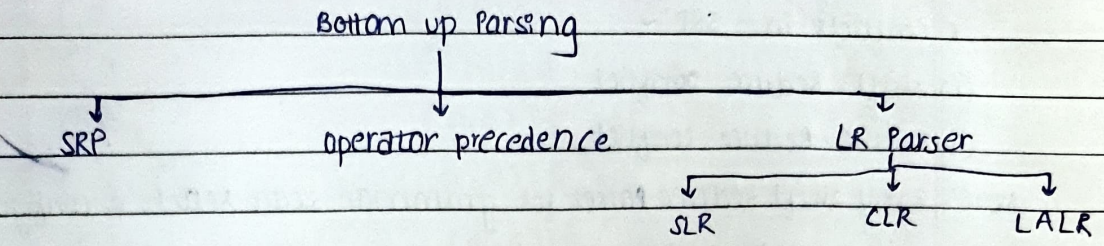
Sessional-2
10/01/24

Page
Date



1) SRP (Shift Reduce Parser)

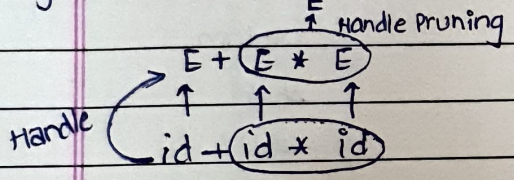
→ Shift - the next input symbol is shifted onto the top of the stack.

→ Reduce - The parser replaces the handle within a stack with a NT.

→ Accept - the parser announces successful completion of parsing.

→ Error - The parser discovers that a syntax error has occurred & calls an error recovery routine.

Eg $E \to E+E / E*E / (E) / id$



Eg Apply SRP:- $E \to E+E / E*E / (E) / id - E$

id + id * id

Ans

	Stack	Input	Action
	$	id+id * id$	shift id
	$id	+id * id $	Reduce E→id
	$E	+id * id $	shift +
	$E+	id * id $	shift id
	$E+id	* id $	Reduce E→id
Reduce-shift conflict problem →	$E+E	* id $	shift *
	$E+E*	id $	shift id
	$E+E*id	$	Reduce E→id
	$E+E*E	$	Reduce E→E*E
	$E+E	$	Reduce E→E+E
	$E	$	Accept

* Conflicts in SRP:-

(i) shift - Reduce conflict

(ii) Reduce - Reduce conflict

NOTE:- Every shift Reduce Parser for grammar can reach a configuration in which the parser knowing the entire stack contents and next ip symbol, cannot decide whether to shift or to reduce then S-R conflict occurs or if parser cannot decide which of several reductions to make then this conflict is called as R-R conflict.

Eg:- E→2E2
E→3E3
E→4

string 32423

stack	Input	Action
$	32423$	shift 3
$3	2423$	shift 2
$32	423$	shift 4
$324	23$	Reduce E→4
$32E	23$	shift 2
$32E2	3$	Reduce E→2E2
$3E	3$	shift 3
$3E3	$	Reduce E→3E3
$E	$	Accepted

Eg. S→(L)/a string: (a,(a,a))
L→L,S/S

stack	Input	Action
$	(a,(a,a))$	shift (
$(	a,(a,a))$	shift a
S-R conflict $(a	,(a,a))$	Reduce S→a
$(S	,(a,a))$	Reduce L→S
$(L	,(a,a))$	shift)

$(S	,(a,a))$	Reduce S→a
$(a	,(a,a))$	shift ,
$(a,	(a,a))$	shift (
$(a,(	a,a))$	
$(L,	(a,a))$	shift (
$(L,(	a,a))$	shift a
$(L,(a	,a))$	Reduce S→a
$(L,(S	,a))$	Reduce S→L
$(L,(L	,a))$	shift ,
$(L,(L,	a))$	shift a
$(L,(L,a	))$	Reduce S→a
$(L,(L,S	))$	Reduce L→L,S
$(L,(L	))$	shift)
$(L,(L)	)$	Reduce S→(L)
$(L,S	)$	Reduce L→L,S
$(L	)$	shift)
$(L)	$	Reduce S→(L)
$S	$	Accepted

Eg S → T L
T → int | float
L → L, id | id

NOTE
id - highest priority
$ - lowest priority

string: int id, id

Stack	Input	Action
$	int id, id $	shift int
$ int	id, id $	Reduce T → int
$ T	id, id $	shift id
$ T id	, id $	Reduce L → id
$ T L	, id $	Reduce S → TL / shift ,
$ T L ,	id $	shift id
$ T L , id	$	Reduce L → L, id
$ T L	$	Reduce S → TL
$ S	$	Accept

11.01.24

⇒ Grammar G is said to be operator precedence if it has following properties
i) No prodⁿ on the RHS is null.
ii) There should not be any prodⁿ rule possessing 2 adjacent non-terminals at the RHS. E → EAB (Both are NT)

* Steps:-
i) Precedence Relation table
ii) Converted string
iii) Stack implementation
iv) Precedence function graph
v) Precedence function table

$\theta_1 \doteq \theta_2$ (θ_1 has highest priority)
$\theta_1 \lessdot \theta_2$ (θ_2 has highest priority)
$\theta_1 \doteq \theta_2$ (θ_1 & θ_2 has same priority)

carot ↑ - power
^ - right associative

⇒ Rules:-
i) ↑ is of highest precedence and right associative
ii) * and / are of next highest precedence and left associative
iii) + and - are of lowest precedence and left associative.
iv) (and) so use (≐)
v) + and + so use + ⋗ + bcoz + has left associative
vi) ↑ and ↑ so use ↑ ⋖ ↑ bcoz ↑ has right associative.

