https://github.com/911-Ciurcau-Leonardo-Iulian/FLCD/tree/Lab4FA

For including the FA in the scanner changes were made in Scanner->determineIdentifierType(std::string identifier),
with the scanner now having two FA's from the files identifierFA.in and integerFA.in

+class FiniteAutomaton
	+struct TransitionPart # used for transitions hash table
		+terminal: String
		+outputState: String
	-states: List<String>
	-initialState: String
	-alphabet: List<String>
	-transitions: HashTable<String, List<TransitionPart>> # example: transitions[A] = { { b, B }, { c, C } }
	-finalStates: List<String>
	-acceptsSequence(in: istream): boolean # checks if sequence is valid by traversing the transitions and checking if they are correct,
at the end it also checks if the last state is a final state

	+enum Stage { STATES, INITIAL_STATE, ALPHABET, TRANSITIONS, FINAL_STATES } # used for input file parsing
	+FiniteAutomaton(inputFile: String): parses the input file as described below in the EBNF
	+isDeterministic(): boolean # checks if the automaton is deterministinc by looking if a state has more than one transition to the same output state
	+acceptsSequenceFromFile(inputFile: String)
	+acceptsSequenceFromString(sequence: String)

EBNF for FA input format:
	letter ::= 'a'|'b'|...|'z'|'A'|'B'|...|'Z'
	specialSymbol ::= '-'|'+'|'_'
	digit ::= '0'|'1'|...|'9'
	terminal ::= letter | digit | specialSymbol
	newline ::= '\n'
	twoNewlines ::= newline newline

	state ::= letter
	states ::= state{,state}
	initialState ::= state
	alphabet ::= terminal{,terminal}
	transition ::= state,terminal,state
	transitions ::= transition {newline transition}
	finalStates ::= state,{,state}
	inputFAFile ::= states twoNewlines initialState twoNewlines alphabet twoNewlines transitions twoNewlines finalStates

+class ProgramInternalForm
	+enum Identifer { ID, CONSTANT }

	+struct PIFItem
		code: Integer # can be ID, CONSTANT, or code for the reserved symbol
		symbolTablePosition: Integer

	-pif: List<PIFItem> # used for storing the items in the pif

	+add(code: Integer, symbolTablePosition: Integer)
	+fprint(outputFile: String, tokensPositionList: List<String) # prints the pif to the given file, using tokensPositionList for reserved symbols

```
+class SyntaxErrorException
      +SyntaxErrorException(message: String, line: Integer) # used for outputing
an error with format: `Syntax error at line {line}: {message}`


+class Scanner
      -tokens: HashTable<String, Integer> # used for storing reserved words
      -tokensPositionList: List<String> # the inverse of the hash table, for
mapping positions to reserved words
      -symbolTable: SymbolTable
      -programInternalForm: ProgramInternalForm
      -variableFA: FiniteAutomaton # automaton for variables
      -integerFA: FiniteAutomaton # automaton for integers
      -lineCount: Integer # used for storing the current line in the input
program
      -computePosition(symbol: String): Integer # retrieves the position of the
symbol from the symbol table, and adds it to the symbol table beforehand if it
does not yet exist there
      +static globalTokenId: Integer # used for attributing and storing codes
for the reserved words

      -addIdentifier(identifier: String) # adds an identifier to the pif
      -determineIdentifierType(identifier: String): Identifier # determines if
identifier is id or constant, and also throws SyntaxError if it is not a valid
one
      +Scanner(tokensFile: String, symbolTable: SymbolTable,
programInternalForm: ProgramInternalForm, variableFAFile: String, integerFAFile:
String) # initializes the scanner, computes and stores the reserved words from
tokensFile, by attributing them a code which is their position in
tokensPositionList
      +scan(programFile: String) # scans the program from programFile
            # there are multiple cases taken into account here, and they can be
found in the comments from the source code
      +getTokensPositionList(): List<String> # returns tokensPositionList



+class SymbolTable
      -hashTable: HashTable<String, Integer> # the hash table used for storing
the symbols
      -positionList: List<String> # the inverse of the hash table, for mapping
positions to symbols
      +static globalPosition: Integer # used for storing the positions of
symbols

      +SymbolTable()
      +add(symbol: String) # adds a symbol to the symbol table with the current
globalPosition as value, then increments globalPosition
      +contains(symbol: String): boolean # return true if a symbol exists in the
table, false otherwise
      +getPosition(symbol: String): Integer # gets the position of the symbol if
it exists, -1 otherwise
      +getSymbol(position: Integer): String # gets the symbol at the given
position
      +fprint(outputFile: String): # prints the symbol table to the given file

+class HashTable<K, V>
      -struct HashBucket
            key: K
            value: V
            next: HashBucket* # next in the bucket chain
```

```
      -static constant loadFactorLimit: double = 0.7
      -size: Integer # the total number of buckets
      -count: Integer # the total number of current elements
      -buckets: HashBucket**

      -hash(value: String): Integer # computes the hash of a string value, the
sum of ascii codes modulo size
      -loadFactor(): double # computes the current load factor as count / size
      -loadFactorExceeded: boolean # returns true when loadFactor() >
loadFactorLimit, false otherwise
      -resize() # resizes the hash table for when the load factor is exceeded by
doubling its size and reinserting the elements
      +HashTable()
      +add(key: K, value: V) # adds the key and the value in the hash table
      +contains(key: K): boolean # returns true if key is found in the hash
table, false otherwise
      +get(key: K): V* # returns a pointer to the value mapped to the key if it
exists, null otherwise
      +~HashTable()
```