コンパイラ開発レポート

メンバー（五十音順）

e18-5405　久保徹朗　　e18-5406　齋藤瑞樹

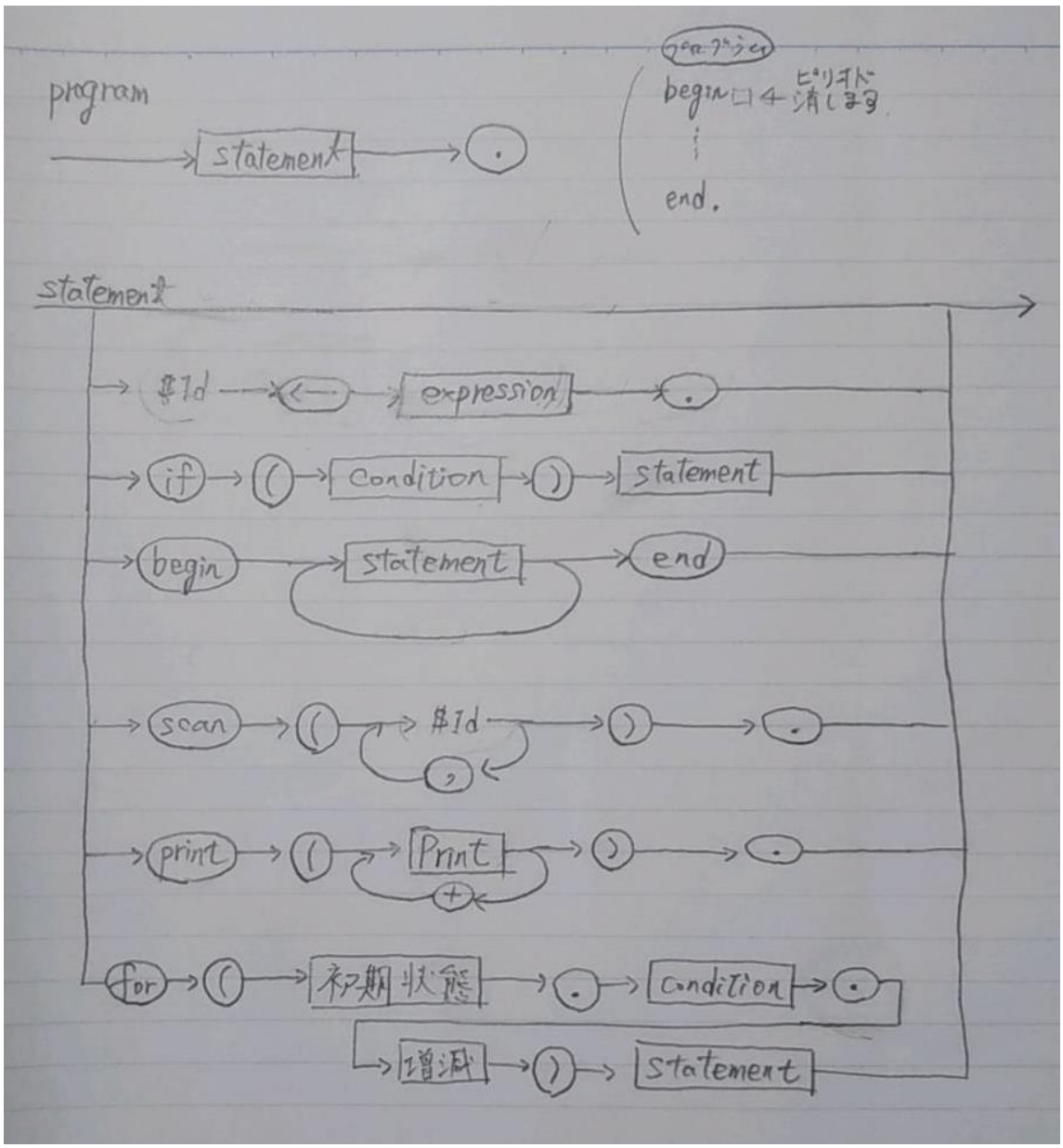
e18-5405　玉井紀光　　e18-5405　美濃谷拓郎

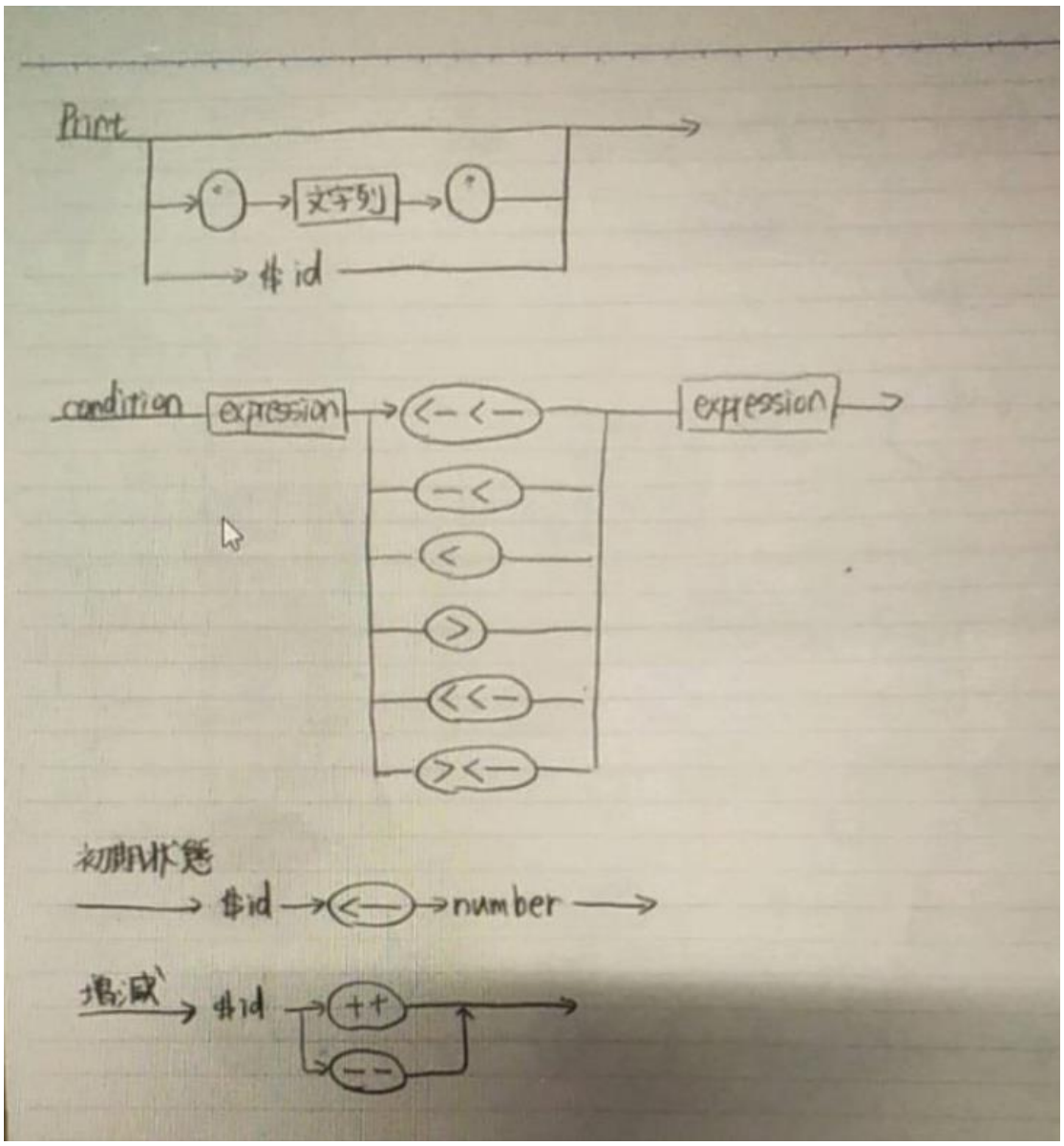
□□独自言語の仕様

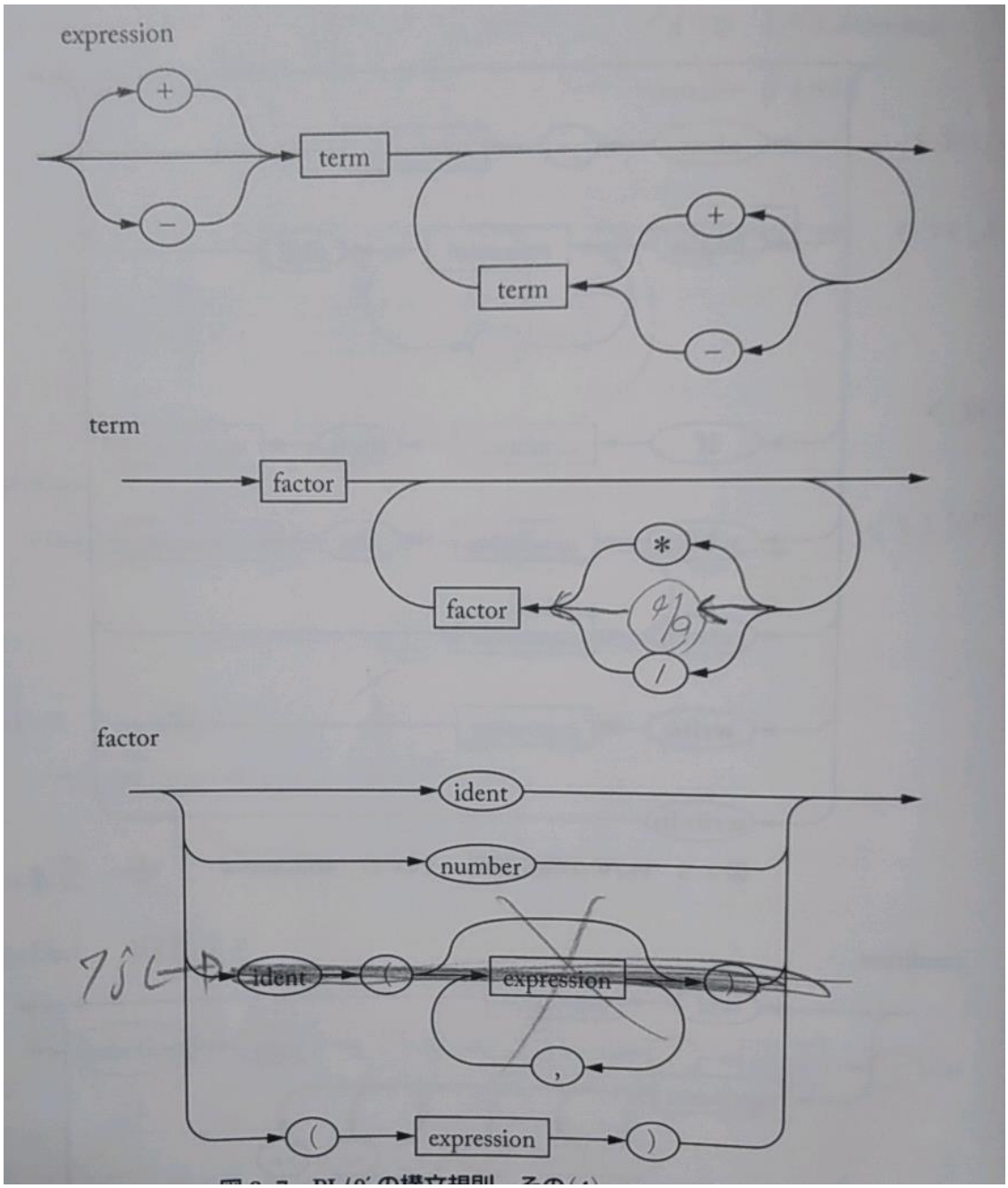
□字句の定義

操作	自作言語	c
＜予約語＞		
・開き中カッコ	begin	{
・閉じ中カッコ	end	}
・スキャン	scan	scanf
・プリント	print	printf
・if	if	if
・for	for	for
・変数	$ a	a
<演算子とその他の記号>		
・代入	<-	=
・比較	<-<-	==
・でないならば	-<	!=
・≦	<<-	<=
・≧	><-	>=
・＜＞は同じ		
・演算子は c と同じ（％も）		
・最後	.	;
・改行	¥¥	¥n
・はじめと終わりは　begin~end.　とする		

□文法の定義（構文図式と BNF）




Print
'
文字列
'
#id
condition
expression
<-<-
-<
<
>
<<-
>=<-
expression
初期状態
#id
<-
number
増減
#id
++
--


expression
+
−
term
term
+
−
term
factor
factor
*
/
factor
ident
number
ident
(
expression
)
,
(
expression
)

BNF

No.
Date . .

