実践的プログラミング　構文解析

ctare

2017/12/27

1　ソースコード

1.1　Main

リスト 1　TreeMain.java

```
1  import newlang4.*;
2  import newlangdef.Program;
3
4  public class TreeMain {
5      // 構文木確認用
6      public static void main(String[] args) throws Exception {
7          LexicalAnalyzer lex;
8          Environment env;
9          Node program;
10
11         System.out.println("basic␣parser");
12         lex = new LexicalAnalyzerImpl("resource/input");
13         env = new Environment(lex);
14
15         while(true) {
16             program = NodeUtil.isMatch(Program.class, env, null);
17             if(program == null || !program.parse()) {
18                 break;
19             }
20         }
21     }
22 }
```

1.2　Node

リスト 2　Node.java

```
1  package newlang4;
2
3  public class Node {
4      NodeType type;
5      Environment env;
6
7      /** Creates a new instance of Node */
8      public Node() {
9      }
10     public Node(NodeType my_type) {
11         type = my_type;
12     }
13     public Node(Environment my_env) {
14         env = my_env;
15     }
16
17     public NodeType getType() {
18         return type;
19     }
```

```
20
21      public boolean parse() throws Exception {
22          return true;
23      }
24
25      public Value getValue() throws Exception {
26          return null;
27      }
28
29      public String toString() {
30          if (type == NodeType.END) return "END";
31          else return "Node";
32      }
33
34  }
```

