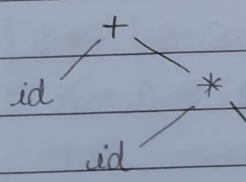
11/3/25

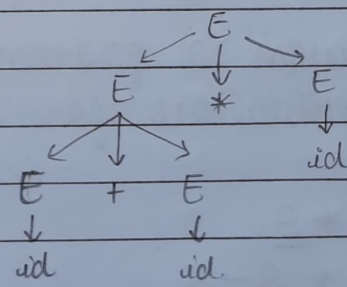
ASSIGNMENT 2

Unit-3 Syntax Analysis

Q.1 Differentiate Syntax Tree and Parse Tree. —4

Syntax Tree	Parse Tree
Here, interior nodes are operator and leaves are operands.	Here, interior nodes are Non-Terminals and leaves are Terminals.
Used for semantic analysis and code generation in the compiler	Primarily used for syntax analysis to check grammar correctness.
Smaller, more abstract representation	Larger, as it shows the entire derivation process.
It contains only meaningful information.	It contains unusable information also.
eg: Grammar: $E \rightarrow E*E \mid E+E \mid id$ String: id + id * id or a + b * c.	eg: Grammar: $E \rightarrow E*E \mid E+E \mid id$ String: id + id * id or a + b * c.





Q.2 Discuss types of derivation with example.

→ Derivations are the process of generating a string from a grammar's start symbol by repeatedly replacing a non-terminal with one of its production rules.

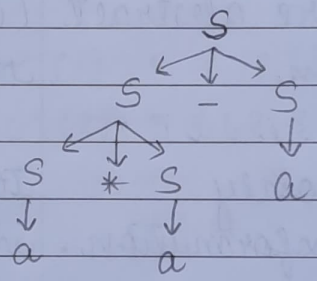
1. Leftmost Derivation (LMD)

The leftmost non-terminal in the string is replaced at every step.

eg: Given a grammar : $S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid a$

∘ Derivation for the string, a * a - a :

→ S
→ S - S
→ S * S - S
→ a * S - S
→ a * a - S
→ a * a - a



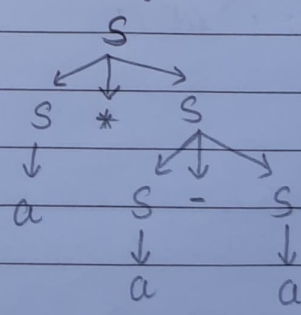
2. Rightmost Derivation (RMD)

The Rightmost non-terminal in the string is replaced at every step.

eg: Given a grammar : $S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid a$

∘ Derivation for the string, a * a - a :

S
→ S * S
→ S * S - S
→ S * S - a
→ S * a - a
→ a * a - a



Q.3 write down C program for Recursive Descend Parser for:

$S \rightarrow Aa \mid bAc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
char input[100];
int pos = 0;
void error() {
    printf("Error: Invalid input\n");
    exit(1);
}
void match(char c) {
    if(input[pos] == c) pos++;
    else error();
}
void S() {
    if(input[pos] == 'd') { // S -> Aa
        match('d'); match('a');
    } else if(input[pos] == 'b') { // S -> bAc
        match('b'); match('d');
        if(input[pos] == 'c') match('c');
        else if(input[pos] == 'a') match('a');
        else error();
    } else error();
}
int main() {
    printf("Enter the input string: ");
```

