COVER PAGE

CS323 Programming Assignments

Fill out all entries 1 - 7. If not, there will be deductions!

Check one

1. Names [1. Andrew Gomez], (MW [ ] or R class [X])

   [2. Zihao Qiu], (MW [ ] or R class [X])

   [**if 3**. Evan Purpura], (MW [ ] or R class [X])

2. Assignment Number [2]

3. Due Dates **Softcopy** [11/12], **Hardcopy** [11/15]

4. Turn-In Dates **Softcopy** [11/15], **Hardcopy** [11/15]

5. Executable FileName [compilers.jar]
   (**A file that can be executed without compilation by the instructor**)

6. LabRoom [CS401]
(Execute your program in a lab in the CS building before submission)

7. Operating System [Windows]

To be filled out by the Instructor:

GRADE:

COMMENTS:

CS323 Documentation

1. Problem Statement

create a syntax analyzer uses the lexer() from assignment #1 and the new method parser() in the compiler_parser class to take a token and check if it fits in the RAT18F grammar. The tokens are distinguished from one another with the use of the lexical analyzer. After getting the tokens, the syntax analyzer parses the token through each function using Top-Down RDP approach with nested if else statements.

2. How to use your program

To utilize our program you must:

a. Download all of the files from the titanium submission

b. Make sure you have the JDK and JRE downloaded so the Java Virtual Machine can compile java files on your machine.

c. Set up your system path variables to recognize the JDK path: resources

d. The important files in this program are:

- i. src <
  1. compilers.jar
  2. **run_jar.bat**
  3. Test.txt
  4. Test2.txt
  5. Test3.txt

e. You will locate the project folder on your computer directory

f. Place your test file in the same directory as the downloaded compilers.jar file

g. Go into the folder on the command prompt

h. Then run the command: "**java -jar compilers.jar**"

OR

i. Double click the batch file "**run_jar.bat" if you are running on a windows** machine.

j. Follow prompt and enter in the name of the file to send to the program

example: **test.txt**

k. And the output on command line will print the Token and lexeme

l. Your output will also be written to the file **output.txt**

3. Design of your program

i. Main.java

1. Main class where the program is ran
2. Utilizes Scanner to scan input file for next "word" or each input that has a space in between it.
3. Removes comments if the comment symbol is seen in the lexeme

a. Checks for end comment symbol in this lexeme and all following lexemes until one is found and the parsing can continue.

4. parseSeparatorOperator(): Function as part of main class that utilizes an **ArrayList** to hold the lexemes that can be parsed from removing separators and operators that do not have a space.

ii. Compiler_LA.java

1. Main Lexical analyzer class that identifies a Token class of 7 types:
    a. Identifier
    b. Real
    c. Keyword
    d. Integer
    e. Operator
    f. Separator
    g. Invalid
2. Has the main **lexer()** method:
    a. Method first checks if any of the defined separator, operator and keyword options match the lexeme and if so returns Token created from lexeme
    b. If not then the functions **isInteger(), isReal()** and **isIdentifier()** are called and return the new Token of specified type.

iii. FSMIdentifier.java

1. Has a two dimensional array of Integers that hold the state table for this given FSM
2. Constructor enters in the **ArrayList's** the valid states and acceptance states and first state
3. **isIdentifier()**: Function that handles checking if the current state is valid and not null. If the state is valid grabs the next state from the current state and next input.
4. **nextState()**: Takes in the current state and the next char input and checks what the table shows the next state to be depending on currentState and input char that is mapped to a column.
5. **char_to_col()**: Function that gets the column number of the input char and returns it.

iv. FSMInteger.java

1. Constructor enters in the **ArrayList's** the valid states and acceptance states and first state.
2. **isIdentifier()**: Function that handles checking if the current state is valid and not null. If the state is valid grabs the next state from the current state and next input.
3. **nextState()**: Since we only have one state besides the initial state in this state table, you can simply represent it as one switch statement in the nextState() that with any other input besides a number goes to an invalid NULL state.

v. FSMReal.java

1. Same structure as FSMIdentifier, but has a different state table for Real numbers.

vi. **Compiler_Parser.java**

1. This is the class that contains the parser logic.
2. Main method is the **parse()** method:
    1. This method takes in the **pre, curr and post** tokens and then utilizing the rules from the **Rat18F** grammar decides whether this token is correctly placed in the grammar and what rule is applied to support this.
    2. This method has a **case** statement that takes in the current **token.getToken()** to determine what rules to apply and what token type we are currently looking at.
    3. Inside each of these case statements is a pre method for each of the rules that are determined in our language, and sometimes post depending on how the first sets of each rule map to follow sets of the productions.
    4. There is an **error()** method that takes in the preToken and the currToken in order to determine if the specific valid lexeme does not fit the rules, what lexeme did we expect and print that in the main. It also has an error counter that will print the amount of errors in the program at the end of runtime.

First, get rid of the left recursion in RAT18F grammar.

<Expression> | <Condition>

<Function Definitions> ::= <Function> <Function Definitions '>
<Function Definitions '> ::= <Function Definitions> | ε

<Parameter List> ::= <Parameter><Parameter List Prime>
<Parameter List Prime> ::= ,<Parameter List> | ε

<Declaration List> := <Declaration> ; <Declaration List Prime>
<Declaration List Prime> := <Declaration List> | ε

<Statement List> ::= <Statement> <Statement List '>
<Statement List '> ::= <Statement List> | ε

<IDs> := <Identifier><IDs Prime>
<IDs Prime> := ,<IDs>| ε

<Expression> ::= <Term><Expression '> | <Term>
<Expression '> ::= + <Term><Expression '> | -<Term><Expression '> | ε

<Term> ::= <Factor> <Term '> | <Factor>
<Term '> ::= * <Factor> <Term '>| / <Factor> <Term '> | ε

4. **Any Limitation: No**
5. **Any shortcomings : No**

Source Code Listing:

Test Case 1:

```
[* TEST CASE 1: this is comment for this sample code which
  converts  Fahrenheit into Celcius *]

function convert1x (fahr : int)
{
        return 5 * (fahr -38.8978978909087) / 9;
}

$$
   int   x,  j, step;     [* declarations *]

   get (x, h, step);
   while (x  <  high )
      {  put (x);
         put (convert1x  (X));
         low =  low  +  step;
      }
    whileend

 $$
```

Results 1:

Token Lexeme
KEYWORD function
<Function Definitions> -> <Function><Function Definitions Prime>
<Function> -> function <Identifier>(<Opt Parameter List>)<Opt Declaration List><Body>

IDENTIFIER convert1x
<Function Definitions> -> <Function><Function Definitions'>
<Function> -> function <Identifier> (<Opt Parameter List>) <Opt Declaration List> <Body>

SEPARATOR (
<Factor> -> <Primary>
<Primary> -> <Identifier>(<IDs>)

IDENTIFIER fahr
<Function> -> function <Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>

SEPARATOR :
<Parameter List> -> <Parameter><Parameter List'>
<Parameter> -> <IDs>:<Qualifier>

KEYWORD int
<Parameter> -> <IDs>:<Qualifier>
<Qualifier> -> int

SEPARATOR)
<Function Definitions> -> <Function><Function Definitions'>
<Function> -> function <Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>

SEPARATOR {
<Function> -> function<Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>
<Body> -> {<Statement List>}

KEYWORD return
<Statement> -> <Return>
<Return> -> return<Expression>;

INTEGER 5
<Return> -> return <Expression>;
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Integer>

OPERATOR *
<Term> -> <Factor><Term'>
<Term'> -> *<Factor><Term'>

SEPARATOR (
<Factor> -> <Primary>
<Primary> -> (<Expression>)

IDENTIFIER fahr
<Function> -> function <Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>

OPERATOR -
<Expression> -> <Term><Expression'>
<Expression'> -> -<Term><Expression'>

REAL 38.8978978909087
<Factor> -> -<Primary>
<Primary> -> <Real>

SEPARATOR)

<Factor> -> <Primary>
<Primary> -> (<Expression>)

OPERATOR /
<Term> -> <Factor><Term'>
<Term'> -> /<Factor><Term'>

INTEGER 9
<Term'> -> /<Factor><Term'>
<Factor> -> <Primary>
<Primary> -> <Integer>

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

SEPARATOR }
<Statement> -> <Compound>
<Compound> -> {<Statement List>}

SEPARATOR $$
<Rat18F> -> <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$

KEYWORD int
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> int

IDENTIFIER x
<Decleration List> -> <Decleration>;<Declaration List'>
<Decleration> -> <int | boolean | real > <IDs>
<IDs> -> <Identifier><IDs'>
<IDs'> -> ,<IDs>

SEPARATOR ,
<IDs> -> <Identifier><IDs'>
<IDs'> -> ,<IDs>

IDENTIFIER j
<IDs'> -> ,<IDs>

SEPARATOR ,
<IDs> -> <Identifier><IDs'>
<IDs'> -> ,<IDs>

IDENTIFIER step
<IDs'> -> ,<IDs>

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

KEYWORD get
<Statement> -> <Scan>
<Scan> -> get(<IDs>);

SEPARATOR (
<Statement> -> <Scan>
<Scan> -> get(<IDs>);
IDENTIFIER x
<Primary> -> <Identifier>(<IDs>)<IDs> -> <Identifier><IDs Prime>
<IDs Prime> -> ,<IDs>

SEPARATOR ,
<IDs> -> <Identifier><IDs'>
<IDs'> -> ,<IDs>

IDENTIFIER h
<IDs'> -> ,<IDs>

SEPARATOR ,
<IDs> -> <Identifier><IDs'>
<IDs'> -> ,<IDs>

IDENTIFIER step
<IDs'> -> ,<IDs>

SEPARATOR)
<Factor> -> <Primary>
<Primary> -> (<Expression>)

SEPARATOR ;
<Statement> -> <Print>
<Print> -> put(<Expression>);

KEYWORD while
<Statement> -> <While>
<While> -> while(<Condition>) <Statement> whileend

SEPARATOR (
<Condition> -> <Expression><Relop><Expression>

IDENTIFIER x
<Function> -> function <Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>

OPERATOR <
<Condition> -> <Expression> <Relop> <Expression>
<Relop> -> <

IDENTIFIER high
<Condition> -> <Expression> <Relop> <Expression>
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Identifier>

SEPARATOR)
<Factor> -> <Primary>
<Primary> -> (<Expression>)

SEPARATOR {
<Function> -> function<Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>
<Body> -> {<Statement List>}

KEYWORD put
<Statement> -> <Print>
<Print> -> put(<Expression>);

SEPARATOR (
<Statement> -> <Print>
<Print> -> put(<Expression>);

IDENTIFIER x
<Primary> -> <Identifier>(<IDs>)
<IDs> -> <Identifier><IDs Prime>
<IDs Prime> -> ε

SEPARATOR)
<Factor> -> <Primary>
<Primary> -> (<Expression>)

SEPARATOR ;
<Statement> -> <Print>
<Print> -> put(<Expression>);

KEYWORD put
<Statement> -> <Print>
<Print> -> put(<Expression>);

SEPARATOR (
<Statement> -> <Print>
<Print> -> put(<Expression>);

IDENTIFIER convert1x
<Function> -> function <Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>

SEPARATOR (
<Factor> -> <Primary>
<Primary> -> <Identifier>(<IDs>)

IDENTIFIER X
<Primary> -> <Identifier>(<IDs>)
<IDs> -> <Identifier><IDs Prime>
<IDs Prime> -> ε

SEPARATOR)
<Factor> -> <Primary>
<Primary> -> (<Expression>)

SEPARATOR)
<Factor> -> <Primary>
<Primary> -> (<Expression>)

SEPARATOR ;

<Statement> -> <Print>
<Print> -> put(<Expression>);

IDENTIFIER low
<Statement>-> <Assign>
<Assign> -><Identifier> = <Expression>;

OPERATOR =
<Assign> -> <Identifier> = <Expression>

IDENTIFIER low
<Assign> -> <Identifier> = <Expression>;
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Identifier>

OPERATOR +
<Expression> -> <Term><Expression'>
<Expression'> -> +<Term><Expression'>

IDENTIFIER step
<Expression'> -> +<Term><Expression'>
<Term> -> <Factor><Term'>
<Factor> -> <Primary>
<Primary> -> <Identifier>

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

SEPARATOR }
<Statement> -> <Compound>
<Compound> -> {<Statement List>}

KEYWORD whileend
<While> -> while(<Condition>) <Statement> whileend

SEPARATOR $$
<Rat18F> -> <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$

Your program parsed with a total of: 0 errors

Test Case 2:

[* TEST CASE 2 *]

```
function multiply (xA : int, yB : int)
{
        return xA * yB;
}
```

```
$$
        int xA, yB;

        xA = 5;
        yA = 1.32;

        int zC = multiply(xA , yB);

        put(zC);
$$
```

Results 2:

Token Lexeme

KEYWORD function
<Function Definitions> -> <Function><Function Definitions Prime>
<Function> -> function <Identifier>(<Opt Parameter List>)<Opt Declaration List><Body>

IDENTIFIER multiply
<Function Definitions> -> <Function><Function Definitions'>
<Function> -> function <Identifier> (<Opt Parameter List>) <Opt Declaration List> <Body>

