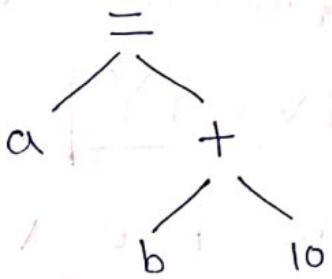
UNIT II : Syntax Analysis (Part-I)

* **Parser** : A Parsing or syntax analysis is a process which takes the input string 'w' & Produces either a Parse tree (syntactic structure) or generates the syntactic errors.

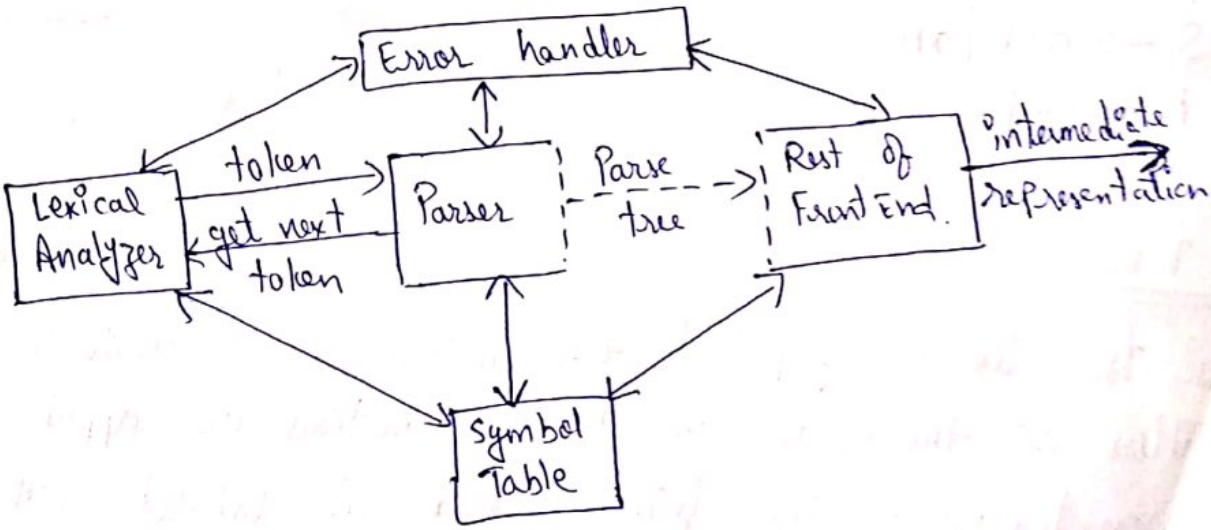
→ The syntax analyzer takes the tokens as input & generates a tree like structure called '**Parse tree**'.

Eg : $a = b + 10$



Parse tree for $a = b + 10$.

(*)



Position of Parser in compiler model

→ Specification of input can be done by using "Context-Free Grammar" (CFG).

Scanned by CamScanner

* Context-Free Grammar:

The CFG, 'G' is a collection of the following:

$$G_1 = (V, T, S, P)$$

- 'V' is a set of non-terminals (All upper-case letters) (Syntactic variables that denote sets of strings)
- 'T' " " " " terminals (basic symbols from which strings are formed)
- 'S' is a start symbol
- 'P' is a set of Production rules.

→ The Production rules are must be in the format:

$$\boxed{\text{Non-terminal} \longrightarrow (V \cup T)^*}$$

(or) $\boxed{A \longrightarrow \alpha}$ where $A \in V$

$\alpha \in (V \cup T \cup \epsilon)^*$.

eg: $S \rightarrow aSb \mid aA$

$A \rightarrow a \mid \epsilon$.

* Parse trees:

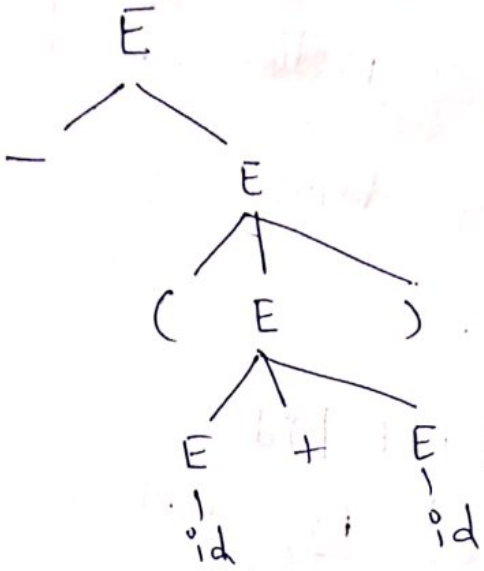
A Parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non-terminals. The interior node is labeled with the non-terminal.

