Backtracking Parsing :-

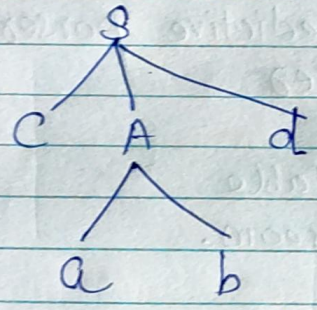
- If we make a sequence of erroneous and subsequently discover a mismatch, we undo the effects and rollback the input pointer.

This method is also known as Brute force parsing.

$S \rightarrow cAd$

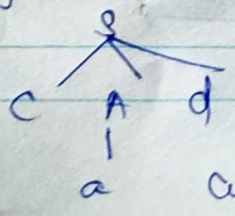
$A \rightarrow ab|a$

Input string $w = cad$ is to generate.



String generated : cabd

* but $w = cad$, the third symbol is not matched.
* So Report error
* go back to A.
* apply alternat production



$w = cad$

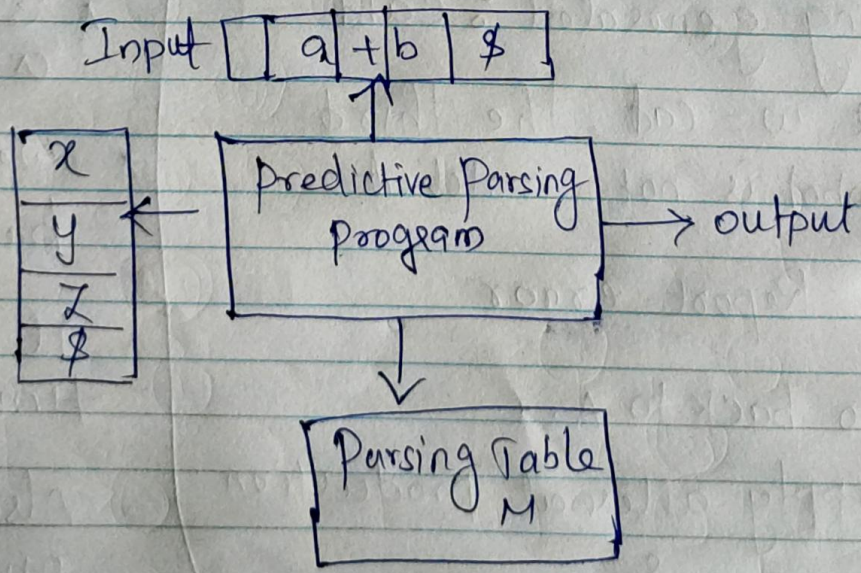
* String generated is $w = cad$
* We have used back tracking
* It is costly and time consuming
* Also outdated one.

Recursive descent Parsing

* Execute a set of recursive procedure to process the input
* The sequence of procedures are called implicitly, defines the parse tree for the input.

Non Recursive predictive Parsing :
(Table driven parsing)

* It maintains a stack explicitly, rather than implicitly via recursive calls.
* A Table driven predictive parser has
  * Input buffer
  * A stack
  * A parsing table
  * Output stream.



Transition diagram for predictive parser:-

In case of predictive parser, one diagram is drawn for each non terminal of the grammar.

The edges of the transition diagram will be either terminals or non terminals.

To construct a transition diagram for a predictive parser,

(i) Eliminate the left recursion from the grammar
(ii) Left factorize the grammar
(iii) For each non terminal A of the grammar do,
  a) Create a initial and final state
  b) For each production of the form,
  $A \rightarrow X_1 X_2 \cdots X_n$
  Create a path from initial to final state, the edges name as $X_1, X_2 \cdots X_n$.

