Unit-I

Formal language & Regular expression:-

Languages:-

→ A Source pgm is divided into modules stored in separate files.

→ The preprocessors collects the source pgm, expands shorthands called source pgm statements.

→ The target pgm from complier is, then translated into assembly code & is then translated using assembler into machine code.

→ A linker in turn, links the library routines with machine code & is then into primary using loader.

Source pgm with assembler directing

↓

Preprocessors

↓

Source pgm.

↓

Complier

↓

target assembly pgm

↓

Assembly

↓

Relocate machine code.

↓

Loader/Linker ← Links Library Relocate obj code.

↓

Absolute machine code.

5/8

Complier:-

→ A complier is a pgm that reads a pgm written in one language. The source language into equivalent program in another language the target pgm.

Soue pgm → complier → Target pgm.

↓

Error msg.

→ The source pgm may be developed using any programming lang such as C, C#, Fortran.

→ The target pgm may be another Programming language or the machine lang of any computers.

Interpreters:-

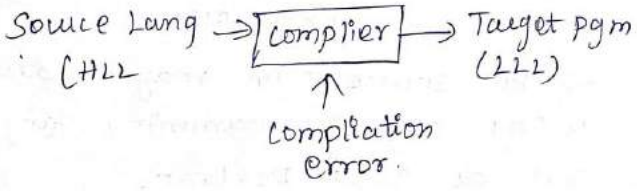
→ Its also a translator which converts the source pgm into target pgm.

→ Interpreter is a pgm in which the code is directly interpreted by the microcode resides in the control memory of a machine and executes the code line by line.

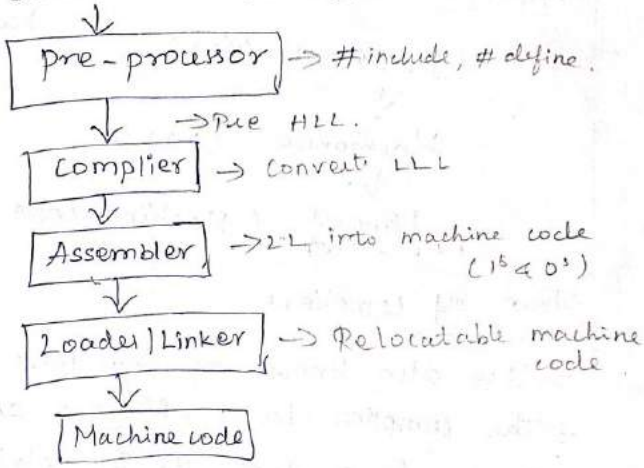
→ This microcode generates the control signals for executions.

Complier design:-

→ Complier is a s/w which converts a program written in HLL into LLL.



Language processing s/y:-



Ex: C-program {HLL}.

→ C complier translate pgm into LLL (Assembly lang)

→ Assembly translate LLL into machine code.

→ Linker used to link all parts of pgm together for execution.

→ Loader loads all them into memory & then the pgm is executed.

→ Complier converts HLL to LLL
→ Assembler converts LLL to machine Level Language.

Statement (HLL)
↓
Mnemonics (LLL)
↓
Binary (Machine code or MLL).

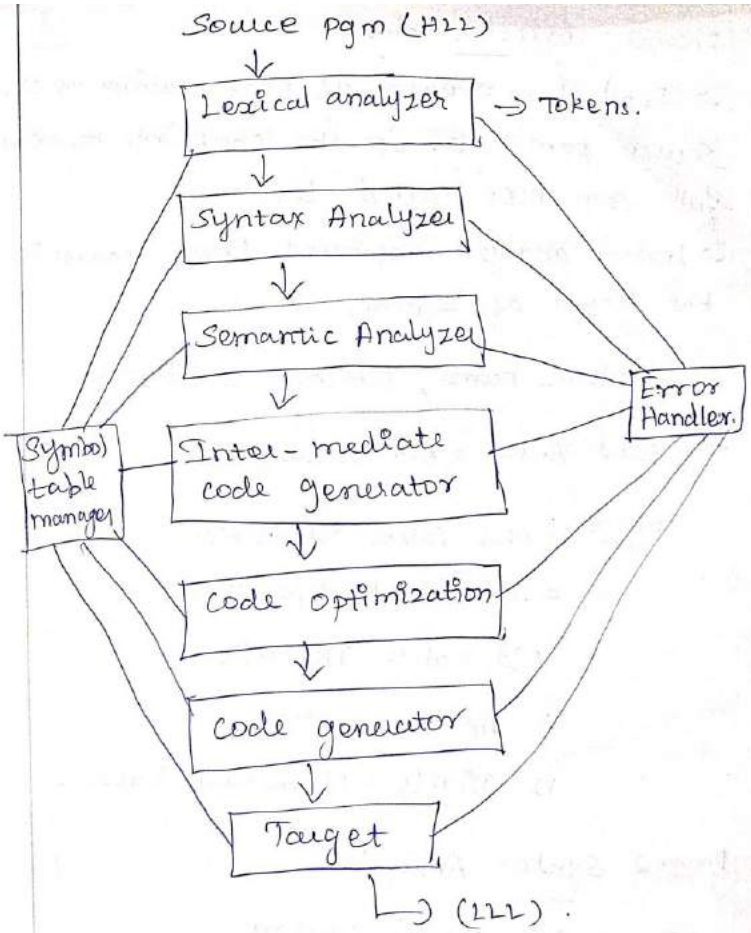
Phase of Compliers:

→ Its also known as Structures compliers.
→ The complier have different phase.
→ phases is a logically interelated operations that takes some pgm in one representation & produce o/p in another representation.

2 - major phase

① Analysis phase → M/c Independent & Lang dependent.

② Synthesis phase → M/c dependent & Lang Independent.



Phase 1 : Lexical analyzer:-

→ Read the stream of char making up the source pgm & group the char into meaning full sequences called lexeme.

→ Lexical analyzer represents these lexems in the form of tokens.

< token_name, attribute_value >.

Eg: new value := old value : +12.

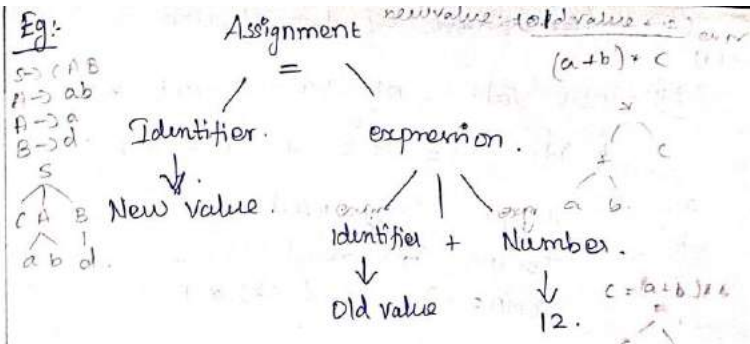
Tokens: → new value identifier
= Assignment Operators
Old value identifier.
+ Add operator.
12 Digits / Numerical Values.

Phase 2 Syntax Analysis:-

→ Its also called parsing.

→ It takes token produced by lexical as I/P & generates a parse tree or Syntax Analyzer.

Eg:



Phase 3 Semantic Analyzer:-

→ It checks whether the parse tree constructed followed the rules of lang. or not

→ Semantics works just check the Syntax

Eg: Sum = a+b.
int a;
double Sum;
Char b;

→ Syntax is correct

Data type mismatch.

Semantic record.
a : int.
Sum : double
b : char.

→ semantically not correct.

Phase 4: Intermediate code generation:-

→ Its the representation of final m/c lang code is produced.

→ This phase bridges the analysis and

Synthesis phase of translation.

Eg: new val := old val + fact *1

id1 := id2 + id3 *1

Temp1 = int read (1)

Temp2 = int read (2).

Temp3 = int read (3). *1

Temp3 = id2 + Temp2;

id1 = Temp3

Phase 5 : Code optimization:

The o/p runs faster & takes less space.

Temp1 = id3 * 1

id1 = id2 + Temp1

Phase 6: Code generation:

Translate the intermediate code into sequence of relocatable in a machine code.

Eg: id1 := id2 + id3 * 1

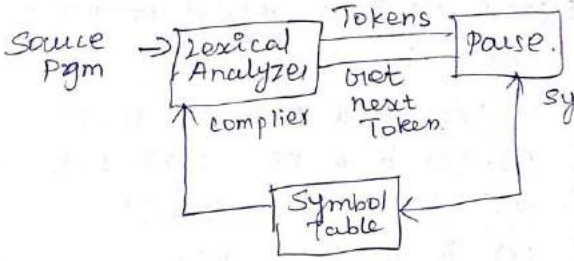
MOV R1, Id3
MUL R1, #1
} Assembly lang (LLL)

Complier converts HLL to LLL.

Lexical Analyzer:

→ Its the first phase of complier also called as linear or scanning.

→ In this phase of stream of characters making up the pgm is read from left to right & grouped into tokens that are seq of characters having collective machine.



Language:

→ A set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet is called language.

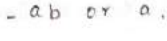
Eg: Lang over the alphabet $\Sigma = \{a, b\}$ that consists of all strings ending with abb.

$L = \{abb, aabb, babb, ababb, \ldots\}$.

