Documentation

Symbol Table

The Symbol Table is composed from 3 separate hash tables, one for identifiers, one for integer constants and one for string constants. Each hash table is represented by a list in which every position is another list, in order to be able to store values that hash to the same position. Also, the hash tables have a size. An element from the symbol table has as position a pair of indices, the first one being the index of the list in which the element is stored and the second one being its actual position in that list. The hash function is value modulo the size of the list, for integer values, and the sum of the ASCII codes of the characters modulo the size of the list, for string constants/identifiers. The implementation of the hash table is generic.

Operations:

Hash table

- hash(key: int): int – computes the position in the ST of the list in which the integer constant will be added
- hash(key: string): int – computes the position in the ST of the list in which the string constant/identifier will be added, based on the sum of the ASCII codes of their characters
- getSize(): int – return the size of the hash table
- getHashValue(key: T): int – return the corresponding position in the ST according to the type of the parameter 'key'
- add(key: T): (int, int) – add the key to the hash table and return its position if the operation is successful; otherwise, throw an exception
- contains(key: T): boolean – return if the given key is in the hash table or not
- getPosition(key: T): (int, int) – return the position in the ST of the given key, if it exists; otherwise, return (-1, -1)
- toString() (overridden method) – return the string representation of the hash table

Symbol table

- has 3 hash tables: for identifiers, for string constants and for integer constants
- addIdentifier(name: string): (int, int) – add an identifier and return its position in the ST
- addIntConstant(constant: int): (int, int) – add an integer constant and return its position in the ST

- addStringConstant(constant: string): (int, int) – add an string constant and return its position in the ST
- hasIdentifier(name: string): boolean – return if the given identifier is in the ST or not
- hasIntConstant(constant: int): boolean – return if the given integer constant is in the ST or not
- hasStringConstant(constant: string): boolean – return if the given string constant is in the ST or not
- getPositionIdentifier(name: string): (int, int) – get the position of the identifier in the ST
- getPositionIntConstant(constant: int): (int, int) – get the position of the integer constant in the ST
- getPositionStringConstant(constant: string): (int, int) – get the position of the string constant in the ST
- toString() (overridden method) – get the string representation of the whole symbol table

ScannerException

This class is used as an exception for the Scanner, it extends the class RuntimeException and overrides one method from it.

Scanner

The Scanner class is responsible with building the program internal form and symbol table, while checking if the input program is lexically correct or not. The program internal form is a list composed from pairs of the form string and position in the symbol table, where string is the actual token, string const, int const or identifier. For the tokens, the position in the symbol table is considered to be (-1, -1). The Scanner also has fields for the program string, a symbol table, an index and the current line of the program.

Operations:

- setProgram(program: string) – setter for the program field
- readTokens() – read from the token.in file the tokens and separate them into reserved words and other tokens
- skipSpaces() – method to skip the spaces from the program and update the current line and index accordingly
- skipComments() – method to skip the comments from the program and update the index accordingly
- treatStringConstant(): Boolean – method to treat the case in which we have a string constant in the program; the method checks if the string is lexically correct (no invalid characters and quotes are closed correctly) and if it is, it adds it to the symbol table if it

does not exist and to the program internal form, updates the index and returns true; if the string constant is invalid, the method returns false

- treatIntConstant(): Boolean – method to treat the case in which we have an integer constant in the program; the method checks if the number if valid (no other characters except numbers) and if it is, it adds it to the symbol table if it does not exist and to the program internal form, updates the index and returns true; if the integer is invalid, the method returns false
- checkIfValid(possibleIdentifier: string, programSubstring: string): Boolean – the method checks if a possible identifier is valid by checking if it is part of a declaration or if it is in the symbol table; if it does not satisfy any of the conditions, the possibleIdentifier is an invalid token and the method returns false; otherwise, the method returns true
- treatIdentifier(): Boolean – method to treat the case in which we have an identifier in the program; the method checks if it is a valid identifier (starts with _ or a letter, contains only letters, digits and _ , is part of a declaration or a previously defined identifier) and if it is, it adds it to the symbol table if it does not exist and to the program internal form, updates the index and returns true; if the identifier is not valid, it returns false
- treatFromTokenList(): Boolean – the method checks if the current element of the program is a reserved word or another token and if it is, it adds it to the program internal form and returns true; otherwise, it returns false
- nextToken() – treats the current case and if there is no corresponding case, throw an exception that there is a lexical error at the current line and index
- scan(programFileName) – reads the program from the given file and checks for the next token until the end of the program; at the end, it writes the program internal form and symbol table to output files and displays a message that the program is lexically correct; if an exception is caught, it displays the message of that exception