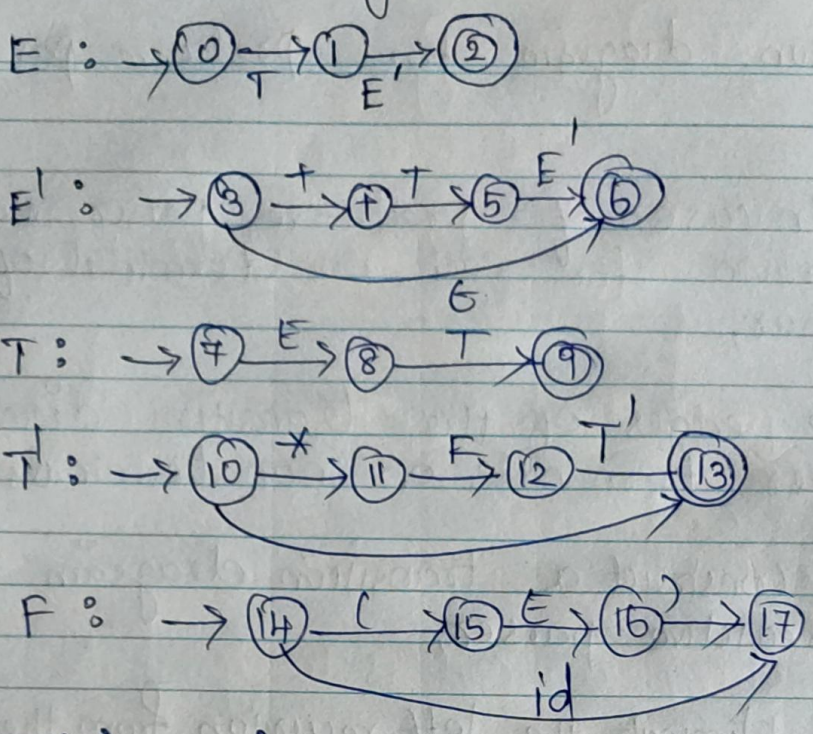
Draw the transition diagram for the given grammar.

$E \rightarrow E+T/T$
$T \rightarrow T*F/F$
$F \rightarrow (E)/id.$

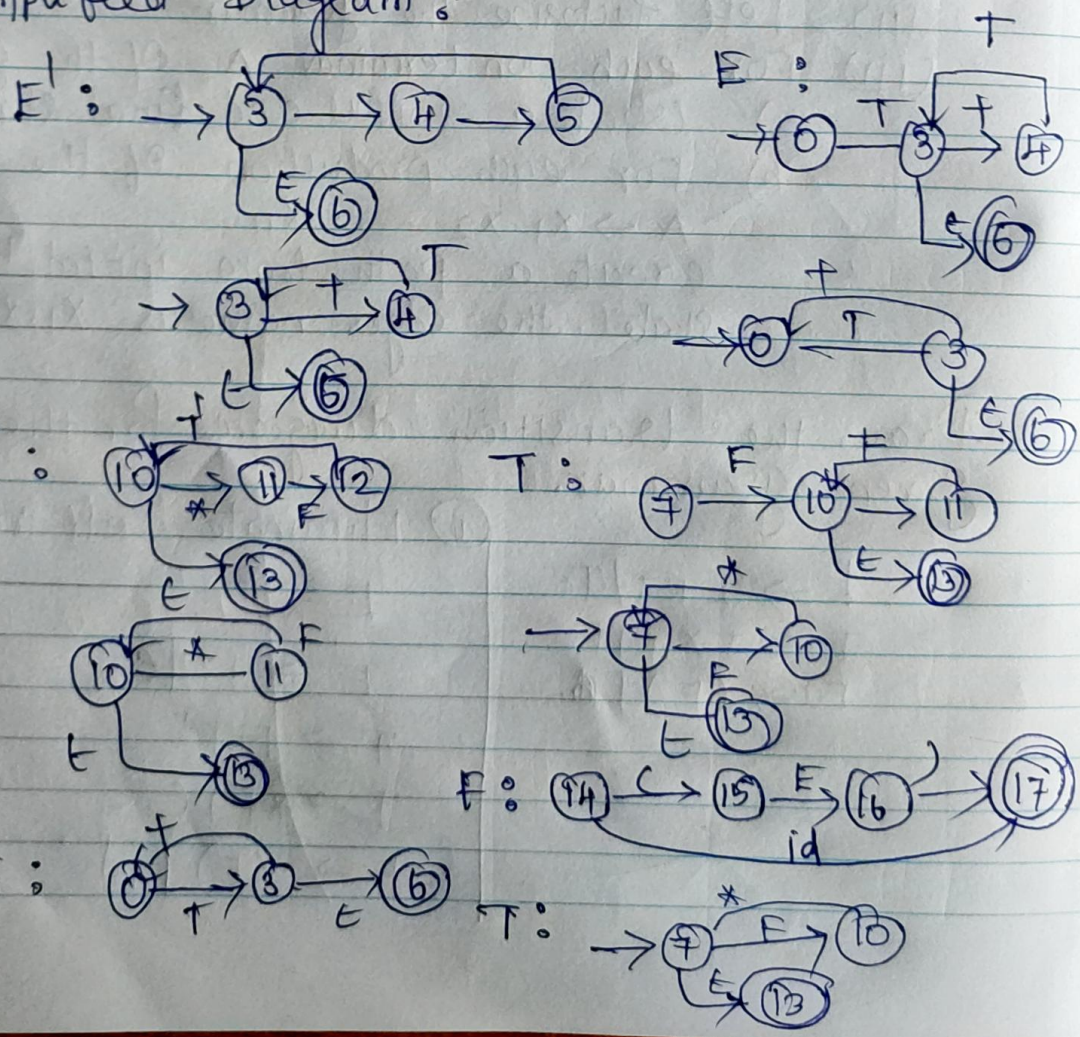
① Eliminate left recursion.

