20/07/2023.

# Analytical Questions.

Bharath
SundaraRaman.J
192011038

1. Find the no. of tokens in the following.

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

Tokens :-

'int': Keyword token.

'main': Identifier token.

'(' and ')': Parentheses tokens

'{' and '}': Curly brace tokens.

'int': Keyword token.

'a' and 'b': Identifier tokens.

'=': Assignment operator token.

'10' and '30': Constant tokens.

';': Semicolon token.

'if': Keyword token.

'<': Less than operator token

'return': Keyword token.

'else': keyword token.

'a': Identifier token.

';': Semicolon token.

∴ Total no. of tokens

= 24 tokens.

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

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

→ $(0|1)^*$ : any sequence of 0's and 1's before substring

→ 101 : matches exact substring

→ $(0|1)^*$ : any sequence of 0's and 1's after substring.

3. Write a regular expression that have at lea two consecutive 0's (or) 1's :

$$(0,\{2,\}|1\{2,\})$$

→ $(0\{2,\})$ : Matches two or more consecutive 0's.

→ '|' : OR operator, want to match either of the conditions.

→ $(1\{2,\})$ : Matches two or more consecut 1's.

4. a = (b+c) * (b+c) * 2 ;

Lexical Analysis :-

Identifies : a

Assignment Operator : =

Left Parenthesis : (

Identifies : b

Addition Operator : +

Identifies : c

Right Parenthesis : )

Multiplication Operator : *

Left Parenthesis : (

Identifies : b

Addition Operator : +

Identifies : c

Right Parenthesis : )

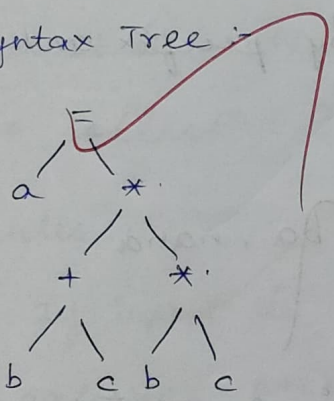
Multiplication Operator : *

Constant : 2

Semicolon : ;

Syntax Analysis :-

Syntax Tree :-




Semantic Analysis :-

→ checks if the variables and expressions are valid and consistent.

Code optimization :-

→ common optimization techniques like constant folding, common subexpression elimination and strength reduction.

Intermediate Code Generation :-

t1 = b + c

t2 = t1 * t1

a = t2 * 2

Code Generation :-

* The final step is to generate the machine code or assembly code from the optimized intermediate code.

Operation on Strings :-

1. Length of a string :- $s = 10101$ $\therefore |s| = 5$

2. Empty string :- $s = 0$ , $\varepsilon = 0$

3. prefix of string :- Any numbes of leading symbols in a string.

Let $s = abc$

prefix = $\varepsilon$, abc, a, ab.

4. suffix of string :- Any number of tail symbols in a string.

Let $s = abc$

suffix = $\varepsilon$, abc, c, bc.

5. Substring :- It is added by prefix and suffix. Let $s$ = Banana.

Substring = $\varepsilon$, Banana, ba, nana

→ A set of string which is generated by alphabets are called languages.

Example :- $\varepsilon = \{a, b\}$

$L = \{a, b, ab, ba, ab \ldots\}$

Operation on Languages :-

1) Union :-

$L = \{0, 1\}$ $M = \{00, 11\}$

$\therefore L \cup M = \{0, 1, 00, 11\}$

ii) Concatenation :-

$L = \{0, 1\}$ $M = \{00, 11\}$

$\therefore LM = \{000, 011, 100, 111\}$

iii) Kleene closure :- $(L^*)$

$\Sigma = \{a\}$

$\therefore \Sigma^* = \varepsilon, a, aa$

iv) Positive closure :- $(L^*)$

$\Sigma = \{a\}$

$\Sigma^* = a, aa$

Recursive Descent parsing :-

→ It is a Top Down Parser.

→ collection of recursive procedure

Rules :-

i) If input is non-terminal (upper-case) then call the procedure

ii) If the input is terminal (lower-case) then matched with input.

iii) If non-terminal produce more than one production then all the production score should be return.

Example :-

$E \rightarrow iE'$ $\quad E' \rightarrow +iE' / \varepsilon$

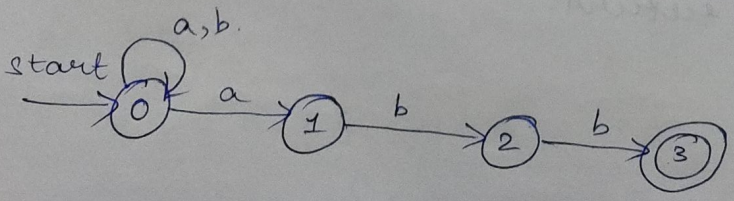
↳ Derive

```
E()
{
  if (input == 'i')
  input ++ ;
  E. PRIME ();
}
E PRIME ()
{
  if (input == '+')
  {
    input ++ ;
  if (input == 'i')
    input ++ ;
    E PRIME ();
  }
  else
    return
}
```

Non Deterministic Finite Automata (NFA)

Find out the NFA of Regular Expression

$(a+b)^* abb$.



context Free Grammar.

