2019

プログラミング言語論とコンパイラ

独自言語作成レポート

作成者

E18-5402　上岡　新平

E18-5408　佐藤　央

目次

1.開発した独自言語の仕様

１．１　概要

・ユークリッドの互除法の実現ができることを最低限の目標とした、四則演算と除算時の余り計算,
及び自由な変数名が設定可能な言語を開発。

1.2　命令

入出力
print　printf に相当．文法は
　　print $hogehoge~　←変数出力
　　print _fugafuga_　←固定文字列出力

get scanf に相当．文法は
　　get $hogehoge~

宣言
def C 言語の int 宣言に準じる．文法は
　　def $hogehoge~

制御系
if C 言語の if と同様の動作を行う．

else C 言語の else と同様の動作を行う．

while C 言語の while と同様の動作を行う．

比較演算子
+ - * / % < > = == ! && ||は C 言語と同等の働きをする．

１．３　実装

今回は Python で実現した．
詳細な実装は後述する．

1.4　制約

・プログラムの始まりには START，終わりには END/n（END のあとに改行）が必要である．
・一行につき 1 命令，また行末には;をつけなければならない．
・print def get の引数を指定しないとコンパイラ自体がエラーを起こす．

1.5 サンプルプログラム

以下にユークリッドの互除法を実現する sample source を示す

```
START
def $abc~;
def $b~;
def $r~;
def $tmp~;

get $abc~;
get $b~;

if($abc~ < $b~){
	$tmp~ = $abc~;
	$abc~ = $b~;
	$b~ = $tmp~;
	}
$r~ = $abc~ % $b~;
while($r~ != 0){
	$abc~ = $b~;
	$b~ = $r~;
	$r~ = $abc~ % $b~;
}

print $b~;
END
```

1.6 文法定義（BNF）

以下に独自言語の文法を BNF で記述する。なお number は 0 以上の整数、id は英数字。

<program> -> “START”<statements>”END”

<statements> -> <G state><statements>|<D state><statements>|<I state>
 <statements>|<E state><statemens>|<W state><statements>|<P state>
 <statements>|<S state><statements>| ε

<G state> -> “get” <var><fin>

<D state> -> “def” <var><fin>

<I state> -> “if””(”<condition>”)””{”<statements>”}”

<E state> -> “else””{”<statements>”}”

<W state> -> “while””(”<condition>”)””{”<statements>”}”

<P state> -> “print” <var><fin>|”print”<>

<S state> -> <var> “=” <exp><fin>

<condition> -> <exp><compare><exp> <conditions><fin>

<conditions> -> <&&||記号> <condition>| ε

<exp> -> <term><terms>

<term> -> <factor> <factors>

<terms> -> <加減符号><term><terms>| ε

<factor> -> <var>|”(“<exp>”)”|number

<factors> -> <乗除符号><factor><factors>| ε

<var> -> “$” id “~”

<compare>　-> “=”|”>”|”<”|”>=”|”<=”|”==”|”!=”

