index 2in1

Name: Y. Harsha Vardhan Reddy

Sub: Compiler Design for lexical Analysis

STD: _____ SEC: _____ Roll No. _____

School/College: Sareetha School of Engineering

S.No.	Date	Title	marks per	Teacher's Sign/ Remarks
1.	4/8/23	Analytical - Day1	10	
2.	4/8/23	Analytical - Day2	10	
3.	4/8/23	Analytical - Day3	10	
4.	4/8/23	Analytical - Day4	10	
5.	4/8/23	Analytical Day5	10	
6.	4/8/23	Analytical Day6	10	
7.	4/8/23	Analytical Day7	10	
8.	4/8/23	Analytical Day8	10	
9.	4/8/23	Analytical Day9	10	
10.	4/8/23	Analytical Day10	10	
11.	4/8/23	Analytical Day11	10	

1920114941

Analytical Day-1

1. Find a no. of token in the following C statement

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

:NO of Token = 31

2. write a regular expression to detect set of all strings over {0,1} containing substring 101.

Expression (0+1)*101(0+1)*

192011494

(A) Explanation

(i) '^' Starting of String

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

(iii) '.*' matches zero or more

(iv) '$' match the end of string.

string contains the substring '101'

(3) write a regular expression that have at least two consecutive 0's or 1's.

Expression ^(.*00.*|.*11.*)$

Explanation

(1) '^' Starting of string

(2) (.*00.*|.*11.*) group that matches consecutive 0 or 1.

(3) '|' logical or operator.

This Expression matches any string that contain two characters

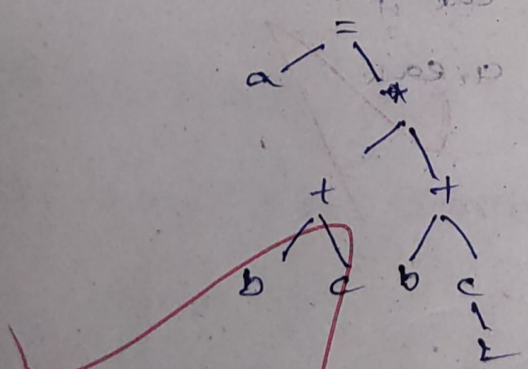
19201149AC

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

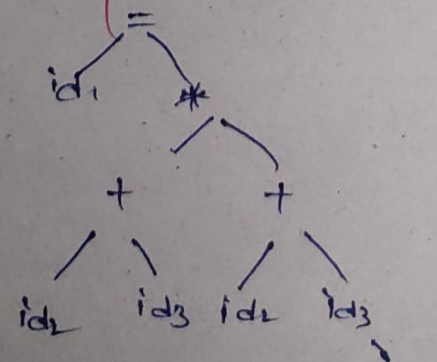
(1) Lexical Analysis → Tokens

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

Syntax Analysis → Parse tree.



Semantic code Analysis



Intermediate code generation

$t_1 = b + c$

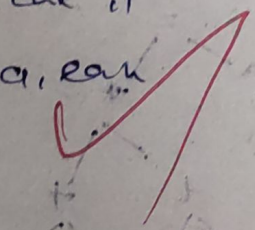
$t_2 = t_1 * t_1$

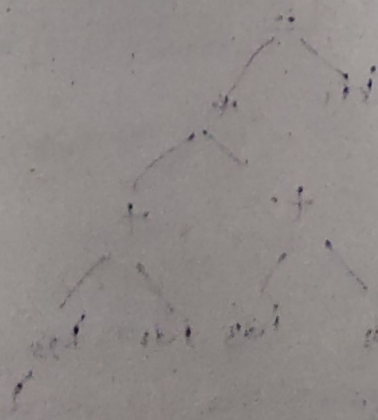
Code optimization

The optimized generated code is transferred into machine code.

Code generator:-

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





Analytical Day-2

① predictive parser

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

$T \rightarrow T, S \mid S$

(A) **Step 1:-** Eliminate left recursion from grammar.

