Project report

Course: Automata

PAKISTAN AIR FORCE KARACHI INSTITUTE OF ECONOMICS & TECHNOLOGY

Project partners:

1) **NAME: MUHAMMAD ADNAN**

   SID: 61717

2) **NAME: NOUMAN**
   SID: 62199
3) **NAME: SIRAJ**

   SID: 61917

Faculty:

Sir Saad Akbar

Introduction: The course introduces some fundamental concepts in automata theory and formal languages including grammar, finite automaton, regular expression, formal language, pushdown automaton, and Turing machine. Not only do they form basic models of computation, they are also the foundation of many branches of computer science, e.g. compilers, software engineering, concurrent systems, etc. our project is based on one of the important topic of the course regular expression. A regular expression (sometimes called a rational expression) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching.

Regular Expression:

(ab + b)(ab + bb)*(a + b)*

Project description:

Programming Language: python.

The list of metacharacters which we use in our project.

[ ] , ( ) , *, +, { }.

[]

The first metacharacters we'll look at are [and]. They're used for specifying a character class, which is a set of characters that you wish to match. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-'. For example, [abc] will match any of the characters a, b, or c; this is the same as [a-c], which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be [a-z].

( )

Groups are marked by the `'('`, `')'` metacharacters. `'('` and `')'` have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside them, and you can repeat the contents of a

group with a repeating qualifier, such as `*`, `+`, `?`, or `{m,n}`. For example, `(ab)*` will match zero or more repetitions of ab.

Repeating Things: (*)

Being able to match varying sets of characters is the first thing regular expressions can do that isn't already possible with the methods available on strings. However, if that was the only additional capability of regexes, they wouldn't be much of an advance. Another capability is that you can specify that portions of the RE must be repeated a certain number of times.

The first metacharacter for repeating things that we'll look at is *. * doesn't match the literal character *; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.

Using Regular Expressions:

Now that we've looked at some simple regular expressions, how do we actually use them in Python? The "**re**" module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

Performing Matches:

Once you have an object representing a compiled regular expression, what do you do with it? Pattern objects have several methods and attributes. Only the most significant ones will be covered here; consult the **re** docs for a complete listing.

Method/Attribute	Purpose
`match()`	Determine if the RE matches at the beginning of the string.
`search()`	Scan through a string, looking for any location where this RE matches.
`findall()`	Find all substrings where the RE matches, and returns them as a list.
`finditer()`	Find all substrings where the RE matches, and returns them as an iterator.

Now you can query the match object for information about the matching string. match object instances also have several methods and attributes; the most important ones are:

Method/Attribute	Purpose
`group()`	Return the string matched by the RE
`start()`	Return the starting position of the match
`end()`	Return the ending position of the match
`span()`	Return a tuple containing the (start, end) positions of the match

Here's a table of the available flags, followed by a more detailed explanation of each one.

Flag	Meaning
`DOTALL, S`	Make `.` match any character, including newlines
`IGNORECASE, I`	Do case-insensitive matches
`LOCALE, L`	Do a locale-aware match
`MULTILINE, M`	Multi-line matching, affecting `^` and `$`
`VERBOSE, X`	Enable verbose REs, which can be organized more cleanly and understandably.
`UNICODE, U`	Makes several escapes like `\w, \b, \s` and `\d` dependent on the Unicode character database.

