

<https://github.com/RazvanAndreiMoga/LFTC>

```
/**
 * This is the constructor and here we initialize the symbolTable, the pif and the filePath
 * @param filePath - represents the filePath from the while from where we are going to read the program
 */
public MyScanner(String filePath) {
    this.filePath = filePath;
    this.symbolTable = new SymbolTable(100);
    this.pif = new ProgramInternalForm();
}
```

```
/**
 * In this method we read the content of the file and replace the tabs with ""
 * @return - We return the content of the read file
 * @throws FileNotFoundException if the file doesn't exist
 */
private String readFile() throws FileNotFoundException {
    StringBuilder fileContent = new StringBuilder();
    Scanner scanner = new Scanner(new File(this.filePath));
    while(scanner.hasNextLine()){
        fileContent.append(scanner.nextLine()).append("\n");
    }

    return fileContent.toString().replace("\t", "");
}
```

```
/**
 * This method is a wrapper for the real tokenize method, which prepares the array for the
 * real process of splitting the tokens.
 * In this method, we call the method for reading the content of the file, we concatenate the
separators into a
 * simple string, we use that string to split the program into a list of string where we have stored
the tokens +
 * identifiers + constants + the separators from the created string. In the end, the tokenize method
is called,
 * method which will create a List of pair which contains the token/idenfitier/constant + the number
of the line
 * on which it was placed.
 * @return - the list of pairs composed of tokens/identifiers/constants + a pair which is composed
from the number of the line and the number of column on which them were placed
 */
private List<Pair<String, Pair<Integer, Integer>>> createListOfProgramsElems(){
```

```
/**
 * Within this method, we go through each string from tokensToBe and look in what case are we:
 * We can have 4 cases:
 * 1) the case when we are managing a string
 * -- a) where we are either at the start of the string and we start to create it
 * -- b) we found the end of the string so we add it to our final list + the line and the column on
which it is situated
 * 2) the case when we are managing a char
 * -- a) where we are either at the start of the char and we start to create it
 * -- b) we found the end of the char so we add it to our final list + the line and the column on
which it is situated
 * 3) the case when we have a new line
 * -- we simply increase the line number in this case
 * -- we make the number column 1 again because we start a new line
 * 4) the case when:
 * -- a) if we have a string, we keep compute the string
 * -- b) if we have a char, we compute the char
 * -- c) if the token is different from " " (space) it means we found a token and we add it to our
final list + the line and the column on which it is situated and we increase the column number
```

```
    * In this method we go through the elements of the program and for each of them, if they compose a
    token/identifier/constant we add it
    * to the final list and we compute also the line number on which each of the are situated. (we
    somehow tokenize the elems which compose the program)
    * @param tokensToBe - the List of program elements (strings) + the separators
    * @return - the list of pairs composed of tokens/identifiers/constants + a pair which is composed
    from the number of the line and the number of column on which them were placed
    */
    private List<Pair<String, Pair<Integer, Integer>>> tokenize(List<String> tokensToBe){
```

```
/**
 * In this method, we scan the list of created tokens and we classify each of them in a category:
 * a) 2 - for reservedWords
 * b) 3 - for operators
 * c) 4 - for separators
 * d) 0 - for constants
 * e) 1 - for identifiers
 * If the token is a constant or an identifier we add it to the Symbol Table
 * After figuring out the category, we add them to the ProgramInternalForm + their position in the
symbol table ( (-1, -1) for anything that is not a constant and an identifier ) + their category (0, 1, 2,
3, 4)
 * If the token is not in any of the categories, we print a message with the line and the column of
the error + the token which is invalid.
 */

public void scan(){
```

```
/**
```

```
 *
```

```
 * @return the ProgramInternalForm
```

```
 */
```

```
public ProgramInternalForm getPif(){  
    return this.pif;  
}
```

```
/**
```

```
 *
```

```
 * @return the SymbolTable
```

```
 */
```

```
public SymbolTable getSymbolTable() {  
    return this.symbolTable;  
}
```

```

/**
 * We initialize the two lists from the class
 */
public ProgramInternalForm() {
    this.tokenPositionPair = new ArrayList<>();
    this.types = new ArrayList<>();
}

/**
 * We add a token/identifier/constant to its list + their position in the symbol table and we also add
the category in the list of types
 * @param pair - Is a pair which is composed of the token/constant/idenfitier + its position in the
symbol table
 * @param type - The category of the token (2, 3, 4) or constant (0) or identifier (1)
 */
public void add(Pair<String, Pair<Integer, Integer>> pair, Integer type){
    this.tokenPositionPair.add(pair);
    this.types.add(type);
}

```