krisztinahorvath/Formal-Languages-and-Compiler-Design (github.com)

Documentation

FA:

The FA class is a finite automaton which implements operations on an automaton read from an input file. The input file contains states (stored as a list of strings), the initial state (stored as a string), final states(stored as a list of strings), the alphabet(stored as a list of strings), and the transitions(stored as a list of Transitions) of the automaton.

Functions:

- readFA(filePath: String): void – reads the file and stores the states, initial state, final states, alphabet and transitions in their corresponding lists
- checkAccepted(sequence: String): Boolean – checks if a given string is accepted by the FA by going through the string character by character and checking if from the initial state a final state can be reached
- printStates(): void – prints the states of the FA
- printAlphabet(): void – prints the alphabet of the FA
- printTransitions(): void – prints the transitions of the FA
- printInitialState(): void – prints the initial state of the FA
- printFinalStates(): void – prints the final states of the FA

Transition:

A class that is used for representing a transition, having the following fields: from (initial state), to (final state) and element (from the alphabet). A transition is of the form (from, to, element).

Input file format for FA in BNF:

digit ::= “0” | ”1” | ... | “9”

letter ::= “a” | “b” | … | “z” | “A” | “B” | … | “Z”

character ::= letter | digit

firstLine ::= “states” “=” “{“ character { “,” character } “}”

secondLine ::= “initial_state” “=” “{” character “}”

thirdLine ::= “final_states” “=” “{“ character { “,” character } ”}”

fourthLine ::= “alphabet” “=” “{“ character { “,” character } “}”

fifthLine ::= “transitions” “=” “{“ triple { “;” triple } “}”

triple ::= “(“ character “,” character “,” character “)”

inputFile ::= firstLine “\n” secondLine “\n” thirdLine “\n” fourthLine “\n” fifthLine

Hash table:

- hashFunction(key: T): int – returns the position in the Symbol table of the list in which the value will be added
- add(key: T): (int, int) – adds the key to the hash table and returns its position on success, otherwise returns null
- contains(key: T): boolean – returns true/false if the key is in the hash table or not
- getPosition(key: T): (int, int) – returns the position in the hash table of the given key
- toString() (overridden method) – returns the string representation of the hash table
- for all the above functions the average-case complexity is O(1), since we can access directly the elements by knowing the position they hash to and we assume that there are no collisions and the hash function is well-distributed. The worst-case complexity for adding and searching is O(n), when all keys hash to the same bucket and there are n keys in it

Symbol table:

The Symbol Table is composed of one hash table with separate chaining. The hash table is represented as a list and every position is another list, to be able to store values that hash to the same position. An element of the symbol table has as position a pair of indices, the first one is the index of the bucket/inner list in which the element is stored, and the second one is the actual position inside the list. The hash function is value modulo the number of buckets in the list, the value is computed by a built-in function. The implementation of the hash table is generic.

Functions:

- has one hash table
- addSymbol(identifier: String): (int, int) – adds a symbol and returns its position in the symbol table
- hasSymbol(identifier: String): boolean – returns true if the symbol is in the symbol table, false otherwise
- getSymbolPosition(identifier: String): (int, int) – returns the position of the symbol in the symbol table
- toString() (overridden method) – returns the string representation of the symbol table and its hash tables
- since all the above functions call and return hash table functions, their average-case complexity is O(1) and worst-case complexity for adding and searching a symbol is O(n) using the same reasoning

Lexical Analyzer:

The Lexical Analyzer is composed of two symbol tables, one for identifiers and one for constants, the program internal form table that holds a pair of an integer value for the code of the token and another pair of integers to store the position from the symbol table, and three lists for reserved words, operators, and separators. It first reads a file with tokens and it adds the value and the position of the reserved words, operators and separators to their respective lists. Then it processes a file by reading it character by character to form atoms until the first separator, adds the separator to the PIF and tries to match the previous token to a reserved word, operator, identifier and integer or string constant. It matches to an identifier or an integer constant by using the FA class with the respective input files, and for matching to a string constant, a regex is used. If the matching fails then it will be considered as a lexical error and it will be appended to an output string.

Functions:

- readTokenFile(filePath: String): void – reads an input file line by line and fills the reserved word, separator and operator lists with the correct token and its position in the file.
- getReservedWordCode(word: string): int – returns the position of the word in the reservedWords list, returns -1 if not found
- getSeparatorsCode(sep: char): int – returns the position of a given separator in the separators list, return -1 if not found
- getOperatorsCode(op: String): int – returns the position of a given operator in the operators list, returns -1 if not found
- writeToFile(inputFilePath: String): void – writes to separate files the PIF table and the two symbol tables
- processFile(filePath: String): String – reads a file character by character and builds an atom until the first separator is read, then the separator is added to PIF and the previous atom is matched to either a reserved word, an operator, an identifier, an integer constant or a string constant. For detecting identifiers and integer constants it uses the FA class with the respective input files, and for matching string constants it uses regular expressions and it makes sure that beginning and end quotes were read for it to be valid, if the atom wasn't matched, then it is considered a lexical error and a string is appended with the error and this is returned at the end of the function. If the file was lexically correct it returns that message, otherwise it returns the errors with their line and the respective token.