BOTTOM-UP Parser :-

* Introduction * Handle pruning
* Types of Bottom-up Parser.

(a) Introduction :-

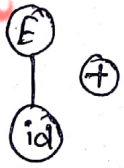
* The process of "constructing a parse tree from leaves nodes to root nodes" is called bottom-up parsing.
* In the construction ~~Parsing~~ process in each step a terminal (or) group of symbols are reduced by non terminal symbol which appears on leftside of production rule.
* It uses reduction technologies.
* It works on e.f.g
* Bottom-up Parser follows "a right most derivation in reverse order" to construct a parse tree.

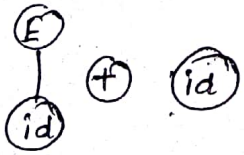
Ex:- consider a grammer $E \to E+E$, $E \to id$ to construct Bottom up Parser for the i/p string id + id + id.

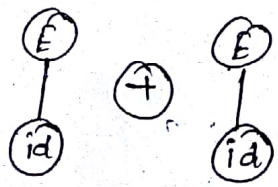
Sol:- Step1:- 

Step2: 

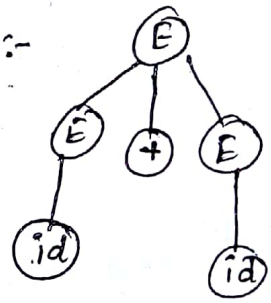
(∵ E → id)

Step3: 

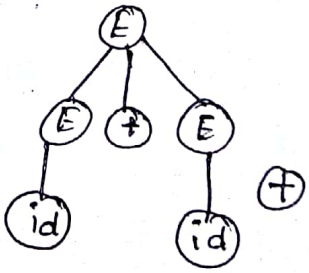
Step4: 

Step5:- 

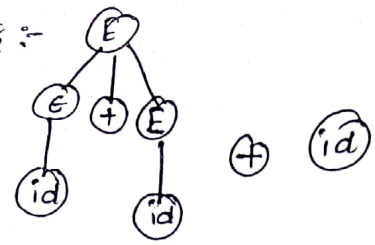
(∵ E → id)

Step6:- 

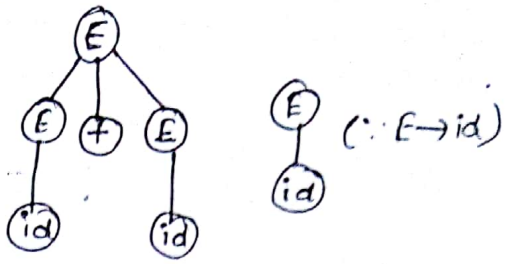
(∵ E → E+E

Step7:- 

Step8 :-

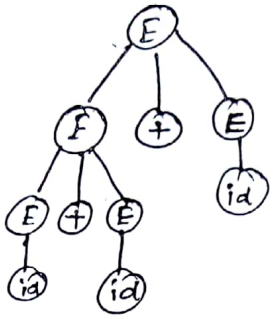


Step9 :-



(∵ E → id)

Step10 :-

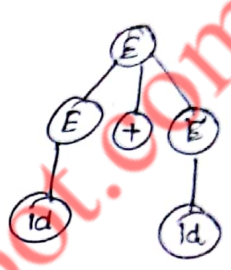


(∵ E → E+E)

RMD :-

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

Parse tree



⇒ Bottom up Parser.

∴ The reverse order of RMD is called construction of bottom up parser.

(b) Handle Pruning :-

Handle : Handle is a substring which is reduced by a non terminal on the left hand side of production rule.

Ex :- consider the grammer E → E+E, E → id and the input string is id+id+id. Find the handles.

Sol :- Given E → E+E
E → id

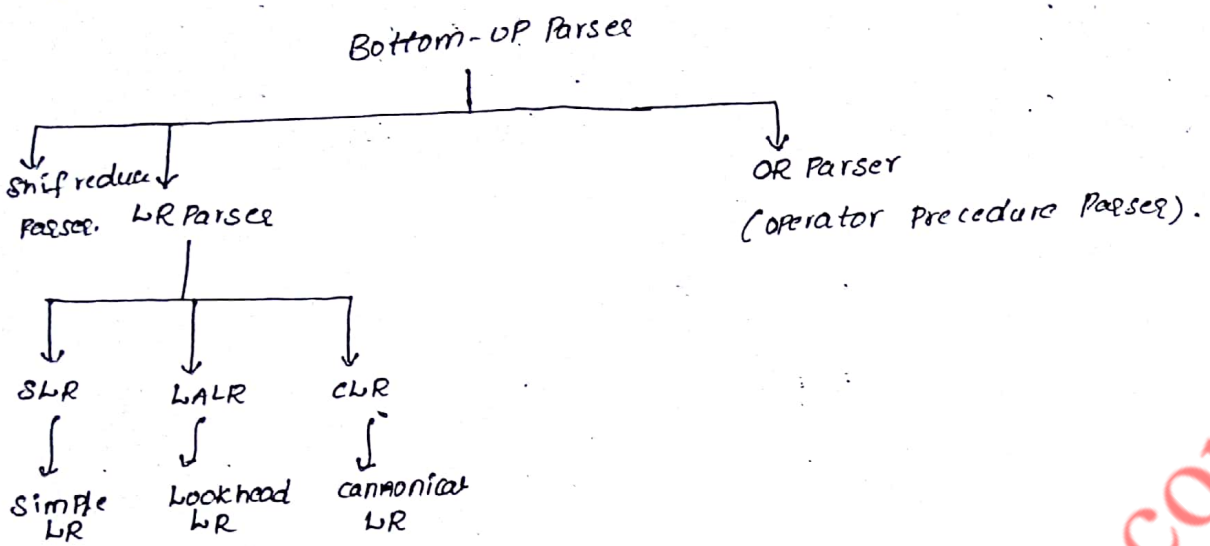
RMD :-

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

The process of detecting handles and use them in reduction is called handle pruning.

Right sentential form	Handle	Production Rule
id+id+id	id	E → id
E+id+id	id	E → id
E+E+id	id	E → id
E+E+E	E+E	E → E+E
E+E	E+E	E → E+E
E		

Types of Bottom-UP Parser :-



Why LR Parser :

* Introduction
* LR Parser Model.

Introduction :-

- LL Parser is a bottom-UP Parser.
- It constructs a Parse tree for the given i/p string from leaves ~~node~~ to root node.
- It performs right most derivation in reverse order. to construct Parse tree.
- Here LR Parser is also called as LR(K) Parser.

