INDEX

NAME: K. Gowtham STD.: SEC.: ROLL NO.: SUB.:

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		Compiler Design		
		SDT	3 - 13	
		Intermediate Code Generation	14 - 20	
		Theory of Computation	21 - 39	
		Operating System	40 - 45	
		AVL Tree Deletion	46 - 50	
		Compile Design (Code optimization)	51 - 70	
		AVL Tree Insertion	71 - 72	
		DBMS (Join algo, log based recovery, B-tree deletion)	73 - 76	
		OBST	77 - 80	

04/10/20

Compiler Design

Syntax Directed Translation

Grammar + Semantic rules → SDT

SDT for evaluation of expression:

$E \to E_1 + T$ $\{E.val = E_1.val + T.val\}$

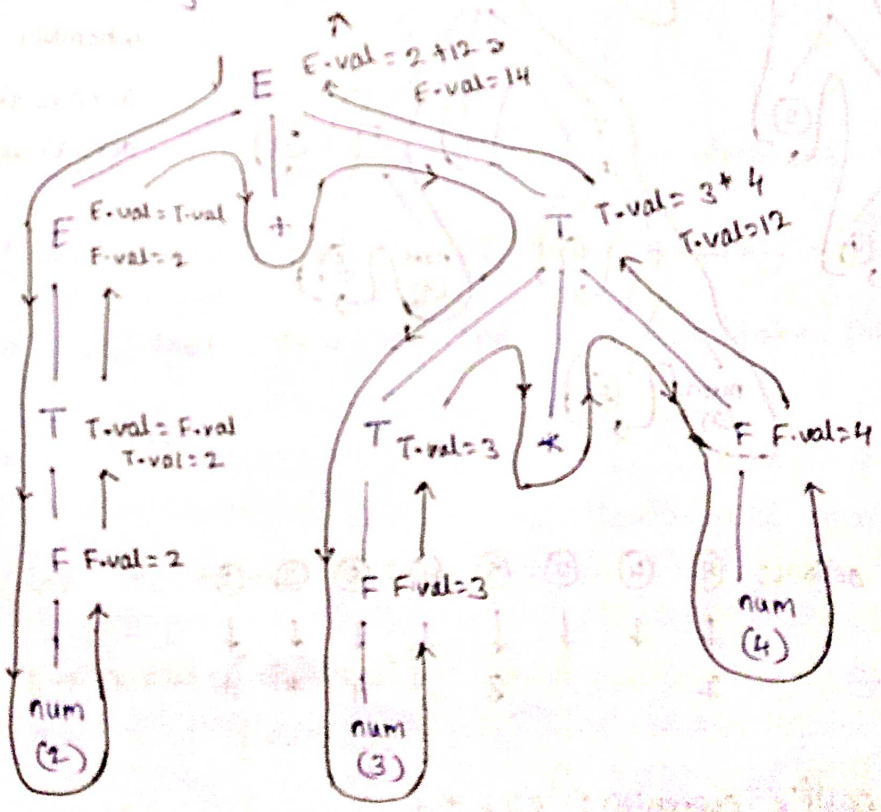
$E \to T$ $\{E.val = T.val\}$

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

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

$F \to num$ $\{F.val = num.lexvalue\}$

Eg: Consider string 2+3*4



→ while traversing, whenever we see a reduction we carry the corresponding operations of the corresponding production.

SDT for infix to postfix conversion:

$E \rightarrow E+T$ { Print("+"); } ------ ①

$E \rightarrow T$ { } ------- ②

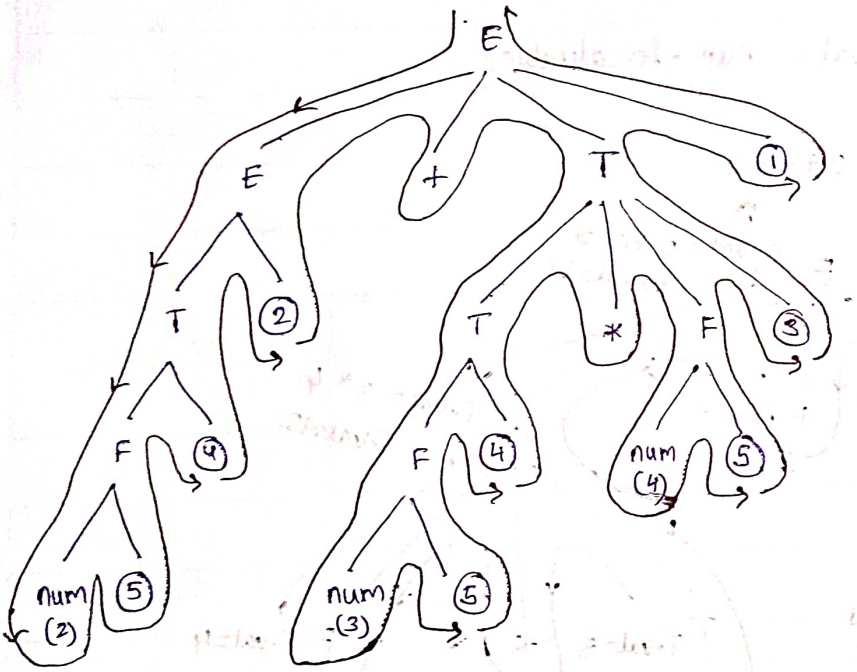
$T \rightarrow T*F$ { Print("*"); } --- ③

$T \rightarrow F$ { } ------- ④

$F \rightarrow num$ { Print(num.lval); } -- ⑤

working of a top down parser for above grammar:

Consider i/p string 2+3*4



while traversing, whenever we come across the semantic action, we perform it.

Semantic actions: ⑤ ④ ② ⑤ ④ ⑤ ③ ①

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

2 3 4 * +

∴ postfix expression: 234*+

working of bottom up parser:

	Reduction	o/p
2 + 3 * 4		
num + num * num (handle)	F → num	2
F + num * num	T → F	–
T + num * num	E → T	–
E + num * num	F → num	3
E + F * num	T → F	–
E + T * num	F → num	4
E + T * F	T → T * F	*
E + T	E → E + T	+
E	–	–

∴ o/p : 2 3 4 * +

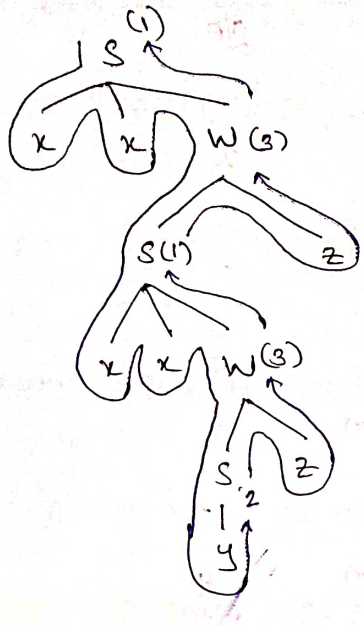
Also the above working can be shown by drawing the parse (like in the case of SDT for evaluation). If we observe, in that example the reductions take place in reverse of RMD.

Q ~~For~~ Consider the below SDT

S → x x W {print(1);}
S → y {print(2);}
W → S z {print(3);}

Find the o/p for string x x x y z z.