1.3 NodeUtil

リスト 3　NodeUtil.java

```
1   package newlang4;
2
3   import java.lang.annotation.*;
4   import java.lang.reflect.Field;
5   import java.util.*;
6   import java.util.stream.Collectors;
7   import java.util.stream.Stream;
8
9   /**
10   * Created by ctare on 2017/12/05.
11   */
12  public final class NodeUtil {
13      public static Node isMatch(Class<? extends Node> cls, Environment my_env,
            LexicalUnit first){
14          Define define = cls.getAnnotation(Define.class);
15          if(define == null) {
16              return null;
17          }
18          try {
19              Field field = cls.getDeclaredField(define.firstSet());
20              FirstSet firstSet = (FirstSet) field.get(null);
21
22              NodeType nodeType = define.type();
23
24              if(firstSet.contains(first)) {
25                  Node instance = cls.newInstance();
26                  instance.env = my_env;
27                  instance.type = nodeType;
28                  if(cls.getAnnotation(SimpleParse.class) != null) {
29                      instance = new SimpleParseDecorator(instance);
30                  }
31                  return instance;
32              }
33              return null;
34          } catch (NoSuchFieldException | IllegalAccessException |
                InstantiationException e) {
35              e.printStackTrace();
36              return null;
37          }
38      }
39
40      public static class FirstSet {
41          private Set<LexicalType> firstSet = new HashSet<>();
42          public FirstSet(LexicalType... types) {
43              add(types);
44          }
45
```

```
46          public FirstSet(FirstSet another) {
47              merge(another);
48          }
49
50          public void add(LexicalType... types) {
51              Collections.addAll(firstSet, types);
52          }
53
54          public boolean contains(LexicalUnit lexicalUnit) {
55              return firstSet.contains(lexicalUnit.type);
56          }
57
58          public FirstSet merge(FirstSet another) {
59              firstSet.addAll(another.firstSet);
60              return this;
61          }
62
63          @Override
64          public String toString() {
65              return firstSet.toString();
66          }
67      }
68
69      @Retention(RetentionPolicy.RUNTIME)
70      @Target(ElementType.TYPE)
71      public @interface Define {
72          NodeType type();
73          String firstSet() default "firstSet";
74          String children() default "children";
75      }
76
77      @Retention(RetentionPolicy.RUNTIME)
78      @Target(ElementType.TYPE)
79      public @interface SimpleParse {
80      }
81
82      public interface HasSyntaxTree {
83          SyntaxTree getSyntaxTree();
84      }
85
86      private static class SimpleParseDecorator extends Node implements HasSyntaxTree
            {
87          private Node node;
88          private SyntaxTree tree;
89          private SimpleParseDecorator(Node node) {
90              this.node = node;
91          }
92
93          private static boolean parseSub(Children.Rule target, Node node, SyntaxTree
                tree) throws Exception {
94              boolean result = true;
95              for (Child child : target) {
96                  LexicalUnit lexicalUnit = node.env.getInput().get();
97
98                  result = false;
99                  if(child.cls != null) {
100                     Node body = NodeUtil.isMatch(child.cls, node.env, lexicalUnit);
101                     if (body != null) {
102                         node.env.getInput().unget(lexicalUnit);
103                         result = body.parse();
104                         if(body instanceof HasSyntaxTree) {
105                             tree.add(((HasSyntaxTree) body).getSyntaxTree());
106                         }
107
108                         if(!result) {
109                             break;
110                         }
111                     } else {
112                         tree.add(new TerminalSymbol(target, lexicalUnit));
```

```
113                     }
114                 } else {
115                     result = child.lexicalType.equals(lexicalUnit.type);
116                     tree.add(new TerminalSymbol(target, lexicalUnit));
117                 }
118 
119                 if(!result) {
120                     break;
121                 }
122             }
123             return result;
124         }
125 
126         private boolean recParse(SyntaxTree mainTree, Set<Children.Rule> rules)
                throws Exception {
127             for (Children.Rule rule : rules) {
128                 SyntaxTree tmp_tree2 = new SyntaxTree(rule);
129                 boolean result_rec = parseSub(rule, node, tmp_tree2);
130 
131                 if(result_rec) {
132                     SyntaxTree tmp = new SyntaxTree(mainTree.rule);
133                     tmp.add(mainTree);
134                     tmp.addAll(tmp_tree2);
135 
136                     recParse(tmp, rules);
137 
138                     this.tree = tmp;
139                     return true;
140                 } else {
141                     tmp_tree2.unget(node.env);
142                 }
143             }
144             return false;
145         }
146 
147         @Override
148         public boolean parse() throws Exception{
149             Define define = node.getClass().getAnnotation(Define.class);
150             Field field = node.getClass().getField(define.children());
151             Children<?> children = (Children<?>)field.get(null);
152 
153             // not recursions parse
154             for (Children.NotRecursiveRule notRecursiveRule : children.
                    not_recursions) {
155                 SyntaxTree tmp_tree1 = new SyntaxTree(notRecursiveRule);
156                 boolean result = parseSub(notRecursiveRule, node, tmp_tree1);
157 
158                 if(result) {
159                     // recursions parse
160                     if(recParse(tmp_tree1, children.recursions)) {
161                         return true;
162                     }
163 
164                     this.tree = tmp_tree1;
165                     return true;
166                 } else {
167                     tmp_tree1.unget(node.env);
168                 }
169             }
170 
171 //              if (lexicalUnit.getType() == LexicalType.END) {
172 //                  super.type = NodeType.END;
