Condrea Adrian
Group 932

Lab 3 - lexical analyser
Documentation

Github link: https://github.com/adriancondrea/FLCD_Project
The **documentation** folder contains the pdf files with documentation.
The **input** folder contains the programs (p1, p2, p3, p1err and test) for the lexical analyser.
The folder **lab1b** contains the lexic, syntax and token files.
The symbol table is the one implemented in lab2 and was documented in lab2 documentation. We are using an unique symbol table for identifiers and constants.

The lexical analyser has the reserved words, operators and separators hardcoded as static final fields in the class.

```
private static final List<String> RESERVED_WORDS = Arrays.asList("startProgram", "input", "declare", "typeCheck",
        "typeDefine", "check", "else", "output", "endProgram", "assign", "loop", "breakLoop", "integer", "boolean",
        "string", "char", "array");
private static final List<String> OPERATORS = Arrays.asList("+", "-", "*", "/", "%", "<=", ">=", "==", "!=", "!",
        "<", ">");
private static final List<String> SEPARATORS = Arrays.asList(" ", ",", "\"", "'", ".", "\n", "[", "]", "(", ")", "<", ">");
```

The symbol table has a program internal form (PIF) modelled as a list of pairs (string, integer). Each PIF entry represents the value of the token, along with the position in the symbol table, or -1 if it does not belong to the symbol table (is a reserved word, operator, or separator).
The boolean flag error signals whether the scanner detected a lexical error or not.

The scan function takes as parameter the file path, and tries to read it line by line and process each line. If the file is not found an error message is printed to the console. After processing the entire file, the scanner either outputs the success message (“Lexical analyzer has found no errors!”), or prints “Lexical error on line _, token _”, mentioning where in the file the error was.

The process line method splits the line at most three times. First, it splits it by “(“ and “)”, adding the separators to the token list also. Then, for each token obtained, checks if it is a valid token. If not, further tries to break it down by the compound operators (<=, >=, ==, !=) and comma.
For each subtoken, if it is still not valid, the scanner splits it by the remaining operators and separators and processes each resulting token again. If they still are not valid, a lexical error was found, and the error message is printed to the screen, along with setting the error flag to true.

The process token method checks if the token is a reserved word, operator or separator. If so, it adds it to the PIF with symbol table position -1, meaning it is not stored in the symbol table.

Otherwise, it checks if it is an identifier or a constant. If true, adds it to the symbol table and inserts the token into PIF with the position returned by the addElement function from the symbol table.
If it is neither reserved word, operator, separator, identifier or constant, the processToken method returns false, meaning it is not a valid token.
For checking if the token is a constant or an identifier, we test it against a regex expression.