Compiler Theory

The 3rd Program Assignment – Scanner Construction

21900628 – Sechang Jang

Index

BNF (Backus-Naur Form)

PROGRAM -> program Identifier BLOCK

BLOCK -> begin STATEMENT end

STATEMENT -> ASSIGNMENT_STATEMENT ; | DECLARATIVE_STATEMENT ; | PRINT_STATEMENT | STATEMENET

ASSIGNMENT_STATEMENT -> Identifier = ASSIGNED | ASSIGNMENT_STATEMENT, Identifier = ASSIGNED

ASSIGNED -> NUMORID ARITHOP ASSIGNED | NUMORID

NUMORID -> number | Identifier

TYPE -> int

DECLARATIVE_STATEMENT -> TYPE ASSIGNMENT_STATEMENT

PRINT -> print_line (“ STRING ”) ;

STRING -> letter | letter STRING

NUMBER -> digit | digit NUMBER

ARITHOP -> *

Attribute Grammar

PROGRAM -> program Identifier BLOCK

BLOCK -> begin STATEMENT end

STATEMENT -> ASSIGNMENT_STATEMENT ; | DECLARATIVE_STATEMENT ; | PRINT_STATEMENT | STATEMENET

ASSIGNMENT STATEMENT -> Identifier.val = ASSIGNED.val | ASSIGNMENT_STATEMENT, Identifier.val = ASSIGNE.val

ASSIGNED.val -> NUMORID.val * ASSIGNED.val | NUMORID.val

NUMORID.val -> number.val | Identifier.val

TYPE.val -> int

DECLARATIVE_STATEMENT -> TYPE.val + ASSIGNMENT_STATEMENT

PRINT -> print_line (“ STRING.val ”) ;

STRING.val -> letter.lexeme

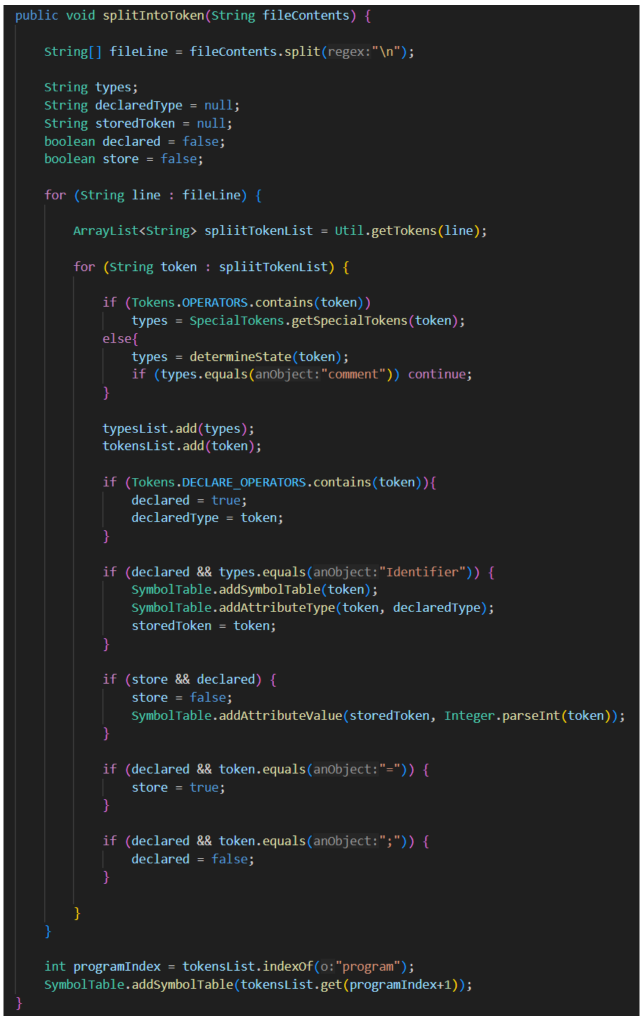
STRING.val -> letter.lexeme + STRING.val

