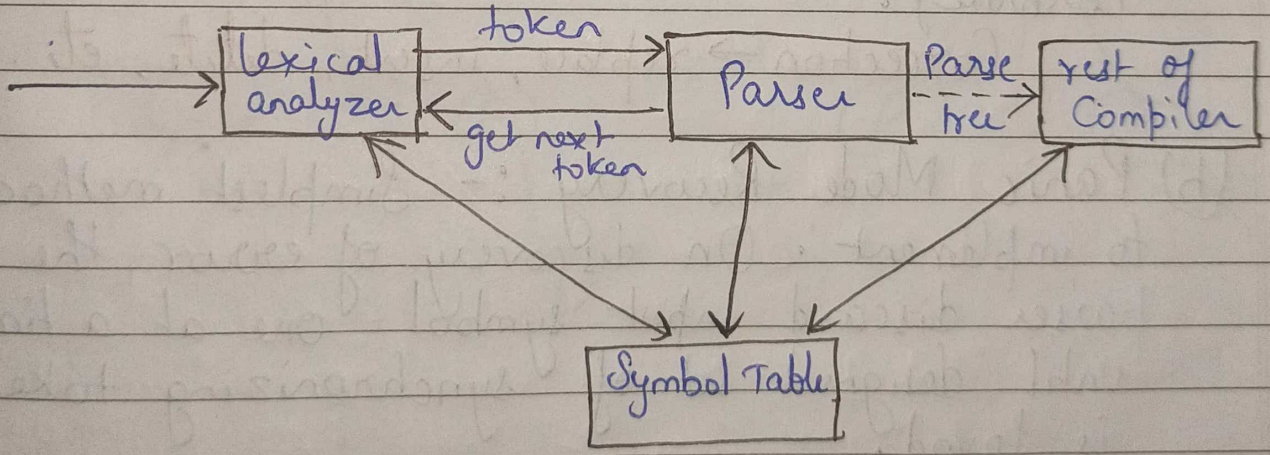
Compiler Design
Unit - 2

Syntax Analysis :- Checking if syntax is correct or not.

The syntax of programming language constructs can be described by context-free grammars or BNF (Backus-Naur form) notation.

* **Role of Parser :-**

The task of parser is to check syntax. The parser obtains a string of tokens from the lexical analyzer and verify that the string can be generated by grammar for the source language.



Syntax Error Handling :-

Types of Errors :-

(i) Lexical → misspelling
(ii) Syntactic → semicolon missing, extra Bracket, Bracket missing
(iii) Logical → infinite loop
(iv) Semantics → operator applied to incompatible operands.

Goals of error handler in a parser :-

→ It should report the presence of error clearly and accurately.
→ It should recover from each error quickly enough to be able to detect subsequent errors.
→ It should not significantly slow down processing of correct program.

* Error Recovery Strategies :-

(a) Phrase level recovery :- On discovery of an error a parser may perform local correction on the remaining input, (i.e it may replace a prefix to some string so that parser will continue).

Correction → replace, insert, delete, etc.

(b) Panic Mode Recovery :- Simplest method to implement. On discovery of error, the parser discard input symbol one at a time until designed set of synchronizing token is found.

(c) Error production → Replace common errors.

(d) Global Correction :- For a given string X with grammar G, we take similar string Y and compare them to find error.

