Chapter-6

Regular Expressions

Regular Expression (RE)

A RE is a string that can be formed according to the following rules:

1. ø is a RE.
2. ε is a RE.
3. Every element in ∑ is a RE.
4. Given two REs α and β,αβ is a RE.
5. Given two REs α and β, α U β is a RE.
6. Given a RE α, α* is a RE.
7. Given a RE α, α+ is a RE.
8. Given a RE α, (α) is a RE.

if ∑ = {a,b}, the following strings are regular expressions:

ø, ε, a,b, (a U b)*, abba U ε.

Semantic interpretation function L for the language of regular expressions:

1. L (ø) = ø, the language that contains no strings.
2. L (ε) = {ε}, the language that contains empty string.
3. For any c϶∑, L(c) = {c}, the language that contains single character string c.
4. For any regular expressions α and β, L (αβ) = L (α) L (β).
5. For any regular expressions α and β, L (α U β) = L (α) U L (β).
6. For any regular expression α, L (α*) = (L (α))*.
7. For any regular expression α, L (α+) = L (αα*) = L (α) (L (α))*
8. For any regular expression α, L ((α)) = L (α).

Analysing Simple Regular Expressions

1.L((a U b)*b) = L((a U b)*)L(b)
= (L((a U b)))*L(b)

= (L(a) U L(b))*L(b)
=({a} U {b})*{b}
= {a,b}*{b}

(a U b)*b is the set of all strings over the alphabet {a, b} that end in b.

2. L(((a U b) (a U b))a(a U b)*)

= L(((a U b)(a U b)))L(a) L((a U b)*)
= L((a U b)(a U b)) {a} (L((a U b)})*
= L((a U b))L((a U b)) {a} {a,b}*
= {a, b} { a, b} {a} {a, b}*

- ((a U b)(a U b))a(a U b)* is

{xay : x and y are strings of a's and b's and lxl = 2}.

Finding RE for a given Language

1.Let L = {w ϵ {a, b }*: |w| is even}.

L = {aa,ab,abba,aabb,ba,baabaa,-------}
RE = ((a U b)(a U b))* or (aa U ab U ba U bb)*

2. Let L = {w ϵ {a, b }*: w starting with string abb}.

L = {abb,abba,abbb,abbab-------}
RE = abb(a U b)*

3. Let L = {w ϵ {a, b }*: w ending with string abb}.

L = {abb,aabb,babb,ababb-------}
RE = (a U b)*abb

4. L = {w ϵ {0, 1}* : w have 001 as a substring}.

L = {<u>001</u>,1<u>001</u>,0<u>001</u>01,-------}
RE = (0 U 1)*001(0 U 1)*

5. L = {w ϵ {0, 1}* : w does not have 001 as a substring}.

L = {0,1,010,110,101,----}
RE = (1 U 01)*0*

6. L = {w ϵ {a, b}* : w contains an odd number of a's}.

L = {a,aaa,ababa,bbaaaaba------}
RE = b*(ab*ab*)* a b*   or   b*ab*(ab*ab*)*

7. L = {w ϵ {a, b}* :#a(w) mod 3 = 0}.
L = {aaa,abbaba,baaaaaa,---}
RE = (b*ab*ab*a)*b*

8. Let L = {w ϵ {a, b }*:#a(w) <= 3}.
L = {a,aa,ba,aaab,bbbabb,-------}
RE = b*(a U ε)b*(a U ε)b*(a U ε)b*

9. L = {w ϵ {0, 1}* : w contains no consecutive 0's}
L={0, ε,1,01,10,1010,110,101,-----}
RE = (0 U ε)(1 U 10)

10. L = {w ϵ {0, 1}* : w contains at least two 0's}
L={00,1010,1100,0001,1010,100,000,-----}
RE = (0 U 1)*0(0 U 1)*0(0 U 1)*

11.L = { $a^n b^m$ / n>=4 and m<= 3}
RE= (aaaa)a*(ε U b U bb U bbb)

12.L = { $a^n b^m$ / n<=4 and m>= 2}
RE= (ε U a U aa U aaa U aaaa)bb(b)*

13. L = { $a^{2n} b^{2m}$ / n>=0 and m>= 0}
RE= (aa)*(bb)*

14. L = { $a^n b^m$:(m+n) is even}
(m+n) is even when both a's and b's are even or both odd.
RE = (aa)*(bb)* U a(aa)*b(bb)*

Three operators of RE in precedence order(highest to lowest)

1. Kleene star
2. Concatenation
3. Union

Eg: (a U bb*a) is evaluated as (a U (b(b*)a))

Kleene's Theorem

Theorem 1:
Any language that can be defined by a regular expression can be accepted by some finite state machine.

Theorem 2:
Any language that can be accepted by a finite state machine can be defined by some regular expressions.

Note: These two theorems are proved further.

Buiding an FSM from a RE

Theorem 1:For Every RE, there is an Equivalent FSM.

Proof: The proof is by construction.
We can show that given a RE α,
we can construct an FSM M such that L (α) = L (M).
Steps:

1. If α is any c∈∑ ,we construct simple FSM shown in Figure(1)

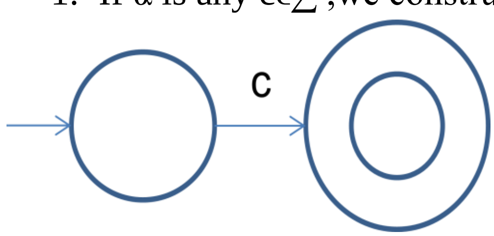


Figure (1)

2. If α is any ø, we construct simple FSM shown in Figure(2).

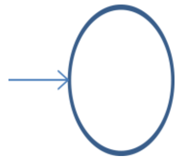


Figure (2)

3. If α is ε,we construct simple FSM shown in Figure(3).

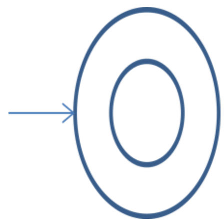
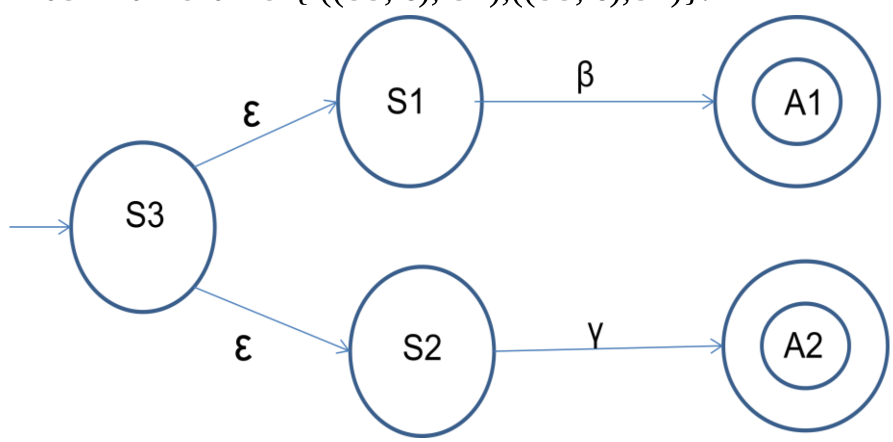


Figure (3)

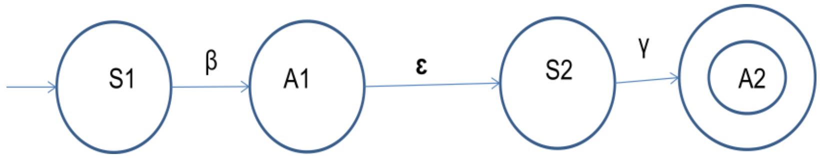
4. Let β and γ be regular expressions.
   If L(β) is regular,then FSM M1 = (K1, ∑ , δ1, s1, A1).
   If L(γ) is regular,then FSM M2 = (K2, ∑ , δ2, s2, A2).
   If α is the RE β U γ, FSM M3=(K3, ∑ , δ3, s3, A3) and
   L(M3)=L(α)=L(β) U L(γ)
   M3 = ({S3} U K1 U K2, ∑ , δ3, s3, A1 U A2), where
   δ3 = δ1 U δ2 U { ((S3, ε), S1),((S3, ε),S2)}.



α = β U γ

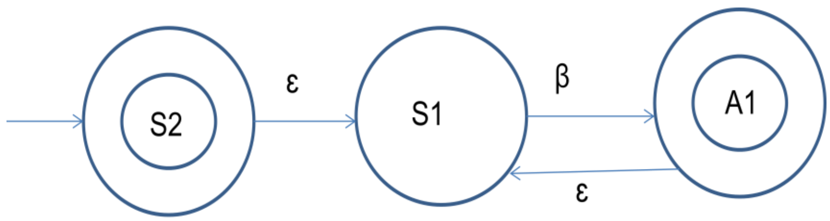
5. If α is the RE βγ, FSM M3=(K3, ∑ , δ3, s3, A3) and
   L(M3)=L(α)=L(β)L(γ)
   M3 = (K1 U K2, ∑ , δ3, s1, A2), where

δ3 = δ1 U δ2 U { ((q, ε), S2):q∊A1}.



α = βγ

6. If α is the regular expression β*, FSM M2 = (K2, ∑, δ2 s2, A2) such that L (M2) = L (α)) = L (β)*.
   M2 = ({S2} U K1, ∑, δ2,S2,{S2} U A1), where
   δ2 = δ1 U {((S2, ε),S1)} U {((q, ε),S1):q ∊ A1}.