R.E:

Union - a+b (or) a|b

Concatination - ab or a.b



Closure - Kleen closure - a^* = {ε, a, aa ...}

Positive " - a^+ = {a, aa, aaa, ...}

Regular Expression:

→ A regular Expression is built out of simpler R.E using a set of definite rules

→ Each rule (R.E) r denotes a language L(r).

1) ε is a regular expression denotes the lang {ε}
2) If 'a' is a symbol in Σ, then a is the regular expression denotes the lang {a}.
3) Suppose r & s are regular expression denoting the lang L(r) & L(s).
   a) (r) | (s) is a R.E L(r) ∪ L(s). Union
   b) (r). (s) is a RE L(r). L(s) concatenation
   c) (r)* is a RE (L(r))* kleen closure
   d) (r) is a RE L(r).

The precedence & associativity of operators:

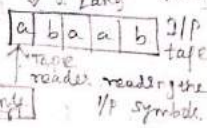
1) Unary operator (*) has highest precedence & is left associative
2) Concatenation operator has 2nd highest precedence & its left associ
3) | has the lowest precedence & its left associatives.

Finite automata or automata:

NFA
DFA

A finite automata has (5-tuples)

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



when Q = no. of Finite set of states

Σ = no. of Finite set of symbols called I/P alphabet

δ : Q × Σ → Q = Transition function (or) Mapping fun

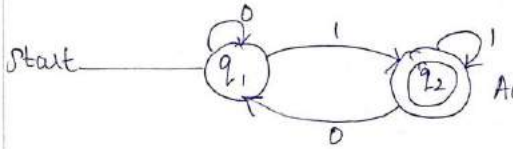
$q_0 \in Q$ = Initial state. (or) Start State

$F \subseteq Q$ = Set of final State.

Transition Diagram:

→ A transition diagram is a directed graph associated with the vertices of the graph corresponds to the states of finite automata.

→ The transition from State 'p' to state 'q'



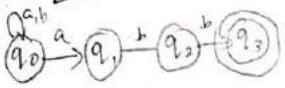
Language Acceptance by Finite automata:

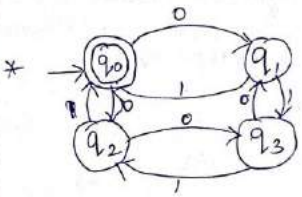
Eg: Given $M = (Q, \Sigma, \delta, q_0, F)$.

$Q = \{q_0, q_1, q_2, q_3\}$. [abb].

$\Sigma = \{0, 1\}$

$F = \{q_0\}$.





String i/p 110101 is a accepted by FA or not.

Sol:

	0	1
→ *q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_1	q_2

~~i/p strings~~.

~~q_0~~

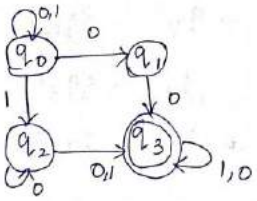
Finite automata can be classified as,

(1) NFA - Non-deterministic Finite automata
(2) DFA - Deterministic Finite automata.

NFA:

Ex1: Design NFA function, with using Transition Table.

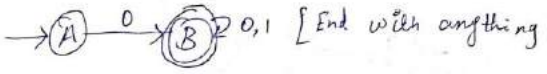
State	0	1
→ *q_0	q_0 q_1	q_0, q_2
→ q_1	q_3	∈
→ q_2	q_2 q_3	q_3
→ *q_3	q_3	q_3.



Ex2: Construct NFA, L = { Set of all strings that start with 0 }

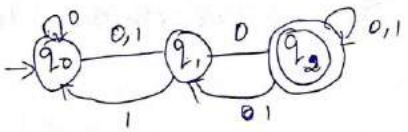
i/p $\Sigma = \{0, 1\}$. Min length = 1.

$L = \{0, 01, 001, 0001, 0101 \ldots\ldots\}$



Ex3: $Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$ $q_0 = \{q_0\}$ $F = \{q_2\}$.

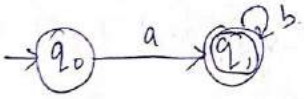


States	0	1
→ q_0	$\{q_0, q_3\}$	$\{q_1\}$
q_1	$\{q_3\}$	$\{q_0\}$
* q_2	$\{q_2\}$	$\{q_2, q_3\}$

Ex4: Construct NFA where language is given as strat with 'a'.

$\Sigma = \{a, b\}$.

Sol: $L = \{a, ab, aa, aab, aab\ abab, \ldots\}$.



NFA: many paths for specific I/p.

→ The FA are called NFA, when there exist many paths for specific I/p from the current state to the next state.

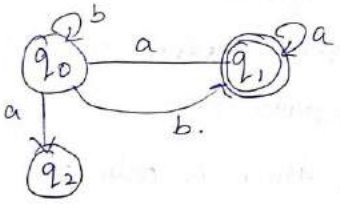
→ Its easy to construct NFA than DFA for a given RL.