Sol :



∴ O/P : 2 3 1 3 1

Q Evaluate $w = 4-2-4*2$ for below SDT

$E \to E_1 * T$	$\{E.val = E_1.val * T.val\}$
$E \to T$	$\{E.val = T.val\}$
$T \to F - T_1$	$\{T.val = F.val - T_1.val\}$
$T \to F$	$\{T.val = F.val\}$
$F \to 2$	$\{F.val = 2\}$
$F \to 4$	$\{F.val = 4\}$

Sol :

Rather than constructing parse tree, we can analyze the grammar

From the grammar,

- has higher precedence &
- is right associative
- * is left associative

$\therefore [(4-(2-4))*2]$

$= (4-(-2))*2$

$= 6*12$

$= 12$

SDT to build syntax tree (abstract syntax tree)

$E \to E_1 + T$ { E.nptr = mknode (E_1.nptr , '+', T.nptr); }

$E \to T$ { E.nptr = T.nptr ; }

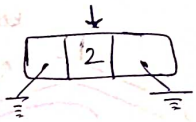
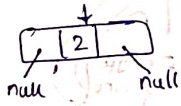
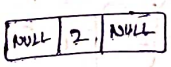
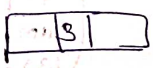
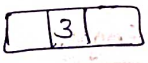
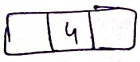
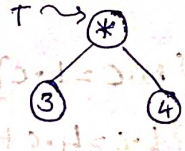
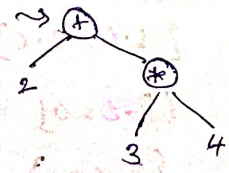
$T \to T_1 * F$ { T.nptr = mknode (T_1.nptr , '*', F.nptr); }

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

$F \to id$ { F.nptr = mknode (null, id.name, null); }

where ~~mkNo~~ mknode (a, b, c) creates a node 'b' with a & c are left & right pointers respectively

Consider the i/p string 2+3*4 & bottom up parsing

~~Stack~~	String	Reduction	
	id+id*id	$F \to id$	F.nptr 
	F+id*id	$T \to id$	T.nptr 
	T+id*id	$E \to T$	E: 
	E+id*id	$F \to id$	F: 
	E+F*id	$T \to F$	T: 
	E+T*id	$F \to id$	F: 
	E+T*F	$T \to T*F$	T 
	E+T	$E \to E+T$	E 
	E	–	

SDT for type checking:

$E \to E_1 + E_2$ { if((E₁.type == E₂.type) && (E₁.type == int)) then
E.type = int
else
error; }

$E \to E_1 == E_2$ { if((E₁.type == E₂.type) && (E₁.type = int/boolean)) then
E.type = boolean
else
error; }

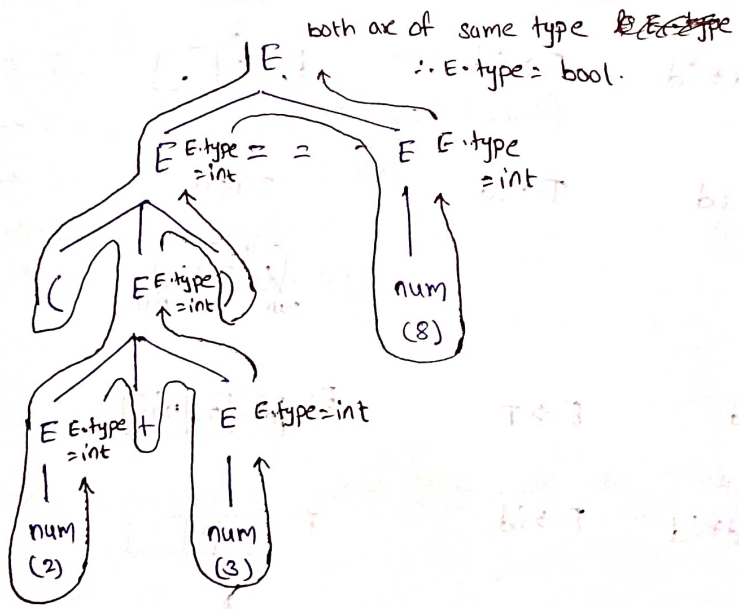
$E \to (E_1)$ { E.type = E₁.type; }

$E \to$ num { E.type = int; }

$E \to$ True { E.type = bool; }

$E \to$ False { E.type = bool; }

Eg: Check whether the i/p string (2+3) == 8 is semantically correct or not using above grammar



SDTs on binary strings:

→ The below is an SDT for counting no of 1's in a binary string

$N \to L$ { N.C = L.C }

$L \to L_1 B$ { L.C = L₁.C + B.C }

$L \to B$ { L.C = B.C }

$B \to 0$ { B.C = 0 }

$B \to 1$ { B.C = 1 }

→ The below is an SDT for calculating the value of binary string

$N \to L$ $\{N.val = L.val\}$

$L \to L_1 B$ $\{L.val = L_1.val * 2 + B.val\}$

$L \to B$ $\{L.val = B.val\}$

$B \to 0$ $\{B.val = 0\}$

$B \to 1$ $\{B.val = 1\}$

→ The below is an SDT for calculating the value of binary string with radix point.

$N \to L_1 . L_2$ $\{N.val = L_1.val + (L_2.val / 2^{L_2.cnt})\}$

$L \to L_1 B$ $\{L.cnt = L_1.cnt + B.cnt,\ L.val = L.val * 2 + B.val\}$

$L \to B$ $\{L.cnt = B.cnt,\ L.val = B.val\}$

$B \to 0$ $\{B.cnt = 1,\ B.val = 0\}$

$B \to 1$ $\{B.cnt = 1,\ B.val = 1\}$

SDT to generate Three Address Code:

$S \to id = E$ $\{gen(id.name = E.place);\}$

$E \to E_1 + T$ $\{E.place = newTemp();\ gen(E.place = E_1.place + T.place);\}$

$E \to T$ $\{E.place = T.place;\}$

$T \to T_1 * F$ $\{T.place = newTemp();\ gen(T.place = T_1.place * F.place);\}$

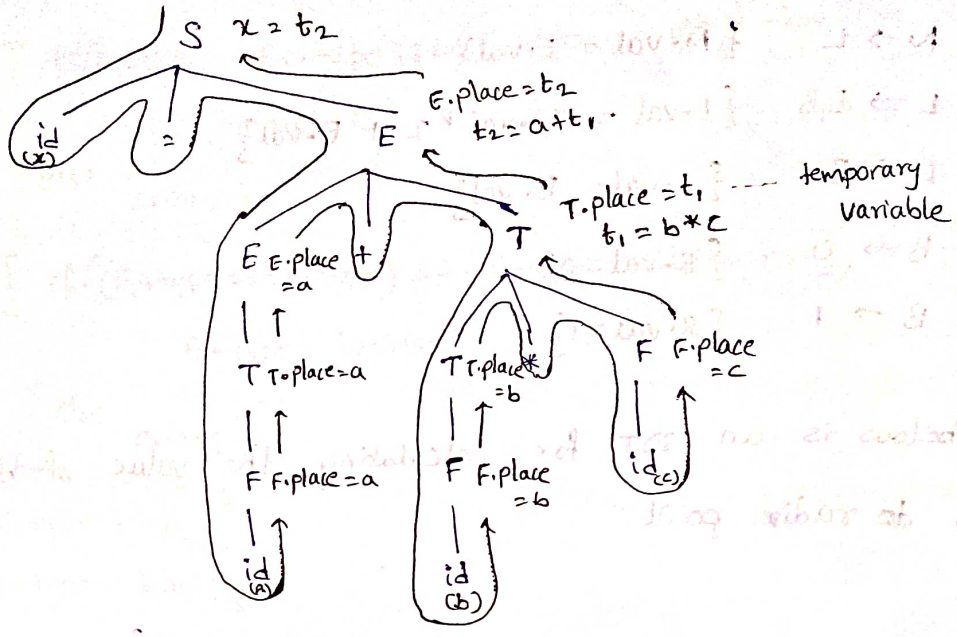
$T \to F$ $\{T.place = F.place;\}$

$F \to id$ $\{F.place = id.name;\}$

newTemp() → generates a temporary variable

gen() → generates required assignment stmt.

