

Internal Document

src/main/java/CharClass.java

```
public enum CharClass {  
    LETTER, DIGIT, UNKNOWN, EOF  
}
```

- Lexeme이 EOF인지 문자인지, 정수인지, Unknown인지 확인하기 위한 enum이다.

src/main/java/Token.java

```
public enum Token {  
    INT_LIT, IDENT, ASSIGN_OP, ADD_OP, MULT_OP, LEFT_PAREN, RIGHT_PAREN, EOF, SEMICOLON, ERROR  
}
```

- Lexeme의 Token Type을 정하기 위해서 확인하기 위한 enum이다.

src/main/java/Pair.java

```
public class Pair {  
    private final String first;  
    private final String second;  
  
    Pair(String first, String second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    public String getFirst() {  
        return first;  
    }  
  
    public String getSecond() {  
        return second;  
    }  
}
```

- Error를 List에 넣을때, Error인지 Warning인지, 그 정확한 내용은 무엇인지를 넣기위해서 제작한 Pair Class이다.
- 간단하게 String 두개를 만든후에 Constructor, Getter를 만들어 완성했다.

src/main/java/Main.java

```
import java.io.File;  
import java.io.FileNotFoundException;  
import java.util.Scanner;  
  
public class Main {  
    //txt파일 불러옴  
    public static void main(String[] args) throws FileNotFoundException {  
  
        boolean optionFlag = false;  
  
        if(args.length == 0 || args.length > 2) {  
            System.out.println("Error");  
            System.exit(1);  
        }  
    }  
}
```

```

    }
    if(args.length == 2 && !args[0].equals("-v")) {
        System.out.println("Error");
        System.exit(1);
    }
    if(args[0].equals("-v")) {
        optionFlag = true;
    }

    Scanner scanner = new Scanner(new File(args[args.length-1]));

    StringBuilder lines = new StringBuilder();
    while(scanner.hasNextLine()) {
        lines.append(scanner.nextLine());
    }
    scanner.close();

    //공백과 개행문자 같이 지움
    String tmp = lines.toString().replaceAll(" ", "");
    tmp = tmp.replaceAll("\n", "");
    if(tmp.isEmpty()) {
        if(optionFlag) {
            return;
        }
        System.out.println("Statement : ");
        System.out.println("ID : 0 CONST : 0 OP : 0");
        System.out.println("(Error) - There is no input.");
        return;
    }

    //개행문자 제거후 한 문장으로 묶기
    String input;
    input = lines.toString().replaceAll("\n", "");
    Program program = new Program(input);
    program.run(optionFlag);
}
}

```

- 프로그램에서 가장 먼저 실행되는 부분이다.
- 우선 args의 길이를 분석하여 길이가 1과 2가 아닐 경우에는 에러가 뜨게 했고, 2일 때는 첫번째 요소가 -v가 아니면 에러가 뜨게 했다.
- 두 검사를 통과했을 경우 배열 맨 첫번째 값이 -v이면 optionflag를 true로 세팅한 후에 파일을 Scanner를 통해 읽고, StringBuilder를 통해 저장한다.
- 그 후에 tmp를 선언하고 replaceAll을 통해서 개행문자와 공백을 다 제거한 이후에 tmp가 비면 오류를 출력하고 프로그램을 종료시킨다.
- 검사를 다 통과했으면 저장한 문자의 개행문자를 모두 제거한 후에 Program에 문장과 OptionFlag를 넣고 돌린다.

src/main/java/Program.java

- 전체 코드

```

import java.util.ArrayList;
import java.util.List;

public class Program {

    private String input;
    private List<String> Statements = new ArrayList<>();

    private boolean semicolonFlag = false;

    Program(String input) {
        this.input = input;
        this.Statements = splitStatements();
    }

    //문장을 ;기준으로 나누어 리스트에 저장
    private List<String> splitStatements() {
        if(input.charAt(input.length() - 1) == ';' ) {
            semicolonFlag = true;
            input = input.substring(0, input.length() - 1);
        }
        List<String> result = new ArrayList<>();
    }

```

```

String[] tmp = input.split(";");
for(String statement : tmp) {
    result.add(statement.trim());
}
for(int i = 0; i < result.size() - 1; i++) {
    result.set(i, result.get(i) + ";");
}
return result;
}

public void run(boolean optionFlag) {
    for(String tmp : Statements) {
        Parser statement = new Parser(tmp, optionFlag);
        if(semicolanFlag && tmp.equals(Statements.get(Statements.size() - 1))) {
            statement.addErrors(new Pair("(Warning)", "Semicolon in the last statement is not required. - Delete semicolon"));
        }
        statement.run();
    }
    if(!optionFlag) Parser.printResult();
}
}

```

- 과제 Document에 있는 문법들 중 Program, Statements에 관련된 부분이다.
- 우선 input을 받은 후에 input바탕으로 splitStatements() 함수를 돌려서 Statement들을 얻는다.

- splitStatements() 함수

```

private List<String> splitStatements() {
    if(input.charAt(input.length() - 1) == ';') {
        semicolanFlag = true;
        input = input.substring(0, input.length() - 1);
    }
    List<String> result = new ArrayList<>();
    String[] tmp = input.split(";");
    for(String statement : tmp) {
        result.add(statement.trim());
    }
    for(int i = 0; i < result.size() - 1; i++) {
        result.set(i, result.get(i) + ";");
    }
    return result;
}

```

- Warning 처리를 위해 문장의 끝이 세미콜론일때, semicolanFlag를 true로 바꾸고 뒤의 세미콜론을 없앤다.
- 그리고 tmp로 input을 세미콜론 기준으로 자르고 이를 result 리스트에 추가한다.
- 그 후에 맨 마지막 문장을 뺀 나머지의 문장 각각의 마지막에 세미콜론을 추가하고 이를 반환한다.

