Unit-2

Syntax Analysis

→ A Parsing of Syntax Analysis is a Process which takes an input string 'w' and produces either a parse tree or generates the syntax free grammar

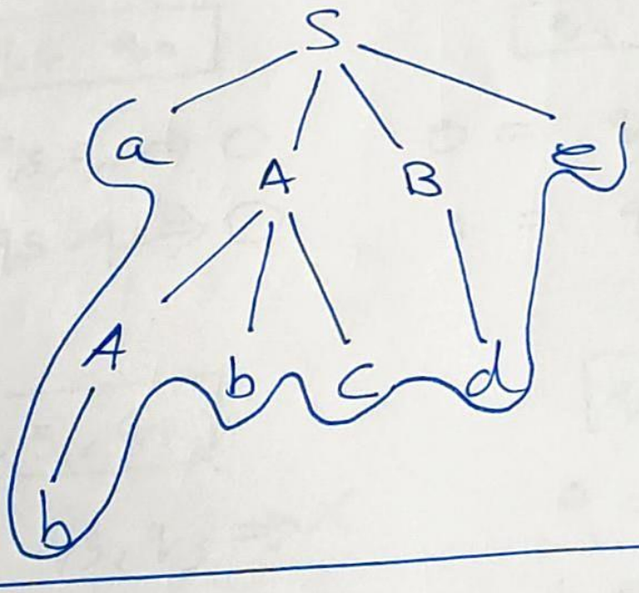
$G = \{V, T, P, S\}$ [Content free grammer]

- V → non terminals
- T → Terminals
- P → Production Rules
- S → Start symbol.

(whichever on the left side is the non-terminals)

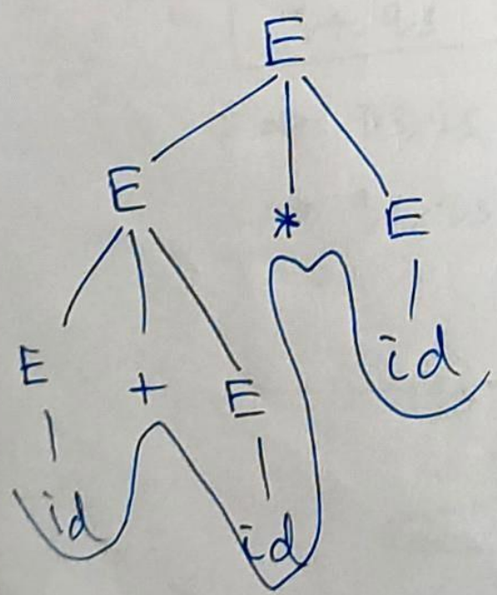
1] $S \rightarrow aABe$
$A \rightarrow Abc / b$
$B \rightarrow d.$

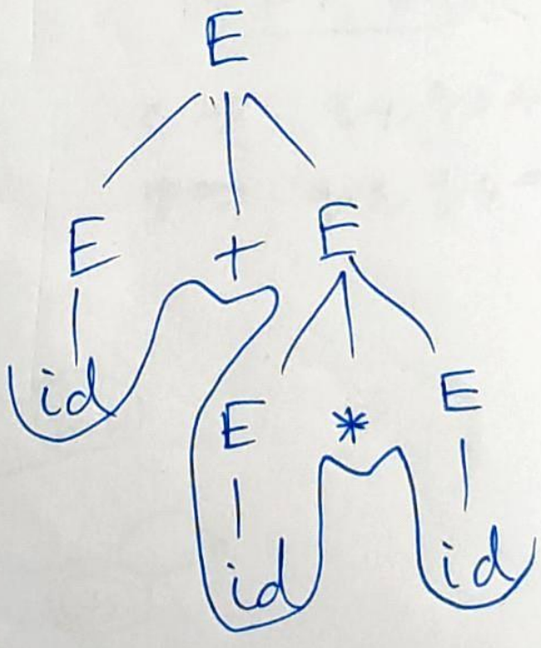
i/p → abbcde.



