Greibach Normal Form (GNF) & CNF to GNF conversion

Greibach Normal form (GNF)

⇒ A CFG is in Greibach Normal form if the productions are in the following forms:

$$A \rightarrow b$$
$$A \rightarrow bC_1C_2\cdots C_n$$

where $A, C_1, \ldots, C_n$ are Non-Terminals and $b \in T$

Steps to Convert a given CFG to GNF:

Step 1: Check if the given CFG has any Unit productions or NULL Productions and Remove if there are any.
[Using Unit and Null production Removal Techniques]

Step 2: Check whether the CFG is already in CNF and Convert it to CNF if it is not.
[Using CFG to CNF conversion technique]

Step 3: Change the names of the Non-Terminal Symbols into some A_i in ascending order of i.

CFG to GNF

Example:

$S \rightarrow CA \mid BB$

$B \rightarrow b \mid SB$

$C \rightarrow b$

$A \rightarrow a$

Replace: S with A_1, C with A_2, A with A_3, B with A_4 (Not Random ↓)

we set:

$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$

$A_4 \rightarrow b \mid A_1 A_4$ *

$A_2 \rightarrow b$

$A_3 \rightarrow a$

~~Step 2~~:

Step 4: Alter the rules so that the Non-Terminals are in ascending order, such that, If the production is of the form $A_i \rightarrow A_j x$, then, $i < j$ and should never be $i \geq j$

* $A_4 \rightarrow b \mid A_1 A_4$

$A_4 \rightarrow b \mid A_2 A_3 A_4 \mid A_4 A_4 A_4$

$A_4 \rightarrow b \mid b A_3 A_4 \mid A_1 A_4 A_4$ [$i = j$]

$A_4 \rightarrow b \mid b A_3 A_4 \mid$ Left ~~Recursion~~ Recursion

[A_4 is calling itself]

CFG to GNF

Step 5: Remove Left Recursion.

⇒ Introduce a new variable to remove the Left Recursion

$A_4 \rightarrow b \mid bA_3A_4 \mid A_4A_4A_4$

$Z \rightarrow A_4A_4Z \mid A_4A_4$ ✓

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$ ✓

Now the grammar is:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$ *

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

$Z \rightarrow A_4A_4 \mid A_4A_4Z$ * *

$A_2 \rightarrow b$

$A_3 \rightarrow a$

GNF

$A_1 \rightarrow bA_3 \mid bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4$

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

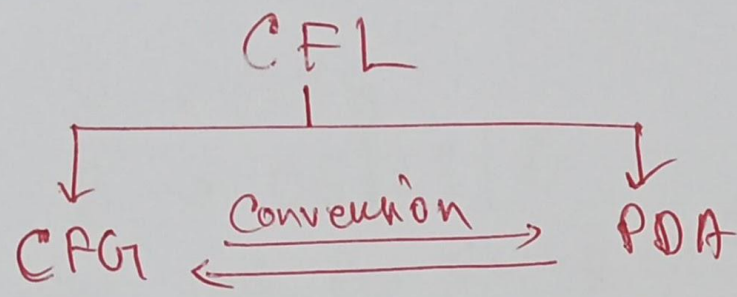
$Z \rightarrow bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4 \mid$
$bA_4Z \mid bA_3A_4A_4Z \mid bZA_4Z \mid bA_3A_4ZA_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Inter-conversion (CFG & PDA)

CFG to PDA Conversion



⇒ CFG → PDA:

Let $L = L(G)$ where $G = (V, \Sigma, P, S)$ is a CFG, we construct a PDA (A) with empty stack as

$$A = (q, \Sigma, \{V \cup \Sigma\}, \delta, q, S, \phi)$$

Rule 1: $\delta(q, \epsilon, A) = \{(q, \alpha) \mid A \to \alpha \text{ is in } P\}$

Rule 2: $\delta(q, \beta, \beta) = \{(q, \epsilon)\}$ for every $\beta \in \Sigma$

CFG to PDA Conversion

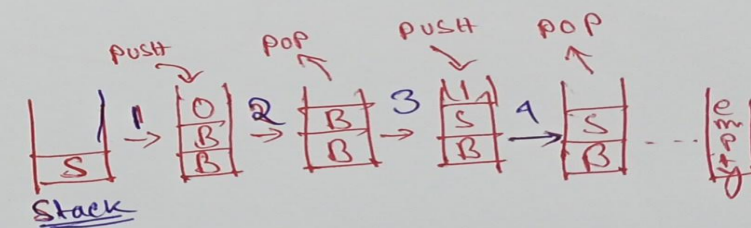
Example:

$S \rightarrow OBB$
$B \rightarrow OS \mid 1S \mid 0$
Also test for string 010^4

R1: $\delta(q, \epsilon, S) = (q, OBB)$
R2: $\delta(q, \epsilon, B) = (q, OS)$
R3: $\delta(q, \epsilon, B) = (q, 1S)$
R4: $\delta(q, \epsilon, B) = (q, 0)$

R5: $\delta(q, 0, 0) = (q, \epsilon)$
R6: $\delta(q, 1, 1) = (q, \epsilon)$

⇒ testing of string 010^4



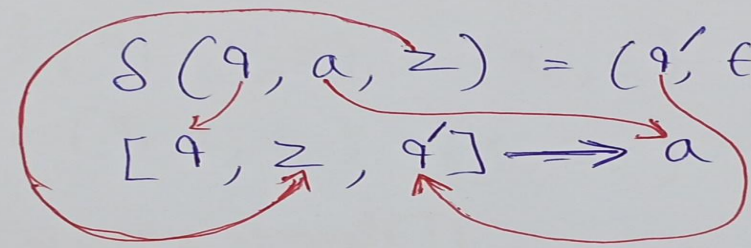
1. $(q, 010^4, S)$ [R1]
2. $(q, 010^4, OBB)$ [by R5]
3. $(q, 10^4, BB)$ [by R3]
4. $(q, 10^4, 1SB)$ [by R6]
5. $(q, 0^4, SB)$ [by R1]
6. $(q, 0^4, OBBB)$ [by R5]
7. $(q, 0^3, BBB)$ [by R4]
8. $(q, 0^3, OBB)$ [by R5]
9. $(q, 0^2, BB)$ [by R4]
10. $(q, 0^2, OB)$
    ↓ by R5 and R4 (11, 12)
13. (q, ϵ, ϵ)

PDA to CFG

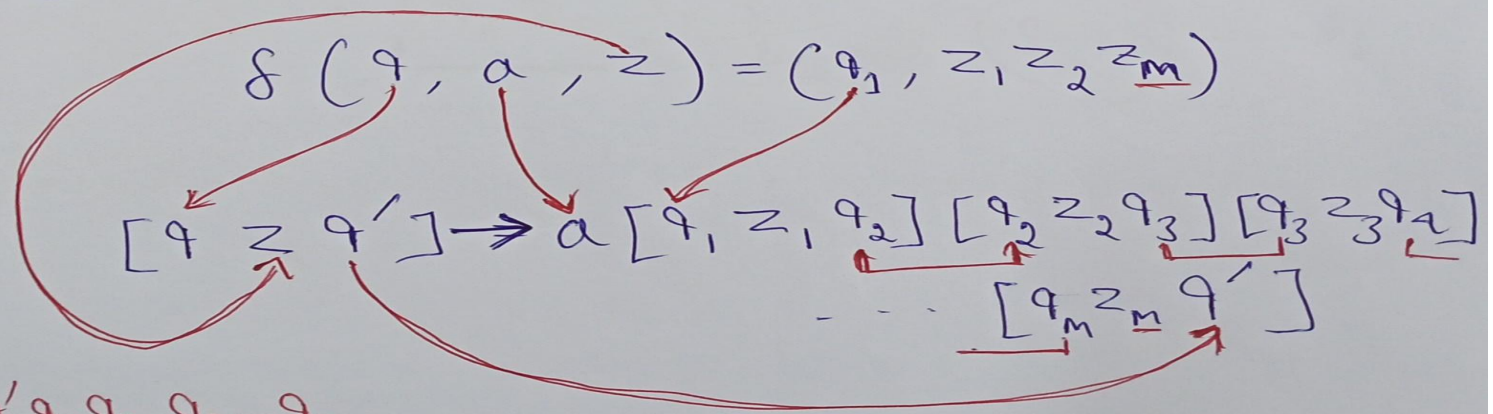
⇒ The productions (P) are produced by following the moves of PDA as follows

1) S productions are given by $S \rightarrow [q_0 Z_0 q]$ for every $q \in Q$

2) Each erasing move, i.e. $\delta(q, a, Z) = (q', \wedge)$, induces production $[q, Z, q'] \rightarrow a$



(3) Non-erasing moves:



where $q', q, q_1, q_2, q_3 \ldots q_m$ can be any state in Q

2^m → combinations

PDA to CFG

Example: $A = (\{q_0, q_1\}, \{a, b\}, \{z_0 z\}, \delta, q_0, z_0, \emptyset)$

$\delta(q_0, b, z_0) = (q_0, z z_0)$

$\delta(q_0, \epsilon, z_0) = (q_0, \epsilon)$

$\delta(q_0, b, z) = (q_0, z z)$

$\delta(q_0, a, z) = (q_1, z)$

$\delta(q_1, b, z) = (q_1, \epsilon)$

$\delta(q_1, a, z_0) = (q_0, z_0)$

For starting symbol (i)