- run() 함수

```

public void run(boolean optionFlag) {
    for(String tmp : Statements) {
        Parser statement = new Parser(tmp, optionFlag);
        if(semicolanFlag && tmp.equals(Statements.get(Statements.size() - 1))) {
            statement.addErrors(new Pair("(Warning)", "Semicolon in the last statement is not required. - Delete semicolon"));
        }
        statement.run();
    }
    if(!optionFlag) Parser.printResult();
}

```

- optionFlag를 input으로 받는다.
- Statements를 for문으로 돌려 각각의 Statement를 Parser를 통해 파싱한다.
- 만약 semicolanFlag가 true이고 tmp가 Statements의 마지막 요소라면 statement에 Warning을 추가한다.
- for문을 다 돌린 후에 OptionFlag가 false라면 printResult() 함수를 통해 마지막 연산결과를 출력한다.

src/main/java/Program.java

```
import java.util.*;

public class Parser {

    private boolean errorFlag = false;
    private static boolean errorFlag2 = false;
    private final boolean optionFlag;
    private boolean assignFlag = false;
    private String charClass;
    private String lexeme;
    private String LHS;
    private static final TreeSet<String> IdentList = new TreeSet<>();
    private final List<String> lexemes = new ArrayList<>();
    private final List<String> tokens = new ArrayList<>();
    private final List<Pair> errors = new ArrayList<>();
    private static final HashMap<String, Integer> IdentValue = new HashMap<>();
    private char nextChar;
    private int lexLen;
    private String nextToken;
    private String expr;
    private final int[] result = new int[3]; //0 ID, 1 CONST, 2 OP

    Parser(String expr, Boolean optionFlag) {
        this.expr = expr;
        this.optionFlag = optionFlag;
        lexLen = 0;
        nextToken = "";
        lexeme = "";
        getchar();
    }

    void setIdentValue(String ident, int value) {
        IdentValue.put(ident, value);
    }

    void updateIdentValue(String ident, Integer value) {
        IdentValue.replace(ident, value);
    }

    public void addErrors(Pair pair) {
        errors.add(pair);
    }

    void addLexeme(String str) {
        lexemes.add(str);
    }

    void resetLexeme() {
        lexeme = "";
        lexLen = 0;
    }

    void lookup(char ch) {
        result[2]++;
        switch (ch) {
            case '(' -> {
                addchar();
                nextToken = String.valueOf(Token.LEFT_PAREN);
                addLexeme(lexeme);
                tokens.add(nextToken);
                result[2]--;
            }
            case ')' -> {
                addchar();
                nextToken = String.valueOf(Token.RIGHT_PAREN);
                addLexeme(lexeme);
                tokens.add(nextToken);
                result[2]--;
            }
            case '+', '-' -> {
                addchar();
                nextToken = String.valueOf(Token.ADD_OP);
                addLexeme(lexeme);
                tokens.add(nextToken);
            }
            case '*', '/' -> {
                addchar();
                nextToken = String.valueOf(Token.MULT_OP);
            }
        }
    }
}
```

```

        addLexeme(lexeme);
        tokens.add(nextToken);
    }
    case ':' -> {
        addchar();
        getchar();
        if (nextChar == '=') {
            addchar();
        } else {
            errors.add(new Pair("Warning", "Change : to := "));
            lexeme += "=";
        }
        nextToken = String.valueOf(Token.ASSIGN_OP);
        addLexeme(lexeme);
        tokens.add(nextToken);
        result[2]--;
    }
    case ';' -> {
        addchar();
        nextToken = String.valueOf(Token.SEMICOLON);
        addLexeme(lexeme);
        tokens.add(nextToken);
        result[2]--;
    }
    case '=' -> {
        errors.add(new Pair("Warning", "Change = to := "));
        lexeme += ":";
        addchar();
        nextToken = String.valueOf(Token.ASSIGN_OP);
        addLexeme(lexeme);
        tokens.add(nextToken);
        result[2]--;
    }
    case (char)-1 -> {
        addchar();
        result[2]--;
        nextToken = String.valueOf(Token.EOF);
        tokens.add(nextToken);
    }
    default -> {
        errorFlag = true;
        errors.add(new Pair("Error", "Unexpected operator"));
        addchar();
        addLexeme(lexeme);
        nextToken = String.valueOf(Token.ERROR);
        tokens.add(nextToken);
        result[2]--;
    }
}

}

void addchar() {
    if (lexLen <= 98) {
        lexeme += nextChar;
        lexLen++;
    } else {
        errorFlag = true;
        errors.add(new Pair("Error", "Identifier too long"));
    }
}

void getchar() {
    if (!expr.isEmpty()) {
        nextChar = expr.charAt(0);
        expr = expr.substring(1);
        if (Character.isLetter(nextChar)) {
            charClass = String.valueOf(CharClass.LETTER);
        } else if (Character.isDigit(nextChar)) {
            charClass = String.valueOf(CharClass.DIGIT);
        } else {
            charClass = String.valueOf(CharClass.UNKNOWN);
        }
    } else {
        charClass = String.valueOf(CharClass.EOF);
    }
}

void lex() {
    lexLen = 0;
    getNonBlank();
    switch (charClass) {
        case "LETTER" -> {
            addchar();
            getchar();
        }
    }
}

```

```

        while (charClass.equals(String.valueOf(CharClass.DIGIT)) || charClass.equals(String.valueOf(CharClass.LETTER))) {
            addchar();
            getchar();
        }
        result[0]++;
        addLexeme(lexeme);
        IdentList.add(lexeme);
        nextToken = String.valueOf(Token.IDENT);
        tokens.add(nextToken);
    }
    case "DIGIT" -> {
        addchar();
        getchar();
        while (charClass.equals(String.valueOf(CharClass.DIGIT))) {
            addchar();
            getchar();
        }
        result[1]++;
        addLexeme(lexeme);
        nextToken = String.valueOf(Token.INT_LIT);
        tokens.add(nextToken);
    }
    case "EOF" -> {
        nextToken = String.valueOf(Token.EOF);
        tokens.add(nextToken);
    }
    case "UNKNOWN" -> {
        lookup(nextChar);
        getchar();
    }
}
resetLexeme();
}

void getNonBlank() {
    while (Character.isWhitespace(nextChar)) {
        getchar();
    }
}

int statement() {
    if (nextToken.equals(String.valueOf(Token.IDENT))) {
        lex();
        if (nextToken.equals(String.valueOf(Token.ASSIGN_OP))) {
            assignFlag = true;
            LHS = lexemes.get(lexemes.size() - 2);
            lex();
            while (nextToken.equals(String.valueOf(Token.ASSIGN_OP))) {
                errors.add(new Pair("Warning", "Consecutive Assignment Operator - Remove Duplicate Assignment Operator"));
                lexemes.remove(lexemes.size() - 1);
                tokens.remove(tokens.size() - 1);
                lex();
            }
            return expression();
        } else {
            errorFlag = true;
            errors.add(new Pair("Error", "No assignment operator"));
        }
    } else {
        if (nextToken.equals(String.valueOf(Token.ASSIGN_OP))) {
            errorFlag = true;
            errors.add(new Pair("Error", "Assignment operator cannot be used without identifier"));
        } else {
            errorFlag = true;
            errors.add(new Pair("Error", "Unexpected identifier"));
        }
    }
}
return 0;
}

int expression() {
    int num1 = term();
    ArrayList<Map<String, Integer>> tmp = termTail();
    if (!tmp.isEmpty()) {
        for (Map<String, Integer> map : tmp) {
            for (String str : map.keySet()) {
                if (str.equals("+")) {
                    num1 += map.get(str);
                } else {
                    num1 -= map.get(str);
                }
            }
        }
    }
}
}

