https://github.com/912-CUCONU-MARIA/FLCD_LAB

Class: FA

The FA class represents a finite automaton. It initializes the automaton from a given file, storing its states, alphabet, and transitions. The class provides functionalities to check if a given word is accepted by the automaton, along with methods to print and access the automaton's attributes.

Attributes:
initialStates: A list of strings representing the initial states of the automaton.

finalStates: A list of strings representing the final states of the automaton.

all_states: A list of strings representing all states of the automaton.

alphabet: A HashMap representing the alphabet of the automaton. Each entry in the HashMap corresponds to an element of the alphabet.

transitions: A HashMap with keys as Pair<String, String> and values as strings, representing the transitions of the automaton.
example: transition p0->element->p1 is represented as a HashMap entry of
key= (p0,element) and value=p1

filename: A string representing the name of the file from which the automaton is initialized.

Constructor:

FA(String filename): Initializes the FA object and reads the automaton configuration from the specified file.

Methods:

initFA(): Reads the automaton configuration from the file and initializes the states, alphabet, and transitions of the automaton.

checkAccepted(String word): Checks if the given word is accepted by the automaton. Returns true if the word is accepted, otherwise false.

getInitialStates(): Returns the list of initial states.

getFinalStates(): Returns the list of final states.

getAll_states(): Returns the list of all states of the automaton.

printAlphabet(): Prints the alphabet of the automaton.

printTransitions(): Prints the transitions of the automaton

EBNF Documentation of FA input file

file.in:=initial_states final_states all_states alphabet transitions
initial_states:="initial_states:" states_list
state:="p" index
index:= digit {digit}
digit := "0" | "1" | "2" | ... | "9"
states_list:={state”,”} state
final_states:="final_states:" states_list
all_states:="all_states:" states_list
alphabet:="alphabet:" alphabet_list
alphabet_list:={element”,”} element
element:= "_" | "digit" | "non_zero_digit" | "letter"
transitions:="transitions:"transition_list
transition_list:={transition”,”} transition
transition:="(" state "," element "," state ")"

Class: Scanner

The Scanner class scans the given input program and builds the program internal form and the symbol table for it, while also checking if the input program is lexically correct.

Attributes:
symbolTable: An instance of the SymbolTable class

pif: A list of pairs representing the program internal form, where each pair consists of a string representing an identifier “identifier”, a constant “constant” or a token and its position in the symbol table.

program: A string representing the entire program to be scanned.

tokens: A list containing the tokens the scanner can recognize.

index: The current position of the scanner in the program string.

currentLine: The current line number being scanned in the program.

Constructor:
Scanner(): Initializes the Scanner object and reads the tokens from the "token.in" file.

Methods:

readTokens(): Reads tokens from the "token.in" file and populates the tokens list. Sorts the tokens by length in descending order for optimized matching.

setProgram(String program): Setter method to set the program attribute.

scanProgram(String programFileName): Reads the input program file and sets the program attribute. Invokes the nextToken method as it parses the whole program string to tokenize the program and populates the program internal form (PIF) and symbol table.At the end, if the program is lexically correct, writes the PIF and symbol table to output files. In case of lexical error, it displays the error and stops parsing.

nextToken(): Extracts the next token from the program. Skips over whitespaces and newlines. Checks for string constants, integer constants, predefined tokens, and identifiers. Throws an exception in case of lexical error.

treatIdentifier(): Tries to match an identifier from the current position in the program. Adds the identifier to the PIF and symbol table if it's valid. Returns true if an identifier is matched, otherwise false.

treatStringConstant(): Tries to match a string constant from the current position in the program. Adds the string constant to the PIF and symbol table if it's valid. Throws an exception if an unclosed string literal is encountered. Returns true if a string constant is matched, otherwise false.

treatIntConstant(): Tries to match an integer constant from the current position in the program. Adds the integer constant to the PIF and symbol table if it's valid.Throws an exception if an invalid integer token is encountered. Returns true if an integer constant is matched, otherwise false.

treatFromTokenList(): Tries to match any token from the predefined tokens list. Adds the matched token to the PIF. Returns true if a token from the list is matched, otherwise false.