2] $E \rightarrow E+E / E*E / id$

i/p → id + id * id



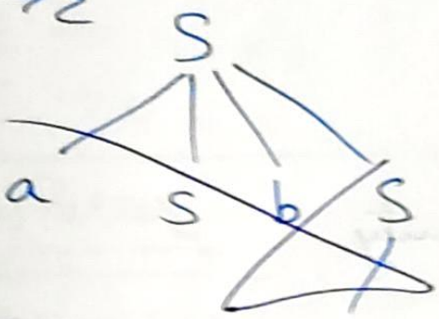


3] $S \to aSbS$

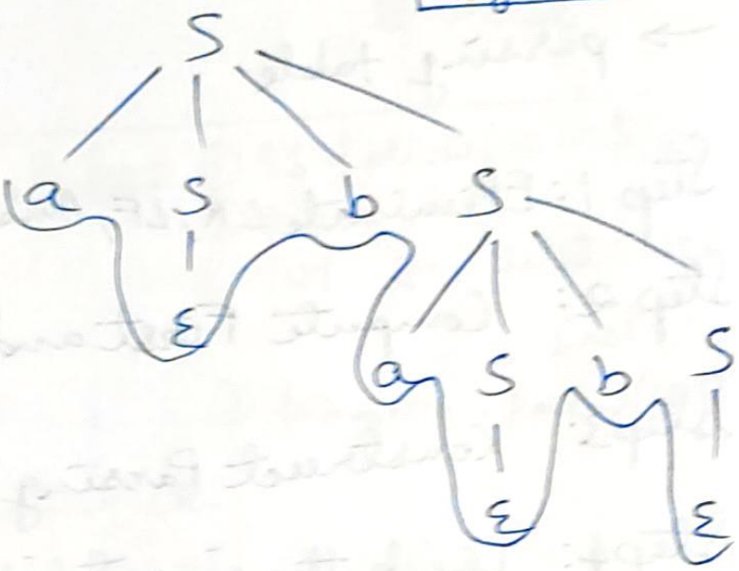
$S \to bSaS$

$S \to \varepsilon$

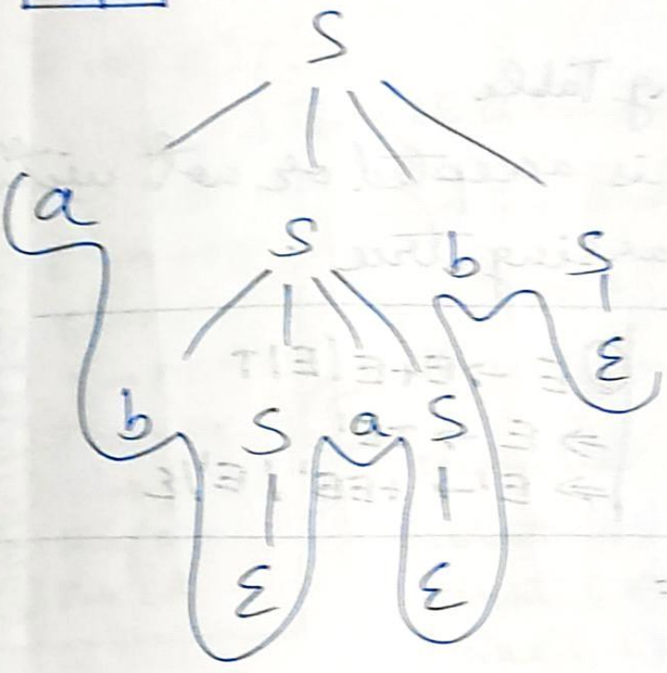
i/p → abab



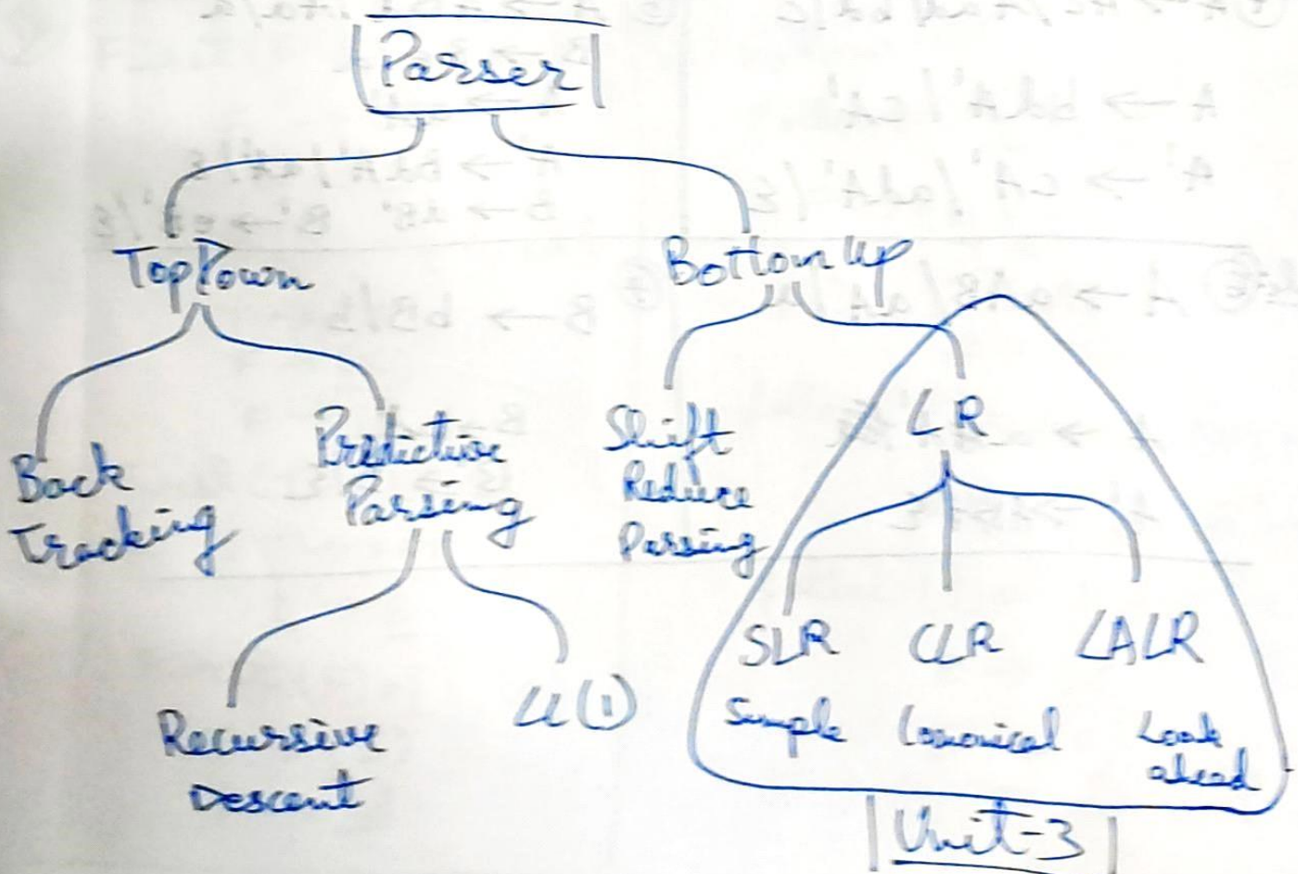
Right



Left



Types of Parsing :



LL(1) Parsing

→ ip tape
→ Stack
→ parsing table

Step 1: Eliminate LR, LF, Ambiguity

Step 2: Compute First and Follow.

Step 3: Construct Parsing Table

Step 4: Check the input is accepted or not with the help of the parsing tree.

Example: ① $E \to E+T/T$

$\Rightarrow E \to TE'$

$\Rightarrow E' \to +TE'/\varepsilon$

② $E \to E+E/E/T$

$\Rightarrow E \to TE'$

$\Rightarrow E' \to +EE'/E'/\varepsilon$

Example: ③ $E \to E+E/E*E/T/F$

$\Rightarrow E \to TE'/FE'$

$\Rightarrow E' \to +EE'/*EE'/\varepsilon$

Example: ④ $A \to Ac/Aad/bd/c$

$A \to bdA'/cA'$

$A' \to cA'/adA'/\varepsilon$

⑤ $A \to ABd/Aa/a$

$B \to Be/d$

$A \to aA'$

$A' \to BdA'/aA'/\varepsilon$

$B \to dB' \quad B' \to eB'/\varepsilon$

Example: ⑥ $A \to aAB/aA/a$

$A \to aA'$

$A' \to AB/A/\varepsilon$

⑦ $B \to bB/b$

$B \to bB'$

$B' \to B/\varepsilon$

Ex8: $S \to iEtS / iEtSeS / a$

$E \to b$

$\Rightarrow S \to iEtSS' / a$

$S' \to \varepsilon / eS$

$E \to b$

S → Statement
i → if
E → expression
t → term
e → else

LL(1) Parser

$E \to E + T / T$

$T \to T * F / F$

$F \to (E) / id$

Rules: ① SS follow(SS) = $

② follow(NT)

NT → NT, first(NT)

or

T, first(T)

③ follow(NT) = follow(LHS)

↓ step 1: Eliminating LR.

