

# 情報工学実験 C

## —コンパイラレポート—

氏名: 中畑 凌成 (Nakahata, Ryosei)  
学生番号: 09B23587

出題日: 2025 年 12 月 xx 日  
提出日: 2026 年 02 月 xx 日  
締切日: 2026 年 02 月 06 日

## 1 実験の目的

コンパイラを作成することを実験の目的とする。ここでこのコンパイラはコードをアセンブリに変換して、最終課題を実行することが出来るということを目指とする。また、コンパイラ作成において `lex` と `yacc` を使用する。

## 2 最終的に作成した言語の定義

`yacc` で定義したルールを記載する。

```
1:
2: my
3: :program
4: ;
5:
6: program //プログラム
7: :declarations statements
8: ;
9:
10: statements //文集合
11: :statement statements
12: |statement
13: ;
14:
15: statement //文
16: :assignment_stmt
17: |loop_stmt
18: |cond_stmt
19: ;
20:
21: assignment_stmt //代入分
22: :idents ASSIGN expression SEMIC {
23:   $$ = build_node2_Assign(ASSIGNMENT_STMT_AST,$1, $3,-1);
24: }
25: |IDENT L_BRACKET NUMBER R_BRACKET ASSIGN expression SEMIC {$$ = build_node2_Assign(ASSIGNMENT_STMT_AST,$1,$2,$3,-1);
26: |IDENT L_BRACKET IDENT R_BRACKET ASSIGN expression SEMIC {$$ = build_node2_Assign(ASSIGNMENT_STMT_AST,$1,$2,$3,-1);
27: |IDENT L_BRACKET IDENT R_BRACKET L_BRACKET IDENT R_BRACKET ASSIGN expression SEMIC {$$ = build_node2_Assign(ASSIGNMENT_STMT_AST,$1,$2,$3,$4,-1);
```

```

28: |IDENT L_BRACKET NUMBER R_BRACKET L_BRACKET NUMBER R_BRACKET ASSIGN expression SEMIC {$$ = build
29: |IDENT L_BRACKET IDENT R_BRACKET L_BRACKET NUMBER R_BRACKET ASSIGN expression SEMIC {$$ = build
30: |IDENT L_BRACKET NUMBER R_BRACKET L_BRACKET IDENT R_BRACKET ASSIGN expression SEMIC {$$ = build
31: ;
32:
33: term //項
34: :term mul_op factor
35: |factor
36: ;
37:
38: factor //因子
39: :var
40: |L_PAREN expression R_PAREN
41: ;
42:
43: add_op //加減演算子
44: :ADD
45: |SUB
46: ;
47:
48: mul_op //乗除演算子
49: :MUL
50: |DIV
51: ;
52:
53: var //変数
54: :idents {$$=build_node1(VAR_AST,$1);}
55: |NUMBER {$$=build_node1(VAR_AST,build_node0(NUMBER_AST,"",$1,""));}
56: |IDENT L_BRACKET NUMBER R_BRACKET {$$=build_node1(VAR_AST,build_node0(IDENT_AST,$1,$3,""));}//mi
57: |IDENT L_BRACKET IDENT R_BRACKET {$$=build_node1(VAR_AST,build_node0(IDENT_AST,$1,0,$3));}
58: |IDENT L_BRACKET NUMBER R_BRACKET L_BRACKET NUMBER R_BRACKET {$$=build_node1(VAR_AST,build_node0
59: |IDENT L_BRACKET IDENT R_BRACKET L_BRACKET IDENT R_BRACKET {$$=build_node1(VAR_AST,build_node0_a
60: |IDENT L_BRACKET IDENT R_BRACKET L_BRACKET NUMBER R_BRACKET {$$=build_node1(VAR_AST,build_node0
61: |IDENT L_BRACKET NUMBER R_BRACKET L_BRACKET IDENT R_BRACKET {$$=build_node1(VAR_AST,build_node0
62: ;
63:
64: declarations //変数宣言部
65: :decl_statement declarations
66: |decl_statement
67: ;
68:
69: decl_statement //宣言文
70: :DEFINE idents SEMIC {$$ = build_node1_Decl(DECL_STATEMENT_AST,$2,-1);}//-1->define
71: |ARRAY IDENT L_BRACKET NUMBER R_BRACKET SEMIC {$$ = build_node2_Decl(DECL_STATEMENT_AST,build_n
72: |ARRAY IDENT L_BRACKET NUMBER R_BRACKET L_BRACKET NUMBER R_BRACKET SEMIC {$$ = build_node3_Decl
73:
74: idents //複数の識別子
75: :IDENT COMMA idents
76: |IDENT
77: ;
78:
79: expression //算術式
80: :expression add_op term
81: |term
82: ;
83:
84: condition //条件式
85: :expression cond_op expression
86: ;
87:
88: cond_op //比較演算子
89: :EQ
90: |GT

