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.

The implemented **Parser** algorithm uses the recursive descendent method. The class **Config** has the current state of the parsing (s), the position of the current symbol in the input sequence (i), the working stack, which stores the way the parse is build (alpha), and the input stack, which is part of the tree to be build (beta):

Config: {

s – string (one of : “q”-NORMAL_STATE, “b”-BACK_STATE, “f”-FINAL_STATE, “e”-ERROR_STATE)

i – integer

alpha – list of productions/terminals

beta – list of terminals/non-terminals

}

The recursive descendant method starts with the default configuration that is in a normal state (s=NORMAL_STATE), the index points at the first symbol from the input sequence (i=0), the working stack is an empty list (alpha=[]), and the input stack is a list containing the start symbol

(beta=[startSymbol]); and keeps going until it reaches the FINAL_STATE or the ERROR_STATE. It has 6 important steps:

1. If the current state is normal, the input stack is empty and position of the current symbol in the input sequence points outside of the sequence (the entire sequence has been parsed), then there is the **SUCCESS** step: the state of the configuration becomes final, and the algorithm ends with no errors => No syntax errors
2. If the current state is normal, and the first element of the input stack is a non-terminal, there is the **EXPAND** step: the nonterminal is removed from the input stack, the first production of the nonterminal is added at the end of the working stack, and each element of the right-side of the production is added at the beginning of the input stack
3. If the current state is normal, and the first element of the input stack is a terminal equal to the current symbol from the input sequence, there is the **ADVANCE** step: the terminal is added at the end of the working stack, removed from the beginning of the input stack, and the index moves to the next symbol from the input stack
4. If the current state is normal, and the first element of the input stack is a terminal NOT equal to the current symbol from the input sequence, there is the **MOMENTARY INSUCCESS** step: the state becomes back state
5. If the current state is back, and the last element of the working stack is a terminal, there is the **BACK** step: the index is moved to the previous element from the input sequence, and the last element of the working stack is removed and added to the front of the input stack
6. If the current state is back, and the last element of the working stack is a non-terminal, there is the **ANOTHER TRY** step:
    a. if there exists another transaction from the current non-terminal symbol that has not been tried before, the state becomes normal, the new production is added at the end of the working stack, and the elements of the right-side of the production are added at the beginning of the input stack
    b. if there doesn't exist another transaction from the current non-terminal that was not tried before, and the index points at the beginning of the input sequence and the last production from the end of the working stack starts from the start symbol, then the state becomes error, and the algorithm ends with errors => Syntax errors
    c. if a and b don't happen, the last element of the working stack is removed and added back to the front of the input stack

The **ParserOutput** class contains the productions – a list of productions in the order in which they need to be applied (returned by the Parser algorithm), and a table – an object of type **Table**, that has a list of rows, where a row is on object of type **Row**, having an index – integer,

an info – string (the symbol), a parent – integer (the index of the parent symbol), and a right sibling – integer (the index of the right sibling, the symbol in front of the current one if it exists).

Starting from the list of productions, it takes the nonterminal from the left-side of the production, and, if the table is empty, a new row is added, having the index 1, the non-terminal symbol as the info, and the parent and right siblings 0, otherwise it means it has already been added, and it takes its index from the table. Then, it considers all the elements from the left-side of the production as the children of the current nonterminal. Every child is added as a new row in the table, having the next index, its symbol as the info, the index of the current nonterminal as a parent and the index of the previous child as right sibling, or 0 if it is the first child.