# 課題 4 レポート

学生番号 09B17020 河越　淳

2020 年 4 月 3 日

## 1　課題の目的

課題の目的は Pascal 風プログラムを字句解析した結果 (ts ファイル) をコンパイラへの入力として、CASLII で記述したプログラム (cas ファイル) を出力するコンパイラを作成することである。なお、コンパイルした際に構文的もしくは意味的な誤りを検出した場合は標準エラーにその誤りの内容を出力して終了し、入力ファイルが見つからない場合は標準エラーに”File not found”と出力して終了する。エラーメッセージの仕様は課題 3 に準じる。また、誤りを発見した場合には cas ファイルを生成しないようにする。

## 2　課題達成の方針と設計

課題達成のための方針としては、まず checker で字句解析した結果を構文木にする。そして、構文木にしたプログラムを compile で呼び出し、構文木の根から順に構文を見ていくことで Pascal 風プログラムを CASLII の記述へと順に変換していく。

## 3　プログラムの実装方針

次にプログラムの実装について詳しく説明する。

### 3.1　下準備

まず、Pascal 風プラグラムを字句解析した結果を入力として受け取った後、checker でプログラムを拡張バッカス・ナウア記法の構文木に変更する。この際、構文解析と意味解析も checker で同時に行う。

### 3.2　構文木

構文木は図 1 のような構成になっており、Program クラスが根となっている。また、灰色で書かれた部分はクラス名または変数名を表しており、灰色と白で書かれた図形はクラスの構成を灰色のみで書かれた図形は変数を示している。また、下矢印はそのクラスが所持しているクラスまたは変数を表す。
矢印なしの線はクラス同士が結ばれている場合はその親クラスと子クラスの関係を示しており、変数とクラスまたは変数と変数が結ばれている場合はその変数のクラスが示されている。また、*がついている

ものはその下が続くが省略されていることを示すが、一つだけ省略されていないものがあり、他のものもそれと同じ構成となっている。例えば、下部 (*7) で Factor クラスは variable という変数を持つが、変数 variable は Variable クラスであることが示されている。また、左中央で Variable クラス (*4) は子クラスとして StandardVariable クラスと SubscriptedVariable クラスを持つことが示されている。また、中央上部で Statement クラスと basicStatement 変数が結ばれており、その先で InOutStatement 変数が結ばれているが、これは basicStatemnt と inOutStatement の 2 つの変数が Statement クラスと同一であり、子クラスとして InputStatement クラスと OutputStatement クラスを持つことを示している。
また変数 record は全て、クラス Record(*8 中央下) であり、name には変数名または数字、文字などが入る。また、token にはトークンが入る。