Eg : Consider i/p string $x = a+b*c$:



∴ O/P :

$$\left.\begin{aligned} t_1 &= b*c \\ t_2 &= a+t_1 \\ x &= t_2 \end{aligned}\right\} \text{TAC}$$

Types of Attributes:

Synthesized attribute:

Computed from child nodes

Inherited attributed:

Computed from parent node or siblings

~~Classified~~

Classification of SDT

S-attributed SDT	L-attributed SDT
(i) Uses only synthesized attributes	(i) Uses both inherited & synthesized attributes. Each inherited attribute is restricted to inherit either from parent or left sibling only.
(ii) Semantic actions are place at right end of the production. Hence S-attributed is also called postfix SDT	(ii) Semantic actions can be place anywhere on RHS Eg: $A \to \{\}BC$ $A \to BD\{\}E$ $A \to FG\{\}$
(iii) Attributes are evaluated during bottom up parsing	(iii) Attributes are evaluated by traversing parse tree depth first, left to right.

→ Every S-attributed is L-attributed, but reverse need not to be true.

04/10/20

Q) Consider below SDT

$A \to LM \{ L.i = f(A.i);\ M.i = f(L.s);\ A.s = f(M.s); \}$

$A \to QR \{ R.i = f(A.i);\ Q.i = f(R.i);\ A.s = f(Q.s); \}$

The SDT is

a) S-attributed b) L-attributed

c) both a & b d) none.

Sol:

$A \to LM \quad \{L.i = f(A.i) \; ; \quad M.i = f(L.s) \; ; \quad A.s = f(M.s)$

$L.i$ → inherited ∴ not S

$M.i$ → computing from left sibling ∴ L

$A.s$ → synthesized

$A \to QR \quad \{R.i = f(A.i) \; ; \quad Q.i = f(R.i) \; ;$

$R.i$ → Inherited

$Q.i$ → computing from right sibling ∴ not L

∴ opt. (d)

SDT to store type information into symbol table:

$D \to TL \quad \{L.in = T.type\}$

This SDT is L-attributed

$T \to int \quad \{T.type = int;\}$

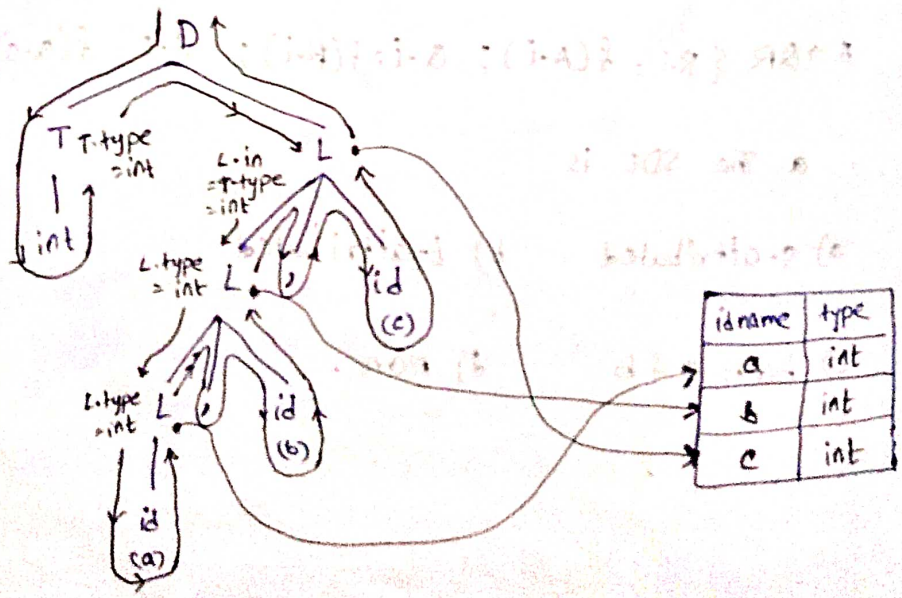
$T \to char \quad \{T.type = char;\}$

$L \to L_1, id \quad \{L_1.in = L.in, \; addtype(id.name, L_1.in);\}$

↳ comes under synthesized

$L \to id \quad \{addtype(id.name, L.in);\}$

Eg: Consider int a, b, c



idname	type
a	int
b	int
c	int

Evaluation of attributes in L-attributed SDT is done as

(i) while traversing evaluate the inherited attributes when the node is visited ~~for~~ for the first time.
↳ This include attribute evaluate from left ~~ch~~ sibling too.

(ii) Evaluate synthesized attributed when the node is visited last time.

S-attributed SDT to store type information into symbol table:

$D \rightarrow D_1, id$ {add_type (id.name, D_1.type)} D.type = D_1.type ;}

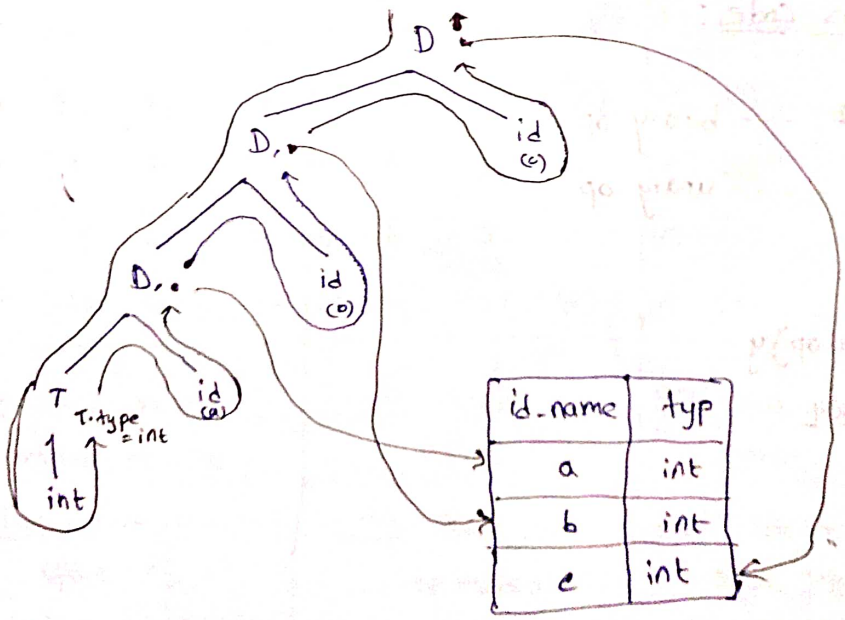
$D \rightarrow T\ id$ {add_type (id.name, T.type), D.Type = T.type ;}

$T \rightarrow int$ {T.type = int ;}

$T \rightarrow char$ {T.type = char ;}

The above SDT is S-attributed.

