# ECOTE - final 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 with functions, syntax tree without functions, and transition table for created DFA.

The solution is written in Python but without the use of external libraries and frameworks. All structures for tree and automata are 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
**|** - 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

The initial preliminary project planned to add a meta-character +, which would mean that a given symbol could appear one or more times. However, it was abandoned due to problems with calculating the syntax tree functions described below.

### *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 simple command-line interface (CLI)
- The program checks the correctness of input regular expression before the syntax tree and DFA procedures begin
- If the input regular expression is incorrect, the program will show an error message and raise an exception
- The program shows constructed syntax tree and three functions with evaluations: firstpos, lastpos and followpos
- The program must be closed simply using the menu option, without the use of a SIGINT signal
- Errors displayed must be strict. If the program found a mistake in the input syntax, they show precisely in which place

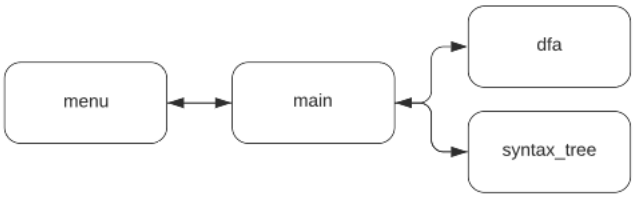
### *2.2 Input requirements*

- 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

## 3. 1General architecture

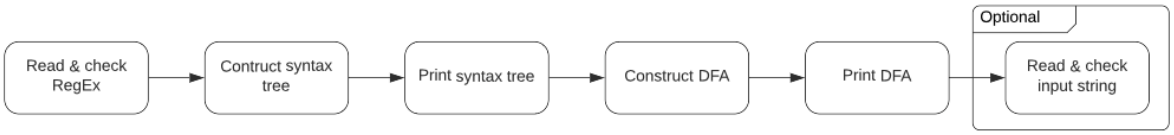
*Program architecture:*




The program consists of four distinctive modules, each responsible for a different task. The menu module determines what the user wants to do and what input data they enter. When the menu

module decides that the regex is correct, it passes it to the main module, calling the syntax tree and dfa modules to perform the procedure.

*Program steps:*




The program begins by loading a regular expression, then checks for syntax errors and formats to contain a concatenation symbol. If regex is correct, the syntax tree procedure launches. It consists of four stages: constructing and printing a syntax tree and then, based on it, creating and printing DFA. It is optional for the user to input a string that checks to see whether an earlier regular expression can generate it. If users are not interested, they can close the app after the printed DFA.

## 3.2 Data structures

regex – user-supplied string that contains a regular expression
Node – a single node of syntax tree
SyntaxTree – syntax tree that consists of nodes
State – single state of automata
DFA – a deterministic finite automaton that consists of states

In the earlier assumptions of the project, I planned to create two separate trees - one without functions and the other with functions. However, this approach duplicates the data that would be the same in both structures. Therefore, to simplify the program and its readability, only one SyntaxTree structure has been implemented.

## 3.3 Module descriptions

*menu*

The menu 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. The user is able to input regular expressions or close the program.

When the module receives a regular expression, it subtracts all unsupported characters, creates an alphabet, checks correctness using is_regex_correct(), and adds concatenation symbols using add_cat_symbol() functions. If at this stage module finds that placed regex is incorrect, the program stops working and shows an error - no syntax tree or DFA is created.

After the DFA construction completes, the module asks the user to provide input strings, using the function read_input_string(), that the program will check whether they can be generated by regular expressions typed earlier.

In the initial version of the project, the module was also supposed to display tree and DFA structures. However, it was abandoned due to the unnecessary transfer of classes in the function parameters. Objects can now show their contents by themselves using implemented functions.

*main*

The main module manages the application process. It is a kind of administrator. The module task is to call functions from the menu, syntax_tree, and dfa modules in the order presented in the graph program steps.

Thanks to the fact that the module is a link, adding additional functionality to the program is effortless because the user interface layer is separate from the logical layer.

*syntax_tree*

Module is, as its name suggests, responsible for creating a tree structure that consists of other smaller structures called Nodes.

Before the building function build_tree() is called, another function called create_tokens() creates a table consisting of characters from regular expression in postfix notation (operator is at the right ex. x*y is xy*)

At first, the created tree does not include nullable, firstpos, lastpos, and followpos functions. All calculations are done by calculate_function(), which runs as the last step of the SyntaxTree initialization procedure.

