実践的プログラミング

氏名: 中村宥哉
学籍番号: C0117237
提出日: 2019/11/28

1. Lexical Analyzerの製作

作成したコードを説明する。

ソースコード1 Main.java

```
1  package newlang3;
2
3
4  public class Main {
5          public static void main (String[] args){
6                  String fileName = "./src/newlang3/src.txt";
7                  LexicalAnalyzer la;
8                  try {
9                          la = new LexicalAnalyzerImpl(fileName);
10                 } catch (Exception e) {
11                         e.printStackTrace();
12                         return;
13                 }
14
15                 LexicalUnit lu;
16
17                 try {
18                         while ((lu = la.get()).getType() != LexicalType.EOF) {
19                                 System.out.println(lu);
20                         }
21                 } catch (Exception e) {
22                         e.printStackTrace();
23                         return;
24                 }
25         }
26 }
27
```

4行目の `LexicalAnalyzer la;` では `LexicalAnalyzer` 型の変数 `la` を宣言し、以後はこれを利用して字句解析を行う。

8行目からの `try - catch` 文ではファイルを読み込む処理を記述している。ファイルの読み込みに失敗した場合はエラーを表示して終了する。

18行目からの `while` 文では `la` から `LexicalUnit` のインスタンスを一つづつ受け取り表示している。 `EOF` に到達した場合は `while` 文の条件式が `false` を返し終了する。

ソースコード2 LexicalAnalyzer.java

```
1  package newlang3;
2
3  public interface LexicalAnalyzer {
4      public LexicalUnit get() throws Exception;
5      public boolean expect(LexicalType type) throws Exception;
6      public void unget(LexicalUnit token) throws Exception;
7  }
8
```

`LexicalAnalyzerImpl` で実装するインタフェースの定義である。

ソースコード3 LexicalAnalyzerImpl.java

```
package newlang3;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PushbackReader;
import java.util.HashMap;

public class LexicalAnalyzerImpl implements LexicalAnalyzer {
        PushbackReader reader;

        static HashMap<String, LexicalUnit> reservedWords = new HashMap<String,
        static HashMap<String, LexicalUnit> punctuators = new HashMap<String, Le

        static {
                reservedWords.put("if", new LexicalUnit(LexicalType.IF));
                reservedWords.put("then", new LexicalUnit(LexicalType.THEN));
                reservedWords.put("else", new LexicalUnit(LexicalType.ELSE));
                reservedWords.put("elseif", new LexicalUnit(LexicalType.ELSEIF))
                reservedWords.put("endif", new LexicalUnit(LexicalType.ENDIF));
                reservedWords.put("for", new LexicalUnit(LexicalType.FOR));
                reservedWords.put("forall", new LexicalUnit(LexicalType.FORALL))
                reservedWords.put("next", new LexicalUnit(LexicalType.NEXT));
                reservedWords.put("sub", new LexicalUnit(LexicalType.SUB));
                reservedWords.put("dim", new LexicalUnit(LexicalType.DIM));
                reservedWords.put("as", new LexicalUnit(LexicalType.AS));
                reservedWords.put("end", new LexicalUnit(LexicalType.END));
                reservedWords.put("while", new LexicalUnit(LexicalType.WHILE));
                reservedWords.put("do", new LexicalUnit(LexicalType.DO));
                reservedWords.put("until", new LexicalUnit(LexicalType.UNTIL));
                reservedWords.put("loop", new LexicalUnit(LexicalType.LOOP));
                reservedWords.put("to", new LexicalUnit(LexicalType.TO));
                reservedWords.put("wend", new LexicalUnit(LexicalType.WEND));

                punctuators.put("=", new LexicalUnit(LexicalType.EQ));
                punctuators.put("<", new LexicalUnit(LexicalType.LT));
                punctuators.put(">", new LexicalUnit(LexicalType.GT));
                punctuators.put("<=", new LexicalUnit(LexicalType.LE));
                punctuators.put("=<", new LexicalUnit(LexicalType.LE));
                punctuators.put(">=", new LexicalUnit(LexicalType.GE));
                punctuators.put("=>", new LexicalUnit(LexicalType.GE));
                punctuators.put("<>", new LexicalUnit(LexicalType.NE));
                punctuators.put("\n", new LexicalUnit(LexicalType.NL));
                punctuators.put(".", new LexicalUnit(LexicalType.DOT));
                punctuators.put("+", new LexicalUnit(LexicalType.ADD));
                punctuators.put("-", new LexicalUnit(LexicalType.SUB));
                punctuators.put("*", new LexicalUnit(LexicalType.MUL));
                punctuators.put("/", new LexicalUnit(LexicalType.DIV));
                punctuators.put("(", new LexicalUnit(LexicalType.LP));
                punctuators.put(")", new LexicalUnit(LexicalType.RP));
                punctuators.put(",", new LexicalUnit(LexicalType.COMMA));
        }