Consider string



id_name	typ
a	int
b	int
c	int

Observation:

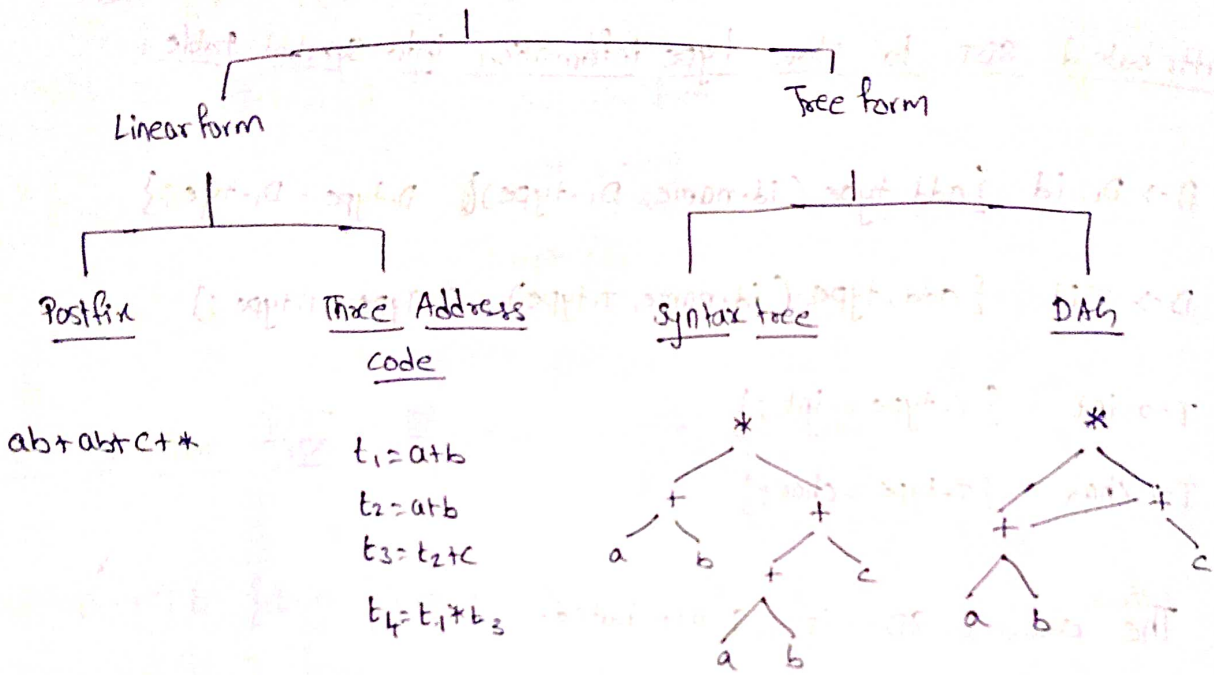
→ In the L-attr, the information (int) ~~flow~~ started at bottom level, then it moved up, then moved right, then moved down.

→ In S-att, the information flowed from bottom to top

09/10/20

Intermediate Code Generation:

Eg: $(a+b)*(a+b+c)$



Types of 3 address code:

(i) $x = y \text{ op } z$ --- binary op

(ii) $x = \text{op } z$ --- unary op

(iii) $x = y$

(iv) if x (relation op) y
goto L

(v) goto L

(vi) $A[i] = x$
$y = A[i]$

(vii) $x = *p$
$y = \&x$

Representation TAC :

Consider $-(a+b) * (c+d) + (a+b+c)$ and its TAC

$t_1 = a+b$

$t_2 = -t_1$

$t_3 = c+d$

$t_4 = t_2 * t_3$

$t_5 = a+b$

$t_6 = t_5 + c$

$t_7 = t_4 + t_6$

The below are representations of TAC

Quadraples					Triples				Indirect triples	
	Oprand	OP1	op2	res		Opr	op1	op2	Prog	pointers
1)	+	a	b	t_1	1)	+	a	b	(i)	(1)
2)	-	t_1	-	t_2	2)	-	(1)		(ii)	(2)
3)	+	c	d	t_3	3)	+	c	d	(iii)	(3) → pointers
4)	*	t_2	t_3	t_4	4)	*	(2)	(3)	(iv)	(4)
5)	+	a	b	t_5	5)	+	a	b	(v)	(5)
6)	+	t_5	c	t_6	6)	+	(5)	c	(vi)	(6)
7)	+	t_4	t_6	t_7	7)	+	(4)	(6)	(vii)	(7)
adv: stmts can be moved arround dis: too much space req					adv: space is not wasted dis: Stmts can't be moved				adv: Stmts can be moved dis: Two memory access req.	

11/10/20

Back patching:

Eg: if (a<b) then t=1
else t=0

TAC:

100: if a<b goto 103
101: t=0
102: goto 104
103: t=1
104:

These blanks are filled using the technique called back patching.

~~Ex: if (a<b and c<d (or) e<f) then~~
~~t=1~~
~~else~~
~~t=0~~

Eg: a<b and c<d or e<f

100: if a<b goto 104
102: $t_1 = 0$
103: goto 105
104: $t_1 = 1$
105: if (c<d) goto 108
106: $t_2 = 0$
107: goto 109
108: $t_2 = 1$
109: if (e<f) goto 112
110: $t_3 = 0$
111: goto 113
112: $t_3 = 1$
113: $t_4 = t_1$ and t_2
114: $t_5 = t_4$ (or) t_3

Eg: while E do S

```
L: If (E==0) goto L1
      S
      goto L
L1:
```

(or)

```
L: If (E) goto L1
      goto L2
L1: S
      goto L
L2:
```

→ Identifying loops in this TAC is difficult.
↳ Required in Code optimization.

Eg: while (a<b) do
x=y+z

```
L: if a<b goto L1
      goto L2
L1:   t=y+z
      x=t
      goto L
L2:
```

Eg: for (i=0; i<10; i++)
a=b+c;

TAC:

```
      i=0
L: if (i<10) goto L1
      goto L2
```

L_1 : $t_1 = b+c$

$a = t_1$

$t = i+1$

$i = t$

goto L

L_2 :

Q: Switch (i+j)

{

case 1 :

a=b+c

break;

case 2:

P=Q+R

break;

default :

x=y+z;

break;

}

t=i+j

if (t==1) goto L_1

if (t==2) goto L_2

goto L_3

L_1: $t_1 = b+c$

$a = t_1$

goto last

L_2 : $t_2 = Q+R$

$P = t_2$

goto last

L_3: $t_3 = y+z$

$x = t_3$

goto last

last :

Ex:

2D array to TAC:

$x = A[y, z]$ $\quad$ $A : 10 \times 20$

TAC:

$t_1 = y * 20$

$t_2 = t_1 + z$ $\quad$ → no of elements to be crossed

$t_3 = t_2 * 4$ $\quad$ → no of bytes to be crossed

t_4 = base address of A

$x = t_4[t_3]$

Theory of Computation

07/12/20

FA to Regular Expression:

There are 3 methods for this:

(i) Arden's Method:

If $R = Q + RP$ and P doesn't contain ϵ then

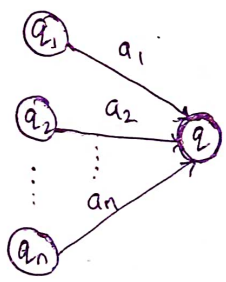
$R = QP^*$ is the solution.

If P contains ϵ then there will be infinite solutions.

→ So this method is applied for DFA & NFA without ϵ moves.

