Segment 4

Syntax analysis

Compiler

Parser

Top Down Parser

(top to down)

Bottom - up Parser (down to top)

Recursive Descent parser.

Back tracking Parser

predictive / Non recursive predictive parser.

LL(1)

Nonback tracking

LR parser

- LR(0)
- SLR(1)
- LALR(1)
- CLR(1)

Operator precedence parser

(1)

a parser is a component that breaks data to smaller elements for easy translation into other language

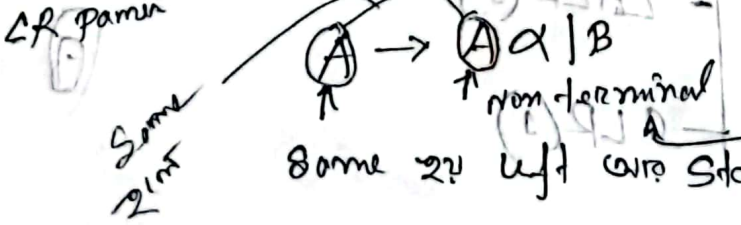
Error Recovery:

When a syntax error occours, in order for the compiler to recover, it usually has to discard the next few tokens, move to the end of the line and resume

There are 4 types of error recovery strategies

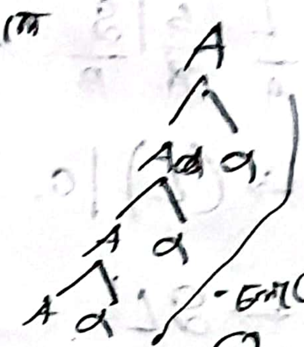
- Panic mode
- Phrase level
- Error production
- Global correction

Left recursion: Left side এবং Non terminal থাকলে



(If the production is left recursive if the left most symbol on the right side is the same the non terminal on the left side) (left recursion বলে)

Top Down parsing LR কে allow করে না, because infinite loop আছে Like $A \rightarrow A\alpha$ হলে



এমন হবে so এর Top Down allow করে না।

Elemination of Left Recursion :

$A \rightarrow A\alpha \mid \beta$ হলে LR eleminate করলে এরকম →

$A \rightarrow \beta A'$
$A' \rightarrow \epsilon \mid \alpha A'$

(সূত্র)

আমাদের হবে এভাবে then এইরকম alterenative দিলাম।

Example * $E \rightarrow E+T \mid T$ ($A = E$, $\alpha = +T$, $\beta = T$)

Sol^n: $E \rightarrow TE'$

$E' \rightarrow \epsilon \mid +TE'$

* $T \rightarrow T*F \mid F$ ($A = T$, $\alpha = *F$, $\beta = F$)

Sol^n: $T \rightarrow FT'$

$T' \rightarrow \epsilon \mid *FT'$

* $A \rightarrow ABd \mid Aa \mid a$ ($A = A$, $\alpha = Bd$; A, $\alpha = a$; $\beta = a$)

$B \rightarrow Be \mid b$

Sol^n: $A \rightarrow aA'$

$A' \rightarrow \epsilon \mid BdA' \mid aA'$

* $E \rightarrow E+E \mid E \times E \mid a$ ($A = E$, $\alpha = +E$; $A = E$, $\alpha = \times E$; $\beta = a$)

$E \rightarrow aE'$

$E' \rightarrow \epsilon \mid +EE' \mid \times EE'$

* $S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

$A \quad \alpha \quad \beta$

Soln $S \rightarrow (L) \mid a$

$L \Rightarrow SL'$

$L' \rightarrow \epsilon \mid ,SL'$

Left Recursion.

- Advantages of

$A \rightarrow A\alpha \mid \beta$

⇒ LR grammer is more intuitive than the transformed grammer

. A left recursive grammer

conclusion.

. Bottom-up parsing takes advantages of the behaviour of left recursion.

Left factoring: Top Down parser S.

It consists in factoring out prefix which are common two or more production.

অর্থ: $A \to a\beta \mid a\gamma$ Soln: $A \to aA'$ — then A' এখানে production ধরা হয়েছে (1) both side common হবে যেটা

$A' \to \beta \mid \gamma$ — then বাকি থাকা common বাদে লিখতে হবে।

Example:

* $S \to iEtS \mid iEtSeS \mid a$

$E \to b$

Ans: $S \to iEtS\,S' \mid a$

$S' \to \epsilon \mid eS$

$E \to b$

* $E \to T+E \mid T$

Ans: $E \to TE'$

$E' \to +E \mid \epsilon$

LR parser: LR (k) are type of bottom up parser that efficiently handle deterministic Context free language in linear time.

First and follow function

first

for any string of grammer symbols α, first(α) is the set of all tokens that may appear as the first symbol in a replacement string of α. | first & follow এ Left recursion Left factoring থাকলে হবে না এগুলো remove করতে হয় আগে।

Rules

* first (Terminal) = terminal

$A \to abE \mid G$ → 1st (a) terminal থাকলে সেটাই first

* If $X \to \epsilon$ then add ϵ to first(X)

$X \to AB \mid \epsilon$ → যেখানে ϵ হবে কেবল তখন ϵ দিব

X আবে X থেকে $ sign put হবে only.

* $A \to AF$, $F \to a \mid \epsilon$ $A \to b \mid \epsilon$ so (A) first {a, b, ε}

যদি Non terminal (A) হয় then যত বাকিগুলো আছে সেগুলো হবে then Non terminal আর put হবে নিবো।

* যদি এমন হয় first একটাই A. } a, ∈
আরেকটা b, ∈ } then {a, b, ∈} হবে total.
আরেকটি c, ∈

Example :* S → abc | def | ghi find the first.

here S → {a} ∪ S → {d} ∪ S → {g}

first {S} = {a, d, g}.

* S → ABC.

A → a | b | ∈.

B → c | d | ∈.

C → e | f | ∈.

∈ টা আমাদের ব্যাপার ∪ দিয়ে A, B, C এর সব ∈ এর জন্য সবগুলো হবে।

Ans

first of (S) → first (ABC)

⇒ {a, b} ∪ {c, d} ∪ {e, f} ∪ f(∈)

f(A) → {a, b, ∈}

= {a, b, c, d, e, f, ∈}

f(B) → {c, d ∈}

f(C) → {e, f, ∈}

(*) $E \to TE'$

$E' \to *TE' \mid \epsilon$

$T \to FT'$

$T' \to \epsilon \mid +FT'$

$F \to id \mid (E)$

Ans first of $E \to TE'$ [এর Non terminal so প্রথম first এর মান বসবে]

$\Rightarrow \{(, id\}$

first $E' \to *TE' \mid \epsilon$

$\Rightarrow \{*, \epsilon\}$

first $T \to FT'$ ← T' এর F এর মান বসবে

$\Rightarrow \{(, id\}$

first $T \to \epsilon \mid +FT$

$\Rightarrow \{\epsilon, +\}$ ⊕ terminal

first $F \to id \mid (E)$

$\Rightarrow \{id, (\}$ ← প্রথম terminal

parenthesis.

Follow

rule :

- start symbol S (S) শুরু করে দিবো direct.
- এখানে S → FT হলে S এর follow বের করতে হলে
  F → ST

  S তার পরে কোন আইকাই আছে খুঁজে then এর পরবর্তী টার first দিবো। Line Example এর T এর first লিখবো

- যদি T value মানে T এর first ε আসছে then T তে ε বসাই T vanish করে দিবো। না হলে বাদ T F থাকা F থাই F এর follow বের করে আবার S বসাই দিবো
- কোন সময় terminal পেলে direct বসাবে S → FTa
  F → ST
  T → SF

  যেকোন follow বের করতে RHS side এ (দেখে) then তার immediate RHS side এ যদি [illegible]

  এখানে T follow {a,}

- Non terminal পেলে তার production যদি [illegible] then (A) → A [illegible] A থাকে যদি [illegible] first লিখবো লিখতে না পেলে production

Example

terminal থাকলে direct নিব

* S → AaAb | BbBa

A → ε

B → ε

(Ans) ⇒ fo(A) → {a, b}

fo(B) = {b, a}

* S → ABC2 ε

A → DEF ⇒ fo(A) = ($) শুধু এখন ($) বসবে

B → ε

C → ε

First and follow

Production	First	Follow
E → TE'	{(, id}	{), $} $ by default
E' → +TE' | ε (+ terminal)	{+, ε}	{$,)} E' তে কিছু নেই তাই E নিতে হবে
T → FT'	{(, id}	{+, $,)}
T' → *FT' | ε (* terminal)	{*, ε}	{+, $,)}
F → (E) | id (id terminal)	{(, id} terminal যেগুলো আগে আসে সেগুলো নিব	{*, +,), $}

Recursive Descent parser

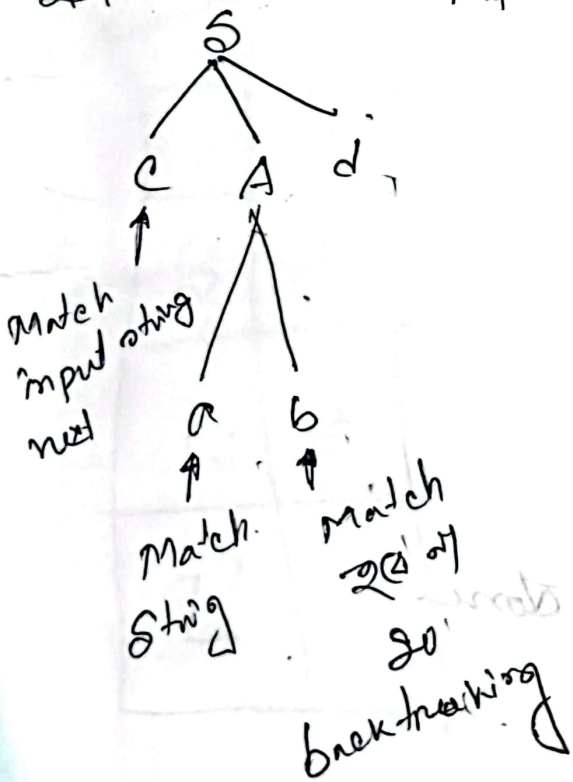
It is uses a set of recursive procedures to scan its input. It involve backtracking. So repeated scan of its input.

Example $S \rightarrow cAd$
$A \rightarrow ab|a$

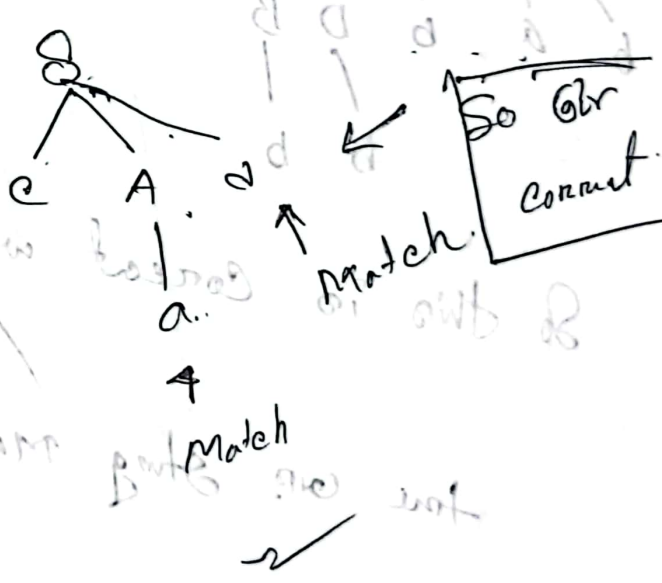
String $w = cad$.