LR(K) = L → means read the i/p string from left to right

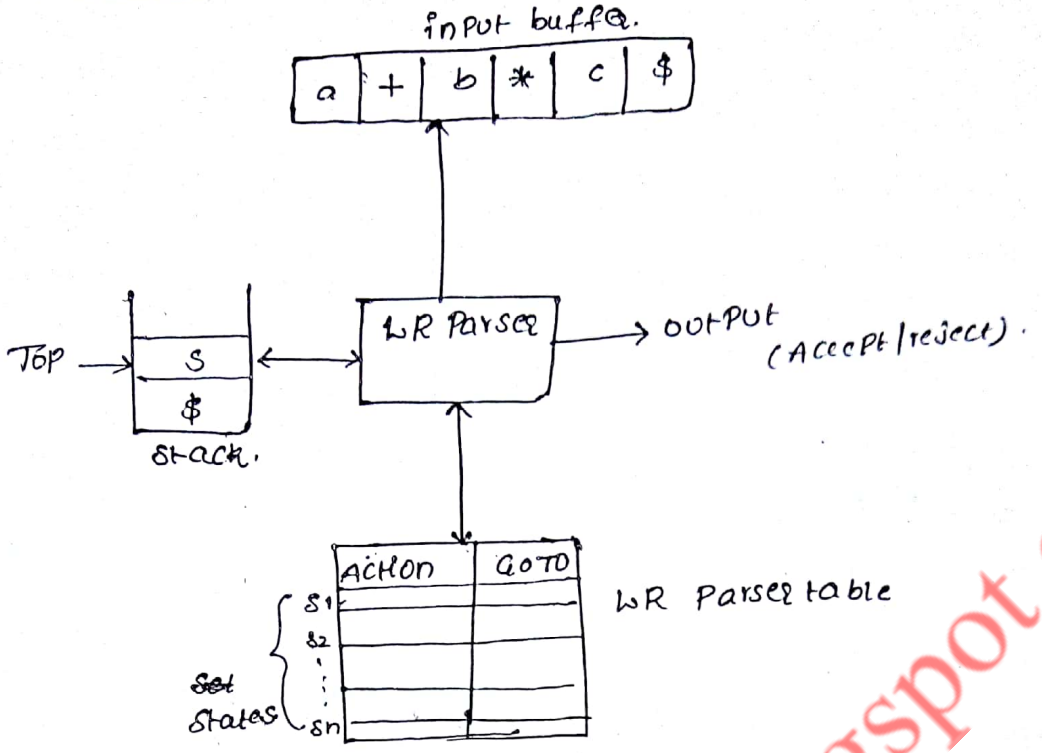
R → means it derives the given i/p string by using right most derivation in reverse order

K → means it reads K no. of i/p symbol at a time.

Property :-

1. LR(K) Parser is always super set of LL(1) Parser. that means the CFG which is accepted by LR(K) is always superclass of the CFG which is accepted by LL(1) Parser.
2. It construct a Parse tree with no back tracking.

(b) Model of LR Parser :-



- LR Parser uses three data structures like
  1. Input buffer. 2. stack 3. Parsing table.

Input buffer :- It is used to store the data of input string. and input ends with end marker. C$);

Stack :- used to perform shift and reduce operations by Reading input string from buffer.
Initially it contains ($), and (S) start symbol of grammar.

LR Parsing table :- It contains collection of rows and columns. row represent state. columns represented by two parts.
1. Action Part 2. GOTO Part.

1. Action Part contains terminal symbols including (C$) sign. and the entry in blw state and action is always one of the following operations.
   1. Shift 2. Reduce 3. Accept 4. Error.

2. GOTO contains set of non-terminal and the entries on state numbers.

Operator Precc

Shift - Reduce Parser:-

* Introduction.
* Implementation of SR parser.
* Parsing Algorithm.

Introduction:-

* Bottom UP - Parser.
* It constructs a parser tree for given i/p string from leave nodes to root node.
* The working principle is "bottom-up parsei".

Implementation of Shift reduce Parsei:-

* SR Parsei can be implemented by using the following data structure
  * Input buffer.
  * stack.
* Input buffer used to store the i/p string and it ends with "$".
* stacks :- stack is used for storing and accessing the bits and RHS of the production rule. It is initially initialized by $.
* Shift reduce Parsei perform the following basic operations.
  * Shift :- Moving of the symbols from i/p buffer on to the stack. This action is called shift.
* Reduce :- If the handle appears on the top of the stack then reduction of it by a production rule. That means RHS of rule is poped and LHS of rule is Pushed on to the stack.
* Accept :- If the stack contains $ and start symbol and i/p buffer is empty at the same time. Then that action is called accept. That is Parsing is done successfully.
* Error :- A situation in which the Parsei can't either shift or reduce the symbols, it can't even perform the accept action is called an Error.

Parsing Algorithm:-

Ex:- Consider the grammer $E \to E+E$
$E \to E*E$
$E \to id$

* This algorithm basicaly consist of 3 colums
* stack * I/P buffer or string * parsing action.

Stack	I/P buffer	Parsing action
$	id * id * id $	shift
$ id	* id * id $	Reduce $E \to id$.
$ E	* id * id $	shift
$ E +	id * id $	shift
$ E + id	* id $	Reduce $E \to id$.
$ E + E	* id $	shift
$ E + E *	id $	shift
$ E + E * id	$	Reduce by $E \to id$
$ E + E * E	$	Reduce $E \to E * E$
$ E + E	$	Reduce by $E \to E + E$
$ E	$	Accept.

*** (CU M)

Operator Precedence Parser:-

* Introduction * operator Grammar * operator Precedure Relations.
* operator Precedence Grammar. * operator Precedence table.
* operator Precedence functions.

(a) Introduction:-

* OR parser is a Bottom-up parser.
* Type of shift-Reduce parser.
* works on operator Grammar.

(b) operator Grammar:-

A Grammar is set to be operator Grammar if that satisfies the following conditions.

1. ~~KNOW~~ no Production rule should have 'ε' on its right
2. no Production rule should have two adjacent non terminals.

Ex:- 1. $E \to E+E$
$E \to E*E$
$E \to id$.

2. $S \to (L)$
$S \to a$
$L \to L,S$
$L \to S$

3. $S \to AS$
$S \to b$
$A \to SA$
$A \to a$.

not operator Grammar.

4. $S \to CC$
$C \to CC$
$C \to d$

(c) operator Precedence Relations:-

1. $t_1 < t_2$
t_1 has less Precedence then t_2.

2. $t_1 = t_2$
t_1 has the same Precedence as t_2.

3. $t_1 > t_2$.
t_1 has more Precedence than t_2.

(d) operator Precedence Grammar:-

It is a operator Grammar having disjoint Precedence relation $(<, =, >)$ b/w any pair of terminals.

(e) operator Precedence table:-

1. It is a collection of rows and columns are represented by terminals only.

The entries are precedence relations b/w rows and columns.

Ex:- consider the grammar $E \to E+E$
$E \to E*E$
$E \to id$.

construct operator precedence parser for the above Grammar and parse the input string id + id * id.

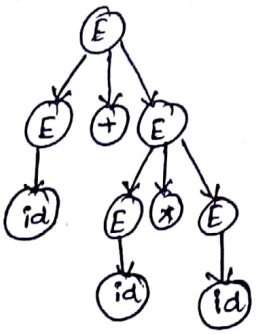
Sol:- step1:- check OPG (or) not.
2:- Operator Precedence relation table.
3. Parse the given input string.
4. Generate Parse tree.

Basic	id, a, b, c → high.
+	$ → low
*	+ > +
id	* > *
$	id ≠ id
	$ A $

	+	*	id	$
+	>	⋖	⋖	>
*	>	>	⋖	>
id	>	>		>
$	⋖	⋖	⋖	A

stack	Relation	input string	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	A	$	Accept.

Parse tree :-



Note:- In the operator precedence table we have n terminals and n operators then it requires order of n^2. memory size. To overcome this problem. why using operator precedence function table.

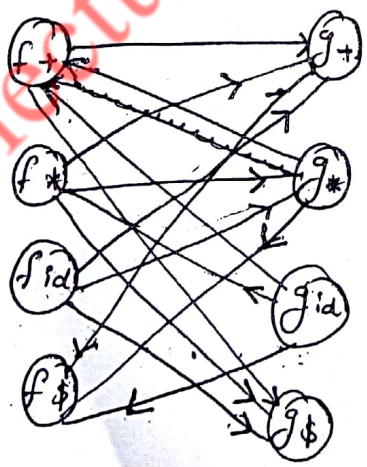
Operator Precedence functions :-

Step1:- create the functions f_a, g_a for each grammar terminal 'a' and for '$'.

Step2:- create a directed graph by adding edges in following manner.

1. If $a < b$ then add an edge from g_b to f_a
2. If $a > b$ then add an edge from f_a to g_b.

Step3:- the graph does not contains cycles



longest path :-

$fid \rightarrow g* \rightarrow f+ \rightarrow g+ \rightarrow f\$$

$gid \rightarrow f* \rightarrow g* \rightarrow f+ \rightarrow g+ \rightarrow f\$$

f The mapping table

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

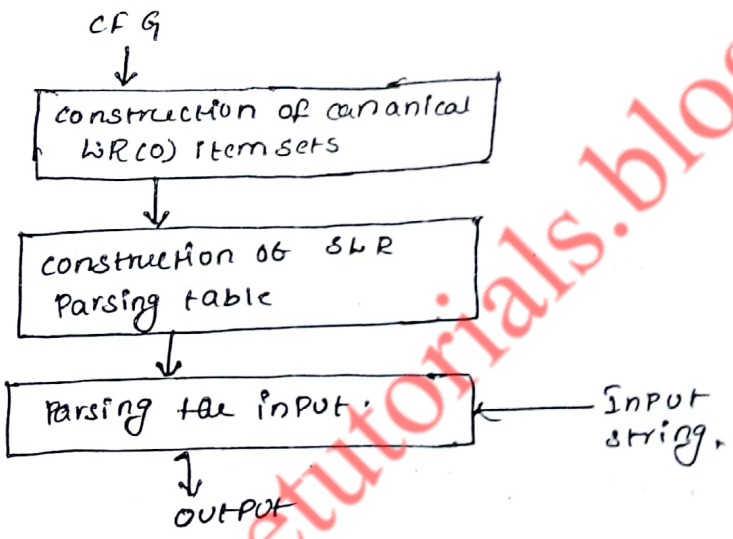
(sum)

** LR Parser:-

* Introduction * Types of LR Parser.
* Model

1, SLR Parser :-

② working of SLR Parser



1, Construction of Canonical LR(0) item sets:-

LR(0) Item :-

consider a Production rule

A → aBb . Now, Place • (dot) at some Position at the RHS of the Production rule.

A → •aBb
A → a•Bb
A → aB•b
A → aBb•

This is are called LR(0) Items of the grammar 'G'.

Argumented Grammar :-

consider the grammar 'G' and 'S' is the start Symbol. The argumented grammar of 'G' is G' with a new start Symbol S', and having a Production S' → S.

Ex:- Consider the Grammar 'G' is like $E \to E+E$, $E \to E*E$, $E \to id$. then

the argumented grammar G' is
$E' \to E$
$E \to E+E$
$E \to E*E$
$E \to id$.

LR(0) Items are computed by using the following functions.

1. CLOSURE 2. GO TO

1. CLOSURE Function:-

Let I be a set of Items in the grammar 'G'. then the CLOSURE(I) can be computed by using the following steps.

1. initially every item in 'I' is added to CLOSURE(I)
2. consider the following

$A \to \alpha X \cdot B Z$ is a Item. and

$B \to y$ is a Production then add the item

$V \to \cdot y$ to I.

Ex:- consider the argumented grammar $E' \to E$
$E \to E+E$
$E \to E*E$
$E \to id$.

CLOUSRE ($E' \to E$):

$E' \to \cdot E$
$E \to \cdot E+E$
$E \to \cdot E*E$
$E \to \cdot id$.

2. GO TO functions:-

It is used for moving the (.) symbol in LR(0) Item from one position to another position.

consider an item in 'I' as $A \to \cdot Bab$ then

GOTO(I, B) = $A \to B \cdot ab$.

** Ex:- construct LR(0) Items for the following grammar $E \to E+T$, $E \to T$, $T \to T*F$, $T \to F$, $F \to (E)$, $F \to id$.

Sol:- The given grammar is $E \to E+T$, $E \to T$, $T \to T*F$, $T \to F$, $F \to (E)$, $F \to id$.

∴ The argumented grammar is $E' \to E$, $E \to E+T$, $E \to T$, $T \to T*F$, $T \to F$, $F \to (E)$, $F \to id$.

CLOSURE $(E' \to E)$

I_0 : $E' \to \cdot E$, $E \to \cdot E+T$, $E \to \cdot T$, $T \to \cdot T*F$, $T \to \cdot F$, $F \to \cdot (E)$, $F \to \cdot id$

1 GOTO (I_0, E):

I_1 : $E' \to E \cdot$, $E \to E \cdot +T$

2 GOTO (I_0, T):

I_2 : $E \to T \cdot$, $T \to T \cdot *F$

3 GOTO (I_0, F)

I_3 : $T \to F \cdot$

4 GOTO $(I_0, ()$

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

I_4:

5 GOTO (I_0, id)

I_5: $F \to id \cdot$

6 GOTO $(I_1, +)$

I_6 : $E \to E+\cdot T$, $T \to \cdot T*F$, $F \to \cdot (E)$, $F \to \cdot id$

7 GOTO $(I_2, *)$

I_7 : $T \to T*\cdot F$, $F \to \cdot (E)$, $F \to \cdot id$

8 GOTO (I_4, E):

I_8: $F \to (E \cdot)$
$E \to E \cdot + T$

9 GOTO (I_4, T)

I_2: $E \to T \cdot$
$T \to T \cdot * F$

10 GOTO (I_4, F)

I_3: $T \to F \cdot$

11 GOTO (I_4, ()

$F \to (\cdot E)$
$E \to \cdot E + T$
I_4: $E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

12 GOTO (I_4, id)

I_5: $F \to id \cdot$

13 GOTO (I_6, ()

$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
I_4: $T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

14 GOTO (I_6, T)

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

15 GOTO (I_6, F)

I_3: $T \to F \cdot$

16 GOTO (I_6, id)

I_5: $F \to id \cdot$

17 GOTO (I_7, F)

I_{10}: $T \to T * F \cdot$

18 GOTO (I_7, ()

$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
I_4: $T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

19 GOTO (I_7, id)

I_5: $F \to id \cdot$

20 GOTO (I_8,))

I_{11}: $F \to (E) \cdot$

21 GOTO (I_8, +)

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

22 GOTO (I_9, *)

$T \to T * \cdot F$
I_7: $F \to \cdot (E)$
$F \to \cdot id$

→ Construction of SLR Parsing table:-

STEP1:- Let 'C' = $\{I_0, I_1, I_2, \ldots I_n\}$ is a collection of LR(0) item sets.

STEP2: Consider I_J an item set in 'C'.

1. If GOTO (I_j, a) = I_K then set ACTION [J, a] = shift K where 'a' is always terminal.
2. If GOTO (I_j, A) = I_K then set GOTO [J, A] = K. where 'A' is always is non-terminal.
3. If $S' \to S \cdot$ is in I_j then set ACTION [j, $] = accept
4. If $A \to X \cdot$ is in I_j then set ACTION [j, a] = reduced by $A \to X$ for all 'a' follow of [A]. If 'X' is a terminal.

Set Action[i, a] = reduce by $A \to x$ for all 'a' in Follow(X).
If 'x' is non terminal.

6. for all unfield Entries are represented by Error

State	Action						Goto		
	+	*	id	(	)	$	E	T	F
0			S5	S4			1	2	3
1	S6					Accept			
2	r2	S7 r2			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	r1			
10	r3	r3			r3	r3			
11	r5	r5			r5	r5			

Completed items:-

I_1 : $E' \to E$

I_2 : $E \to T$.

I_3 : $T \to E$

I_5 : $F \to id$

I_8 : $T \to F$

I_5 : $F \to id$

I_{10} : $T \to T * F$.

I_{11} : $F \to (E)$.

I_1 : $E' \to E \Longrightarrow a[1, \$]$: accept

I_9 : $E \to E + T$

I_9 : $E \to E + T$

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

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

FOLLOW(T) = FOLLOW(E) ∪ {*}
= {$, +, *,)}

FOLLOW(F) = FOLLOW(T)
= {$, +, *,)}

1. E → E+T
E → T
T → T*F
T → F
F → (E)
F → id

I3 : T → F
{$, +, *,)}
r4

I8 : F → id
{$, +, *,)}
r6

I10 :- T → T * F:
r3 {$, +, *,)}

I11 : F → (E)
{$, +, *,)}
r5

I2 :- E → T.
FOLLOW(T) {$, +, *,)}
r2

I9 : E → E+T
FOLLOW {$, +, *,)}
r1

- conflict occurs.