→ In this we write an equation for each state

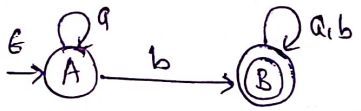
Eg:



$\Rightarrow q = q_1 a_1 + q_2 a_2 + \dots + q_n a_n$

If 'q' is initial state then add ϵ on RHS of equation.

Eg:



$A = \epsilon + Aa$

$B = Ab + Ba + Bb = Ab + B(a+b)$

Solving A:

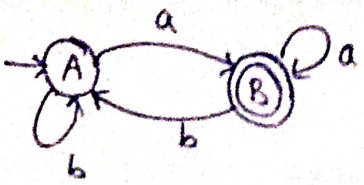
$A = \epsilon a^* = a^*$

$\Rightarrow B = a^*b + B(a+b)$

$\Rightarrow B = a^*b(a+b)^*$

∴ Reg Exp corresponding to the automata is $a^*b(a+b)^*$

Eg:



$\Rightarrow A = \epsilon + Ab + Bb$

$B = Aa + Ba$

Solving A:

$A = (\epsilon + Bb) + Ab$

$\Rightarrow A = (\epsilon + Bb)b^*$

$\Rightarrow B = (\epsilon + Bb)b^*a + Ba$

$\cancel{= \epsilon + B}$

$B = b^*a + Bbb^*a + Ba$

$B = b^*a + B(bb^*a + a)$

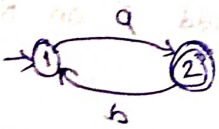
$\Rightarrow B = b^*a(bb^*a + a)^*$

$= b^*a((bb^* + \epsilon)a)^*$

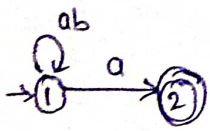
$= b^*a((b^+ + \epsilon)a)$

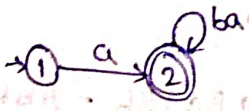
$B = b^*a(b^*a)^*$

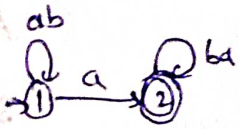
Eg: Consider



This DFA can be modified as below 3 ways





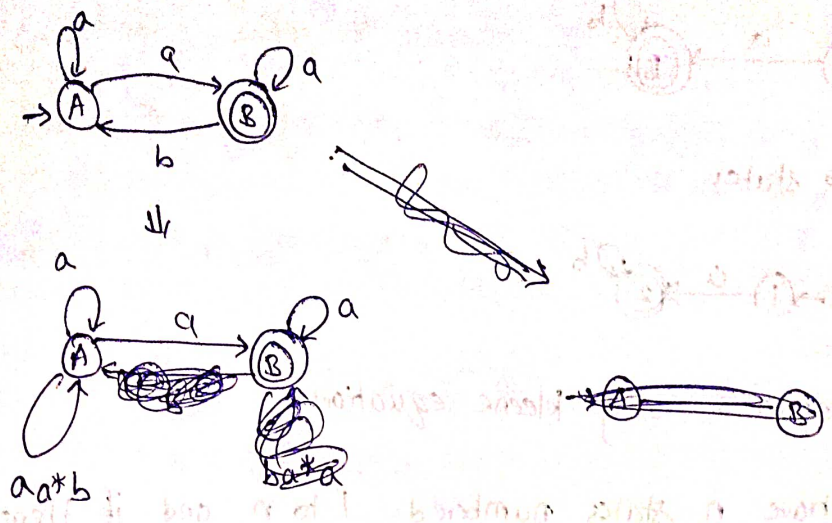


$\therefore$ Reg Exp: $(ab)^*a$

Reg Exp: $a(ba)^*$

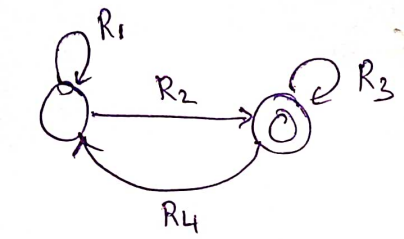
Reg Exp: $(ab)^*a(ba)^*$

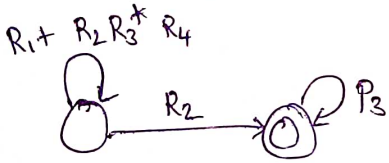
Eg:



$\Rightarrow (a+aa^*b)^* a a^*$

Note:





$\therefore (R_1 + R_2 R_3^* R_4)^* R_2 R_3^*$

Kleene Method:

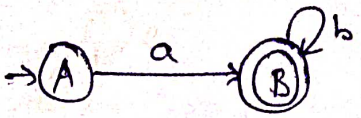
Kleene equation:

$$R_{ij}^{k} = R_{ij}^{k-1} + R_{ik}^{k-1} \left(R_{kk}^{k-1}\right)^* R_{kj}^{k-1}$$

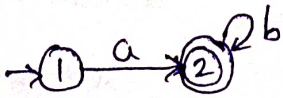
Time Complexity: $O(n^3)$

$n \rightarrow$ no of states

Eg: Consider



Step 1: Number the states



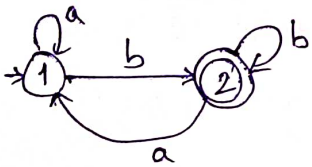
Step 2: Find R_{12}^{2} equation using kleene equation

If DFA have n states numbered 1 to n and if there are k final states $f_1, f_2, \ldots f_k$ then we find

$$R_{1f_1}^{n} + R_{1f_2}^{n} + R_{1f_3}^{n} + \cdots + R_{1f_k}^{n}$$

Here 1 is initial state

Eg:



$R_{12}^{2} = ?$

Base cases:

$R_{11}^{0} = \epsilon + a$

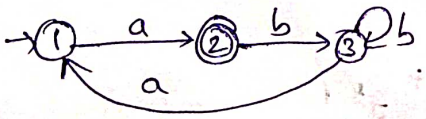
$R_{12}^{0} = b$

$R_{21}^{0} = a$

$R_{22}^{0} = \epsilon + b$

$R_{kk}^{0} = \epsilon +$ (self loop label)

Eg:



$R_{12}^{3} = ?$

Base Cases:

$R_{11}^{0} = \epsilon$ $\quad R_{21}^{0} = \phi$ $\quad R_{31}^{0} = a$

$R_{12}^{0} = a$ $\quad R_{22}^{0} = \epsilon$ $\quad R_{32}^{0} = \phi$

$R_{13}^{0} = \phi$ $\quad R_{23}^{0} = b$ $\quad R_{33}^{0} = \epsilon + b$

Now Continuing with 1st example

$$R_{12}^{2} = R_{12}^{1} + R_{12}^{1} (R_{22}^{1})^{*} R_{22}^{1} \longrightarrow (1)$$

$$\Rightarrow R_{12}^{1} = R_{12}^{0} + R_{11}^{0} (R_{11}^{0})^{*} R_{12}^{0}$$

$$= b + (\epsilon + a)(\epsilon + a)^{*} b$$

$$= b + (\epsilon + a) a^{*} b$$

$$= b + a^{*} b$$

$$= (\epsilon + a^{*}) b$$

$$= a^{*} b$$

$$\Rightarrow R_{22}^{1} = R_{22}^{0} + R_{21}^{0} (R_{11}^{0})^{*} R_{12}$$

$$= (\epsilon + b) + a (\epsilon + a)^{*} b$$

$$= (\epsilon + b) + a a^{*} b$$

$$= \epsilon + (\epsilon + a^{+}) b$$

$$= \epsilon + a^{*} b$$

Substituting these in eq (1)

$$\Rightarrow R_{12}^{2} = a^{*}b + a^{*}b (\epsilon + a^{*}b)^{*} (\epsilon + a^{*}b)$$

$$= a^{*}b + a^{*}b (a^{*}b)^{*} (\epsilon + a^{*}b)$$

$$= a^{*}b + a^{*}b \left[(a^{*}b)^{*} + (a^{*}b)^{+} \right]$$

$$= a^{*}b + a^{*}b (a^{*}b)^{*}$$

$$= a^{*}b \left[\epsilon + (a^{*}b)^{*} \right] \qquad (\because a + b = b, \text{ if } a \subseteq b)$$

$$= a^{*}b (a^{*}b)^{*}$$

All pair shortest path
Kleene method
Finding transitive closure

} Similar way of Dynamic Programming

Regular Grammar to FA :

Regular Languages

Reg Exp ⟷ FA ⟷ Reg Grammar

FA to reg grammar :

For every state we create a non-terminal.