(I)

① $E \to TE'$
② $E' \to +TE'$
③ $E' \to \varepsilon$
④ $T \to FT'$
⑤ $T' \to *FT'$
⑥ $T' \to \varepsilon$
⑦ $F \to (E)$
⑧ $F \to id$

Step 2: find first and follow

$T \to +, *, (,), id$

Non-terminal → E, E', T, T', F

(II) Terminal → First

First(+) → {+}
First(*) → {*}
First(( ) → {(}
First( )) → {)}
First(id) → {id}

(III) NonTerminal → First

First(E) = { (, id}

$E \to TE'$
$T \to FT'$
$F \to (E)$
$F \to id$

(IV)

First(E') = { +, ε}

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

First(T) = { (, id}

$T \to FT'$
$F \to (E)$
$F \to id$

First(T') = {*, ε}

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

First(F) = { (, id}

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

(V) Follow-NT

Follow(E) = { $,) }

$F \to (E)$ → first( ))

follow(E') = follow(E) = {$,)}

$E \to +E'$, $E' \to +TE'$

follow(T) = first(E') - ε ∨ follow(E)

$E \to TE'$

$E' \to +TE'$

= {+, $,)}

→ Follow(T')

T → F T'
T' → * F T'

follow(LHS) = follow(T) = {+, $,)}

→ follow(F)

T → F T'
T' → * F T' ⇒ first(T') − ε ∪ follow(T) ∪ follow(T')
⇒ {*, +, $,)}

	+	*	(	)	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→(E)		F→id	

Parsing Table

① eq E → T E'
first(T) = {(, id}

② eq E' → + T E'
first(+) = {+}

③ eq E' → ε
follow(E') = {$,)}

④ eq T → F T'
first(F) = {(, id}

⑤ eq T' → ε
follow(T') = {+, $,)}

⑥ eq T' → * F T'
first(*) = {*}

⑦ eq F → (E)
first(( ) = {(}

⑧ eq F → id
first(id) = {id}

I/P: id + id * id

Stack	input	Action
$E	id + id * id $	E, id E → TE'
$E'T	id + id * id $	T, id T → FT'
$E'T'F	id + id * id $	F, id F → id
$E'T'id	id + id * id $	T', + T' → ε
$E'	+ id * id $	E', + E' → +TE'
$E'T+	+ id * id $	T, id T → FT'
$E'T'F	id * id $	F, id F → id
$E'T'id	id * id $	T', * T' → * FT'
$E'T'F*	* id $	F, id F → id
$E'T'id	id $	T', $ T' → ε
$E'	$	E', $ E' → ε
$	$	

24/2/23

① E → E * E
E → E + E
E → id
id + id * id

② S → TL;
T → int | float
L → L, id | id
int id, id;

③ S → (L) | a
L → L, S | S
(a, (a, a))

④ S → S + S
S → S - S
S → (S)
S → a
a - (a - a)

⑤ S → cc
c → cc
c → d
ccdd

2 $S \to aAB \mid bA \mid \varepsilon$

$A \to aAb \mid \varepsilon$

$B \to bB \mid \varepsilon$

NO LR & LF

First (S) = {a, b, ε}

First (A) = {a, ε}

First (B) = {b, ε}

follow (S) = {$}

follow (A) = {first (B) − ε ∪ follow (S)} = {b, $}

follow (B) = follow (S) = {$}

	a	b	$
S	S → aAB	S → bA	S → ε
A	A → aAb	A → ε	A → ε
B		B → bB	B → ε

3 $T \to A \mid L$

$A \to no \mid id$

$L \to (S)$

$S \to T, S \mid T \Rightarrow$ $S \to TS'$, $S' \to ,S \mid \varepsilon$

⇒ first (T) = {no, id, (}

first (A) = {no, id}

first (L) = {(}

first (S) = {no, id, (}

first (S') = {, , ε}

follow (T) = {first (S')} = {, , − ε ∪ follow (S)} = {, ,), $}

follow (A) = {follow (T)} = {, ,), $}

follow (L) = {follow (T)} = {, ,), $}

follow (S) = {), follow (S')} = {)}

follow (S') = {follow (S)} = {)}

T → A
T → L
A → no
A → id
L → (S).
S → TS'
S' → ,S/ε

	no	id	(	)	,	$
T	T → A	T → A	T → L			
A	A → no	A → id				
L			L → (S)			
S	S → TS'	S → TS'	S → TS'			
S'				S' → ε	S' → ,S	

4 S → (L)/a

L → L,S/S

⇓

L → SL'

L' → ,SL'/ε

first(S) = {(, a}

first(L) = {(, a}

first(L') = {, , ε}

follow(S) = {first(L') − ε} = {, , $,)}
∪ follow(L)

follow(L) = {)}

follow(L') = {follow(L)} = {)}

S → (L)
S → a
L → SL'
L' → ,SL'
L' → ε

	a	(	)	,	$
S	S → a	S → (L)			
L	L → SL'	L → SL'			
L'			L' → ε	L' → ,SL'	

↓ follow(L')

(a, a)

5] $S \to a \mid \wedge \mid (T)$

$T \to T, S \mid S$

$T \to ST'$

$T' \to ,ST'$

Stack	Input	Relation	Action
$	(id*id)+id$	$ < (	Push (
$ (	id*id)+id$	(< id	Push id
$ (id	*id)+id$	id > *	Pop id
$ (	*id)+id$	(< *	Push *
$ (*	id)+id$	* < id	Push id
$ (*id	)+id$	id >)	Pop id
$ (*	)+id$	* >)	Pop *
$ (	)+id$	(=)	Push)
$ ()	+id$	] > +	Pop)
$ (	+id$	(< +	Push +
$ (+	id$	+ < id	Push id
$ (+id	$		Pop
$			Pop
			Pop
			Accepted.