173 //                  return true;
174 //              }
175 
176             return false;
177         }
178 
179         @Override
```

```
180         public SyntaxTree getSyntaxTree() {
181             return this.tree;
182         }
183     }
184 
185     public static class SyntaxTree extends ArrayList<SyntaxTree> {
186         private Children.Rule rule;
187         SyntaxTree(Children.Rule rule) {
188             this.rule = rule;
189         }
190 
191         void unget(Environment env) throws Exception {
192             Collections.reverse(this);
193             for (SyntaxTree syntaxTree : this) {
194                 if(syntaxTree != null) {
195                     syntaxTree.unget(env);
196                 }
197             }
198         }
199 
200 //         public Value getValue() {
201 //             return rule.getValue();
202 //         }
203     }
204 
205     public static class TerminalSymbol extends SyntaxTree {
206         private LexicalUnit terminal;
207         public TerminalSymbol(Children.Rule rule, LexicalUnit terminal) {
208             super(rule);
209             this.terminal = terminal;
210         }
211 
212         public LexicalUnit getTerminal() {
213             return terminal;
214         }
215 
216         @Override
217         void unget(Environment env) throws Exception {
218             env.getInput().unget(terminal);
219         }
220 
221         @Override
222         public boolean add(SyntaxTree tree) {
223             throw new RuntimeException("can't␣add␣tree␣to␣terminal");
224         }
225 
226         @Override
227         public String toString() {
228             return terminal.toString();
229         }
230     }
231 
232     public static class Child {
233         private Class<? extends Node> cls;
234         private LexicalType lexicalType;
235 
236         public Child(Class<? extends Node> cls) {
237             this.cls = cls;
238         }
239 
240         public Child(LexicalType lexicalType) {
241             this.lexicalType = lexicalType;
242         }
243 
244         public boolean match(Class<? extends Node> cls) {
245             return this.cls != null && this.cls.equals(cls);
246         }
247 
248         public boolean match(LexicalType lexicalType) {
```

```
249             return this.lexicalType != null && this.lexicalType.equals(lexicalType
                    );
250         }
251 
252         @Override
253         public String toString() {
254             return lexicalType == null ? cls.toString() : lexicalType.toString();
255         }
256     }
257 
258     public static class Children<T extends Node> {
259         private final Set<NotRecursiveRule> not_recursions = new TreeSet<>((o1, o2)
                -> o1.length() < o2.length() ? 1 : -1);
260         private final Set<Rule> recursions = new TreeSet<>((o1, o2) -> o1.length()
                < o2.length() ? 1: -1);
261         private final Class<? extends Node> parent;
262 
263         @SuppressWarnings("unchecked")
264         public Children(T... dummy) {
265             parent = (Class<? extends Node>) dummy.getClass().getComponentType();
266         }
267 
268         public final Children<T> or(Child... children) {
269             if(children[0].match(parent)) {
270                 recursions.add(new Rule(children));
271             } else {
272                 not_recursions.add(new NotRecursiveRule(children));
273             }
274             return this;
275         }
276 
277         public final Children<T> or(Object... childrenWithConvert) {
278             return or(convert(childrenWithConvert));
279         }
280 
281         @SuppressWarnings("unchecked")
282         public static Child[] convert(Object... children){
283             List<Child> c = Stream.of(children).map(e -> {
284                 if(e instanceof LexicalType) {
285                     return new Child((LexicalType) e);
286                 }
287                 Class cls = (Class) e;
288                 if(cls.getGenericSuperclass() == Node.class) {
289                     return new Child((Class<? extends Node>) e);
290                 }
291                 return null;
292             }).filter(Objects::nonNull).collect(Collectors.toList());
293             Child[] result = new Child[c.size()];
294             return c.toArray(result);
295         }
296 
297         private static class Rule implements Iterator<Child>, Iterable<Child>{
298             List<Child> children = new ArrayList<>();
299             Iterator<Child> iterator;
300 
301             private Rule(Child... children) {
302                 Collections.addAll(this.children, children);
303             }
304 
305             @Override
306             public Iterator<Child> iterator() {
307                 this.iterator = children.iterator();
308                 if(this.iterator.hasNext()) {
309                     this.iterator.next();
310                 }
311                 return this.iterator;
312             }
313 
314             @Override
```

```
315             public boolean hasNext() {
316                 return iterator.hasNext();
317             }
318 
319             @Override
320             public Child next() {
321                 return iterator.next();
322             }
323 
324             public int length(){
325                 return children.size();
326             }
327 
328             @Override
329             public String toString() {
330                 return children.toString();
331             }
332         }
333 
334         private static class NotRecursiveRule extends Rule {
335             private NotRecursiveRule(Child... children) {
336                 super(children);
337             }
338 
339             @Override
340             public Iterator<Child> iterator() {
341                 this.iterator = children.iterator();
342                 return this.iterator;
343             }
344         }
345 
346         @Override
347         public String toString() {
348 
349             return "===␣recursions␣===\n" +
350                     recursions.stream().map(Object::toString).collect(Collectors.
                            joining("\n")) +
351                     "\n" +
352                     "===␣not␣recursions␣===\n" +
353                     not_recursions.stream().map(Object::toString).collect(
                            Collectors.joining("\n")) +
354                     "\n";
355         }
356     }
357 }
```