All symbols in Σ are terminals.

For every transition we create a production.

Initial state's corresponding non-terminal is start symbol.

Eg:

(At every state, write productions for outgoing transition)

$X \to aX$

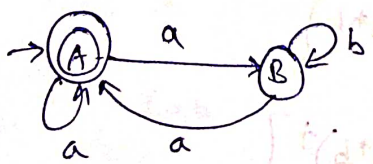
$X \to bY$

$Y \to \epsilon$ (For every final state add Production $F \to \epsilon$)

~~→ For every~~

This method gives Right Linear grammar.

Eg:

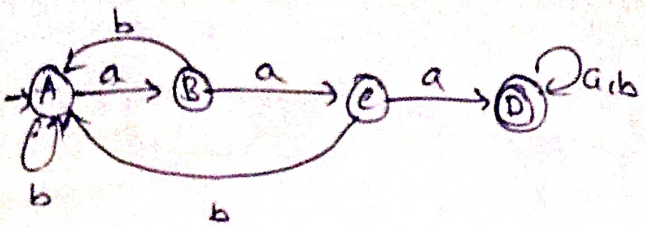


$A \to aA / aB / \epsilon$

$B \to bB / aA$

Reversing above productions will give regular grammar (LLG) for reverse of FA.

Eg:



$A \to bA \mid aB$

$B \to aC \mid bA$

$C \to baD \mid bA$

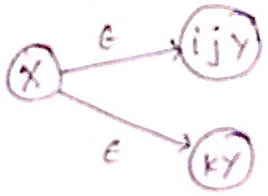
$D \to aD \mid bD \mid \epsilon$

<u>Regular Grammar (RLG)</u> to <u>FA</u>:

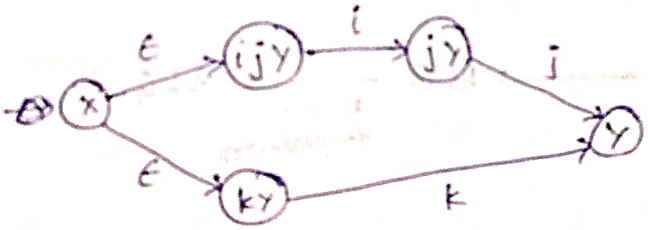
→ This applicable only for RLG

For every LHS part of production create a state and we move to this state using ϵ-move

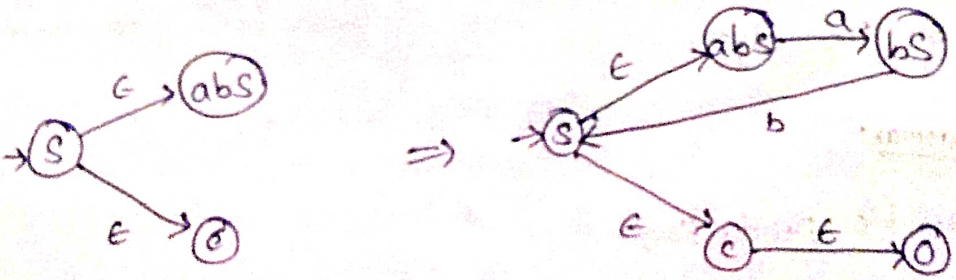
Eg: $X \to ijY \mid kY$



Now we do the below modification

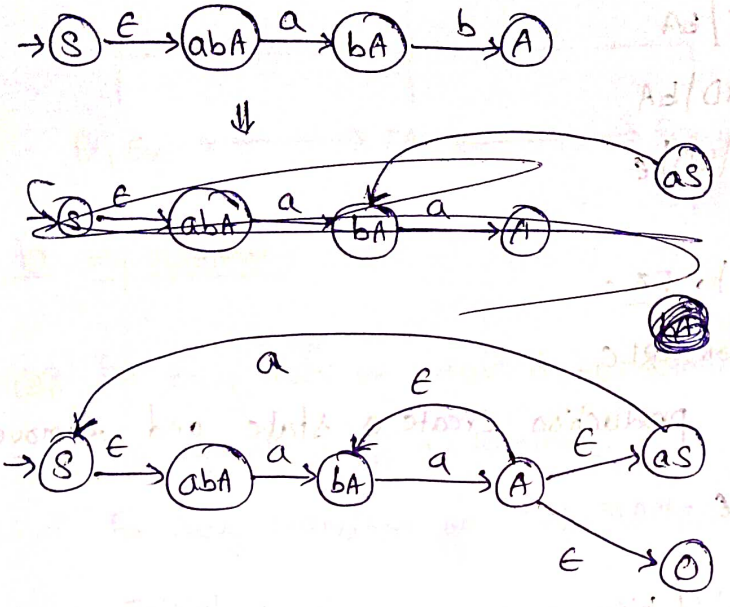


Eg: $S \to abS \mid c$



Eg: $S \to abA$

$A \to aS \mid bA \mid \epsilon$



Eg: ~~S → aA | abB~~ | bc

FA to left linear grammar:

Procedure:

FA ⟶ Reverse of FA ⟶ RLG ⟶ (reverse the productions) LLG.

Reversal of FA:

→ Interchange initial & final states.

→ Reverse all the edges.

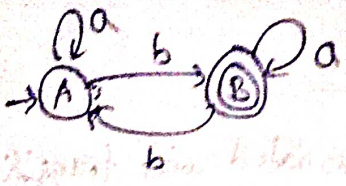
Reversing grammar:

Reverse the productions

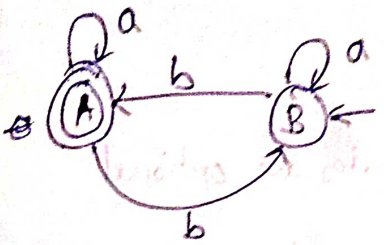
Eg: $B \to bA$ ⟹ $B \to Ab$

$A \to aA \mid \epsilon$ ⟹ $A \to Aa \mid \epsilon$

Eg:



⇓ Reversal



⇓

$B \to aB \mid bA$

$A \to aA \mid bB \mid \epsilon$

⇓

$B \to Ba \mid Ab$

$A \to Aa \mid Bb \mid \epsilon$ (B is start symbol)

When reversing
If there are more than 1 final states, create an initial state and add ϵ transition to every final state.