Program

<program> ⟶ <statement> "."

statement

<statement> ⟶ ε | $id "<—" <expression> "." |
"if" "(" <condition> ")" <statement> |
"begin" <statement> <addstatement> "end" |
"scan" "(" #id <idlist> ")" "." |
"print" "(" <print> <printlist> ")" "." |
"for" "(" <初期状態> "." <condition> "." <増減> ")"
<statement>

addstatement

<addstatement> ⟶ <statement> <addstatement> | ε

idlist

<idlist> ⟶ "," #id <idlist> | ε

print

<print> ⟶ ε | "'" <文字列> "'" | #id

printlist

<printlist> ⟶ "+" <print> <printlist> | ε

condition

<condition> ⟶ <expression> <conditionlist> <expression>

Conditionlist

<conditionlist> ⟶ "<-<-" | "-<" | "<" | ">" | "<<-" | ">><-"

Date

初期状態

<初期状態> → #id "<-" "number"

増減

<増減> → #id "++" | #id "--"

expression

<expression> → <mark><term><expressionlist>

mark

<mark> → ε | "+" | "-"

mark2

<mark2> → "+" | "-"

expressionlist

<expressionlist> → <mark2><term><expressionlist> | ε

term

<term> → <factor><termlist>

termlist

<termlist> → <mark3><factor><termlist> | ε

mark3

<mark3> → "*" | "%" | "/"

factor

<factor> → "ident" | "number" | "(" <expression> ")"

□意味の定義

✓始めと終わりは
begin
・・・・
end.
とする。

✓begin は c 言語でいう ｛ （開き中カッコ）
　end は c 言語でいう ｝ （ 閉じ中カッコ）

✓変数は定義無しで適宜使える。

✓for 文は

for(ＢＮＦ参考)
begin
・・・
end

とする

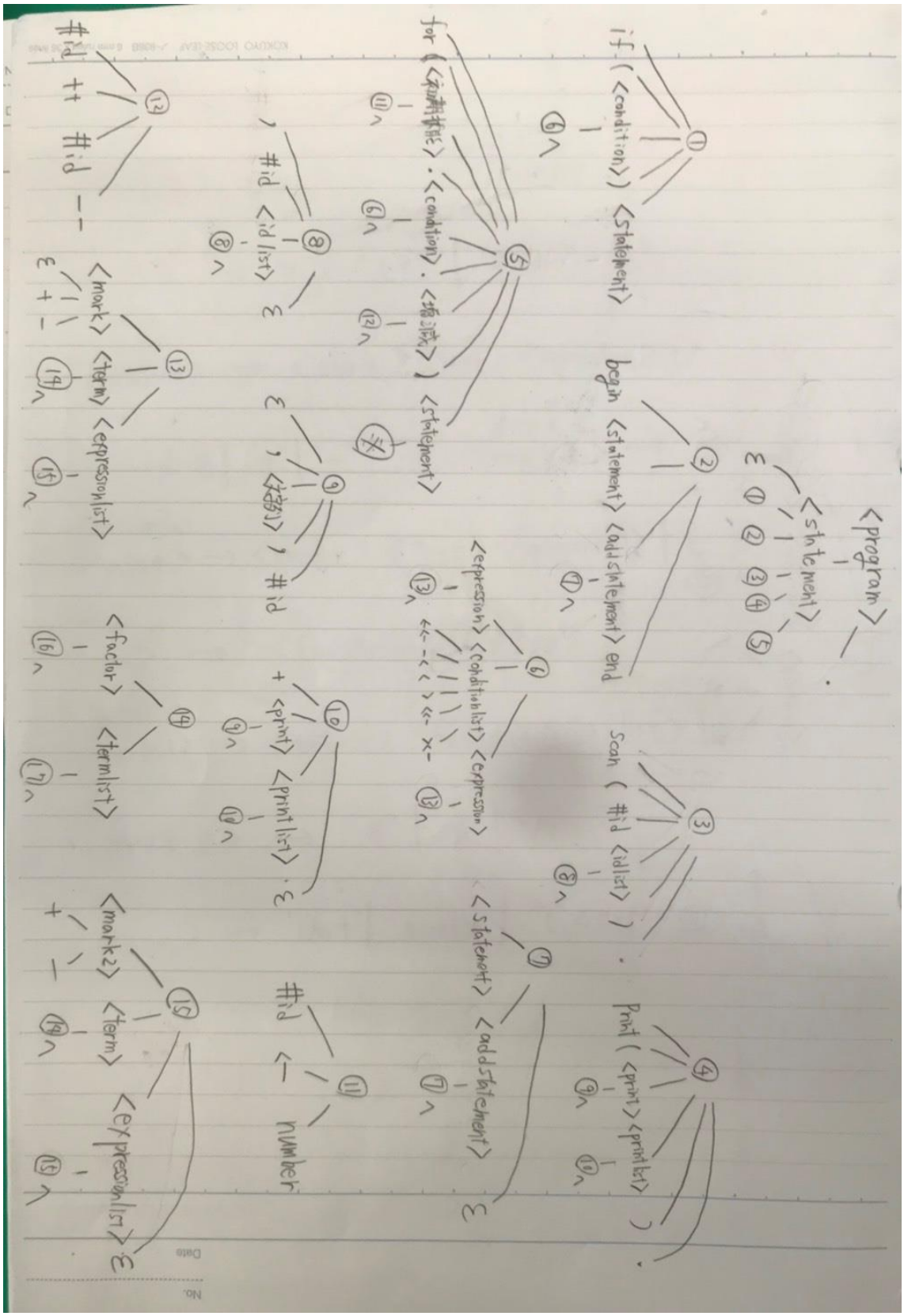
✓改行（¥¥） は print（“¥¥”）.
と使い。print(“moziretu¥¥”). とは使えない。但し、print（“文字列”+”¥¥”）. とはできる。

□□プログラムの１例

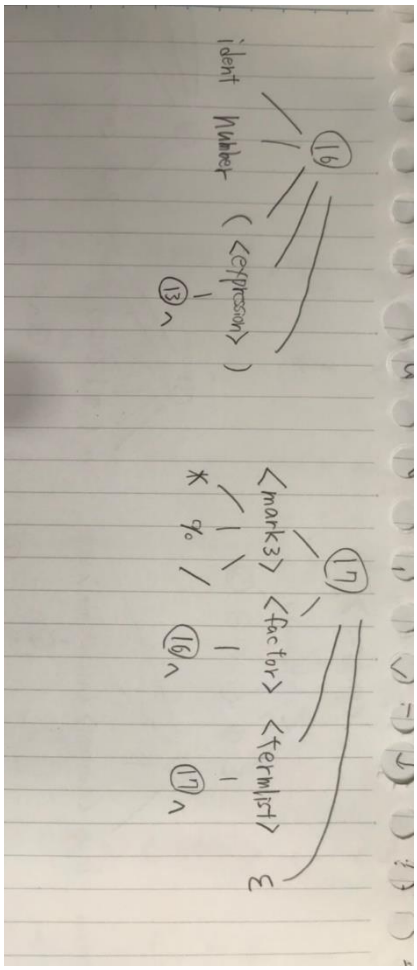
□最大公約数を求める独自言語のプログラム

```
begin
 print("num1"+"¥¥").
 scan($a).
 print("num2"+"¥¥").
 scan($b).

 if($a<$b)
 begin
  $tmp<-$a.
  $a<-$b.
  $b<-$tmp.
 end

 $r<-$a%$b.

 for($i<-0.$r-<0.$i++)
 begin
  $a<-$b.
  $b<-$r.
  $r<-$a%$b.
 end


 print("G.C.D is"+$b+"¥¥").
end.
```