→ Every NFA is not DFA, but each NFA can be translated into DFA.

→ NFA is defined in the same way as DFA but with two exceptions.

- It contains multiple next state.
- It contains ∈ transitions.

Eg:



Formal definition of NFA:

→ NFA also have five states same as DFA, but with different transistion function.

$\delta : Q \times \Sigma \rightarrow 2^Q$

where, Q : Finite set of states
Σ : Finite set of I/P symbol.
q_0 : Initial State
F : Final state
δ : Transition function.

DFA:

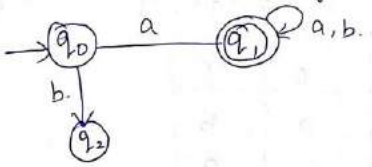
→ The finite automata are called deterministic finite automata if the m/c is read an I/P string one symbol at a time.

→ Deterministic refers to the uniqueness of the computation.

→ In DFA, there is only one path for specific I/P from the current state to the next state.

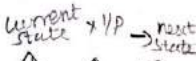
→ DFA does not accept the null move, is DFA cannot change state any I/P characters.

→ DFA can contain multiple final states, its used in lexical analysis in complier.



Formal definition of DFA:

Q : Finite set of state.
Σ : Finite set of I/P symbol
q_0 : initial state
F : Final "
δ : Transition Function → $\delta : Q \times \Sigma \rightarrow Q$.



Accepting Language:

Eg: $L = \{\phi\}$ ⇒ (q_0).

$L = \{\Sigma\}$ ⇒ (q_0)

DFA to accept 'a' ⇒ $(q_0) \xrightarrow{a} ((q_0))$

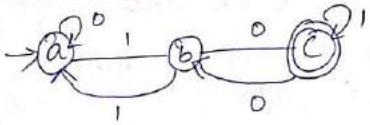
DFA to accept zero & more 'a' → $((q_0))$ a.

$L = \{\epsilon, a, aa, aaa, \ldots\}$.

→ only one path

Ex1: Construct DFA. $Q = \{a, b, c\}$ $q_0 = \{a\}$ $\Sigma = \{0, 1\}$ $F = \{c\}$

Present State	0	1
→ a	a	b
b	c	a
* c	b	c.



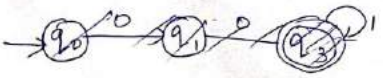
Ex2:
Design DFA with $\Sigma = \{0, 1\}$ accepts those string which starts with 1 & ends with 0.

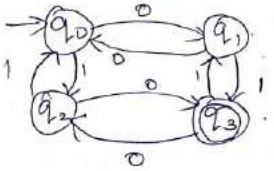
$L = \{1, 10, \ldots\} \Rightarrow \{10, 100, 1010, 111110 \ldots\}$

10 accepted
100 accepted

Ex3: Design FA with $\Sigma = \{0, 1\}$ accepts even no. of 0's & Even no. of 1's.

$L = \{001, 110, 100 \ldots\}$

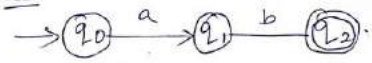




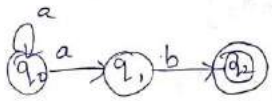
Ex4: Construct DFA accepting all strings over $\{a, b\}$ ending with 'ab'?

$L = \{ab, aab, bab, bbab, aaab \ldots\}$

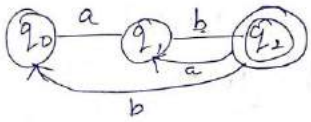
ab:



aab

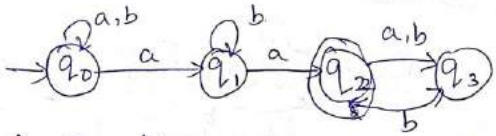


bab



Conversion of NFA to DFA:-

Ex1:



Step 1: Constructing Transition Table.

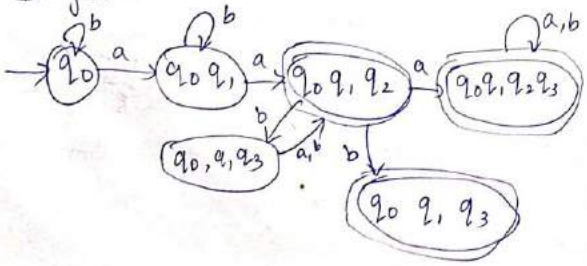
NFA Transition Table:-

δ	a	b
→ q_0	$\{q_0 q_1\}$	q_0
q_1	q_2	q_1
* q_2	q_3	q_3
* q_3	-	q_2

DFA Table:-

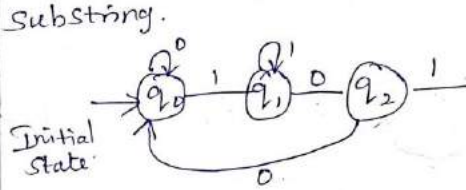
δ	a	b
→ q_0	$[q_0, q_1]$	q_0
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_0 q_1]$
$[q_0 q_1 q_2]$	$[q_0 q_1 q_2 q_3]$	$[q_0 q_1 q_3]$
$[q_0 q_1 q_2 q_3]$	$[q_0 q_1 q_2 q_3]$	$[q_0 q_1 q_3 q_2]$
$[q_0 q_1 q_3]$	$[q_0 q_1 q_2 q_3]$	$[q_0 q_1 q_2]$

DFA Diagram.



