情報科学演習Ｄ　課題2レポート

担当教員：松本　真佑
提出者：市川　達大
学籍番号：09B17010
メールアドレス：u243952e@ecs.osaka-u.ac.jp

提出年月日：2020年11月6日

1　課題の目的

コンパイラを作成する第 2 段階で、課題一で字句解析をしてプログラムをトークン分解したものを今回の課題 2 で構文的に正しいかどうかを判断していく。

2　課題の達成の方針と設計

2.1　外部仕様

課題 1 で字句解析した結果ファイル (ts ファイル) を読み込み、構文的に誤りがあるかどうかを判定する。正しい場合は標準出力に”OK”を出力し、構文が正しくない場合は”Syntax error: line”という文字列とともに最初の誤り部分の行番号を出力する。

2.2　方針と設計

以上の課題を達成するために主に以下の方針で設計し実装する。課題達成の要件として主に以下 3 つの課題がある。

1. トークンの前後関係の表現
2. EBNF の条件のメソッド化
3. 構文が正しいかどうかの判定

2.2.1　トークンの前後関係の表現

まず、課題 l でトークンれつに分化した ts ファイルを読み込み、トークン ID 部分と行数部分をリストへ別々に格納していく。今回はこの ID を格納したリストを前から順番に見ていくことで、構文の条件に当てはまっているかどうかを判定していき、条件に当てはまらないものが出てきた時点でエラーとして、行数を出力する。

2.2.2　EBNF の条件のメソッド化

条件で与えられた、EBNG 式を全てメソッド化し、大きい規則の中に小さな規則が内包される場合は、メソッド化した小さい規則の組み合わせで大きい規則を実現する。例えば、「ブロック」は「変数宣言」を判定するメソッドと「副プログラム宣言郡」を判定するメソッドを同時に真にしたときに真になる。

2.2.3　構文が正しいかどうかの判定

プログラムが正しい構文になっているかどうかの判定は、各式を判定するメソッドの組み合わせで実現する。具合的にいうと、プログラム = ”program” プログラム名 ”;” ブロック 複合文 ”.”の形になっていればよく、各「プログラム名」や「ブロック」、「複合文」などはより小さい単位のメソッドの組み合わせで実現されている。

3　プログラムの実装方法

上記の設計方針をもとに具体的にプログラムを実装していく。

3.1　トークンの前後関係の表現

まず読み込んだ ts ファイルから、トークンの ID 部分と行番号部分のみをリストへ格納する。

```
String line;
while((line = in.readLine()) != null) {//分割したトークンをリストへ格納
        String[] get_token = line.split("\t");

        //ID と行数のみを取得する
        get_id_list.add(get_token[2]);
        get_token_line.add(get_token[3]);
                        }
```

この get_id_list を順番に読み込んで見ていくことで、正しい構文になっているかを判定していく。また、このとき get_id_line のインデックスを表すグローバル変数 gr_index を宣言しておく。

3.2　EBNF 条件のメソッド化

書かれている条件を全てメソッド化する。こうすることで、「因子 = 変数 — 定数 — ”(” 式 ”)” — ”not” 因子.」 のような場合でも再起的にメソッドを呼び出すことで、条件にあっているかどうかの判定が楽になる。また、「基本文 = 代入文 — 手続き呼出し文 — 入出力文 — 複合文.」 のような場合でも「代入文」、「手続き呼び出し文」、「入出力文」、「複合文」の判定をメソッド化しているので、動作を考慮せずメソッドを組み合わせるだけで構文が正しいかどうかを判定でき、楽に実装が可能である。ここでメソッド化した式を以下に示す。

1. 「プログラム」- is_program
2. 「ブロック」- is_block
3. 「変数宣言」- is_variable_dec
4. 「副プログラム宣言郡」- is_subprogram_head
5. 「複合文」- is_compound
6. 「式の並び」- is_sentence
7. 「基本文」- is_basic_sentence
8. 「代入分」- is_subsituation
9. 「手続き呼び出し文」- is_procedure_call
10. 「入出力文」- is_inout
11. 「式」- is_fomula
12. 「因子」- is_factor