Derivation :-

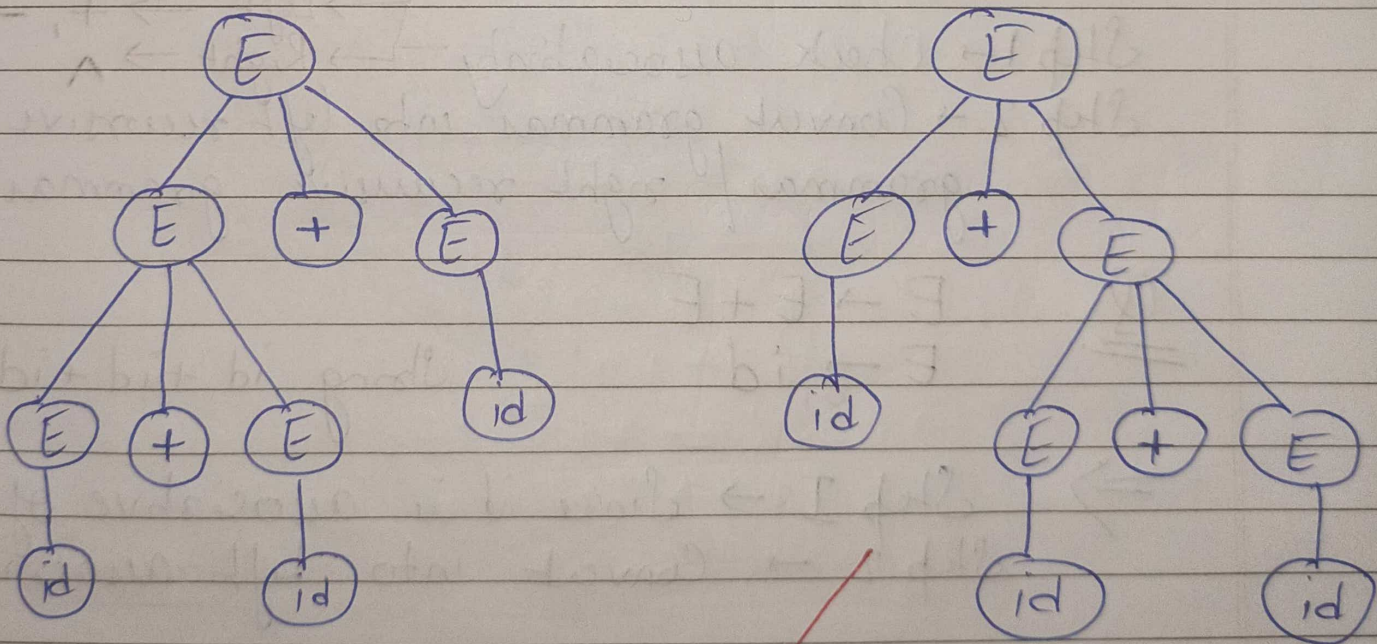
$E \rightarrow E + E$
$E \rightarrow id$

① Left-Most Derivation

$E \rightarrow E + E$
$E \rightarrow (E+E) + E$
$E \rightarrow id + E + E$
$E \rightarrow id + id + E$
$E \rightarrow id + id + id$

② Right-Most Derivation

$E \rightarrow E + E$
$E \rightarrow E + (E+E)$
$E \rightarrow E + E + id$
$E \rightarrow E + id + id$
$E \rightarrow id + id + id$



2-Different Parse tree
⇒ Ambigous

When such situation of ambigous happens and we have to choose which one to take. We follow following rules :-

(a) Associativity → (when symbols are same)
e.g id + id + id

(b) Precedence → (when symbols are different)
e.g id + id * id

So basically, to remove ambiguity we check for whether ~~or no~~ it is associative or Predence and then convert accordingly.

Step 1 → Check associativity → Left → +, −, *, /
→ Right → ^

Step 2 → Convert grammar into left recursive grammar/ right recursive grammar.

Q. $E \rightarrow E + E$
$E \rightarrow id$

String id + id + id

⇒ Step I → Since it is associative of '+'
Step 2 → Convert into left associative

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

⇒

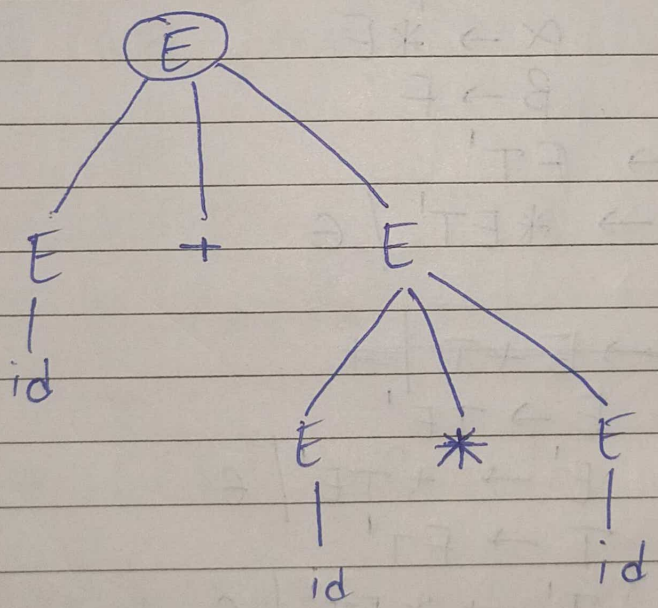
$E \rightarrow E + id$
$E \rightarrow id$

Q $E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow id$ String id + id * id

⇒ Since string contain different symbol so precedence will follow.