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$.

$A_1 \rightarrow bA_2$.

$A_2 \rightarrow bA_3$.

$A_3 \rightarrow \varepsilon$.

Grammer → Production.

$G = (V, T, P, S)$ → Start

Grammar. Variable. Terminal.

$F_2 \rightarrow F_2 + F_2$.
$\mid F_2 * F_2$.
$\mid id$.

3 Production

$\therefore V = \{E\} \rightarrow$ it contains only one variable on the left side So $V = \{E\}$.

$T = \{+, *, id\}$.

id + id * id → Generate the string

Left Most Derivation (LMD)

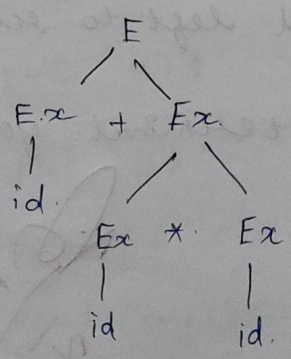
$E \Rightarrow E + F_2$.
$\Rightarrow id + F$
$\Rightarrow id + E * E$.
$\Rightarrow id + id * E$.
$\Rightarrow id + id * id$.

Right Most Derivation (RMD).

$F_2 \Rightarrow F_2 + E$.
$\Rightarrow F_2 + E * F_2$.
$\Rightarrow E + E * id$.
$\Rightarrow E + id * id$.
$\Rightarrow id + id * id$.

Parse Tree.

Construct a Parse Tree for id+id*id. Parse Tree.



11/07/2023

# Analytical Questions -2

i) Predictive parser

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

First(S) = {a, ↑, (}  Follow(S) = {, , ), $}

First(T) = {a, ↑, (}  Follow(T) = {), $}

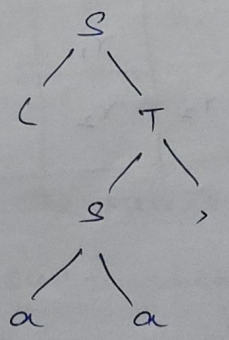
| | a | ↑ | ( |
|---|---|---|---|
| S | S → a | S → ↑ | S → (T) |
| T | T → S, T | T → S | T → (T) |

ii) Behaviour of the parser on the sentence

Parsing steps :-

1. stack. '$S'
2. Input '(a, a)$'
3. Input 'a, a)$'
4. Input ', a)$'
5. Input 'a)$'
6. Input ')$'
7. Input '$'
8. Input 'empty'

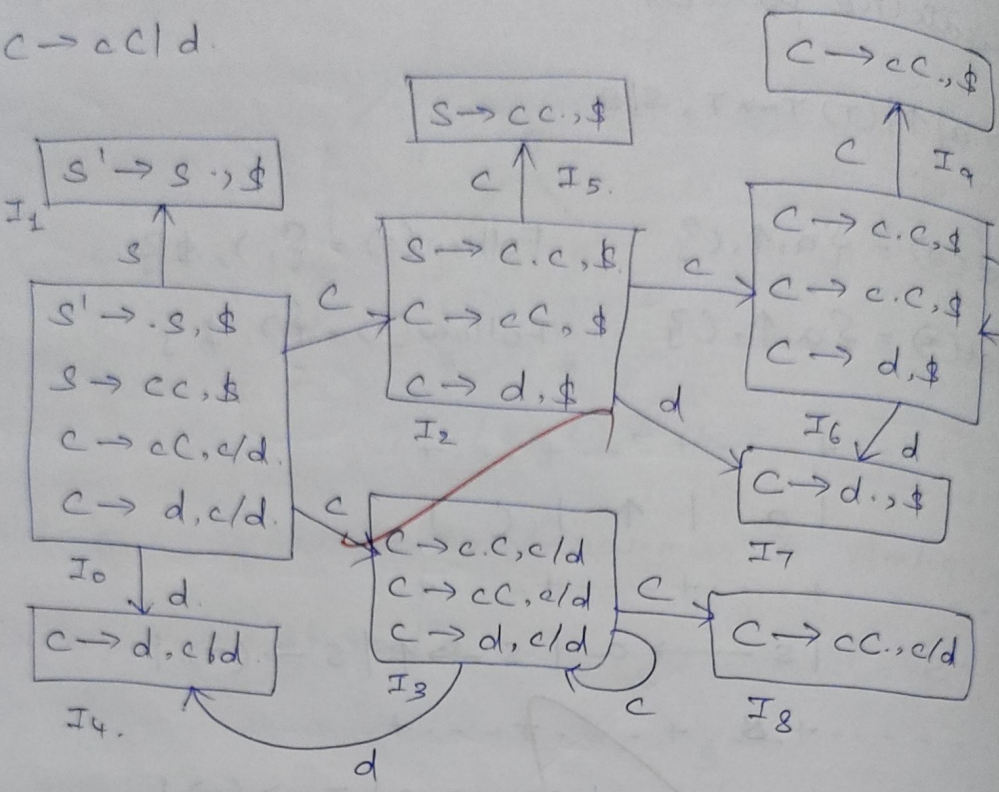
The parse Tree will be.