$S \rightarrow [q_0 z_0 q_0]$ ... A

$S \rightarrow [q_0 z_0 q_1]$ ... B

$S \rightarrow A \longrightarrow P_1$

$S \rightarrow B \longrightarrow P_2$

PDA → CFG

$\delta(q_0, b, Z_0) = (q_0, Z\,Z_0)$

$2^2 = 4$ combinations

$[q_0\ Z_0\ ] \to b\,[q_0\ Z\ ][\ Z_0\ ]$

$[q_0\ Z_0\ ] \to b\,[q_0\ Z\ ][\ Z_0\ ]$

$[q_0\ Z_0\ ] \to b\,[q_0\ Z\ ][\ Z_0\ ]$

$[q_0\ Z_0\ ] \to b\,[q_0\ Z\ ][\ Z_0\ ]$

$Q = \{q_0, q_1\}$

$[q_0\ Z_0\ q_0\ ] \to b\,[q_0\ Z\ q_0]\,[q_0\ Z_0\ q_0]$ — P_3

$[q_0\ Z_0\ q_0\ ] \to b\,[q_0\ Z\ q_1]\,[q_1\ Z_0\ q_0]$ — P_4

$[q_0\ Z_0\ q_1\ ] \to b\,[q_0\ Z\ q_0]\,[q_0\ Z_0\ q_1]$ — P_5

$[q_0\ Z_0\ q_1\ ] \to b\,[q_0\ Z\ q_1]\,[q_1\ Z_0\ q_1]$ — P_6

$\Rightarrow \delta(q_0, \epsilon, Z_0) = (q_0, \epsilon)$

$[q_0\ Z_0\ q_0\ ] \to \epsilon$ — P_7

PDA → CFG

$\Rightarrow \delta(q_0, b, Z) = (q_0, ZZ)$ $\quad 2^2 = 4$ combinations

2. $\quad P_8 \rightarrow P_{11}$

$\Rightarrow \delta(q_0, a, Z) = (q_1, Z)$ $\quad 2^1 = 2$ combinations

$[q_0 Z q_0] \Rightarrow a [q_1 Z q_0]$ — P_{12}

$[q_0 Z q_1] \rightarrow a [q_1 Z q_1]$ — P_{13}

$\Rightarrow \delta(q_1, b, Z) = (q_1, \epsilon)$

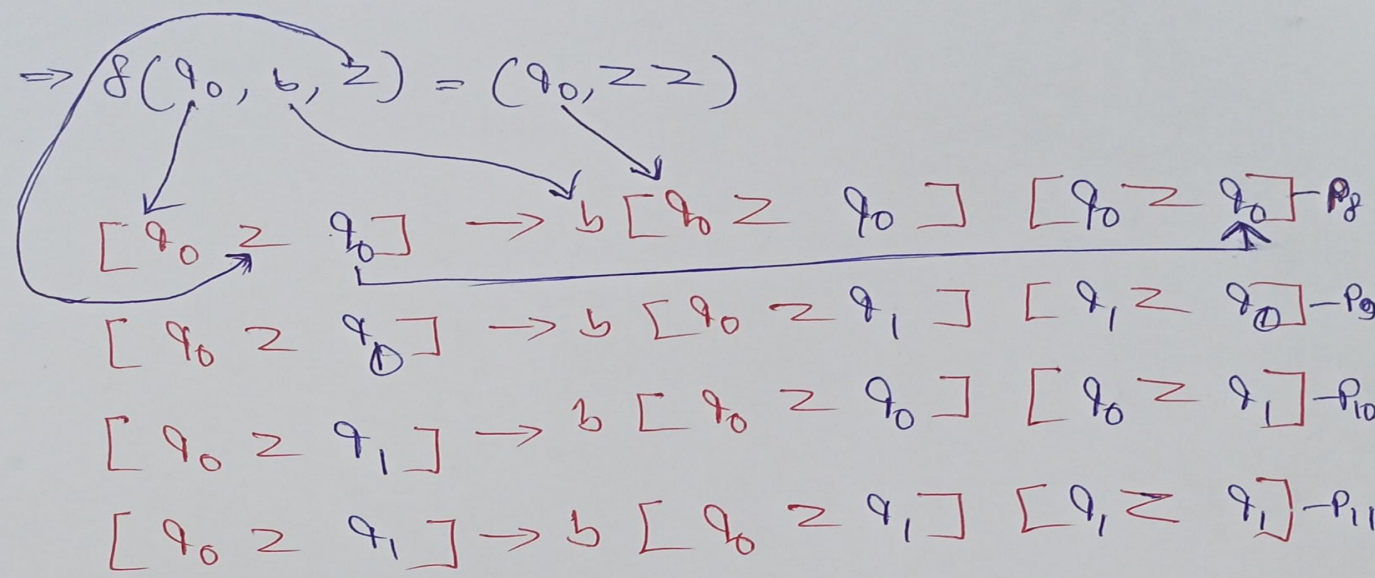
$[q_1 Z q_1] \rightarrow b$ — P_{14}

$\Rightarrow \delta(q_1, a, Z_0) = (q_0, Z_0)$ $\quad 2^1$

$[q_1 Z_0 q_0] \rightarrow a [q_0 Z_0 q_0]$ — P_{15}

$[q_1 Z_0 q_1] \rightarrow a [q_0 Z_0 q_1]$ — P_{16}

PDA → CFG



States (Assumed)

$[q_0 Z_0 q_0] = A$

$[q_0 Z q_0] = D$

$[q_0 Z_0 q_1] = B$

⋮

$S \to A$

$S \to B$

$S \to A \mid B$

P8 → $A \to b\,A\,A$

Pumping Lemma for CFL

Pumping Lemma for CFL

⇒ Pumping Lemma (for CFL) is used to prove that a language is NOT Context Free.

If A is a CFL, then, A has a Pumping Length 'p' such that any string 'S', where $|S| \geq p$ may be divided into 5 pieces $S = uvxyz$ such that the following conditions must be true:

1) uv^ixy^iz is in A for every $i \geq 0$
2) $|vy| > 0$
3) $|vxy| \leq p$

⇒ Following are the steps to prove a language is not CFL; [Using Contradiction]

1. Assume that A is Context free
2. It has to have a pumping length (say p)
3. All string longer than p can be pumped $|S| \geq p$
4. Now find a string 'S' in A such that $|S| \geq p$
5. Divide S into uvxyz
6. Show that $uv^ixy^iz \notin A$ for some i
7. Then Consider the ways that S can be divided into uvxyz
8. Show that none of these can satisfy all the 3 pumping conditions at the same time.
9. S can not be pumped == CONTRADICTION

Example – 1

⇒ Show that $L = \{ a^N b^N c^N \mid N \geq 0 \}$ is not Context Free.

① Assume that L is Context Free

② L must have a pumping length (say P)

③ Now we take a string S such that $S = a^P b^P c^P$

④ We divide S into parts uvxyz

Ex. P = 4 So, $S = a^4 b^4 c^4$

Case I: v and y each contain only one type of symbol.

aaaabbbbcccc

u v x y z

$uv^i x y^i z$ [i = 2]

$uv^2 x y^2 z$

aaaaaabbbbccccc

$a^6 b^4 c^5 \notin L$

Case II: Either v or y has more than one kind of symbols.

aaaabbbbcccc

u v x y z

$uv^i x y^i z$ [i = 2]

$uv^2 x y^2 z$

aaaabbaabbbbbcccc

$\notin L$

Example – 2

Show that $L = \{ww \mid w \in \{0,1\}^*\}$ is NOT Context Free.

→ Assume that L is Context Free

→ L must have a pumping length (say P)

→ Now we take a string S such that $S = 0^P 1^P 0^P 1^P$

→ we divide S into parts uvxyz

Ex. $P = 5$ so, $S = 0^5 1^5 0^5 1^5$

Case 1: vxy does not straddle a boundary
→ doesn't cross the boundary.

$\underbrace{0\,0\,0\,0\,0^*}_{u}\underbrace{1\,1\,1\,1}_{vxy}\underbrace{1^*\,0\,0\,0\,0\,0^*\,1\,1\,1\,1\,1}_{z}$