NUMBER.val -> digit.lexeme | digit NUMBER

NUMBER.val -> digit.lexeme + NUMBER.val

ARITHOP.lexeme -> *

Explanation of the Source Code about How to Obtain the Result Based on Attribute Semantic Rules



```java
public void splitIntoToken(String fileContents) {

    String[] fileLine = fileContents.split(regex:"\n");

    String types;
    String declaredType = null;
    String storedToken = null;
    boolean declared = false;
    boolean store = false;

    for (String line : fileLine) {

        ArrayList<String> spliitTokenList = Util.getTokens(line);

        for (String token : spliitTokenList) {

            if (Tokens.OPERATORS.contains(token))
                types = SpecialTokens.getSpecialTokens(token);
            else{
                types = determineState(token);
                if (types.equals(anObject:"comment")) continue;
            }

            typesList.add(types);
            tokensList.add(token);

            if (Tokens.DECLARE_OPERATORS.contains(token)){
                declared = true;
                declaredType = token;
            }

            if (declared && types.equals(anObject:"Identifier")) {
                SymbolTable.addSymbolTable(token);
                SymbolTable.addAttributeType(token, declaredType);
                storedToken = token;
            }

            if (store && declared) {
                store = false;
                SymbolTable.addAttributeValue(storedToken, Integer.parseInt(token));
            }

            if (declared && token.equals(anObject:"=")) {
                store = true;
            }

            if (declared && token.equals(anObject:";")) {
                declared = false;
            }

        }
    }

    int programIndex = tokensList.indexOf(o:"program");
    SymbolTable.addSymbolTable(tokensList.get(programIndex+1));
}
```

The splitIntoToken method processes a string representing the contents of a file. It tokenizes each line of the file, categorizes each token, and updates a symbol table with information about declared identifiers and their types and values.

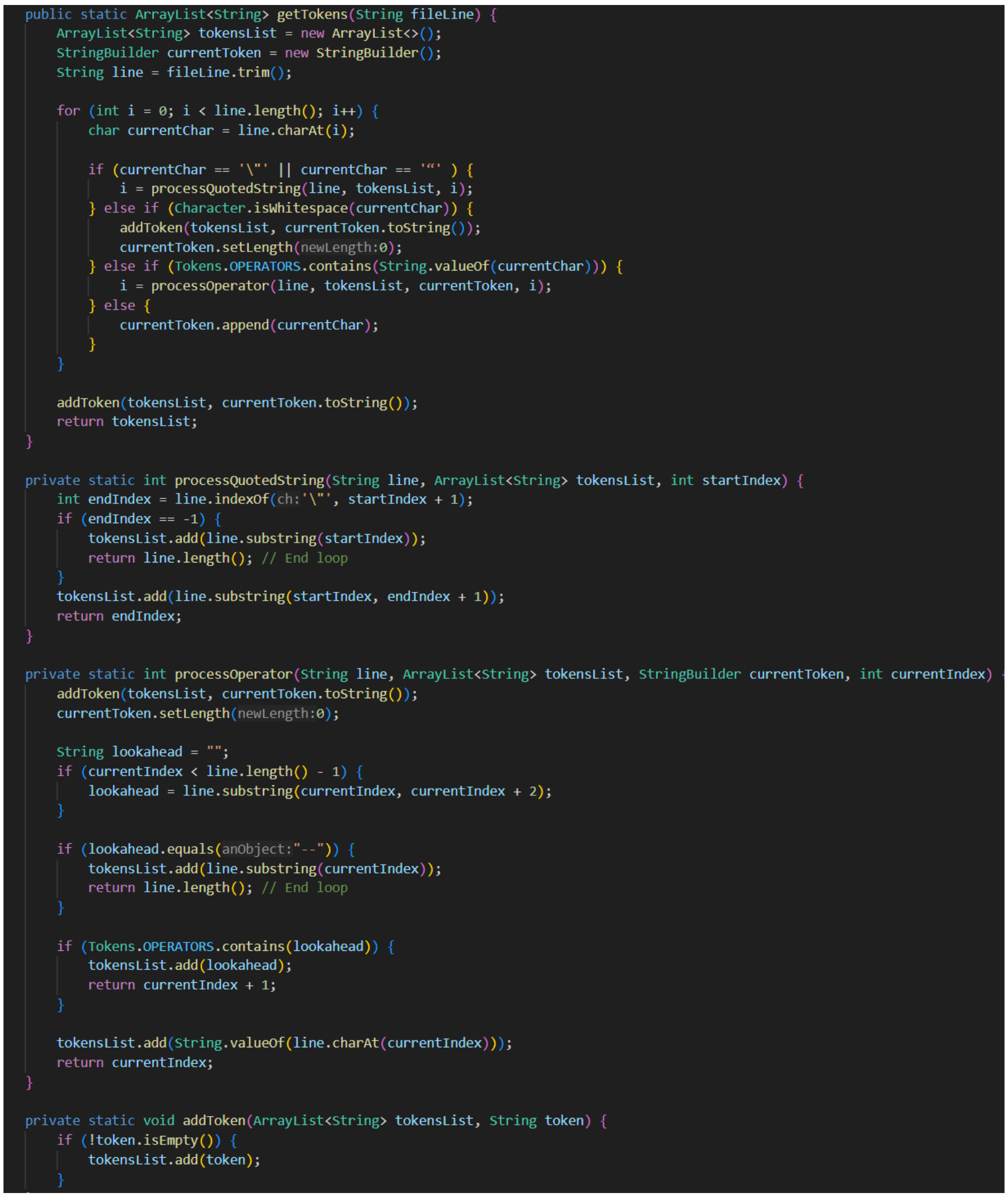
types: The type of the current token.

