Rusu Horia-Radu, 936/1
https://github.com/HoriaRaduRusu/Formal-Languages-And-Compiler-Design/tree/main/SymbolTable

Hash table:

The hash table has a Map attribute, which holds as keys the hash values of the added elements, and as values, lists of elements referred to as buckets. It will hold values of type V. It exposes the following methods:

- Integer add(V value):
  Adds the given value to the hashtable and returns the key of the added element in the HashTable.
  The key of the element will be of the form c(h(v)) * capacity + h(v), where:
  - h(v) is the result of hashing the given value, also representing the key of the bucket in which the element will be added
  - c(h(v)) is the number of elements already in the bucket at key h(v)
  - capacity is the number of available buckets (in our case 389)

- V get(Integer key):
  Returns the value stored at the given key, or null if no value is stored at that key.

- Integer search(V value):
  Returns the key where the value can be found in the table, or -1 if the value is not stored in the table.

- String toString():
  Returns the string representation of the hash table.

  The hash function used is the following:
  In our case, capacity = 389
  If the added value is an integer, h(v) = v % capacity. Otherwise, denote with s the sum of the ASCII values of the characters used in the string representation of the value - the hash function will be h(v) = s % capacity.

Symbol Table:

The symbol table uses the above specified hash table. It exposes the following methods:

- Integer addValue(Object value):
  If the value is already in the table, return the key of the value in the table. Otherwise add the element to the table and return the key it was added at.

- Integer searchForValue(Object value):
  Returns the key at which the given value can be found at in the table, or -1 if it cannot be found in the table.

- Object getValue(Integer key):
  Returns the element found in the table at the given key, or null if there is no element at the given key.
- String toString():
  Returns the string representation of the symbol table, giving informations about how the data is stored.

PIF:

The PIF holds an ordered list of pairs, formed from a String and an Integer, the integer representing the position of the element described by the string in the symbol table (the key if the element is an identifier or constant, otherwise -1). It exposes the following methods:

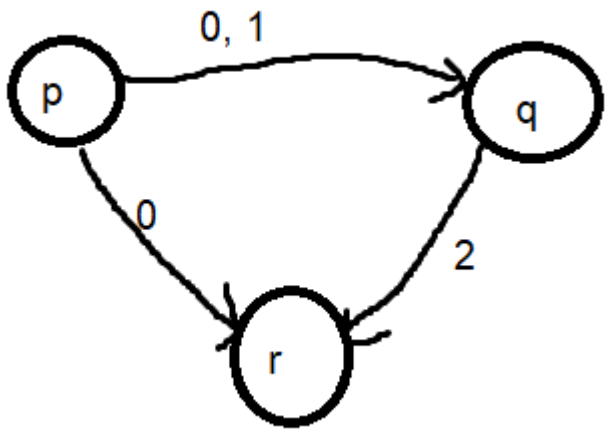
- void addElement(Pair<String, Integer> element):
  Adds the element to the PIF.
- String toString():
  Returns the string representation of the PIF.

Vertex:

Vertex is a data class used to represent a node in the graph described below. It has a label attribute, representing the identifier of the vertex.

GraphWithLabeledAdjacency:

This class describes an oriented graph with the property that each edge is labeled with at least one String. It has a Map attribute, which describes the graph in the following way: the keys are the starting vertices of the edges, and the values are Maps, holding as keys the labels of the edges that start from the vertex, and as values, a list of the vertices that can be reached from the starting vertex by going through edges labeled with a given label. For example, if we take the following graph:



the map would look like this:
p -> (0 -> [q, r], 1->[q])
q -> (2 -> [r])

It has the following methods:

- List<Vertex> getNeighborsThroughElement(Vertex starter, String element):
  Returns the vertices that neighbor the “starter” vertex through edges labeled with “element”.

- Map<Vertex, Map<String, List<Vertex>>> getAdjMap():
  Returns the adjacency map.

