REPORT

과 제 명: Lexical Analyzer 구현

과목명	컴파일러 02분반
교수명	김진성
학번	20206319
학과	소프트웨어학부 소프트웨어전공
이름	김가연

[소스코드 설명]

```java
package Running;
public enum Token {
    INTEGER, BOOLEAN_STRING, LITERAL_STRING, IDENTIFIER, KEYWORLD, TYPE, KEYWORD,
    ARITHMETIC_OPERATOR, ASSIGNMENT_OPERATOR, COMPARISON_OPERATOR, SEMI, LBRACE, RBRACE, LPAREN, RPAREN,
    LBRACK, RBRACK, COMMA, WHITESPACE, INVALID
}
```

<2-5줄> 편리를 위하여 Token.java에 토큰의 이름들만 저장하였습니다.

```java

public class Lexical_Analyzer {
    private Map<String, Token> keywordsAndOperatorsMap;

    public Lexical_Analyzer() {
        this.keywordsAndOperatorsMap = new HashMap<>();
        keywordsAndOperatorsMap.put("true", Token.BOOLEAN_STRING);
        keywordsAndOperatorsMap.put("false", Token.BOOLEAN_STRING);

        keywordsAndOperatorsMap.put("if", Token.KEYWORD);
        keywordsAndOperatorsMap.put("else", Token.KEYWORD);
        keywordsAndOperatorsMap.put("while", Token.KEYWORD);
        keywordsAndOperatorsMap.put("class", Token.KEYWORD);
        keywordsAndOperatorsMap.put("return", Token.KEYWORD);
        keywordsAndOperatorsMap.put("public", Token.KEYWORD);
        keywordsAndOperatorsMap.put("static", Token.KEYWORD);
        keywordsAndOperatorsMap.put("void", Token.KEYWORD);
        keywordsAndOperatorsMap.put("main", Token.KEYWORD);
        keywordsAndOperatorsMap.put("args", Token.KEYWORD);
        keywordsAndOperatorsMap.put("package", Token.KEYWORD);

        keywordsAndOperatorsMap.put("int", Token.TYPE);
        keywordsAndOperatorsMap.put("char", Token.TYPE);
        keywordsAndOperatorsMap.put("boolean", Token.TYPE);
        keywordsAndOperatorsMap.put("String", Token.TYPE);

        keywordsAndOperatorsMap.put("+", Token.ARITHMETIC_OPERATOR);
        keywordsAndOperatorsMap.put("-", Token.ARITHMETIC_OPERATOR);
        keywordsAndOperatorsMap.put("*", Token.ARITHMETIC_OPERATOR);
        keywordsAndOperatorsMap.put("/", Token.ARITHMETIC_OPERATOR);
        keywordsAndOperatorsMap.put("=", Token.ASSIGNMENT_OPERATOR);
        keywordsAndOperatorsMap.put(">", Token.COMPARISON_OPERATOR);
        keywordsAndOperatorsMap.put("<", Token.COMPARISON_OPERATOR);
        keywordsAndOperatorsMap.put("==", Token.COMPARISON_OPERATOR);
        keywordsAndOperatorsMap.put("!=", Token.COMPARISON_OPERATOR);
        keywordsAndOperatorsMap.put("<=", Token.COMPARISON_OPERATOR);
        keywordsAndOperatorsMap.put(">=", Token.COMPARISON_OPERATOR);
```

```java
        keywordsAndOperatorsMap.put(";", Token.SEMI);

        keywordsAndOperatorsMap.put("{", Token.LBRACE);
        keywordsAndOperatorsMap.put("}", Token.RBRACE);

        keywordsAndOperatorsMap.put("(", Token.LPAREN);
        keywordsAndOperatorsMap.put(")", Token.RPAREN);

        keywordsAndOperatorsMap.put("[", Token.LBRACK);
        keywordsAndOperatorsMap.put("]", Token.RBRACK);

        keywordsAndOperatorsMap.put(",", Token.COMMA);
    }
```

