RegexParser Implementation Notes

This tool aim to draw the final state machine related to the regular expression given as input.

REGULAR EXPRESSION GRAMMAR

Draw a finite state machine related to a regular expression means to parse the related grammar. The grammar knowledge allows to create the right recursive process to create the graph you want.

Grammar:

RE	->	SS
SS	->	C_Σ
SS	->	C_Σ AX
SS	->	C_OA SS
SS	->	SS C_OC
SS	->	SS U SS
C_OA	->	(C_OA
C_OA	->	[C_OA
C_OA	->	ε
C_OC	->	) C_OC
C_OC	->	) STAR
C_OC	->	) CROSS
C_OC	->	) AX
C_OC	->	] C_OC
C_OC	->	] AS
C_OC	->	SS
C_OC	->	ε
STAR	->	* SS
CROSS	->	+ SS
AX	->	VAL SS
AS	->	AP SB SS
AP	->	VAL
SB	->	VALS
C_Σ	->	ONE alphabet symbol

NON-Terminal symbol Legend

RE	:	Regular Expression
SS	:	Symbols Sequence
C_Σ	:	alphabet char
C_OA	:	opening operative char
C_OC	:	closing operative char
AX	:	generic Apex
AS	:	Apex Subscript couple
AP	:	Apex after]
SB	:	Subscript
STAR	:	star symbol
CROSS	:	cross symbol
VAL	:	number >0
VALS	:	number ≥ 0

our application allows only ONE-digit number

special case (it isn't a non-terminal symbol)

U : Union symbol

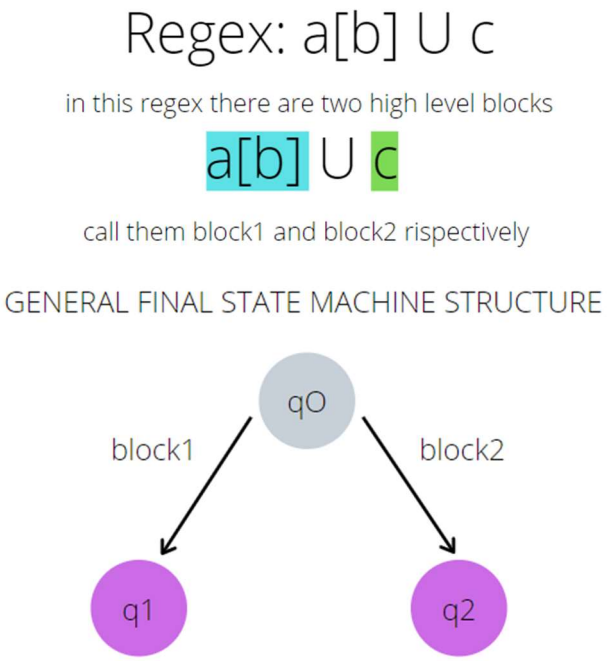
IMPLEMENTATION

IDEA

First, we check the right lexicon and syntax of the regular expression, so we check symbol by symbol the regex to avoid errors. In case of error the user is alerting.
After that, we must process the expression to draw the related finite state machine:
we can see a regular expression how a set of *blocks*. Each block consists of a set of symbols, of both alphabet and non-alphabet.
So, the main implementation idea is work recursively on each block, with a top-down approach:

1. We divide the regular expression in a set of blocks spaced by the symbol "U", only if this symbol is not into a pair of round brackets. This operation provides an high level structural point of view of the regular expression and allow to create the general graph structure which later we will change with the content of each block.
2. Recursively, we want to know the internal structure of each block. So, our idea is identifying the round brackets block, ex (aaa), and create a symbolic representation of this in such a way to easily compute them.
   We repeat this process in a similar way for the square brackets block, taking care to read any apex and subscript present after the].
3. When we have the complete block structure of the regular expression, we do a recursive substitution, on the graph, of each block with its right implementation.

The following images show the idea with a simple example.