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

(2) If (G1) is already Left factorized



Simplified Diagram :-



Constructions of predictive parser is aided by two functions :-

* FIRST * FOLLOW

These functions are used to fill the entries in the table of predictive parser.

Computation of FIRST :-

(i) If there is a production of the form

$X \to a$ (or) $X \to a\alpha$

where $X \to$ is a Non terminal
$a \to$ is a terminal
$\alpha \to$ is a grammar symbol

then, FIRST $(X) = \{a\}$

(ii) If there is a production $X \to \epsilon$, then

FIRST $(X) = \{\epsilon\}$

(iii) If there is a production $X \to A$ or $X \to A\alpha$ then

FIRST(X) = FIRST(A)

(iv) If $X \to Y_1 Y_2 \ldots Y_n$ is a production with y_i is a nonterminals then

FIRST (X) = FIRST(Y_1), if FIRST(Y_1) contains

'ε' then goto Y_2 add the FIRST(Y_2) with Y_1 and continue the process until no 'ε' is found in the first.

Computation of Follow.

(i) place $ in FOLLOW(S) where S is a start symbol and $ is the input end marker.

(ii) If there is a production of the form $A \to \alpha B\beta$

where α and β are grammar symbols and β is a non-terminals then Follow(β) = First(β) except 'ε'.

if β is a non-terminal and Follow(β) = β is a terminal symbol.

(iii) If the production is the form $A \to \alpha B$ or $A \to \alpha B\beta$ then Follow(B) = FOLLOW(A) (if β is ε)

Construction of predictive parsing Table:-

(i) For each terminal 'a' in FIRST(A) add a production $A \to a$ in $M[A, a]$

(ii) If ϵ is in FIRST(A), for each terminal 'b' in FOLLOW(A) add a production $A \to \epsilon$ in $M[A, b]$

1. Construct a predictive parser for the following grammar.

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

Left Recursive

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

Remove Left Recursive:-

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

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

Calculation of FIRST and FOLLOW.