Eg $ ⋖ (, (⋖ id , (⋖ (,) ⋗) ,) ⋖ id ,) ⋗ $

f is in stack, g is in string

stack (low) high
string (high) low

f \ g	+	-	*	/	↑	id	(	)	$
+	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
-	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
/	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
↑	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
id	⋗	⋗	⋗	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗	⋗	⋗	⋗			⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖		

Eg E → EAE/(E)/-E/id
string id + id * id
Ans Step1:- check prodⁿ rule
Here E → EAE (2 adjacent NT so change)

Step2:-
$ ⋖ id ⋗ + ⋖ id ⋗ * ⋖ id ⋗ $

Given in Ques A → +/-/*//↑

Now,
E → E+E/E-E/E*E/E/E/E↑E/(E)/-E/id

operator precedence relation table

f/g	+	*	id	$
+	⋗	⋖	⋖	⋗
*	⋗	⋗	⋖	⋗
id	⋗	⋗		⋗
$	⋖	⋖	⋖	

Advantage
→ Simple

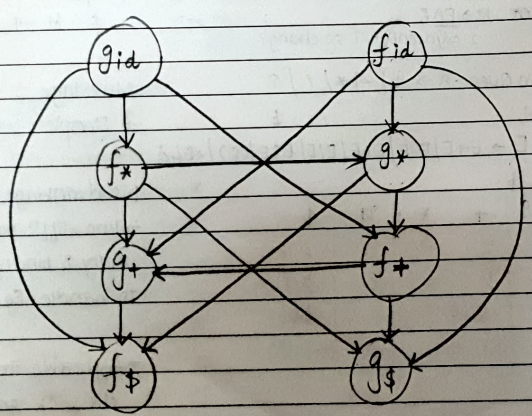
Disadvantage
→ Two diffⁿ precedence like unary & binary. so it is hard to handle. Eg +, + , -, -

→ Applicable to only small class of grammar.

step (iii) Operator precedence table/stack implementation:-

Stack	input	Action
$	id+id*id$	$ <· id, shift id
$id	+id*id$	id ·> +, reduce E→id
$E	+id*id$	$ <· +, shift +
$E+	id*id$	+ <· id, shift id
$E+id	*id$	id ·> *, reduce E→id
$E+E	*id$	+ <· *, shift *
$E+E*	id$	* <· id, shift id
$E+E*id	$	id ·> $, reduce E→id
$E+E*E	$	* ·> $, reduce E→E*E
$E+E	$	+ ·> $, reduce E→E+E
$E	$	accept

step iv) Precedence function graph



step (v) precedence table

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

Eg
E → E+T/T
T → T*V/V
V → a/b/c/d
string a+b*c*d

Ans (i)

f/g	+	*	a	b	c	d	$
+	·>	<·	<·	<·	<·	<·	·>
*	·>	·>	<·	<·	<·	<·	·>
a	·>	·>					·>
b	·>	·>					·>
c	·>	·>					·>
d	·>	·>					·>
$	<·	<·	<·	<·	<·	<·	

(ii) $ <· a ·> + <· b ·> * <· c ·> * <· d ·> $

·>, reduce
<·, shift

(iii)

Stack	input	Action
$	a+b*c*d$	$ <· a, shift a
$a	+b*c*d$	a ·> +, reduce V→a
$V	+b*c*d$	$ <· +, shift +
$V+	b*c*d$	+ <· b, shift b
$V+b	*c*d$	b ·> *, reduce V→b
$V+V	*c*d$	+ <· *, shift *
$V+V*	c*d$	* <· c, shift c
$V+V*c	*d$	c ·> *, reduce V→c
$V+V*V	*d$	* ·> *, reduce V→T
$V+V*T	*d$	* ·> *, reduce E→T
$V+V*E	*d$	* ·> *

fail

$\$ <\cdot$, $< + - < * / ,$ ↑ $<$

id id → NO entry
$ $ → accept

Eg S → (L) / a
L → L, S / S
string (a, a)

Ans (i)

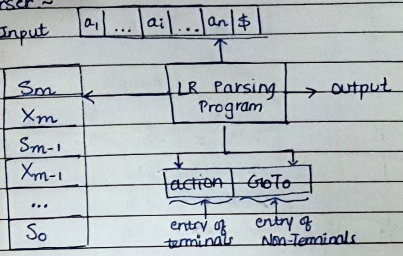
f | g	(	a	,	)	$
(	<·	<·	·>	≐	·>
a	·>		·>	·>	·>
,	<·	<·	·>	<·	·>
)	·>	<·	·>	·>	·>
$	<·	<·	<·	<·	

(ii) $ <· (<· a ·> , <· a ·>) ·> $

(iii)

Stack	Input	Action

* Model of LR Parser :-



LR(K)
L for left to right scanning of i/p
R for constructing right most derivation in reverse
K for the no. of i/p symbols, of lookahead

Steps of LR(K) :-
i) Augment the grammar & number it.
ii) Draw DFA diagram.
iii) Parsing Table.
iv) Stack Implementation.
v) Parse Tree.