→ Each interior node of a Parse tree represents the application of a Production.

Scanned by CamScanner

Eg: $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$.

generate Parse tree for $-(id+id)$

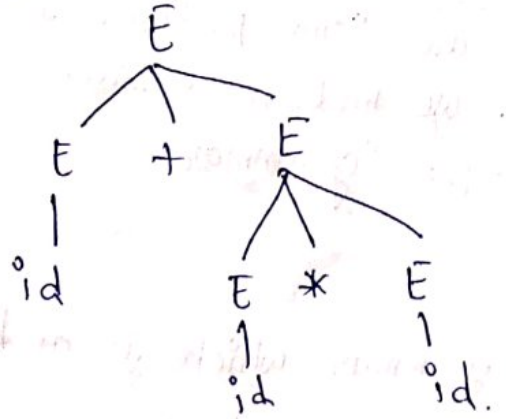


Parse tree

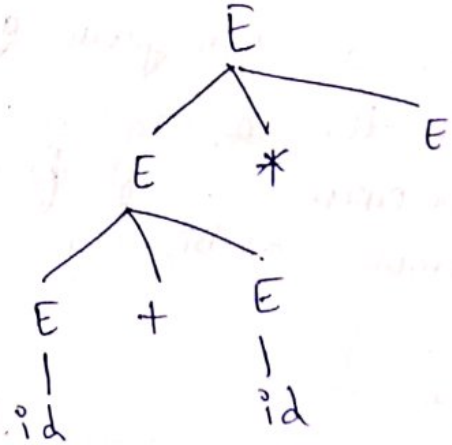
* **Ambiguous Grammar:** A grammar is said to be ambiguous if it generates more than one parse trees for same sentence of language $L(G)$.

Eg: $E \rightarrow E+E \mid E*E \mid id$.

id+id*id:



Parse tree 1



Parse tree 2

Since we have more than one Parse tree for id+id*id, the given grammar is Ambiguous Grammar.

Scanned by CamScanner

* **Derivation** : Derivation from 'S' means generation of string 'w' from 'S'. For constructing derivation two things are important.
  - choice of Non-terminal from several others.
  - choice of rule from Production rules for corresponding non-terminal.

→ we have two types of derivations:
  a) Left most derivation
  b) Right most derivation.

* **Example** : $E \rightarrow E+E \mid E*E \mid id$

Leftmost derivation

$E \rightarrow E+E$
$E \rightarrow id+E$
$E \rightarrow id+E*E$
$E \rightarrow id+id*E$
$E \rightarrow id+id*id$.

Right most derivation

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

* **Eliminating ambiguity** :

Sometimes an Ambiguous grammar can be rewritten to eliminate the ambiguity. There are some problems like left recursion, left factoring. we need to eliminate these to remove ambiguity in the grammar.

(1) **Left Recursion** :

The left recursive grammar is a grammar which is as below :

$A \overset{+}{\Longrightarrow} A\alpha$ where A is a non-terminal
α denotes some i/p string.

(A ... same) $\overset{+}{\Longrightarrow}$ means deriving the i/p in one or more steps.

Scanned by CamScanner

→ Because of left recursion, the top down parser can enter into infinite loop. So we need to eliminate it.

* Removing left recursion

> $A \longrightarrow A\alpha \mid \beta$ is rewritten as :
>
> $A \longrightarrow \beta A'$
> $A' \longrightarrow \alpha A' \mid \epsilon$

Example :

$E \longrightarrow E+T \mid T$
$T \longrightarrow T*F \mid F$
$F \longrightarrow id \mid (E)$

we have left recursion in the following Production rules.

$E \longrightarrow E+T \mid T$ (same A; α = $+T$; β = T)

$T \longrightarrow T*F \mid F$ (same A; α = $*F$; β = F)

removing left recursion we get :

$E \longrightarrow TE'$
$E' \longrightarrow +TE' \mid \epsilon$
$T \longrightarrow FT'$
$T' \longrightarrow *FT' \mid \epsilon$
$F \longrightarrow id \mid (E)$

$\frac{E}{A} \longrightarrow \frac{E}{A}\frac{+T}{\alpha} \mid \frac{T}{\beta}$

$\frac{T}{A} \longrightarrow \frac{T}{A}\frac{*F}{\alpha} \mid \frac{F}{\beta}$

Scanned by CamScanner

② Left factoring :

Consider a grammar :

$$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$$

Here it is not possible for us to take a decision whether to chose first rule or second. The above grammar can be left factored as : rewritten as :

$$A \longrightarrow \alpha A'$$
$$A' \longrightarrow \beta_1 \mid \beta_2$$

* Example : Grammar : $S \longrightarrow iEtS \mid iEtSeS \mid a$
$E \longrightarrow b$

The above grammar after left factoring becomes as :

$S \longrightarrow iEtSS' \mid a$
$S' \longrightarrow eS \mid \epsilon$
$E \longrightarrow b.$

$S \longrightarrow \underset{\alpha}{iEtS} \mid \underset{\alpha}{iEtSeS} \mid \underset{\beta_2}{a}$

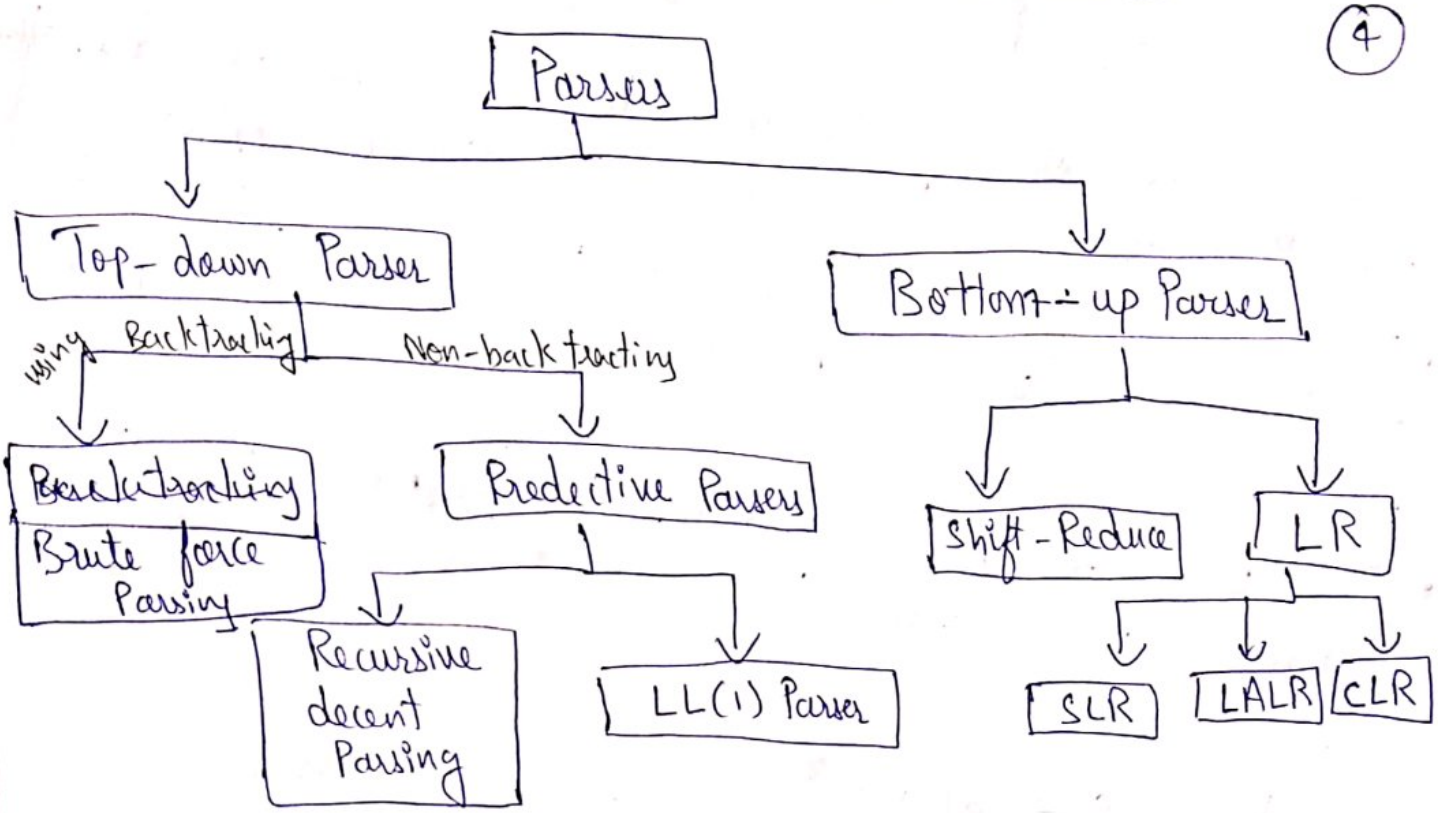
→ By left factoring we may be able to rewrite the Production in which the decision can be deferred until enough of the input is seen to make the right choice.

