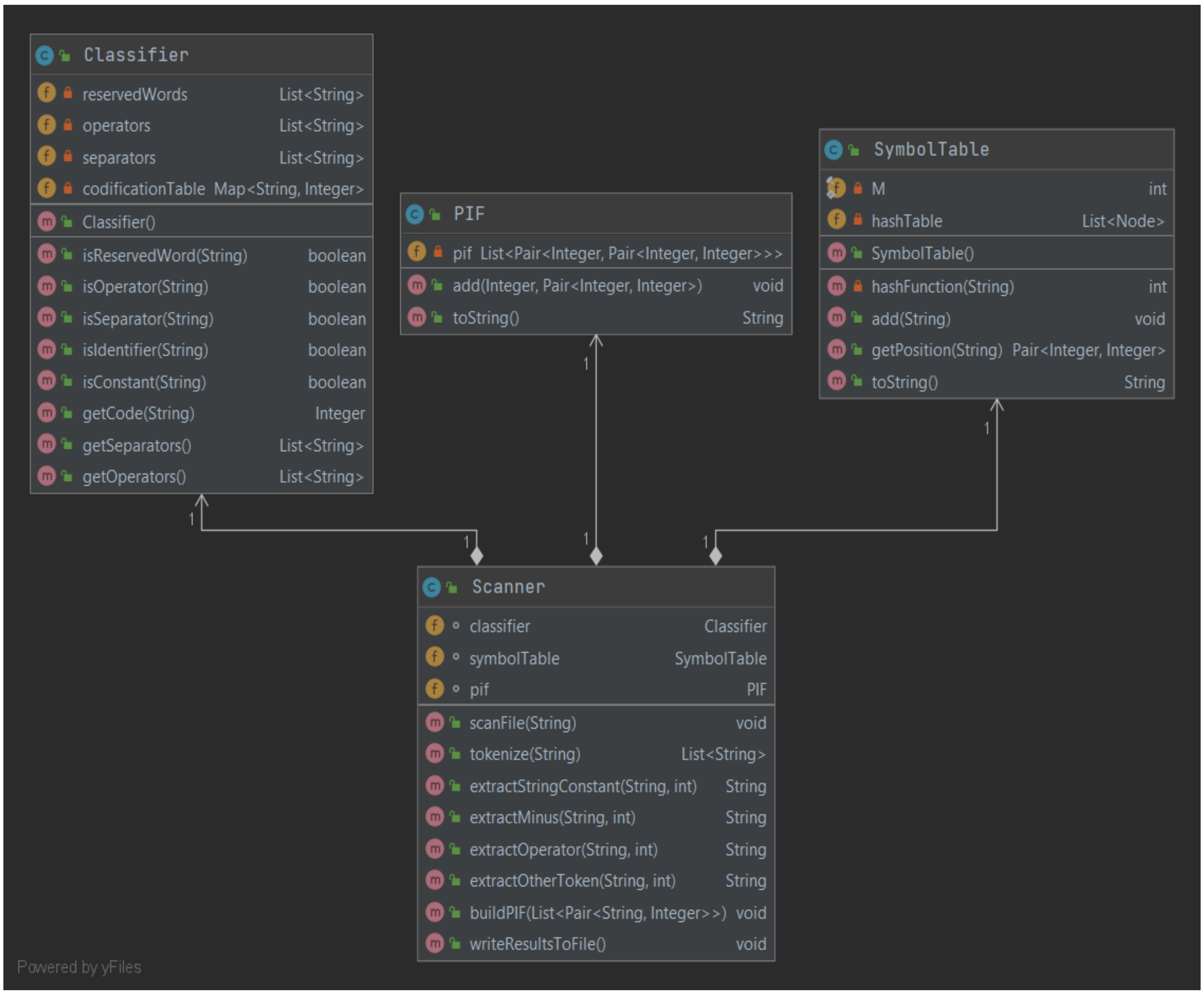
**Requirement:** Implement a scanner (lexical analyzer): Implement the scanning algorithm and use ST from lab 2 for the symbol table.