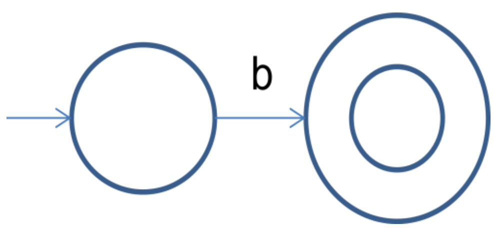
α = β*

Algorithm to construct FSM, given a regular expression α

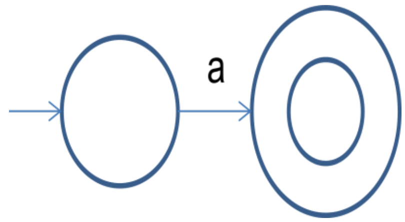
regextofsm(α : regular expression) =
Beginning with the primitive subexpressions of α and working outwards until an FSM for an of α has been built do:
Construct an FSM as described in previous theorem.

Building an FSM from a Regular Expression

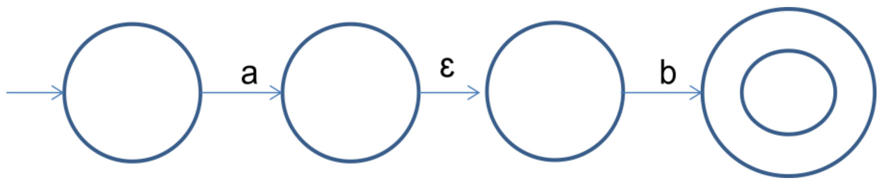
1. Consider the regular expression (b U ab)*.



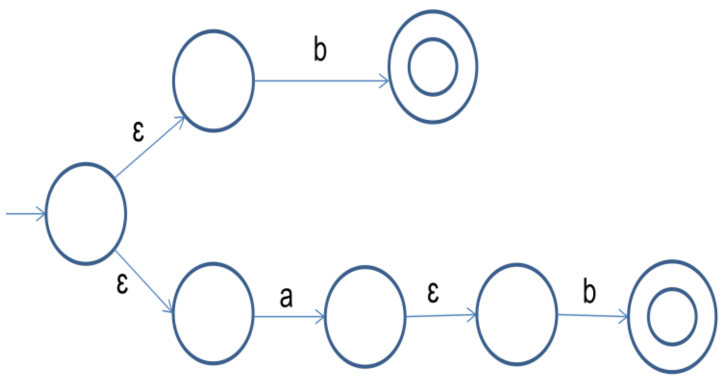
An FSM for b



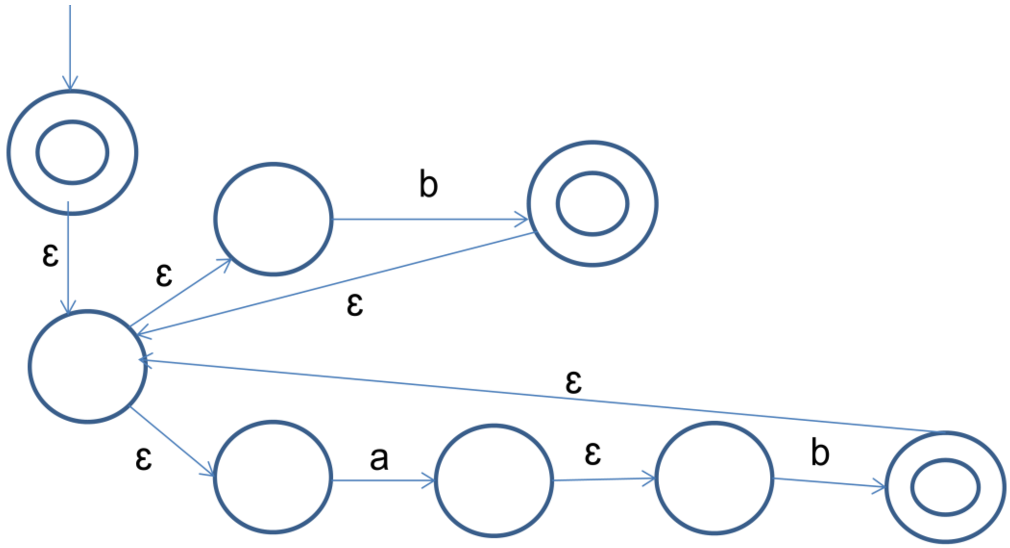
An FSM for a



An FSM for ab

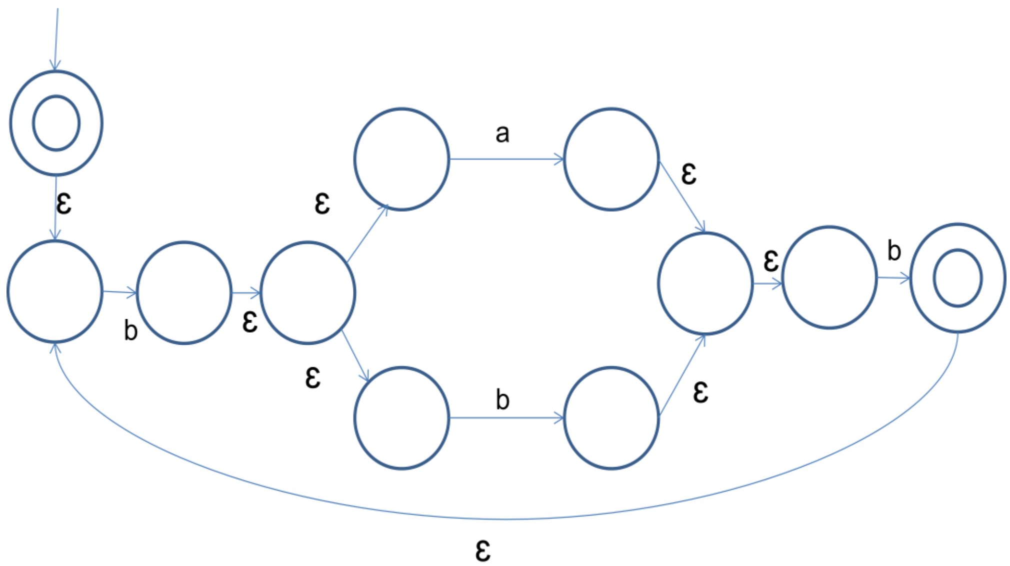


An FSM for (b U ab)

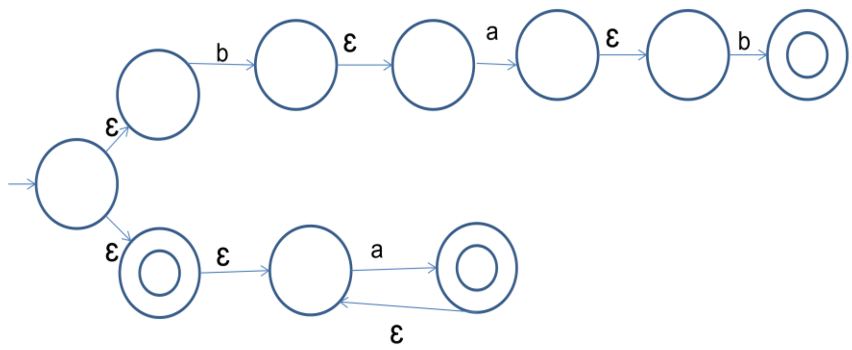


An FSM for (b U ab)*

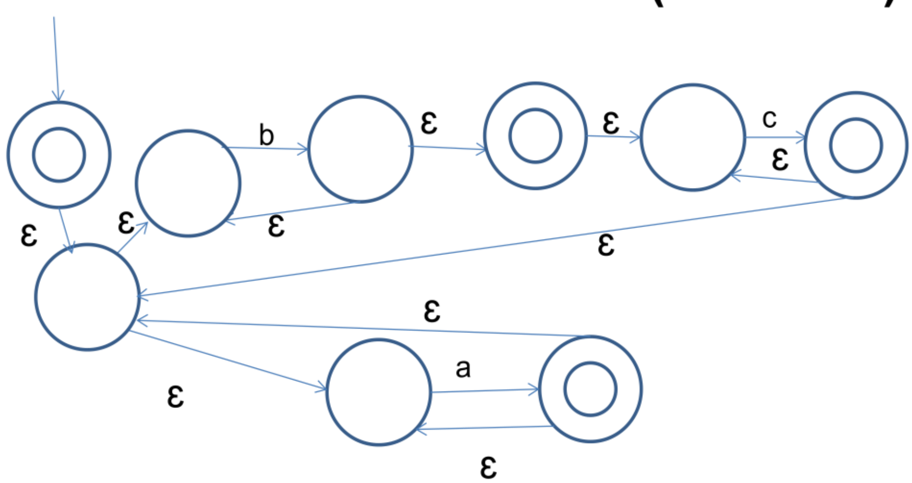
2. Construct FSM for the RE (b(a U b)b)*



3. Construct FSM for the RE **bab U a***

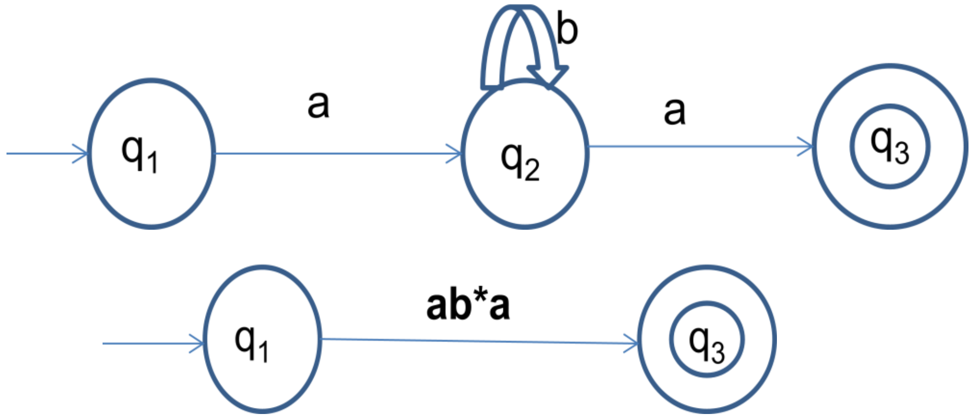


FSM for RE = (a* U b*c*)*



Building a Regular Expression from an FSM

Building an Equivalent Machine M



Algorithm for FSM to RE(heuristic)

fsmtoregexheuristic(M: FSM) =

1. Remove from M-any unreachable states.
2. No accepting states then return the RE ø.
3. If the start state of M is has incoming transitions into it, create a new start state s.
4. If there is more than one accepting state of M or one accepting state with outgoing transitions from it, create a new accepting state.
5. M has only one state, So L (M} = { ε } and return RE ε.