$|vxy| \leq P$

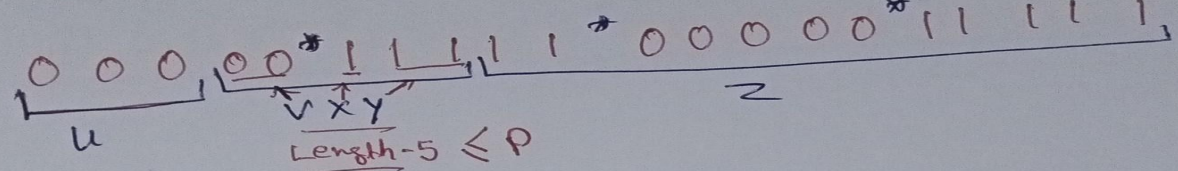
$uv^i xy^i z \Rightarrow uv^2 xy^2 z$

$0\,0\,0\,0\,0\,1\,1\,1\,1\,1\,1\,0\,0\,0\,0\,0\,1\,1\,1\,1\,1$

$0^5 1^7 0^5 1^5 \notin L$

Example 2

Case 2a: vxy straddles the first boundary

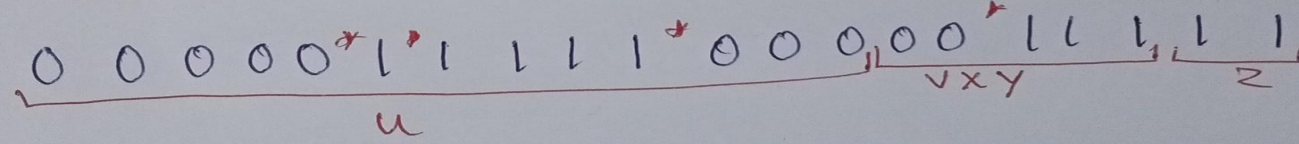


uv^2xy^2z

0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1

$0^7 1^7 0^5 1^5 \notin L$

Case 2b: vxy straddles the third boundary



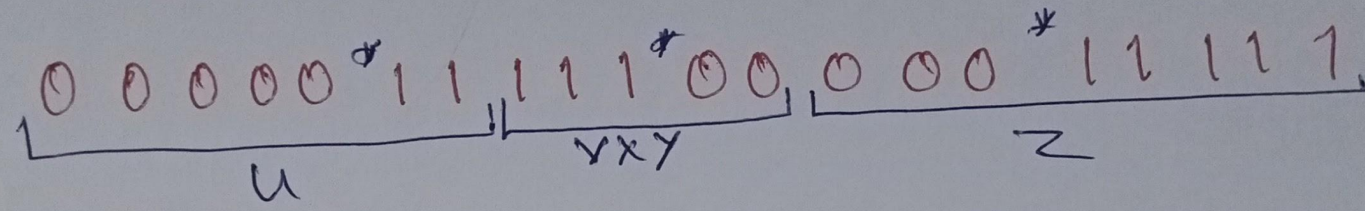
uv^2xy^2z

0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1

$0^5 1^5 0^7 1^7 \notin L$

Example 2

Case 3: vxy straddles the midpoint



uv^2xy^2z

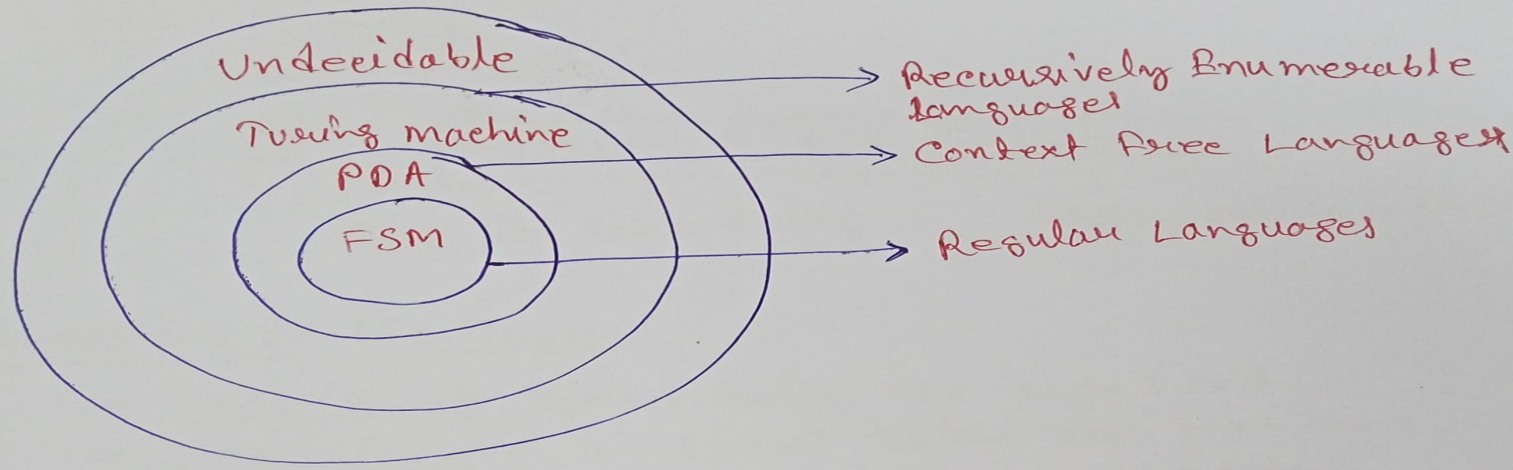
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1

$0^5 1^7 0^7 1^5 \notin L$

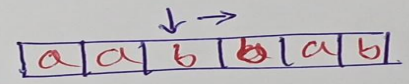
L is not Context Free.

Turing Machine

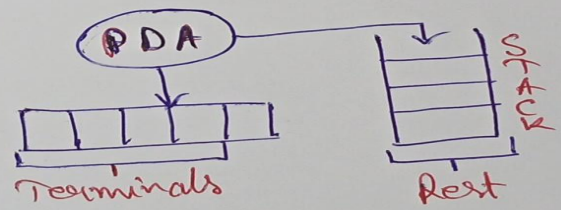
Turing Machine - Introduction



FSM: → The Input String

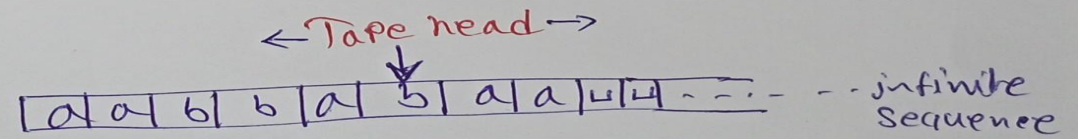


PDA: → The Input String
→ A Stack



TURING MACHINE:

→ A Tape



Tape Alphabets: $\Sigma = \{0, 1, a, b, x, z_0\}$

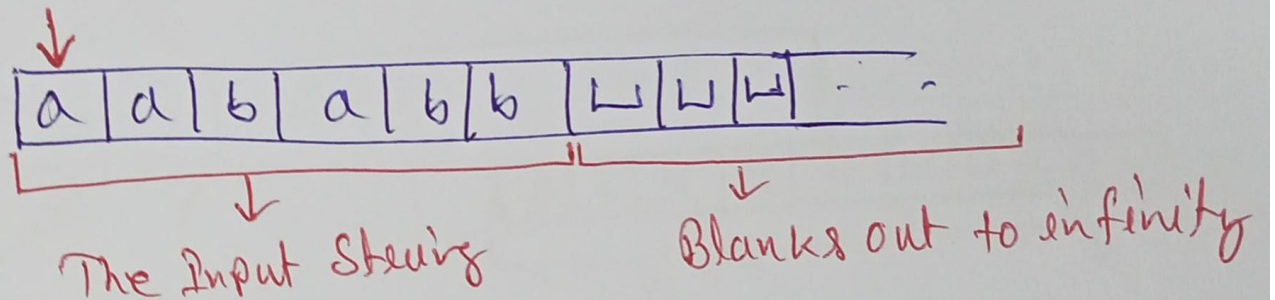
The Blank ␣ is a special symbol, where $␣ \notin \Sigma$

↓

It is a special symbol used to fill the infinite tape.

TM - Introduction

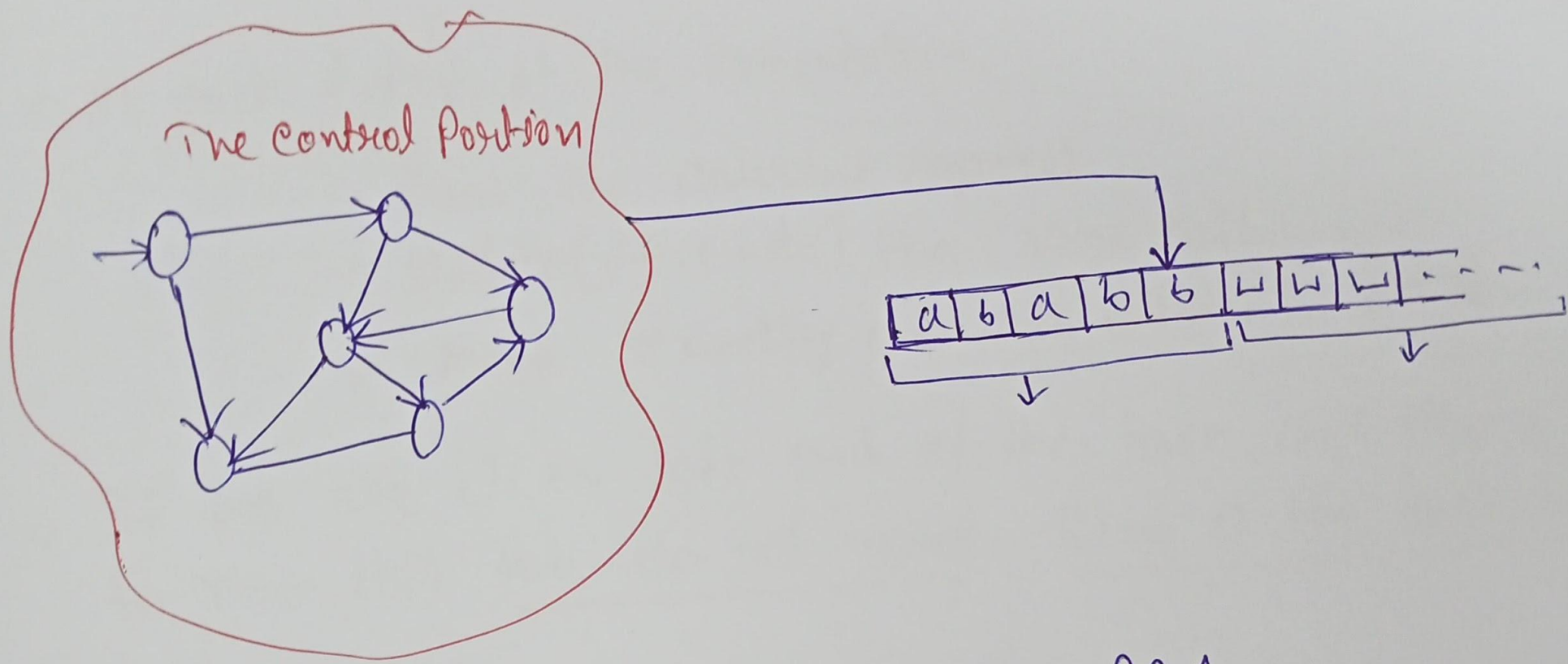
Initial Configuration:



Operations on the Tape:

- → Read/Scan symbol below the tape head.
- → Update/Write a symbol below the tape head.
- → Move the tape head one step LEFT
- → Move the tape head one step Right.