```

```

        return num1;
    }

    ArrayList<Map<String, Integer>> termTail() {
        if (nextToken.equals(String.valueOf(Token.ADD_OP))) {
            String op = lexemes.get(lexemes.size() - 1);
            lex();
            while (!nextToken.equals(String.valueOf(Token.INT_LIT)) && !nextToken.equals(String.valueOf(Token.IDENT))
                    && !nextToken.equals(String.valueOf(Token.LEFT_PAREN))) {
                //ADD_OP일때
                if (nextToken.equals(String.valueOf(Token.ADD_OP))) {
                    if (lexemes.get(lexemes.size() - 1).equals(op)) {
                        errors.add(new Pair("Warning", "Duplicate operators were found - Remove duplicate operator (" + op + ")"));
                        lexemes.remove(lexemes.size() - 1);
                        tokens.remove(tokens.size() - 1);
                    } else {
                        lexemes.remove(lexemes.size() - 1);
                        tokens.remove(tokens.size() - 1);
                        errors.add(new Pair("Warning", "Consecutive operators were found - Remove Backward operator " + op));
                    }
                }
                //MULT_OP일때
                else if (nextToken.equals(String.valueOf(Token.MULT_OP))) {
                    errors.add(new Pair("Warning", "Consecutive operators were found - Remove Backward Operator " + lexemes.get(lexemes.size() - 1)));
                    lexemes.remove(lexemes.size() - 1);
                    tokens.remove(tokens.size() - 1);
                }
                else {
                    errorFlag = true;
                    errors.add(new Pair("Error", "Unexpected token"));
                    return new ArrayList<>();
                }
            }
            result[2]--;
            lex();
        }
        int num = term();
        Map<String, Integer> tmp = new HashMap<>();
        tmp.put(op, num);
        ArrayList<Map<String, Integer>> additonal = termTail();
        ArrayList<Map<String, Integer>> tmpMap = new ArrayList<>();
        tmpMap.add(tmp);
        tmpMap.addAll(additonal);
        return tmpMap;
    }
    else if (!nextToken.equals(String.valueOf(Token.SEMICOLON)) || nextToken.equals(String.valueOf(Token.EOF))
            || nextToken.equals(String.valueOf(Token.RIGHT_PAREN)) || nextToken.equals(String.valueOf(Token.ERROR))) {
        errorFlag = true;
        errors.add(new Pair("Error", "Unexpected token"));
    }
    return new ArrayList<>();
}

int term() {
    int num1 = factor();
    ArrayList<Map<String, Integer>> tmp = factorTail();
    if (!tmp.isEmpty()) {
        for (Map<String, Integer> map : tmp) {
            for (String str : map.keySet()) {
                if (str.equals("")) {
                    num1 *= map.get(str);
                } else {
                    num1 /= map.get(str);
                }
            }
        }
    }
    return num1;
}

ArrayList<Map<String, Integer>> factorTail() {
    if (nextToken.equals(String.valueOf(Token.MULT_OP))) {
        String op = lexemes.get(lexemes.size() - 1);
        lex();
        while (!nextToken.equals(String.valueOf(Token.INT_LIT)) && !nextToken.equals(String.valueOf(Token.IDENT))
                && !nextToken.equals(String.valueOf(Token.LEFT_PAREN))) {
            //MULT_OP일때
            if (nextToken.equals(String.valueOf(Token.MULT_OP))) {
                if (lexemes.get(lexemes.size() - 1).equals(op)) {
                    errors.add(new Pair("Warning", "Duplicate operators were found - Remove duplicate operator (" + op + ")"));
                    lexemes.remove(lexemes.size() - 1);
                    tokens.remove(tokens.size() - 1);
                } else {
                    lexemes.remove(lexemes.size() - 1);
                }
            }
        }
    }
}

```

```

        errors.add(new Pair("Warning)", "Consecutive operators were found - Remove Backward operator " + op));
        tokens.remove(tokens.size() - 1);
    }
} else if (nextToken.equals(String.valueOf(Token.ADD_OP))) {
    errors.add(new Pair("Warning)", "Consecutive operators were found - Remove Backward operator " + lexemes.get(lexemes.size() - 1));
    lexemes.remove(lexemes.size() - 1);
    tokens.remove(tokens.size() - 1);
}
else {
    errorFlag = true;
    errors.add(new Pair("Error)", "Unexpected token"));
    return new ArrayList<>();
}
result[2]--;
lex();
}
int num = factor();
Map<String, Integer> tmp = new HashMap<>();
tmp.put(op, num);
ArrayList<Map<String, Integer>> additonal = factorTail();
ArrayList<Map<String, Integer>> tmpMap = new ArrayList<>();
tmpMap.add(tmp);
tmpMap.addAll(additonal);
return tmpMap;
}
else if (!(nextToken.equals(String.valueOf(Token.SEMICOLON)) || nextToken.equals(String.valueOf(Token.EOF))
|| nextToken.equals(String.valueOf(Token.ADD_OP)) || nextToken.equals(String.valueOf(Token.RIGHT_PAREN)) || nextToken.equals(String.valueOf(Token.LEFT_PAREN)))) {
    errorFlag = true;
    errors.add(new Pair("Error)", "Unexpected token"));
}
return new ArrayList<>();
}

int factor() {
    if (nextToken.equals(String.valueOf(Token.LEFT_PAREN))) {
        lex();
        int tmp = expression();
        if (nextToken.equals(String.valueOf(Token.RIGHT_PAREN))) {
            lex();
            return tmp;
        } else {
            errors.add(new Pair("Warning)", "not found right parenthesis - add right parenthesis in the end"));
            lexemes.add(")");
            tokens.add(String.valueOf(Token.RIGHT_PAREN));
            if(optionFlag) System.out.println(")");
            return tmp;
        }
    }
    else if (nextToken.equals(String.valueOf(Token.IDENT))) {
        if (!IdentValue.containsKey(lexemes.get(lexemes.size() - 1))) {
            if (!(IdentList.contains(lexemes.get(lexemes.size() - 1)) && !errorFlag2)) {
                errorFlag = true;
                errors.add(new Pair("Error)", "Undefined identifier (" + lexemes.get(lexemes.size() - 1) + ")"));
                IdentValue.put(lexemes.get(lexemes.size() - 1), null);
            }
            else if (IdentList.contains(lexemes.get(lexemes.size() - 1))) {
                errorFlag = true;
                errors.add(new Pair("Error)", "Undefined identifier (" + lexemes.get(lexemes.size() - 1) + ")"));
            }
        }
        lex();
        if (!errorFlag && !errorFlag2) {
            if (nextToken.equals(String.valueOf(Token.EOF))) {
                return IdentValue.get(lexemes.get(lexemes.size() - 1));
            }
            return IdentValue.get(lexemes.get(lexemes.size() - 2));
        }
        return 0;
    }
    else if (nextToken.equals(String.valueOf(Token.INT_LIT))) {
        lex();
        if (nextToken.equals(String.valueOf(Token.EOF))) {
            return Integer.parseInt(lexemes.get(lexemes.size() - 1));
        }
        else if (nextToken.equals(String.valueOf(Token.ERROR))) {
            return 0;
        }
        return Integer.parseInt(lexemes.get(lexemes.size() - 2));
    }
    else {
        errorFlag = true;
        errors.add(new Pair("Error)", "Unexpected token"));
        return 0;
    }
}

void run() {

