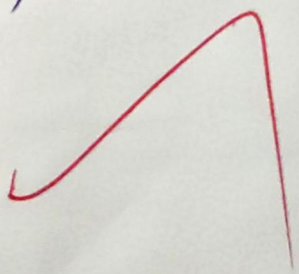
Analytical Day-1.

(1). Find a number of tokens in the following c statement is

```
int main()
{
/* Find maxm of a and b*/
int a=10, b=30;
if(a<b)
 return(b);
 else
 return(a);
 }
```

```
1   2  3 4
int main()
{5
/* Find maxm of a and b */
 6  7  8 9 10 11 12 13 14
int a=10, b=30;
 14 15 16 17 18
if(a<b) 19
 20 21 22 23 24
return(b);
else 25
 26 27 28 29 30
 return(a);
}31
```



∴ No. of token = 31

(2) Write a regular expression to denote set of all strings over $\{0,1\}^*$ containing substring 101.

Expression : $\wedge(?=.*101).*$

Explanation:

(i) '^' starting of string

(ii) (?=.*101) for checking if "101" exists anywhere in the string or not.

(iii) '*' matches zero or more.

(iv) $ asserts the end of string.

So, the regular expression ensures that the string contains the substring "101".

(3) Write a regular expression that have at least two consecutive 0's (00) 1's

Expression: $\wedge(.*00.*|1.*11.*)\$$

Explanation:

1. '^' starting of string.
2. (.*00*.|1.*11.*) group that matches consecutive 0 or 1
3. '|' logical or operator.

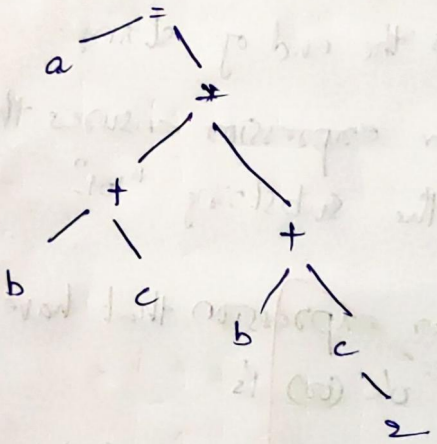
This expression matches any string that has atleast two characters.

④. How would you trace the program segment "a = (b+c)* (b+c) * 2" for all phases of compiler

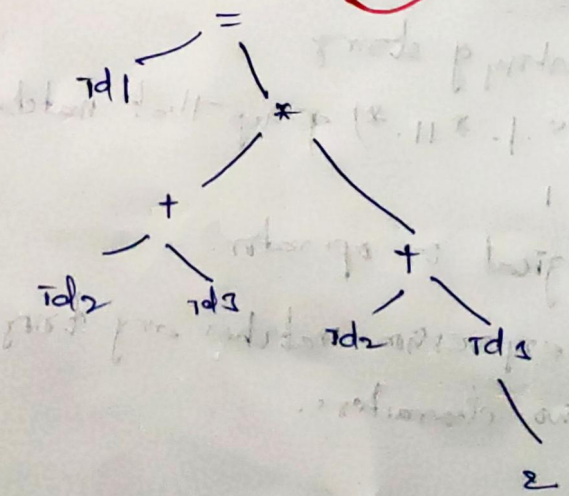
Lexical Analysis: 'Token'

"a", "=", "(", "b", "+", "c", ")", "*", "(", "b", "+", "c", ")", "*", "2"

Syntax Analysis: Parse tree



Semantic code Analysis:-



Intermediate Code generation:

$t_1 = b + c$

$t_2 = t_1 * t_1$

$a = t_2 << 1$

Code optimization:

The optimized generated code is transferred into machine code

Code Generation:

```
MOV  eax, b
ADD  eax, b
IMUL eax, eax
SHL  eax, 1
MOV  a, eax
```

Analytical Day-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 01: Eliminate left-recursion from grammar

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

$T \to ST'$

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

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

set-1

First(S) = {a, ↑, c}

First(T) = {a, ↑, c}

set-2

Follow(S) = {$,)}

Follow(T) = {, $,)}

step 1: Predictive parsing table

(i)

	a	↑	c	,	$
S	a	↑	c		

Parsing table for T:

	a	↑	c	,	)	$
T	a	↑	c			
T'					,	

(2) SLR

S → CC C → cC / d

step 01: Augment the grammar

S' → S

S → CC

C → cC / d.