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = { +, ε }

FIRST(T') = FIRST(F) = { (, id }

FIRST(T') = { *, ε }

FIRST(F) = { (, id }

FOLLOW(E) = {), $ }

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

FOLLOW(T) = FIRST(E') = { +,), $ }

FOLLOW(T') = FOLLOW(T) = { +,), $ }

FOLLOW(F) = FIRST(T') = { *, +,), $ }

Computation of the Table :-

Non Terminal / Terminal	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	$F \to id$			$F \to (E)$		

Parsing :- **id + id * id**

Stack	Input	Behaviour
$ E	id+id*id$	Start
$ E'T	id+id*id$	Push
$ E'T'F	id+id*id$	push
$ E'T'id	id+id*id$	push
$ E'T'	+id*id$	pop
$ E'	+id*id$	Push
$ E'T'+	+id*id$	push
$ E'T'	id*id$	Pop
$ E'T'F	id*id$	push
$ E'T'id	id*id$	push
$ E'T'	*id$	pop
$ E'T'F*	*id$	push
$ E'T'F	id$	pop

$E'T'id$	$id\ \$$	push
ET'	$\$$	pop
E'	$\$$	push
E	$\$$	push.

Since, the Table has no multiple entries, a given grammar is an LL(1) grammar.

2) Construct a predictive parser for the given grammar.

$S \to iEtS \mid iEtSeS \mid a$
$E \to b.$

|| Left Factor

→ No Left Recursion

→ Left Factoring

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

First(S) = {i, a}
First(S') = {e, ∈}
First(E) = {b}

Follow(S) = {First(S')} = {e, $}
Follow(S') = {e, $}
Follow(E) = {b}

NT \ T	i	t	a	e	∈	b	$
S	$S \to iEtSS'$		$S \to a$				
S'				$S' \to eS$ $S \to \epsilon$			$S \to \epsilon$
E						b	

Multiple entries so no LL(1) grammar

3. Construct a predictive parser for the given grammar.

$S \rightarrow AaAb \mid BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

(1) Eliminate Left Recursive and Left factoring

The given grammar is already Removed Left Recursion and Left factoring.

(2) Calculating First and Follow.

First(S) = {a, b}
First(A) = {ϵ}
First(B) = {ϵ}

Follow(A) = {a, b}
Follow(B) = {a, b}
Follow(S) = {$}

(3) Computation of the table.

NT \ T	a	b	$
S	S → AaAb	S → BbBa	
A	A → ϵ	A → ϵ	
B	B → ϵ	B → ϵ	

Show the behaviour of ab by predictive parser.

Stack	Input	Description.
$ S	ab $	Start
$ bAaA	ab$	push
$ bAa	ab$	push
$ bA	b$	pop
~~$ bAaBbB~~	~~b$~~	~~Push~~
~~$ baBb~~	~~b$~~	~~push~~
$ B	$	push
$	$	pop.

Since the table has no multiple entries give grammar is a LL(1) grammar.

4) Construct a predictive parser for the given grammar.

$S \rightarrow (L) | a$

$L \rightarrow L, S | S.$

1) Remove Left Recursion and Left factoring.

$S \rightarrow (L) | a$

$L \rightarrow SL'$

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

2. The grammar is already left Factorized.

3. Calculation of First and Follow:-

First(S) = {(, a}
First(L) = {(, a}
First(L') = {,, ∈}

Follow(S) = {$, ,,)}
Follow(L) = {)}
Follow(L') = {)}

4. Computation of Table.

Non Terminal / Terminal	(	)	a	,	$
S	S→(L)		S→a		
L	L→SL'		L→SL'		
L'		L'→∈		L'→,SL'	

show the behavious of (a,a) by the predictive parser

Stack	Input	Behaviour
$ S	(a,a) $	start
$) L (	(a,a) $	push
$) L	a,a) $	pop
$) L' S	a,a) $	push
$) L' a	a,a) $	push
$) L'	,a) $	pop
$) L' S ,	,a) $	push
$) L' S	a) $	pop
$) L' a	a) $	push
$) L'	) $	pop
$)	) $	push
$	$	pop [accept.

Bottom up Parsing.
[Shift - Reduce parsing]

Types of parsing:-

(i) OR parsing (operator precedence parsing)
(ii) LR parsing (Left to Right parsing)
→ Right most derivation in Reverse
→ Left to Right scanning.