Stack	input	Relation	Action
$	id*id+id$	$ < id	Push id
$ id	*id+id$	id > *	Pop id
$	*id+id$	$ < *	Push *
$ *	id+id$	* < id	Push id
$*id	+id$	id > +	Pop id
$ *	+id$	* > +	Pop *
$	+id$	$ < +	Push +
$+	id$	+ < id	Push id
$+id	$	id > $	Pop id
$+	$	+ > $	Pop +

Bottom Up Parsing

- Shift Reduce Parsing
- Operator Precedence Parsing
- LR
  - SLR (Simple)
  - CLR (Canonical)
  - LALR (LookAhead)

The Grammer G is said operator precidence Parser:
If they follow these rules:

1] No Production on the right hand side should be 'ε'.

2] There should not be any production rules posessing any two adjecent nodes non terminals.

OPP:

⇒ E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

⇒
Leading(E) = { + * } ∪ Leading(T) } = {+, *, (, id}
Trailing(E) = {Trailing(T) ∪ +} = {+, *,), id}
Leading(T) = Leading(F) ∪ * = {(, id, *}
Trailing(T) = {Trailing(F) ∪ *} = {*,), id}
Leading(F) = {(, id}
Trailing(F) = {), id}

	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
$	<	<	<		<	

① E → E + T
'+' < leading(T)
row → '+' < ((id *)
Trailing(E) > '+'
row → (+ *) id) > '+'

③ T → T * F
'*' < leading(F)
row → '*' < {(, id}
Trailing(T) > '*'
row → {*,), id} ⊇ {*}

⑤ F → (E)

'(' < leading(E)

row → '(' < (+ * (id)

Trailing(E) > ')'

row → (+ *) id) > ')'

F → (E)

↓

=

---

[2] S → TL;

T → int | float

L → L, id | id

Stack	Input	Action
$	int id,id;$	Shift
$ int	id,id;$	Reduce
$ T	id,id;$	Shift
$ T id	,id;$	Reduce
$ TL	,id;$	Shift
$ TL,	id;$	Shift
$ TL,id	;$	Reduce
$ TL	;$	Shift
$ TL;	$	Reduce
$ S	$	

id*id+id for ↑ for 1st.

(id*id)+id for 1st.

② $S \to L = R$
$S \to R$
$L \to * R$
$L \to id$
$R \to L$

Note:
If '<' operator we will "Push".
If top of the stack is '>' than the 1st symbol Pop (stack)
If = 'Push'.

find leading and trailing

leading(S) = {'=' ∪ leading(R)} ∪ {leading(L)} = {=, *, id}
leading(R) = {leading(L)} = {*, id}
leading(L) = {*, id}

trailing(S) = {Trailing(R) ∪ =} = {=, *, id}
trailing(L) = {Trailing(R) ∪ * ∪ id} = {*, id}
trailing(R) = {Trailing(L)} = {*, id}

	=	*	id	$
=		<	<	>
*	>	<	<	>
id	>		e	>
$	<	<	<	Accept

① = < leading(R)
Row → '=' < {*, id} — I
Trailing(L) > '='
Row → {*, id} > '=' — II
$L \to * R$.

* < leading(R)
Row → '*' < {*, id}

Stack	Input	Relation	Action
$	*id=id$	$ < *	push *
$*	id=id$	* < id	push id
$*id	=id$	id > =	pop id
$*	=id$	* > =	pop *
$	=id$	$ < =	push =
$=	id$	= < id	push id
$=id	$	id > $	pop id
$=	$	= > $	pop =
$	$		Accept.

Direct Method

$E \to E+E$
$E \to E*E$
$E \to id$
id is High
$ is Low

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	e	>
$	<	<	<	Accepted

+ → equal
+ → equal
^ * →
^ ^ ←

⇒ $ < id > * < id > + < id > $

E * < id > + < id > $

E * E + < id > $

E * E + E

Drop 'E'

⇒ $ < * > + > $

$ E + > $ [Can't do it like that]

$ < + > $ [Dropped '< * >']

$ $

→ Accept.

---

③ $S \to (L) \mid a$

$L \to L,S \mid S$

	,	a	(	)	$
,	>	<	<	<	>
a	>	e	>	>	>
(	>	<	<	=	>
)	>	>	>	>	>
$	<	<	<	<	A

* will have higher precedence.

⇒

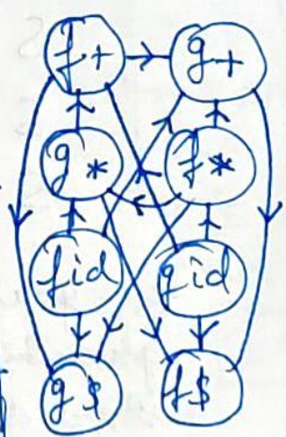
Stack	Input	Relation	Action
$	(a,(a,a))$		

And Direct

Third Method.

E → E+E / E*E / id

f \ g	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>	e	·>
$	<·	<·	<·	Accept



Precedence fn — f
Precedence graph — g
Length of the graph

→ If 'f' is less than 'g' for any terminal then draw an edge from 'g' to 'f'.
→ If 'f' is greater than 'g' draw an edge from 'f' to 'g'.

Symbol table

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

(4M)

Direct Method.

Operator Precedence Parsing.

LR Parsing:

L → Left to Right Scaning
R → Right most derivation.
LR(K)
↓
0, 1 . . . .

SLR CLR LALR
LR(0)

SLR (Parsing) : CFG as Input → canonical set of item.
→ construction of SLR Parsing table.
→ Parsing Input string.
→ Final output.

Q1] What is LR(0) item?

→ A Grammer 'g' some production having a symbol (•) inserted in any position on the right hand side, then that item is called LR(0) item.

eg: S → •ABC
S → A•BC
S → AB•C
S → ABC•

If a grammer g is having a start symbol S then argmented grammer is a new grammer g' in which S' is a new start symbol such that S' → S• is the new rule.

S' → •S ✓

Kernel items → It is the collection of items $S \to \cdot S$ and all the items whose '·' are not at the leftmost end of the right hand side of the rule.

Non Kernel items ⇒ collection of items where '·' are at the leftend of the right hand side of the rule.

Viable prefix ⇒ $K \to \alpha$

set of prefix in the right sentencial form.

SLR Parsing

$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

$\Rightarrow$

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

} I_0 [Item set]

goto(I0,E)
goto(I0,T)
goto(I0,F)
goto(I0,()
goto(I0,id)

⇒ Because there is E, T, F after the dot, we are writing there "rules".