TM - Introduction



The control portion similar to FSM or PDA

The program

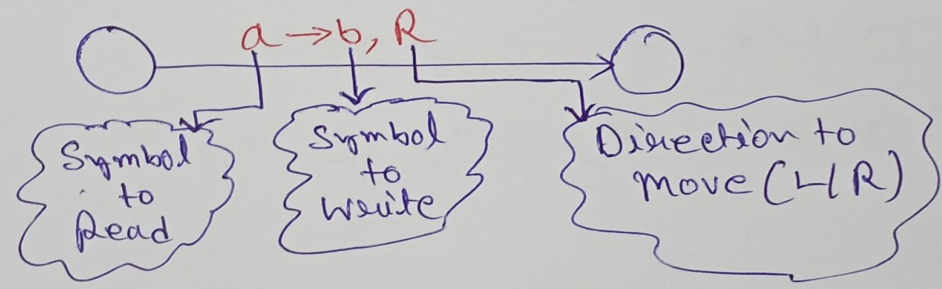
It is deterministic

Rules of Operation - 1 (TM)

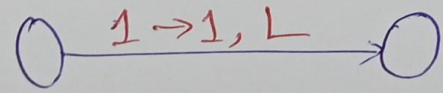
⇒ At each step of the computation:

- → Read the current symbol
- → Update (i.e. write) the same cell
- → move exactly one cell either L or R

⇒ If we are at the left end of the tape, and trying to move left, the do not move. Stay at the left end.



⇒ If you don't want to update the cell, just write the same symbol,



Rules of operation - 2 (TM)

→ Control is with a sort of FSM

→ Initial State

→ Final States: (there are two final states)

1) The ACCEPT state
2) The REJECT state

→ Computation can either

1) HALT and ACCEPT
2) HALT and REJECT
3) LOOP (the machine fails to HALT)

Turing machine (Formal Definition)

⇒ A Turing machine (TM) can be defined as a set of 7 tuples

$$(Q, \Sigma, \Gamma, \delta, q_0, b, F)$$

Q → Non empty set of States

Σ → Non empty set of Symbols

Γ → Non empty set of Tape symbols

δ → Transition function defined as

$$Q \times \Sigma \rightarrow \Gamma \times (R/L) \times Q$$

q_0 → Initial State

b → Blank Symbol

F → Set of final states (ACCEPT & REJECT States)

⇒ Thus, the production rule of TM will be written as

$$\delta(q_0, a) \rightarrow (q_1, Y, R)$$

Turing Thesis:

⇒ It states that any computation that can be carried out by mechanical means can be performed by some Turing Machine.

Few arguments for accepting this thesis are:

1) Anything that can be done on existing digital computer can also be done by TM.

2) No one has ~~yet~~ yet been able to suggest a problem solvable by what we consider an algorithm, for which a TM program cannot be written.

Recursively Enumerable Language:

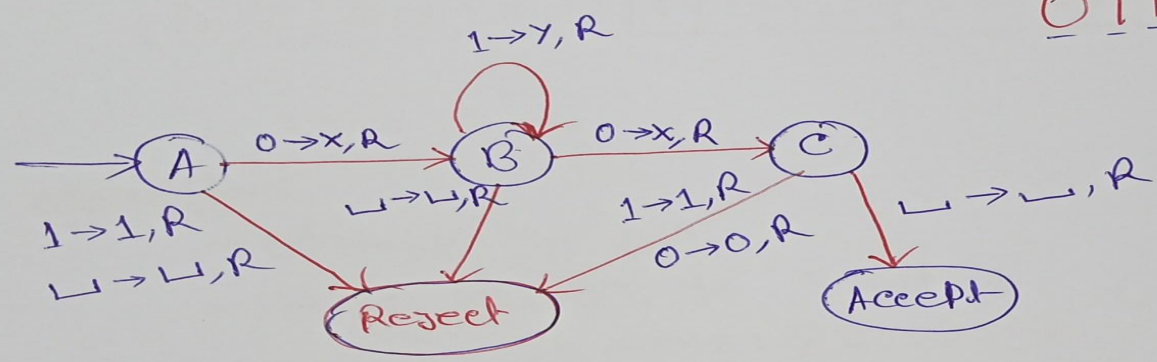
A Language L and Σ is said to be Recursively Enumerable if there exists a TM that accepts it.

Turing Machine - Example

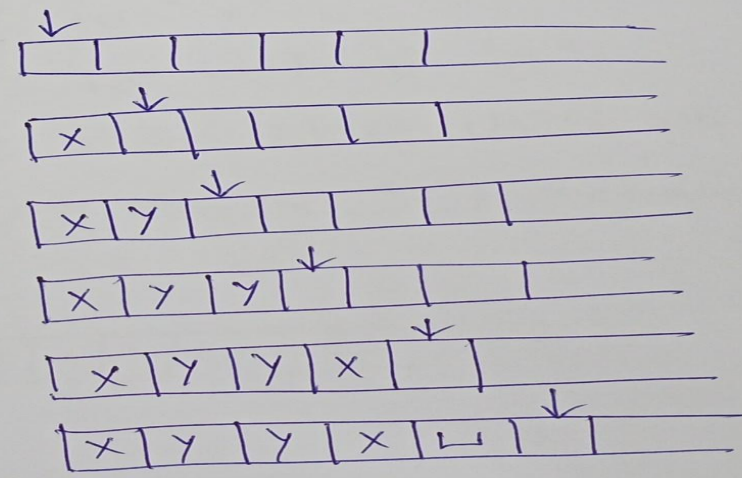
⇒ Design a TM which recognizes the language

$$L = 01^*0$$

0 1 1 0 ␣



Tape →



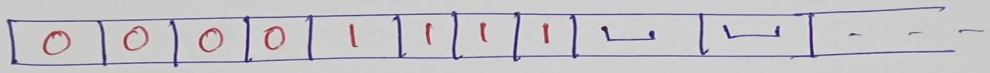
⇒ Deterministic TM [$\Sigma = \{0, 1\}$, $b = ␣$]

⇒ TM is more powerful than PSM and PDA

Turing Machine - Example

Design a TM which recognises the language

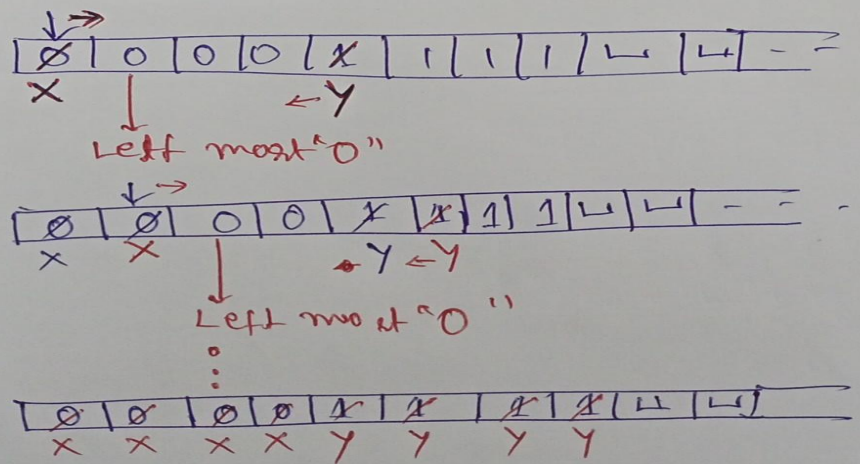
$L = 0^N 1^N$



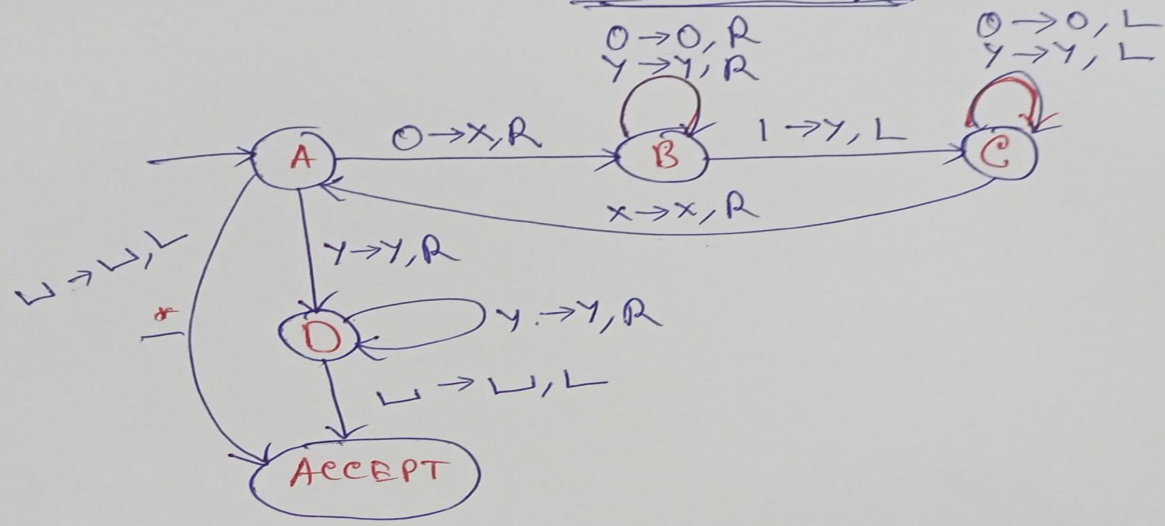
Algorithm:

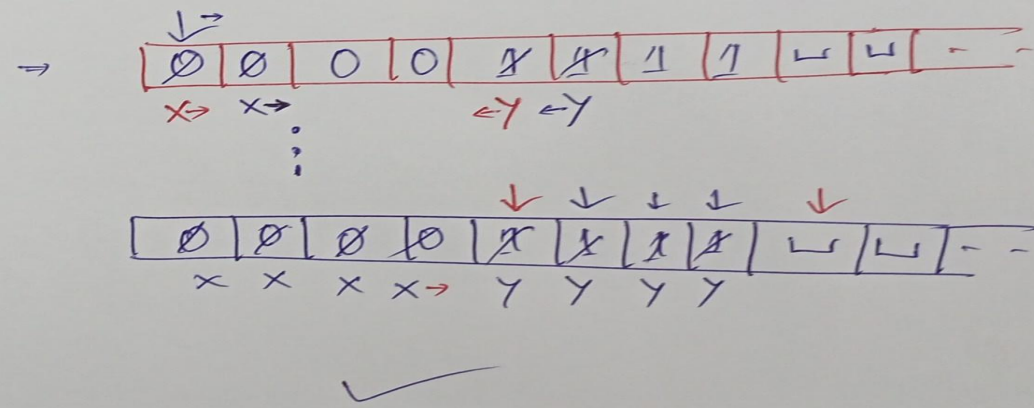
→ Change "0" to "x"
→ move RIGHT to first "1"
If none: REJECT

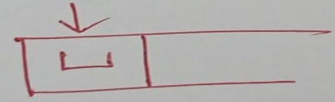
→ Change "1" to "y"
→ move LEFT to Leftmost "0"
→ Repeat the above steps until no more "0"s
→ make sure no more "1"s remain.



TM - Example

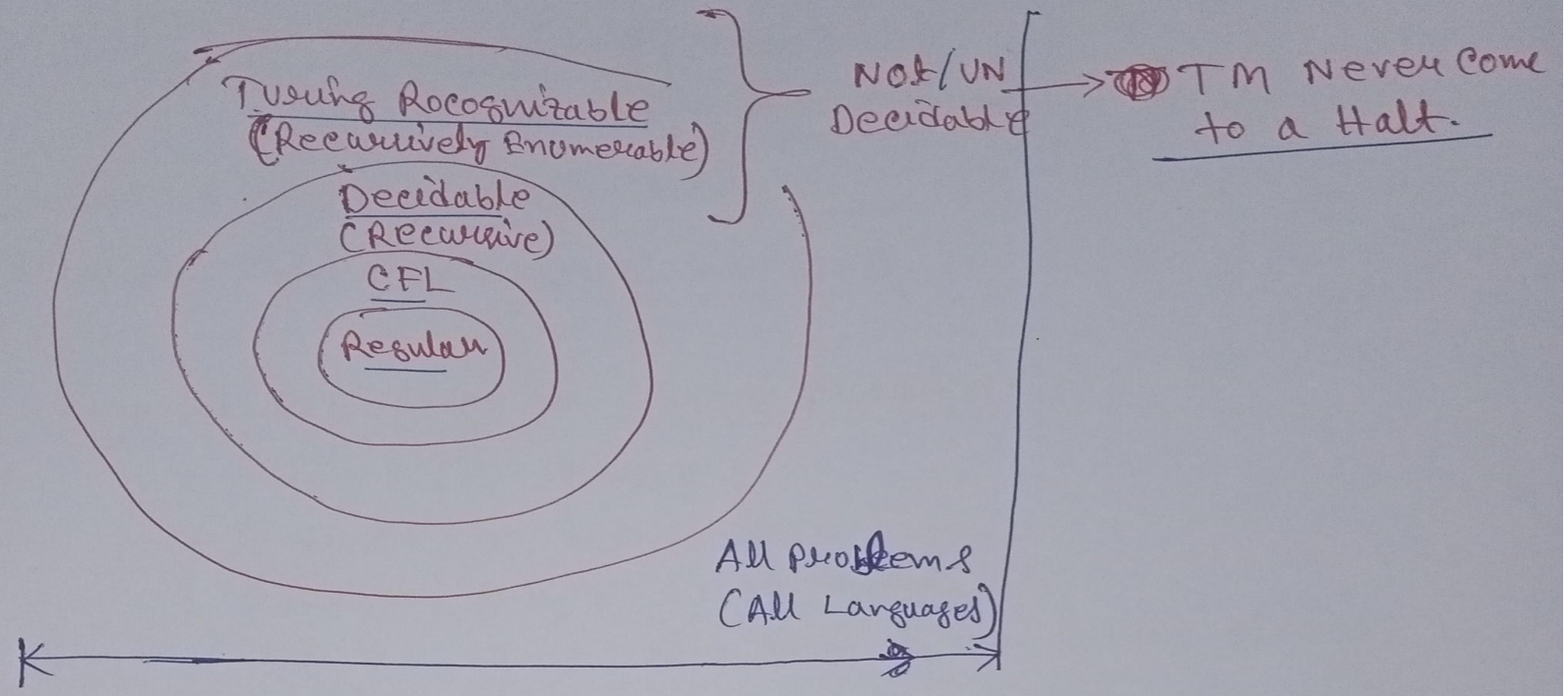




* 

NOTE! Any edge that has a missing transition that will go to the REJECT State

Different Classes of Languages



Turing machine Languages:

⇒ Decidable (Recursive) → that can be decided by a TM.

① Accept and Halt.
② Reject and Halt.

→ Always come to a Halt.

⇒ Turing Recognizable → (Recursively Enumerable)

① Accept and Halt
② Loop (may not Halt)

The CHURCH-TURING Thesis

⇒ What does COMPUTABLE mean?

→ FUZZY Term

① Alonzo Church - LAMDA CALCULUS

② Allen Turing - TURING MACHINE

Several Variations of Turing Machine:

- → One Tape or many
- → Infinite on both ends
- → Alphabets only {0, 1} or more?
- → Can the Head also stay in the same place?
- → Allow Non-Determinism

All variations are equivalent in Computing Capability

Turing machine and Lambda calculus are also equivalent in power

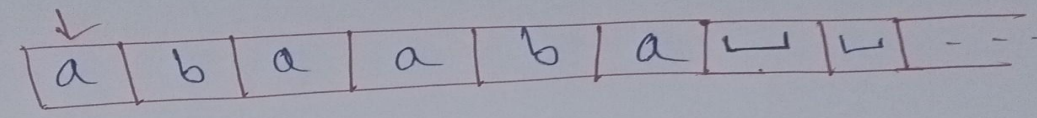
Algorithmically Computable means Computable by Turing machine

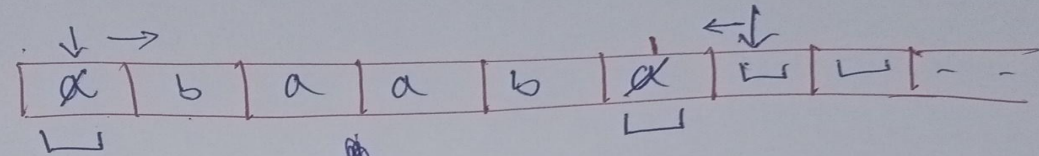
NOTE: TURING TEST ≠ TURING MACHINE

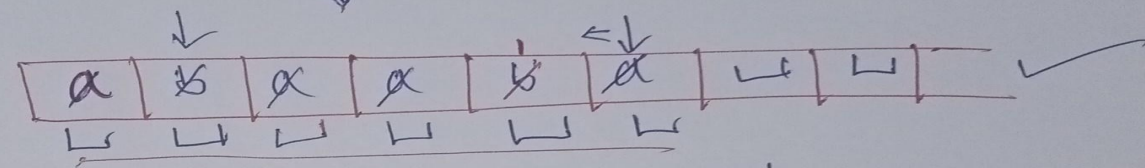
Turing machine for Even Palindromes

⇒ Design a TM that accepts Even palindromes over the alphabet $\Sigma = \{a, b\}$