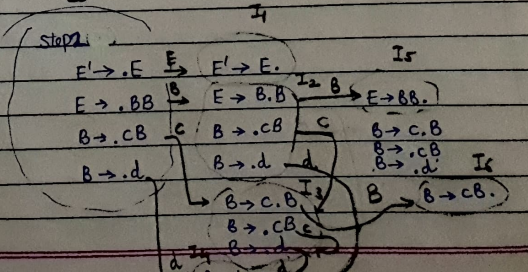
Eg E → BB
B → cB / d
string ccdd

. के बाद NT है तो as it is NT के सारे Prod^n लिख दो।
. के बाद T है तो do nothing

Ans

	Step1(i)
0.	E' → E
1.	E → BB
2.	B → cB
3.	B → d

Step2



Step3

state	Action c	Action d	Action $	GoTo B	GoTo E
I_0	S_3	S_4		2	1
I_1			Accept		
I_2	S_3	S_4		5	
I_3	S_3	S_4		6	
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Step4

stack	input	action
$0	ccdd$	shift $C \to S_3$
$0c3	cdd$	shift $C \to S_3$
$0C3C3	dd$	shift $d \to S_4$
$0C3C3d4	d$	reduce $B \to d$
$0C3C3B6	d$	reduce $B \to CB$
$0B2	d$	reduce $B \to CB$
$0B2d4	$	reduce $B \to d$
$0B2B5	$	reduce $E \to BB$
$0E1	$	Accept

Step5

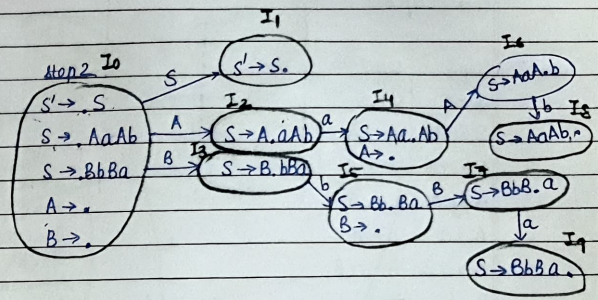
Eg $S \to AaAb / BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

Ans

	Step1
0	$S' \to S$
1	$S \to AaAb$
2	$S \to BbBa$
3	$A \to \varepsilon$
4	$B \to \varepsilon$



Step3

state	Action a	Action b	Action $	GoTo S	GoTo A	GoTo B
I_0	r_3/r_4	r_3/r_4	r_3/r_4 Accept	1	2	3
I_1						
I_2	S_4					
I_3		S_5				6
I_4						7
I_5						
I_6		S_8				
I_7	S_9					
I_8	r_1	r_1	r_1			
I_9	r_2	r_2	r_2			

SR (Shift Reduce) or RR (Reduce Reduce)

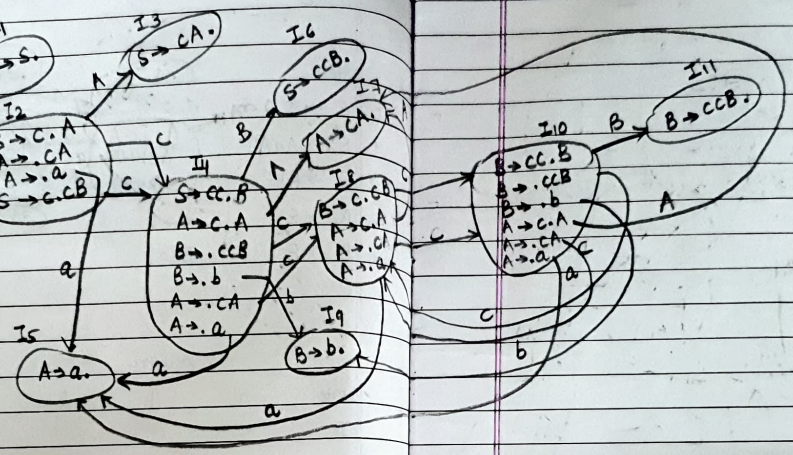
If SR and RR in one cell then it is not LR(0) parser.

Augmentation If after . NT is there then only do entry of prodn rule of that NT

Eg. S → cA | ccB
A → cA | a
B → ccB | b

Ans.

	Step1	Step2
0	S' → S	S' → .S
1	S → cA	S → .cA
2	S → ccB	S → .ccB
3	A → cA	A → .cA
4	A → a	A → .a
5	B → ccB	B → .ccB
6	B → b	B → .b