SEPARATOR (
<Factor> -> <Primary>
<Primary> -> <Identifier>(<IDs>)

IDENTIFIER xA
<Function> -> function <Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>

SEPARATOR :
<Parameter List> -> <Parameter><Parameter List'>
<Parameter> -> <IDs>:<Qualifier>

KEYWORD int
<Parameter> -> <IDs>:<Qualifier>
<Qualifier> -> int

SEPARATOR ,
<IDs> -> <Identifier><IDs'>
<IDs'> -> ,<IDs>

IDENTIFIER yB
<IDs'> -> ,<IDs>

SEPARATOR :
<Parameter List> -> <Parameter><Parameter List'>
<Parameter> -> <IDs>:<Qualifier>

KEYWORD int
<Parameter> -> <IDs>:<Qualifier>
<Qualifier> -> int

SEPARATOR)
<Function Definitions> -> <Function><Function Definitions'>
<Function> -> function <Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>

SEPARATOR {
<Function> -> function<Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>
<Body> -> {<Statement List>}

KEYWORD return
<Statement> -> <Return>
<Return> -> return<Expression>;

IDENTIFIER xA
<Statement> -> <Return>
<Return> -> return <Expression>

OPERATOR *
<Term> -> <Factor><Term'>
<Term'> -> *<Factor><Term'>

IDENTIFIER yB
<Term'> -> *<Factor><Term'>
<Factor> -> <Primary>
<Primary> -> <Identifier>

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

SEPARATOR }
<Statement> -> <Compound>
<Compound> -> {<Statement List>}

SEPARATOR $$
<Rat18F> -> <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$

KEYWORD int
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> int

IDENTIFIER xA
<Decleration List> -> <Decleration>;<Declaration List'>
<Decleration> -> <int | boolean | real > <IDs>
<IDs> -> <Identifier><IDs'>
<IDs'> -> ,<IDs>

SEPARATOR ,
<IDs> -> <Identifier><IDs'>
<IDs'> -> ,<IDs>

IDENTIFIER yB
<IDs'> -> ,<IDs>

SEPARATOR ;

<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

IDENTIFIER xA
<Statement>-> <Assign>
<Assign> -><Identifier> = <Expression>;

OPERATOR =
<Assign> -> <Identifier> = <Expression>

INTEGER 5
<Assign> -> <Identifier> = <Expression>;
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Integer>

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

IDENTIFIER yA
<Statement>-> <Assign>
<Assign> -><Identifier> = <Expression>;

OPERATOR =
<Assign> -> <Identifier> = <Expression>

REAL 1.32
<Assign> -> <Identifier> = <Expression>;
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Real>

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

KEYWORD int
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> int

IDENTIFIER zC
<Decleration List> -> <Decleration>;<Declaration List'>
<Decleration> -> <int | boolean | real> <IDs>
<IDs> -> <Identifier><IDs'>

OPERATOR =
<Assign> -> <Identifier> = <Expression>

IDENTIFIER multiply
<Assign> -> <Identifier> = <Expression>;
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Identifier>

SEPARATOR (
<Factor> -> <Primary>
<Primary> -> <Identifier>(<IDs>)

IDENTIFIER xA
<Primary> -> <Identifier>(<IDs>)<IDs> -> <Identifier><IDs Prime>
<IDs Prime> -> ,<IDs>

SEPARATOR ,
<IDs> -> <Identifier><IDs'>
<IDs'> -> ,<IDs>

IDENTIFIER yB
<IDs'> -> ,<IDs>

SEPARATOR)
<Factor> -> <Primary>
<Primary> -> (<Expression>)

SEPARATOR ;
<Statement> -> <Print>
<Print> -> put(<Expression>);

KEYWORD put
<Statement> -> <Print>
<Print> -> put(<Expression>);

SEPARATOR (
<Statement> -> <Print>
<Print> -> put(<Expression>);

IDENTIFIER zC
<Primary> -> <Identifier>(<IDs>)
<IDs> -> <Identifier><IDs Prime>
<IDs Prime> -> ε

SEPARATOR)
<Factor> -> <Primary>
<Primary> -> (<Expression>)

SEPARATOR ;
<Statement> -> <Print>
<Print> -> put(<Expression>);

SEPARATOR $$
<Rat18F> -> <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$

Your program has a total of: 0 errors

Test Case 3:

```
[* TEST CASE 3 *]

$$

        int count = 0;
        int aMeaninglessNumber = 0;

        boolean is100x = true;

        while(is100x){
                count = 2 * 3;
        }

        aMeaninglessNumber = count * 2322222;
$$
```

Results 3:

```
Token                   Lexeme
SEPARATOR               $$
<Rat18F> -> <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$

KEYWORD                 int
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> int

IDENTIFIER              count
<Decleration List> -> <Decleration>;<Declaration List'>
<Decleration> -> <int | boolean | real> <IDs>
<IDs> -> <Identifier><IDs'>

OPERATOR                =
<Assign> -> <Identifier> = <Expression>

INTEGER                 0
<Assign> -> <Identifier> = <Expression>;
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Integer>

SEPARATOR               ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

KEYWORD                 int
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> int

IDENTIFIER              aMeaninglessNumber
<Decleration List> -> <Decleration>;<Declaration List'>
```

<Decleration> -> <int | boolean | real> <IDs>
<IDs> -> <Identifier><IDs'>

OPERATOR =
<Assign> -> <Identifier> = <Expression>

INTEGER 0
<Assign> -> <Identifier> = <Expression>;
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Integer>

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

KEYWORD boolean
<Declaration> -> <Qualifier> <IDs>
<Qualifier> -> boolean

IDENTIFIER is100x
<Decleration List> -> <Decleration>;<Declaration List'>
<Decleration> -> <int | boolean | real> <IDs>
<IDs> -> <Identifier><IDs'>

OPERATOR =
<Assign> -> <Identifier> = <Expression>

KEYWORD true
<Factor> -> <Primary>
<Primary> -> true

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

KEYWORD while
<Statement> -> <While>
<While> -> while(<Condition>) <Statement> whileend

SEPARATOR (
<Condition> -> <Expression><Relop><Expression>

IDENTIFIER is100x
<Primary> -> <Identifier>(<IDs>)
<IDs> -> <Identifier><IDs Prime>
<IDs Prime> -> ε

SEPARATOR)
<Factor> -> <Primary>
<Primary> -> (<Expression>)

SEPARATOR {
<Function> -> function<Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>

<Body> -> {<Statement List>}

IDENTIFIER count
<Statement>-> <Assign>
<Assign> -><Identifier> = <Expression>;