2. Construct SLR parsing Table

$S \to CC$

$C \to cC \mid d$




| State | ACTION | | | GOTO | |
|---|---|---|---|---|---|
| | c | d | $ | S | C |
| 0 | $S_{36}$ | $S_{47}$ | | 1 | 2 |
| 1 | | | acc | | |
| 2 | $S_{36}$ | $S_{47}$ | | | 5 |
| 36 | $S_{36}$ | $S_{47}$ | | | 89 |
| 47 | $r_3$ | $r_3$ | $r_3$ | | |
| 5 | | | $r_1$ | | |
| 89 | $r_2$ | $r_2$ | $r_2$ | | |

3. Given Grammar :-

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

FIRST sets :

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

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

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

FOLLOW SETS :-

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

$FOLLOW(A) = \{a, b\}$

$FOLLOW(B) = \{a, b\}$

Check for LL(1) Conditions :

i) No Two Productions with the same left-Hand side.

ii) FIRST/FOLLOW sets Conflict

$FIRST(A) = \{\varepsilon\}$ and $FOLLOW(S) = \{\$\}$

$FIRST(B) = \{\varepsilon\}$ and $FOLLOW(S) = \{\$\}$

3. Empty Productions :-

There are empty productions for A and B. Howeves. since they have distinct right-hand sides, they do not cause conflicts.

∴ The given grammar is LL(1).

4. $E \rightarrow 2E2$.

$E \rightarrow 3E3$.

$E \rightarrow A$.

S : Shift.

R : Reduce.

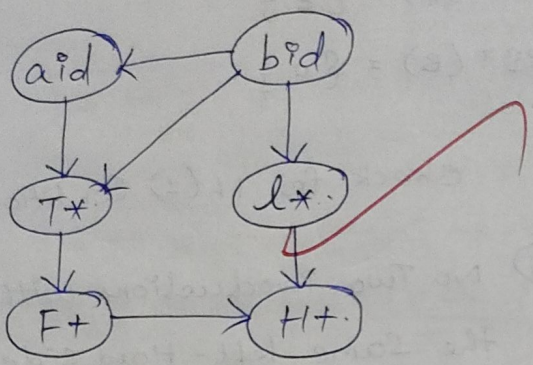
Parsing Table.

| State | 2 | 3 | 4 |
|---|---|---|---|
| 0 | S1 | S2 | S3 |
| 1 | S1 | S2 | S3 |
| 2 | R3 | R3 | R3 |
| 3 | R1 | R1 | R1 |

∴ The parsing is successful, and the input string "32 433" is valid according to the given grammar.

22/07/2023.

operator precedence flow graph.




operator precedence relation function.

| | id | * | * |
|---|---|---|---|
| F | | | |
| T | | | 2 |
| l | | | 1 |
| H | 4 | | |

Longest path.

aid → T* → F+ → H+.

bid → T* → F+ → H+.

bid → aid → T* → F+ → H+

T* → F+ → H+.

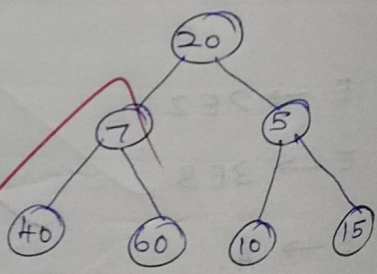
l* → H+.

bid → l* → H+.

YACC → Yet Another compiler compiler

Top Down Parser

→ This parser construct from ~~group~~ root to leaves with non-terminal.



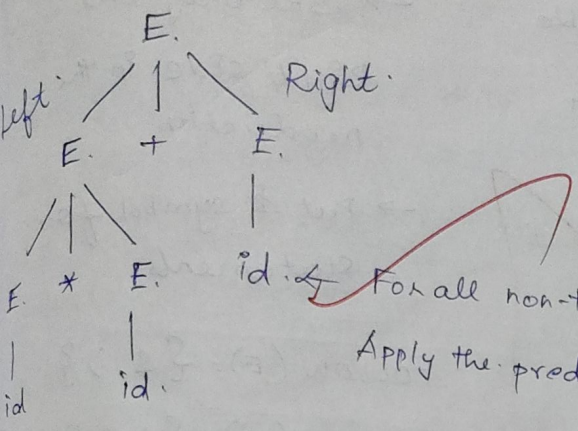

(pre-order traversal)

20 → 7 → 40 → 60

→ Top-Down parser construct parse tree as output

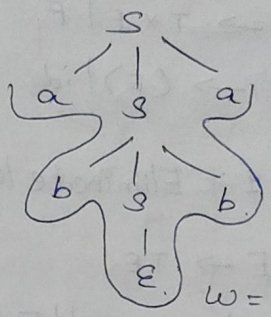
→ It gives us pre-ordered traversal (Root, left, Right)

$E \rightarrow E+E \mid E*E \mid id$.




Eg :-

$P = \{ S \rightarrow aSa,\ S \rightarrow bSb,\ S \rightarrow \varepsilon \}$ Apply with word $w = abba$.