Here 2 pointer. String Start point

Step 1.



Step 2 → backtracking [repeated]



Que. How could a top down parser decide which production for S to use to derive babbb?

$S \rightarrow AB \mid CD$

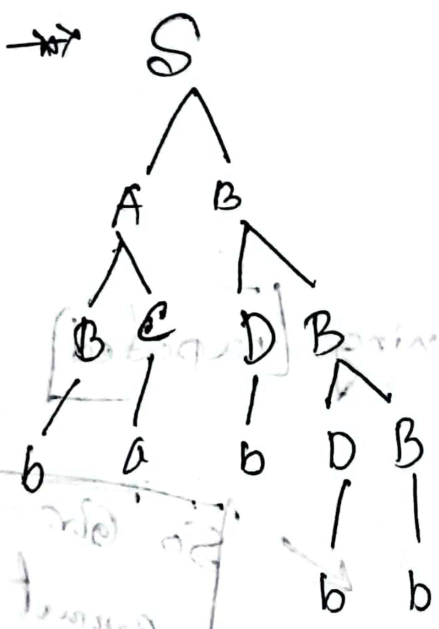
$A \rightarrow BC \mid CA \mid a.$

$B \rightarrow CA \mid DB \mid b$

$C \rightarrow BA \mid AD \mid a.$

$D \rightarrow AC \mid BD \mid b.$

input string babbb.



So this is correct way

true case string match হলো done

LL parsing / ~~Predictive~~ parser

An LL parser parses the input from left to right (L) and uses a Leftmost derivation (L).

- first follow আগে 1st then Left variable এবং terminal.

Example

		first	follow
$S \rightarrow (L) \mid a$ (1, 2)	→	{(, a}	{$, ,,)}
$L \rightarrow SL'$ (3)	→	{(, a}	{)}
$L' \rightarrow \epsilon \mid ,SL'$ (4, 5)	→	{ε, ,}	{)}

total 5 ta production

(parse table)

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

$L' \rightarrow \epsilon$ তাহলে L' follow দেখব

1. production. এখানে $S \rightarrow (L)$ এর first {(} so তাহলে 1 না {(}

So this is LL parsing.

* $S \rightarrow aSbS \mid bSaS \mid \epsilon$ (productions numbered 1, 2, 3)

first → {a, b}, ε}

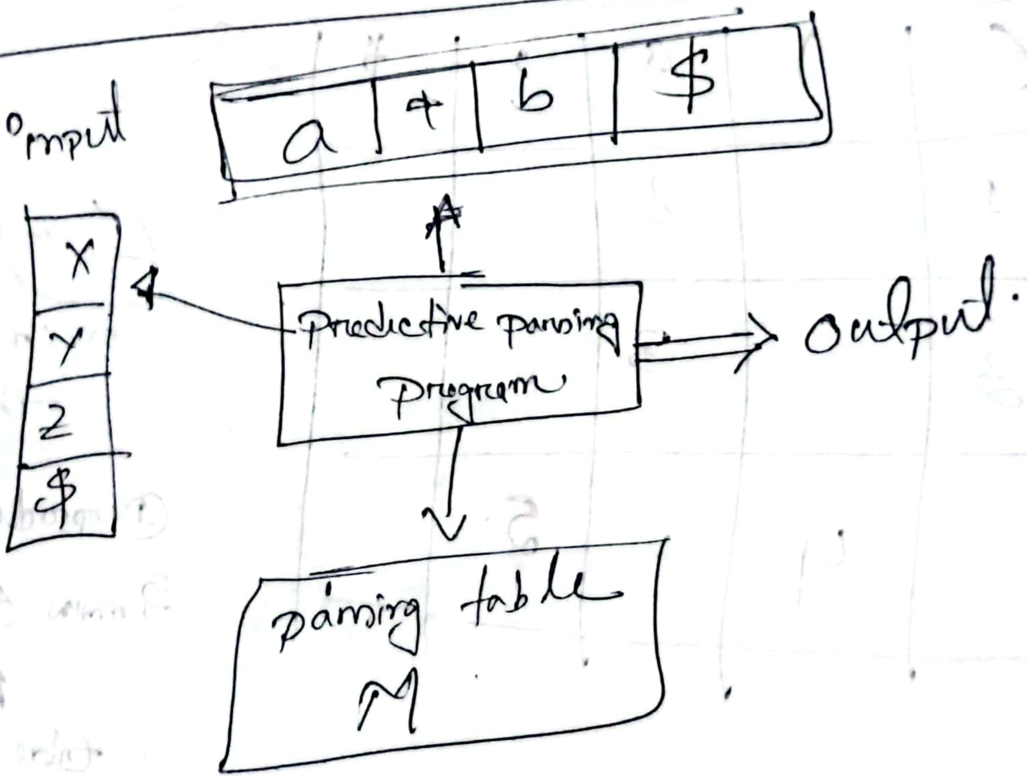
follow → {b, a, $}

	a	b	$
S	1/3	2/3	3

S→ε হলে S কি follow এর ঘরে বসবে — একের অধিক তাহলে

So it is not LL grammar.

Show the LL parser model. (Non recursive predictive)



LL(1) : It is called Non-recursive predictive parser.

The basic idea of a Nonrecursive descent parser is to use the current input symbol to decide which alternative to choose. Grammar which have the property are (LL-1).

* LR(k) L R (k) input symbol

Left to right scanning — Construction rightmost derivation in reverse

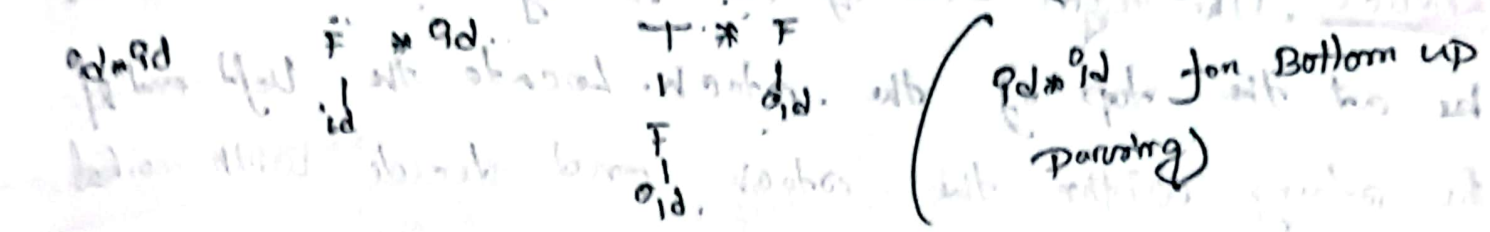
Top down : It is the parse tree is generated in the to bottom (ie from root to leaves). [Left Most]

Bottom up : It is a strategy for parsing strings of CFG that attempts to construct parse tree leaf to root

Segment 3

Bottom up Parsing

Bottom up parser traverses the parse tree from bottom up.



(id*id for Bottom up parsing)

The key decisions during bottom up parsing.

- when to reduce (reducing string)
- what production to apply.(

Shift reduce Parsing is a form of bottom up parsing

- a stack hold grammar symbols.
- a input buffer holds the rest of the string to be parsed.
- parser reports (this until) detected error or stack contains Start symbol.

Shift reduce parser.

Shift: Shift the next input symbol onto the top of the stack.

Reduce: The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.

Accept: Announce successful completion of parsing.

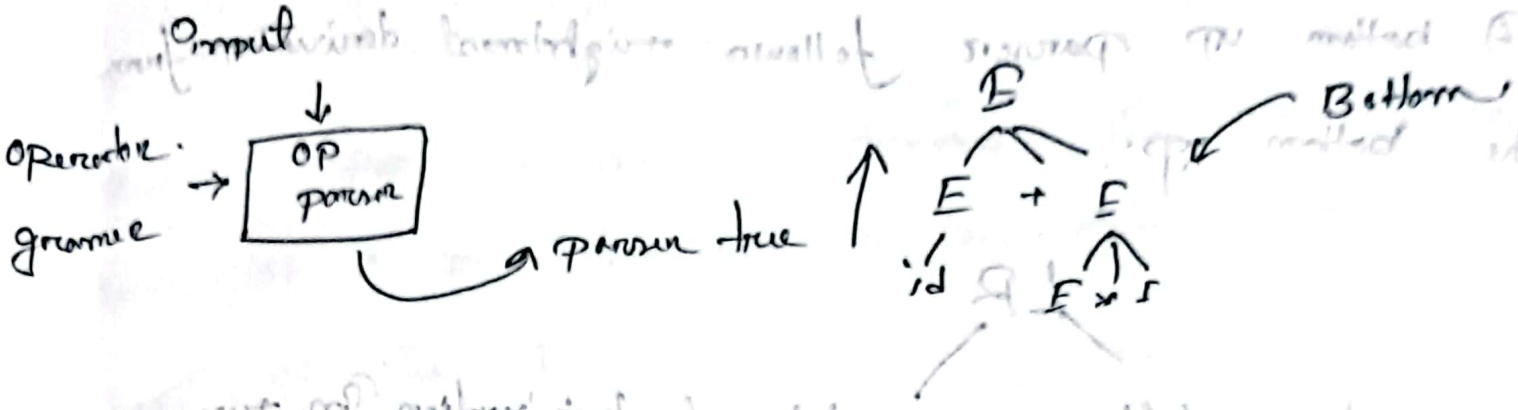
Error: Discover a syntax error and call an error recovery routine

Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.

[RHS (Right hand side) of a production rule is replaced by the symbol on LHS of production.)

Example - LR Parsing

Operator precedence parser



Operator grammer

(i) No Epsilon in right side.

$E \to EE \mid E*E \mid id$

no epsilon

(ii) no adjacent variable.

AB ← (গুণফলের ক্ষেত্রে)

A+B ← যোগ

A*B ← গুণ

মাঝখানে operator থাকা

$E \to a*b \mid E+E \mid id$ ← operator precedence grammer

$E \to EAE$ ← non operator grammer

A → + / *

So convert

$E \to E+E \mid E*E \mid id$

LR Parser

- A bottom up parser follows rightmost derivation from the bottom up.

Process tokens left to right

rightmost derivation in reverse order

- LR parsing : The production is applied only after the pattern has been matched.

- In LL (Predictive) Parsing, the production was selected and then the tokens were matched to it.

- In LR parser we have
  - A parse table
  - An input buffer
  - A stack of LR(0) states.

LR is PDA

LR parser perform three types of operation:

- Shift: A token from the input buffer to the stack.
- Reduce: Pop sequence of tokens that match the body of a production and replace them with the head of the production.
- Goto: a new state.

* LR(0): a look ahead reduces as soon as the body of a production is matched of next tokens.

SLR (Simple LR): Depending on the next token

LR(1) parser: 1 (look ahead): incorporates the next token as part of the current 'state' of the process.

LALR (Look ahead LR) is an LR(1) parser with its state much consolidated.

LR(0) items

It is a production with a special marker (·) that marks a position within the body of the production.