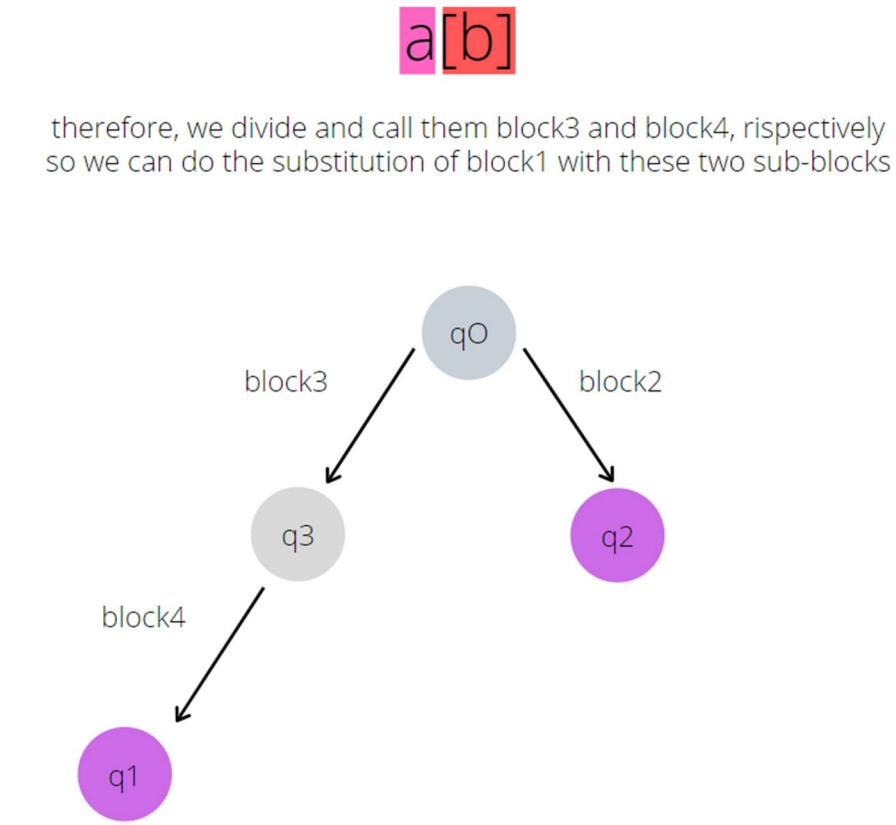


Figure 1 - IDEA example. Violet nodes are the final nodes. q0 is the initial node.

Following image show the same Regex of the example, processed by the application.