```

```

91: |LT
92: |GT ASSIGN
93: |LT ASSIGN
94: ;
95:
96: loop_stmt //ループ文
97: :WHILE L_PAREN condition R_PAREN L_BRACE statements R_BRACE
98: ;
99:
100: cond_stmt //条件分岐文
101: :IF L_PAREN condition R_PAREN L_BRACE statements R_BRACE
102: |IF L_PAREN condition R_PAREN L_BRACE statements R_BRACE ELSE L_BRACE statements R_BRACE
103: ;
104:

```

終端記号の意味を明らかにするため,lex ファイルの一部を示す.

```

1: define {return DEFINE;}
2: if {return IF;}
3: array {return ARRAY;}
4: while {return WHILE;}
5: else {return ELSE;}
6: = {return ASSIGN;}
7: ; {return SEMIC;}
8: [a-zA-Z][a-zA-Z0-9]* {
9:   yylval.sp= (char*)malloc(sizeof(char) * yyleng);
10:  strncpy(yylval.sp , yytext, yyleng);
11:  return IDENT;
12: }
13: [0-9]+ { yylval.ival = atoi(yytext);
14:  return NUMBER;}
15: , {return COMMA;}
16: [ \t]+
17: "+" {return ADD;}
18: "-" {return SUB;}
19: "/" {return DIV;}
20: "*" {return MUL;}
21: "==" {return EQ;}
22: "<" {return LT;}
23: ">" {return GT;}
24: "{" {return L_BRACE;}
25: "}" {return R_BRACE;}
26: "[" {return L_BRACKET;}
27: "]" {return R_BRACKET;}
28: "(" {return L_PAREN;}
29: ")" {return R_PAREN;}

```

### 3 定義した言語で受理されるプログラムの例

```

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: }

```

## 4 コード生成の概要

### 4.1 メモリの使い方

データセグメントを `0x10004000` から開始し, テキストセグメントを `0x00001000` から開始させた. また, メモリは算術式において計算結果を一時的に確保する役割でスタックメモリを使用した. 宣言した変数や配列要素はそれぞれ 4 バイトと大きさを定めている.

### 4.2 レジスタの使い方

作成したコンパイラでのレジスタの使用用途を表でまとめる.

レジスタ	用途
v0	計算結果の格納
t0	データセグメントのアドレスの格納
t6	変数のアドレスの格納
t7,t8	変数の添字の計算
t1	スタックに格納して計算する値を格納
t3,t5	計算のため, スタックからロード
s0,s1	条件式の各辺の計算結果を格納
s2	条件式の結果を格納

### 4.3 算術式のコード生成の方法

EXPRESSION\_AST ノードが最初に現れたら `turn_on_expression` を実行して `expressioning` を 1 にする. そして `expressioning` が 1 の間に IDENT\_AST や NUMBER\_AST を訪問したら, 訪

問完了したらその値をスタックに入れていく。そして, `EXPRESSION_AST` ノードを訪問完了したら `add_node, sub_node` をしてスタックから取り出して計算する。また掛け算や割り算を行う `TERM_AST` は `EXPRESSION` よりも木構造的に下の位置に生成されるようになっており, そのノードの子ノードの探索を終えて, 兄弟ノードを探索する前にスタックに入れていけば, 演算子の優先順位を壊さずに計算を行える。なお, 代入や計算において, 演算子を伴わない算術式ではスタックに入れず, 直接レジスタに値を入れるようにしている。