1.4 LexicalAnalyzerImpl

リスト 4　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,
```

```
148                 LexicalType.DIM,
149                 LexicalType.AS,
150                 LexicalType.END,
151                 LexicalType.WHILE,
152                 LexicalType.DO,
153                 LexicalType.UNTIL,
154                 LexicalType.LOOP,
155                 LexicalType.TO,
156                 LexicalType.WEND,
157                 LexicalType.EOF,
158         };
159 
160         PatternNameDef(Pattern pattern) {
161             super(pattern);
162         }
163 
164         @Override
165         public Return match(LexicalType type, String target) {
166             Return ret = super.match(type, target);
167             if(ret == null) return null;
168             for (LexicalType lexicalType : types) {
169                 if(lexicalType.name().equals(ret.value)) {
170                     ret.lexicalUnit.type = lexicalType;
171                     break;
172                 }
173             }
174             return ret;
175         }
176 
177         @Override
178         Value createValue(String value) {
179             return new ValueImpl(value, ValueType.STRING);
180         }
181     }
182 
183     private static class PatternDefWithValueType extends PatternDef {
184         private ValueType type;
185         private Cast cast;
186 
187         PatternDefWithValueType(Pattern pattern, ValueType type, Cast cast) {
188             super(pattern);
189             this.type = type;
190             this.cast = cast;
191         }
192 
193         @Override
194         Value createValue(String value) {
195             return new ValueImpl(cast.exec(value), type);
196         }
197 
198         @FunctionalInterface
199         interface Cast{
200             Object exec(String value);
201         }
202     }
203 }
```

1.5 Block

リスト 5　Block.java

```
1  package newlangdef;
2 
3  import newlang4.*;
4 
5  /**
6   * Created by ctare on 2017/12/15.
```

```
 */
@NodeUtil.SimpleParse
@NodeUtil.Define(type = NodeType.BLOCK)
public class Block extends Node {
    public final static NodeUtil.FirstSet firstSet = new NodeUtil.FirstSet(
        LexicalType.DO, LexicalType.WHILE);
    public final static NodeUtil.Children children = new NodeUtil.Children<StmtList
        >()
            .or(LexicalType.DO, LexicalType.WHILE, Cond.class, LexicalType.NL,
                StmtList.class, LexicalType.LOOP, LexicalType.NL)
            .or(LexicalType.DO, LexicalType.UNTIL, Cond.class, LexicalType.NL,
                StmtList.class, LexicalType.LOOP, LexicalType.NL);
}
```

1.6 Cond

リスト 6　Cond.java

```
package newlangdef;

import newlang4.LexicalType;
import newlang4.Node;
import newlang4.NodeType;
import newlang4.NodeUtil;

/**
 * Created by ctare on 2017/12/20.
 */
@NodeUtil.SimpleParse
@NodeUtil.Define(type = NodeType.COND)
public class Cond extends Node {
    public final static NodeUtil.FirstSet firstSet = new NodeUtil.FirstSet(Expr.
        firstSet);
    public final static NodeUtil.Children children = new NodeUtil.Children<CallFunc
        >()
            .or(Expr.class, LexicalType.EQ, Expr.class)
            .or(Expr.class, LexicalType.GT, Expr.class)
            .or(Expr.class, LexicalType.LT, Expr.class)
            .or(Expr.class, LexicalType.GE, Expr.class)
            .or(Expr.class, LexicalType.LE, Expr.class)
            .or(Expr.class, LexicalType.NE, Expr.class);
}
```

1.7 ExprList

リスト 7　ExprList.java

```
package newlangdef;

import newlang4.LexicalType;
import newlang4.Node;
import newlang4.NodeType;
import newlang4.NodeUtil;

/**
 * Created by ctare on 2017/12/19.
 */
@NodeUtil.SimpleParse
@NodeUtil.Define(type = NodeType.EXPR_LIST)
public class ExprList extends Node {
    public final static NodeUtil.FirstSet firstSet = new NodeUtil.FirstSet(Expr.
        firstSet);
    public final static NodeUtil.Children children = new NodeUtil.Children<ExprList
        >()
            .or(Expr.class)
```

```
17              .or(ExprList.class, LexicalType.COMMA, Expr.class);
18  }
```

1.8 Stmt

リスト 8　Stmt.java

```
1   package newlangdef;
2
3   import newlang4.LexicalType;
4   import newlang4.Node;
5   import newlang4.NodeType;
6   import newlang4.NodeUtil;
7
8   /**
9    * Created by ctare on 2017/12/07.
10   */
11
12  @NodeUtil.SimpleParse
13  @NodeUtil.Define(type = NodeType.STMT)
14  public class Stmt extends Node {
15      public static NodeUtil.FirstSet firstSet = new NodeUtil.FirstSet(LexicalType.
            END).merge(CallFunc.firstSet).merge(Subst.firstSet);
16
17      public static NodeUtil.Children children = new NodeUtil.Children<Stmt>()
18              .or(Subst.class)
19              .or(LexicalType.END)
20              .or(CallFunc.class);
21  }
```

1.9 Subst

リスト 9　Subst.java

