Day-1 Analytical -1

1. Find the no of tokens in the following statement

```
int main()
{
  int a=10, b=30;
  if (a>b)
    return (b);
  else
    return (a);
}
```

int : keyword
Main : Identifier
( ) : Left and Right parentheses.
{ : left curly brace
int : keyword
a : identifier.
= : operator
10 : Integral literal
, : Comma
b : Identifier
= : operator
30 : Integer
; : Semicolon
if : keyword
( ) : left & right parentheses

a : Identifier

< : less than

b : Identifier

Clr : keyword

return : keyword

() : left & right paranthesis

a : Identifier

; : Semicolon.

→ There are 30 tokens in given C code.

② write a regular Expression to denote set of all strings over $\{0,1\}^*$ Containg Substring 101:

$(0+1)^*$ 101 $(0+1)^*$

the regular Expression

$(0+1)^*$ 101 $(0+1)^*$ will match any string that Contain the Substring "101" as a Con quous Sequence Character

③ Write a regular Expression that have at least two Consective 0's (00) 1's.

(00+ | 11+)

| → act as OR operator

00+ → least two Consective 0's

11+ → least two Consective 1's

For example the regular Expression will match string like-

"1100", "0011", "1111", "000000", "111111" etc.

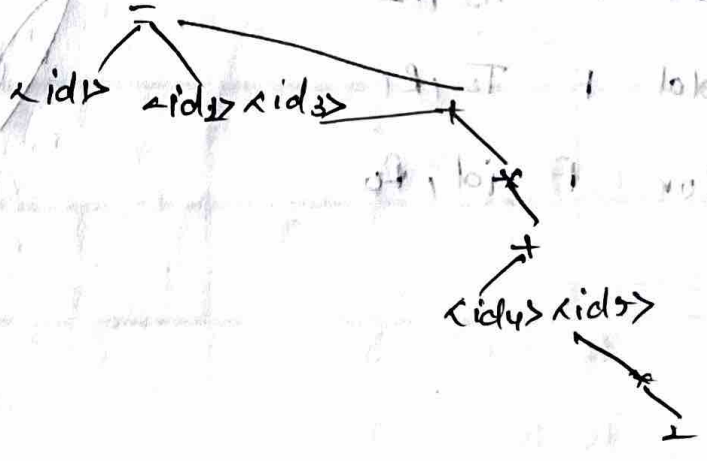
④ How would you trace the program Sequence " a=(b+c)* (b+c)*2 " for all phrases.

1. Lexical Analysis

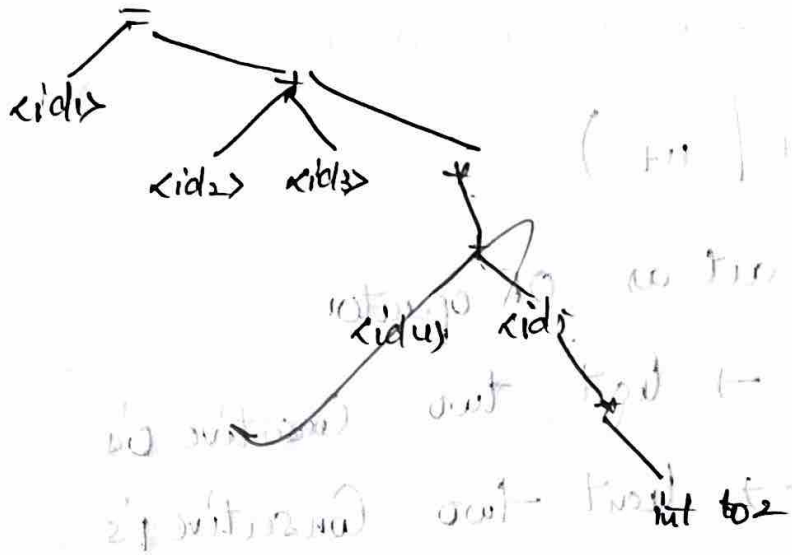
<a><=> <(> <b> <+> <c> <)> <*> <(> <b> <+>