step 2: Computer the closure and go to sets for item

LR(0) item:

1. $S' \rightarrow .S$
2. $S \rightarrow .CC$
3. $S \rightarrow .Cd$
4. $S \rightarrow .\phi$
5. $C \rightarrow .CC$
6. $C \rightarrow .\phi$

I0:

closure (I0)

1. $S' \rightarrow .S$

Go To (I0):

Go To (I0, S) = I1

1. $S' \rightarrow S.$

Go To (I1):

Go To (I1, C) = I2

I2: closure(I2)

1. $S \rightarrow .CC$
2. $S \rightarrow .Cd$
3. $S \rightarrow .\phi$
4. $C \rightarrow .CC$
5. $C \rightarrow .\phi$

Go To (I2):

Go To (I2, C) = I3

Go To (I2, C) = I4

Go To (I2, d) = I5

I0:

closure (I0):

1. S → .CC
2. C → .CC
3. C → .d

GO TO (I0):

- GO TO (I0, C) = I1
- GO TO (I0, S) = I2
- GO TO (I0, d) = I3

I1:

closure (I1):

S → C.C

C → .CC

C → .d

GO TO (I1):

- GO TO (I1, C) = I4
- GO TO (I1, d) = I5
- GO TO (I1, $) = Accept

I2:

closure (I2):

S → CC.

C → .CC

C → .d

GO TO (I2):

- GO TO (I2, C) = I7
- GO TO (I2, d) = I8
- GO TO (I2, $) = I9

I3:

closure (I3):

C → d.

I4:

closure (I4):

C → C.C

C → .CC

C → .d

GO TO (I4):

- GO TO (I4, C) = I10
- GO TO (I4, d) = I11
- GO TO (I4, $) = I12

I5:

closure (I5):

C → d.

I6:

closure (I6):

C → C.C

C → .CC

C → .d

GO TO (I6):

- GO TO (I6, C) = I13
- GO TO (I6, d) = I14
- GO TO (I6, $) = I15

I7:

closure (I7):

C → CC.

I8:

closure (I8):

C → d.

I9:

closure (I9):

S → CC.

I10:

closure (I10):

C → C.C

C → .CC

C → .d

GO TO (I10):

- GO TO (I10, C) = I16
- GO TO (I10, d) = I17
- GO TO (I10, $) = I18

I11:

closure (I11):

C → d.

I12:

closure (I12):

S → CC.

I13:

closure (I13):

C → C.C

C → .CC

C → .d

GO TO (I13):

- GO TO (I13, C) = I16
- GO TO (I13, d) = I17
- GO TO (I13, $) = I18

I14:

closure (I14):

C → d.

I15:

closure (I15):

S → CC.

I16:

closure (I16):

C → C.C

C → .CC

C → .d

GO TO (I16):

- GO TO (I16, C) = I18
- GO TO (I16, d) = I19
- GO TO (I16, $) = I20

I17:

closure (I17):

C → d.

I18:

closure (I18):

C → CC.

I19:

closure (I19):

C → d.

I20:

closure (I20):

S → CC.

	C	d	$	S	C
10				Shift 1	
11			Accept		
12	Shift 3	Shift 5			
13	Shift 3	Shift 5		Shift 4	Shift 6
14	"	"		"	"
15				"	"
16			Reduce 2		
17					
18	"	"			

3A) grammar

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

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

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Step 01: first step for non-terminal

$FIRST(A) = \{\varepsilon\}$

$FIRST(B) = \{\varepsilon\}$

$FIRST(S) = \{a, b\}$

Not LL(1)

4A) grammar

$E \rightarrow 2E2$

$E \rightarrow 1E1$

$E \rightarrow 4$

Passing input string 3242$.