* Precedence of operators :-
1) Higher level will have operator of less priority.
2) Lower level will have operator of high priority.

⇒



Elimination of Left Recursion :-

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

$\Downarrow$

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

Q $E \to E+T \mid T$
$T \to T*E \mid F$
$F \to (E) \mid id$

$\Rightarrow$ $E \to E+T \mid T$
$\alpha \to +T$
$\beta \to T$
$E \to TE'$
$E' \to +TE' \mid \epsilon$

$T \to T*E \mid F$
$\alpha \to *E$
$\beta \to F$
$T \to FT'$
$T' \to *ET' \mid \epsilon$

$\Rightarrow$ ~~$E \to E+T \mid +$~~
$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *ET' \mid \epsilon$
$F \to (E) \mid id$

Q $S \to Aa \mid b$
$A \to Ac \mid Sd \mid \epsilon$

$\Rightarrow$ $S \to Aa \mid b$
$A \to Ac \mid Aad \mid bd \mid \epsilon$

$A \to Ac$ | $A \to Aad$

$\alpha \to c$ | $\alpha \to ad$

$\beta \to bd \mid \epsilon$ | $\beta \to bd \mid \epsilon$

$A \to bdA' \mid \epsilon A' \Rightarrow A \to bdA' \mid A'$

$A' \to adA' \mid cA' \mid \epsilon$

Left factoring :- (Common Prefix Problem) :-

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \gamma$

$\Downarrow$

$A \to \alpha A' \mid \gamma$

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

Q. $S \to iEtS \mid iEtSeS \mid \alpha$

$\Rightarrow$ $\alpha \to iEtS$

$\beta_1 \to \epsilon$

$\beta_2 \to eS$

$\gamma \to \alpha$

$\Rightarrow$ $S \to iEtSS' \mid \alpha$

$S' \to e \mid eS$

Q. $S \rightarrow aSSbSS / aSaSb / abb / b$

$\Rightarrow$ $\alpha \rightarrow a$
$\beta_1 \rightarrow SSbSS$
$\beta_2 \rightarrow SaSb$
$\beta_3 \rightarrow bb$
$\gamma \rightarrow b$

$S \rightarrow aS' / b$ — (1)
$S' \rightarrow SSbSS / SaSb / bb$
$\alpha \rightarrow S$
$\beta_1 \rightarrow SbSS$
$\beta_2 \rightarrow aSb$
$\gamma \rightarrow bb$
$S' \rightarrow SS'' / bb$ — (2)
$S'' \rightarrow SbSS / aSb$ — (3)

For Practice :-

Q. $S \rightarrow bSSaaS / bSSaSb / bSb / a$

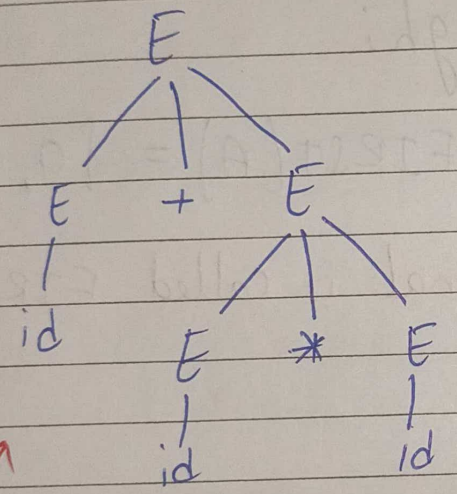
Parsing Methods

(1) Top Down Parsing	(2) Bottom-up Parsing
→ Start from start symbol	→ Just opposite

Q $E \rightarrow E+E \mid E*E \mid id$

Generate id + id * id using both method

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



Top-Down

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

Step 1 → E + id * id
id

Step 2 → E + E * id
id id

Step 3 → E * id
E + E
id id

Step 4 → E * E
E + E id
id id

Step 5 → E
E * E
E + E id
id id

Bottom-up

Top-Down Parser

LL(1) Parser

- L → Left to right
- L → Left most derivation
- 1 → lookahead symbol

* Computation of FIRST :-

Q. $A \rightarrow abc / def / ghi$

$\Rightarrow$ First of A $\Rightarrow$ FIRST(A) = {a, d, g}

* Basically first terminal is called FIRST.