<13-60줄> 정의한 토큰들에 대하여 Token이름을 부여하고 이를 위하여 Map을 이용하였습니다. 제시된 Lexical Specification 이외에 JAVA로 구현할 때 필요한 단어들은 부득이하게 KEYWORD로 취급하게 되었습니다.

```java
    public Map<Integer, Map<String, Token>> analyzeCode(Path filePath) {
            Map<Integer, Map<String, Token>> mapTokensLine = new LinkedHashMap<>();
            try {
                List<String> lines = Files.readAllLines(filePath, Charset.forName("UTF-8"));
                for (int i = 0; i < lines.size(); i++) {
                    Map<String, Token> lineTokens = analyzeLine(lines.get(i).strip());
                    mapTokensLine.put(i + 1, lineTokens);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            return mapTokensLine;
        }
        private Map<String, Token> analyzeLine(String line) {
            Map<String, Token> lineTokens = new HashMap<>();
            Automata automaton = new Automata();
            for (String str : line.split(" ")) {
                if (keywordsAndOperatorsMap.containsKey(str.toLowerCase()))
                    lineTokens.put(str, keywordsAndOperatorsMap.get(str.toLowerCase()));
                else
                    lineTokens.put(str, automaton.evaluate(str));
            }
            return lineTokens;
        }
```

String java.lang.String.toLo

Converts all of the characters in
is equivalent to calling toLowe

<62-84줄> input파일에서 줄별로 읽어내기 위한 코드를 작성하였습니다.

```java
public enum State {
    INITIAL,Q1,Q2,Q3,Q4,INVALIDATION_STATE
}
```

<3-4줄> Regular Expression으로 표현한 Token중 문법이 필요했던 표현들에 대하여 문법을 통해 분류를 하는데에 있어 편리함을 위하여 상태에 대해 미리 표현을 해두었습니다.

```java
public class Automata {
    private Map<State, Token> finalStates;
    public Automata() {
        finalStates = new HashMap<>();
        finalStates.put(State.Q1, Token.INTEGER);
        finalStates.put(State.Q2, Token.LITERAL_STRING);
        finalStates.put(State.Q3, Token.IDENTIFIER);

    }
    private State executeTransition(State currentState, char entry) {
        switch (currentState) {
            case INITIAL: {
                if ((entry >= '0' && entry <= '9') || entry == '-')
                    return State.Q1;
                else if (entry == '"')
                    return State.Q2;
                else if ((entry >= 'A' && entry <= 'Z') || (entry >= 'a' && entry <= 'z') || entry == '_')
                    return State.Q3;
                else
                    return State.INVALIDATION_STATE;
            }
            case Q1: {
                if (entry >= '0' && entry <= '9')
                    return State.Q1;
                else
                    return State.INVALIDATION_STATE;
            }
            case Q2: {
                return (entry == '\0')
                        ||(entry >= 'A' && entry <= 'Z')
                        || (entry >= 'a' && entry <= 'z')
                        ? State.Q2 : State.INVALIDATION_STATE;
            }
            case Q3: {
                return (entry >= 'A' && entry <= 'Z')
                        || (entry >= 'a' && entry <= 'z')
                        || (entry == '_')
                        || (entry >= '0' && entry <= '9')
                        ? State.Q3 : State.INVALIDATION_STATE;
            }
            case Q4:

            default:
                return State.INVALIDATION_STATE;
        }
    }
    public Token evaluate(String str){
        State state = State.INITIAL;
        for(char c : str.toCharArray()){
            state = executeTransition(state, c);
        }
        return finalStates.getOrDefault(state, Token.INVALID);
    }
}

```

<8-49줄> Regular Expression을 코드로 작성하였습니다. 문법으로 표현해야하는 INTEGER, LITERAL_STRING, IDENTIFIER에 대하여 작성하였습니다.
<52-57줄>은 패턴이 있는 Token으로 읽혀지는 것들은 Token 이름을 상태로 저장하고, 아무 소속이 없는 글자는 INVALID로 표현하기 위한 코드를 작성하였습니다.