□独自言語プログラムのサンプル解析木



続き・・・



□□コンパイラ開発

□字句解析(＊コンパイラの教科書参考)

字句解析は main.c と others.c で実現されている。others.c には字句解析と意味解析に使われる新しい型の定義や、その型の変数の定義・宣言、記号表に関する変数の定義がある。main.c は自作言語のプログラムを予め定義した数値に置き換える（字句解析）のプログラムである。エラー検出も 2 つ組み込んだ。

（新しい型とその変数を詳しく説明すると・・・）

字句解析を行うに当たり、いくつか型を定義した。

✓KeyId 型（others.c に定義）

予約語、演算子、その他記号、変数、数、文字列などに数値を割振りふった。 Struct と Enum で実現した。字句解析で完成した、数値コードは、keyId 型（新しい型と書いたが実際は integer 型）で表現されている。

以下は、「予約語」、「全ての定義した記号」に割り振った番号の表。

予約語の名前

Begin 0
End 1
Endp 2 いらん
Scan 3
Print 4
Else 5 いらん
If 6
For 7
end_of_KeyWd 8

演算子と区切り記号の名前

Dollar 9 (&)
Assign 10 (<-)
Compare 11 (<-<-)
Equal 12 (=)使ってない
Lparen 13 (
Rparen 14)
Period 15 (.)
Double_Quotation 16 (")
Plus 17 (+)
Minus 18 (-)
Mult 19 (*)
Div 20 (/)
Remainder 21 (%)
Backslash 22 (¥)使ってない
Ln 23 (¥¥)
NotEq 24 (-<)
Comma 25 (.)
PlusPlus 26 (++)
MinusMinus 27 (--)
Lss 28 (<)
Gtr 29 (>)
LssEq 30 (<<-)
GtrEq 31 (><-)
end_of_KeySym 32

トークンの種類

Id	33
Num	34
nul	35
end_of_Token,	36

上記以外の文字の種類

letter	37	文字列
digit	38	いらん
colon	39	いらん
others	40	
mmm	41	

・下は、gcd を字句解析して得られた、keyId 型で表現されたコード。zisaku_token ファイルに出力される。

```
0
4 13 16 37 16 17 16 37 16 14 15
3 13 33 14 15
4 13 16 37 16 17 16 37 16 14 15
3 13 33 14 15

6 13 33 28 33 14
0
33 10 33 15
33 10 33 15
33 10 33 15
1

33 10 33 21 33 15

7 13 33 10 34 15 33 24 34 15 33 26 14
0
33 10 33 15
33 10 33 15
33 10 33 21 33 15
1



4 13 16 37 16 17 33 17 16 37 16 14 15
1 15
```

keyId の変数として、keyId　charClassT[256]を置いた。（mian.c）内容については

```
charClassT['0']～charClassT['9']を digit
charClassT['A']～charClassT['Z']を letter
charClassT['a']～charClassT['z']を letter

charClassT['$']=Dollar;
charClassT['"']=Double_Quotation;
charClassT['%']=Remainder;
charClassT['+'] = Plus; charClassT['-'] = Minus;
charClassT['*'] = Mult; charClassT['/'] = Div;
charClassT['('] = Lparen; charClassT[')'] = Rparen;
charClassT['='] = Equal; charClassT['<'] = Lss;
charClassT['>'] = Gtr; charClassT[','] = Comma;
charClassT['.'] = Period;
charClassT['¥¥'] = Backslash;
charClassT['¥n'] = mmm;//終了合図

cahrClassT['その他']を others
```

と置いた。

✓struct keyWd 型　字句解析で検出した文字列がどの予約語かを判断するのに使われる。

✓struct keyWd2 型　意味解析で使われる。主に、字句解析で作られた数値コードが c 言語のどれにも対応するか使われる。

（mian.c で使われる関数に関して・・・）

・nectChar()
　自作言語のプログラムの次の 1 文字を返す関数。nextToken()から呼ばれる。

・nextToken()
　nectChar()から返された文字を使って、“「予約語」、「全ての定義した記号」に割り振った番号の表”と照らし合わせて、keyId 型のコードを出力する関数。mina()から呼び出される。

□構文解析

構文解析は koubun.c で実現されている。koubun.c は字句解析で作られたコードファイル（zisaku_token）を使って、自作言語のプログラムの構文上のエラーがないか確認するプログラムである。コンパイラの教科書に沿って、作成したので特に説明は要らないと思う。エラーを検出したときに、どこでエラーが発生したのかをできる限り示すようにした。

□意味解析

意味解析は、means.c と others.c で実現されている。means.c はコードファイル（zisaku_token）と記号表を使って、自作言語のプログラムを c プログラムに変換するプログラムである。C プログラムは cLang.c に書き込まれるようにした。

プログラムの考えとしては、switch－case 文で zisaku_token ファイルの数値コードをコードの終わりまで回して、最後に end. が来た時に終わるよにした。Swich-case 文の選択肢は

case Begin
case End
case Scan
case Printf
case For
case Id
case Num
default de()

とした。

例えば、case For では以下のプログラムで数値コードが処理されていく。

```
case For:
        Token2=getToken2();//(とる
        Token2=getToken2();// .or その他をとる
        fprintf(fl4,"for(");

        while(Token2!=Period){
          de();

          //printf("%d¥n",Token2);
        }

        fprintf(fl4,";");
        Token2=getToken2();
```

de()に関して、de()は数値コードが上に入っていないとき、コードと対応する c プログラムをそのまま出力する関数。これには、字句解析で述べた、struct keyWd2 型の変数 KeyWdT2[添え字]が使われている。