① goto (I_0, E) = $E' \to E \cdot$, $E \to E \cdot + T$ } I_1

$[E' \to \cdot E]$
$[E \to \cdot E + T]$

② goto (I_0, T) = $E \to T \cdot$, $T \to T \cdot * F$ } I_2

$[E \to \cdot T]$
$[T \to \cdot T * F]$

③ goto (I_0, F) = $T \to F \cdot$ } I_3

$[T \to \cdot F]$

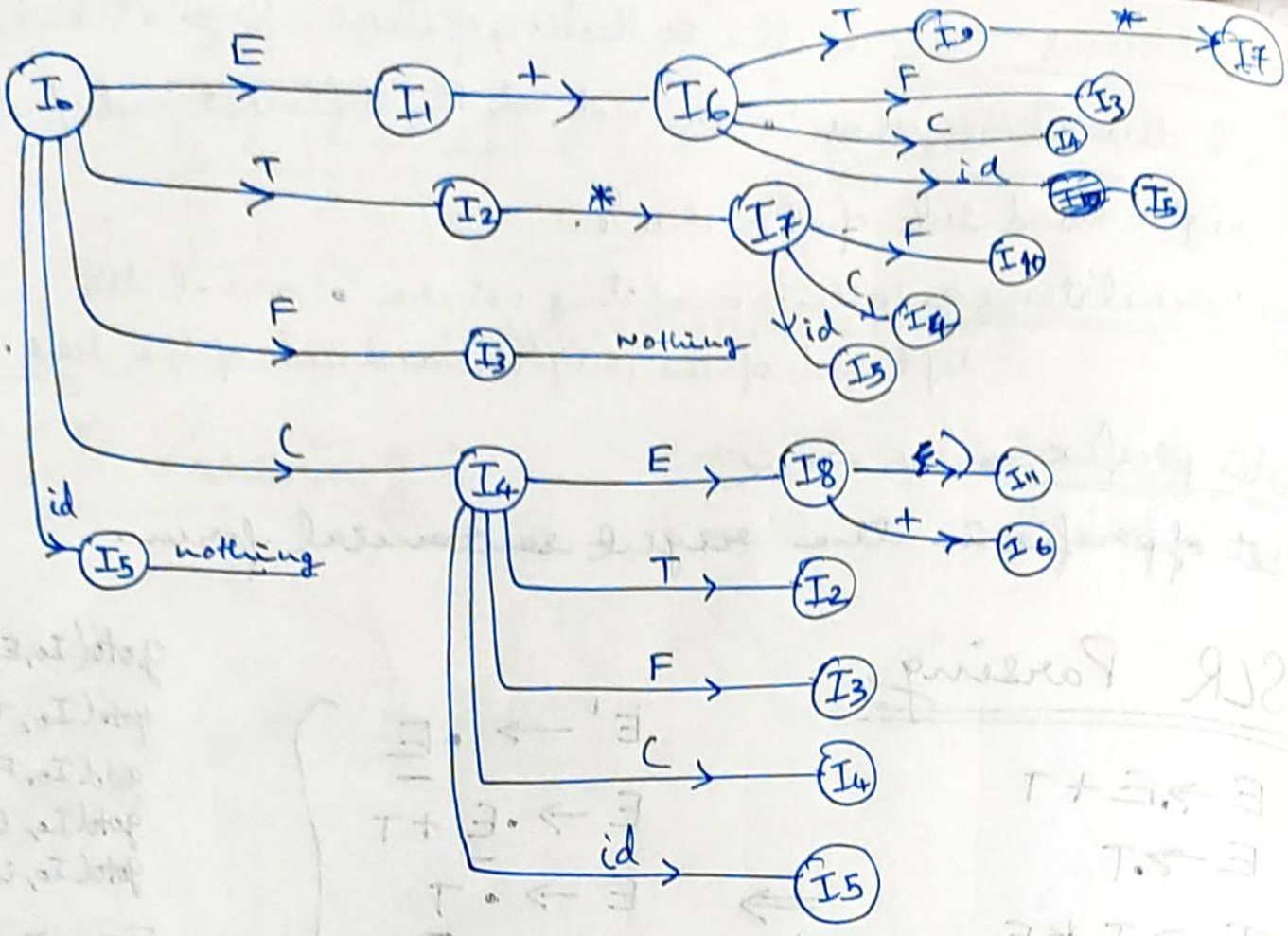
④ goto $(I_0, ()$ = $F \to (\cdot E)$

$[F \to \cdot (E)]$

$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T *$
$T \to \cdot F$
$F \to \cdot (E$
$F \to \cdot id$

⑤ goto (I_0, id) → $F \to id \cdot$ } I_5

$[F \to \cdot id]$



	+	*	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1	S_6					Accept			
2	γ_2	S_7		γ_2		γ_2			
3	γ_4	γ_4		γ_4		γ_4			
4			S_4		S_5		8	2	3
5	γ_6	γ_6		γ_6		γ_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8	S_6			S_{11}					
9	γ_1	S_7		γ_1		γ_1			
10	γ_3	γ_3		γ_3		γ_3			
11	γ_5	γ_5		γ_5		γ_5			

I_1 goto(I_1, +) $E' \to E\cdot$ $E \to E \cdot + T$

$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$
} I_6 (T rules)

I_2 goto(I_2, *) $E \to T \cdot$ $T \to T \cdot * F$ } I_7

$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$
} I_8 (F rules)

I_4 goto(I_4, E) = $F \to (E \cdot)$
$E \to E \cdot + T$
} I_8

I_4 goto(I_4, T) = $E \to T \cdot$
$T \to T \cdot * F$
} I_2

I_4 goto(I_4, F) = $T \to F \cdot$ } I_3

$T \to \cdot F$ } I_4

I_6 goto(I_6, T) = $E \to E + \cdot T$
$T \to \cdot T * F$
} I_6

$E \to E + T \cdot$
$T \to T \cdot * F$
} I_9

I_6 goto(I_6, F) = $T \to \cdot F$
$T \to F \cdot$ } I_3

I_6 goto(I_6, () = I_4

I_6 goto(I_6, id) = $F \to id \cdot$ } I_5

I_7 goto(I_7, F) = $T \to T * F \cdot$ } I_{10}

I_7 goto(I_7, () = $F \to (\cdot E)$
$F \to \cdot (E)$
$F \to \cdot id$
} I_6

I_7 goto(I_7, id) = $F \to id \cdot$ } I_5

I_8 goto(I_8, F) = $E \to (E) \cdot$ } I_{11}

goto(I_8, +) = $E \to E + \cdot T$
$F \to \cdot (E)$
$F \to \cdot id$
} I_6

r_2 I_2 E → T·

r_4 I_3 T → F·

r_6 I_5 F → id·

~~r_2 I8 E → T~~

r_1 I_9 E → E+T·

r_3 I_{10} T → T*F·

r_5 I_{11} F → (E)·

① E → ·E+T

② E → ·T

③ T → ·T*F

④ T → ·F

⑤ F → ·(E)