This parsing constructs the parse tree for an input string beginning at the leaves and working up towards the root (i.e) a string is reduced to the start symbol.

At each step a particular substring matching the right side of a production

it replaced by the non terminal on the left of the production.

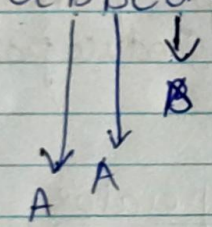
If a substring is choosen correctly at each step, a right most derivation is traced out in reverse.

Example:-

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

In bottomup parsing the string 'abbcde' is verified as follows.

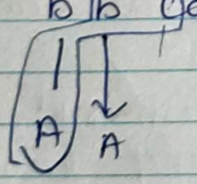
abbcde



Handle $A \rightarrow b$

aAAcBe Can't arrive start symbol.

a b b c d e



Handle
$A \rightarrow Abc$

aAde

Handle
$B \rightarrow d$

d

aABe

Handle
$S \rightarrow aABe$

S → start symbol.

NOTE : Right most Derivation is in Reverse.

$S \rightarrow aABe$

$A \rightarrow d$

$aAbcde$

$abbcde$.

Handle of a String :- substring that matches the RHS of some production and whose reduction to the nonterminal on the LHS is a step along the reverse of some rightmost derivation.

$S \rightarrow \alpha A \gamma$ (rm)

$A \rightarrow \alpha \beta \gamma$. (rm).

NOTE :-

Right Sentential form of a unambiguous grammar have one unique handle.

Example :-

Handle pruning :- The process of discovering a handle and reducing it to the appropriate left hand side is called handle pruning.

Handle pruning is the basics for bottomup parsing.

To construct the rightmost derivation, Apply the following steps:-

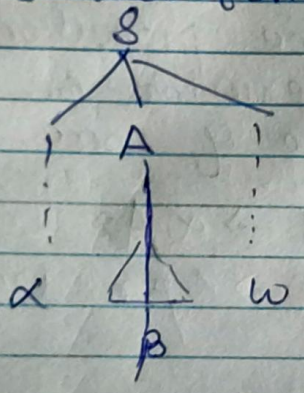
$S = r_0 = r_1 = r_2 \cdots r_{n-1} = w$

For $i \leftarrow n$ to 1

Find the handle $A_i \rightarrow \beta_i$ in r_i

replace β_i with A_i to generate r_{i-1}

Consider the part of parse tree of a certain right sential form,



$A \rightarrow \beta$ is a handle for $\alpha\beta w$.

Shift - Reduce parsing with a stack:-

There are 2 problems with this technique.

(i) To locate the handle

(ii) Decide to which production to use

General construction using stack:-

1. "Shift" input symbols onto the stack until a handle is found on top of it.
2. "Reduce" the handle to the corresponding non Terminal
3. "Accept" when the input is consumed and only the Start Symbol is on the stack.
4. "Error" - call an error reporting/recovery routine.

Stack Implementation of Shift-Reduce parser:

* The shift Reduce parser consists of input buffer, stack and parse Table.
* The input buffer consists of strings, with each cell containing only one input symbol.
* The stack contains the grammar symbol,

  Shift → insert grammar symbols
  Reduce → reduce " "
  after obtaining the handle from the buffer.

Go to / action :-

Which are constructed using terminal, non terminal and compiler items.

Example for stack implementation :-

G: S → AA , A → aA , A → b

Input string :- abab$

Stack	Input string	Action
$	abab$	Shift
$a	bab$	Shift
$ab	ab$	Reduce (A → b)
$aA	ab$	Reduce (A → aA)
$A	ab$	Shift
$Aa	b$	Shift
$Aab	$	Reduce (A → b)
$AaA	$	Reduce (A → aA)
$AA	$	Reduce (S → AA)
$S	$	Accept

Viable prefixes :-

The set of prefixes of a right sentential form that can appear on the stack of a shift reduce parser are called viable prefixes.

Conflicts

- Shift Reduce conflicts
- Reduce Reduce conflict.