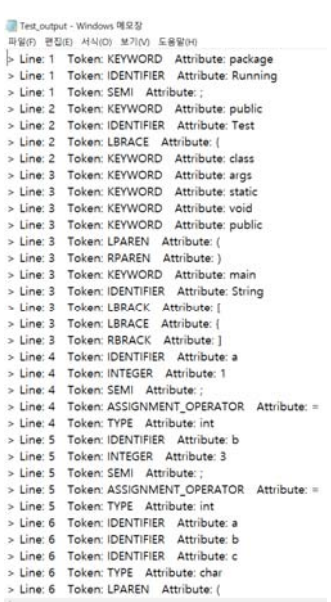
```java
public class Main {
    public static void main(String[] args){
        File file = new File("C:\\Users\\USER\\eclipse-workspace\\Com_0511\\src\\Running\\Test.java");
        evaluate(file);
    }
    static String result = "";
    private static void evaluate(File file) {
        Lexical_Analyzer Lexana = new Lexical_Analyzer();
        Map<Integer, Map<String, Token>> tokensTable = Lexana.analyzeCode(file.toPath());

        try{
            File file2 = new File("C:\\Users\\USER\\eclipse-workspace\\Com_0511\\Test_output.txt");
            FileWriter fileWriter = new FileWriter(file2);
            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
            tokensTable.forEach((k, v) ->
            v.forEach((str, token) ->
                {
                        result= "> Line: " + k + "   Token: " + token + "   Attribute: " + str + "\n";
                        try {
                            bufferedWriter.write(result);
                        } catch (IOException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        }
                        return;
                }
                )
            );
            bufferedWriter.close();
        }catch(IOException e) {
            e.printStackTrace();
        }
    }

}

```

<15-16줄> input파일을 읽기위한 코드입니다.
<20-21줄> Lexical_Analyzer.java에서 미리 작성한 인터페이스를 가져온 것입니다.
<19-43줄> 읽은 input파일을 읽어서 정의된 Token으로 분류하고 이를 output파일에 저장하는 코드입니다. output파일에 저장하기 위한 코드들로 try-catch문을 이용하였습니다.

[결과]
원활한 인식을 위하여 input파일의 코드는 전부 띄어쓰기를 하였습니다.
아래는 테스트하기위한 input파일과 output파일입니다.

```
1  package Running ;
2  public class Test {
3      public static void main ( String [ ] args ) {
4          int a = 1 ;
5          int b = 3 ;
6          char c = ( char ) ( ( char ) a + b ) ;
7          int d = - 15 ;
8          boolean e = false ;
9          String f ;
10         if ( a >= 1 ) {
11             f = " HELLO " ;
12         }
13         else
14             f = " OOPS " ;
15     }
16 }
17
```