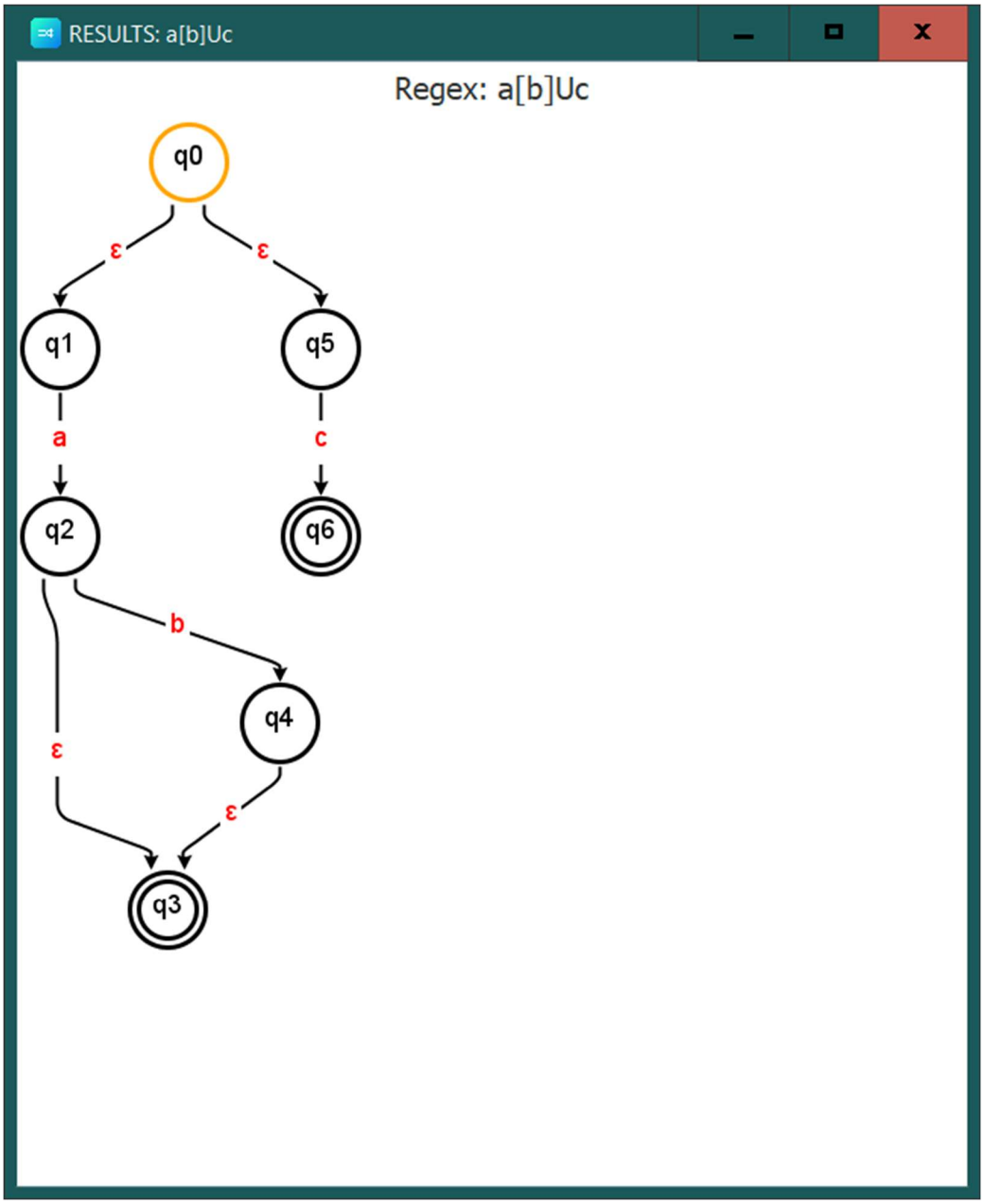


Figure 2 - previous example processed by RegexParser

About the program structure, we decided to use an MVC pattern that perfectly match with our needs and the problem structure.

DEVELOPMENT

PATTERN

The follow figure shows the components which act as Controller, Model and View parts of the MVC pattern.

