Page NO.

1. 1 to 16 – Mid Term
2. 17 to 54- Final Term

MD IFTAKHAR KABIR SAKUR

25th BATCH

COMPUTER AND COMMUNICATION ENGINEERING

International Islamic University Chittagong

COURSE CODE: CCE-4829

COURSE TITLE: Machine Learning

COURSE TEACHER:

Adjunct Faculty

Computer and Communication Engineering

Compiler

Terminal:- যে জিনিসগুলো পরিবর্তন করা যায় না। অর্থাৎ, replace করা না গেলে Terminal।

S → AB.
A → a
B → b → Terminal

Non-Terminal:- যে জিনিসগুলো পরিবর্তন করা যায় অর্থাৎ, replace করা যায় তাকে Non-Terminal বলে।

Needed of Symbol Table:-

→ Used to manage information about identifiers (such as variables, functions & constants) encountered during the compilation or interpretation process.

(1) Identifier Management:- Stores information including their names, types, scopes & memory location.

(2) Scope Resolution:- Maintain the scope hierarchy and allow efficient lookup & resolution of identifiers.

(3) Type checking:-
Helps to check & ensure that operations involving identifiers are semantically valid.

(4) Memory management:- Tracks memory locations & enable efficient memory management.

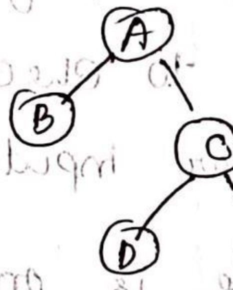
(5) Error Detection & Reporting:-
Facilitate error detection & reporting during the compilation or interpretation process.

(6) Optimization:- Compiler can optimize the generated code for improved performance & efficiency through symbol tables.

Derivation of Grammar, parse Tree.

=> Parse Tree:-
A parse tree or derivation tree is an ordered rooted tree that graphically represents the semantic information of a string derived from a content free grammar.

(যে string আছে তার graphical representation হই parse ~~Table~~ Tree)



□ <u>Ambiguous Grammar</u>:-

Same Grammar হতে different দুইটি parse Tree বানানো গেলে তা Ambiguous Grammar. <u>Example</u>:

For Grammar, G, For some string $w \in L(G)$ হলে তা Ambiguous Grammar

DFA / NFA draw:-

Automata:- A machine or program that run on a given input to check whether the input is a regular input or not

DFA:- Where there is only one possibility

NFA:-

Ambiguous or NOT:- (More than one → Ambiguous; Exactly one → Unambiguous)

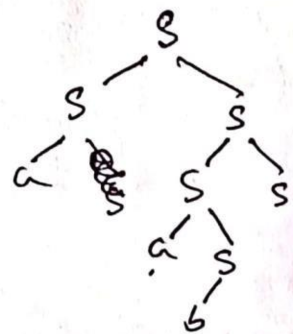
Ex:- $G = (\{S\}, \{a+b, +, *\}, P, S\}$

Where P consists of $S \to S+S \mid S*S \mid a \mid b$

The string a+a*b can be generated as:-

Ans:-

$S \to S+S$	$S \to S*S$
$\to a+S$	$\to S+S*S$
$\to a+S*S$	$\to a+S*S$
$\to a+a*S$	$\to a+a*S$
$\to a+a*b$	$\to a+a*b$



Used left derivation trees And got same result. So the grammar is ambiguous

② S → AB

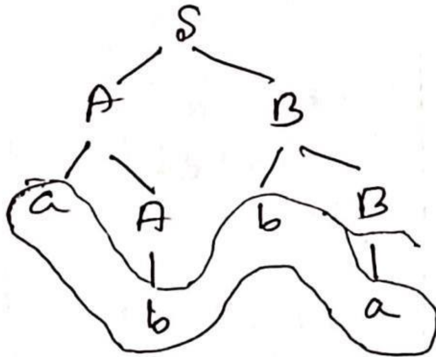
A → aA | b

B → bB | a

word: abba

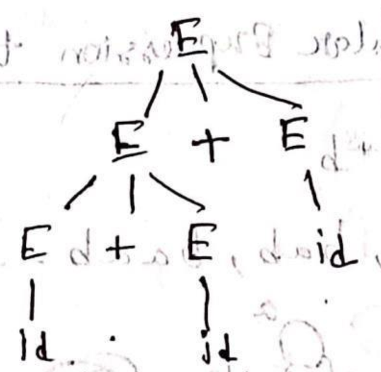
S → AB

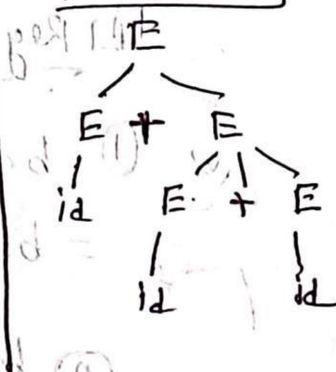
→



③ E → E+E | id

w → id+id+id





As there are 2 different trees it is Unambiguous.

④ x → x+x | x*x | a   w = a+a*a

x → x+x
→ x+x*x
→ a+x*x
→ a+a*x

a+a*a

$x \to x * x$

$\to x + x * x$

$\to a + x * x$

$\to a + a * x$

$\to a + a * a$

এটি Ambiguous.

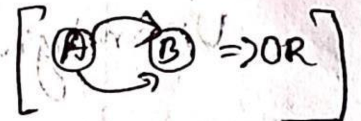
Regular Expression to Finite Automata:-

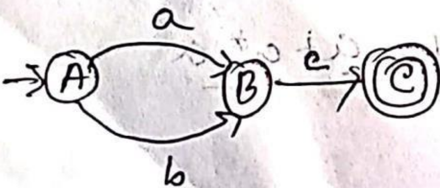
① ba^*b

$= bb, bab, baab$ ---

[① Whenever closer a^* make self loop
② whenever & operation make straigh state]

② $(a+b)c$ → [+ symbol মানে OR] Strings: ac, bc

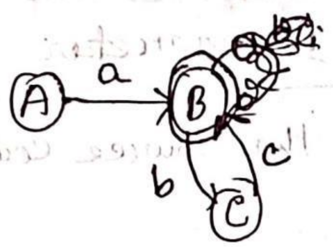




It means bc can be equal to zero or two more

So, a, abc, abcbc, abcbcbc

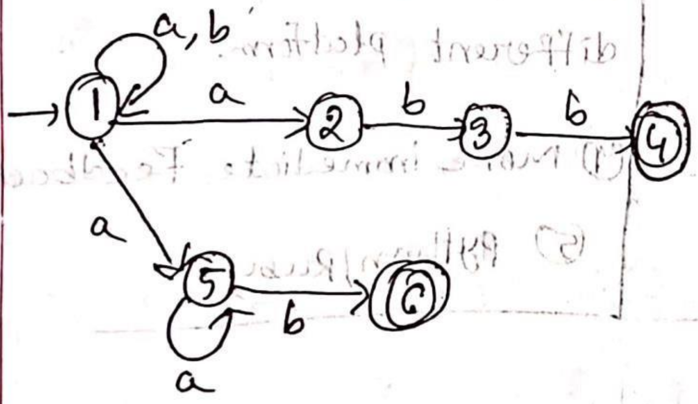
(3) $a(bc)^*$



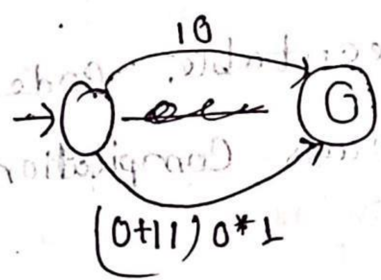
(4) $(a|b)^* . (abb \mid a^+ b)$ → [equal to '+']

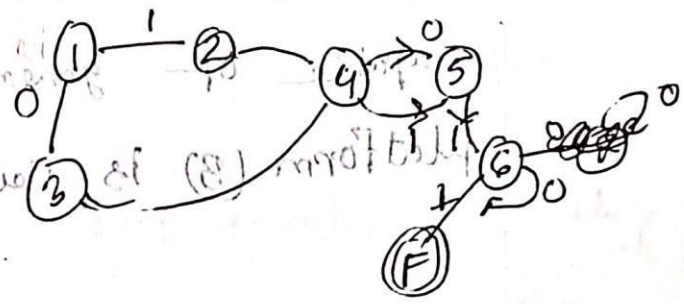
$[a^+ = \{a, aa, aaa...\}]$

$a^* = \{\epsilon, a, aa, aaa\}]$



(5) $10 + (0+11)0^*1$





Compiler vs Interpreter:-

Compiler	Interpreter
① Translates entire source code	① Reads the source code line by line.
② Compilation happens before execution.	② It happens in runtime
③ particular hardware/software	③ Can be executed on different platform.
④ Debugging challenging	④ More immediate Feedback.
⑤ C, C++, Rust	⑤ Python, Ruby

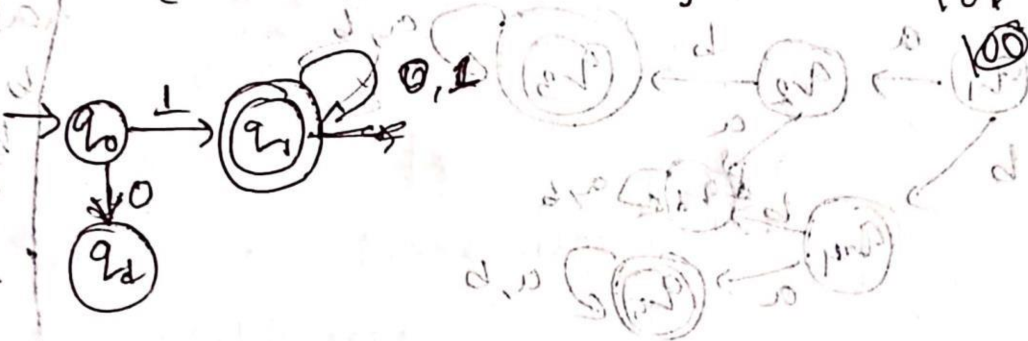
Cross Compiler:-

A compiler that's run on platform (A) & is capable of generating executable code for platform (B) is called a cross compilation.