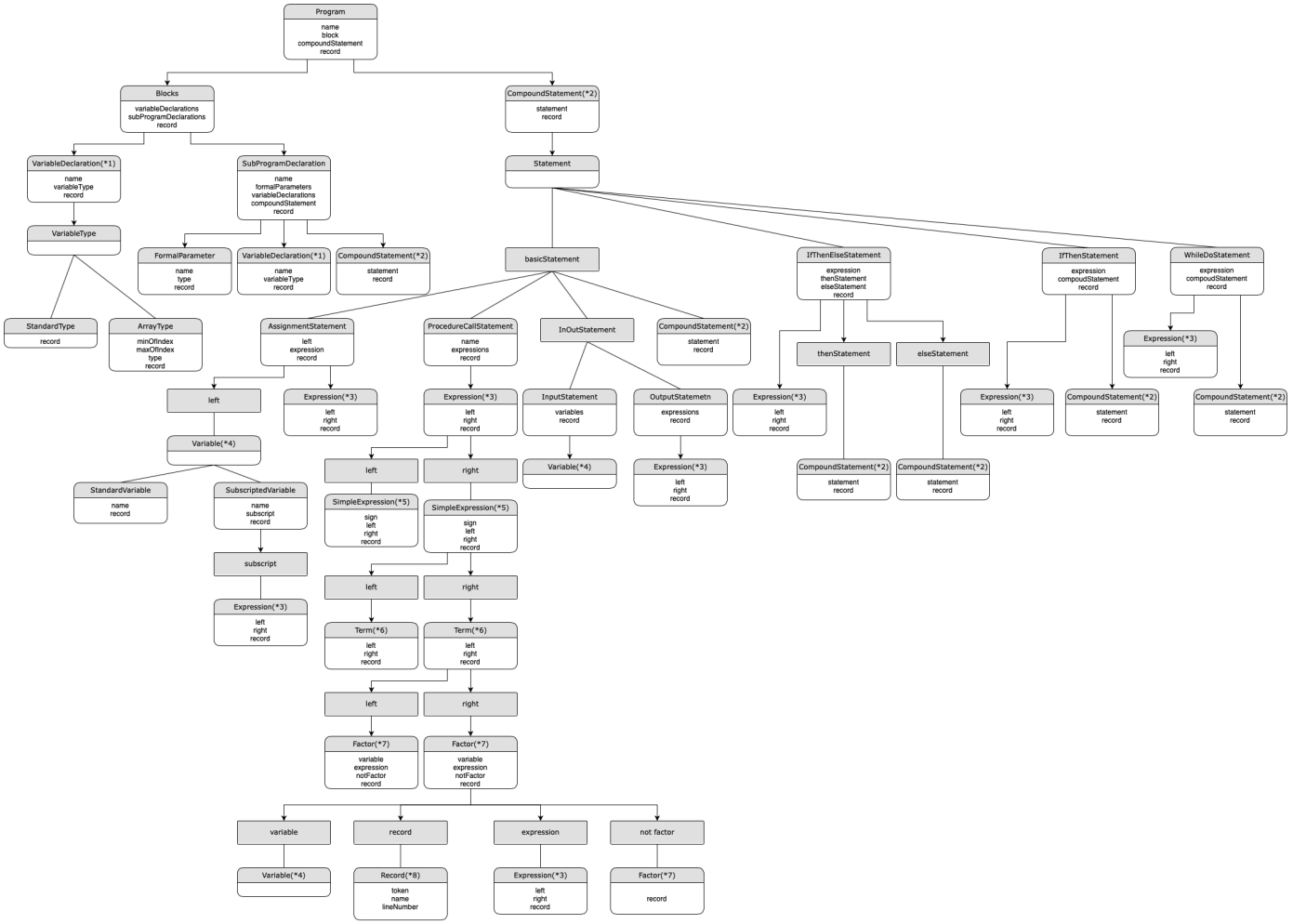


図 1　構文木

# 4　コンパイラ

コンパイラは checker で作成された、図 1 の構成となった構文木を根からみていき、それに合わせた CASLII プログラムを書き込むことで行う。

## 4.1　構文木作成と構文・意味解析

構文木作成と構文・意味解析は 1 行目 checker.run() で行い、2 行目 checker.success() で構文や意味のエラーがないかを確認する。エラーあればエラーを出して処理を終了し、無ければ 4 行目で構文木を入力として CASLII プログラムを作成する convertCASL2 クラスに、引数として checker で作成した構文木の根を与えることで CASLII プログラムへの変換を行う。

```
1 checker.run(inputFileName);
2   if(checker.success()){
3     ConvertCASL2 convert=new ConvertCASL2();
4     convert.run(checker.getRoot());
```

## 4.2　CASLII 命令の書き込み

ファイルへの CASLII 命令の書き込みは次のように行い、3 行目で CASLII 命令を取り出し、4 行目で DC 命令に使うプログラム中で使われた文字列を取り出す。それらを順にファイルに書き込んでいくことで CASL 命令の書き込みを行う。

```
1           try(FileWriter fw=new FileWriter(new File(outputFileName))){
2             BufferedWriter bw=new BufferedWriter(fw);
3             ArrayList<String> commands = convert.getCommand();
4             LinkedHashMap<String,String> names = convert.getNames();
5             for(String co : commands) {
6               bw.write(co+"\n");
7                       }
8             for(Map.Entry<String, String> name:names.entrySet()) {
9               bw.write(name.getKey()+"\tDC\t"+name.getValue()+"\n");
10            }
11            bw.write("LIBBUF\tDS\t256;\n");
12            bw.write("\tEND\t;\n");
```

## 4.3　変数宣言・スコープの管理方法

まず変数宣言の処理について説明する。変数が宣言されると、その変数のためのメモリ番地を用意する。これは変数の名前と宣言された番号を連想配列に格納することで行う。宣言された番号とはその変数が何番目に