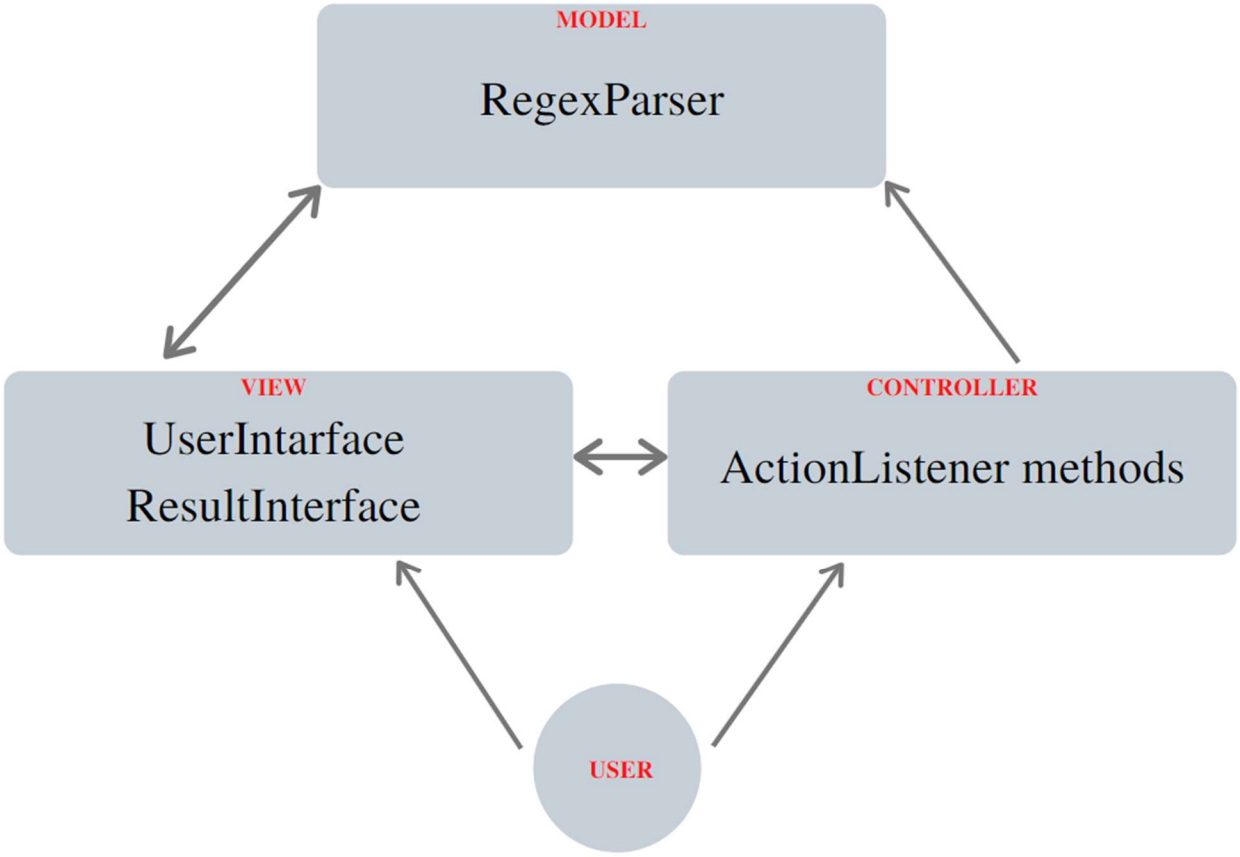


Figure 3 - MVC pattern

SCANNING AND PARSING THE REGULAR EXPRESSION

The check of lexicon and syntax is into the RegexParser Class and is develop in the Scanner method and Parser method, respectively. There are a lot of rules to respect, so for more information about this aspect you can read the RegexParser Guide that shows how write a regular expression in the right way.

PROCESS THE REGULAR EXPRESSION

Let us see, in detail, how the core process of RegexParser has been developed:
the dividing process, illustrated in the previous paragraph, is into the RegexParser Class:

- Is a sum of two methods, Resolve and create_hashTable.

- The first method, Resolve, do the *point 1* and the *point 2* of the idea, using also the second method. So, it prepares the necessary for the recursive substitution process and starts this process. See returnBlock method for more details about the symbolic blocks representation.
- The second method create_hashTable, do the substitution process recursively creating an hash table that will be used to draw the graph. It has a various parameter because it is used to compute both the creation of the general structure of the finite state machine, the recursive substitution process and the creation of the "sub-graph" for the substitution process.

For more details about the code that performs these operations, you can directly see the comments into the Class files.

DRAW THE FINITE STATE MACHINE

The method that draws the finite state machine is named create_graph and simply parses the hash table create before using the library jgraphx (https://github.com/jgraph/jgraphx).
For finding the *final nodes* of the machine that have a different layout, we used a little trick that allow to distinguish the real final nodes from the others: we use a special character "&" for some edges for skip the check for final nodes. In facts, we consider a node as final only if it does not have outgoing edges or its outgoing edges are labelled with ϵ or are a loop. So, into the substitution process we connect the various blocks implementation with ϵ-labelled edges and this is a problem for the final node finder. Therefore, we use &-labelled edges for the connections instead of ϵ-labelled.
In the result draw, you do not see &-labelled edges because they were modified before showing.

CREDITS

Tool designed and developed by

Dott. Davide Cesani @ Università degli Studi di Bergamo
d.cesani@studenti.unibg.it
Dott. Federico Nespoli @ Università degli Studi di Bergamo
f.nespoli1@studenti.unibg.it

as end-of-class project.

Class:

Formal languages and Compilers
Prof. Giuseppe Psaila