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

$T \rightarrow ST'$

$T' \rightarrow ST'$

step 2:- create the first and follow set for each non-terminal.

Step ①

first (S) = {a, ↑, (}

First (T) = {a, ↑, (}

Step ②

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

follow(T) = {), $}

step 3 predictive parsing table

	a	↑	(	,	$
S	a	↑	(		
T	a	↑	(		
T'					

(2) SLR

$S \to SCC \quad C \to CC|d$

(A) Step 1:- Augment the grammar.

$S' \to S$

$S \to CC$

$C \to CC|d$

Step 2:- Compute the closure to go to sets for the.

LR(0) items:

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

I0:- closure (I0)

1. $S' \to .S$

Goto (I0);

GOTO (I0,I1) = I2

Clause (12)

S → .ec

S → .cc

S → .d

C → .cc

5. C → d.

Clause (13):

S → cc

C → .cc

C → d

GOTO (13, C) = 10

(34) Grammar

S → AaAb | BbBa

A → ε

B → ε is LL(1) or not?

Step 1: Find step for Non-terminal.

First (A) = {ε}

First (B) = {ε}

First (S) = {a, b}

Not LL(1).

(44)

Grammar

$E \rightarrow 2E_2$

$E \rightarrow 3E_3$

$E \rightarrow 4$

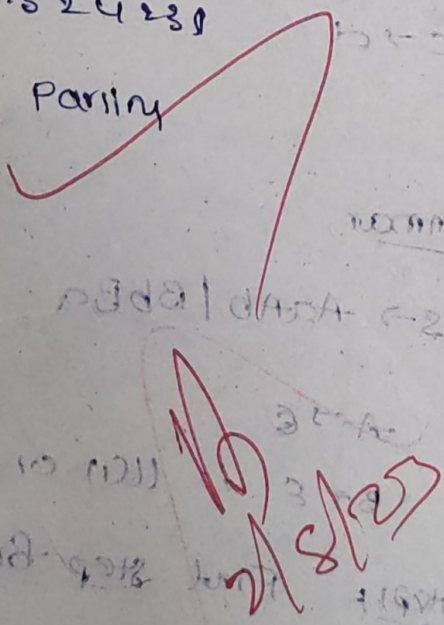
Parsing input string 32423

Step 1: initialization

Stack = $

input : 32423$

Step 2: Parsing



Analytical Day 3 1920114941

1. Syntax Directed translation

2*(4+5)

E → E+T / E - T / T

T → T*F / T / F / F

E → Represent expression

F → (E) / num.

T → Term

F → factor.

SDT Actions:

E → E+T {right = POP(); left POP(); ert

T {result = POP(); Push (result());

T → T*F {right : POP · left : POP · ert Put

+right + ('*'); ·

F → (E) {result = POP(), Push (result).

(2) SSD Scheme.

3*5+6*3

E → E+T / E → T / T

T → T*F / T / F / F

F → (E) / num

Semantic rules

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

   $E_1 - T$ { E.val = E_1.val - T.val }

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

   num { E.val = num.val }

(3) Grammar G

$E' \rightarrow E$

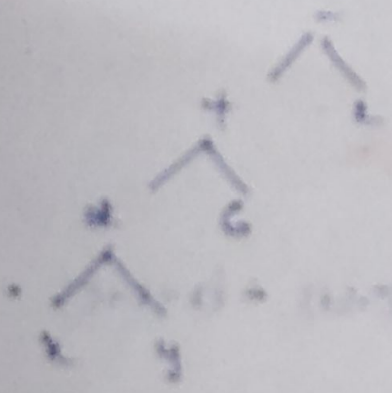
$E \rightarrow E + n / n$

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
$E'	$	Reduce E' → 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) | id

Ro
24/8/23

Analytical Day-4

1. Translate the expression (a+b)*(c+d)+(a+b+c).

(i) quadruples

(ii) Triples

(iii) Indirect triples.

(A) Given (a+b) * (c+d) + (a+b+c).