・means.c で使われているその他の関数について、

getToken2()　：zisaku_token ファイルから数値コードを読み込み・出力する関数。

put_IdstrTb()：変数文字列表を参照して、変数を cLang.c に出力する関数。c プログラムで使われる変数を始めに示す時に使われる。(Mian(){ int a,b,c;のところ)

put_outIdTb()：変数出力表を参照して、変数を cLang.c に出力する関数。変数を意味する数値コードが出て来くたびに実行される。

・変数、文字列、数値を記憶した配列について、

意味解析を実現するために、字句解析で変数、pirnt で使われる文字列、数値を来た順にそれぞれの配列記憶していった。当該配列は以下の 3 つである。

outIdTb[]　：自作言語のプログラムで使われる変数名を来た順に格納する Char 型の配列。 strTb[]　　：自作言語のプログラムの print で使われる文字列を来た順に格納する char 型の配列。 numTb[]　：自作言語のプログラムで使われる数値を来た順に格納する integer 型の配列。

例えば、gcd を求める自作言語のプログラムにおいて、

outIdTb[]には a&b&a&b&tmp&a&a&b&b&tmp&r&a&b&i&r&i&a&b&b&r&r&a&b&b&

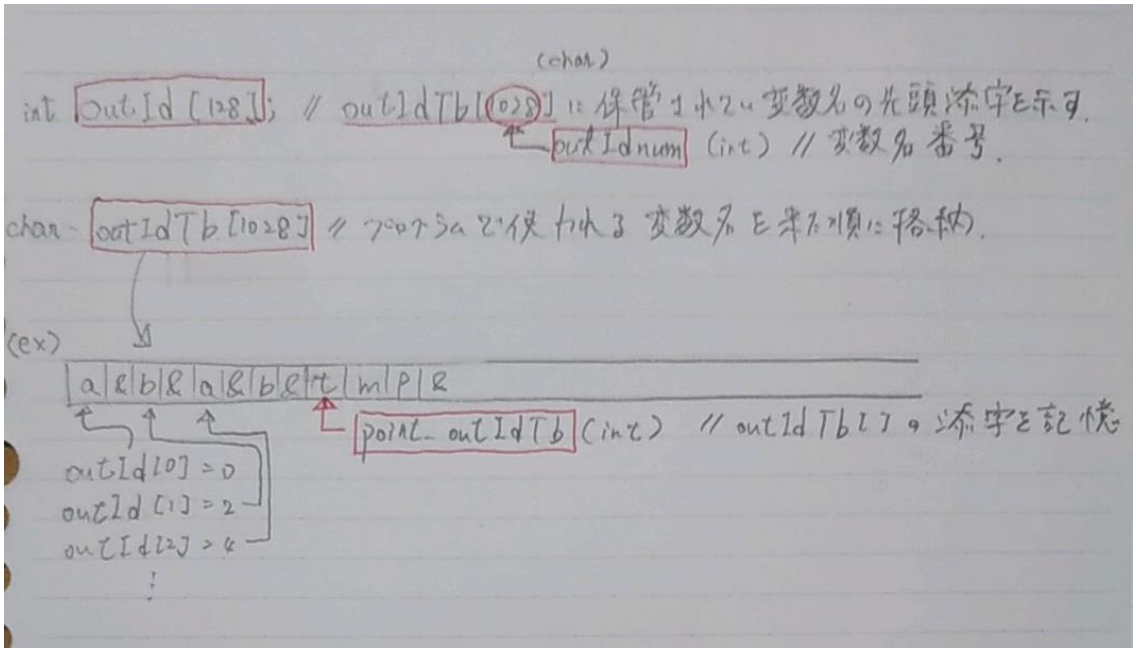
strTb[]には num1&¥¥&num2&¥¥&G.C.D is &¥¥&

numTb[]には　numTb[0]=0 numTb[1]=0

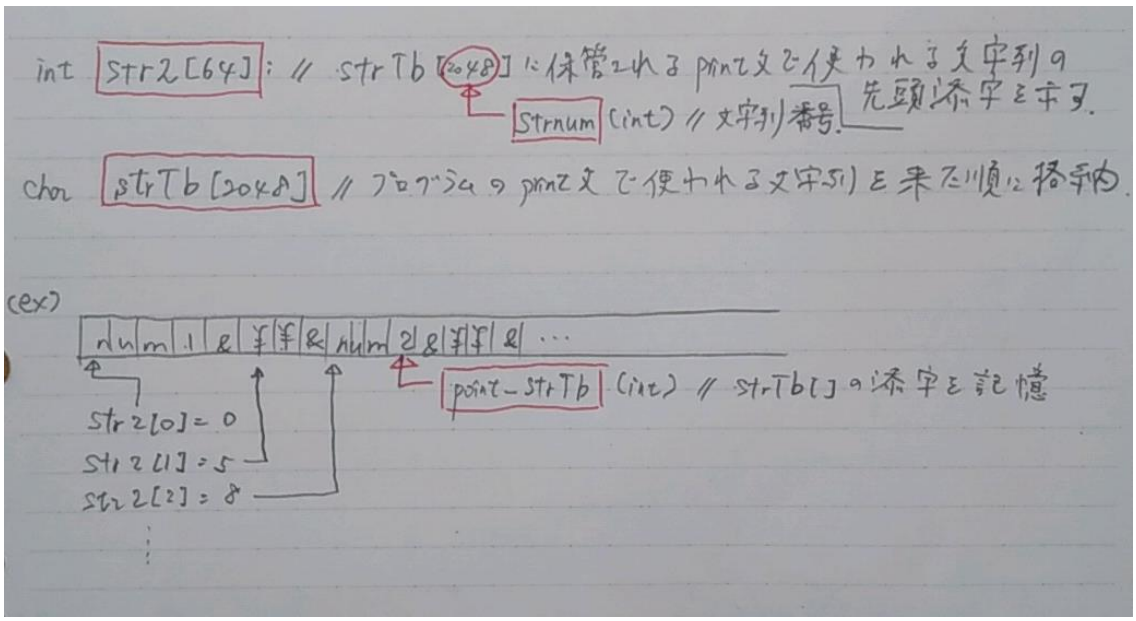
が格納されている。

・それぞれの配列に関わる変数、その意味、関係を以下に示す。

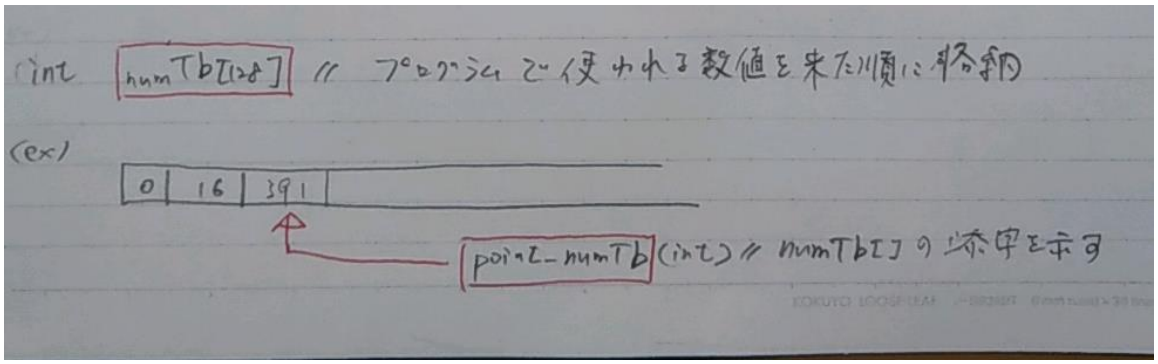
・outIdTb[]