**Details:**

- ST.out should give information about the data structure used in representation
- If there exists an error the program should give a description and the location (line and token)

## Analysis and Design:



Classifier
reservedWords List<String>
operators List<String>
separators List<String>
codificationTable Map<String, Integer>
Classifier()
isReservedWord(String) boolean
isOperator(String) boolean
isSeparator(String) boolean
isIdentifier(String) boolean
isConstant(String) boolean
getCode(String) Integer
getSeparators() List<String>
getOperators() List<String>
PIF
pif List<Pair<Integer, Pair<Integer, Integer>>>
add(Integer, Pair<Integer, Integer>) void
toString() String
SymbolTable
M int
hashTable List<Node>
SymbolTable()
hashFunction(String) int
add(String) void
getPosition(String) Pair<Integer, Integer>
toString() String
Scanner
classifier Classifier
symbolTable SymbolTable
pif PIF
scanFile(String) void
tokenize(String) List<String>
extractStringConstant(String, int) String
extractMinus(String, int) String
extractOperator(String, int) String
extractOtherToken(String, int) String
buildPIF(List<Pair<String, Integer>>) void
writeResultsToFile() void
1
1
1
1
1
1
Powered by yFiles

- **Scanner** – it scans the source code, extract the tokens from it and creates the symbol table and the program internal form. The tokens are retrieved by scanning the file line by line, and then each line, character by character.
    - In the Scanner class, there are a Symbol Table, a PIF and a Classifier.
    - The methods are:

  - void scanFile(String fileName) - scans the file given as parameter line by line, creating the list of tokens from the file.

  - List<String> tokenize(String line) - returns

      – the list which contains all the tokens from the line given as parameter.

  - String extractStringConstant(String line, int currentPosition) - returns

- the substring that starts with the quote mark at the currentPosition and ends at the complementary quote mark or at the end of the line.

- String extractCharacterConstant(String line, int currentPosition) - returns

    - the substring that starts with the apostrophe at the currentPosition and ends at the complementary apostrophe or at the end of the line.

- String extractMinus(String line, int currentPosition) - returns

    - "-" if the minus sign at the currentPosition is an operator or
    - the numerical constant which the "-" is part of.

- String extractOperator(String line, int currentPosition) - returns

    - the found operator at the currentPosition(if it does not belong to a compound operator) or
    - the compound operator starting at currentPosition or
    - empty string if there is no operator at currentPosition.

- String extractOtherTokens(String line, int currentPosition) - returns

    - the token that starts at the currentPosition and ends at the first separator or operator found.

> The tokenize method scans the line using an index (the current position in that line). It analyses the character from the current position and depending on its type, it calls the methods that extract and return a token. This token is added in the list of tokens and the index of the current position will be placed immediately after that token in the line.

- boolean buildPIF(List<Pair<String, Integer>> tokens):

    - input: a list of pairs of tokens and the line of the source file at which that token appears

- returns: true if the source code has a lexical error, false otherwise.

> The method checks every token from the list as follows:

- if the token is keyword, separator or operator, it is added in PIF with the code from the codification table and the position (-1, -1)
- if the token is identifier or constant, it is added in the Symbol Table (if it has not been added already) and in PIF with the code 0 for identifier, 1 for constants and the position in the ST
- otherwise, it is invalid, and a message will be displayed.

- void writeResultsToFile(): writes the symbol table and the PIF in the corresponding output files.

- **Classifier** – it is used for classifying a token, which can be a reserved word, a separator, an operator, an identifier or a constant.
    - In the Classifier class, there are 3 lists, each of them containing the keywords, the separators and, respectively, the operators that appear in the language specifications.
    - It also contains a Map which is populated in the constructor and associates each atom with an integer number (code) - the codification table.
    - The methods used for classification are:

- boolean isReservedWord(String token) - returns

– true if the token given as parameter is a reserved word
– false otherwise.

- boolean isOperator(String token) -

– returns true if the token given as parameter is an operator
– false otherwise.

- boolean isSeparator(String token) - returns

– true if the token given as parameter is a separator
– false otherwise.

> For these 3 methods, the checking is done by verifying if the token appears in the list of corresponding atoms.

