Regular expressions :-

Another way of giving finite representations for class of Regular languages. But unlike DFA/NFA these are not Machines.

→ Regular expressions (or) regex are used almost every where where we need to do any pattern matching.

ex :- "grep" tool in linux. (look it up).

→ in applications we will convert our regex into either a DFA (or) NFA to use it as a machine easily.

→ Every regular expression represents a Language.

→ Most operations in regular expressions are similar to ϵ-NFAs. We will see how easily we can convert a regex into a ϵ-NFA soon.

Common notation

for a given regular expression 'A'.

We denote the language 'A' is representing by $L(A)$.

→ We will see some basic regex examples and operations in regular expressions and then we will see how to write regular expressions for different complex languages starting from small expressions and compounding them using our operations allowed.

Basic regular expressions

(1) $L_1 = \{\text{"a"}\}$, $\Sigma = \{a, b\}$

$\rightarrow A = a$

(2) $L_1 = \{\text{"a"}, \text{"b"}\}$, $\Sigma = \{a, b\}$

$\rightarrow A = a + b$

(3) $L_3 = \{x_1, \dots, x_k\}$,

$x_1, x_2, \dots, x_k \in \Sigma^*$, $\Sigma = \{a, b\}$

$\rightarrow A = x_1 + x_2 + \dots + x_k$

(4) $L_4 = \{\epsilon\}$, $\Sigma = \{a, b\}$

$\rightarrow A = \epsilon$

(5) $L_5 = \phi$, $\Sigma = \{a, b\}$

$\rightarrow A = \phi$

Basic operations on regular expressions

Assume regular expressions

A, B represent languages $L(A), L(B)$ respectively, then

(1) regular expression for $L(A) \cup L(B)$ will be $A + B$

(or) Shortly

$L(A) \cup L(B) = L(A + B)$

(2) $\sim L(A) = L(\hat{}A)$ → just notation

(3) $L(A) \cdot L(B) = L(A.B)$

(4) $(L(A))^* = L(A^*)$

(5) $(L(A))^k = L(A^k)$, $k \geq 0$

(6) $L(A) \cap L(B)$

Using De-morgans law

$= \sim((\sim L(A)) \cup (\sim L(B)))$

$= L(\hat{}(\hat{}A + \hat{}B))$

→ Using these basic operations we will slowly construct small and complex regular expressions now.

Example regular expressions for Languages

① $L_1 = \{"aba"\}$, $\Sigma = \{a, b\}$

$A_1 =$ aba

② $L_2 = \{"aba" + "bab"\}$, $\Sigma = \{a, b\}$

$A_2 =$ aba + bab → union of $\{"aba"\}$ and $\{"bab"\}$ so +.

③ $L_3 = \Sigma^*$, $\Sigma = \{a, b\}$

$A_3 = (a+b)^*$

→ explanation :-

$\Sigma = \{a, b\}$

→ if you write regex for single character strings you get $\{"a", "b"\}$ as $(a+b)$.

then Σ^* is just $(\{"a", "b"\})^*$

so regex will be $(a+b)^*$

④ $L_4 = \{a^n, n \geq 0\}$, $\Sigma = \{a, b\}$

$A_4 = a^*$ → remember this and previous one. we will use them a lot.

⑤ L_5 = set of all strings that contain exactly one 'b'. $\Sigma = \{a, b\}$

$A_5 = a^* b a^*$

explanation :- your all possible strings contain one b. before and after that it can be concatenated with any number of a's.

⑥ L_6 = set of all strings containing exactly 3 b's. $\Sigma = \{a, b\}$

$A_6 = a^* b a^* b a^* b a^*$ (or) $(A_5)^3 = (a^* b a^*)^3$ → they look different but they represent exact same set of strings.

explanation : same as before.

⑦ L_7 = set of all strings containing exactly k b's, $\Sigma = \{a, b\}$

$A_7 = (a^* b a^*)^k$ (or) $a^* (b a^*)^k$

both valid.