OPERATOR =
<Assign> -> <Identifier> = <Expression>

INTEGER 2
<Assign> -> <Identifier> = <Expression>;
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Integer>

OPERATOR *
<Term> -> <Factor><Term'>
<Term'> -> *<Factor><Term'>

INTEGER 3
<Term'> -> *<Factor><Term'>
<Factor> -> <Primary>
<Primary> -> <Integer>

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

SEPARATOR }
<Statement> -> <Compound>
<Compound> -> {<Statement List>}

IDENTIFIER aMeaninglessNumber
<Statement>-> <Assign>
<Assign> -><Identifier> = <Expression>;

OPERATOR =
<Assign> -> <Identifier> = <Expression>

IDENTIFIER count
<Assign> -> <Identifier> = <Expression>;
<Expression> -> <Term><Expression'>
<Factor> -> <Primary>
<Primary> -> <Identifier>

OPERATOR *
<Term> -> <Factor><Term'>
<Term'> -> *<Factor><Term'>

INTEGER 2322222
<Term'> -> *<Factor><Term'>
<Factor> -> <Primary>
<Primary> -> <Integer>

SEPARATOR ;
<Statement> -> <Assign>
<Assign> -> <Identifier>=<Expression>;

SEPARATOR $$
<Rat18F> -> <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$

Your program parsed with a total of: 0 errors

Source Listing

main.java <

```
import java.util.Scanner;
import java.io.*;
import java.util.ArrayList;
import java.util.HashSet;
public class main {

   public static final char[] separatorsAndOps = { '(', ')', '{', '}', '[', ']', ',', ';', ':', '=', '+', '-', '/', '*', '>', '<', '|', '^' };
   public static final HashSet<String> dubOps = new HashSet<>();
   public static ArrayList<String> separatedLexemes = new ArrayList<>();
   public static ArrayList<Compiler_LA.Token> allLexemes = new ArrayList<>();

   public static void main(String[] args) throws IOException {
      initDubOps();
      BufferedWriter writer =  new BufferedWriter( new FileWriter("output.txt"));
      writer.write("Token" + "\t\t\t" + "Lexeme");
      writer.newLine();

      //HERE ARE THE THREE LINES I ADDED TO TAKE IN FILE FROM USER
      System.out.print("Please input the filename that you want to test in the directory \nExample:
test.txt\n:");
      Scanner readInFile = new Scanner(System.in);
      String fileName = readInFile.next();


      File file = new File(fileName);
      Scanner input = new Scanner(file);


      //init local vars
      String lexeme = "", preLexeme = "";
      boolean isComment = false;
      int commStart, commEnd, lineNum = 1;

      //init local parser vars
      String parserHolder = "";

      //instantiate lexer class object in main
      Compiler_LA Compiler = new Compiler_LA();
      Compiler_LA.Token thisToken;
```

```
        //instantiate parser class object in main
        Compiler_Parser Parser = new Compiler_Parser();

        //while we have another lexeme to verify loop
        while(input.hasNext()){
            lexeme = input.next();
            if (input.hasNext(System.lineSeparator()) ) {lineNum++; System.out.println("A NEW LINE");
continue;}
            //ignore comments comments
            if(lexeme.contains("[*")) {
                commStart = lexeme.indexOf("[*");
                preLexeme = lexeme.substring(0, commStart);

                isComment = true;
                while(isComment) {
                    if (lexeme.contains("*]")) {
                        commEnd = lexeme.indexOf("*]"); //index might be off
                        lexeme = lexeme.substring(commEnd + 1, lexeme.length() - 1);
                        isComment = false;
                    }
                    else{
                        if(input.hasNext()){
                            lexeme = input.next();
                        }
                    }
                }

                if(!preLexeme.isEmpty()) {
                    parseSeparatorOperator(preLexeme);
                    for(String lexemeOpSep: separatedLexemes) {
                        //CALL TO LEXER ON PRELEXEME HERE
                        thisToken = Compiler.lexer(lexemeOpSep, lineNum);
                        //add token to universal program list of ordered lexemes
                        allLexemes.add(thisToken);

                        preLexeme = "";
                        //THIS IS WHERE OLD PRINT HAPPENED FOR PROJ 1
                        //System.out.println(thisToken);
                       // writer.write(thisToken.toString());
                       // writer.newLine();
                    }
                }
            }


                parseSeparatorOperator(lexeme);

            for(String lexemeOpSep: separatedLexemes) {
                //CALL TO LEXER HERE
                thisToken = Compiler.lexer(lexemeOpSep, lineNum);
                //add token to universal program list of ordered lexemes
                allLexemes.add(thisToken);
                //THIS IS WHERE OLD PRINT HAPPENED FOR PROJ 1
                //System.out.println(thisToken);
```

```
            //writer.write(thisToken.toString());
            //writer.newLine();
        }
    }

    //We call the parser on an iteration over the list of all lexemes in the program
    for(int i = 0; i < allLexemes.size() - 1; i++) {
        //CALL PARSER HERE FOR PRE, CURR AND POST LEXEME
        if(i == 0){
            parserHolder = Parser.parse(allLexemes.get(i), allLexemes.get(i), allLexemes.get(i + 1));
        } else {
            parserHolder = Parser.parse(allLexemes.get(i - 1), allLexemes.get(i), allLexemes.get(i + 1));
        }
        //This is where the output format printing occurs
        System.out.println(allLexemes.get(i));
        writer.write(allLexemes.get(i).toString());
        writer.newLine();
        System.out.println(parserHolder);
        writer.write(parserHolder);
        writer.newLine();
    }

    //parse the very last input
    parserHolder = Parser.parse(allLexemes.get(allLexemes.size()-2), allLexemes.get(allLexemes.size() -
1), allLexemes.get(allLexemes.size() - 1));
    System.out.println(allLexemes.get(allLexemes.size()-1));
    writer.write(allLexemes.get(allLexemes.size()-1).toString());
    writer.newLine();
    System.out.println(parserHolder);
    writer.write(parserHolder);
    writer.newLine();

    //print the number of errors in program
    System.out.println("Your program parsed with a total of: " + Parser.errors + " errors\n");
    writer.write("Your program parsed with a total of: " + Parser.errors + " errors");
    writer.newLine();

    //close file for proper program shutdown
    writer.flush();
    writer.close();
}

//function that allows for special case of operator or separator
public static void parseSeparatorOperator(String lexeme){
    separatedLexemes.clear();
    StringBuilder lexemeHolder = new StringBuilder();
    boolean charSepOp = false;

    for(int i = 0; i < lexeme.length(); i++) {
        charSepOp = false;

        for (char separatorOp : separatorsAndOps) {
            if (lexeme.charAt(i) == separatorOp) {
                charSepOp = true;
```

```
            if(lexemeHolder.length() > 0) {
               separatedLexemes.add(lexemeHolder.toString());
               lexemeHolder.setLength(0);
            }
            //if is for double operator being one lexeme and else runs for any single operator
            if( i + 1 < lexeme.length() && isValidDubOp(lexeme.charAt(i), lexeme.charAt(i+1))) {
               lexemeHolder.append(String.valueOf(lexeme.charAt(i)));
               lexemeHolder.append(String.valueOf(lexeme.charAt(i+1)));
               separatedLexemes.add(lexemeHolder.toString());
               lexemeHolder.setLength(0);
               i++;
               break;
            } else {
               separatedLexemes.add(String.valueOf(separatorOp));
               break;
            }
         }
      }

      if(!charSepOp) {
         lexemeHolder.append(String.valueOf(lexeme.charAt(i)));
      }
   }

   if(lexemeHolder.length() > 0) {
      separatedLexemes.add(lexemeHolder.toString());
   }
}

public static boolean isValidDubOp(Character firstChar, Character secondChar) {
   StringBuilder combineOps = new StringBuilder();
   boolean isValid = false;

   combineOps.append(firstChar);
   combineOps.append(secondChar);

   if(dubOps.contains(combineOps.toString())) {
      isValid = true;
   }

   return isValid;
}

public static void initDubOps() {
   dubOps.add("=<");
   dubOps.add("=>");
   dubOps.add("^=");
   dubOps.add("==");
   dubOps.add("+=");
   dubOps.add("-=");
   dubOps.add("*=");
   dubOps.add("/=");
}
```

```
}
```

