Formal Languages and Compiler Design
Third laboratory – Documentation
Scanner – Lexical Analyzer

Link to Git: https://github.com/albcristi/formal-languages-and-compiler-design

The Scanner Class (MyScanner class in the program)

- Contains the paths to:
    - o *token.in* – file that stores all the language separators, reserved words and operators specific to our mini language
    - o file containing the source program
- The scanner uses the class Symbol Table in order to represent the ***symbol table*** that will be constructed during the lexical analysis process.
- In order to form the ***PIF*** an array containing of elements of form pair<String, pair<Int, Int>> will be formed. An element from the PIF will contain a pair, first element being the special token specific to the mini language or ID, in case of identifiers, or CONST, in case of constants (numbers, strings, Booleans).
- As the scanning happens, any type of lexical errors will be stored in a list and written in a file named ***lexical-errors.out*** in the ***lexical-analysis-out*** folder

Scanning process

for each line in source program

1. We split the line of code into tokens
    - 1.1 Split based on the space character
    - 1.2 Split the result from 1.1 based on the special language separators, reserved words and operators presented in the token.in file
    - 1.3 Taken the result from 1.2 we then apply special union operations that are used to reconstruct possible compound tokens that were destroyed during the splitting process from step 1.2:
        - 1.3.1 We reconstruct the possible tokens formed with “=”, namely the following cases: “>=”, “<=” and “==”
        - 1.3.2 We reconstruct the possible input/output compounded commands, namely the ones that were specially made for inti values, namely “sayInti” and “giveInti” from “say/give”+”Inti”
        - 1.3.3 We reconstruct the string constants that have been separated due to the spaces that were included in the string
        - 1.3.4 We reconstruct the constant integers that might have been separated from their sign

2 We start classifying the tokens:

2.1 Tokens that can be found in the token.in file

2.2 Tokens that represent identifiers and constants, those tokens have to pass the identifier/constant language specific format (we use the ***isConstant*** or ***isIdentifier*** methods)

3 We form the PIF and the Symbol Table and have the following cases:

3.1 for tokens identified at 2.1 we add in the PIF the following (token, (0,-1))

3.2 for tokens identified at 2.2 we search for their position in the symbol table:

3.2.1 if found in ST we add (CONST/ID, position in ST) to PIF

3.2.2 otherwise we add in the ST the token, and add to PIF (CONST/ID, position where added in ST)

3.3 for tokens that were not classified we add a new lexical error for an unclassified token and the line number to the list were we keep track of the errors that occurred during the lexical analysis

END FOR EACH

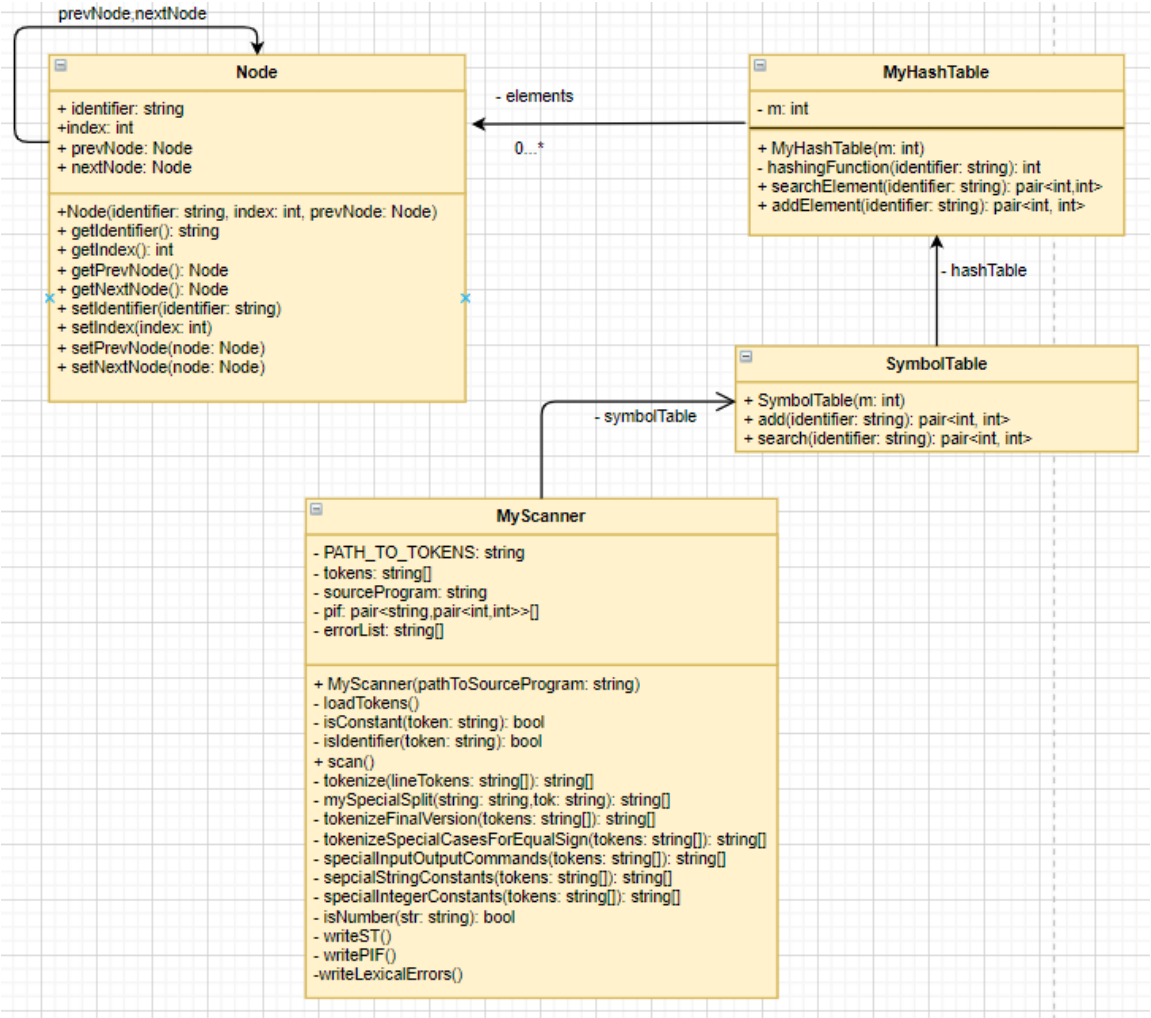
We now write the results of the scanning process in the ***lexical-analysis-out*** folder:

1. **pif.out** - file representing the formed PIF during the scanning process in the following format: on the first line we have the size of the PIF and after a line for each element having the index and the pair (string, (int, int))
2. **st.out** – file representing the formed symbol table formed during the scanning process having the format: on the first line is the size of the symbol table, after that a line for each element from the table with the index in the table and the list of corresponding elements of that table entry
3. **lexical-errors.out** – table where on the first line we have the number of errors that occurred during the scanning, on the following line a message that states if the program is lexically correct and followed on the next lines, if it is the case, each error per line

Class Diagram

-see next page-

The class diagram will also include the classes involved in the construction of the SymbolTable class, namely: Node and MyHashTable



Alb Dan-Cristian
931