35%

(8) $L_8 = \{x \mid x \text{ ends with an 'a'}\}$, $\Sigma = \{a,b\}$

$A_8 = (a+b)^* a$

explanation :- last char is 'a', before that, Prefix can be anything so Concatenate Σ^*, {'a'} languages, you will get A_8

(9) $L_9 = \{x \mid x \text{ ends with "aba"}\}$, $\Sigma = \{a,b\}$

$A_9 = (a+b)^* . aba$

(10) $L_{10} = \{x \mid x \text{ starts with "bba"}\}$, $\Sigma = \{a,b\}$

$A_{10} = bba . (a+b)^*$

(11) $L_{11} = \{x \mid x \text{ starts with "ab", ends with "aa"}\}$, $\Sigma = \{a,b\}$

$A_{11} = ab . (a+b)^* aa$

(12) $L_{12} = \{x \mid x \text{ starts with "aab", ends with "ab"}\}$ $\Sigma = \{a,b\}$

$A_{12} = aab . (a+b)^* . ab$ ✗

This is a trick question. A_{12} is not representing L_{12} because "aab" is vadid string in L_{12} but it will be not matched with A_{12} regex.

$A_{12} = aab . (\epsilon + (a+b)^* ab)$ will work, it'll accept "aab" too.

(13) $L_{13} = \{x \mid x \text{ contains only one 'a'}\}$, $\Sigma = \{a,b\}$

$A = b^* a b^*$

(14) $L_{14} = \{x \mid x \text{ contains 3 a's only}\}$, $\Sigma = \{a,b\}$

$A = b^* a b^* a b^* a b^*$

39%

(15) $L_{15} = \{x \mid x$ contains even no. of a's$\}$, $\Sigma = \{a, b\}$

$A_{15} = (b^* a b^* a b^*)^* + b^*(a b^* a b^*)^*$

(16) $L_{16} = \{x \mid x$ contains no consecutive b's$\}$,

$\Sigma = \{a, b\}$

$A_{16} = a^*(baa^*)^*(\epsilon + b)$

$(baa^*)^*$ → every b except last b in string is followed by at least 1 a.

$(\epsilon + b)$ → Final can also end with 'b'.

(17) $L_{17} = \{x \mid x$ ends with even no. of a's$\}$, $\Sigma = \{a, b\}$

$A_{17} = ((a+b)^* b + \epsilon)(aa)^*$

Explanation : last $(aa)^*$ will match even no. of a's, its Prefix should be either ending with b to make sure it ending with a doesnt mess up our end with even a's (or) the prefix is just an empty string.

Informal discussion on how to check if a string matches a given pattern (or) whether the string belongs to the language being represented by the regular expression. :-

Note :-

(1) if you see a pattern A.B.C and a string 'x'.

just try to identify parts of string x as $x_1.x_2.x_3$ such that x_1 matches A, x_2 matches B, x_3 matches C.

→ you cannot ignore any pieces of your string 'x'.

Simple example

String $x =$ "aa bbb aaaa", regular expression $= a^*b^*a^*$, $\Sigma = \{a,b\}$

→ look at the expression $a^* \cdot b^* \cdot a^*$

- a^*: any number of a's, a^n $n \geq 0$
- b^*: b^n, $n \geq 0$
- a^*: Same

Our string is of this form. aa bbb aaaa

Simple example 2

String $x =$ "ababba", regular expression $=$ aba b^* a^*

- aba: all strings of the language start with "aba"
- b^*: followed by b^n, $n \geq 0$
- a^*: Same

Our x is of the given form.

starting, end are constant so match them directly

Simple example 3

String = abaaba

ab a^* ba

ex 4 :-

$x =$ ab

$(a+b)^*$ ab

- $(a+b)^*$: Prefix matches ← any string in Σ^*
- ab: Const endings

here it matched with empty string.