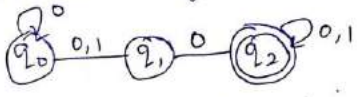
Finite automata for DFA. Grate Question:-

Design a DFA that accepts '101' as substring.



101
1101
10101
101
1101
10101
10110

Ex2:- conversion of NFA to DFA.

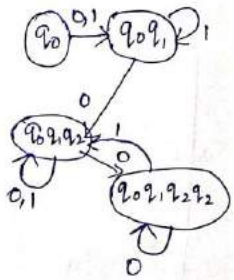


Transition Table for NFA.

δ	0	1
→ q_0	$\{q_0, q_1\}$	$[q_0 q_1]$
q_1	q_2	-
* q_2	q_2	q_2

DFA Table.

δ	0	1
→ q_0	$[q_0, q_1]$	$[q_0 q_1]$
$[q_0 q_1]$	$[q_0, q_1, q_2]$	$[q_0 q_1]$
$[q_0 q_1 q_2]$	$[q_0 q_1 q_2 q_2]$	$[q_0 q_1 q_2]$
$[q_0 q_1 q_2 q_2]$	$[q_0 q_1 q_2 q_2]$	$[q_0 q_1 q_2]$



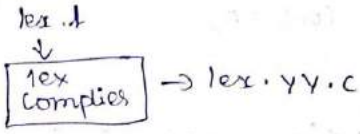
Application of finite automata to lexical Analysis:

LEX Tool:-

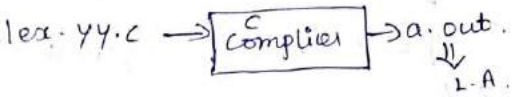
→ Its main Job is to beat I/P an I/P Stream into the token.

→ lex is a tool for automatically generating a lexer.

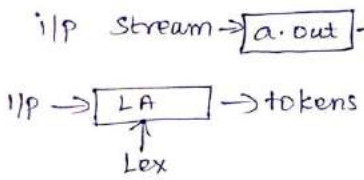
Step 1: lex same prog



Step 2:



Step 3:



Structure of lex program:-

{ declaration } → Declare the Variables.

%%

{ translation rules } → have the pattern { Action }.

%%

{ auxilary functions } => funs can be compiled separately.

Ex: lex program:

→ program to count the no. of vowels & constants in a given grammer.

I/P stream → Grammer

```
%{ #include <stdio.h>
   int vowels=0;
   int cons=0;
%}
%%
   [aeiou AEIOU] {vowels++;}
   [a-z A-Z]     {cons++;}
%%
int yywrap()
{ return1;
}
main() {
   Pf("Enter the string at End press ^d \n");
   yy lex();  → lex tool.
Pf("no. of vowels = %d \n
    no. of constants = %d \n",
    vowels, cons);
}
```

Unit-2

Context free Grammer

sentence formation by given the set of rules

CFG stands for context free Grammer. Its a formal grammer which is used to generate all possible patterns of string in a given finite Language (FL).

CFG can defined as 4 tuples.

(G = VTPS)

G - Grammer.

V - Non-Terminal / Uppercase.

T - Terminal / Lowercase

P - Set of production rules.

↳ non-terminal symbols (LS productio

↳ Terminal symbol (RS productio

S - Start symbol. used to derive the string.

Ex1: construct CFG for the language having any no. of a's over the set $\Sigma = \{a\}$.

Sol:

$G = \{V, T, P, S\}$.

Σ (I/P symbol) $= \{a\}$.

$L = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$

R.E = a^* derive i/p = "aaaaaa"

Production rule:-

$S \to aS$ —①

$S \to \in$ —②

S
as
aas
aaas
aaaas
aaaaas
aaaaas

Ex2: Construct a CFG for the lang $L = a^n b^{2n}$ where $n >= 1$.

n=1 n=2 n=3

L = { abb, aabbbb aaabbbbbb ..... }

The grammer could be,

$S \to asbb$.

$S \to abb$.

aaabbbbbb. to derive.

$S \to asbb$.