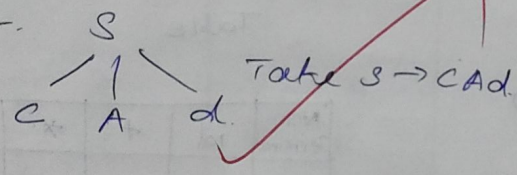



Q) Backtracking in Top Down Parser :-

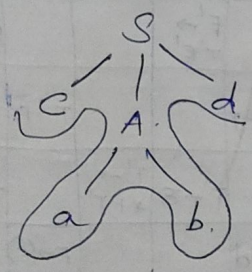
Construct the Grammar.

$S \rightarrow cAd$.
$A \rightarrow ab \mid d$ } Apply with word $w = cdd$.

Step 1 :-




Step 2 :-

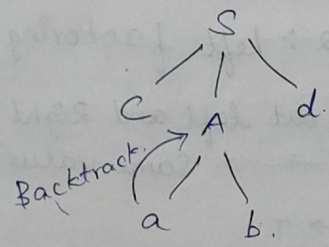



Now take $A \rightarrow ab$
↓
First variable.

Step 3 :- Now Form the yield. whether it matches $w = cdd$

No it forms cabb., so we backtrack.

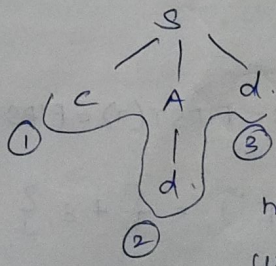
Step 4 :-




Now take the next alternative.

$A \rightarrow abld$.
↓
Next ~~Alternative~~ variable

Step 5 :-




Now form the yield.
$w = cdd$.
Yes it matches with $w = cdd$

# Analytical Questions - 3

1. SDT Scheme for 2*(4+5)

Expression → Term. { code = Term. code }

Term → Factor. { code = Factor. code }

Term → Term '*' Factor {

t = new_temporary_variable()

code = Term.code + Factor.code + t + '=' + Term.place

+ '*' + Factor.place + '\n'

place = t.

}

Factor → Number { code = '' ; place = Number.value }

Factor → '(' Expression. ')' { code = Expression.code ;

place = Expression.place. }

Number → [0-9] + { value = parseInt($1) }

2. Expression : $3*5+6*3$.

1. For $F \rightarrow num$ (num, val = 3):

   F. val = 3.

2. For $F \rightarrow num$ (num, val = 5):

   F. val = 5.

3. For $T \rightarrow F$:

   T. val = F. val = 5.

4. For $F \rightarrow num$ (num, val = 6):

   F. val = 6.

5. For $F \rightarrow num$ (num. val = 3):

   F. val = 3.

6. For $T \rightarrow F$:

   T. val = F. val = 3

7. For $T \rightarrow T*F$:

   T. val = 3 3*

8. For $E \rightarrow T$:

   E. val = T. val = 3

9. For $T \rightarrow F$:

   T. val = F. val = 5

10. For $T \rightarrow T*F$.

    T. val = 5 6*

3. $E' \rightarrow E$

   $E \rightarrow E+n \mid n$.

Stack : {}, Input : "n+n$"

Stack : {n}, Input : "+n$".

Stack : {n, +}, Input : "n$".

Stack : {E}., Input : "n$"

Stack : {E}, Input : "$"

Stack : {E'}, Input : "$",

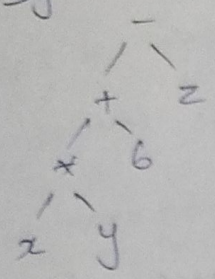
Final Configuration

Stack : {E'}.

Input : "$"

Action : Parsing Completed successfully.

4. Syntax Tree :-




5. LR Parsing :-

Stack : [0]

Input : 5*3+2 ; $

Action : Shift

Stack : [0, 5]

Input : *3+2 ; $

Action : Reduce

Stack : [0, 3]

Input : *3+2 ; $

Action : Shift

Stack : [0, 3, 6]

Input : 3+2 ; $

Action : Shift

Stack : [0, 3, 6, 3]

Input : +2 ; $

Action : Reduce

Stack : [0, 3, 6, 3, 4]

Input : +2 ; $

Action : Reduce

Stack : [0, 3, 6, 2]

Input : +2 ; $

Action : Shift

Stack : [0, 3, 6, 2, 7]

Input : 2 ; $

Action : Shift

stack : [0, 10]

Input : $

Action : Reduce

∴ The string is accepted

24/07/2023

Operator Precedence :-

→ Bottom up parsing

→ class of Grammars

The properties are no two variables are adjacent

No ε production.

id - highest precedence when compared with all the operators.

$ - Lowest precedence when compared with all the operators.

