→ $S \to iEtS \mid iEtSeS \mid a$ —①
$E \to b$ —②

For $A \to \alpha A'$

$A \to iEtSA' \mid a$

$A' \to \epsilon \mid eS$

$E \to b$

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

⇓

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

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

→ Top-Down Parsing

Mainly Focuses on LL(1) class grammar

Left to Right Scanning — Left most derivation

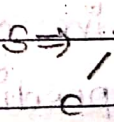
The basic idea behind Top-Down parsing is to construct an efficient non-backtracking form of Top-Down parser called a Predictive Parser. We define the class of LL(1) Grammar from which predictive parser can be constructed automatically.

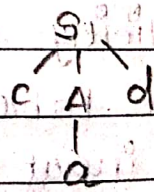
→ Recursive Descent Parser [Chance for Backtracking] less efficient.

$S \to cAd$

$A \to ab \mid a$

Input :- cad

$S \Rightarrow$ 

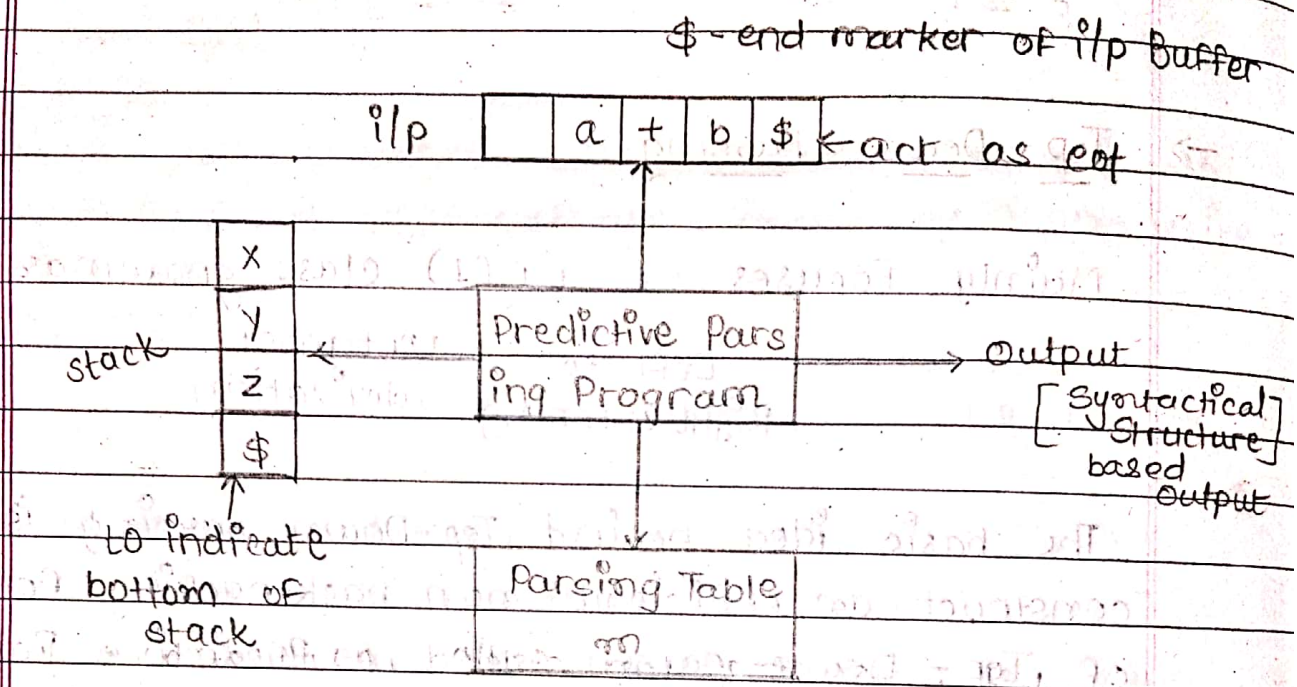
$\to$ 

→ Predictive Parser

In many cases, by carefully writing a grammar, eliminating (L.R.) from it, and left

Factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive descent parser that needs no backtracking i.e., a predictive parser.