* Parsing Techniques :

There are two Parsing techniques :

1. Top - Down Parsing (we will Parse from Root to leaf node)
2. Bottom - Up Parsing. (" " " leaf to root node)

Scanned by CamScanner



Types of Parsing Techniques

* (1) Top Down Parsing :-

The top-down construction of a Parse tree is done by starting with the root, labeled with the starting non-terminal, & repeatedly Performing the following two steps

- At node n, labeled with non-terminal A, select one of the Productions for 'A' & construct children at 'n' for the Symbols on the right side of the Production.
- Find the next node at which a sub-tree is to constructed.

→ Top-down Parsing can be viewed as an attempt to find a left-most derivation for an input string. A general form of top-down Parsing is called "Recursive descent Parsing".

Scanned by CamScanner

(a) Brute-force Parsing :

- Given a particular non-terminal that is to be expanded, the first Production for this non-terminal can be applied.
- Then, within this newly expanded string, the left most non-terminal is selected for expansion & its first Production is applied.
- This Process of applying Productions is repeated for all subsequent non-terminals that are selected untill the Process cannot be continued.

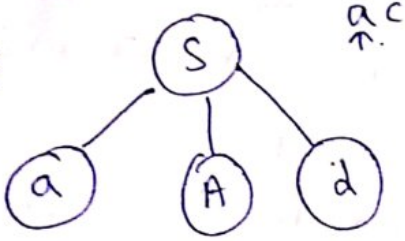
* Example : S → aAd | aB , w = accd.

A → b | c

B → ccd | ddc

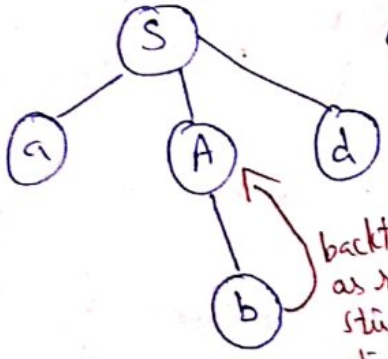
step 1 : S — accd.

step 2 :



accd

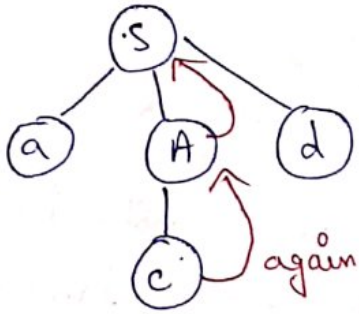
step 3 :



accd

Generated : ab.

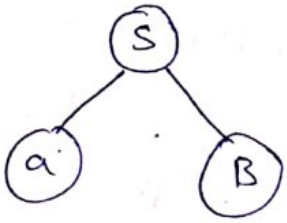
step 4 :



acd we got

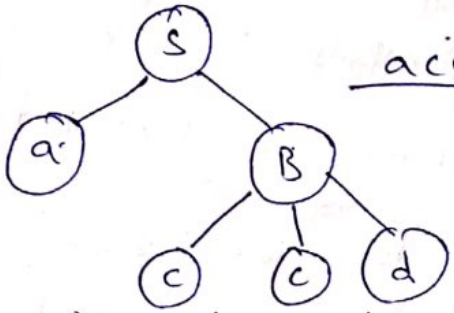
accd

step 5 :



accd

step 6 :



accd

here the given string match with generated string is done so it is ~~success~~ successful Passing.

Scanned by CamScanner

(b) Recursive Descent Parsing:

A Recursive-descent parsing program consists of a set of Procedures, one for each non-terminal. Execution begins with the Procedure for the start symbol, which halts & announces success if its Procedure body scans the entire input string.

→ CFG is used to build the recursive routines. The RHS of the Production rule is directly converted to a Program. For each non-terminal a separate Procedure is written & body of the Procedure (code) is RHS of the corresponding non-terminal.

* Basic steps for construction of Recursive descent Parser:

The RHS of the rule is directly converted into program code symbol by symbol

(1) If the input symbol is non-terminal then a call to the Procedure corresponding to the non-terminal is made.

(2) If the input symbol is terminal then it is matched with the lookahead input. The lookahead pointer has to be advanced on matching of the input symbol.

(3) If the Production rule has many alternates then all these alternates has to combined into a single body of Procedure

(4) The Parser should be activated by a Procedure corresponding to the start symbol.

Scanned by CamScanner

```
Void A() {
    Choose an A-production, A → X1 X2 .... Xk ;
    for (i = 1 to k) {
        if (Xi is a nonterminal)
            call Procedure Xi() ;
        else if (Xi equals the current input symbol a)
            advance the input to the next symbol ;
        else /* an error has occured */ ;
    }
}
```

A typical procedure for a non-terminal in a top-down Parser

(c) Predictive LL(1) Parsing :

Predictive LL(1) Parsing is of non-recursive type. In this type of Parsing a table is built.

