

<https://github.com/roxanazachman01/FLCD>

Documentation:

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"

Deliverables: input, output, source code, documentation











Details:

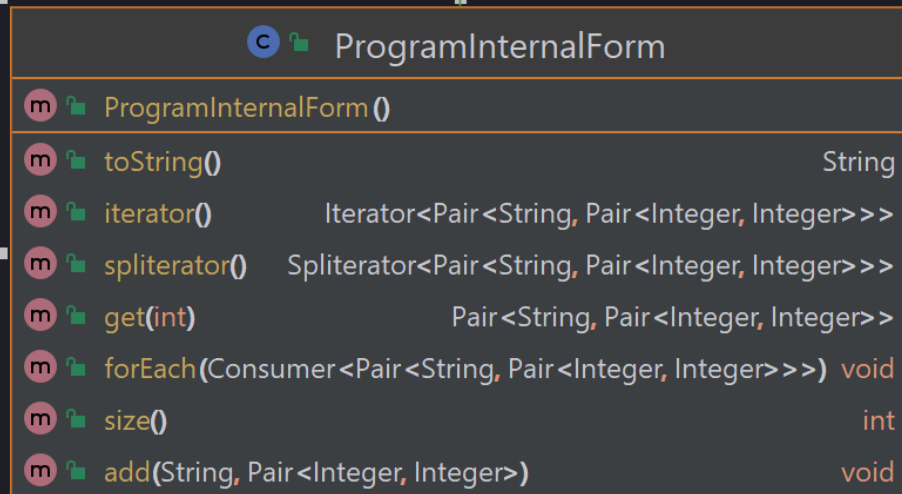
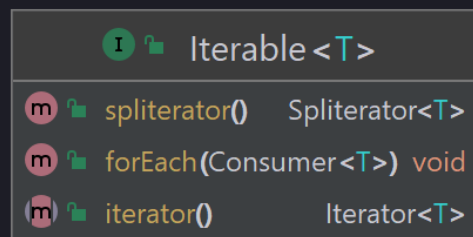
- ST.out should give information about the data structure used in representation
- If there exists an error the program should give a description and the location (line and token)

Analysis: The Scanner should be able to read a given program line by line, split the lines into correct tokens, classify them into the five categories (operators, separators, reserved words, identifier, constants), and if a token is not part of those five categories, print that there is a lexical error on that line number. After that, there should be the codify part, where the tokens get placed into the program internal form (PIF) and symbol table (ST).

Implementation: the algorithm implements the splitting into tokens (detection), classifying and codifying. Splitting is done by using java split function on space as delimiter, with special care for the cases where there are more tokens not delimited by space (ex: x=3;) and for the case of string constants, to not split the actual content of the strings by space. The tokens for classifying are read from the file token.in, with the format "token_category:token1,token2,...," (ex: "separator:[,{,},]" "operator:+,-,==,<="). For identifiers and constants, instead of actual tokens, there are regex rules for identifying if a token is an identifier or constant. Constants include regex for numerical constants as well as for string constants. Codifying makes use of the symbol table done previously. The PIF is a list of pairs of pairs, where first pair contains the token and the pair of positions, where position is -1,-1 if the token is not a symbol and the position in the hash table and linked list from the symbol table, if it is a symbol.

Scanner

		<code>Scanner(String, String, String, String)</code>	
		<code>detect(String)</code>	<code>List<String></code>
		<code>writeST()</code>	<code>void</code>
		<code>isSymbol(String)</code>	<code>boolean</code>
		<code>checkIfOperator(String, int)</code>	<code>String?</code>
		<code>codify(List<String>)</code>	<code>void</code>
		<code>classify(List<String>, int)</code>	<code>String</code>
		<code>printFileContent()</code>	<code>void</code>
		<code>writePIF()</code>	<code>void</code>
		<code>isValidConstant(String)</code>	<code>boolean</code>
		<code>isValidIdentifier(String)</code>	<code>boolean</code>
		<code>loadTokens()</code>	<code>void</code>
		<code>writeToFile()</code>	<code>void</code>
		<code>tokenize()</code>	<code>void</code>
		<code>isWordChar(char)</code>	<code>boolean</code>



SymbolTable		
m	SymbolTable(int)	
m	SymbolTable()	
f	size	int
m	hash(String)	int
m	insert(String)	Pair<Integer, Integer>
m	search(String)	Pair<Integer, Integer>
m	toString()	String
p	size	int

Testing: tests the scanning for the p2.txt and p1err.txt programs work as intended: correctly adds to PIF and ST and correctly detect lexical errors.

```

class ScannerTest {

    @Test
    void tokenize() {
        Scanner scanner = new Scanner( filePath: "in/p2.txt", tokensPath: "in/token.in", pifPath: "out/PIF.out", stPath: "out/ST.out");
        scanner.tokenize();
        var pif : ProgramInternalForm = scanner.getPif();
        var st : SymbolTable = scanner.getSymbolTable();
        var errors : String = scanner.getErrors();
        assertThat(errors.trim()).isEmpty();
        assertThat(st.search( symbol: "x").getKey()).isNotEqualTo( other: -1);
        assertThat(st.search( symbol: "y").getKey()).isNotEqualTo( other: -1);
        for (var pair : pif) {
            if (pair.getKey().equals("x")) {
                assertThat(st.search( symbol: "x").getKey()).isEqualTo(pair.getValue().getKey());
            }
            if (pair.getKey().equals("y")) {
                assertThat(st.search( symbol: "y").getKey()).isEqualTo(pair.getValue().getKey());
            }
        }

        scanner = new Scanner( filePath: "in/p1err.txt", tokensPath: "in/token.in", pifPath: "out/PIF.out", stPath: "out/ST.out");
        scanner.tokenize();
        errors = scanner.getErrors();
        assertThat(errors).isNotEmpty();
    }
}

```