Non-Recursive Predictive Parsing.



Model of N-R-P-P

★. Note :- Model of N-R-P-P = P-P = Top-down.

It is possible to build a N-R-P-P by maintaining a stack explicitly, rather than implicitly via recursive calls.

A table driven predictive parser [P-P] has an input buffer, a stack, a parsing table and an ~~Input~~ output stream.

The I/P Buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of input

string.

Initially the stack contains the start symbol of the grammar on top of the stack above $ sign and the parsing table is a 2-Dimensional array.

1. if $X = a = \$$
2. if $X = a \neq \$$

   pops X

   advance i/p ptr
3. if X is a non-terminal

   eg. $M[X, a] = X \rightarrow UVW$

* Algo. Non-Recursive-Predictive-Parsing [N-R-P-P]

set ip to point to the first symbol of w$;
repeat
  Let X be the top stack symbol and a the symbol pointed to by ip;
  if X is a terminal or $ then
    if X = a then
      pop X from the stack and advance ip
    else error()
  else
    if $M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ then begin
      pop X from the stack;
      push $Y_k, Y_{k-1} \ldots Y_1$ on to stack with Y_1 on top;
      output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$
    end
    else error()
until X = $

→ FIRST AND FOLLOW

→ **FIRST**

If α is any string grammar symbols, Let FIRST(α), be the set of terminals that begin the strings derived from α.

→ Rules

1. If X is a terminal then first(X) is {X} itself.
2. If $X \to \epsilon$, then add ϵ to first(X).
3. If X is a non-terminal then, use production $X \to Y_1 Y_2 \dots Y_K$.

→ **FOLLOW**

Follow(A), for non-terminal A to be the set of terminals a that can appear immediately to the right of A in some sentencial form.

→ Rules

1. Place $ in FOLLOW(S), where S is the start symbol.
2. If there is a production $A \to \alpha B\beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B)
3. If there is a production $A \to \alpha B$ or $A \to \alpha B\beta$ where FIRST(β) contains ϵ then everything

in FOLLOW(A) is in FOLLOW(B).

Ex. Grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Start Symbol	FIRST	FOLLOW
E	(, id	$,)
E'	+, ϵ	$,)
T	(, id	+, $,)
T'	*, ϵ	+, $,)
F	(, id	*, +, $,)

Note:- In First identify particular Terminals based on all expression.

i.e., $E \rightarrow \boxed{T}E'$ Symbol E contains FIRST(c, id)

$T \rightarrow \boxed{F}T'$

$F \rightarrow (E) \mid id$

1. $E \rightarrow TE'$

Applying Rule 2 of FOLLOW.

$A \rightarrow \alpha B \beta$.

According to Rule, place ϵ as

$A \rightarrow \epsilon TE'$

FIRST(E') contains +, ϵ

Take + and put in FOLLOW(T)

* Construction of Predictive Parsing Table :-

I/p :- G
o/p :- parsing tabe M

1. For such production $A \to \alpha$ of the grammar, do step 2 and 3

2. For each terminal a in FIRST(α) add $A \to \alpha$ to M[A, a]

3. If ϵ is in FIRST(α), Add $A \to \alpha$ to M[A, b] for each terminal b in Follow[A]. If ϵ is in FIRST(α) and \$ is in FOLLOW(A) add $A \to \alpha$ to M[A, \$]

4. Make each undefined entry of m be error

eg. Grammar

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

	FIRST	FOLLOW
E	(, id	), \$
E'	+, ϵ	), \$
T	(, id	*,), \$
T'	*, ϵ	*,), \$
F	(, id	+, *,), \$

* I/p symbols.

Non ter-minal.	Input symbol.					
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E → ∈	E' → ∈
T	T → FT'			T → FT'		
T'		T' → ∈	T' → *FT'		T' → ∈	T' → ∈
F	F → id			F → (E)		

* moves made by predictive parser :
i/p :- id + id * id

STACK	INPUT	OUTPUT
$E	id + id * id $	
$E'T	id + id * id $	E → TE'
$E'T'F	id + id * id $	T → FT'
$E'T'id	id + id * id $	F → id
$E'T'	+ id * id $	
$E'	+ id * id $	T' → ∈
$E'T+	+ id * id $	E' → +TE'
$E'T	id * id $	
$E'T'F	id * id $	T → FT'
$E'T'id	id * id $	F → id
$E'T'	* id $	
$E'T'F*	* id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'∈	$	T' → ∈
$	$	E' → ∈

→ Bottom-up Parsing

In general, bottom-up syntax analyzer, known as shift reduce parsing.