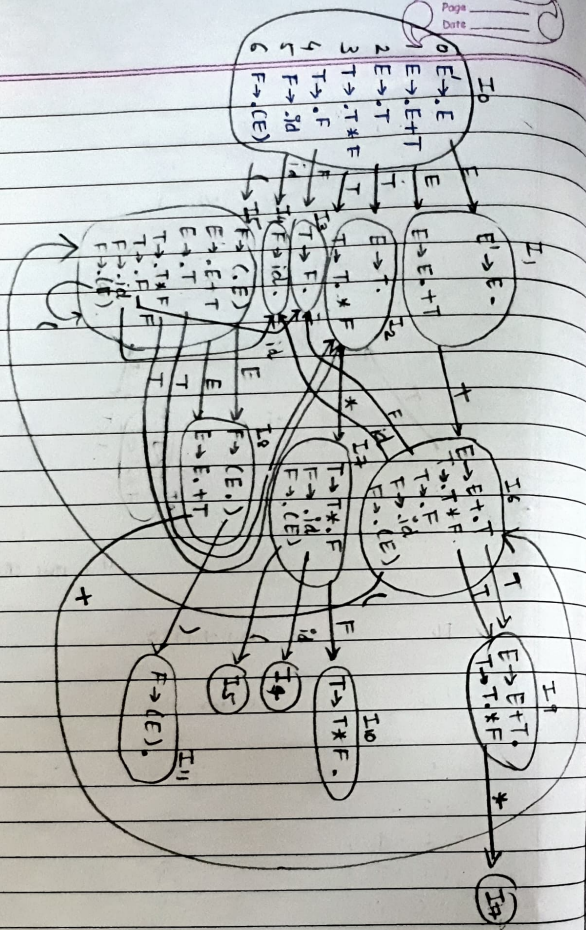
Step3

State	Action a	b	c	$	GoTo S	A	B
I_0			S_2		1		
I_1				Accept			
I_2	S_5		S_4			3	
I_3	r_1	r_1	r_1	r_1			
I_4	S_5	S_9	S_8			7	6
I_5	r_4	r_4	r_4	r_4			
I_6	r_2	r_2	r_2	r_2			
I_7	r_3	r_3	r_3	r_3			
I_8	S_5		S_{10}			7	
I_9	r_6	r_6	r_6	r_6			
I_{10}	S_5	S_9	S_8			7	11
I_{11}	r_5	r_5	r_5	r_5			

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

String: Id+id*Id

Ans 0. $E' \to E$
1. $E \to E+T$
2. $E \to T$
3. $T \to T*F$
4. $T \to F$
5. $F \to id$
6. $F \to (E)$



state	id	(	)	+	*	$	E	T	F
I_0	S_4	S_5					1	2	3
I_1				S_6		Accept			
I_2	r_2	r_2	r_2	r_2	S_7/r_2	r_2			
I_3	r_4	r_4	r_4	r_4	r_4	r_4			
I_4	r_5	r_5	r_5	r_5	r_5	r_5			
I_5	S_4	S_5					8	2	3
I_6	S_4	S_5						9	3
I_7	S_4	S_5							10
I_8			S_{11}	S_6					
I_9	r_1	r_1	r_1	r_1	S_7/r_1	r_1			
I_{10}	r_3	r_3	r_3	r_3	r_3	r_3			
I_{11}	r_6	r_6	r_6	r_6	r_6	r_6			

Goto

Action

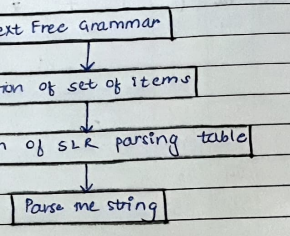
∴ Acc to LR(0), string can't be parse as it is not LR(0) as SR conflict.

copy

Stack	Input	Action
$0	id+id*id$	shift id → S_4
$0id4	+id*id$	reduce F → id
$0F3	+id*id$	reduce T → F
$0T2	+id*id$	reduce E → T
$0E1	+id*id$	shift + → S_6
$0E1+6	id*id$	shift id → S_4
$0E1+6id4	*id$	reduce F → id
$0E1+6F3	*id$	reduce T → F
$0E1+6T9	*id$	Shift/Reduce Conflict so stop.

In a if ε entry so find first.

* SLR



Steps:-

i) Augment the grammar and no. it.
ii) Find first and follow (find only follows)
iii) Construct DFA.
iv) Parsing Table
v) Stack implementation
vi) Parse Tree

instep(v) NOTE Reduction entry will be done in follows of respective NT.

Eg Apply SLR in previous question.

Ans (ii) first & follow

(i) Augment the grammar :- here left recursion & left factoring shld not be done. so E→E+T|T so neglect E and start from +

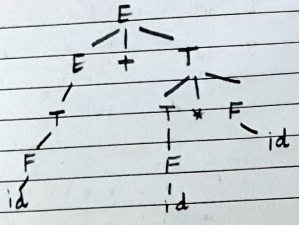
0	E'→E	E'
1	E→E+T	E
2	E→T	E
3	T→T*F	T
4	T→F	T
5	F→id	F
6	F→(E)	F

	first	follow
E→E+T/T	E	{+,*, id, (} | {+,), $}
T→T*F/F	T	{* , id, (} | {+,), $, *}
F→id/(E)	F	{id, (} | {+,), $, *}

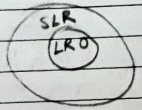
All LR called SLR NO

State	id	(	)	+	*	$	E	T	F
	Action						Goto		
I_0	S_4	S_5					1	2	3
I_1				S_6		Accept			
I_2			r_2	r_2	S_7	r_2			
I_3			r_4	r_4	r_4	r_4			
I_4			r_5	r_5	r_5	r_5			
I_5	S_4	S_5					8	2	3
I_6	S_4	S_5						9	3
I_7	S_4	S_5							10
I_8			S_{11}	S_6					
I_9			r_1	r_1	S_7	r_1			
I_{10}			r_3	r_3	r_3	r_3			
I_{11}			r_6	r_6	r_6	r_6			

Stack	Input	Action
$0	id+id*id$	shift id → S4
$0id4	+id*id$	reduce F→id
$0F3	+id*id$	reduce T→F
$0T2	+id*id$	reduce E→T
$0E1	+id*id$	shift + → S6
$0E1+6	id*id$	shift id → S4
$0E1+6id4	*id$	reduce F→id
$0E1+6F3	*id$	reduce T→F
$0E1+6T9	*id$	shift * → S7
$0E1+6T9*7	id$	shift id → S4
$0E1+6T9*7id4	$	reduce F→id
$0E1+6T9*7F10	$	reduce T→T*F
$0E1+6T9	$	reduce E→E+T
$0E1	$	Accept



