Github link: https://github.com/DayanaRalucaMardari/FLCD/tree/main/lab%203

SymbolTable

Implemented a SymbolTable based on a **HashTable** which can be used for both the IDENTIFIERS and CONSTANTS,
in the same table.
The methods of the **HashTable** are also implemented int the **SymbolTable**, except the `hash(String key)` method.
For tokens, the position in the SymbolTable is considered to be `(-1, -1,)`.

Pair
Implemented the pair class which has the integer `firstItem` and `secondItem` attributes, used for storing the position of an element from the SymbolTable (HashTable).

HashTable
It is implemented as a list of lists, meaning that on each position from a list there is another list.

It has the following methods:
* `getSize()` - returns the size of the list which has as elements other lists
* `hash(String key)` - returns the hash value of a given element as parameter
* `findElemPosition(String elem)` - returns the position of the element as a `Pair` object if found; otherwise, it returns null
* `containsElem(String elem)` - returns `true` whether the given element is found; otherwise, it returns `false`
* `findByPosition(Pair position)` - returns the element found on the given position. If the position is invalid, it throws `IndexOutOfBoundsException`
* `addElem(String elem)` - adds a new element into the table. It returns a pair with the position where it is located into the table. If the element already exists in the table, it throws an `Exception`
* `toString()` - override method that returns the size of the hash table and the elements

ProgramInternalForm
It is implemented as a list of pairs where the first elements is the `element` and the second one is its `position`.

* `add(Pair<String, Pair<Integer, Integer>> tokenPair)` - adds the passed as parameter to the list
* `toString()` - prints all the elements of the PIF as: `element --> (listIndex, elemIndexInList)`

MyScanner
It builds the **Program Internal Form (PIF)** and the **Symbol

Table**, and it checks the given **source code** for
lexical errors. It also gets the **reserved words** and the
tokens from a given file.

* `readTokens()` - reads the tokens file and adds the *reserved works* into `reservedWords` field and the *separators & operators* into `tokens`. Returns `IOException` if the tokens file was not found.
* `setSourceCode(String sourceCode)` - sets the `sourceCode` with the value passed as param.
* `start(String fileName)` - starts the sourceCode scanning and checks for lexical errors. The scanning action includes reading the source code, going through the source code and identifying the identifiers, string constants, integer constants and tokens. Also, all the identified elements are then memorized into the SymbolTable and all their positions into the PIF.
If no errors were found, the contents of the **SymbolTable** and **PIF** are written into files, and it informs the user that the source code is lexically correct. Otherwise, it informs the user that there are lexical errors, and it specifies the line where the error occurred.
* `skipSpaces()` - skips spaces and increased the line number when the new line char is found.
* `tokenizer()` - it tokenizes the source code by searching for identifiers, constant integers, constant string and tokens.
If none of the searched elements where not found, it throws `MyScannerException` and tells the user where the lexical error occurred.
* `tryIdentifierMatching()` - tries to match the current substring indicated by the index with an identifier. The matching flow is made of checking if the beginning of the substring matches format of an identifier by using regular expressions.
If there is a match and the identifier is valid, the identifier is added to the SymbolTable (if it is not already contained in the SymbolTable) or it gets its position from the SymbolTable in order to be added to the PIF. It returns `false`, if there is no match or if the identifier is not valid. Otherwise, `true`.
* `validateIdentifier(String possibleIdentifier, String sourceCodeSubstr)` - validates the possible identifier by checking if it is a reserved word, it matches the identifier pattern, or if it already is found in the SymbolTable.
* `tryTokenListMatching()` - tries to match with a token from tokens list. First it checks whether it is a reserved word, then if it is a separator or an operator. Returns `true`, if it is a reserved word or a token from the list. Otherwise, `false`.
* `tryIntConstMatching()` - tries to match with a constant integer. Checks if the current element is a valid number, meaning it only contains digits (no other characters), starts with a digit greater than 0, and it might have the +/- specified in the beginning. Or it is simply 0. If it is a valid number, it is added to the SymbolTable (if it's not already) and added its position to the PIF. Returns `true`, if the number is valid. Otherwise, `false`.

* `tryStringConstMatching()` - tries to match with a constant string. Checks whether the current element is a valid string, meaning that the quotes are closed correctly and has no invalid chars. Returns `true`, if the string is valid. Otherwise, `false`.
* `getSymbolTable()` - returns the `symbolTable`.
* `getPIF()` - returns the `PIF`.

MyScannerException
It is an exception class used for `MyScanner`, extending the `RuntimeException` class and overriding the `message` method.