declaredType: The type of a declared variable (e.g., int).

storedToken: The identifier token being processed.

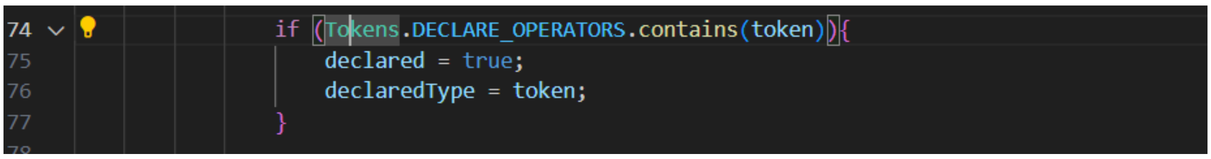
declared: A flag indicating if a declaration is being processed.

store: A flag indicating if the next token should be stored as a value.



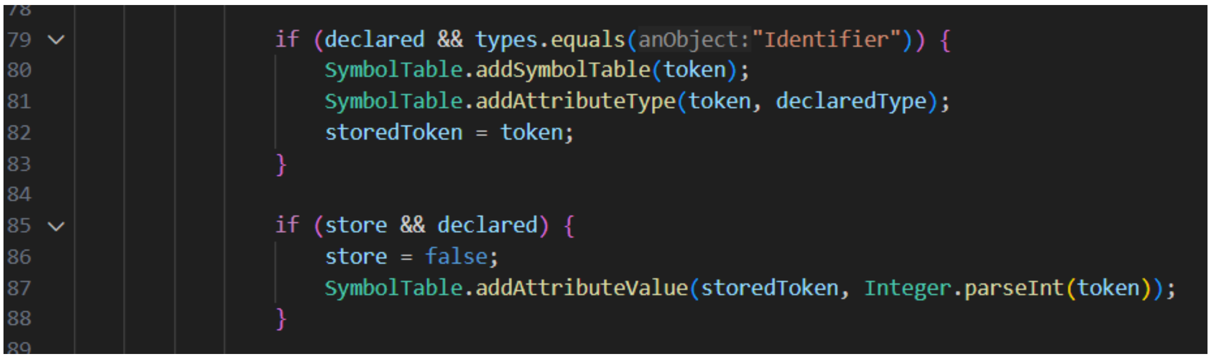
```java
public static ArrayList<String> getTokens(String fileLine) {
    ArrayList<String> tokensList = new ArrayList<>();
    StringBuilder currentToken = new StringBuilder();
    String line = fileLine.trim();

    for (int i = 0; i < line.length(); i++) {
        char currentChar = line.charAt(i);

        if (currentChar == '\"' || currentChar == '“' ) {
            i = processQuotedString(line, tokensList, i);
        } else if (Character.isWhitespace(currentChar)) {
            addToken(tokensList, currentToken.toString());
            currentToken.setLength(newLength:0);
        } else if (Tokens.OPERATORS.contains(String.valueOf(currentChar))) {
            i = processOperator(line, tokensList, currentToken, i);
        } else {
            currentToken.append(currentChar);
        }
    }

    addToken(tokensList, currentToken.toString());
    return tokensList;
}

private static int processQuotedString(String line, ArrayList<String> tokensList, int startIndex) {
    int endIndex = line.indexOf(ch:'\"', startIndex + 1);
    if (endIndex == -1) {
        tokensList.add(line.substring(startIndex));
        return line.length(); // End loop
    }
    tokensList.add(line.substring(startIndex, endIndex + 1));
    return endIndex;
}

private static int processOperator(String line, ArrayList<String> tokensList, StringBuilder currentToken, int currentIndex)
    addToken(tokensList, currentToken.toString());
    currentToken.setLength(newLength:0);

    String lookahead = "";
    if (currentIndex < line.length() - 1) {
        lookahead = line.substring(currentIndex, currentIndex + 2);
    }

    if (lookahead.equals(anObject:"--")) {
        tokensList.add(line.substring(currentIndex));
        return line.length(); // End loop
    }

    if (Tokens.OPERATORS.contains(lookahead)) {
        tokensList.add(lookahead);
        return currentIndex + 1;
    }

    tokensList.add(String.valueOf(line.charAt(currentIndex)));
    return currentIndex;
}

private static void addToken(ArrayList<String> tokensList, String token) {
    if (!token.isEmpty()) {
        tokensList.add(token);
    }
```

