課題 3 レポート

学生番号 09B17020 河越　淳

2020 年 3 月 28 日

1　課題の目的

Pascal 風プログラムを字句解析した結果を意味解析器への入力として、Pascal 風言語で記述されたプログラムが構文的に正しいかどうか判定すると共に、変数等の宣言や有効範囲の誤りや、演算子と被演算子の間や仮パラメータと実パラメータの間などでの型の不整合といった、意味的な誤り意味的な誤りが含まれるかどうかの判定を出力とするプログラムを作成する。また、構文的な誤りが含まれるかどうか (課題 2 の内容) も同時に判定する。

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、checker で字句解析されたファイルを一行ずつ読み込み、文字列とトークンと行番号に分けて、それぞれリスト型配列に格納する。その後、一番上の行から順にトークンの構文が合っているかどうかを確認し構文解析を行うと同時に拡張バッカス・ナウア記法に従って木構造を作成し、変数の範囲や型が合っているかどうかの意味解析を行う。また、どうしても前に戻らなければならない時のみ、前の行に戻る処理を行う。

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

まず結果の出力の仕方、トークンの判別方法、前に戻って判別する方法について説明する。
まず、出力は構文が間違っていた際には下の関数を読み込みエラーを出し、構文が合っていた際には”OK”を表示する。この関数は変数 fail(初期値は false) を用いて、何度も呼び出されても最初の一度だけしか出力をしない。また、ts ファイルの何行目のトークンでエラーが出たのかを変数 rowNumber で判別し、そのトークンのある行番号 (コンパイル前のプログラム上の行番号) を出力する。変数 rowNumber は判別中の行番号 (ts ファイル上の行番号) が入っており、リスト型配列 lineNumber にはある行目のトークン (ts ファイル上) がプログラム上で何行目にあるかが順に格納されている。

トークンの判別は下の関数 match を用いる。この関数は引数で選択したトークンと調べている行のトークンが合っていれば、関数 read を呼び、間違っていれば false を返す。関数 read は変数 rowNumber を +1 して判別対象を次の行にしてから true を返す。

```
1 private boolean match(final Token Token){
2   if(lookahead.tokenCheck(Token)){
3     return read();
4   }
5
6 throw new Error("expecting "+Token.toString()+"; "+lookahead.getToken().
    toString()+" found.");
7 }
```

引数の Token とは前回 enum クラスで定義したもので、全てのトークンが定義されている。

```
1 public enum Token {
2   SAND("and"),
3   SARRAY("array"),
4   SBEGIN("begin"),
5   SBOOLEAN("boolean"),
6   SCHAR("char"),
7   SDIVD("div"),
8     '''中略'''
9 }
```

調べる行を前に戻す際には、下の関数 store と関数 back を用いて行い、保存しておきたい行で関数 store を呼び出すことで保存処理を行い、前に戻りたいときに関数 back を呼び出すことで戻る処理を行う。

```
1 private int store(){
2   return rowNumber;
3 }
4
5 private void back(int previous){
6   rowNumber=previous;
7   lookahead=records.get(rowNumber);
8 }
```

3.2 意味解析

次に意味解析について説明する。

意味解析は関数 run 内で下の関数 program を呼び出すことで行う。先ほど説明した関数 match などを使っ

て、最初がトークン”PROGRAM”で始まっているか、その次にプログラム名があるか、などを順に判別している。そして構文が間違っていれば関数 syntaxError を呼び出し、間違っている行を出力する。また、関数 block や関数 compoundStatement を呼ぶことで入れ子構造の中を深さ優先探索で判別していく。

```
1  private Program program(){
2    if(lookahead.tokenCheck(Token.SPROGRAM)){
3      Records r=records.get(rowNumber);
4      match(Token.SPROGRAM);
5      String name=programName();
6      nameTypes.put(name, null);
7
8      if(null!=name && lookahead.tokenCheck(Token.SSEMICOLON)){
9        match(Token.SSEMICOLON);
10       Blocks block=block();
11       CompoundStatement compoundStatement=compoundStatement();
12
13       if(null!=block && null!=compoundStatement &&lookahead.tokenCheck(
             Token.SDOT)){
14         match(Token.SDOT);
15         checkCompoundStatement(compoundStatement,nameTypes);//semErrorCheck
16         return new Program(name, block, compoundStatement, r);
17         }
18       }
19     }
20     syntaxError();
21     return null;
22 }
```