<加減符号>　-> “+”|”-”| ε

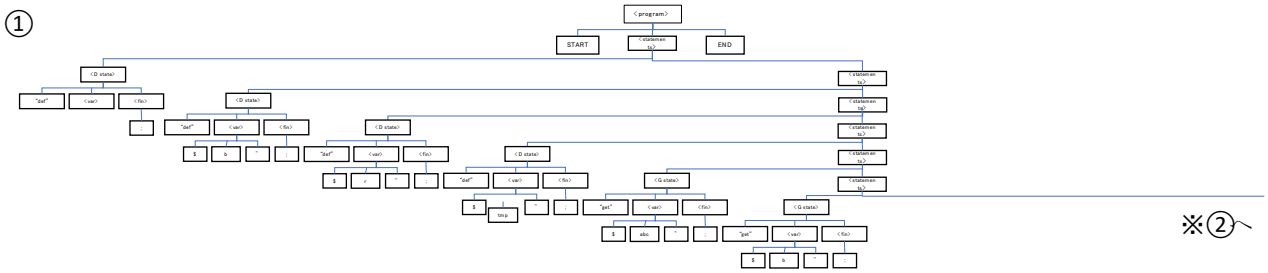
<乗除符号> -> “*”|”/”|”%”| ε

<&&||記号> -> “&&”|”||”| ε
<fin> -> “;”| ε

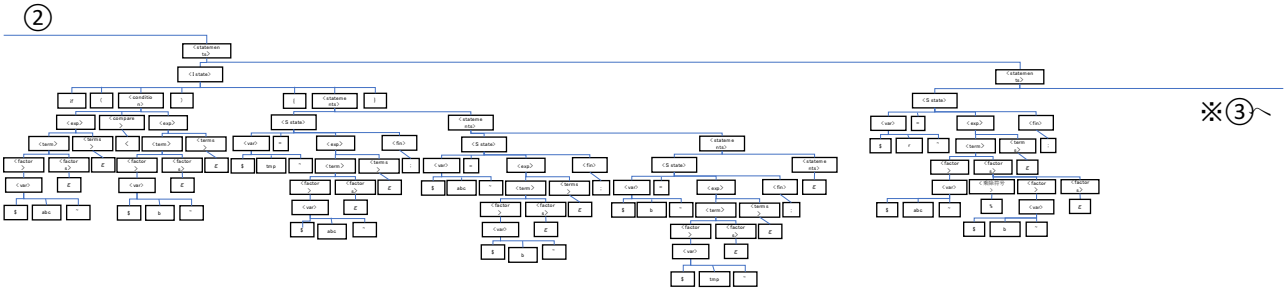
1.7 解析木

サンプルプログラムの解析木を以下に示す.なお解析木は①②③の三つに分けて載せた.

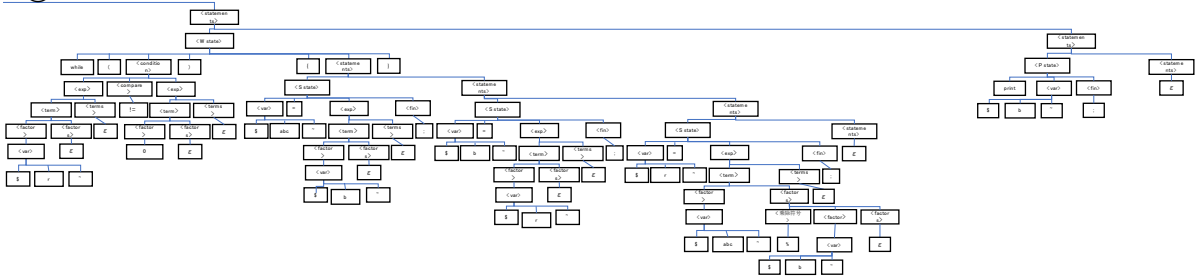
①



②



③



2.コンパイラの開発

2.1 字句解析

本コンパイラでは，外部に保存された txt ファイルを Python の readlines 関数を用いて字句解析を行っている．読み込む段階で，ソースコードは行ごとに配列に代入される．

```
# Reading source code
path1 = "samplesource.txt"
file1 = open(path1, 'r', encoding='utf-8')
f1 = file1.readlines()
file1.close()


fileobj = open("Source.C", "w")
```

2.2 構文解析と意味解析

読み込まれたソースコードは行ごとに解釈される．
if 文の羅列により，どの構文に合致するか判断され，規定された文法に合致すると判断された場合は，その文法に対応する C 言語ソースコードが外部ファイルに書き込まれる．
また先述したとおり，def,get といった文は文法規定から外れた場合エラーを吐く．
例：def の解析部

```
      if(f1[PG][n] == "d"):
         if(f1[PG][n+1] == "e"):
            if(f1[PG][n+2] == "f"):
               # print("DEF 10")
               fileobj.write("int ")
               if (f1[PG][n + 4] != "$"):
                  de_flg = True
                  e_flg = True
```

2.3 記号表　別途 PDF による．

2.4 エラーの種類

①START が無いと通知
②END がない，若しくは文章の最後にないと通知
③print の引数が$~で挟まれていないと通知
④def の引数が$~で挟まれていないと通知
⑤get の引数が$~で挟まれていないと通知
⑥print get def の文末に;が無いと通知

2.5 独自言語のプログラム例

例１：最大公約数を求めるアルゴリズム（ユークリッドの互除法）
独自言語

```
START
def $abc~;
def $b~;
def $r~;
def $tmp~;

get $abc~;
get $b~;

if($abc~ < $b~){
    $tmp~ = $abc~;
    $abc~ = $b~;
    $b~ = $tmp~;
    }
$r~ = $abc~ % $b~;
while($r~ != 0){
    $abc~ = $b~;
    $b~ = $r~;
    $r~ = $abc~ % $b~;
}

print $b~;
END
```

対応する C 言語

```c
#include <stdio.h>
int main(){

int abc;
int b;
int r;
int tmp;

scanf("%d",&abc);
scanf("%d",&b);

if(abc<b){
tmp=abc;
abc=b;
b=tmp;
}
r=abc%b;
while(r!=0){
abc=b;
b=r;
r=abc%b;
}

printf("%d",b);

return 0;
}
```

例２：FizzBuzz 問題

```
START
def $i~;
def $a~;
def $b~;

$i~ = 1;

while($i~ <= 100){

    if($i~ % 3 == 0 && $i~ % 5 == 0){
        print(_FizzBuzz¥n_);
    }
    else if($i~ % 3 == 0){
        print(_Fizz¥n_);
    }
    else if($i~ % 5 == 0){

        print(_Buzz¥n_);
    }
    else{
        print($i~);
        print(_¥n_);
    }
    $i~ += 1;


}

END
```

