https://github.com/andreeall00/LFTC

I have a class **SymbolTable**, represented as a binary search tree, with an inner class Node.

The SymbolTable object has a root of type Node, and a position (pos).

A Node has a left and right Node, a key which is the value I take into consideration when adding a new Node into the tree, and a position (pos).

When adding a Node, I start by considering the current node to be the root of the tree. When the current node is empty, I simply add the new Node having the given key and the position 0, increment the position in the SymbolTable (pos), and return its position. When the current node is not empty, i check if its value is the one I'm looking for. If it is, then i return its position. If it's not, then i check if the key of the current node is alphabetically greater than the key I'm searching for. If it is then I'll search in the left side of the tree (the left Node of the current one), and if it's not then I'll search in the right side of the tree (the right Node of the current one).

I continue this way recursively until I find the key I'm looking for, returning its existing position, or until I get to an empty node where I add a new one containing the key and returning its new position.

The class **Scanner** has a SymbolTable and a PIF, which is represented here as a list of tuples [eq: (=, 1)]. It has a list for reserved words, one for operators and one for separators. It also contains a regex for identifiers, one for integers, one for characters, one for strings and one for booleans.

Identifier Regex: "^[a-zA-Z]([a-zA-Z]|_|[0-9])*$" - an identifier can start with an uppercase/lowercase letter and contain any number of letters, digits and '_'.

Integer Regex: "^(([+-]?[1-9][0-9]*)|0)$" - an integer can be 0, or a signed or unsigned number that starts with a non-zero digit and continues with any number of digits (including 0)

Character Regex: "^'([a-zA-Z]|[0-9])'$" - a character has to start with a single quote ('), contain a single letter or a single digit, and end with another single quote

String Regex: "^\"([a-zA-Z]|[0-9])([a-zA-Z]|[0-9])*\"$" - a string has to start and end with quotes ("…"), and can contain one or more letters and digits

Boolean Regex: "^(T|F)$" -- a Boolean can be T for true and F for false

There is a getTokens function which receives a string representing a line of code, and parses each character, constructing tokens. A number of consecutive characters are considered to be a

token when we get to a separator, an operator, or the end of the line. If there are opened “ or ‘ and they are not closed by the end of the line, it can’t be continued on the next one.

The scan function parses the given file and gets the tokens from each line in the file. For each token it checks if it is a reserved word/operator/separator, and if so, it adds it in the pif with a position of -1. If the token is an identifier, it adds (add if it doesn’t exist, or search if it does) it in the symbol table and then in the pif as an ‘id’ with the position received from adding it in the symbol table. If the token is any of the defined constants above, it checks to match the regex and if it does it adds in the symbol table and then in the pif as a ‘const’ with the position from the symbol table. If the token doesn’t match any of the above cases, then it is a lexical error at it gets printed along with the line it occurred at. At the end the symbol table and the pif are written in separate files.

The class **FiniteAutomaton** has a finite set of states (Q), a list containing the finite alphabet (E), a map of transitions (delta), an initial state (q0), a set of final states (F), and the input file containing all the data:

FiniteAutomaton: {

file – strings

Q – list of strings

E – list of strings

delta – map having as key a tuple containing the state from which the transition begins, and the value needed to proceed to the next state, and as value the state in which you end up

q0 – string

F – list of strings

}

The FA.in file has 5 rows:

1. The states, divided by column:

   states = letter {“,” letter}

2. The alphabet, divided by column:

   alphabet = character {“,” character}

   character = letter | ”_” | digit | “+” | “-"

3. The transitions, written as: p+0=q, where p and q are states, and from p, using 0, we get to q:

   transitions = transition {“,” transition}

   transition = letter “+” character “=” letter

4. The initial state:

   initialState = letter

5. The final states, divided by column:

   finalStates = states

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

digit = “0” | “1” | … | “9”

The isAccepted function from FiniteAutomata checks if a sequence is accepted by the finite automata. It starts from the initial state and, for each character from the sequence, moves to the next state if there exists a transition from the current state to another one using the current character. If there doesn’t exist such a transition, or if the state we end up in after parsing the entire sequence is not a final state, it means the sequence is not accepted by the fa, otherwise, it is.