- Build LR parse table, 1st find LR (0) items

Ex

$E \to E + T$

- the possible LR(0) items.

$E \to \cdot E + T$, $E \to E \cdot + T$, $E \to E + \cdot T$, $E \to E + T \cdot$

$\#[A \to \alpha \cdot B]$ is "we have processed α and we might process B next"

- we are build a PDA whose states are sets of LR(0) items.

LR parsing

Steps

- First we augment the grammar with the new start symbol.

$[S' \to S]$

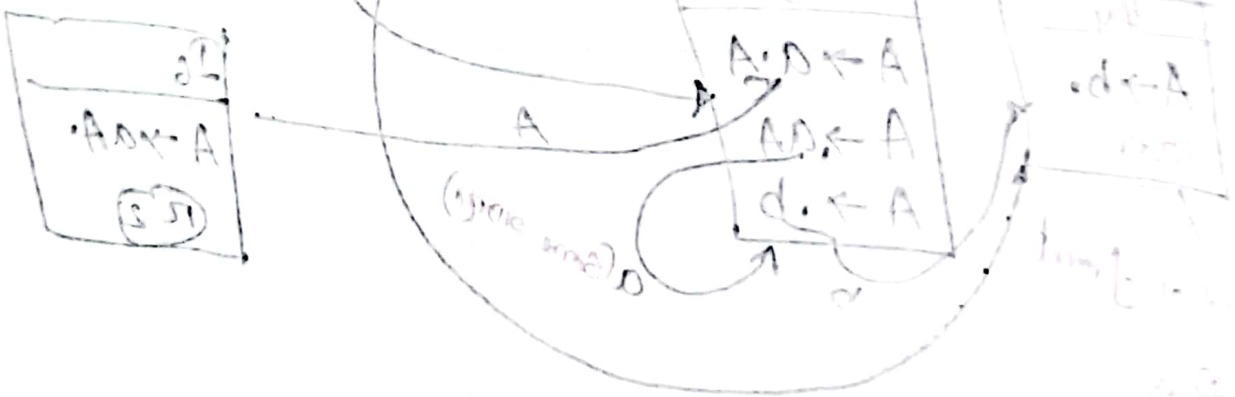
- Initial state I_0 called I_0 (item 0)
- State I_0 is the closure of the set

$\{S' \to \cdot S\}$

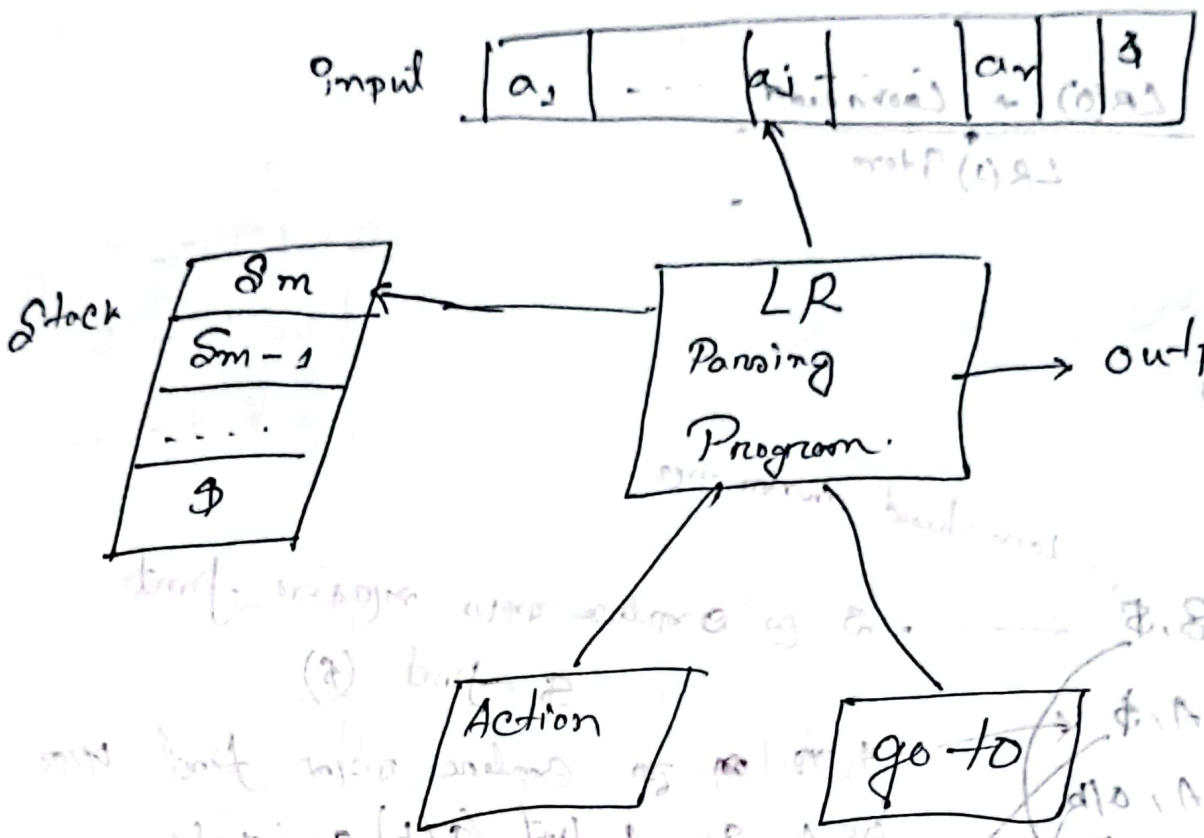
- for each item $[A \to \alpha \cdot B\beta]$ set each production $B \to \gamma$ in grammar add the item $[B \to \cdot \gamma]$.

Continue ...

- We call $[B \to \cdot \gamma]$ an initial B-item.



LR Parsing Model



* $A \to \alpha a B \cdot$ [reduce q item]

$A \to \alpha a \cdot B$ [shift q item]

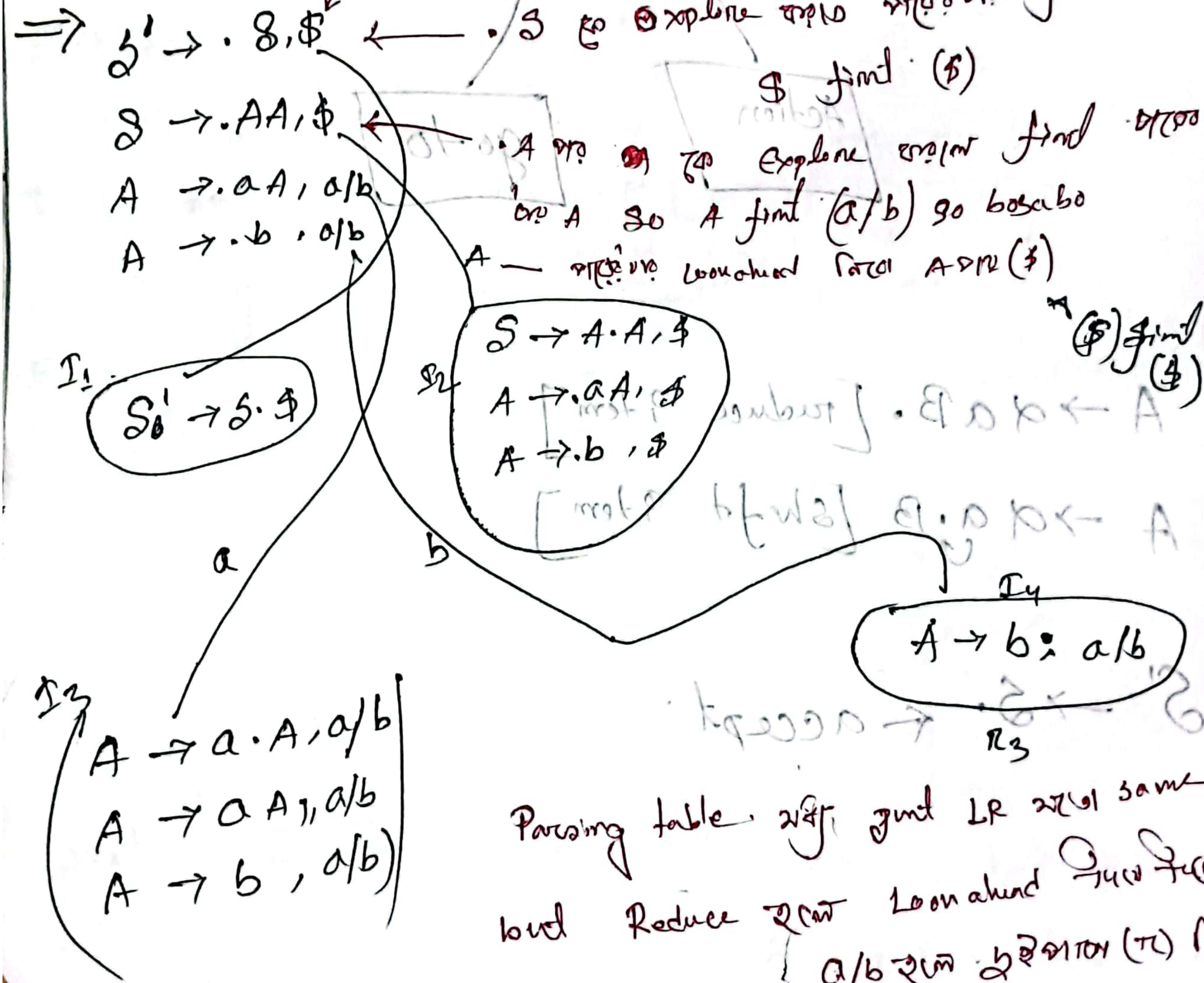
* $S' \to S \cdot$ ← accept.