<c> <)> <*> <2>

Syntax Analysis:



(3) Semantic Code:



Intermediate Code:

~~Temp1 = int to2~~

Temp2 = id4 + id5 * Temp1

Temp3 = id2 + id3 * ~~Temp2~~ Temp3

id1 = Temp3

Code optimizer

~~temp1~~ = T3 * 2

id1 = T2 + temp1

Code generator

MOV F #3, R2

MOV F #2, R2

Add F T2, R1

MOV F id1, R0

22/7/23

Analytical-2

① Predictive parser

$S \to a \mid \uparrow \mid (T)$ $T \to T, S \mid S$

(i) $S \to a \mid \uparrow \mid (T)$

$T \to T, S \mid S$

Step-1 Eliminate left-recursive from grammar

$S \to a \mid \uparrow \mid (T)$

$T \to ST'$

$T' \to ,ST' \mid \varepsilon$

Step-2: Create the first and follow sets for each non-terminal.

Set-1

First(S) = {a, ↑, (}

First(T) = {a, ↑, (}

Set-2

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

Follow(T) = {), $}

Step-3: predicitive parsing table.

	a	↑	(	)	$
S	a	↑	(		

Parsing table for T:

	a	↑	(	)	,	$
T	a	↑	(			
T_1					)	

20) SCR

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

Step-1: Augmit the grammer

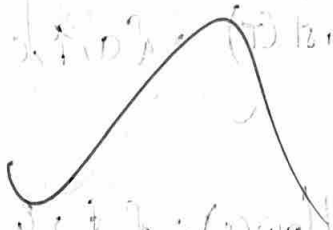
$S' \to S$

$S \to CC$

$C \to \cdot cC/d$

Step-2: Computer the Closure are, Go To set for item

LR(0) item.



1. $S' \to \cdot S$
2. $S \to \cdot CC$
3. $S \to \cdot CC$
4. $S \to \cdot d$
5. $C \to c\cdot C$
6. $C \to \cdot d$

I0:
Closure(I0):

1. $S' \to \cdot S$

Go To (I0):

· Go To(I0,S) = I1

I2 : Closure(I2)

1. $S \to \cdot CC$
2. $S \to \cdot CC$
3. $S \to \cdot d$
4. $C \to \cdot cC$

1. $S' \rightarrow S$

Go to (I1):

Go to (I1, C) = I2

Go To (I2):

Go To (I2, C) = I3

Go To (I2, C) = I4

Go To (I2, d) = I5

I3:

Closure (I3)

1. $S \rightarrow CC$

2. $C \rightarrow .CC$

3. $C \rightarrow .d$

Go to (I3):

Go to (I3, C) = I6

I4:

Closure (I4):

1. $S \rightarrow CC$

2. $C \rightarrow .CC$

3. $C \rightarrow .d$

Go To (I4):

Go to (I4, C) = I7

I5:

Closure (I5)

I6:

Closure (I6):

1. $C \rightarrow C.C$

2. $C \rightarrow .CC$

3. $C \rightarrow .d$

Go To (I6, C) = I8

I7:

Closure (I7):

$C \rightarrow cC.$

I8:

Closure (I8)

1. $C \rightarrow cd$

	c	d	$	S	C
10				Shift 1	
11			Accept		
12	Shift	Shift 5			
13	Shift	Shift 5		Shift 4	Shift 6
14	Shift	Shift 5		Shift 4	11
15				4	11
16					
17					
18					

3) Grammer

$S \rightarrow Aa\,Ab \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$ is LL(1) or not?