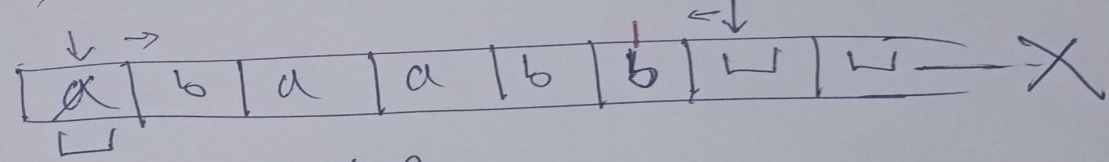
Ex. ①

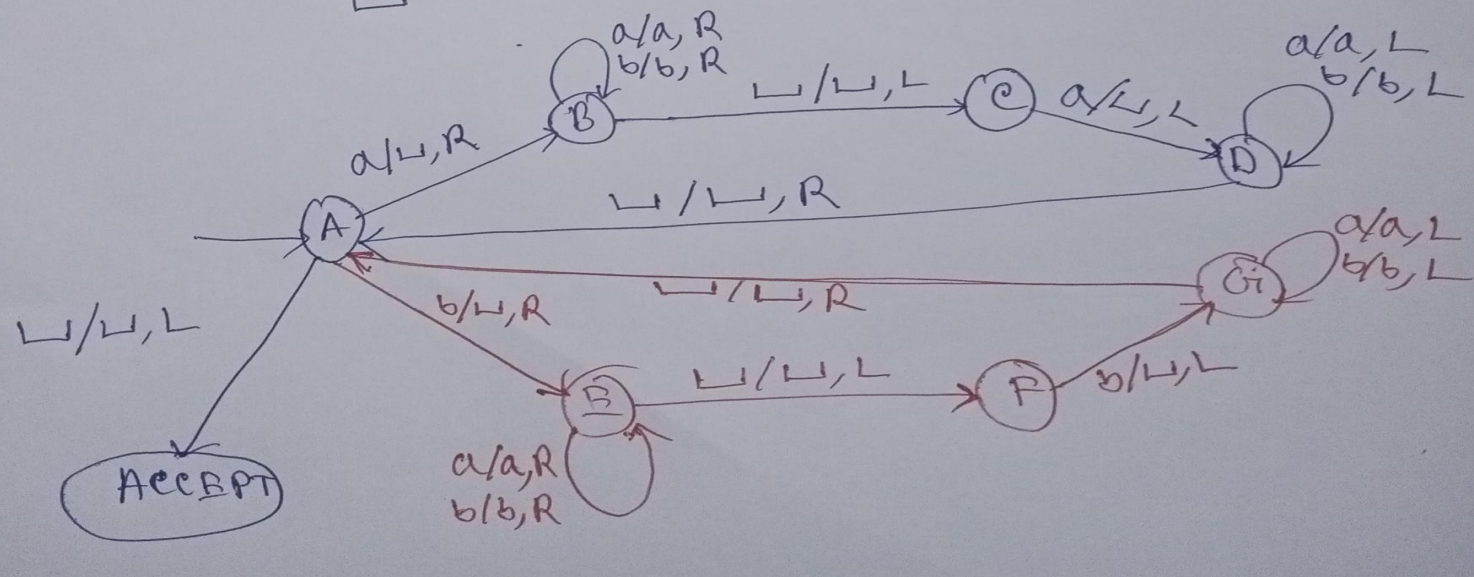






②





Multitape TM and Decidability & Undecidability

Multitape Turing machine

Theorem: Every multitape TM has an equivalent Single Tape TM.

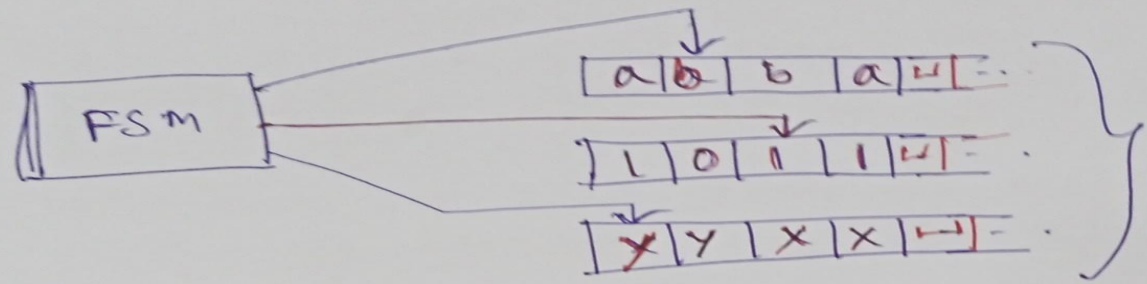
Proof: Given a multitape TM show how to build a single tape TM

→ Need to store all tapes on a single tape
Show the data representation

→ Each tape has a tape head.
Show how to store that info

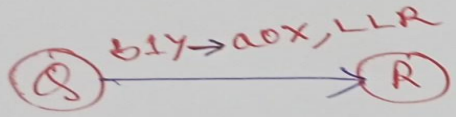
→ Need to transform a move in the Multitape TM into one or more moves in the Single Tape TM.

Multitape TM



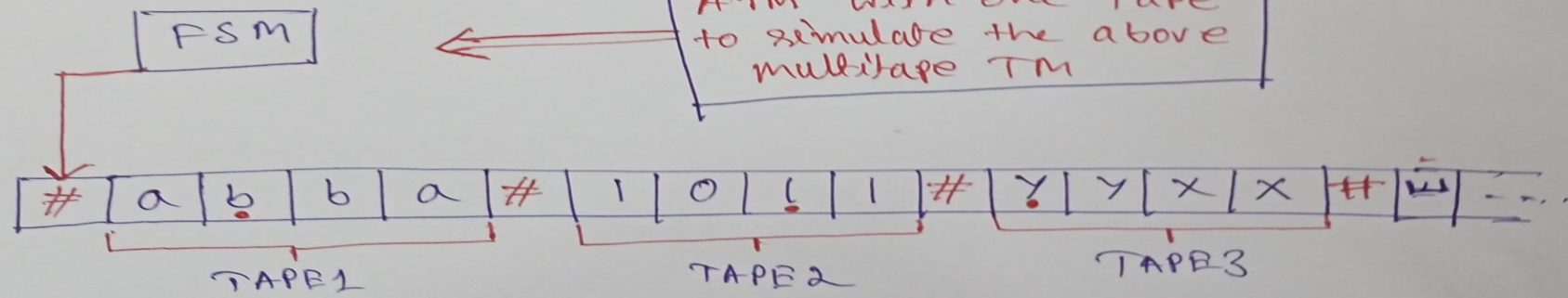
Several Tapes. Each has its own TAPE HEAD.

An Example with K = 3 Tapes TM.



Single TAPE TM

A TM with one Tape to simulate the above multitape TM



→ '#' is used to seperate the TAPE's.

→ Add "dots" to show where Head "K" is [Marking process]

→ To simulate a transition from state q, we must scan our Tape to see which symbols are under the K Tape heads.

Single Tape TM

→ Once we determine this and are ready to make the transition, we must scan across the tape again to update the cells (1) and move the dots (2).

→ Whenever one head moves off the right end, we must shift our tape so we can insert a ␣.

Decidability and Undecidability

Recursive Language:

→ A language L is said to be recursive if there exists a TM which will accept all the strings in 'L' and reject all the strings not in 'L'.

→ The TM will halt every time and give an answer (accepted or rejected) for each and every string input.

[Always HALT/STOP]

Recursively Enumerable Language:

→ A language 'L' is said to be a recursively enumerable language if there exists a TM which will accept (and therefore HALT) for all the input strings which are in 'L'.

→ But may or may not HALT/STOP for all input strings which are not in 'L'.

Decidability and Undecidability

Decidable Language:

A language 'L' is decidable if it is a recursive language and vice-versa.

Partially Decidable Language:

A language 'L' is partially decidable if 'L' is a recursively enumerable language.

Undecidable language →

→ If a language is not partially decidable then that language is undecidable

→ A language is undecidable if it is not decidable.

→ NO TM for undecidable languages.

Summary:

Recursive Language	TM always HALT/STOP
Recursively enumerable Languages	TM may or may not HALT.
Decidable Languages	Recursive language
Partially Decidable Language	Recursively Enumerable language
UNDECIDABLE	NO TM for that Language.

TM halting problem, Linear Bounded Automata, FA simulator & Parsers

Halting Problem

Problem.

⇒ Given a Program, WILL IT HALT ? [IN GENERAL]

Question ? ⇒ Can we have an algorithm that will tell you ~~what~~ whether a given program halt or not ?

⇒ Given a TM, will it halt when run on some particular given input?

⇒ Reframed

Given some program written in some language (Java/C etc.) will it ever get into an infinite loop or will it always terminate?

Answer → we can not design a TM or program (Algorithm) that will tell you ~~whether~~ whether a given TM or Program will not get into an infinite loop

1. In General we can't always know.
2. The best we can do is run the program and see whether it halts.
3. For many programs we can see that it will always halt or some times loop.

BUT FOR PROGRAMS IN GENERAL THE QUESTION IS UNDECIDABLE.