```



```

lex();
int num = statement();
remainLexer();
if (!optionFlag) {
    System.out.print("Statement : ");
    StringBuilder stringBuilder = new StringBuilder();
    for (String str : lexemes) {
        if(str.equals("(")) stringBuilder.append(str);
        else if(str.equals("(") || str.equals(";")) {
            if(stringBuilder.charAt(stringBuilder.length() - 1) == ' ') {
                stringBuilder.replace(stringBuilder.length() - 1, stringBuilder.length(), str);
                stringBuilder.append(" ");
            }
            else stringBuilder.append(str).append(" ");
        }
        else stringBuilder.append(str).append(" ");
    }
    System.out.println(stringBuilder);
    System.out.println("ID : " + result[0] + " CONST : " + result[1] + " OP : " + result[2]);
    if(!errors.isEmpty()) {
        for(Pair pair : errors) {
            System.out.println(pair.getFirst() + " - " + pair.getSecond());
        }
    } else {
        System.out.println("OK");
    }
}
if (!errorFlag) {
    if (IdentValue.containsKey(LHS)) {
        updateIdentValue(LHS, num);
    } else {
        setIdentValue(LHS, num);
    }
    //System.out.println("Result => " + LHS + " = " + IdentValue.get(LHS));
} else {
    updateIdentValue(LHS, null);
    //System.out.println("Result => " + LHS + " = Unknown ");
}
errorFlag2 = errorFlag || errorFlag2;
}
else {
    for(String str : tokens) {
        System.out.println(str);
    }
}
}

public static void printResult() {
    System.out.print("=> Result : ");
    for (String str : IdentList) {
        if (IdentValue.get(str) == null) {
            System.out.print(str + " = Unknown ");
        } else {
            System.out.print(str + " = " + IdentValue.get(str) + " ");
        }
    }
}

public void remainLexer() {
    while (!nextToken.equals(String.valueOf(Token.EOF))) {
        lex();
        if(assignFlag && nextToken.equals(String.valueOf(Token.ASSIGN_OP))) {
            errors.add(new Pair("Error", "Assignment Operator must be one"));
        }
        else if(nextToken.equals(String.valueOf(Token.IDENT))) {
            if(!IdentValue.containsKey(lexemes.get(lexemes.size() - 1))) {
                errors.add(new Pair("Error", "Undefined identifier (" + lexemes.get(lexemes.size() - 1) + ")"));
            }
        }
    }
}
}
}
}

```

- Statement에서 factor까지의 기능을 구현하고, 결과, 에러 출력과, 에러가 발생했을때 나머지를 Lex하는 함수를 구현한 클래스이다.

- 변수

```

private boolean errorFlag = false;
private static boolean errorFlag2 = false;
private final boolean optionFlag;

```

```

private boolean assignFlag = false;
private String charClass;
private String lexeme;
private String LHS;
private static final TreeSet<String> IdentList = new TreeSet<>();
private final List<String> lexemes = new ArrayList<>();
private final List<String> tokens = new ArrayList<>();
private final List<Pair> errors = new ArrayList<>();
private static final HashMap<String, Integer> IdentValue = new HashMap<>();
private char nextChar;
private int lexLen;
private String nextToken;
private String expr;
private final int[] result = new int[3]; //0 ID, 1 CONST, 2 OP

```

- 에러를 판단 하는 errorFlag와 errorFlag2, -v 옵션을 판단하는 optionFlag, assignment Operator가 들어왔는지 판단하는 assignFlag가 있다.
- 다음 Lexeme의 Class를 판단하는 charClass변수, LHS를 저장하는 LHS변수, lexeme을 저장하는 lexeme변수와 Ident를 모아두는 IdentList, lexeme들을 모아두는 lexemes, token들을 모아두는 tokens, error들을 모아두는 errors, Ident와 그에 대한 값을 모아두는 IdentValue가 있다.
- 다음 Char를 판단하는 nextChar, lexeme의 길이를 판단하는 lexLen, 다음 lexeme의 Token을 판단하는 nextToken, 그리고 Statement를 받아들이는 expr, Ident, Constant, Operator의 값을 모아두는 result 배열이 있다.

- Constructor ~ resetLexeme함수

```

Parser(String expr, Boolean optionFlag) {
    this.expr = expr;
    this.optionFlag = optionFlag;
    lexLen = 0;
    nextToken = "";
    lexeme = "";
    getchar();
}

void setIdentValue(String ident, int value) {
    IdentValue.put(ident, value);
}

void updateIdentValue(String ident, Integer value) {
    IdentValue.replace(ident, value);
}

public void addErrors(Pair pair) {
    errors.add(pair);
}

void addLexeme(String str) {
    lexemes.add(str);
}

void resetLexeme() {
    lexeme = "";
    lexLen = 0;
}