Step1: Initialization

Stack = $

Input: 3242$

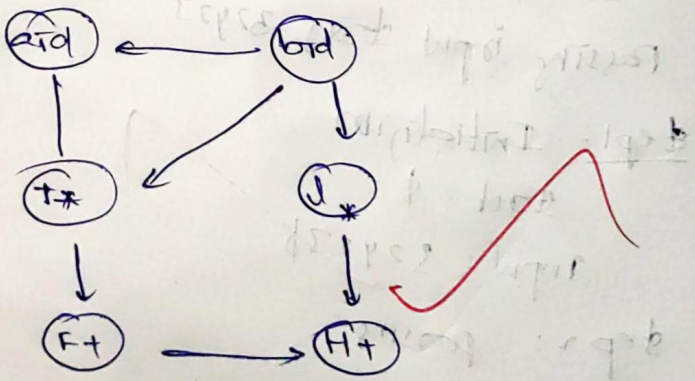
Step 2: parsing

Stack	Input	Action
$	3242$	Shift 1, Push 1
$3	242$	shift 2, Push 2
$32	42$	Reduce E → 4
		Pop 4

$E	4*2$	shift 4, Push 4
$E4	*2$	shift *, Push *
$E4*	2$	Reduce E→4 Pop 4
$E*	2$	Reduce E→2E2 Pop E
$E	$	shift 3, Push 3
$E3	$	shift 2, Push 2
$E32$	—	Reduce E→3E2
$E	—	Accept

22/07/23

Operator procedure flow graph.



aid → T* → F+ → H+ ⇒ a$ ⇒ 1
bid → T* → F+ → H+ ⇒ b$ ⇒ 1
T* → F+ → H+ ⇒ T$ ⇒ 2
F+ → H+ ⇒ F$ ⇒ 1
bid → I* → H+ ⇒ b$ ⇒ 2

22/2/23 Analytical Day-1

① Syntax Directed-translation

$2^0(4+5)$

$E \rightarrow E+T / E-T / T$

$T \rightarrow T*F / T / F / f$

$f \rightarrow (E) / num$

E → Represent expression

T → Term

F → Factor.

SDT Action:-

$E \rightarrow E+T$ { 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); }

$T \rightarrow T*F$ { right = pop; left = pop; emit(left + right + '*'); push(left + Right + '*'); }

$/ F$ { result = pop(); push (result); }

$F \rightarrow (E)$ { Result = pop(); push (result); }

/num { emit (num); push (num); }

(2). SSD 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. F → (E) { E.val = E.val }
   | num { E.val = num.val }

(3). Grammar G

E' → E

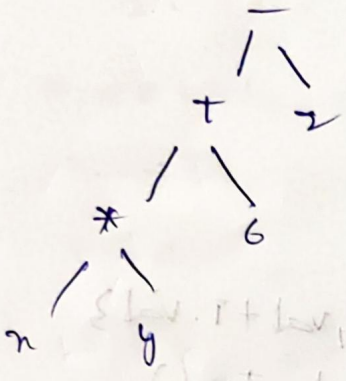
E → E+n | n

Parsing Action.

Stack	Input	Action.
$	n+n$	shift n
$n	+n$	shift '+'
$n+	n$	shift 'n'
$n+n	$	E → n.
$E	$	Reduce E → E+n
$E1	$	Reduce S → E

④. Syntax tree

x * y + 6 - z



⑤. Syntax directed

S → EN

E → E + T | E - T | T

T → T * F | T F | F

F → (E) | digit

input

5 * 3 + 2

Day-4 – Analytical :

Translate the expression $-(a+b)*(c+d)+(a+b+c)$ into

(i) quadruples

(ii) Triples

(iii) Indirect triples

Given $-(a+b)*(c+d)+(a+b+c)$

$t_1 = a+b$

$t_2 = 0-t_1$

$t_3 = c+d$

$t_4 = t_2 * t_3$

$t_5 = t_1 + c$

$t_6 = t_4 + t_5$

Quadruples

	op	arg1	arg2	Result
(0)	+	a	b	t_1
(1)	−	0	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

Triples

	op	arg1	arg2
(0)	+	a	b
(1)	−	(0)	−
(2)	+	c	d
(3)	*	(0)	(2)
(4)	+	(0)	c
(5)	+	(3)	(4)

Indirect triple

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)
105	(5)

1 (2). Translate the expression $a^* -(b+c)$

a) Quadruples

b) Postfix Notation

c) Three-address code.