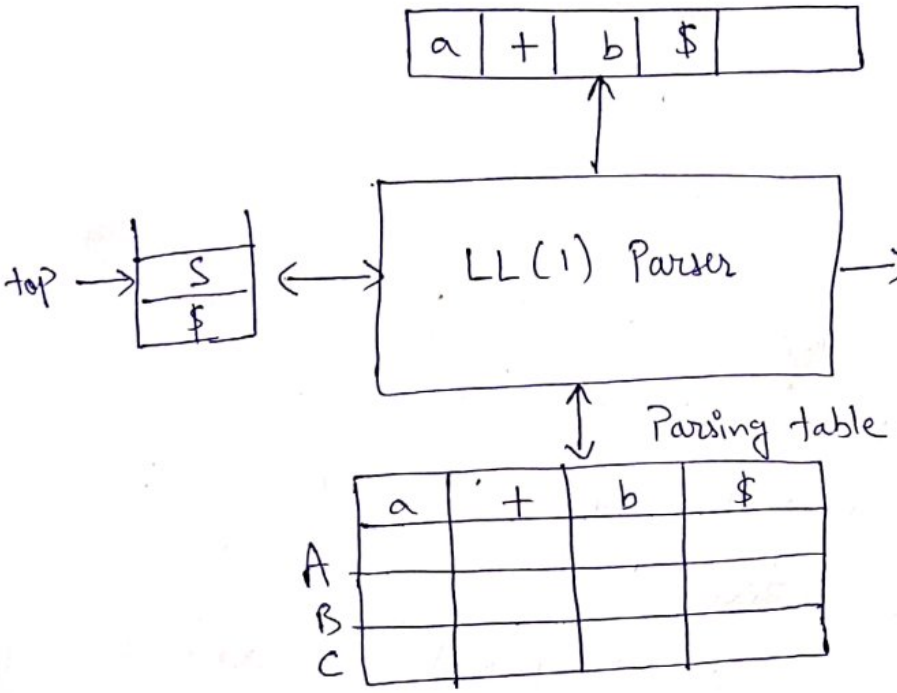
LL (1) → lookahead (uses only one i/p symbol to Predict Parsing Process)

- i/p is scanned from left to right.
- leftmost derivation

→ The data structures used by LL(1) are a) input buffer, b) stack, c) Parsing table. The LL(1) parser uses input buffer to store the input tokens. The stack is used to hold the left sentential form. The symbols in RHS of rule are pushed into the stack in reverse order i.e from right to left.

Scanned by CamScanner

Use of stack makes this parsing non-recursive. The table is basically a two dimensional array. The table can be represented as M[A, a] where 'A' is a non-terminal & 'a' is current input symbol..



Model for LL(1) Parser

→ The Parsing Program reads top of the stack & a current input symbol. with the help of these two symbols the Parsing action is determined. The Parser consults the table M[A, a] each time while taking the parsing actions. Hence this type of Parsing method is called "table drive Parsing".

* Construction of Predictive LL(1) Parsing:

~~Step 1:~~ ~~Comp~~

→ In LL(1) Parsing, Pre-Processing steps are as follows:

Scanned by CamScanner

1. Elimination of left Recursion.
2. left-factoring the grammar.
3. Computation of FIRST & FOLLOW.
4. Constructing Predictive Parsing Table using FIRST & FOLLOW.
5. Parse the input string with the help of Predictive Parsing table.

* FIRST( ) :

FIRST(α) is a set of terminal symbols that are first symbols appearing at RHS in derivation of 'α'.

→ Following are the rules used to compute FIRST functions:

- If 'X' is a terminal then First(X) is just 'X'.
- If there is a Production $X \rightarrow \epsilon$(epsilon) then add 'ϵ' to first(X).
- If there is a Production $X \rightarrow aB$ then add 'a' to first(X) where 'a' is a terminal & 'B' is non-terminal.
- If there is a Production $X \rightarrow Y_1Y_2 \cdots Y_k$ then first($Y_1Y_2 \cdots Y_k$) to first(X).
- First($Y_1Y_2 \cdots Y_k$) is either:
  - i) First(Y_1) (if First(Y_1) doesn't contain 'ϵ')
  - ii) OR (if First(Y_1) does contain 'ϵ') then First($Y_1Y_2 \cdots Y_k$) is everything in First(Y_1) <except for 'ϵ'> as well as everything in First($Y_2 \cdots Y_k$).

Scanned by CamScanner

iii) If First(Y_1) First(Y_2) ... First(Y_k) all contain 'ϵ' then add 'ϵ' to First($Y_1 Y_2 \cdots Y_k$) as well.