Q
$A \rightarrow BCD$
$B \rightarrow C$
$C \rightarrow d$
$D \rightarrow e / \epsilon$

$\Rightarrow$ FIRST(A) = FIRST(B) = {c}
FIRST(B) = {c}
FIRST(C) = {d}
FIRST(D) = {e, ϵ}

Q
$S \rightarrow ABCD$
$A \rightarrow bDC$
$B \rightarrow CD$
$C \rightarrow d$
$D \rightarrow e / ABC / \epsilon$

⇒ FIRST(S) → {b}
FIRST(A) → {b}
FIRST(B) → {d}
FIRST(C) → {d}
FIRST(D) → {e, ϵ, b}

* Computation of FOLLOW :-

Q. A → AbC → Just after A what is there (terminal).

FOLLOW(A) = {b}

Q. A → ABCc ⇒ FOLLOW(A) = {d}
B → d FOLLOW(B) = {c}

Q. S → ABCD ⇒ FOLLOW(A) = {c}
A → b FOLLOW(B) = {d}
C → d/e FOLLOW(D) = {$}
D → e FOLLOW(C) = {d e}
B → c FOLLOW(S) = {$}

* Note → Start symbol always have a $

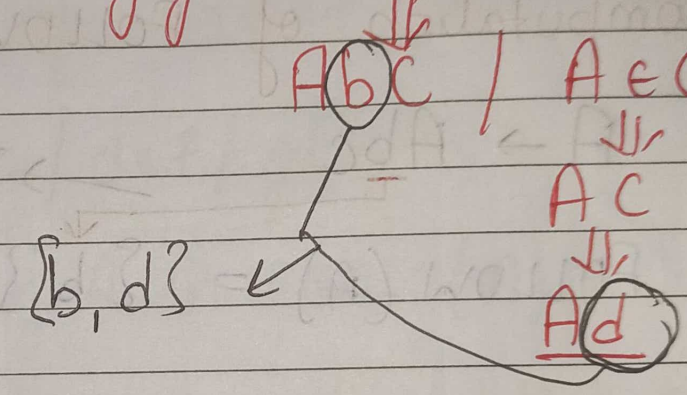
S → ABCD

D end me h ⇒ S ka follow copy hoga.

Q. $S \to ABC$
$A \to a$
$B \to b / \epsilon$
$C \to d$

$\Rightarrow$ { $ }
{ b, ϵ } = { b, d }
{ d }
{ $ }

Now, follow me ϵ nhi hoga
fo, ϵ ka value put karne ke bad
kya ayega uo check karenge.



Practise :-

Q. $S \to ABCD$
$A \to b / \epsilon$
$C \to d / \epsilon$
$D \to e$
$B \to c / \epsilon$

$\Rightarrow$ { $ }
{ c, d, e }
{ e }
{ $ }
{ d, e }

Q. $S \to Bb / Cd$
$B \to aB / \epsilon$
$C \to cC / \epsilon$

LL(1) Parsing Table :-

Q. $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

⇒ First of all we make sure there is no left recursion & left factoring in production rule

We already done this question in left recursion

⇒ $E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *ET' \mid \epsilon$
$F \to (E) \mid id$

Now, Calculate first and follow.

		FIRST	FOLLOW
$E \to TE'$	⇒	{ (, id }	{ $,) }
$E' \to +TE' \mid \epsilon$	⇒	{ +, ϵ }	{ $,) }
$T \to FT'$	⇒	{ (, id }	{ $,), + }
$T' \to *ET' \mid \epsilon$	⇒	{ *, ϵ }	{ $,), + }
$F \to (E) \mid id$	⇒	{ (, id }	{ +, *,), $ }.

Now Creat Parsing Table

	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to *ET'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to id$	

Q. $S \to AbAa \mid BaBb$
$A \to \epsilon$
$B \to \epsilon$

$\Rightarrow$ $S \to AbAa \mid BaBb$
$A \to \epsilon$
$B \to \epsilon$

	FIRST	FOLLOW
S	{b, a}	{$}
A	{ε}	{b, a}
B	{ε}	{a, b}

	a	b	$
S	$S \to AbAa$	$S \to BaBb$	
A	$A \to \epsilon$	$A \to \epsilon$	
B	$B \to \epsilon$	$B \to \epsilon$	

Note :- If there is 2 production in any block of this Parse tree then LL(1) is not ~~at~~ obtained. (We can't Generate True).

