Unit -2

Types of grammar:

* Type 3
* Type 2
* Type 1
* Type 0

Type 3:

* It is a regular language which describes regular expression
* Grammar regular.
* It can be modelled by NFA or DFA.

eg: $a^* b^*$

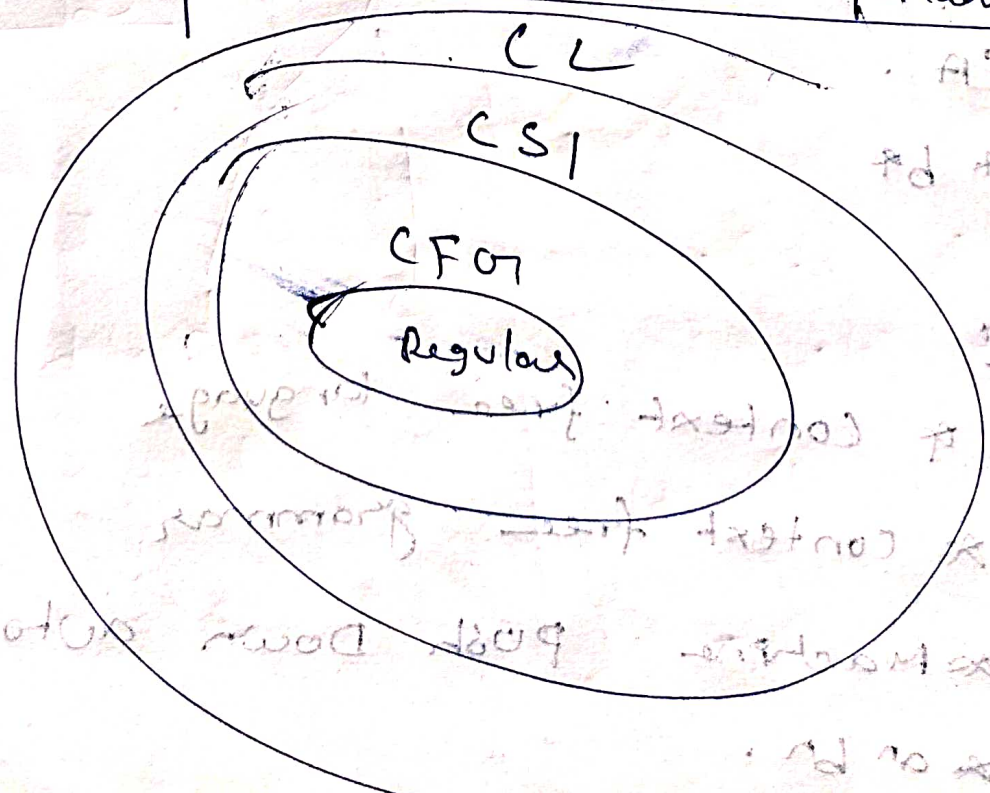
Type 2:

* Context free language
* Context free grammar
* Machine push Down automata
* $a^n b^n$.

Type 1

* Decidable language.
* content sensitive
* Linear bounded automata
* $a^n b^n c^n$.

Language class	language	Grammar	Machine	eg
Type 3	Regular	Regular grammar	NFA DFA	$a^n b$
Type 2	Context free	Context free grammar	PDA	$a^n b^n$
Type 1	Decidable language	Context sensitive	Linear bound auto	$a^n b^n$
Type 0	Computable lang	Unrestricted grammar	Turing Machine	$n!$



CFG.