```

```
55          public LexicalAnalyzerImpl (String fileName) throws FileNotFoundExceptic
56                  reader = new PushbackReader(new BufferedReader(new FileReader(fi
57          }
58
59          @Override
60          public LexicalUnit get() throws IOException {
61                  char ch = (char)reader.read();
62
63                  char EOF = (char)-1;
64
65                  while (ch == ' ' || ch == '\t' || ch == '\r') {
66                          ch = (char)reader.read();
67                          if (ch == EOF) {
68                                  return new LexicalUnit(LexicalType.EOF);
69                          }
70                  }
71
72                  if (ch == EOF) {
73                          return new LexicalUnit(LexicalType.EOF);
74                  }
75
76                  String input = String.valueOf(ch);
77
78                  reader.unread(ch);
79
80                  if (input.matches("[0-9]")) {
81                          return getNumber();
82                  }
83                  if (input.matches("[a-zA-Z_]")) {
84                          return getString();
85                  }
86                  if (input.matches("\"")) {
87                          return getLiteral();
88                  }
89                  return getPunctuator();
90          }
91
92          public boolean findUnit () {
93                  return false;
94          }
95
96          @Override
97          public boolean expect(LexicalType type) throws Exception {
98
99                  return false;
100         }
101
102         @Override
103         public void unget(LexicalUnit token) throws Exception {
104
105         }
106
107         private LexicalUnit getNumber () throws IOException {
108                 String target = "";
109
```

```
110                 char ch;
111
112                 boolean isDouble = false;
113
114                 while (true) {
115                         ch = (char)reader.read();
116                         if (ch < 0) break;
117                         if ((target + ch).matches("[0-9]+(\\.[0-9]*)?")) {
118                                 target += ch;
119                                 if (target.matches("[0-9]+\\.[0-9]+")) {
120                                         isDouble = true;
121                                 }
122                                 continue;
123                         }
124                         reader.unread(ch);
125                         break;
126                 }
127
128                 return isDouble
129                 ? new LexicalUnit(
130                         LexicalType.DOUBLEVAL,
131                         new ValueImpl(target, ValueType.DOUBLE)
132                 )
133                 : new LexicalUnit(
134                         LexicalType.INTVAL,
135                         new ValueImpl(target, ValueType.INTEGER)
136                 );
137         }
138
139         private LexicalUnit getString () throws IOException {
140                 String target = "";
141
142                 char ch;
143
144                 while (true) {
145                         ch = (char)reader.read();
146                         if (ch < 0) break;
147
148                         if ((target + ch).matches("[a-zA-Z_][a-zA-Z0-9_]*")) {
149                                 target += ch;
150                                 continue;
151                         }
152                         reader.unread(ch);
153                         break;
154                 }
155
156                 String key = target.toLowerCase();
157
158                 LexicalUnit lu = reservedWords.get(key);
159
160                 if (lu != null) {
161                         return lu;
162                 }
163
164                 return new LexicalUnit(
```

```
165                         LexicalType.NAME,
166                         new ValueImpl(target, ValueType.STRING)
167                 );
168         }
169
170         private LexicalUnit getLiteral () throws IOException {
171                 String target = "";
172
173                 char ch;
174
175                 reader.read();
176
177                 while (true) {
178                         ch = (char)reader.read();
179                         if (ch < 0) break;
180                         if (ch != '"') {
181                                 target += ch;
182                                 continue;
183                         }
184                         break;
185                 }
186
187                 return new LexicalUnit(
188                         LexicalType.LITERAL,
189                         new ValueImpl(target)
190                 );
191         }
192
193         private LexicalUnit getPunctuator () throws IOException {
194                 String target = "";
195
196                 char ch;
197                 String str;
198                 LexicalUnit prev = null;
199
200                 while (true) {
201                         ch = (char)reader.read();
202                         str = String.valueOf(ch);
203                         if (ch < 0) break;
204                         if (str.matches("=|<|>|\\+|-|\\*|/|\\(|\\)|,|\\n")) {
205                                 target += str;
206                                 if (punctuators.get(target) != null || prev == nu
207                                         prev = punctuators.get(target);
208                                         continue;
209                                 }
210                         }
211                         reader.unread(ch);
212                         break;
213                 }
214                 return prev;
215         }
216 }
217
```