So, Question can be asked to check whether it is a LL(1) or not.

Q. $S \to A \mid a$
$A \to a$

$\Rightarrow$ $S \to A \mid a$ | {a} | $
$A \to a$ | {a} | $

	a	$
S	$S \to A / S \to a$	
A	$A \to a$	

→ 2 Production in one block.
⇒ Not LL(1).

For Practise :-

Q. Check if following grammar is LL(1) or not.

1) $S \to aSbS / bSaS / \epsilon$

2) $S \to aABb$
$A \to c / \epsilon$
$B \to d / \epsilon$

3) $S \to aB / \epsilon$
$B \to bC / \epsilon$
$C \to cS / \epsilon$

4) $S \to aAa / \epsilon$
$A \to abS / \epsilon$.

* Making LL(1) Parsing tree from LL(1) Parsing table.

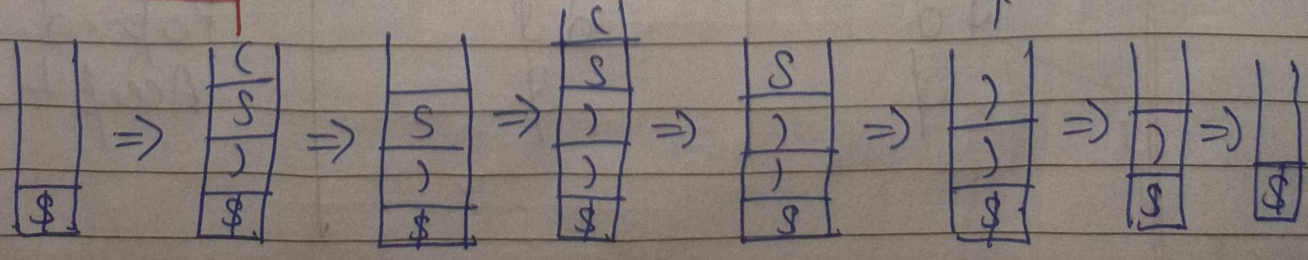
Q. $S \to (S) / \epsilon$ Generate (( ))

Sol. $\Rightarrow$ $S \to (S) / \epsilon$ $\Rightarrow$ $\{(, \epsilon\}$ $\{\$,)\}$

	(	)	$
S	$S \to (S)$	$S \to \epsilon$	$S \to \epsilon$

Same Symbol pe Pop operation

(( ))



How to do in exam?

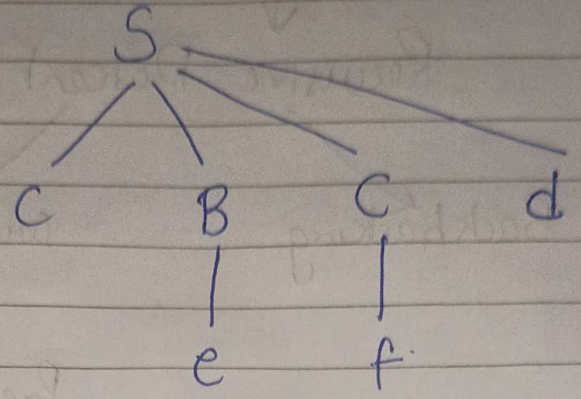
Q. $S \to cBCd$
$B \to e/\epsilon$
$C \to f/\epsilon$ Generate LL(1) for cefd.

Sol → $S \to cBCd$ | {c} | {$}
$B \to e/\epsilon$ | {e, ε} | {f, d}
$C \to f/\epsilon$ | {f, ε} | {d}

	c	e	f	d	$
S	S→cBCd				
B		B→e	B→ε	B→ε	
C			C→f	C→ε	

Stack	Input	Action
$	cefd $	
$S	cefd $	S→cBCd
$dCBc	cefd $	Pop c
$dCB	efd $	B→e
$dCe	efd $	Pop e
$dC	fd $	C→f
$df	fd $	Pop f
$d	d $	Pop d
$	$	Accept

