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

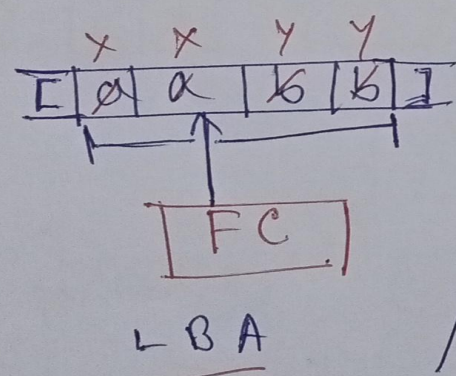
Context Sensitive Language

Formal Definition →

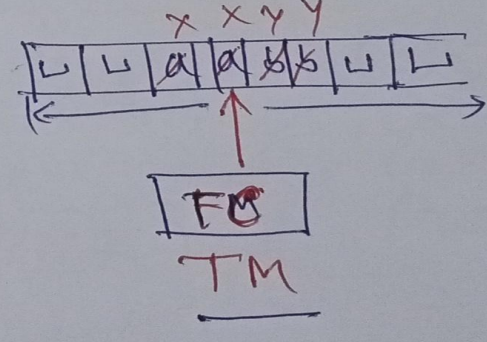
$(Q, \Sigma, q_0, F, \Gamma, M, \delta)$