Step-1: First & Foll for non-terminal

First(A) = {ε}

First(B) = {ε}

First(S) = {a, b}

NOT LL(1)

4) Grammer

$E \rightarrow 2E2$

$E \rightarrow 3E3$

$E \rightarrow 4$

Parsing input string 32423

Step-1: Initialization

Stack = $

Input: 3*4+2$

Step-2: parsing

Stack	Input	Action
$	3*4+2$	Shift 3, push 3
$3	*4+2$	shift *, push *
$3*	4+2$	Reduce E→4 pop 4
$ E	+2$	shift +, push +
$ E+	2$	shift 2, push 2
$ E+2	$	Reduce E→2 pop 2
$ E+	$	Reduce E→E+E pop 3
$ E	$	Accept.

7/3

Analytical Day-8

Syntax Directed Translation

2* (LL5)

CFG for arithmetic Expression:

E → E+T | E-T | T E → represent Expression

T → T*F | T/F | F T → Terms

F → (F) | num F → factor

SDT Rules:

E → E+T { right = pop(); left = pop(); Emit(left + right + '+'); } push(left + right + '+'); }

| E-T { right = pop(); left = pop(); Emit(left + right + '-'); push(left + right + '-'); }

| T { result = pop(); push(result); }

T → T*F { right = pop(); left = pop(); Emit(left + right + '*'); push(left + right + '*'); }

| T/F { right = pop(); left = pop(); Emit(left + right + '/'); push(left + right + '/'); }

| F { result = pop(); push(result); }

F → (E) { result = pop(); push(result); }

| num { Emit(num); push(num); }

SDD Scheme

3 * 5 + 6 * 3

E → E+T | E-T | T

T → T*F | T/F | F

F → (E) | num

Semantic Rule

1. E → E1 + T { E.val = E1.val + T.val }

   | E1 - T { E.val = E1.val - T.val }

   | T { E.val = T.val }

2. T → T1 * F { T.val = T1.val * F.val }

   | T1 / F { T.val = T1.val / F.val }

   | F { T.val = F.val }

3. F → (E) { F.val = E.val }

   | num. { F.val = num.val }

Grammer G

$E' \to E$

$E \to E+n/n$

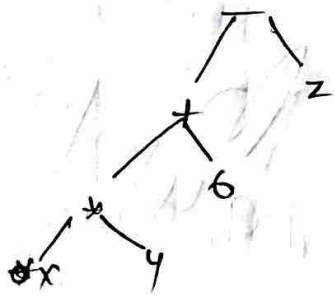
Passing Action

Stack	Input	Action
$	n+n$	Shift 'n'
$n	+n$	Shift '+'
$n+	n$	Shift 'n'
$n+n	$	Reduce $E \to n$
$ E	$	Reduce $E \to n+n$
$ E'	$	Reduce $E' \to n$

Syntax tree for Expression

$x*y+6-z$

Syntax tree



Syntax - directed

S → E N
E → E + T / E - T / T
T → T * F / T / F / F
F → (E) / digit
N → ;

Input : 5 * 3 + 2

1. Attributed Grammer

S → E {S.val = E.val}
E → E1 + T {E.val = E1.val + T.val}
| E1 - T {E.val = E1.val - T.val}
| T {E.val = T.val}
T → T1 * F {T.val = T1.val * F.val}
| T1 / F {T.val = T1.val / F.val}
| F {T.val = F.val}
F → (E) {F.val = E.val}
| digit {E.val = digit.val}
N → ; {N.val = ;}

→ result = 5 * 3 + 2

5 * 3 = 15
15 + 2 = 17
5 * 3 + 2 = 17 //

24/7/25

24/7/23 Analytical day-4

1) -(a+b)*(c+d)+(a+b+c) into

1. Quadruples

1. - a, b, t1
2. + c, d, t2
3. * t1, t2, t3
4. + a, b, t4
5. + t4, c, t5
6. + t3, t5, result

2 Triples:

1. - a, b
2. + c, d
3. * (1), (2)
4. + a, b
5. + (4), c
6. + (3), (5)