DFA:

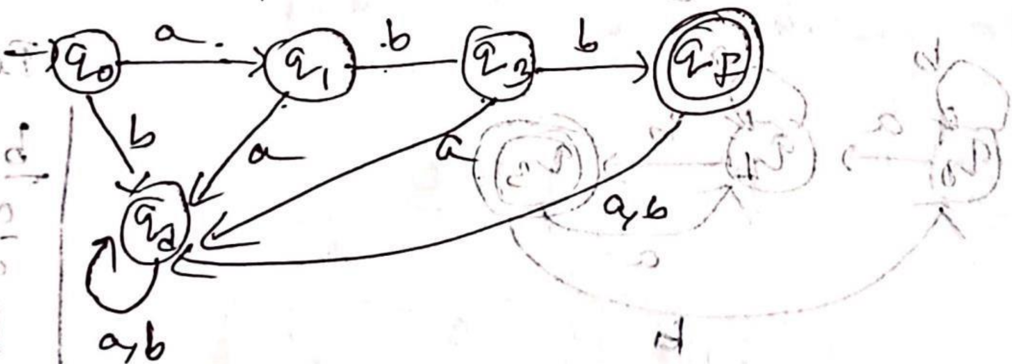
(*) Σ = {0, 1}

L = {w | w start with 1}



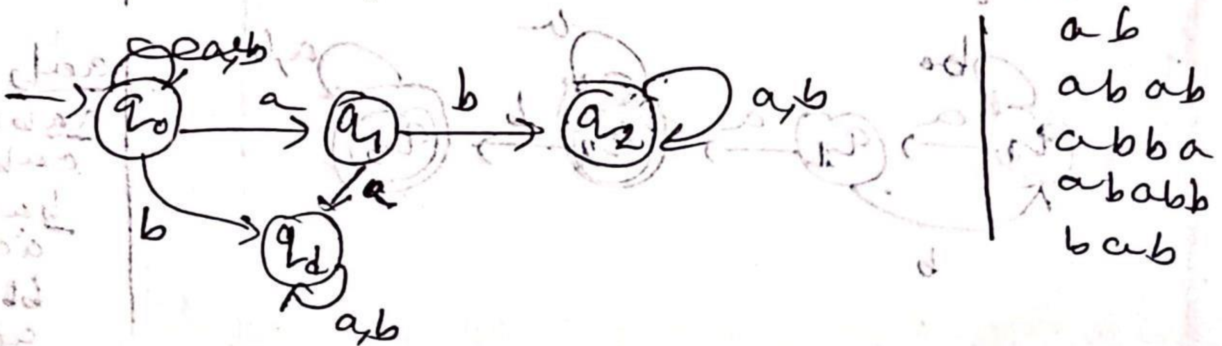
(*) Σ = {a, b}

L = {w | w accept only abb}



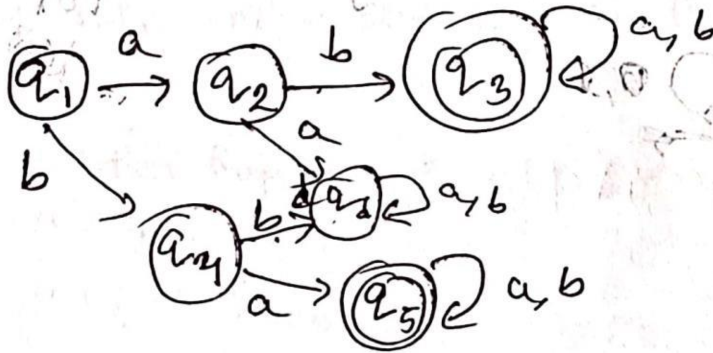
(*) Σ = {a, b}

L = {w | w only accept which starts with ab}



⊛ $\Sigma = \{a,b\}$

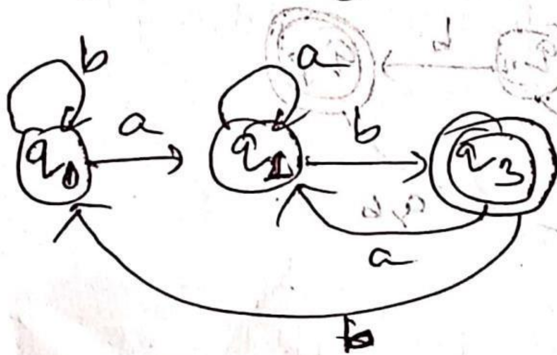
L = {w | w : Only accept which starts with ab/ba}



Test case
abba
baab
baba
abab
aaa
bbb
aab
bba

⊛ $\Sigma = \{a,b\}$

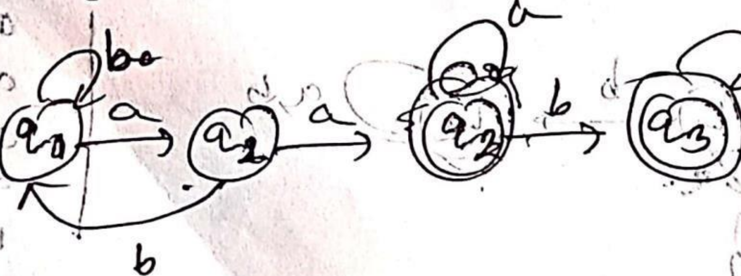
L = {w | w : only accept which ends with ab}



Test case
bab
aab
abab
baab
aaaba
bbab
abaab
bbabab

⊛ $\Sigma = \{a,b\}$

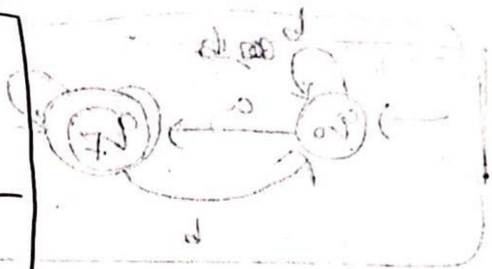
L = {w | w accept Contain aab}



aab
aaba
aabb
baab
aaab
bbaaab
aaaab

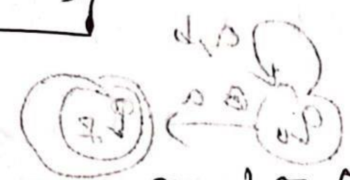
→ does not contain বললে, সুতরাং same ভাবে এঁকে ফেলবো

But Final state বাদে বাকি সবগুলোকে Final state হিসেবে count করতে হবে।



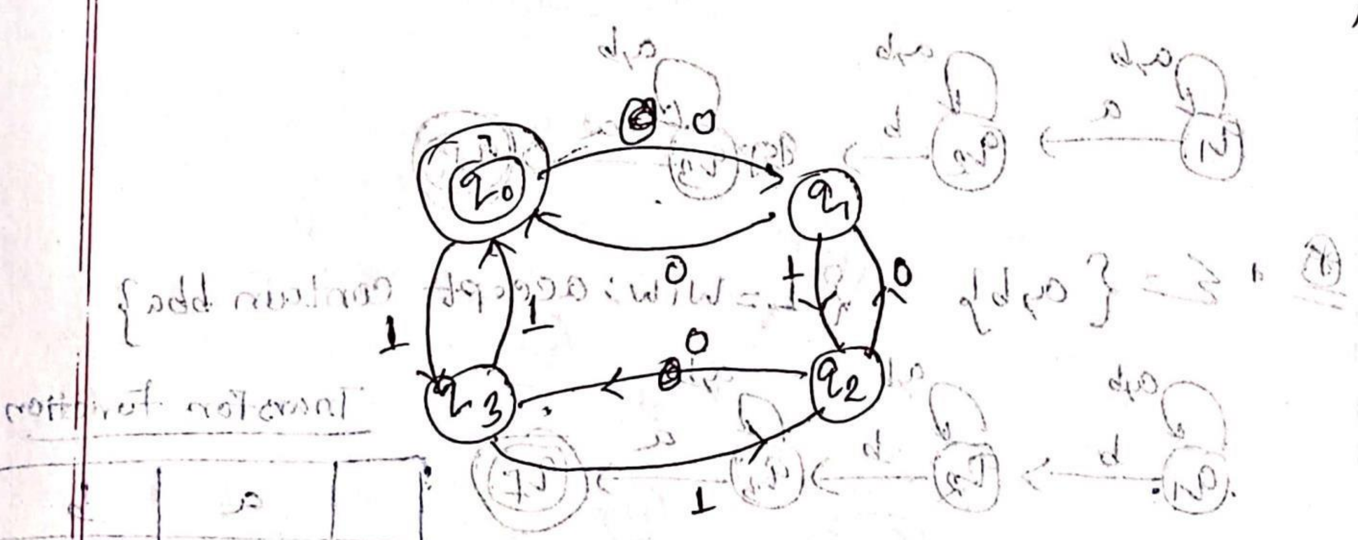
Regular Expressions:

- (+) হলে → divide
- তবে n → state বাড়বে
- * → self loop



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

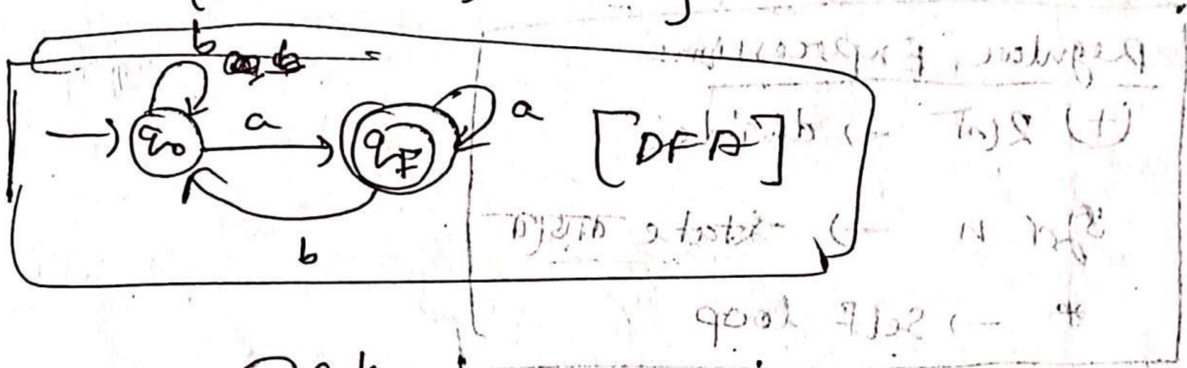
$L = \{w \mid w$ accept even number of 0's & even number of 1's$\}$



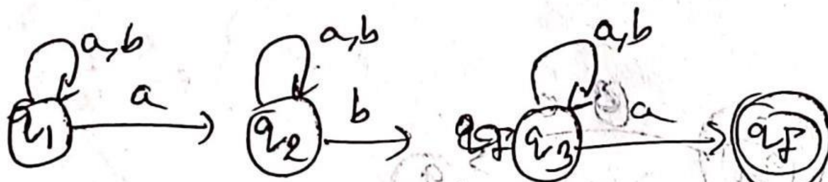
NFA:-

(Q) $\Sigma = \{a, b\}$

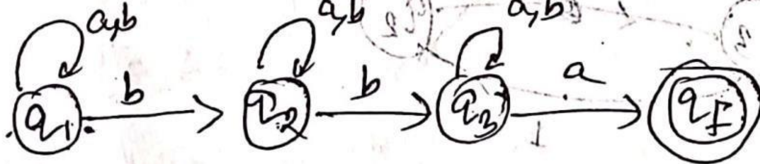
$L = \{w | w$ ends with $a\}$



(Q) $\Sigma = \{a, b\}$ $L = \{w | w$, ends with aba$\}$



(Q) $\Sigma = \{a, b\}$ $\{L = w | w$; accept contain bba$\}$

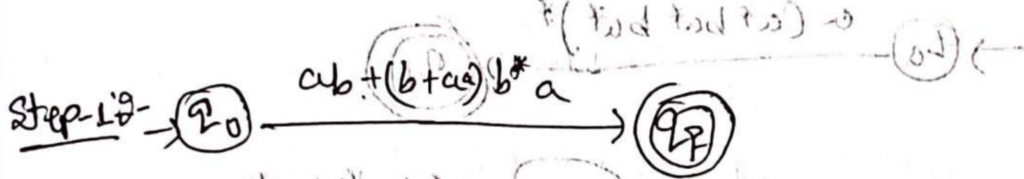


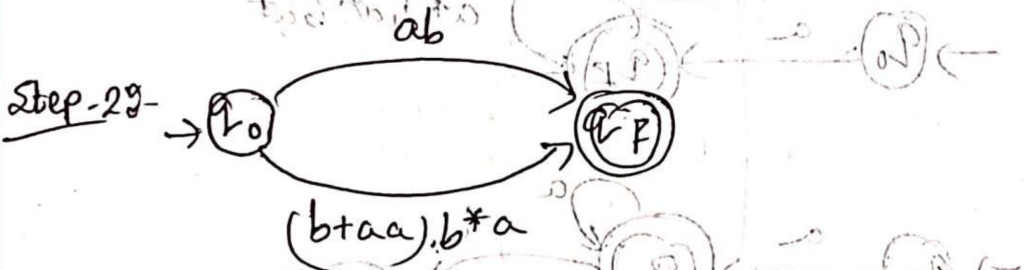
Transfor function

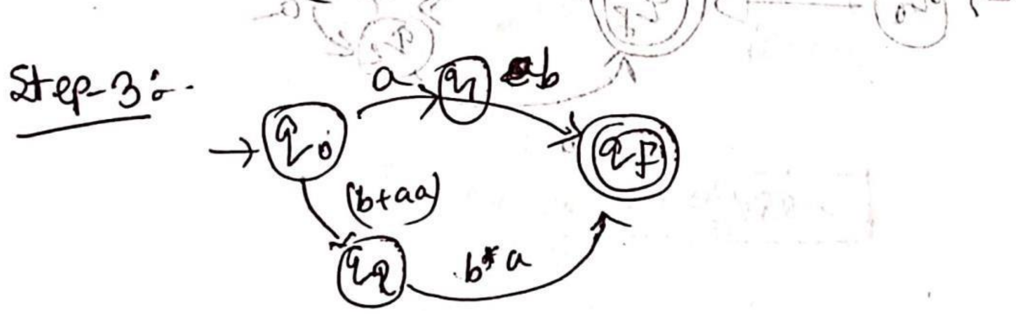
	a	b
q_1	q_1	q_2, q_1
q_2	q_2	q_2, q_3
q_3	—	—
	—	—
q_f		

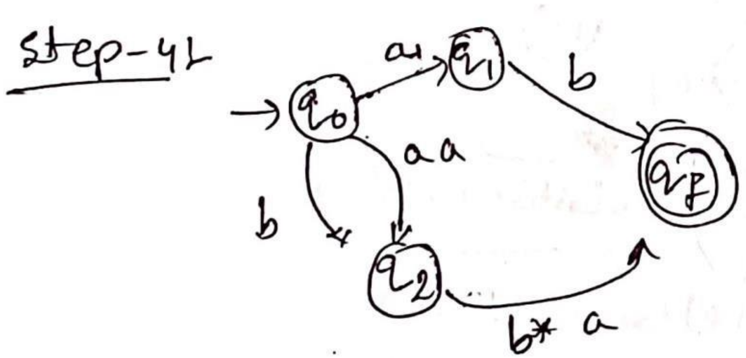
Regular Expression :-

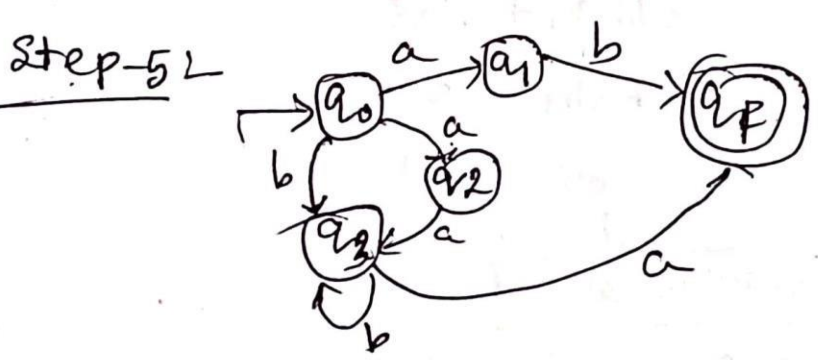
~~Gutam~~ [ab+ (b+aa)b*a]





