない合成数の要素が 0 となる．配列 a はアドレス 0x10004010 からはじまる領域に格納され，各要素が 4 バイトとなっており，変数のメモリにおいて，最初の 4 バイトが添字として 0 に該当する要素である. このことから，実行後の変数メモリで 0，1 番目の要素を除いて 2，3，5，7，11，13，17，19，23 番目の値が 1 となっており，正しく素数の探索処理が行えたことが確認できる．

■実行時の命令・サイクル数

```
*** multicycle statistics ***
*** total inst. count:       294858
*** total cycle count:      1212350
***               IPC:   0.243 (inst/cycle)
***               CPI:   4.107 (cycle/inst)
```

■使用した変数の情報

```
# ------ [ Symbols ] ------
#         symbol_0         size: 4         offset:     0(0)         [N]
#         symbol_1         size: 4         offset:   0x4(4)         [i]
#         symbol_2         size: 4         offset:   0x8(8)         [j]
#         symbol_3         size: 4         offset:   0xc(12)         [k]
#         symbol_4         size: 4004         offset:  0x10(16)         [a]
```

■実行後の変数メモリ　出力されたメモリ領域が大きいため，省略した冒頭の一部を示す.

```
  # tag=256 index=4
10004000: 000003e8 000001f5 00000003 000003e8
10004010: 00000000 00000001 00000001 00000001
10004020: 00000000 00000001 00000000 00000001
10004030: 00000000 00000000 00000000 00000001
10004040: 00000000 00000001 00000000 00000000
10004050: 00000000 00000001 00000000 00000001
10004060: 00000000 00000000 00000000 00000001
```

5.5 課題 5 2 つの 2×2 行列の積の計算

■ソースコード

```
 1: array matrix1[2][2];
 2: array matrix2[2][2];
 3: array matrix3[2][2];
 4: define i, j, k;
 5: matrix1[0][0] = 1;
 6: matrix1[0][1] = 2;
 7: matrix1[1][0] = 3;
 8: matrix1[1][1] = 4;
 9: matrix2[0][0] = 5;
10: matrix2[0][1] = 6;
11: matrix2[1][0] = 7;
12: matrix2[1][1] = 8;
13: for(i=0;i<2;i++){
14:   for(j=0;j<2;j++){
15:    matrix3[i][j] = 0;
16:   }
17: }
18: for(i=0;i<2;i++){
19:   for(j=0;j<2;j++){
20:     for(k=0;k<2;k++){
21:       matrix3[i][j] = matrix3[i][j] + matrix1[i][k] * matrix2[k][j];
22:     }
23:   }
24: }
```

5.5.1 実行・出力結果

実装した 2 次元配列では, 1 次元の配列が順に行数文メモリに格納される仕様であり, 例として 2 次元配列 a[3][2] が格納されるメモリ領域では, 先頭から a[0][0], a[0][1], a[1][0], a[1][1]... の順に要素が並ぶ.

このプログラムで計算する行列の積は,

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

であり, 実行後の変数メモリで積の値が格納される matrix3 に該当する 0x10004020 からはじまる 16 バイトの領域において

第 (1, 1) **要素** (a[0][0]) $13_{(16)} = 19$
第 (1, 2) **要素** (a[0][1]) $16_{(16)} = 22$
第 (2, 1) **要素** (a[1][0]) $2b_{(16)} = 43$
第 (2, 2) **要素** (a[1][1]) $32_{(16)} = 50$

となっており, 正しく行列の積が計算されたことが確認できる.

■実行時の命令・サイクル数

```
*** multicycle statistics ***
*** total inst. count:         1097
*** total cycle count:         4450
```

```
***               IPC:   0.247 (inst/cycle)
***               CPI:   4.054 (cycle/inst)
```

■使用した変数の情報

```
# ------ [ Symbols ] ------
#         symbol_0         size: 16         offset:     0(0)         [matrix1]
#         symbol_1         size: 16         offset:  0x10(16)         [matrix2]
#         symbol_2         size: 16         offset:  0x20(32)         [matrix3]
#         symbol_3         size: 4         offset:  0x30(48)         [i]
#         symbol_4         size: 4         offset:  0x34(52)         [j]
#         symbol_5         size: 4         offset:  0x38(56)         [k]
```

■実行後の変数メモリ　出力されたメモリ領域が大きいため，省略した冒頭の一部を示す．

```
# tag=256 index=4
10004000: 00000001 00000002 00000003 00000004
10004010: 00000005 00000006 00000007 00000008
10004020: 00000013 00000016 0000002b 00000032
10004030: 00000002 00000002 00000002 00000000
```

6　考察

今回のコンパイラプログラムの作成においては，処理の追加実装や改良が行い易いように抽象構文木を再帰的に探索してコード生成ができるノードの構成になるよう作成を行った．これにより変数宣言以降の処理文集合について，内部にさらに文が続くループ文や分岐処理文の実装が比較的容易に行えたため，文集合だけでなく，特に他の処理でも多く利用される算術式などは個々の演算子のノードのみを構文木に残すのではなく，算術式という括りのノードを残すことによって，これを用いる様々な文法規則への応用や機能の実装と拡張がより容易に行えると考えられる．

また，プログラムの実装の観点において，先述の様に再帰的な解析が可能な抽象構文木の作成とその構造やノードに合わせたコード生成の関数などを用いる方針によってプログラムの実装と機能追加は容易になったが，その反面再帰的処理の流れを変えてしまうとコードの生成に支障をきたす箇所が多く，最適化や算術式の計算結果の保持などに関する処理は多くの分岐処理を加えることが必要となった．そのため，単に構文木を探索して都度コードの生成を行うだけでなく，解析の後にコード生成における処理を選択して切り替える，変数の記号表の様に全体の構造を抽象化したデータを作成するといった様々な実装の方針を立てることがさらなる拡張や修正における柔軟性を上げるために重要であると考えられる．