NOTE



All SLR are LR

But All LR are not SLR

H.W

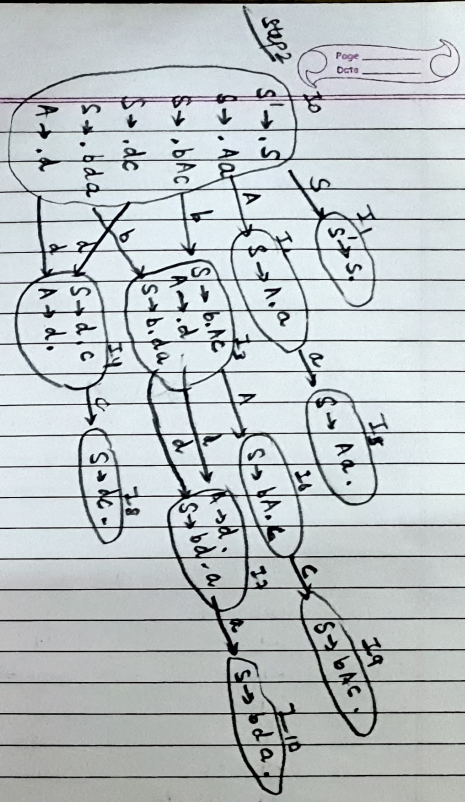
Apply LR

Eg $S \to Aa / bAc / dc / bda$

Aug $A \to d$

		State	Action					GoTo	
			a	b	c	d	$	S	A
0	$S' \to S$	I_0		S_3		S_4		1	2
1	$S \to Aa$	I_1					accept		
2	$S \to bAc$	I_2	S_5						
3	$S \to dc$	I_3				S_7			6
4	$S \to bda$	I_4	r_5	r_5	r_5	S_8	r_5	r_5	
5	$A \to d$	I_5	r_4	r_1	r_1	r_4	r_1		
		I_6			S_9				
		I_7	S_{10}						
		I_8	r_3	r_3	r_3	r_3	r_3		
		I_9	r_2	r_2	r_2	r_2	r_2		
		I_{10}	r_4	r_4	r_4	r_4	r_4		

Not LR(0) parser.



24.01.24

Page
Date

* Syntax directed definition :~ (SDD)

A syntax directed definition is generalisation of CFG in which each grammar symbol has an associative set of attributes & partitioned into 2 subsets called the synthesized and inherited attributes of that grammar symbol.

SDD
CFG + Semantic Rules

Production	Semantic Rule
E → E+T	E.val = E.val + T.val

a → symbol
x → attribute
a.x

p → symbol
q → attribute
p.q

⇒ Types of Attributes :~

(i) Synthesized :~ If a node takes value from its children then it is called as synthesized attributes.

x → yzw
x.a = y.a
x.a = z.a
x.a = w.a

(ii) Inherited :~ If a node takes value from either parent or siblings.

x → yzw
z.b = x.b
z.b = y.b
z.b = w.b

⇒ Types of SDD

(i) S-Attributed SDD :~ A SDD that uses only synthesized attributes.

Eg A → BCD Here S → Attribute

A.S = B.S
A.S = C.S
A.S = D.S

→ semantic rules are written on RHS

(iii) L-Attributed SDD :- It uses both synthesized & inherited attribute but each inherited attribute is restricted to inherits from left parent or left sibling.

Eg $A \to XYZ$ S → Attribute

$Y.S = A.S$ ✓

$Y.S = X.S$ ✓

$Y.S = Z.S$ ✗ → this won't come as RHS node is Z. only left sibling and parent is taken

Ques Syntax directed def^n of a simple desk calculator.

Token digit has a synthesized attribute lexval whose value is supplied by the lexical analyzer.

The rule associated with the prod^n $L \to En$ for the starting NT, L is just procedure that prints as a o/p the value of the arithmetic expression generated by E.

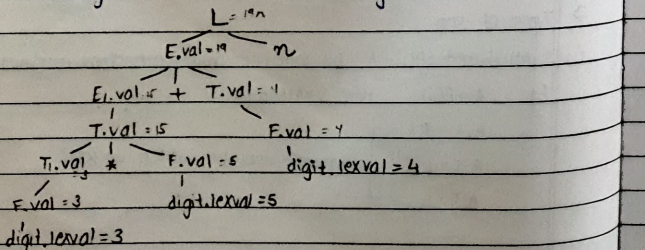
string :~ 3 * 5 + 4n

n - new line

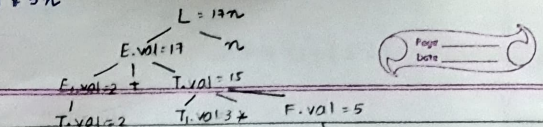
Step 1

Production	Semantic Rules
$L \to En$	Print (E.val)
$E \to E_1 + T$	E.val := E_1.val + T.val
$E \to T$	E.val := T.val
$T \to T_1 * F$	T.val := T_1.val * F.val
$T \to F$	T.val := F.val
$F \to (E)$	F.val := E.val
$F \to$ digit	F.val := digit.lexval

Step 2 Annotated parse tree



2 + 3 * 5n

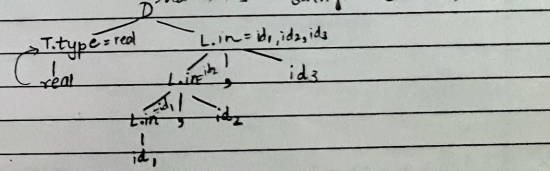


* Inherited Attribute :-

Production	
$D \to TL$	L.in := T.type
$T \to int$	T.type := integer
$T \to real$	T.type := real
$L \to L_1, id$	L.in := L.in addtype(id.entry, L.in)
$L \to id$	addtype(id.entry, L.in)

→ The NT T has a synthesized attribute type whose value is determined by the keyword in the declaration.