6. Until only the start state and the accepting state remain do:

   6.1. Select some state rip of M.

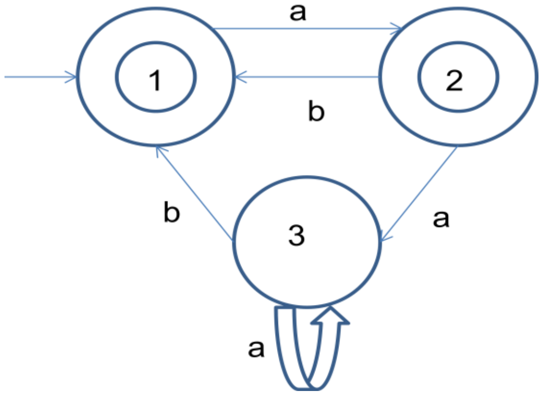
   6.2. Remove rip from M.

   6.3. Modify the transitions. The labels on the rewritten transitions may be any regular expression.

7. Return the regular expression that labels from the start state to the accepting state.

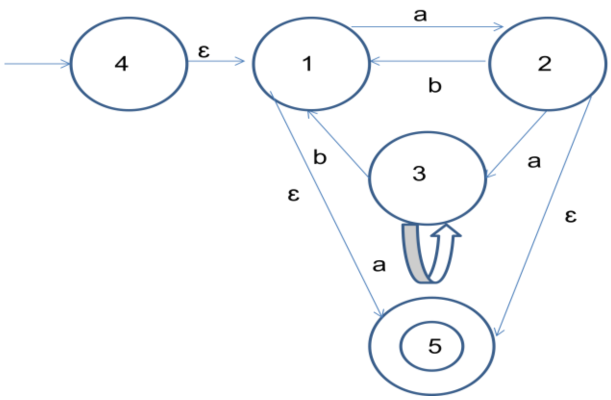
Example 1 for building a RE from FSM

Let M be:

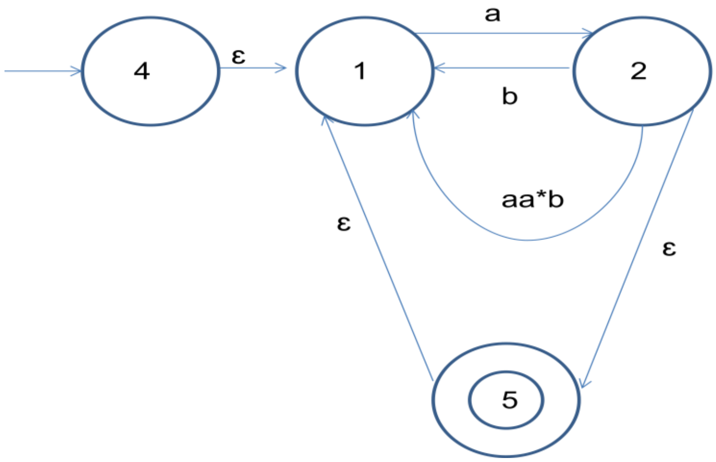


Step 1:Create a new start state and a new accepting state and link them to M

After adding new start state 4 and accepting state 5



Step 2: let rip be state 3



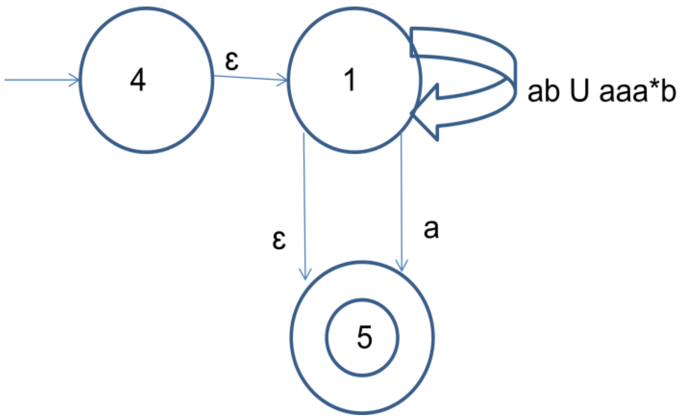
After removing rip state 3

1-2-1:ab U aaa*b

1-2-5:a

Step 3: Let rip be state 2

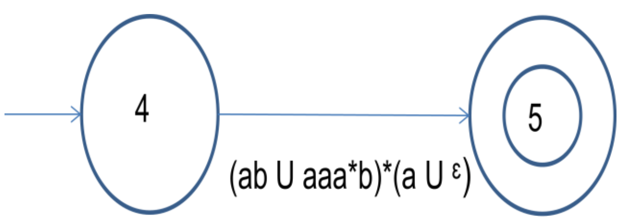
After removing rip state 2



4-1-5: (ab U aaa*b)*(a U ε)

Step 4: Let rip be state 1

After removing rip state 1



RE = (ab U aaa*b)*(a U ε)

Theorem 2 :For Every FSM ,there is an equivalent regular expression

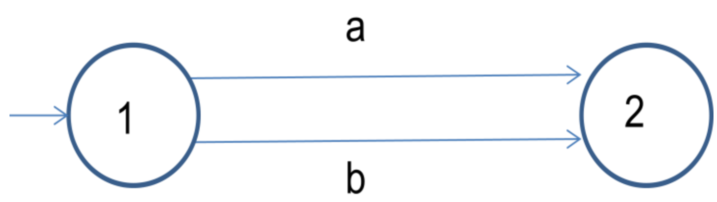
Statement : Every regular language can be defined with a regular expression.

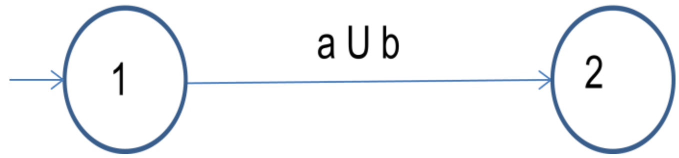
Proof : By Construction

Let FSM M = (K,∑,δ,S,A),construct a regular expression α such that

L(M) = L(α)

Collapsing Multiple Transitions

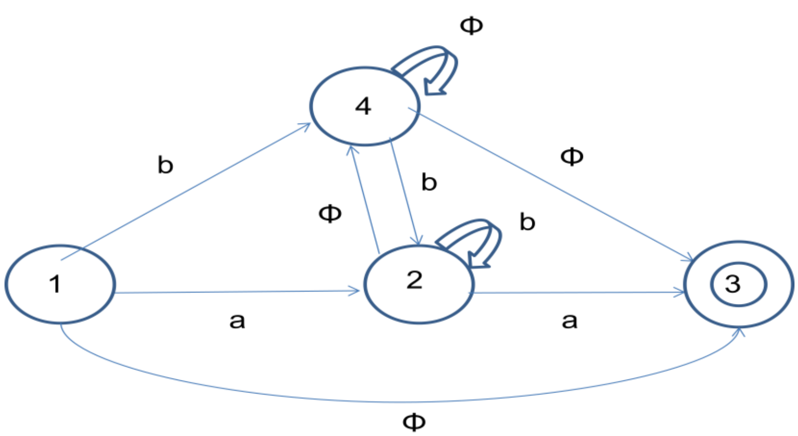




{C1,C2,C3.......Cn} - Multiple Transition

Delete and replace by {C1 U C2 U C3.......U Cn}

If any of the transitions are missing, add them without changing L(M) by labeling all of the new transitions with the RE ø.



Select a state rip and remove it and modify the transitions as shown below.

Consider any states p and q.once we remove rip,how can M get from p to q?