An easy to implement form of shift-reduce parsing, called operator-precedence parsing.

A more general method of shift-reduce parsing called LR-parsing.

shift reduced parsing, attempts to constructs a parse tree for an input string beginning at the bottom and working up towards the top.

So, it is a process of reducing a string "ω" to start symbol of whole grammar.

Ex:-

Grammar

S → aABe

A → Abc | b

B → d

Input :- abbcde

→ abbcde

→ aAbcde

→ aAde

→ aABe

→ S

Reduction steps.

Bottom to top ⇒ (Left to Right).

$S \xrightarrow{rm} aABe \xrightarrow{rm} aAde \xrightarrow{rm} aAbcde \xrightarrow{rm} abbcde$

Right most derivation for Reduction.

→ Handles (substring)

Handle of a string is a substring that matches the right side of production, and where

reduction to the non-terminal on the left side of the production represents one step along the reverse of right most derivation.

→ Handle pruning

Reducing β to A in input sentence is referred as pruning the handle.

It is process of removing the children of A from the parse tree.

Right most derivation in reverse can be obtained by handle pruning.

Ex:- $E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$.

Input :- id + id * id

$E \Rightarrow E+E$
$\Rightarrow E+E*E$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow id+id*id$

$E \Rightarrow E*E$
$\Rightarrow E*id$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow id+id*id$

⇒. id + id * id

E + id * id

E + E * id

E + E * E

E + E

E

⇒ id + id * id

E + id * id

E + E * id

E * id

E * E

E

→ Operator Precedence Parsing

operate - a, b

Relation	Meaning
1. $a \lessdot b$	a give precident to b
2. $a \doteq b$	a has same precident as b
3. $a \gtrdot b$	a takes precident over b

1. b is priority over a.
2. a is priority over b.
3. a and b is priority.

→ left end marker $\lessdot$
→ right end marker $\gtrdot$

* using operator-precedence relations :-

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

Eg. operator-precedence relations

Eg. id + id * id
→ $id + id * id $ (Put $ in both side)
→ $ $\lessdot$ id $\gtrdot$ + $\lessdot$ id $\gtrdot$ * $\lessdot$ id $\gtrdot$ $

(i) Left to Right scanning :-
- first $\gtrdot$

(ii) $\gtrdot$ find them backward scanning till $\lessdot$ is

(iii) $\gtrdot$ $\lessdot$ whatever in this true it is drop including and marker a particular path.

→ ($ <· id ·> + <· id ·> * <· id ·> $)

(i) $ + <· id ·> * <· id ·> $
$ + * <· id ·> $
$ + * $
} (i to iii)

(sequence of operator is only required no. is also not required, work is till then operator is

→ $ + * $

$ <· + <· * ·> $

a <· b

$ + * $

$ + → <·

+ * → <·

* $ → ·>

* Algorithm of operator precedence parsing.

set ip to point to the 1st symbol of w$;
repeat forever
  if $ is on top of the start and
    ip points to $ then
  return
  else begin
  Let a be the top most terminal symbol on
    the stack and b be the symbol pointed
    by ip;
  if a <· b or a ≐ b then begin push b
    onto the stack;
  advance ip to the next i/p symbol
  end;
  else if a ·> b then
  repeat

pop the stack
until the top stack terminal is related <. to the terminal most recently popped.
else error()
end.

* Priority of operators

1. * and / are of highest precident and left associative.
2. + and - are of lowest precident and left associative

→ LR [Left to Right - Right most derivation] Parser

→ The techniques is called LR(K) Parsing.
→ The "L" is for left to right scanning of the input.
→ The "R" for constructing a right most derivation in reverse.
→ "K" reference no of i/p symbol.