Linear Bounded Automat (LBA)

Order of power (Greater to Lesser)

TM > LBA > PDA > FA

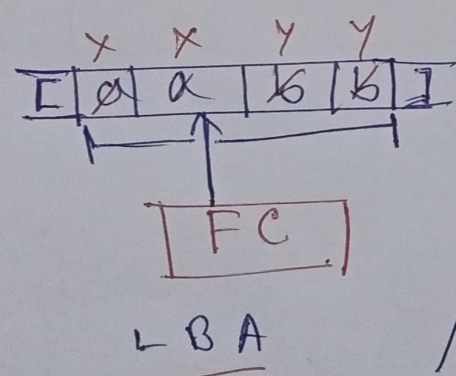
Context Sensitive Language

Formal Definition →

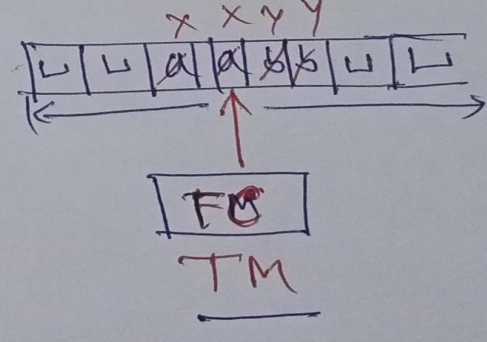
$(Q, \Sigma, q_0, F, \Gamma, M, \delta)$

- Q → Set of states
- Σ → Input Alphabets
- q_0 → Initial State
- F → Set of final states
- Γ → Tape Symbols
- M → Left marker / Right marker
- δ → Transition function

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$



LBA

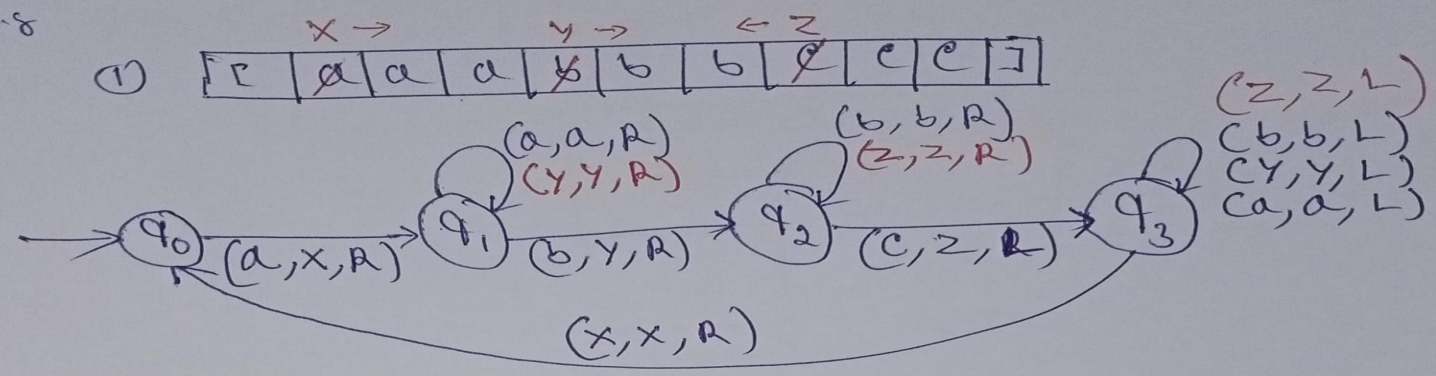


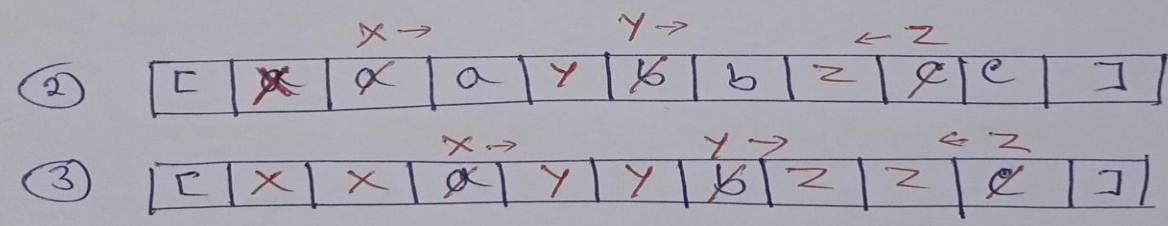
TM

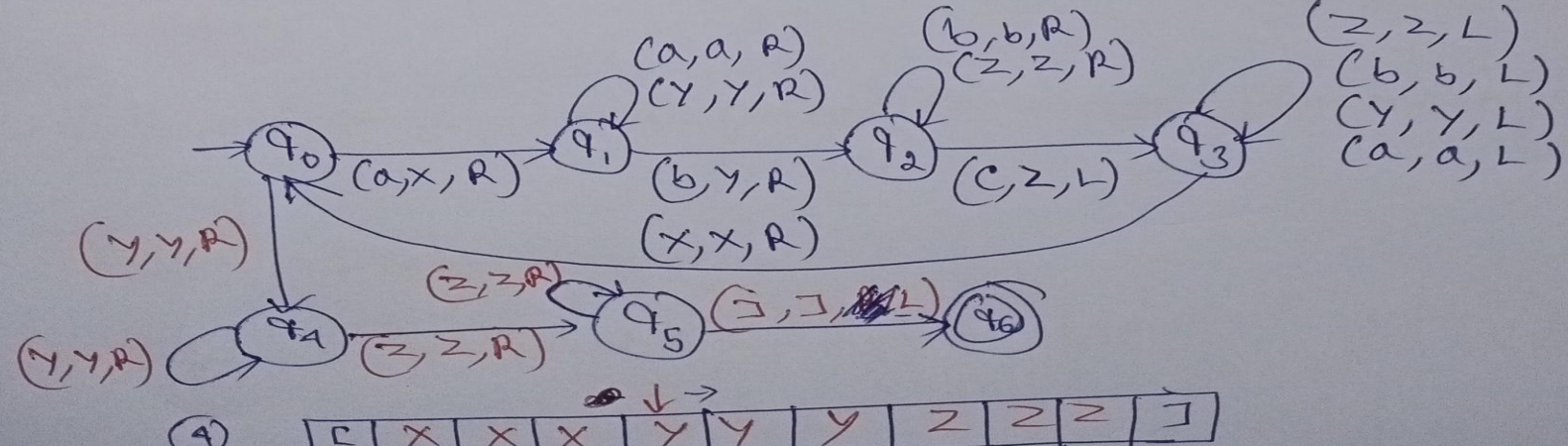
PDA (Example)

Construct LBA for $L = \{a^n b^n c^n \mid n \geq 1\}$

e.g

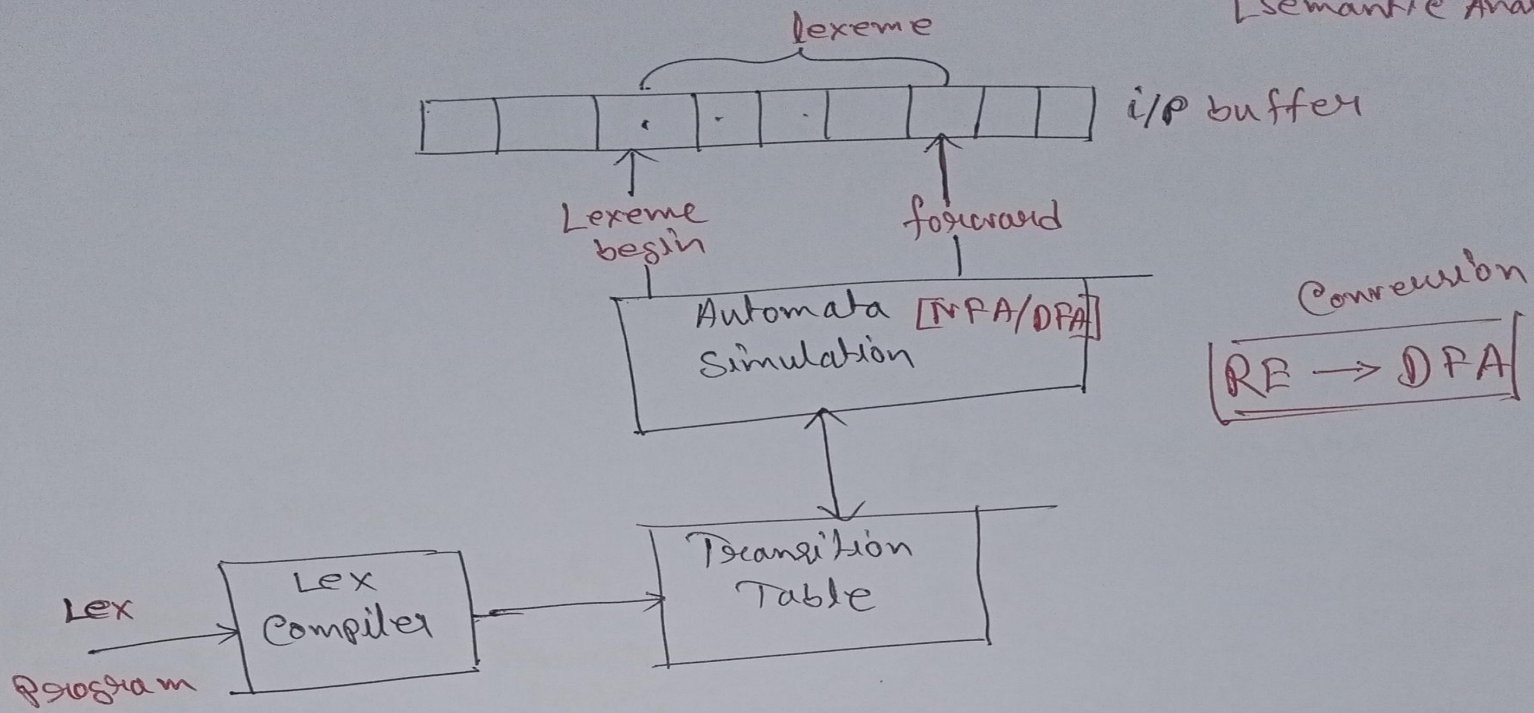






Lexical Analyzer Generator - FA simulation

[Semantic Analysis]

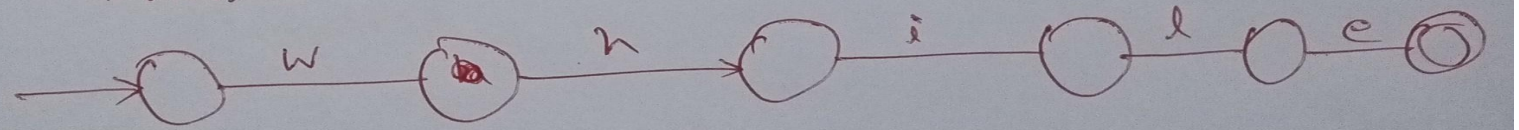


-> Lex program contains transition rules. [RE → FA]

-> It matches a input string with any of a pattern/token.