LLG to FA:

LLG $\xrightarrow{\text{reverse}}$ RLG $\longrightarrow$ FA $\xrightarrow{\text{reverse}}$ Req FA

FA without O/P:

- Moore
- Mealy

→ In FA with O/P we have O/P alphabet also & we don't have any final states here.

It is represented as 6 tuple

$(Q, \Sigma, \delta, q_0, \Delta, \lambda)$

↳ O/P transition function

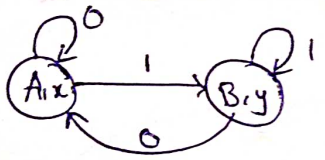
Moore M/C

→ O/p is associated with state.

$\lambda : Q \rightarrow \Delta$

→ DFA

→ Final states are optional

→ 

I/P : 0 1 1 0

$A \xrightarrow{0} A \xrightarrow{1} B \xrightarrow{1} B \xrightarrow{0} A$

x x y y x

∴ O/P : x x y y x

The 1st o/p (due to initial state) may be ignored if needed.

→ I/P length = n then

O/P length = n+1

→ No delay in producing O/P symbol after reading i/P symbol.

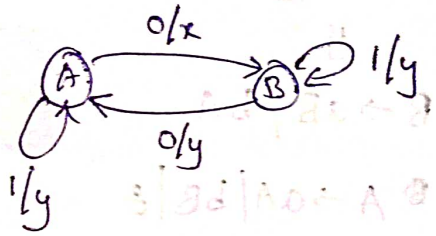
Mealy M/C

→ O/p is associated with transition.

$\lambda : Q \times \Sigma \rightarrow \Delta$

→ DFA

→ Final states are optional.

→ 

I/P : 01101

O/P : x y y x y

→ If I/P length = n then

O/P length = n+1

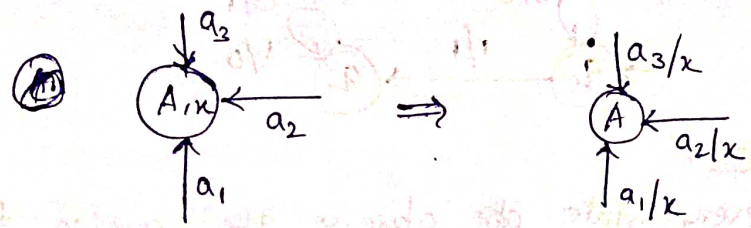
→ Delay is present as transition has to be made.

Moore to Mealy :

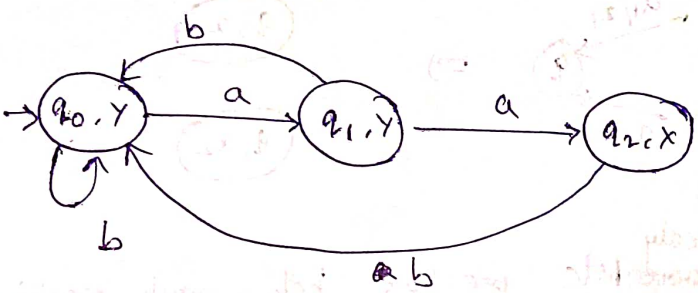
From moore draw the same FA.

The only change is for all incoming transitions shift o/p to that

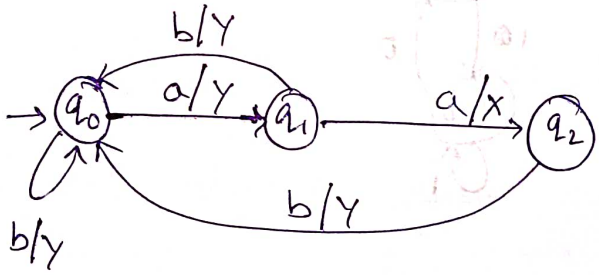
Eg :



Eg : Consider below Moore that print x if string ends with aa and Y otherwise.



⇓ Moore to ~~Mealy~~ Mealy



If λ is o/p function of Moore and λ' is of Mealy then

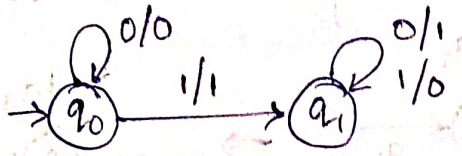
$$\lambda'(q, a) = \lambda[\delta(q, a)]$$

where δ is transition function of Moore machine.

Mealy to Moore:

Consider below mealy machine that finds 2' complement.

Here i/p is given in reverse

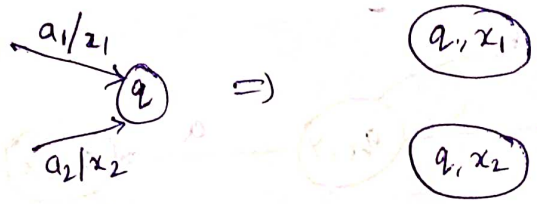


Conversion:

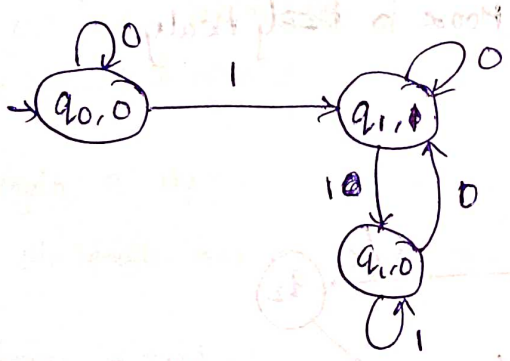
→ For every state observe the incoming transitions.

If no of different o/ps to the incoming state are k then create k state from it.

Eg:



∴ For above ~~Moore~~ Mealy M/c we have below mealy machine



$$\delta'([q, ax], a) = [\delta(q, a), \lambda[q, a)]$$

δ → transition function of Mealy M/c

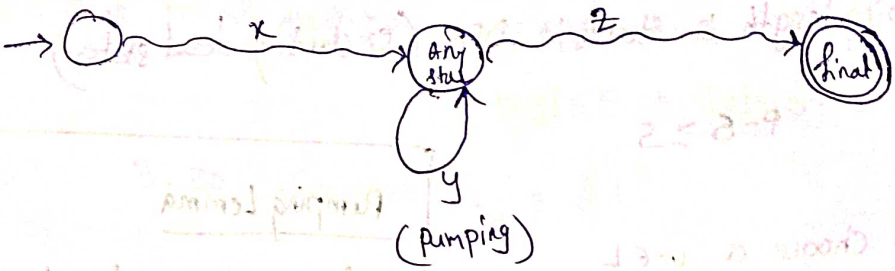
λ → o/p func of Mealy M/c

δ' → transition func of Moore M/c

8/12/20

Pumping Lemma for regular languages :

→ If a regular language is infinite, then there must be some part of the string that repeats. ~~for infinite~~



$\forall i,\ xy^iz \in L$ & L is regular

(i) Choose a constant 'p' (pumping constant) / (Pumping length)

(ii) Choose a string w such that $|w| \geq p$

(iii) Divide w into 3 parts

such that $w = xyz$ where $|xy| \leq p$ & $|y| \neq 0$

(iv) $\forall i,\ xy^iz \in L$ iff L is regular

Eg : If reg exp is $a(ba)^*a$

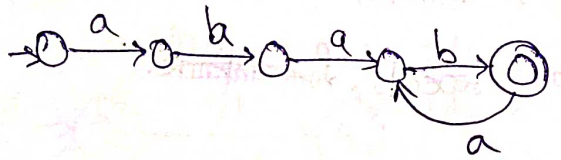
Consider string $w = \underset{x}{a}\underset{y}{ba}\underset{z}{a}$

Choose x, y, z so that we prove the lang is regular

$a(ba)^ia \in L\ \forall i$

Eg: $R = a(ba)^+ab$

we choose pumping length from DFA.