Compiler_Parser.java <

```
import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;
import java.util.ArrayList;
public class Compiler_Parser {

   public enum ParserState {
      VALID, ERROR
   }

   private HashMap<Compiler_LA.Type, String> expectedToken = new HashMap<>();
   private StringBuilder outputforTrio;
   public int errors;

   public Compiler_Parser(){
      initHashMap();
      errors = 0;
   }

   //implement an expected token to pass into the error() function
   public String parse(Compiler_LA.Token preToken, Compiler_LA.Token currToken,
Compiler_LA.Token postToken) throws IOException {

      ParserState state = ParserState.ERROR;
      outputforTrio = new StringBuilder();

      // IDENTIFIER, KEYWORD, INTEGER, REAL, OPERATOR, SEPARATOR, INVALID
      //utilize past, current and future prediction to associate trio of lexemes to production rules
      //If any lexeme does not fulfill a production rule for its current position then it is deemed invalid and
sent to the error function
      switch(currToken.getToken()) {
         case IDENTIFIER:
            if (preToken.getLexeme().equals("function")){
               outputforTrio.append("<Function Definitions> -> <Function><Function Definitions'>\n" +
"<Function> -> function <Identifier> ( <Opt Parameter List> )  <Opt Declaration List>  <Body> \n");
            } else if (preToken.getLexeme().equals("return")) {
               outputforTrio.append( "<Statement> -> <Return>\n" + "<Return> -> return <Expression>\n");
            } else if(preToken.getLexeme().equals("int") || preToken.getLexeme().equals("boolean") ||
preToken.getLexeme().equals("real")){
               if(postToken.getLexeme().equals(",")) {
                  outputforTrio.append("<Decleration List> -> <Decleration>;<Declaration List'>\n" +
"<Decleration> -> <int | boolean | real > <IDs>\n" + "<IDs> -> <Identifier><IDs'>\n" + "<IDs'> ->
,<IDs>\n");
               } else {
                  outputforTrio.append("<Decleration List> -> <Decleration>;<Declaration List'>\n" +
"<Decleration> -> <int | boolean | real> <IDs>\n" + "<IDs> -> <Identifier><IDs'>\n");
               }
            } else if(preToken.getLexeme().equals("-")) {
               outputforTrio.append("<Factor> -> -<Primary>\n" + " <Primary> -> <Identifier>\n");
            } else if(preToken.getLexeme().equals("+")){
```

```
                outputforTrio.append("<Expression'> -> +<Term><Expression'>\n" + "<Term> ->
<Factor><Term'>\n" + "<Factor> -> <Primary>\n" + "<Primary> -> <Identifier>\n");
            } else if(preToken.getLexeme().equals("*")){
                outputforTrio.append("<Term'> -> *<Factor><Term'>\n" + "<Factor> -> <Primary>\n" +
"<Primary> -> <Identifier>\n");
            } else if(preToken.getLexeme().equals("/")){
                outputforTrio.append("<Term'> -> /<Factor><Term'>\n" + "<Factor> -> <Primary>\n" +
"<Primary> -> <Identifier>\n");
            } else if(preToken.getLexeme().equals(",")){
                outputforTrio.append("<IDs'>  ->  ,<IDs>\n");
            } else if(preToken.getLexeme().equals("=")){
                outputforTrio.append("<Assign> -> <Identifier> = <Expression>;\n" + "<Expression> ->
<Term><Expression'>\n" + "<Factor> -> <Primary>\n" + "<Primary> -> <Identifier>\n");
            } else if(preToken.getLexeme().equals(">") || preToken.getLexeme().equals("<") ){
                outputforTrio.append("<Condition> -> <Expression> <Relop> <Expression>\n" +
"<Expression> -> <Term><Expression'>\n" + "<Factor> -> <Primary>\n" + "<Primary> ->
<Identifier>\n");
            } else if(preToken.getLexeme().equals("(") || preToken.getLexeme().equals(")") ){
                if(postToken.getLexeme().equals(",")) {
                    outputforTrio.append("<Primary> -> <Identifier>(<IDs>)" + "<IDs> -> <Identifier><IDs
Prime>\n" + "<IDs Prime> -> ,<IDs>\n");
                } else if(postToken.getLexeme().equals(")")){
                    outputforTrio.append("<Primary> -> <Identifier>(<IDs>)\n" + "<IDs> -> <Identifier><IDs
Prime>\n" + "<IDs Prime> -> ε\n");
                } else {
                    outputforTrio.append("<Function> -> function <Identifier> (<Opt Parameter List>)<Opt
Declaration List><Body>\n");
                }
            } //else if(preToken.getLexeme().equals("") ){
                //possibly need to add one prod rule here for put but it should be counted in keyword section
              else if(postToken.getLexeme().equals("=") ){
                outputforTrio.append("<Statement>-> <Assign>\n" + "<Assign> -><Identifier> =
<Expression>;\n");
            } else {
                error(preToken, currToken);
            }
            break;
          case INTEGER:
            if(preToken.getLexeme().equals("-")){
                outputforTrio.append("<Factor> -> -<Primary>\n" + "<Primary> -> <Integer>\n");
            } else if(preToken.getLexeme().equals("+")){
                outputforTrio.append("<Expression'> -> +<Term><Expression'>\n" + "<Term> ->
<Factor><Term'>\n" + "<Factor> -> <Primary>\n" + "<Primary> -> <Integer>\n");
            } else if(preToken.getLexeme().equals("*")){
                outputforTrio.append("<Term'> -> *<Factor><Term'>\n" + "<Factor> -> <Primary>\n" +
"<Primary> -> <Integer>\n");
            } else if(preToken.getLexeme().equals("/")){
                outputforTrio.append("<Term'> -> /<Factor><Term'>\n" + "<Factor> -> <Primary>\n" +
"<Primary> -> <Integer>\n");
            } else if(preToken.getLexeme().equals(">") || preToken.getLexeme().equals("<")){
                outputforTrio.append("<Condition> -> <Expression><Relop><Expression>\n" +
"<Expression> -> <Term><Expression'>\n" + "<Factor> -> <Primary>\n" + "<Primary> -> <Integer>\n");
            } else if(preToken.getLexeme().equals("=")){
                outputforTrio.append("<Assign> -> <Identifier> = <Expression>;\n" + "<Expression> ->
```

```
<Term><Expression'>\n" + "<Factor> -> <Primary>\n" + "<Primary> -> <Integer>\n");
            } else if(preToken.getLexeme().equals("return")){
               outputforTrio.append("<Return> -> return <Expression>;\n" + "<Expression> ->
<Term><Expression'>\n" + "<Factor> -> <Primary>\n" + "<Primary> -> <Integer>\n");
            }else {
               error(preToken, currToken);
            }
            break;