35%

Note :-

(2) if you have regex as $A + B + C$ where A, B, C are valid regex too, then for checking for string 'x', identify atleast one of A, B, C which matches with 'x'.

if you find multiple ways of splitting your string to view it in the form of our regular expression, that is okay. they are analogous to NFAs having multiple paths reaching final states for a given string.

we will see that they are exactly equivalent later when we see that regular expressions and all regular language machine models are equivalent. i.e., powerful enough to represent exactly same class of regular languages

Simple example - 1

$x = ab$ $\quad A = a + a^*b$

(3) if you have A^* and string 'x'. then try to break the string x into $x_1, x_2, \dots$ as many as needed where all x_i's match with A.

ex :- a^* matches all $a^n \ \forall \ n \geq 0$

ex - 2 :- $(ba^*)^*$, $x = baa\ ba\ b\ baa$

they all of form ba

ex 3 :- $(a+b)^*$

↳ your Σ^*, $\Sigma = \{a, b\}$

$x = abaaba$

every single character matches with $(a+b)$.

ex 4 :- $\epsilon\ baba\ b$ ✓ $\quad a^*(baa^*)(\epsilon + b)$

$babba$ ✗ rejected

Always in regular expressions, Precedence

$() > * > . > +$

ex :- $ab^* + a \Rightarrow (a.(b^*)) + a$

34%

Properties of regular expressions :-

① Union operation :- let regular expressions A, B represent languages L_A, L_B

$L_A \cup L_B$ has regular expression as $A + B$.

(i) $L_A \cup L_B = L_B \cup L_A \Rightarrow A+B = B+A$ Commutative under Union.

(ii) $L_A \cup (L_B \cup L_C) = (L_A \cup L_B) \cup L_C \Rightarrow A+(B+C) = (A+B)+C$ associative

(iii) $L_A \cup L_A = L_A \Rightarrow A + A = A$

(iv) $L_A \cup \phi = L_A \Rightarrow A + \phi = A$

② Concatenation :-

(i) $L_A . L_B \neq L_B . L_A \Rightarrow A.B \neq B.A$ not Commutative.

$L_A . L_B$ — Concatenation of Languages

(ii) $L_A . (L_B . L_C) = (L_A . L_B) . L_C \Rightarrow A.(B.C) = (A.B).C$ Associative.

(iii) $A.(B+C) = A.B + A.C$ distributive under

(ii) $L_A \cdot (L_B \cdot L_C) = (L_A \cdot L_B) \cdot L_C \Rightarrow A \cdot (B \cdot C) = (A \cdot B) \cdot C$ Associative.

(iii) $A \cdot (B + C) = A \cdot B + A \cdot C$ distributive under concatenation, union.

(iv) $A \cdot \phi = \phi$, $A \cdot \epsilon = A$

(3) Kleene star operator (*) :-

(i) $(L_A^*)^* = L_A^* \Longrightarrow (A^*)^* = A^*$

Short proof :

$L_A^* = L_A^0 \cup L_A^1 \cup \cdots$

$(L_A^*)^0 = \{\epsilon\} = L_A^0$

$(L_A^*)^1 = L_A^*$

$(L_A^*)^{2 \text{ or } k}$ = will add some $(L_A)^k$ terms but they already covered in L_A^*.

So $(L_A^*)^* = L_A^*$

(ii) $(A+B)^* \neq A^* + B^*$

$(AB)^* \neq A^* B^*$

(iii) $A^* A^* = A^*$

(iv) $A^+ = A \cdot A^* = A^* A$

$A^* = \epsilon + A^+$

Additionally

(i) $\phi^* = \epsilon$, $\epsilon^* = \epsilon$

Regular Languages :-

We have seen DFA, NFA, ε-NFA are different finite state machine models capable of representing regular languages.

→ In fact, we have defined the class of regular languages as the set of all languages for which we can construct a finite state machine DFA/NFA. (also an ε-NFA too).

→ Then we saw that we can construct a DFA that represents the same language from a given NFA.

→ Also if we have a DFA we can also call it an NFA because it still follows all constraints of an NFA.