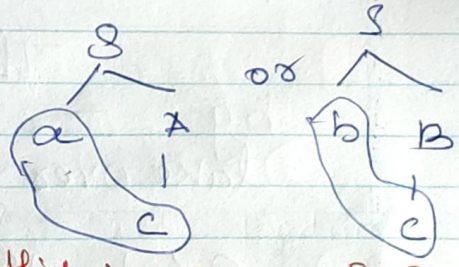
Shift / Reduce conflict:-

Ex:- stmt → if expr then stmt | if exp then stat else stmt | any other stmt.

if exp then stmt is on the stack, in this case, we can't tell whether it is a handle or not.
(i.e) Shift Reduce conflict.

Reduce / Reduce conflict:-

Ex: S → aA | bB
A → c
B → c



Input string = ac.
(i.e) Reduce Reduce conflict. LMD RMD

Operator precedence parser:-

* The only parser that accepts the ambigious grammar.
* no ∈ production and no two non terminal adjacent to each other on the right side.
* The small and important class of bottom up parser.
* It defines the mathematical operators and their relations in the grammar.

Ex1 : $E \to E+E \mid E-E \mid id$ is operator grammar.

Ex2 : $E \to AB$ — 2 non Terminal adjacent to each other

$E \to a$

$B \to b$

So not an operator grammar.

Ex3 : ~~$E \to EOE$~~

$E \to E \Theta E \mid id$

not operator grammar.

Operator precedence relation :-

The relations are represented by $\doteq$, $\lessdot$, $\gtrdot$ (Equal, yield and take over) precedence.

$a \lessdot b$ then, b has the highest precedence than a

$a \doteq b$ Same precedence

$a \gtrdot b$ then, a has the highest precedence than b.

Common ways for determining the precedence relation between pair of terminals :-

1. Traditional notations of associativity and precedence

2. First construct an unambiguous grammar for the language with correct

associativity and precedence in its parse tree.

Construction of operator precedence parser:-

① Leading
② Trailing have to be computed

for OR or operator grammar.

Computation of Leading :- L(A)

(i) a is in leading of A if,

1) There is a production A→ay where y is any grammar symbol

2) A→ Xay where y is any grammar symbol.

(ii) Lead(A) = Lead(B) : If there is a production A→B.

Computation of Trailing :- T(A)

(i) a is a trailing(A), if,

(1) There is a production A→Xa where X is any grammar symbol.

(2) There is a production A→Xay where X is a single non terminal.

(ii) Trailing(A) = Trailing(B), if there is a production A→B.

NOTE:-

If two continuous terminal symbols appear in the production on the right side then those two terminals are copies to the leading an trailing.

Example:- $A \to abcd$

$L(A) = \{a, b\}$

$T(A) = \{c, d\}$.

Computation of precedence Relation Table:-

1. $\$ < \circ$ all terminal symbols present in the leading of start symbol of the grammar. And all terminals in the trailing of the start symbols $\circ >$ over $\$$.

2. If $X \to ab$ is a production where a and b are terminal symbols then $a = b$ and if $X \to aBb$ is a production with a and b are terminal symbols and B is a non terminal $a \doteq b$.

3. if $X \to ay$ is a production with a as terminal and y as non terminal then $a < \circ$ all terminal present in Leading(y).

4. If $X \to Ya$ is a production with Y as non terminal and a as terminal then all trailing symbols $(Y) \Rightarrow a$

5. All Undefined Symbols/entries in the precedence table are errors.

Behaviours of the Input String:-

1. If $ & $ on the top the stack and current input symbol is $ then accept and announce successful completion.

2. If the top of the Stack is 'a' and the current input symbol is 'b'
   If ($a \lessdot b$) or $a \doteq b$ then shift 'b' all with the precedence relation.

3. If the top of the Stack is 'a' and the current input symbol is 'b' with the relation $a \gtrdot b$ then pop all the symbols from the stack until the first $\lessdot$ symbol is reached in the Stack.