CLR(1) Parsing

LR(0) + Lookahead = LR(1) item

$S \to AA$
$A \to aA$
$A \to b$

Lookahead mean করে



.S কে explore করলে পরবর্তীর first $ first ($)

.A এর পর A কে explore করলে যা find করবো এখানে A so A first (a/b) so bosabo

A — পরবর্তীতে lookahead দিবো A এর ($)

($) first ($)

Parsing table সবই first LR এর মত same but Reduce হবে Lookahead দিয়ে দিবো { a/b হলে দুইজায়গায় (r) দিবো

~~Shift reduce parsing~~

LL(1) using string

$A \to Q$
$Q \to St \mid Rr$
$S \to sS \mid \epsilon$
$R \to qR \mid \epsilon$

String ⇒ "qqqr"

		FIRST	FOLLOW
①	$A \to Q$	{s, ε, q, t, r}	($)
②	$Q \to St \mid Rr$	{s, ε, q, t, r}	($)
③	$S \to sS \mid \epsilon$	s, ε	(t)
④	$R \to qR \mid \epsilon$	q, ε	(r)

(LL) parser

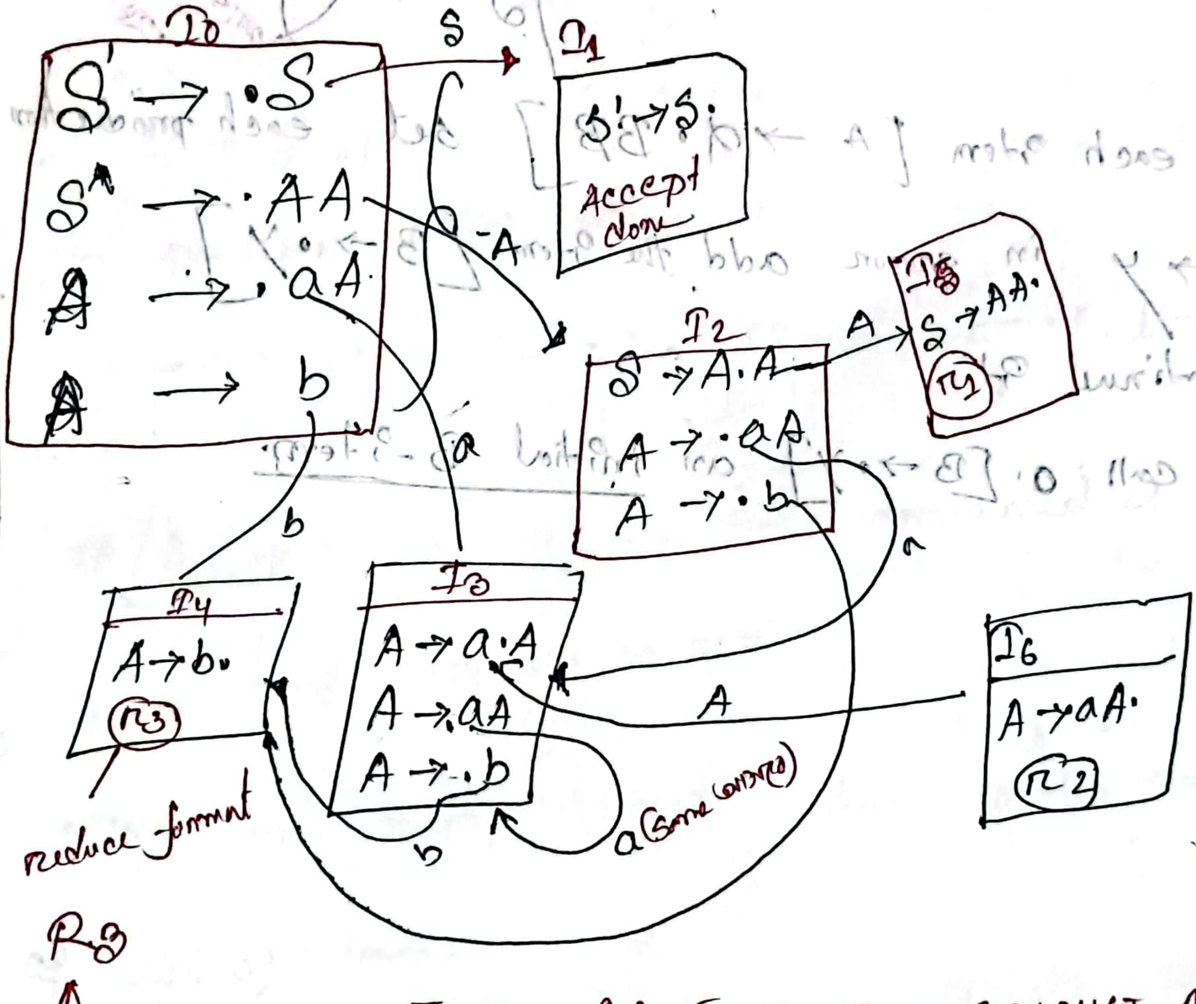
	s	t	q	r	$
A	1	1	1	1	
Q	1	1	1	1	
S	1	1			
R			4	④	

Check the given grammar is LR(0) or not

S → AA — Production 1
A → aA — 2
A → b — 3

[Shift and reduce
Same table এ হবে
LR(0)]

• Start symbol থেকে ⓪ দিয়ে New start symbol then আর একটা, but ① same জায়গাতেই থাকবে.



[Note: •AA থেকে একটা গেলে মানে A•A হবে তখন •A আছে So A এর দুইটা Production গুলো নিতে পারবে Same process চলবে এ
AA• , b• reduce format এ আসবে]

LR(0) Parsing table

States	Action (terminal)			Goto (variable)	
	a	b	$	S	A
I_0 → (shift)	S_3	S_4		1	2
I_1 →			accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	R_3	R_3	R_3 Acting reduction correct as box		
I_5	R_1	R_1	R_1		
I_6	R_2	R_2	R_2		

So here Dont arrange Shift reduce in Same Line

So this LR(0) parsing

Example 2 LR(0) or not

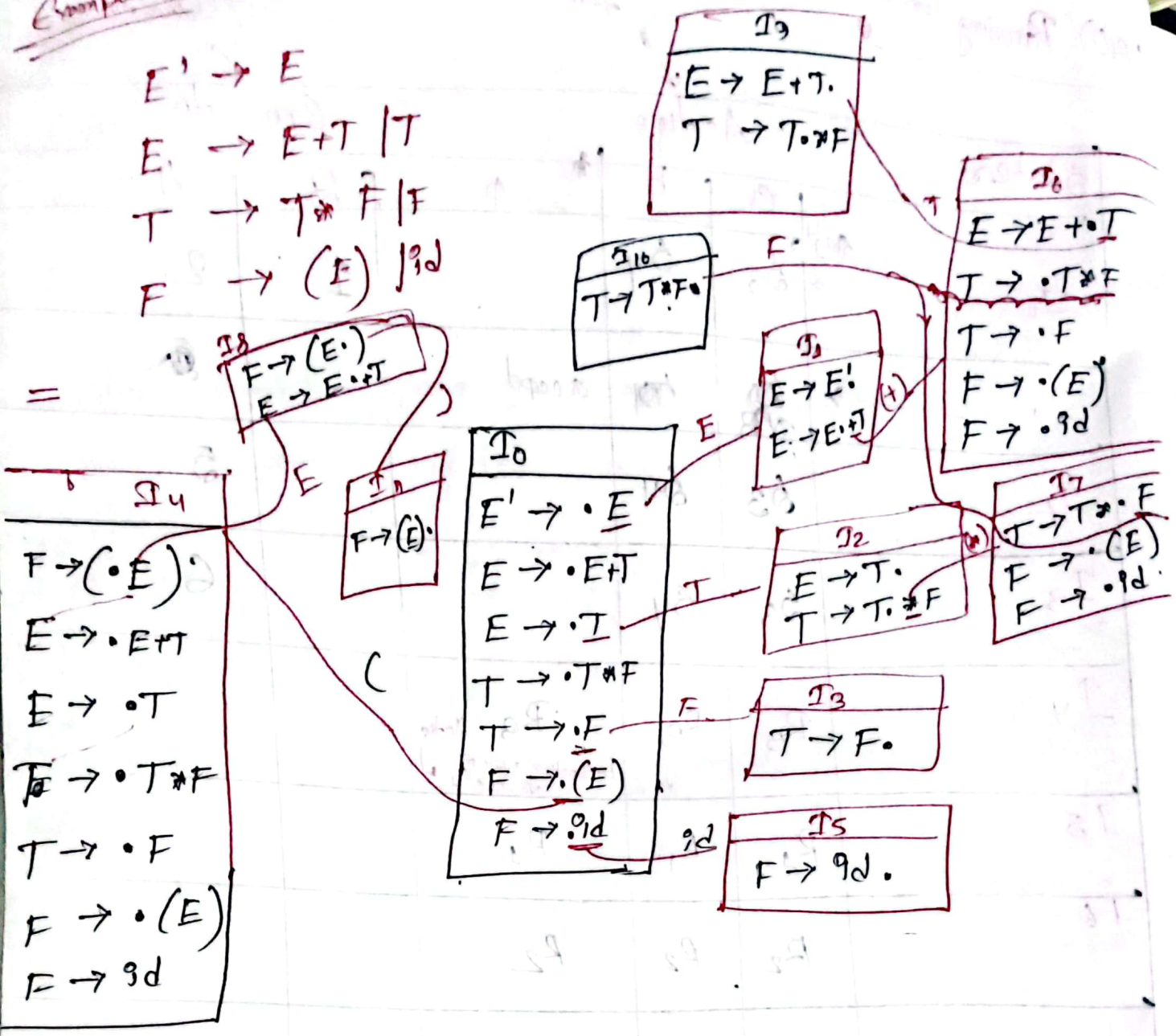
$E' \to E$

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

=



Stack	Input	Action
$	1111r$	—
$A (1st production)	1111r$	A → S (red)
$ SRr	1111R$	S → Rr
$ 1Rr	11	R → (1r)r
1111 RR	R$	
Rr		
∈ r	r $	
$	$	Accept

Lec 6

Syntax Directed Translation

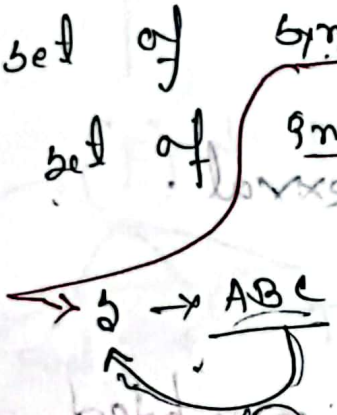
Syntax Directed definition (SDD):

It is a context-free grammar with attributes added to the grammar symbols.

SDD → CFG + Semantic rules.

SDD each node has:

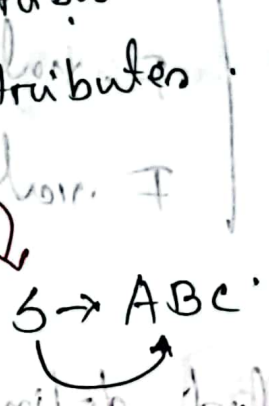
- a set of synthesized attributes
- a set of inherited attributes



child to parent info.

synthesized attributes.

If its value at a parent node can be determined from its children]



A value will be parent to sibling (or) value to sibling

B C sibling

[If determined from the value of attributes at the sibling and Parent, and itself that node]

A → B C

Syn - A ⇒ B.b

B ⇒ C.c

Production rule & Semantic rules.