宣言されたかを表すもので、0 から始まり純変数が呼び出されると数字が 1 増え、添字付き変数が呼び出されると添字の数だけ番号が増える。例えば i,a[1..10]、j の順で呼び出されると i の番号は 0、a の番号は 1、j の番号は 11 となる。この番号と変数名を大域変数であれば vLenList に、局所変数であれば tmpvLenList に、仮パラメータであれば parLenList に格納する。また、下の valLen 関数で変数名を入力として、その変数の番地を出力として得ることが出来る。また、大域変数、局所変数、仮パラメータに同一の名前があった場合は仮パラメータ、局所変数、大域変数の順で優先される。
また、手続き名は scope に格納され、valLen 関数で入力として手続き名を入れることで何番目に宣言された手続き名であるかも得ることが出来る。

```
1   public int valLen(String vName) {
2     for(Map.Entry<String, Integer> entry : parLenList.entrySet()) {
3           if(vName.equals(entry.getKey())) return entry.getValue();
4         }
5     for(Map.Entry<String, Integer> entry : tmpvLenList.entrySet()) {
6           if(vName.equals(entry.getKey())) return entry.getValue();
7         }
8     for(Map.Entry<String, Integer> entry : vLenList.entrySet()) {
9           if(vName.equals(entry.getKey())) return entry.getValue();
10        }
11    for(String s:scope) {
12      if(vName.equals(s)) return scope.indexOf(s);
13    }
14    return 0;
15  }
```

## 4.4 変数の代入の方法

変数の代入は次のように行う。まず、1 行目で代入文の式を得、それを 2 行目で式を処理する関数に入れる。すると計算された値が PUSH されるため、5 行目で値を GR1 に代入する。また、3 行目で左辺の変数を得、4 行目で GR2 に変数の番地を代入する。そして 6 行目で右辺の値を左辺の変数の番地に代入することで変数の代入を行う。

```
1     Expression assignExpress = assign.getExpression();
2     expression(assignExpress);
3     Variables assignV = assign.getVariable();
4     variable(assignV);
5     caslList.add("\tPOP\tGR1");
6     caslList.add("\tST\tGR1, VAR, GR2");
```

変数の番地の計算は純変数であれば次のプログラムのように行う。2 行目で先ほど説明した valLen 関数を用いて、3 行目で GR2 に変数の番地を代入している。

```
1 public void pureVariable(Variables pv) throws Exception{
2     int i = valLen(pv.getRecord().getName());
3     caslList.add("\tLD\tGR2\t,="+String.valueOf(i));
4 }
```

また、添字付き変数の場合は 3 行目で添字の計算をして、PUSH された添字の値を 4 行目で GR2 に代入している。そして 6 行目で添字の値に変数の最初の番地から 1 を引いた数を加えることで、添字付き変数の番地を GR2 に代入している。ちなみに添字の最小値を 1 として処理をしているが、6 行目で 1 を引く代わりに変数の添字の最小値を引くことで添字の最小値が 1 でない場合も対応できる。

```
1 public void subscriptedVariable(Variables sv) throws Exception{
2     Expression subscript = sv.getSubscript();
3     expression(subscript);
4     caslList.add("\tPOP\tGR2");
5     int i = valLen(sv.getRecord().getName());
6     caslList.add("\tADDA\tGR2, ="+String.valueOf(i-1));
7     return;
8 }
```

## 4.5 式の処理方法

### 4.5.1 式

式の処理はまず 1 行目で式のレコードをみる。レコードが null であれば (-3) などの () のついた負数であるため、単純式に移行する。(作成した構文の仕様上そうなる。)
また、そうでなければトークンを取ってくる。トークンは関係演算子、加法演算子、乗法演算子、因子のいづれかであり、関係演算子であれば処理を行い、そうでなく式が単純式型であれば単純式に移行、式が項型であれば項に移行、因子型であれば因子に移行する。(9〜15 行目)

```
1     if(exp.getRecord()!=null) {
2         t = exp.getRecord().getToken();
3     }else {
4       SimpleExpression sExp = (SimpleExpression)exp;
5       simpleExpression(sExp);
6       return;
7     }
8         ` ` ` 省略 ` ` `
9         else if(exp instanceof SimpleExpression) {
10      simpleExpression((SimpleExpression)exp);
11    }else if(exp instanceof Term) {
12      term((Term)exp);
13    }else if(exp instanceof Factor) {
```

```
14        factor((Factor)exp);
15      }
```