$t_1 = a+b$

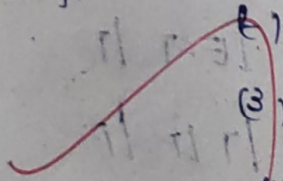
$t_2 = a+c$

$t_3 = c+d$

$t_4 = t_2 * t_3$

$t_5 = t_1 + c$

$t_6 = t_4 + t_5$



Quadruples

	op	arg1	arg2	
(0)	+	a	b	t_1
(1)	+	a	c	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)	+	a	c
(2)	+	c	d
(3)	*	(0)	(2)
(4)	+	(0)	c
(5)	+	(3)	(4)

Indirect tuples

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

Translate the Expression

(a) Quadruples

(b) Triples

(c) Three-address code

(A) Quadruples

$a^* -(b+c)$

$t_1 = b+c$

$t_2 = -c$

$t_3 = a+a$

$t_4 = t_3 * t_2$

	op	arg1	arg2	result
(0)	+	b	c	t_1
(1)	-	c		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)

$t_1 = c + d$

$t_2 = a + b$

$\phi = t_2 - t_1$

(4)

$t_1 = a < b$

$t_2 = a < b$

$t_3 = d < e$

$t_4 = t_1$ or t_2

$t_5 = t_4$ and t_3

Back Patching

if t_4 goto 4

if not t_5 goto t_2

4:

goto

t_2

$t_3 : 1$

Analytical Day-5

1920114941

1. DAG for following three address code.

$a = b + c$

$t_1 = a \times a$

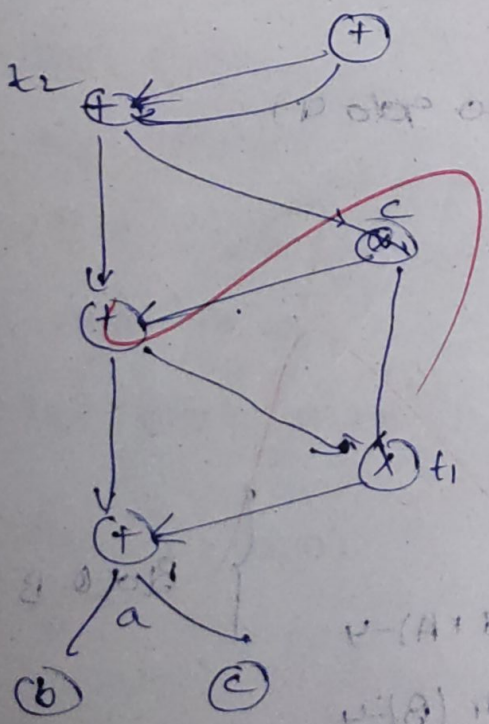
$b = t_1 + a$

$c = t_1 \times b$

$t_2 = c + b$

$a = t_2 + b$

A. Directed cyclic graphs



(2) Basic Block

① PROD = 0

② I = 1

③ I_2 = addr(A) − 4

④ T_1 = 4×1

⑤ T_1 = 4×1

⑥ T_3 = T_2[T_1].

⑦ T_6 = T_3 × T_5

⑧ PROD = PROD + T_6

⑨ I = I + 1

⑩ If I<20 goto ④

(4)

PROD = 0

T_1 = 4×1

(11) PROD = G

② I = 1

③ T_2 = addr(A) − 4

④ T_4 = addr(B) − 4

⑤ T_1 = 4×1

⑥ T_2 = T_2[T_1]

⑦ T_3 = T_4[T_1]

⑧ T_6 = T_3 × T_5

Block B

Block B2

③ $w = (A+B) + (A+C) + (A+C)$

$t_1 = A+C$

$t_2 = A+C$

$t_3 = A+B$

$t_4 = t_1 + t_2 + t_3$

$w = t_4$

④ Code:

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

⑤ Basic Block and flow graph

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

(Ans)

PROD = 0
I = 1
} Block 1

T_2 = addr (A1) - 4
T_4 = addr (B1) - 4
T_1 = 4 x I
T_5 = T_4(T_1)
T_6 = T_3 x T_5
} Block 2

PROD = PROD + T_6

21/8/25

Analytical Day-6

(2) Consider the following grammar and eliminate left recursion.