```
1   package newlangdef;
2
3   import newlang4.*;
4
5   /**
6    * Created by ctare on 2017/12/19.
7    */
8   @NodeUtil.SimpleParse
9   @NodeUtil.Define(type = NodeType.STMT_LIST)
10  public class Subst extends Node {
11      public final static NodeUtil.FirstSet firstSet = new NodeUtil.FirstSet(
            LexicalType.NAME);
12      public final static NodeUtil.Children children = new NodeUtil.Children<Subst>()
13              .or(LexicalType.NAME, LexicalType.EQ, Expr.class);
14  }
```

1.10 CallFunc

リスト 10　CallFunc.java

```
1   package newlangdef;
2
3   import newlang4.*;
4
5   /**
6    * Created by ctare on 2017/12/19.
7    */
8   @NodeUtil.SimpleParse
9   @NodeUtil.Define(type = NodeType.FUNCTION_CALL)
```

```
10 public class CallFunc extends Node {
11     public final static NodeUtil.FirstSet firstSet = new NodeUtil.FirstSet(
           LexicalType.NAME);
12     public final static NodeUtil.Children children = new NodeUtil.Children<CallFunc
           >()
13             .or(LexicalType.NAME, LexicalType.LP, ExprList.class, LexicalType.RP);
14 }
```

1.11 Expr

リスト 11　Expr.java

```
1  package newlangdef;
2
3  import newlang4.LexicalType;
4  import newlang4.Node;
5  import newlang4.NodeType;
6  import newlang4.NodeUtil;
7
8  /**
9   * Created by ctare on 2017/12/19.
10  */
11 @NodeUtil.SimpleParse
12 @NodeUtil.Define(type = NodeType.EXPR)
13 public class Expr extends Node {
14     public final static NodeUtil.FirstSet firstSet = new NodeUtil.FirstSet(
15             LexicalType.SUB,
16             LexicalType.LP,
17             LexicalType.NAME,
18             LexicalType.INTVAL,
19             LexicalType.DOUBLEVAL,
20             LexicalType.LITERAL)
21             .merge(CallFunc.firstSet);
22     public final static NodeUtil.Children children = new NodeUtil.Children<Expr>()
23             .or(Expr.class, LexicalType.ADD, Expr.class)
24             .or(Expr.class, LexicalType.SUB, Expr.class)
25             .or(Expr.class, LexicalType.MUL, Expr.class)
26             .or(Expr.class, LexicalType.DIV, Expr.class)
27             .or(LexicalType.SUB, Expr.class)
28             .or(LexicalType.LP, Expr.class, LexicalType.RP)
29             .or(LexicalType.NAME)
30             .or(LexicalType.INTVAL)
31             .or(LexicalType.DOUBLEVAL)
32             .or(LexicalType.LITERAL)
33             .or(CallFunc.class);
34 }
```

1.12 Program

リスト 12　Program.java

```
1  package newlangdef;
2
3
4  import newlang4.*;
5
6  import java.lang.reflect.Field;
7
8  /**
9   * Created by ctare on 2017/12/05.
10  */
11 @NodeUtil.Define(type = NodeType.PROGRAM, firstSet = "f")
12 public class Program extends Node {
13     public final static NodeUtil.FirstSet f = new NodeUtil.FirstSet() {
14         @Override
```

```
15          public boolean contains(LexicalUnit lexicalUnit) {
16              return true;
17          }
18      };
19
20      @Override
21      public boolean parse() throws Exception {
22          Environment env = getEnv();
23          LexicalUnit lexicalUnit = env.getInput().get();
24          env.getInput().unget(lexicalUnit);
25
26          Node stmtList = NodeUtil.isMatch(StmtList.class, env, lexicalUnit);
27          boolean result = stmtList != null && stmtList.parse();
28          if(stmtList instanceof NodeUtil.HasSyntaxTree) {
29              System.out.println(((NodeUtil.HasSyntaxTree) stmtList).getSyntaxTree
                    ());
30          }
31          return result;
32      }
33
34      private Environment getEnv() throws NoSuchFieldException,
            IllegalAccessException {
35          Field f = Node.class.getDeclaredField("env");
36          f.setAccessible(true);
37          return (Environment) f.get(this);
38      }
39  }
```

1.13 StmtList

リスト 13　StmtList.java