Let R(p,q) be RE that labels the transition in M from P to Q.Then the new machine M' will be removing rip,so R'(p,q)

R'(p,q) = R(p,q) U R(p,rip)R(rip,rip)*R(rip,q)

Ripping States out one at a time

R'(1,3) = R(1,3) U R(1,rip)R(rip,rip)*R(rip,3)

= R(1,3) U R(1,2)R(2,2)*R(2,3)

= ø U ab*a

= ab*a

Algorithm to build RE that describes L(M) from any FSM M = (K,∑,δ,S,A)

Two Sub Routines:

1. **standardize** : To convert M to the required form
2. **buildregex** : Construct the required RE from

   modified machine M

<u>1.Standardize (M:FSM)</u>

i. Remove unreachable states from M
ii. Modify start state
iii. Modify accepting states
iv. If there is more than one transition between states p and q ,collapse them to single transition
v. If there is no transition between p and q and p ∉A, q ∉S,then create a transiton between p and q labled Φ

2.buildregex(M:FSM)

i. If M has no accepting states then return RE Φ

ii. If M has only one accepting state ,return RE ε

iii. until only the start state and the accepting state remain do:

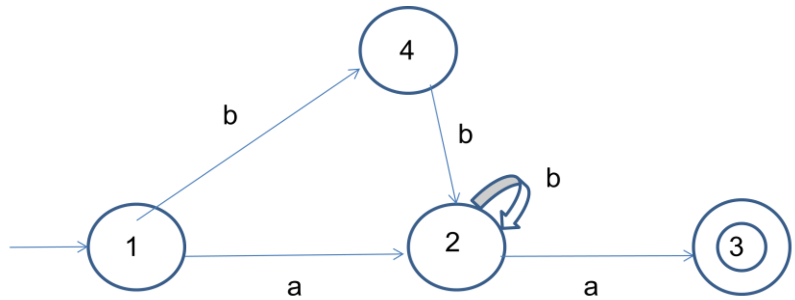
   a. Select some state rip of M

   b. Find R'(p,q) = R(p,q) U R(p,rip).R(rip,rip)*.R (rip,q)

   c. Remove rip on d all transitions into ad out of it

iv. Return the RE that labels from start state to the accepting state

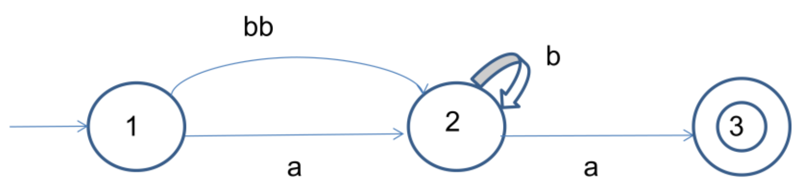
Example 2: Build RE from FSM



Step 1: let RIP be state 4

1-4-2 : bb

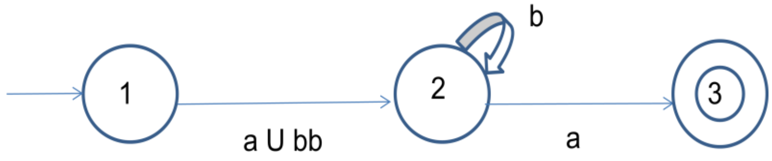
After removing rip state 4



Step 2: Collapse multiple transitions from state 1 to state 2

1-2: a U bb

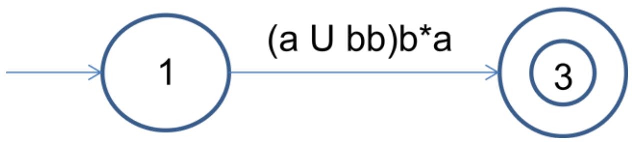
After collapsing multiple transitions from state 1 to state 2



Step 3: let rip be state 2

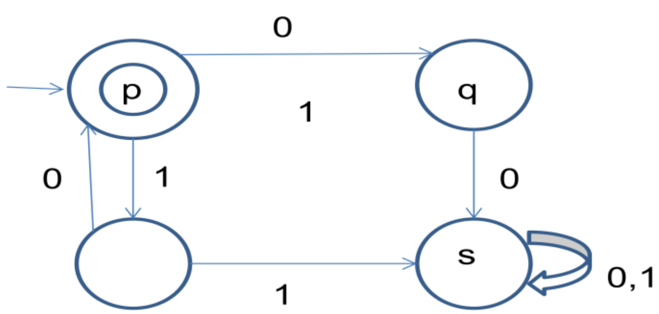
1-3: (a U bb)b*a

After removing rip state 2



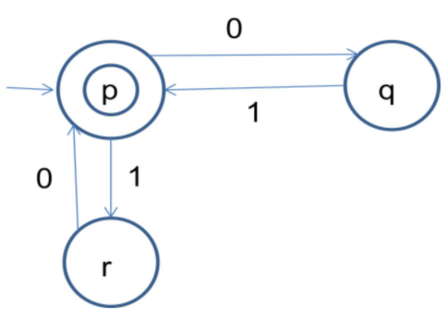
RE = (a U bb)b*a

Example 3: Build RE From FSM



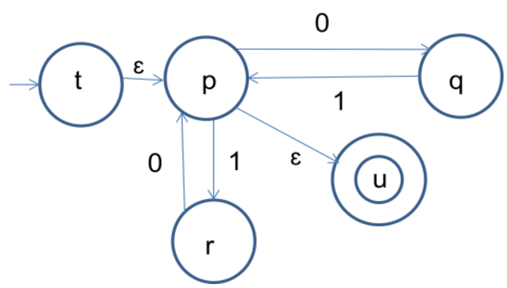
Step 1: Remove state s as it is dead state

After removing state s



Step 2: Add new start state t and new accepting state u

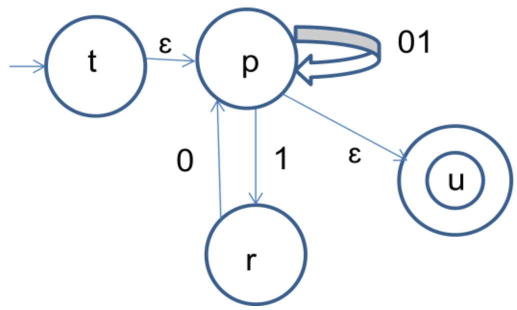
After adding t and u



Step 3: Let rip be state q

p-q-p: 01

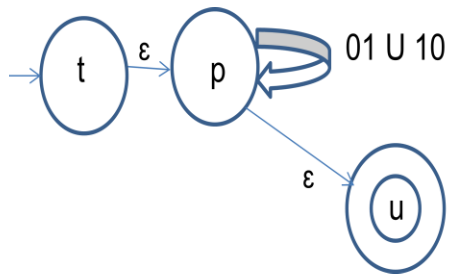
After removing rip state q



Step 4: Let rip be state r

p-r-p: 10

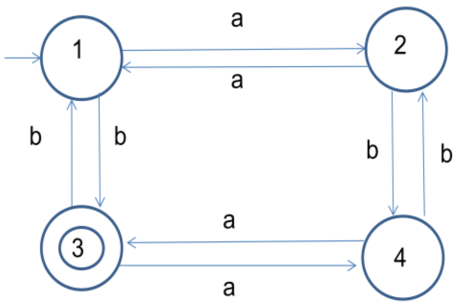
After removing rip state r



RE = (01 U 10)*

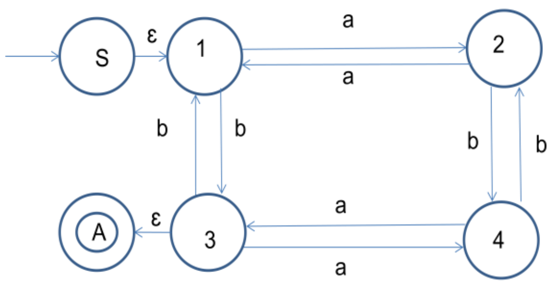
Example 4:A simple FSM with no simple RE

L = {w ε {a,b}* : w contains an even no of a's and an odd number of b's}



[3] even a's odd b's

Step 1: Add new start state S and new accepting state A.

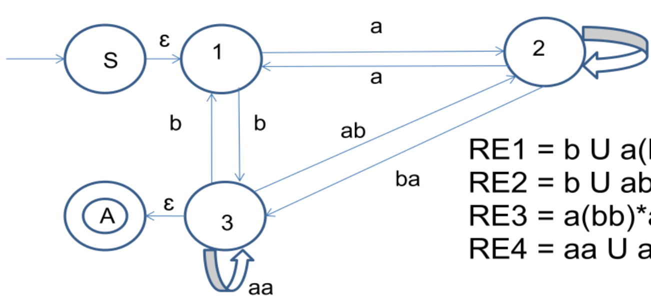


2-4-2: bb
3-4-3: aa
2-4-3: ba
3-4-2: ab

Step 2: let rip be state 4

Result after removing rip state 4

1-2-1: a(bb)*a
1-2-3: a(bb)*ba
3-2-1: ab(bb)*a
3-2-3: ab(bb)*ba

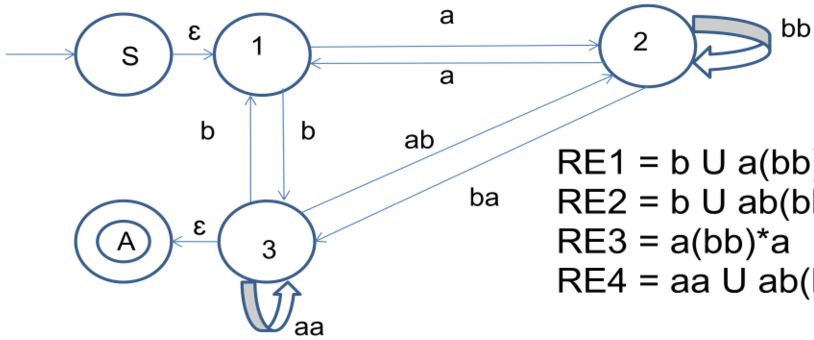


RE1 = b U a(bb)*ba
RE2 = b U ab(bb)*a
RE3 = a(bb)*a
RE4 = aa U ab(bb)*ba

Step 3: let rip be state 2

Result after removing rip state 4

1-2-1: a(bb)*a
1-2-3: a(bb)*ba
3-2-1: ab(bb)*a
3-2-3: ab(bb)*ba

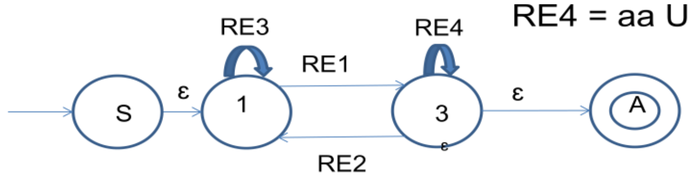


RE1 = b U a(bb)*ba
RE2 = b U ab(bb)*a
RE3 = a(bb)*a
RE4 = aa U ab(bb)*ba

Step 3: let rip be state 2

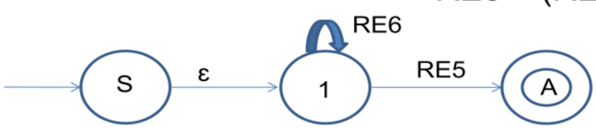
Redrawn

RE1 = b U a(bb)*ba
RE2 = b U ab(bb)*a
RE3 = a(bb)*a
RE4 = aa U ab(bb)*ba



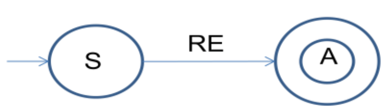
Step 4: let rip be state 3

RE5 = (RE1)(RE4)*
RE6 = (RE3) U (RE1)(RE4)*(RE2)