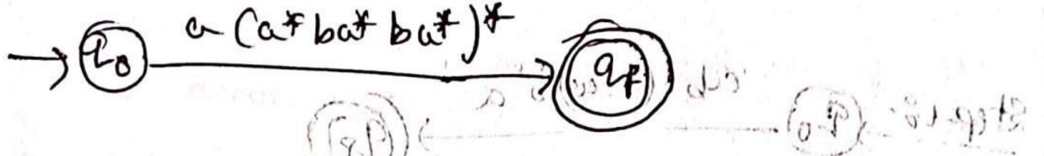






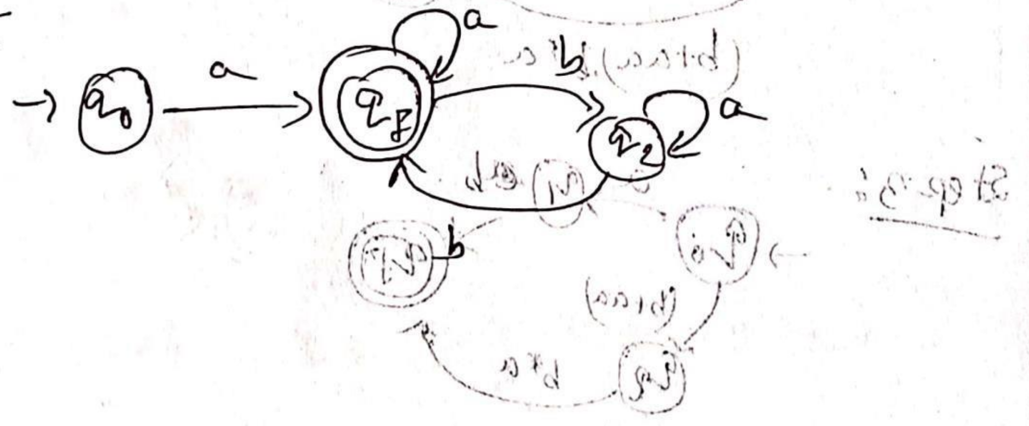
Example-02:- $[a(a^* ba^* ba^*)^*]$

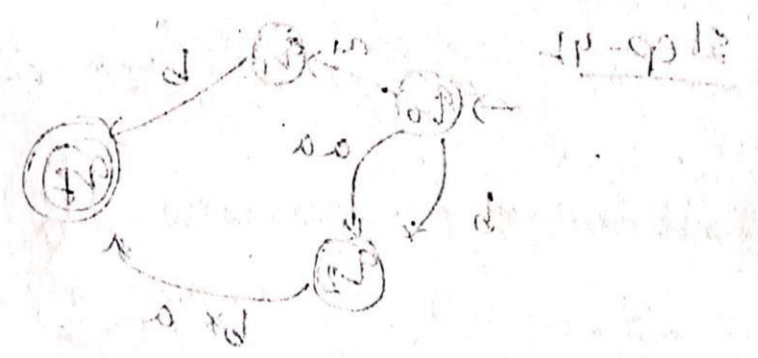
Step-1:

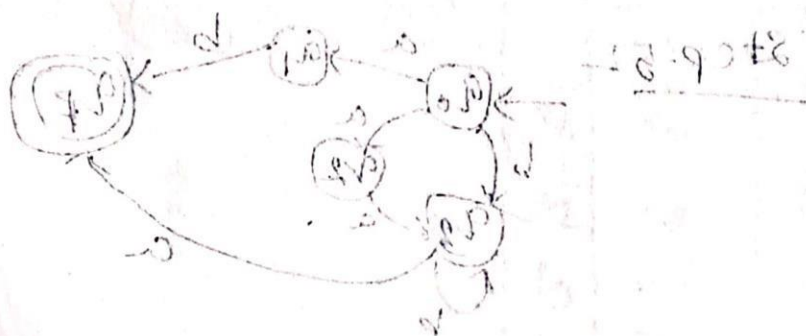


Step-2:

Step-3:







Translation of an assignment Statement :-

⇒ position = initial + rate * 60

↓

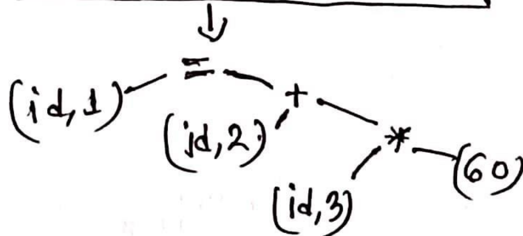
Lexical Analyzer

{id, 1} (=) {id, 2} + {id, 3} (*) (60)

↓

Syntax Analyzer

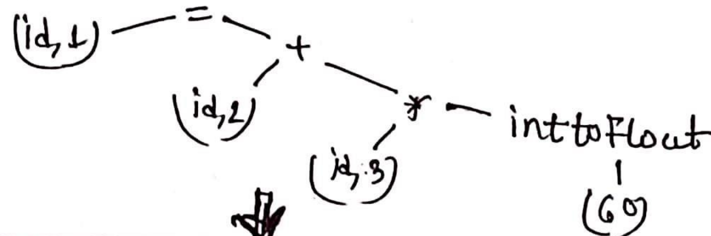
↓



↓

Semantic Analyzer

↓



↓

Intermediate Code Generator

↓

t1 = inttofloat(60)

t2 = t1 * id3

t3 = id2 + t2

id1 = t3

↓

Code optimizer → **Code Generator**

t1 = id3 * 60.0

id1 = id2 + t1

☐ Translation of an assignment Statement :-

⇒ position = initial + rate * 60

↓

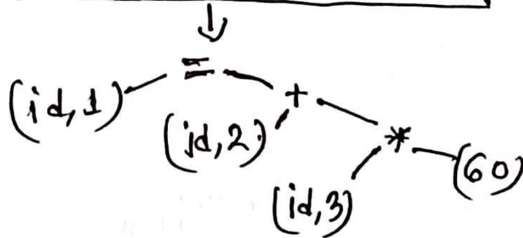
Lexical Analyzer

{id, 1} (=) {id, 2} + {id, 3} (*) (60)

↓

Syntax Analyzer

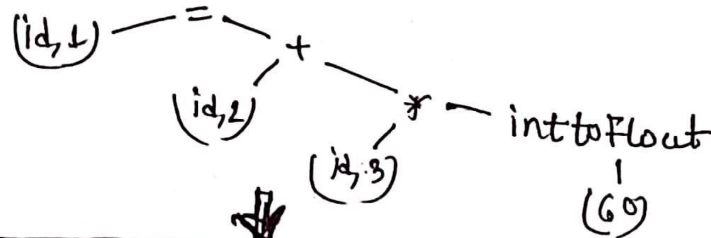
↓



↓

Semantic Analyzer

↓



↓

Intermediate Code Generator

↓

t_1 = inttofloat(60)

t_2 = t_1 * id3

t_3 = id2 + t_2

id1 = t_3

↓

Code optimizer → **Code Generator**

t_1 = id3 * 60.0

id1 = id2 + t_1

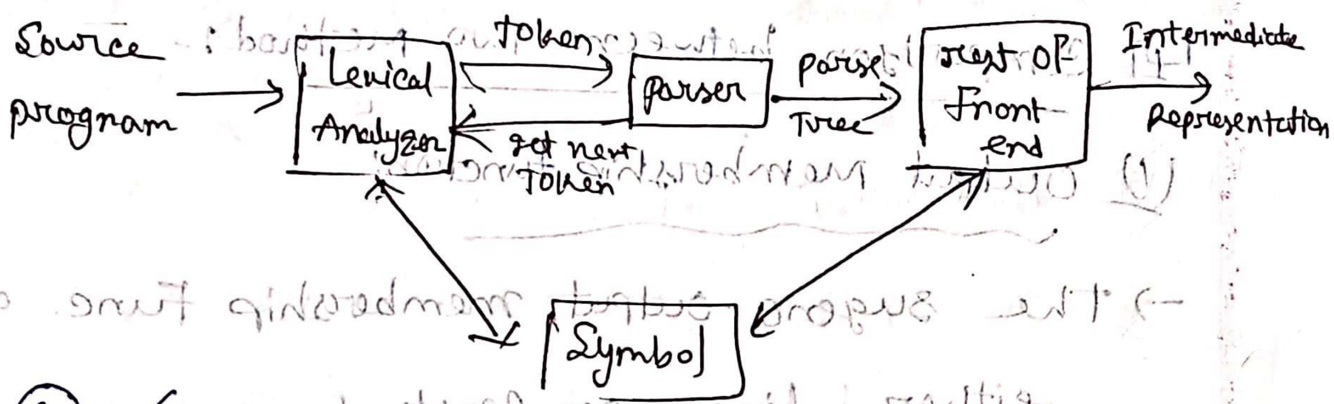
KEEP
CALM
ITS TIME FOR THE
FINAL
EXAM

Compiler (FINAL)

4) Parser:

→ Compiler or interpreter component that breaks data into smaller elements for easy translation into another language.

→ Takes input in the form of a sequence of tokens & usually builds a data structure in the form of a parse tree.



4) Error Recovery Strategies:

4 types:

① Panic mode

② Phrase level

③ Error production

④ Global Correction

① Panic Mode:-

→ Simplest method to implement & can be used by most parsing method.

→ if error is discovered, the parser discards input symbol one at a time until one of a designated set of synchronizing tokens is found.

② Phrase Level:-

→ Error occur, parser may perform local correction on the remaining input.

→ May replace prefix of the remaining input by some string that allows the parser to continue

③ Error production:-

→ Common error can occur.

→ Deginer can create augmented grammar to be used. As productions that generate erroneous constructs when these errors are encountered.

④ Global Correction:-

→ Few changes as possible for incorrect input string.

→ Choosing a minimal sequence of changes to obtain a globally least cost correction.

→ Too costly to implement

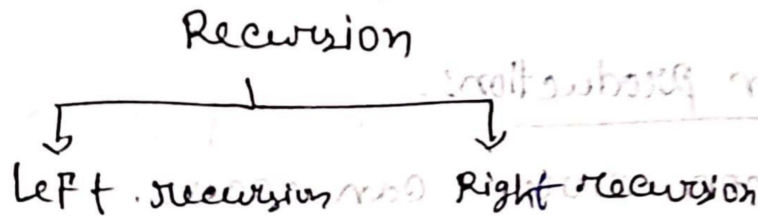
→ So, these are theoretical.