* Example :

(1). $E \rightarrow E+T \mid T$ (Left Recursion exists.)
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

sol : we need to remove left recursion from the grammar.
After removal of left recursion the grammar is :

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id.$

(*) First(E) = $E \rightarrow T$ now go to 'T' RHS production & look.
$T \rightarrow FT'$ (again non-terminal)
so go to 'F' RHS side production
$F \rightarrow (E) \mid id$ (id → terminal.)
∴ First(E) = { (, id }

- First(E) = First(T) = First(F) = { (, id }
- First(E') = { +, ϵ }
- First(T') = { *, ϵ }

(2). $S \rightarrow aB \mid aC \mid Sd \mid Se$
$B \rightarrow bBc \mid f$
$C \rightarrow g.$

sol : First(S) = { a }
First(B) = { b, f }
First(C) = { g }

Scanned by CamScanner

* FOLLOW ( ) :

Let $S \rightarrow \alpha A a \beta$ where α & β may be terminals or non-terminals. FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of 'A'.

→ Rules for computing FOLLOW are :

- For the start symbol 'S' place "$" in FOLLOW (S).
- If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST (β) without 'ϵ' is to be placed in FOLLOW(B).
- If there is a Production $A \rightarrow \alpha B \beta$ @L $A \rightarrow \alpha B$ and FIRST (β) = {ϵ} then FOLLOW (A) = FOLLOW (B) i.e everything in FOLLOW (A) is in FOLLOW (B).

* Examples :

(1)
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$.

Sol : • FOLLOW (E) = { $,) }

check all RHS side Productions where 'E' is there, we have
$F \rightarrow (E) \mid id$
↳ first(')') = ')'.
∴ FOLLOW(E) = { $,) }
since 'E' is start symbol place '$' in its follow.

- FOLLOW(E') = { $,) }
- FOLLOW (T) = { +, $,) } ⇒ $E \rightarrow TE'$
  first(E') = {+, ε} → to follow (E) = FOLLOW (T)
  FOLLOW(T) = {+, $,) }
- FOLLOW (T') = { +, $,) }
- FOLLOW (F) = { *, +, $,) }

Scanned by CamScanner

(2). $S \rightarrow aB \mid aC \mid Sd \mid Se$

$B \rightarrow bBc \mid f$

$C \rightarrow g$.

Sol: FOLLOW(S) = { $, d, e }

FOLLOW(B) = { c }

FOLLOW(C) = { d, e, $ }

* Construction of Predictive Parsing table:

The construction of Predictive Parsing table is an important activity in Predictive Parsing method. This algorithm requires FIRST & FOLLOW functions.

Input: Context-free grammar 'G'.

Output: Predictive Parsing table 'M'.

Algorithm: For the rule $A \rightarrow \alpha$ of grammar 'G':

- For each 'a' in FIRST(α) create entry M[A, a] = $A \rightarrow \alpha$ where 'a' is a terminal symbol.
- For 'ε' in FIRST(α) create entry M[A, b] = $A \rightarrow \alpha$ where 'b' is the symbols from FOLLOW(A).
- All the remaining entries in the table 'M' are marked as syntax error.

→ This algorithm can be applied to any grammar 'G' to produce a Parsing table 'M'. If the grammar 'G' is left recursive or ambiguous then 'M' will have atleast one multiple defined entry.

Scanned by CamScanner

* LL(1) grammars :- A grammar whose Parsing table has no multiple-defined entries is said to be LL(1). The LL(1) grammars are:

- Scanned from left-to-right.
- are Parsed by a leftmost derivation
- have one symbol look ahead.

* Problems :-

(1) Verify whether the following grammar is LL(1) or not.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id.$

sol :- we need to remove left recursion. After removal the grammar is:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

- First(E) = First(T) = First(F) = {(, id}
- First(E') = {+, ε}
- First(T') = {*, ε}

- Follow(E) = {$,)}
- Follow(E') = {$,)}
- Follow(T) = {+, $,)}
- Follow(T') = {+, $,)}
- Follow(F) = {*, +, $,)}

Scanned by CamScanner

* Predictive Parsing Table :

Non-terminals → Input | terminals → id, +, *, (,), $ | First Productions → (| Follow sets →), $

Input	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'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

Since there are no multiple entries in the above table, the given grammar is LL(1) grammar.

Input string : id + id * id $ is parsed using above table as below :

Stack	Input	Action
$ E' T → store in reverse order	id + id * id $	$E \to TE'$
$ E' T' F	id + id * id $	$E \to FT'$
$ E' T' id	id + id * id $	$F \to id$
$ E' T'	+ id * id $	
$ E'	+ id * id $	$T' \to \epsilon$
$ E' T +	+ id * id $	$E' \to +TE'$
$ E' T	id * id $	stack top & current pointer of input points to same character. Hence Pop it.

