Unit I

Ch 5 Finite Automata

DATE: / /
PAGE NO.: 1

Another method for Defining Languages :-

Game - playing board & pieces, Dice are thrown, generate random no.'s accordingly pieces are rearranged using rules. Child has no option / skill to change board. Whether white pieces win the game is dependent on sequence of numbers generated by dice.

States - all possible positions of pieces on board. Game changes from one state to another in a fashion determined by i/p of certain No. For each possible No. there is one & only one resulting state. There is possibility that after a No. is entered, the game is still in the same state. [ex. player is in jail & needs 2 to come out] After certain No. of rolls, board arrives at a state that means victory & game is over. This is final state. There might be many possible final states. In comp^n theory, these are also called as halting states; terminal / accepting st.

Beginning with initial state (unique), some i/p sequences of Nos. lead to victory & some do not.

ex. Child has a comp^n (i/p device, processing unit, memory, o/p device) & wishes to calculate 3+4. He writes prog.— sequence of instructions that are fed into m/c one at a time. Each instn is executed as soon as it is read. If all goes well, m/c generates o/p 7. & terminates exec^n. This is similar to board game. Here board in comp^n & diff arrangements of pieces on the board corresponds to " " of 0's & 1's in the cells of memory. 2 m/c's are in same state if their o/p pages look the same & their memories look same cell by cell.

Comp^n is deterministic, on i/p instn., m/c changes state (or remains on same state) based on prior state & i/p. No choice involved, no knowledge reqd. of state the m/c was in & instn. ago.

Some sequences of i/p instns. may lead to success & some may not.

In ex.1, consider set of all possible dice rolls to be letters of an alphabet. Define lang as set of strings of those letters that lead to success.
ex.2, consider set of all comp^{n} instr^{s} as letters of alphabet. Define lang. to be set of all words over this alphabet that lead to success.
This is a lang. whose words are all prog that print a 7.

The most general model, of which both of these ex. are instances, is called finite automaton – finite : No. of possible states & No. of letters in the alphabet are both finite & automaton because change of states is totally governed by i/p. The determination of what state is next is automatic (involuntary & mechanical) motion of hands of clock vs hands of human.
↳ automatic. ↳ will full

Plural of automaton is automata. accept str & reject. ↗ does not print o/p or play music.
(FA) (finite acceptor). [lang recognizer] whether i/p is lan
Def^{n} – A finite automaton is a collection of 3 things:
① A finite set of states, one is start & some are final states.
② An alphabet Σ of possible i/p letters.
③ finite set of transitions that tell for each state & for each letter of the i/p alphabet which state to go to next.

working – i/p str. of letters. Begin at start state, letters determine sequence of states.

FA with o/p – does something.

ex. $\Sigma = \{a, b\}$, 3 states - x, y, z.

transition rules -

1. from state x & i/p a, goto st. y.
2. — " — x b, " z
3. — " — y a " x
4. — " — y b " z
5. — " — z a/b, stay at z.

ex. i/p - aaa - unsuccessful termination of run.
It is not in the lang. of all str. that leave this FA in z.
Set of all str. that do leave us in a final state is called the lang. defined by the FA.

aaa is not accepted or is rejected by FA.
Set of all str. accepted is the lang. associated with FA.
FA accepts lang. L. , L is lang. of FA; lang. associated with FA.
If L_1 is contained in L_2 & FA accepts L_2 then it also accepts L_1 (all words in L_2 are accepted & all i/ps accepted are words in L_2) then this FA must also accept all words in L_1 (because they are also words in L_2).
However we don't say "L_1 is accepted by FA" ∵ it means that all words FA accepts are in L_1.

ex. aabca is accepted by FA.
If i/p has just a, it won't be accepted. It needs a b. to reach final state.
lang. is defined by RE: $(a+b)^* b (a+b)^*$.

Transition table - rows — states in FA,
cols — letters in alphabet.
entries — transition states.

	a	b
start x	y	z
y	x	z
final z	z	z

defines FA.

FA is considered to be m/c., having dynamic capabilities. It moves, processes i/p.

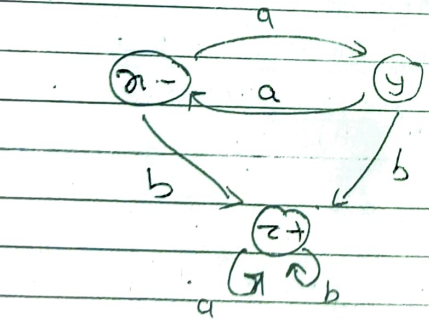
Replace transition table with transition f^n δ, whose i/p is pair of state & alphabet letter & o/p is single state.

Abstract def^n of FA:

1. Finite set of states $Q = \{q_0, q_1, q_2, \dots\}$, q_0 - start
2. Subset of Q called final states.
3. alphabet $\Sigma = \{x_1, x_2, x_3, \dots\}$.
4. transition f^n δ associating each pair of state & letter with a state:

$$\delta(q_i, x_j) = x_k.$$

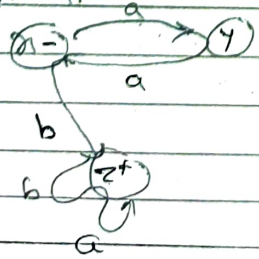
Transition diagram – gives feel of motion.
states – circles — directed graph.
arrows – transitions (loops) — (directed) edge.
label – letters.
start state – label "start" or minus sign.
final " – " final or plus "



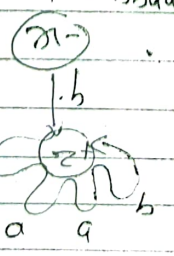
OR start st. indicated by arrow & final by drawing a box or another circle.
OR – & + signs are drawn inside or outside circle.

OR → (a) (z) or start (x) (z) final

ex. i/p - aabba

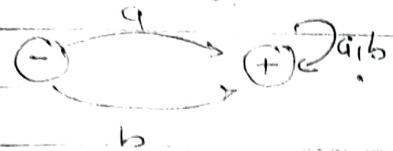


ex i/p - bbaab

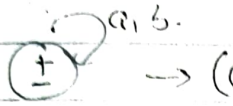


Every state has as many outgoing edges as there are letters in alphabet.
A state can have no incoming edges or many.

It is not necessary to give specific names to states.

ex. 

lang. accepted by FA is set of all str. except Λ:
$(a+b)(a+b)^* = (a+b)^+$

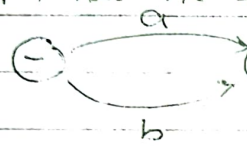
ex. 

$\rightarrow (a+b)^*$ accepts all words.

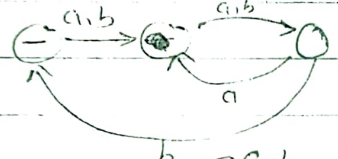
convention — Λ starts in FA state & ends right there on all FAs.

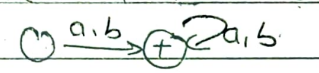
ex. FA accepts no lang.

(1) FA has no final states



(2) Final state can't be reached from start state.

ex. 



disconnected graph.

ex. 

- FAs & their languages —

(1) Start with m/c & analyze it to see what lang. it accepts. or

(2) Start with desired lang. in mind & construct FA as lang. recognizer or definer. (for practical algorithmic def.)

Comparison: RE

(1) When lang. is defined by a RE, it is easy to produce some arbitrary words that are in the lang. by making set of choices for + & *'s, but it is harder to recognize whether a given string of letters is or is not in the lang. defined by the expression.

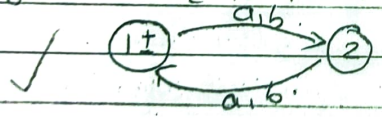
(2) It is just opposite with FA. If we are given a specific str., we can decide by an algorithmic proc. whether or

not if it is in the lang. defined by the mlc.
But given a lang. defined by FA, it is not so easy to write down a bunch of words that we know in advance the mlc will accept.

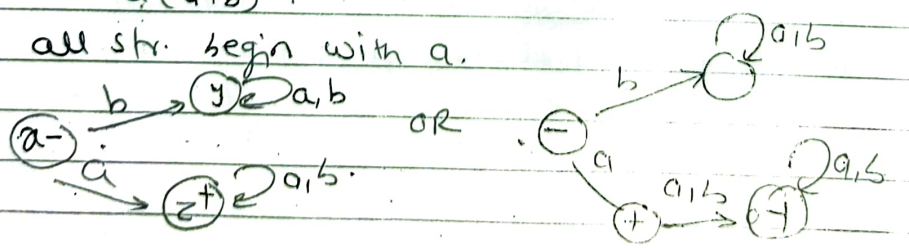
ex. Build a mlc that accepts lang. of all words over alphabet {a b} with even No. of letters.
→ Mathematical approach – count letters in i/p & decide.
It involves many calculations to generate extraneous info.
Programming approach – use Boolean flag E. initially TRUE.
On reading a letter from i/p, reverse its value.
At end, if TRUE then even letters.

```
prg:  E = TRUE
      while not out of data do
          read i/p letter
          E = not(E).
      if E = TRUE, accept i/p str
          else reject.
```

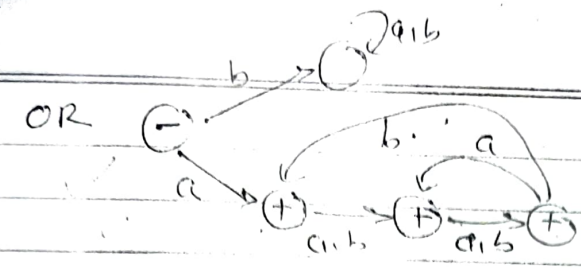
Comp^{n}. employs only one storage loc^{n} which contains one of 2 diff. values, FA requires only 2 states:
State 1 : E is TRUE, start & final state
State 2 : E is FALSE



ex. Build FA that accepts all words in lang $a(a+b)^*$.
– all str. begin with a.



all other words starting with an a reach 2nd + state.
Word a ends in 1st + state.



There is not a unique m/c for given lang.

Is there always at least one FA that accepts each possible lang. ?

Every lang. that can be accepted by an FA can be defined by a RE & every lang. that can be ~~accepted~~ defined by ~~FA~~ RE can be ~~defined by RE~~ accepted FA.

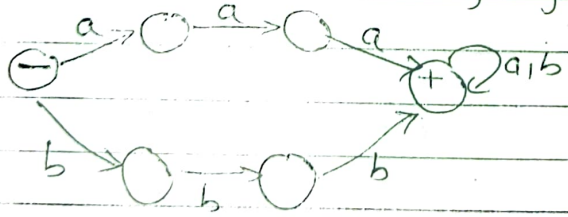
There are lang. that ~~can~~ are neither definable by RE nor accepted by FA.

For a lang. to be the lang. accepted by an FA means not only that all the words in the lang. run to final states, but also that no strings not in the lang. do.

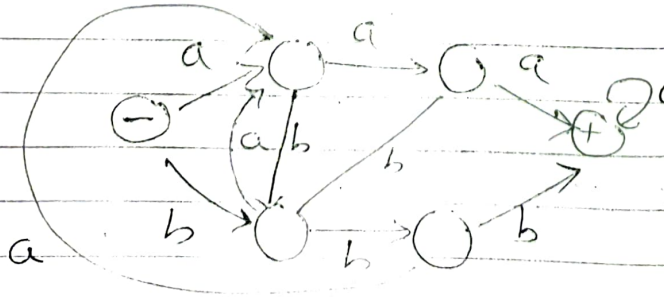
ex. Build FA that accepts all words containing a triple letters, either aaa or bbb & only those words.

→ Consider aaa

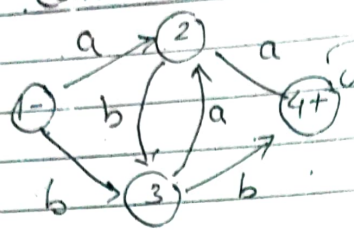
for bbb, need 2 separate states, otherwise we could mix a's & b's & mistakenly get to t.



If we are moving anywhere along a-path & we read a b before the 3rd a, we jump to b-path in progress & vice versa.



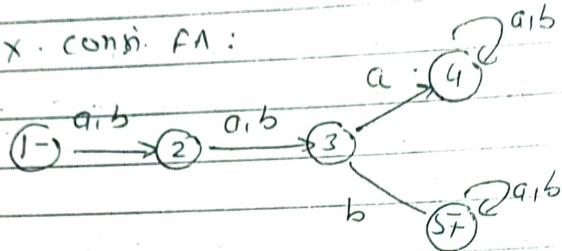
ex. consider FA:



⇒ words that have a double letter.

⇒ $(a+b)^*(aa+bb)(a+b)^*$.

ex. consi. FA:

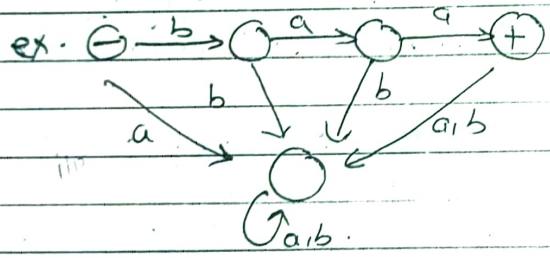


accepts strings with minimum 3 letters where 3rd letter must be b.

R.E. $(aab+abb+bab+bbb)(a+b)^*$ or

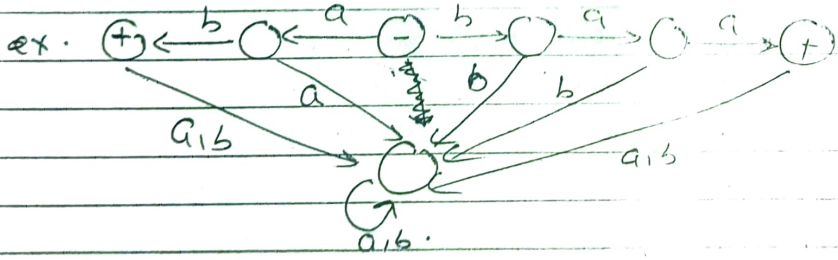
$(a+b)(a+b)(b)(a+b)^* = (a+b)^2 b (a+b)^*$.

↓ (not RE ∵ 2 is not in def?

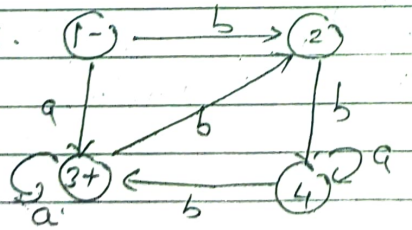


Only word accepted is baa

∴ $L = \{baa\}$.



Only 2 str. baa & ab. Big m/c, small lang.



ex.

It accepts aa^*,

If str has b, no. of b's must be divisible by 3.

i.e. str with tot. no. of b's divisible by 3 is accepted.

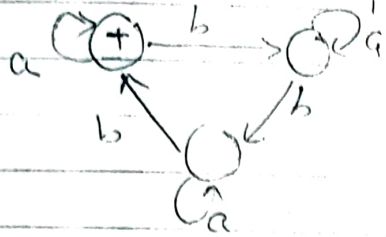
R.E. $a^*(a^*ba^*ba^*ba^*)^*(a+a^*ba^*ba^*ba^*)$

↓

to guarantee that Λ is not a possibility because it is not accepted by m/c

DATE: / /
PAGE NO.:

If Λ is included, FA:

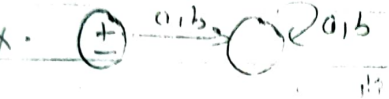


RE : $(a+ba^*ba^*b)^+$

↳ for original FA

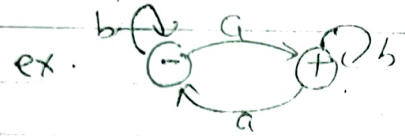
RE : $(a^*ba^*ba^*ba^*)^*$ +

$(a^+ + ba^*ba^*b)^*$

ex.  accepts only Λ. all other words go to right state & stay there.

$(a+b)^* a$, any str. ends with a.

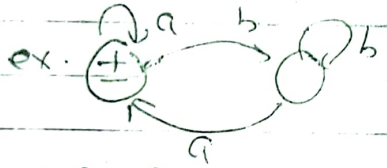
OR $b^* a^+$ & this won't accept aba



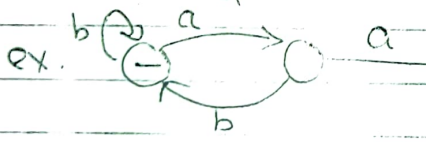
words with odd No. of a's.

$b^*ab^*(ab^*ab^*)^*$.

$b^*a[b^*(ab^*a)^*]^*$

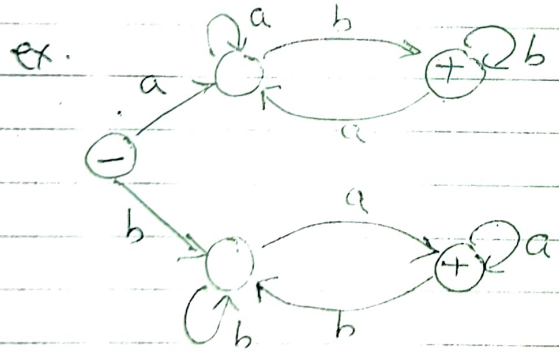


includes Λ, words that do not end in b. $((a+b)^*a)^*$



— all words with double a.

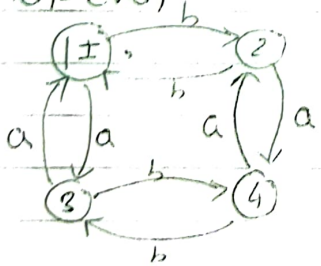
$(a+b)^*aa(a+b)^*$.



all words that have diff. 1st & last letters.

$(a^+(a+b)^*b^*)+(b^+(a+b)^*a^*)$ &

ex. Even-Even



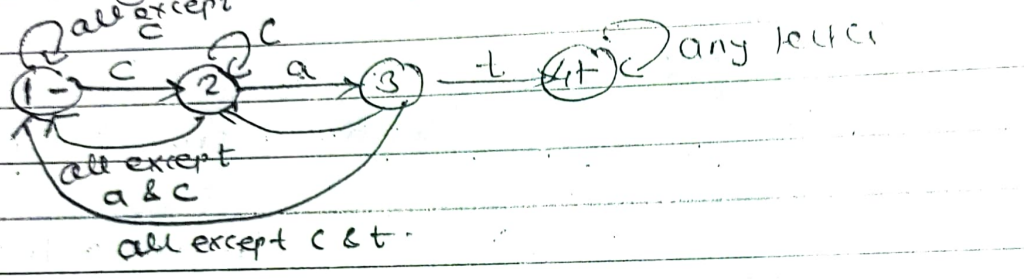
All words that return to state 1 are in Even-Even.

State 2 — even no. of a's & odd no. of b's.

St. 3 — even b's, odd a's.

St. 4 — odd a, odd b.

ex. We are programmers hired to create a word proc?. As a part of this major prog., we must build a subroutine that scans i/p str. & spaces & locates the occurence of substring cat whether it is a word standing alone or part of a longer word.

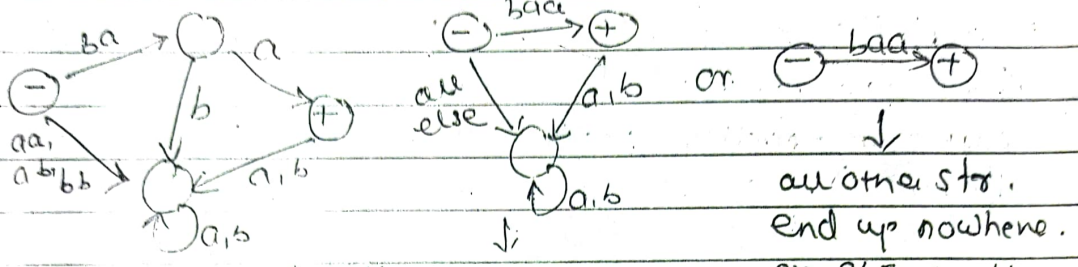


DATE: / /
PAGE NO.:

Ch.6 Transition Graphs.

Relaxing the restriction on inputs -

FA reads only 1 letter from i/p str. at a time. We design powerful m/c that could read 1 or 2 letters. ex. accept baa.

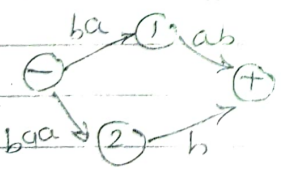


for all other str. go to garbage collection state.

Assume that there is some trash-can state, in case of failure. last 2 fig. above are equivalent, accept exact same lang. trivial difference — diff. no. of states.

Def.^n When an i/p string that has not been completely read reaches a state (final or otherwise) that it can't leave because there is no outgoing edge that it may follow, we say that the i/p (or the m/c) crashes at that state. Exec^n then terminates & i/p must be rejected.

With this approach, there is foll. problem:



We can accept baab in 2 ways.
i) ba, ab ii) baa, b.

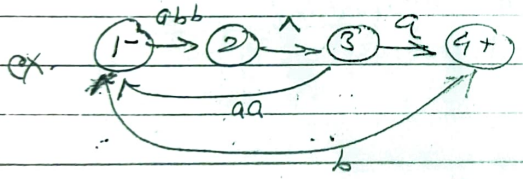
Previously we had unique path for every i/p str., now some have no paths while some have several.

The new m/c with word-labeled edges (reading more than 1 letter of i/p at a time)., is called as transition graph (instead of TA) because they are easily understood when defined directly as graphs than as tables later turned into pictures.

Transi. Gr. TG is a collection of 3 things:
1. A finite set of states, at least one is start (→) & some (none) are final states (+)
2. An alphabet Σ of possible i/p letters from which i/p str. are formed.
3. A finite set of transitions (edge labels) that show how to go from some states to some others, based on reading specified substrings of i/p letters (even Λ).

Every edge is labeled by some str., not necessarily 1 i/p letter. Some states may have no edge coming out of them at all & some may have thousands (eg. edges labeled a, aa, aaa, ...)

successful path – series of edges from start to final state



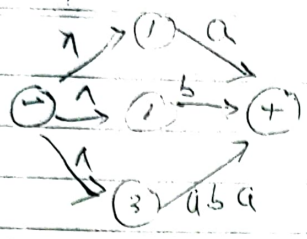
Str. accepted by TG:
abbaab = (abb)(Λ)(aa)(b)
abba
abbaaabba
b.

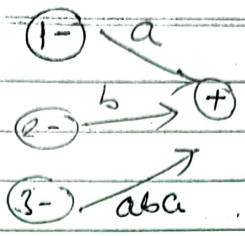
When edge is labeled with str. Λ, we need not follow the edge, but we can if we want to without consuming any letters from i/p str.

Given i/p str. to run on a given TG, we must decide how to break the word into substrings that might correspond to the labels of edges in a path.
ex. abbab can be factored as (abb)(Λ)(a) but b is left : rejected.

We can introduce more start states, all by connecting them to original start state by edges labeled Λ.
ex. There is no difference bet? full 2 TG's.

DATE: / /
PAGE NO.:

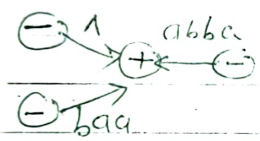




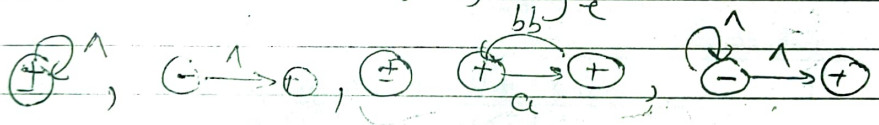
Diff. is def? for no. of states but as lang. acceptors they are equ.

Every FA is TG but not every TG satisfies def? of FA.

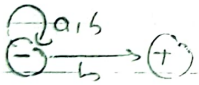
ex. ⊖ - TG accepts nothing, not even Λ., ∵ there is no final state.
⊕ accepts only Λ. Any other str can't have successful path to final state thro' labels of edges because there are no edges.



Accepts Λ, baa, abba
Anything read while in the + state cause a crash, ∵ there is no outgoing edge

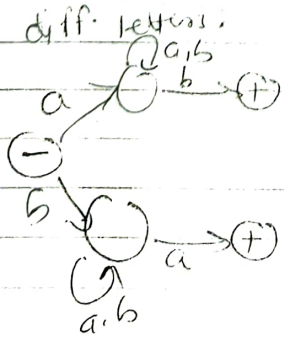


These TGs accept only Λ.

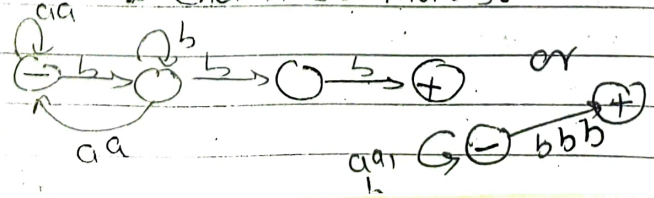


if b is last letter in i/p str, we go to +.
But we can also follow unsuccessful path.
for last b, be in ⊖ or for nonfinal b, enter ⊕.
Lang accepted by TG is all words ends in b.
RE $(a+b)^*b$, FA is:

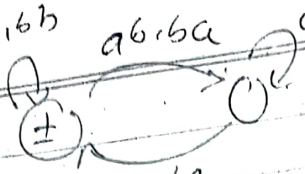
ex. TG accepts lang. of words that begin & end with diff. letters.



ex. TG accepts lang. of all words, a occurs in even clumps & end in 3 or more b's.



Pf: [illegible]

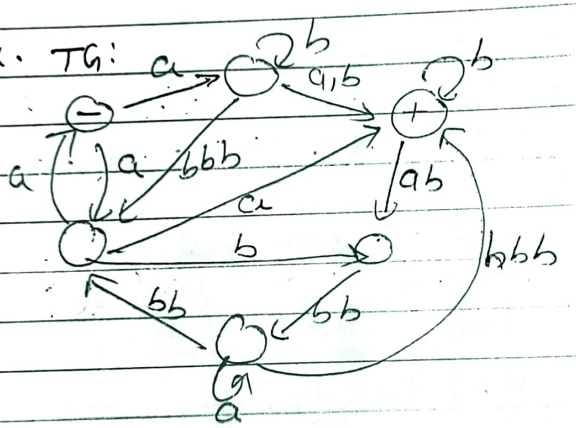


ex even-even

For lang. even-even, out of PI, TM, TG, TGs is most understandable

Practical problems with TGs - There are occassionally so many possible ways of grouping the letters of the i/p string that we must examine many possibilities before we know whether a given str is accepted or rejected.

ex. TG:



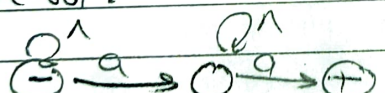
Is the word abbbabbbabbba accepted by this m/c ans - Yes, in 3 ways.

Λ edges - we have infinite No. of ways of grouping the letters of an i/p str. ex. i/p str ab is factored as (a)(b), (a)(Λ)(b), (a)(Λ)(Λ)(b), ...

This allows infinite ~~str~~ things to happen in seemingly finite sol^ns.

Λ - infinite grouping

TG:



word aa can be accepted by infinitely many diff. paths.

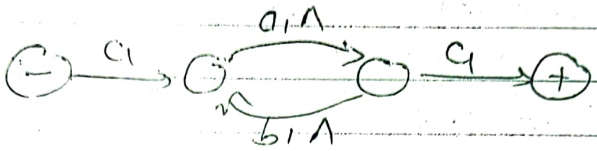
Λ-edges makes life difficult, have no utility. If we take any TG with Λ loops & take away these loops, resultant picture is still a TG & accepts same set of i/p str. Then why we allow Λ-loops ?

① def^n should be simple & universal sounding (any edges, anywhere, any labels)

DATE: / /
PAGE NO.:

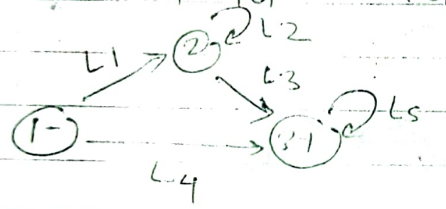
② Λ loops are not the only way of getting an infinite path all of a finite i/p str.

ex. It is obvious how to eliminate full Λ circuit:

But 

if Λ is erased, resultant lang ② is changed.

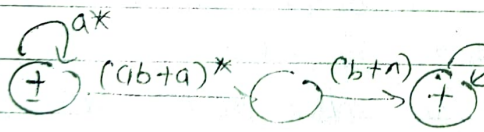
Generalized TG - GTG - defn



① finite set of states, start & final

② Σ

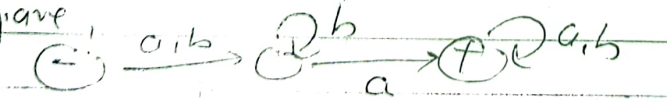
③ Directed edges connecting some pairs of states, each labeled with a RE.

ex. 

accepts all str. w/o (double) b.

word b takes Λ-edge from start to middle.

There is no difference bet? Kleene star closure for RE & a loop in our previous TG's or FA's.

compare 

In 1st picture, we may loop in the middle state as many times as we want or go straight to 3rd state. To not loop corresponds to taking Λ choice from b^* in 2nd ex.

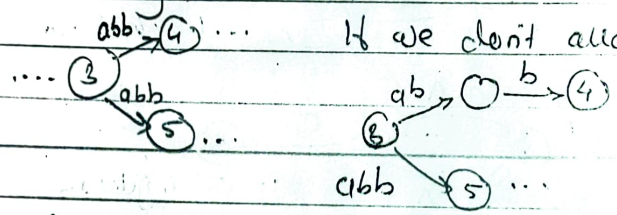
DATE: / /
PAGE NO.:

Nondeterminism – disturbing fact with TG.

Just as the * & + in RE represent a potential multiplicity of choices, so does the possible multiplicity of paths to be selected from a TG.

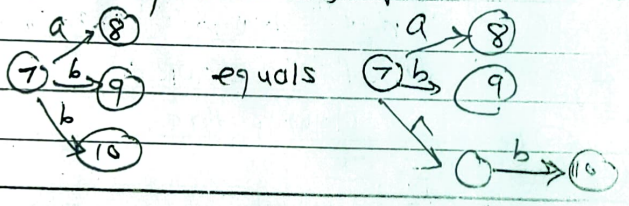
In GTG, the choice are both static & dynamic. We often have a range of choices of edges, each labeled with an infinite lang. alternatives. The No. of ways of going from state 1 to state 4 might be infinite.

ex. showing need for choice:



If we don't allow this, one can do:

Even if we restrict labels to strings of only 1 letter or Λ, we may indirectly permit these equi. situations.



In TG, a particular str. of i/p may trace thro. the m/c on diff. paths, depending on our choice of grouping.

ex. abb can go from state 3 to 4 or 5.

The ultimate path thro. the m/c is not determined by the i/p alone. ∴ it is nondeterministic.

DATE: / /
PAGE NO.:

Ch 7 Kleene's theorem.

Th^m – Any lang that can be defined by RE or FA or TG can be defined by all 3 methods.

Proof – part 1 – Every lang. that can be defined by a FA can also be defined by a TG.

Part 2 – —"— TG —"— RE

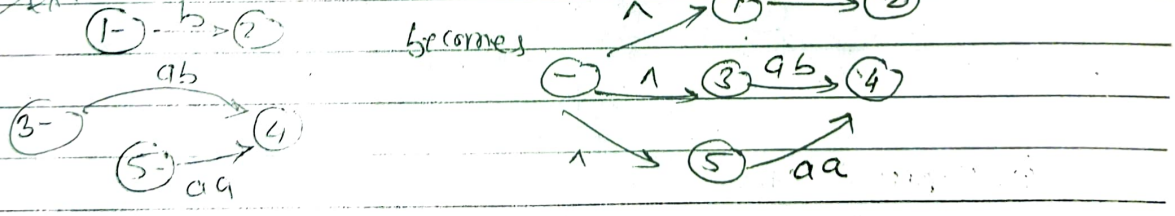
3 – —"— RE —"— FA.

Proof of part 1 -- Every FA is itself already a TG.
∴ every lang. that has been def^nd by a FA has already been def^nd by a TG.

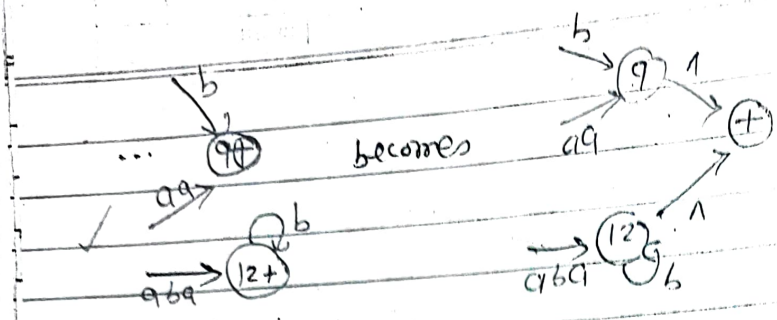
Part 2 – Turning TG into RE.

(1) TG T may have many start states. Simplify T so that it has only 1 start state that has no incoming edges.
Introduce a new state with – sign, connect to all previous start states by edges labeled with Λ. Drop – from prev. st states.
This does not make any changes in prev. m/c logically.

Ex.



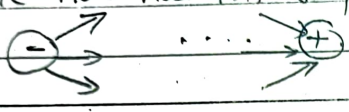
(2) T is modified to have a unique unexitable final state w/o changing the lang. it accepts.
If T had no final st., then it has no lang. & we produce no RE other than null or empty expn ∅.
If T has several final states, unfinal them & introduce a new unique final state labeled with a + sign. Draw Λ edges from prev. fin. st to new one. When i/p str. runs out of letters & it is in old final st., it can now take a free Λ edge over to new fin. st.



Now fin. st. has no outgoing edge.

We require that the unique final st. be a diff state from unique start state. If an old state used to have $\pm$, then both signs are removed from it to newly created states.

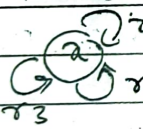
m/c now has fall shape:



where there are no other $-$ or $+$ states.

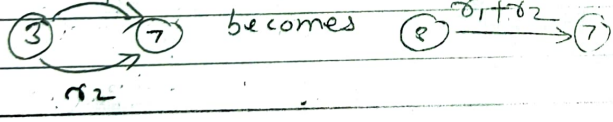
(3) now we build piece by piece the RE that defines same lang. as T: Change T into GTG.

Let T has state q (not $-$ or $+$) having many loops.

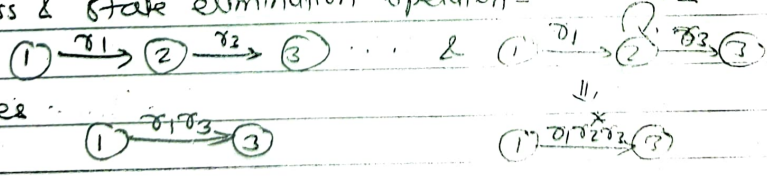


where r_1, r_2, r_3 are RE or strings.

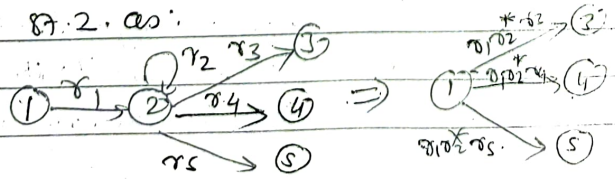
Replace 3 loops by 1 loop labeled with RE $(r_1 + r_2 + r_3)^*$



(4) Bypass & state elimination operation -



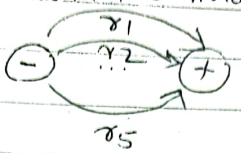
If 87.1 is connected to 87.2 which is connected to 87 3, 4 & 5. then we can eliminate edge from 1 to 2 & bypass 87.2. as:



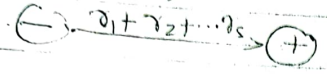
DATE: / /
PAGE NO.:

Every state that leads into S7.2 can be made to bypass S1.2, by adding edges.
We can repeat this process until we have eliminated all states from T except unique start & final S7.

We get a picture:



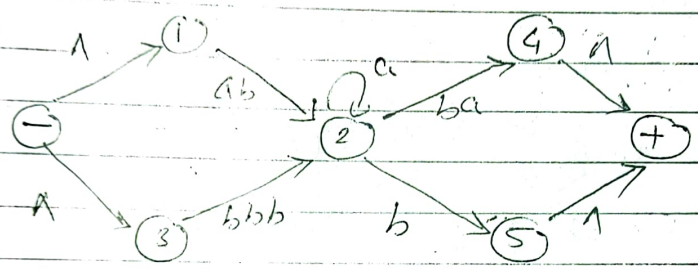
combine this,



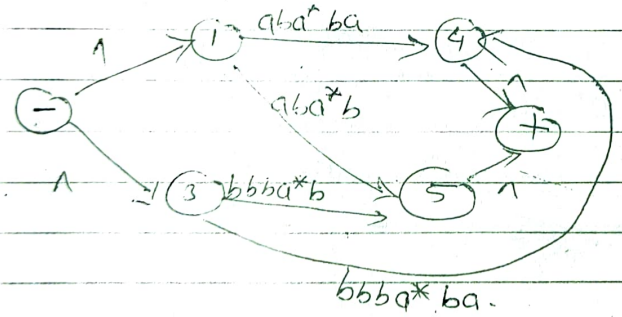
Resultant RE is defines same lang. T did originally.

pre

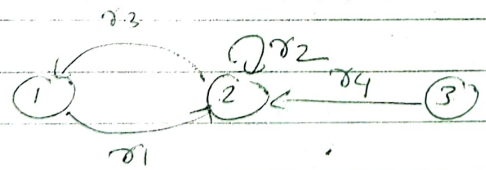
ex.



Bypass state 2,

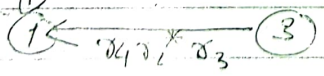


1-2, 2-4 ⇒ 1-4
1-2, 2-5 ⇒ 1-5
3-2, 2-4 ⇒ 3-4
3-2, 2-5 ⇒ 3-5.

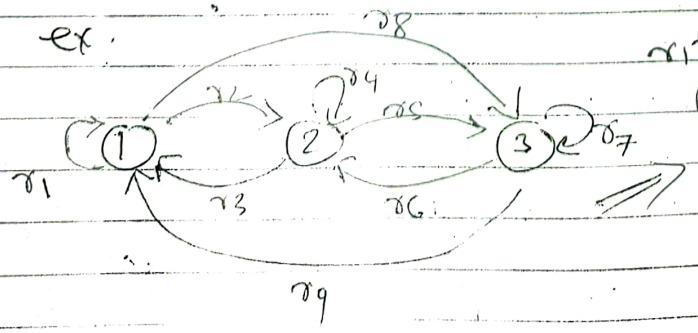


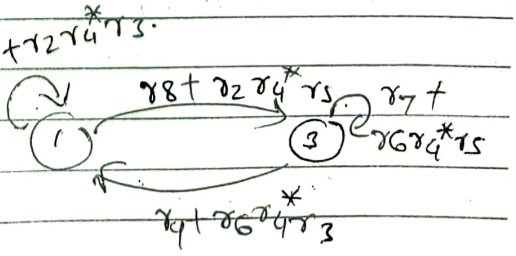
ex.

3-2, 2-1 ⇒ 3-1
1→2, 2-1→1-1



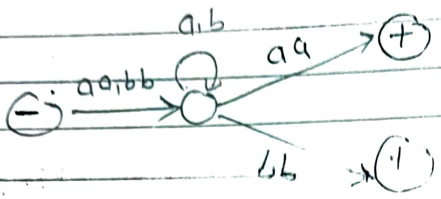
ex.



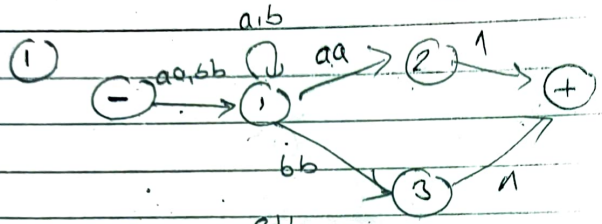


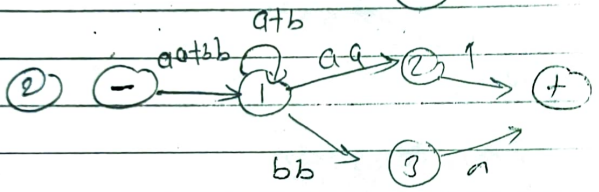
Algo. terminates in finite No. of steps, because T has only finitely many states, & 1 state is eliminated with each iteration of bypass procedure.
It works on all TGs.
Thus there is a RE for each TG.

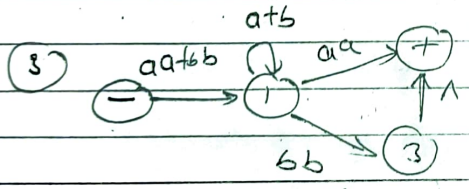
ex.

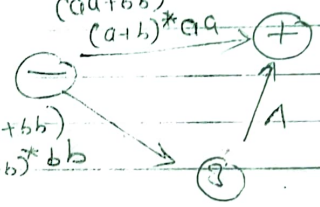


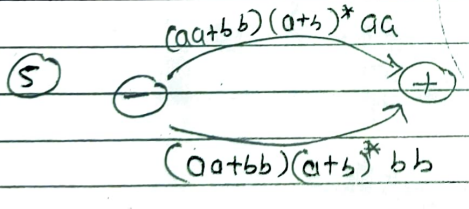
words that begin & end with double letters (having at least length 4)





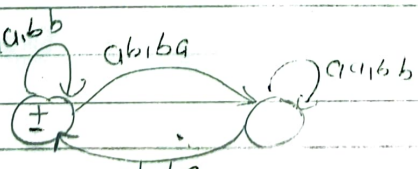


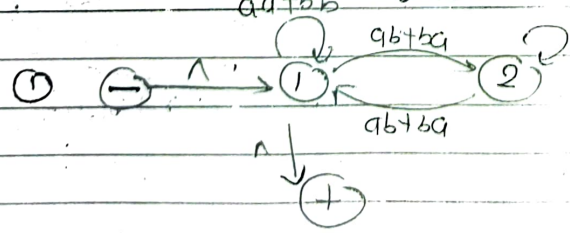


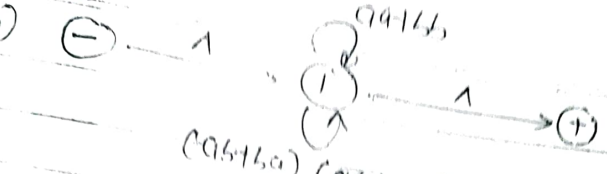


RE: $(aa+bb)(a+b)^*aa + (aa+bb)(a+b)^*bb$
$= (aa+bb)(a+b)^*(aa+bb)$
by distributive law

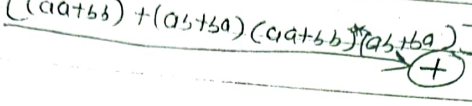
ex Even-Even.



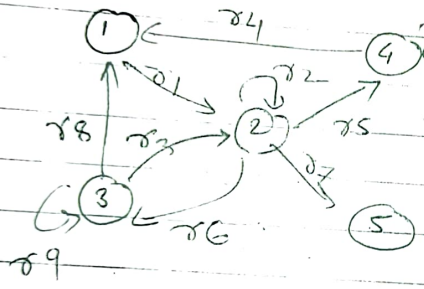


(2) 

(3) 

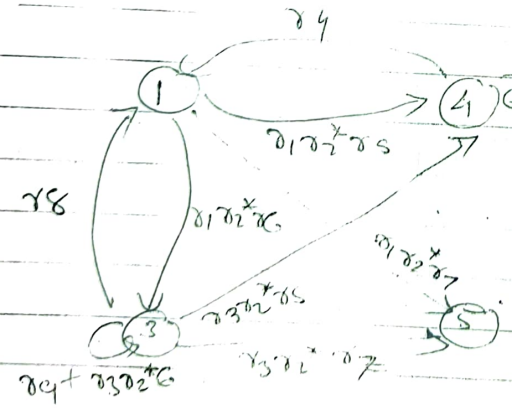
(4) 

ex. fragment TG



bypass 2
incoming edges from 1, 3
out 3, 4, 5.

from	To	labeled
1	3	$r_1 r_2^* r_6$
1	4	$r_1 r_2^* r_5$
1	5	$r_1 r_2^* r_7$
3	3	$r_3 r_2^* r_6$
3	4	$r_3 r_2^* r_5$
3	5	$r_3 r_2^* r_7$



old trip thro. states
1-2-4-4-1-2-3-3-2-5
will be made thro'.
1-4-4-1-3-3-5 whose
concatenation of RE's is
exactly same as before.

Algorithm that proves that all TGs can be turned into RE that define exact same lang.
Step 1: Create a unique, unenterable - state & a unique, unleavable + st.
2: one by one, in any order, bypass & eliminate all the non - or + states in TG.

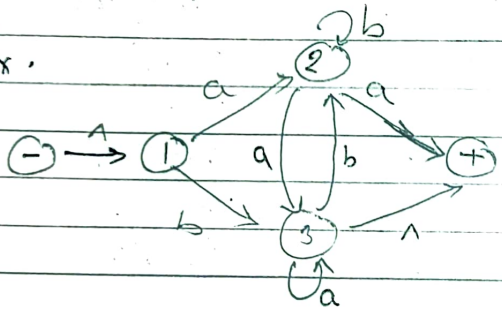
A state is bypassed by connecting each incoming edge with each outgoing edge. Label of each resultant edge is the concatenation of the label on the incoming edge with the label on the loop edge if any & label on outgoing edge.

3: When 2 states are joined by more than 1 edge going in same dir^n, unify them by adding their labels.

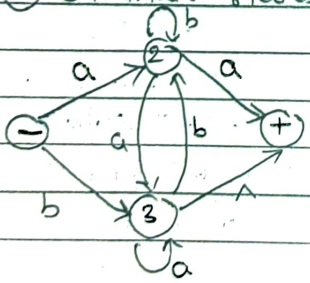
4: Finally, 1 edge is left from − to +, its label is RE that generates same lang. as recognized by original m/c.

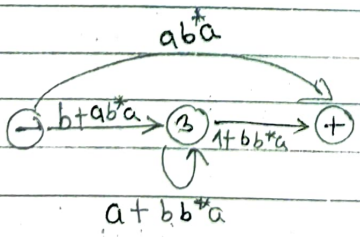
When we finish st. 3, there may be no path left from − to +, i.e. original m/c accepted null lang. ϕ, whose RE has no symbols.

Ex.



(1) eliminate states 1, 2, 3.





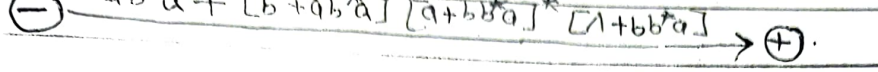
State 2 -

incoming edges from −, 2, 3

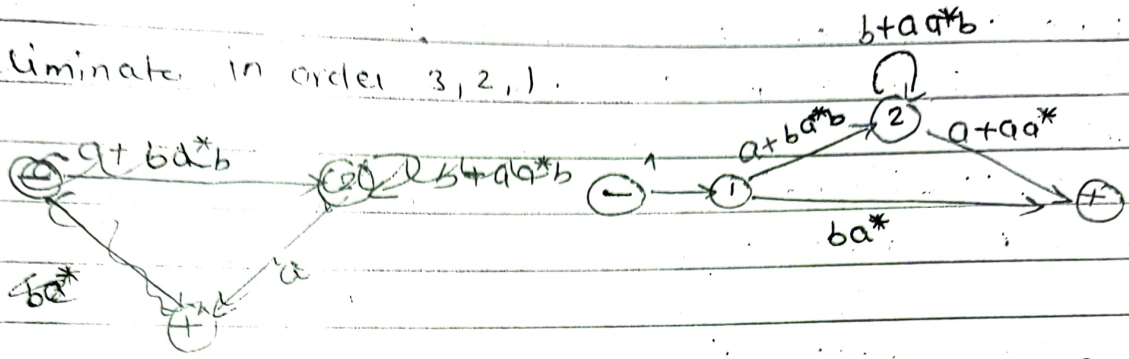
outgoing " to 2, 3, +

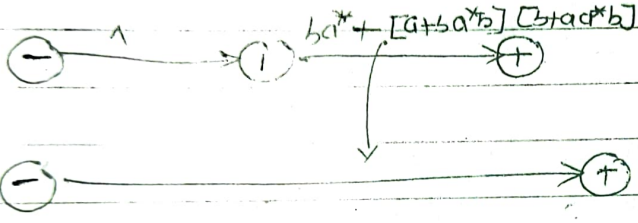
from	to	from	to	edge
−	2	−	3	ab^*a
−	3	−	+	ab^*a
−	+	3	3	bb^*a
2	2	3	+	bb^*a
2	3			
2	+	− 2, 2 3	⇒	− 3
3	2	− 2, 2 +	⇒	− +
3	3	3 2, 2 3	⇒	3 3
3	+	3 2, 2 +	⇒	3 +

DATE: / /
PAGE NO.:



(2) Eliminate in order 3, 2, 1.





If we had not seen how they were derived, we might have no clue as to whether these 2 RE's define the same lang.

Converting RE's into FAs: (Proof of part 3)

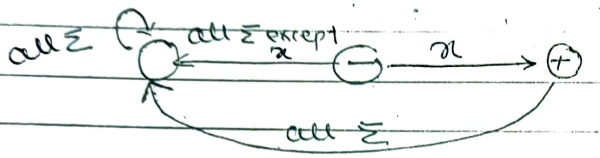
Rule 1: There is an FA that accepts any particular letter of the alphabet. FA accepts only the word Λ.