・strTb[]

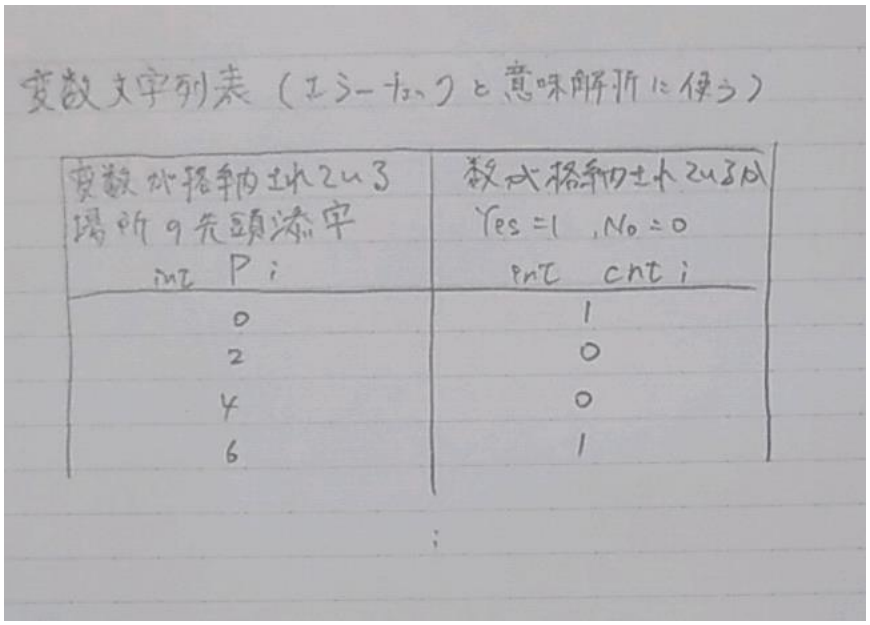


・numTb[]



□記号表

記号表は 1 つ作成した。変数文字列表と称する。表は以下の通り。



・変数文字列表を作るのに新しく型（Id_Tb 型）を定義した。

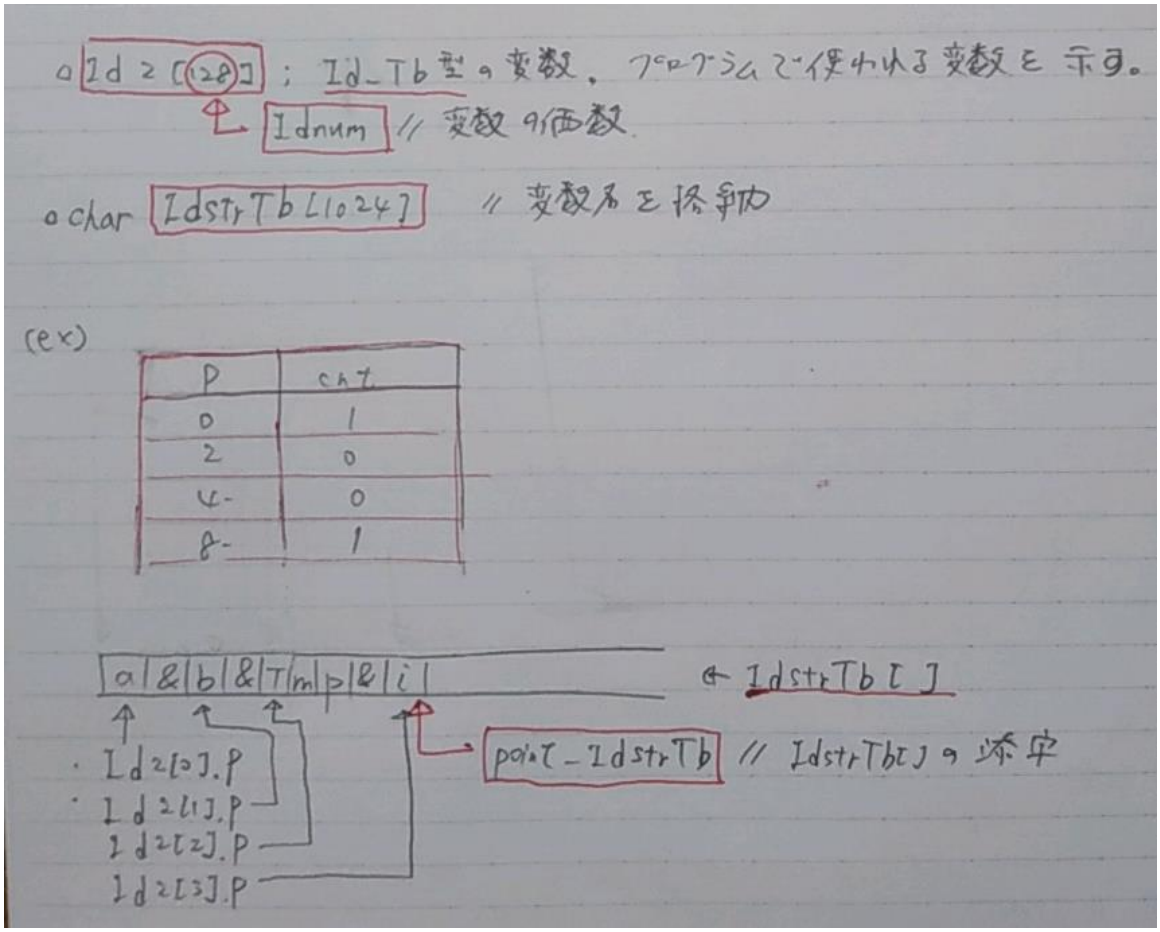
```
typedef struct Tb{
    int p;
    int cnt;
}Id_Tb;
```