→ SLR Parsing Algorithm:-

step1:- initially push '0' as initial state on to the stack. and place input string along with '$' in to the i/p buffer.

step2:- If 's' is top of the stack and 'a' is the i/p symbol from input buffer. then

1. If action of [s, a] = shift j then push 'j' a on to the stack.
2. If action of [s, a] = reduce A → β then pop 2*|β|.
If 'i' is on the top of the stack then push 'A' then push 'GoTo'[i, A] on the top of the stack.
3. If Action of [s, a] = Accept then stop the Parsing process and announce Parsing is successful.

Stack	Input Buffer	Action
0	id + id * id $	shift (S5)
0 id 5	+ id * id $	reduced by F → id (r6)
0 F 3	+ id * id $	reduced T → F (r4)
0 T 2	+ id * id $	reduced by E → T (r2)
0 E 1	+ id * id $	shift (S6)
0 E 1 + 6	id * id $	shift (S5)
0 E 1 + 6 id 5	* id $	reduced by F → id (r6)
0 E 1 + 6 F 3	* id $	reduced by T → F (r4)
0 E 1 + 6 T 9	* id $	shift (S7)
0 E 1 + 6 T 9 * 7	id $	shift (S5)
0 E 1 + 6 T 9 * 7 id 5	$	reduced by F → id (r6)
0 E 1 + 6 T 9 * 7 F 10	$	reduced by T → T * F (r3)
0 E 1 + 6 T 9	$	reduced by E → E + T (r1)
0 E 1	$	Accept.

CLR Parser:-

* Introduction.
* construction of CLR Parser.
1. construction of ~~CLR~~ LR(1) items.
2. construction of CLR parser table.
3. construction of CLR Parsing Algorithm.

(a) Introduction:-

* Bottom-up Parser.
* most powerful Parser.
* works on LR(1) items.
* constructed by using LR(1) items.

LR(1) Items:-

General form is.

- $A \rightarrow x \cdot y, a$ where
- A is non terminal.
- x, y are Grammar Symbols.

. a is always terminal, which is caued as a lookahead symbol" that gives an extra information.

→ Construction of CLR Parser table:-

↓ CLOUSRE function:-

consider I be a set of LR(1) items. then CLOUSRE(I) can be computed by using the following steps.

step1:- Initially every item in I is added to CLOUSRE(I).

step2:- consider $A \to x \cdot By, a$

$B \to z$ are two productions and x, y, z are Grammar symbols. then add $B \to \cdot z$, FIRST(ya) to new LR(1) items.

↓ GOTO functions:-

consider an LR(1) item is in I as $A \to \cdot x\, By, a$ then GOTO of I, x will be $A \to x \cdot By, a$ and including CLOUSRE (B) productions.

Ex:- construct LR(1) items for the following Grammar.

$S \to CC$
$C \to cC$
$C \to d$

FIRST(Y,a)

SOL:- The Given Grammar is $S \to CC$
$S \to cC$
$C \to d$

∴ The argumented grammar is $S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

CLOSURE (S' → S):

$S' \to \cdot S, \$$

I_0:
$S \to \cdot CC, \$$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

↓ GOTO (I_0, S):

I_1: $S' \to S \cdot, \$$

2, GOTO (I0, C)

$S \rightarrow C \cdot C, \$$

I_2: $C \rightarrow \cdot CC, \$$

$C \rightarrow \cdot d, \$$

3, GOTO (I0, c):

$C \rightarrow C \cdot C, c|d$

I_3: $C \rightarrow \cdot CC, c|d$

$C \rightarrow \cdot d, c|d$

4, GOTO (I0, d):

I_4: $C \rightarrow d \cdot, c|d$

5, GOTO (I2, C)

I_5: $S \rightarrow CC \cdot, \$$

6, GOTO (I2, C)

$C \rightarrow C \cdot C, \$$

I_6: $C \rightarrow \cdot CC, \$$

$C \rightarrow \cdot d, \$$

7, GOTO (I2, d):

I_7: $C \rightarrow d \cdot, \$$

8, GOTO (I3, C)

I_8: $C \rightarrow CC \cdot, c|d$

9, GOTO (I3, C)

$C \rightarrow C \cdot C, c|d$

I_3 $C \rightarrow \cdot CC, c|d$

$C \rightarrow \cdot d, c|d$

10, GOTO (I3, d)

I_4: $C \rightarrow d \cdot, c|d$

11, GOTO (I6, C)

I_9: $C \rightarrow CC \cdot, \$$

12, GOTO (I6, c)

$C \rightarrow C \cdot C, \$$

I_6 $C \rightarrow \cdot CC, \$$

$C \rightarrow \cdot d, \$$

13, GOTO (I6, d):

I_7: $C \rightarrow d \cdot, \$$

Construction of CLR Parser table:-

CLR Parser table:

State	Action	GOTO

Step1: Let C = {I1, I2, I3, ... In} is a collection of LR(1) Item sets.

Step2:- consider I_i as a set in C:

1. If GOTO of $(I_i, a) = I_k$ then set Action $[i, a]$ = shift k. where a is terminal
2. If $S' \rightarrow S \cdot, \$$ is in I_i then set Action $[i, \$]$ = Accept.
3. If $A \rightarrow X \cdot, a$ is in I_i then set Action $[i, a]$ = Reduced by $A \rightarrow X$.

Step3:- If GOTO $(I_i, A) = I_k$ then set GOTO $[i, A] = k$.

Step4:- All the unfilled entries are Errors.

CLR Parser table

State	Action			Go To	
	c	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	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

GOTO $(I_0, S) = 2$, GOTO $(I_0, C) = I_2$.

completed:-

→ I_1: $S' \to S \cdot$, $\$$

I_5: $S \to CC \cdot$, $\$$

I_9: $C \to cC \cdot$ $\$$

I_4: $C \to d \cdot$, c/d.

I_7: $C \to d \cdot$, $\$$

I_8: $C \to cC \cdot$, c/d

$S \to CC$

$S \to cC$

$C \to d$

CLR Parsing Algorithm:-

Step1:- Initially push '0' as initial state on the stack. and place input string along with '$'. in to the i/p Buffer.

Step2:- If 's' is top of the stack and 'a' is the i/p symbol from input buffer then.

1. If action of [s,a] = Shift j then push 'j','a' on to the stack
2. If action of [s,A] : reduce $A \to B$ then pop 2*|B| If 'r' is on the top of the stack then push 'A' then push Goto [r,A] on the top of the stack.
3. If Action of [s,a] = Accept then stop the Parsing process and announce Parsing is successful.

Example :- Consider the input string ccdd

CLR Parsing Algorithm

Stack	input buffer	Action
0	ccdd$	shift (s_3)
0 c_3	cdd$	shift (s_3).
0 c_3c_3	dd$	shift (s_4)
0 $c_3c_3d_4$	d$	reduced by $c \to d$
0 $c_3c_3C_8$	d$	reduced by $c \to dcC$
0 c_3C_8	d$	reduced by $c \to cC$
0 C_2	d$	shift (s_7)
0 C_2d_7	$	reduced by $c \to d$
0 C_2C_5	$	reduced by $S \to cc$
0 S_1	$	Accept.

*** (14m)

LALR Parser :

* Introduction.
* Construction of LALR Parser.

(a) Introduction :-

* Bottom-up parser.
* similar to CLR and SLR Parser.
* SLR ⊆ LALR ⊆ CLR
* Merge the LR(1) items which have same LR(0) items but different lookahead symbols.
* LR(1) Parser
* LALR → Lookahead LR parser.

(b) Constructions of LALR parser :-

1. construction of LR(1) items. (Merge them if necessary)
2. construction of LALR Parsing Table.
3. construction of LALR parsing algorithm.