Syntax Directed definition

Production	Semantic rules
$L \rightarrow En$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

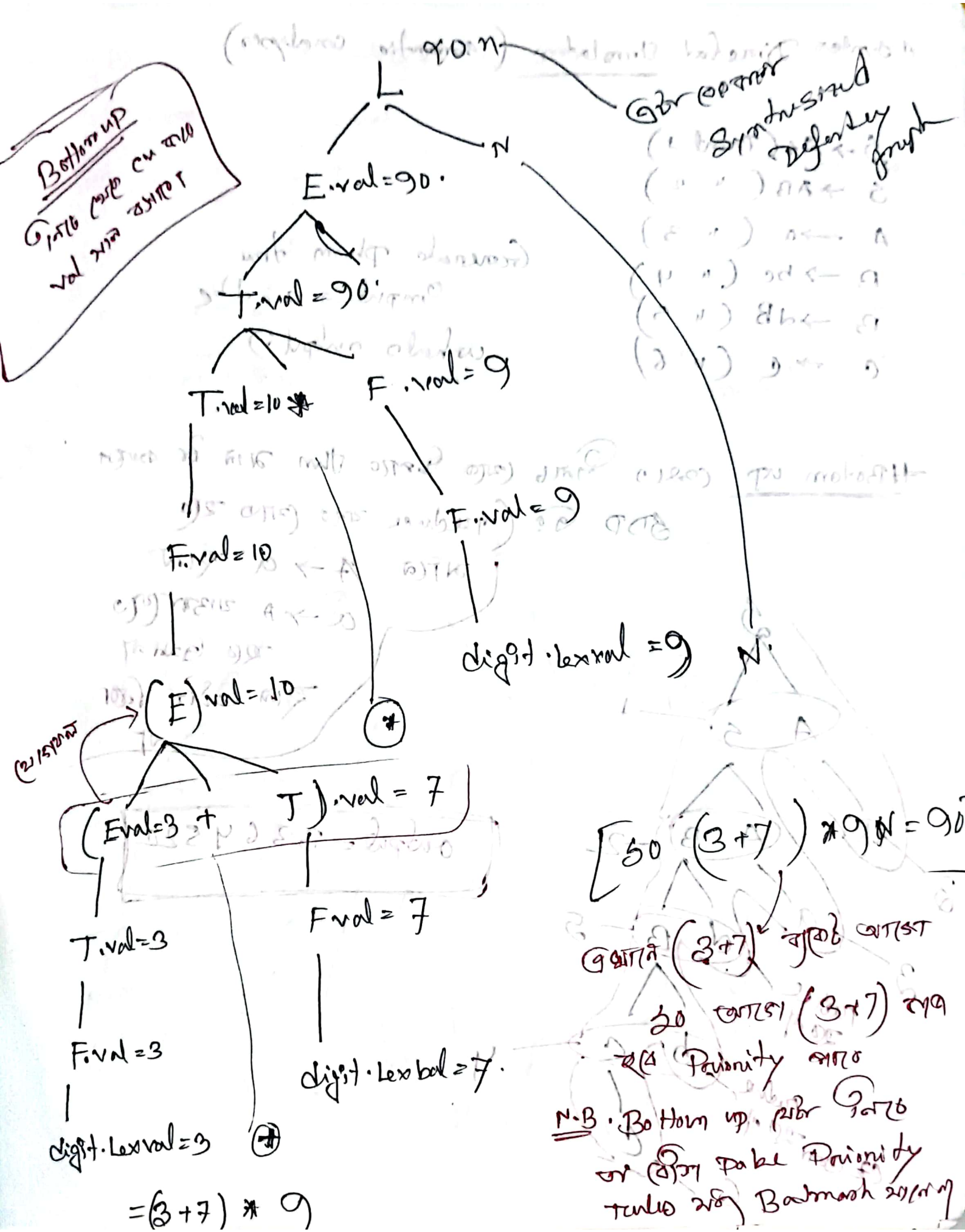
Syntax directed definition also annotated.

Parse tree

Previous example of SDD also.

$(3+7)*9$ annotated

NB> [illegible]

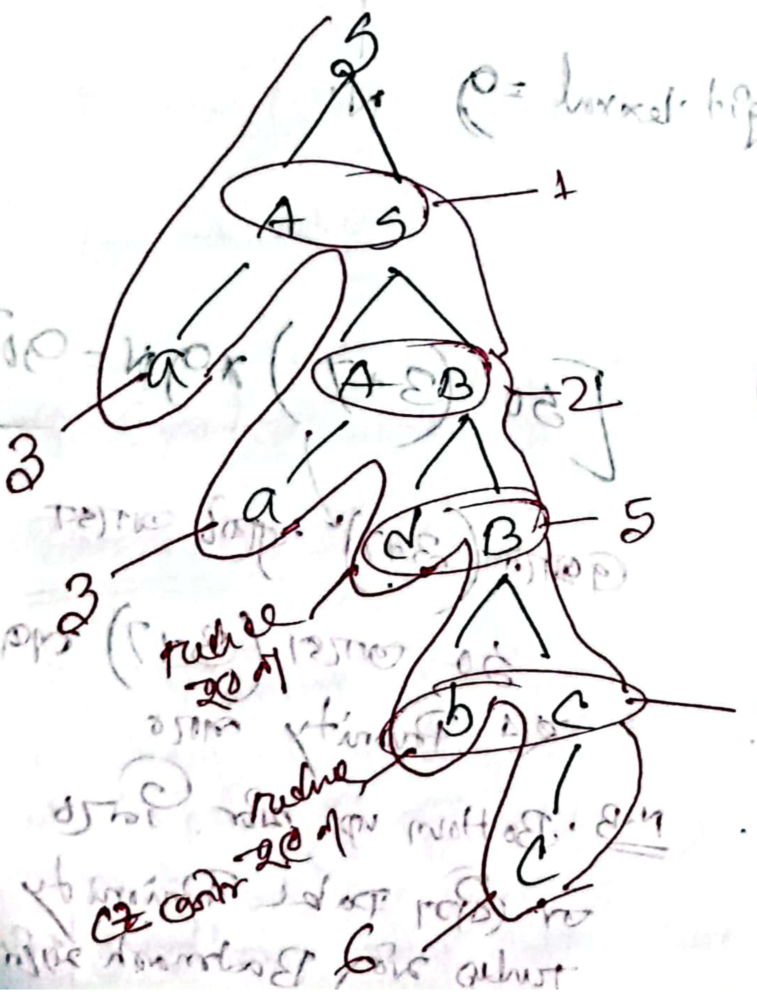

Bottom up
L
N
E.val=90.
T.val=90.
T.val=10
F.val=9
F.val=10
F.val=9
digit.lexval=9
N.
(E) val=10
*
E val=3 + T).val = 7
T.val=3
F.val=3
digit.Lexval=3
F val = 7
digit.Lexval=7.
=(3+7) * 9
[50 (3+7) *9 N = 90

Syntax Directed Translation (Semantic analysis)

S → AS (Print 1)
S → AB (" 2)
A → a (" 3)
B → bc (" 4)
B → dB (" 5)
C → c (" 6)

Generate parse tree
input = aadbc
what's output?

Botom up কেননা নিচের থেকে উপরে then মান টা করবে SDD তে (reduce করে গেলে হবে মানে A → a হলো a → A বাতকর থেকে হবে উল্টা হলো ওইটা নিতে



output = 3364521

$E \to E \& T$ { E.val = E.val * T.val }

$E \to T$ { E.val = T.val }

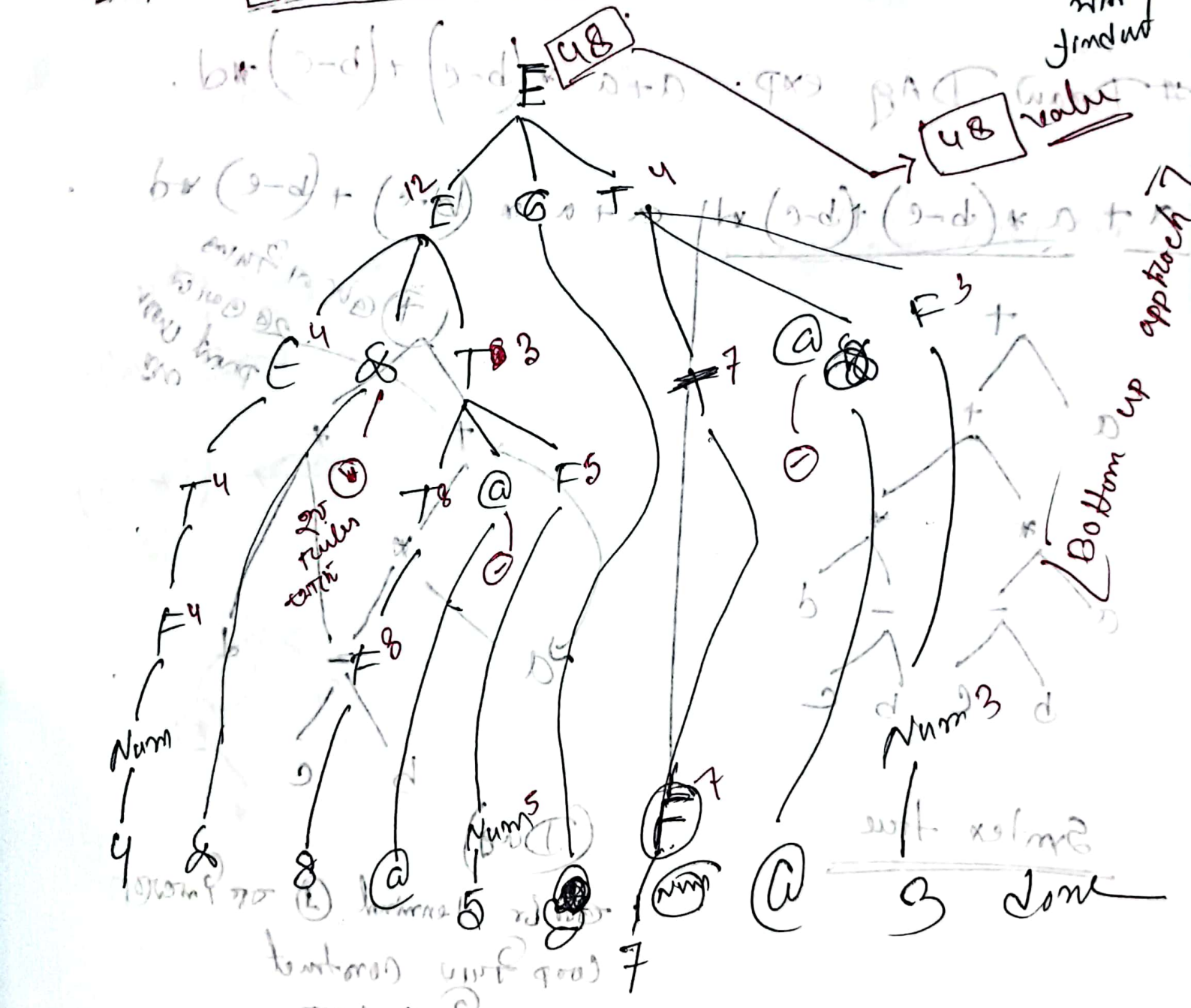
$T \to T @ F$ { T.val = T.val - F.val }

$T \to F$ { T.val = F.val }

$F \to Num$ { F.val = num }.