Using the Grammar $E \rightarrow E+E \mid E*E \mid id$

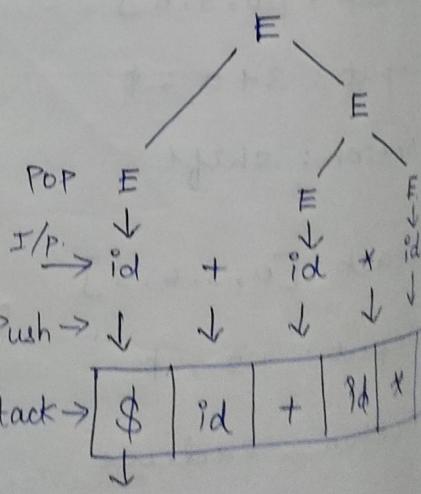
Form a operator precedence table

| | id | + | * | $ |
|---|---|---|---|---|
| id | - | > | > | > |
| + | < | > | < | > |
| * | < | > | > | > |
| $ | < | < | < | - |

{ Top of the Stack < Input String
Top of the Stack > Input String

Using the operator precedence table parse the assumed input.

Input = id + id * id $




→ whenever the top of the stack is less than or equal in the input string then you have to PUSH

Procedure :-

step 1 :- Compare $ and id in the op. precedence table. It shows < so push id inside the stack.

Now increment the pointer.

It goes to +.

step 2 :- Now compare id and + in the table, It shows > so pop id out of the stack and reduce the pointer, so the pointer goes to id again since it is popped.

step 3 :- Now compare $ and + it shows < in the table, so push + inside the stack, now increment the pointer, it goes to id.

Step 4 :- Now compare + and id, it shows <, so push id inside the stack, Now increment the pointer it goes to *.

Step 5 :- Now compare id and *, it shows > in the table, so pop id out of the stack and reduce the pointer, it moves to +.

Step 6 :- Now compare + and id., it shows < symbol in the table., so push * inside the stack., now increment the pointer, it moves to id.

Step 7 :- Now compare * and id., it shows < in the table, so push id inside the stack, and increment the pointer, it moves to $.

# Analytical Topic-4.

1. $-(a+b)*(c+d)*+(a+b+c)$

Quadruples :-

Quadruple 1 : $t1 = a+b$

$t2 = -t1$

$t3 = c+d$

$t4 = t2 * t3$

$t5 = a+b$

$t6 = t5 + c$

$t7 = t4 + t6$

Triples :-

$t1 = a+b$

$t2 = -t1$

$t3 = c+d$

$t4 = t2 * t3$

$t5 = a+b$

$t6 = t5 + c$

$t7 = t6 + t4$

Indirect Triples :-

$*t1 = a+b$

$*t2 = -*t1$

$*t3 = c+d$

$*t4 = *t2 * *t3$

$*t5 = a+b$

$*t6 = *t5 + c$

$*t7 = *t6 + *t4$

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

Quadruples :-

$t1 = b+c$

$t2 = -t1$

$t3 = a * t2$

Postfix :-

$a\ b\ c\ +\ -\ *$

Three-Address code :-

$t1 = b+c$

$t2 = -t1$

$t3 = a * t2$

3. Translation scheme :-

$t1 := a+b$

$t2 := c * d$

$g := t1 - t2$

Result three-address code

$t1 = a+b$

$t2 = c * d$

$g = t1 - t2$

H. Resulting three-address code using backpatching :-

```
if (a<b) goto L1.
t1 = 1.
goto L2.
L1 : t1 = 0.
if (c<d) goto L3.
t2 = 1.
goto L2.
L3 : t2 = 0.
L2 : t3 = t1 and t2.
```

25/7/23

25/07/2023.