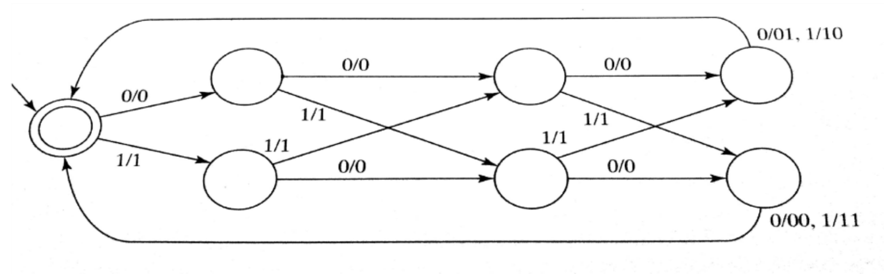
RE = (RE6)* (RE5)

Last Step: let rip be state 1



RE = (RE6)*(RE5)
= ((RE3) U (RE1)(RE4)*(RE2))*((RE1)(RE4)*)
= ((a(bb)*a) U (b U a(bb)*ba)(aa U ab(bb)*ba)*(b U ab(bb)*a))*((b U a(bb)*ba)((aa U ab(bb)*ba)*)

Example 5:Using fsmtoregexheuristic construct a RE for the following FSM(Example 5.3 from textbook)



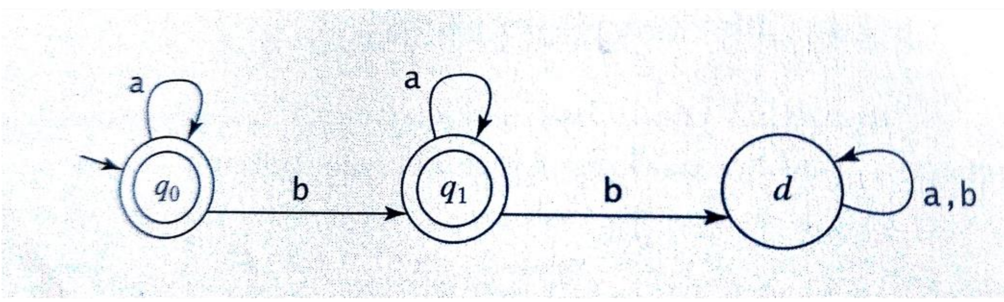
RE = (0000 U 0001 U 1100 U 1101 U 0010 U 1110 U
1100 U 0100 U 0011 U 1111 U 1101 U 0101)

Writing Regular Expressions

- Let L = {w ϵ {a,b}*: there is no more than one b}

  L = {**ε,**b,a,aa,ab,ba,aba,baa,abaa,aabaa,-----}

  RE = a*(b U ε)a*

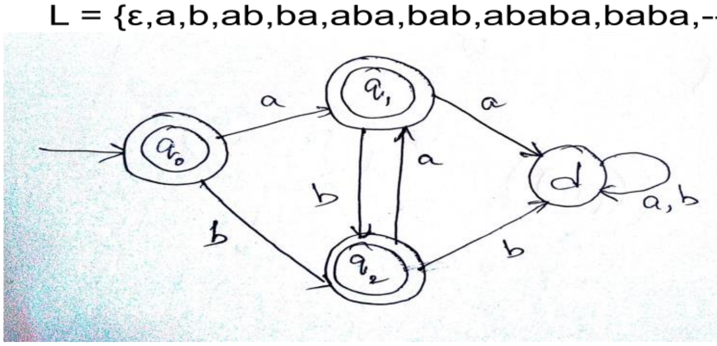


Writing Regular Expressions

- Let L = {w ϵ {a,b}*: No two consecutive letters are same}

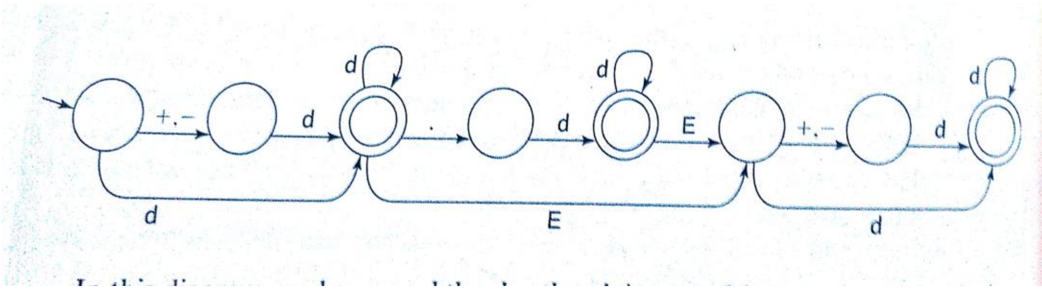
  RE = (b U ε)(ab)*(a U ε) or (a U ε)(ba)*(b U ε)

  L = {ε,a,b,ab,ba,aba,bab,ababa,baba,-----}



Writing Regular Expressions

- Floating point Numbers
  D stands for (0 U 1 U 2 U 3 U 4 U 5 U 6 U 7 U 8 U 9)
  RE = (ε U + U -)D⁺(ε U .D⁺)(ε U (E(ε U + U -)D⁺)
  L = { 24.06, +24.97E-05,------}



Building DFSM

- It is possible to construct a DFSM directly from a set of patterns
- Suppose we are given a set K of n keywords and a text string s.
- Find the occurences of s in keywords K
- K can be defined by RE

  $(\Sigma^*(K_1 \text{ U } K_2 \text{ U........U } Kn)\Sigma^*)^+$

- Accept any string in which at least one keyword occurs

Algorithm- buildkeywordFSM

- To build dfsm that accepts any string with atleast one of the specified keywords

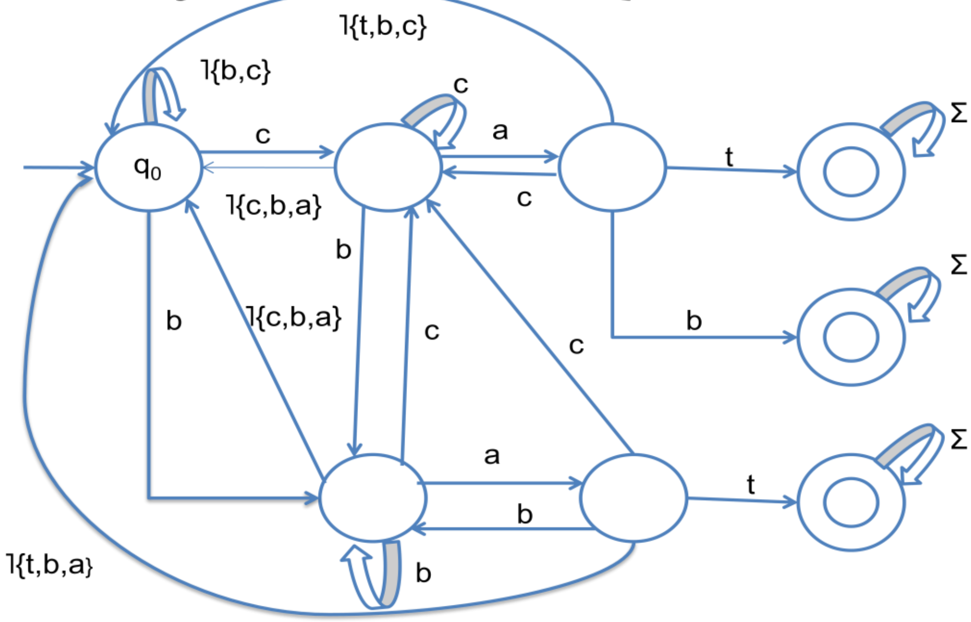
Buildkeyword(K:Set of keywords)

- Create a start state q_0
- For each element k of K do

  Create a branch corresponding to k

- Create a set of transitions that describe what to do when a branch dies
- Make the states at the end of each branch accepting

Ex:Keywords Set = {cat,bat,cab}



Applications Of Regular Expressions

- Many Programming languages and scripting systems provide support for regular expression matching
- Re's are used in emails to find spam messages
- Meaningful words in protein sequences are called motifs
- Used in lexical analysis
- To Find Patterns in Web
- To Create Legal passwords
- Regular expressions are useful in a wide variety of text processing tasks,

- More generally string processing, where the data need not be textual.
- Common applications include data validation, data scraping (especially web scraping), data wrangling, simple parsing, the production of syntax highlighting systems, and many other tasks.

RE for Decimal Numbers

RE = -? ([0-9]$^+$(\.[0-9]*)? | \.[0-9]$^+$)

- (α)? means the RE α can occur 0 or 1 time.
- (α)* means the RE α can repeat 0 or more times.
- (α)$^+$ means the RE α can repeat 1 or more times.