```
1   package newlangdef;
2
3   import newlang4.LexicalType;
4   import newlang4.Node;
5   import newlang4.NodeType;
6   import newlang4.NodeUtil;
7
8   /**
9    * Created by ctare on 2017/12/05.
10   */
11  @NodeUtil.SimpleParse
12  @NodeUtil.Define(type = NodeType.STMT_LIST)
13  public class StmtList extends Node {
14      public final static NodeUtil.FirstSet firstSet = new NodeUtil.FirstSet(Stmt.
            firstSet).merge(Block.firstSet);
15      public final static NodeUtil.Children children = new NodeUtil.Children<StmtList
            >()
16              .or(Stmt.class)
17              .or(StmtList.class, LexicalType.NL, Stmt.class)
18              .or(StmtList.class, LexicalType.NL)
19              .or(Block.class)
20              .or(Block.class, LexicalType.NL);
21  }
```

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、構文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 を作成し、Program クラスを引数に isMatch を呼び出し、その戻り値が null になるか、Program クラスの parse が false を返すまで構文解析を行う。

2.2 Node.java

リスト **2** は、講義中に配布された Node クラスをほぼそのまま使用している。講義中に配布された Node クラスでは、Parse メソッドが定義されていたが、名前を parse に変更した。

2.3 NodeUtil.java

リスト **3** では、構文解析を行う処理を定義している。

2.3.1 isMatch メソッド

字句解析にて、先頭に来ている LexicalUnit が、次に見るべき構文定義のファースト集合に含まれているかを確認する。含まれていた場合、引数で渡されたクラスのインスタンスを作成し、戻り値とする。また、その際に、対象クラスに SimpleParse アノテーションが付与されていた場合、SimpleParseDecorator のインスタンスを戻り値とする。

2.3.2 FirstSet クラス

ファースト集合を表すクラスである。Set を拡張する形で実装している。

merge メソッドでは、引数のファースト集合の中身を全て自分に追加する。

2.3.3 Define アノテーション

構文定義を行なっているクラスに付与する。NodeType と、ファースト集合、構文定義の指定を行う。

2.3.4 SimpleParse アノテーション

SimpleParseDecorator クラスで拡張したいクラスに付与する。

2.3.5 HasSyntaxTree インターフェイス

構文解析中に構文木を保存している場合、このインターフェイスを実装して構文木を取り出せるようにする。

2.3.6 SimpleParseDecorator クラス

Node の子クラスを拡張するクラスである。parse メソッドを拡張し、構文解析を行う。また、構文木の保存も行う。

parse メソッドでは、構文定義と字句解析の結果を参照し、構文が一致しているかを確認している。構文定義の参照順序は、先頭の定義が自身の定義を指していないものから参照している。構文定義と字句解析の結果が一致した場合、recParse メソッドを呼び出し、戻り値を true とする。一致しなかった場合、読み進めた字句解析の結果を、このメソッドが呼ばれた際の状態に戻し、戻り値を false とする。

parseSub メソッドでは、構文定義と字句解析の結果の一致確認を行う処理を定義している。引数には、構文定義を表す target、対象ノードを表す node、構文木の保存先である tree を定義している。target から一つずつ定義を読み取り、字句解析にて現在読んでいる箇所と一致しているか確認する。一致していた場合は、字句解析を読み進め、構文木に保存する。target から取り出した定義が、終端記号以外を指していた場合、その定義に対応しているクラスを引数に isMatch にてファースト集合との一致を確認し、その戻り値の parse を呼び出す。また、その際に一つ字句を読み戻す。