         case REAL:
            if(preToken.getLexeme().equals("-")){
               outputforTrio.append("<Factor> -> -<Primary>\n" + "<Primary> -> <Real>\n");
            } else if(preToken.getLexeme().equals("+")){
               outputforTrio.append("<Expression'> -> +<Term><Expression'>\n" + "<Term> ->
<Factor><Term'>\n" + "<Factor> -> <Primary>\n" + "<Primary> -> <Real>\n");
            } else if(preToken.getLexeme().equals("*")){
               outputforTrio.append("<Term'> -> *<Factor><Term'>\n" + "<Factor> -> <Primary>\n" +
"<Primary> -> <Real>\n");
            } else if(preToken.getLexeme().equals("/")){
               outputforTrio.append("<Term'> -> /<Factor><Term'>\n" + "<Factor> -> <Primary>\n" +
"<Primary> -> <Real>\n");
            } else if(preToken.getLexeme().equals(">") || preToken.getLexeme().equals("<")){
               outputforTrio.append("<Condition> -> <Expression> <Relop> <Expression>\n" +
"<Expression> -> <Term><Expression'>\n" + "<Factor> -> <Primary>\n" + "<Primary> -> <Real>\n");
            } else if(preToken.getLexeme().equals("=")){
               outputforTrio.append("<Assign> -> <Identifier> = <Expression>;\n" + "<Expression> ->
<Term><Expression'>\n" + "<Factor> -> <Primary>\n" + "<Primary> -> <Real>\n");
            } else {
               error(preToken, currToken);
            }
            break;
         case OPERATOR:
            if(currToken.getLexeme().equals("-") || currToken.getLexeme().equals("+")){
               //according to rules we need to check for all primary
               if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER || preToken.getToken() ==
Compiler_LA.Type.REAL || preToken.getToken() == Compiler_LA.Type.INTEGER ||
preToken.getLexeme().equals("true") || preToken.getLexeme().equals("false") ||
preToken.getLexeme().equals("(") || preToken.getLexeme().equals(")")) {
                  if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER || postToken.getToken() ==
Compiler_LA.Type.REAL || postToken.getToken() == Compiler_LA.Type.INTEGER ||
postToken.getLexeme().equals("true") || postToken.getLexeme().equals("false") ||
postToken.getLexeme().equals("(")){
                     //pre and post follow same rules so we passed the - or + check
                     if(currToken.getLexeme().equals("-")) {
                        outputforTrio.append("<Expression> -> <Term><Expression'>\n" + "<Expression'> ->
-<Term><Expression'>\n");
                     } else {
                        outputforTrio.append("<Expression> -> <Term><Expression'>\n" + "<Expression'> ->
+<Term><Expression'>\n");
                     }
                  } else {
                     error(preToken, currToken);
                  }
               } else {
                  error(preToken, currToken);
```

```
                }
            } else if(currToken.getLexeme().equals("*") || currToken.getLexeme().equals("/")){
                if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER || preToken.getToken() ==
Compiler_LA.Type.REAL || preToken.getToken() == Compiler_LA.Type.INTEGER ||
preToken.getLexeme().equals("true") || preToken.getLexeme().equals("false") ||
preToken.getLexeme().equals("(") || preToken.getLexeme().equals(")")) {
                    if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER || postToken.getToken() ==
Compiler_LA.Type.REAL || postToken.getToken() == Compiler_LA.Type.INTEGER ||
postToken.getLexeme().equals("true") || postToken.getLexeme().equals("false") ||
postToken.getLexeme().equals("(")){
                        //pre and post follow same rules so we passed the * or / check
                        if(currToken.getLexeme().equals("/")) {
                            outputforTrio.append("<Term> -> <Factor><Term'>\n" + "<Term'> ->
/<Factor><Term'>\n");
                        } else {
                            outputforTrio.append("<Term> -> <Factor><Term'>\n" + "<Term'> ->
*<Factor><Term'>\n");
                        }
                    } else {
                        error(preToken, currToken);
                    }
                } else {
                    error(preToken, currToken);
                }
            } else if(currToken.getLexeme().equals(">") || currToken.getLexeme().equals("<") ||
currToken.getLexeme().equals("=<") || currToken.getLexeme().equals("=>") ||
currToken.getLexeme().equals("^=") || currToken.getLexeme().equals("==")){
                if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER || preToken.getToken() ==
Compiler_LA.Type.REAL || preToken.getToken() == Compiler_LA.Type.INTEGER ||
preToken.getLexeme().equals("true") || preToken.getLexeme().equals("false") ||
preToken.getLexeme().equals("(")) {
                    if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER || postToken.getToken() ==
Compiler_LA.Type.REAL || postToken.getToken() == Compiler_LA.Type.INTEGER ||
postToken.getLexeme().equals("true") || postToken.getLexeme().equals("false") ||
postToken.getLexeme().equals("(")){
                        //pre and post follow same rules so we passed the relop check
                        if(currToken.getLexeme().equals(">")) {
                            outputforTrio.append("<Condition> -> <Expression> <Relop> <Expression>\n" +
"<Relop> -> >\n");
                        } else if(currToken.getLexeme().equals("<")){
                            outputforTrio.append("<Condition> -> <Expression> <Relop> <Expression>\n" +
"<Relop> -> <\n");
                        } else if(currToken.getLexeme().equals("=<")){
                            outputforTrio.append("<Condition> -> <Expression> <Relop> <Expression>\n" +
"<Relop> -> =<\n");
                        } else if(currToken.getLexeme().equals("=>")){
                            outputforTrio.append("<Condition> -> <Expression> <Relop> <Expression>\n" +
"<Relop> -> =>\n");
                        } else if(currToken.getLexeme().equals("^=")){
                            outputforTrio.append("<Condition> -> <Expression> <Relop> <Expression>\n" +
"<Relop> -> ^=\n");
                        } else if(currToken.getLexeme().equals("==")){
                            outputforTrio.append("<Condition> -> <Expression> <Relop> <Expression>\n" +
"<Relop> -> ==\n");
```

```
                    }
                } else {