関係演算子であった場合は次の処理を行う。まず、1 行目で式の左側と右側が null であった場合は (式) の形となっているため、() の中の式を取り出し 4 行目のように関数"式"に移行する。また、式の左側が単純式型であれば単純式に移行、式が項型であれば項に移行、因子型であれば因子に移行する。(9〜14 行目)
これと同様のことを式の右側でも行う。

```
1         if(exp.getLeft()==null && exp.getRight()==null) {
2           if(exp instanceof Factor) {
3             Expression e =((Factor) exp).getExpression();
4             expression(e);
5           }
6           return;
7         }
8         if(exp.getLeft() instanceof SimpleExpression) {
9           simpleExpression((SimpleExpression)exp.getLeft());
10        }else if(exp.getLeft() instanceof Term) {
11          term((Term)exp.getLeft());
12        }else if(exp.getLeft() instanceof Factor) {
13          factor((Factor)exp.getLeft());
14        }
```

また、その後関係演算子の処理を行う。関係演算子の処理はまず 6 行目の 1 つ目の POP で右辺の値を GR2 に代入し、2 つ目の POP で左辺の値を GR1 に代入し、比較を行う。その後、関係演算子の種類によって処理を変える (下のテーブル)。処理では式を満たせば GR1 に 0 が代入され、満たさなければ GR1 に FFFF が代入されるようになっている。

```
1         String L1="TRUE"+String.valueOf(trueNum);
2         String L2="BOTH"+String.valueOf(bothNum);
3         trueNum+=1;
4         bothNum+=1;
5
6         switch(t){
7             caslList.add("\tPOP\tGR2");
8             caslList.add("\tPOP\tGR1");
9             caslList.add("\tCPA\tGR1,GR2");
```

| = | <> | < | <= |
|---|---|---|---|
| JZE L1<br>LD GR1 =#FFFF<br>JUMP L2<br>L1 LD GR1,=#0000<br>L2 PUSH 0,GR1 | JNZ L1<br>LD GR1 =#FFFF<br>JUMP L2<br>L1 LD GR1,=#0000<br>L2 PUSH 0,GR1 | JMI L1<br>LD GR1 =#FFFF<br>JUMP L2<br>L1 LD GR1,=#0000<br>L2 PUSH 0,GR1 | JPL L1<br>LD GR1 =#0000<br>JUMP L2<br>L1 LD GR1,=#FFFF<br>L2 PUSH 0,GR1 |
| | | > | >= |
| | | JPL L1<br>LD GR1 =#FFFF<br>JUMP L2<br>L1 LD GR1,=#0000<br>L2 PUSH 0,GR1 | JMI L1<br>LD GR1 =#0000<br>JUMP L2<br>L1 LD GR1,=#FFFF<br>L2 PUSH 0,GR1 |

#### 4.5.2 単純式

単純式ではまずレコードをみる。レコードが null であれば (-3) などの () のついた負数であり、left のレコードに数値や変数が格納されているため、それを token に代入する。

```
1     if(expression.getRecord()!=null) {
2       token = expression.getRecord().getToken();
3     }else {
4       token = expression.getLeft().getRecord().getToken();
5     }
```

トークンは加法演算子、乗法演算子、因子のいづれかであり、加法演算子であれば処理を行い、そうでければ (-3) など () のついた負数であるとして処理を行う。() のついた負数である場合の処理は 3 行目で値を PUSH でスタックに格納し、次に負数の処理をする。

```
1 if(expression.getLeft() instanceof Factor) {
2         Factor left = (Factor)expression.getLeft();
3         factor(left);
4       }
5       sign(expression.getSign());
```

負数の処理は sign 関数で行う。負数であれば sign が false であるため、2 行目で負数かどうかの判断をして、負数であれば 3 行目の POP で GR2 に値を入れ、0 からその値を引くことで正負を反転させる。そして、その値を PUSH でスタックに格納する。

```
1   public void sign(boolean s) throws Exception{
2     if(s == false) {
3       caslList.add("\tPOP\tGR2");
4       caslList.add("\tLD\tGR1,=0");
```

```
5         caslList.add("\tSUBA\tGR1,GR2");
6         caslList.add("\tPUSH\t0,GR1");
7       }
8     }
```

