実践的プログラミング　字句解析

ctare

2017 年 11 月 29 日

1　ソースコード

1.1　Main

リスト 1　Main.java

```
1  import newlang4.*;
2
3  import java.io.FileNotFoundException;
4
5  /**
6   * Created by ctare on 2017/09/26.
7   */
8  public class LexMain {
9      // 字句解析確認用
10     public static void main(String[] args) throws FileNotFoundException {
11         String fileName = "resource/input";
12         LexicalAnalyzerImpl lexicalAnalyzer = new LexicalAnalyzerImpl(fileName);
13         while(true) {
14             try {
15                 LexicalUnit unit = lexicalAnalyzer.get();
16                 System.out.println(unit);
17                 if (unit.getType() == LexicalType.END) break;
18             } catch (Exception e) {
19                 e.printStackTrace();
20                 break;
21             }
22         }
23     }
24 }
```

1.2　LexicalAnalyzer

リスト 2　LexicalAnalyzerImpl.java

```
1  package newlang4;
2
3  import java.io.FileNotFoundException;
4  import java.io.FileReader;
5  import java.io.IOException;
6  import java.io.PushbackReader;
7  import java.util.*;
8  import java.util.regex.Matcher;
9  import java.util.regex.Pattern;
10
11 /**
12  * Created by ctare on 2017/09/26.
13  */
14 public class LexicalAnalyzerImpl implements LexicalAnalyzer {
15     private Deque<LexicalUnit> back = new ArrayDeque<>();
16     private PushbackReader in;
17     private String all;
```

```
18      private LinkedHashMap<LexicalType, Def> patternMap = new LinkedHashMap<
            LexicalType, Def>(){{
19          put(LexicalType.INTVAL, new PatternDefWithValueType(Pattern.compile("
                ^([0-9]+)"), ValueType.INTEGER, Integer::parseInt));
20          put(LexicalType.DOUBLEVAL, new PatternDefWithValueType(Pattern.compile("
                ^([0-9]+(\\.[0-9]+)?)"), ValueType.DOUBLE, Double::parseDouble));
21          put(LexicalType.NAME, new PatternNameDef(Pattern.compile("^([a-zA-Z_]+[0-9_
                ]*)")));
22          put(LexicalType.ADD, new PatternDef(Pattern.compile("^(\\+)")));
23          put(LexicalType.SUB, new PatternDef(Pattern.compile("^(-)")));
24          put(LexicalType.MUL, new PatternDef(Pattern.compile("^(\\*)")));
25          put(LexicalType.DIV, new PatternDef(Pattern.compile("^(/)")));
26          put(LexicalType.COMMA, new PatternDef(Pattern.compile("^(,)")));
27          put(LexicalType.DOT, new PatternDef(Pattern.compile("^(\\.)")));
28          put(LexicalType.LE, new PatternDef(Pattern.compile("^(<=|=<)")));
29          put(LexicalType.GE, new PatternDef(Pattern.compile("^(>=|=>)")));
30          put(LexicalType.GE, new PatternDef(Pattern.compile("^(<>)")));
31          put(LexicalType.EQ, new PatternDef(Pattern.compile("^(=)")));
32          put(LexicalType.LT, new PatternDef(Pattern.compile("^(<)")));
33          put(LexicalType.GT, new PatternDef(Pattern.compile("^(>)")));
34          put(LexicalType.LP, new PatternDef(Pattern.compile("^(\\()")));
35          put(LexicalType.RP, new PatternDef(Pattern.compile("^(\\))")));
36          put(LexicalType.LITERAL, new PatternDefWithValueType(Pattern.compile("^\"
                (.*)\""), ValueType.STRING, value -> value));
37          put(LexicalType.NL, new PatternDef(Pattern.compile("^(\n)")));
38      }};
39
40      private Pattern[] ignore = {
41              Pattern.compile("^([␣\t])"),
42      };
43
44      public LexicalAnalyzerImpl(String fileName) throws FileNotFoundException {
45          this.in = new PushbackReader(new FileReader(fileName));
46          StringBuilder all_s = new StringBuilder();
47          while(true) {
48              int c = -1;
49              try {
50                  c = this.in.read();
51              } catch (IOException e) {
52                  // ignored
53              }
54              if(c == -1) break;
55              all_s.append((char) c);
56          }
57
58          all = all_s.toString();
59      }
60
61      @Override
62      public LexicalUnit get() throws Exception{
63          if(!back.isEmpty()) {
64              return back.pop();
65          }
66          while(true) {
67              boolean break_flg = true;