SYNTAX DIRECT TRANSLATION (SDT) :-

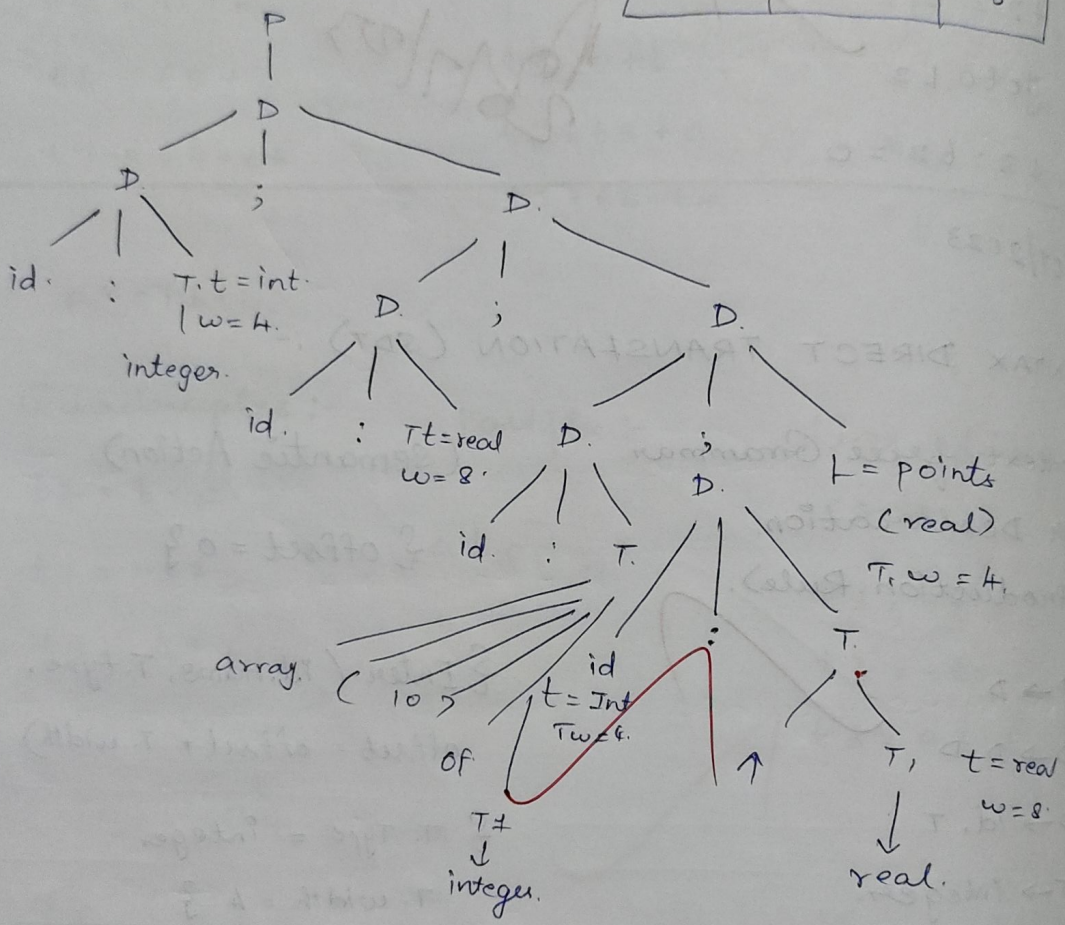
| Context free Grammar For Declaration. (Production Rule). | (Semantic Action) |
|---|---|
| | { offset = 0 } |
| P → D. | |
| D → D. D. | { Enter (id.name, T.type, offset = offset + T.width) |
| D → id. T | { T.Type = integer. T.width = 4 } |
| T → integer. | |
| T → real. | { T.Type = real. T.width = 8 } |
| T → array [num] of T. | { T.Type = array (num.val, T.type) } |
| T → ↑ T. | T.width = num.val * T1. width. ) |

Grammar.

{ a : integer
  b : real
  c : array [10] of integer
  d : ↑ real
}

| Name | Type | Offset |
|---|---|---|
| a | Integer | 0 |
| b | Real | 4 |
| c | array[10] | 10 |
| d | ↑ real | 18 |

Parse Tree.




INTERMEDIATE CODE GENERATION.

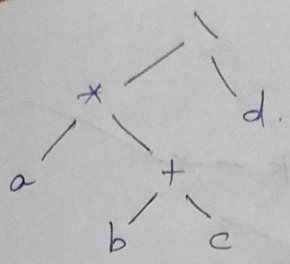
i) SYNTAX TREE. → The root node represents operator
leaf node represents operands

a * (b+c)/d

Eg :- construct a syntax tree for the following grammar

→ Brackets as the highest priority

→ Then the operator as the second priority




Post-fix expression :-

i) $(a+b)*c \rightarrow ab+c*$

ii) $a+(b*c) \rightarrow abc*+$

iii) $(a-b)*(c/d) \rightarrow ab-cd/*$

Three-address code :-

Condition :-

i) each instruction should contain three address

ii) one operator on right side, example :-

Using the grammar $a = b*-c+b*-c$

Find out the three address code representation having :- a) quadruples b) Triples c) Indirect Triples.

Four fields.
↓
operator,
argument 1,
argument 2,
result.

$t1 = -c$
$t2 = b*t1$
$t3 = -c$
$t4 = b*t3$
$t5 = t2+t4$

b) TRIPLE.

| | operator | Arg 1 | Arg 2 |
|---|---|---|---|
| t1 (0) | - | c | |
| t2 (1) | * | b | (0) t2 |
| t3 (2) | - | c | |
| t4 (3) | * | b | (2) t3 |
| t5 (4) | + | (1) | (3) t4 |

| | op | arg(1) | arg(2) | Res |
|---|---|---|---|---|
| (0) | - | c | | t1 |
| (1) | * | b | t1 | t2 |
| (2) | - | c | | t3 |
| (3) | * | b | t3 | t4 |
| (4) | + | t2 | t4 | t5 |