Now draw tree :-



Question for Practise :-

Q. $E \rightarrow TE'$

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

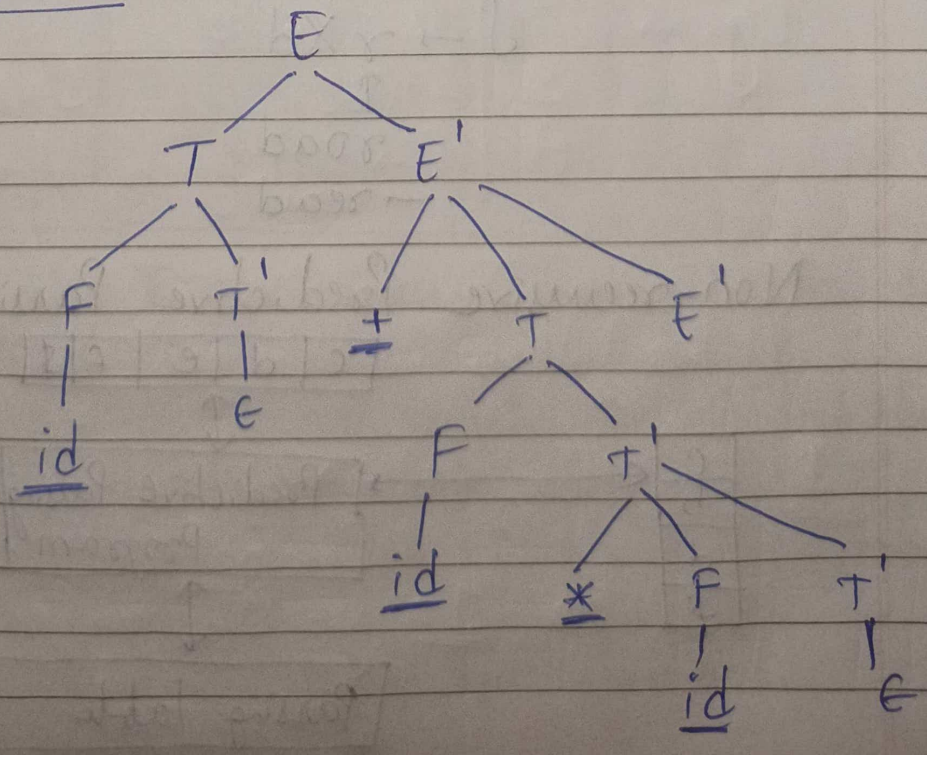
$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

Generate LL(1) tree for id + id * id

Ans to Match :-



Top-Down Parser

- Recursive Descent Parsing
  - Backtracking
  - Without Backtracking
    - Predictive Parser
      - LL(1) Parser
        - If generate error then don't backtrack.