→ The semantic rule L.in = T.type, associated with production $D \to TL$, sets inherited attribute L.in to the type in the declaration.

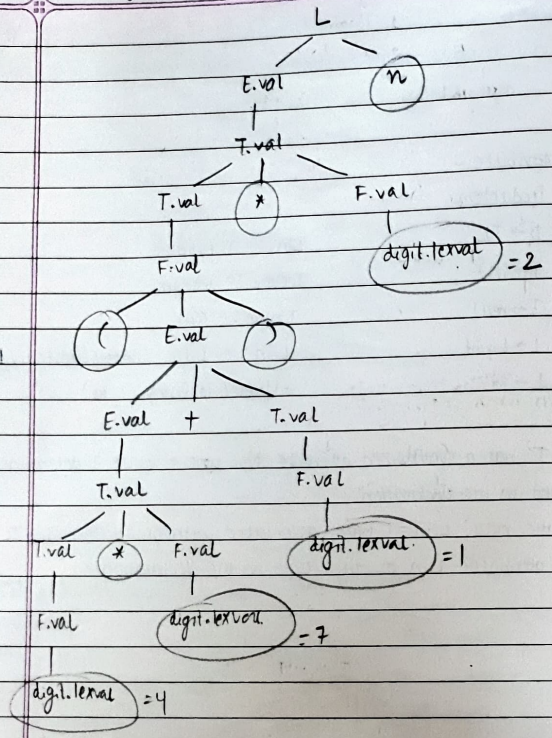


Eg For the i/p expression (4*7+1)*2n, construct an annotated parse tree acc. to the SDD.

Ans

$L \to En$	Print (E.val)
$E \to E_1 + T$	E.val := E_1.val + T.val
$E \to T$	E.val := T.val
$T \to T_1 * F$	T.val := T_1.val * F.val
$T \to F$	T.val := F.val
$F \to (E)$	F.val := E.val
$F \to$ digit	F.val := digit.lexval

String (4*7+1) * 2n



Eg (3↑2+3) * 2n

Ans step1 E → E↑E | E*E | E+E | (E) | digit

step2 L → En

step3 E → T↑E | T

T → T*F | F

F → F+G | G

G → (E) | digit

wrong

L → En

E → E↑T | T

T → T*F | F

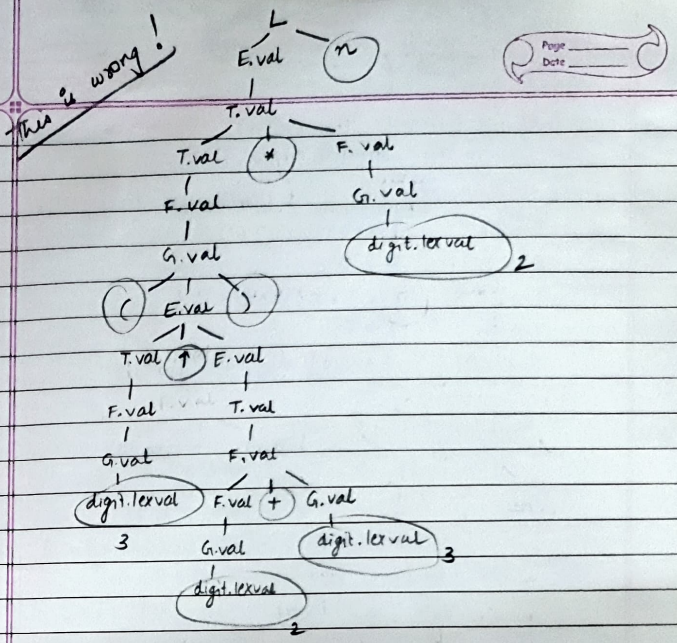
F → G↑F | G

G → (E) | digit

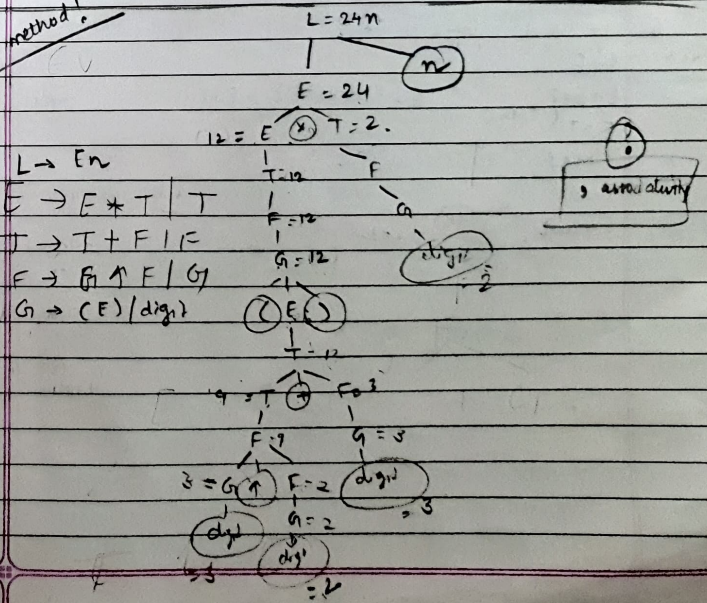
NOTE Always generate prodⁿ in ~~reverse~~ priority wise ~~stack~~ of string.

highest priority last prodⁿ.

lowest priority first prodⁿ.



Right method!