コンパイル結果

```c
#include <stdio.h>
int main(){

int i;
```

```
int a;
int b;

i=1;

while(i<=100){

if(i%3==0&&i%5==0){
printf("FizzBuzz¥n");
}
else if(i%3==0){
printf("Fizz¥n");
}
else if(i%5==0){

printf("Buzz¥n");
}
else {
printf("%d",(i));
printf("¥n");
}
i+=1;


}


return 0;
}
```

例 3　def エラーの例

ソースコード

```
START
def $i~;
def $a~;
def $b~;
def error

$i~ = 1;




END
```

コンパイラのメッセージ

ERROR_Def の引数が無いか，変数の前に $ がありません(5 行目)

例４　入力した数値の桁数を求めるプログラム

ソースコード

```
START
def $number~;
def $digit~;
$digit~ = 0;

get $number~;

while($number~ != 0){
    $number~ = $number~ / 10;
    $digit~ = $digit~ + 1;
}
print($digit~);

END
```

コンパイル結果

```c
#include <stdio.h>
int main(){

int number;
int digit;
digit=0;

scanf("%d",&number);

while(number!=0){
number=number/10;
digit=digit+1;
}
printf("%d",(digit));


return 0;
```

```
}
```

例５　入力した数値を逆順にするプログラム

ソースコード

```
START
def $number~;
def $reverce~;
$reverce~ = 0;

get $number~;

while($number~ > 0){

    $reverce~ = $reverce~ * 10 + $number~ % 10;
    $number~ = $number~ / 10;



}
print($reverce~);

END
```

コンパイル結果

```
#include <stdio.h>
int main(){

int number;
int reverce;
reverce=0;

scanf("%d",&number);

while(number>0){
```

```
reverce=reverce*10+number%10;
number=number/10;



}
printf("%d",(reverce));


return 0;
}
```

考察と感想

E18-5408　佐藤央

考察

独自言語開発に於いては，仕様で示された言語の実装全般を行った．

今回は Python を用いてコンパイラを作成した．C 言語のように配列といった概念ではなく，標準でリストが使用でき，また外部ファイルに保存された文字を各行ごとに代入できるといった便利なライブラリが揃っていることが，２人という少人数ではあるがなんとか完成までこぎつけたことの大きな助けになったと考えた．

ただし，今回の言語では自分たちで実装したのは実質３つの関数と少しの制御命令にとどまっており，とてもミニマムなコンパイラとなった．変数も int のみを用意するにとどまっている．

また授業を通して字句解析、構文解析、意味解析の仕組みは理解できたが，今回の実開発に於いてはその機能を分けて開発を行う余裕はなく，最低限の解析を行った上で，とにかく独自言語と C 言語をスムースに受け渡すことを念頭に置いた結果であろう．また変数を最初から数個こちらで用意し使ってもらうアセンブラ言語に似た文法を考えたが，流石にそれでは味気ないということで自由変数名設定のみは，上級言語に見られるような形で実装した．

＄〜で挟むという少しわかりにくい方式ではあるが，こうすることによって Python でどこからどこまでが変数名なのかを区切ることが用意に可能になった．

感想

今回自分でコンパイラを実装する経験を通して，いかに普段使っている Cs や Java が偉大な言語なのかを思いしった．また解析木の作成及びプログラムの確認で、構文解析のプログラムを確認している際、解析木の大切さを実感した．

エラーコードに関しても，それぞれどのステップに由来するものなのかといった，込み入ったレベルで知ることができてよかった．今回の経験は今後のシステム開発にとって大きなプラスになるであろうと実感している．

また機会があれば，コンパイラの再制作に取り組みたい．

E18-5402 上岡新平

考察

独自言語を作成するにあたって、言語の基本的な仕様策定と BNF による文法定義、解析木作成を行った。

今回の課題への取り組みにおいて、字句解析、文法解析、意味解析がそれぞれ具体的にどういった働きをし、またそれらをどう実装していけばよいかの理解が遅れたのは反省点として挙げられる。ただ Python のライブラリが非常に優秀であり構文解析以降では内部コードを使う必要が無くなったこと、また変数の型は int のみで関数・命令もユークリッドの互除法を表現できる最低限のものに絞ることで、実装のほか BNF や解析木の作成なども比較的容易となった。

今回実際に BNF の記述をやってみて、想定する動作をちゃんとすべて行うか、定義に穴が無いかを確認する作業が予想よりずっと煩雑だった。今まで何気なく扱っていた言語がどれだけ緻密に作り上げられたかを思い知らされた。

感想

二人班ということで一人一人の負担が大きく、結局内部コードがうまく活用できなかった,実装してみてから BNF に大きく変更点が出たなど反省点は残ったが、なんとか形になったことによる達成感は大きい。特に実装を担ってくれた佐藤君には深く感謝したい。また各々の担当範囲が広かったことでより言語構造、コンパイラについて理解が深まったように思う。今後のシステム開発などでこの経験は生きてくるだろう。