11行目では字句解析に用いる PushbackReader 型変数を宣言している。
13行目では HashMap<String, LexicalUnit> 型の予約語を保持する変数を宣言している。
14行目では HashMap<String, LexicalUnit> 型の演算子等の記号を保持する変数を宣言している。

16~53行目では予約語と演算子等の記号をそれぞれの変数に登録している。

55行目からのコンストラクタではファイル名を受け取って PushbackReader 型のインスタンスを生成している。ファイルが見つからなかった場合は FileNotFoundException をスローする。

60行目からの get メソッドでは現在位置からみて最初に現れる字句を返す処理を行っている。文字の読み込みに失敗した場合は IOException をスローする。

65行目からの while 文では ' ', '\t', '\r' を読み飛ばす処理を行っている。読み飛ばしている間に EOF に到達した場合は new LexicalUnit(LexicalType.EOF) を返す。

改行の直後に EOF が存在する場合は while 文内の return 文が実行されないため、72行目で EOF の LexicalUnit を返す。

76行目では後に正規表現を利用するため String 型変数に文字を代入している。

80~88行目では正規表現を利用して 数値, 文字列, リテラル, 記号 を取得するメソッドを呼び出している。

107行目からの getNumber メソッドでは INTEGER 型, DOUBLE 型の字句を取得して返す処理を行う。

117行目の if 文では取得した文字列が INTEGER 型か DOUBLE 型である場合に内部が実行され、 target に取得した文字列を格納する。targetに格納された文字列が DOUBLE 型である場合には120行目の isDouble = true が実行される。

128行目からの return 文では isDouble の値によって分岐し、 target が整数である場合には INTEGER の LexicalUnit を、浮動小数点数である場合は DOUBLE の LexicalUnit を返却する。

139行目からの getString メソッドでは、予約語や変数名等の文字列を取得して返す処理を行う。

予約語の大文字小文字を区別せずに検索するため、156行目でヒットした文字列をすべて小文字に変換する処理を記述している。

158行目ではヒットした文字列をキーとする予約語を取得している。該当がない場合は null が代入される。

160行目からの if 文ではヒットした文字列が予約語であった場合にその予約語の LexicalUnit を返却する処理を記述している。

164行目の return では予約語以外の文字列の LexicalUnit を返却している。

170行目からの `getLiteral` メソッドでは、文字列リテラルを取得して返す処理を行う。

177行目からの `while` 文では次 `"` が出現するまでリテラル内部の文字列を `target` に追記している。(`"` のエスケープには対応していない。)

187行目では `target` から生成した `LITERAL` の `LexicalUnit` を返却している。

193行目からの `getPunctuator` メソッドでは、記号や改行を取得して返す処理を行う。

200行目からの `while` 文、 `return` 文では一文字ずつ読み進め、該当する記号が存在しなくなるまで回し、存在しなくなる直前に該当した記号の `LexicalUnit` を返却する。これにより `>` や `>=` を識別する。