Input → 4 * 8 @ 5 & 7 @ 3.

what is output? annoted tree with output value

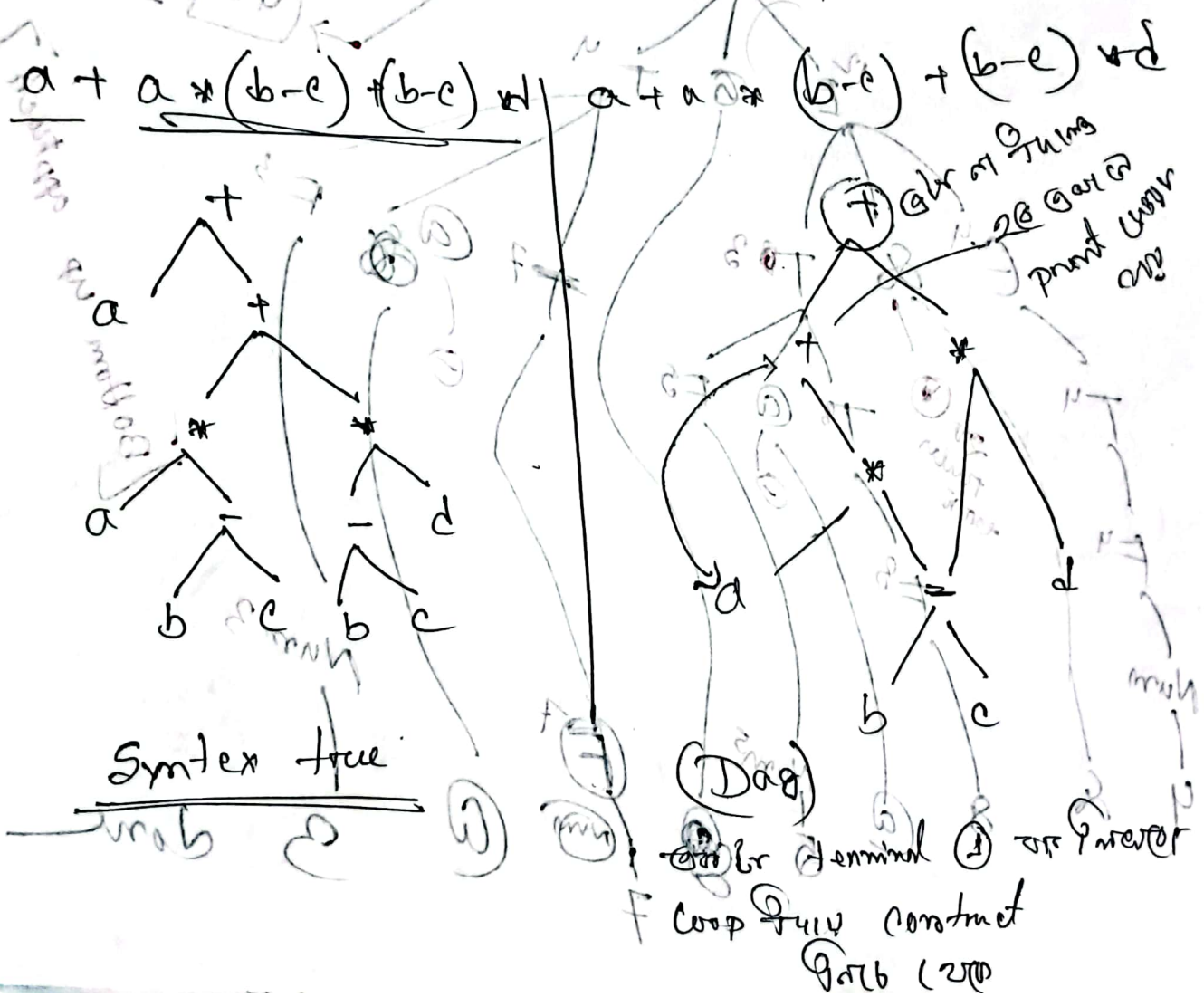


DAG (Directed Cycle Graph)

* A DAG for an expression identifies the common sub expression in the expression.

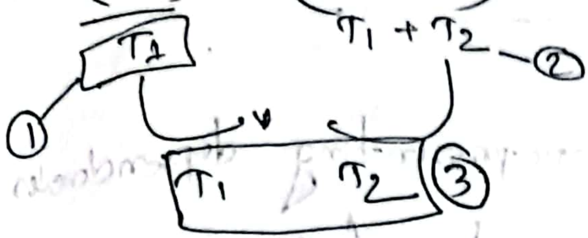
Syntax tree: It is condensed form of parse tree useful for representing language constructs.

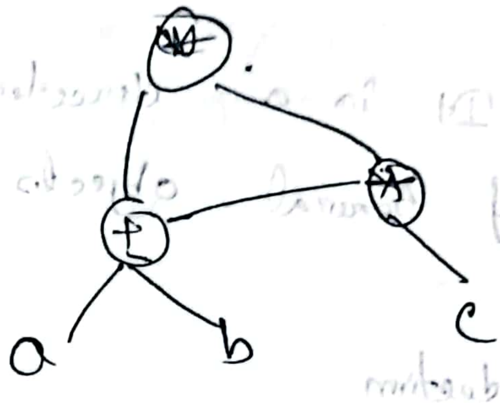
* Draw DAG exp. $a+a*(b-c)+(b-c)*d$



$(a+b) \times (a+b+c)$ Draw Dag.

Bodmas এর
() আগে Precedence





[illegible] এর:

$T_1 = a+b$

$T_2 = T_1 + c$

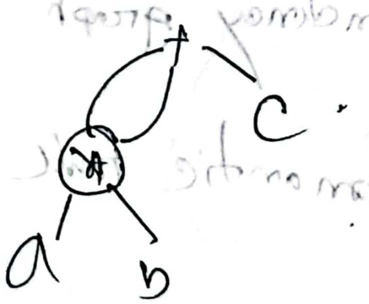
$T_3 = T_1 \times T_2$

Dag কেমন দিয়ে কেমন Diffrence Syntax tree থেকে

DAG টিতে যেমন উদাহরণ Line (A*b) দুইবার আছে হবে then same value দুবার নেবে।

$(a*b) + (a*b)+c$

t_1

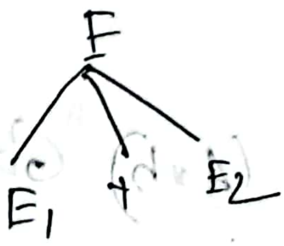


Dependency graph

- It is a directed graph representing dependencies of several objects towards each other

Production — Symantatic rule

$E \rightarrow E_1 + E_2$ —— $E.val = E_1.val + E_2.val$



$A \rightarrow BCD$ ← Synthesized BCD থেকে A তে যায় মান
$A \rightarrow BCD$, C A থেকে + B, D থেকে নিতে পারে

- Conceptual view of ~~STD~~ (SDT)

Input string → parse tree → dependency graph → evaluation order for semantic rule.

Ques.

D → TL
T → int | real.
L → L1, id | id

- Find semantic rule.
- Draw noted parse tree with inherited attributes and dependency graph for real id1, id2, id3 by grammer

Ans

Semantic Rules:

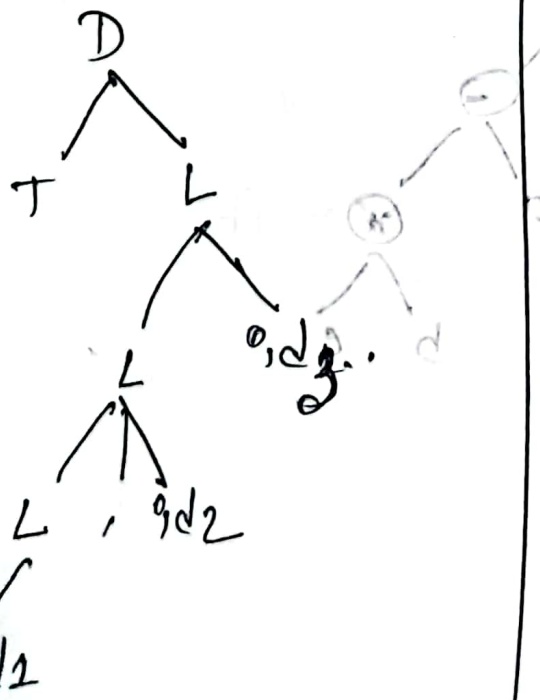
D → TL { D.val → T.val + L.val }

T → int | real { T.val = int.val, T.val = real.val }

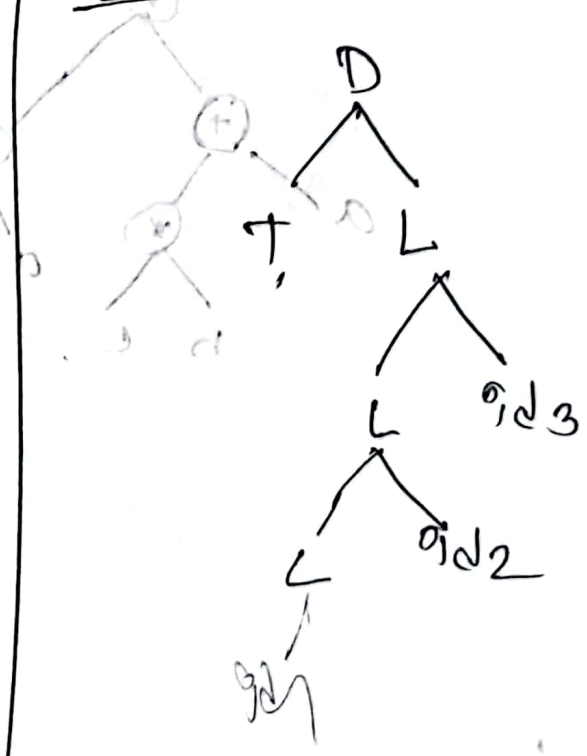
L → L1, id | id { L.val = L1.val + [id.val], id.val }

Parse trees for the sentence real id1, id2, id3 using

Phase inherited tree :



Dependency graph



log 7 & 8

Intermediate Code generation

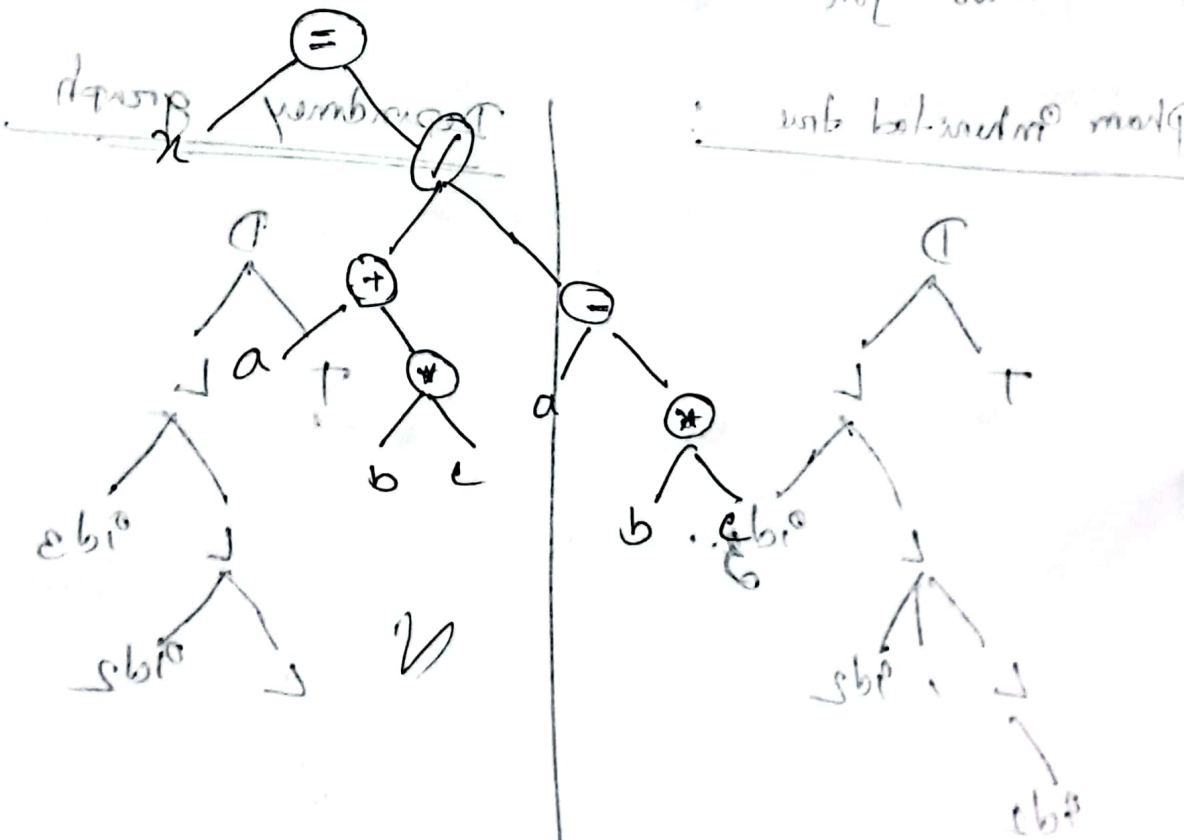
Source code → Intermediate code generation → Target Code