```

- Constructor에서는 expr와 optionFlag를 불러와서 Parser Class의 변수에 저장하고, LexLen과 nextToken, lexeme의 기본값을 맞춘 후에 getchar()함수를 한번 실행시켜 문장이 Statement에 진입할 준비를 하게한다.
- setIdentValue()는 Ident와 value를 인자로 받아서 IdentValue에 그 값들을 넣게 한다.
- updateIdentValue()는 Ident와 value를 인자로 받아 IdentValue의 값을 업데이트 한다.
- addErrors()는 pair하나를 받아서 그 값을 errors 리스트에 추가한다.
- addlexeme()는 string하나를 인자로 받아 lexemes 리스트에 추가한다.
- restLexeme()는 lexeme을 초기화하고, lexLen도 0으로 초기화한다.
- lookup()

```

void lookup(char ch) {
    result[2]++;
    switch (ch) {
        case '(' -> {
            addchar();
            nextToken = String.valueOf(Token.LEFT_PAREN);
            addLexeme(lexeme);
            tokens.add(nextToken);
            result[2]--;
        }
        case ')' -> {
            addchar();
            nextToken = String.valueOf(Token.RIGHT_PAREN);
            addLexeme(lexeme);
            tokens.add(nextToken);
            result[2]--;
        }
        case '+', '-' -> {
            addchar();
            nextToken = String.valueOf(Token.ADD_OP);
            addLexeme(lexeme);
            tokens.add(nextToken);
        }
        case '*', '/' -> {
            addchar();
            nextToken = String.valueOf(Token.MULT_OP);
            addLexeme(lexeme);
            tokens.add(nextToken);
        }
        case ':' -> {
            addchar();
            getchar();
            if (nextChar == '=') {
                addchar();
            } else {
                errors.add(new Pair("Warning", "Change : to := "));
                lexeme += "=";
            }
            nextToken = String.valueOf(Token.ASSIGN_OP);
            addLexeme(lexeme);
            tokens.add(nextToken);
            result[2]--;
        }
        case ';' -> {
            addchar();
            nextToken = String.valueOf(Token.SEMICOLON);
            addLexeme(lexeme);
            tokens.add(nextToken);
            result[2]--;
        }
        case '=' -> {
            errors.add(new Pair("Warning", "Change = to := "));
            lexeme += "=";
            addchar();
            nextToken = String.valueOf(Token.ASSIGN_OP);
            addLexeme(lexeme);
            tokens.add(nextToken);
            result[2]--;
        }
        case (char)-1 -> {
            addchar();
            result[2]--;
            nextToken = String.valueOf(Token.EOF);
            tokens.add(nextToken);
        }
        default -> {
            errorFlag = true;
            errors.add(new Pair("Error", "Unexpected operator"));
            addchar();
            addLexeme(lexeme);
            nextToken = String.valueOf(Token.ERROR);
            tokens.add(nextToken);
            result[2]--;
        }
    }
}
}

```

- 문자기호와 EOF를 판단하는 함수이다.
- Char하나를 인자로 받아 우선 Operator쪽 배열의 값을 하나 추가하고, switch-case문을 통해서 문자를 판단해서 그 값일때의 nextToken을 설정하고 tokens에 nextToken을 추가하고, 문자를 Lexeme에 세팅하고, lexemes에 이를 추가한다.

- 혹시 문법에서 정의된 기호가 아닌 다른 기호들이 들어온다면 errorFlag를 세팅하고, errors에 에러 코드와 error를 추가한다. 하지만 그 후에도 Lexer는 계속 작업을 이어가야하므로 nextToken을 세팅하고 lexemes와 tokens에 이를 추가한다.
- 혹시 `:` 이나 `=` 이 들어왔을 경우 errors에 warning 코드와 warning을 추가하고, 이들을 lexeme을 직접 수정하는 작업을 거쳐 `:=` 로 바꾼다. 그 후에 이것들을 nextToken을 설정하고, tokens에 이를 추가하고, lexemes에 `:=` 을 추가함으로써 정상적으로 문장이 진행되도록 한다.

- addchar() ~ getchar()

```
void addchar() {
    if (lexLen <= 98) {
        lexeme += nextChar;
        lexLen++;
    } else {
        errorFlag = true;
        errors.add(new Pair("(Error)", "Identifier too long"));
    }
}

void getchar() {
    if (!expr.isEmpty()) {
        nextChar = expr.charAt(0);
        expr = expr.substring(1);
        if (Character.isLetter(nextChar)) {
            charClass = String.valueOf(CharClass.LETTER);
        } else if (Character.isDigit(nextChar)) {
            charClass = String.valueOf(CharClass.DIGIT);
        } else {
            charClass = String.valueOf(CharClass.UNKNOWN);
        }
    } else {
        charClass = String.valueOf(CharClass.EOF);
    }
}
```

- addchar()는 lexeme에 nextchar를 추가하고 lexLen을 증가시킨다. 만약 lexLen이 98을 넘을 경우 에러를 배출하게 한다.
- getchar()는 expr가 비지 않았을 경우, nextchar를 expr의 가장 첫번째 문자로 세팅하고, expr에 그 문자를 뺀다. 그 후에 nextchar이 숫자인지, 문자인지, 기호인지를 판단하여 charClass를 세팅한다. 만약 expr가 비어있을 경우, charClass를 EOF로 세팅한다.

- lex()

```
void lex() {
    lexLen = 0;
    getNonBlank();
    switch (charClass) {
        case "LETTER" -> {
            addchar();
            getchar();
            while (charClass.equals(String.valueOf(CharClass.DIGIT)) || charClass.equals(String.valueOf(CharClass.LETTER))) {
                addchar();
                getchar();
            }
            result[0]++;
            addLexeme(lexeme);
            IdentList.add(lexeme);
            nextToken = String.valueOf(Token.IDENT);
            tokens.add(nextToken);
        }
        case "DIGIT" -> {
            addchar();
            getchar();
            while (charClass.equals(String.valueOf(CharClass.DIGIT))) {
                addchar();
                getchar();
            }
            result[1]++;
            addLexeme(lexeme);
            nextToken = String.valueOf(Token.INT_LIT);
            tokens.add(nextToken);
        }
        case "EOF" -> {
            nextToken = String.valueOf(Token.EOF);
            tokens.add(nextToken);
        }
    }
}
```

```

    }
    case "UNKNOWN" -> {
        lookup(nextChar);
        getchar();
    }
}
resetLexeme();
}

```

- 우선 lexlen을 0으로 세팅한 후, getNonBlank() 함수를 호출하여 공백을 지우고, switch-case문으로 charClass에 따라서 addchar()와 getchar()를 사용하여 lexeme을 완성한 후에 이를 lexemes에 추가하고, 각각의 charClass에 맞는 result의 값을 증가시키고, nextToken을 세팅하고, tokens에 nextToken을 추가했다.
- 혹시 charClass의 값이 unknown일 경우에는 lookup 함수를 호출하여 기호를 판단하고 lexeme을 세팅하게 했다.
- 그 후에 lexeme을 초기화함으로써 다음 lexeme을 쓸 수 있게 세팅했다.

• getNonBlank()

```

void getNonBlank() {
    while (Character.isWhitespace(nextChar)) {
        getchar();
    }
}

```

- 공백을 무시하기 위해서 만들어진 함수이다.
- nextChar가 공백일 경우, getChar를 한번 더 호출해 공백을 무시하도록 세팅했다.

• statement()

```

int statement() {
    if (nextToken.equals(String.valueOf(Token.IDENT))) {
        lex();
        if (nextToken.equals(String.valueOf(Token.ASSIGN_OP))) {
            assignFlag = true;
            LHS = lexemes.get(lexemes.size() - 2);
            lex();
            while (nextToken.equals(String.valueOf(Token.ASSIGN_OP))) {
                errors.add(new Pair("Warning", "Consecutive Assignment Operator - Remove Duplicate Assignment Operator"));
                lexemes.remove(lexemes.size() - 1);
                tokens.remove(tokens.size() - 1);
                lex();
            }
            return expression();
        } else {
            errorFlag = true;
            errors.add(new Pair("Error", "No assignment operator"));
        }
    } else {
        if (nextToken.equals(String.valueOf(Token.ASSIGN_OP))) {
            errorFlag = true;
            errors.add(new Pair("Error", "Assignment operator cannot be used without identifier"));
        } else {
            errorFlag = true;
            errors.add(new Pair("Error", "Unexpected identifier"));
        }
    }
    return 0;
}

```

- 문법에서의 statement를 담당하는 함수이다.
- 만약 nextToken이 Ident일 경우에는 lex() 함수를 실행시키고, nextToken이 Assignment Operator일 경우에는, assignFlag를 true로 둔 다음에 LHS를 세팅한다. 그 후에 lex를 돌려, Assignment Operator가 연속적으로 나올 경우에는 warning과 warning code를 errors에 추가하고, 이 Assignment operator는 무시한다. 아닐 경우에는 expression()을 실행시키고 그 값을 반환한다.
- 만약 Ident 다음 토큰이 Assignment Operator가 아닐 경우에는 Assignment Operator가 없기 때문에 오류를 추가하고 0을 반환한다.

- 만약 첫번째 Token이 Ident가 아닐경우, 경우를 판단해서 첫번째 Token이 Assignment Operator일 때와 아닐때에 다른 에러를 띄우도록 설계했다.

- expression()

```
int expression() {
    int num1 = term();
    ArrayList<Map<String, Integer>> tmp = termTail();
    if (!tmp.isEmpty()) {
        for (Map<String, Integer> map : tmp) {
            for (String str : map.keySet()) {
                if (str.equals("+")) {
                    num1 += map.get(str);
                } else {
                    num1 -= map.get(str);
                }
            }
        }
    }
    return num1;
}
```

- 첫번째 변수 num1은 Term()의 값으로 세팅하고, termTail()의 결과값을 ArrayList로 받아 for문을 돌려 tmp List의 Map의 key값에 따라 Map의 Value값을 num1에 더하고 뺀다. 그 후에 num1을 반환하여 계산결과를 Statement에 전달한다.

- termTail()

```
ArrayList<Map<String, Integer>> termTail() {
    if (nextToken.equals(String.valueOf(Token.ADD_OP))) {
        String op = lexemes.get(lexemes.size() - 1);
        lex();
        while (!nextToken.equals(String.valueOf(Token.INT_LIT)) && !nextToken.equals(String.valueOf(Token.IDENT))
            && !nextToken.equals(String.valueOf(Token.LEFT_PAREN))) {
            //ADD_OP일때
            if (nextToken.equals(String.valueOf(Token.ADD_OP))) {
                if (lexemes.get(lexemes.size() - 1).equals(op)) {
                    errors.add(new Pair("Warning", "Duplicate operators were found - Remove duplicate operator (" + op + ")"));
                    lexemes.remove(lexemes.size() - 1);
                    tokens.remove(tokens.size() - 1);
                } else {
                    lexemes.remove(lexemes.size() - 1);
                    tokens.remove(tokens.size() - 1);
                    errors.add(new Pair("Warning", "Consecutive operators were found - Remove Backward operator " + op));
                }
            }
            //MULT_OP일때
            else if (nextToken.equals(String.valueOf(Token.MULT_OP))) {
                errors.add(new Pair("Warning", "Consecutive operators were found - Remove Backward Operator " + lexemes.get(lexemes.size() - 1)));
                lexemes.remove(lexemes.size() - 1);
                tokens.remove(tokens.size() - 1);
            }
            else {
                errorFlag = true;
                errors.add(new Pair("Error", "Unexpected token"));
                return new ArrayList<>();
            }
        }
        result[2]--;
        lex();
    }
    int num = term();
    Map<String, Integer> tmp = new HashMap<>();
    tmp.put(op, num);
    ArrayList<Map<String, Integer>> additonal = termTail();
    ArrayList<Map<String, Integer>> tmpMap = new ArrayList<>();
    tmpMap.add(tmp);
    tmpMap.addAll(additonal);
    return tmpMap;
}
else if(!(nextToken.equals(String.valueOf(Token.SEMICOLON)) || nextToken.equals(String.valueOf(Token.EOF))
    || nextToken.equals(String.valueOf(Token.RIGHT_PAREN)) || nextToken.equals(String.valueOf(Token.ERROR)))) {
    errorFlag = true;
    errors.add(new Pair("Error", "Unexpected token"));
}
```

```

        return new ArrayList<>();
    }

```

- 만약 nextToken의 값이 ADD_OP이면 op의 값을 lexemes에서 불러와서 저장하고, lex를 돌린다. 그 후에도 nextToken의 값이 ADD_OP라면 op의 값과 Lexeme의 값과 비교하여 같을 경우와 다를 경우의 warning 메시지를 다르게 출력하고, 뒤의 OP는 Lexemes에서 삭제시키고 tokens에 마지막 Token을 삭제시킴으로써 중복된 Op를 처리했다. 만약 nextToken의 값이 MULT_OP여도 뒤의 OP와 token을 삭제시키고 warning메시지를 추가한다. 그 외의 경우는 errorFlag를 true로 세팅하고 에러 메시지를 errors에 추가한 후에 빈 ArrayList를 반환한다. 아닐 경우에는 result[2] 즉, Operator의 갯수를 하나 감소시키고 lex를 다시 돌린다. 이는 nextToken이 변수나 Constant, 또는 왼쪽 괄호가 나올때까지 반복한다.
- 다 돌린 후에 Operand의 값은 term()의 값을 세팅해두고, 새로운 HashMap을 만들어 op와 num을 넣는다. 그 후에 termTail의 ArrayList의 값과 합쳐, tmpMap에다가 저장하고 이를 반환한다.
- 만약 nextToken의 값이 ADD_OP의 값이 아니고, 세미콜론, EOF, 오른쪽 괄호, 그리고 에러가 아니라면 errorFlag의 값을 세팅하고 errors에 에러를 추가한다. 그 후에 빈 ArrayList를 반환한다.