ソースコード4 LexicalType.java

```
package newlang3;

public enum LexicalType {
        LITERAL,        // 文字列定数   （例：  “文字列”）
        INTVAL,         // 整数定数     （例：  ３）
        DOUBLEVAL,      // 小数点定数   （例：  １．２）
        NAME,           // 変数         （例：  i）
        IF,                     // IF
        THEN,           // THEN
        ELSE,           // ELSE
        ELSEIF,         // ELSEIF
        ENDIF,          // ENDIF
        FOR,            // FOR
        FORALL,         // FORALL
        NEXT,       // NEXT
        EQ,                     // =
        LT,                     // <
        GT,                     // >
        LE,                     // <=, =<
        GE,                     // >=, =>
        NE,                     // <>
        DIM,            // DIM
        AS,                     // AS
        END,            // END
        NL,                     // 改行
        DOT,            // .
        WHILE,          // WHILE
        DO,                     // DO
        UNTIL,          // UNTIL
        ADD,            // +
        SUB,            // -
        MUL,            // *
        DIV,            // /
        LP,                     // )
        RP,                     // (
        COMMA,          // ,
        LOOP,           // LOOP
        TO,                     // TO
        WEND,           // WEND
        EOF,            // end of file
}

```

終端記号の種類を定義している。

ソースコード5 LexicalUnit.java

```
package newlang3;


public class LexicalUnit {
        LexicalType type;
        Value value;
        LexicalUnit link;

        public LexicalUnit(LexicalType this_type) {
            type = this_type;
        }

        public LexicalUnit(LexicalType this_type, Value this_value) {
                type = this_type;
                value = this_value;
        }

        public Value getValue() {
                return value;
        }

        public LexicalType getType() {
                return type;
        }

        public String toString() {
                switch(type) {
            case LITERAL:
            return "LITERAL:\t" + value.getSValue();
            case NAME:
                return "NAME:\t" + value.getSValue();
            case DOUBLEVAL:
                return "DOUBLEVAL:\t" + value.getSValue();
            case INTVAL:
                return "INTVAL:\t" + value.getSValue();
            case IF:
            return ("IF");
            case THEN:
            return ("THEN");
            case ELSE:
            return ("ELSE");
            case FOR:
            return ("FOR");
            case FORALL:
            return ("FORALL");
            case NEXT:
            return ("NEXT");
            case SUB:
            return ("SUB");
            case DIM:
            return ("DIM");
            case AS:
            return ("AS");
            case END:
```

```
55              return ("END");
56              case EOF:
57              return ("EOF");
58              case NL:
59              return ("NL");
60              case EQ:
61              return ("EQ");
62              case LT:
63                  return ("LT");
64              case GT:
65                  return ("GT");
66              case LE:
67                  return ("LE");
68              case GE:
69                  return ("GE");
70              case DOT:
71                  return ("DOT");
72              case WHILE:
73                  return ("WHILE");
74              case DO:
75                  return ("DO");
76              case UNTIL:
77                  return ("UNTIL");
78              case ADD:
79                  return ("ADD");
80              case MUL:
81                  return ("MUL");
82              case DIV:
83                  return ("DIV");
84              case LP:
85                  return ("LP");
86              case RP:
87                  return ("RP");
88              case COMMA:
89                  return ("COMMA");
90              case LOOP:
91                  return ("LOOP");
92              case TO:
93                  return ("TO");
94              case WEND:
95                  return ("WEND");
96              case ELSEIF:
97                  return ("ELSEIF");
98              case NE:
99                  return ("NE");
100             case ENDIF:
101                 return ("ENDIF");
102             }
103             return "";
104         }
105 }
106
```

トークンの情報を保持するクラスである。

ソースコード6 Value.java

```
1  package newlang3;
2
3  public abstract class Value {
4  // 実装すべきコンストラクタ
5      public Value(String s) {};
6      public Value(int i) {};
7      public Value(double d) {};
8      public Value(boolean b) {};
9      public Value(String s, ValueType t) {};
10     public abstract String get_sValue();
11         public abstract String getSValue();
12         // ストリング型で値を取り出す。必要があれば、型変換を行う。
13     public abstract int getIValue();
14         // 整数型で値を取り出す。必要があれば、型変換を行う。
15     public abstract double getDValue();
16         // 小数点型で値を取り出す。必要があれば、型変換を行う。
17     public abstract boolean getBValue();
18         // 論理型で値を取り出す。必要があれば、型変換を行う。
19     public abstract ValueType getType();
20 }
21
```