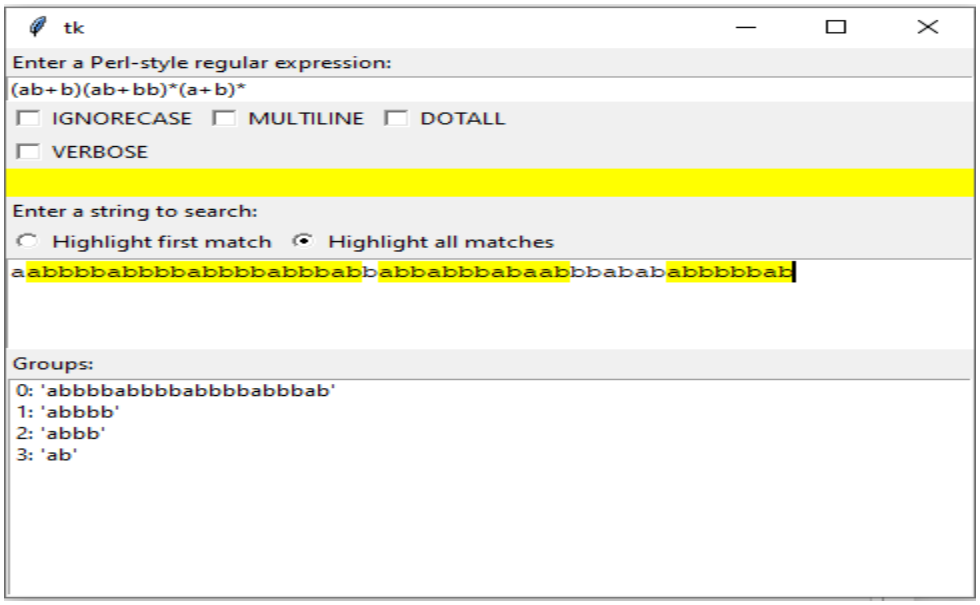
Running Project screenshots and video link:

Running project Video link:

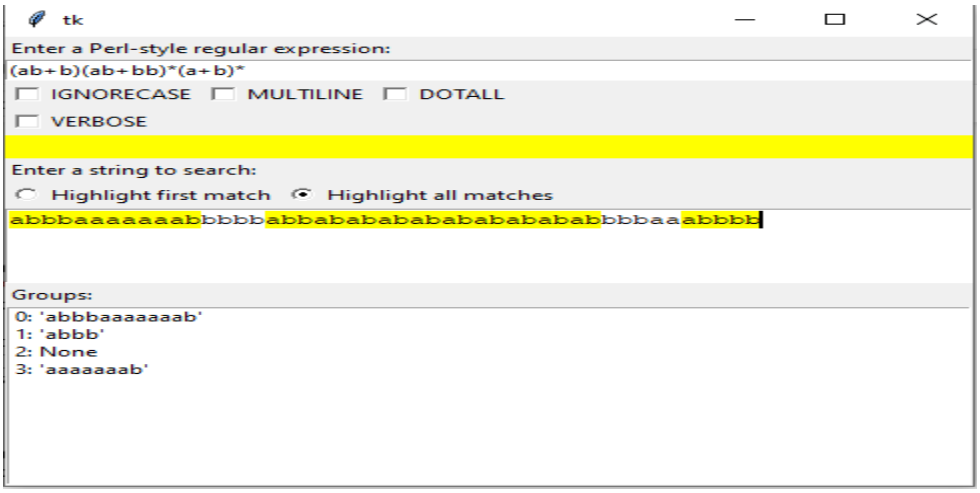
https://drive.google.com/open?id=1lBy579iyL8q5JIwMKnZ1TOezAIHToSnE

Running project screenshots:

1)