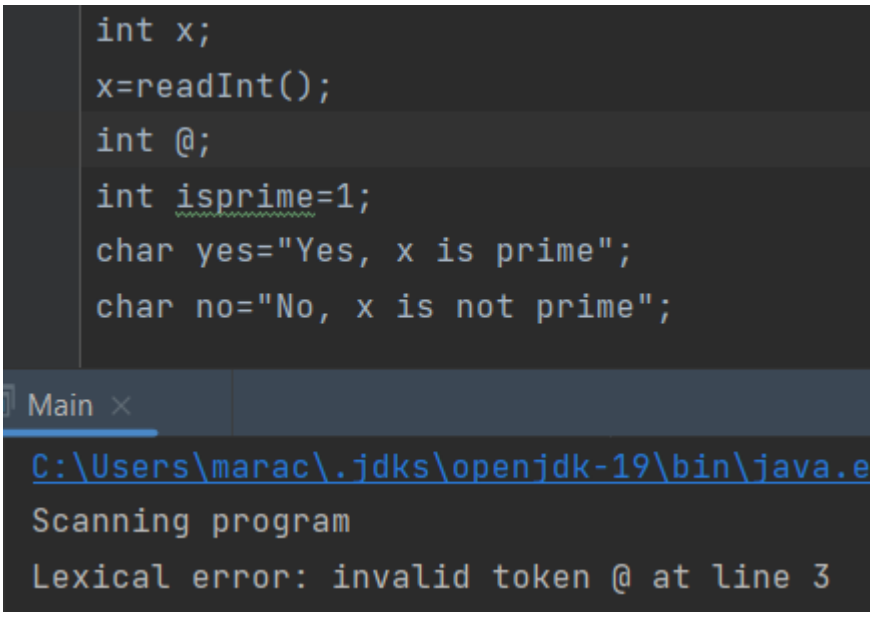
Class: ScannerTest : provides testing functionalities for the `Scanner` class. It tests the scanner against various input program files to evaluate its behavior and correctness.

Attributes:
program1: A string representing the file name of the first test program.
program2: A string representing the file name of the second test program.
program3: A string representing the file name of the third test program.
program1err: A string representing the file name of a test program that is expected to contain errors.

Methods:
testScanner(String program): Creates a new Scanner object. Prints a message indicating the start of the scanning process. Invokes the scanProgram method of the Scanner class to scan the input program.

testAllPrograms(): Tests the Scanner class against all predefined test program files (program1, program2, program3, and program1err).

Example of ScannerTest testScanner(program) method output in case of different lexical errors



Documentation from previous lab2, in case needed:
Class: SymbolTable

The SymbolTable is composed from 2 separate hash tables, one for identifiers, one for integer and string constants. Each hash table is represented by a list of lists, in order to be able to handle hashing collisions.An element from the symbol table has as position a pair of indices, the first one being the index of the list in which the element is stored and the second one being its actual position in that list.

Attributes:
size (Integer): Size of the symbol table.
identifiers (HashTable<String>): Hash table to store identifiers.
constants (HashTable<String>): Hash table to store integer and string constants.
Constructor:
SymbolTable(Integer size): Initializes a new instance of the SymbolTable with the specified size and initializes the hash tables.

Methods:
addIdentifier(String identifier) -> Pair<Integer, Integer>: Adds an identifier to the identifiers hash table and returns its position.

addConstant(String constant) -> Pair<Integer, Integer>: Adds a constant to the constants hash table and returns its position.

getPositionIdentifier(String name) -> Pair<Integer, Integer>: Retrieves the position of a given identifier in the identifiers hash table.

getPositionConstant(String constant) -> Pair<Integer, Integer>: Retrieves the position of a given constant in the constants hash table.

containsIdentifier(String name) -> boolean: Checks if the given identifier exists within the identifiers hash table. Returns true if found, false otherwise.

containsConstant(String constant) -> boolean: Checks if the given constant exists within the constants hash table. Returns true if found, false otherwise.

toString() -> String: Overridden method that provides a string representation of the symbol table.

Class: HashTable<T>

The HashTable class provides a basic implementation of a hash table. The hash function is value modulo the size of the list, for integer values, and the sum of the ASCII codes of the characters modulo the size of the list, for string constants/identifiers.

Attributes:
buckets (ArrayList<ArrayList<T>>): A list of "buckets" which are used to store items. Each "bucket" is itself a list.

size (Integer): Size of the hash table.

Constructor:
HashTable(Integer size): Initializes a new instance of the HashTable with specified size and creates the buckets.
Methods:
getSize() -> Integer: Returns the size of the hash table.

hash(Integer key) -> Integer: Hash function for integer keys based on absolute Integer value modulo the size of the list.

hash(String key) -> Integer: Hash function for string keys based on the sum of ASCII codes of the characters.

getHashValue(T key) -> Integer: Determines the appropriate hash function to use based on the type of key and computes its hash value.

add(T key) -> Pair<Integer, Integer>: Adds a key to the hash table and returns its position if the operation is successful; otherwise, it throws an exception.

contains(T key) -> boolean: Checks if the given key is present in the hash table.

getPosition(T key) -> Pair<Integer, Integer>: Retrieves the position of a given key in the hash table. If the key is not present, returns (-1, -1).

toString() -> String: Overridden method that provides a string representation of the hash table.

Class: Pair<T1, T2>

The Pair class represents a tuple with two values.

Attributes:
first (T1): The first element in the pair.
second (T2): The second element in the pair.
Constructors:
Pair(T1 first, T2 second): Initializes a new instance of the Pair with specified values.

Pair(): Default constructor.

Methods:
getFirst() -> T1: Returns the first value in the pair.

setFirst(T1 first): Sets the first value in the pair.

getSecond() -> T2: Returns the second value in the pair.

setSecond(T2 second): Sets the second value in the pair.

toString() -> String: Overridden method that provides a string representation of the pair.

equals(Object o) -> boolean: Overridden method to check for equality of two pairs based on their content.

hashCode() -> int: Overridden method to compute hash code based on the pair content.