GitHub link: https://github.com/hognogicristina/FLCD/tree/main/Lab4

For my SymbolTable I chose to implement one HashTable which can be used for both the identifiers table and constants table, as well as one single table which contains both of them.

HashTable Class

The HashTable class represents a basic hash table data structure. It is initialized with a specified capacity, which determines the number of buckets in the hash table. Buckets are used to store key-value pairs. This function creates an empty hash table with the specified capacity.

- **get_capacity(self)** - This method allows you to retrieve the capacity of the hash table. It returns the number of buckets in the hash table.

- **hash(self, key)** - The hash method is responsible for calculating the hash value of a given key. If the key is an integer, it calculates the hash using the modulo operation. If the key is a string, it employs a hashing algorithm that iterates through the characters of the string to compute a hash value.

- **contains(self, key)** - The contains method checks if a given key exists in the hash table. It calculates the hash value for the key and then searches for the key within the corresponding bucket. It returns True if the key is found, and False if it is not found.

- **get_hash_value(self, key)** - This function retrieves the hash value for a given key. It distinguishes between integer and string keys and calculates the hash value accordingly.

- **add(self, key)** - The add method adds a key to the hash table. It calculates the hash value for the key and appends it to the corresponding bucket if the key is not already present in the table.

- **get_position(self, key)** - This method returns the position (bucket index) where a given key would be stored in the hash table.

- **__str__(self)** - Return a string representation of the hash table

SymbolTable Class

The SymbolTable class is a data structure used for managing and storing symbols, such as identifiers and constants, in a programming language compiler. It uses a hash table to efficiently organize and retrieve symbols.

- **add_hash(self, name)** - Adds a symbol (name) to the Symbola Table

- **has_hash(self, name)** - Checks if a given symbol (name) exists in the hash table and returns True if found, or False if not found.

- **get_position_hash(self, name)** - Retrieves the position of a specific symbol (name) in the hash table and returns the position or None if the symbol (name) is not found.

- **__str__(self)** - Returns a string representation of the SymbolTable, including details about both the hash tables that includes identifiers and constants in the same symbol table.

Scanner Class

- **read_tokens(self)** - Reads token definitions from a file and populates the reserved_words and tokens lists. Stores the positions of tokens in the token_positions dictionary.

- **set_program(self, program)** - Sets the program source code to be scanned.

- **skip_spaces(self)** - Advances the character index while skipping whitespace characters. Updates the current line number when a newline character is encountered.

- **skip_comments(self)** - Advances the character index while skipping comments (lines starting with "//"). Calls skip_spaces to handle whitespace within and after comments.

- **treat_string_constant(self)** - Attempts to identify and process string constants. Adds string constants to the symbol table, and their positions and hash values to the Program Internal Form (PIF).

- **treat_int_constant(self)** - Attempts to identify and process integer constants. Adds integer constants to the symbol table, and their positions and hash values to the PIF.

- **check_if_valid(self, possible_identifier, program_substring)** - Checks if a possible identifier is valid based on specific criteria. Verifies that the identifier is not a reserved word and adheres to specific patterns in the program.

- **treat_identifier(self)** - Attempts to identify and process identifiers (variable names or keywords). Adds valid identifiers to the symbol table, and their positions and hash values to the PIF.

- **treat_from_token_list(self)** - Attempts to identify tokens from a list of predefined tokens. Adds recognized tokens to the PIF.

- **next_token(self)** - Calls various methods to identify the next token in the program source code. Handles spaces, comments, identifiers, string constants, integer constants, and tokens from the token list. Raises an exception for any unrecognized or invalid tokens.

- **scan(self, program_file_name)** - Reads the program source code from a file. Calls next_token iteratively to tokenize the entire program. Writes the PIF and symbol table to output files. Prints "Lexically correct" if the scanning process completes without errors.

Finite Automaton (FA) Class

The FA class represents a finite automaton, which is a computational model used to recognize patterns in strings. This class is designed to initialize and work with finite automata defined in a configuration file.

- **__init__(self, filename)** - The constructor initializes an instance of the FA class by parsing the given configuration file and storing its contents. It initializes the following attributes: filename (the name of the configuration file), states (a list to store states), alphabet (a list to store the alphabet symbols), transitions (a list to store transitions between states), initial_state (the initial state of the finite automaton), output_states (a list to store output states).

- **init(self)** - This method is called by the constructor to initialize the finite automaton by parsing the configuration file. It reads the file line by line and extracts information about states, alphabet, transitions, initial state, and output states. If the file format is invalid or if any error occurs during parsing, it raises an exception.

- **print_states(self)** - This method prints the list of states in the finite automaton.

- **print_alphabet(self)** - This method prints the list of alphabet symbols used in the finite automaton.

- **print_output_states(self)** - This method prints the list of output states in the finite automaton.

- **print_initial_state(self)** - This method prints the initial state of the finite automaton.

- **print_transitions(self)** - This method prints the list of transitions between states in the finite automaton.

- **check_accepted(self, word)** - This method checks whether a given input word is accepted by the finite automaton. It iterates through the characters of the word and follows transitions between states according to the alphabet. If the finite automaton reaches an output state after processing the entire word, it returns True. Otherwise, it returns False.

- **get_next_accepted(self, word)** - This method finds the longest prefix of a given input word that is accepted by the finite automaton. It iterates through the characters of the word and follows transitions between states according to the alphabet. If the finite automaton reaches an output state, it returns the longest accepted prefix. If no prefix is accepted, it returns None.

Transition Class

The Transition class represents a transition between two states in a finite automaton.

- **__init__(self, from_state, to_state, label) -** from_state (str) (the source state of the transition), to_state (str) (the target state of the transition), label (str) (the label associated with the transition).