* Types of LR Parser.

1. SLR → Simple LR
- It is less powerful but easy to implement parsing techniques.

2. Canonical LR

- It is most powerful but expensive also.

3. LALR → loop ahead LR.
- It is intermediate for above two parsing technique. In terms of performance and cost.

→ Stack implementation of shift-reduce parsing

Stack	I/P
$	w$

Stack	I/P
$s	$

Model of bottom-up parsing

→ Given Grammar

E → E+E
E → E*E
E → (E)
E → id

I/p:- id+id*id

STACK	INPUT	ACTION
$	id+id*id$	shift
$id	+id*id$	reduced by E→id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduced by E→id
$E+E	*id$	shift
$E+E*	id$	shift
$E+E*id	$	reduced by E→id
$E+E*E	$	reduced by E→E*E
$E+E	$	reduced by E→E+E
$E	$	accept

To implement a shift reduce parser, it is required to use a stack to hold grammar symbols and an input buffer to hold the string "w" to be parsed.

Following are primary operation or possible action a shift reduce parser can make.

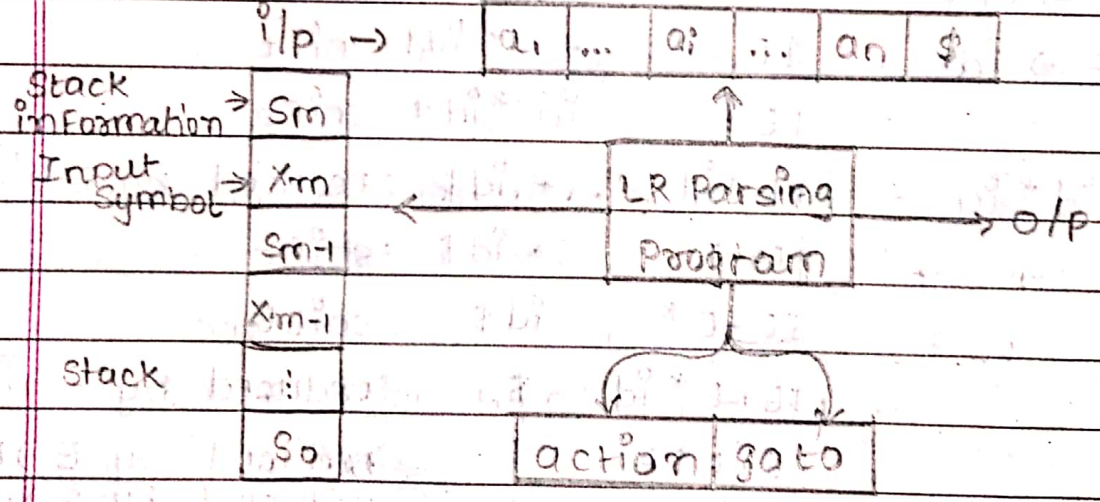
1. shift
2. Reduce
3. Accept
4. Error

→ operator Precedence Parsing

operator Grammar

Grammar's have the property that no production right side is ϵ nor has two adjacent non-terminal said to be operator grammar.

→ Model of an L-R Parser / Bottom-up Parser



Model of an LR Parser consist of an input, an output, a stack, a driver program and a parsing table that has two parts [action and goto]

A configuration of an LR parser is a pair whose first component is the stack content and whose second component is unexpected input.

Initial $(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_i a_{i+1} \dots a_n \$)$

1. If action $[s_m, a_i] = \text{shift } s$
   $(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i s, a_{i+1} \dots a_n \$)$
2. If action $[s_m, a_i] = \text{reduce } A \to \beta$
   $(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} A s, a_i a_{i+1} \dots a_n \$)$

   ↑ reduced length got by [AS] Non-Terminal.

   /* r is the length of β */
3. If action $[s_m, a_i]$ = accept, parsing is completed.
4. If action $[s_m, a_i]$ = error, the parser has discovered an error.

