Cheran Bianca Paula, group 932/1

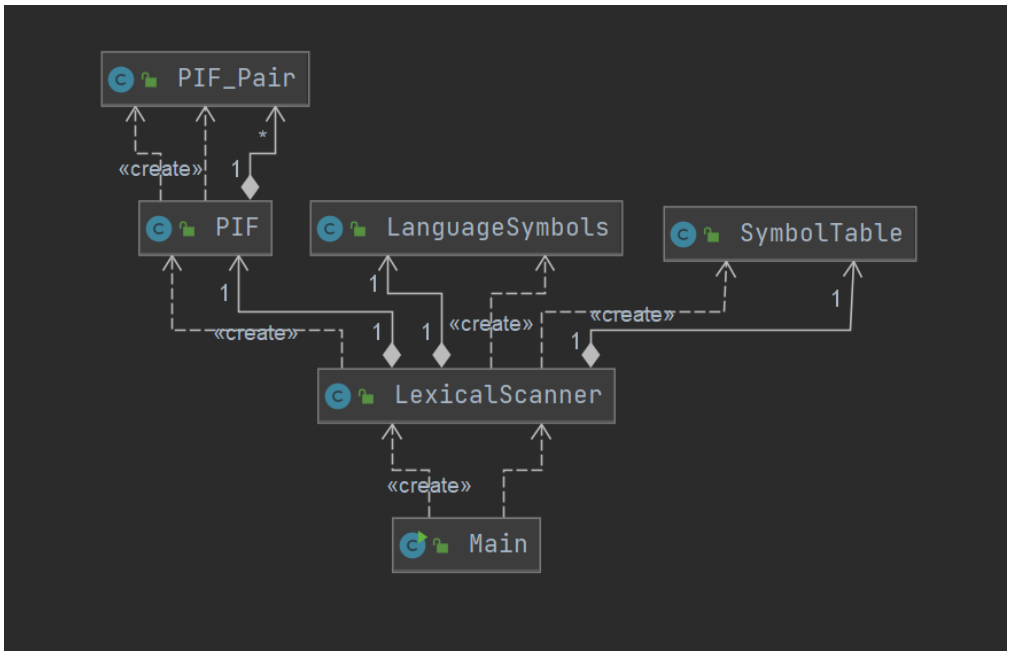
Lab 3 – Scanner (Lexical Analyzer)

Link to GitHub code: https://github.com/bianca-paula/Formal-Languages-and-Compiler-Design/tree/main/Lab3%20-%20Scanner

Statement: Implement a scanner (lexical analyzer): Implement the scanning algorithm and use ST from lab 2 for the symbol table.
Input: Programs p1/p2/p3/p1err and token.in (see Lab 1a)
Output: PIF.out, ST.out, message "lexically correct" or "lexical error + location"

Solution:

The class diagram of the implementation is:



The SymbolTable uses the HashTable implemented in the second laboratory.

Tokens can be classified into: • Identifiers • Constants • Reserved words (keywords) • Separators • Operators**.** In the **Symbol Table**, we are going to add only identifiers and constants.

For the **PIF (Program Internal Form)**, I have defined two classes: PIF and PIF_Pair. A PIF_Pair consists of a (token, positionInST), where positionInST represents the position of the given token in the Symbol Table. If the element is not in the Symbol Table, that is it is an operator, separator or reserved word, its position on the ST will be -1. The PIF is kept as a Java List with elements of type PIF_Pair. It has the add and print functionalities implemented.

```java
public class PIF {
    public List<PIF_Pair> ProgramInternalForm;
    public PIF(){
        this.ProgramInternalForm = new ArrayList<>();
    }

    public void addToPIF(String token, Integer tokenPositionInST){
        PIF_Pair element = new PIF_Pair(token, tokenPositionInST);
        this.ProgramInternalForm.add(element);
    }

    public void printPIF(){
        for(PIF_Pair element : ProgramInternalForm){
            System.out.println(element.toString());
        }
    }
```

```java
public PIF_Pair(String token, Integer positionInST){
    this.token = token;
    this.positionInST=positionInST;
}
```

The Scanning algorithm is inspired by the algorithm from Course2:

INPUT: source program

OUTPUT: PIF + ST

While (not(eof)) do

detect(token);

if token is reserved word OR operator OR separator

then genPIF(token, 0)

else

if token is identifier OR constant

then index = pos(token, ST);

genPIF(token_type, index)

else message “Lexical error”

endif

endif

endwhile

The scanning algorithm is based on two methods: parseLine() and parseFile().

