Link to GitHub: <https://github.com/thediciman/FLCD/tree/master/Lab3>

Description:

The entry point of the program is in the **main** method from the **Main** class, where the **Scanner** class that I have implemented receives a filename, it scans the file and returns a pair containing the generated **SymbolTable** and **ProgramInternalForm** associated with the program from the input file, or it will throw an exception in case of a lexical error.

The Scanner class contains the static method **scan** which receives a filename, it creates an empty SymbolTable and ProgramInternalForm, and then proceeds with opening the file and reading, one by one, the lines from the file.

Each line that is being read is being first split into tokens by the **tokenize** method, which receives the line and returns a list of tokens. The **tokenize** method makes use of a rather long regex – more specifically, the regex includes the **compound operators** (i.e. “<=”, “==”, “>=” etc.), the **separators**, a regex that matches **multiple whitespaces**, a regex that matches **any combination of letters and digits**, a regex matching **identifiers**, a regex matching **constants** and a regex matching **simple operators** (“<”, “>”, “=”, “!”, etc.) to split the line into tokens, while also keeping the separators. The above mentioned regex was made in such a way to handle all kinds of combinations of identifiers, constants, operators, whitespace etc., and it supports instructions which are not split by space (i.e. “max=a+b-c”). If anything, only this method should be modified to tokenize the line in a different way.

After the line is tokenized, each token is then identified as a separator, operator, reserved word – in which case it will be added to the PIF with an invalid position (-1, -1) and its corresponding code, or if it is an identifier or constant, it will be added to the SymbolTable first, and then added to the PIF with its corresponding code and the position from the SymbolTable.

If the token can’t be identified as one of the above, an exception is thrown indicating the line and the token that caused issues.

The code for each token is hardcoded – for each separator, operator, reserved word a unique code is generated and put into a map, and also, identifiers and constants have special codes, namely 0 and 1.

Checking if a token is an identifier or a constant is done as well via regexes – for identifiers, the regex checks that the token respects the specification, namely that it starts with a letter and the rest of it is either a letter or a digit, and for constants, it is checked that it is either an integer, character or string constant, and each of these have their own regex which also respects the specification, so I’m not going to go into much detail since it’s pretty self explanatory.

After the scanning is done, the SymbolTable and PIF are written to files in order to be checked easily, as mentioned in the assignment requirements.

Class diagram:

