* Hierarchy:

1) Top Down:

- Start with a general model, and add restrictions to get the other members.

*2) Bottom Up:

- Start with the simplest models, and add more functionality to get a general model.

---

* Finite Automaton: (Denoted $M, M_1, \ldots$) (Cannot store inputs) (Read input character by character)

---

* • Alphabet (Σ) = Set of Symbols used to define a language.

• Language (L) = Set of strings of a given alphabet.

⇒ Every such machine will have a finite
* number of states, denoted by (Q).

- $q_0 \in Q$ is the start state
- A set (F) $\subseteq Q$ are called final/accepted states

⇒ State transition function denoted by δ.

- Domain of $\delta = Q \times \Sigma$ ← Input Symbols
  (↑ Current State)
- Range of $\delta = Q$ ← Next State

- Computing on a finite automata,

$$M = (Q, \Sigma, q_0, \delta, F)$$

$\Rightarrow$ M starts at q_0

$\Rightarrow$ Input are symbols $\in \Sigma$

$\Rightarrow$ δ gives us next state from current state and input.

$\Rightarrow$ On reaching a state in F, we have found our final state.

* $\Rightarrow$ A machine is said to <u>accept an input</u> if it would lead the machine to reach one of its final states(F).

* • <u>Language accepted by a machine:</u>

$$L(M) = \{w \mid M \text{ accepts } w\}$$

(M: Machine; w: String (Input))

———x———

<u>Note</u>: Computation is a sequence of state transitions taken by the machine for an input w.

———x———

* Regular Languages:

- Languages that are accepted by some finite automata are called regular languages.

———x———

* Note: $L1 \cup L2 = \{w \mid w \text{ in } L1 \text{ or } w \text{ in } L2\}$

* $L1 \circ L2 = \{w_1 w_2 \mid w_1 \text{ in } L1 \text{ and } w_2 \text{ in } L2\}$

Concatenation

Star $L^* = \{w_1 w_2 \ldots w_k \mid k >= 0, w_i \text{ in } L \text{ for } 1 <= i <= k\}$

We get ε for $k=0$.

—x—

* Note: ε = Empty String

—x—

Closure of Languages: (Regular)

1) Union:

(This also works if the alphabet of L1 & L2 are different, complete proof)

- Let L1 & L2 be two regular languages with finite automatons M_1 & M_2 that recognize the languages respectively.

⇒ For $L = L1 \cup L2$ to be regular, we need to show there exists some M which L is accepted by.

- We can define our machine M as,

$\delta((q, r), a) \longrightarrow (q', r')$

where $q' = \delta_1(q, a)$ } For M1

$r' = \delta_2(r, a)$ } For M2

⇒ M's states are tuples $(p, s) \in Q$

- Accepted states of M are → p accepted in M1 or s accepted in M2.

F(M)

∴ We have formulated our finite automata M. So the union is also a regular language.

—x—

*2) Concatenation:

* $L = L1 \circ L2$

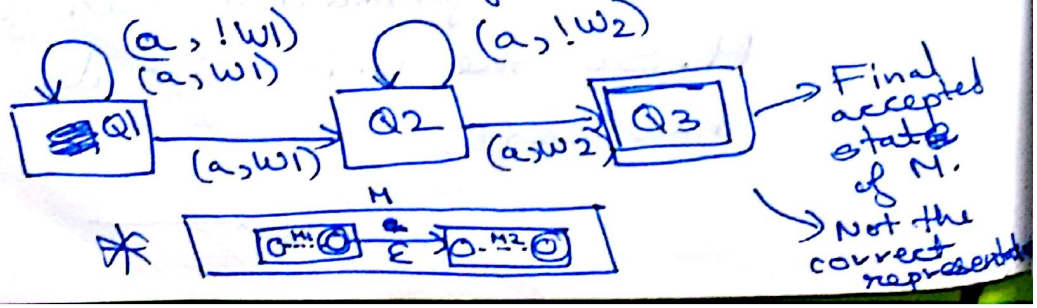
⇒ Given an input W, if we can ~~know~~ & know that $W = W1 \circ W2$ with W1 in L1 & W2 in L2, then we are done.

- If we know where W1 ends & W2 begins in W, then our machine M could run M1 till W1 ends & then run M2 till W2 ends & if both are true, then our machine M is defined.

* ⇒ We use non-determinism to solve this.
  - → From a state, on a given input we can go to 0 or more resultant states.
  - → If any of these states possible are final states then we consider the input to be accepted.

⇒ Now using non-determinism, our machine M will keep checking for M1 on W, and if we get an accepted state on M1 at some part of W, we then go to checking with M2 as well as continuing check with M1.

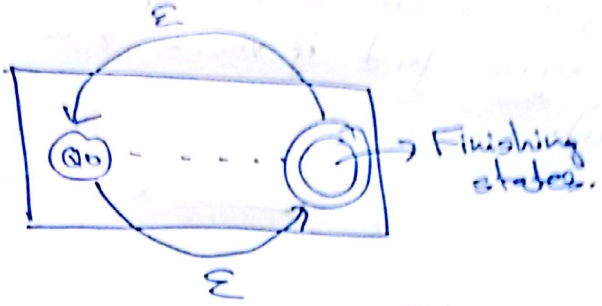
Since this part of the string might also be a prefix of another W1.



*3) Star:

$L \longrightarrow L^*$

- This can be thought of a similar to concatenation.
- With the help of non-determinism,



———x———

*NFA is equivalent to DFA: (Example at the back of the book)

(NFA → Nondeterministic, DFA → Deterministic)

- Suppose our NFA N, $\delta(q,a) = \{q1, q2, \ldots\}$

⇒ Now let us consider a DFA M with the states to be the powerset of the states of N.

$$\therefore \delta_M(\langle S \rangle, a) = \bigcup_{s \text{ in } S} \delta_N(s,a)$$

(δ_M: Deterministic; $\bigcup$: Union; δ_N: Nondeterministic)

- So now, if our set $\langle S \rangle$ consists of a goal state, we can say, our automata accepted the input. (M starts at $\{q_0\}$)

———x———

*Note: Suppose the input ε is given to the DFA defined above, then what would happen? (Textbook)

⇒ ~~It would be similar to the transition from one M1 to M2 as defined previously.~~ (we could add another ~~set~~ number to our set which can determine if we are in M1 or M2 and proceed)

———x———

*Redundency of States:

- If a state has no incoming states, then that state is redundant.

⇒ All states which only have redundant incoming states are also redundant.

(Think of it as an iterative process, find all redundant, then again find if more have been added and so on).

—— x ——

*Regular Expressions:

- $R = a$
- $R = \varepsilon$
- $R = \phi$
- $R = R_1 \cup R_2$
- $R = R_1 \circ R_2$
- $R = (R_1)^*$

- Any of these R's are defined as regular expressions where R_1 & R_2 are regular expressions.

⇒ (Σ^*) matches any length of any of over alphabet Σ. (Matches any string made from characters of Σ)

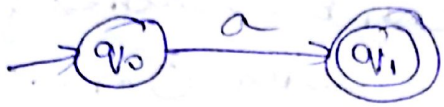
—— x ——

Note:

(Every regular language has an NFA which can recognize it, so it has a DFA which can recognize it).

* Regular Expression → NFA :

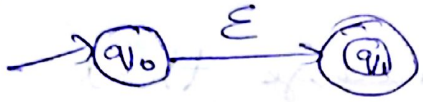
- We can show that a regular expression can be described by an NFA, so a regular expression corresponds to a regular language.



} For $R = a$

(For union, concatenation and star, we proved earlier.)

} For $R = \phi$



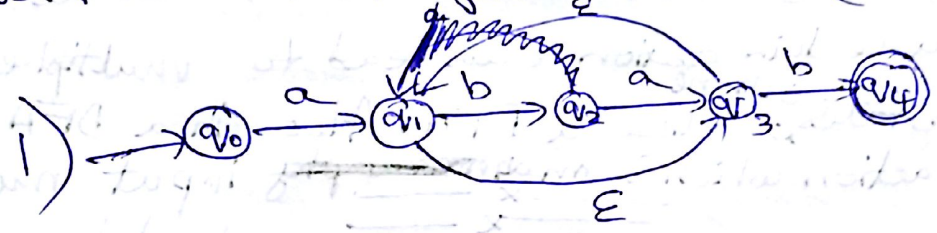
} For $R = \varepsilon$

⇒ Smaller RE's are ones which require lesser no. of operations to build them. (Using the given 6 operations/definitions).

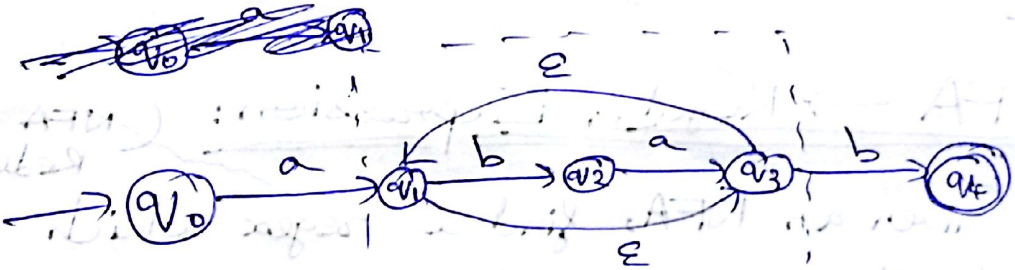
---

* ex: Draw NFA for $R = a(ba)^*b$

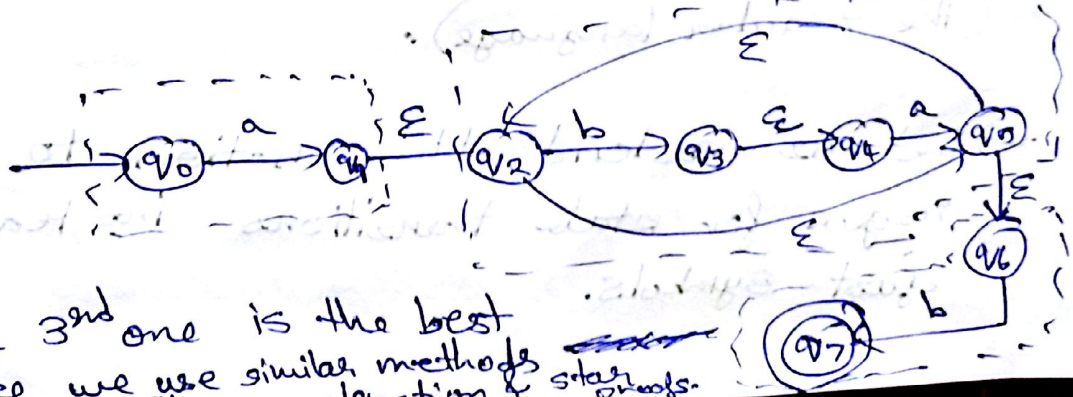
1)



2)



3)



- The 3rd one is the best since we use similar methods like concatenation & star proofs.

Note: In NFA to DFA conversion, suppose we allow ε inputs as well, then let,

$E(R) = \{q \mid q$ can be reached from R by using 0 or more $\varepsilon\}$

- Now $\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a))$ for all $r \in R\}$

$\Rightarrow$ Also one more change that needs to be done is, to consider all $E(q_0)$ to be the start states instead of just q_0.

Note: An NFA is where a certain state on a certain action can lead to multiple states, or none, unlike a DFA. Also in a DFA, every action which is on an input must lead to a state change.

NFA $\longrightarrow$ Regular Expression: (NFA $\rightarrow$ GNFA) Reduce GNFA to get regex

- Given an NFA, find a regex which corresponds to the set of strings accepted by the NFA (the regular language).

$\Rightarrow$ Let us extend this notion to using regex for state transitions instead of just symbols.

• GNFA → Generalized NFA.

• In a GNFA, there are no incoming transitions to the start state & no outgoing transitions from the end state. A single start & accept state. Transitions can be on regular expressions.

* to the start state & no outgoing transitions from the end state. A single start & accept state. Transitions can be on regular expressions.

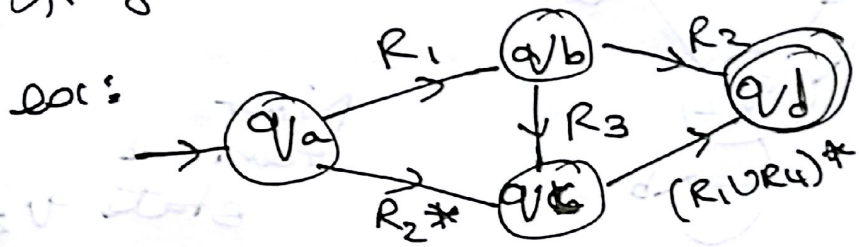
—x—

1) ⇒ To convert an NFA to a GNFA, add
* a new start state with arcs on ε to all start states, and add a new end state with arcs on ε from all end states.

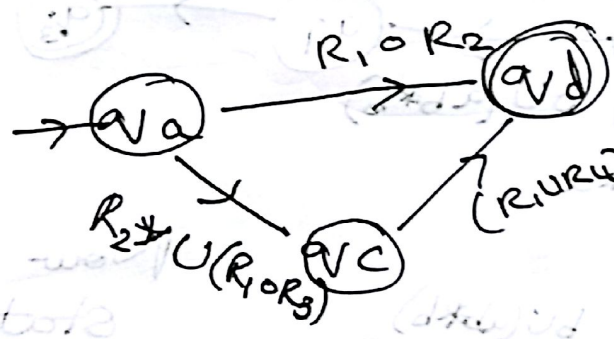
Now GNFA → RE,

⇒ We then try to remove states
** from our NFA, one by one, to get our final regex.

Eg:



⇓ Remove q_b



• Suppose the state we are removing has a self loop, then the regex from $q_i \to q_j$ after removing q_x is,

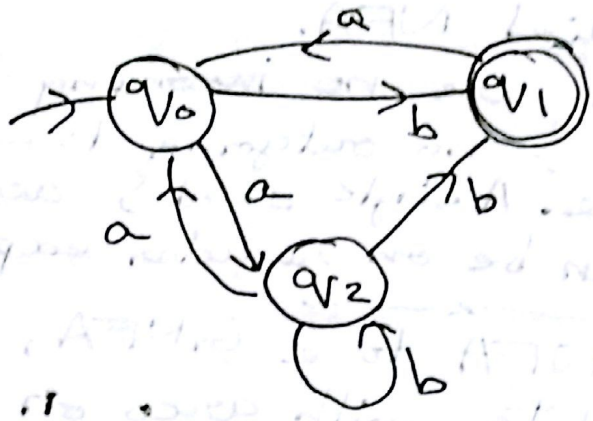
$R(q_i, q_j) \circ R(q_x, q_x)* \circ R(q_x, q_j)$

• In this manner we can reduce our NFA to get a final regex corresponding to it. (When only 2 states are left in our machine)

∴ We can convert an NFA to its corresponding regex. (Example in next page)

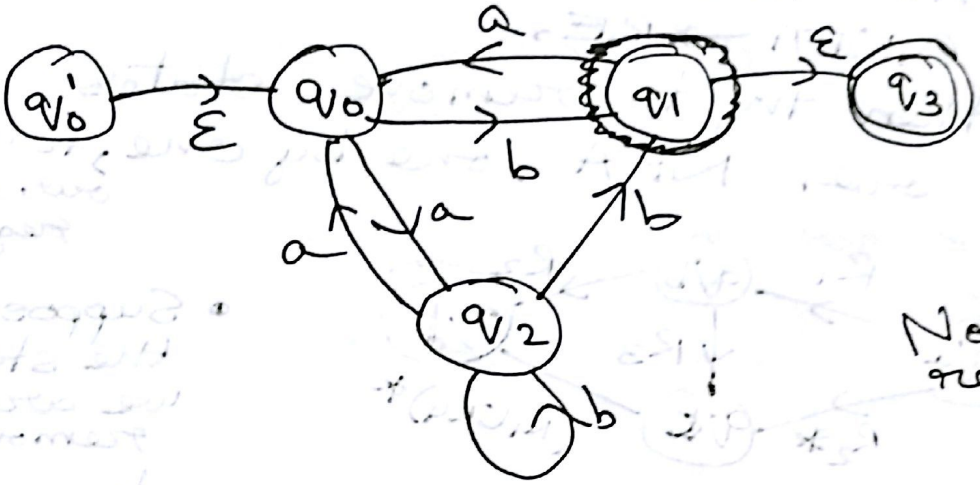
—x—

ex:

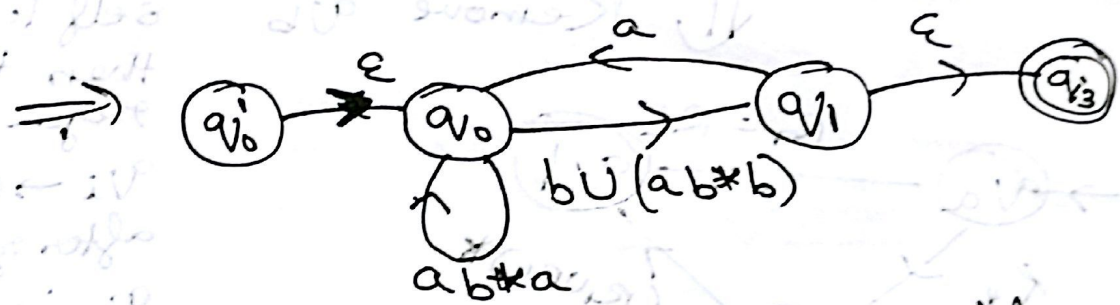


(This is not a GNFA) So we make it a GNFA.

- Now we try remove node q_2.
After we get a GNFA.

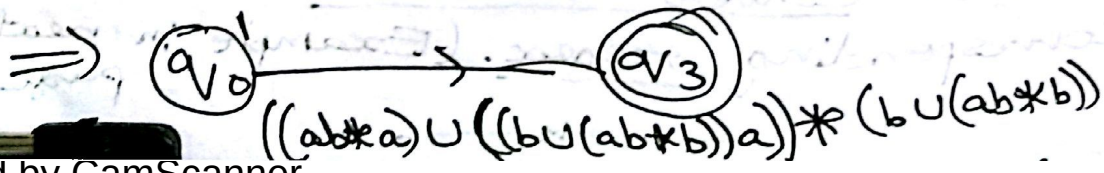


Now remove state q_2.



Now remove State q_1

Now remove state q_0.



* Non Regular Languages: (Generally ∞ states)

ex: $0^n 1^n \mid n \geq 0$ → This expression does not have a DFA which accepts the expression.

ex: ww^R → Palindrome for w in $\{0,1\}^*$

—— x ——

* Pumping Lemma:

⇒ Given a regular language L, there exists p = Pumping length, such that all strings w in L of length ≥ p can be split as $w = x \circ y \circ z$ such that,

- $i \geq 0$, $xy^i z \in L$ } Belongs to L.
- $|y| > 0$ & $|xy| \leq p$ } Length of string y & length of $(x \circ y)$

—— x ——

* Note: If the no. of strings accepted by a machine is greater than no. of states, then some states repeat, so we can see that these cycles correspond to y, $w = xyz$.

—— x ——

⇒ We can let the pumping length to
* be ⇒ No. of states in DFA M (Accepting L)

(If a DFA has M states, without repetition of states, it can only accept strings ≤ M−1 length)

* • So if $|w| \geq p$, states repeat.

∴ These parts b/w where the state repeats can be treated as y.

* ⇒ We can see that we can pick x, y such that $|xy| \leq p$, since we
* will definitely get a repeat state on string of length = p or greater, so we can set x & y.

* ex: We can use pumping lemma to show
* that a language L is not regular.

ex: $L = \{0^n 1^n \mid n \geq 0\}$

⇒ Let us assume L is a regular language so it must satisfy the pumping lemma with pumping length p.

- Let us consider $w = 0^p 1^p \in L$, $|w| \geq p$, we will show that we cannot divide w into $w = xoyoz$.

⇒ If y only has zeros, $xyyz$ will have more no. of zeros than 1's.
Similarly y cannot have only 1's.

⇒ So y contains both 0's & 1's, but then on repetition ⇒ $xyyz$, all 0's & all 1's won't be on two sides of the string.

∴ We cannot pick a y, $|y| > 0$ such that $xy^iz \in L$.

∴ Our assumption was false, L is not a regular language.

———x———

** ex: If L = Strings with equal no of 0's & 1's

⇒ Proof as above; but if $y = 01$ etc then $xyyx \in L$, but we can find a $w = 0^p 1^p$, where $|w| \geq p$ but for any y, $xyyz \in L$ but $|xy| > p$
⇒ Min $y = 01$, but $0^{p-1} \cdot 01 \cdot 1^{p-1} = w$
$= xyz$

But $xyyz = 0^{p-1} \cdot 01 \cdot 01 \cdot 1^{p-1}$

$\Rightarrow |xy^i z| = p+1$, So we cannot pick any y for this w.

$\therefore$ L cannot be a regular language

———x———

* Grammar:

- Grammar specifies how to create strings of the given language.

$\Rightarrow$ Grammar $\longrightarrow$ 4 tuple (Variables, Terminals, Productions, Start Variables) (G_1).

* exc: $V = \{A\}$
$T = \{0, 1\}$
$P = \{A \rightarrow 0A1, A \rightarrow \varepsilon\}$
$S = A$

Strings following grammar G_1 are strings of language generated by $G_1 = L(G_1)$

Derivation: exc: $A \longrightarrow 0A1 \longrightarrow 00A11 \longrightarrow 0011$
(of a string $\in L(G_1)$)

$\Rightarrow$ We should completely eliminate all variables to get a string.

———x———

Note: These kinds of grammars are context free grammars, so the rules only have single variable on LHS. (Context Sensitive grammar supports this though, Natural Languages)

$\therefore$ Natural Languages are more complex compared to programming languages.

———x———

- A string x yields another string y if using some number of derivations we can get y from x.

* ex: $\Rightarrow G_e = (V, T, P, S)$, Create a+axa

$V = S = \{EXPR\}$

$T = \{a, +, x, (,)\}$

$P = \{EXPR \Rightarrow EXPR + EXPR \mid EXPR \times EXPR \mid (EXPR) \mid a\}$

a) EXPR ⟶ EXPR + EXPR ⟶ a + EXPR x EXPR

a + a x a

or,

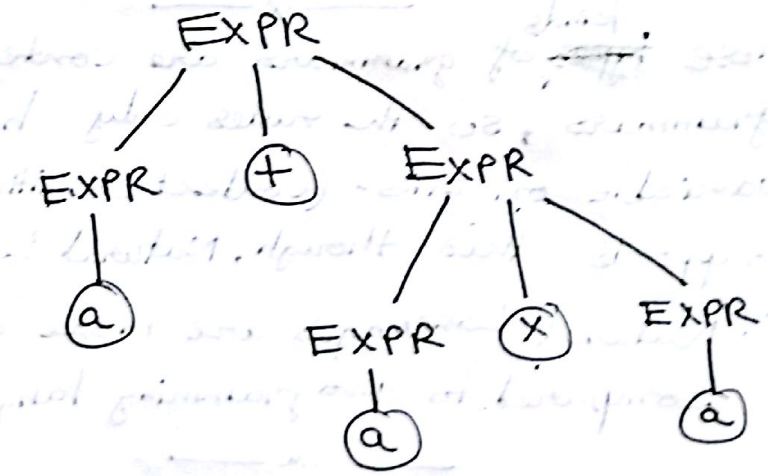
b) EXPR ⟶ EXPR x EXPR ⟶ EXPR + EXPR x EXPR

a + a x a

* Parse Trees:

- Representing the derivations using parse trees are more efficient.

ex: For the above example (a) derivation,



⟹ a + a x a.

⟹ Using example derivation (b) above,

we will get a different parse tree; but it also constitutes to a+a*a. This is called ambiguity.

⇒ Ambiguity: (Getting different parse trees for same expression)

* • To get rid of it (so we get only one parse tree for a certain string), we could follow,
  a) Leftmost derivation:
  → Always replace the leftmost variable with a production

This removes ambiguity for our previous example.
(If we use a proper set of productions which follow leftmost derivation)

• This need not always work, since there might be 2 or more valid derivations which can lead to the same result. (Mainly natural languages)

---

* Eg: Non-Ambiguous Expression grammar.

(Derive a+a*a)
(Only 1 possible derivation)
(So non-ambiguous)

$G_{re} = (V, T, P, S)$

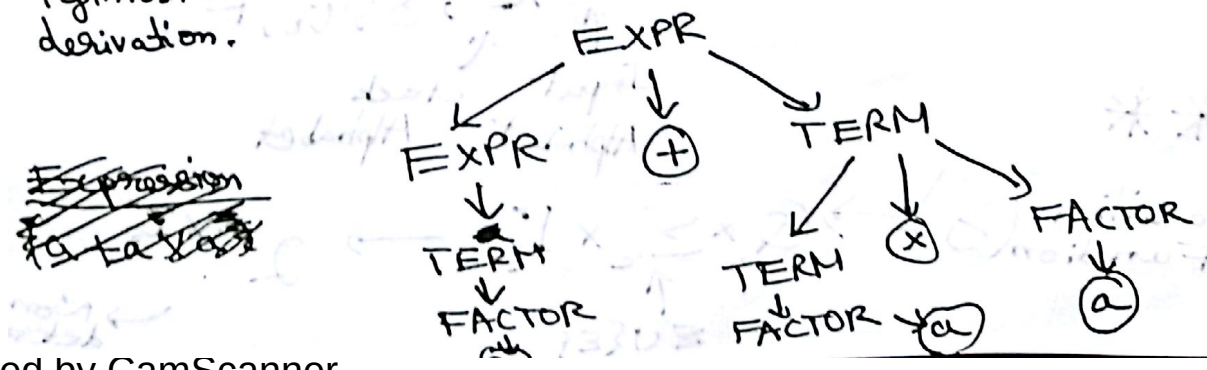
$V = \{EXPR, TERM, FACTOR\}$

$T = \{a, +, *, (,)\}$

$P = \{EXPR \rightarrow EXPR + TERM \mid TERM$
$TERM \rightarrow TERM * FACTOR \mid FACTOR$
$FACTOR \rightarrow (EXPR) \mid a\}$

This follows leftmost derivation.

$S = EXPR$



*Note: DFA's cannot recognize all languages generated using CFG's (Context Free Grammars) (Since CFG's also work for non regular languages)

—— x ——

*Pushdown Automata: (PDA)

- We allow the PDA to be non-deterministic
- Stack present → Allows for a lot more operations. (We can read the element on top of the stack)

⇒ These can recognize CFG's.

* ex: $L = \{0^n 1^n \mid n \geq 0\}$ → 3 states + stack

↳ The reason why we couldn't recognize this language using a DFA/NFA is because we could not store the number of 0's that have occurred till now.

↳ Here we can do that using the stack.

⇒ PDAs can behave like ~~programs~~ a computer (LIFO), since they have a stack & can maintain states like a DFA.

- PDA $M \rightarrow (Q, \Sigma, \Gamma, \delta, q_0, F)$ (F: Subset of Q (Accepted states))

(Σ: Input Alphabet, Γ: Stack Alphabet)

**

Transition Function $(\delta) \Rightarrow Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow 2^{Q \times \Gamma_\varepsilon}$

($\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$; → Non deterministic)

* ex: $L = \{a^i b^j c^k \mid i,j,k \geq 0$, either $i=j$ or $i=k\}$

- Here non determinism is useful.

⇒ Once we get either b or c,

✓ a	✓ a	x a
a	a	a
b	c	b
b	c	c

- In state 1, we use non-determinism to pop an 'a' when we encounter a 'b'. ~~or pop on 'a' when we encounter a 'c'.~~

- In state 0, we push 'a's onto the stack, state 3, we pop 'a's when we encounter 'b's. In state 4 we dont pop at all.

- State 2 is the failure state

⇒ State 1, 3 with stack empty in one of the non-deterministic elements (In states 1 & 3, we do popping & try to check if input has ended).

- In state 3 if stack is empty & the input is not over yet, we still have encountered an accept state.

⇒ Non-determinism → State 0 → State 3, State 4 on input b.

---

* ex: $L = \{ww^R$ for w in $\{0,1\}^*\}$ (PDA can do this) → Can be done using non determinism

N.D for determining middle of string.

Normal Forms:

* Chomsky Normal Form: (CNF)

- A CFG is in CNF if for every rule,

1) $\Rightarrow A \longrightarrow BC$

or

$A \longrightarrow a$ → Terminal symbol

- B, C cannot be the start symbol S.

2) $\Rightarrow$ Only S can derive ε.

---

* Conversion to CNF: (Grammar $G_1 \longrightarrow$ CNF G_2)

1) To ensure B, C are not start symbols, we assume a new start state S', $S' \longrightarrow S$, so now S can be used anywhere.

2) If we have an $A \longrightarrow \varepsilon$, we recursively ensure to ~~remove~~ keep rules $X \longrightarrow S_1 A S_2$ and add $X \longrightarrow S_1 S_2$. If X now goes to ε, apply same rule again & again.

3) Remove rules of form $A \longrightarrow B$ (unit rules), suppose $A \to B$, $B \to C$, ..., $X \to x$, then we remove all these rules & add $A \to x$. This will ensure that there are no single variable rules on the right hand side so CNF rule 1(a) is ensured.

4) Reduce long expressions. $A \to BCD$, this is simple $A \to BB'$, $B' \to CD$.

- So just introducing new states we can solve this problem.

(Also reduce $A \to aa$ to $A \to A'A'$, $A' \to a$)

5) $A \to BC$ & $A \to x$ — 2 terminates

---

* Describing PDAs

- Transitions b/w states are shown as,

$a, b \to c$ } On input a, Stack top $b \to c$

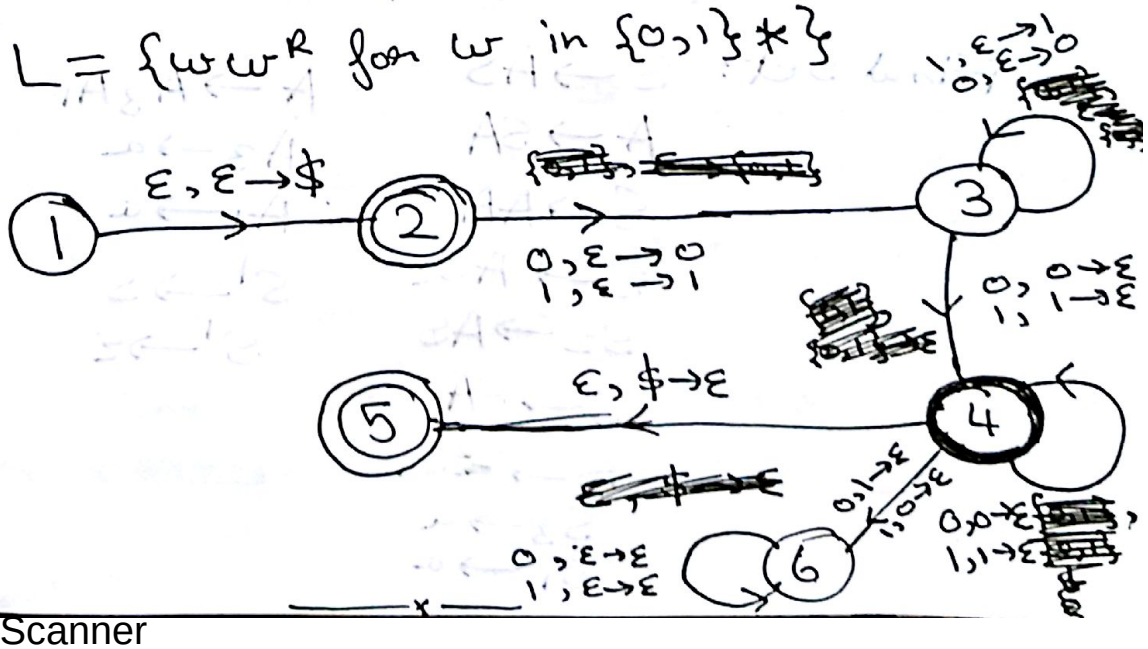
* • b, c can be ε: If b is ε, then it means that no matter what is present on the top of the stack, it is replaced with c.

$\Rightarrow$ If $c = \varepsilon$, then we pop b from the stack.

- We should always first put a '$' symbol etc on the stack (Denotes bottom of stack) and then remove this at the last.

$(\varepsilon, \varepsilon \to \$)$ & $(\varepsilon, \$ \to \varepsilon)$

** ex: * $L = \{w w^R$ for w in $\{0,1\}^*\}$



*

ex: Convert CFG to CNF (Chomsky Normal Form)

**

- $S \to AS$
  $S \to AAS$
  $A \to SA$
  $A \to aa$
  $S \to \varepsilon$

a) $\Rightarrow$ $S' \to S, S' \to \varepsilon$, So we don't repeat the start state is added to G!

b) $S \to A$
$S \to AA$
$A \to A$ } We can remove this
~~$A \to aa$~~ added &
$S \to \varepsilon$ is removed.

c) ~~$S \to A$~~
~~$S \to AA$~~
~~$A \to aa$~~
~~$S \to aa$~~
~~$S \to AA$~~
~~$A \to aa$~~

~~$A \to SA$~~ Add
$S \to aa$
$S \to SA$
Remove
$S \to A$
$A \to A$

d) $S \to aS_1$ } Add
$S_1 \to a$
$A \to aA_1$
$A_1 \to a$
~~$S \to AA$~~

$S \to AS_2$; $S_2 \to AS$

Final Set:
$S \to AS$
$A \to SA$
$S \to AA$
$S \to AS_2$
$S_2 \to AS$
$S \to SA$
$S \to S_3 S_1$
$S_3 \to a$
$S_1 \to a$
$A \to A_3 A_1$
$A_3 \to a$
$A_1 \to a$
$S' \to S$
$S' \to \varepsilon$

But we need to remove $S^1 \to S$ (cycle again)

- Find Set of rules:

$S \to AS$
$A \to SA$
$S \to AA$
$S \to AS_2$
$S_2 \to AS$
$S \to SA$
$S \to S_3S_1$
$S_3 \to a$
$S_1 \to a$
$A \to A_3A_1$
$A_3 \to a$
$A_1 \to a$
$S^1 \to \varepsilon$
$S^1 \to AS$
$S^1 \to AA$
$S^1 \to AS_2$
$S^1 \to SA$
$S^1 \to S_3S_1$

---

* Equivalence of ~~E~~F CFGs & PDAs:

- We know that every regular language can be derived from a CFG. (Inverse need not be true). (PDA can recognize all regular langu

*

* CFG → PDA (Example in slides) (Rules in slides)

- A CFG 'G' corresponds to a language L(G)

→ We show that a PDA M which accepts L(M) ~~is~~ exists such that L(M) = L(G).

- We allow transitions of the form $a, b \to X$

* where X is a string. We could do this on our PDA by introducing additional states & pushing one character at a time.

(Inserting string $a_1a_2a_3$ results in a_1 on top of stack).

⇒ We prove this with the help of non-determinism.

- We somehow treat our production rules as state transitions, so that we check if we can derive a string w, $w \in L(G)$ using M.

* • ~~If we have~~ We use the stack to
* store our current string along with variables. On reading terminal symbols on top, we pop it & consume input of our string. Since we use leftmost derivation, we can pop terminal symbols as soon as they occur, else if its not a terminal symbol, we push our production rule onto the stack. (Multiple production rules for same variable → nondeterminism)

Since it is leftmost derivation, we replace the variable on top of the stack first.

⇒ Our PDA will have 3
* states.
(Start, Repeat & Accept)

- q_0 has one transition,

$\varepsilon, \varepsilon \rightarrow S\$$

- q_{loop} has 3 transitions, (Push) (Repeat)

$\varepsilon, A \rightarrow w$ } Production rule
$a, a \rightarrow \varepsilon$ } Pop
When input is also empty → $\varepsilon, \$ \rightarrow \varepsilon$ } Final transition

- On the final transition we move from q_{loop} to q_{accept} with $\varepsilon, \$ \to \varepsilon$. If we are in q_{final}, then we accept the string.

---

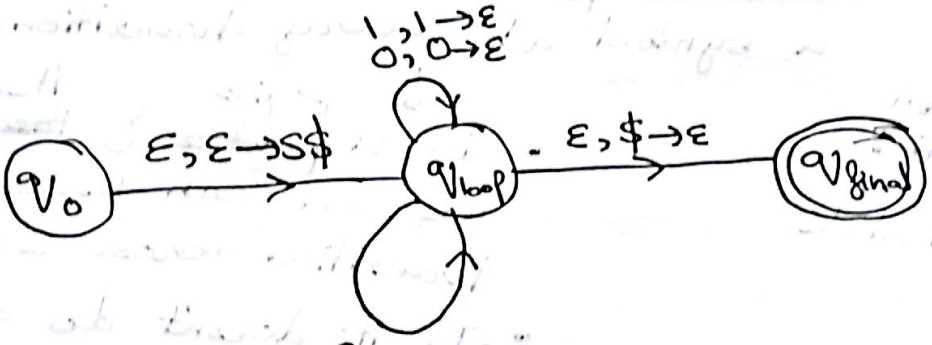
* EOC: Construct an equivalent PDA for the given CFG. *

$S \to TT \mid U$

$T \to 0T \mid T0 \mid 1$

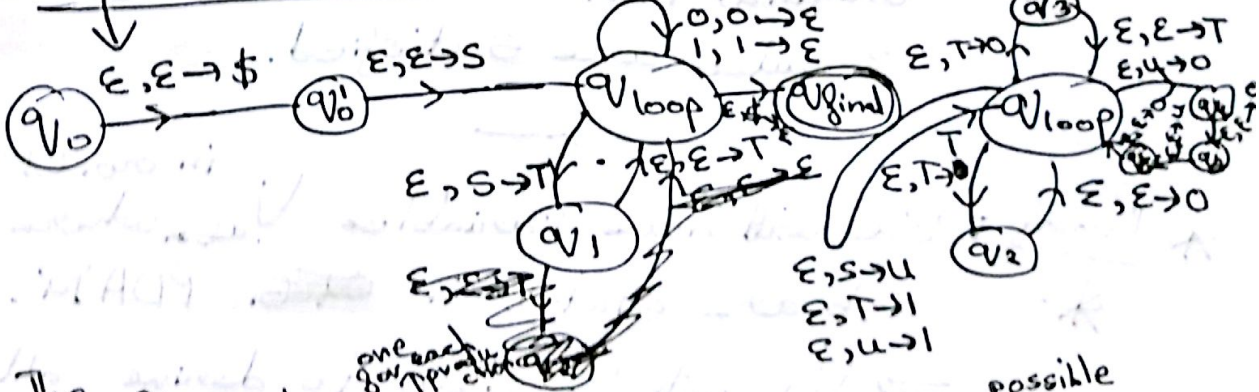
$U \to 0U00 \mid 1$

Terminals ↓ $\{0, 1\}$



ε, S → TT
ε, S → U
ε, T → 0T
ε, T → T0
ε, T → 1
ε, U → 0U00
ε, U → 1

Show each transition without using string transitions.



The q_{loop} has 7 transitions (loops) with possible intermediate states, one for each production.

---

*PDA → CFG (Not there as of now)

- Given a PDA M, we need to show there exists a CFG G such that $L(M) = L(G)$.

⇒ We will consider a type of PDAs such that,

* a) M has one accept state } Make a new accept state from old one
  b) M empties its stack before accepting
  c) M either pushes a symbol or pops a symbol at every transition.
     - If M pops & then pushes, we can can make this transition across 2 states.
     - If it doesn't do anything to the stack, we push a dummy variable onto the stack & then pop it.

Empty stack one symbol at a time if at accept state

∴ We were able to convert our original PDA to M such that the 3 rules are satisfied.

—x—

* Proof: We will have variables V_{abx} where a, b are states in PDA M'. in our CFG

⇒ V_{ab} will be able to derive all strings 'w' such that M in a stack a on consuming 'w' goes to state b, starting & ending with empty stack

*Non-CF languages:

*Pumping Lemma for CFLs:

- Let $|V|$ = No. of non-terminals in G.
  (b = branching factor (max) = Largest no. of symbols across any production rule)
- Consider string w of length atleast p

$$p = b^{|V|+2} \rightarrow \text{height} = |V|+2 \text{ (of parse tree)}$$

$\Rightarrow$ Consider a string 's' of length $\geq p$

- Because of our string chosen, the parse tree will have atleast $|V|+2$ nodes.

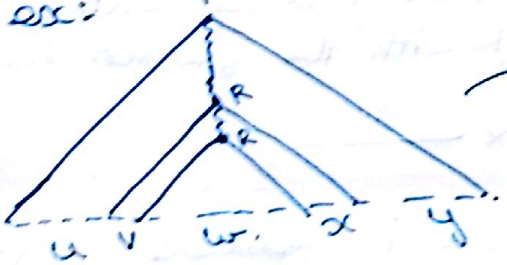
(tree height $\geq |V|+2$)

$\Rightarrow$ On the longest path (ignoring the terminal symbol), we have $|V|+1$ non-terminal nodes. $\therefore$ Atleast one variable will repeat $\rightarrow$ Lets say this is R.

- To keep R unique, we pick the R with the longest height.

T $\rightarrow$ is the tree with the smallest no. of nodes for string S.

ex:



$\rightarrow$ Parse tree for S,

$S = uvwxy$

- Now we can see that strings,

$S' = uvvwxxy$,

$\Rightarrow S^i = uv^iwx^iy$ (On repeating the part b/w R to R multiple times)

(i could also be 0)

$\therefore S^0 = uwy$ also belongs to the CFL.

—x—

*Pumping Lemma for CFLs statement:

- Given a CFL(L), there exists p = pumping length, such that all strings w in the CFL of length $\geq p$ can be split as $uvwxy$ such that,
  - i) $i \geq 0$, $uv^iwx^iy \in L$
  - ii) $|vx| \geq 0$, If both $|v|=0$ & $|x|=0$, then the tree we consider is not the smallest tree for string S. (So the lemma doesn't apply)
  - iii) $|vwx| \leq p$,
    - Since we can find a repeated state R such that the gap b/w them is atmost $|v|+1$.
    - ∴ We can see that $b^{|v|+2}$ is always possible to be the max length of $|vwx|$
      - ↳ We can always select an R, such that this is satisfied (R with the greatest height)

Note: We often use this to prove languages are CFLs. (Similar to how we used the pumping lemma to show non regular languages)

---

*ex: $L = \{a^n b^n c^n \mid n \text{ at least } 0\}$

→ Show that it is not a CFL.

⇒ Consider the string $S = a^p b^p c^p$

- we cannot find any split as $uvwxy$ such that $|vwx| \leq p$. → so S cannot be pumped.

---

ex: $L = \{ww \mid w \in \{0,1,*\}\}$

*

- This is not context free whereas ww^R is context free.

---

*Turing Machine:

- Consider a finite state automata along with an infinitely long tape starting from location 0. » Unlike the limited memory of our computers

⇒ The machine can move a head left/right and read/write data on the cell pointed by the head.

- This allows us to store our input & read the input multiple times.

*⇒ This machine can recognize non CFLs,

ex: $L = \{a^n b^n c^n \mid n \text{ at least } 0\}$

→ One way is to identify the length & then use it.

→ Another can be to cross out a single 'a', 'b', 'c' on each pass & perform n passes.

* ⇒ Turing Machine(M) = $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$

*

Q = Set of states

Σ = Input Alphabet

Γ = Tape Alphabet → superset of Σ

$\delta = Q \times \Gamma \longrightarrow Q \times \{L, R\}$ (Transition function)

q_0 = Start State

q_a = Accept state

q_r = Reject state.

⇒ * $\delta(a, q) \longrightarrow (b, q', R)$

- On reading input a from state q, replace symbol a with b on (tape) and goes to a state q', and moves left or right on the tape.

- Input is always present on tape starting from index 0 (leftmost of tape).

——x——

Note: * Turing machine can face the halting problem, → q_r or q_a never reached.

——x——

*Configuration: (How we represent the machine during computation)

- The current state
- The position of the head.
- The contents of the tape.

⇒ If head is positioned at w_i

⇒ Contents of tape ⇒ $w = w_1 w_2 \ldots\ldots w_m$

- We represent the configuration as,

*Config = $w_1 w_2 \ldots\ldots w_{i-1} q w_i w_{i+1} \ldots\ldots w_m$

⇒ We put our state just before the position of the head so a single string has all the info we need.

———x———

*ex: a configuration $u\,a\,q\,b\,v$ yields a configuration $u\,q'\,a\,b'\,v$.

⇒ $\delta(q, b) \longrightarrow (q', b', 'L')$

↳ Move left

———x———

Note: Turing decidable: Machine halts on all inputs

** Turing recognizable: Machine may not halt on all inputs.

Turing recognizable > Turing decidable

———x———

eg: Turing machine recognizable languages.

** → Multiplication, Powers of 2 etc are
- Turing machine decidable.

→ $L = \{wrw \mid w \in \{a,b\}^*\}$

$L = \{a^i b^j c^k \mid i = j = k\}$

$L = \{a^i b^j \mid j = 2^i, i \neq 0\}$

- These are all turing machine decidable.

→ $L = \{a^i b^j c^k \mid k = i+j\}$

$L = \{a^i b^j c^k \mid k = i \times j\}$

- These are turing machine decidable.

---

* Types of Turing Machines:

*1) Multitape TM:

- 2 tapes with one head each.
- $L = \{wrw \mid w \text{ in } \{a,b\}^*\}$. → Move second head to the middle of string on tape 2 after copying input from tape 1 to tape 2.

↓

⇒ Match character by character and go to accept state on reaching end of string

$\Rightarrow \delta : Q \times T^k \longrightarrow Q \times T^k \times \{L,R\}^k$

- Only one state q of our TM though.

* Equivalence of k-tape TM & single tape TM:

- One of the solution is to use a multitrack turing machine $\longrightarrow$ k tracks so a single head can read k values at the same time.

$\Rightarrow$ Another solution is to split a single tape into chunks of size k.

- Chunk$_i$ = All the i^{th} values of all the k tapes.

---

*2) Non-deterministic TM:

$(Q \times T \times \{L,R\})$

- $\delta : Q \times T \longrightarrow 2$

ex: $L = \{a^i \mid i \text{ is not prime}\}$

- Use two tapes $\longrightarrow$ on the 2nd tape it will write numbers b/w $2 \& i-1$
- If the number on the 2nd tape divides i, then TM accepts.

*

* Equivalence of Nondeterministic TM & normal TM

- We can show equivalence using a three tape deterministic TM. (assuming NDTM has a single tape)

Tape 1 → Input
Tape 2 → Working tape
Tape 3 → Encoding

- Consider a tree describing our NDTM

⇒ Let us assume the max. no of choices
* at any state is b.

↳ Each node can be labeled with a string over the b symbols

- ex: 15238...2 ⇒ first choice in step 1, fifth choice in step 2 and so on...

⇒ Every node has string telling us how to get there.

* Use a breadth wise approach so that a non-halting branch will not effect the other branches.

↳ Follow a lexicographic ordering of the strings (This gives us the breadth wise ordering)

- At each stage copy this string onto tape 3, and then copy input from tape 1 to tape 2 & apply corresponding actions to tape 2.

———x———

* Enumerator:

- A TM which can list / enumerate all the members of the language in any order

⟹ If we are given an input string w, start the enumerator E, compare each string with w using the TM

- So this is equivalent to a Language L.

→ With multiple inputs, iterate over a max step size & each of the strings trying to check for an accept state on string(i).

↳ This way we won't be stuck on a non-accepting string forever.

↳ max step size → limits the number of actions the E can perform.

Note: Recursive languages → TM-Decidable

Recursively enumerable → TM-recognizable

Note: The union of a Recursive T.M (M_1) & Recursive T.M (M_2) is also a Recursive TM (M)

- M_1 for L_1
  M_2 for L_2
  M for $L_1 \cup L_2$

(Proof: Write like a C program)
if M_1 accepts ...
else if ...
...

— x —

Note: The union of an R.E T.M (M_1) & an R.E T.M (M_2) $\Rightarrow$ R.E T.M(M)

- On input w ($t=1$)

Repeat
  Run M_1 on w for t steps
  Run M_2 on w for t steps
Until either M_1 halts by accept
  or M_2 halts by accept.

- TM M accepts & halts

— x —

Note: Similarly for the intersection of two T.M s ⟶ Also a TM

— x —

*Restricted Turing Machine: (LBA)

* L → Context Sensitive grammars
  ↳ Cannot produce ε

- In CFGrs, $A \to w$ rules exist

⇒ Here we also allow,

$$a_1 A a_2 \to a_1 w a_2 \quad (w \neq \varepsilon)$$

(A: Context Sensitive)

- a_1, a_2 can be ε or strings of variables & terminals.

———×———

* *esc: $L = \{a^i b^i c^i \mid i >= 1\}$

$S \to aPbc \mid abc$
$aPb \to aaPbbQ$
$Qc \to cc$
$Qb \to bQ$
$P \to \varepsilon$ } Need to replace this rule

} Context Sensitive Grammar

———×———

⇒ A restricted TM → Size of Tape is bounded to be linear in the size of the input

* • This TM ⇒ Linear Bounded Automata (LBA)

- Tape size = O(Length of input)

* ⇒ Every CSL has an LBA & vice versa (except ε)