Ex:- Construct LALR parser table for the following grammar

$S \to CC$
$S \to cC$
$C \to d$. and check the input string ccdd. is accepted (or) not.

Sol:- Construction of LR(1) Item

$S \to CC$
$C \to cC$
$C \to d$

∴ The argumented grammar is $S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

CLOUSERE $(S' \to S)$

I_0 $S' \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c/d$
$C \to .d, c/d$

1) GOTO (I_0, S)
I_1 $S' \to S., \$$

2) GOTO (I_0, C)
I_2 $S \to C.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

3) GOTO (I_0, c):
$C \to c.C, c/d$
I_3: $C \to .cC, c/d$
$C \to .d, c/d$

4) GOTO (I_0, d)
I_4: $C \to d., c/d$

5) GOTO (I_2, C)
I_5: $S \to CC., \$$

6) GOTO (I_2, c)
I_6 $C \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

7) GOTO (I_2, d)
I_7 $C \to d., \$$

8) GOTO (I_3, C)
I_8 $C \to cC., c/d$

9) GOTO (I_3, C)
$C \to C.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$

10) GOTO (I_3, d)
I_4: $C \to d., c/d$

11) GOTO (I_6, C)
I_9: $C \to CC., \$$

12) GOTO (I_6, C)
$C \to C.C, \$$
I_6 $C \to .cC, \$$
$C \to .d, \$$

We have got I_3 and I_6 with similar LR(0) items and different lookahead symbol. then we consider this two states as same by merging them that is $I_3 + I_6 = I_{36}$

$\therefore I_{36} = C \to C.C, c/d/\$$
$C \to .cC, c/d/\$$
$C \to .d, c/d/\$$

similarly $I_4 + I_7 = I_{47}$
$\therefore I_{47} = C \to d., c/d/\$$

$I_8 + I_9 = I_{89}$
$\therefore I_{89} = C \to cC., c/d/\$$

∴ I_{47}: $C \to d\cdot, c|d|\$$.

Construction of LALR Parse table:-

State	ACTION			GOTO	
	c	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	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Completed Items:-

I_1: $S' \to S\cdot, \$$

I_5: $S \to CC\cdot, \$$

I_{47}: $C \to d\cdot, c|d|\$$

I_{89}: $C \to cC\cdot, c|d|\$$

Parsing Algorithm:-

Stack	Input buffer	Action
0	ccdd$	shift (s36)
0c36	cdd$	shift (s36)
0c36c36	dd$	shift (s47)
0c36c36d47	d$	reduced by C→d
0c36c36C89	d$	Reduced by C→cC
0c36C89	d$	reduced by C→cC
0c2	d$	shift (s47)
0c2d47	$	reduced by C→d
0C5	$	reduced by S→CC
0S1	$	Accept

Comparision of LR Parsers

SLR Parser	LALR Parser	CLR Parser.
* It is simplest in size.	* LALR & SLR have the same size	* It is largest in size
* It is easiest method based on FOLLOW function	* This method is applicable to some class other than SLR.	* This method is most powerful than SLR & LALR.
* Error Detection is not immediate in SLR.	* Error Dection is not immediate in LALR	* Error Dection is not immediate in CLR.
* It requires less time and space complexity.	* The time and space complexity is more than SLR.	* The time and space complexity is more than LALR.

Using Ambigeous Grammar:-

Dangling else ambiguity:-

(a) Ambigeous Grammar:-

A grammar which has two or more LMD or RMD for the same i/p string passed.

* Due to ambiguity there are two types of conflicts occured.
* If two entries in the parse table m[A,a] are reduction actions then "reduce-reduce" conflicts occurs.
* If one entry is shift action and another entry is reduced action in m[A,a] then "shift-reduce." conflicts occurs.

Ex: consider the grammar $S \rightarrow ises \mid is \mid a$

where ises = if expression then statement else statement

is = if expression then statement

a = all other statement

Construction of SLR Parse table

Given grammar is $S \rightarrow ises \mid is \mid a$

Argument grammar is $S' \rightarrow S$

$S \rightarrow ises$

$S \rightarrow is$

$S \rightarrow a$

CLOSURE (S'→S)

S'→·S

S →· iSeS

I0 S →·iS

S →·a

1, GOTO (I0, S):

I1: S'→S·

2, GOTO (I0, i)

S→ i·SeS

S→ i·S

I2: S→·iSeS

S→·iS

S→·a

3, GOTO (I0,a)

I3: S→a·

4, GOTO (I0, S):

I4: S→iS·eS

S→iS·

5, GOTO (I0,i)

S→i·SeS

S→i·S

I2: S→·iSeS

S→·iS

S→·a

6, GOTO (I2,a)

I3: S→a·

7, GOTO (I4,e)

S→iSe·S

I5: S→·iSeS

S→·iS

S→·a

8, GOTO (I5,S)

I6: S→iSeS·

9, GOTO (I5,i)

S→i·SeS

I2 S→i·S

S→·iSeS

S→·iS

S→·a

10, GOTO (I5,a)

I3: S→a·

Completed Items

I1: S'→S· Jaction[1,$]=Acc

I3: S→a· → Follow(S)={$,e}

R3

I4: S→iS·→ {$,e}

R2

I6: S→iSeS→ {$,e}

R1

1. S→iSeS
2. S→iS
3. S→a

SLR Parser table:-

State	ACTION				GOTO
	i	e	a	$	S
0	S2		S3		1
1				Acc	
2	S2		S3		4
3		r3		r3	
4		S5 / r2		r2	
5	S2		S3		6
6		r1		r1	

1. This means choice of R2 in Action of [4, e] is not valid. Hence we will try it by creating the shift Action [3e].

Parsing algorithm:-

Consider the I/P string iiaea

Stack	Input buffer	Action
0	iiaea$	Shift(S2)
0i2	iaea$	Shift(S2)
0i2i2	aea$	Shift(S3)
0i2i2a3	ea$	reduced by S→a
0i2i2S	ea$	reduced by S→iS
0i2S4	ea$	reduced by S→iS
0S1	ea$	Error

Stack	Input buffer	Action
0	iieaea$	Shift (S2)
0i2	ieaea$	Shift (S2)
0i2i2	eaea$	Shift (S3)
0i2i2a3	aea$	reduced by S→a
0i2i2S4	ea$	Shift (S6)
0i2i2S4e5	a$	~~reduce~~ Shift (S3)
0i2i2S4e5a3	$	Reduced by S→a
0i2i2S4e5S6	$	Reduced by S→ iSeS
0i2S4	$	reduced by S→ iS
0S1	$	Accept.

∴ The give the grammar for shift operation.

Error Recovery in LR Parser:-

Generally the unfilled entries in the parser table. gives Errors. The process of detecting and reducing number of errors in the parser table is caled ERR recovery.

LR Parser uses Phase level ERR recovery techniques to reduce no. of errors.

Ex:- consider the grammar
E → E+E
E → E*E
E → (E)
E → id.

The Argumented grammar is
E' → E
E → E+E
E → E*E
E → (E)
E → id

~~Consider~~
CLOUSRE (E' → E)

E' → .E
E → .E+E
E → .E*E

I0 $E \to \cdot(E)$
$E \to \cdot id$.

GOTO (I0, E):

~~I1 $E \to E \cdot$~~

1, GOTO (I0, E)

$E \to E \cdot + E$

I1 $E \to E \cdot + E$

$E \to E \cdot * E$

2, GOTO (I0, ( )

$E \to (\cdot E)$

$E \to \cdot E + E$

I2 $E \to \cdot E \cdot * E$

$E \to \cdot (E)$

$E \to \cdot id$

3, GOTO (I0, id)

I3: $E \to id \cdot$

4, GOTO (I1, +):

$E \to E + \cdot E$

$E \to \cdot E + E$

I4 $E \to \cdot E * E$

$E \to \cdot (E)$

$E \to \cdot id$

5, GOTO (I4, *):

$E \to E * \cdot E$

I5 $E \to \cdot E + E$

$E \to \cdot E * E$

$E \to \cdot (E)$

$E \to \cdot id$

6, GOTO (I2, +)

$E \to E + \cdot E$

$E \to \cdot E + E$

I4: $E \to \cdot E * E$

$E \to \cdot (E)$

$E \to \cdot id$.