## 5 特に工夫した点についての説明

### 5.1 記号表の作成

変数や配列を記号表を作成して管理した `define` や `array` が入力されたら記号表用の構造体に変数名, オフセット, インデックスなどの値を記録するようにした。具体的に記号表が持つ値を表でまとめる。記号表が `index` を持つのは整数の添字のときの効率のためである。

name	定義した変数の名前
next	次のデータへのポインタ
offset	データセグメントのオフセット
index	配列要素の添字
ide index	2次元配列用の添字ノード

### 5.2 配列要素の扱い

整数の添字の配列要素を扱う際, 直接オフセットを記号表から持ってくるようにした。整数の添字の配列要素を扱う方法として2つあると考える。具体的には今回, プログラムで実装した, 直接オフセットを記号表から参照する方法と, 配列の先頭のアドレスに添字の数だけ加算してアドレスを入手するという方法である。前者の方は記号表の項目を増やさなければならないが, 新たにアドレスを計算する過程がなくなるのでインストラクション数を少なくすることが出来ると考える。

## 6 コンパイラのソースプログラムのある場所

## 7 最終課題を解くために (定義した言語で) 書いたプログラム

### 7.1 課題 1

```

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: }
```

### 7.2 課題 2

```

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: }

```

### 7.3 課題 3

```

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: }

```

### 7.4 課題 4

```

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: }

```

## 7.5 課題 5

```
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: }
```

## 8 それをコンパイルして maps で実行したときの実行結果とステップ数

### 8.1 課題 1

メモリは以下ようになった.

```
1: 0x10004000 (268451840) = 0x0000000b (11)
2: 0x10004004 (268451844) = 0x00000037 (55)
3: 0x10004008 (268451848) = 0x00000000 (0)
4: 0x1000400c (268451852) = 0x00000000 (0)
5: 0x10004010 (268451856) = 0x00000000 (0)
6: 0x10004014 (268451860) = 0x00000000 (0)
7: 0x10004018 (268451864) = 0x00000000 (0)
8: 0x1000401c (268451868) = 0x00000000 (0)
```

インストラクション数は 408 になった.

## 8.2 課題 2

メモリは以下ようになった.

```
1: 0x10004000 (268451840) = 0x00000006 (6)
2: 0x10004004 (268451844) = 0x00000078 (120)
3: 0x10004008 (268451848) = 0x00000000 (0)
4: 0x1000400c (268451852) = 0x00000000 (0)
5: 0x10004010 (268451856) = 0x00000000 (0)
6: 0x10004014 (268451860) = 0x00000000 (0)
7:
```

インストラクション数は 233 になった.

## 8.3 課題 3

メモリは以下ようになった.

```
1: 0x10004000 (268451840) = 0x00000008 (8)
2: 0x10004004 (268451844) = 0x00000004 (4)
3: 0x10004008 (268451848) = 0x00000002 (2)
4: 0x1000400c (268451852) = 0x00000010 (16)
5: 0x10004010 (268451856) = 0x0000001f (31)
6: 0x10004014 (268451860) = 0x00000000 (0)
7:
```

インストラクション数は 3596 になった.

## 8.4 課題 4

メモリは以下ようになった.(一部省略)

```
1: 10004000: 000003e8 000001f5 00000003 000003e8
2: 10004010: 00000000 00000001 00000001 00000001
3: 10004020: 00000000 00000001 00000000 00000001
4: 10004030: 00000000 00000000 00000000 00000001
5: 10004040: 00000000 00000001 00000000 00000000
6: 10004050: 00000000 00000001 00000000 00000001
7: 10004060: 00000000 00000000 00000000 00000001
8: 10004080: 00000000 00000001 00000000 00000001
9: 100040a0: 00000000 00000001 00000000 00000000
10: 100040b0: 00000000 00000001 00000000 00000001
11: 100040c0: 00000000 00000000 00000000 00000001
12: 100040e0: 00000000 00000001 00000000 00000000
13: 100040f0: 00000000 00000000 00000000 00000001
14: 10004100: 00000000 00000001 00000000 00000000
15: 10004110: 00000000 00000000 00000000 00000001
16: 10004120: 00000000 00000000 00000000 00000001
17: 10004130: 00000000 00000001 00000000 00000000
18: 10004140: 00000000 00000000 00000000 00000001
19: 10004150: 00000000 00000000 00000000 00000001
20: 10004170: 00000000 00000001 00000000 00000000
21: 10004190: 00000000 00000001 00000000 00000000
22: 100041a0: 00000000 00000001 00000000 00000001
23: 100041b0: 00000000 00000000 00000000 00000001
24: 100041c0: 00000000 00000001 00000000 00000000
25:
```

インストラクション数は 413252 になった.