From these two above results we can say that,

If a Language can be represented by an NFA $\xrightarrow{\text{implies}}$ That Language can also be represented by some DFA.

and

If a Language can be represented by a DFA $\Longrightarrow$ That language can also be represented by some NFA.

→ Now these two statements mean that DFA, NFA Machine models are capable of representing the same class of Languages.

→ Which we later define as the class of regular Languages.

So we say that DFA, NFAs are equivalent machine models with same power

↳ (capabilits of representing regular languages in this case)

→ Similarly, from ε-NFA to DFA conversion we can say that ε-NFAs are equivalently powerful too.

$$DFA \equiv NFA \equiv \varepsilon\text{-}NFA$$

→ if you have a machine in one form, you can convert into other two.

Regular expressions and Regular languages

DFA / NFA / ε-NFA are machine models using which we can build machines that are just some black box machine which takes some string x and outputs $x \in L$ or not.

→ Regular expressions are similar, they are some syntactic expressions and not machines.

**Regular expressions are also capable of representing the whole class of Regular Languages.

-> this above point is proved by proving regular expressions are equivalent to DFA.

i.e., NEED TO PROVE : for every DFA construction that represents some language , there also exists some regular expressions representing same language and vice versa. for every regular expression representing a language there also exists a DFA representing the same language

39%

→ We will not look at proof of first part.

→ We will now look at constructing (or) converting a regular expression into an ε-NFA representing same language.

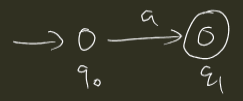
→ Since ε-NFA can later be converted into an equivalent DFA. this conversion is enough to support our argument.

Constructing ϵ-NFA's from Regular expressions

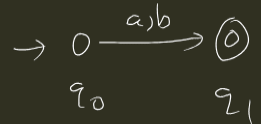
→ We will first build NFAs for small pieces in our regular expressions and then wherever we have . (OR) + (OR)* operators, we combine our small machines and eventually build the ϵ-NFA for whole regular expression (or regex).

Examples NFAs of small expressions

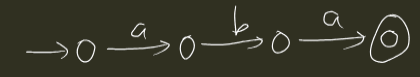
(i) $A = a$, $\Sigma = \{a, b\}$



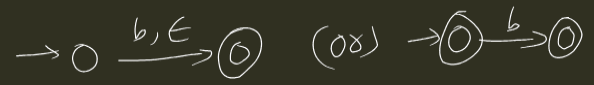
(ii) $A = a + b$



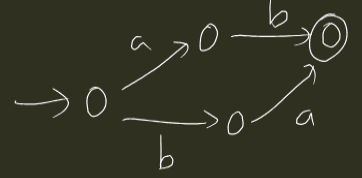
(iii) $A = aba$



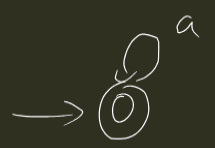
(iv) $A = \epsilon + b$



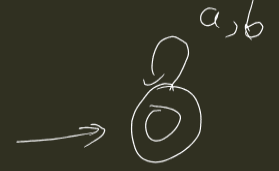
(v) $A = ab + ba$



(vi) $A = a^*$



(vii) $A = (a+b)^*$



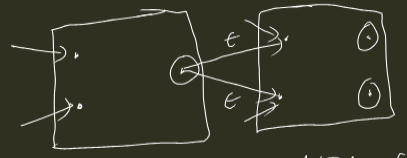
39%

Compounding machines over ., +, * operators

for . Concatenating A.B :-

if NFA of A, B are both given, then

ϵ- NFA of A.B will be



NFA of A NFA of B

Write down NFA of A, B side by side.

→ From all final states in A, draw ϵ- transitions to all start states in B,

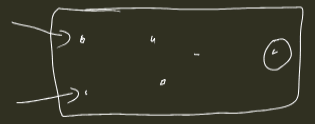
→ Then make final states in A as non-final and start states in B as non-start states.