68              for (Map.Entry<LexicalType, Def> lexicalTypePatternEntry : patternMap.
                    entrySet()) {
69                  Return ret = lexicalTypePatternEntry.getValue().match(
                        lexicalTypePatternEntry.getKey(), all);
70                  if(ret != null) {
71                      all = ret.result;
72                      return ret.lexicalUnit;
73                  }
74              }
75
76              for (Pattern pattern : ignore) {
77                  Matcher matcher = pattern.matcher(all);
78                  if(matcher.find()){
79                      all = all.substring(matcher.end());
```

```
                    break_flg = false;
                }
            }
            if(break_flg) break;
        }
        return new LexicalUnit(LexicalType.EOF);
    }

    @Override
    public boolean expect(LexicalType type) throws Exception {
        return false;
    }

    @Override
    public void unget(LexicalUnit token) throws Exception {
        back.push(token);
    }

    private interface Def {
        Return match(LexicalType type, String target);
    }

    private static class Return {
        private String result;
        private String value;
        private LexicalUnit lexicalUnit;

        Return(String result, String value, LexicalUnit lexicalUnit) {
            this.result = result;
            this.value = value;
            this.lexicalUnit = lexicalUnit;
        }
    }

    private static class PatternDef implements Def {
        private Pattern pattern;

        PatternDef(Pattern pattern) {
            this.pattern = pattern;
        }

        @Override
        public Return match(LexicalType type, String target) {
            Matcher matcher = pattern.matcher(target);
            if(matcher.find()) {
                target = target.substring(matcher.end());
                String value = matcher.group(1);
                return new Return(target, value, new LexicalUnit(type, createValue(
                    value)));
            }
            return null;
        }

        Value createValue(String value) {
            return null;
        }
    }

    private static class PatternNameDef extends PatternDef {
        private LexicalType[] types = new LexicalType[]{
                LexicalType.IF,
                LexicalType.THEN,
                LexicalType.ELSE,
                LexicalType.ELSEIF,
                LexicalType.ENDIF,
                LexicalType.FOR,
                LexicalType.FORALL,
                LexicalType.NEXT,
                LexicalType.FUNC,
```

```
                LexicalType.DIM,
                LexicalType.AS,
                LexicalType.END,
                LexicalType.WHILE,
                LexicalType.DO,
                LexicalType.UNTIL,
                LexicalType.LOOP,
                LexicalType.TO,
                LexicalType.WEND,
                LexicalType.EOF,
        };

        PatternNameDef(Pattern pattern) {
            super(pattern);
        }

        @Override
        public Return match(LexicalType type, String target) {
            Return ret = super.match(type, target);
            if(ret == null) return null;
            for (LexicalType lexicalType : types) {
                if(lexicalType.name().equals(ret.value)) {
                    ret.lexicalUnit.type = lexicalType;
                    break;
                }
            }
            return ret;
        }

        @Override
        Value createValue(String value) {
            return new ValueImpl(value, ValueType.STRING);
        }
    }

    private static class PatternDefWithValueType extends PatternDef {
        private ValueType type;
        private Cast cast;

        PatternDefWithValueType(Pattern pattern, ValueType type, Cast cast) {
            super(pattern);
            this.type = type;
            this.cast = cast;
        }

        @Override
        Value createValue(String value) {
            return new ValueImpl(cast.exec(value), type);
        }

        @FunctionalInterface
        interface Cast{
            Object exec(String value);
        }
    }
}
```