The getTokens method takes a single line of text and splits it into individual tokens. Tokens can be identifiers, numbers, operators, or strings. The method handles whitespace, quotes, and operators, and adds the tokens to a list which it then returns.



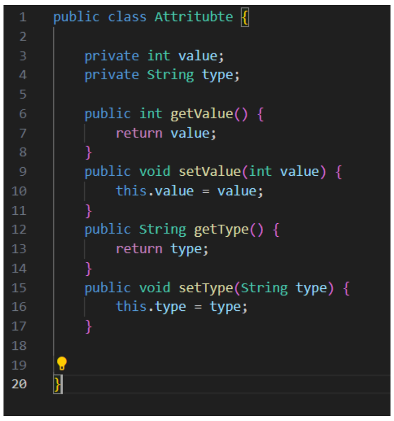
```java
if (Tokens.DECLARE_OPERATORS.contains(token)){
    declared = true;
    declaredType = token;
}
```

Context: This part of the code checks if the current token is a declaration operator (like int).
It sets the type value that is the Inherited Attributes.



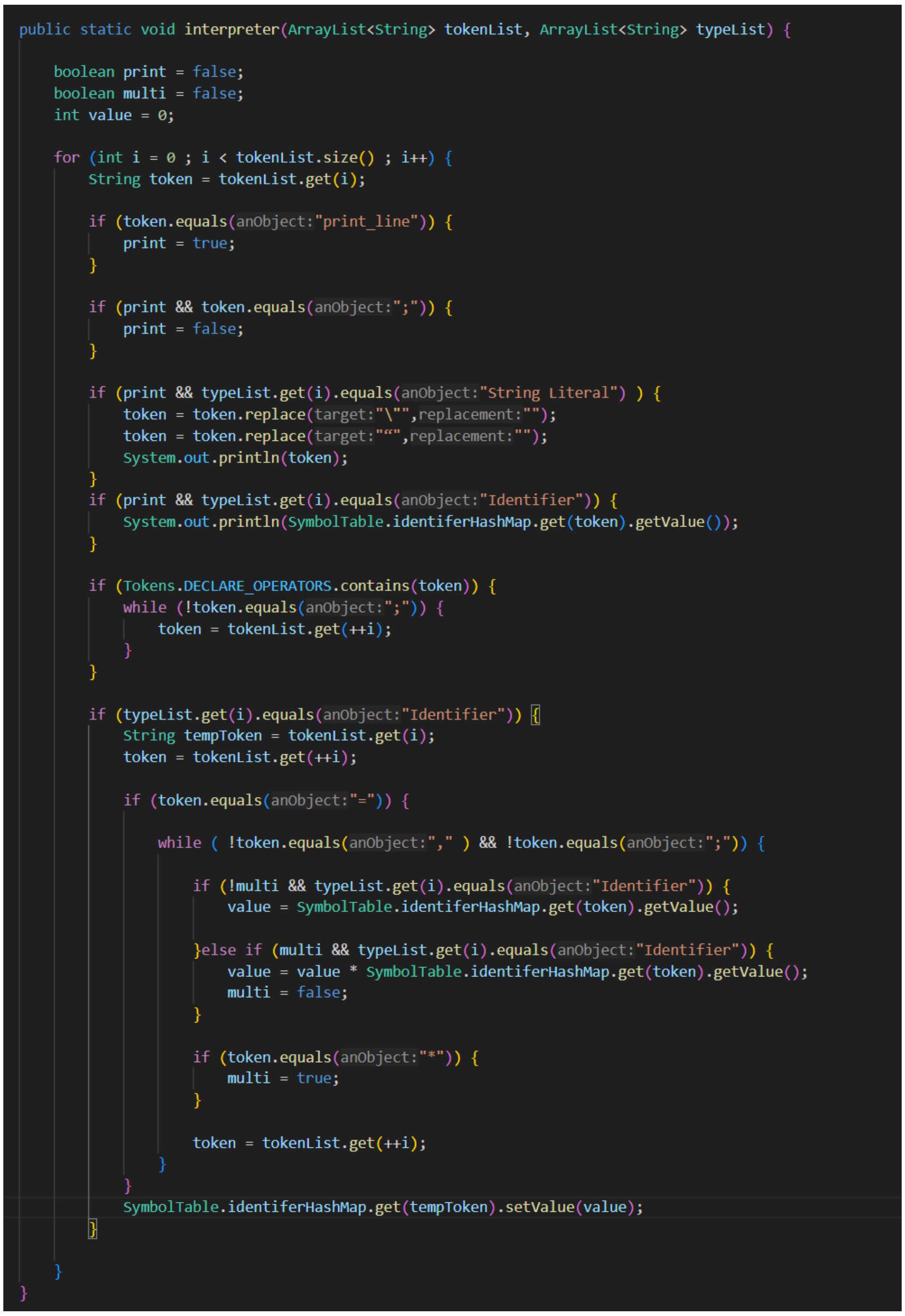
```
            if (declared && types.equals(anObject:"Identifier")) {
                SymbolTable.addSymbolTable(token);
                SymbolTable.addAttributeType(token, declaredType);
                storedToken = token;
            }

            if (store && declared) {
                store = false;
                SymbolTable.addAttributeValue(storedToken, Integer.parseInt(token));
            }
```

Context: If a declaration has been detected (declared is true) and the current token is an identifier, it adds the identifier to the symbol table with its type(inherited attribute). If an assignment is being processed (store is true) and it is part of a declaration (declared is true), the assigned value(synthesized attributes) is stored in the symbol table.



```
public class Attritubte {

    private int value;
    private String type;

    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }


}
```

Each Identifier has the attribute class to store its types and values.