$S \to aasbbbb$. {$S \to asbb$}

$S \to aaabbbbbb$. {$S \to abb$}

$S \to aAB$
$A \to aA \mid a$
$B \to bB \mid b$

G = (V, T, P, S)

G = ({A, B, S}, {a, b}, P, S)

Derivations:

Derivation is a seq of production rules. It is used to get i/p strings. During paring we have to take two decision:

* we have to decide the non-terminal which is to be replaced.
* we have to decide the production rule by which the non-terminal will be replaced.

→ we have two options to be decide, which non terminal to be placed with production rule.

① left most derivation.

② Right most derivation

Left most derivation: [Left to Right].

→ In the left most derivation, the i/p. is scanned & replaced with the production rule from left to right.

→ so we have to read i/p string from left to right.

Ex1: Derive the string "abb" for left most & Right most derivation using CFG given by.

$S \rightarrow AB \mid \epsilon$
$A \rightarrow aB$
$B \rightarrow sb.$

Left most

S
AB
aB
asb B
ab B
ab sb
abb

$S \rightarrow AB.$
$A \rightarrow aB.$
$B \rightarrow sb.$
$S \rightarrow \epsilon.$
$B \rightarrow sb.$
$S \rightarrow \epsilon.$

Right most derivation

S
AB → S → AB.
Asb → B → sb.
Aεb S → ε.
aBb A → aB.
asbb B → sb.
abb S → ε.

Ex2: Derive string '00101' L.M.D & R.M.D.

$S \rightarrow A \mid B$
$A \rightarrow 0A \mid \epsilon$
$B \rightarrow 0B \mid 1B \mid \epsilon$

Left most derivation:

S	
A/B	(S→A/B)
0A/B	A→0A
00A/B	A→0A
00/B	A→ε
00 1 0B	B→0B.
00101B	(B→1B) 00101

Right most derivation

S	
A/B	(S→1B)
A10B	(S→0B)
A101B	(B→1B)
A1010	(B→ε)
0A101	(A→0A)
00A101	(A→0A)
00101	(A→ε)

Ex: String aabbabba L.M.D & R.M.D.

$S \rightarrow aB \mid bA$
$A \rightarrow a \mid aS \mid bAA$
$B \rightarrow b \mid bS \mid aBB.$

Sol:

L.M.D.

S	
aB	S→aB.
aaBB	B→aB.
aabB	B→b.
aabbS	B→bS.
aabbaB	S→aB.
aabbabS	B→bS.
aabbabbA	S→bA.
aabbabba	A→a.

R.M.D.

S	
aB	S→aB.
aaBB	B→aBB.
aaBbS	B→bS.
aaBbbA	S→bA.
aaBbba	A→a.
aabSbba	B→bS.
aabbAbba	S→bA.
aabbabba	A→a.

Parse tree:-

→ It a graphical representation for the derivation of the given production rules for a given CFG.

→ Its simple way to show how the derivation can be done to obtain some string from a given set of production rules.

→ Its also called as parse tree.

→ The parse tree follows the precedence of operators.
→ The deepest subtree traversed first
→ The operator in the parent node has less precedence over the operator in the subtree

Properties:

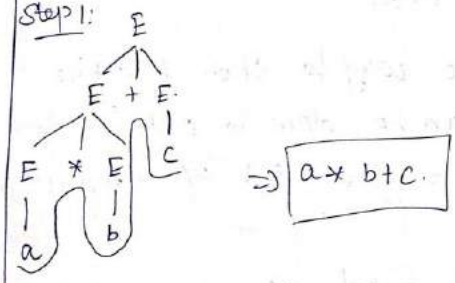
1. The root node is always a node indicating start symbols.
2. The derivation is read from left to Right.
3. The leaf node is

Ex: Production Rules:

E = E+E
E = E*E
E = a|b|c.

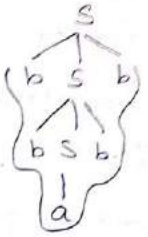
I/P:
a*b+c.

Step 1:



⇒ a * b+c.

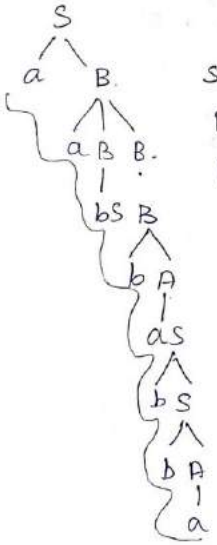
Ex1: parse tree for the string bbabb from the CFG given by, S → bSb|a|b



⇒ bbabb

Ex2: parse tree for the string aabbabba

S → aB | bA
A → a|aS|bAA.
B → b|bS|aBB.



S → aB.
B → aBB.
B → bS
B → bA.