Commonly used intermediate code representation

① Syntax tree ② Postfix notation ③ Directed Acyclic graph (DAG)

④ 3 Address [code]

Syntax tree : $x = (a + (b * c)) / (a - (b * c))$



postfix Notation:

$a+b$ হলে $ab+$ হবে।

Diagram 5hb লাগে দিলাম।

$(a+b)*c \Rightarrow$ [PN] $\rightarrow ab+c*$

$(a-b)*(c+d)+(a-b) \Rightarrow ab-cd+*ab-+$

$((a*b)+c)*(c-(d/a)) \Rightarrow ab*c+$ ~~da/a/*~~ $cda/-*$

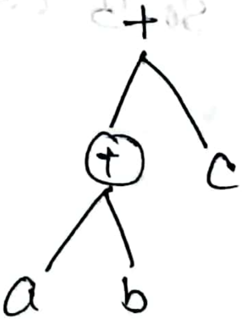
da আগে হবে, * হবে

then $x-a$ হবে ...

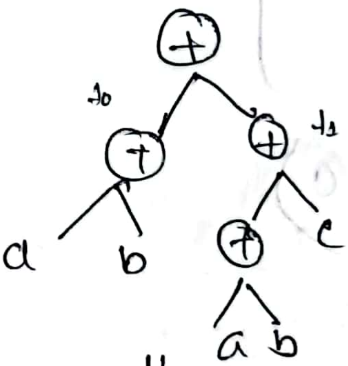
DAG: $a+b+c$

$t_0 = a+b$, $t_1 = t_0+c$, $d = t_0+t_1$

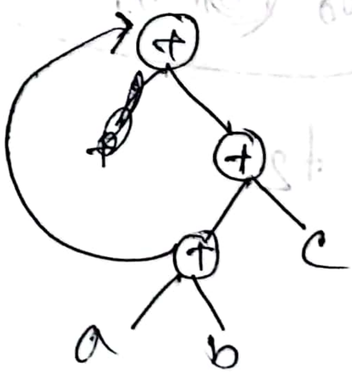
$t_0 = a+b$



$t_1 = t_0+c$



⇓ — shortcut করলে



একটা দিলাম then add করে ... দিলে 1 ও 2 দিলাম না

3 Address code (**)

$a = b + c * d$

let $t_1 = c * d$.

$t_2 = b + t_1$.

$a = t_2$.

Example

$x = (a * b) + (c * d)$

1 2 3 4 5

let

$t_1 = c * d$.

$t_2 = a * b$

$t_3 = t_2 + t_2$

$x = t_3$.

(ii) $a * -b\ (b + c)$

$t_1 = b + c$.

$t_2 = \text{uminus}\ t_1$

$t_3 = a * t_2$.

(ii) $a = b + c * d$.

$t_1 = c * d$.

$t_2 = b + t_1$

$t_3 = t_2$.

3 address statement.

① assignment — $x = y$ opcode 2 — $x = y + z$ (value x zero)

$x = y$ (value y zero)

② jump — conditional 'if $x > y$ go to L.

uncon — go to l.

③ array — $x = y[i]$

④ pointer address assign — $x = *y$

$x = \#y$

Translate expression quadruple, triple, indirect.

triple $a = b * -c + b * -c$.

$\Rightarrow$

$t_1 = \text{uminus } c$

$t_2 = b * t_1$

$t_3 = \text{uminus } b\ c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$.

OP code
+, *, -, /, uminus.

① Quadruple.

#	OP	arg1	arg2	result
(0)	uminus	c	–	t_1
(1)	*	b	t_1	t_2
(2)	uminus	c	–	t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5	–	a

(ii) triple

#	opcode	Arg1	Arg2
(0) →	uminus	c	–
(1) →	*	b	(0)
(2) →	uminus	c	–
(3) →	*	b	(2)
(4) →	+	(3)	(1)
(5) →	=	a	(4).

(0) — (0) location এর রেজাল্ট তোর গুণের (*).

Indirect triple :

#	Segment
(0) →	triple এর (0)
(1) →	" " (1)
(2) →	" " (2)
(3) →	" " (3)
(4) →	" " (4)
(5) →	" " 5

(*) Describe 3-address code, Define Quadruples Triples, Indirect triples.

• 3 address code is a type of Intermediate code which is easy to generate and can be easily converted to machine code. Ex: $x := y \text{ op } z$

(i) Quadruples: It is a record structure with 4 fields, which we call operation, arg1, arg2 & result. op code is internal code. Three stated statement $x = y \text{ op } z$ where y is ~~read number~~ Arg1, z arg2 op is operation (*, +, -) and x are result

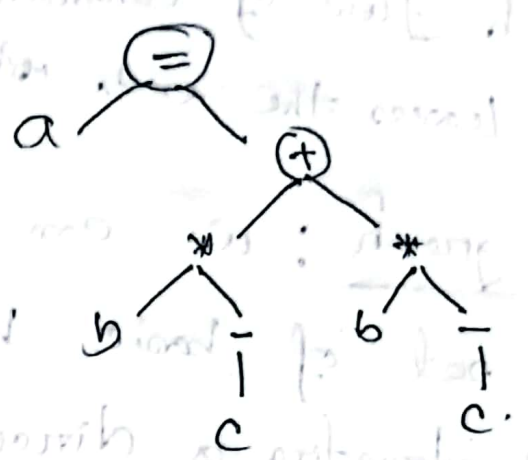
(ii) Triples: It doesn't use of extra temporary variable to represent a operation. It can represent by 3 fields OP, Arg1 arg2.

(iii) ~~(D)~~ Indirect triple: It has been considered is that of listing pointer to triples. It is require less space. It has only operation & Segment

(4) write 3 address code for syntax tree & DAG

$a = b * -c + b * -c$.

Code for syntax tree :- here $a = (b * (-c)) + (b * (-c))$



$t_1 = -c$

$t_2 = b * t_1$

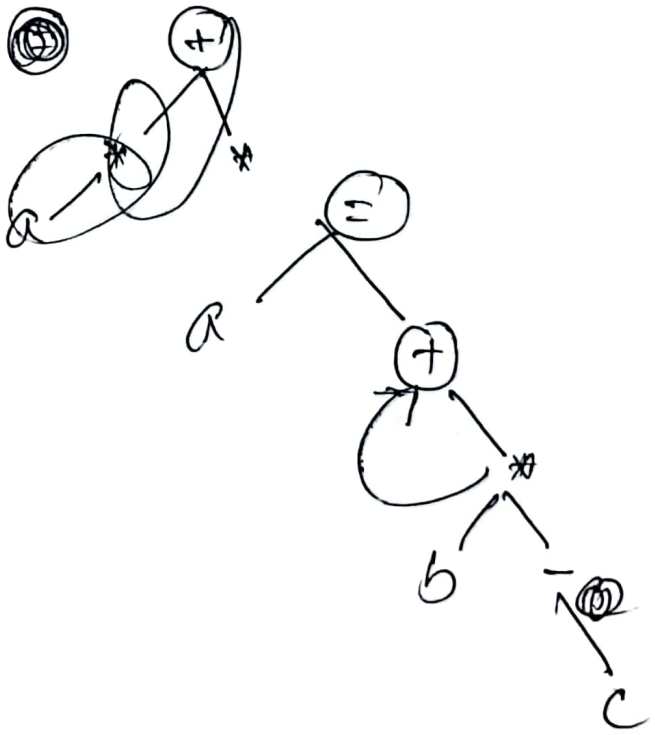
$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$.

$a = t_5$.

code for DAG



$t_1 = -c$

$t_2 = b * t_1$

$t_3 = t_2 + t_2$.

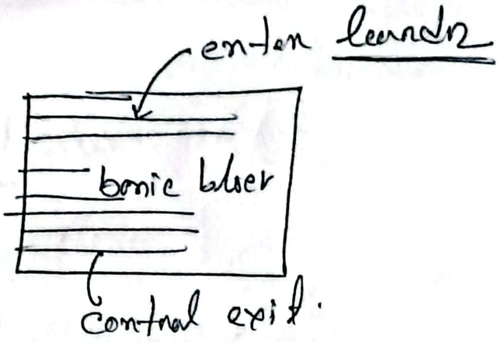
$a = t_5$

#Basic Block & flow graph.

Basic block: It is a sequence of statements in which flow of controls enter at the beginning and leaves the end.

Flow graph: We can add flow of control info to the set of basic blocks making up a program by constructing a directed graph is flow graph

Example of Basic Block & flow graph



* first statement leader.

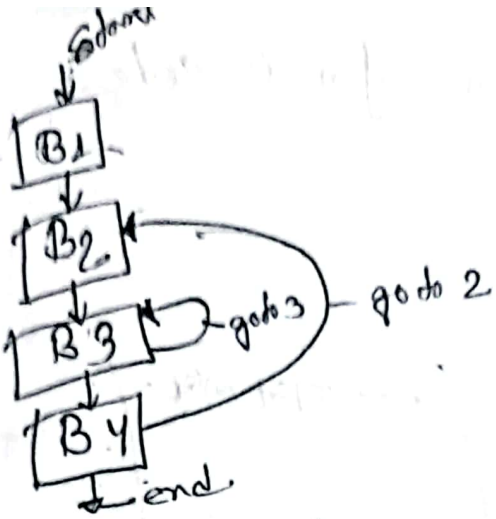
* i address of conditional, unconditional go to are leaders. If $J \leq 5$ goto (3)
  go to যেখানে যাবে সেইটা leader

* Next line of conditional and unconditional go to are leaders (go যাবে পরের leader)

Example.

1. $i = 1$ (Leader) (B1)
2. $j = 1$ (Leader (B2)
3. $t_1 = 5 * i$ (Leader) (B3)
4. $t_2 = t_1 + j$
5. $t_3 = 4 * t_2$
6. $t_4 = t_3$
7. $a[t_4] = -1$
8. $J = J + 1$.
9. if $J <= 5$ go to 3
10. $i = i + 1$ (Leader) (B4)
11. if $i < 5$ go to 2.

find the nodes & edge of flow graph



Ans: 4 Nodes 5 edge

Example

$$a[i] \begin{cases} = \text{array}(a) - t_3 \\ = i * 4 \quad - t_2 \\ = t_1[t_2] \end{cases}$$

3 address Code representation for C code

(ii) int a[10], b[10], i, dp = 0;

for (i=0 ; i<10 ; i++)
{ dp+ = a[i] * b[i];

i++ করা যাবে না =

3 address code

1. dp = 0
2. i = 0
3. If i >= 10 goto (16) — end করে দিবে ; false হলে
4. t_1 = addr (a)
5. t_2 = i * 4 — a[i] এর জন্য
6. t_3 = t_1 [t_2]
7. t_4 = addr (b) — become — i দিলে যায় যায় ck করে
8. t_5 = i * 4 — b(i) — 4 bit
9. t_6 = t_4 [t_5]
10. t_7 = t_3 + t_6.
11. t_8 = dp + t_7. — dp+ = a[i] * b[i]
12. dp = t_8. — dp = dp+ " ; এভাবে ভেঙে লিখলে মনে হয়.