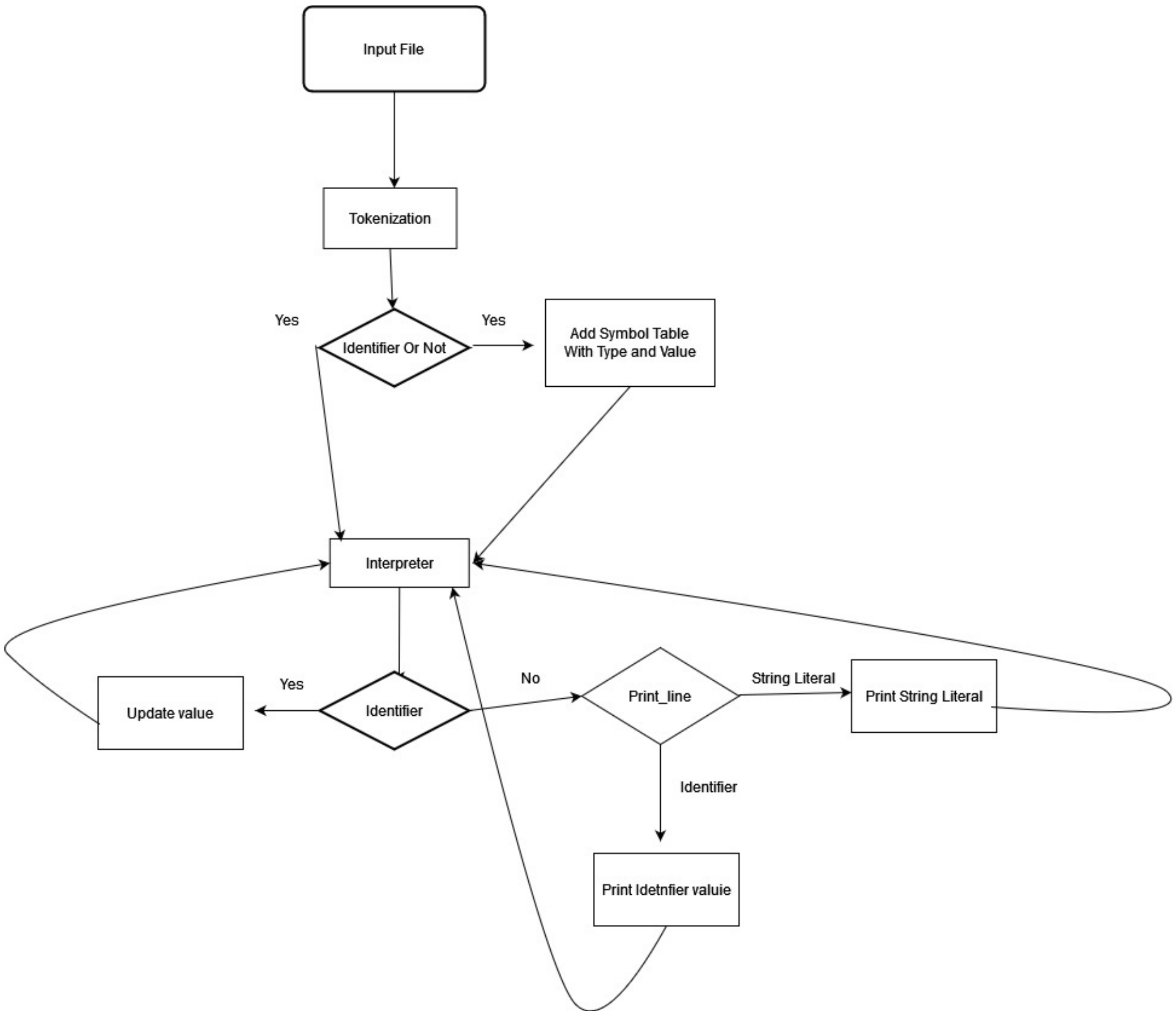


```java
public static void interpreter(ArrayList<String> tokenList, ArrayList<String> typeList) {

    boolean print = false;
    boolean multi = false;
    int value = 0;

    for (int i = 0 ; i < tokenList.size() ; i++) {
        String token = tokenList.get(i);

        if (token.equals(anObject:"print_line")) {
            print = true;
        }

        if (print && token.equals(anObject:";")) {
            print = false;
        }

        if (print && typeList.get(i).equals(anObject:"String Literal") ) {
            token = token.replace(target:"\"",replacement:"");
            token = token.replace(target:"“",replacement:"");
            System.out.println(token);
        }
        if (print && typeList.get(i).equals(anObject:"Identifier")) {
            System.out.println(SymbolTable.identiferHashMap.get(token).getValue());
        }

        if (Tokens.DECLARE_OPERATORS.contains(token)) {
            while (!token.equals(anObject:";")) {
                token = tokenList.get(++i);
            }
        }

        if (typeList.get(i).equals(anObject:"Identifier")) {
            String tempToken = tokenList.get(i);
            token = tokenList.get(++i);

            if (token.equals(anObject:"=")) {

                while ( !token.equals(anObject:"," ) && !token.equals(anObject:";")) {

                    if (!multi && typeList.get(i).equals(anObject:"Identifier")) {
                        value = SymbolTable.identiferHashMap.get(token).getValue();

                    }else if (multi && typeList.get(i).equals(anObject:"Identifier")) {
                        value = value * SymbolTable.identiferHashMap.get(token).getValue();
                        multi = false;
                    }

                    if (token.equals(anObject:"*")) {
                        multi = true;
                    }

                    token = tokenList.get(++i);
                }
            }
            SymbolTable.identiferHashMap.get(tempToken).setValue(value);
        }

    }
}
```