加法演算子であれば、まず左辺の処理をする。左辺が単純式型であれば単純式に移行、項型であれば項に移行、因子型であれば因子に移行する。その後、左辺が例えば $-3+4$ や $-3*2+4$ のような負数でないかどうかの判断を上記の sign 関数を使うことで行う。

右辺も左辺と同様、単純式型であれば単純式に移行、項型であれば項に移行、因子型であれば因子に移行する処理を行う。その後、下のプログラムの 1 行目で GR2 に右辺の値を代入し、2 行目で GR1 に左辺の値を代入する。そして、加法演算子の計算を行い PUSH で計算された値をスタックに格納する。

```
1           caslList.add("\tPOP\tGR2");
2           caslList.add("\tPOP\tGR1");
3           if(token ==Token.SPLUS) {
4             caslList.add("\tADDA\tGR1, GR2");
5           }else if(token == Token.SMINUS) {
6             caslList.add("\tSUBA\tGR1, GR2");
7           }else if(token ==Token.SOR) {
8             caslList.add("\tOR\tGR1,GR2");
9           }
10          caslList.add("\tPUSH\t0, GR1");
```

#### 4.5.3 項

項でもトークンをみるが、トークンは必ず乗法演算子であるため乗法演算子の処理を行う。まず、左辺をみて、左辺が項型であれば項に移行、因子型であれば因子に移行する。

右辺も同様のことを行うが、その後に乗法演算子の処理を行う。処理は下のプログラムの 1 行目、2 行目で左辺と右辺を取ってきて GR1、GR2 に代入し乗法演算子の計算を行う。そして、その結果を PUSH でスタックに格納する。(3〜15 行目)

```
1           caslList.add("\tPOP\tGR2");
2           caslList.add("\tPOP\tGR1");
3           if(token == Token.SSTAR) {
4             caslList.add("\tCALL\tMULT");
5             caslList.add("\tPUSH\t0,GR2");
6           }else if(token == Token.SDIVD ) {
7             caslList.add("\tCALL\tDIV");
8             caslList.add("\tPUSH\t0,GR2");
9           }else if(token==Token.SMOD) {
10            caslList.add("\tCALL\tDIV");
11            caslList.add("\tPUSH\t0,GR1");
```

```
12          }else if(token==Token.SAND) {
13            caslList.add("\tAND\tGR1,GR2");
14            caslList.add("\tPUSH\t0,GR1");
15          }
```

#### 4.5.4 因子

因子ではまず、() でないかどうかを確認し () であればその中身を取ってきて、関数"式"に代入する。そうではなく数字であればその値を PUSH し、true であれば 0 を PUSH、false であれば FFFF を PUSH する。文字であればその文字を PUSH する。
また、文字列であれば文字列を格納しているラベルから文字列を呼び出し、GR2 に代入する。そして文字列を PUSH する。(4、5 行目)
not 因子であれば factor 関数で値を PUSH して、その値を反転させる。(9〜11 行目)
変数であれば variable 関数で変数の番地を GR2 に代入し、15 行目の LD で変数の番地に格納されている値を GR1 に代入する。そして、その値を PUSH する。
これらによって、値がスタックに格納される。

```
1  public void factor(Factor f) throws Exception{
2  ` ` ` 省略 ` ` `
3      }else if(f.getValType()==ValType.tString){
4        caslList.add("\tLAD\tGR2,CHAR"+String.valueOf(charNum));
5        caslList.add("\tPUSH\t0,GR2");
6        nameLists.put("CHAR"+String.valueOf(charNum),f.getRecord().getName
             ());
7        charNum+=1;
8      }else if(f.getNotFactor() !=null) {
9        factor(f.getNotFactor());
10       caslList.add("\tPOP\tGR1");
11       caslList.add("\tXOR\tGR1, =#FFFF");
12       caslList.add("\tPUSH\t0, GR1");
13     }else if(f.getVariable()!=null) {
14       variable(f.getVariable());
15       caslList.add("\tLD\tGR1, VAR, GR2");
16       caslList.add("\tPUSH\t0,GR1");
17     }
18 }
```