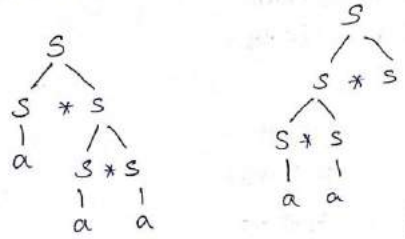
⇒ aabbabba//

Ambiguity LL(k) grammers:

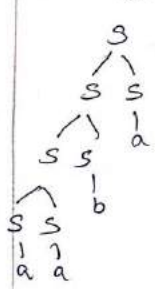
→ If two or more left derivation trees are possible is called ambiguity.

→ One or more than one tree can be generated for the same string.

Ex1: If G is the Grammer, S→S+S/S*S/a Show that G is ambiguous with the string w = a*a*a.



Ex2: P: S→SS/a/b.
w = aaba.



LL parser:

→ It accepts (LL) grammer. It is denoted as LL(K).

→ The first L → represents the I/p from left to right.

→ The second L → represents the left most derivation.

→ K represents the no. of look a heads (How many time you trag the list generally k=1

∴ LL(K) = LL(1).

→ A gramma G is LL(1), If there are two distinct production.

A→(α/β)

condition:

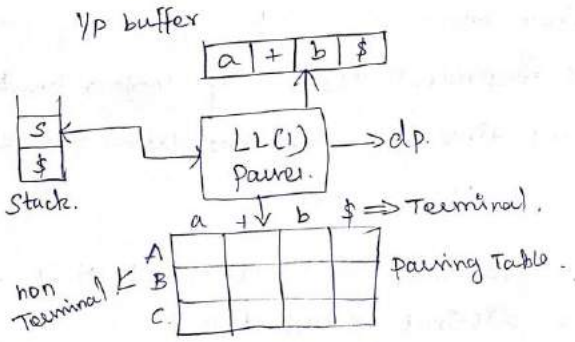
① For no teeminal α & β derive string beginning with a.

② At most one of α & β can derive empty string.

③ If β→ϵ, then 'a' does not derive any string begnning with a teeminal in Follow(A).

→ LL(1) use data structure are,
- i/p buffer
- stack
- parsing table.

Structure of LL(1):



Construction of predective LL(1)

① FIRST ( ) | Leading ( ).
FOLLOW( ) | Trailing ( ).

② Predective parsing table by using first & follow functions.

③ Parse the i/p string with the help of the table.

Ex1: construction of predective parse LL(1)

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \in$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \in$
$F \rightarrow (E) \mid id$

① First & Follow.
② Parsing Table.
③ Stack Implementation
④ Parse tree.

Step1: First function (→ Left side)

First (F) = {(, id}
First (T') = {*, ∈}
First (T) = {(, id}
First (E') = {+, ∈}
First (E) = {(, id}

Step 1: Follow function. (→ Right side)

Follow (E) = {$,)}
Follow (E') = {$,)}
Follow (T) = {(first (E') − ∈) ∪ Follow (E)}
= {+, $,)}
Follow (T') = {+, $,)}
follow (F) = {*, +,), $}

Step2: Whenever no ∈ need to go follow fun

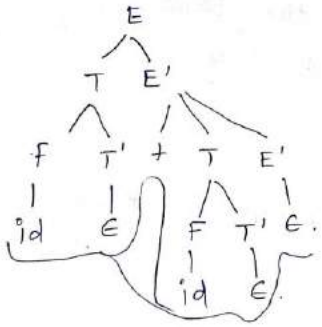
construct the parsing Table.

NT / T	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→∈	E'→$
T	T→FT'			T→FT'		
T'		T'→∈	T'→*FT'		T'→∈	T'→∈
F	F→id			F→(E)		

Step 3: Implements the Stack.

Stack	I/P	Action.
Symbol ← E $ (Start, Follow)	id+id $	E → TE'
TE'$	id+id $	T → FT'.
FT'E'$	id+id $	F → id.
id T'E'$	id+id $	pop id.
T'E'$	+id$	T' → ∈.
E'$	+id$	E' → +TE'
+TE'$	+id $	pop +.
TE'$	id $	T → FT'
FT'E'	id$	F → id
id T'E'	id$	pop id.
T'E'	$	T' → ∈
E'$	$	E' → ∈.
$	$	Accepted.

Step 4: parse tree:



⇒ id + id

Ex2: LL(1) parsing, the given grammar.

I/p string : abd $

~~S → A~~
~~A → aB / Ad~~
~~B → b~~
~~∈ → g.~~

S → A
A → aBA'
A' → dA' / ∈
B → b
C → g

Step 1:

First function:

First (S) = First (A) = {a}.
First (A) = {a}.
First (A') = {d, ∈}.
First (B) = {b}.
First (C) = {g}

Follow function.