→ at the end, your start states of A.B are just start states in A, final states only exist in B.

for Union of 2 languages A + B :-

NFA of A, B individually are given.

ϵ- NFA of A + B will be



NFA of A

NFA of B

just draw them together. set of states will be union of states in both. transitions are same.

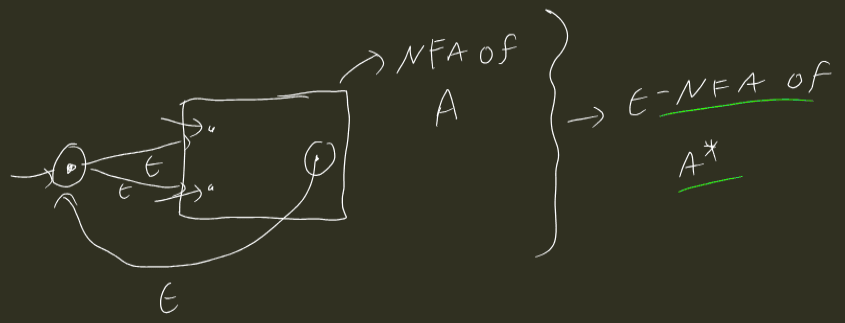
→ Set of start states are combined

→ Set of final states are combined.

→ still an NFA, only new thing is that we have 2 disjoint set of states.

ϵ-NFA for A^* :-

NFA of A is provided,



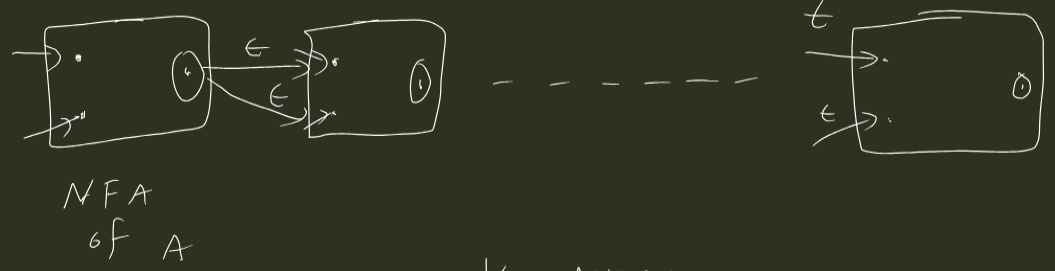
→ one new start state. Draw ϵ-transitions from it to all start states.

→ Draw ϵ-transitions from all final states to our new start state.

→ At the end our one new state will be the only start and final state. No other start (or) final states.

ϵ-NFA for A^K :-

Concatenate using previous A.B method for A.A K-1 times.



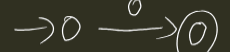
→ at the end, start states exist only in 1st machine, final states exist in final machine.

→ rest all start states, Final states make them not start, not final.

Examples of regex to ε-NFA Conversion :-

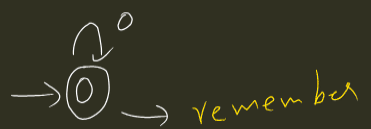
(1) $A = 0 + 1.0^*$

NFA for 0

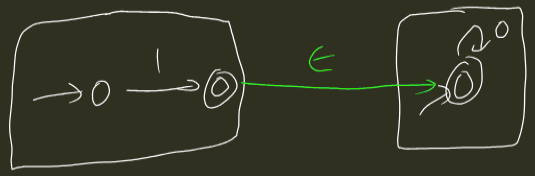


NFA for 1

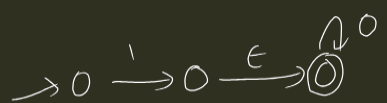
NFA for 0^*



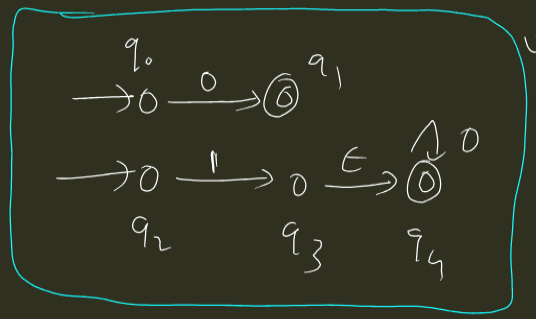
now join 1.0^*



and remove final states in 1 and start states in 0^*.