As mentioned in the description of the menu module, the module can draw a created structure. It starts with the root node and goes down the tree. A structure without functions is illustrated to show data clearly. After that, a tree with functions is displayed.

*dfa*

The dfa module creates the Deterministic Finite Automaton based on the tree made earlier from the syntax_tree module. DFA consists of smaller classes called states, the single state in the automaton.

An automaton is created in the function build_dfa(), which will create a single initial state. Based on possible transitions produced by the transition() function, it makes and concatenates the following states. Creation ends when all possible transitions have been taken into account.

The program can draw DFA to the user with the function print_dfa(), which prints the transition table. However, the automata are gently formatted with the format_states() function before anything is displayed. It causes the user to see the target state instead of seeing all possible transitions from the state.

Optionally, the can_be_generated() function is called to test if the typed expression can be generated by the regex inserted to create an automaton.

## 3.4 Input/output description

All inputs and outputs are handled in the console. The program has a straightforward menu located in the menu 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 an earlier given regex can generate it. There are three outputs: syntax tree without functions, syntax tree with firstpos and lastpos functions, and transition table for the created DFA.

## IV. Functional test cases

### *4.1 TC1: Correct regular expression a* and two input strings: correct aaa and incorrect b*

**Input:**

What would you like to do?
0 - Convert regex to DFA
1 - exit
Input: 0
Input regex: a*

**Output:**

SYNTAX TREE WITHOUT FUNCTIONS:

```
CAT
|
|___STAR___a
|
|___#
```

SYNTAX TREE WITH FUNCTIONS:

```
CAT first_pos(1,2) last_pos(2)
|
|___STAR first_pos(1) last_pos(1)___a first_pos(1) last_pos(1)
|
|___# first_pos(2) last_pos(2)
```

TRANSITION TABLE:

```
    STATE    a
BEGIN 1      | 1 | FINAL
```

**Input:**

What would you like to do?
0 - Input string to check, if it can be generated by regex
1 - exit
Input: 0
Input string: aaa

**Output:**

aaa can be generated by regex

**Input:**

What would you like to do?
0 - Input string to check, if it can be generated by regex
1 - exit
Input: 0
Input string: b

**Output:**

b cannot be generated by regex

### *4.2 TC2: Correct regular expression a|b|c and two input strings: incorrect dd and correct b*

**Input:**

What would you like to do?
0 - Convert regex to DFA
1 - exit
Input: 0
Input regex: a|b|c

**Output:**

SYNTAX TREE WITHOUT FUNCTIONS:

```
CAT
|
|___OR
|   |
|   |___OR
|   |   |
|   |   |___c
|   |   |
|   |   |___b
|   |
|   |___a
|
|___#
```

SYNTAX TREE WITH FUNCTIONS:

```
CAT first_pos(1,2,3) last_pos(4)
|
|___OR first_pos(1,2,3) last_pos(1,2,3)
|   |
|   |___OR first_pos(2,3) last_pos(2,3)
|   |   |
|   |   |___c first_pos(3) last_pos(3)
|   |   |
|   |   |___b first_pos(2) last_pos(2)
|   |
|   |___a first_pos(1) last_pos(1)
|
|___# first_pos(4) last_pos(4)
```

TRANSITION TABLE:

```
    STATE     a   b   c
BEGIN 1       | 2 | 2 | 2 |
    2         |   |   |   | FINAL
```

**Input:**

What would you like to do?
0 - Input string to check, if it can be generated by regex
1 - exit
Input: 0
Input string: dd

**Output:**
dd cannot be generated by regex

**Input:**
What would you like to do?
0 - Input string to check, if it can be generated by regex
1 - exit
Input: 0
Input string: b

**Output:**
b can be generated by regex

### *4.3 TC3: Correct but tricky regular expression (((a)*)*)* and no input strings*

**Input:**
What would you like to do?
0 - Convert regex to DFA
1 - exit
Input: 0
Input regex: (((a)*)*)*

**Output:**
SYNTAX TREE WITHOUT FUNCTIONS:
CAT
|
|___STAR___STAR___STAR___a
|
|___#

SYNTAX TREE WITH FUNCTIONS:
CAT first_pos(1,2) last_pos(2)
|
|___STAR first_pos(1) last_pos(1)___STAR first_pos(1) last_pos(1)___STAR first_pos(1) last_pos(1)___a first_pos(1) last_pos(1)
|
|___# first_pos(2) last_pos(2)