### 4.6 手続きの呼び出し方法

手続きはまず関数"式"で引数を PUSH する。そして、下のプログラムの 7 行目で手続きを呼び出すことで行う。6 行目変数 i は呼び出された手続きの番号を表し、同一の名前であれば同一の番号となる。

```
1       if(n.getExpressions() != null) {
2         for(Expression e: n.getExpressions()) {
3           expression(e);
4         }
5       }
6       int i = valLen(n.getName());
7       caslList.add("\tCALL\tPROC"+String.valueOf(i));
```

## 4.7 手続き作成方法

### 4.7.1 手続き

手続きは Block で作成する。手続きは同じ名前であれば、PROC の後の番号は同じとなる。(4、5 行目) また、手続きプログラムが終わるごとに RET を書き込み、範囲外は参照出来ないようにする。(7 行目) そして、1 つの手続きプログラムを書き込むごとに仮パラメータと局所変数は削除する。(8〜11 行目)

```
1    public void block(Blocks b) throws Exception{
2      if(b.getSubProgramDeclarations()!=null){
3        for(SubProgramDeclaration s : b.getSubProgramDeclarations()){
4          int i = valLen(s.getName());
5          caslList.add("PROC"+String.valueOf(i)+"\tNOP");
6          subProgramDeclaration(s);
7          caslList.add("\tRET");
8          tmpvList.clear();
9          tmpvLenList.clear();
10         parList.clear();
11         parLenList.clear();
12       }
13     }
14   }
```

### 4.7.2 仮パラメータ

仮パラメータは下のプログラムで作成し、引数の左から順に PUSH されているため 3 行目 Collections.reverse でパラメータの順番を反転させ、右から順に取り出すようにする。例えば procedure subproc(x,y,z: integer) で subproc(99,80,7) と呼び出した場合、99、80、7 の順に PUSH されるため、z に 7、y に 80、x に 99 の順に変数の番地に値を格納していく。(4〜15 行目)

```
1       for(FormalParameter param : p){
2         ArrayList<String> paName = param.getNames();
3         Collections.reverse(paName);
4         for(String pp: paName) {
```

```
5             int i = valLen(pp);
6             caslList.add("\tLD\tGR1,GR8");
7             caslList.add("\tADDA\tGR1,=1");
8             caslList.add("\tLD\tGR2, 0, GR1");
9             caslList.add("\tLD\tGR3,="+String.valueOf(i));
10            caslList.add("\tST\tGR2,VAR,GR3");
11            caslList.add("\tSUBA\tGR1,=1");
12            caslList.add("\tLD\tGR1, 0, GR8");
13            caslList.add("\tADDA\tGR8, =1");
14            caslList.add("\tST\tGR1, 0, GR8");
15          }
16        }
17      }else {
18        caslList.add("\tLD\tGR1,GR8");
19        caslList.add("\tADDA\tGR1,=0");
20      }
```

## 4.8 分岐の処理方法

### 4.8.1 ifThen 文

ifThen 文の場合、まず式の値をスタックに格納する。(2 行目) そして式の値が FFFF のときは ELSE に移動し、0 の時は複合文を実行する。(4、5、10 行目)

```
1   public void ifThenStatement(IfThenStatement n) throws Exception{
2     expression(n.getExpression());
3     caslList.add("\tPOP\tGR1");
4     caslList.add("\tCPA\tGR1, =#FFFF");
5     caslList.add("\tJZE\tELSE"+String.valueOf(elseNum));
6     int elNum=elseNum;
7     elseNum+=1;
8     Statement comState = n.getThenStatement();
9     if(comState instanceof CompoundStatement) {
10      CompoundStatement((CompoundStatement)comState);
11      caslList.add("ELSE"+String.valueOf(elNum)+"\tNOP");
12    }
13  }
```

### 4.8.2 ifThenElse 文

ifThenElse 文では ifThen 文と同様に式の値が FFFF のときは ELSE に移動し、0 の時は複合文を実行するが ELSE で移動した先でも複合文を実行する。(3、17 行目) そして Then での複合文が実行された後に Else