7, GOTO (I

SLR Parser table:-

State	ACTION						GOTO
	+	*	(	)	id	$	E
0	E_1	E_1	S_2	E_2	S_3	E_1	1
1	S_4	S_5	E_3	E_2	E_3	Acc	
2	E_1	E_1	S_2	E_2	S_3	E_1	6
3	r_4	r_4	E_3	r_4	E_3	r_4	
4	E_1	E_1	S_2	E_2	S_3	E_1	7
5	E_1	E_1	S_2	E_2	S_3	E_1	8
6	S_4	S_5	E_3	S_9	E_3	E_1	
7	r_1/S_4	r_1/S_5	E_3	r_1	E_3	r_1	
8	r_2/S_4	r_2/S_5	E_3	r_2	E_3	r_2	
9	r_3	r_3	E_3	r_3	E_3	r_3	

completed

$E' \rightarrow E$ ↳ action['$']: Accept.

I_3: $E \rightarrow id.$

I_7: $E \rightarrow E+E.$

I_8: $E \rightarrow E*E.$

I_9: $E \rightarrow (E).$

$A \rightarrow X.$ (non-terminal)

↓ FOLLOW(X)

terminal

FOLLOW(A).

FOLLOW(E) = {$, +, *,)}

I_0: $E' \rightarrow .E$

stack	input	Error
$...	+	Missing operand
$...	*	Missing operand
$...	)	~~Missing operand~~ unbalanced Right Parenthesis.
$	$	Missing operand

I_1: $E' \rightarrow E.$

stack	input	Error.
$..... id	(	Missing operator.
$..... id	)	unbalanced Right Parenthesis.
$..... id	$ id	Missing operator.

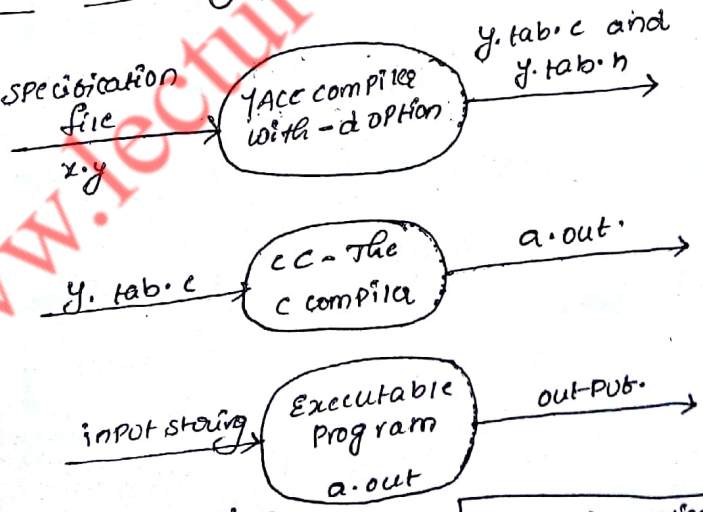
*** (4M)
# Automatic Parser Generator (YACC)

* Introduction
* YACC: Parser Generation Model.
* YACC: Parser Specification.
* Example.

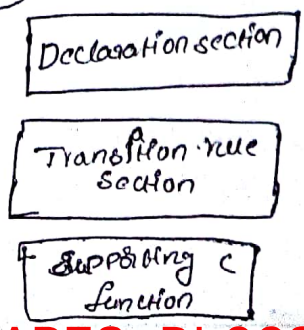
(a) Introduction :-

- The manual method of construction of LR Parser. Involves a lot of work. for the Parsing the given input string.
- There is a need for automation on this Process in order to archeive the Efficience in Parsing the input string.
- YACC is an automatic tool for generating the Parser Programm.
- YACC standands for "Yet Another compiler compiler".
- It works on UNIX UTility.
- YACC is a LALR Parser generator.
- YACC can report conflicts (or) ambiguities in the form of Error messages.
  (two or more operations are existings.)

(b) YACC : Parser generator Model :-



(c) YACC specification :-



General form is

```
%{
    Declaration section.
%}
```

→ Translation rule section :-

```
rule1      {ACTION1}
rule2      {ACTION2}
  :
  :
rulen      {ACTIONn}
```

LHS → alternative1

⇓

LHS : {alternative1 ACTION}

LHS → alternative1 | alternative2 | ..... | alternation.

```
LHS → alternative1 |
      alternative2 |   { ACTION rule }
        :
      alternaten |
```

→ Procedure section

→ Example :-

Consider the following grammar $E \rightarrow E+T \mid T$ ~~$T*T$~~

$T \rightarrow T*F \mid F$

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

design LALR bottom up Parser to compute the input Expression 2+3.

```
SOL:- %{
|
    %}
    %union
    {
      double dval;
    }
    %token <dVal> DIGIT
    %token <dval> EXPR
    %token <dval> term
    %token <d,val> factor
    %%
```

```
line : expr '\n'      {printf("\n%d", $1);}

expr: expr '+' term   {$$ = $1 + $3;}
    | term
    ;
term: term '*' factor {$$ = $1 * $3;}
    | factor
    ;
factor : '(' expr ')'  {$$ = $2;}
    | DIGIT
    ;
%%

int main()
{
  yyparse();
}
yyerror(char *s)
{
  printf("%s", s);
}
```

Output:-

```
$ vi x.y
$ yacc -d x.y
$ cc y.tab.c -lm
$ ./a.out
      2+3
    5.000000
```

Symantic Anaysis:- ~~Gives~~

A symantic anaysis gives a specific meaning of the program.

The need of symantic anaysis is

1. It build a symbol table to keep the information about tokens.
2. It determines 1. Name of the variable. 2. DataType of the variable. 3. Memory address 4. Scope of Variable 5. life time of Variable 6. Type conversions. 7. Type checking.

***(7m)

Syntax - Directed Definition:-

* Introduction.
* Synthesised Attribute.
* Inherited Attribute.
* Annotated Parse tree.

→ Introduction:-

* Syntax directed definition is a cfg with Symantic rules associated with every production of grammar.
* The symantic rules defines the attribute values of grammar symbols.

1. Attributes → They are associated with grammar symbols.

Attributes can be 1. number 2. memory address 3. data type 4. string & 5. scope.

Ex:- Let 'X' be a grammar symbol and 'b' is one of it's attribute value. then X.b denotes the value of 'b' at node of 'X' in the Parse tree.

→ Symantic rule:-

They are associated with grammar productions.

Ex:- $E \to E+T$ is a grammar production then the corresponding symantic rule is $E.val = E.val + T.val$.

Attributes are classified into two types

1. synthesised attribute (SA)
2. Inherited Attribute (IA).

→ Synthesised Attribute (SA):-

An Attribute is synthesised if it's value can be computed from the value of the it's children.

Ex:- consider the CFG production rule $A \to X$.

Here A.val = X.value.

A.val is a synthesised attribute. because it's value depending on the children attribute value; X.val

. If $X \rightarrow x_1, x_2, \ldots x_n$ is a production. If the computing rule of X attribute is of the form.

$A(x) = f(A(x_1), A(x_2), \ldots A(x_n))$ is called synthesized attribute.

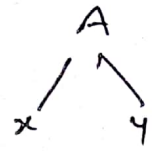
. synthesized attributes are values that can be computed by using bottom up Parsing technique.