Quadruple:

$a^* -(b+c)$

$t_1 = b+c$

$t_2 = -t_1$

$t_3 = a+a$

$t_4 = t_3 * t_2$

	op	arg1	arg2	result
(0)	+	b	c	t_1
(1)	−	0	1	t_2
(2)	+	a	a	t_3
(3)	*	t_3	t_2	t_4

b) **Postfix Notation:**

$a^* -(b+c) \Rightarrow abc+-*$

c) **Three Address code:**

$t_1 = b+c$

$t_2 = -t_1$

$t_3 = a * t_2$

(3) $t_1 = c * d$

$t_2 = a + b$

$q = t_2 - t_1$

(4) $t_1 = a < b$

$t_2 = c < d$

$t_3 = d < c$

$t_4 = t_1 \text{ or } t_2$

$t_5 = t_4 \text{ and } t_3$

Back Patching :-

if t_4 goto 4

if not t_5 goto 12

4 :

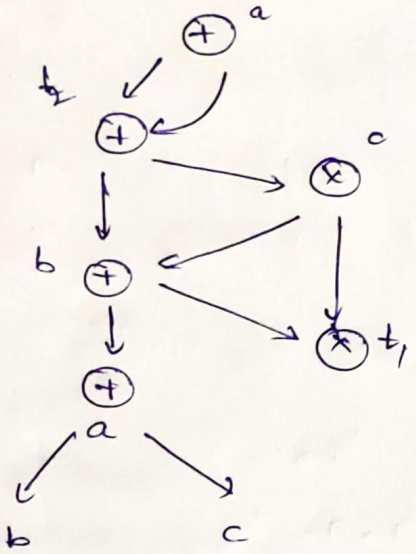
goto 4 :

t_2

12 : ;

Analytical Day-5

① Directed cyclic graph.



②

PROD = 0

$T_1 = 4 * 1$

(1) PROD = 0

(2) I = 1

(3) T_2 = addr(A) - 4 } Block B.

(4) T_4 = addr(B) - 4

(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 T_6 (7) } Block B

(7). Code :-

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

(8).

PROD = 0
I = 1
} Block 1.

T_2 = addr(A) → 4

T_4 = addr(B) → 4

T_1 = 4 × I
I = I + 1

T_3 = 2[T_1]

T_5 = T_4[T_1]

T_6 = T_3 × T_5.

} Block-2

PROD = PROD + T_6

Analytical Day-6

① Identify left-recursion variables:

Step ①:-

The left recursive variable in the grammar are A & B, as they appear as the leftmost symbol in some of their production rules

Step ②: Eliminate left recursion:

$A \rightarrow aA'$

$A' \rightarrow BaA'/E$

$B \rightarrow bB'$

$B' \rightarrow eB'/\epsilon$

② left recursion is eliminated by converting the grammar into a right recursive grammar

$A \rightarrow BA'$

$A' \rightarrow AdA'/\epsilon$

Explanation:-

For nonterminal A, we create a new non-terminal A' to handle the recursion. Now, A can produce strings starting with B. A' can produce strings starting with Ad & ε. This way, we avoid left recursion for A.

(11).

a) $S \rightarrow bSSaS / bSSaSb / bSb / a$.

step (1) :- Identify common prefixes

common prefix in this grammar is 'bSS'

step (2) :- Perform left-factoring:

factor out common prefix "bSS" & create new non terminals to handle the different suffixes

$S \rightarrow bSSS'$

$S' \rightarrow aaS / aSb / b.$

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

(1). Common prefix is "aS"

$S \rightarrow aSS'S'$

$S' \rightarrow bS / aSb / \varepsilon$

(21).

$FIRST(S) = \{;, a\}$

$FIRST(E) = \{b\}$

Follow sets :-

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

$Follow(E) = \{;, e, \$\}$

Analytical Day-7:

① .

```
Procedure E()
{
    T();
    E'();
}
```

$E \rightarrow TE'$

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

$E \rightarrow +TE'$

```
Procedure T()
{ F();
  T'();
}
```

$T \rightarrow +T'$

```
Procedure T'()
{ if input symbol/c "*" then
  advance();
    F();
    T'();
}
```

$T' \rightarrow *FT'$

Procedure F()

{ if input symbol = "id" then
advance();
else if input symbol = 'c' then
advance();
E();
If input symbol = ')"
advance();
else error();
}

$T' \rightarrow *FT'$

$F \rightarrow (E)$

(2).

