## https://github.com/KovacsAndrea/FLCD/tree/main/Assignment3

## Lab 3 DOCUMENTATION

**Statement: Implement the Symbol Table (ST) as the specified data structure, with the corresponding operations**

Symbol Table: Implemented as a. unique for identifiers and constants (create one instance of ST)

# **Exceptions Package**

LiteralException Class

* Extends Exception class
* Input: line number, position, file and message
  + message: can be one of the following “ ” expected”, “ ‘ expected”, “Illegal literal”
  + file: the path of the program file being scanned
  + line number: the number of the line where the Lexical Error is found
  + position: the position in the line where the error is encountered
* getMessage function will return a String regarding the details of the Lexical Error encountered

# **Specification Package**

Constants

* static Class
* has three static attributes: a regex for numeric patterns (signed and unsigned), a regex for char patterns, and a regex for string patterns
* has three static methods that are getters for the three regexes

Identifiers

* has the regex for identifiers and a getter

Operators

* stores all operators
* getAll() getter for the stored operators
* isOperator function returns true if a string is an operator and false otherwise
* isPartOfOperator is used for operators that contain more than one character, such as ==, !=, <=, =>, ++, --, returns true if the given string is =, !, <, + or – and false otherwise

ReservedWords

* Stores all reserved words
* getAll(): getter for reserved words
* isReservedWord function returns true if a string is a reserved word and false otherwise;

Separators

* Stores all separators
* getAll(): getter for separators
* isSeparator function returns true if a string is a separator and false otherwise;

***Specifications***

* Attributes
  + final static HashMap<String, Integer> codification – will hold sets of Strings and Integers, where the Strings are Reserved Words, Operators, Separators, indicator for Constants and Identifiers. Will hold 0 for identifiers and 1 for constants;
* Methods
  + isConstant(String s) returns true if s is a Constant and false otherwise
  + isIdentifier(String s) returns true is f is an Identifier and false otherwise
  + isSymbol(String s) returns true if s is an Operator, Reserved Word, or Separator and false otherwise
  + createCodifications() places tuples 0 “identifiers” and 1 “constants” in the codification HashMap; for all other symbols, increments the code and places the tuple in the HashMap
  + getCode(String token) returns the code of a token

# **Domain**

* migrated class SymbolTable to MyScanner
* added new clas PIF (Program Internal Form)
* added new attributes to class MyScanner
  + final PIF \_pif
  + final String \_PIFFile – initialized in constructor
  + final String \_STFile– initialized in constructor
  + boolean \_lexicalCorrect
* changed name of getSymbolTable method to scan

Pair

* parametrized record class
* getKey() returns the key of the element
* getValue() returns the value of the element
* equals() override for the equals function, returns true if the keys and the values are equal

HashTable

* class has 2 attributes
* \_variablesAndConstants is an array of arrays (hash table)
* \_size is an integer, initialized in the constructor
* getSize(): returns the value stored in \_size
* hash(): computes the hash value of a token(string); hash value will be computed by sum%\_size where sum is the sum of the ascii values of all characters in the string, and the \_size is the size given initially to the hash table
* add(): will add a token in the table, if the key already exists, the token will not be added again; if two elements hash to the same position, the new element will be added on the next position in the array
* contains(): returns true if the hash table contains the string and false otherwise
* getPosition(): returns Pair<Integer, Integer>, where the key is the position of the secondary array in the main array, and the value is the position of the element in the secondary array
* remove(): removes an element from the hash table

PIF

* Attributes
  + final List<Pair<Integer, Pair<Integer, Integer>>> pif – a list of pairs where the first value is an Integer and the second value is a pair of integer and integer
* Methods
  + void add(Integer code, Pair<Integer, Integer> value)
  + receives a code and a Pair of integer and integer and adds is to the pif attribute