• term()

```

int term() {
    int num1 = factor();
    ArrayList<Map<String, Integer>> tmp = factorTail();
    if (!tmp.isEmpty()) {
        for (Map<String, Integer> map : tmp) {
            for (String str : map.keySet()) {
                if (str.equals("**")) {
                    num1 *= map.get(str);
                } else {
                    num1 /= map.get(str);
                }
            }
        }
    }
    return num1;
}

```

- num1의 값을 factor()의 값으로 세팅한다. 그후에 factorTail()의 값을 tmp에 저장하고, for문을 돌려 tmp의 key값에 따라 tmp의 value값을 num1에 곱하고 나눈다. 그 후에 num1의 값을 반환한다.

• factorTail()

```

ArrayList<Map<String, Integer>> factorTail() {
    if (nextToken.equals(String.valueOf(Token.MULT_OP))) {
        String op = lexemes.get(lexemes.size() - 1);
        lex();
        while (!nextToken.equals(String.valueOf(Token.INT_LIT)) && !nextToken.equals(String.valueOf(Token.IDENT))
            && !nextToken.equals(String.valueOf(Token.LEFT_PAREN))) {
            //MULT_OP일때
            if (nextToken.equals(String.valueOf(Token.MULT_OP))) {
                if (lexemes.get(lexemes.size() - 1).equals(op)) {
                    errors.add(new Pair("Warning", "Duplicate operators were found - Remove duplicate operator (" + op + ")"));
                    lexemes.remove(lexemes.size() - 1);
                    tokens.remove(tokens.size() - 1);
                } else {
                    lexemes.remove(lexemes.size() - 1);
                    errors.add(new Pair("Warning", "Consecutive operators were found - Remove Backward operator " + op));
                    tokens.remove(tokens.size() - 1);
                }
            } else if (nextToken.equals(String.valueOf(Token.ADD_OP))) {
                errors.add(new Pair("Warning", "Consecutive operators were found - Remove Backward operator " + lexemes.get(lexemes.size() - 1)));
                lexemes.remove(lexemes.size() - 1);
                tokens.remove(tokens.size() - 1);
            } else {
                errorFlag = true;
                errors.add(new Pair("Error", "Unexpected token"));
                return new ArrayList<>();
            }
            result[2]--;
            lex();
        }
        int num = factor();
    }
}

```

```

        Map<String, Integer> tmp = new HashMap<>();
        tmp.put(op, num);
        ArrayList<Map<String, Integer>> additonal = factorTail();
        ArrayList<Map<String, Integer>> tmpMap = new ArrayList<>();
        tmpMap.add(tmp);
        tmpMap.addAll(additonal);
        return tmpMap;
    }
    else if(!(nextToken.equals(String.valueOf(Token.SEMICOLON)) || nextToken.equals(String.valueOf(Token.EOF))
        || nextToken.equals(String.valueOf(Token.ADD_OP)) || nextToken.equals(String.valueOf(Token.RIGHT_PAREN)) || nextToken.e
            errorFlag = true;
            errors.add(new Pair("(Error)", "Unexpected token"));
        }
    }
    return new ArrayList<>();
}

```

- 만약 nextToken의 값이 MULT_OP이면 op의 값을 lexemes에서 불러와서 저장하고, lex를 돌린다. 그 후에도 nextToken의 값이 MULT_OP라면 op의 값과 nextToken의 Lexeme의 값과 비교하여 같을 경우와 다를 경우의 warning 메시지를 다르게 출력하고, 뒤의 OP는 Lexemes에서 삭제시키고 tokens에서 token을 삭제시킴으로써 중복된 Op를 처리했다. 만약 nextToken의 값이 ADD_OP 여도 뒤의 OP를 삭제시키고, warning메시지를 추가한다. 그 외의 경우는 errorFlag를 true로 세팅하고 에러 메시지를 errors에 추가한 후에 빈 ArrayList를 반환한다. 아닐 경우에는 result[2] 즉, Operator의 갯수를 하나 감소시키고 lex를 다시 돌린다. 이는 nextToken이 변수나 Constant, 또는 왼쪽 괄호가 나올때까지 반복한다.
- 다 돌린 후에 Operand의 값을 factor()의 값을 세팅해두고, 새로운 HashMap을 만들어 op와 num을 넣는다. 그 후에 factorTail()의 ArrayList의 값과 합쳐, tmpMap에다가 저장하고 이를 반환한다.
- 만약 nextToken의 값이 MULT_OP가 아니고, ADD_OP, 세미콜론, EOF, 오른쪽 괄호, 그리고 에러가 아니라면 errorFlag의 값을 세팅하고 errors에 에러를 추가한다. 그 후에 빈 ArrayList를 반환한다.

• factor()

```

int factor() {
    if (nextToken.equals(String.valueOf(Token.LEFT_PAREN))) {
        lex();
        int tmp = expression();
        if (nextToken.equals(String.valueOf(Token.RIGHT_PAREN))) {
            lex();
            return tmp;
        } else {
            errors.add(new Pair("(Warning)", "not found right parenthesis - add right parenthesis in the end"));
            lexemes.add("");
            tokens.add(String.valueOf(Token.RIGHT_PAREN));
            return tmp;
        }
    } else if (nextToken.equals(String.valueOf(Token.IDENT))) {
        if (!IdentValue.containsKey(lexemes.get(lexemes.size() - 1))) {
            if (!IdentList.contains(lexemes.get(lexemes.size() - 1)) && !errorFlag2) {
                errorFlag = true;
                errors.add(new Pair("(Error)", "Undefined identifier (" + lexemes.get(lexemes.size() - 1) + ")"));
                IdentValue.put(lexemes.get(lexemes.size() - 1), null);
            } else if (IdentList.contains(lexemes.get(lexemes.size() - 1))) {
                errorFlag = true;
                errors.add(new Pair("(Error)", "Undefined identifier (" + lexemes.get(lexemes.size() - 1) + ")"));
            }
        }
        lex();
        if (!errorFlag && !errorFlag2) {
            if (nextToken.equals(String.valueOf(Token.EOF))) {
                return IdentValue.get(lexemes.get(lexemes.size() - 1));
            }
            return IdentValue.get(lexemes.get(lexemes.size() - 2));
        }
        return 0;
    } else if (nextToken.equals(String.valueOf(Token.INT_LIT))) {
        lex();
        if (nextToken.equals(String.valueOf(Token.EOF))) {
            return Integer.parseInt(lexemes.get(lexemes.size() - 1));
        } else if (nextToken.equals(String.valueOf(Token.ERROR))) {
            return 0;
        }
        return Integer.parseInt(lexemes.get(lexemes.size() - 2));
    } else {
        errorFlag = true;
        errors.add(new Pair("(Error)", "Unexpected token"));
    }
}

