ECOTE - preliminary project

Semester: 2022L

Author: Jakub Tomkiewicz

Subject: Constructing DFA using syntax tree for given regular expression

I. General overview and assumptions

1.1 Overview

The aim of this project is to write a program that will create directly deterministic finite automata using syntax free procedure for regular expression given by the user. The user will be able to enter input strings and check if the inputs can be generated by the regular expression typed earlier.

1.2 Assumptions

After checking the correctness of the entered regular expression, the program displays to the user the constructed syntax tree and functions during the procedure.

The solution will be written in Python but without the use of external libraries and frameworks. All functions will be implemented manually.

1.3 Regular expression

A regular expression (shortly called regex or RE) is a character's sequence used to describe a search pattern. Its common purpose is to check whether a given string meets the requirements specified in the expression. An example application can be, for example, checking if the entered text is an email: it contains a char sequence followed by the @ symbol and then a domain.

Regular expressions contain so-called meta-characters that have special meanings. In this project, the user will be able to use the following four:
***** - means that a given symbol may appear 0 or more times
+ - is similar to *, but means that a given symbol may appear 1 or more times
| - stands for basically 'or', which means that the first or second symbol can occur
() - are grouping symbols, and characters inside are treated as a single unit

1.4 DFA

Deterministic finite automata (DFA) is a finite state machine (mathematical model of computation) that accepts or rejects a given string of characters. It parses chars through a sequence dictated by each string.

In deterministic automata, each state sequence is distinct. Thus, for a single input symbol, there is one and only one result state. DFA is the opposite of NFA (nondeterministic finite automata), in which there is more than one possible transition from a single state.

1.5 Syntax tree

The syntax tree procedure is a procedure to construct DFA directly from a regular expression. It results in a minimized DFA. Compared to the second possible procedure, called Thompson's algorithm, it has fewer states and does not have to be minimized using a Table filling algorithm.

II. Functional requirements

2.1 Program requirements

- The program runs in console and has a command-line interface (CLI)
- The program checks correctness of input regular expression before syntax tree procedure begins
- If the input regular expression is incorrect, the program will show an error message and stop working
- The program shows constructed syntax tree and four functions with evaluations: nullable, firstpos, lastpos and followpos
- The program must be closed simply, without the use of a SIGINT signal

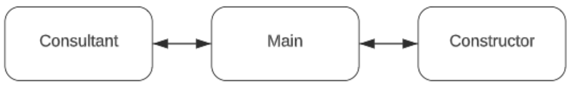
2.2 Input requirements

- Length of user input regular expression and input strings is not limited
- The program will read only four types of meta-characters: *, +, | and ()
- The program will ignore other not supported regular expression meta-characters
- The program will read only three types of regular characters: [a-z], [A-Z] and [0-9]
- The program will ignore other not supported regular characters

III. Implementation

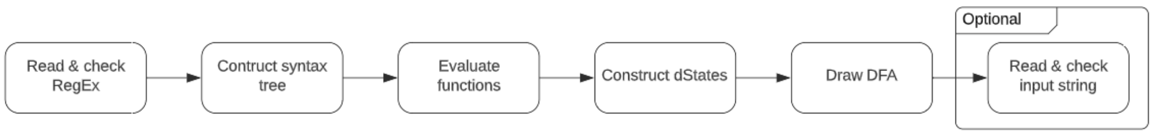
General architecture

Program architecture:



The program consists of three distinctive modules, each of which is responsible for a different task. Consultant's task is to determine what the user wants to do and what input data he/she enters. Then Consultant refers to the Main module with the input regex, which Main then passes on to the Constructor that builds the DFA.

Program steps:



The program process begins by loading a regular expression, which is then checked for syntax errors. If it does not contain any, the syntax tree procedure is launched, which consists of four stages:

constructing a syntax tree, calculating a function for each node, and creating states for which the DFA is then drawn. It is optional for the user to write a string, which the program will check with regex. If the user is not interested, he/she can close the app after the DFA is drawn.