24.23,-24.23, .12, 12. ----- are some examples

Requirements for legal password

- A password must begin with a letter
- A password may contain only letters numbers and a underscore character
- A password must contain atleast 4 characters and no more than 8 characters

((a-z) U (A-Z))

((a-z) U (A-Z) U (0-9) U _)

((a-z) U (A-Z) U (0-9) U _)

((a-z) U (A-Z) U (0-9) U _)

((a-z) U (A-Z) U (0-9) U _ U ε)

((a-z) U (A-Z) U (0-9) U _ U ε)

((a-z) U (A-Z) U (0-9) U _ U ε)

((a-z) U (A-Z) U (0-9) U _ U ε)

Very lengthy regular expression

Different notation for writing RE

- α means that the pattern α must occur exactly once.
- α* means that the pattern may occur any number of times(including zero).
- α^+ means that the pattern α must occur atleast once.
- α{n,m} means that the pattern must occur **atleast n times** but not more than **m times**
- α{n} means that the pattern must occur **n times exactly**
- So RE of a legal password is :

RE = ((a-z) U (A-Z))((a-z) U (A-Z) U (0-9) U _){3,7}

Examples: RNSIT_17,Bangalor, VTU_2017 etc

- RE for an ip address is :

RE = ((0-9){1,3}(\.(0-9){1,3}){3})

Examples: 121.123.123.123

118.102.248.226

10.1.23.45

Manipulating and Simplifying Regular Expressions

Let α, β, γ represent regular expressions and we have the following identities.

1. Identities involving union

2. Identities involving concatenation

3. Identities involving Kleene Star

Identities involving Union

- Union is Commutative

α U β = β U α

- Union is Associative

  $(\alpha \cup \beta) \cup \gamma = \alpha \cup (\beta \cup \gamma)$

- Φ is the identity for union

  $\alpha \cup \Phi = \Phi \cup \alpha = \alpha$

- union is idempotent

  $\alpha \cup \alpha = \alpha$

- For any 2 sets A and B, if $B \subseteq A$, then $A \cup B = A$

  $a^* \cup aa = a^*$, since $L(aa) \subseteq L(a^*)$.

<u>Identities involving concatenation</u>

- Concatenation is associative

  $(\alpha\beta)\gamma = \alpha(\beta\gamma)$

- ε is the identity for concatenation

  $\alpha\varepsilon = \varepsilon\alpha = \alpha$

- Φ is a zero for concatenation.

  $\alpha\Phi = \Phi\alpha = \Phi$

- Concatenation distributes over union

  $(\alpha \cup \beta)\gamma = (\alpha\gamma) \cup (\beta\gamma)$

  $\gamma(\alpha \cup \beta) = (\gamma\alpha) \cup (\gamma\beta)$

<u>Identities involving Kleene Star</u>

- $\Phi^* = \varepsilon$
- $\varepsilon^* = \varepsilon$
- $(\alpha^*)^* = \alpha^*$
- $\alpha^*\alpha^* = \alpha^*$

- If $\alpha^* \subseteq \beta^*$ then $\alpha^*\beta^* = \beta^*$
- Similarly If $\beta^* \subseteq \alpha^*$ then $\alpha^*\beta^* = \alpha^*$

  a*(a U b)* = (a U b)*, since L(a*) ⊆ L((a U b)*).
- $(\alpha \cup \beta)^* = (\alpha^*\beta^*)^*$
- If $L(\beta) \subseteq L(\alpha)$ then $(\alpha \cup \beta)^* = \alpha^*$

(a U ε)* = a*,since {ε} ⊆ L(a*).

Simplification of Regular Expressions

1. ((a* U Φ)* U aa) = (a*)* U aa //L(Φ) ⊆ L(a*)

   = a* U aa //(α*)* = α*

   = a* // L(aa) ⊆ L(a*)

2. (b U bb)*b* = b*b* //L(bb) ⊆ L(b*)

   = b* // α*α* = α*

3. ((a U b)* b* U ab)*

   = ((a U b)* U ab)* //L(b*) ⊆ L(a U b)*

   = (a U b)* //L(a*) ⊆ L(a U b)*)

4. ((a U b)* (a U ε)b* = (a U b)* //L((a U ε)b*) ⊆ L(a u b)*

5. (Φ* U b)b* = (ε U b)b* //Φ* = ε

   = b* //L(ε U b) ⊆ L(b*)

6. (a U b)*a* U b = (a U b)* U b // L(a*) ⊆ L((a U b)*)

   = (a U b)* // L(b) ⊆ L((a U b)*)

7.((a U b)$^+$)* = (a U b)*

Chapter-7

Regular Grammars

Regular grammars sometimes called as right linear grammars.

A regular grammar G is a quadruple (V, ∑ , R, S)

- V is the rule alphabet which contains nonterminals

  and terminals.

- ∑ (the set of terminals) is a subset of V
- R (the set of rules) is a finite set of rules of the form

  X → Y

- S (the start symbol) is a nonterminal.

All rules in R must:

- Left-hand side should be a single nonterminal.
- Right-hand side is **ε** or a single terminal or a single terminal followed by a single nonterminal.

Legal Rules

S→a

S→ε

T→aS

Not legal rules

S→aSa

S→TT

aSa→T

S→T

- The language generated by a grammar G = (V, ∑ , R, S) denoted by L(G) is the set of all strings w in ∑* such that it is possible to start with S.
- Apply some finite set of rules in R, and derive w.
- Start symbol of any grammar G will be the symbol on the left-hand side of the first rule in R_G

Example of Regular Grammar

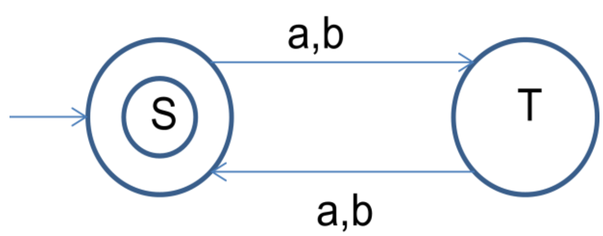
Example 1:Even Length strings

Let L = {w∈ {a, b }*: lwl is even}.

The following regular expression defines L:

((aa) U (ab) U (ba) U (bb))* or ((a U b)(a U b))*

DFSM accepting L



Regular Grammar G defining L

S→ε

S→aT

S→bT

T→aS

T→bS

Derivation of string using Rules

Derivation of string "abab"

S => aT

=> abT

=> abaS

=> ababS

=> abab

Regular Grammars and Regular Languages

THEOREM

Regular Grammars Define Exactly the Regular Languages

Statement:

The class of languages that can be defined with regular grammars is exactly the regular languages.

Proof: Regular grammar → FSM

FSM → Regular grammar

The following algorithm constructs an FSM M from a regular grammar G = (V, ∑ , R, S) and assures that

L (M) = L (G):

Algorithm-Grammar to FSM

grammartofsm (**G: regular grammar**) =

1. Create in M a separate state for each nonterminal in V.
2. Make the state corresponding to S the start state.
3. If there are any rules in R of the form X→w, for some

   w ϵ ∑, then create an additional state labeled #.

4. For each rule of the form X→ wY,

add a transition from X to Y labeled w.

5. For each rule of the form X→w, add a transition from X to # labeled w.
6. For each rule of the form X→**ε**, mark state X as accepting.
7. Mark state # as accepting.
8. If M is incomplete then M requires a dead state.

   Add a new state D. For every (q, i) pair for which no transition has already been defined, create a transition from q to D labeled i. For every i in Σ, create a transition from D to D labeled i.

<u>Example 2:Grammar→FSM</u>

Strings that end with aaaa

Let L = {w∈ {a, b }*: w end with the pattern aaaa}.

RE = (a U b)*aaaa

Regular Grammar G

S→aS

S→bS

S→aB

B→aC

C→aD

D→a

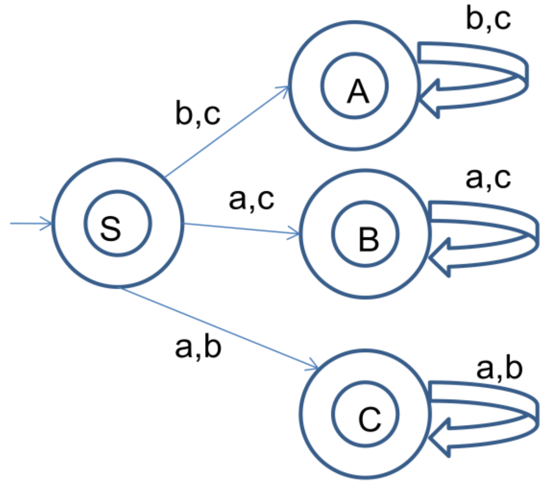
Example 3:The Missing Letter Language

Let ∑ = {a, b, c}.

$L_{Missing}$ = { w : there is a symbol a € ∑ not appearing in w}.

Grammar G generating $L_{Missing}$

FSM for Missing Letter Language



S→ε
S→aB
S→aC
S→bA
S→bC
S→cA
S→CB

A→ ε
A→bA
A→cA
B→ ε
B→aB
B→cB

C→ ε
C→aC
C→bC

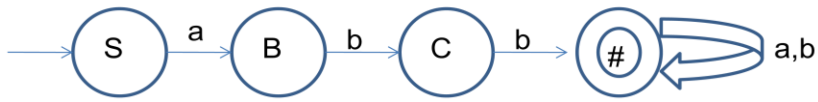
Example 4 :Strings that start with abb.

Let L = {w ϵ {a, b }*: w starting with string abb}.

