Documentation Scanner – Laboratory 1

*Group 933: Bianca Ioana Cioată*

*1. Identifiers:*

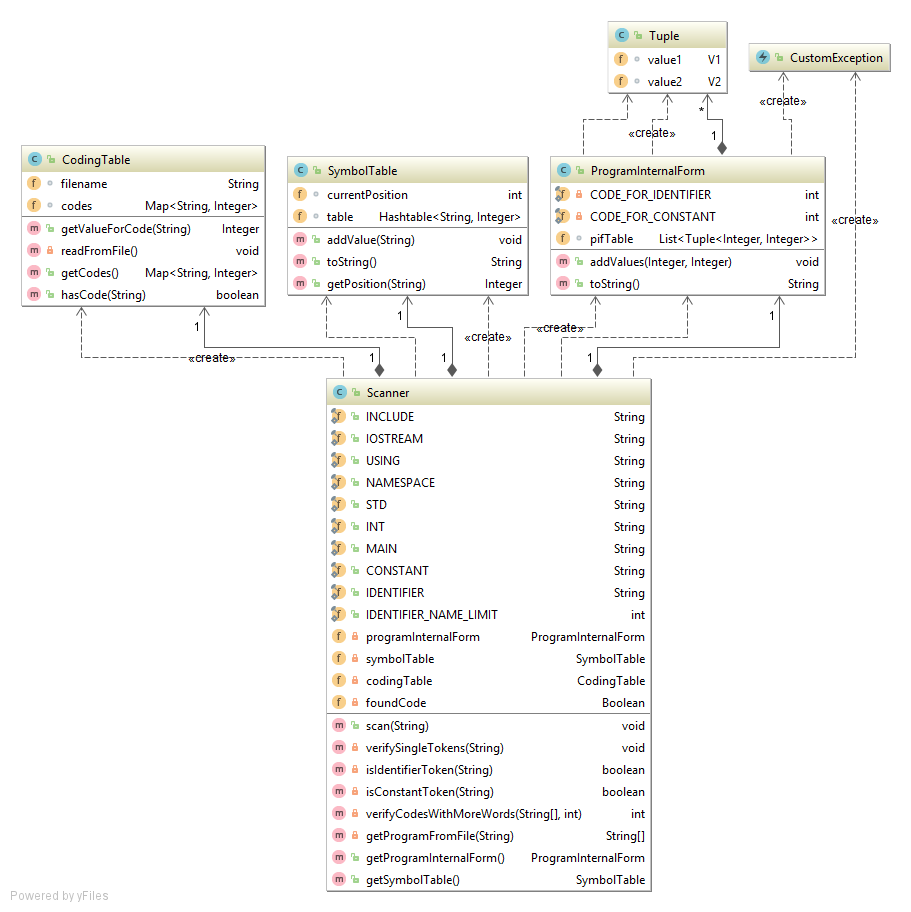
*a. length at most 8 characters*

*2. Symbol Table*

*: a. unique for identifiers and constants*

*3. Symbol Table Organization:*

*c. hashing table*



**CodingTable class** contains a map with the numeric codes of the possible keywords in the program. This map is filled in the constructor of the class, it calls a private method that reads from a given file the data.

Exceptions : The constructor may throw an CustomException if there is an IO Exception thrown, related to reading the file.

The codes for the scanner are:

0 identifier  
1 constant  
2 int  
3 float  
4 int  
5 const  
6 #include <iostream>  
7 using namespace std  
8 int main()  
9 cin  
10 cout  
11 ;  
12 (  
13 )  
14 {  
15 }  
16 <<  
17 >>  
18 =  
19 ==  
20 !=  
21 >  
22 <  
23 <=  
24 >=  
25 +  
26 -  
27 \*  
28 /  
29 %  
30 ,  
31 return  
32 if  
33 else  
34 while

**SymbolTable class** contains a hashtable that holds constants and identifiers. The key of the hashtable is the id of the identifier or the constant (numeric values) and their position in the code. These identifiers and constants are unique.

*/\*\*  
 \* Adds a constant or an identifier in the symbol table, if it does't already exist  
 \** ***@param value*** *- String  
 \*/***public void** addValue(String value};

**ProgramInternalForm class** contains a list of tuples, representing the atoms of the program, more specifically their code, associated with their position in the symbol table, in the case of constants and identifiers.

*/\*\*  
 \** ***@param code*** *- Integer  
 \** ***@param positionSymbolTable*** *- Integer number identifier or constant, null otherwise  
 \** ***@throws*** *CustomException if we try to add a null code or we do not insert the position from the symbol table in  
 \* the case of identifiers and constants; we may insert a null position  
 \*/***public void** addValues(Integer code, Integer positionSymbolTable) **throws** CustomException

**Scanner class** uses all the other classes. In the constructor it initializes the CodingTable and creates the symbolTable and the programInternalForm.