3 Indirect Triples:

1. - a, b, t1
2. + c, d, t2
3. * t1, t2, t3
4. + a, b, t4
5. + t4, c, t5
6. + t3, t5, result

② Translate Expression

$a * -(b+c)$

③ Quadruple:

1. + b, c, t1
2. - 0, t1, t2
3. * a, t2, result

④ Postfix Notation

$a * -(b+c)^{*}$

⇒ a b c + − *

⑤ Three-address Code:

result = operand1 operator operand2

$t_1 = b + c$

$t_2 = -t_1$

result = $a * t_2$;

⑥ Translation Scheme to generate three address code

$g = a+b-c*d$.

1. S → g := E
2. E → E + T
3. E → E − T
4. E → T
5. T → T * f
6. T → f
7. f → a
8. f → b
9. f → c
10. f → d

Three address code for

$g := a+b-c*d$

1. $t_1 = a+b$
2. $t_2 = c*d$
3. $t_3 = t_1 - t_2$
4. $g = t_3$

⑦ Translate (a<b) or (c<d) and (d<e) into three address statment

⑧ 1. if a<b goto L1

2. goto L2
3. goto L1: if c<d goto L3
4. goto L2
5. L3: if d<e goto L4
6. goto L2
7. L4: $t_1 = 1$
8. goto L5
9. L2 : $t_1 = 0$
10. t_5 : ....

⇒ L1, L3, L3 and L4 are placeholder labels

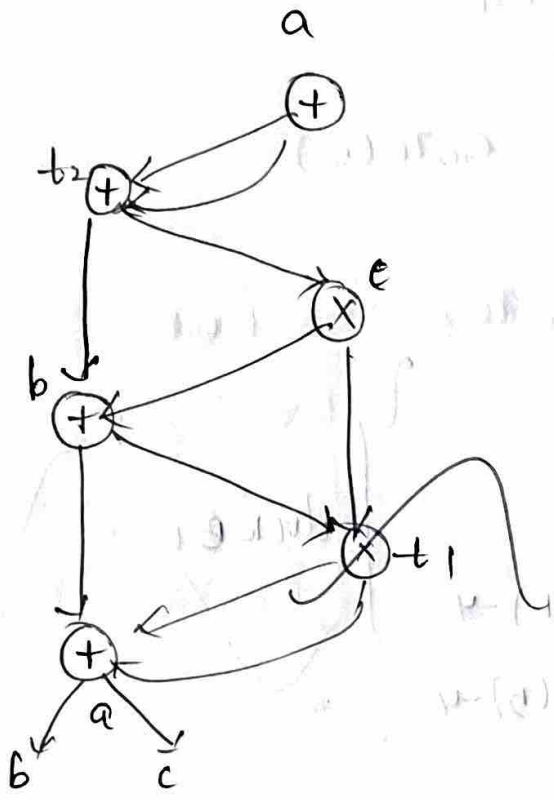
1. if a<b goto L6
2. goto L7
3. L6 : if c<d goto L8
4. goto L7
5. L8 : if d<e goto L9
6. goto L7
7. L9 : $t_1 = 1$
8. goto L10
9. L7 : $t_1 = 0$
10. L10 : ....

25/7/23

① DAG for the following address Code:

1. $a = b + c$
2. $t_1 = a \times a$
3. $b = t_1 + a$
4. $c = t_1 \times b$
5. $t_2 = c + b$
6. $a = t_2 + t_2$

Directed Acyclic Graph



(2) Basic Blocks

1. PROD = 0
   I = 1
   T_2 = addr(A) - 4
   T_4 = addr(B) - 4
   T_1 = 4 × I
   T_3 = T_2[T_1]
   T_5 = T_4[T_1]
   T_6 = T_3 × T_5
   PROD = PROD + T_6
   I = I + 1
   IF I ≤ 20 GOTO (5)