```
    scanf("%s", input);
    S();
    printf(input[pos] ? "Error : Input is invalid.
        \n" : "Input is valid.\n");
    return 0;
}
```

// Output : Enter the input string : da
Input is valid.

(bdc, bda) → valid , (d, bd, bdcx, abc) → Invalid

Q.4 Compute the operator precedence matrix and precedence function for the following grammar if it exists : +, *, -, /, id, num, (and) are terminal symbols.

G → E
E → E+T | E-T | T
T → T*F | T/F | F
F → num | id | ∈

→ Operator Precedence Table (Matrix) :-

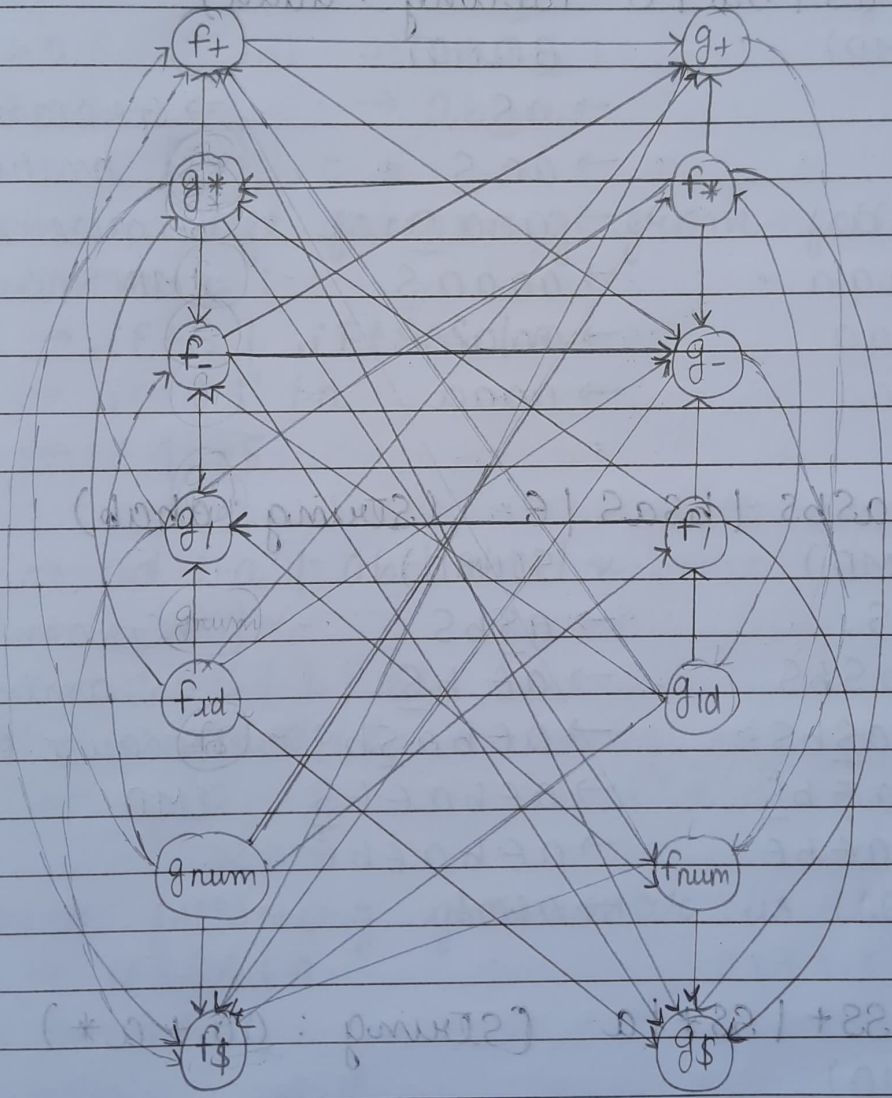
g →

f ↓	+	*	-	/	id	num	$
+	>	<	>	<	<	<	>
*	>	>	>	>	<	<	>
-	>	<	>	<	<	<	>
/	>	>	>	>	<	<	>
id	>	>	>	>	-	-	>
num	>	>	>	>	-	-	>
$	<	<	<	<	<	<	-

- Functions fa & ga for 'a' are :-

f_+	$g*$	$f-$	$g/$	f_{id}	g_{num}	$f\$$
$g+$	$f*$	$g-$	$f/$	g_{id}	f_{num}	$g\$$

- Computing precedence functions to check cycle :-



Function Table :-

	+	*	-	/	id	num	$
f	2	4	2	4	6	6	0
g	3	5	3	5	-	-	0

*NOT ACCURATE (COUNT AGAIN)

Q.5 Check whether following grammar are ambiguous or not :

1. $S \rightarrow aS \mid Sa \mid \epsilon$ (String : aaaa)

—↓

S (LMD)	S (LMD)	
→ Sa	→ aS	⇒ Grammar is ambiguous because of multiple LMD.
→ Saa	→ aaS	
→ Saaa	→ aaaS	
→ Saaaa	→ aaaaS	
→ ϵaaaa	→ aaaaϵ	
→ aaaa	→ aaaa	

2. $S \rightarrow aSbS \mid bSaS \mid \epsilon$ (String : abab)

—↓

S (LMD)	S (LMD)	
→ aSbS	→ aSbS	⇒ Grammar is ambiguous because of multiple LMD.
→ abSaSbS	→ aϵbS	
→ abϵaSbS	→ aϵbaSbS	
→ abϵaϵbS	→ aϵbaϵbS	
→ abϵaϵbϵ	→ aϵbaϵbϵ	
→ abab	→ abab	

3. $S \rightarrow SS+ \mid SS* \mid a$ (String : aa+a*)

—↓

S (LMD)

→ SS*

→ SS+S*

→ aS+S*

→ aa+S*

→ aa+a*

⇒ Grammar is unambiguous because of single LMD.

4. Show that the CGF with productions :
S → a | SS | bSS | SSb | SbS · is ambiguous (Assume string).

—4 Assumed string : abaa

S (LMD)	S (LMD)
→ SS	→ SbS
→ aS	→ abS
→ abSS	→ abSS
→ abaS	→ abaS
→ abaa	→ abaa.

⇒ So, this is how grammar is ambiguous.

Q.6 Remove left factoring from following grammar :

1. S → iEtS | iEtSeS | a.

—4 S → iEtS S' | a
S' → ∈ | eS

2. A → ad | a | ab | abc | x.

—4 A → a A' | x
A' → d | ∈ | b | bc
A' → bA'' | d | ∈
A'' → ∈ | c

Q.7 Check following grammar is LL(1) or not ?
S → iCtSA | a.
A → eS | ∈
C → b.

—4 Step-1 : No need for left factoring or recursion.

Step-2: Finding first and follow of non-terminal.

NT	First	Follow
S	{i, a}	{e, $}
A	{e, ϵ}	{e, $}
C	{b}	{t}

Step-3: Constructing Predictive Parsing Table.

NT	a	b	e	i	t	$
S	S→a			S→iCtSA		
A			A→eS A→ϵ			A→ϵ
C		C→b				

∴ Here, the given grammar is not LL(1) because there are multiple entries in a cell. (overlap between First(A) and Follow(A)).

Q.8 Given the grammar, evaluate the string a * (b+c) using Shift Reduce Parser