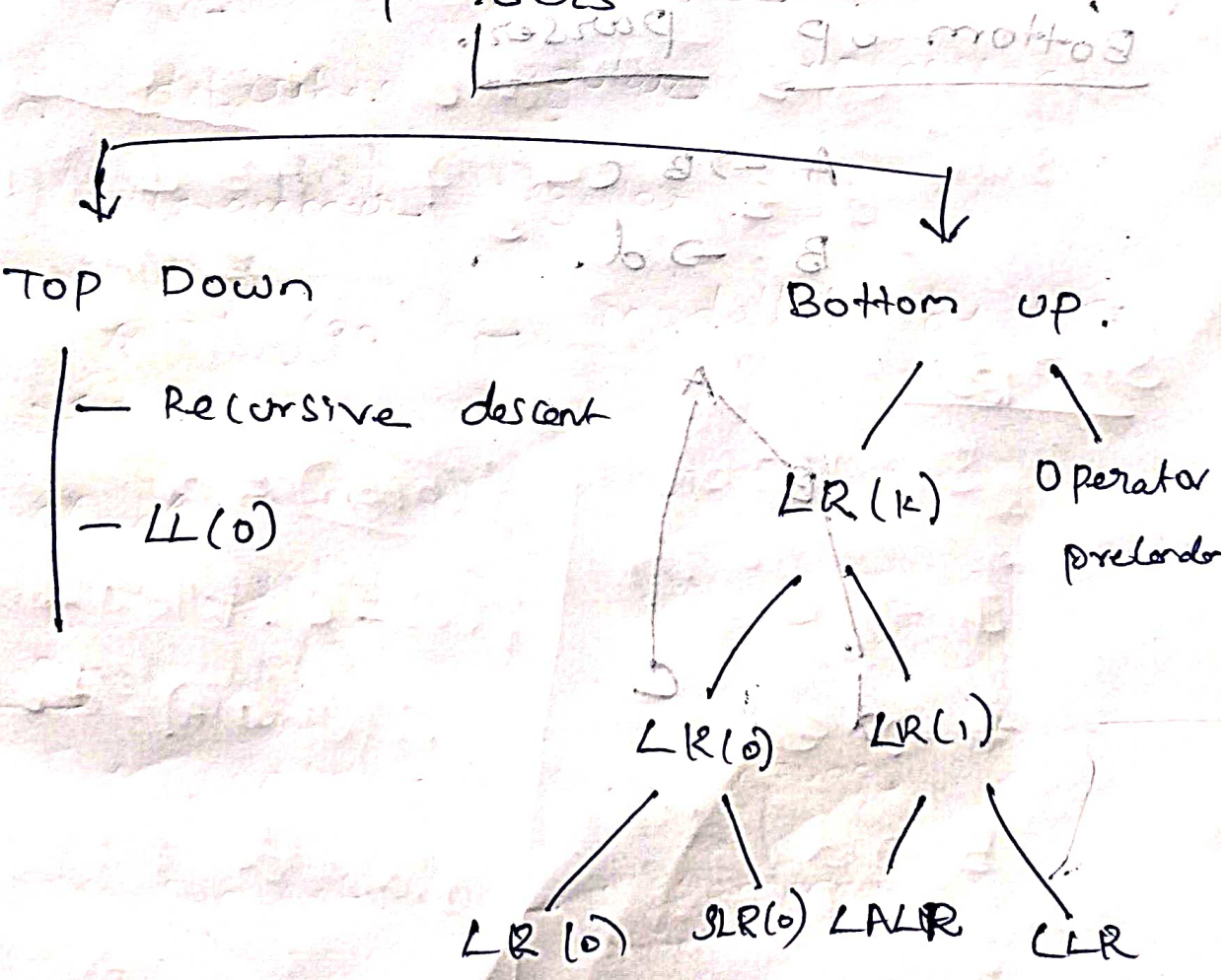
1) Convert CFL to CFG

$a^n b^n \quad n \geq 0$

$n = 3$

$a^3 b^3 = aaa\ bbb$

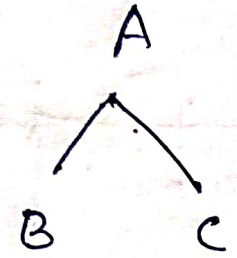
Parsers.



Top down parser.