Rule 2: If there is an FA_1 accepts lang. defined by RE r_1 & FA_2 accepts lang. by r_2, then there is FA_3 accept lang. by $(r_1 + r_2)$.

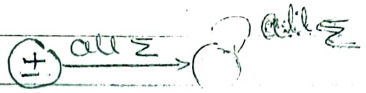
Rule 3: —"— by $r_1 r_2$, the product lang.

R.4: FA_1 accepts r, FA_2 accepts r^*.

Proof of Rule 1: If x is in Σ then FA



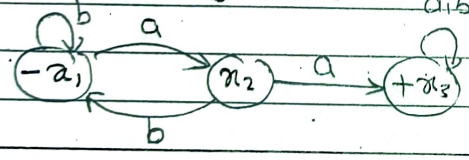
accepts only the word x.

FA accepts only Λ is 

It would be easier to design these m/c's as TG's, but it is important to keep them as FA's.

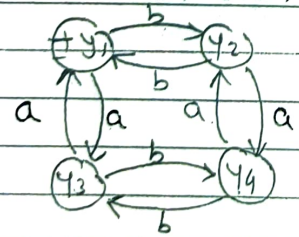
Proof of Rule 2: We show how to construct new m/c from old m/c's.

FA_1: words with aa



	a	b
$-x_1$	x_2	x_1
x_2	x_3	x_1
$+x_3$	x_3	x_3

FA_2: Even-Even



	a	b
$\pm y_1$	y_3	y_2
y_2	y_4	y_1
y_3	y_1	y_4
y_4	y_2	y_3

The new m/c accepts union of these 2 lang. i.e. words either have an aa or are in Even-Even. New m/c will simultaneously keep track of where the i/p would be if it were running on FA_1 alone & where

On m/c 3, we follow both the path the i/p would make on FA1 & on FA2 at same time. (By keeping track of both paths, we know when i/p str ends, whether or not it has reached a final st. on either m/c)

DATE: / /
PAGE NO.:

the i/p would be if it were running on FA2 alone.

Start state z_1 — combine x_1 & y_1 st. states of 2 m/c. All z states in m/c 3 keep track of which x state the string would be in & which y state the str would be in.

$\pm z_1$ means either x_1 or y_1. m/c 1 m/c 2 ∵ y_1 is final st. for FA2, z_1 is also fin.

$z_1 \to$ i/p a $\to$ (x_1 or y_3) $\to$ call it z_2

$z_1 \to$ i/p b $\to$ x_1 or y_2 $\to$ call it z_3

(Refer transition tables of FA1 & FA2)

$z_2 \to a \to x_3$ or $y_1 \to z_4 + (\text{fin. st})$

$z_2 \to b \to x_1$ or $y_4 \to z_5$.

$z_3 \to a \to x_2, y_4 \to z_6$

$z_3 \to b \to x_1, y_1 \to z_1$ (no need to create new states)

$z_4 \to a \to x_3, y_3 \to +z_7$

$z_4 \to b \to x_3, y_2 \to +z_8$ (both final, ∵ x_3 is final in FA1)

$z_5 \to a \to x_2, y_2 \to z_9$

$z_5 \to b \to x_1, y_3 \to z_{10}$

$z_6 \to a \to x_3, y_2 \to +z_8$

$z_6 \to b \to x_1, y_3 \to z_{10}$

$z_7 \to a \to x_3, y_1 \to +z_4$

$z_7 \to b \to x_3, y_4 \to +z_{11}$

$z_8 \to a \to x_3, y_4 \to +z_{11}$

$z_8 \to b \to x_3, y_1 \to +z_4$

$z_9 \to a \to x_3, y_4 \to +z_{11}$

$z_9 \to b \to x_1, y_1 \to +z_1$

$z_{10} \to a \to x_2, y_1 \to +z_{12}$

$z_{10} \to b \to x_1, y_4 \to z_5$

$z_{11} \to a \to x_3, y_2 \to +z_8$

$z_{11} \to b \to x_3, y_3 \to +z_7$

$z_{12} \to a \to x_2, y_3 \to +z_7$

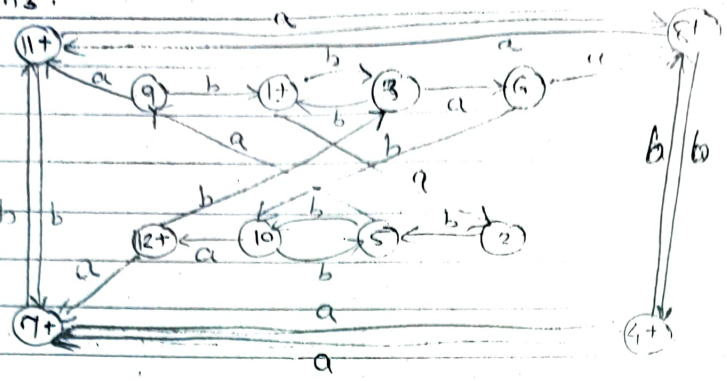
$z_{12} \to b \to x_1, y_2 \to z_3$

Transition Table

	a	b
$\pm z_1$	z_2	z_3
z_2	z_4	z_5
z_3	z_6	z_1
$+z_4$	z_7	z_8
z_5	z_9	z_{10}
z_6	z_8	z_{10}
$+z_7$	z_4	z_{11}
$+z_8$	z_{11}	z_4
z_9	z_{11}	z_1
z_{10}	z_{12}	z_5
$+z_{11}$	z_8	z_7
$+z_{12}$	z_7	z_3

DATE: / /
PAGE NO.:

FA3 :



Algorithm: Starting with 2 mlc's FA_1, with states $x_1, x_2, x_3, \cdots$ & FA_2 with st. $y_1, y_2, \cdots$, build a new mlc or FA_3 with states $z_1, z_2, \cdots$, where each z is of the form "x something or y something". The combination st. x part or y part is the - state of FA_3. If either x part or y part is a final state, then corresponding z is a final state. To go from one z to another by reading a letter from the i/p string, we see what happens to the x part & y part & go to new z accordingly.

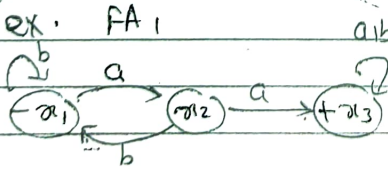
formula:

$z_{new\ after\ letter\ p} = [x_{new\ after\ p\ on\ FA_1}$ or $y_{new\ after\ p\ on\ FA_2}]$

Because there are only finitely many x's & y's, there can be only finitely many possible z's. Not all of them will necessarily be used in FA_3, if no i/p string beginning at - can get to them.

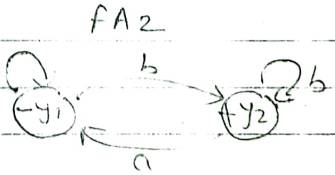
Thus we can build a mlc that can accept sum of 2 RE's if we already know mlc's to accept each of the component RE's separately.

ex. FA_1



words with double a

FA_2



words end with b.

Mlc that accepts union of these 2 lang:

$-z_1 = x_1$ or y_1

$z_1 \rightarrow a \rightarrow x_2, *y_1, z_2$

$b \rightarrow x_1, +y_2, +z_3$.