(4)
- PROD = 0
- T_1 = 4 × 0

(1) PROD = 0
(2) I = 1
(3) T_2 = addr(A) - 4
(4) T_4 = addr(B) - 4

Block B1

5 T_1 = 4 × I
6 T_3 = T_2[T_1]
7 T_5 = T_4[T_1]
8 T_6 = T_3 × T_5
9 PROD = PROD + T_6
10 I = I + 1
11 If I <= 20 GOTO(5)

Block B2

(3) w = (A+B) + (A+C) + (A+C)

t_1 = (A+B)
t_2 = (A+C)
t_3 = (A+B)
t_4 = t_1 + t_2 + t_3
w = t_4

Code.

```
MOV A, R0
ADD B, R0
ADD A, R0
ADD C, R0
ADD A, R0
ADD C, R0
MOV R0, w
```

(4) Basic Block and Flow Graph

```
for (i=1 to n)
{
   s = u;
   while (j <= n)
   {
      A = B * C(0);
      j = j + 1
   }
}
```

(4) PROD = 0
I = 1 → Block-1

T2 = addr(A) - 4

T4 = addr(B) - 4

T1 = 4 x I

T3 = T2[T1]

T5 = T4[T1]

T6 = T3 x T5

PROD = PROD + T6

I = I + 1 → Block-2

If I <= 20 GOTO (2) → Block-3

Analytical-6

i) Consider the following grammer and eliminate left recursion

A → ABd/Aa/a

B → Be/b

Step-1: Remove direct left recursion from the non-terminal A.

A → AaBd/aBd/Aa/A

Step-2: Factor out the common prefix from the left hand side of production.

A → aA'

A' → aBdA'/BdA'/ε

The resulting grammer with eliminated left recursion is:

A → aA'

A' → aBdA'/BdA'/ε

ii) Consider the following grammer and eliminate left recursion

A → AA∝/β

Step-1: A → βA'

Step-2: Introduce the new non-terminal to replace the left part

S → AA∝

Step-3: Replace the left-recursive production with new non-terminal:

$A \rightarrow \beta/\beta$

Step-4: Factor out Common prefix from the right hand

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

The resulting grammar will be after elimination of left recursion.

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

(iii) Do left factoring in the following grammar.

a. $S \rightarrow bSSaaS / bSSaSb / bSb / a$

b. $S \rightarrow aSSbS / aSaSb / abb / b$

To left factor this grammar, let Common for Common prefix.

1. $S \rightarrow bS'$
2. $S' \rightarrow SSaaS / SaSb / Sb / \varepsilon$
3. $S \rightarrow a$

b. $S \rightarrow aSSbS / aSaSb / abb / b$

1. $S \rightarrow aS'$
2. $S' \rightarrow SbS / SaSb / \varepsilon$
3. $S \rightarrow abb$
4. $S \rightarrow b$

(iv) Check whether the following grammar is a LL(1) or not

$S \rightarrow iEtS / iEtSeS / a$

$E \rightarrow b$

First and follow Set

1. First (S) = {i, a}
2. First (E) = {b}
3. follow (S) = {$, e}
4. Follow (E) = {t, e}

Checking the Condition for the given grammar:

1. The first set of S = {i, a}
2. The First set of E = {b} and follow set of E = {t, e}

Analytical - 7

① Construct a recursive descent parser for the following grammer using

$E \to TE'$

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

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) / id$

```
Procedure E()
{
    T();
    E'();
}
```
$E \to TE'$

```
Procedure E'()
{
  if input symbol = '+' then
    advance();
    T();
    E'();
}
```
$E' \to +TE'$

```
Procedure T()
{
    F();
    T'();
}
```
$T \to FT'$

```
procedure T'()
{
  if input symbol = '*' then
    advance();
    F();
    T'();
}
```
$T' \to *FT'$

```
Procedure F()
{
  if input symbol = 'id' then
  advance();
  else if input-symbol = '(' then
  advance();
  E();
  if input-symbol = ')'
  advance();
  else error();
  else error();
}
```
$F \to id$

$F \to (E)$

② Construct the following grammer for operator precedence parser.

solution.

	a	(	)	,	$
a		>	>	>	>
(	<	>	>	>	>
)	<	>	>	>	>
,	<	<	>	>	>
$	<	<	<	<	

Operator precedence Table.