$A \rightarrow ABd / Aa / a$

$B \rightarrow Bc / b$

(A) **Step ①:** Identify left-recursive variable.

The left-recursive variable in the grammar are: A & B as they appear after leftmost symbol in some of their productive rules.

Step ② Eliminate left recursion.

$A \rightarrow \alpha A'$

$A' \rightarrow \beta A' / \epsilon$

$B \rightarrow bB'$

$B' \rightarrow cB' / \epsilon$

(iii) Consider the following grammar & elimⁿ- -nate left recursion

$A \rightarrow AA\alpha / \beta$

(A) left recursion is eliminated by converting the grammar into a right recursive grammar.

$$A \rightarrow BA'$$
$$A' \rightarrow \alpha A' / \epsilon$$

Explanation:

for non-terminal A, we create a new terminal A', to handle the recursion. Now A can produce string with BA' (or) more by

(iii) Do left factoring in the following grammar

(a) $S \rightarrow bSSaS / bSSa / bSb / a$

(b) $S \rightarrow aSSbS / aSaSb$

(A) Step 1: identify common prefixes

Common prefix in the grammar is bA

Step 2: Perform left factoring.

factor out common prefix.

1920114944

(b) (i) common prefix is 'a'

S → ass'

S' → bs | asbb

(ii) Check whether the following grammar is LL(1) or not.

S → iE | s | iE | sEsa

E → b

(A) FIRST (S) = {i, a}

FIRST (E) = {b}

FOLLOW (S) = {$, e}

FOLLOW (E) = {t}

28/25

DAY -1

-Analytical

19/2/14

1. Construct a recursive descent parser for the following grammar string

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

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

$E \rightarrow TE'$

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

$E \rightarrow +TE'$

```
Procedure T()
{
   F();
   T();
}
```

$T \rightarrow *T'$

19201149

```
procedure T()
{
  if input symbol = '('
    advance();
    F();
    T();
}
procedure F()
{
  if input symbol = id
    advance()
  else if input symbol = '('
    advance();
    E();
```

② construct the following grammar for operator precedence parser

	a	(	)	,	$
a	·	>	>	>	>
(	<	≐	≐	·	·
)	·	>	>	>	>
,	<	·	>	>	>
$	<	<	<	<	

Step 1:

$ (a, (a,a)) $

we insert precedence operators between symbols.

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

Step 2:

we can scan and parse the string

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

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

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

$ < S, < (S) > > $

$ < (S, S) > $

$ < (S) > $

$ < S > $

$ $

(3) Determine the code parse graph for the following trees

$E \rightarrow (E_1 + E_2)$

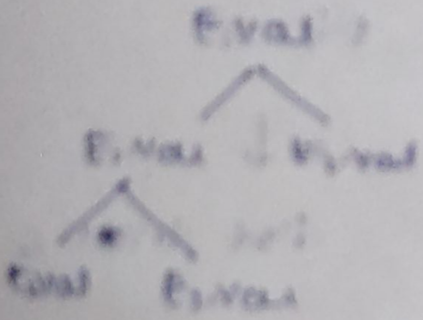
$E \rightarrow E_1 * E_2$

Mounika

Production   Semantic Rules

$E \rightarrow E_1 + E_2$ $E.val \rightarrow E_1.val + E_2.val$

$E \rightarrow E_1 * E_2$ $E.val \rightarrow E_1.val * E_2.val$



Q) Design a dependency graph for the following grammar

$S \rightarrow T$ list

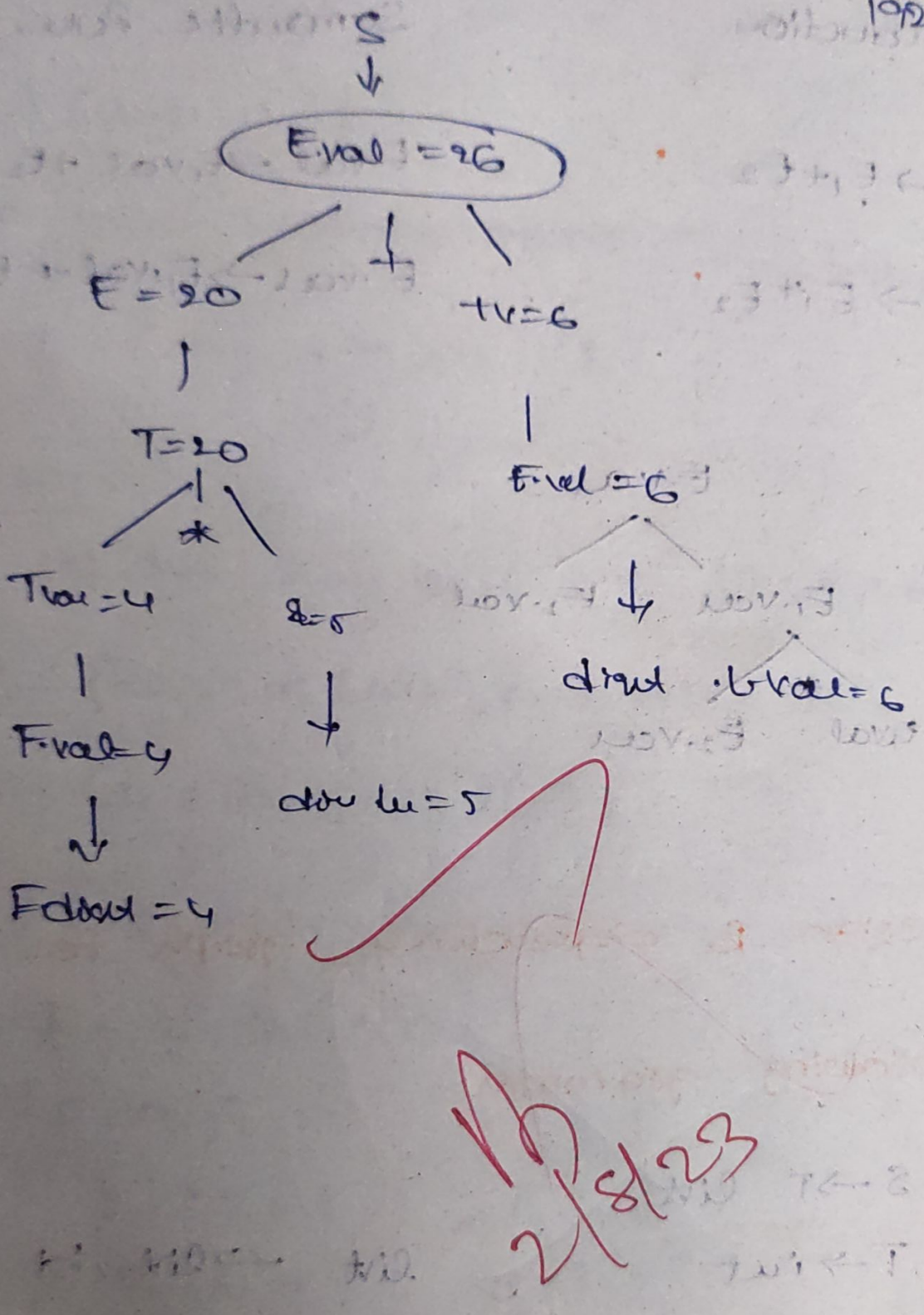
$T \rightarrow$ int   list $\rightarrow$ list , id

$T \rightarrow$ float   list $\rightarrow$ id

$T \rightarrow$ char

$T \rightarrow$ double

Design for above grammar of dependency graph.


S
E.val=26
E=20
+
T.val=6
T=20
F.val=6
*
T.val=4
digit.lval=6
F.val=4
digit.lval=5
Fdigit=4

2/8/23

Analytical Day 8

PRAVEEN

① illustrate the SDD for simple desk Calculator and show annotated Parse tree for the expression (6+4) * (4+3) n.

