https://github.com/danielmalancioiu/LFTC

Documentation

Finite Automaton:
The finite automaton class utilizes a constructor to parse information from the "fa.in" file, which includes details such as states, alphabet, transitions, output states, and the initial state of the automaton.
The checkAccepted method evaluates whether a provided word is accepted by the automaton, returning true if it is accepted.
Additionally, the getNextAccepted method returns the longest accepted prefix of a given word.
This functionality serves to replace regular expressions used in the lexical analyzer for matching integer constants and identifiers.
The data structure employed to store the finite automaton consists of separate lists for states, the alphabet, output states, and transitions.
Transitions are represented by a dedicated class, which encapsulates three variables.
Another variable is employed to maintain the input state.
The checkAccepted method is implemented by initiating the process from the initial state.
For each letter in a given word, the algorithm checks if there exists a transition originating from the current state with a label matching the letter in the word.
If no such transition is found as required, or if the final state is not an output state, the word is deemed not accepted; otherwise, it is accepted.

The Symbol Table comprises three distinct hash tables: one for identifiers, one for integer constants, and one for string constants.
Each hash table is represented as a list where each position is another list, allowing the storage of values that hash to the same position.
Additionally, the hash tables have defined sizes.
An element in the symbol table is identified by a pair of indices: the first index denotes the list in which the element is stored, and the second index indicates the element's position within that list.
The hash function used for integer values is the value modulo the size of the list, while for string constants and identifiers, it's the sum of the ASCII codes of the characters modulo the size of the list.
The implementation of the hash table is designed to be generic.

Operations:
Hash Table

hash(key: int): int – calculates the position in the Symbol Table (ST) for storing the integer constant.
hash(key: string): int – calculates the position in the ST for storing the string constant/identifier, based on the sum of the ASCII codes of their characters.
getSize(): int – returns the size of the hash table.
getHashValue(key: T): int – retrieves the corresponding position in the ST based on the type of the parameter 'key'.
add(key: T): (int, int) – adds the key to the hash table and returns its position if the operation succeeds; otherwise, raises an exception.
contains(key: T): boolean – checks if the given key exists in the hash table.
getPosition(key: T): (int, int) – retrieves the position in the ST of the given key, if it exists; otherwise, returns (-1, -1).

Symbol Table

Consists of three hash tables: one for identifiers, one for string constants, and one for integer constants.
addIdentifier(name: string): (int, int) – adds an identifier and returns its position in the ST.
addIntConstant(constant: int): (int, int) – adds an integer constant and returns its position in the ST.

addStringConstant(constant: string): (int, int) – adds a string constant and returns its position in the ST.
hasIdentifier(name: string): boolean – checks if the given identifier exists in the ST.
hasIntConstant(constant: int): boolean – checks if the given integer constant exists in the ST.
hasStringConstant(constant: string): boolean – checks if the given string constant exists in the ST.

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 has the dual responsibility of constructing the program's internal representation and sy mbol table while simultaneously verifying the lexical correctness of the input program.
The program's internal form is represented as a list, consisting of pairs.
Each pair includes a string denoting a token, which can be a string constant, integer constant, or an identi fier, along with its respective position within the symbol table.
Tokens are designated by (-1, -1) as their position in the symbol table.
The Scanner class also maintains fields for the program string, the symbol table, an index, and the curren t line of the program.

Operations:
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

treatStringConstant(): Boolean - is a method designed to handle scenarios involving string constants in th e program.
This method conducts checks to determine if the string is lexically correct, ensuring there are no invalid ch aracters and that the quotes are properly closed.
If the string constant is deemed valid, it is added to the symbol table (if it doesn't already exist) and incorp orated into the program's internal representation.
Subsequently, the method updates the index and returns true. However, if the string constant is found to be invalid, the method returns false.

- treatIntConstant(): Boolean - is a method tailored to address situations involving integer constants within the program.
This method performs checks to verify the validity of the integer, ensuring that it consists only of numerica l characters.
When a valid integer is encountered, it is included in the symbol table (if it is not already present) and inte grated into the program's internal representation.
The method then proceeds to update the index and returns true.
In contrast, if an invalid integer is encountered, the method returns false.

- checkIfValid(possibleIdentifier: string, programSubstring: string): Boolean – is a method designed to det ermine the validity of a potential identifier.
It accomplishes this by checking whether the possible identifier is either part of a declaration or already ex ists in the symbol table.
If the possibleIdentifier fails to meet any of these conditions, it is considered an invalid token, prompting th e method to return false.
Conversely, if the identifier satisfies one of these conditions, the method returns true.

- treatIdentifier(): Boolean – is a method for handling identifiers in the program.
It validates the identifier, ensuring it starts with an underscore or a letter, contains only letters, digits, and underscores, and is part of a declaration or a previously defined identifier.
If these conditions are met, the method adds the identifier to the symbol table (if not already present) and

the program’s internal form.
It then updates the index and returns true. If the identifier is not valid, it returns false.

- treatFromTokenList(): Boolean – is a method that examines the current program element to determine if it is either a reserved word or another token.
If it is, the method includes it in the program’s internal representation and returns true.
If the element doesn’t match these criteria, the method returns false.

scan(programFileName) – reads the program from the specified file and scans for the next token until the end of the program is reached.
Upon completion, it writes the program’s internal form and symbol table to output files and presents a mes sage confirming the program’s lexical correctness.
In the event an exception is encountered, it displays the exception’s message.