Follow (S) = {$}.
Follow (A) = Follow (S) = {$}
Follow (A') = Follow (A) = {$}
Follow (B) = Follow {d, $}.
Follow (C) = NA.

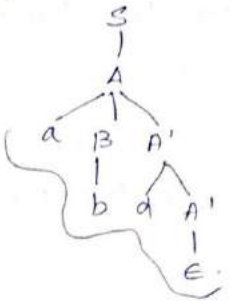
Step2:
Constructing the table.

	a	b	d	g	$
S	S→A.				
A	A→aBA'				
A'	~~A'→dA'/ε~~		A'→dA'		A'→ε
B		B→b			
C				C→g.	

Step3 Implementation of stack by using Parsing Table.

Stack	I/P	Action
S$	abd$	S→A
A$	abd$	A→aBA'
aBA'$	abd$	pop a
BA'$	bd$	B→b.
bA'$	bd$	pop b
A'$	d$	A'→dA'
dA'$	d$	pop d.
A'$	$	A'→ε.
$	$	Accept (The I/P is properly parsed)

Step-1: parse tree.



abdε //.

Bottom up parsing (Shift Reduce parsing):-

→ Shift Reduce parsing is a process of reducing a string to the start symbol of a grammer.

→ It uses of a Stack to hold the grammar & an I/p to hold the string.

A string $\xrightarrow{\text{Reduce}}$ the starting symbol.

→ Shift Reducing parsing perform two action.
① Shift ② Reduce.

→ Shift action, the current symbol in the I/p string is pushed on stack.

→ At each reduction, the symbol will be replaced by the non-terminals.

→ The symbol is the right side of production & non-terminal is the left side of the production.

Ex1:

Grammer

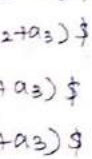
$S \rightarrow S+S$
$S \rightarrow S-S$
$S \rightarrow (S)$
$S \rightarrow a$

I/p string $a_1-(a_2+a_3)$

Stack	I/p string	Action
$	$a_1-(a_2+a_3)$	shift a_1
$ a_1	$-(a_2+a_3)$	reduce by $S \rightarrow a$
$ S	$-(a_2+a_3)$	Shift -
$ S-	(a_2+a_3)	Shift a_2
$ S-(	$a_2+a_3)$	reduced by $S \rightarrow a$
$ S-(a_2	$+a_3)$	Shift +
$ S-(S	$+a_3)$	Shift a_3
$ S-(S+a_3	$)$	reduced by $S \rightarrow S+S$
$ S-(S+S)	$	reduced by $S \rightarrow (S)$
$ S-(S)	$	reduced by $S \rightarrow S-S$
$ S-S	$	reduced $S \rightarrow S+S$
$	$	Accept parsing.

Ex2: $E \rightarrow 2E2$, $E \rightarrow 3E3$, $E \rightarrow 4$ I/p string 32423.

Stack	I/p string	Action
Empty stack $	32423$	shift 3.
$3	2423$	shift 2.
$32	423$	shift 4.
$324	23$	Reduce $E \rightarrow 4$.
→ $32E	23$	shift 2.
→ $32E2	3$	Reduce $E \rightarrow 2E2$
→ $3E	3$	shift 3.
→ $3E3	$	Reduce $E \rightarrow 3E3$.
→ $E	$	Accept the string.



Types of Parsing.

- Top down parse.
  - Backtracking
  - Predictive parse.
- Bottom - up parse
  - Operator precedence parse.
  - LR Parser
    - CLR, SLR, LA, LAL

Handle pruning:-

→ "Handle" is a string of substring that matches the right side of the production and we can reduce such string by a Non-terminal on left hand side production

→ "Handle" of right sentimental form α is production of β where the string β may be found & replaced by A to produce the previous right sentimental form in RMD of β.

$E \rightarrow E + E$
$E \rightarrow id$

id + id + id in RMD

$E \underset{rm}{\Rightarrow} E + E$
$E \underset{rm}{\Rightarrow} E + E + E$
~~$E \Rightarrow E + E + E + E$~~
~~$E \Rightarrow id + id + id$~~
$E \underset{rm}{\Rightarrow} E + E + id$
$E \underset{rm}{\Rightarrow} E + id + id$
$E \Rightarrow id + id + id$

Right sentimental form	Handle	production
id + id + id	id	E → id
E + id + id	id	E → id
E + E + id	id	E → id
E + E + E	E+E	E → E+E
E + E	E+E	E → E+E
E → Root		

LR Grammar paring:-

Various steps involved in paring

→ For the given i/p string write context free grammar.
→ check ambiguity of the grammar.
→ Add augment production in the given grammar.
→ Create canonical collection of LR(0) items
→ Draw a data flow diagram (DFA)
→ Construct LR(0) pausing table.