S → En

E → E+T | E-T | T

T → T*F | T/F | F

F → (E) digit

(A) S → E{n.val = E.val}

E → E+T{E.val = E.val + T.val}

E → E-T{E.val = E.val - T.val}

E → T{E.val = T.val}

T → T* F{T.val = T.val * F.val}

T → T/F{T.val = T.val / F.val}

Now let's construct the annotated Parse tree for the expression (6+4)* (4+3);

Token Stream:

(6+4)*(4+3)

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

(A) Lexical Analysis

Token Stream:

(, 4, +, 7.5, *, 3, /, 2,).

Steps: Build the Parse Tree using the grammar Rule and animate the tree's attribute value:

We'll start with the root of the parse tree, representing the start symbol S and then apply the grammar rules to construct the tree.

digit $\rightarrow$ value 4

digit $\rightarrow$ value 7.5

F.1 $\rightarrow$ F.val = F.val = 3

3. Give a Syntax directed definition to differentiate expressions formed by applying the arithmetic operators + and * to the variable x and constants, express

x * (3 * x + x * x)

The SDD is as follows,

S → E { S.derivate = E.derivate }

E → E + T { E.derivate = E1.derivate + T.derivate }

E → T * F { T.derivate = T1.derivate }

F → (E) { F.derivate = E.derivate }

→ Here x represents the variable and 'c' represents a constant value.

→ Now let's apply the SDD to the given expression x * (3 * x + x * x).

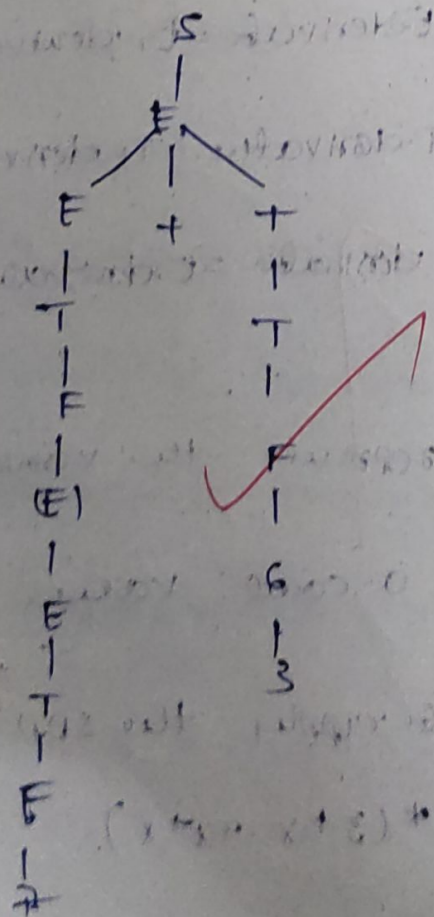
Analytical Day 10

1. SDD for simple desk parse tree (7+6)*3.

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

(A) Parse tree

(7+4)* (6+6)



S → E { n.val = E.val }

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

Problem

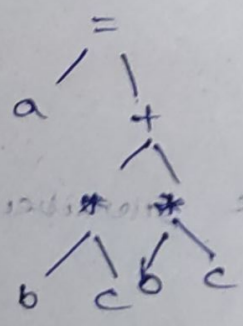
$T \rightarrow T_1 \{T.val = T.val\} * F \{T.val = T_1.val * F.val\}$

$F \rightarrow \{E\ \{E.val = E.val\}\} \mid digit\ \{F.val = digit.val\}$

2. Construct the Syntax tree for Expression

$a = b*c + b*c$

(A) $a = b*c + b*c$



3. Illustrate the Bottom up Evaluation for Attributes.

$5*6+4$

$S \rightarrow EN$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow T/F$

$T \rightarrow F$

$F \rightarrow E$

$F \rightarrow digit$

19201149

(A) Grammar

$S \rightarrow EN$

input : 5*6+4

Step1: Shift the digit in stack

stack : [5]

input : *6+4

Step2: Shift the * in stack.

stack : [5, *]

input : 6+4

Step 3: Shift the number in stack

stack : [5, *, 6]

input : +4

Step4 :- Reduce $\rightarrow T$

stack : [E]

input [+4]

Step5: [E, +]

Step6: $F \rightarrow$ digit

[E, +, F]

Step7: Reduce $T \rightarrow F$

[E + T]

Step8 Reduce $E \rightarrow E+T$

(E)

Analytical Day - 10

1. Construct Quadruples for the expression

(a+b)+(c+d) - (a+b+c+d).