$A \rightarrow BC$

$B \rightarrow d.$

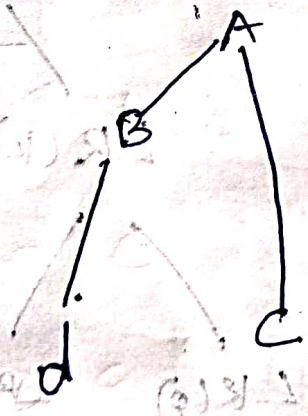


d.

Bottom up parser.

$A \rightarrow B\ C$

$B \rightarrow d.$



LL(1) Parser.

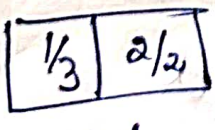
* It is also known as predictive parser

Rules:

* No left recursion
* CFG grammar is used
* No ambigous grammar
* The grammar should be left factorized if required

Steps:

1) find first and follow for every variable
2) construct parsing Table
3) If the cell has more than one entry, then the grammar is not LL(1)



wrong

correct

eg:

$S \rightarrow aABb$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow d \mid \varepsilon$

$S \rightarrow aABb$

$A \rightarrow c$

$A \rightarrow \varepsilon$

$B \rightarrow d$

$B \rightarrow \varepsilon$

Step 1:

First	Follow
a	$
c, ε	d, b
d, ε	b

Step 2:

$S \rightarrow aABb$

$A \rightarrow c$

$A \rightarrow \varepsilon$

$B \rightarrow d$

$B \rightarrow \varepsilon$

$f(aABb) = a$

$f(c) = c$ = follow(left hand side)

$\therefore$ follow(A) = d, b

$f(d) = d$

$f(B) = follow(B) = b$

Step 3:

	a	b	c	d	$
S	S→aABb				
A		A→ε	A→c	A→ε	
B		B→c		B→d	

LR parsers:

* It scans input from left to right
* LR parsers can parse a strictly larger class of grammar than top down grammar.
* LR parser can usually recognize all programming language
* LR parser detects errors quickly

eg:

A → BC

B → d

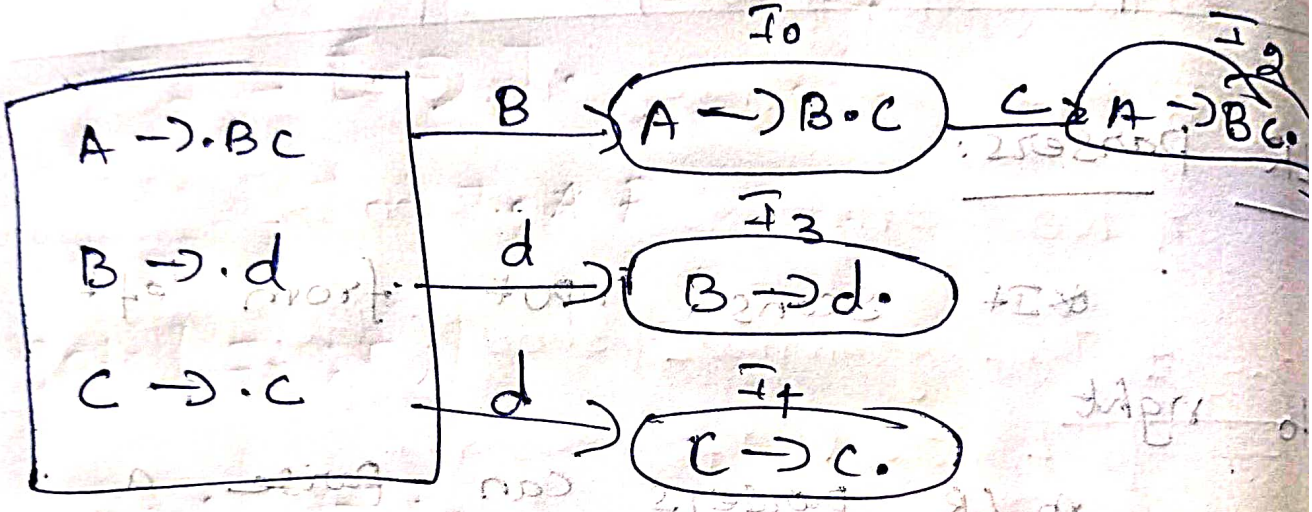
C → c

Step 1:

$A \to \cdot BC \longrightarrow R_1$

$B \to \cdot d \longrightarrow R_2$

$C \to \cdot C \longrightarrow R_3$



State	Terminal d	c	$	Varial A	B	C
0						
1						
2						
3						
4						

SLR:

* It is known as Simple LR parsing
* It is type of Bottom up parser

Step 1: Argumented grammar.

$E \to E+T$	$E' \to E$
$E \to T$	$E \to E+T$
$T \to T * F$	$E \to T$
$T \to F$	$T \to T * F$
$F \to (E)$	$T \to F$
$F \to id$	$F \to (E)$
	$F \to id$

Step 2: Canonical LR(0) collection.

$I_0 : E' \to \cdot E$

$E \to \cdot E+T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

goto (I0, E)

$I_1 = E' \to E.$

$E \to E. + T$

goto (I0, T)

$I_2 = E \to T.$

$= T \to T. * F$

goto (I0, F)

$I_3 : T \to F.$

goto (I0, ()

$I_4 : F \to (. E)$

$E \to . E + T$

$E \to . T$

$T \to . T * F$

$T \to . F$

$F \to . (E)$

$F \to . id$

goto (I_0, id)

I_5 : F → id.

goto (I_6, +)

I_6 : E → E + . T

T → . T * F

T → . F

F → . (E)

F → . id

goto (I_2, *)

I_7 : T → T * . F

F → . (E)

F → . id

goto (I_4, E)

I_8 : F → (E .)

E → E . + T

goto (I_6, T)

I_9 : E → E + T .

T → T . * F

goto (I_7, F)

I_{10} : T → T * F .

goto (I_8,)

I_{11} : F → (E) .

CLR(1)

Step 1: Write Argumented grammar

eg:

$S \to CC$

$C \to cC/d$

$S' \to .S$

$S \to .CC$

$C \to .cC$

$C \to .d$

Step 2: Find lookahead value:

$S' \to .S, \$$

$S \to .CC, \$$

$C \to .cC, c/d$

$C \to .d, c/d$

Step 3: find SLR for above grammar

Step 4: Make a parsing table

Step 5: Reduce the repeated grammar

6) Error handling and recovery:

* Error handling and recovery are important aspects of Syntax analyzer. in a compiler or interpereter.

*Syntax errors are encountered by Syntax analayzer

* Here are some of common techniques used for error handling and recovery

1) Error Reporting:

*When Syntax error is detected the parser provides clear and informat error msg to the programmer.

* error information includes such as line number, position and brief description of the error

2) Panic Mode Relovery:

In this mode relovery, the parser skips a certain number of tokens

3. Insertion, Deletion, Substitution

4) Global correction

5) Interactive error Handling

7) YACC:

* YACC Stands for Yet Another Compiler Compiler

* YACC tool is used to produce a parser for a given grammar

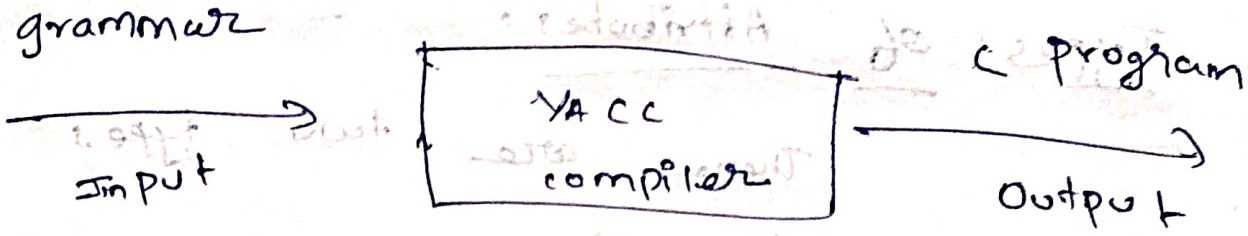
* It is designed to compile a LALR(1) grammar