`ValueImpl` クラスで継承する抽象クラスである。

ソースコード7 ValueImpl.java

```
1  package newlang3;
2
3  public class ValueImpl extends Value {
4
5          private ValueType type;
6          private String valString;
7          private int valInteger;
8          private double valDouble;
9          private boolean valBoolean;
10
11         public ValueImpl (String s) {
12                 super(s);
13                 valString = s;
14                 type = ValueType.STRING;
15         }
16     public ValueImpl (int i) {
17         super(i);
18         valInteger = i;
19         type = ValueType.INTEGER;
20     }
21     public ValueImpl (double d) {
22         super(d);
23         valDouble = d;
24         type = ValueType.DOUBLE;
25     }
26     public ValueImpl (boolean b) {
27         super(b);
28         valBoolean = b;
29         type = ValueType.BOOL;
30     }
31     public ValueImpl (String s, ValueType t) {
32         super(s, t);
33         switch(t) {
34                 case STRING:
35                         valString = s;
36                         break;
37                 case INTEGER:
38                         valInteger = Integer.parseInt(s);
39                         break;
40                 case DOUBLE:
41                         valDouble = Double.parseDouble(s);
42                         break;
43                 case BOOL:
44                         valBoolean = Boolean.parseBoolean(s);
45                         break;
46                 case VOID:
47                         break;
48                 }
49                 type = t;
50     }
51
52         @Override
53         public String get_sValue() {
54                 return getSValue();
```

```
55          }
56
57          @Override
58          public String getSValue() {
59                  switch(type) {
60                  case STRING:
61                          return valString;
62                  case INTEGER:
63                          return String.valueOf(valInteger);
64                  case DOUBLE:
65                          return String.valueOf(valDouble);
66                  case BOOL:
67                          return String.valueOf(valBoolean);
68                  default:
69                          return null;
70                  }
71          }
72
73          @Override
74          public int getIValue() {
75                  switch(type) {
76                  case STRING:
77                          return Integer.parseInt(valString);
78                  case INTEGER:
79                          return valInteger;
80                  case DOUBLE:
81                          return (int)valDouble;
82                  case BOOL:
83                          return valBoolean ? 1 : 0;
84                  default:
85                          return 0;
86                  }
87          }
88
89          @Override
90          public double getDValue() {
91                  switch(type) {
92                  case STRING:
93                          return Double.parseDouble(valString);
94                  case INTEGER:
95                          return (double)valInteger;
96                  case DOUBLE:
97                          return valDouble;
98                  case BOOL:
99                          return valBoolean ? 1.0 : 0.0;
100                 default:
101                         return 0.0;
102                 }
103         }
104
105         @Override
106         public boolean getBValue() {
107                 switch(type) {
108                 case STRING:
109                         return Boolean.parseBoolean(valString);
```

```
110                 case INTEGER:
111                         return (valInteger != 0) ? true : false;
112                 case DOUBLE:
113                         return (valDouble != 0) ? true : false;
114                 case BOOL:
115                         return valBoolean;
116                 default:
117                         return false;
118                 }
119         }
120
121         @Override
122         public ValueType getType() {
123                 return type;
124         }
125
126  }
127
```

11行目からのコンストラクタ群は各型の引数をとり、その値を保持する処理を行う。

57行目からの `getType` 以外の `get~` メソッド群は型変換を行いその結果を返す処理を行う。

122行目の `getType` メソッドはインスタンスが保持している値の型を返す。

ソースコード8 ValueType.java

```
1  package newlang3;
2
3  public enum ValueType {
4          VOID,
5          INTEGER,
6          STRING,
7          DOUBLE,
8          BOOL,
9  }
10
```

値の型を定義している。

2. 実行例

ソースコード9 src.txt

```
a = 500

DO UNTIL a < 1

PRINT ("Hello")

a = a - 1

LOOP

END
```

ソースコード9を制作したLexicalAnalyzerで解析すると実行結果1の出力が得られた。

実行結果1 実行結果

```
NAME:   a
EQ
INTVAL: 500
NL
NL
DO
UNTIL
NAME:   a
LT
INTVAL: 1
NL
NL
NAME:   PRINT
LP
LITERAL:        Hello
RP
NL
NL
NAME:   a
EQ
NAME:   a
SUB
INTVAL: 1
NL
NL
LOOP
NL
NL
END
NL
```