

```
1 package dragon;
2
3 import org.antlr.v4.runtime.CharStream;
4 import org.antlr.v4.runtime.CharStreams;
5 import org.antlr.v4.runtime.Token;
6 import org.testng.annotations.AfterMethod;
7 import org.testng.annotations.BeforeMethod;
8 import org.testng.annotations.Test;
9
10 import java.io.FileInputStream;
11 import java.io.IOException;
12 import java.io.InputStream;
13 import java.nio.file.Path;
14
15 public class DragonLexerGrammarTest {
16     InputStream is = System.in;
17
18     @BeforeMethod
19     public void setUp() throws IOException {
20         is = new FileInputStream(Path.of("src/test/java/dragon/dragon0.txt")
21             .toFile());
22     }
23
24     @AfterMethod
25     public void tearDown() {
26     }
27
28     @Test
29     public void testGetAllTokens() throws IOException {
30         CharStream input = CharStreams.fromStream(is);
31         DragonLexerGrammar lexer = new DragonLexerGrammar(input);
32
33         for (Token token : lexer.getAllTokens()) {
34             System.out.println(token);
35         }
36     }
37 }
```

```
1 package dragon;
2
3 import org.testng.annotations.AfterMethod;
4 import org.testng.annotations.BeforeMethod;
5 import org.testng.annotations.Test;
6
7 import java.io.IOException;
8 import java.nio.file.Files;
9 import java.nio.file.Path;
10
11 public class DragonLexerTest {
12     private String input;
13     private DragonLexer lexer;
14
15     @BeforeMethod
16     public void setUp() throws IOException {
17         input = Files.readString(Path.of("src/test/java/dragon/dragon0.txt
18     ));
19         lexer = new DragonLexer(input);
20     }
21
22     @AfterMethod
23     public void tearDown() {
24     }
25
26     @Test
27     public void testNextToken() {
28         Token token = lexer.nextToken();
29
30         while (token != Token.EOF && token != Token.WS) {
31             System.out.println(token);
32             token = lexer.nextToken();
33         }
34 }
```

```
1 package dragon;
2
3 public class DragonLexer extends Lexer {
4     private final KeywordTable kwTable = new KeywordTable();
5
6     public DragonLexer(String input) {
7         super(input);
8     }
9
10    @Override
11    public Token nextToken() {
12        if (peek == EOF) {
13            return Token.EOF;
14        }
15
16        if (Character.isWhitespace(peek)) {
17            WS();
18        }
19
20        if (Character.isLetter(peek)) {
21            return ID();
22        }
23
24        if (Character.isDigit(peek)) {
25            return NUMBER();
26        }
27
28        if (peek == '=') {
29            consume();
30            return Token.EQ;
31        }
32
33        if (peek == '<') {
34            consume();
35            if (peek == '=') {
36                consume();
37                return Token.LE;
38            }
39
40            if (peek == '>') {
41                consume();
42                return Token.NE;
43            }
44
45            return Token.LT;
46        }
47
48        if (peek == '>') {
49            consume();
50
51            if (peek == '=') {
52                consume();
53                return Token.GE;
```

```
54     }
55
56     return Token.GT;
57 }
58
59 Token unknown = new Token(TokenType.UNKNOWN, Character.toString(
peek));
60 consume();
61 return unknown;
62 }
63
64 private Token WS() {
65     while (Character.isWhitespace(this.peek)) {
66         consume();
67     }
68
69     return Token.WS;
70 }
71
72 private Token ID() {
73     StringBuilder sb = new StringBuilder();
74
75     do {
76         sb.append(peek);
77         consume();
78     } while (Character.isLetterOrDigit(peek));
79
80     Token token = this.kwTable.getKeyword(sb.toString());
81     if (token == null) {
82         return new Token(TokenType.ID, sb.toString());
83     }
84
85     return token;
86 }
87
88 private Token INT() {
89     StringBuilder sb = new StringBuilder();
90
91     do {
92         sb.append(peek);
93         consume();
94     } while (Character.isDigit(peek));
95
96     return new Token(TokenType.INT, sb.toString());
97 }
98
99 private Token NUMBER() {
100     StringBuilder intStr = new StringBuilder();
101     intStr.append(peek);
102     consume();
103
104     int intPos = -1;
105     int realPos = -1;
```

```
106
107     StringBuilder realStr = new StringBuilder();
108     StringBuilder sciStr = new StringBuilder();
109
110     int state = 13;
111     while (true) {
112         switch (state) {
113             case 13:
114                 intPos = pos;
115                 if (Character.isDigit(peek)) {
116                     intStr.append(peek);
117                     consume();
118                     state = 13;
119                     break;
120                 } else if (peek == '.') {
121                     realStr.append(peek);
122                     consume();
123                     state = 14;
124                 } else if (peek == 'E' || peek == 'e') {
125                     sciStr.append(peek);
126                     consume();
127                     state = 16;
128                     break;
129                 } else {
130                     return new Token(TokenType.INT, intStr.toString());
131                 }
132             case 14:
133                 if (Character.isDigit(peek)) {
134                     realStr.append(peek);
135                     consume();
136                     state = 15;
137                     break;
138                 } else {
139                     this.reset(intPos);
140                     return new Token(TokenType.INT, intStr.toString());
141                 }
142             case 15:
143                 realPos = pos;
144                 if (Character.isDigit(peek)) {
145                     realStr.append(peek);
146                     consume();
147                     state = 15;
148                     break;
149                 } else if (peek == 'E') {
150                     sciStr.append(peek);
151                     consume();
152                     state = 16;
153                     break;
154                 } else {
155                     return new Token(TokenType.REAL, intStr.append(realStr).