4 詳細

次に意味解析の詳細について述べる。

4.1 変数解析

変数の重複と型の照合 (変数が現れた際にその変数の型を照合すること) については nameTypes という型と変数名をひとまとめにした連想配列を用いることで解析を行った。まず、構文”ブロック”の変数宣言で宣言された変数を nameTypes 配列に名前と型をひとまとめにして格納する。次に構文”副プログラム宣言”の変数宣言で宣言された変数を tempNameTypes という連想配列に同様に名前と型をひとまとめにして格納する。これらにより、全ての宣言された変数とその型が連想配列に格納される。また、手続き名は procedureNames という配列に格納した。これらの配列を変数の型の照合や名前が正しいかの判別に利用した。

nameTypes 配列には大域変数が格納されており、tempNameTypes には局所変数が格納されている為、新しい変数を格納する際にこれらを確認することで重複や宣言されていない変数が使用されていないかを判別した。
使われていない手続き名を使用していないかどうかは手続き名を呼び出す際に procedureNames に格納されている名前が使われているかどうかを判別することで行った。

```
1   HashMap<String,ValType> nameTypes = new HashMap<String,ValType>();
2   private ArrayList<String> procedureNames = new ArrayList<String>();
```

4.2 型の判別

正しい型が使われているかの判別は主に checkCompoundStatement、checkFactor、checkFactorType という 3 つの関数で行った。

4.2.1 checkCompoundStatement

checkCompoundStatement 関数は”プログラム”の複合文、”副プログラム宣言”の複合文で使用し、正しい複合文かどうかを判別する。使い方としては引数として局所変数と複合文を与えると、まず複合文が if 文、while 文、基本文などの中のどれであるかを判別し、その後、意味判別を行う。
例えば、6 行目から始まる代入文であれば 13 行目の nameCheck 関数で、使われている変数が宣言されたものかを判別し宣言されていなければ 14 行目の関数でエラーを出す。また、35 行目では配列の添字が int 型かどうかを判別している。また、9 行目で checkFactor 関数を 22 行目で checkFactorType 関数を使用して、式が正しいかどうかを判別している。

```
1     private void checkCompoundStatement(CompoundStatement compoundStatement
          ,HashMap<String,ValType> tempNameTypes) {
2       if(compoundStatement!=null) {
3         ArrayList<Statement> statements = compoundStatement.getStatements
              ();
4
5         for(Statement state:statements) {
6           if(state instanceof AssignmentStatement) {
7             AssignmentStatement assignState = (AssignmentStatement) state;
8             Expression assignExpress = assignState.getExpression();
9             checkFactor(assignExpress);
10            Variables assignV = assignState.getVariable();
11            String assignVname = assignV.getName();
12            int assignVrow = assignV.getLineNumber();
13            if(nameCheck(assignVname,nameTypes) && nameCheck(assignVname,
                  tempNameTypes)) {
14              semanticErrorOutput(assignVrow);
15            }
```

```
16              ValType assignType = getType(assignVname,tempNameTypes);
17              if(assignType == null) assignType = getType(assignVname,
                    nameTypes);
18              if(assignType == ValType.tBooleanArray || assignType == ValType
                    .tIntegerArray) {
19                if(assignV.getSubscript() == null) semanticErrorOutput(
                      assignVrow);
20              }
21
22              checkFactorType(assignExpress, assignType,tempNameTypes);
23
24              Expression subscript = assignV.getSubscript();
25              if(subscript!=null) {
26
27                int subRow = subscript.getLineNumber();
28                if(subscript instanceof Factor) {
29                  Factor subFactor = (Factor)subscript;
30                  Variables subV = subFactor.getVariable();
31                  if(subV!=null) {String subVName = subV.getName();
32                    ValType subVType = getType(subVName,tempNameTypes);
33                    if(subVType==null) subVType = getType(subVName,nameTypes
                          );
34                    if(subVType != ValType.tInteger) {
35                      semanticErrorOutput(subRow);
36                    }
37                  }
38                }
39              }
40            }
41            else if(state instanceof ProcedureCallStatement) {
42              ProcedureCallStatement proState = (ProcedureCallStatement)
                    state;
43              int proRow = proState.getLineNumber();
44              boolean flag = false;
45              for(String s: procedureNames) {
46                if(s.equals( proState.getName())) flag = true;
47
48                                  ` ` ` 省略 ` ` `
```