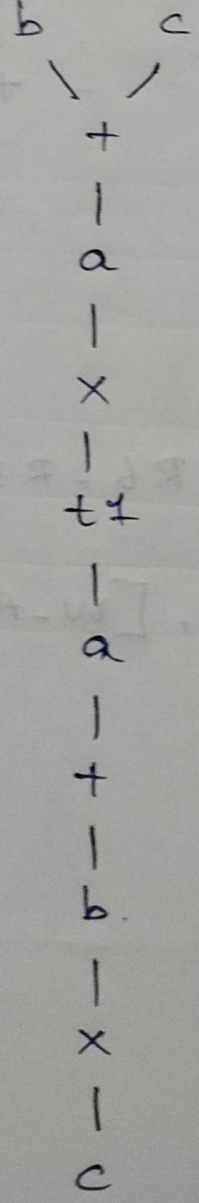
c) INDIRECT TRIPLE

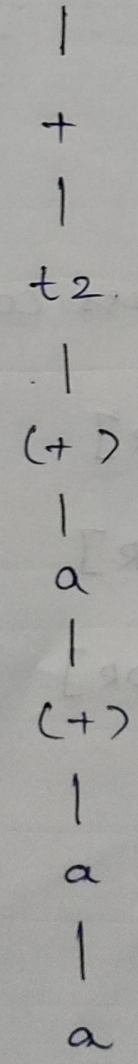
| | statement | op | arg1 | arg2 |
|---|---|---|---|---|
| (0) | 100 | - | c | |
| (1) | 101 | * | b | (100) |
| (2) | 102 | - | c | |
| (3) | 103 | * | b | (102) |
| (4) | 104 | + | 104 | (103) |

25/07/2023.

# Analytical Topic-5.

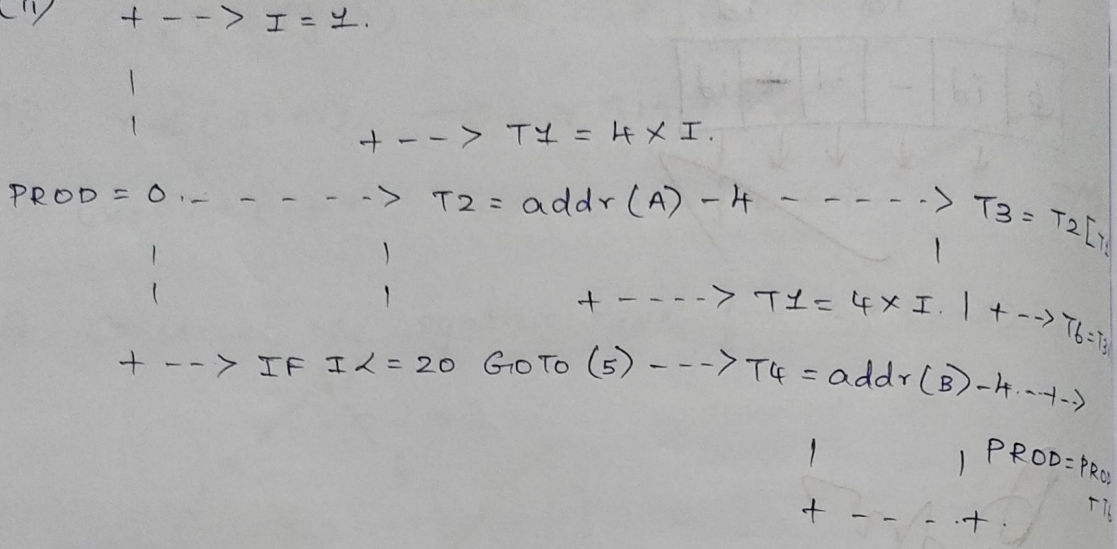
i) Directed Acyclic Graph :- (DAG).



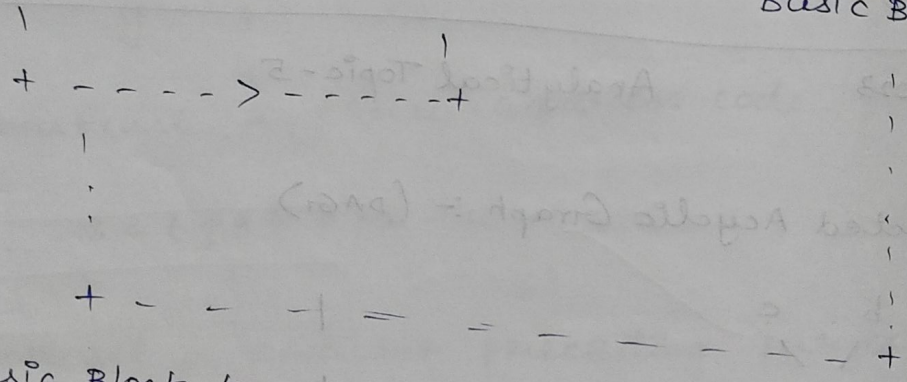



→ In the DAG, each node represents an operation, and the arrows indicate the flow of data from one operation to another.

(ii)




Basic Block 1. Basic Block 2. Basic Block 3.



Basic Block 4.

(iii). Three address code.

```
LOAD R1, [A_ADDR]
LOAD R2, [B_ADDR]
LOAD R3, [C_ADDR]
ADD R4, R1, R2
ADD R5, R1, R3
ADD R6, R4, R5
ADD R7, R6, R5
STORE R7, [W_ADDR]
```

iv). Construct Basic Blocks.

Basic Block 1 :-

PROD = 0.

I = 1.

Basic Block 2 :-

T2 = addr (A) - 4.

T4 = addr (B) - 4.

T1 = 4 x I.

Basic Block 3 :-

T3 = T2 [T1].

T5 = T4 [T1]

T6 = T3 x T5.