<input 코드>



```
Test_output - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
> Line: 1  Token: KEYWORD  Attribute: package
> Line: 1  Token: IDENTIFIER  Attribute: Running
> Line: 1  Token: SEMI  Attribute: ;
> Line: 2  Token: KEYWORD  Attribute: public
> Line: 2  Token: IDENTIFIER  Attribute: Test
> Line: 2  Token: LBRACE  Attribute: {
> Line: 2  Token: KEYWORD  Attribute: class
> Line: 3  Token: KEYWORD  Attribute: args
> Line: 3  Token: KEYWORD  Attribute: static
> Line: 3  Token: KEYWORD  Attribute: void
> Line: 3  Token: KEYWORD  Attribute: public
> Line: 3  Token: LPAREN  Attribute: (
> Line: 3  Token: RPAREN  Attribute: )
> Line: 3  Token: KEYWORD  Attribute: main
> Line: 3  Token: IDENTIFIER  Attribute: String
> Line: 3  Token: LBRACK  Attribute: [
> Line: 3  Token: LBRACE  Attribute: {
> Line: 3  Token: RBRACK  Attribute: ]
> Line: 4  Token: IDENTIFIER  Attribute: a
> Line: 4  Token: INTEGER  Attribute: 1
> Line: 4  Token: SEMI  Attribute: ;
> Line: 4  Token: ASSIGNMENT_OPERATOR  Attribute: =
> Line: 4  Token: TYPE  Attribute: int
> Line: 5  Token: IDENTIFIER  Attribute: b
> Line: 5  Token: INTEGER  Attribute: 3
> Line: 5  Token: SEMI  Attribute: ;
> Line: 5  Token: ASSIGNMENT_OPERATOR  Attribute: =
> Line: 5  Token: TYPE  Attribute: int
> Line: 6  Token: IDENTIFIER  Attribute: a
> Line: 6  Token: IDENTIFIER  Attribute: b
> Line: 6  Token: IDENTIFIER  Attribute: c
> Line: 6  Token: TYPE  Attribute: char
> Line: 6  Token: LPAREN  Attribute: (
```

```
Test_output - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
> Line: 6  Token: RPAREN  Attribute: )
> Line: 6  Token: ARITHMETIC_OPERATOR  Attribute: +
> Line: 6  Token: SEMI  Attribute: ;
> Line: 6  Token: ASSIGNMENT_OPERATOR  Attribute: =
> Line: 7  Token: IDENTIFIER  Attribute: d
> Line: 7  Token: INTEGER  Attribute: 15
> Line: 7  Token: SEMI  Attribute: ;
> Line: 7  Token: ASSIGNMENT_OPERATOR  Attribute: =
> Line: 7  Token: ARITHMETIC_OPERATOR  Attribute: -
> Line: 7  Token: TYPE  Attribute: int
> Line: 8  Token: TYPE  Attribute: boolean
> Line: 8  Token: IDENTIFIER  Attribute: e
> Line: 8  Token: BOOLEAN_STRING  Attribute: false
> Line: 8  Token: SEMI  Attribute: ;
> Line: 8  Token: ASSIGNMENT_OPERATOR  Attribute: =
> Line: 9  Token: IDENTIFIER  Attribute: f
> Line: 9  Token: IDENTIFIER  Attribute: String
> Line: 9  Token: SEMI  Attribute: ;
> Line: 10  Token: IDENTIFIER  Attribute: a
> Line: 10  Token: INTEGER  Attribute: 1
> Line: 10  Token: LPAREN  Attribute: (
> Line: 10  Token: RPAREN  Attribute: )
> Line: 10  Token: LBRACE  Attribute: {
> Line: 10  Token: KEYWORD  Attribute: if
> Line: 10  Token: COMPARISON_OPERATOR  Attribute: >=
> Line: 11  Token: LITERAL_STRING  Attribute: "
> Line: 11  Token: IDENTIFIER  Attribute: HELLO
> Line: 11  Token: IDENTIFIER  Attribute: f
> Line: 11  Token: SEMI  Attribute: ;
> Line: 11  Token: ASSIGNMENT_OPERATOR  Attribute: =
> Line: 12  Token: RBRACE  Attribute: }
> Line: 13  Token: KEYWORD  Attribute: else
> Line: 14  Token: LITERAL_STRING  Attribute: "
```

<output 파일>

여기서 에러가 발생합니다. Test.java에서 7줄을 보면, -15를 의도하였지만 output파일을 보면, -를 Arithmetic_Operator로 인식하였습니다. 이를 해결하기 위해서는 음수를 표현할 때, -와 숫자를 붙여서 인식시켜야 합니다.

```
> Line: 14  Token: IDENTIFIER  Attribute: f
> Line: 14  Token: IDENTIFIER  Attribute: OOPS
> Line: 14  Token: SEMI  Attribute: ;
> Line: 14  Token: ASSIGNMENT_OPERATOR  Attribute: =
> Line: 15  Token: RBRACE  Attribute: }
> Line: 16  Token: RBRACE  Attribute: }
```

```
1  package Running ;
2  public class Test {
3      public static void main ( String [ ] args ) {
4          int a = 1 ;
5          int b = 3 ;
6          char c = ( char ) ( ( char ) a + b ) ;
7          int d = -15 ;
8          boolean e = false ;
9          String f ;
10         if ( a >= 1 ) {
11             f = " HELLO " ;
12         }
13         else
14             f = " OOPS " ;
15     }
16 }
17
```

