Documentation

GitHub: https://github.com/917tapoimarius/LFTC/tree/main/lab4

I implemented the Symbol Table as a hash table which uses the Separate Chaining Technique to resolve collisions. It has 2 classes, HashNode and SymbolTable. The Program Internal Form uses a Map which stores (key, value) pairs of type (String, Integer).

Class HashNode<K, V>: A class representing a node of chains containing key, value pairs:

Attributes:

key: K – represents the key of the node.

value: V – represents the value stored inside the node.

hashCode: integer – represents the hash code of the node.

next: HashNode<K, V> - represents a reference to the next node in the chain.

Methods:

HashNode(K key, V value, int hashCode): Constructor of the class which initializes the key, value and hash code of the hash node.

params:

key: K – represents the key of the node.

value: V – represents the value stored inside the node.

hashCode: integer – represents the hash code of the node.

returns: -

Class SymbolTable <K>: A class representing the entire symbol table:

Attributes:

THRESHOLD: double – represents the indicator which tells when to resize the symbol table; it is set to 0.75 by default.

symbolTable: ArrayList<HashNode<K, Integer> > - represents the Symbol Table as an ArrayList of HashNodes, used for storing the symbol table's elements (array of chains).

capacity: integer – represents the current capacity of the symbol table; it increases as more elements are added to the symbol table; initialized to 10 by default.

size: integer – represents the current number of elements inside of the symbol table.

Methods:

SymbolTable (): Constructor of the class; initializes the capacity, size and creates empty chains in the table.

params: -

returns: -

size (): Method used for retrieving the number of elements from the symbol.

params: -

returns: an integer representing the size attribute.

isEmpty (): Method used for checking if the symbol table is empty (if the size is equal to 0)

params: -

returns: a boolean, true if the table is empty or false if it contains elements (if the size is greater than 0)

hashCode (K key): Method used for generating a hash code for the given key using a built-in Java function.

params:

key: K – a key.

returns: an integer representing the hash code for the given key.

getSymbolTableIndex (K key): Method used for implementing the hash function to find the index for a key. It generates the hash code for the given key and finds the index for the key corresponding to its hash code by using modulo '%' capacity on the generated code.

params:

key: K – a key.

returns: an integer representing the index of the key.

elementIsEqualToNode (HashNode<K, Integer> node, K key, int hashCode): Method used to check if a key is equal to a node's key and if it has the same hash code.

params:

node: HashNode<K, Integer> - a node which is checked for equality.

key: K – a key which is checked for equality.

hashCode: integer – an integer representing the hash code which is compared with the node's hash code.

returns: a boolean which is true if the key is equal to the node's key and if it has the same hash code or false otherwise.

resize (): Method used for resizing the symbol table when the size is greater than the THRESHOLD. It copies the current symbol table, doubles its capacity, reinitializes it (its size and buckets) and adds back the elements from the copy in the same positions.

params: -

returns: -

remove (K key): Method used to remove a given key from the symbol table. It finds the head of the chain for the given key, then searches for the key in the chain. If it is not found, it returns null. Otherwise, it removes the (key, value) pair from the symbol table and returns the value of the pair which was removed.

params:

key: K – the key of the (key, value) pair, which needs to be removed from the symbol table.

returns: an Integer which represents the value of the (key, value) pair, which was removed from the symbol table, or null, if the key was not found in the table.

get (K key): Method used to retrieve the value for a given key. It finds the head of the chain for the given key, then searches for the key in the chain.

params:

key: K – a key used to retrieve the value for the given key.

returns: an Integer which represents the value for the given key or null if the key does not exist in the symbol table.

add (K key): Method used to add a key to the symbol table. It finds the head of the chain for the given key, then searches for the key in the chain. If it already exists, it returns the value having the key K. Otherwise, it adds it, and its attached value will be the size before it was added (current size – 1) to the symbol table and returns the *value (the old size)*. When the threshold is exceeded, the symbol table is resized.

params:

key: K – the key which needs to be added.

returns: a value V, representing the value which is stored having the key K.

toString(): Overridden method which is used for printing the (key, value) pairs stored in the Symbol Table in a table like manner.

params: -

returns: a String containing a table like representation of the Symbol Table.

Class ProgramInternalForm: A class representing the program internal form:

Attributes:

pif: Map<String, Integer> – represents the program internal form.