                    error(preToken, currToken);
                }
            } else {
                error(preToken, currToken);
            }
        } else if(currToken.getLexeme().equals("=")){
            if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER){
                if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER || postToken.getToken() ==
Compiler_LA.Type.REAL || postToken.getToken() == Compiler_LA.Type.INTEGER ||
postToken.getLexeme().equals("true") || postToken.getLexeme().equals("false") ||
postToken.getLexeme().equals("(")){
                    outputforTrio.append("<Assign> -> <Identifier> = <Expression>\n");
                } else {
                    error(preToken, currToken);
                }
            } else {
                error(preToken, currToken);
            }

        }
        break;
      case KEYWORD:
        if(currToken.getLexeme().equals("int") ||currToken.getLexeme().equals("real") ||
currToken.getLexeme().equals("boolean") ) {
            if(preToken.getLexeme().equals(":")){
                outputforTrio.append("<Parameter> -> <IDs>:<Qualifier>\n" + "<Qualifier> -> " +
currToken.getLexeme() + "\n");
            } else if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER) {
                outputforTrio.append("<Declaration> -> <Qualifier> <IDs>\n" + "<Qualifier> -> " +
currToken.getLexeme() + "\n");
            } else {
                error(preToken, currToken);
            }
        } else if(currToken.getLexeme().equals("return")) {
            if(postToken.getLexeme().equals(";")) {
                outputforTrio.append("<Statement> -> <Return>\n" + "   <Return> -> return;\n");
            } else if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER || postToken.getToken()
== Compiler_LA.Type.REAL || postToken.getToken() == Compiler_LA.Type.INTEGER ||
postToken.getLexeme().equals("true") || postToken.getLexeme().equals("false") ||
postToken.getLexeme().equals("(")){
                outputforTrio.append("<Statement> -> <Return>\n" + "<Return> ->
return<Expression>;\n");
            } else {
                error(preToken, currToken);
            }

        } else if(currToken.getLexeme().equals("function")) {
            if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER) {
                outputforTrio.append("<Function Definitions> -> <Function><Function Definitions
Prime>\n" + "<Function> -> function  <Identifier>(<Opt Parameter List>)<Opt Declaration
List><Body>\n");
            } else {
```

```
                error(preToken, currToken);
            }

        } else if(currToken.getLexeme().equals("put") || currToken.getLexeme().equals("get")) {
            if(postToken.getLexeme().equals("(")) {
                if(currToken.getLexeme().equals("put")){
                    outputforTrio.append("<Statement> -> <Print>\n" + "<Print> -> put(<Expression>);\n");
                } else {
                    outputforTrio.append("<Statement> -> <Scan>\n" + "<Scan> -> get(<IDs>);\n");
                }
            } else {
                error(preToken, currToken);
            }

        } else if(currToken.getLexeme().equals("true") || currToken.getLexeme().equals("false")) {
            if(preToken.getLexeme().equals("-")){
                outputforTrio.append("<Factor> -> -<Primary>\n" + "<Primary> -> " +
currToken.getLexeme() + "\n");
            } else {
                outputforTrio.append("<Factor> -> <Primary>\n" + "<Primary> -> " +
currToken.getLexeme() + "\n");
            }

        } else if(currToken.getLexeme().equals("if") || currToken.getLexeme().equals("while")) {
            if(postToken.getLexeme().equals("(")) {
                if(currToken.getLexeme().equals("if")) {
                    outputforTrio.append("<Statement> -> <If>\n" + "<If> -> if (<Condition>) <Statement>
ifend\n");
                } else {
                    outputforTrio.append("<Statement> -> <While>\n" + "<While> ->  while(<Condition>)
<Statement> whileend\n");
                }
            } else {
                error(preToken, currToken);
            }

        } else if(currToken.getLexeme().equals("else") || currToken.getLexeme().equals("ifend") ||
currToken.getLexeme().equals("whileend")) {
            if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER || preToken.getToken() ==
Compiler_LA.Type.REAL || preToken.getToken() == Compiler_LA.Type.INTEGER ||
preToken.getLexeme().equals("(") || preToken.getLexeme().equals("}") ||
preToken.getLexeme().equals("true") || preToken.getLexeme().equals("false") ||
preToken.getLexeme().equals("=")) {
                if(currToken.getLexeme().equals("else")) {
                    outputforTrio.append("<Statement> -> <If>\n" + "<If> -> if (<Condition>) <Statement>
else <Statement> ifend\n");
                } else if(currToken.getLexeme().equals("ifend")){
                    outputforTrio.append("<If> -> if (<Condition>) <Statement> else <Statement> ifend\n");
                } else {
                    outputforTrio.append("<While> ->  while(<Condition>) <Statement> whileend\n");
                }
            } else {
                error(preToken, currToken);
            }
```

```
            } else {
               error(preToken, currToken);
            }
            break;
         case SEPARATOR:
            if (currToken.getLexeme().equals("(")) {
               if(preToken.getLexeme().equals("put")) {
                  if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER || postToken.getToken() ==
Compiler_LA.Type.REAL || postToken.getToken() == Compiler_LA.Type.INTEGER ||
postToken.getLexeme().equals("true") || postToken.getLexeme().equals("false")){
                     outputforTrio.append("<Statement> -> <Print>\n" + "<Print> -> put(<Expression>);\n");
                  }
               } else if(preToken.getLexeme().equals("get")) {
                  if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER){
                     outputforTrio.append("<Statement> -> <Scan>\n" + "   <Scan> -> get(<IDs>);");
                  }
               } else if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER &&
postToken.getToken() == Compiler_LA.Type.IDENTIFIER) {
                  outputforTrio.append("<Factor> -> <Primary>\n" + "<Primary> -> <Identifier>(<IDs>)\n");