toString());
156                 }
157             case 16:
```

```
158         if (peek == '+' || peek == '-') {
159             sciStr.append(peek);
160             consume();
161             state = 17;
162             break;
163         } else if (Character.isDigit(peek)) {
164             sciStr.append(peek);
165             consume();
166             state = 18;
167             break;
168         } else {
169             this.reset(realPos);
170             return new Token(TokenType.REAL, intStr.append(realStr).
toString());
171         }
172         case 17:
173             if (Character.isDigit(peek)) {
174                 sciStr.append(peek);
175                 consume();
176                 state = 18;
177                 break;
178             } else {
179                 this.reset(realPos);
180                 return new Token(TokenType.REAL, intStr.append(realStr).
toString());
181             }
182         case 18:
183             if (Character.isDigit(peek)) {
184                 sciStr.append(peek);
185                 consume();
186                 state = 18;
187                 break;
188             } else {
189                 return new Token(TokenType.SCI, intStr.append(realStr).
append(sciStr).toString());
190             }
191         default:
192             System.err.println("Unreachable");
193     }
194 }
195 }
196 }
197
```

```
1 package dragon;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class KeywordTable {
7     private final Map<String, Token> keywords = new HashMap<>();
8
9     public KeywordTable() {
10         this.reserve(Token.IF);
11         this.reserve(Token.ELSE);
12     }
13
14     public Token getKeyword(String text) {
15         return this.keywords.get(text);
16     }
17
18     private void reserve(Token token) {
19         keywords.put(token.getText(), token);
20     }
21 }
```

```
1 package dragon;
2
3 public abstract class Lexer {
4     public static final char EOF = (char) -1;
5
6     private final String input;
7     char peek;
8     int pos;
9
10    public Lexer(String input) {
11        this.input = input;
12        this.pos = 0;
13        this.peek = input.charAt(pos);
14    }
15
16    public abstract Token nextToken();
17
18    public void reset(int pos) {
19        this.pos = pos;
20        this.peek = input.charAt(pos);
21    }
22
23    public void consume() {
24        this.pos++;
25        if (this.pos >= this.input.length()) {
26            this.peek = EOF;
27        } else {
28            this.peek = input.charAt(this.pos);
29        }
30    }
31 }
```



```
1 package dragon;
2
3 public class Token {
4     public static final Token EOF = new Token(TokenType.EOF, "EOF");
5     public static final Token WS = new Token(TokenType.WS, " ");
6
7     public static final Token IF = new Token(TokenType.IF, "if");
8     public static final Token ELSE = new Token(TokenType.ELSE, "else");
9
10    public static final Token EQ = new Token(TokenType.EQ, "=");
11    public static final Token NE = new Token(TokenType.NE, "<>");
12    public static final Token LT = new Token(TokenType.LT, "<");
13    public static final Token LE = new Token(TokenType.LE, "<=");
14    public static final Token GT = new Token(TokenType.GT, ">");
15    public static final Token GE = new Token(TokenType.GE, ">=");
16
17    private final TokenType type;
18    private final String text;
19
20    public Token(TokenType type, String text) {
21        this.type = type;
22        this.text = text;
23    }
24
25    public TokenType getType() {
26        return type;
27    }
28
29    public String getText() {
30        return this.text;
31    }
32
33    @Override
34    public String toString() {
35        return String.format("token {type : %s, text : %s}",
36            this.type, this.text);
37    }
38 }
```

```
1 package dragon;
2
3 /**
4  * Types of tokens
5  */
6 public enum TokenType {
7     // Group 0
8     EOF,
9     WS,
10    UNKNOWN,
11
12    // Group 1
13    IF, ELSE,
14    ID,
15    INT,
16
17    // Group 2
18    // =, <>, <, <=, >, >=
19    EQ, NE, LT, LE, GT, GE,
20
21    // Group 3
22    REAL,
23    SCI,
24 }
```