Scanned by CamScanner

stack	Input	Action
$ E' T' F	id * id $	T → F T'
$ E' T' id	id * id $	F → id
$ E' T'	* id $	
$ E' T' F *	* id $	T → * F T'
$ E' T' F	id $	
$ E' T' id	id $	F → id
$ E' T'	$	
$ E'	$	T → ϵ
$	$	E' → ϵ

∴ Thus input string gets parsed.

(2) check whether the following grammar is LL(1) grammar or not.

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

Sol:- After left factoring, the above grammar becomes as:

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

First & Follow for the above grammar:-

Scanned by CamScanner

- First (S) = { i, a }
- First (S') = { e, ϵ }
- First (E) = { b }
- Follow (S) = { $, e }
- Follow (S') = { $, e }
- Follow (E) = { t }

→ Predictive Parsing table can be constructed as:

	a	b	e	i	t	$
S	S → a			S → i E t S S'		
S'			S' → ϵ S' → e S	multiple entries.		S' → ϵ
E		E → b				

Since we got multiple entries in M[S', e], this grammar is not LL(1) grammar.

* Error Recovery in Predictive Parsing:

e/n error is detected during Predictive Parsing when the terminal on top of stack does not match the next input symbol or when non-terminal 'A' is on top of the stack, 'a' is the next input symbol, & the parsing table entry M[A, a] is empty.

Scanned by CamScanner

There are two types of error recovery techniques :-

(a) Panic-mode error recovery :- It is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

(b) Phase level Recovery :- It is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert or delete symbols on the input & issue appropriate error messages. They may also pop from the stack.

* Bottom-Up Parsing :-

A Bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) & working up towards the root (the top).

→ Input string is taken first & we try to reduce this string with the help of grammar & try to obtain the start symbol. The process of Parsing halts successfully as soon as we reach to start symbol.

Scanned by CamScanner

* Reduction :- Parser tries to identify RHS of production rule & replace it by corresponding LHS. This activity is called "Reduction".

→ The Primary task in bottom-up Parsing is to find the productions that can be used for reduction. The bottom up Parse tree construction process indicates that the tracing of derivations are to be done in reverse order.

* Handle Pruning :-

Handle is a string of substring that matches the right side of Production & we can reduce such string by a non-terminal on left hand side Production.

(OR)

Handle of right sentential form γ is a production $A \rightarrow \beta$ & a position of γ where the string β may be found & replaced by 'A' to produce the previous right sentential form in rightmost derivation of γ.

*Example :- $E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow (E)$
$E \rightarrow id$

Consider string id. + id. * id.

$E \xrightarrow{rm} E+\underline{E}$
$E \xrightarrow{rm} E+\underline{E*E}$
$E \xrightarrow{rm} E+E*\underline{id}$

Scanned by CamScanner

$E \xrightarrow{rm} E + \underline{id} * id$

$E \xrightarrow{rm} \underline{id} + id * id$

→ A right most derivation in reverse order can be obtained by "Handle Pruning".

Right Sentential Form	Handle ← Handle	Reduced Production.
$\underline{id} + id * id$	id	$E \rightarrow id$
$E + \underline{id} * id$	id	$E \rightarrow id$
$E + E * \underline{id}$	id	$E \rightarrow id$
$E + \underline{E * E}$	$E * E$	$E \rightarrow E * E$
$\underline{E + E}$	$E + E$	$E \rightarrow E + E$
E		

* Shift - Reduce Parsing :

Shift-reduce Parsing is a form of bottom-up Parsing in which a stack holds grammar symbols & an input buffer holds the rest of the string to be parsed.

→ The initial configuration of shift reduce Parser is as below:

STACK	INPUT
$	ω $

where stack is empty & 'ω' is a string.

Scanned by CamScanner

The shift-Reduce Parser can Perform the following operations:

(1) Shift : Moving of symbols from input buffer onto the stack.

(2) Reduce : If the handle appears on the top of the stack then reduction of it by appropriate rule is done. That means RHS of rule is popped off & LHS is pushed in. This action is called 'Reduce action'.

(3) Accept : Announce successful completion of Parsing if the stack contains start symbol only & input buffer is empty at the same time.

(4) Error : A situation in which parser cannot either shift or reduce the symbols. It cannot even. Perform the accept action. This is called as error.

* Example :

(1) E → E-E
E → E*E
E → id

Perform shift-Reduce parsing of the input string "id1 - id2 * id3".