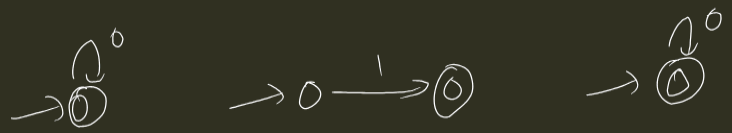


union this with NFA of '0'.

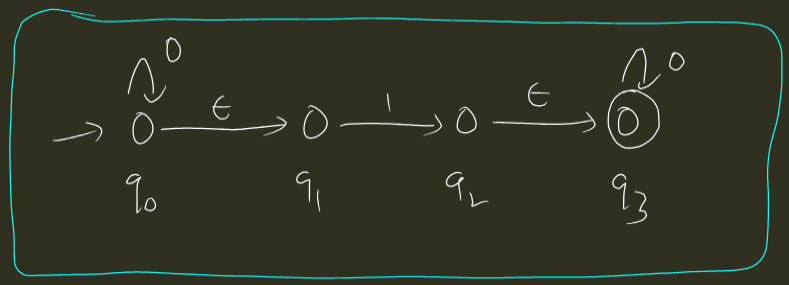


ε-NFA of $0 + 1.0^*$

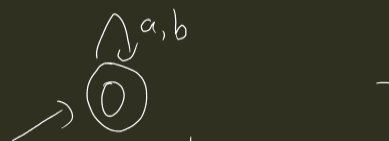
(2) $0^* 1 0^*$



Use the concatenating NFAs method.

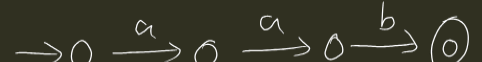


(3) $(a+b)^* aab$

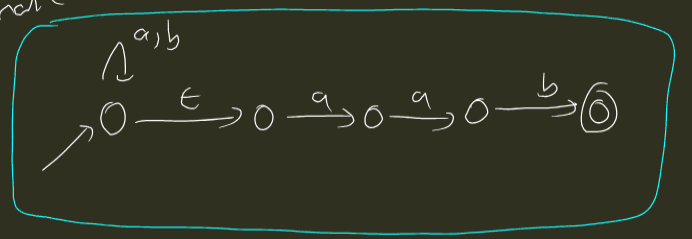


for $(a+b)^*$

remember this one too

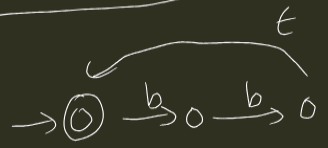


concatenate



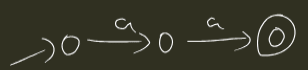
(4) $(aa)^*(bb)^*b$

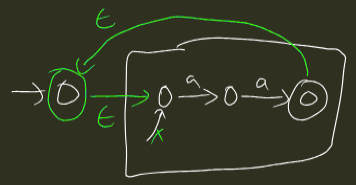
NFA of $(aa)^*$ → Similarly for $(bb)^*$