変数は Id_Tb　Id2[128];を置いた。

しかし、本コンパイラでは、IdstrTb[]は意味解析のみに使われる。エラー検出でも使えるように実装したが、今回はエラー検出の部分は使用しない(cnt のこと)。エラー検出の部分

を消さない理由は、今後改善するときに、使用するためである。

・変数文字列表を実現するために使用した、変数、その意味、関係を以下に示す。



□エラーに関して

　エラーは 3 種類とする。

エラー①：予約語のスペルを間違えた時。

予約語が間違っていることを教える。例えば、標準出力は print が適切であるが、printf になっていた時に予約語が間違っていることを教える。字句解析の時に、発見するものである。

　仕組みに関しては、文字列は、

begin　0

end　　1

scan　 3

print　4

if　　6

for　　7

のどれかであるが、辿った文字列がどれにも当てはまらなかった時にエラーを返す。エラーコード１とする。

エラー②：定義されていない記号（例えば、；や：や,など）が記述されていた時、定義されていないものが使われていることを教える。
例えば、最後を　.（ドット）でなく；(セミコロン）にしたときに、定義されてていないものが使われていることを教える。字句解析に導入されている。
　仕組みに関しては、上と同じで、辿った文字または文字列が定義された記号でなかった時エラーを返す。エラーコード０とする。

エラー③：文法上のミスが合った時。
　例えば、print 文は文字列は“”で囲むが、print(“iii);のときに print 文が間違っていることを教える。構文解析に組み込んだ。
　仕組み関しては、例えば、print では

```
void print(){
      ・・・略・・・
     if(token2==letter){
        token2=getToken();
     }else{
        error2(17);
     }



        if(token2==Double_Quotation){
           token2=getToken();
        }else{
           error2(18);
        }

        break;
```

であるが、letter の後に　“　がない時に、error2(18)が組み込まれている。
エラーコードは２から２７とする。(koubun.c ファイル の error２(int i)関数にあります。)

□□動作例

①最大公約数を求める

・プログラム

略・・・

・実行例

```
student@debian:~/compiler2$ gcc main.c others.c koubun.c means.c
[1]-  終了                  emacs test6
student@debian:~/compiler2$ ./a.out
自作言語を書き込んだファイル名を入力してください。
test5
字句解析終了
構文解析終了
意味解析終了
正常終了です。
zisaku_token に自作言語の token が書き込まれています。
cLang.c に命令が書き込まれています。実行してください。
student@debian:~/compiler2$ gcc cLang.c
student@debian:~/compiler2$ ./a.out
num1
36
num2
12
G.C.D is 12
```

②右下直角二等辺三角形を作る

・プログラム

```
begin

 print("we make triangle"+"¥¥").
 print("short edge"+"¥¥").
 scan($len).

 for($i<-1.$i<<-$len.$i++)
 begin
  for($j<-1.$j<<-$len-$i.$j++)
  begin
   print(" ").
  end
  for($j<-1.$j<<-$i.$j++)
  begin
   print("*").
  end
 print("¥¥").
 end
 print("great!!!!"+"¥¥").
end.
```

・実行例

```
student@debian:~/compiler2$ gcc cLang.c
student@debian:~/compiler2$ ./a.out
we make triangle
short edge
11
          *
         **
        ***
       ****
      *****
     ******
    *******
   ********
  *********
 **********
***********
great!!!!
```

③入力した数字の約数とその個数を求める

・プログラム

```
begin
  print("put integer value"+"¥¥").
  scan($n).
  $count<-0.

  for($i<-1.$i<<-$n.$i++)
  begin
    if($n%$i<-<-0)
    begin
      print($i+"¥¥").
      $count<-$count+1.
    end
  end

  print("the number of divisor is "+$count+"¥¥").



end.
```

・動作例

```
student@debian:~/compiler2$ gcc cLang.c
student@debian:~/compiler2$ ./a.out
put integer value
12
1
2
3
4
6
12
the number of divisor is 6
```

④読み込んだ２つの整数値の小さい数から大きい数までの和を求める

・プログラム

```
begin

 print("integerA:"+"¥¥").
 scan($a).
 print("integerB:"+"¥¥").
 scan($b).

 if($a>$b)
 begin
  $lower<-$b.
  $upper<-$a.
 end
 if($a<<-$b)
 begin
 $lower<-$a.
 $upper<-$b.
 end

 $sum<-0.
 $no<-$lower.
 for($i<-0.$no<<-$upper.$i++)
 begin
  $sum<-$sum+$no.
  $no<-$no+1.
 end
 print($lower+"以上"+$upper+"以下の全整数の和は"+$sum+"ですよ。").
end.
```

・動作例

```
student@debian:~/compiler2$ gcc cLang.c
student@debian:~/compiler2$ ./a.out
integerA:
10
integerB:
1
1 以上 10 以下の全整数の和は 55 ですよ。student@debian:~/compiler2$
```

ここからエラーの動作例（gcd を求めるプログラムを使う）

⑤文法上の間違い

・プログラム

```
begin
  print("num1"+"¥¥").
  ・・・略・・・
  print("G.C.D is "$b+"¥¥").
end.
```

print 文は文法上、文字列と文字列または変数を繋げるときは＋をいれる。今回は、＋を除いて入れてコンパイラした。

・動作例

```
自作言語を書き込んだファイル名を入力してください。
test5
字句解析終了
print 文に関するエラー。
終了します。error code(11)
```

⑥定義されていない記号が使われた時

・プログラム

```
begin
・・・略・・・
begin
   $tmp<-$a;
   $a<-$b.
・・・略・・・
end.
```

プログラム 1 文の最後は ． で終わる。しかし、今回は定義されいていない　；　を使ってみた。

・動作例

```
自作言語を書き込んだファイル名を入力してください。
test5
予約語にない文字列や定義されていない記号が使われています。error code(0)
```

⑦予約語のスペルが間違っている時

・プログラム

```
begin
  printf("num1"+"¥¥").
  scan($a).
  ・・・略・・・
end.
```

print をスペルミスで printf にしたとき。

```
自作言語を書き込んだファイル名を入力してください。
test5
予約語のスペルが間違っています。error code(1)
```

□注意点

✓お手数ですが、自作言語のプログラムは改行で終わらせてください。

Begin

・・・

End.(改行しない)では正しく実行できませんので、よろしくお願いします。

✓自作言語のプログラムは任意名前のファイルに書き込んでください。ファイル名は、後に標準入力でプログラムが認識することになります。プログラム実行中にファイル（zisaku_token）に字句解析で出力されたものが作られます。ファイル名はそれ以外でお願いします。