*/\*\*  
 \* It fills the programInternalForm and the symbolTable with the   
 \* associated data  
 \** ***@param filename*** *- the name of the file with the program we scan  
 \** ***@throws*** *CustomException - in case there is an unaccepted token or  
 \* if there is a IOException thrown when trying to read the file  
 \*/***public void** scan(String filename) **throws** CustomException ;

*/\*\*  
 \* Reads and splits the text in an array of string which were separated by space  
 \** ***@param filename*** *- the file from which it reads the program   
 \** ***@return*** *an array of possible tokens  
 \** ***@throws*** *CustomException in case of an IOException (cannot find or open the file)  
 \*/***private** String[] getProgramFromFile(String filename) **throws** CustomException

*/\*\*  
 \* Identifies tokens of longer length and returns the position of the next token.  
 \* (Finds the tokens : "#include <iostream>", "using namespace std", "int main()" )  
 \** ***@param tokensVal*** *- list of tokens in the program  
 \** ***@param i*** *- the current position in the list of program tokens/atoms  
 \** ***@return*** *the position of the next token to analyze  
 \** ***@throws*** *CustomException if it cannot add in the programInternalForm these values  
 \*/***private int** verifyCodesWithMoreWords(String[] tokensVal, **int** i) **throws** CustomException

*/\*\*  
 \* Verifies tokens composed of a single word - simple keywords, constants, identifiers  
 \* Exits the function if an empty string is found  
 \** ***@param token*** *- the token to be checked  
 \** ***@throws*** *CustomException - if it cannot identify a type for this token.  
 \* It allerts that there is a mistake in the written program from the file, that does  
 \* not respect the alphabet of this language  
 \*/***private void** verifySingleTokens(String token) **throws** CustomException

*/\*\*  
 \** ***@param s*** *- the string to verify  
 \** ***@return*** *true if is an identifier (contains only letters and is not a keyword),  
 \* false otherwise  
 \** ***@throws*** *CustomException if the string s is an identifier longer than IDENTIFIER\_NAME\_LIMIT (=8)  
 \*/***private boolean** isIdentifierToken(String s) **throws** CustomException

*/\*\*  
 \** ***@param s*** *- the string to verify  
 \** ***@return*** *true if s is a valid number, false otherwise  
 \* Examples unvalid numbers: -0 , 01, -01.10, 2.0, 2000.000020  
 \*/***private boolean** isConstantToken(String s)

Documentation Finite automaton – Laboratory 2, part 1

***Representation 1***

The content of the file is :

First line – keyword : **INITIAL** - for the initial state

INITIAL:  
q0

Second line – string representing the initial state

Third line – keyword : **FINAL –** for the final states

FINAL:  
q2 q3 q4

Fourth line – name of the final states, separated by space

Fifth line – keyword : **TRANSITIONS –** for the transitions

Remaining lines – the transitions, with the components separated by space.

The components are:

1. First state – String
2. Second state – String
3. The symbols in {}, separated by comma (,)
4. TRANSITIONS:  
   q0 q1 {-}  
   q1 q2 {1,2,3,4, 5,6,7,8,9}  
   q0 q2 {1,2,3,4,5,6,7,8,9}  
   q2 q2 {0,1,2,3,4,5,6,7,8,9}

***Representation 2***

The content of the file is :

First line – keyword : **INITIAL** - for the initial state

INITIAL:  
q0

Second line – string representing the initial state

Third line – keyword : **FINAL –** for the final states

FINAL:  
q2 q3 q4

Fourth line – name of the final states, separated by space

Fifth line – keyword : **TRANSITIONS –** for the transitions

Remaining lines – the transitions, with the components separated by space.

The components are:

1. First state – String
2. Symbol– String
3. The second possible states in {}, separated by comma (,)
4. TRANSITIONS:  
   q0 - {q1}  
   q1 1 {q2}  
   q1 2 {q2}  
   q1 3 {q2}  
   q1 4 {q2}  
   q1 5 {q2}  
   q1 6 {q2}  
   q1 7 {q2}  
   q1 8 {q2}  
   q1 9 {q2}  
   q0 1 {q2}  
   q0 2 {q2}

***Program***

**FiniteAutomaton class** is an abstract class that contains the abstract method :

**public abstract void** readAutomaton() **throws** CustomException;

that defines the way the automaton is read.

It has the fields :

**protected** Set<String> **states**; - the states of the automaton  
**protected** Set<String> **alphabet**; - set of symbols, accepted by the atomaton  
**protected** String **initialState**; - initial state of the automaton  
**protected** Set<String> **finalStates**; - accepted final states, accepted by the automaton  
**protected** Set<Transition> **transitions**; - the transitions

The Transitions are represented in the following way, using the ***helper class Transions***:

**public class** Transition   
 **private** String **state1**;  
 **private** String **state2**;  
 **private** Set<String> **symbols**;

It has the methods :

*/\*\*  
 \* Verifies if the sequence is accepted by the automaton. (works for deterministic automatons)  
 \*  
 \** ***@param sequence*** *- String sequence of characters  
 \** ***@return*** *the longest prefix that is accepted by the automaton or the same string given if it is all correct  
 \*/***public** String verifySequence(String sequence)

*/\*\*  
 \** ***@return*** *true if the automaton is deterministic or not  
 \*/***public boolean** isDeterministic()

**FiniteAutomatonFromFile class** is an implementation of the class Finite