FiniteAutomaton:

This class describes a finite automaton described in a given file. It holds a set of strings for the states and one for the alphabet (since the order of elements in the states and alphabet don’t matter and they can’t repeat), a string representing the starting state, a set of strings representing the end states (with the same reasoning as above), and a GraphWithLabeledAdjacency for the transitions. When creating a FiniteAutomaton, if any state or alphabet element used in the file is not present in the set of states or alphabet elements, an error is thrown. It exposes the following methods:

- Set<String> getStates():
  Returns the set of states.

- Set<String> getAlphabet():
  Returns the set of alphabet elements.

- String getStartingState():
  Returns the starting state.

- Set<String> getEndStates():
  Returns the set of end states.

- String getTransitionsForPrinting():
  Returns the transitions graph in a form that can be printed.

- boolean isDFA():
  Returns true if the FiniteAutomaton represents a deterministic finite automaton, otherwise false.

- boolean isSequenceAccepted(String sequence):
  Returns true if the finite automaton described accepts the given sequence, otherwise false.

The file that specifies the finite automaton should be written in the format described in the following way (in EBNF form, and with “\n” representing a new line):

nZDigit = “1” | … | “9”
digit = “0” | nZDigit
state = “s” nZDigit {digit}
alphabetElement = digit | “a” | … | “z” | “A” | … | “Z” | “_”

stateList = state {“ ” state}
alphabet = alphabetElement {“ ” alphabetElement}
transition = state “ ” alphabetElement “ ” stateList
transitionList = transition {“\n” transition}

file = stateList ”\n” alphabet “\n” state “\n” stateList “\n” transitionList

LexicalAnalyzer:

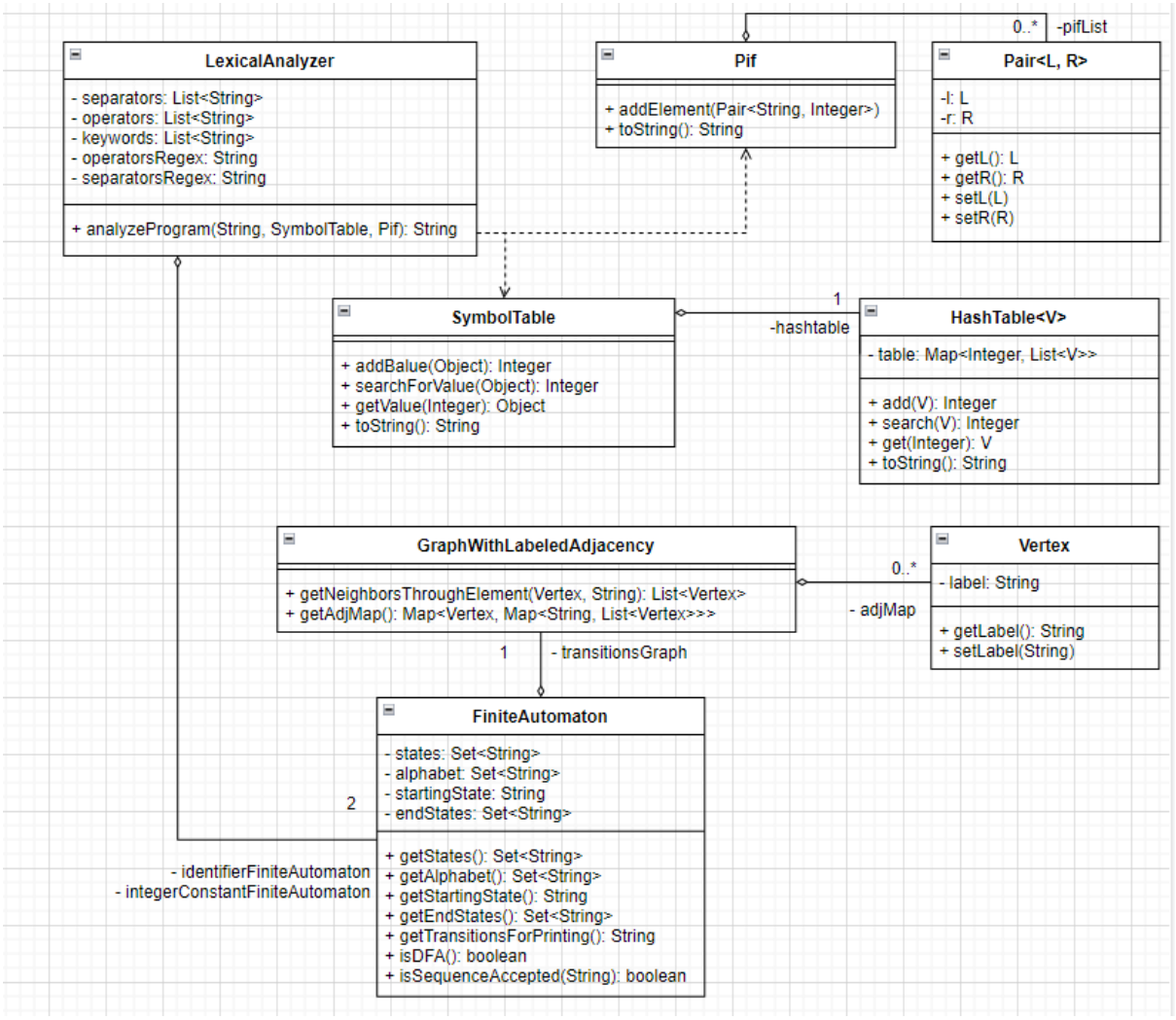
The lexical analyzer has methods used to analyze a given program from a lexical point of view, producing the symbol table and the PIF. It exposes the following methods:

- String analyzeProgram(String programLocation, SymbolTable symbolTable, Pif pif): Analyzes the program found in the file described by programLocation from a lexical point of view. The keywords, operators and separators taken into accounts are the ones given in the lists used to instantiate the object. While analyzing, it populates the symbol table and PIF accordingly. If no lexical errors were found, it returns the message “lexically correct”; otherwise, for each lexical error found, it gives the line and the token that caused the problem.

  It uses the following regexes:
- To match char constants: ^'[a-zA-Z\\d]'$; Char constants are represented by a single letter or digit in between apostrophes
- To match string constants: \"[a-zA-Z\\d]+\"; String constants are arrays of at least one letter or digit in between quotes
- To separate tokens, regexes of the form (?<=%1$s)|(?=%1$s) are joined, where %1$s represents a given separator or operator

  To match identifiers and integer constants, it uses two finite automata.

Class diagram:


LexicalAnalyzer
- separators: List<String>
- operators: List<String>
- keywords: List<String>
- operatorsRegex: String
- separatorsRegex: String
+ analyzeProgram(String, SymbolTable, Pif): String
Pif
+ addElement(Pair<String, Integer>)
+ toString(): String
0..*
-pifList
Pair<L, R>
-l: L
-r: R
+ getL(): L
+ getR(): R
+ setL(L)
+ setR(R)
SymbolTable
+ addBalue(Object): Integer
+ searchForValue(Object): Integer
+ getValue(Integer): Object
+ toString(): String
1
-hashtable
HashTable<V>
- table: Map<Integer, List<V>>
+ add(V): Integer
+ search(V): Integer
+ get(Integer): V
+ toString(): String
GraphWithLabeledAdjacency
+ getNeighborsThroughElement(Vertex, String): List<Vertex>
+ getAdjMap(): Map<Vertex, Map<String, List<Vertex>>>
0..*
- adjMap
Vertex
- label: String
+ getLabel(): String
+ setLabel(String)
1
- transitionsGraph
FiniteAutomaton
- states: Set<String>
- alphabet: Set<String>
- startingState: String
- endStates: Set<String>
+ getStates(): Set<String>
+ getAlphabet(): Set<String>
+ getStartingState(): String
+ getEndStates(): Set<String>
+ getTransitionsForPrinting(): String
+ isDFA(): boolean
+ isSequenceAccepted(String): boolean
2
- identifierFiniteAutomaton
- integerConstantFiniteAutomaton