- 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

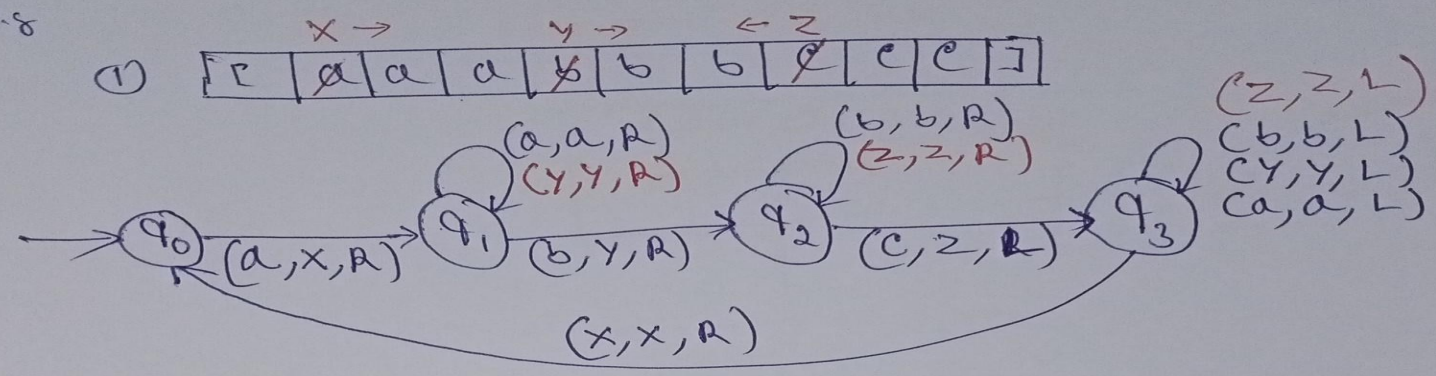


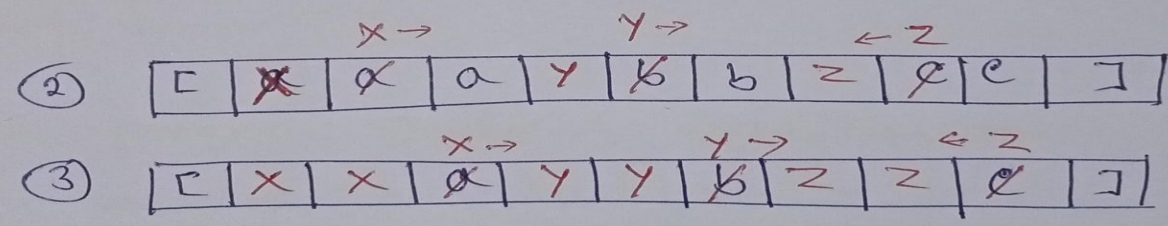
TM

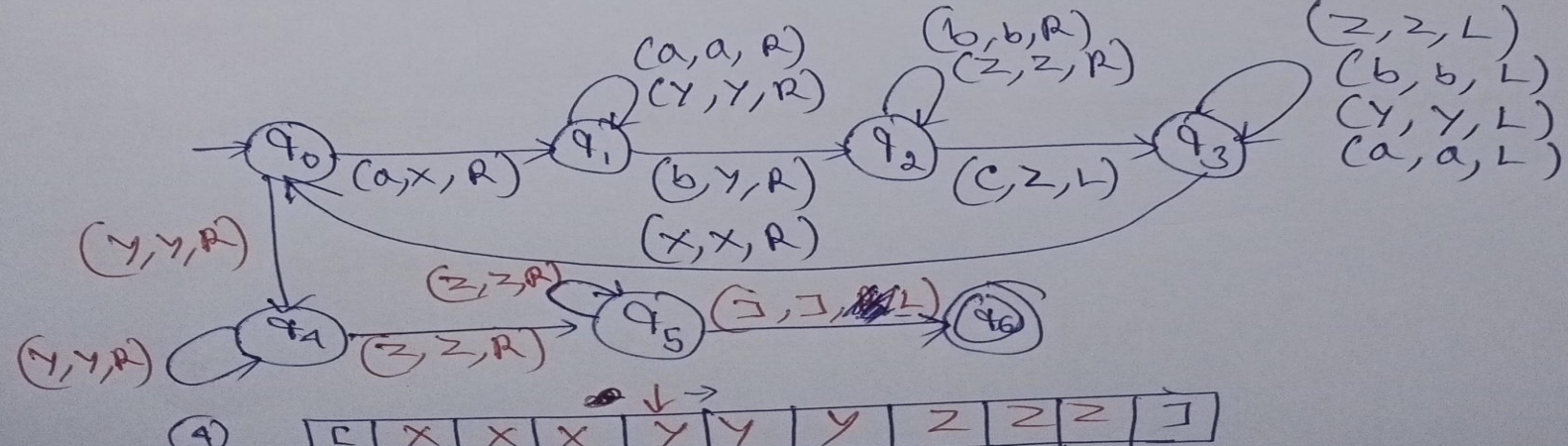
PDA (Example)

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

e.g

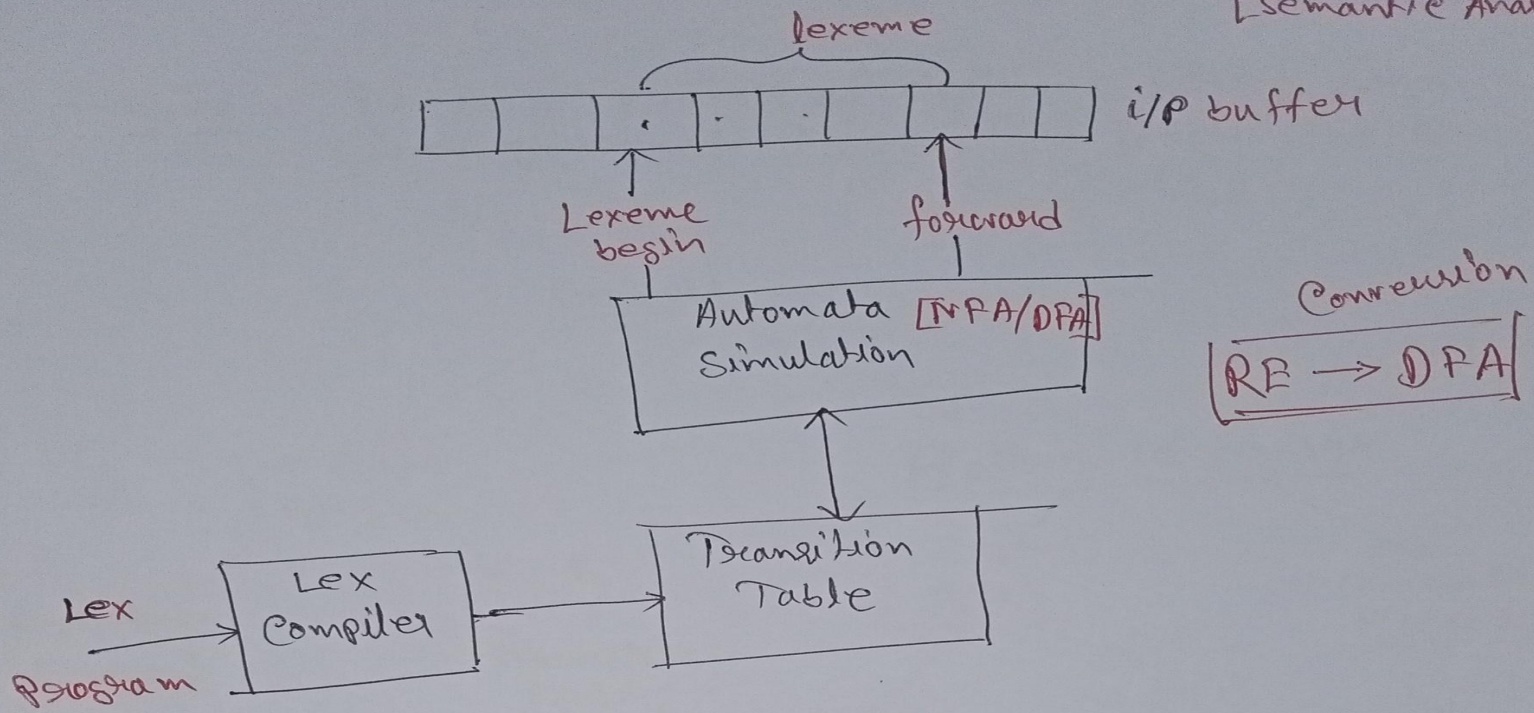






Lexical Analyzer Generator - FA simulation

[Semantic Analysis]