Algorithm of the left recursion :-

Ex:- $P \to P + Q \mid Q$ - - -

Answer :-

Recursion:- Function खुद Function को Call करता



$(A \to A\alpha \mid B$

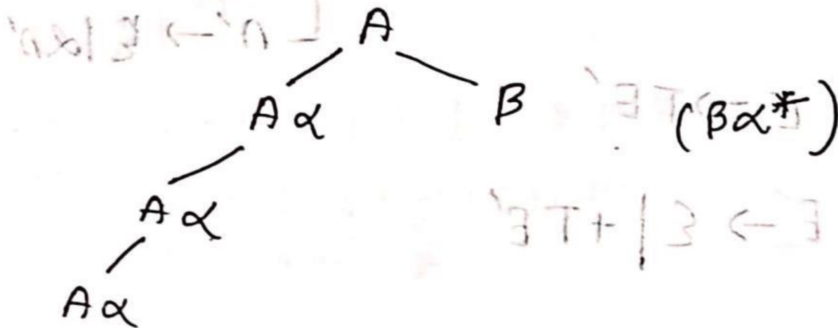
Same

$B \to B\alpha \mid B$

Same or LR

'Ter. ক → Small Letter
Non Ter → Capital ✓

• কারণ: $A \to A\alpha \mid \beta$

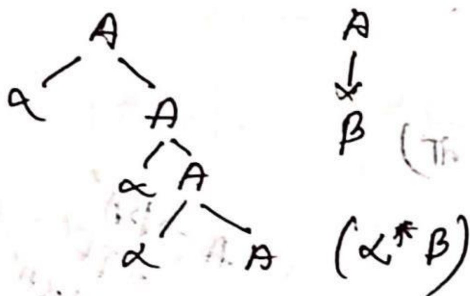


$(\beta\alpha^*)$

(Same ভাবে চলতে থাকে তাই left recursion.

Right Recursion:

$A \to \alpha A \mid \beta$



A ↓ β

$(\alpha^* \beta)$

(এটা TOP Down parsing)
TOP Down parser Infinite call এরকম করতে পারে না তাই এটা Left Rec. use করে না)

Infinity দূর করা:-

Formula:- $A \to A\alpha \mid \beta$

Ex: $A \to A\alpha \mid \beta$ [Target string $\beta\alpha^*$]

$A \to \beta A'$

$A' \to \varepsilon \mid \alpha A'$ ⇒ [মূলত এটা Formula]

(Left recursion)

Ex:- $E \rightarrow E + T \mid T$ (A = E, α = +T, β = T)

[Formula]
$A \rightarrow A\alpha \mid \beta$
$A \rightarrow \beta A'$
$A' \rightarrow \varepsilon \mid \alpha A'$

Solution:- $E \rightarrow TE'$

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

Ex: (2) $T \rightarrow T * F \mid F$ (A = T, α = *F, β = F)

$T \rightarrow FT'$

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

Ex: (3) $F \rightarrow (E) \mid id$

(Left recursion नहीं)

Ex-4: $E \rightarrow E + T \mid T$ (A = E, α = +T, β = T)

$T \rightarrow T * F \mid F$ (A = T, α = *F, β = F)

$F \rightarrow (E) \mid id$

$A \rightarrow \beta A'$
$= TE'$
$A' = \varepsilon \mid \alpha A'$

$\Rightarrow$ ~~E → E + T~~ $E \rightarrow TE'$ ✓

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

Again,

$T \rightarrow FT'$

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

(Ex: 5) $S \rightarrow (L) \mid a \mid b$ (Left Recursion नहीं)

$\underset{A}{L} \rightarrow \underset{A}{L}\,\underset{\alpha}{,S} \mid \underset{\beta}{S}$

Solution:-

$L \rightarrow SL'$

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

(F) ~~Aexz A~~ Formula OF multiple Left Recursion :-

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \Rightarrow A \rightarrow A\alpha \mid \beta$

$\beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$

$A \rightarrow \beta A' \Rightarrow A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \ldots \mid \beta_n A'$

$A' \rightarrow \epsilon \mid \alpha A' \Rightarrow A' \rightarrow \epsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A'$

Ex-1: $A \rightarrow A\beta\alpha \mid Aa \mid a$

$\underset{A}{A} \rightarrow \underset{A}{A}\,\underset{\alpha_1}{\beta\alpha} \mid \underset{A}{A}\,\underset{\alpha_2}{a} \mid \underset{\beta}{a}$

$A \to aA'$

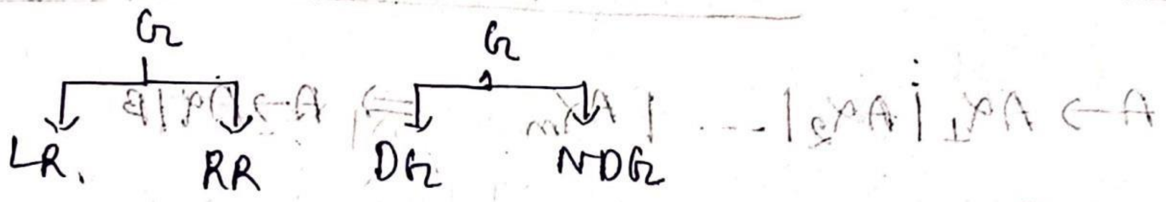
$A' \to \varepsilon \mid \beta\alpha A' \mid aA'$

Ex:-02 $A \to Ac \mid Aad \mid bd \mid c$

(underneath: A, $A\ \alpha_1$, $A\ \alpha_2$, β_1, β_2)

$A = bdA' \mid cA'$

$A' = \varepsilon \mid cA' \mid adA'$



G → LR, RR

G → DG, NDG

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$

$\Rightarrow A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \beta_n$

[প্রথমে Common α এরপর বাকি অংশের এরজন্য A']

Ex-01:

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

$E \to b$

Formula:

$A \to \alpha B_1 \mid \alpha B_2$

$A \to \alpha A'$

$A' \to B_1 \mid B_2$

[Formula]

$\Rightarrow S \to IEtSS' \mid a$

$S' \to \varepsilon \mid eS$

$E \to b$

Ex-02:

$A \to aAB \mid aA$

$B \to bB \mid b$

Solution:- $A \to aA'$

$A' \to AB \mid A$

Ex-03 $A \to aAB \mid aA \mid a \mid \varepsilon$

$A \to aA'$

$A' \to AB \mid A \mid \varepsilon$

Ex:-04 $E \rightarrow T+E \mid T \cdot \varepsilon$

$E \rightarrow TE'$

$E' \rightarrow +E \mid \varepsilon$

Ex-05: $S \rightarrow bSSaaS \mid bSSasb \mid bSb \mid a \cdot \varepsilon$

Step-1

$S \rightarrow bSS' \mid a$

$S' \rightarrow SaaS \mid Sasb \mid b$

Step-2:

$S \rightarrow bSS' \mid a$

$S' \rightarrow SaS'' \mid b \mid \varepsilon$

$S'' \rightarrow aS \mid Sb$

Show the model of a non-recursive predictive parser:

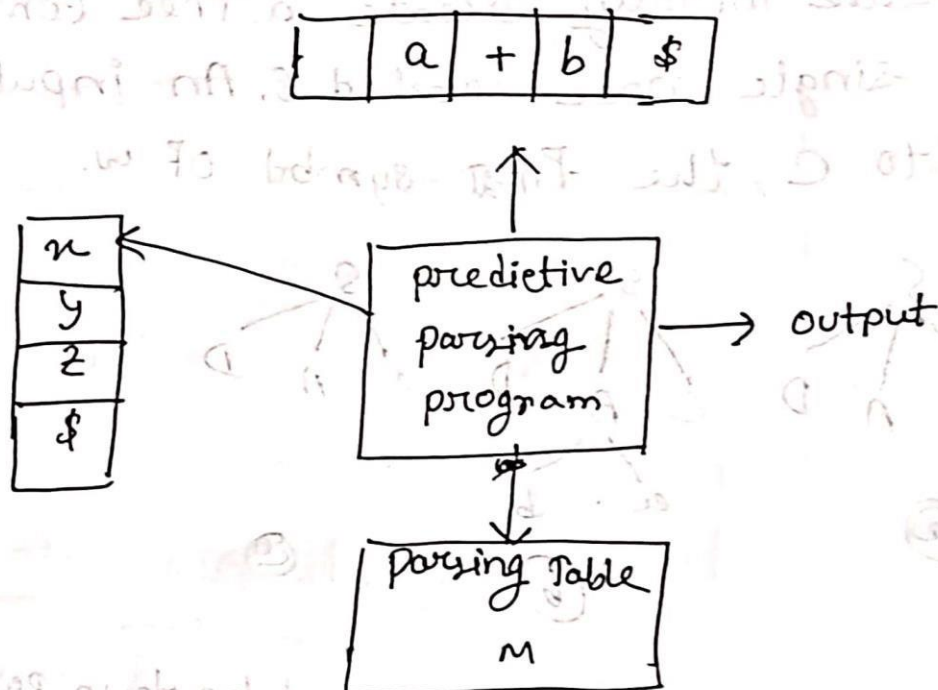


Fig:- A Non-recursive predictive parser

Step of top-down parsing, Draw the position of the parser.

⟹ Consider the grammer

$S \to cAd$

$A \to ab \mid a$

And the input, w = cad. To Construct the parse tree for this string top down, we initially create a tree consisting of a single node labeled S. An input point to c; the first symbol of w.