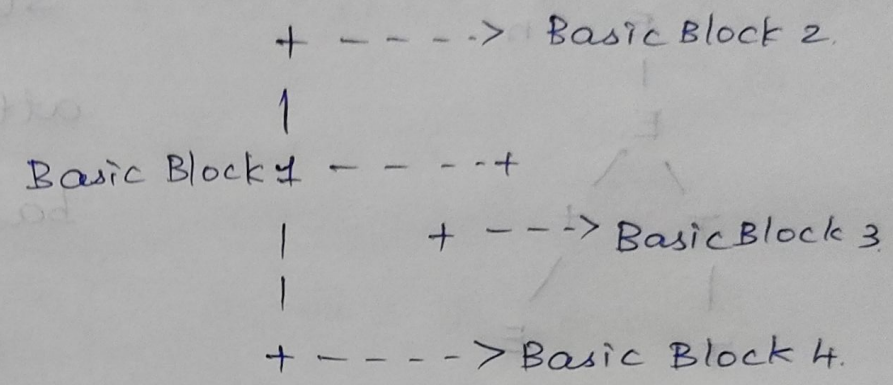
PROD = PROD + T6

I = I + 1.

Basic Block 4 :-

IF I <= 20 GOTO (5)

Flow Graph :-




Loop Invariant Statements :-

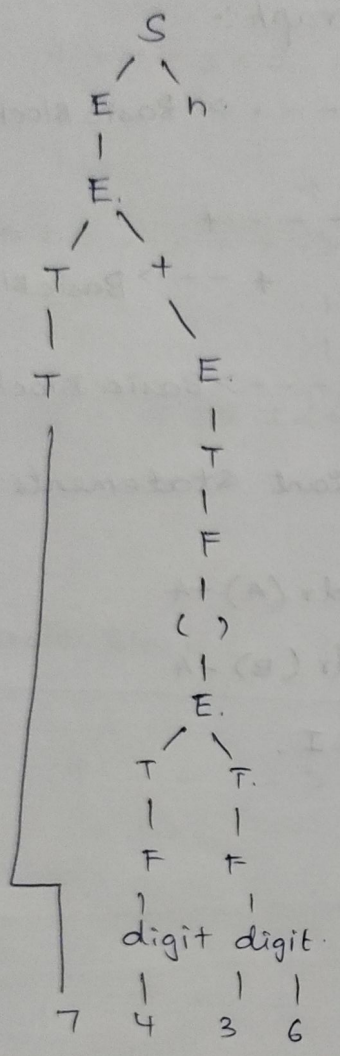
T2 = addr (A) - 4.

T4 = addr (B) - 4.

T1 = 4 x I.

# Analytical Question.

1. Step 1 :- Tokenize the expression and build the parse tree.




Step 2 :- Evaluate the attributes for each node based on the SDD rules

F. attr = 7

F. attr = 4.

T. attr = F. attr = 4.

F. attr = 3.

F. attr = 6.

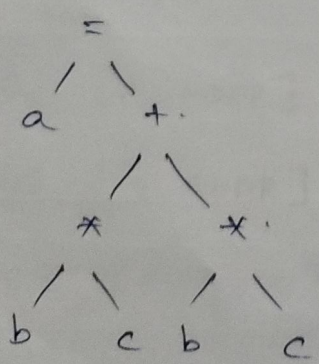
T. attr = F. attr = 6.

E. attr = T. attr = 6.

T. attr = F. attr = 10.

E. attr = E1. attr + T. attr =

2. Syntax Tree :-



3. E.val = 5. (digit)

F.val = 5 (E.val)

T.val = 5 (F.val)

E.val = 5 (T.val)

E.val = 6. (digit)

F.val = 6. (E.val)

T.val = 6 (F.val)

E.val = 6. (T.val)

T.val = 5 (T1.val) * 6 (F.val)
= 30.

T.val = 30 (T.val)

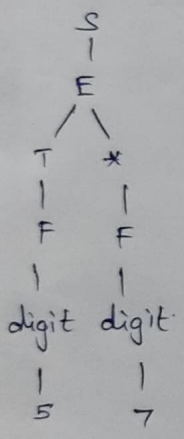
E.val = 30 (E1.val) + 4 (T.val)
= 34.

E.val = 34 (E.val)

S.val = 34 (E.val)

4. Input : '5*7'

Parse Tree :-




E.postfix = T.postfix = F.postfix =
digit.lexeme = 5

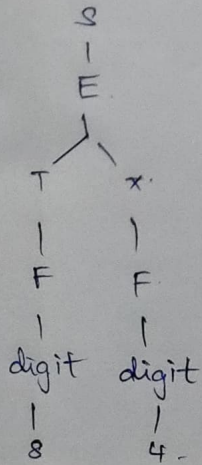
T.postfix = F.postfix + " " + digit.lexe-me
+ "*" = 5 + " " + 7 + "*" = "57*"

E.postfix = T.postfix = "57*"

S.postfix = E.postfix = "57*"

Input : "8*4".

Parse Tree :-




E.postfix = T.postfix = F.postfix =
digit.lexeme = 8.

T.postfix = F.postfix + " " + digit.lexeme
+ "*" = 8 + " " + 4 + "*" = "84*"

E.postfix = T.postfix = "84*"

S.postfix = E.postfix = "84*"