4.2.2 checkFactor

checkFactor 関数は式が出てきた際に使用し、引数として式をとる。9 行目で + や*が出てきた際に被演算子として char 型や boolean 型が使用されていないかを判別し、使用されていればエラーを出す。また、16 行目のように再帰的に使用することで式が複雑になっても判別が可能となる。プログラムの省略された部分では左辺と同様のことを右辺でも行っている。

```
1     private boolean checkFactor(Expression expression) {
2       Token token = expression.getRecord().getToken();
3       if(token.isAdditiveOperator() || token.isMultiplicativeOperator()) {
4         if(expression.getLeft() instanceof Factor) {
5           Factor left = (Factor)expression.getLeft();
6
7           if(left != null) {
8             int leftRow = left.getLineNumber();
9             if(left.getValType() == ValType.tBoolean || left.getValType()
                == ValType.tChar) {
10              semanticErrorOutput(leftRow);
11              return false;
12            }
13            if(left.getExpression() != null) checkFactor(left.getExpression
                ());
14          }
15        }else {
16          checkFactor(expression.getLeft());
17        }
18
19        if(expression.getRight() instanceof Factor){
20          Factor right = (Factor)expression.getRight();
21          ` ` ` 省略 ` ` `
```

4.2.3 checkFactorType

checkFactorType 関数は引数として式と式の型と局所変数を取り、式の中の項の型が正しいかどうかを判別している。18 行目では左辺の型が式の型に一致しているかを判別し、違っていればエラーを出すようにしている。プログラムの省略された部分では左辺と同様のことを右辺でも行っている。

```
1     private boolean checkFactorType(Expression expression, ValType
        checkType, HashMap<String,ValType> tempNameTypes) {
2       Token token = expression.getRecord().getToken();
3       if(checkType == ValType.tIntegerArray) checkType = ValType.tInteger;
4       if(checkType == ValType.tBooleanArray) checkType = ValType.tBoolean;
```

```
5       if(token.isAdditiveOperator() || token.isMultiplicativeOperator()) {
6         if(expression.getLeft() instanceof Factor) {
7           Factor left = (Factor)expression.getLeft();
8
9           if(left != null) {
10            int leftRow = left.getLineNumber();
11            if(left.getExpression() != null) checkFactor(left.getExpression
                  ());
12            else {
13              ValType leftValType = left.getValType();
14              if(leftValType == null) {
15                leftValType = getType(left.getVariable().getName(),
                      tempNameTypes);
16                if(leftValType == null) leftValType = getType(left.
                      getVariable().getName(),nameTypes);
17              }
18              if(leftValType != checkType) {
19                semanticErrorOutput(leftRow);
20                return false;
21              }
22            }
23          }
24        }else {
25          checkFactor(expression.getLeft());
26        }
27
28        if(expression.getRight() instanceof Factor){
29          Factor right = (Factor)expression.getRight();
30          ` ` ` 省略 ` ` `
```

5 まとめ

今回の意味解析では構文”プログラム”から順に入れ子構造の中を深さ優先探索することで解析を行った。特に複合文では再帰呼び出しを使用するで代入文や手続き呼出し文や入出力文がどれだけ長くなろうとも右辺と左辺の型が一致しているかや宣言していない変数を使っていないかなどの意味解析を行えるようにした。

6　感想

今回の演習では拡張バッカス・ナウア記法で書かれたプログラムを構文木にするのに大変苦労しました。特に木を作成する為にたくさんのクラスを作成するのにかなりの時間を使いましたが、変数の型が正しいかどうかだけを見るだけで、かなり多くのテストが正解になる為、クラスを作成するのではなく変数を配列に入れて、その変数の型や使い方が正しいかどうかを判別するということを行えばかなり時間の節約になったのではないかと思いました。