Methods:

addOperatorSeparatorReservedWord(String token): Method used for adding to the program internal form an operator, separator or reserved word. An integer value is not given, as these types of special symbols should have value -1 by default.

params:

token: String – the token which needs to be added to the pif.

returns:

addIdentifierConstant(String token, Integer symbolTablePosition): Method used for adding to the program internal form an identifier or a constant

params:

token: String – the token which needs to be added to the program internal form.

symbolTablePosition: String – the value of the token added to the program internal form, which represents the position of the token in the symbol table.

returns:

toString(): Overridden method which is used for printing the (key, value) pairs stored in the program internal form in a table like manner.

params: -

returns: a String containing a table like representation of the Program Internal Form.

Class MyScanner: A class which represents the lexical analyzer.

Attributes:

operators: ArrayList<String> – preinitialized list containing the specific operators of the programming language.

separators: ArrayList<String> – preinitialized list containing the specific separators of the programming language.

reservedWords: ArrayList<String> – preinitialized list containing the specific reserved words of the programming language.

operatorsForPattern: ArrayList<String> preinitialized list containing the specific operators used for building a pattern (regex) to check if a token is an operator; the elements inside of it are different from the *operators* list, as some characters are written with double backlash to define a single backlash, so that they can be used in the regex.

separatorsForPattern: ArrayList<String> – preinitialized list containing the specific separators used for building a pattern (regex) to check if a token is a separator; the elements inside of it are different from the *separators* list, as some characters are written with double backlash to define a single backlash, so that they can be used in the regex.

pattern: Pattern – a pattern created by using a regex which puts between round parentheses both *separatorsForPattern* and *operatorsForPattern*, separates them by "|", and separates each element from both lists by "|". This effectively creates a pattern that matches either separators or operators. The resulting regular expression will look like this: (separator1|separator2|...)|(operator1|operator2|...).

symbolTable: SymbolTable<String> - the symbol table of the compiler.

programInternalForm: ProgramInternalForm – the program internal form of the compiler.

programLines: List<String> - a list of strings representing each line of the read file.

finiteAutomataIdentifier: FiniteAutomata – FiniteAutomata object under construction

finiteAutomataIntegerConstant: FiniteAutomata – FiniteAutomata object under construction

<u>Methods</u>:

MyScanner(String filePath): Constructor of the class. It initializes the *symbolTable* and *programInternalForm,* a BufferedReader which reads from the FileReader's given *filePath*, and the *programLines*, list of Strings which stores the bufferedReader's lines. It also creates instances of type FiniteAutomata for finiteAutomataIdentifier to which it passes the location of the file containing the rules for identifiers and for finiteAutomataIntegerConstant to which it passes the location of the file containing the rules for integers and constants.

params:

filePath: String – the path of the program which will be scanned.

returns: -

isIdentifier(String token): Method used for checking whether a given token is a valid identifier in the programming language. It uses the Finite Automata's checker method (checkSequence(String token)) from the finiteAutomataIdentifier object to validate the identifier.

params:

token: String – the token which needs to be checked.

returns: true if the token is a valid identifier or false otherwise.

isIntegerConstant(String token): Method used for checking whether a given token is a valid integer or constant in the programming language. It uses the Finite Automata's checker method (checkSequence(String token)) from the finiteAutomataIntegerConstant object to validate the identifier.

params:

token: String – the token which needs to be checked.

returns: true if the token is a valid integer or constant or false otherwise.

processString(Iterator<String> tokenizer): Method used to handle the reading and processing of strings within the source code. It builds a string out of the characters it finds and checks if it is outside of the string or not when it encounters a space inside the token list so that it can append it to the final string.

params:

tokenizer: Iterator<String> – iterator for the list of tokens.

returns: -

scan (): Method used to read through each line of the input file by breaking them into individual tokens, and categorizing those tokens based on their types. It takes each line and uses a matcher to apply the precompiled regex on them. It adds each match to the token list and iterates through it. If a token is a space or a tab, it skips it. Otherwise, it tries to categorize it as an operator, separator, keyword, identifier or constant, and adds them to their specific tables, or if it fails, it means that there is a lexical error. It also handles special cases for single-quote and double-quote tokens, adding the next token as a character constant or processing a string by calling the method *processString(Iterator<String> tokenizer)*. After the whole file was checked, it prints if there were errors or not, and if there were errors it prints their line and the possible token which causes it, then it calls the *writeToFiles()* method.

params: -

returns: -

writeToFiles(): Method used for creating files for PIF.out and ST.out, by using FileWriter objects, in which their string format is written.

params: -

returns: -

Class FiniteAutomata: A class which represents the finite automata of the compiler.