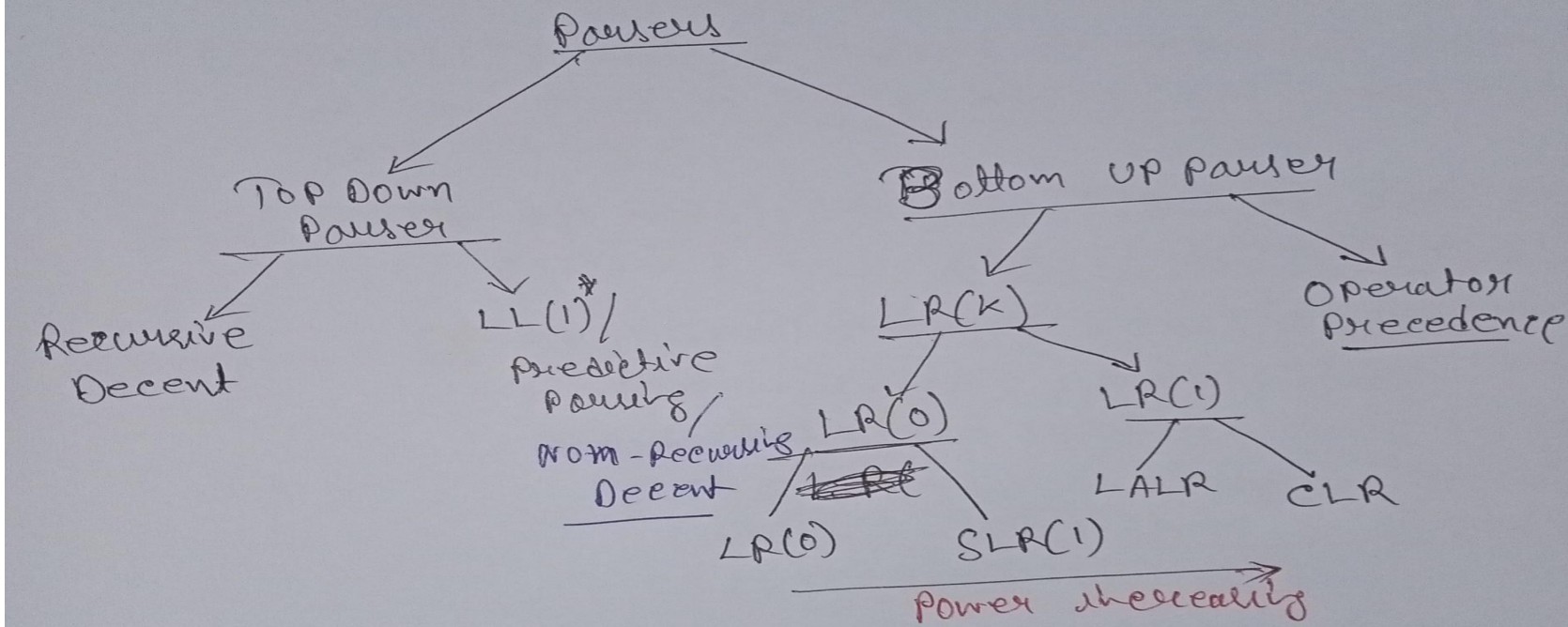
-> If matches then output considered. [Stream of Tokens]

e.g. -> while



Parsers [Syntax Analysis]

Parsing → It is a process of deriving string from a given grammar. [tokens are follows the grammar or not] [CFG]



* LL(1)
- L → Left to Right scan of input string
- L → Left most Derivation
- 1 → Lookahead

Finding FIRST() and FOLLOW()

FIRST() [S → ABC : FIRST(S) = FIRST(ABC)]

Rules:

1. FIRST (Terminal) = Terminal
2. FIRST (ϵ) = ϵ
3. (a) FIRST (Variable) → Contains all terminals present in first place of every string [VUT] derived by that variable.
   - (b) If FIRST(ABC), if FIRST(A) does not contain ϵ then FIRST(ABC) = FIRST(A).
   - (c) In FIRST(ABC), if FIRST(A) contains ϵ then FIRST(ABC) = FIRST(BC), and Repeat it for each variables in FIRST(ABC)
     - I) If Right most variable in FIRST(ABC) Contains ϵ the include ϵ in the FIRST (variable).
     - II) Other wise exclude ϵ in the FIRST (variable).

---

e.g.

S → ABCDE {a, b, c}
A → a/ϵ {a, ϵ}
B → b/ϵ {b, ϵ}
C → c {c}
D → d/ϵ {d, ϵ}
E → e/ϵ {e, ϵ}

S → ABC | shi | jkl {a, s, j}
A → a | b | c {a, b, c}
B → b {b}
D → d {d}

S → ABCDE {a, b, c, d, e, ϵ}
A → a | b | ϵ {a, b, ϵ}
B → c | d | ϵ {c, d, ϵ}
C → e | f | ϵ {e, f, ϵ}

Follow( )

S → ABC : Follow(A), Follow(B), Follow(C)

Rules

① Follow(variable) contains set of all terminals present immediately in right of variable.

② Follow of start symbol is $

③ If Follow(variable) contains a variable at the immediately in right. then Follow(variable) = First (immediate in right variable)

- ① If follow(variable) ~~first(ϵ), then~~ followed by ϵ then, follow(variable) = follow(Left hand side variable of the arrow symbol)

e.g. S → ABCϵ

④ Follow never contain 'ϵ'

e.g S → Eϵ then Follow(E) = Follow(S)

S → A B C D E ϵ ϵ ϵ ϵ ϵ {$}
A → a/ϵ {b,c}
B → b/ϵ {c}
C → c {d,e,$}
~~D → ϵ~~ {e,$}
D → d/ϵ {$}
E → e/ϵ

S → ABC|shi|jkl {$}
A → a|B|C {b}
B → b {c}
D → d

S → A B C ϵ ϵ ϵ {$}
A → a|b|ϵ {c,d,e,f,$}
B → c|d|ϵ {e,f,$}
C → e|f|ϵ {$}

Recursive descent Parser

$$E \rightarrow iE'$$
$$E' \rightarrow +iE'/\epsilon$$

```
E()
{
1. if(l=='i')
   {
2.   match('i');
3.   E'();
   }
4. }
```

```
E'()
{
1. if(l=='+')
   {
2.   match('+')
3.   match('i')
4.   E'();
   }
5. else
6.   return;
}
```

```
main()
{
1. E();
2. if(l=='$')
3.   Printf("Parsing Success")
}
```

match the symbol

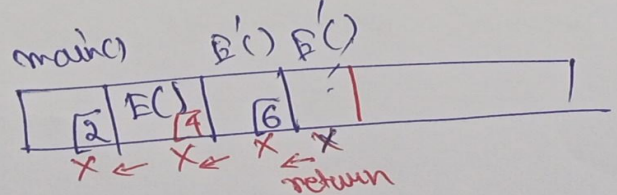
```
match(char c)
{
  if(l==c)
    l = getchar();
  else
    printf("error");
}
```

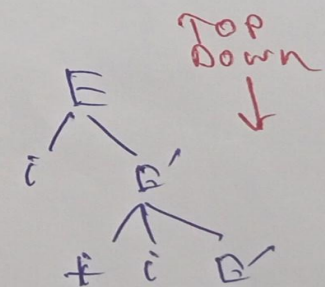
eg- i + i $

Lookahead

l = i

Recursion stack →





LL(1) grammar - Example

① $S \rightarrow (L) \mid a$

② $L \rightarrow SL'$

③ $L' \rightarrow \epsilon / , SL'$

	First()	Follow
①	{(, a} First(S)	{$,), ,} Follow(S)
②	{(, a} First(L)	{)} Follow(L)
③	{, , ϵ} First(L')	{)} Follow(L')

Parse Table

	(	)	a	,	$
S	1		2		
L	3		3		
L'		4		5	

LL(1) ✓

$S \rightarrow (L)$ — ①
$S \rightarrow a$ — ②
$L \rightarrow SL'$ — ③
$L' \rightarrow \epsilon$ — ④
$L' \rightarrow ,SL'$ — ⑤

follow of $L' \rightarrow \epsilon$

If parse table contains more than one entry within a cell position, then the Given grammar is not LL(1)

LL(1) or Not

$S \rightarrow aSbS / bSaS / \epsilon$

First(S) = {a, b, ϵ}

Follow(S) = {b, a, $}

Parsing Table

	a	b	$
S	1/3	2/3	3

Not a LL(1) Grammar

$S \rightarrow aSbS$ — (1)

$S \rightarrow bSaS$ — (2)

$S \rightarrow \epsilon$ — (3)

Follow(S)