RE = abb(a U b)*

Regular Grammar G

S→aB
B→bC
C→bT
T→aT
T→bT
T→ε



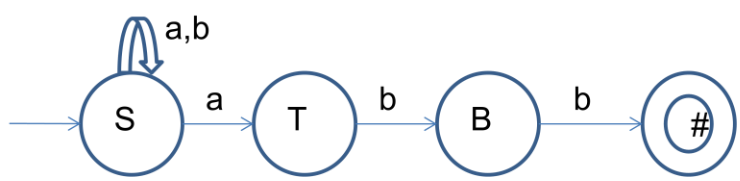
Example 5 :Strings that end with abb.

Let L = {w ϵ {a, b }*: w ending with string abb}.

RE = (a U b)*abb

Regular Grammar G

S→aS
S→bS
S→aT
T→bB
B→b



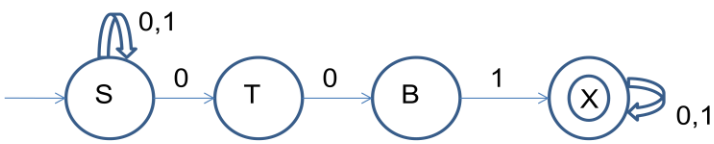
Example 6:Strings that contain substring 001.

Let L = {w ϵ {0, 1 }*: w containing the substring 001}.

RE = (0 U 1)*001(0 U 1)*

Regular Grammar G

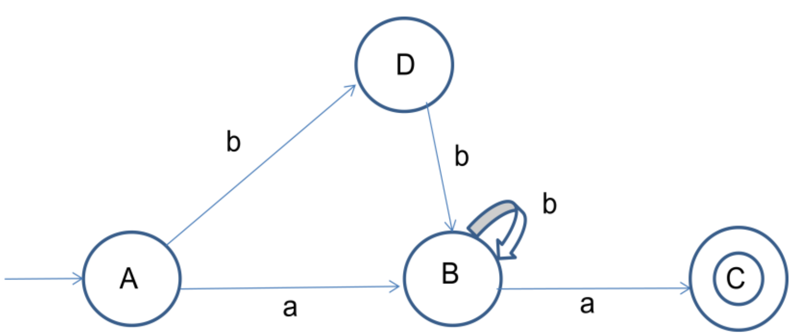
S→0S
S→1S
S→0T
T→0P
P→1X
X→0X
X→1X
X →ε



<u>Algorithm FSM to Grammar</u>

1. Make M deterministic (to get rid of ε-transitions).
2. Create a nonterminal for each state in the new M.
3. The start state becomes the starting nonterminal.
4. For each transition δ(T, a) = U, make a rule of the form T → aU.
5. For each accepting state T, make a rule of the form T → ε.

Example 7:Build grammar from FSM



RE = (a U bb)b*a

Grammar

A→aB

A→bD

B→bB

B→aC

D→bB

C→ε

Derivation of string "aba"

A => aB

=> abB

=> abaC

=> aba

Derivation of string "bba"

A => bB

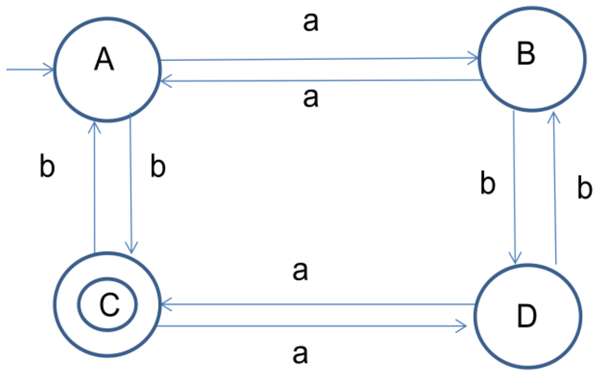
=> bbB

=> bbaC

=> bba

<u>Example 8:A simple FSM with no simple RE</u>

L = {w ε {a,b}* : w contains an even no of a's and an odd number of b's}



Grammar

A→aB

A→bC

B→aA

B→bD

C→bA

C→aD

D→bB

D→aC

C→ε

Derivation of string "ababb"

A => aB

=> abD

=> abaC

=> ababA

=> ababbC

=> ababb

RE,RG and FSM for given Language

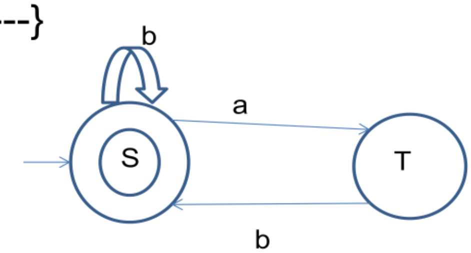
Let L = { w϶ {a, b }*: every a in w is immediately followed by atleast one b.}

L = { b,ab,abab,abb,-------}

RE = (ab U b)*

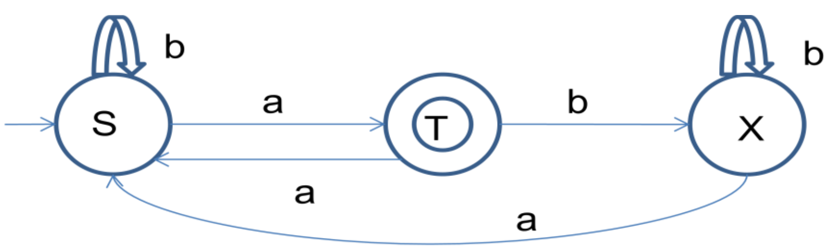
Regular Grammar

S→aT
S→bS
S→ε
T→bS



Satisfying Multiple Criteria

Let L = { w϶ {a, b }*: w contain an **odd number of a's** and **w ends in a**}.



S→bS

S→aT

T→ ε

T→aS

T→bX

X→aS

X→bX

Conclusion on Regular Grammars

- Regular grammars define exactly the regular languages.
- But regular grammars are often used in practice as FSMs and REs are easier to work.
- But as we move further there will no longer exist a technique like regular expressions.
- So we discuss about context-free languages and context-free-grammars are very important to define the languages of push-down automata.

Chapter-8

Regular and Nonregular Languages

- The language a*b* is regular.
- The language $A^nB^n = \{a^nb^n : n>=0\}$ is not regular.
- The language {w Є {a,b}*:every a is immediately followed by b} is regular.
- The language {w Є {a, b}*:every a has a matching b somewhere and no b matches more than one a} is not regular.

- Given a new language L, how can we know whether or not it is regular?

Theorem 1: The Regular languages are countably infinite

Statement:

There are countably infinite number of regular languages.

Proof:

- We can enumerate all the legal DFSMs with input alphabet ∑.
- Every regular language is accepted by at least one of them.

- So there cannot be more regular languages than there are DFSMs.

- But the number of regular languages is infinite because it includes the following infinite set of languages:

  {a}, { aa} , { aaa}, { aaaa}. { aaaaa}, { aaaaaa } ----

- Thus there are at most a countably infinite number of regular languages.

Theorem 2 : The finite Languages

Statement: Every finite language is regular.

Proof:

- If L is the empty set, then it is defined by the R.E Ø and so is regular.
- If it is any finite language composed of the strings $s_1, s_2, \ldots s_n$ for some positive integer n, then it is defined by the R.E: $s_1 \cup s_2 \cup \ldots \cup s_n$
- So it too is regular

❖ Regular expressions are most useful when the elements of L match one or more patterns.

❖ FSMs are most useful when the elements of L share some simple structural properties.

Examples:

- **L_1 = {w Є {0-9}*: w is the social security number of the current US president}.**

  L_1 is clearly finite and thus regular. There exists a simple FSM to accept it.

- **L_2 = {1 if Santa Claus exists and 0 otherwise}.**
- **L_3 = {1 if God exists and 0 otherwise}.**

  L_2 and L_3 are perhaps a little less clear.

  So either the simple FSM that accepts { 0} or the simple FSM that accepts { 1} and nothing else accepts L_2 and L_3.

- **L_4 = {1 if there were people in north America more than 10000 years age and 0 otherwise}.**
- **L_5 = {1 if there were people in north America more than 15000 years age and 0 otherwise}.**

L_4 is clear. It is the set { 1}.

L_5 is also finite and thus regular.

- **L_6 = {w Є {0-9}*: w is the decimal representation, without leading 0's, of a prime Fermat number}**

- Fermat numbers are defined by

  $$F_n = 2^{2^n} + 1, \quad n >= 0.$$

- The first five elements of F are {3, 5, 17, 257,65537}.
- All of them are prime. It appears likely that no other Fermat numbers are prime. If that is true,then L_6 is finite and thus regular.

- lf it turns out that the set of Fermat numbers is infinite,then it is almost surely not regular.

Four techniques for showing that a language L(finite or infinite) is regular:

1. Exhibit a R.E for L.
2. Exhibit an FSM for L.
3. Show that the number of equivalence of $\approx_L$ is finite.
4. Exhibit a regular grammar for L.

Closure Properties of Regular Languages

The Regular languages are closed under

- Union
- Concatenation
- Kleene star
- Complement
- Intersection
- Difference
- Reverse
- Letter substitution

Closure under Union, Concatenation and Kleene star

Theorem: The regular languages are closed under union, concatenation and Kleene star.

Proof: By the same constructions that were used in the proof of Kleene's theorem.

Closure under Complement

Theorem:

The regular languages are closed under complement.

Proof:

- If L_1 is regular, then there exists a DFSM $M_1=(K,\sum,\delta,s,A)$ that accepts it.
- The DFSM $M_2=(K, \sum,\delta,s,K-A)$, namely M_1 with accepting and nonaccepting states swapped, accepts $\neg(L(M_1)$ because it rejects all strings that M_1 accepts and rejects all strings that M_1 accepts.