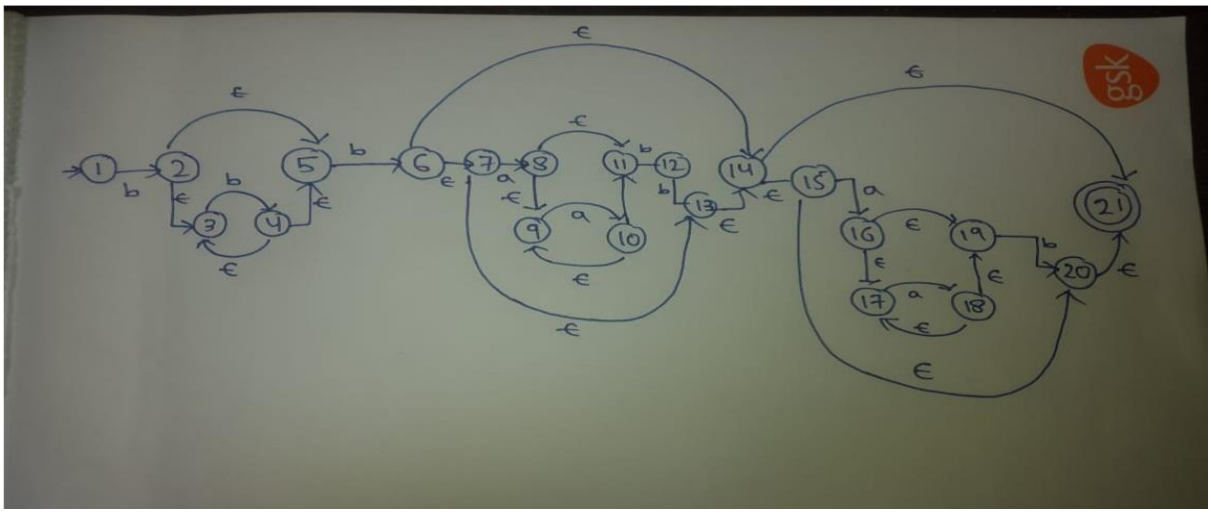


2)



NFA:

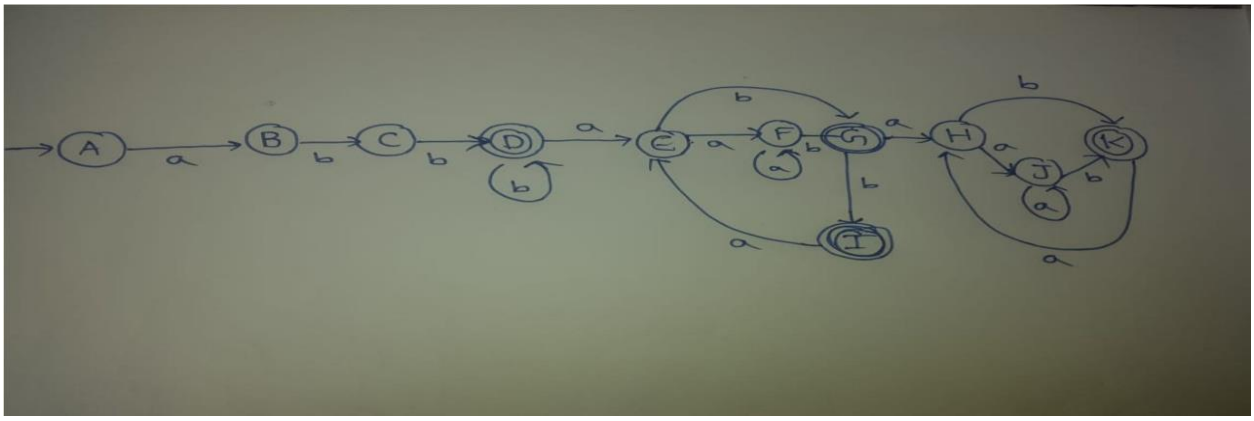
STATE	a	b
0	B	
1		C
2,3,5		D
3,4,5,6,7,14,15,21	E	D
8,9,11,16,17,19	F	G
9,10,11,17,18,19	F	G
12,15,20,21	H	I
16,17,19	J	K
7,13,14,15,21	E	
17,18,19	J	K
15,20,21	H	



DFA:

STATE	a	b
A	B	
B		C
C		D
D	E	D
E	F	G
F	F	G
G	H	I
H	J	K
I	E	

J	J	K
K	H	



Conclusion:

In this project we use regular expression and some python built methods. Because regular expressions usually contain characters that have special meaning to the shell, regular expressions must be quoted correctly. You have to ensure that the syntax is correct.