⑥ F → ·id

Follow(E) = {$, +,)}

follow(T) = {*, $, +,)}

follow(F) = {*, $, +,)}

Stack	input Buffer	ActionT.	GotoT.	Parsing action
$0	id*id+id$	[0, id] = S_5		Shift
$0 id5	*id+id$	[5, *] = r_6	[0, F] = 3	Reduce F→
$0 F3	*id+id$	[3, *] = r_4	[0, T] = 2	Reduce T→F
$0 T2	*id+id$	[2, *] = S7		Shift
$0 T2*7	id+id$	[7, id] = S_5		Shift
$0 T2*7id5	+id$	[5, +] = r_6	[7, F] = 10	Reduce F→id
$0 T2*7F10	+id$	[10, +] = r_3	[0, T] = 2	Reduce T→T*
$0 T 2	+id$	[2, +] = ~~S2~~	[0, E] = 1	~~Shift~~ Reduce E→T
$ ~~0 T 2 T~~ 0 E 1	+id$	[~~T~~1, ~~T~~+] = S_6		Shift
$ 0E1+6	id$	[~~T~~6, id] = S_5		Shift
$ 0E1+6id5	$	[5, $] = r_6	[6, F] = 3	Reduce F→id
$ 0E1+6F3	$	[3, $] = r_4	[6, T] = 9	Reduce T→F
$ 0E1+6T9	$	[9, $] = r_1	[0, E] = 1	Reduce E→E+T
$ 0E1	$	[1, $] = Accepted.		Accepted.

	a	d	$	S	C
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	γ_3	γ_3			
5			γ_1		
6	S_6	S_7			9
7			γ_3		
8	γ_2	γ_2			
9			γ_2		

$C \to a.C, a/d$
$C \to .aC, a/d$
$C \to .d, a/d$
} I_3

$C \to a.C, \$$
$C \to .aC, \$$
$C \to .d, \$$
} I_6

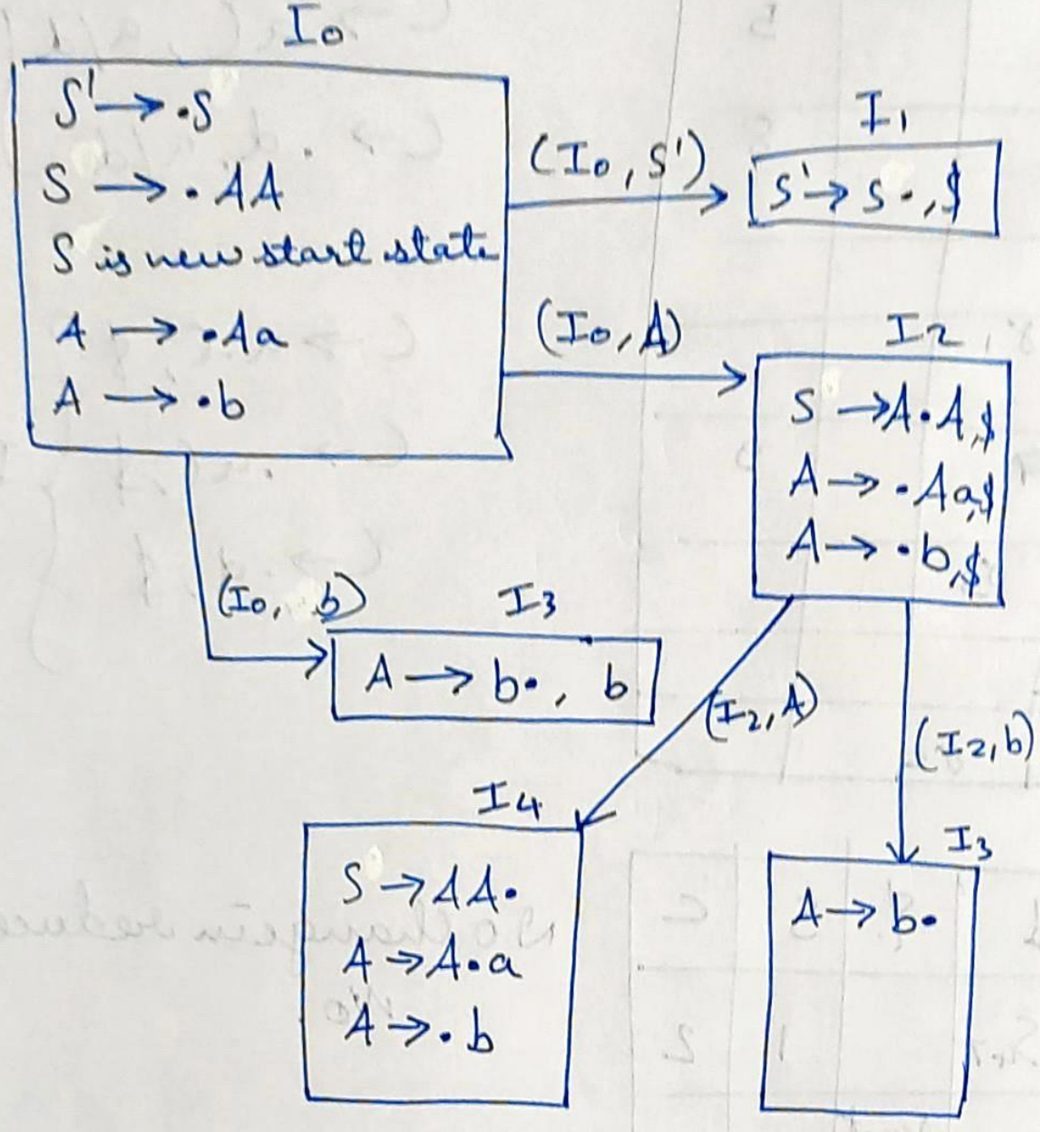
	a	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	γ_3	γ_3	γ_3		
5			γ_1		
89	γ_2	γ_2	γ_2		

No Change in reduce No.