* The input of YACC is rule or grammar

* The output of YACC is a program

eg: IP: file.y

Output: tab.c



Unit - 3

1.) SDD

* Syntax Directed - definition.

* It is a combination of context free grammar and Semmantic rules.

SDD = CFG + Semmantic Rules

* Attributes are associated with grammar Symbols

* Semmantic rules are associated with production

* Attributes may number, string reference, datatypes

Productions	Semantic rule
$E \Rightarrow E + T$	$E.val = E.val + T.Val$
$E \rightarrow T$	$E.val = T.val$

Types of Attributes:

There are two types of attributes they are

1) Synthesized Attribute
2) Inherited Attribute

Synthesized Attribute:

If a node takes a value from it's children then it is said to be synthesized attribute

eg:

A → BCD

A be a parent node and B, C, D are children node

Inherited Attribute:

If a node takes a value from it's parent node or sibling node then it is said to be

2) Construction of Syntax Tree:

functions:

Mknode (op, left, right)

Mkleaf (id, entry)

Mkleaf (num, val)

eg:

Expression: x * y - 5 + z

Step1: Convert the expression from infix to postfix

Step 2: make use of Mknode(), Mkleaf() and Mkleaf (id, num) functions

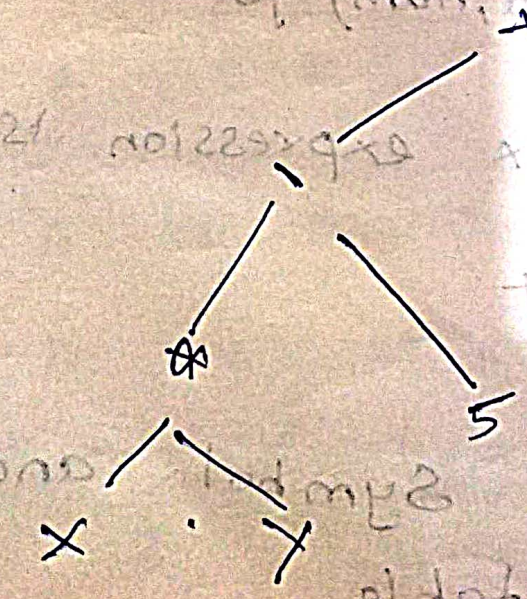
Step 3: The postfix expression is

x y * 5 - z +

Step 4: Create a Symbol and operation table.

Symbol	Operation
x	P_1 = MKleaf (id, ptr entry to x)
y	P_2 = MKleaf (id, ptr entry to y)
*	P_3 = MKnode (*, P_1, P_2)
5	P_4 = MKleaf (num, 5)
-	P_5 = MKinode (-, P_3, P_4)
z	P_6 = MKleaf (id, ptr entry to z)
+	P_7 = MKnode (+, P_5, P_6)

Step 6: Draw syntax tree:



Step 6: write SDD.. for above grammar.

Production rule	Semmantic Operation
$E \rightarrow E_1 + T$	new node ('+', E_1.Node, T.Node)
$E \rightarrow E_1 - T$	new node ('-', E_1.node, T.Node)
$E \rightarrow E_1 * T$	new node ('*', E_1.node, T.Node)
$E \rightarrow T$	E.node = T.node
$F \rightarrow id$	T.node = new leaf (id, id, ints)
$T \rightarrow num$.	T.node = new leaf (num, value)

3) Three Address Code:

* Maximum three addresses in each instruction
* Maximum one operator in RHS
* It would be equivalent for Syntax tree or DAG

Eg:

$a = b + c$ ✓ $a = x + y + z$ ✗

eg:

$a = x + y + z$

Convert it into three adress code

$t_1 = x + y$

$t_2 = t_1 + z$

$a = t_2$

eg2.