Step-1

$ (a,(a,a)) $

we insert precedence operator b/w symbol

$ < (< a >, < (< a >, < a >) >) > $

Step-2:

we can scan and parse the string as

$ < (< a >, < (< a >, < a >) >) > $

$ < (S, < (< a > , < a >) >) > $

$ < (S, < (< a > , < a >) >) > $

$ < (S, < (S, < a >) >) > $

$ < (S, < (S, S) >) > $

$ < (S, < (L, S) >) > $

$ < (S, < (L) >) > $

$ < (S, S) > $

$ < (L, S) > $

$ < (L) > $

$ < S > $

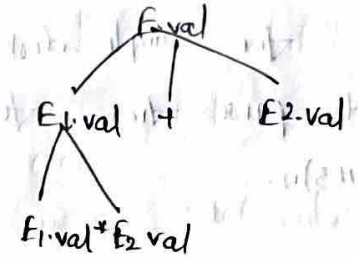
$ $

Design the dependency graph for the following grammer.

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

Production	Semantic 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



4. Design the dependency graph for the following grammer.

S → T list

T → int

T → float

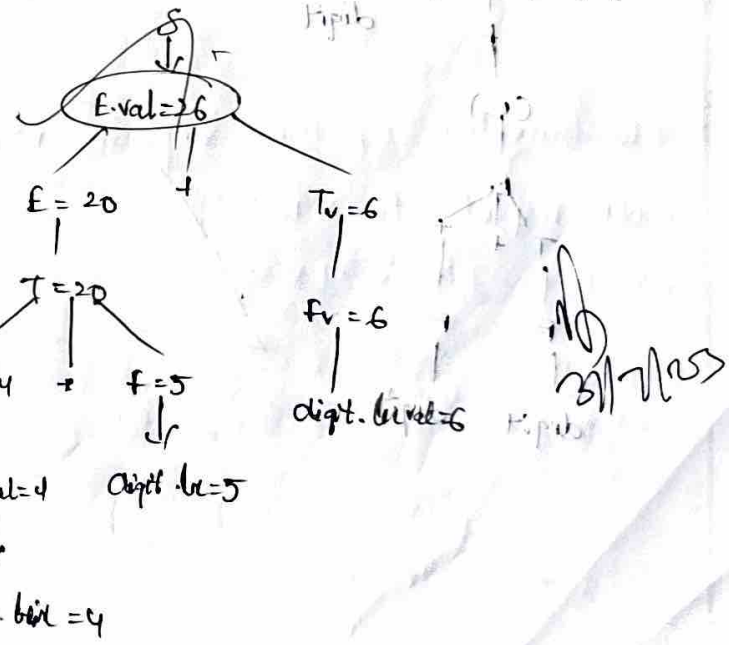
T → char

T → double

List → List1, id

List → id.

Design for above grammer of dependency graph.

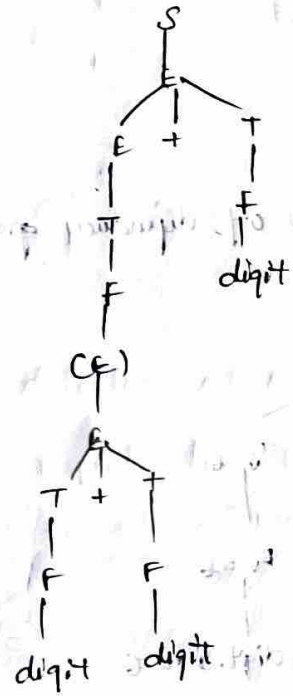


Analytical Page 5

① Illustrate the SDD for simple desk calculator and draw annotated parse tree for the expression (6+4)* (4+3).