이렇게 수정 후 output파일을 보면,

```
> Line: 6   Token: ASSIGNMENT_OPERATOR   Attribute: =
> Line: 7   Token: INTEGER   Attribute: -15
> Line: 7   Token: IDENTIFIER   Attribute: d
> Line: 7   Token: SEMI   Attribute: ;
> Line: 7   Token: ASSIGNMENT_OPERATOR   Attribute: =
```

정상적으로 출력됩니다. 문자에 한해서도 확인해보겠습니다.

```
2  public class Test {
3      public static void main ( String [ ] args ) {
4          int a = 1 ;
5          int b = 3 ;
6          char c = ( char ) ( ( char ) a + b ) ;
7          int d = -15 ;
8          int sum = b -a ;
9          boolean e = false ;
10         String f ;
11         if ( a >= 1 ) {
12             f = " HELLO " ;
13         }
14         else
15             f = " OOPS " ;
16     }
17 }
18
```

```
> Line: 8   Token: IDENTIFIER   Attribute: b
> Line: 8   Token: INVALID   Attribute: -a
> Line: 8   Token: IDENTIFIER   Attribute: sum
```

왼쪽의 input파일로 출력하였을 때, 결과는 오른쪽과 같습니다.
띄어쓰기와 문자를 붙이게 되면, 둘을 합쳐서 ARITHMETIC_OPERATOR로 인식합니다.
인식의 문제로 전부 띄어쓰기를 부득이하게 하였으나 '-'에 대해 이러한 오류가 생겨났습니다.
이를 해결하기 위해 Automata.java코드를 아래와 같이 수정하였습니다.

```
6  public class Automata {
7      private Map<State, Token> finalStates;
8      public Automata() {
9          finalStates = new HashMap<>();
10         finalStates.put(State.Q1, Token.INTEGER);
11         finalStates.put(State.Q2, Token.LITERAL_STRING);
12         finalStates.put(State.Q3, Token.IDENTIFIER);
13         finalStates.put(State.Q6, Token.ARITHMETIC_OPERATOR);
14
15     }
16     private State executeTransition(State currentState, char entry) {
17         switch (currentState) {
18             case INITIAL: {
19                 if ((entry >= '0' && entry <= '9'))
20                     return State.Q1;
21                 else if (entry == '"')
22                     return State.Q2;
23                 else if ((entry >= 'A' && entry <= 'Z') || (entry >= 'a' && entry <= 'z') || entry == '_')
24                     return State.Q3;
25                 else if(entry == '-')
26                     return State.Q4;
27                 else
28                     return State.INVALIDATION_STATE;
29             }
30             case Q1: {
31                 if (entry >= '0' && entry <= '9')
32                     return State.Q1;
33                 else
34                     return State.INVALIDATION_STATE;
35             }
53             case Q5: {
54                 if (entry >= '0' && entry <= '9')
55                     return State.Q1;
56                 else if((entry >= 'A' && entry <= 'Z') || (entry >= 'a' && entry <= 'z'))
57                     return State.Q6;
58             }
59             case Q6:
60             default:
61                 return State.INVALIDATION_STATE;
62         }
```

```
> Line: 8   Token: ARITHMETIC_OPERATOR   Attribute: -a
> Line: 8   Token: IDENTIFIER   Attribute: sum
```

결과는 오른쪽 사진과 같이 문자와 '-'를 하나로 인식하고 -를 ARITHMETIC_OPERATOR로 인식하였습니다.