Steps:

1. Given an arbitrary NDFSM M_1,construct an equivalent DFSM M' using the algorithm ndfsmtodfsm.
2. If M_1 is already deterministic, M' = M_1.
3. M' must be stated completely, so if needed add dead state and all transitions to it.
4. Begin building M_2 by setting it equal to M'.
5. Swap accepting and nonaccepting states. So $M_2=(K, \sum,\delta,s,K-A)$

Example:

- Let L = {w Є {0,1}* : w is the string ending with 01}
  RE = (0 U 1)*01
- The complement of L(M) is the DFSM that will accept strings that do not end with 01.

Closure under Intersection

Theorem:

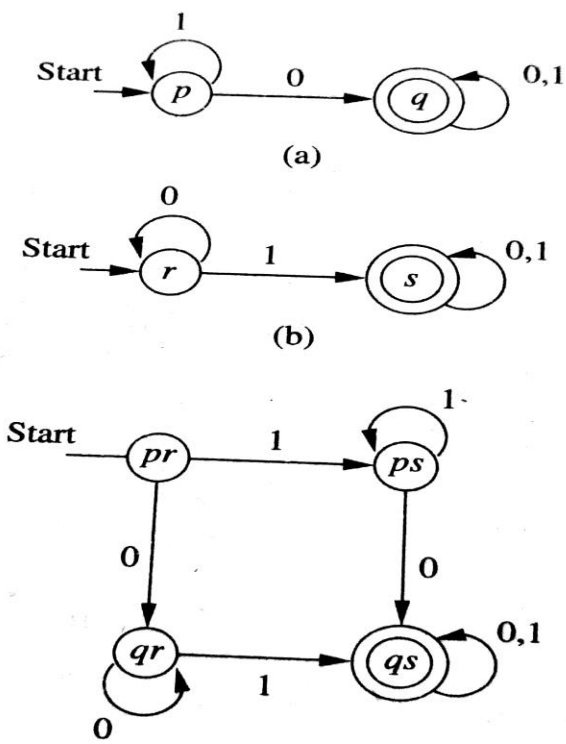
The regular languages are closed under intersection.

Proof:

- Note that

$L(M_1) \cap L(M_2) = \neg(\neg L(M_1) \cup \neg L(M_2))$.

- We have already shown that the regular languages are closed under both complement and union.
- Thus they are closed under intersection.
- Example:



- Fig (a) is DFSM L1 which accepts strings that have 0.
- Fig(b) is DFSM L2 which accepts strings that have 1.

- Fig(c) is Intersection or product construction which accepts that have both 0 and 1.

The Divide and Conquer Approach

- Let L = {w Є {a,b}* : w contains an even number of a's and an odd number of b's and all a's come in runs of three }.
- L is regular because it is the intersection of two regular languages,

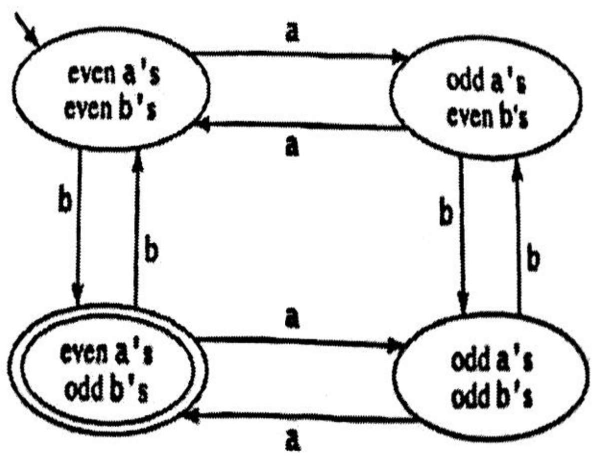
  $L = L_1 \cap L_2$, where

- L1 = {w Є {a,b}* : w contains an even number of a's

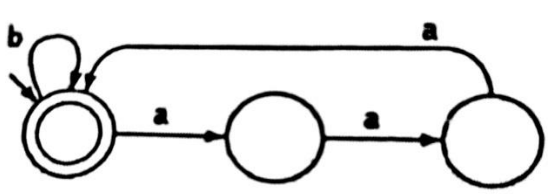
  and an odd number of b's},and

  L2 = {w Є {a,b}*: all a's come in runs of three}.

- L1 is regular as we have an FSM accepting L1



- L2 = {w Є {a,b}*: all a's come in runs of three}.
- L2 is regular as we have an FSM accepting L2



L = {w Є {a,b}* : w contains an even number of a's and an odd number of b's and all a's come in runs of three }.

L is regular because it is the intersection of two regular languages,L = $L_1 \cap L_2$

Closure under Set difference

Theorem:

The regular languages are closed under set difference.

Proof:

$L(M_1) - L(M_2) = L(M_1) \cap \neg L(M_2)$

- Regular languages are closed under both complement

and intersection is shown.

- Thus regular languages are closed under set difference.

Closure under Reverse

Theorem:

The regular languages are closed under reverse.

Proof:

- L^R = { w Є ∑* : w = x^R for some x Є L}.

Example:

1. Let L = {001,10,111} then L^R = {100,01,111}
2. Let L be defined by RE (0 U 1)0* then L^R is 0*(0 U 1)

reverse(L) = {x ∈ Σ* : x = w^R for some w ∈ L}.

By construction.

- Let M = (K, Σ, δ, s, A) be any FSM that accepts L.
- Initially, let M′ be M.

- Reverse the direction of every transition in M′.
- Construct a new state q. Make it the start state of M′.
- Create an ε-transition from q to every state that was an accepting state in M.
- M′ has a single accepting state, the start state of M.

Closure under letter substitution or Homomorphism

- The regular languages are closed under letter substitution.
- Consider any two alphabets, $\sum_1$ and ∑2.
- Let **sub** be any function from $\sum_1$ to $\sum_2^*$.
- Then **letsub** is a letter substitution function from L_1 to L_2 iff **letsub**(L_1) = { w Є $\sum_2^*$:Ǝy Є L_1(w = y except that every character c of y has been replaced by **sub**(c))}.
- Example 1

Consider $\sum_1$ = {a,b} and $\sum_2$ = {0,1}

Let **sub** be any function from $\sum_1$ to $\sum_2^*$.

sub(a) = 0, sub(b) = 11

letsub**(**$a^n b^n$: n >= 0}) = { $0^n 1^{2n}$: n >= 0}

- Example 2

Consider $\sum_1$ = {0,1,2} and $\sum_2$ = {a,b}

Let **h** be any function from $\sum_1$ to $\sum_2^*$.

h(0) = a, h(1) = ab, h(2) = ba

h(0120) = h(0)h(1)h(2)h(0)

= aabbaa

h(01*2) = h(0)(h(1))*h(2)

= a(ab)*ba

Long Strings Force Repeated States

Theorem: Let M=(K,∑,δ,s,A) be any DFSM. If M accepts any string of length |K | or greater, then that string will force M to visit some state more than once.

Proof:

- M must start in one of its states.
- Each time it reads an input character, it visits some state. So ,in processing a string of length n, M creates a total of n+1 state visits.
- If n+1 > | K |, then, by the pigeonhole principle, some state must get more than one visit.
- So, if n>= | K |,then M must visit at least one state more than once.

<u>The Pumping Theorem for Regular Languages</u>

Theorem: If L is regular language, then:

Ǝk >= 1 (∀strings w ϵ L, where |w| >= k (Ǝx, y, z (w = xyz,

|xy| <= k,

y ≠ **ε**,and

∀q >= 0(xy^qz ϵ L)))).

Proof:

- If L is regular then it is accepted by some DFSM M=(K,∑,δ,s,A).

  Let k be |K|
- Let w be any string in L of length k or greater.
- By previous theorem to accept w, M must traverse some loop at least once.

- We can carve w up and assign the name y to the first substring to drive M through a loop.
- Then x is the part of w that precedes y and z is the part of w that follows y.
- We show that each of the last three conditions must then hold:
- **|xy| <= k**

  M must not traverse thru a loop.

  It can read k - 1 characters without revisiting any states.

  But kth character will take M to a state visited before.
- **y ≠ ε**

  Since M is deterministic, there are no loops traversed by **ε.**
- **∀q >= 0 (xy^qz ϵ L)**

  y can be pumped out once and the resulting string must

  be in L.

Steps to prove Language is not regular by contradiction method.

1. Assume L is regular.
2. Apply pumping theorem for the given language.
3. Choose a string w, where w ϵ L and IwI >= k.
4. Split w into xyz such that |xy| <= k and y ≠ **ε.**
5. Choose a value for q such that xy^qz is not in L.
6. Our assumption is wrong and hence the given language is not regular.

Problems on Pumping theorem (Showing that the language is not regular)

1. **Show that A^nB^n is not Regular**

Let L be $A^nB^n = \{ a^nb^n : n >= 0\}$.

Proof by contradiction.

Assume the given language is regular.

Apply pumping theorem and split the string w into xyz

Choose w to be a^kb^k (We get to choose any w).

1 | 2

a a a a a … a | a a a a b b b b … b b b b b b

x y z

We show that there is no x, y, z with the required properties:

$k \leq |xy|$,

$y \neq \varepsilon$

$\forall$ q > = 0 (xy^qz is in L y must be in region 1.

So y = a^p≤Since |xy| 1.≥for some p Let q = 2, producing: ak+pbk L, since it has more a∉which 's than b' s.

2. **$\{a^ib^j : i, j \geq 0$ and $i - j = 5\}$.**

- Not regular.
- L consists of all strings of the form a*b* where the number of a's is five more than the number of b's.
- We can show that L is not regular by pumping.
- Let w = $a^{k+5}b^k$.
- Since $|xy| \leq k$, y must equal a^p for some p > 0.

- We can pump y out once, which will generate the string $a^{k+5-p}b^k$, which is not in L because the number of a's is is less than 5 more than the number of b's.