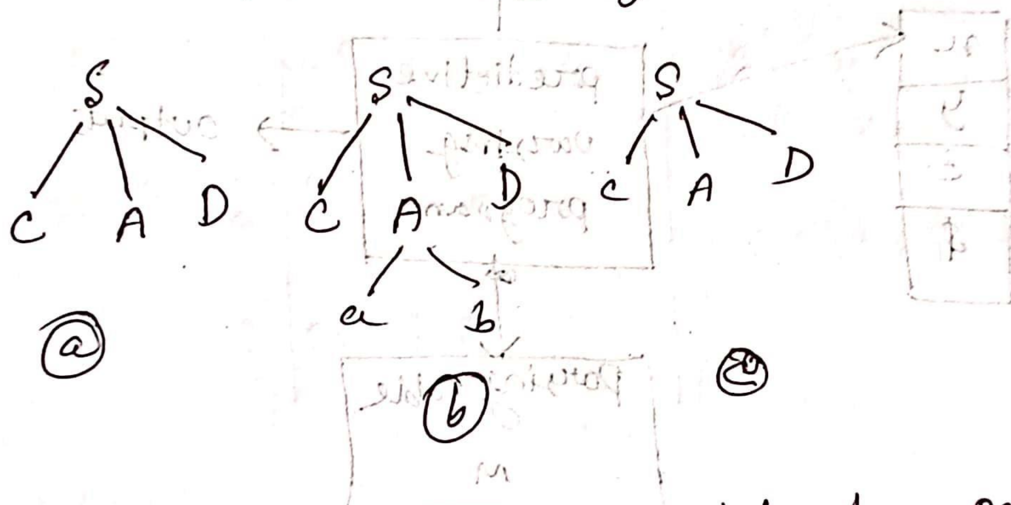


Fig:- Steps in the top-down parse

Topic: First & Follow

→ Left Recursion & Left Factoring থাকলে না

	FIRST	Follow

Rules:- (FIRST)

$A \to aB$ (a → Terminal)

$FIRST(A) = \{a\}$

$A \to a \quad \{a\}$

① $A \to a \mid \epsilon$

$A \to a$

$A \to \epsilon \quad \therefore FIRST(A) = \{a, \epsilon\}$

② $A \to (aB) \mid \epsilon$ → First এ আছে যে terminal

$FIRST(A) = \{(, \epsilon\}$ → Second টির First ε

③ $A \rightarrow Ta$

$T \rightarrow *FT'$

$FIRST(A) = FIRST(T)$ → কারণ T Non-Terminal

$= \{*\}$

④ $A \rightarrow Ta$

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

$FIRST(A) = FIRST(T)$

$= \{*, \epsilon\}$

$= \{*, a\}$

⑤

→ Follow(E) কারণ E' এর পরে আর কেউ নেই

	FIRST	Follow
$E \to TE'$	{id, (}	{\$,)}
$E' \to +TE' \mid \varepsilon$	{+, ε}	{\$,)}
$T \to FT'$	{id, (}	{+, \$,)}
$T' \to *FT' \mid \varepsilon$	{*, ε}	{+, \$,)}
$F \to id \mid (E)$	{id, (}	{*, +, \$,)}

~~FOR~~

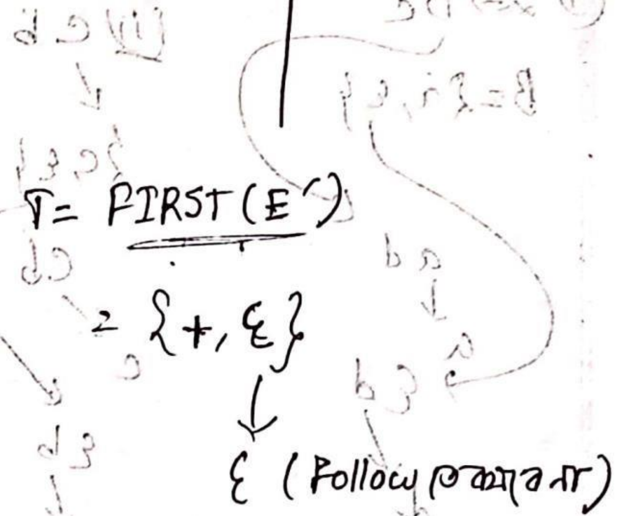
FIRST(E) = TE'

FIRST(T) = FT'

FIRST(F) = {id, (}

Follow(F) = FIRST(T')

= {*, ε}



T = FIRST(E')

= {+, ε}

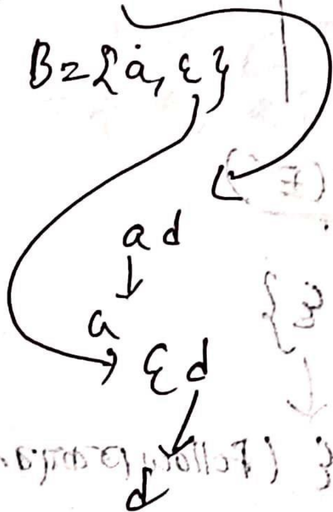
ε (Follow দেখতে হবে)

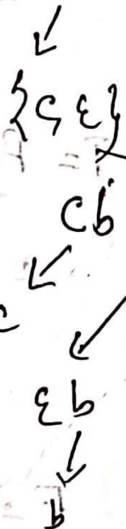
E = TE' → [E' এ ε বসালে ফাঁকা হয় তার মানে T এর Follow হবে E। আর E এর Follow বুঝেছে]

Ex-02

	FIRST	FOLLOW
$S \to Bbd \mid Cb$	{a, d, c, b}	{\$}
$B \to aB \mid \varepsilon$	{a, ε}	{d}
$C \to eC \mid \varepsilon$	{e, ε}	{b}

right এ কোনো S নাই





Ex-03

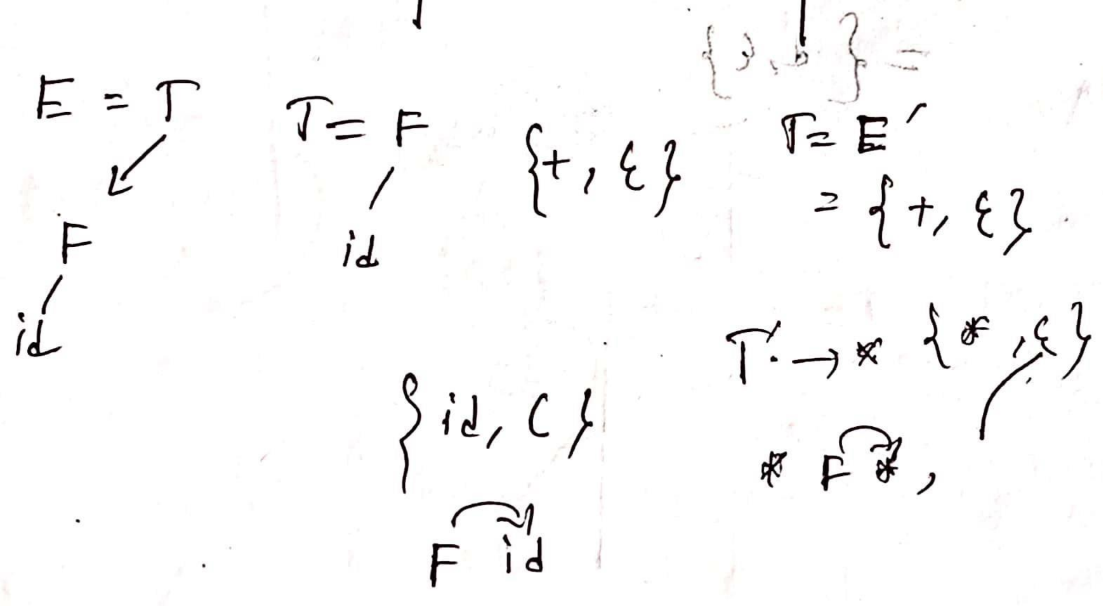
	FIRST	FOLLOW
$S \rightarrow aABbc$	{a}	{$}
$A \rightarrow c \mid \varepsilon$	{c, ε}	{d, b}
$B \rightarrow d \mid \varepsilon$	{d, ε}	{b}

Follow(A) = FIRST(B)

= {d, ε}

Topic :- LL(1)

	FIRST	FOLLOW
E → TE'	{ id, (}	{ $,) }
E' → +TE' | ε	{ +, ε }	{ $,) }
T → FT'	{ id, (}	{ +, $,) }
T' → *FT' | ε	{ *, ε }	{ +, $,) }
F → id | (E)	{ id, (}	{ *, +, $,) }



	id	+	*	(	)	$
E	E→TE′			E→TE′		
E′		E′→+TE′			E′→ε	E′→ε
T	T→FT′			T→FT′		
T′		T′→*ε	T′→*FT′	T′→ε		T′→ε
F	F→id			F→(E)		

E′→ε → হলে E′ এর follow বরাবর বসবে।

Define LL(1) & LR(K) parser:

LR parser:- A type of bottom up parsers that efficiently handle deterministic content free language in guaranteed linear time.

Left Recursion:-

The production is left recursive if the leftmost symbol on the right side is the same the non terminal on the left side.

ex: $A \rightarrow A\alpha \mid \beta$

Left Factoring:- Consist in factoring out prefix which are common two or more production.

Top-Down parsing:-

⇒ An attempt to find a leftmost derivation of an input string.