以下に例として「複合文」の部分を記述する。

```
1         //複合文判定
2         public String is_compound(ArrayList<String> id, ArrayList<String> line, int index) {
3                 String check = null;
4                 //begin
5                 if(! id.get(index).equals("2")) { return line.get(index); }
6                 index ++;
7
8                 //文の並び
9                 check = is_sentence(id,line,index);
10                if( ! (check.equals("0"))) { return check; }
11                index = gr_index;
12
13                //end
14                if(! id.get(index).equals("8")) { return line.get(index); }
15                index ++;
16
17                gr_index = index;
18                return "0";
19
20        }
```

上記を見ると複合分は、まず「begin」であるかを判定し、次のトークンから「式の並び」を満たすかどうかを判定し、最後に「end」がくれば真と判定できる。このようにメソッドを組み合わせることで比較的簡単に実装することができる。また、各メソッドでは int 型の数値を返り値とし、0 であれば真、それ以外であればエラーの行番号を返す。

3.3 構文が正しいかどうかの判定

構文判定のメソッドが定義できたら、最後にプログラム全体が構文規則に沿っているかを判定する。ここでは先ほど定義したメソッドを持ちいて、ID を格納したリストを前から順番に読み込んでいき、エラーになった時点でエラーの行番号を出力する。具体的にいうと、プログラムが「プログラム = ”program” プログラム名 ”;” ブロック 複合文 ”.”.」に沿っているかを判定していく。このとき、前から読んでいくインデックスはグローバル変数として宣言した `gr_index` で表し、各メソッドを呼び出すときは引数に `gr_index` を与えることで、どこまで読み込んでいるかを合わせる。以下にプログラム判定の箇所を記述する。

```
1  //”program” プログラム名 ”;”
2                         check = is_program(get_id_list,get_token_line,gr_index);
3                         if(! check.equals("0")) {
4                                 System.err.println("Syntax␣error:␣line␣" + check);
5                                 return;
6                         }
7                         //System.out.println(”program ok”);
8                         //System.out.println(gr_index);
9                         //ブロック
10                        check = is_block(get_id_list,get_token_line,gr_index);
11                        if(! check.equals("0")) {
12                                System.err.println("Syntax␣error:␣line␣" + check);
13                                return;
14                        }
15
16                        //複合文
17                        check = is_compound(get_id_list,get_token_line,gr_index);
18
19                        if(! check.equals("0")) {
20                                System.err.println("Syntax␣error:␣line␣" + check);
21                                return;
22                        }
```

```
23
24
25                          //.
26                          if(! get_id_list.get(gr_index).equals("42")) {
27                                  System.err.println("Syntax␣error:␣line␣" + get_token_line.
                                        get(gr_index));
28                                  return;
29                          }
30
31                          System.out.println("OK");
```

上記のようにプログラム全体がプログラム = ”program” プログラム名 ”;” ブロック 複合文 ”.”.」の規則に沿っていれば、標準出力に”OK”を出力する。また、規則に沿っていないものがあるとき、全てのメソッドの返り値として、誤りがある行番号を返り値としているので、その時点でプログラムを終了する。

4 まとめ

今回の課題 2 では、構文解析を作成するのを通して、連続したデータに関する処理を学ぶことができた。今回はリストにトークン列の ID を保持して、前から読み込むことで構文判定を行ってきたが、課題 3 の意味解析のことを考えれば、木構造に保存しておいた方が次の実装が楽になる。

5 感想

今回の課題では、EBNF 記法を細かくメソッドに分けたがクラス分けはしていないので、デバッグの際に依存関係がややこしくなった場面があったのでクラス分けをした方がよかったなと思った。また今回のような複雑なコードでエラーが出た際、挙動がわかりにくくデバッグに時間がかかったので、機能ごとにデバッグをするなどをしてこまめにテストを行っていけば、エラー修正もやりやすかったのではないかと思う。