The parseLine() method takes as an input a line from the file, which is of type String, and splits it into tokens. For each token, it checks whether it is an identifier, constants, operator, separator or reserved word, and, in the case mentioned above, adds it to the Symbol Table if necessary and then to PIF. If a token cannot be classified, a lexical error message, along with the line where it is found, will be printed:

```
public void parseLine(String line) throws IOException, ParseException {
    this.languageSymbols.readLanguageSymbols();
    // remove all blank spaces
    String[] lineElements = line.split("\\s+");

    for(String element: lineElements) {
        if(element.length() > 0) {
            int i = 0;
            StringBuilder newString = new StringBuilder();

            while(i < element.length()) {
                if(isCharacterInAlphabet(element.charAt(i))) {
                    newString.append(element.charAt(i));
                    //System.out.println(element.charAt(i));
                }
                else {
                    if(newString.length() > 0) {
                        this.symbolTable.addElement(newString.toString());

this.ProgramInternalForm.addToPIF(newString.toString(),
this.symbolTable.searchElement(newString.toString()));
                    }
                    newString = new StringBuilder();

                    String newToken = Character.toString(element.charAt(i));
                    if(this.languageSymbols.operators.contains(newToken) ||
this.languageSymbols.separators.contains(newToken)) {

ProgramInternalForm.addToPIF(Character.toString(element.charAt(i)),this.symbo
lTable.searchElement(Character.toString(element.charAt(i))));
                    }
                    else {
                        String errorMessage="lexical error - at line:
"+getFileLineNumber(line);
                        throw new
ParseException(errorMessage,getFileLineNumber(line));
```

```
                    }

                }
                i++;
            }

            if (newString.length() > 0 &&
this.languageSymbols.reservedWords.contains(newString.toString())){
                this.ProgramInternalForm.addToPIF(newString.toString(),
this.symbolTable.searchElement(newString.toString()));
            }

            if (newString.length() > 0 &&
!(this.languageSymbols.reservedWords.contains(newString.toString()))) {
                    this.symbolTable.addElement(newString.toString());
                    this.ProgramInternalForm.addToPIF(newString.toString(),
this.symbolTable.searchElement(newString.toString()));
                }

        }
    }
}
```

The parseFile() method uses a Scanner to scan the program file, it goes through each line of the file as long as there are any lines left, and calls the parseLine() method for each line of the program in order to split each line into tokens. Finally, it uses a BufferedWriter in order to write the SymbolTable to the file ST.out and the PIF to the PIF.out file.

Tests:

For p1err.txt:

p1err.txt | LexicalScanner.java | Top-Level Package | PIF_Pair.java

```
declare
    var myArray: integer []
    var index: integer
    var minNumber: integer
enddeclare
program
    myArray := [400, 9, 2, 7, 9, 3, 8, 5]
    minNumber := myArray(0)
    for (index := 0, index < 7; index:=index+1)
        if (minNumber > myArray(index)
            minNumber := myArray(index)
        endif
    endfor
            write ($minNumber)
endprogram
```

We have the following output (there should be an error on line 14, ‘$’ is not a recognized symbol):

```
Main
"C:\Program Files\Java\jdk-15\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2020.2.2\lib\idea_rt.jar=49255:C:\Program
java.text.ParseException Create breakpoint : lexical error - at line: 14
    at LexicalScanner.parseLine(LexicalScanner.java:70)
    at LexicalScanner.parseFile(LexicalScanner.java:106)
    at Main.main(Main.java:10)

Process finished with exit code 0
```

For p1.txt:

```
p1.txt   p1err.txt   LexicalScanner.java   Top-Level Package   PIF_Pair.ja
1   declare
2       var myArray: integer []
3       var index: integer
4       var minNumber: integer
5   enddeclare
6   program
7       myArray := [400, 9, 2, 7, 9, 3, 8, 5]
8       minNumber := myArray(0)
9       for (index := 0, index < 7; index:=index+1)
10          if (minNumber > myArray(index)
11              minNumber := myArray(index)
12          endif
13      endfor
14              write (minNumber)
15  endprogram
16
```

We have the following ST.out:

```
ST.out   p1.txt   p1err.txt   LexicalScanner.ja
1   0   |   null
2   1   |   myArray
3   2   |   minNumber
4   3   |   null
5   4   |   null
6   5   |   null
7   6   |   null
8   7   |   null
9   8   |   400
10  9   |   0
11  10  |   2
12  11  |   3
13  12  |   1
14  13  |   5
15  14  |   null
16  15  |   7
17  16  |   index
18  17  |   9
19  18  |   8
20  19  |   null
21
```

And PIF.out:

```
declare,  -1
var,  -1
myArray,  1
:,  -1
integer,  -1
[,  -1
],  -1
var,  -1
index,  16
:,  -1
integer,  -1
var,  -1
minNumber,  2
:,  -1
integer,  -1
enddeclare,  -1
program,  -1
myArray,  1
:,  -1
=,  -1
[,  -1
400,  8
,,  -1
9,  17
,,  -1
2,  10
,,  -1
7,  15
,,  -1
9,  17
,,  -1
3,  11
,,  -1
8,  18
,,  -1
5,  13
],  -1
minNumber,  2
:,  -1
=,  -1
myArray,  1
(,  -1
0,  9
),  -1
for,  -1
(,  -1
index,  16
:,  -1
=,  -1
0,  9
```

```
,,  -1
index,  16
<,  -1
7,  15
;,  -1
index,  16
:,  -1
=,  -1
index,  16
+,  -1
1,  12
),  -1
if,  -1
(,  -1
minNumber,  2
>,  -1
myArray,  1
(,  -1
index,  16
),  -1
minNumber,  2
:,  -1
=,  -1
myArray,  1
(,  -1
index,  16
),  -1
endif,  -1
endfor,  -1
write,  -1
(,  -1
minNumber,  2
),  -1
endprogram,  -1
```

And the output:

```
Main
"C:\Program Files\Java\jdk-15\bin\java.exe" "-javaagent:C:\P
Lexically correct!

Process finished with exit code 0
```