→ Start from root node &

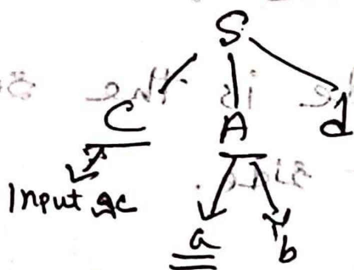
→ May need back tracking

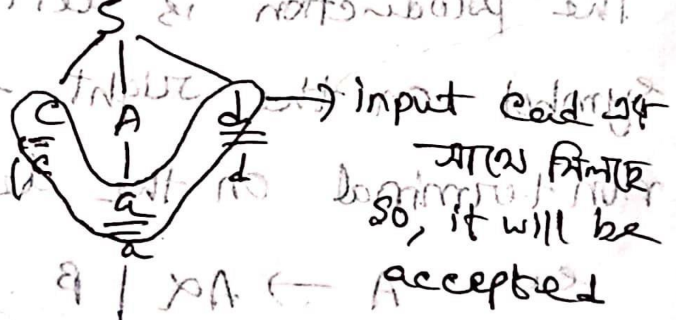
ex:- Let us consider the following grammar

S → cAd

A → ab|a

[Let input string, w = cad]





TOP-Down Parsing Math

VVIm (F)

Consider a grammar:-

$E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to id \mid (E)$

(F) Also apply LL(1) for the String (id + id) → Already done

⇒ ~~Stac~~

TOP-Down parsing

Stack	input	production
E $	id+id $	$E \to TE'$
TE' $	id+id $	$T \to FT'$
FT'E' $	id+id $	$F \to id$
id T'E' $	id + id $	~~(F) → id~~ pop
T'E' $	+id $	$T' \to \varepsilon$
~~FT'E' $~~	~~+id~~	~~F → id~~

Q1

Stack	input	production
$E'\$$	$+id\$$	$E' \to +TE'$
$+TE'\$$	$+id\$$	pop
~~$TE'\$$~~	~~id~~	~~$E' \to +TE'$~~
~~$+TE'\$$~~	~~id~~	
$TE'\$$	$id\$$	$T \to FT'$
$FT'E'\$$	$id\$$	$F \to id$
$idT'E'\$$	$id\$$	pop
$T'E'\$$	$\$$	$T \to \varepsilon$
$E'\$$	$\$$	$E' \to \varepsilon$
$\$$	$\$$	Accepted

Bottom up parsing:-

→ নিচ হতে উপরে যেতে হবে।

→ Right side - এর গুলোর

Ex:

$E \to E + T$

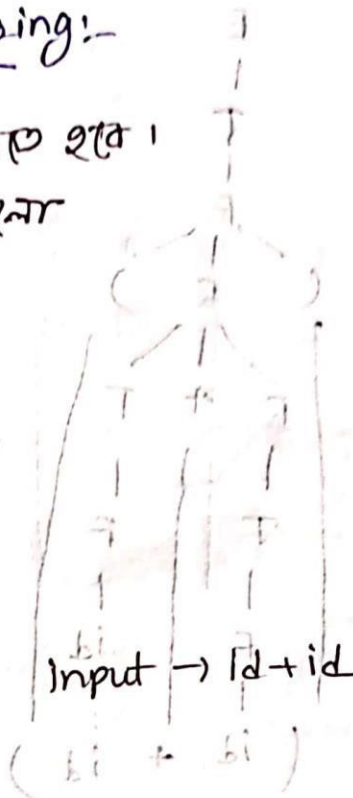
$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to id$



Input → id+id

Solution

Rules used	Right sentential Form
$E \to T$	(E)
$T \to F$	(T)
$F \to (E)$	(F)
$E \to E+T$	(E)
$E \to T$	$(E+T)$
$F \to id$	$(E+F)$
$E \to T$	$(E+id)$
$T \to F$	$(T+id)$
$F \to id$	$(F+id)$
	$(id+id)$

E
|
T
|
F
(E)
E + T
| |
T F
| |
F id
|
(id + id)

LR(0):

E → T+E / T

T → id

Solution:

Item-0:

E' → .

E → .T

E → .T

T → .

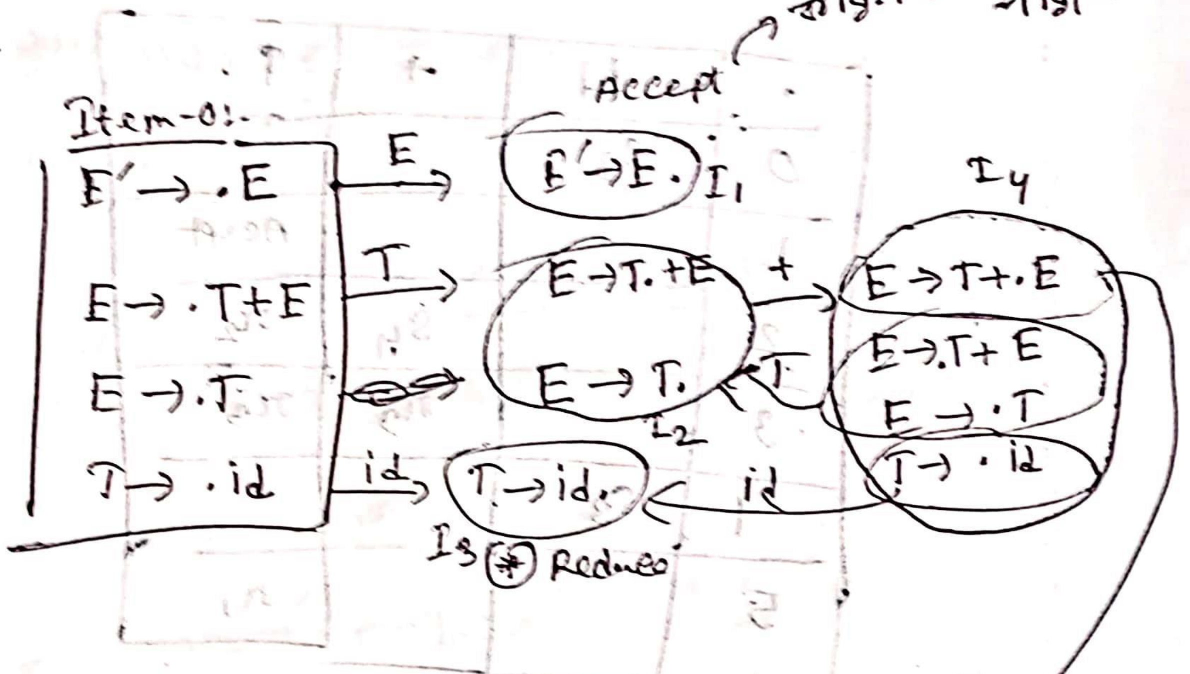
State	
0	
1	
2	
3	
4	
5	

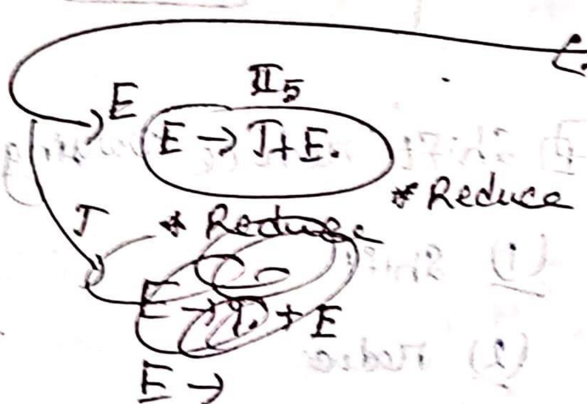
LR(0) :-

$E \rightarrow T+E / T$

$T \rightarrow id$

Solution:





State	id + $	E	T
0	S_3	1	2
1	Acept		
2	r_2 (S_4/r_2) r_2		
3	r_3 r_3 r_3		
4	S_3	5	2
5	r_1 r_1 r_1		

Here, For state 2 S_4/r_2 That means it is not LR(0).

SLR(1):-

Using prev ex:-

	id	+	$
0	S_3		
1			Accept
2		S_4	r_2
3		r_3	r_3
4	S_3		
5			r_1

Shift reduce parsing can make 4 things

(1) Shift

(2) reduce

(3) Accept

(4) Error

2 more things: (1) Stack

(2) Input Buffer

2 conflict:- (1) S/R

(2) RR

LR(0) :-

4 টা Rule :-

(1) Find augmented grammar

(2) I_0 = Closure

(3)

Part - B (Math part)

Syntax Directed Defination :-

SDD = Grammar + Attribute

$E \rightarrow T+F$

Theory [Part A (বাকিগুলো)]

Shift Reduce parsing :-

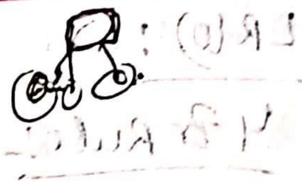
→ process of reducing a string w to the start symbol of a grammar

→ A particular substring matching RHS of a production rule is replaced by the symbol on the LHS of the production.

ex. - LR parsing.

Sakur_E201037

Part-B



(F) * Synthesized Attribute:-

→ If its value at a parent node can be determined from attributes of its children.

(F) * Inherited Attribute:- The value of an Inherited att. is determined from the value of attributes at the sibling & parent of that node.

W Im (F) * DAG:- A directed a cycle Graph for an expression identified the common sub expression in the expression.

(F) * Code Generator & its Issue:

→ Code generator:-

→ Final phase of compiler

→ Takes input an intermediate representation of the source code. And produce equivalent output for that.

Issues!

(i) Input to CG

(ii) Target program

(iii) Memory Management

(iv) Instruction Selection.

(v) Register allocation.