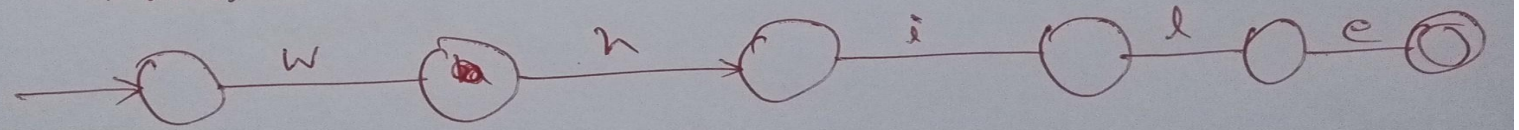


-> Lex program contains transition rules. [RE → FA]

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

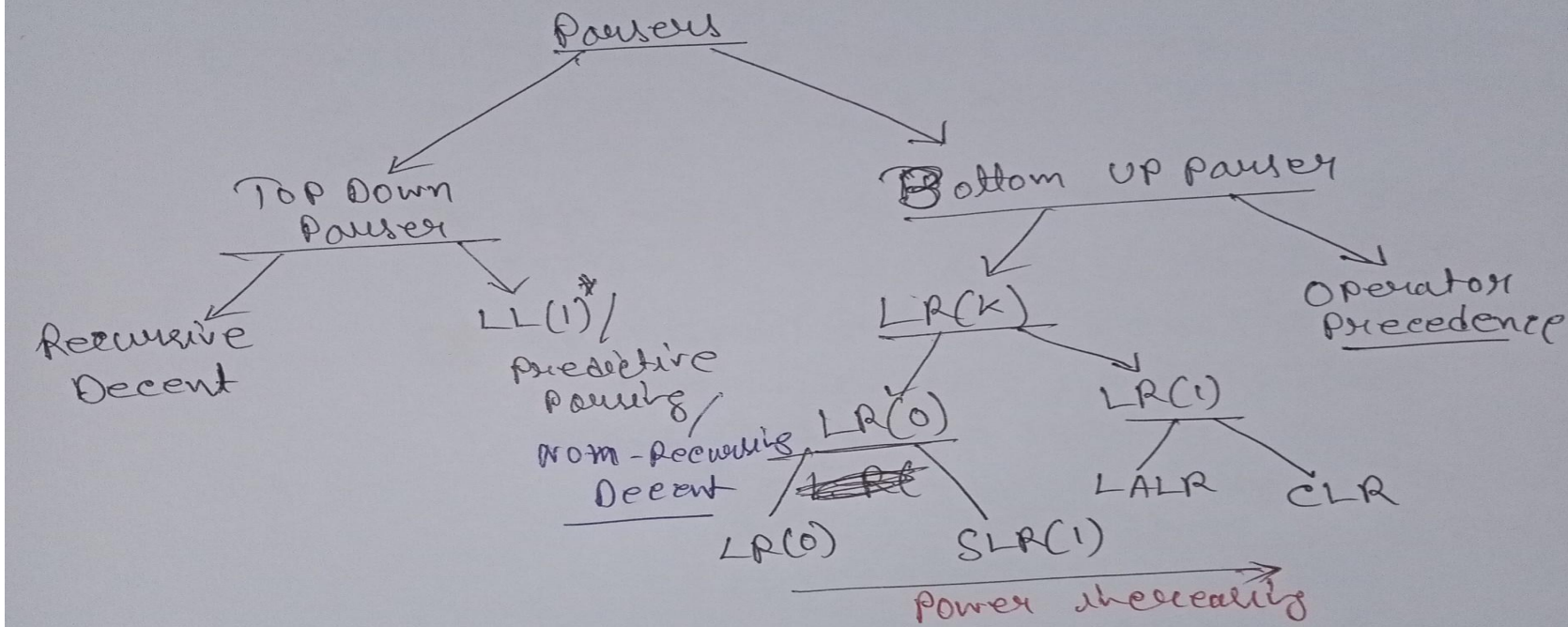
-> If matches then output considered. [Stream of Tokens]

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

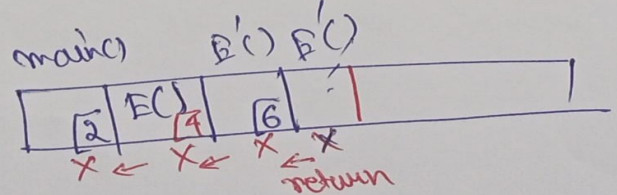
```
match(char c)
{
  if(l==c)
    l = getchar();
  else
    printf("error");
}
```

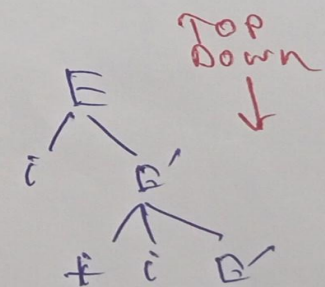
eg- i + i $

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)