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

Step 01:

$ (a, (a,a)) $

we insert procedure operator b/w symbol

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

step 02:-

we can scan and parse the string as

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

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

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

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

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

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

$ <(S,S)> $

$ <(L,S)> $

$ <(L)> $

$ <S> $

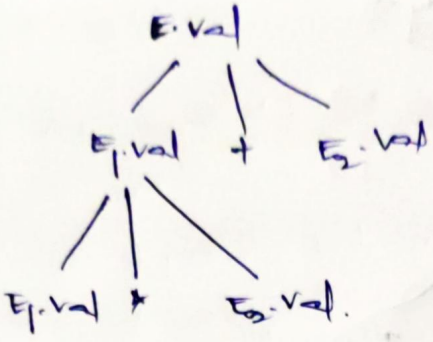
$ $

(3)

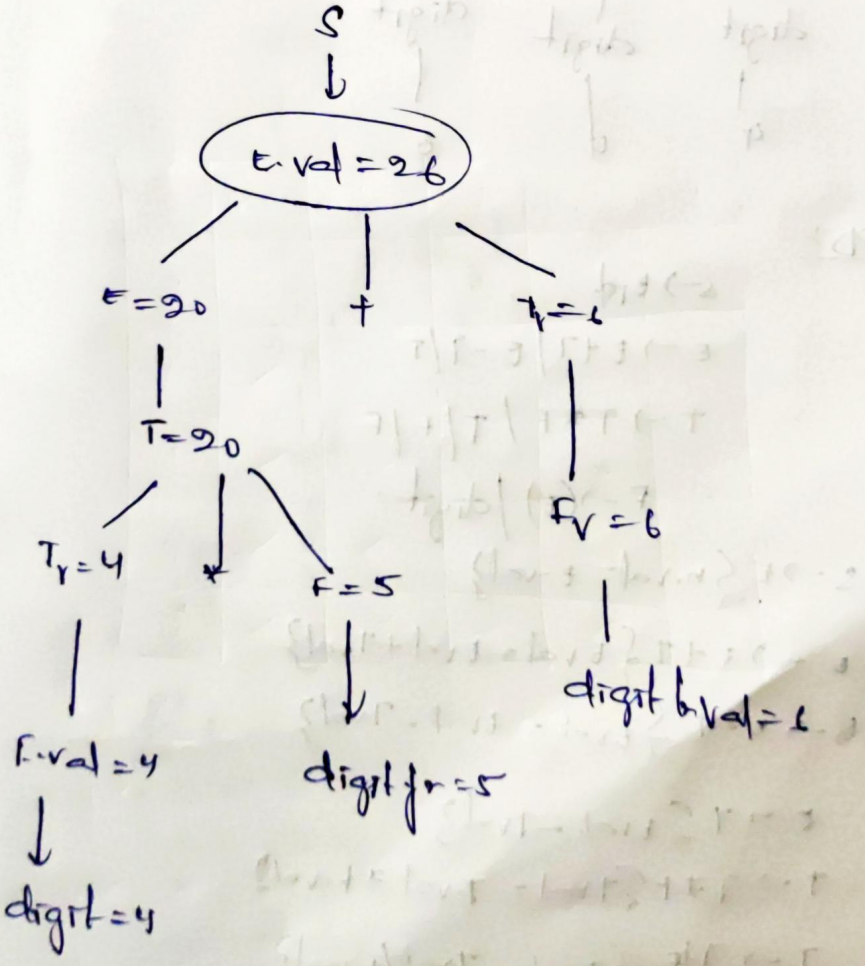
$E \rightarrow (E_1 + E_2)$

$E \rightarrow E_1 * E_2$

Production	Semantic Rules
$E \rightarrow E_1 + E_2$	$E.val \rightarrow E_1.val + E_2.val$
$E \rightarrow E_1 * E_2$	$E.val \rightarrow E_1.val + E_2.val$

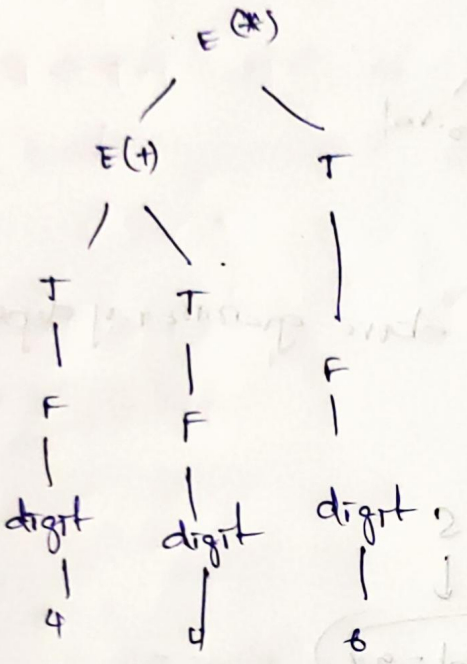


(7) Design for above grammar of dependency graph.