→ LR parsing program.

```
set ip to point to the first symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
        a be the symbol pointed to by ip;
    if action [s,a] = shift s' then begin
        push a then s' on the top of the stack;
        advance ip to the next input symbol
    end
    else if action [s,a] = reduce A → β then begin
        pop 2* |β| symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto [s',A] on top of the stack;
```

```
    end
    else if action [s,a] = accept then
        return
    else error()
end
```

* LR Grammar.

The grammar for which we can construct a parsing table is said to be an LR grammar.

LR(K) grammar

→ Constructing SLR parsing table

SLR → simple.

ex:- $A \to XYZ$

$\Rightarrow A \to .XYZ$

$\Rightarrow A \to X.YZ$

$\Rightarrow A \to XY.Z$

$\Rightarrow A \to XYZ.$

} Process.

ex:- $A \to \epsilon$

$\Downarrow$

$A \to .\epsilon$

$A \to \epsilon.$

$A \to .$

[Hint:- $S\epsilon = \epsilon S = S$ (Omit)]

1. The closure operation:-

Eg:-

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

Step 2:- Argument given grammar

(S.S) $E' \to E$

$E \to E + T$

$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to id$

Step 3 $E' \to .E$

$E \to .E + T$

$E \to .T$

$T \to .T * F$

$T \to .F$

$F \to .(E)$

$F \to .id$

Step 1:- Initially, every item in I is added to closure (I).

If $A \to \alpha . B\beta$ is in closure of I and $B \to \gamma$ is a production, then add the item $B \to .\gamma$ to I, if its not already there. We apply these rules until no more new items can be added to closure of I.

1. Kernel items:- It include the initial item, $S' \to .S$, and all items whose dots are not at the left end.

2. Non-kernel items :- which have there dot is at the left end.

(2) goto operation :-

In goto (I, x), I is a set of items and x is a grammar symbol, goto (I, x) is defined to be the closure of the set of all items, $A \to \alpha X . \beta$ such that $A \to \alpha . X\beta$ is in I.

eg. :- $E' \to E$. Caftey. nothing is not goto oper. otherwise its go to operator.

eg. $E' \to E . + T$

$E \to E . + T$

$E \to E + . T$

$T \to . T * F$

$T \to . F$

$F \to . (E)$

$F \to . id$.

(3) The sets of construction item :-

```
procedure items(G');
begin
    C := {closure({[S' → .S]})};
    repeat
        for each set of items I in C and each
                grammar symbol X
        such that goto(I, X) is not empty and not
                in C do
            add goto(I, X) to C
    until no more sets of items can be added to C
end.
```

2. Given Grammar

$E' \to E+T \mid T$

$E \to T$

$T \to T*F \mid F$

$T \to F$

$F \to (E) \mid id$

$F \to id.$

Start Symbol	FIRST	FOLLOW.
E		
T		
F		

1. Generation of Canonical LR(0) items :-

I_0	$I_1 = E$	$I_4 = ($	$I_5 = id$	$I_7 = *$
$E' \to .E$	$E' \to E.$	$F \to (.E)$	$F \to id.$	$T \to T*.F$
$E \to .E+T$	$E \to E.+T$	$E \to .E+T$	$I_6 = +$	$F \to .(E)$
$E \to .T$	$I_2 = T$	$E \to .T$	$E \to E+.T$	$F \to .id$
$T \to .T*F$	$E \to T.$	$T \to .T*F$	$T \to .T*F$	$I_8 = E$
$T \to .F$	$T \to T.*F$	$T \to .F$	$T \to .F$	$F \to (E.)$
$F \to .(E)$	$I_3 = F$	$F \to .(E)$	$F \to .(E)$	$E \to E.+T$
$F \to .id$	$T \to F.$	$F \to .id$	$F \to .id$	$I_9 = T$
		$I_{11} =)$	$I_{10} = F$	$E \to E+T.$
		$F \to (E).$	$T \to T*F.$	$T \to T.*F$

Algorithm

→ Constant C LR(0)

→ Parsing actions

1. $A \to \alpha.a\beta$ (shift)
2. $A \to \alpha.$ (reduce)

For all a in Follow(A)

A may not be S'

3. $S' \to S.$ accept ($)

-goto -error -initial state

	action						goto		
State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			