E → E+T | T

T → T * F | F

F → (E) | id

String : a * (b+c) = id * (id+id)

Stack	Input	Action
$	id * (id + id)	Shift id
id	* (id + id)	Reduce F → id
F	* (id + id)	Reduce T → F
T	* (id + id)	Shift *
T *	(id + id)	Shift (
T * (	id + id)	Shift id
T * (id	+ id)	Reduce F → id
T * (F	+ id)	Reduce T → F
T * (T	+ id)	Reduce E → T
T * (E	+ id)	Shift +
T * (E +	id)	Shift id
T * (E + id	)	Reduce F → id
T * (E + F	)	Reduce T → F
T * (E + T	)	Shift)
T * (E + T)	-	Reduce E → E+T
T * (E)	-	Reduce F → (E)
T * F	-	Reduce T → T*F
T	-	Reduce E → T
E	-	ACCEPT

Q.9 Construct SLR parsing table for the given grammar:

S → Aa | bAC | bBa

A → d

B → d

Step-1: Augmented Grammar.

S' → S — r_0

S → Aa — r_1

S → bAC — r_2

S → bBa — r_3

A → d — r_4

B → d — r_5

Step-2: canonical collection of LR(0) items.

$I_0 = S' \to .S$
$S \to .Aa$
$S \to .bAC$
$S \to .bBa$
$A \to .d$

- goto $(I_0, S) = S' \to S.$ — I_1
- goto $(I_0, A) = S \to A.a$ — I_2
- goto $(I_0, b) = S \to b.AC$ — I_3
  $A \to .d$
  $S \to b.Ba$
  $B \to .d$
- goto $(I_0, d) = A \to d.$ — I_4
- goto $(I_2, a) = S \to Aa.$ — I_5
- goto $(I_3, A) = S \to bA.C$ — I_6
- goto $(I_3, d) = A \to d.$ — I_7
  $B \to d.$
- goto $(I_3, B) = S \to bB.a$ — I_8
- goto $(I_6, C) = S \to bAC.$ — I_9
- goto $(I_8, a) = S \to bBa.$ — I_{10}

Step-3: Finding follow ups.

NT	Follow
S	{$}
A	{a, c}
B	{a}

Step-4: SLR Parsing Table. [(r)educe : production rule Number (s)hift : state Number]

	Action					goto		
State	a	b	c	d	$	S	A	B
I_0		S_3		S_4		1	2	
I_1					Accept			
I_2	S_5							
I_3				S_7			6	8
I_4	r_4		r_4					
I_5					r_1			
I_6			S_9					
I_7	r_4/r_5		r_4		r_2			
I_8	S_{10}				r_2			
I_9					r_2			
I_{10}					r_3			

Q.10 Develop CLR parsing table for the following grammar:

$S \to CC$

$C \to aC / d$.

→ Step-1: Augmented Grammar.

$S' \to S$ — r_0 $\quad$ $C \to aC$ — r_2

$S \to CC$ — r_1 $\quad$ $C \to d$ — r_3

Step-2: Canonical forms.

I_0 = $S' \to .S$, \$
$S \to .CC$, \$
$C \to .aC$, a/d
$C \to .d$, a/d

- goto(I_0, S) = $S' \to S.$, \$ — I_1
- goto(I_0, C) = $S \to C.C$, \$ — I_2
  $C \to .aC$, \$
  $C \to .d$, \$
- goto(I_0, a) = $C \to a.C$, a/d — I_3
  $C \to .aC$, a/d
  $C \to .d$, a/d
- goto(I_0, d) = $C \to d.$, a/d — I_4
- goto(I_2, C) = $S \to CC.$, \$ — I_5
- goto(I_2, a) = $C \to a.C$, \$/d — I_6
  $C \to .aC$, \$
  $C \to .d$, \$
- goto(I_2, d) = $C \to d.$, \$ — I_7
- goto(I_3, C) = $C \to aC.$, a/d — I_8
- goto(I_3, a) = $C \to a.C$, a/d — I_3
  $C \to .aC$, a/d
  $C \to .d$, a/d

- goto(I_3, d) = $C \to d\cdot$, a/d — (I_4)
- goto(I_6, C) = $C \to aC\cdot$, \$ — (I_9)
- goto(I_6, a) = $C \to a\cdot C$, \$ — (I_6)
  $C \to \cdot aC$, \$
  $C \to \cdot d$, \$