recParse メソッドでは、構文定義の先頭が自身の定義を指している場合の、構文定義と字句解析の結果の一致確認を行う処理を定義している。引数には、構文木の保存先である mainTree と、構文定義の集合である rules を定義している。rules から定義を一つずつ取り出し、parseSub にて構文定義が字句解析の結果と一致するかを確認する。一致した場合、recParse を呼び、一致しなくなるまで処理を続ける。

getSyntaxTree メソッドでは、保存されている構文木を戻り値とする。

2.3.7 SyntaxTree クラス

構文木を表すクラスである。ArrayList を拡張する形で実装している。

unget メソッドでは、保存されている構文木を削除し、字句解析で読み進めた箇所を戻す処理を定義する。自クラスに保存されている構文木の unget を再帰的に呼び出す実装とした。

2.3.8 TerminalSymbol クラス

構文木に挿入する終端記号を表すクラスである。add メソッドをオーバーライドし、add が呼ばれた際に例外を投げ、このクラスに構文木が追加されてないように定義している。また、unget メソッドでは、LexicalAnalyzerImpl にて定義されている unget を呼び出し、字句の読み戻しを行なっている。

2.3.9 Child クラス

構文定義の定義要素を表すクラスである。LexicalType か、Node の子クラスを保存する。

2.3.10 Children クラス

構文定義の集合を表すクラスである。構文を定義する際に、定義の先頭が自身の定義を指しているものと、そうでないものに分けて保存する。

Rule クラスは、定義の先頭が自身の定義を指している場合の構文定義を表している。また、構文定義が長い順に並び替える。

NotRecursiveRule クラスは、Rule クラスを拡張しており、定義の先頭が自身の定義を指していない場合の構文定義を表している。

2.3.11 LexicalAnalyzerImpl

リスト 4 では、字句解析を行うクラスを定義している。前レポートにて定義した LexicalAnalyzerImpl クラスに、unget の機能を追加したものである。unget にて読み戻された字句をスタックに保存し、字句の読み出しの際にスタックの中身が存在した場合それを戻り値とする。

2.3.12 Block

リスト 5 では、block の構文を定義している。講義ページにて定義されている構文から、今回の入力で用いる箇所のみを定義した。また、ファースト集合を予測して定義している。

2.3.13 Cond

リスト 6 では、cond の構文を定義している。講義ページにて定義されている構文から、今回の入力で用いる箇所のみを定義した。また、ファースト集合を予測して定義している。

2.3.14 ExprList

リスト 7 では、expr_list の構文を定義している。講義ページにて定義されている構文から、今回の入力で用いる箇所のみを定義した。また、ファースト集合を予測して定義している。

2.3.15 Stmt

リスト 8 では、stmt の構文を定義している。講義ページにて定義されている構文から、今回の入力で用いる箇所のみを定義した。また、ファースト集合を予測して定義している。

2.3.16 Subst

リスト 9 では、subst の構文を定義している。講義ページにて定義されている構文から、今回の入力で用いる箇所のみを定義した。また、ファースト集合を予測して定義している。

2.3.17 CallFunc

リスト 10 では、call_func の構文を定義している。講義ページにて定義されている構文から、今回の入力で用いる箇所のみを定義した。また、ファースト集合を予測して定義している。

2.3.18 Expr

リスト 11 では、expr の構文を定義している。講義ページにて定義されている構文から、今回の入力で用いる箇所のみを定義した。また、ファースト集合を予測して定義している。

2.3.19 Program

リスト 12 では、program の構文を定義している。ファースト集合との一致確認にて、全てと一致するようにファースト集合を定義している。

parse メソッドでは、StmtList を引数に isMatch を呼び出し、戻り値があった場合、その parse を呼び出す。StmtList のインスタンスが構文木を保存していた場合、構文木を取得して表示する。

2.3.20 StmtList

リスト 13 では、stmt_list の構文を定義している。講義ページにて定義されている構文から、今回の入力で用いる箇所のみを定義した。StmtList にて定義されている構文の後には改行の有無を無視したかったため、¡stmt list¿ NL の構文を追加した。

また、ファースト集合を予測して定義している。

3 結果

3.1 入力

リスト 14 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 15 出力

```
1 basic parser
2 [[[[NAME: a, EQ, [INTVAL: 5]]]], NL]
3 [[DO, UNTIL, [[NAME:  a], LT, [INTVAL:  1]], NL, [[[[NAME:  PRINT, LP, [[LITERAL:
      Hello]], RP]]], NL, [[NAME: a, EQ, [[NAME:  a], SUB, [INTVAL: 1]]]]], LOOP, NL
      ]]
4 [[[END]], NL]
```