(A) To construct quadruple for the expression we can follow the three - address code representation.

1. Assign the product of a and b to a temporary variable t_1.
2. Assign the sum of c and d to a temporary variable t_2.
3. Assign the sum of a, b, c and d to a temporary variable t_3.
4. Compute the negation of t_1 and store it to a temporary variable t_4.
5. Compute the sum of t_4 and t_2.

Let write them as a series of quadruples

(1) $t_1 = a * b$

(2) $t_2 = c + d$

(3) $t_3 = a + b + c + d$

(4) $t_4 = -t_1$

(5) $t_5 = t_4 + t_2$

Result $= t_5 - t_3$

2. Construct three Address code for the following expression. if A<B and C<D then t=1 else t=0.

(A) To construct three Address code for the given expression we break it down into smaller parts.

① (t1) t1 = A<B

② (t2) t2 = C<D

③ (t3) if t1 goto L1

goto L2

goto L3

* L1: t = 0

* L3:

in the TAC representation we assume that the expression A<B and C<D are Boolean

(3) Generate three Address code for the following code:

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

19201A04L

To generate the three - Address code for the given code, we'll use temporary variables to hold the intermediate value.

1. i = 0
2. dp = 0
3. L1:
4. if i >= 10 goto L2
5. t1 = a[i]
6. t2 = b[i]
7. t3 = t1 * t2
8. dp = dp + t3
9. i = i + 1
10. goto L1
11. L2.

In this TAC representation, we use temporary variables to hold the values of a[i] and b[i] during each iteration of the loop.

04/8/23

Day - 11

Analysis

(1) Three Address code for the given code

```
t1 = 0
i = 0
L1: i
    t2 = i * 4
    t3 = a[t2]
    t4 = b[t2]
    t5 = t3 * t4
    dp = dp + t5
```

(2) Using the generated Three Address code, analyze the efficiency of the code in terms of the number of instructions.

(A) Efficiency Analysis

The given code contains a loop that iterates from i=0 to i=9.

Temporary variables

The code uses four variables t1, t2, t3 and t4.

Prabhakar

t1 is used to initialize the loop counter.

and t2, t3 and t4 are used for intermediate values in memory.

Possible optimizations:-

loop unrolling:-

if the target architecture supports, you can unroll the loop, reduce the number of loop iterations.

(2) Strength Reduction:

instead of using multiplication operation (t2 = i * 4), you can use a bit shift operation to calculate the offset, which is known to multiply by 4.

(2) Consider the given code and the corresponding Three Address code. Discuss the role of 'i' variable and the 'dp' variable in the loop and how they are used to control

(4) The 'i' variable.

The 'i' variable serves as the loop counter for the given code. It is initialized to 0 before the loop starts.

* During each iteration of the loop, the value of 'i' is used to calculate the offset for array access.

1. The dp variable

The dp variable is used to store the accumulated result of the dot product calculation.

After the loop completes the dp variable holds the final value of the dot product.

(5) The given code calculates the dot product of two arrays 'a' and 'b'. Assuming 'a' and 'b' are large arrays

192044241

Suggest strategies or modifications in the three-address code that can handle

(A) memory usage.

The memory usage of the given code depends on the size of arrays 'a', 'b' and the number of temporary variables

* The prime concern is the size of 'a' and 'b'. The code assumes that both arrays have at least 10 elements.

Potential Risk of Buffer overflows.

The given code does not include any checks to ensure that the loop does not access elements beyond the bounds.

5/8/25