13. $t_9 = i+1$ ← i++ এর

14. $i = t_9$ } পুনরায় 8 নং line এ ফেরত যাবে

15. go to (8)

16. end

(11) C code ⇒ 3 Address code.

```
C = 0
do
{
  if (a<b) then
     x++;
  else
     x--;
  C++;
}
while (C<5)
```

3 address code

1. $C = 0$
2. if $(a<b)$ go to (4) (if সত্য হলে then else হলে loop
3. go to
4. $t_1 = x+1$ ← $(x++)$ এর জন্য ওর
5. $x = t_1$
6. go to (9)
7. $t_2 = x-1$
8. $x = t_2$ } $(x--)$ এর জন্য ওর
9. $t_3 = C+1$
10. $C = t_3$
11. if $(C<5)$ go to (2)
12. end

C code to 3 address

while (A < C and B > D) do

if A = 1 then C = C + 1

else

while A ≤ D

do A = A + B

Ans 3 address code

1. if A < C then go to (3) ⟵ হইলে 3 (তেমনে) মানে next compare
2. go to (15) ———— না হলে end এ
3. if B > D go to (5) ———— Condition ok.
4. go to (15) ———— না হলে end এ
5. if A = 1 go to (7)
6. go to (10)
7. $t_1 = C + 1$
8. $C = t_1$
9. go to (1)
10. If (A ≤ D) go to (12)
11. $t_2 = A + B$
12. $A = t_2$
14. go to (10)

Example 5

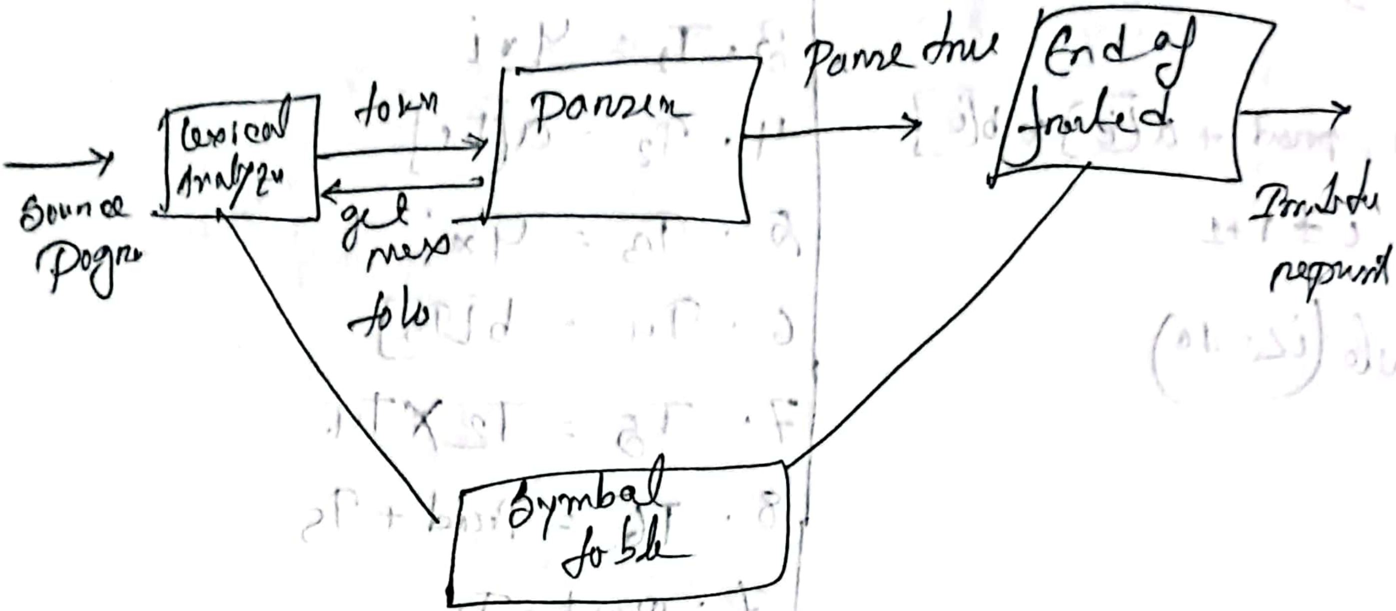
```
prod = 0
i = 1
do {
prod = prod + a[i] x b[i]
    i = i + 1
} while (i <= 10)
```

8 address code

1. prod = 0
2. i = 1
3. $T_1 = 4 \times i$
4. $T_2 = a[T_1]$
5. $T_3 = 4 \times i$
6. $T_4 = b[T_3]$
7. $T_5 = T_2 \times T_4$
8. $T_6 = Prod + T_5$
7. $Prod = T_6$
8. $T_7 = i + 1$
9. $i = T_7$
10. if ($i <= 10$) goto 3.

ex:6

Role of Parser



⑤ Error Recovery Strategies.

<u>Panic mode</u>: It is the easiest way of error recovery. This ~~stops~~ prevent-করে the parser from developing (এটার parser কে Delp' করে যাতে কোন error না থাকে)

If there are less error, then it is best

<u>Phrase Level</u>: In this stage If error find then parser perform local correction on this input. It can replace a prefix of the input.

(কোন কোন local correction করে হয় কোনো input এর prefix দিয়ে replace করে)

Error production : Its occur in the code. So that the designer can create enhanced grammer for use so that production create incorrect structure.

code এ হয়, তার জন্য better grammer use করা হয়।

Global correction : We want a compiler that makes few changes in processing to give correct input string. if x input string and grammer G the algorithm itself can find pharse tree for error easily and time লাগে। কোনো steps solve হয় না এখন দিয়ে।

First & follow

$S \to Bd \mid cb$ → (a, b, c, d) → ($,)

$B \to aB \mid \epsilon$ → (a, ε) → {d;

$C \to Co \mid \epsilon$ → (c, ε) → {b

$S \to i\epsilon ds \mid i\epsilon tses \mid a.$

$\epsilon \to b$

$S \to i\epsilon tS\,S' \mid a.$ → (i, a) → {$, e}

$S' \to \epsilon \mid eS \mid$ → (ε, e) → ($, e)

$\epsilon \to b.$ → (b) → {t

LL(1) parsing table

① আগে first আর follow বের করে LL table

① $E \to TE'$	→	(id, ( ) →	), $
② $E' \to +TE' \mid \epsilon$	→	(+, ε) →	(), $)
③ $T \to FT'$	→	(id, ( ) →	(+,), $)
④ $T' \to *FT' \mid \epsilon$	→	(*, ε) →	(+,), $)
⑤ $F \to id \mid (E)$	→	(id, ( ) →	(*, +,), $)

	id	(	*	+	$	)
E	①	①				
E'				②	②	②
T	③	③				
T'			④	④	④	④
F	⑤	⑤				

২নং $E' \to +TE' \mid \epsilon$ এর জন্য follow নিতে

$E' \to \epsilon$ এর জন্য follow E' follow নিতে

(iv) a(a,a) to rightmost derivation in shift reduce parser.

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

(iv) (a(a,a)

$\Rightarrow$ (S(a,a)

$\Rightarrow$ (S(S,a)

$\Rightarrow$ (S(L,a)

$\Rightarrow$ (S(L,S)

$\Rightarrow$ (S(L)

$\Rightarrow$ (S(L)S)

$\Rightarrow$ (L,S)

$\Rightarrow$ (L)

$\Rightarrow S$

... parser in comp...

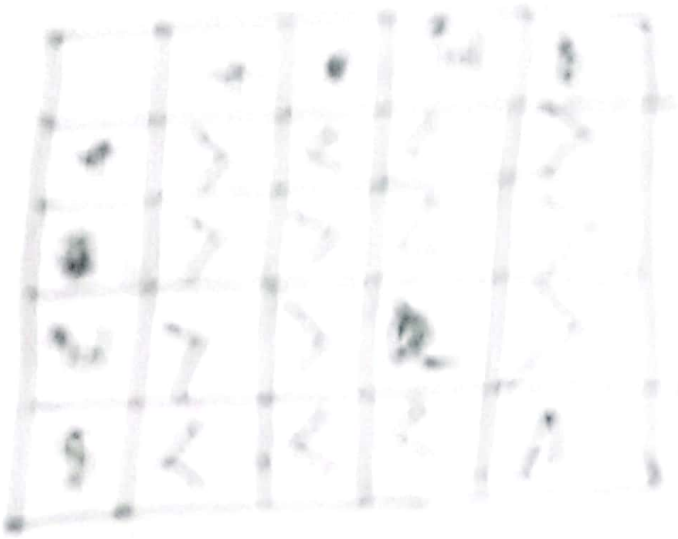
E → E+E
E → E*E
E → (E)
E → id.

String: id * (id + id)

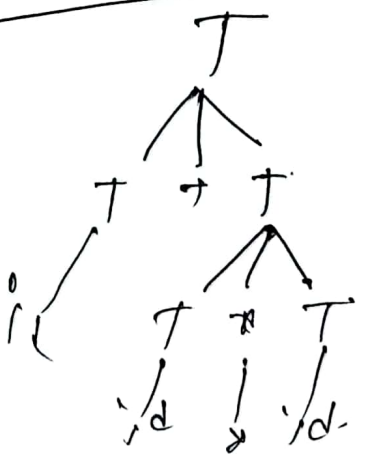
[আমি এখন থেকে table আকারে লিখবো -দেখবো]

Stack	String	Action
$	id * (id + id) $	
$ id	* (id + id) $	E → id, reduce
$ E	* (id + id) $	শিফট
$ E *	(id + id) $	শিফট
$ E * (	id + id) $	শিফট
$ E * (id	+ id) $	রি E → id (R)
$ E * (E +	id) $	শিফট
$ E * (E + id	) $	রি E → id (R)
$ E * (E + E	) $	শিফট
$ E * (E)	$	শিফট
$ E * E	$	
$ E	$	

$ E — 1st production — ($) sign - এখানে correct



Stack	relation	Input	Action
$	<	id + id * id $	shift id
$ id	>	+ id * id $	reduce
$ T	<	+ id * id $	shift
$ T +	<	id * id $	shift
$ T + id	>	* id $	reduce
$ T + T	<	* id $	shift
$ T + T *	<	id $	shift
$ T + T * id	>	$	reduce
$ T + T * T	>	$	R2
$ T + T	>	$	R2
$ T	Accept	$	



(M) operator Precedence

$$A \rightarrow A+A \mid A*A \mid a$$

String $a+a*a*a+a*a\$$

table

	+	*	a	$
+	>	<	<	>
*	>	>	<	>
a	>	>		>
$	<	<	<	A