Github repository: <https://github.com/Dospinescu-Rares/FLCD-Project>

FLCD week 6 assignment

Dospinescu Rares

Group 932/1

Informatics in English Specialization

***class SymbolTable:***

The SymbolTable class provides a simple interface to manage symbols (identifiers and constants) using a custom hash table implementation. It allows adding, removing, checking existence, and finding the position of symbols based on their keys.

**def add(self, key):**

Adds a key to the SymbolTable

* **key**: The key being inserted
* **return**: The position in which the key was inserted

**def remove(self, key):**

Removes a key from the SymbolTable

* **key**: The key being removed
* **return:** None

**def contains(self, key):**

Checks for the presence of a key in the SymbolTable

* **key**: The key we are searching for
* **return:** The location of the key, -1 otherwise

**def get\_position(self, key):**

Searches for the key’s position in the SymbolTable

* **key**: The key whose position we are searching for
* **return:** The position of the key

**def \_\_str\_\_(self) -> str:**

* **return:** The SymbolTable as a string

**class HashTable:**

Custom implementation of a hash table data structure. This hash table allows storing keys and provides basic operations like hash, add, remove, contains and get\_position.

* **size:** The size of the HashTable

**def hash(self, key):**

Returns the hash of a given key

* **key:** The key whose hash we are looking for
* **return:** The hash of a key

**def add(self, key):**

Adds a key to the HashTable

* **key:** The key being added to the HashTable
* **return:** The position in which the key was inserted

**def remove(self, key):**

Removes a key from the HashTable

* **key:** The key being removed from the HashTable
* **return:** None

**def contains(self, key):**

Searches for the key in the HashTable

* **key:** The key we are searching for
* **return: True if the key is part of the HashTable, False otherwise**

**def get\_position(self, key):**

Finds the position of a key in the HashTable

* **key:** The key we are searching for
* **returnn:** The position and the index of the deque where the key is located

**def \_\_str\_\_(self):**

* **return:** Returns the HashTable as a string

**class ProgramInternalForm:**

Initializes the ProgramInternalForm with an empty list

**def add(self, token, internal\_form):**

Adds a (token, internal\_form) pair to the list

* **token:** The token being added to the list
* **internal\_form:** The token’s internal form
* **return:** None

def \_\_str\_\_(self):

* **return:** Returns the ProgramInternalForm as a string

**class Scanner:**

Initializes a Scanner with a file path to the Token.in file and then reads the tokens this file. This scanner is responsible for finding lexical errors in programs.

**def read\_tokens(self):**

Reads the tokens from the Token.in file and saves them in self.separators, self.operators and self.reserved\_words.

* **return:** None

**def check\_lexical\_errors(self, file\_path, st, pif):**

Checks for lexical errors in a given program and writes the Symbol Table and PIF in separate files

* **file\_path:** The file path of the program being checked
* **st:** The file path of the symbol table output file
* **pif:** The file path of the PIF output file
* **return:** The lexical errors, if any were found

**def tokenize\_function(self, line):**

Finds tokens in a given line

* **line:** The line we need to look for tokens in
* **return:** A list of tokens found in a given line

**def is\_part\_of\_operator(self, char):**

Looks to see if a character is part of an operator

* **char:** The char we are checking
* **return:** True if the character is part of an operator, False otherwise

**def get\_operator\_token(self, line, index):**

Finds an operator token after being given a line and an index

* **line:** The line where we need to search
* **index:** The index where we need to search
* **return:** The token and the index where we found it

**def check\_if\_identifier(token):**

Checks if a token is an identifier using a regex

* **token:** The token we are checking
* **return:** True if the token is an identifier, False otherwise

**def check\_if\_constant(token):**

Checks if a token is a constant

* **token:** The token we are checking
* **return:** True if the token is a constant, False otherwise

**def get\_string\_token(line, index):**

Finds the full string token

* **line:** The line where we are searching
* **index:** The index where we are searching
* **return:** The full string token found at the given line and index

**def check\_if\_string(token):**

Checks if a token is a string

* **token:** The token we are checking
* **return:** True if the token is a string, False otherwise

**class FiniteAutomata:**

Initializes empty variables in order to hold Finite Automata elements. When creating a FiniteAutomata class object a file path needs to be provided to the fa.in file.

**def read\_fa(fa\_file\_path):**

Reads and saves the elements of the Finite Automata. This function is called when the FiniteAutomata class object is first created.

* **fa\_file\_path:** The file path of the fa.in file
* **return:** None

**def is\_dfa():**

Checks if the Finite Automata is a Deterministic Finite Automaton (DFA)

* **return:** True if the FA is a DFA, False otherwise

**def check\_sequence(state, sequence):**

Checks if a sequencce is accepted by the FA by recursively checking each possible route.

* **state:** The current state
* **sequence:** The sequence of the characters that need to be checked
* **return:** True if the sequence is accepted by the FA, False otherwise.

**def \_\_str\_\_():**

Returns the elements of the Finite Automata as a printable string

* **return:** The elements of the FA as a string

Here is an example of how the fa.in file should look like:

A screenshot of a computer program

Description automatically generated

Here is the format in which the fa.in file should be written:

A computer screen shot of a black screen

Description automatically generated