Attributes:

states: List<String> - a list of strings which represent the states of the Finite Automaton *alphabet*: List<String> - a list of strings which represent the alphabet of the Finite Automaton.

transitions: HashMap<Pair<String, String>, String>- a HashMap which contains <Key, Value> pairs. The Key is formed by a Pair of two Strings which represents the current state and the symbol through which a next state can be reached and the Value is a String which represents the next state that can be reached.

initialState: String – a string which represents the initial state of the Finite Automaton

finalStates: List<String> - a list of strings which represent the final states of the Finite Automaton.

isDeterministic: boolean – a Boolean value which indicates whether the given Finite Automaton is deterministic (DFA) or not.

Methods:

FiniteAutomata(String filePath): Constructor of the class. It reads from a file (from the given filePath) using a buffered reader. It reads the first line containing the states list, the second line containing the alphabet, the third which contains the initial state, the fourth which contains the final states list, and then it reads all the remaining lines representing the transitions. The transitions are written in the following form: currentSteate,symbol,nextState. The buffered reader is closed and then the checkDFA() method is called so it can determine whether the given Finite Automaton is deterministic or not, and assigns the returned value (true/false) to the isDeterministic attribute.

params:

filePath: String - the path to the file containing the Finite Automata's rules from which the constructor will read its input.

returns: -

checkSequence(String sequence): Method used for checking whether a sequence is valid for the defined Finite Automaton (valid). If the Finite Automaton is not deterministic, the method will return an appropriate message. Otherwise, it goes through each character of the given sequence and tries to find a transition from the current state to another state by the current character (the current character is used as a symbol). If no next state is found, it means that the transition is invalid. Otherwise, the current state changes its value to the next state and the algorithm continues until no character remains. If the final state list contains the current state (the last obtained state), it means that the sequence is accepted by the DFA (valid). Appropriate messages are returned for both cases (valid and invalid).

params:

sequence: String – the sequence which needs to be validated.

returns: a string which based on the result of the checker can be: "Language is not DFA!", "Invalid transition for input: {i}" (where 'i' represents the character for which a transition was not found), "Sequence accepted by DFA!" or "Sequence rejected by DFA!"

isValidSequence(String sequence): Method used for determining whether the given sequence is valid for the defined Finite Automaton. It calls the *checkSequence(String sequence)* method and based on the message (string) it received, it verifies if it is valid or not.

params:

sequence: String – the sequence which needs to be validated.

returns: true if the sequence is valid or false otherwise.

statesToString(): Method used for printing the states of the Finite Automaton.

params: -

returns: A string containing the states of the Finite Automaton.

*alphabetToString():*Method used for printing the alphabet of the Finite Automaton.

params: -

returns: A string containing the alphabet of the Finite Automaton.

transitionsToString(): Method used for printing the transitions of the Finite Automaton.

params: -

returns: A string containing the transitions of the Finite Automaton.

initialStateToString(): Method used for printing the initial state of the Finite Automaton.

params: -

returns: A string containing the initial state of the Finite Automaton.

finalStatesToString(): Method used for printing the final states of the Finite Automaton.

params: -

returns: A string containing the final states of the Finite Automaton.

checkDFA(): Method used for checking whether a Finite Automaton is deterministic or not (DFA). It iterates through the transitions' list and checks if a state can reach more than one state through the same symbol. If it can, then the Automaton is not deterministic.

params: -

returns: true if the Finite Automaton is deterministic or false otherwise.

OBSERVATION:

FA_BNF.txt:

<FA>::=<states><alphabets><initialState><finalState><transitions>

<states>::=<state>|<state><states>

<alphabets>::=<alphabet>|<alphabet><alphabets>

<initialState>::=<state>

<finalState>::=<state>|<state><finalState>

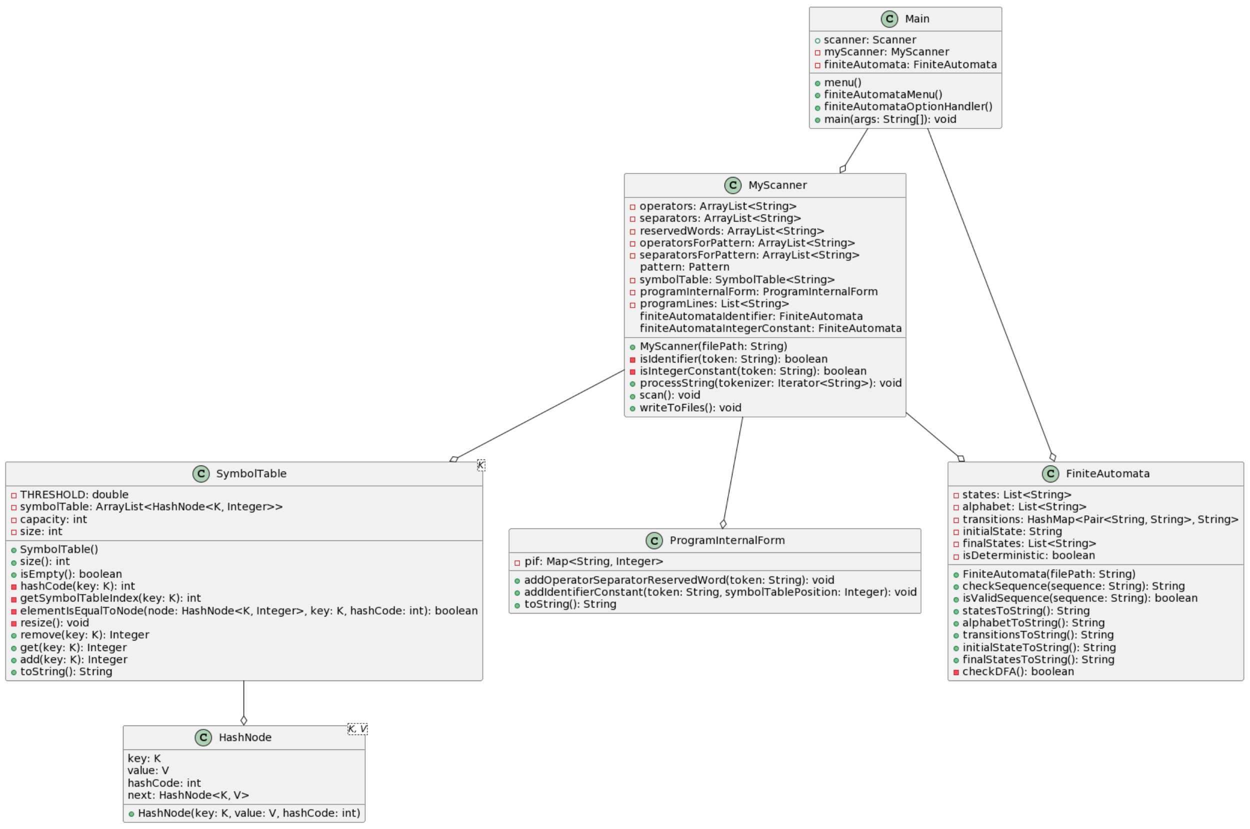
<transitions>::=<transition>|<transition>"\n"<transitions>

<transition>::=<state>,<alphabet>,<state>|

<state>::=a|b|c|...|z

<alphabet>::=a|b|c|...|z|0|1|...|9

Class Diagram:


Main
scanner: Scanner
myScanner: MyScanner
finiteAutomata: FiniteAutomata
menu()
finiteAutomataMenu()
finiteAutomataOptionHandler()
main(args: String[]): void
MyScanner
operators: ArrayList<String>
separators: ArrayList<String>
reservedWords: ArrayList<String>
operatorsForPattern: ArrayList<String>
separatorsForPattern: ArrayList<String>
pattern: Pattern
symbolTable: SymbolTable<String>
programInternalForm: ProgramInternalForm
programLines: List<String>
finiteAutomataIdentifier: FiniteAutomata
finiteAutomataIntegerConstant: FiniteAutomata
MyScanner(filePath: String)
isIdentifier(token: String): boolean
isIntegerConstant(token: String): boolean
processString(tokenizer: Iterator<String>): void
scan(): void
writeToFiles(): void
SymbolTable
K
THRESHOLD: double
symbolTable: ArrayList<HashNode<K, Integer>>
capacity: int
size: int
SymbolTable()
size(): int
isEmpty(): boolean
hashCode(key: K): int
getSymbolTableIndex(key: K): int
elementIsEqualToNode(node: HashNode<K, Integer>, key: K, hashCode: int): boolean
resize(): void
remove(key: K): Integer
get(key: K): Integer
add(key: K): Integer
toString(): String
ProgramInternalForm
pif: Map<String, Integer>
addOperatorSeparatorReservedWord(token: String): void
addIdentifierConstant(token: String, symbolTablePosition: Integer): void
toString(): String
FiniteAutomata
states: List<String>
alphabet: List<String>
transitions: HashMap<Pair<String, String>, String>
initialState: String
finalStates: List<String>
isDeterministic: boolean
FiniteAutomata(filePath: String)
checkSequence(sequence: String): String
isValidSequence(sequence: String): boolean
statesToString(): String
alphabetToString(): String
transitionsToString(): String
initialStateToString(): String
finalStatesToString(): String
checkDFA(): boolean
HashNode
K, V
key: K
value: V
hashCode: int
next: HashNode<K, V>
HashNode(key: K, value: V, hashCode: int)