1.3 ValueImpl

リスト 3　ValueImpl.java

```
package newlang4;

import java.util.HashMap;

/**
 * Created by ctare on 2017/11/07.
```

```
 */
public class ValueImpl implements Value {
    private Object value;
    private ValueType type;

    public ValueImpl(Object value, ValueType type) {
        this.value = value;
        this.type = type;
    }

    @Override
    public String getSValue() {
        return value.toString();
    }

    @Override
    public int getIValue() {
        return (int) value;
    }

    @Override
    public double getDValue() {
        return (double) value;
    }

    @Override
    public boolean getBValue() {
        return (boolean) value;
    }

    @Override
    public ValueType getType() {
        return type;
    }
}
```

2 コード説明

2.1 Main.java

リスト 1 では、メインメソッドを定義している。メインメソッドでは、字句解析が正しく行われているか確認する処理を書いている。LexicalAnalyzer から get メソッドを LexicalType が EOF になるまで呼び続け、どのような LexicalUnit が取得されているかを出力して確認している。

2.2 LexicalAnalyzerImpl.java

リスト 2 では、字句解析の処理を定義している。コンストラクタでは、ファイル名を受け取り、始めに全文字読み込んでいる。

字句はそれぞれ正規表現を用いて定義している。また、字句はそれぞれ前方一致として定義しており、コードの先頭から一致するものを探している。読み飛ばす字句は別途定義しており、読み飛ばさない字句を一致するか全て試した後、読み飛ばす字句の一致確認を行っている。

get メソッドでは、始めに、読み飛ばさない字句を全て確認し、一致した場合、現在見ているコードの位置を一致した文字列分後方にずらした後、LexicalUnit を生成して戻り値としている。いずれにも一致しなかった場合、読み飛ばす字句を全て確認し、一致した場合、現在見ているコードの位置を一致した文字列文後方にずらした後、再度読み飛ばさない字句の確認を行う。さらに、いずれにも一致しなかった場合、EOF として LexicalUnit を戻り値としている。

98 行目以降に定義してあるクラスは、Pattern クラスが final で定義されており、継承をする事が出来なかったため、委譲を行う形で Pattern を拡張するためのクラスである。

2.2.1 Def

対象文字列が、定義されている字句に一致するかを確認する。一致していた場合、Return を戻り値としている。

2.2.2 Return

Def にて用いられている戻り値の定義である。result は、現在見ているコード位置を一致文字列分後方にずらした後の文字列である。value は、一致文字列である。lexicalUnit は、一致した字句である。

2.2.3 PatternDef

Pattern クラスを委譲するクラスである。正規表現との一致を確認し、一致した場合 Return を戻り値とし、一致しなかった場合 null を戻り値としている。

2.2.4 PatternNameDef

PatternDef クラスを拡張している。PatternDef クラスにて一致を確認した後、予約語の確認を行っている。定義されている予約語と PatternDef にて取り出した文字列が完全一致した場合、lexicalUnit の type を変更している。

2.2.5 PatternDefWithValueType

PatternDef クラスを拡張している。インスタンスを作る際に ValueType と値の変換方法を指定することで、字句を生成する際に、それらを用いて ValueImpl クラスのインスタンスを生成する。

2.3 ValueImpl.java

想定される型に変換済みの値と、型をコンストラクタの引数に取る。getSValue を除く、それぞれの getValue では、渡された値が変換済みである事を利用してただキャストして戻り値としている。

3 結果

3.1 入力

リスト 4 入力

```
1  a = 5
2  DO UNTIL a < 1
3      PRINT ("Hello")
4      a = a - 1
5  LOOP
6  END
```

3.2 出力

リスト 5 出力

```
1  DIM
2  NAME: i
3  NL
4  FOR
5  NAME: i
6  EQ
7  INTVAL: 1
8  TO
9  INTVAL: 100
10 NL
11 NAME: PRINT
12 LITERAL:  Hello World
```

```
13 NL
14 NEXT
15 NAME: i
16 EOF
```