               } else if(preToken.getLexeme().equals("+") || preToken.getLexeme().equals("-") ||
preToken.getLexeme().equals("*") || preToken.getLexeme().equals("/")) {
                  outputforTrio.append("<Factor> -> <Primary>\n" + "<Primary> -> (<Expression>)\n");
               } else if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER) {
                  if( postToken.getLexeme().equals(")")) {   //I BELIEVE THIS PART STILL NEEDS TO
BE ADJUSTED FOR FUNCTIONS THAT HAVE PARAMETERS
                     outputforTrio.append("<Function Definitions> -> <Function><Function Definitions'>\n"
+ "<Function> -> function <Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>\n");
                  }
               } else if(preToken.getLexeme().equals("while") || preToken.getLexeme().equals("if")) {
                  outputforTrio.append("<Condition> -> <Expression><Relop><Expression>\n");
               } else {
                  error(preToken, currToken);
               }

            } else if(currToken.getLexeme().equals(")")) {
               if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER || preToken.getToken() ==
Compiler_LA.Type.REAL || preToken.getToken() == Compiler_LA.Type.INTEGER ||
preToken.getLexeme().equals("true") || preToken.getLexeme().equals("false") ||
preToken.getLexeme().equals(")")){
                  outputforTrio.append("<Factor> -> <Primary>\n" + "<Primary> -> (<Expression>)\n");
               } else if(preToken.getLexeme().equals("int") || preToken.getLexeme().equals("boolean")||
preToken.getLexeme().equals("real") || preToken.getLexeme().equals("(")){
                  outputforTrio.append("<Function Definitions> -> <Function><Function Definitions'>\n" +
"<Function> -> function <Identifier> (<Opt Parameter List>)<Opt Declaration List><Body>\n");
               } else {
                  error(preToken, currToken);
               }
            } else if(currToken.getLexeme().equals("{")) {
               if(postToken.getToken() == Compiler_LA.Type.IDENTIFIER || postToken.getToken() ==
Compiler_LA.Type.KEYWORD  ) {
                  if(preToken.getLexeme().equals(")")) {
                     outputforTrio.append("<Function> -> function<Identifier> (<Opt Parameter List>)<Opt
Declaration List><Body>\n" + "<Body> -> {<Statement List>}\n");
```

```
                } else {
                    outputforTrio.append("<Statement> -> <Compound>\n" + "<Compound> -> {<Statement
List>}\n");
                }
            } else {
                error(preToken, currToken);
            }
        } else if(currToken.getLexeme().equals("}")) {
            if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER || preToken.getToken() ==
Compiler_LA.Type.SEPARATOR || preToken.getLexeme().equals("ifend") ||
preToken.getLexeme().equals("whileend")) {
                outputforTrio.append("<Statement> -> <Compound>\n" + "<Compound> -> {<Statement
List>}\n");
            } else {
                error(preToken, currToken);
            }
        } else if(currToken.getLexeme().equals(";")) {   //NOT SURE HOW TO DO THIS
            if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER || preToken.getToken() ==
Compiler_LA.Type.REAL || preToken.getToken() == Compiler_LA.Type.INTEGER ||
preToken.getLexeme().equals("true") || preToken.getLexeme().equals("false")) {
                outputforTrio.append("<Statement> -> <Assign>\n" + "<Assign> ->
<Identifier>=<Expression>;\n");
            } else if(preToken.getLexeme().equals(")")) {
                outputforTrio.append("<Statement> -> <Print>\n" + "<Print> -> put(<Expression>);\n");
            } else {
                error(preToken, currToken);
            }
        } else if(currToken.getLexeme().equals(":")) {
            if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER &&
(postToken.getLexeme().equals("boolean") || postToken.getLexeme().equals("int") ||
postToken.getLexeme().equals("real"))) {
                outputforTrio.append("<Parameter List> -> <Parameter><Parameter List'>\n" +
"<Parameter> -> <IDs>:<Qualifier>\n");
            } else {
                error(preToken, currToken);
            }
        } else if(currToken.getLexeme().equals(",")) {
            if(preToken.getToken() == Compiler_LA.Type.IDENTIFIER || postToken.getToken() ==
Compiler_LA.Type.IDENTIFIER) {
                outputforTrio.append("<IDs> -> <Identifier><IDs'>\n" + "<IDs'> -> ,<IDs>\n");
            } else {
                error(preToken, currToken);
            }

        } else if(currToken.getLexeme().equals("$$")) {  //need to check for end and begin
            outputforTrio.append("<Rat18F> -> <Opt Function Definitions> $$ <Opt Declaration List>
<Statement List> $$\n");
        }
        break;
      case INVALID:
        error(preToken, currToken);
        break;
    }
```

```
        return outputforTrio.toString();
    }

    //utility function that inits a hashmap for error printing of expected lexeme using follow sets
    public void initHashMap() {
        expectedToken.put(Compiler_LA.Type.IDENTIFIER, " (, ), +, -, /, *, :, ;, >, <, ==, ^=, =>, =< or , ");
        expectedToken.put(Compiler_LA.Type.KEYWORD, " IDENTIFIER, (, $$, INTEGER, REAL, ) or
,");
        expectedToken.put(Compiler_LA.Type.INTEGER, " ), +, -, /, *, ;, >, <, ==, ^=, =>, =< or , ");
        expectedToken.put(Compiler_LA.Type.REAL, " ), +, -, /, *, ;, >, <, ==, ^=, =>, =< or , ");
        expectedToken.put(Compiler_LA.Type.OPERATOR, " IDENTIFIER, (, INTEGER, REAL, true,
false ");
        expectedToken.put(Compiler_LA.Type.SEPARATOR, "IDENTIFIER, INTEGER, REAL,
KEYWORD, $$ ");
    }

    //adds an error print to any lexeme that could not fit the production rules applied
    private void error(Compiler_LA.Token preToken, Compiler_LA.Token errToken) {

        // print error
        outputforTrio.append("ERROR: at line " + errToken.getLineNumber() +" Expected " +
expectedToken.get(preToken.getToken()) + "but got token: " + errToken.getToken() + " " +
errToken.getLexeme() + "\n");

       errors++; // increment error counter
    }
}
```

All of these classes have not been changed since the last source listing was submitted

FSMInteger.java >
SAME CODE AS BEFORE

FSMReal.java >
SAME CODE AS BEFORE

FSMIdentifier.java >
SAME CODE AS BEFORE

Compiler_LA.java >
SAME CODE AS BEFORE