```



```

        return 0;
    }
}

```

- 만약 nextToken의 값이 왼쪽괄호라면, lex를 돌린후에 tmp의 값을 expression()의 값으로 세팅한다. expression()을 다 돌고 난 이후, 만약 nextToken의 값이 오른쪽 괄호라면 lex를 한번 더 한 후에 tmp의 값을 반환한다. 만약 nextToken의 값이 오른쪽 괄호가 아니라면, warning을 띄우고, lexemes에 오른쪽 괄호를 추가하고 tokens에 RIGHT_PAREN을 넣는다. 만약 optionFlag가 true였다면 오른쪽 괄호를 출력한다. 그 후에 lex를 한번 더 한 이후에 tmp의 값을 반환한다.
- 만약 nextToken의 값이 변수라면 IdentValue에 해당 lexeme에 대한 값이 있는지 파악한 후에 만약 없으면 IdentList에 값이 없고, errorFlag2의 값이 false라면 errorFlag를 true로하고, errors에 해당 에러 코드를 추가한다. 그 후에 IdentValue에 key값과 null값을 삽입한다. 만약 IdentList에 값이 있으면 errorFlag를 true로 하고 errors에 해당 에러 코드를 추가한다. 그 후에 lex를 돌린 후에 만약 errorFlag와 errorFlag2의 값이 모두 False라면 nextToken이 EOF인지 아닌지에 따라서 lexemes의 있는 값의 위치를 달리하여 IdentValue의 값을 반환한다. 만약 errorFlag나 errorFlag2의 값중에 하나라도 true일 경우 0을 반환한다.
- 만약 nextToken의 값이 상수라면 lex를 한번 돌리고, 만약 nextToken이 EOF인지 아닌지에 따라 위치를 다르게 가져가서 lexemes의 값을 반환한다. 만약 nextToken의 값이 error라면 0을 반환한다.
- 만약 nextToken의 값이 왼쪽괄호, 변수, 상수가 아니라면, errorFlag를 true로 세팅하고 errors에 에러코드를 추가한뒤에 0을 반환한다.

- run()


```

void run() {
    lex();
    int num = statement();
    remainLexer();
    if (!optionFlag) {
        System.out.print("Statement : ");
        StringBuilder stringBuilder = new StringBuilder();
        for (String str : lexemes) {
            if(str.equals("(")) stringBuilder.append(str);
            else if(str.equals("(") || str.equals(";")) {
                if(stringBuilder.charAt(stringBuilder.length() - 1) == ' ') {
                    stringBuilder.replace(stringBuilder.length() - 1, stringBuilder.length(), str);
                    stringBuilder.append(" ");
                }
                else stringBuilder.append(str).append(" ");
            }
            else stringBuilder.append(str).append(" ");
        }
        System.out.println(stringBuilder);
        System.out.println("ID : " + result[0] + " CONST : " + result[1] + " OP : " + result[2]);
        if(!errors.isEmpty()) {
            for(Pair pair : errors) {
                System.out.println(pair.getFirst() + " - " + pair.getSecond());
            }
        } else {
            System.out.println("OK");
        }
        if (!errorFlag) {
            if (IdentValue.containsKey(LHS)) {
                updateIdentValue(LHS, num);
            } else {
                setIdentValue(LHS, num);
            }
            //System.out.println("Result => " + LHS + " = " + IdentValue.get(LHS));
        } else {
            updateIdentValue(LHS, null);
            //System.out.println("Result => " + LHS + " = Unknown ");
        }
        errorFlag2 = errorFlag || errorFlag2;
    }
    else {
        for(String str : tokens) {
            System.out.println(str);
        }
    }
}

```

- 처음에 lex를 돌린후에 num의 값을 statement()의 반환값으로 세팅한다. 그 후에 오류가 나서 중단된 부분을 다 Lexing하기 위해 remainLexer()를 돌린다. 그 후에 optionFlag가 false라면 Statement를 StringBuilder를 사용해서 lexemes의 값들을 이어붙여서 출력하고, Ident, Const, Operator의 갯수를 출력한다. 만약 errors List가 비지 않았으면 에러들과 Warning들을 출력하고, 아니라면

(OK)를 출력한다. 만약 errorFlag가 false라면 LHS가 IdentValue에 있으면 IdentValue의 LHS값을 num으로 업데이트하고, 아니면 LHS와 num을 IdentValue에 삽입한다. 만약 errorFlag가 true라면 IdentValue의 LHS의 값에 null을 삽입한다. 그 후에 errorFlag2의 값을 errorFlag와 errorFlag2의 or값으로 세팅한다.

- 만약 optionFlag가 true라면  옵션에 따라 for문을 돌려서 문장의 token들을 출력한다.

- printResult()

```
public static void printResult() {
    System.out.print("=> Result : ");
    for (String str : IdentList) {
        if (IdentValue.get(str) == null) {
            System.out.print(str + " = Unknown ");
        } else {
            System.out.print(str + " = " + IdentValue.get(str) + " ");
        }
    }
}
```

- Result의 값을 출력한다. IdentList의 값을 for문으로 돌려서 만약 IdentValue에서 IdentList가 키인 값이 null일 경우에 값을 Unknown으로 출력하고 아니면 값을 IdentValue의 값으로 출력한다.

- remainLexer()

```
public void remainLexer() {
    while (!nextToken.equals(String.valueOf(Token.EOF))) {
        lex();
        if(assignFlag && nextToken.equals(String.valueOf(Token.ASSIGN_OP))) {
            errors.add(new Pair("(Error)", "Assignment Operator must be one"));
        }
        else if(nextToken.equals(String.valueOf(Token.IDENT))) {
            if(!IdentValue.containsKey(lexemes.get(lexemes.size() - 1))) {
                errors.add(new Pair("(Error)", "Undefined identifier (" + lexemes.get(lexemes.size() - 1) + ")"));
            }
        }
    }
}
```

- error가 생긴이후에 중단된 Lex를 끝까지 마저 하기 위해서 만든 함수이다.
- nextToken이 EOF가 나올때까지 lex를 돌리고, 만약 assignFlag가 true이고 nextToken이 Assignment Operator가 나오면, error를 추가하고, 만약 nextToken이 Ident면 IdentValue의 key가 lexeme일때의 값이 존재하지 않으면 error를 추가한다.