MyScanner

* Attributes
  + private final HashTable \_symbolTable – will hold a reference to an instance of HashTable class
  + private final PIF \_pif - will hold a reference to an instance of PIF class
  + private final String \_programFile – will hold a reference to the program file
  + private final String \_PIFFile – will hold a reference to the file where the resulting symbol table will be written
  + private final String \_STFile – will hold a reference to the file where the resulting program internal form will be written
  + private int lineNr – will hold the number of the current line that is being scanned; initialized with 1
  + private boolean \_lexicalCorrect – will be true if the program is lexically correct and false otherwise; initialized with true
* Methods
  + void scan() – lineNr is initialized with 1, program file is opened. For each line calls the runTokens() function and receives an Array of strings representing the tokens found on that specific line. Adds the tokens in a List of pairs, where the first value is a token and the second value is the line number. After iterating the file, if \_lexicalCorrect is true, it prints a message stating as such. Calls buildPIF() and writeResults() functions.
  + List<String> runTokens(String line) - a wrapper for tokenize() function. It runs the tokenize() function and if it encounters any exceptions, it prints its message and sets \_lexicalCorrect to false
  + List<String> tokenize() receives a line and checks for constants, identifiers, operators, separators and reserved words, and adds them in the \_hashTable
  + String getStringConstant(String line, int position) checks for the next position of the character “. If it is not fount, it throws a new LiteralException with the message “” expected”. If it is found, it generates a substring from the first “ and checks if it matches the string pattern
  + String getCharConstant(String line, int position) checks for the next position of the character ‘. If it is not fount, it throws a new LiteralException with the message “’ expected”. If it is found, it generates a substring from the first ‘ and checks if it matches the string pattern
  + String getIdentifier(String line, int position) checks for the position of the next separator, operator or white space. If it is not found, a substring from the given position and to the end of the line is generator. If it is found, a substring from the given position and the new found position is generated. Returns the substring if it matches the identifier pattern, throws a LiteralException otherwise. Function will also return reserved words. In the tokenize function e will check if the string is a reserved word. If it is, we will not add it to the hash table
  + String getInteger(String line, int position): If integer is signed, checks if there is a space between + or - and the integer. If what follows after the sign is an identifier, it returns null. Otherwise it checks for the position of the next white space, arithmetic operator or relational operator. If it is found, generates a string from the given position to the found symbol, otherwise, to the end of the line. If the string does not match the numeric pattern, it throws a Literal Exception, or returns it otherwise.
  + void buildPIF(List<Pair<String, Integer>> tokens) – receives a list of Pairs of String and Integer.
    - If it is a symbol, \_pif will add the code saved in Specifications Class, and the Pair will fold values -1 and -1
    - If it is an identifier \_pif will add code 0 and the position of the identifier saved in the symbol table
    - If it is a constant \_pif will add code 1 and the position of the constant saved in the symbol table
  + void writeResults() will open the \_PIFFile and will write the data in \_pif. Will open \_STFile and will write the data in \_symbolTable.

SymbolTable

* class has three attributes
* \_hashTable – an instance of the HashTable class, \_programFile – string containing the path to the file where the program is stored, and lineNr – initialized with 0
* getSymbolTable(): opens the \_programFile, scans every line and calls tokenize() function for each line; increments lineNr for each line scanned
* tokenize(): for a line in the program identifies the constants, identifiers and reserved words, and adds them in the \_hashTable
* getStringConstant() checks for the next position of the character “. If it is not fount, it throws a new LiteralException with the message “” expected”. If it is found, it generates a substring from the first “ and checks if it matches the string pattern
* getStringConstant() checks for the next position of the character ‘. If it is not fount, it throws a new LiteralException with the message “’ expected”. If it is found, it generates a substring from the first ‘ and checks if it matches the string pattern
* getIdentifier() checks for the position of the next separator, operator or white space. If it is not found, a substring from the given position and to the end of the line is generator. If it is found, a substring from the given position and the new found position is generated. Returns the substring if it matches the identifier pattern, throws a LiteralException otherwise. Function will also return reserved words. In the tokenize function e will check if the string is a reserved word. If it is, we will not add it to the hash table
* getInteger(): If integer is signed, checks if there is a space between + or - and the integer. If what follows after the sign is an identifier, it returns null. Otherwise it checks for the position of the next white space, arithmetic operator or relational operator. If it is found, generates a string from the given position to the found symbol, otherwise, to the end of the line. If the string does not match the numeric pattern, it throws a Literal Exception, or returns it otherwise.