CLR if combining then it is LALR.

Example: $S \rightarrow AA$
$A \rightarrow Aa$
$A \rightarrow b$

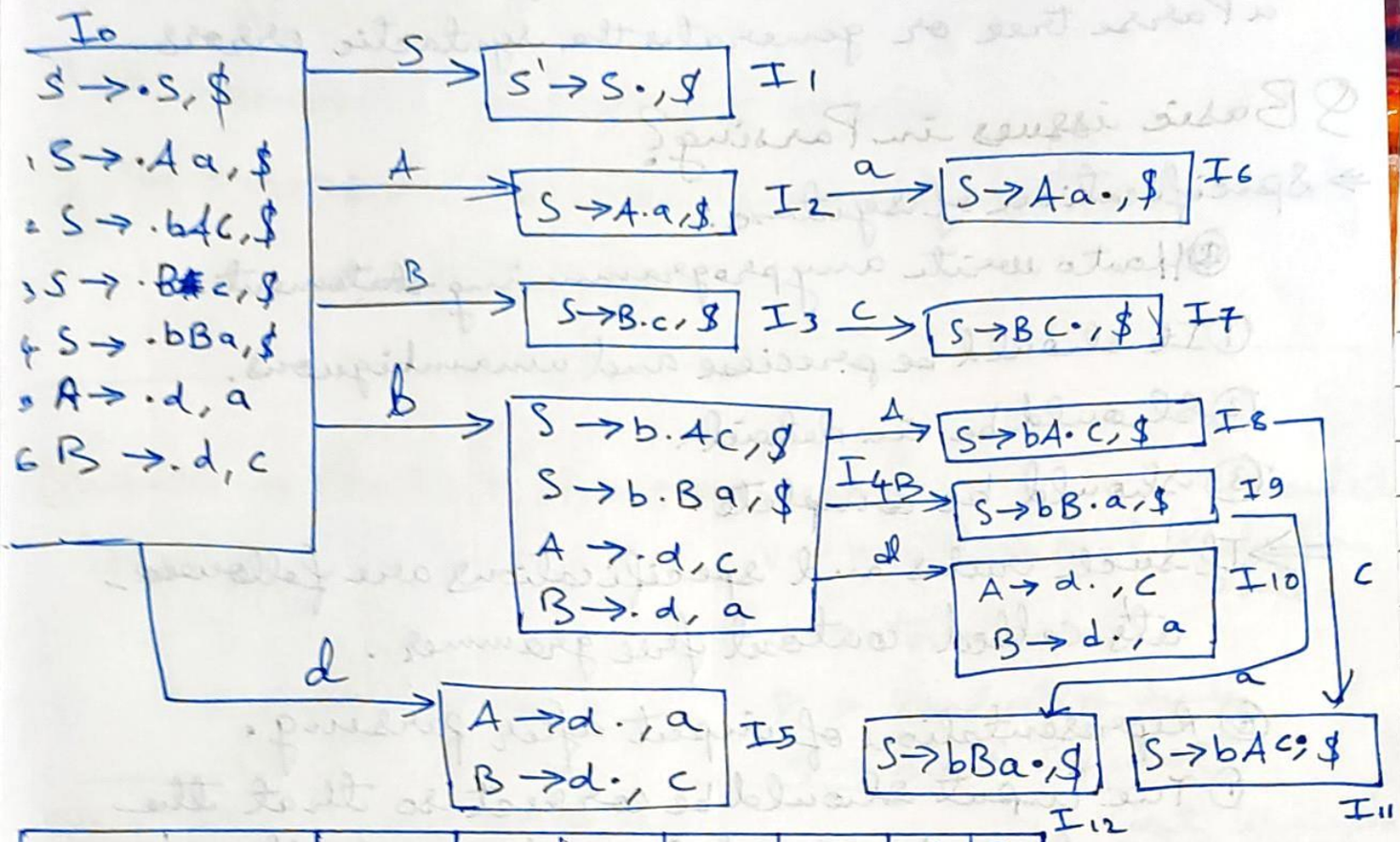
I/P → b a



(3) $S \to Aa / bAc / Bc / bBa$

$A \to d$

$B \to d$ check it is LR(1) not LALR.



	a	b	c	d	$	S	A	B
0		S_4		S_5		1	2	3
1					Accept			
2	S_6							
3			S_7					
4				S_{10}			8	9
5	γ_5		γ_6					
6					γ_1			
7					γ_3			
8		S_{11}						
9	S_{12}							
10	γ_6		γ_5					
11					γ_2			
12					γ_4			

$A \to d., a$

$B \to d., c$

Reduce Reduce conflict that's why CLR and not LALR Parser.

5 10 γ_5/γ_6 γ_6/γ_5

what is a Parsar?
(Parsing)

⇒ A Parsar ~~rathmang~~ or syntax analysis is a Process which takes an input string w and Produces either a Parse tree or generates the syntactic errors.

Q Basic issues in Parsing?

⇒ Specification of syntax:

(a) How to write any programming statement.

① It should be precise and unambiguous.

② Should be in detail.

③ Should be complete.

⇒ If such rules and specifications are followed, it's called context free grammer.