Data structures

DStates – set of states that are used for creating DFA
regex – user-supplied string that contains a regular expression
SyntaxTree – syntax tree created for given regular expression
Node – syntax tree node
SyntaxTreeWithFunctions – SyntaxTree structure with additional attributes (nullable, firstpos, lastpos and followpos)
DFA – deterministic finite automata

Module descriptions

Consultant

Consultant module is responsible for interactions with the user. A straightforward menu has been implemented in it, which after starting the program, is displayed using the *show_menu()* function.

When Consultant receives a regular expression from the user, it passes it to the Main module.

Consultant is called by the Main module when some data has to be presented. Functions *show_syntax_tree()* and *show_functions()* are run during construction and *show_dfa()* function shows an already created DFA.

After the DFA construction is completed, Consultant asks the user to provide input strings that will be checked whether they can be generated by regular expression.

Main

Main module manages the application process. It is a kind of administrator. The module task is to call functions from the Consultant and Constructor modules in the order presented in the graph program steps.

Thanks to the fact that Main is a link, adding additional functionality to the program is very simple because the visual layer is separate from the logical layer.

Constructor

The first task of Constructor is to check if the given regex is correct and does not contain syntax errors, which is done by *is_regex_correct()* function.

Then the constructor of the SyntaxTree structure is called, which is a tree that consists of other smaller Node structures. Already having a tree for each node, the *nullable()*, *firstpos()*, *lastpos()* and *followpos()* functions are run and a new structure with additional attributes called SyntaxTreeWithFunctions is created. Constructor then creates the DStates structure, from which the DFA is finally created.

If a user wants to check if his string can be generated by regex, this is done using the *is_generated_by_regex()* function.

Input/output description

All inputs and outputs are handled in the console. The program implemented has a straightforward menu located in the Consultant module, which is responsible for all interactions with the user through the CLI.

The program has only two inputs: an input regular expression and an input string checked to see if a given regular expression can generate it. There are three outputs: syntax tree, procedure functions with their results, and the created DFA.

IV. Functional test cases

4.1 TC1: Incorrect input regular expression

Input: a(a|b))b

Output: Input regular expression is incorrect!

*4.2 TC2: Correct input regular expression (a|b)*abb*

Input: (a|b)*abb

Output:

```
Syntax Tree:
- + CONCAT
  |
  + - + STAR
  |   |
  |   + - + OR
  |       |
  |       + - a
  |       |
  |       + - b
  |
  + - a
  |
  + - b
  |
  + - b
  |
  + - #
```

```
Syntax Tree with functions:
- + CONCAT
  |
  + - + STAR (firstpos: {1,2}, lastpos: {1,2})
  |   |
  |   + - + OR (firstpos: {1,2}, lastpos: {1,2})
  |       |
  |       + - a (firstpos: 1, lastpos: 1)
  |       |
  |       + - b (firstpos: 2, lastpos: 2)
  |
  + (firstpos: {1,2,3}, lastpos: 3) - a (firstpos: 3, lastpos: 3)
  |
  + (firstpos: {1,2,3}, lastpos: 4) - b (firstpos: 4, lastpos: 4)
  |
  + (firstpos: {1,2,3}, lastpos: 5) - b (firstpos: 5, lastpos: 5)
  |
  + (firstpos: {1,2,3}, lastpos: 6) - # (firstpos: 6, lastpos: 6)

Followpos:
Node 1 followpos: {1,2,3}
Node 2 followpos: {1,2,3}
Node 3 followpos: 4
Node 4 followpos: 5
Node 5 followpos: 6
Node 6 followpos: none

States:
A (start) = {1,2,3}
B = {1,2,3,4}
C = {1,2,3,5}
D (final) = {1,2,3,6}

DFA:
State | input a | input b
A | B | A
B | B | C
C | B | D
D | B | A
```

*4.3 TC3: Correct input regular expression (a|b)*abb and correct input string*

Input: (a|b)*abb

Output: (same as in TC2)

Input: aaabb

Output: Input string aaabb is generated by regex (a|b)*abb

*4.4 TC4: Correct input regular expression (a|b)*abb and incorrect input string*

Input: (a|b)*abb

Output: (same as in TC2)

Input: aabba

Output: Input string aabba is NOT generated by regex (a|b)*abb