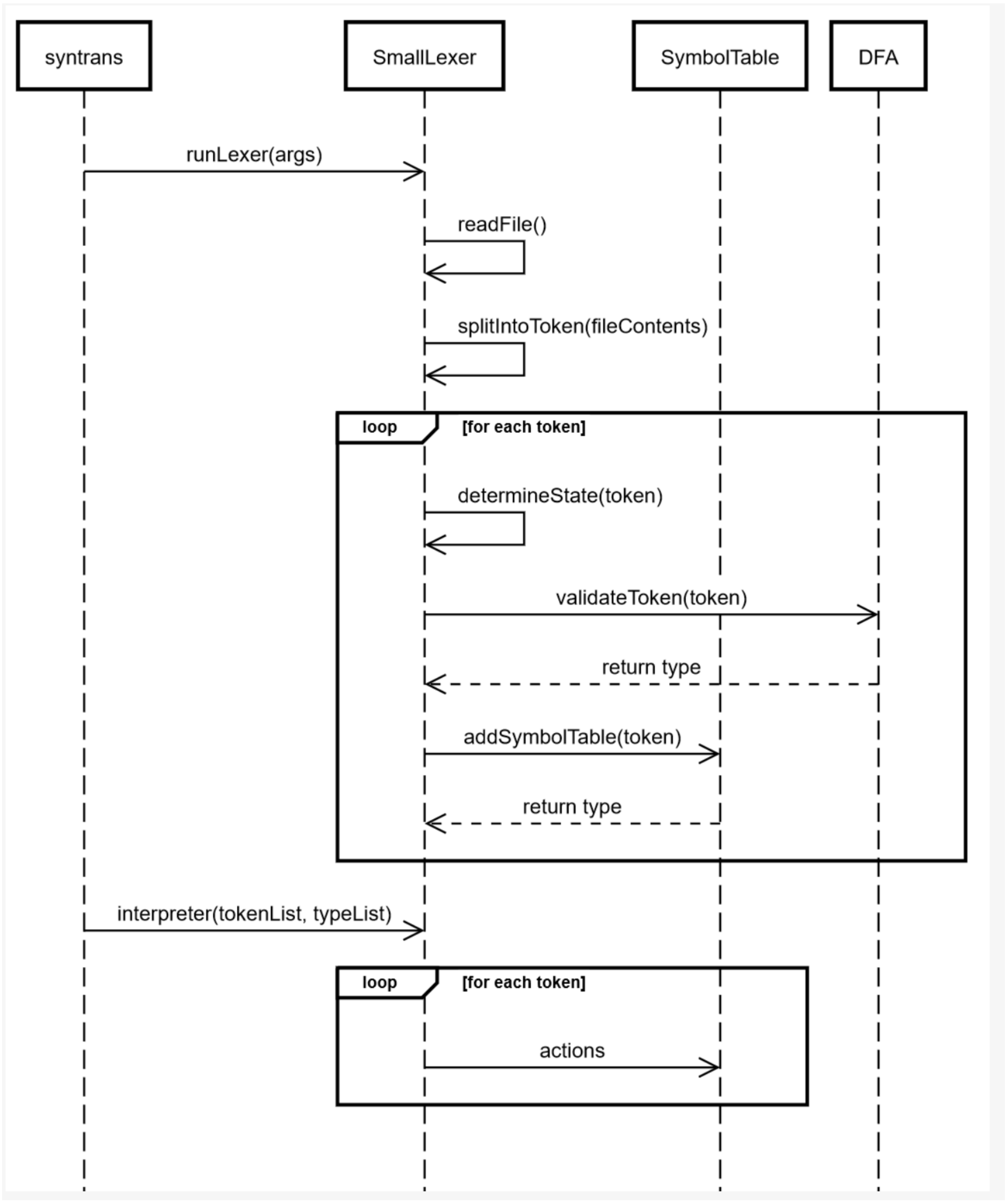
When an identifier is found, it checks if it is part of an assignment (=). Iterates over the tokens within the assignment statement until a comma (,) or semicolon (;) is found. If an identifier is found and multi is false, it retrieves the identifier's value from the symbol table. If multi is true, it multiplies the current value with the identifier's value. If a multiplication operator (*) is found, it sets the multi flag to true. After processing the assignment, it updates the identifier's value in the symbol table. It calls the actual value of the Identifier.

Diagram

Flow Chart



Sequence Diagram


syntrans
SmallLexer
SymbolTable
DFA
runLexer(args)
readFile()
splitIntoToken(fileContents)
loop
[for each token]
determineState(token)
validateToken(token)
return type
addSymbolTable(token)
return type
interpreter(tokenList, typeList)
loop
[for each token]
actions