→ Inherited Attribute (&A):-

An Attribute is Inherited if its value can be computed from the values of it's Parent (or) sibings.

Ex:- consider the CFG production rule $A \rightarrow xy$ then

$x.in = A.val$ (or) $x.in = y.in$



$y.in = A.val$ (or) $y.in = x.in$.

. If $X \rightarrow x_1 x_2 \ldots x_n$ is a production. If the computing value of "x_j" is of the form $A(x_j) = f(x_1, A(x_2) A(x_3) \ldots A(x_j))$ (or) $A(x)$ is called Inherited attribute.

. Inherited attributes are values that can be computed by using top-down Parsing technique (or) Pre order traversal.

Annoted Parse tree :-

A Parse tree that shows the values of its attributes at each node is called an annoted Parse tree (or) decorated Parse tree.

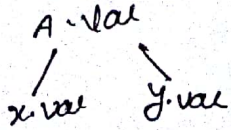
Ex:- consider the grammar CFG with the Production rule $A \rightarrow xy$.

The corresponding Parse tree is

Construct syntax directed definition for the following grammar by using synthesized attributes and construct an annoted Parse tree for the input string $5*6+7)\$$

∴ The decorated Parse tree (or) Annoted Parse tree. for the above Parse tree. is

Annotated Parse tree.



Construct syntax directed definition for the following grammar by using synthesized Attributes and construct an Annoated ~~tree~~ Parse tree for the input string 5*6+7,

$L \to En$
$E \to E+T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to digit.$

→ Syntax-Directed Definition :-

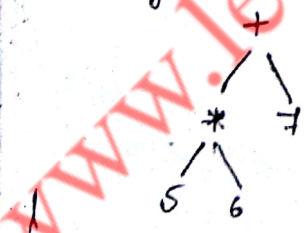
Production rule	Semantic rule
$L \to En$	print (E.val)
$E \to E+T$	E.val = E.val + T.val
$E \to T$	E.val = T.val
$T \to T*F$	T.val = T.val * F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val = E.val
$F \to digit.$	F.val = digit.lex val

Annotated Parse tree :-

The given I/P string is

5*6+7

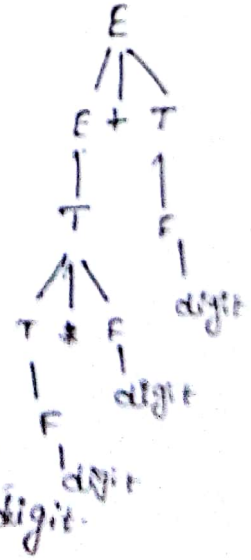
(a) syntax tree



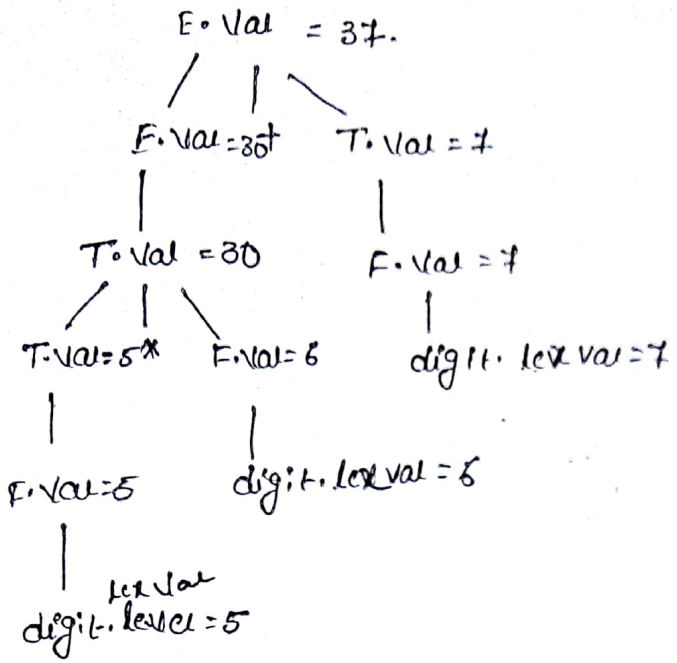
Parse tree

RmD :

$E \to E+T$
$\to E+F$
$\to E+digit.$
$\to T+digit$
$\to T*F+digit$
$\to T*digit+digit$
$\to F*digit+digit$
$\to digit*digit+digit.$



(5) Annotated Parse tree :-



(6). Construct syntax directed definition for the following grammar by using inherited attributes

$D \rightarrow TL;$

$T \rightarrow$ int / float / char / double.

$L \rightarrow L_1$, id.

$L \rightarrow$ id.

and construct Annotated Parse tree for the i/p string

int a, b, c;

Sol:- The given grammar is $D \rightarrow TL;$

$T \rightarrow$ int / char / float / double

$L \rightarrow L_1$, id

$L \rightarrow$ id

Syntax-Directed definition:-

Production rule	Semantic rule.
$D \rightarrow TL;$	L.in = T.type.
	t.type = D.val
$T \rightarrow$ int	T.Type = Integer.
$T \rightarrow$ char	T.Type = char
$T \rightarrow$ float	T.type = float
$T \rightarrow$ double	T.Type = double
$L \rightarrow L_1$, id	L_1.in = lookup (id.entry) or L_1.in = L.in (or) id.entry = L.in & L_1.in
id. ~~entry~~ $L \rightarrow$ id	id.entry := L.in

Annotated tree:-

The given input string.

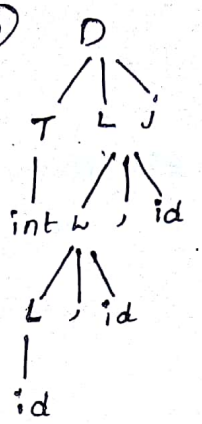
int a, b, c;

(a) Parse tree:-

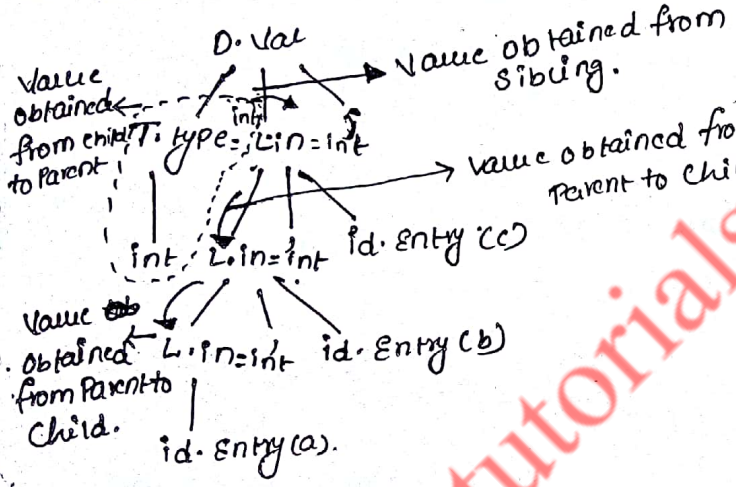
LMD:-

D → TL;
→ int L;
→ int L, id;
→ int L, id, id;
→ int id, id, id;

(b) Parse tree:-



(c) Annotated Parse tree:-



→ Evaluation Order for Syntax directed definition:-

1. Dependency graph
2. S - attributed definition
3. L - attributed definition
4. Syntax directed translation (SDT).

→ Dependency graph:-

It is a directed graph that represents the inter dependencies b/w the synthesized attributes and inherited attributes at nodes in the Parse tree. is caued "dependency graph".

Ex:- for the rule $x \rightarrow y$ the symantic rule is given by $x.val = f(y.val, z.val)$. then the synthesized attribute $x.val$ is depends on the attributes $y.val$ and $z.val$.