Sol :

Stack	Input Buffer	Parsing Action
$	id1 - id2 * id3 $	shift
$id1	- id2 * id3 $	Reduce by E → id
$ E	- id2 * id3 $	shift
$ E -	id2 * id3 $	shift
$ E - id2	* id3 $	Reduce by E → id
$ E - E	* id3 $	shift

Scanned by CamScanner

$ E-E↑	id3 $	shift
$ E-E*id3	$	Reduce by E→id
$ E-E*E	$	Reduce by E→E*E
$ E-E	$	Reduce by E→E-E
$ E	$	Accept

Here we have followed two rules:

1. If the incoming operator has more priority than in stack operator then perform 'shift'.
2. If the stack operator has same or less priority than the priority of incoming operator then perform 'reduce'.

* Conflicts during shift-Reduce Parsing:

Shift-reduce Parsing cannot be used for all types of CFG's. For some CFG's, shift-reduce parser can reach a configuration in which the parser knowing the entire stack contents & the next input symbol, cannot decide whether to shift to reduce (a shift/reduce conflict) or cannot decide which of several reductions to make (a reduce/reduce conflict).

* Example: Consider the following grammar where the productions are numbered as below:

E → E + T { Print '1' }

E → T { Print '2' }

Scanned by CamScanner

$T \rightarrow T * F$ { Print '3' }

$T \rightarrow F$ { Print '4' }

$E \rightarrow (E)$ { Print '5' }

$F \rightarrow id$ { Print '6' }

If a shift-reduce Parser writes the Production number immedi-tely after Performing any Production, what string will be Printed if the Parser input is id+id*id.

Sol:

Stack	Input	Operation
$	id+id*id$	shift
$id	+id*id$	Reduced by F→id Print '6'.
$F	+id*id$	Reduced by T→F Print '4'.
$T	+id*id$	Reduced by E→T Print '2'
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	Reduced by F→id Print '6
$E+F	*id$	Reduced by T→F Print '4
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	Reduced by F→id Print '6
$E+T*F	$	Reduced by T→T*F Print '3'
$E+T	$	Reduced by E→E+T Print '1'.
$E	$	

Scanned by CamScanner

∴ the final string obtained is 64264631.

* Operator Precedence Parsing :

A grammar 'G' is said to be operator Precedence if it Posses following Properties :

- No Production on the right side is 'ϵ' (epsilon).
- There should not be any Production rule Possessing two adjacent non-terminals at the right hand side.

* Example :

$$E \rightarrow EAE \mid (E) \mid -E \mid id$$
$$A \rightarrow + \mid - \mid * \mid / \mid \wedge$$

This grammar is not an operator Precedent grammar as in the Production rule $E \rightarrow EAE$.

→ It contains two consecutive non-terminals. Hence first we will convert it into equivalent operator Precedence grammar by removing 'A'.

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \wedge E$$
$$E \rightarrow (E) \mid -E \mid id$$

→ In operator Precedence Parsing, we will first define Precedence relations $<\cdot$, $=$, and $\cdot>$ between pair of terminals.

	means
• $P <\cdot q$	P gives more Precedence than q.
• $P = q$	P has same " as q.
• $P \cdot> q$	P takes Precedence over q.

Scanned by CamScanner

Operator Precedence (Relation) table :

	id	+	*	$
id		.>	.>	.>
+	<.	.>	.>	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Consider the string : id + id * id .

We will insert $ symbols at the start & end of the input string. We will also insert Precedence operator by referring the Precedence relation table.

$$\$ <\cdot \text{ id } \cdot> + <\cdot \text{ id } \cdot> * <\text{ id } \cdot> \$$$

* Steps to Parse the given string :

- Scan the input from left to right until first .> is encountered.
- Scan backwards over = until <. is encountered.
- The handle is a string between <. and .>.

Parsing is done as follows :

$ <. id .> + <. id .> * <. id .> $	Handle id is obtained b/w <. .> Reduce this by E → id
E + <. id .> * <. id .> $	Handle id is obtained b/w <. .> Reduce this by E → id
E + E * <. id .> $	Handle id is obtained b/w <. .> Reduce this by E → id

Scanned by CamScanner

E + E * E	Remove all the non-terminals
+ *	Insert $ at the beginning & at the end. Also insert the Precedence operators.
$ <· + <· * ·> $	The * operator is surrounded by <· ·>. This indicates that * becomes handle. That means we have to reduce E*E operation first.
$ <· + ·> $	Now '+' becomes handle. hence we evaluate E+E
$ $	Parsing is done.

* Advantage : It is simple to implement.

* Disadvantages :

- The operator like minus has two different Precedence (unary & binary). Hence it is hard to handle such tokens.
- This kind of Parsing is applicable to only small class of grammars.

* Application :

SNOBOL language uses operator Precedence Parsing.

Scanned by CamScanner