課題 2 レポート

学生番号 09B17020 河越　淳

2019 年 11 月 10 日

1　課題の目的

課題 2 では第一引数で指定された ts ファイルを読み込み、構文解析を行う。構文が正しい場合は標準出力に”OK”を出力して、構文が正しくない場合は標準エラーに”Syntax error: line”という文字列と共に最初の誤りを見つけた行の番号を出力して終了する処理を行う。また、入力ファイル内に複数の誤りが含まれる場合は、最初に見つけた誤りのみを出力し、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。

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、lexer 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ていく。ただし、どうしても前に戻らなければならない時やプログラムが見にくくなるのを防ぐ時のみ、前の行に戻る処理を行う。構文解析は拡張バッカス・ナウア記法で書かれた構文通りに判別していく、つまり構文”プログラム”(プログラム = ”program” プログラム名 ”;” ブロック 複合文 ”.”) が合っているかどうかを判別する。ちなみに、構文”プログラム”が合っているかを判別するには、構文”ブロック”(ブロック = 変数宣言 副プログラム宣言群)、構文”複合文”(複合文 = ”begin” 文の並び ”end”) なども合っているか判別しなければならなく、これらの入れ子構造の中を深さ優先で順に判別していくことで解析を行う。

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。
出力は構文が間違っていた際には下の関数を読み込むことで行い、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。

```
1   private void syntaxErrorOutput() {
2     if (!fail) {
3       System.err.println("Syntax error: line " + lineNumber.get(rowNumber
            ));
4       fail = true;
5     }
6   }
```

トークンの判別は下の関数を用いる。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 nextLine を呼び、間違っていれば false を返す。関数 nextLine 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 tokenSyntax(Token Token) {
2     return tokens.get(rowNumber) == Token ? nextLine() : false;
3   }
```

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 previousLine と関数 lineStack を用いて行い、保存しておきたい行で関数 lineStack 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 previousLine を呼び出すことで戻る処理を行う。

```
1   private void previousLine() {
2     while(rowNumber > rowStack) rowNumber -= 1;
3   }
4
5   private void lineStack(int rowNumber) {
6     rowStack = rowNumber;
7   }
```

3.2 構文解析

次に構文解析について説明する。
構文解析は関数 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 tokenSyntax 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 syntaxErrorOutput 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。つまり、まずトークン”PROGRAM”かを判別し、次にプログラム名かを判別し、その次にトークン”SEMICOLON”かどうかを判別するといったように順に判別を行っていくことで構文が正しいかどうかを判別する。

```
1  private boolean program() {// プログラム
2    if (tokenSyntax(Token.SPROGRAM) && programName() && tokenSyntax(Token.
       SSEMICOLON)
3        && block() && compoundStatement()
4        && tokenSyntax(Token.SDOT)) {
5      return true;
6    }
7    syntaxErrorOutput();
8    return false;
9  }
```

4 工夫した点

次にその構文解析の中で工夫した点を 3 つ述べる。

4.1 繰り返し文

0 回または 1 回の繰り返し文では、まず最初のトークンが出てくるかどうか確認し、出てくればその先の構文を解析し、最初のトークンが出てこなければ 0 回であるとして true を返す。
構文”変数宣言の並び”のように 0 個以上の繰り返しをする際は while 文を使って、最初に変数名の並びが出てくるかどうかを確認し、出てくれば次の”COLON”と”型”と”SEMICOLON”を順に判別する。出てこなければ true を返すことで 0 個以上の繰り返しを実装する。

```
1   private boolean variableDeclearationsLine() {//変数宣言の並び
2     if(variableNameLine() && tokenSyntax(Token.SCOLON) && type() &&
          tokenSyntax(Token.SSEMICOLON)) {
3       while(variableNameLine()) {
4         if(!tokenSyntax(Token.SCOLON) || !type() ||
5                   !tokenSyntax(Token.SSEMICOLON))  return false;
6       }
7       return true;
8     }
9     syntaxErrorOutput();
10    return false;
11  }
```

4.2 左再帰性の除去

構文"文"の構文解析では"if" 式 "then" 複合文 "else" 複合文と"if" 式 "then" 複合文の両方を確認するが、if" 式 "then" 複合文が被っているため、先に if" 式 "then" 複合文の部分を判別してから"else" があるかどうかでその先を確認するかどうかを判別している。これによって左再帰性を除去している。

```
1 else if(tokenSyntax(Token.SIF) && expression() && tokenSyntax(Token.STHEN)
    && compoundStatement()) {
2       if (tokenSyntax(Token.SELSE)) {
3         if(compoundStatement()) return true;
4         syntaxErrorOutput();
5         return false;
6       }
7       return true;
8     }
```

4.3 左再帰性

一方、構文"基本文"の構文解析では左再帰性を残している。これは"代入文"と"手続き呼出し文"でどちらも最初のトークンが"SIDENTIFIER"であるという左再帰性があるのだが、それぞれが何重もの入れ子構造となっていて左再帰性をなくすとプログラムがとてもややこしくなり、新たに関数を複数作成する必要が出てきてしまう。そのため、一度"代入文"を判別する前にその時点の行番号を保存し、"代入文"の構文が間違っていた場合は保存した行番号に戻ってから"手続き呼び出し文"を呼び出すことで構文解析を行うようにしている。

```
1 else {
2         lineStack(rowNumber);
3         if(assignmentStatement()) return true;
4         previousLine();
5         if(procedureCallstatement()) return true;
6         return false;
7     }
```

5 まとめ

今回の構文解析では構文”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。0個以上の繰り返し文では while 文を使うことで解析を行い、左再帰性があるところでは共通部分を抜き出すことで出来るだけ左再帰性をなくした。ただし、構文”基本文”のみプログラムを見やすくするため左再帰性を残して解析をした。これらを用いたりして、拡張バッカス・ナウア記法で書かれた構文定義通りに関数を作成することで見やすく分かりやすい Parser を作成した。

6 感想

今回の演習ではあまりエラーがなくスムーズにプログラムを作成できました。プログラムは長くなってしまいましたが、ほぼ拡張バッカス・ナウア記法で書かれた構文定義通りに関数を作成しただけなのでプログラムとしてはとても見やすく、誰が見てもすぐに理解できるプログラムが作成できたと思います。Parser 自体のエラーはほとんど無かったのですが、ただ一つテストを実行する際に ParserTest プログラムで Unhandled exception type FileNotFoundException というエラーが出てきて、何十時間もこのエラーを直すのに使ってしまいました。TA さんに質問したところ、すぐにエラーが解消できたため、分からないところがあれば、すぐに TA さんに質問することがとても大切だと感じました。