- boolean isIdentifier(String token) - returns

– true if the token is and identifier (respects the rule for identifiers)
– false otherwise.

- boolean isConstant(String token) - returns

– true if the token is constant (numeric, character or string)
– false otherwise.

> The last 2 methods check the matching of the token with a regex expression.

o Other methods:

- int getCode(String token) - returns

– the code of the given token from the codification table.

- **PIF** – the class for the Program Internal Form entity. It is represented as a list of Pairs of the token code and their position in the Symbol Table (the position is (-1, -1) if the token is a keyword, a separator or an operator and thus, it does not appear in the symbol table).

  o It has only one method:

  - void add(Integer code, Pair<Integer, Integer> position): adds in the program internal form the pair (code, position).

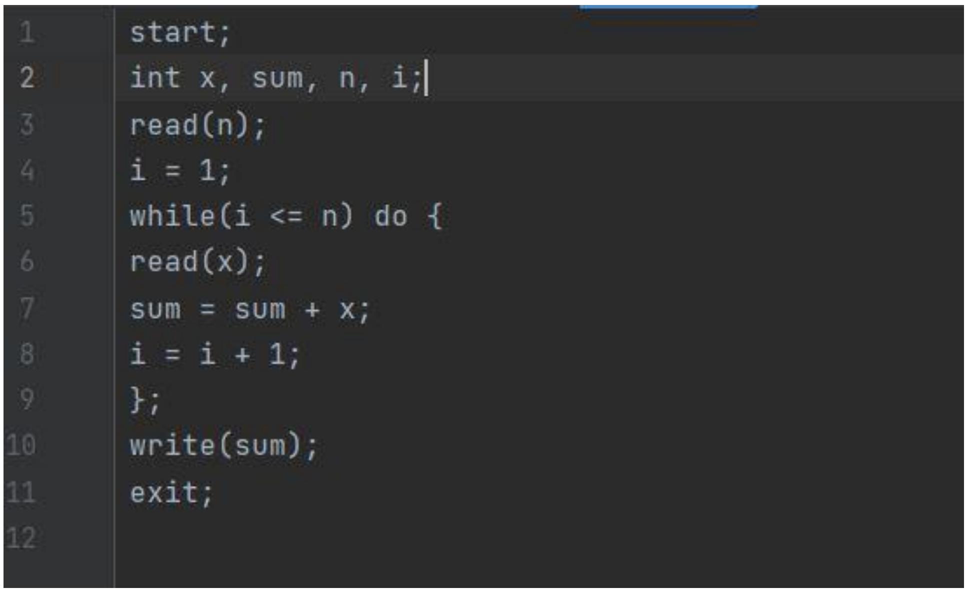
- **Symbol Table** – represented as a hash table, for which the collisions are solved by using a linked list. The hash function is the sum of the ascii code of the characters from the token, modulo M.

## Implementation:

https://github.com/LaviniaGalan/FLCD/tree/master/Lab3

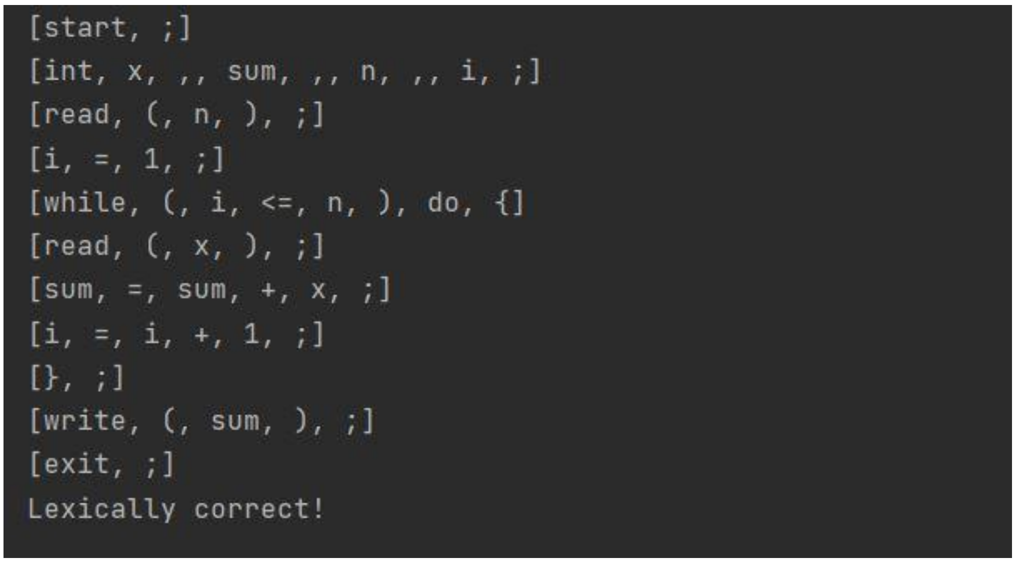
## Test:

For a lexically correct program:



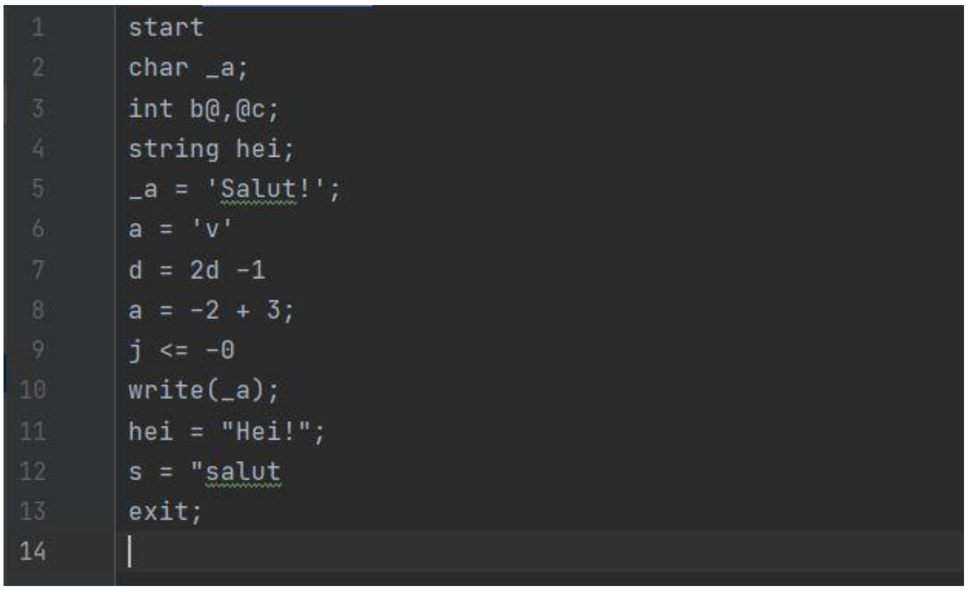
```
start;
int x, sum, n, i;
read(n);
i = 1;
while(i <= n) do {
read(x);
sum = sum + x;
i = i + 1;
};
write(sum);
exit;

```

The tokens from each line and the message are:



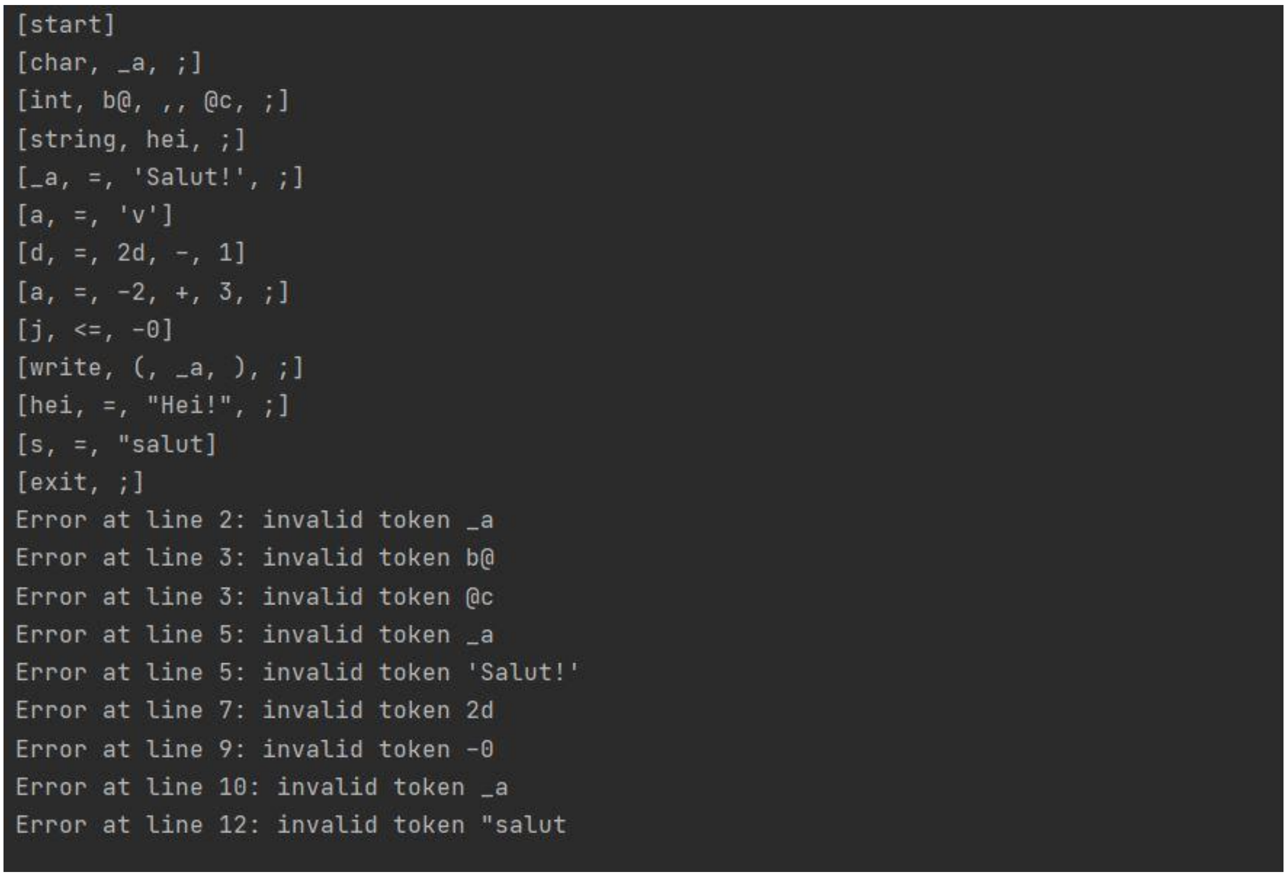
```
[start, ;]
[int, x, ,, sum, ,, n, ,, i, ;]
[read, (, n, ), ;]
[i, =, 1, ;]
[while, (, i, <=, n, ), do, {]
[read, (, x, ), ;]
[sum, =, sum, +, x, ;]
[i, =, i, +, 1, ;]
[}, ;]
[write, (, sum, ), ;]
[exit, ;]
Lexically correct!
```

For a program that contains lexical errors:



```
start
char _a;
int b@,@c;
string hei;
_a = 'Salut!';
a = 'v'
d = 2d -1
a = -2 + 3;
j <= -0
write(_a);
hei = "Hei!";
s = "salut
exit;

```

The list of tokens and the message are:



```
[start]
[char, _a, ;]
[int, b@, ,, @c, ;]
[string, hei, ;]
[_a, =, 'Salut!', ;]
[a, =, 'v']
[d, =, 2d, -, 1]
[a, =, -2, +, 3, ;]
[j, <=, -0]
[write, (, _a, ), ;]
[hei, =, "Hei!", ;]
[s, =, "salut]
[exit, ;]
Error at line 2: invalid token _a
Error at line 3: invalid token b@
Error at line 3: invalid token @c
Error at line 5: invalid token _a
Error at line 5: invalid token 'Salut!'
Error at line 7: invalid token 2d
Error at line 9: invalid token -0
Error at line 10: invalid token _a
Error at line 12: invalid token "salut
```