S → En
E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | digit

④

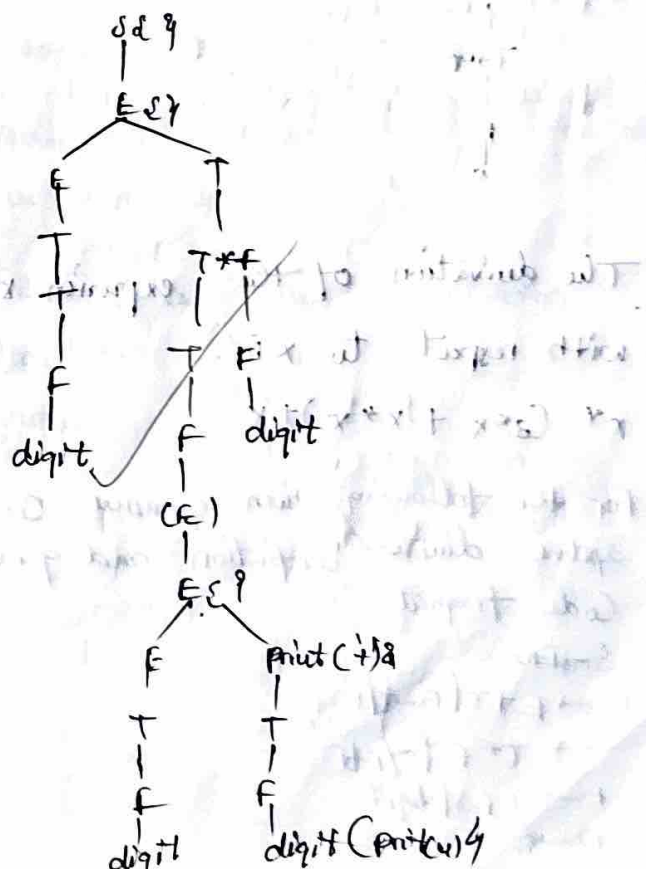


2. Construct a decorated parse tree according to the syntax directed definition for the following input stated (4+7.5*3)/2.

SDD for "Simple Desk Calculator":

1. S → E
2. E → E+T | E-T | T
3. T → T*F | T/F | F
4. F → (E) | digit

* The decorated parse tree:



③ Give a syntax directed difinition to differentiate expression formed by applying the arithmetic operator + and * to variable x ; exp: x*(3*x+x*x)

SDD for Differentiation of Expression

S→E
E→E+T | T
T→T*F | F
F→x | constant

The derivation of the expression x*(3*x+x*x) with respect to x is:-

x* (3*x + x*x)+x

④ For the following given grammer construct the syntax-directed definition and generate the code fragment

S→EN
E→E+T | E-T | T
T→T*F | T/F | F
F→(E) | digit
N→;

⑤ SDD for the given Grammer.

S→E {print(E.val)}
E→E+T {E.v = E1.v + T.v} | E.+LT
T→T*f {T.v = T1.val * f.val} | T/f {T.val = T.val / f.val} | f {T.val = f.val}
F→(E) {F.val = E.val} | digit
N→;

Step-1: Initializing the parsing stack.
Stack: $
Input: 2*3+4;

Step-2: Applying LR parsing action.