(vi) evolution order

(vii) Approaches to code generator

Type checks:- Compiler should report an error if an ~~e~~ error of an operator applied to an incompatible operand. This is called type checking.

[S-R, R-R, Intermediate code generator diagram, Front, end, back end Compiler]

~~S-R~~ Conflict:- Shift reduce ~~comp~~ conflict. Compiler বুঝতে পারে না যে Shift use হবে নাকি Reduce use হবে। এই দুই জিনিস একত্র হয়ে গেলে তাকে S-R conflict বলে।

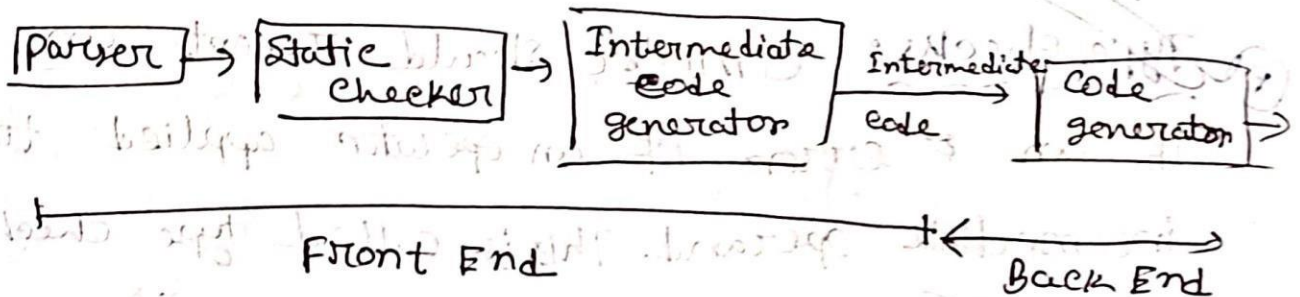
R-R conflict:- Reduce - Reduce conflict একত্র হয়ে গেলে তাকে R-R Conflict বলে। →

আর ঝামেলা হতে বাঁচতে LR(0) এর পরিবর্তে SLR use করা হয়।

Intermediate Code Generator Diagram:

=> ~~Easier to apply source~~

=> process of converting source code into an intermediate representation that is easier for compilers to analyze & optimize before generating target code.



Front End:- Translates source code into an intermediate representation

Back end:- Uses representation to produce code in a computer output language.

Part-B (Maithi)

[4] Syntax Directed Definition:-

SDD = Grammar + Attribute

E = T+F

F → 1

SDD:- A Context Free grammar along with an attribute & rule.

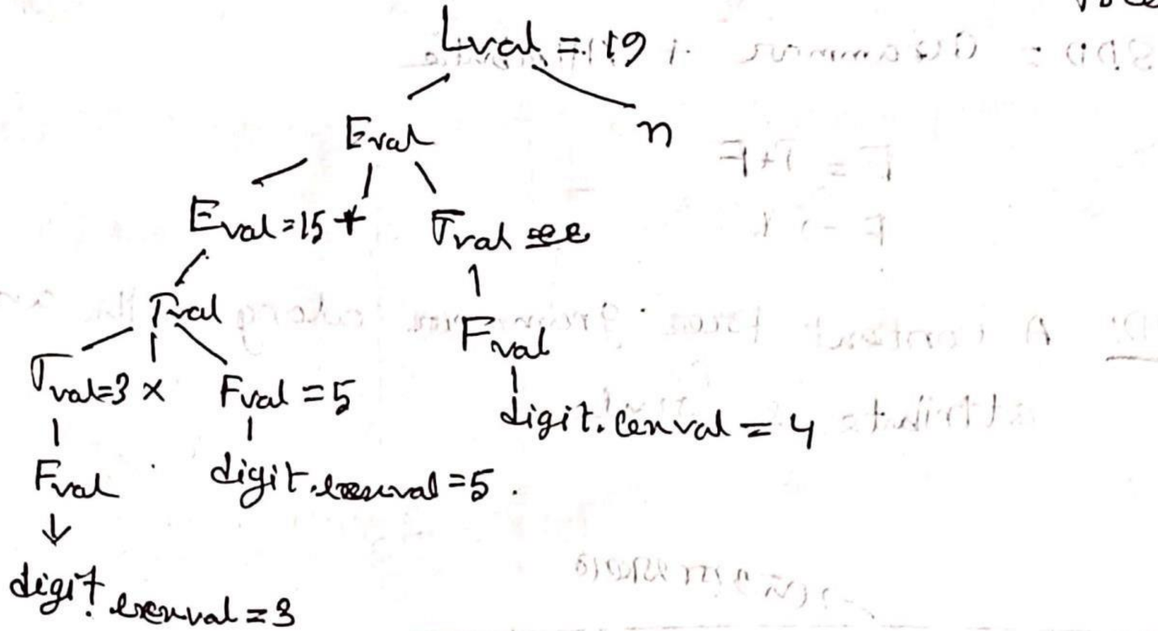
→ সেমান্টিক রুলস

Grammar	Semantic Rule
L → E	Lvalue → Evalue
E → E+T	Eval → Eval + Tval
T → T×F	Tval → Tval × Fval
T → F	Tval → Fval
F → (E)	Fval → Eval
F → digit	Fval → digit lexval

(Book-319 page) → Wim

(F)

3×5+4n এর Parse Tree:- (Anoted parse Tree)



Construction of Syntax tree:-

	Semantic rule
1) E → E+T	E.node = new node(+ ; E.node, T.node)
2) E → E-T	E.node = new node(- , E.node, T.node)
3) E → T	E.node = T.node
4) T → (E)	T.node = E.node
5) T → id	T.node = new leaf(id, id.entry)
6) T → num	T.node → new leaf(num, num.val)

a-4+c make syntax tree for this string:-

E.node

E.node + T.node

E.node - T.node id

T.node E.node

id

Fig:- Syntax Tree

① P_1 = new leaf (id, entry-a)

② P_2 = new leaf (num, 4)

③ P_3 = new node('-', P_1, P_2)

④ P_4 = new leaf (id, entry-c)

⑤ P_5 = new node('+', P_3, P_4)

~~Three Address Code:~~

Three Address Code:

Backtracking:

Type checker is a module of a compiler developed to type checking task.

- int
- Float
- Index; array

Type: 2 types:-

- Static
- Dynamic

DAG

4. Three Address Code:-

Is built From two Concept address and instruction.

```
do (i = i + 1)          t1 = i + 1
{ i*8                   i
while (a[i] < n)        i = t1
                        t2 = i*8
                        t3 = a[i] < n
```

$a = x * -y + x * -y$ এর Quadraples, triples, Indirect tiples বের কর।

⇒ three address code: Quadraple

		OP	arg1	arg2	result
$t_1 = -y$	0	−	y		t_1
$t_2 = x * t_1$	1	*	x	t_1	t_2
$t_3 = -y$	2	−	y		t_3
$t_4 = x * t_3$	3	*	x	t_3	t_4
$t_5 = t_2 + t_4$	4	+	t_2	t_4	t_5
$a = t_5$	5	=	t_5		a

Tripple

	OP	arg 1	arg 2
0	−	y	
1	*	x	(0)
2	−	y	
3	*	x	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect triples:- (101 থেকে start হবে)

101	(0)
102	(1)
103	(2)
104	(3)
105	(4)
106	(5)

④ Basic Blocks:-

(1) The First three address instruction in the intermediate code generator is a leader

(2) Any instruction that is the target of a conditional & unconditional Jump is a leader

(3) Any instruction that immediately follows a conditional or unconditional is a leader.

Find or always leader

B_1 * (1) i = 1

B_2 * (2) j = 1

B_3 * (3) t_1 = 10 * i

(4) $t_2 = t_1 + j$

(5) $t_3 = 8 * t_2$

(6) $t_4 = t_3 - 88$

(7) a[t_4] = 0.0

(8) j = j + 1

(9) if j <= 10 goto (3)

B_4

* (10) i = i + 1

(11) if i <= 10 goto (2)

B_5 * (12) i = 1

B_6

* (13) $t_5 = i - 1$

(14) $t_6 = 88 * t_5$

(15) a[t_6] = 1.0

(16) i = i + 1

(17) if i <= 10 goto (13)

[Leader Find out করতে হবে]

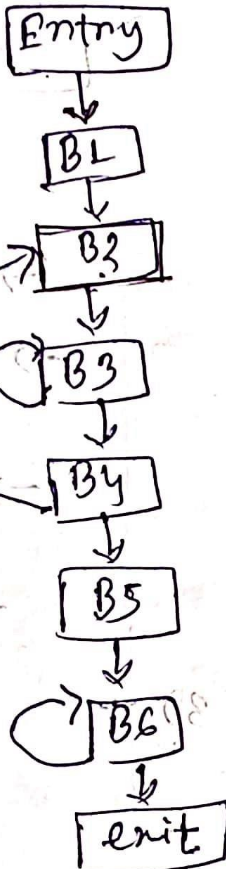
→ প্রথমটি leader

→ condition check করে leader বের করো

→ Block Leader হতে পরের আগের টা পর্যন্ত

→ Same block এ গেলে self loop

Basic Block Diagram



DAG: ~~[struck out]~~ parse tree। কোন step হতে কোন step এ যাচ্ছে তা।

Instruction cost:- (Book-8.2.6

Shift Reduce parsing:-

⇒ Process of reducing a string to the start symbol of a grammer.

string $\xrightarrow{\text{reduce to}}$ the starting symbol.