の複合文が実行されないように移動する。(9、17、18 行目)

```
1   public void ifThenElseStatement(IfThenElseStatement n) throws Exception{
2 ` ` ` 省略 ` ` `
3     caslList.add("\tJZE\tELSE"+String.valueOf(elseNum));
4     int elNum=elseNum;
5     elseNum+=1;
6     Statement thenState = n.getThenStatement();
7     if(thenState instanceof CompoundStatement) {
8       CompoundStatement((CompoundStatement)thenState);
9       caslList.add("\tJUMP\tENDIF"+String.valueOf(endifNum));
10    }
11
12    int enNum=endifNum;
13    endifNum+=1;
14    caslList.add("ELSE"+String.valueOf(elNum)+"\tNOP");
15    Statement elseState = n.getElseStatement();
16    if(elseState instanceof CompoundStatement) {
17      CompoundStatement((CompoundStatement)elseState);
18      caslList.add("ENDIF"+String.valueOf(enNum)+"\tNOP");
19    }
20  }
```

#### 4.8.3 whileDo 文

whileDo 文ではまず LOOP を書き込む。(2 行目)
ループは式が満たされている間行われ、式が満たされない、つまり式の値が FFFF となったときに ENDLP に移動するようにした。(8、9、18 行目)

```
1 public void whileDoStatement(WhileDoStatement n) throws Exception{
2     caslList.add("LOOP"+String.valueOf(loopNum)+"\tNOP");
3     int loNum=loopNum;
4     loopNum+=1;
5
6     expression(n.getCondition());
7     caslList.add("\tPOP\tGR1");
8     caslList.add("\tCPL\tGR1, =#FFFF");
9     caslList.add("\tJZE\tENDLP"+String.valueOf(endlpNum));
10    int enNum=endlpNum;
11    endlpNum+=1;
12
13    Statement comState = n.getDoStatement();
```

```
14      if(comState instanceof CompoundStatement) {
15        CompoundStatement((CompoundStatement)comState);
16        caslList.add("\tJUMP\tLOOP"+String.valueOf(loNum));
17      }
18      caslList.add("ENDLP"+String.valueOf(enNum)+"\tNOP");
19    }
```

### 4.9 レジスタやメモリの利用方法

レジスタは GR1 や GR2 を値の計算や計算した値の PUSH、POP などに用いた。また、メモリは変数を代入するために利用し、変数の数 (添字付き変数がある場合は添字の数も足して) のメモリを空けるようにした、例えば変数が i のみだと後ろに VAR DS 1 を変数が i,j だと後ろに VAR DS 2 を加えるようにして、メモリを空けた。また、文字列が使われた場合は、例えば'ABC' の場合 CHAR0 DC 'ABC' というのを加えることで文字列を参照できるようにした。

### 4.10 ラベルの管理方法

ラベルは手続き文を呼び出す時はその手続き文が呼び出された順に 0 から番号をつけていくようにした。また、分岐の処理ではラベルはラベルを呼び出したところでその値を保存し、番号を +1 してから複合文を呼び出すことで階層が深くなっても正しく分岐処理が行われるようにした。

### 4.11 プログラムの再利用

プログラムは構文木にならって関数を作成したため、同じ処理が行われる際にはそのプログラムを書き込む関数に移動すれば良いようにした。図 1 の構文木の*がついているところのような再帰的な処理もプログラムを再利用することで行えるようになった。

## 5 まとめ

今回の課題では構文"プログラム"から順に入れ子構造の中を深さ優先探索して、Pascal 風プログラムを CASLII プログラムに変換していくことでコンパイルが出来ることを学びました。また、全体を通してコンパイラがあるプログラムを入力として、それをアセンブラ言語に変換するまでの流れを理解することができ、また情報科学実験 C でアセンブラ言語が機械語に変換されるまでの流れを学んだため、それら合わせることでプログラムが機械語に変換されるまでの流れを理解することができました。

## 6 感想

今回の演習で、あるプログラムを別のプログラムの書き方に変更するということを、コンパイラが行なっていたということを十分理解しました。また、コンパイラの変換の仕方によってプログラムの長さが全然変わり、実行速度にも影響してくるということも分かり、命令の省略化ができる為コンパイラ言語がインタプリタ言語よりも実行速度が速いということを身をもって体験することができ、とても良い学習になりました。ま

た、今回最適化を行うことが出来なかったため、最適化もまた時間があれば挑戦してみたいと思いました。