Step-3: Apply the LR parsing action for the couple of this

Step-4: Reduce using rule, S→EN action
Stack: $
Input: ;

Analytical Day-7

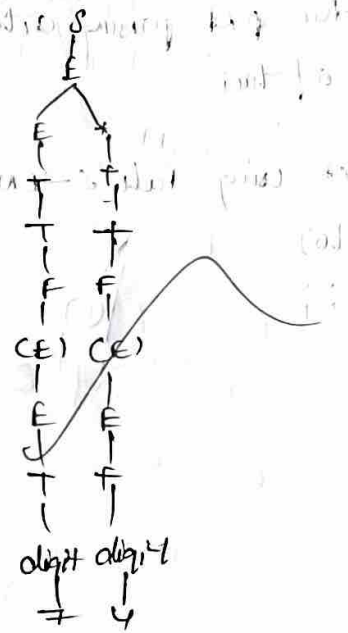
① Illustrate the SDD for simple desk calculator and showed expression: (7+4)*(8+6)n

S → En
E → E+T | E
T | F
T → T*F | T/F | F
F → (E) | digit

SDD for simple Desk Calculator:

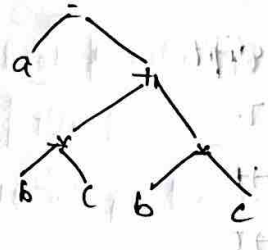
1. S → E { Print (E.val) }
2. E → E+T { E.val = E1.val + T.val } | T { E.val = T.val }
3. T → T*F { T.val = T1.val * F.val } | T/F { T.val = T1.val / F.val }
4. F → (E) { F.val = E.val }

Expression: (7+4)* (8+6)



② Construct the syntax tree for the expression
a = b*c + b*c

The syntax for given expression is:



③ Illustrate the bottom up evaluation for S attribute for the input 5*6+4

S → EN
E → E+T
E → E-T
E → T
T → T*F
T → T/F
T → F
F → E
E → digit

Given Grammer:

S → EN
E → E+T | E-T | T
T → T*F | T/F | F
F → E | digit

Bottom-up evaluation:

Input : 5*6+4

1. Shift "5" onto the stack.
2. Shift "*" onto the stack.
3. Shift "6" on the stack
4. Reduce Using T → digit
5. Reduce Using F → T.
6. Reduce Using T → F
7. Reduce Using E → T
8. Shift "+" onto the stack
9. Shift "4" on the shift.
10. Reduce Using T → digit
11. Reduce Using F → T.
12. Reduce Using S → E.

① Quadruples for expression
-(a*b)+(c+d)-(a+b/c+d)

② Let, $t_1 = a+b$
$t_2 = -t_1$
$t_3 = c+d$
$t_4 = t_2 * t_3$
$t_5 = t_1 + c$
$t_6 = t_4 - t_5$

Quadruple:

	Operator	Arg1	Arg2	Result
(0)	+	a	b	t_1
1	-	t_1		t_2
2	+	c	d	t_3
3	*	t_2	t_3	t_4
4	+	t_1	c	t_5
5	-	t_4	t_5	t_6

③ Construct the address code following erected if A<B and C<D then t=1 else t=0

④ If A<B and C<D
t=1

Use

t=0;

Three address code

(1) if (A<B) goto 3

(2) goto (4)

(3) if (c<0) goto 6

(4) t=0

(5) goto (7)

(6) t=1

(3) Generator the address code

```
C=0
do
{ if (a<b) then
    x++;
  else
    x--;
  C++;
} while (C<5)
```

(4) Three address code

1. C=0
2. if(a<b) goto (4)
3. goto (7)
4. T1(x+5)
5. x=T1
6. goto (9)
7. $T_2 = x-1$
8. x=12
9. T8=C+1
10. C=T3
11. if (C<5)
12. Goto (2)

Generate three address Code

```
int a[10], b[10], i, dp=0;
for (i=0; i<10; i++)
{
  dp += a[i]*b[i];
}
```

(4) 1. i=0
2. t_1 = i<10
3. if not t_1 goto 9
4. t_2 = i*4
5. t_3 = a[t_2]
6. t_4 = i*4
7. t_5 = b[t_4]
8. t_6 = t_3 * t_5
9. t_7 = dp + t_6
10. dp = t_7
11. i = i+1
12. goto 2.