ECOTE - final project

Semester: 2022, summer

Author: Alessandro Cavallotti (K-6416)

Subject:

Write a program reading regular expressions, then constructing directly DFA using a syntax tree for the expression. Show syntax tree and all functions. The program should check if input strings are generated by this expression.

Read regular expressions from the text file or from the command line.

Use the syntax as given on the lecture:

Closure *

Positive closure + (optional)

Or operator |

To group subexpressions ()

E.g.: (a | b)* or a | ba*

I. General overview and assumptions

The program has the goal to generate the syntax tree from a given regular expression and its relative DFA graph representation. Consequently, the resulting structure will be used to check if input strings match the regex.

The program starts with the adding of concatenation symbols to the expression and performs the infix to postfix conversion. While doing this operation, the program looks for syntax errors that if found the execution is stopped and an error message is displayed.

The special characters are the that will be used as filters are '|' the or operator, '(', ')' to group subexpressions and '*' closure that points out that the expression before that can be present from 0 to many times. The concatenation symbol that is added is the '.' and this character cannot be used inside the regular expression. There are no other constraints for the input of the regexp.

The program has five main task to complete that must be done consecutively:

1. Adding of the concatenation symbol
   The '.' is added to the output expression R generated by adding one char at the time from the inputted expression by the user, only if all the following condition are verified:
   a. lenght(R) > 0
   b. the next char that has to enter R is not ')', '*' or '!'
   c. the last char in R is not '(' or '|'
2. Conversion from Infix to Postfix notation

The conversion is performed using a personal implementation of the *Shunting Yard Algorithm*

3. Generation of the syntax Tree
   The construction is made from the leaf to the root node (bottom-up), scanning the postfix expression and putting each symbol on a dynamic stack that links the two objects on top of the stack and replacing them with the new node.
4. Generation of the transition table
   After computing the first, last and the follow table, it is possible to generate the transitional table for each input symbol just by computing the possible moves starting from a state A = first(root), and performing the union of the follow for each input symbol.
5. Evaluation of the input string
   Using the transition table the evaluation of the string is performed easily just by following the states given a specific input character. If the algorithm ends in the final state or to an empty state, then it is an accepting state, so the string matches the regular expression.

The program expect the input from the keyboard and the regular expression must end with the '#' symbol. After a pre-elaboration of the regular expression the program asks for the input of the string to check. The final output of the execution is the result of whether the string matches or not the regular expression and if a mismatch is found its position is returned.

To implement the program the Python language is chosen thanks to its wide community, portability and ease of the code understanding.

II. Functional requirements

The regular expressions given as input must follow the infix notation and cannot contains '.' and must end with the '#' char.

The special characters that the program looks at are: '|', '*', '(', ')' and '.'. All the characters besides those are treated as free text.

The application runs in the console and the user has to input the text after the prompted messages displayed by the program.

III. Implementation

General architecture

The program is written in Python and is composed of a main function that contains the logic to run the full flow of the application and several functions that act like modules that return partial results of the execution. The application will make use of external libraries for the data structures and the handle of the I/O operations. This allows a more agile, safe and well tested development.

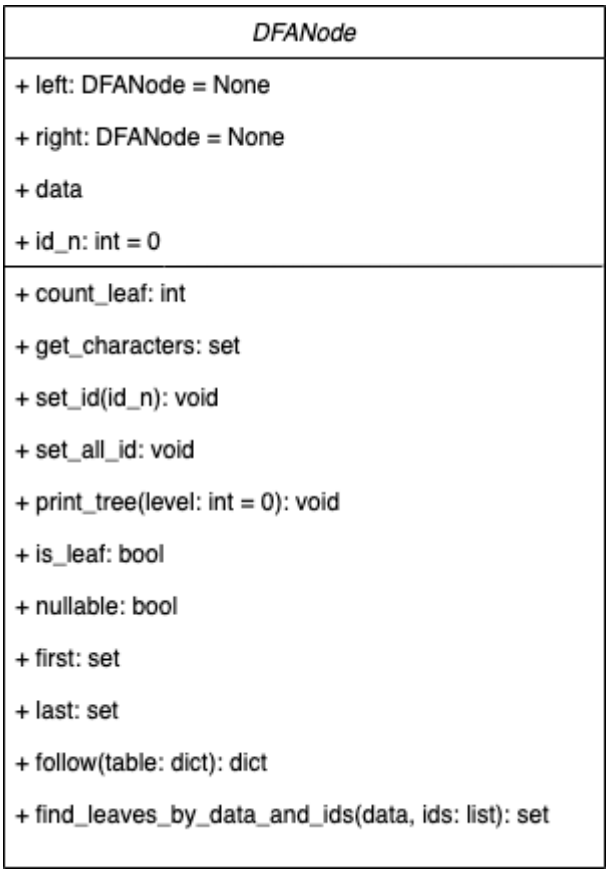
Data structures

The data structures used in the program are the following:

- Stack, used in the *Shunting Yard Algorithm* of the conversion from the infix to the postfix notation.
- Another Stack, used in the *postfix-to-tree algorithm*.
- Tree, used for the representation of the syntax tree.