4. Announce error if unspecified entries are encountered.

1. Construct the operator precedence parser for the following grammar.

$S \rightarrow L=R$
$S \rightarrow R$
$L \rightarrow *R$
$L \rightarrow id$
$R \rightarrow L$

1. Grammar is in correct form.
   - NO ϵ productions
   - NO two non terminals are adjacent.

2. Computation of Leading and Trailing :-

Leading(S) = Leading(R) = Leading(L) = {=, *, id}

Leading(L) = {*, id}

Leading(R) = {*, id}

Trailing(S) = {=, Trailing(L)}
= {=, id, *}

Trailing(R) = {id, *}

Trailing(L) = {id, *}

3. Computation of Operator Relation Table.

1. $ ⋖ = , $ ⋖ * , $ ⋖ id
   = ⋗ $, * ⋗ $, id ⋗ $
2. = ⋖ * , = ⋖ id
   id ⋗ = , * ⋗ =
3. * ⋖ id , * ⋖ *

$ id = id $

$ ⋖

	id	=	*	$
id		⋗	⋗	⋗
=	⋖		⋖	⋗
*	⋖	⋗	⋖	⋗
$	⋖	⋖	⋖	

Stack	Input	Action	
$	id = id $	shift	
$id	= id $	pop	(id ⇒ =)
$ =	id $	shift	
$ = id	$	shift	= ⇐ id
$ =	$	pop	id ⇒ $
$	$	pop	= ⇒ $
$	$	accept	

2) Construct the operator precedence parser for the following grammar.

1. Grammar:- $S \to (L) \mid a$
   $L \to L, S \mid S$ is in correct form

   (i) no ε production
   (ii) no two non terminals are adjacent.

2. Computation of Leading & Trailing.

   Leading(S) = {(, a}

   Leading(L) = {, , Leading(S)}
   = {, , (, a}

   Trailing(S) = {), a}

   Trailing(L) = {, ,), a}

3. Computation of operator Relation Table.

	(	)	a	,	$
(	$<\cdot$	$\doteq$	$<\cdot$	$<\cdot$	
)		$\cdot>$		$\cdot>$	$\cdot>$
a		$\cdot>$		$\cdot>$	$\cdot>$
,	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$	
$	$<\cdot$		$<\cdot$		

Stack	Input	Action.
$	(a)$	start | shift
$<\cdot(	a)$	shift
$<\cdot(<\cdot a	)$	~~shift~~
~~$<\cdot(<\cdot a \cdot>$~~	~~$~~	~~pop~~
~~$<\cdot(\doteq$~~	~~$~~	
$<\cdot(\doteq	)$	pop(a,)
$<\cdot(\doteq)	$	shift
$<\cdot	$	pop
$	$	accept.

3. Construct the operator precedence parser for the following grammar

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

1. The grammar is in correct form.
2. Computation of Leading & Trailing.

Leading(E) = {+, *, id}

Trailing(E) = {+, *, id}

3. Computation of operator precedence relation Table.

Rule 1. $\$ \lessdot +$, $\$ \lessdot *$, $\$ \lessdot id$
$+ \gtrdot \$$, $* \gtrdot \$$, $id \gtrdot \$$

Rule 3) $+ \lessdot +$ $+ \lessdot *$, $+ \lessdot id$
$*$

Rule (4) $+ \gtrdot +$ $* \gtrdot +$ $id \gtrdot +$
$+$

It is not a LR grammar

Operator precedence Table:-

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

Another approach for constructing precedence relation Table.

(1) $Q_1 \gtrdot Q_2$ if Q_1 has higher precedence than Q_2. and $Q_2 \lessdot Q_1$

ex:- $+ \lessdot *$, $* \gtrdot +$.

(2) If Q_1 and Q_2 are of equal precedence and if they are left associative then $Q_1 \gtrdot Q_2$ and $Q_2 \gtrdot Q_1$.

ex:- $+ \gtrdot -$, $- \lessdot +$

(3) id $\gtrdot$ all operator and $\$ \lessdot$ to all operators.

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