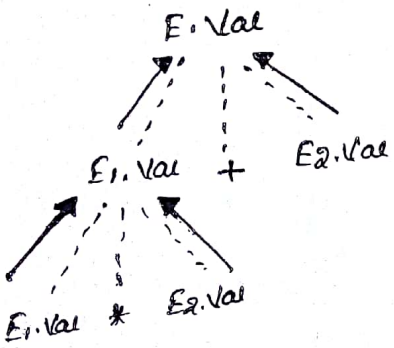
Ex:- consider the grammar $E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$ construct dependency graph for the above grammar.

Sol:- syntax directed Definition by using synthesized attributes:-

Production rule	Symantic rule
$E \rightarrow E_1 + E_2$	$E.Val = E_1.Val + E_2.Val$
$E \rightarrow E_1 * E_2$	$E.Val = E_1.Val * E_2.Val$

Dependency graph:-



1). S-attributed Definition:-

The syntax directed definition that uses only the synthesised attributes, is called as S-attributed definition.

Completion of S-attributed definition:-

Step1:- write the syntax directed definition for the given grammar.

Step2:- generate the annotated parse tree. Here the completion is done in bottomup monner.

Step3:- the value obtained at the root node is the final o/p.

Ex:- consider the following grammar $L \rightarrow En$; $E \rightarrow E+T$, $E \rightarrow T$, $T \rightarrow T*F$, $T \rightarrow F$, $F \rightarrow (E)$, $F \rightarrow$ digit.

construct S-attributed definition for the above Grammar with the help of i/p string 5 * 6 + 7.

3) L-attributed definition:-

The syntax directed definition that uses only the inherited attributes is called L-attributed definition.

Completion of L-ADG:-

Step1:- Write the syntax directed definition for the given grammar.

Step2:- Generate the annotated parse tree where the completion is done in top-down manner.

3:- The value obtained at the leaves is the final o/p.

Ex:- Consider the grammar D → TL; T → int | char | float | double

L → L1, id, L → id.

Construct L-attributed definition for the above grammar with the help of i/p string into an a, b, c;

Syntax Directed Translation (SDT)

During the process of parsing the evaluation of attributes takes place by putting semantic actions in blco ({}) at the right of the Grammar symbol is called SDT.

Ex:-

SDD	Production rule	Semantic rule
	$T \rightarrow T * F$	$T.val = T.val * F.val$
	$T \rightarrow F$	$T.val = F.val$
	$F \rightarrow digit$	$F.val = digit.lexval$
SDT	$T \rightarrow T * F$	$T.val = \{T.val * F.val\}$
	$T \rightarrow F$	$T.val = \{F.val\}$
	$F \rightarrow digit$	$F.val = \{digit.lexval\}$

SDT's are implemented during parsing without building a parse tree.

1. S-attributed SDD is based on LR-Parser table grammar.
2. L-attributed SDD is based on LL-Parser table grammar.

Applications of SDT:-

1. construction of syntax tree.
2. The structure of type.

1. construction of syntax tree:-

Syntax tree:-

A bstract representation of SDT is caued ST.

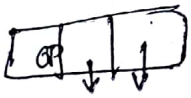
2. Syntax trees are used to write SDT by using SDD.

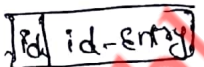
* Construction of a syntax tree for an expression means translation of an expression into postfix

3. the nodes for Each operator and operand is created.

4. Each node can be implemented as a record with multiple fields.

Fields.

1. mknode (op, left, right) ⇒ 

2. mkleaf (id, id-entry) ⇒ 

3 mkleaf (num, value) ⇒ 

Ex:- Construct the syntax tree for the expression

x * y - 5 + z.

Sol:- The given expression is x+y-5+z

then the post fix form of the given expression is

= x * y - 5 + z.

= xy * - 5 + z

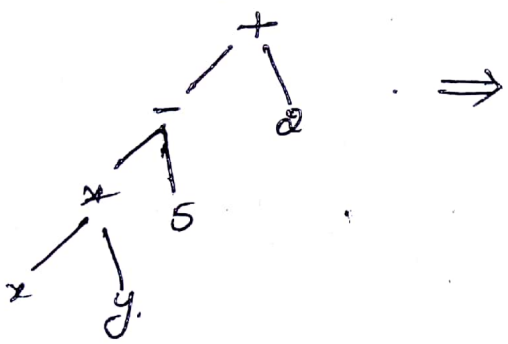
= xy * 5 - + z

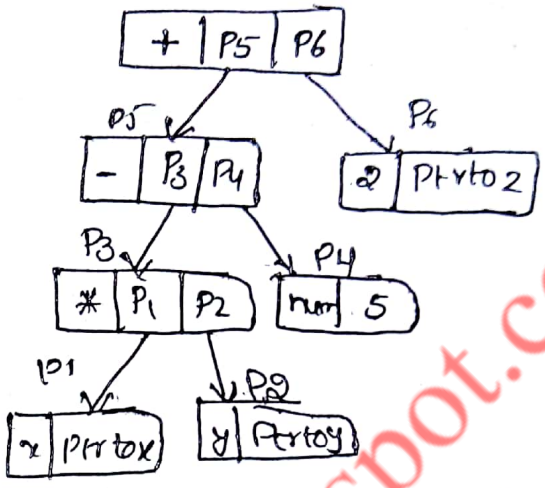
= xy * 5 - z +

Symbol	Operation
x	P_1 := mkleaf (x, ptr to x)
y	P_2 := mkleaf (y, ptr to y)
*	P_3 = Mknode ('*', P_1, P_2)
5	P_4 = Mkleaf (num, 5)
-	P_5 = Mknode ('-', P_3, P_4)
z	P_6 = Mkleaf (z, ptr to z)
+	P_7 = mknode ('+', P_5, P_6)

syntax tree

x * y - 5 + 2





Syntax Directed Translation schemes:- (SDT)

1. Post fix Translation schemes.
2. Parser - stack implementation of post fix SDT's.
3. eliminating left recursion from SDT's.

(a) Introduction

→ deals with the discussion of implementation of SDT during parsing without construction of parse tree.

- SDT is S-attributed then the grammar is LR-Parser.
- SDT is L-attributed then that grammar is LL-Parser

(b) Post fix Translation schemes:-

→ ~~P~~ Post fix SDT schemas are SDT with all actions at the right ends ~~the~~ of the production body.

Ex:-

Production rule	semantic rule
$L \to En$	{print (E.val)}
$E \to E+T$	{E.val = E.val + T.val}
$E \to T$	{E.val = T.val}
$E \to T * F$	{E.val = T.val * F.val}
$T \to F$	{T.val = F.val}
$F \to (E)$	{F.val = E.val}
$F \to digit$	{F.val = digit.lexval}.

2. Parser - stack implementation of Post fix SDT's.

Post fix SDT are implemented by considering the attributes of each grammar on the stack in a place where the grammar symbol is found during reduction ~~Ex~~.

Ex:- consider the grammar Production $A \to xyz$. then SDD is $A.val = f(x.val, y.val, z.val)$.

parser stack with Synthesized attributes.

	state Grammar Symbol	Synthesized attributes
	⋮	
	X	X.val
	Y	Y.val
TOP→	Z	Z.val.
	⋮	
	⋮	

Here we place all attributes in R.H.S of production into parser stack. after applying reduction technique X.val, Y.val, Z.val are replaced by A.val.

3, Eliminating left Recursion from SDT's

consider a syntax directed translation with left recursion $E \rightarrow E+T$ {print("T")}

$E \rightarrow T$. where elimination left recursion from SDT?

Left recursion
$A \rightarrow A\alpha / B$
How to elimination of left recursion
1. $A \rightarrow BA'$
2. $A' \rightarrow \alpha A'$
3. $A' \rightarrow \epsilon$

where
$A = E$
$\alpha = +T$ {print("T")}
$B = T$.

Sol:- $E \rightarrow TE'$

$E' \rightarrow +T$ {print("T")} E'

$E' \rightarrow \epsilon$ without left recursion.