- Node, used for the representation of the node in the syntax tree. The class has the following components:
  - id: the node id
  - first(): returns the set of the nodes in the *first function*
  - last(): returns the set of the nodes in the *last function*
  - follow(): returns the set of the nodes in the *follow function*

  Many other methods are implemented as helper functions to the main program.



UML class diagram

Module descriptions

There are several algorithms involved in the program:

- The first module is the converter of the RE in from infix to postfix notation using the *Shunting Yard Algorithm*.

  Each special symbol has a value:
  - '*' → value 4
  - '.' → value 3
  - '|' → value 2
  - '(' and ')' → value 1

  The program starts scanning from the start of the string and if the character is not one of the special symbols, it is passed the output queue. Else if the current token is '(', the token is put in the operator stack. If instead the token is the ')' the stack is emptied up to the first occurrence of the character '('. Else if the token is one of '|' or '*' the stack is emptied up to the first occurrence of a token with a value greater than the current symbol. After that the character is put in the operator stack. The only exception is when those operators occur after the closing brackets. In this case the symbol is directly passed to the output queue.

If two free text symbols appear consecutively a '.' is put after to indicate the concatenation of the two characters or also in the case in which there is a '*' followed by free text.

After analyzing the whole regex and filling up the operator and partially the output queue, the stack is popped until no symbol is left.

- Next one that passes from the postfix to the suffix tree using the *postfix-to-tree algorithm*. This algorithm works like this:
    1. Start scanning the postfix regex from left to right
    2. For each symbol a new node is created
    3. If it is a free text the node is put in the Stack
    4. If the symbol is the '|', the left and right children of the node are connected to the last 2 nodes in the stack.
    5. If the symbol is the '*', the left child of the node is connected to the last node in the stack.
    6. If the symbol is the '.', the left and right children of the node are connected to the last 2 nodes in the stack.

- After this step the computation of the functions (nullable, first, last, follow) in each node is performed following the reported rules:

Node n	**Nullable**	**Fist**	**Last**
n is a leaf node labeled €	true	∅	∅
n is a leaf node labeled with position i	false	{ i }	{ i }
n is an or node with left child c1 and right child c2	nullable(c1) or nullable(c2)	first(c1) ∪ first(c2)	last(c1) ∪ last(c2)
n is a cat node with left child c1 and right child c2	nullable(c1) and nullable(c2)	If nullable(c 1) then first(c1) ∪ first(c2) else first(c1)	If nullable(c2) then last(c2) ∪ last(c1) else last(c2)
n is a star node with child node c1	true	first(c1)	last(c1)

Follow rules

1. If n is a cat-node with left child c1 and right child c2 and i is a position in last(c1), then all positions in first(c2) are in follow(i).
2. If n is a star-node and i is a position in last(n), then all positions in first(n) are in follow(i).

Then from all the data gathered by the *Follow* function it is possible to compute the transition table taking as the starting state the *First* function of the Root node.

The data structure used at this step is a dictionary where the state is stored together with the symbol passed and the next state. The result after this state is the DFA.

Finally, the program can check if the string matches the regular expression just following the state stored in the DFA structure.

Input/output description

Input

There is one ways in which the program accepts the input:

- giving the input to the console

  first input the regex → if it passes the checking phase → input the string

The regular expressions given as input must follow the infix notation. The special characters that the program looks at are: '|', '*', '(', ')', '.'. All the characters besides those are treated as free text. Input strings don't have any constraints.

Output

The first output of the program is given after the regex checking phase. After that if the result is positive the program will go on to the next step otherwise a syntax error is displayed. Next the postfix translation is performed and the resulting expression is shown in the console. Subsequently the three generations start, after which a png image is generated and saved in the project folder. Finally the input string is analyzed and the applications starts to look for the match. If no mismatch is found a positive message is displayed otherwise the position of the mismatch is displayed.

IV. Functional test cases

The following table shows the output resulting from a given Regex and the relative input text given to the program.

Regex Infix Notation	Input Text	Regex Postfix Notation	Output
(a|b)*ab#	abab	ab|*a.b.#.	The string 'abab' MATCH the regexp '(a|b)*ab#'
(a|b)*ab#	abbaba	ab|*a.b.#.	MISMATCH, the expression does not match the ending
(a|b)*ab#	ab	ab|*a.b.#.	The string 'ab' MATCH the regexp '(a|b)*ab#'
(a|b)*ab#	bbabab	ab|*a.b.#.	The string 'bbabab' MATCH the regexp '(a|b)*ab#'
a|ba*#	abaa	aba*.#.|	MISMATCH in pos '2': 'ab'
a|ba*#	baaa	aba*.#.|	The string 'baaa' MATCH the regexp 'a|ba*#'
a|ba*#	aaab	aba*.#.|	MISMATCH in pos '2': 'aa'

Regex Infix Notation	Input Text	Regex Postfix Notation	Output
(a|b*#			Syntax error: missing matching brackets in regular expression
a|b)*#			Syntax error: missing matching brackets in regular expression
()#			Syntax error: empty regular expression
)ab(#			Syntax error: missing matching brackets in regular expression
|#			Syntax error: missing arguments in the | operator
a|b)*			The regular expression must end with the # terminator. Re-enter the expression
a..#			The regular expression must not contain the '.' symbol. Re-enter the expression