3 associativity

1+2*3
1|2*3
1+2|3
1*2|3
1|2*3|1

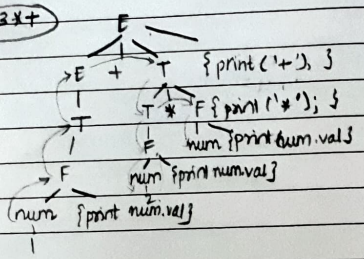
NOTE
* / same priority
Prod rule
then right vaadu pehla

* SDT (Syntax Directed Translation)
CFG + Semantic Rules

Q) 1+2*3 (Convert the string from infix to postfix).

	Semantic actions
Step1 S→E	
E→E+T	E→E+T{print('+');}
E→T	E→T
T→T*F	T→T*F{print('*');}
T→F	T→F
F→num	F→num{print num.val;}

123*+



Q) 1+2*3 infix to prefix

S→E
E→E+T E→{print('+');} E+T
E→T
T→T*F
T→F
F→num

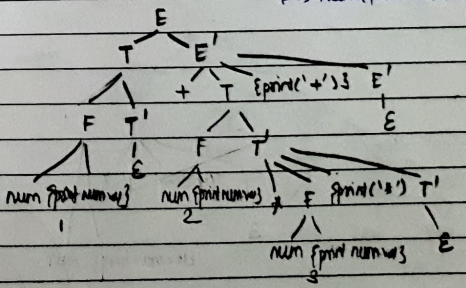
* Applications:-
1) Executing arithmetic expression
2) convert infix to postfix, infix to prefix, binary to decimal
3) Creating syntax tree
4) generating intermediate code
5) Type checking
6) Counting no. of reductions.
7) storing type information into symbol table

Q) Remove left recursion then convert infix into postfix. (1+2*3)

Ans E→E+T|T
T→T*F|F
F→num
E'→+TE'|ε

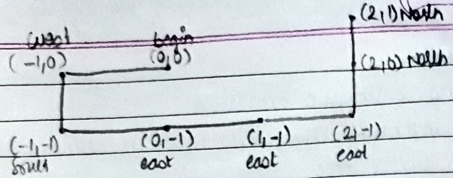
E→E+T	E→E+T {print('+');}	E→TE'
E→T	E→T	E'→+T{print('+')}E'
T→T*F	T→T*F{print('*');}	E'→ε
T→F	T→F	T→FT'
F→num	F→num{print num.val}	T'→*F{print('*')T'
		T'→ε
		F→num{print num.val}

123*+



Q) Suppose a Robot can be instructed to move one step east, north, west or south from its current position. Hint: changes in the position of robot 1) begin 2) west 3) south 4) east 5) east 6) east 7) north 8) north

answer)



Using SDT

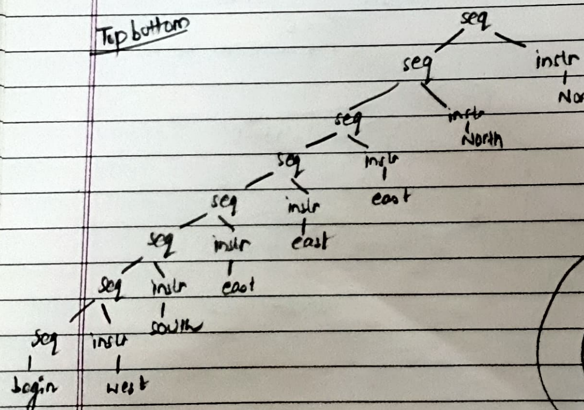
CFG

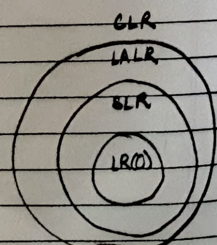
seq → seq instr | begin

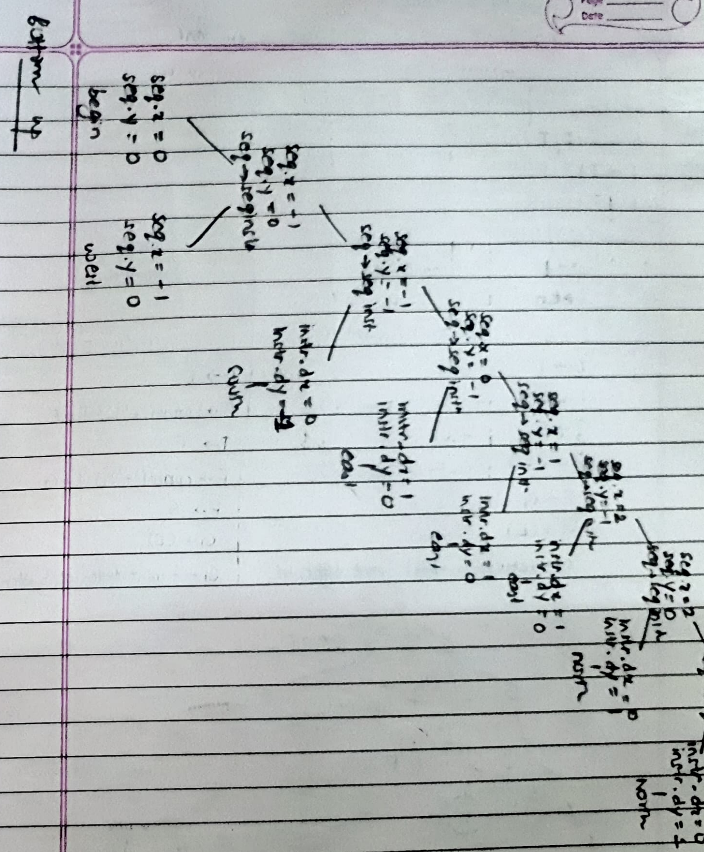
instr → east | north | west | south

Production	Semantic Rule
seq → begin	seq.x := 0 seq.y := 0
seq → seq instr	seq.x := seq.x + instr.dx seq.y := seq.y + instr.dy
instr → east	instr.dx := 1 instr.dy := 0
instr → north	instr.dx := 0 instr.dy := 1
instr → west	instr.dx := -1 instr.dy := 0
instr → south	instr.dx := 0 instr.dy := -1