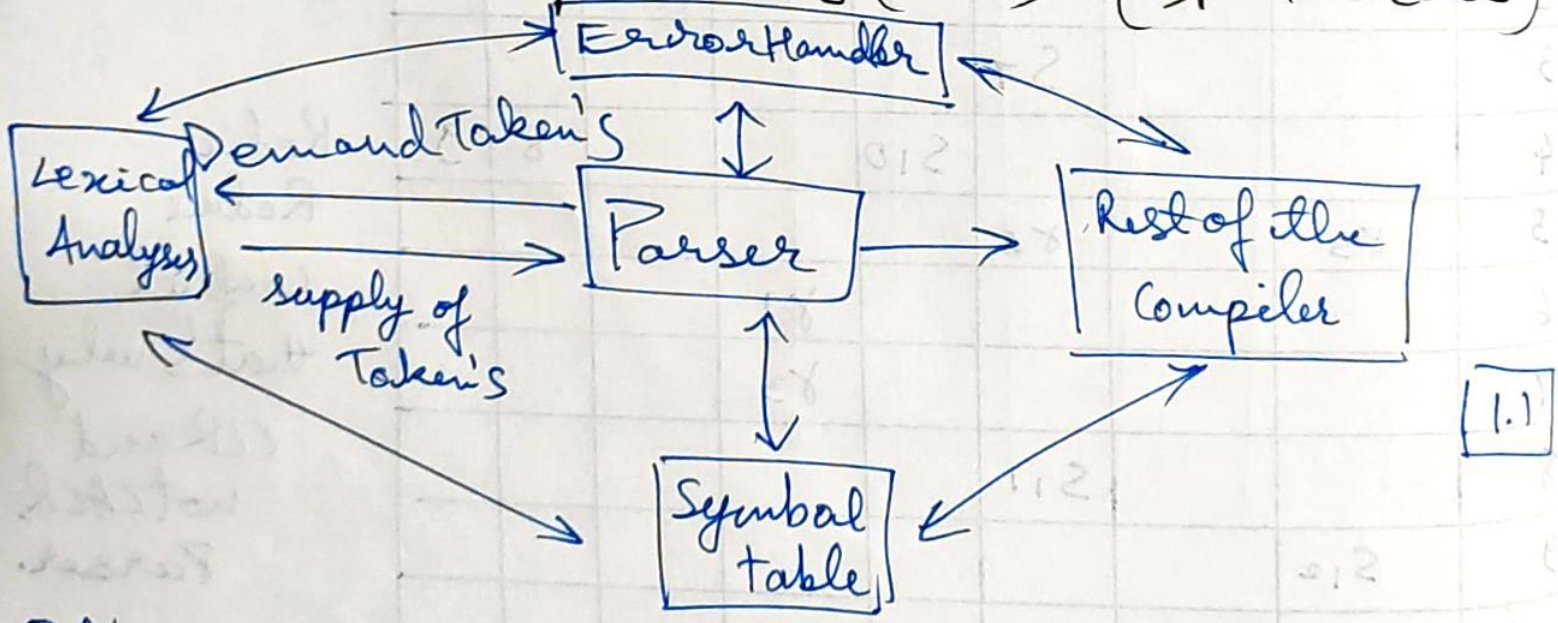
(b) Representation of input after parsing.

① The input should be correct so that the ~~rest~~ it can be used in subsequent phases.

② Parsing Algorithm.

③ How the Parsing algo works.

Q Need and role of Parser (4M) (3pt+draw)



1.1

① The Tokens will be requested from the LA from the Parse Tree.

② The Parser will report the syntactic errors handeled by Error H.

③ The token and updated speci. of the Parsar is stored in the symbol table.

Why Lexical and Syntax Analysis are separated?

1] It accelerated the Process of compiler

2] The errors in the source i/p can be identified precisely.

3] Diagram 1.1

What is Content free grammer?

It is defined as G = VTPS

V → nonterminal / Production rule. (Left hand side)

T → terminals.

P → Production rule

S → start symbol.

$V \rightarrow (VVT)^*$

Ex: $a^n b^n$

⇒ S → aSb

S → ab

Ex: int id, id;

or

float id, id;

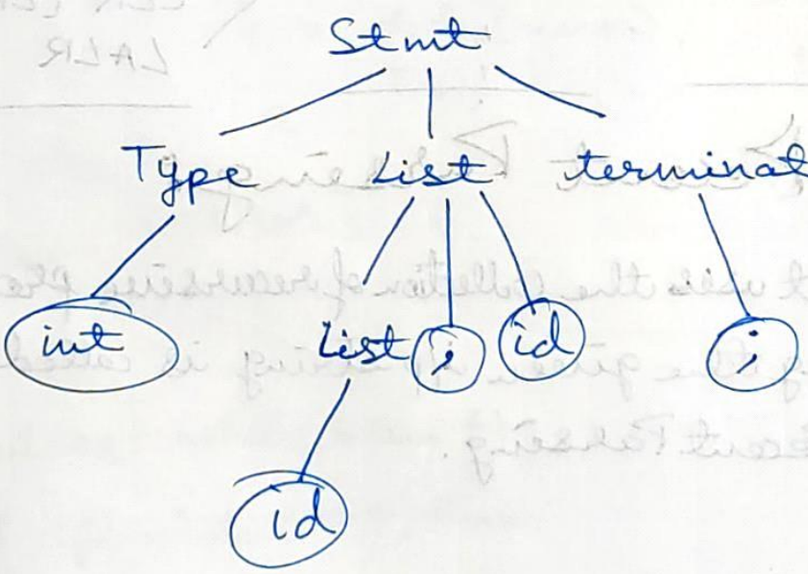
⇒ Stmt → Type List terminator

Type → int / float

List → List, id

List → id

terminator → ;



Q Rules for writing CFG.

→ A single non-t should be on LHS.

⟶ on RHS → (V ∪ T)*

⟹ NT → ε

⟹ one non terminal will be a stack symbol.

Q Comparisor of Top down Parsing and Bottom up Pars

1] Going from root Node to leaf.	1] Going from leaf to root
2] Top down is so seimple	2] Bottom up is complex
3] Less Efficient (elimination of Left Recursion, Left Factoring and Ambiguity is needed)	3] It works in ambiguous grammer.
4] It will work in small class of languages.	4] It will work on Big cl of languages.
5] Backtracking Predictive Parsing ↙ ↘ Recursive Decent LL(1)	5] Shift reduce Parsing operator Precidence LR — SLR (LR(0)) CLR (LR(1)) LALR

Recursive Decent Parsing:

The Parser that uses the collection of recursive procedure for Parsing the given i/p string is called as Recursive Decent Parsing.

Construction:

1] If i/p is Non terminal, Call Procedure.

2] If i/p is terminal match Lookahead with i/p, Advance the Lookahead Pointer on matching the i/p string

3] If the Production Rule has many alternatives, combine all the alternatives to form a single Procedure.

4] The parser has to be activated with the start symbol.

Ex: ① E → num T
T → * num T / ε

CFG

S1] Procedure E
```
{ if (Lookhead = 'num') then
    { match (num);
      T();
    }
  else { error }
  if (Lookahead == '$')
    { declare Success }
  else
    {error}
}
```

S2] Procedure T
```
{ if (Lookahead = '*') then
    { match(*);
      if (Lookahead = 'num') then
        { match(num);
          T();
        }
      else
        {error} }
  else
    {NULL}
}
```

S3] Procedure match(token t)
```
{ if(Lookahead == t) then
    { Lookahead = next_token;
    }
  else {error}
}
```

S4] Procedure error
```
{ print ("Error"); }
```