$a = b + c * d$

$t_1 = b + c$

$t_2 = t_1 * d$

$a = t_2$

eg3.

$a = (c + d) + (e + f)$

$t_1 = c + d$

$t_2 = e + f$

$t_3 = t_1 + t_2$

$a = t_3$

eg4:

do i = i + 1; while (a[i] < v);

L : t_1 = i + 1

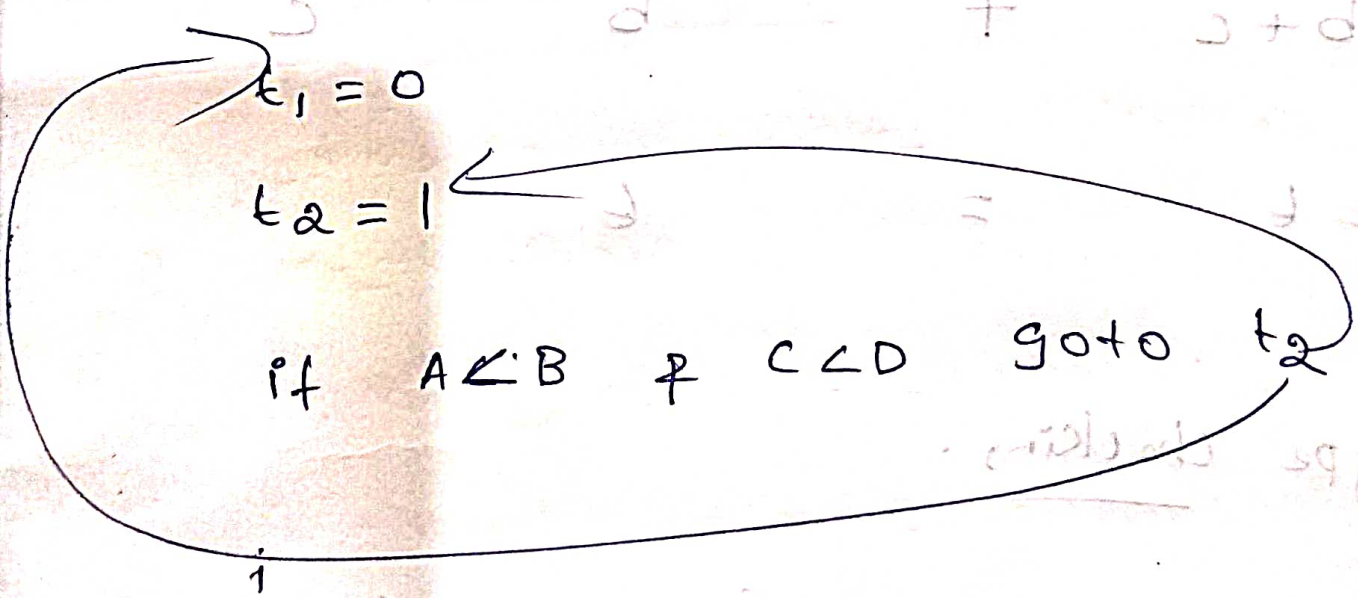
i = t_1

t_2 = a[i]

if t_2 < v goto L

eg5:

if A < B and C < D then t = 1 else t = 0 .



Quadruples.

eg	Operator	arg1	arg2	result
a= b+c	+	b	c	a
d = e-f	-	e	f	d
a = - d	-	d		a.

Triples:

eg	operator	arg1	arg2.
t= b+c	+	b	c
d = t	=	t	

Type checking.

Static → compile time.

Dynamic → Run time