Backtracking :-

$S \rightarrow \gamma X d / \gamma Z d$

$X \rightarrow Oa / ea$

$Z \rightarrow ai d$

Generate string read

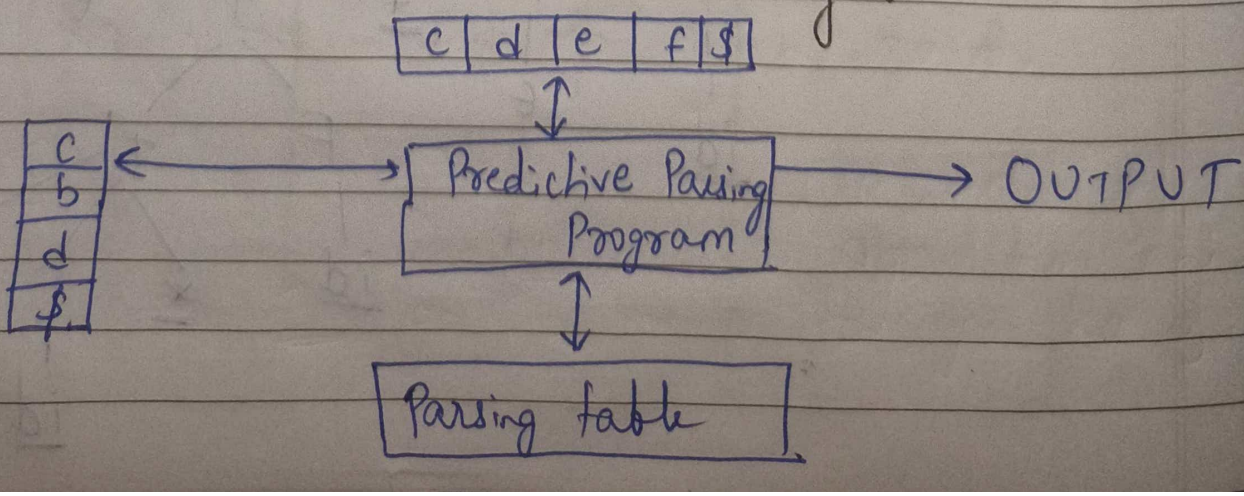
$S \rightarrow \gamma X d$

↑

γoad

→ read

Non-recursive Predictive Parsing :-



Overview :-

TOP DOWN PARSER

↓ Remove left Recursion & Left factoring

Predictive Parser

↓ Remove Backtracking.

LL(1) Parser

Q. $S \to AbS/e/\epsilon$ ⇒ {e, ϵ, a, c} | {$}
$A \to a/cAd$ {a, c} | {b, d}

String aab$.

	a	b	c	d	e	$
S	S→AbS		S→AbS		A→e	A→ϵ
A	A→a		A→cAd			

Input string	Stack	Action
aab$	$S	S→AbS
aab$	$SbA	A→a
aab$	$Sba	Pop a
ab$	$Sb	Error

↳ Now, If Error occur. Then how to deal.

1) Panic Mode :-
   a. If top of stack does not match then pop top element
   b. If Synch. then pop top element
   c. If A→a is empty then skip input Symbol.
2) Phrase Mode :-
   insert/ replace or delete I/P Symbol.

Let understand how to see it :-

1.a.

Ari error aaya tha usi ko confirm karte h.

Stack	Input String → Yha aaya tha error. Now, solve	Action
$Sb	ab$	Remove top of Stack
$S	ab$	S → AbS
$SbA	ab$	A → a
$Sba	ab$	Pop a
$Sb	b$	Pop b
$S	$	S → ϵ
$	$	Accept String

1.b. Hmlog karte ye h ki J is v ~~follow~~ first ka symbol me ϵ nhi h. Usko me we put Synch for follow.

Eg. →

	a	b	c	d	e	$
S	S→AbS		S→AbS		S→ϵ	S→ϵ
A	A→a	Synch	A→c	Synch		

Ye J to last ari solve kar rhe the usko A ka follow wale me synch. dalo.

1.c. For A → e , we just skip input Symbol. (i.e agar koi nhi ho 1a ya 1b me se to ye karo).

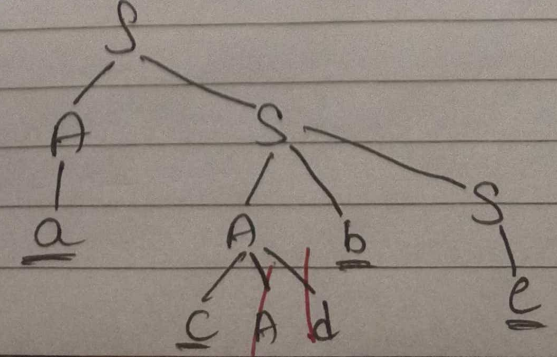
Question → $S \to AbS \mid e \mid \epsilon$
$A \to a \mid cAd$

Generate parse tree for input acbe.

$\Rightarrow$ $S \to AbS \mid e \mid \epsilon$ $\Rightarrow$ {a, c, e, ε} | {$}
$A \to a \mid cAd$ {a, c} | {b, d}.

	a	b	c	d	e	$
S	S→AbS		S→AbS		S→e	S→ε
A	A→a	Synch.	A→cAd	Synch.		

Stack	Input String	Action
$S	acbe$	S → AbS
$SbA	acbe$	A → a
$Sba	acbe$	Pop a
$Sb	cbe$	Pop b (Remove top of stack)
$S	cbe$	S → AbS
$SbA	cbe$	A → cAd
$SbdAc	cbe$	Pop c
$SbdA	be$	Pop A
$Sbd	be$	Pop d
$Sb	be$	Pop b
$S	e$	~~Pop~~ $ S → e
$e	e$	Pop e
$	$	Accept



For Practise :-

Q. $E \rightarrow \bullet TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

Generate Parse tree for)id*+id