Analytical Day-8

① Parse Tree:



SDD:

$S \to E$

$E \to E+T \mid E-T \mid T$

$T \to T*F \mid T/F \mid F$

$F \to (E) \mid digit$

$S \to E \{n.val = E.val\}$

$E \to E+T \{E.val = E.val + T.val\}$

$E \to E-T \{E.val = E.val - T.val\}$

$E \to T \{E.val = T.val\}$

$T \to T*F \{T.val = T.val * F.val\}$

$T \to T/F \{T.val = T.val / F.val\}$

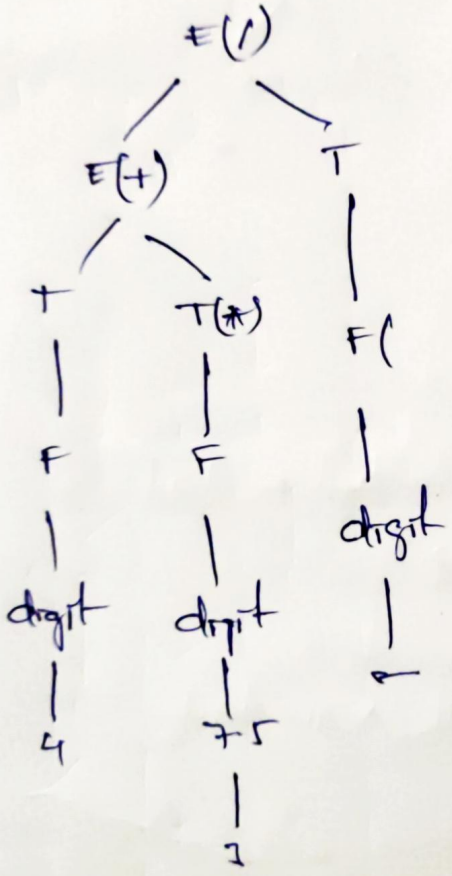
$T \to F \{T.val = F.val\}$

F → (E) {F.val = E.val}

F → digit {F.val = digit.val}

(2)