TRANSITION TABLE:
STATE a
BEGIN 1 | 1 | FINAL

### *4.4 TC4: Incorrect regular expression (((a)*)* with too many opening parentheses*

**Input:**

0 - Convert regex to DFA
1 - exit
Input: 0
Input regex: (((a)*)*

**Output:**

Exception: Given regex contain different number of closing and opening parentheses!

### *4.5 TC5: Correct regular expression (&b|%c)* that contain characters & and % that are being ignored (no input strings)*

**Input:**

What would you like to do?
0 - Convert regex to DFA
1 - exit
Input: 0
Input regex: (&b|%c)*

**Output:**

SYNTAX TREE WITHOUT FUNCTIONS:

```
CAT
|
|___STAR___OR
|        |
|        |___c
|        |
|        |___b
|
|___#
```

SYNTAX TREE WITH FUNCTIONS:

```
CAT first_pos(1,2,3) last_pos(3)
|
|___STAR first_pos(1,2) last_pos(1,2)___OR first_pos(1,2) last_pos(1,2)
|        |
|        |___c first_pos(2) last_pos(2)
|        |
|        |___b first_pos(1) last_pos(1)
|
|___# first_pos(3) last_pos(3)
```

TRANSITION TABLE:

```
    STATE    b   c
BEGIN 1      | 1 | 1 | FINAL
```

### *4.6 TC6: Exiting application at the beginning*

**Input:**

What would you like to do?
0 - Convert regex to DFA
1 - exit
Input: 1

### *4.7 TC7: Correct regular expression a(b|c|d)* with four parameters and no input string*

**Input:**

What would you like to do?
0 - Convert regex to DFA
1 - exit
Input: 0
Input regex: a(b|c|d)*

**Output:**

SYNTAX TREE WITHOUT FUNCTIONS:

```
CAT
|
|___CAT
|   |
|   |___STAR___OR
|   |        |
|   |        |___OR
|   |        |   |
|   |        |   |___d
|   |        |   |
|   |        |   |___c
|   |        |
|   |        |___b
|   |
|   |___a
|
|___#
```

SYNTAX TREE WITH FUNCTIONS:

```
CAT first_pos(1,2,3,4) last_pos(5)
|
|___CAT first_pos(1,2,3,4) last_pos(1)
|   |
|   |___STAR first_pos(2,3,4) last_pos(2,3,4)___OR first_pos(2,3,4) last_pos(2,3,4)
|   |        |
|   |        |___OR first_pos(3,4) last_pos(3,4)
|   |        |   |
|   |        |   |___d first_pos(4) last_pos(4)
|   |        |   |
|   |        |   |___c first_pos(3) last_pos(3)
|   |        |
|   |        |___b first_pos(2) last_pos(2)
|   |
|   |___a first_pos(1) last_pos(1)
```

|
|___# first_pos(5) last_pos(5)

TRANSITION TABLE:

| STATE | a | b | c | d |
|---|---|---|---|---|
| BEGIN 1 | 2 | 1 | 1 | 1 |
| 2 | | | | FINAL |

### *4.8 TC8: Incorrect regular expression a)(b)* with closing before opening parentheses*

**Input:**

What would you like to do?
0 - Convert regex to DFA
1 - exit
Input: 0
Input regex: a)(b)*

**Output:**

Exception: Given regex contain closing parentheses at index 1 before opening parentheses!

### *4.9 TC9: Incorrect regular expression a(b||c)*d with two | (or) operators next to each other*

**Input:**

What would you like to do?
0 - Convert regex to DFA
1 - exit
Input: 0
Input regex: a(b||c)*d

**Output:**

Exception: Given regex contain | at index 3 but it is not between two characters!

### *4.10 TC10: Incorrect regular expression a(|b)*d with | (or) operator in incorrect position*

**Input:**

What would you like to do?
0 - Convert regex to DFA
1 - exit
Input: 0
Input regex: a(|b)*d

**Output:**

Exception: Given regex contain | at index 2 but it is not between two characters!