## 8.5 課題 5

メモリは以下ようになった.

```
1:
2: 10004000: 00000001 00000002 00000003 00000004
3: 10004010: 00000005 00000006 00000007 00000008
4: 10004020: 00000013 00000016 0000002b 00000032
5: 10004030: 00000002 00000002 00000002 00000000
```

インストラクション数は 1357 になった.

## 9 考察

### 9.1 再帰関数について

今回のプログラムでは `gen_code()` のみを再帰関数にしてノードをたどっていったが, 再帰関数を複数作成してノードを探索したほうが簡潔にプログラムをかけるのではないかと考える. どの再帰関数を実行しているかという情報があれば, `turn_on_assign` や `turn_on_define` などの関数を作成しなくても済んだと考える. 具体的には `turn_on_assign` や `turn_on_define`, `turn_on_expression` などは初めて `assign` や `define, expression` ノードを訪問したときに, それぞれのフラグを立てることを目的とした関数である. これら最初のノードの下ノードでは `assigning` や `defining, expressioning` といったフラグが立っているので, 立っているフラグに応じた処理を行うようにした.

### 9.2 expression

今回のプログラムでは

$$2 + 3$$

や

$$2 * 3$$

のような式の場合でも一度スタックに値をそれぞれ格納してからそれを取り出して計算しているが, 演算子が 1 つのときはスタックに入れずに直接値を保持してそれを計算したほうがインストラクション数を少なく出来ると考える.

### 9.3 配列の実装

今回, 配列を実装するにあたって, 配列の添字を `build_node0` を拡張して渡した. しかし, `IDENT_AST` のようにノードを作成して, そのノードが添字の値を持つようにしたほうが, 拡張性が高かったのではないかと考える.

## 9.4 記号表の意義

記号表を作成せずに、変数を記号表を使わずにラベル付を駆使して実装する方法もあるが、安全ではない実装であるを考える。これは変数名が `while` や `if` で使用するラベルと同じ名前になると、同じラベルが2つ存在するところになるからである。絶対にこのような問題を起きないようにするには、変数名を使用したラベルを作成する時に、特殊な記号を追加したりなどのプロトコルを実装する必要があると考える。

## 9.5 yacc ファイルで起きた還元競合

### 9.5.1 還元競合 1

`lex` ファイルの `assignment_stmt` において還元競合が起きてしまった。本来、2次元配列への代入は

```
1: IDENT L_BRACKET expression R_BRACKET L_BRACKET expression R_BRACKET ASSIGN expression SEMIC
```

とする予定であったが、このようにすると還元競合が起きてしまった。これは `var` における

```
1: IDENT L_BRACKET IDENT R_BRACKET
```

が `expression` に還元するのか、`I`

```
1: IDENT L_BRACKET expression (IDENT) R_BRACKET
```

に還元するのかがわからないので還元競合が起きたと考える。

### 9.5.2 還元競合 2

本来、配列要素へ `idents` で

```
1: |IDENT L_BRACKET IDENT R_BRACKET
```

とする予定であったが、このようにすると還元競合が起きてしまった。これはこの文が

```
1: idents L_BRACKET idents R_BRACKET
```

と還元するか `idents` と還元するかの2択が生まれてしまうからであると考えてる。

## 10 作成したプログラムのソースコード

作成したプログラムを以下に添付する。なお、??章に示した課題については、??章で示したようにすべて正常に動作したことを付記しておく。

```
1:
```

## 参考文献

- [1] 平田富雄, アルゴリズムとデータ構造, 森北出版, 1990.
- [2] 著者名, 書名, 出版社, 発行年.

[3] WWW ページタイトル, URL, アクセス日.