Parse Tree



Evaluate (7.5 * 3) = 22.5

Evaluate (4 + 22.5) = 21.5

Evaluate (21.5 / 2) = 13.25

result = 13.25

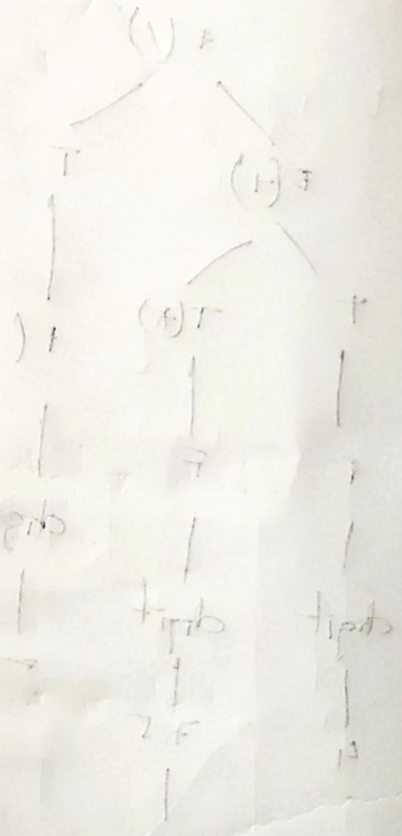
(5). $S \to E$

$E \to E+T$ $\quad F \to x$

$E \to T$ $\quad (E)$

$T \to T*F$

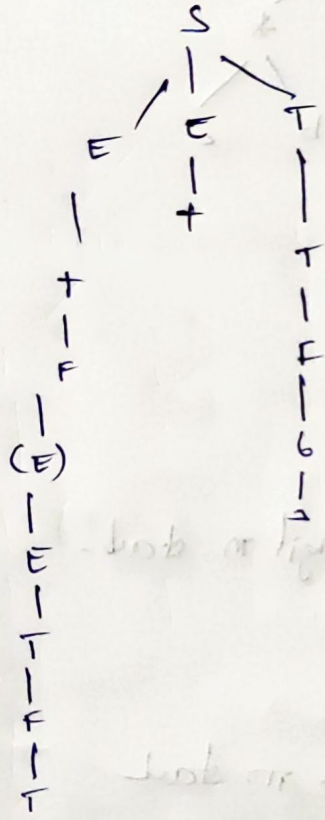
$/F$



Analytics Day-9.

① . Parse tree:-

(7+4)*(1+6)



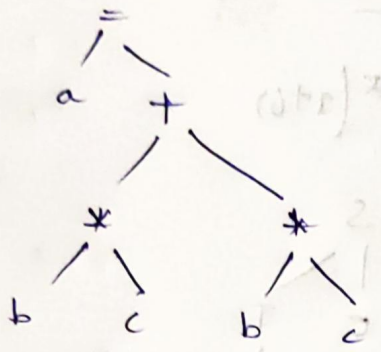
$S \rightarrow E\{n.val = E.val\}$

$E \rightarrow E_1\{E.val = E_1.val\} + T\{E.val = E.val + T.val\} / + \{E.val = T.val\}$

$T \rightarrow T_1\{T.val = T_1.val\} * F\{T.val = T_1.val + F.val\} / F\{T.val = F.val\}$

$F \rightarrow (E\{E.val = E.val\}) / digit\{F.val = digit.val\}$

(2). $a = b*c + b*c$



(1). Grammar:-

$S \rightarrow EN$

Input: 5*6+4

Step 01: shift the digit in stack

Stack: [5]

Input: *6+4

Step 02: shift the * in stack

stack: [5, *]

Input: 6+4

Step 03: shift the 6 number in stack

stack: [5, *, F]

Input: +4

$T \rightarrow F$ ($F \rightarrow E \rightarrow$ dig)

Step 4: Redu E → T

Stack: [E]

Input: +4

Step 5: [E, +]

4

Step 06:

F → E → digit

[E, +, F]

Step 07: Redu T → F

[E, +, +]

(4) Grammer:-

1. Input 5 * 7

Synth = E.val

E1.synth = T.val

T1.synth = F.val

F3.synth = F.val

F1.val = 5, F1.Ord = 5.

F1.val = 7, F2.Ord = 7

T2.val = 7, T2.Ord = 7

$T_1.val = 5$, $T_1.code = 5$

$E_1.val = 5$, $E_1.code = 5$

$E.val = 5$, $E.code = 5$

o/p = 5 ✗

2. Input 8 + 4

$S.inh = E.val$

$E_1.inh = T.val$

$T_1.inh = F_1.val$

$T_2.inh = F_2.val$

$F_1.val = 8$, $F_1.code = 8$

$F_2.val = 4$, $F_2.code = 4$

$T_2.val = 4$,

$E_1.val = 8$, $T_2.code = 4$

$E.val = 8$, $E.code = 8$

o/p : 84 ✗

Analytical Day-10

① $t_1 = a+b$

$t_2 = -t_1$

$t_3 = c+d$

$t_4 = t_2 * t_3$

$t_5 = t_1+c$

$t_6 = t_4 - t_5$

Quadruples:

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

② If A<B and C<D

t = 1

else

t = 0;

These address code

(5) if (A<B) goto 3

(7) goto 4

(3) If (c<0) goto 6

(4) t=0

(5) goto (7)

(6) t=1

(3).

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

These address code

(1) c=0

(2) If (a<b) goto 4

(3). goto 4

(4). $T_1 = (x+5)$

(5). x=t

(6) goto (9)

(7). $T_2 = x-1$

(8). x=t2

(9). $T_8 = (t+1)$

(10) $c = T_3$

(11). If (c<5)

(12). Goto (2)

(A). Generate three address code.

```
int a[40], b[10], i, dp=0;
for (i=0; i<10; i++)
{
    dp += a[i]+b[i];
}
```

(A). (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.