Pair.java

Pair(F first, S second)

Initializes the two elements of the custom pair.

F getFirst()

Returns the first element of the pair.

S getSecond()

Returns the second element of the pair.

String toString()

Provides a string representation of the pair.

HashTable.java

HashTable(int size)

Initializes the size of the hash table and creates an ArrayList of ArrayLists to store the elements.

O(capacity) - Iterates over the size in order to initialize the hash table.

int hash(T key)

Calculates the hash index for a given key based on its type (Integer or String).

O(1) - Involves simple operations and doesn't depend on the size of the hash table.

void add(T key)

Adds a key to the hash table. It calculates the hash index for the given key and checks if the key is already present at that index. If not, it adds the key to the appropriate inner ArrayList.

On average has a O(1) complexity since it involves calculating the hash index and adding the element to the ArrayList at that index. However, if there are collisions, the worst-case time complexity could be O(n).

int search(T key)

Searches for a key in the hash table. It calculates the hash index for the given key and checks if the key is present at that index. It returns a pair formed by the position of the list in which the key was found and the position that it has in that list.

On average has a O(1) complexity since it involves calculating the hash index and checking if the key is present. However, if there are collisions, the worst-case time complexity could be O(n).

String toString()

Provides a string representation of the hash table.

O(n) - Iterates through the entire hash table to access each element.

LexicalAnalyzer.java

LexicalAnalyzer()

Initializes the token lists, the ST and the PIF. It also calls the function that populates the token lists.

void initializeTokenLists()

Populates the reservedWords, operators and separators lists using the file ‘token.in’. The content is read line by line and a counter keeps track of the current token type in order to add the token to the appropriate list. The space character is also added in order to be used as a separator for detecting tokens.

String getSymbolTable()

Returns a string representation of the symbol table.

String getProgramInternalForm()

Returns a string representation of the program internal form.

Boolean isReservedWord(String token)

Checks if the token is present in the reservedWords list.

Boolean isOperator(String token)

Checks if the token is present in the operators list.

Boolean isSeparator(String token)

Checks if the token is present in the separators list.

Boolean isIdentifier(String token)

Checks if the token matches the regex for an identifier. Updated version uses the Finite Automata for identifier.

Boolean isConstant(String token)

Checks if the token matches the regex for a constant (integer, character or string). Updated version uses the Finite Automata for integer constant.

int getTokenCode(String token)

Returns the code associated with the token. If it is a reserved word, an operator or a separator, it takes the number of the line on which it appears in the ‘token.in’ file, by using the position in the lists and adding the appropriate number. If it is an identifier it returns -1, and if it is a constant it returns -2.

ArrayList<Pair<String, Integer>> detectTokensByChar(String inputFile)

Returns a list of pairs formed by the found tokens along with the line on which they appear in the input file. The content of the program is read line by line and a line number is incremented appropriately. If the current character is a separator, then it is added to the tokens list (exception is the space character). Any previously read characters are added to the list as a single token. If the current character is an operator and the next character is not an operator or is a '-', then a check is made to see if the operator is made out of one character or two characters. If the previous character along with the current character form an operator, then it is added to the tokens list, as well as a token formed by any previously read characters. If not, the current character is added to the tokens list (except for when it is a '-' but the previous token is not an identifier or a constant), as well as a token formed by any previously read characters.

void scanProgram(String inputFile, String outputFileST, String outputFilePIF)

Implements the scanning algorithm by iterating through the tokens found in the input program and handling the possible cases. If the current token is a reserved word, an operator or a separator, then a pair formed by the token code and a pair (-1,-1) is added to the program internal form. If the current token is an identifier or a constant, then it is added to the symbol table and a pair formed by the token code and the position from the symbol table is added to the program internal form. If the current token is still not categorized, then it is a lexical error and an appropriate message is added. In the end, the symbol table and the program internal form are printed in their associated output files.

<u>Transition.java</u>

Transition(String sourceState, String alphabetValue, String destinationState)

Initializes the transition with a source state, an alphabet value and a destination state.

String getSourceState()

Returns the source state of the transition.

String getAlphabetValue()

Returns the alphabet value of the transition.

String getDestinationState()

Returns the destination state of the transition.

String toString()

Provides a string representation of the transition.

<u>FiniteAutomata.java</u>

FiniteAutomata(String inputFile)

Initializes the finite automata elements. It also calls the function that populates the initial states, states, final states, alphabet and transitions.

String getInitialState()

Returns the initial state of the finite automata.

List<String> getStates()

Returns the states of the finite automata.

List<String> getFinalStates()

Returns the final states of the finite automata.

List<String> getAlphabet()

Returns the alphabet of the finite automata.

List<Transition> getTransitions()

Returns the transitions of the finite automata.

void initializeElements()

Populates the initial states, states, final states, alphabet and transitions from the input file. The content is read line by line, with the first four lines representing the first four elements of the finite automata and the remaining lines representing the transitions.

Transition findNextTransition(String currentState, String inputSymbol)

Iterates through the transitions of the finite automata and returns the transition that matches the source state and the alphabet value with the given input. If no transition is found it returns null.

boolean isAcceptedByFA(String input)

Checks whether a given input string is accepted by the finite automata. It iterates through each character in the input, finding the next transition based on the current state and input symbol. If a valid transition is found, it updates the current state. After processing the entire input, the method returns true if the last state is present in the final states, and false otherwise.

BNF format of FA.in:

file ::= initial state "\n" states "\n" final states "\n" alphabet "\n" transitions

initial state ::= character

states ::= initial state { ", " character } ", " final states

final states ::= character { ", " character }

alphabet ::= character { ", " character } transitions ::= character “, ” character “, ” character {“\n” character “, ” character “, ” character }

character ::= letter | digit

letter ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"

digit ::= "0" | "1" | ... | "9"