✓ c 言語への出力は cLang.c に書き込まれます。別に実行してください。

✓実行の時は、

gcc main.c　others.c　koubun.c　means.c でコマンド命令してください。

□□考察

久保徹朗

プログラムは概ね意図した通りに動いた。当初理想としていた形に持っていけたのは良かった。しかし、今回のプログラムではセグメンテーションフォルトが出てしまう時がある。変数を格納できる数は、128 コ。文字列を格納する領域は 1024 バイトのみである。それを超えた時、領域不足でセグフォが発生してしまう。コンパイラのテスト時は、変数や文字列を大量に書くことをしなかったので、セグフォが確認ず、完成後までこの欠陥に気がつかなかった。他に作っていたプログラムで配列によるセグフォが起こりこの欠陥が明るみになった。この解決方法としては、malloc、calloc を利用し、動的配列を使うことが考えられる。この出来事より、ユーザーの操作によって配列の利用状況か変わる時は、動的配列を使う必要があることを強く実感した。同じミスをしないために、しっかり覚えておこうと思う。

その他の反省点は、作成した自作言語の機能が少なかったことが挙げられる。例えば、私たちの言語には関数概念がない。また、else を利用できる予約語としたが、それを実装するしなかった。（予約語として else は定義されたままだ）そして特に、エラー検出に関しては内容が非常に乏しくなってしまった。今回のエラーは、

①予約語が間違えた時にそれを知らせる。

②定義されていない記号が使われていない時にそれを知らせる。

③文法上のエラーが発生した時にそれを知らせる。

の 3 点だ。エラー内容が乏しいと感ずるのは、「①予約語が間違えた時・・」は、「②定義されていない・・」と同義に捉えるのこともできてしまうからだ。また、③では明確な場所を示すことが出来なかったので、それも内容不足と感じる 1 つの原因だ。エラーの 1 例として先生は、「変数に値が格納されていない状況でその変数が利用された時にエラーを出力する機能」を挙げられていたが、それを実現出来なかったことが残念である。そして、それを実現するために変数文字列表を用意したが、それ使ずじまいになってしまったのが口惜しい。

反省点が考察の大部分を占めたが、実際に私がテストしたプログラムは全て意図した通り通りに動いた。自作言語の機能は少ないものの動作は、しっかりコンパイラされるのでそこは良かった。

自作言語に改善の余地が有り余っている。今後、時間が取れればコンパイラを改修していきたいと思う。

齋藤瑞樹

齋藤瑞樹

意味解析を担当した。その中で、今回変数を定理するにあたって事前に定義せず、都度"$"を頭に着けることで変数の定義を行った。これが通常とは違う開発を行うことになった。記号表を作る必要性がなく、これにより意味解析でエラーの検出が少なくなった。これが良い

ことかといわれるとそうでもなく、使われていない・値が代入されていない変数を急に使われてしまった場合、エラーとして検出されなくなってしまっている。これでは、途中から誤って存在しない変数が使われていたとしてもそのまま計算を行ってしまい、コンピュータに大きな負荷をかけてしまう。
このような場合でも、エラー検出ができるようにもっと考察をしながら開発を行うべきであったと思われる。コンパイラ開発の難しさを思い知らされた。

玉井紀光

意味解析を担当したが自作言語を C 言語に変換することしか実装できなかった。本来なら、意味解析で記号表を基にエラーを検出したかった。今回は int 型しか定義しなかったため記号表を作る必要性があまりなく、しっかりと作らなかった。また、エラーに関しては字句解析と構文解析組に丸投げにしてしまい反省が残る。

Int 型だけでなく float 型や関数を使った自作言語の実装ができたらすごいなと感じた。

美濃谷拓郎

私は主に構文解析を担当した。BNF、構文木、それを参考にしたプログラムとあるが、全て完璧に作用しないと言語が成り立たないため、非常に重要なブロックであることを痛感した。
特に解析木に関しては、ループの表現が非常に複雑。おそらく間違っている。
今回はエラーも 3 つほどしか実装されておらず、できることも限られており、ユーザビリティへの配慮を考えると、これ以上に多様な構文の導入、解析が必要になる。解決方法ではないが、代替案として構文解析に構文木作成（抽象構文木）を取り入れることが考えられる。実際に紙に書いて視覚化したほうが人間が後々手を加える時に都合がよいかもしれない。だがプログラムでやった場合は、プログラムの近くに(//)を用いて説明を入れておけば後々対処のしやすいコンパイラ開発ができるため、そこはやっておけばよかったという反省点としてあげる。時間があるときにプログラム化ができないか検討してみたいと思う。
エラーメッセージに関して、本来は意味解析の段階でエラーを出力するのが一般的（？）なのかもしれないが、字句解析と構文解析の段階でエラーメッセージの表示を実現した。できるだけどの場面で、どの記号が間違っているのか、どう直したらよいかを印字できるよう工夫してみた。プログラムの使用上できることは限られているため、できる範囲でエラーメッセージのバリエーションを増やせたところは評価したい。

□□感想

久保徹朗

今回、始めてグループで１つの大きなプログラムを作成することを経験しました。
コミュニケーションの難しさを感じました。
GitHub などのバージョン管理システムを使わなかったので、プログラムの共有が遅くなってしまったことが反省です。次からは使いたいです。
今後、チームでプログラムを作る機会が出てくると思うので、今回は良い練習となりました。

齋藤瑞樹

今回のコンパイラ開発レポートの課題において自分が何かチームのために慣れたかというとほとんど力になれていなかったと思います。浅はかな知識しかなく、チームメイトに多く助けられました。自分一人では課題に取り組むことはできなかったと思います。
今後、このようなチームで何かを開発する課題があるときはチームの足を引っ張らないようにしっかりとした知識を身に着けて挑みたいと思います。

玉井紀光

自分は自分のできるところを担当したので難しいところは周りの人にかなり任せきりになってしまいました。これを一から一人で作れとなった時に作り上げる自信はありません。グループワークにおいて、平等な分担作業をしなければならなかった、もっと積極的に参加しなければならなかった、と反省しています。それと同時にチームメイトに感謝しています。
今後もグループワークの機会があると思うので、その時には今回の反省を生かして臨みたいと思いました。

美濃谷拓郎

時間もない中、まず一つのものを仕上げることができて安心しています。しかし、作成量や手間の部分を見ると、主にプログラム開発を率先してやってくれた久保君に多大な負担をかけてしまったことを反省しています。グループでやる以上、もう少し負担を分担できたのではないかなと思います。そこから最初の計画の時点でその後の制作の見通しを十分に立てることの大切さを学びました。今後もグループ活動は必ずあるので、今回の成功・失敗経験を生かせるようにしたいです。