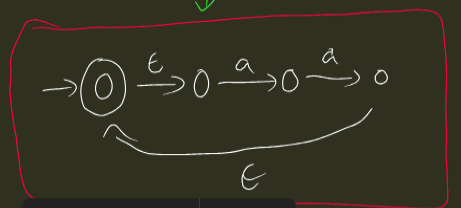


(or) if you want to construct for $(aa)^*$ from scratch,

NFA for aa

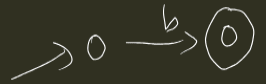






$(aa)^*$

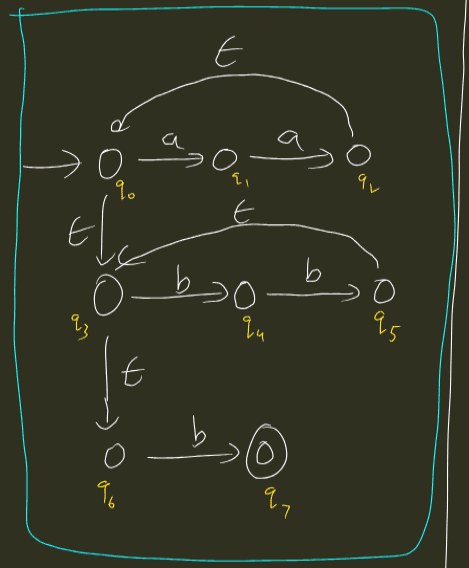
NFA of b



Now concatenate all these in order

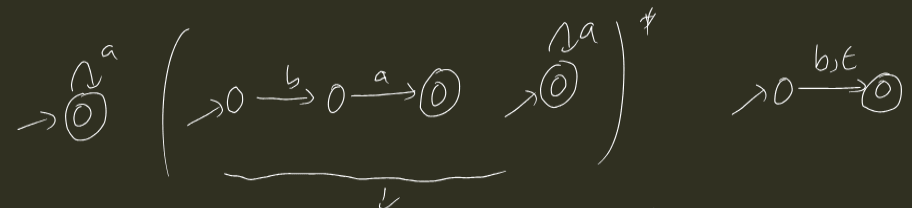
(36)

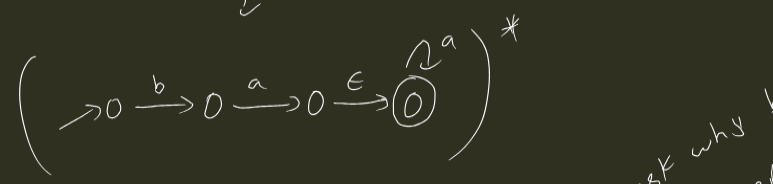
but lets use above one, it looks simpler and a minimized version of this.

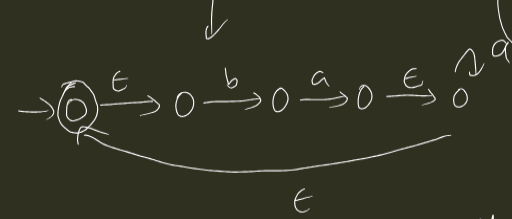


NFA of $(aa)^*(bb)^*b$

(5) $a^*(baa^*)^*(\epsilon+b)$

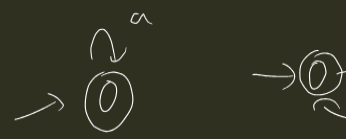


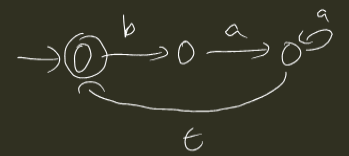


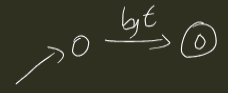


Dont ask why but thes both represent same languages. in exam you dont have to do this simplifications

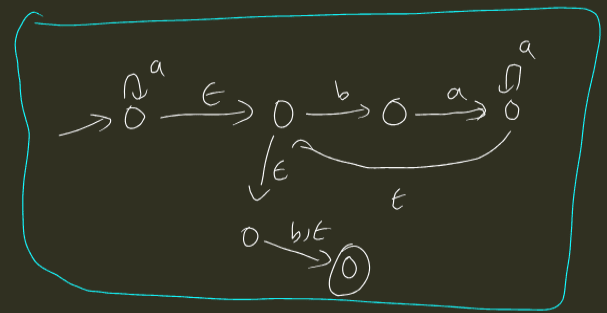
⇓ can be actually simplified into







Now concatenate all



37%