Top bottom







Q) (A+B)* C - (D * E) - (F+G) — infix to postfix, infix to prefix

Ans (AB+) * C - (DE *) - (FG+)

order ( ), +, *, - Right first & left last

S→En SDT and ab aave

AB+C* DE* -FG+ -

S→En

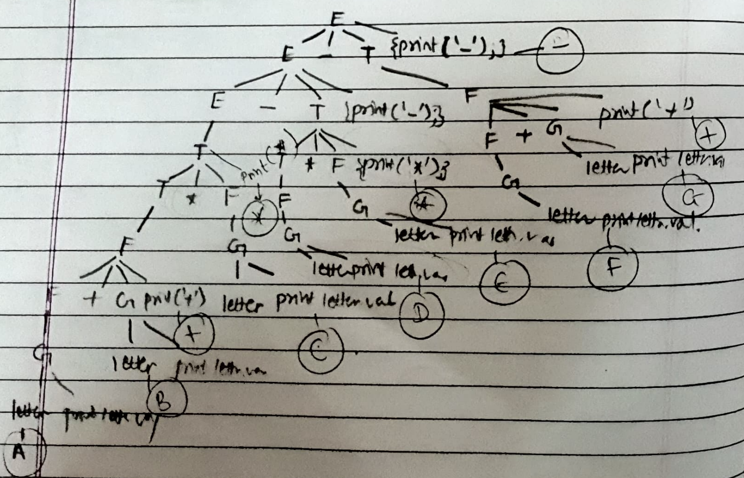
E→ E-T/T

T→T*F/F

F→F+G/G

G→(E)/letter

Prodn	Postfix Semantic Rule	Prefix Semantic Rule
S→En		
E→E-T	E→E-T{print('-');}	E→{print('-');} E-T
E→T	E→T	E→T
T→T*F	T→T*F{print('*');}	T→{print('*');} T*F
T→F	T→F	T→ F
F→F+G	F→F+G {print('+');}	F→{print('+');} F+G
F→G	F→G	F→ G
G→(E)	G→(E)	G→ (E)
G→letter	G→ {print letter.val;	G→{ print letter.val;} letter



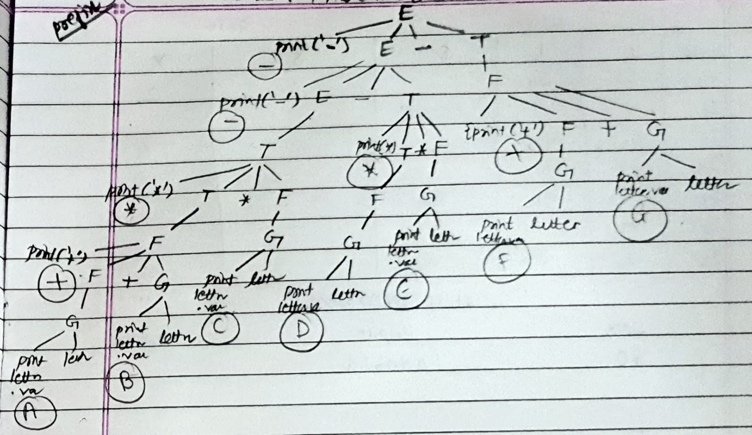
(+AB) * C - (* DE) - (+ F G)

(* + ABC) - (* DE) - (+ F G) Here associativity of minus is left to right so left vaala minus pehle solve hoga.

(- * + ABC * DE) - (+ F G)

- - * + ABC * DE + FG

prefix



Q) S→XX

X→aX|b

string aaabb

Ans step 1

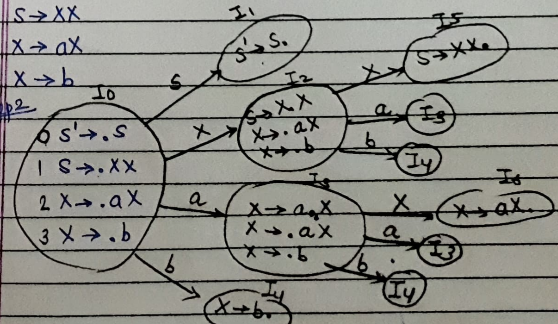
S'→S

S→XX

X→aX

X→b

step 2



	Action			GoTo	
State	a	b	$	S	X
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

It is LR(0) parser.

Stack	Input	Action
$0	aaabb$	

OS

TB → O.S (dinosaur) by Galvin Gagne (6th edition)

* **Operating System**: An OS is a program that manages the computer hardware and provides a basis for appl^n programs.
- Acts as intermediate b/w computer hardware & computer user.
- Two goals:
  - Convenient to use
  - Efficiency of Task
  - Ability to Evolve.

Objectives – Efficiency, hardware abstraction, convenience, system resource management

* **Functions of OS:~**

1) Memory Management
2) Device Management
3) Processor Management
4) Security
5) Error Detection
6) Coordination b/w software & user
7) Job Scheduling
8) File Management

Scheduling

Job Scheduling

If several jobs are ready to be brought to main memory but not enough space then system must choose among them.

CPU Scheduling

Several jobs are ready to run at the same time, the system must choose among them.