- goto(I_6, d) = $C \to d\cdot$, \$ — (I_7)

Step-3: CLR Parsing Table

State	Action			goto	
	a	d	\$	S	C
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

Q.10 Given grammar is CLR or LALR justify your answer. If LALR then construct the parsing table for LALR:

i) $S \to Aa \mid bAc \mid dc \mid bda$.

ii) $A \to d$.

Step-1: Augmented Grammar

$S' \to S$ — r_0 $\quad$ $S \to dc$ — r_3

$S \to Aa$ — r_1 $\quad$ $S \to bda$ — r_4

$S \to bAC$ — r_2 $\quad$ $A \to d$ — r_5

$S \to$

Step-2: Canonical Form

$I_0 = S' \to .S$, \$

$S \to .Aa$, \$

$S \to .bAC$, \$

$S \to .dc$, \$

$S \to .bda$, \$

$A \to .d$, d

- goto(I_0, S) = $S' \to S.$, \$ — (I_1)
- goto(I_0, A) = $S \to A.a$, \$ — (I_2)
- goto(I_0, b) = $S \to b.AC$, \$ — (I_3)

  $S \to b.da$, \$

  $A \to .d$, \$
- goto(I_0, d) = $S \to d.c$, \$ — (I_4)

  $A \to d.$, \$
- goto(I_2, a) = $S \to Aa.$, \$ — (I_5)
- goto(I_3, A) = $S \to bA.C$, \$ — (I_6)
- goto(I_3, d) = $S \to bd.a$, \$ — (I_7)

  $A \to d.$, \$

- goto(I_4, C) = $S \rightarrow dc.$, \$ — (I_8)
- goto(I_6, C) = $S \rightarrow bAC.$, \$ — (I_9)
- goto(I_7, a) = $S \rightarrow bda.$, \$ — (I_{10})

∴ Here, there is no similar production which can be combined, so it is NOT LALR.

(iii) $S \rightarrow CC$.
$C \rightarrow aC \mid d$.

→ Step-1: Augmented Grammar

$S' \rightarrow S$ — r_0 $C \rightarrow aC$ — r_2
$S \rightarrow CC$ — r_1 $C \rightarrow d$ — r_3

Step-2: Canonical Form

I_0 = $S' \rightarrow .S$, \$
$S \rightarrow .CC$, \$
$C \rightarrow .aC$, a/d
$C \rightarrow .d$, a/d

- goto(I_0, S) = $S' \rightarrow S.$, \$ — (I_1)
- goto(I_0, C) = $S \rightarrow C.C$, \$ — (I_2)
  $C \rightarrow .aC$, \$
  $C \rightarrow .d$, \$
- goto(I_0, a) = $C \rightarrow a.C$, a/d — (I_3)
  $C \rightarrow .aC$, a/d
  $C \rightarrow .d$, a/d

- goto(I_0, d) = $C \to d.$, a/d — (I_4)
- goto(I_2, C) = $S \to CC.$, \$ — (I_5)
- goto(I_2, a) = $C \to a.C$, \$ — (I_6)
  $C \to .aC$, \$
  $C \to .d$, \$
- goto(I_2, d) = $C \to d.$, \$ — (I_7)
- goto(I_3, C) = $C \to aC.$, a/d — (I_8)
- goto(I_3, a) = $C \to a.C$, a/d — (I_3)
  $C \to .aC$, a/d
  $C \to .d$, a/d
- goto(I_3, d) = $C \to d.$, a/d — (I_4)
- goto(I_6, C) = $C \to aC.$, \$ — (I_9)
- goto(I_6, a) = $C \to a.C$, \$ — (I_6)
  $C \to .aC$, \$
  $C \to .d$, \$
- goto(I_6, d) = $C \to d.$, \$ — (I_7)

Step-3: LALR Testing.

Here, (I_3, I_6) , (I_4, I_7) and (I_8, I_9) have the same production of LL(0) but only lookahead symbols are different.

So, it is LALR where;

$I_3, I_6 = I_{36}$

$I_4, I_7 = I_{47}$

$I_8, I_9 = I_{89}$

Step-4: LALR Parsing Table.

	Action			goto	
State	a	d	$	S	C
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		