(i) pumping length $\geq$ states in DFA (excluding dead states)

$P = 6 \geq 5$

(ii) Choose a $w \in L$

$w = \underbrace{a}_{x}\underbrace{bab}_{y}\underbrace{ab}_{z} \in L$

$\forall i,\ xy^iz \in L$

$i=0 \Rightarrow abab$

$i=1 \Rightarrow ababab$

$\therefore L$ is regular

Pumping Lemma

If L is a regular language, then $\exists P \geq 0$ such that for all string $w \in L$ where $|w| \geq P$ there exists a partition such that $w = xyz$ such that $|xy| \leq P$ and $|y| > 0$ and for all $i \geq 0$, $xy^iz \in L$

proving a language is non regular using pumping lemma:

~~Assume L is regular.~~

Choose pumping length P

Choose string $w \in L$ such that $|w| \geq P$

divide w into 3 parts such that $w = xyz$ / $|xy| \leq P$ $|y| \neq 0$

L is non regular if $\exists i,\ xy^iz \notin L$

we prove a language is not regular using contrapositive form of pumping lemma which is shown below

Let $L \subseteq \Sigma^*$

if $\forall p \geq 0$

$\exists w \in L$ such that $|w| \geq p$

such that for all partitions $w = xyz$ where

$|xy| \leq p \wedge |y| > 0$

$\exists i \geq 0$ such that $xy^iz \notin L$

$\therefore$ pumping lemma is used to prove a language is not regular. But we can't use it to prove a language is regular.

Eg: P.T $L = \{0^n 1^n \mid n \geq 0\}$ is not regular

Given - $p \geq 0$ (p can be any number)

Choose $w = 0^p 1^p$

Now ~~t~~ partitions

$w = xyz$ $\quad \underbrace{000\cdots0}_{p}\underbrace{111\cdots1}_{p}$

~~let~~ $|xy| \leq p$

and xy does have 1

So xy^iz ~~a~~ for any i

won't ~~a~~ be string of L

$\therefore L$ is not regular.

Grammars

→ Chomsky classification of grammars:

(i) Type 3

(ii) Type 2

(iii) Type 1

(iv) Type 0

Type 3 / Regular Grammar:

Type 3 is one of the below forms

$A \to \alpha B \mid \beta$ $A, B \in V$ $\alpha, \beta \in T^*$	$A \to B\alpha \mid \beta$ $A, B \in V$ $\alpha, \beta \in T^*$
Right linear grammar	Left linear grammar

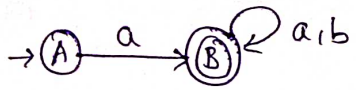
FA to RG (RLG):

For every $A \xrightarrow{a} B$

add production $A \to aB$

and for every final state Q

add production $Q \to \epsilon$

Eg: 

$$\left.\begin{array}{l} A \to aB \\ B \to aB \mid bB \mid \epsilon \end{array}\right\} \xrightarrow{\text{Reversing}} \underbrace{\begin{array}{l} A \to Ba \\ B \to Ba \mid Bb \mid \epsilon \end{array}}_{\text{gives grammar for reverse of the FA}}$$

$$\underset{(L)}{FA} \longrightarrow \underset{(L)}{RLG} \xrightarrow{Rev} \underset{(L^R)}{LLG}$$

FA to LLG:

$$FA \xrightarrow{Rev} \underset{(L^R)}{FA} \longrightarrow \underset{(L^R)}{RLG} \xrightarrow{Rev} \underset{(L)}{LLG}$$

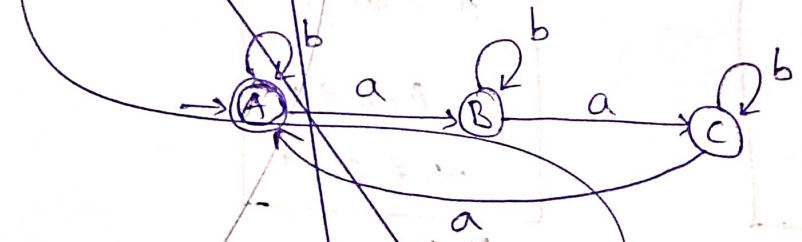
RG to FA:

Consider

$A \to aB \mid bA \mid b$

$B \to aC \mid bB$

$C \to aA \mid bC \mid a$



Finding final state:

Observe productions $A \to bA \mid b$ & $C \to aA \mid a$

$A \to b$ is obtained by ~~adding~~ using $A \to \epsilon$ in $A \to bA$

$C \to a$ is obtained by using $A \to \epsilon$ in $C \to aA$

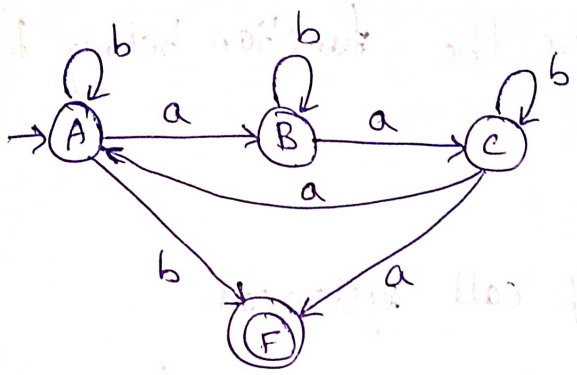
Since $A \to \epsilon$ is used, so A is the final state

RG to FA :

$A \rightarrow aB \mid bA \mid b$

$B \rightarrow aC \mid bB$

$C \rightarrow aA \mid bC \mid a$



27/12/20

Operating System.

System Calls:

A function call is call within the same process

System call is function call where the function belong to OS.

Implementation:

- Some systems may directly call system call.
- Some ~~systems~~ Higher level language provides functions which inturn calls system calls
- Some systems may compile program into assembly language where machine instns call system call.

→ System call requires a context switch (user mode to kernel mode)

parameter passing for system calls:

Using registers:

→ parameters are placed in register by user process. But this has a drawback that when large no of parameters are to be passed, ~~no~~ enough registers will not be present.

Using block of memory:

→ parameters are stored in a fixed memory location.

Stack:

→ parameters are stored on a stack.

Types of System Calls:

Process Control System Calls:

→ End, abort
→ load, execute
→ Create process, terminate process
→ Get process attributes, set process attribute
→ wait for time (sleep)
→ wait event, signal event (use to implement semaphore operations)
→ allocate and free memory

File manipulation system calls:

→ Create, delete
→ open, close
→ read, write, reposition (seek)
→ get file attributes, set file attributes

Device related System Calls

→ Request device, release device
→ Read, write, reposition
→ get device attributes, set device attributes
→ logically attach or detach devices (Enabling device drivers & disabling)

Information Maintainance System Calls:

→ Get time or date, set time or date
→ get system data, set system data
→ get process, file or device data
→ set process, file or device data

Communication related system calls:

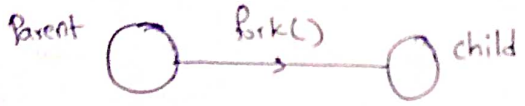
→ Create, delete communication connected

→ send, receive messages

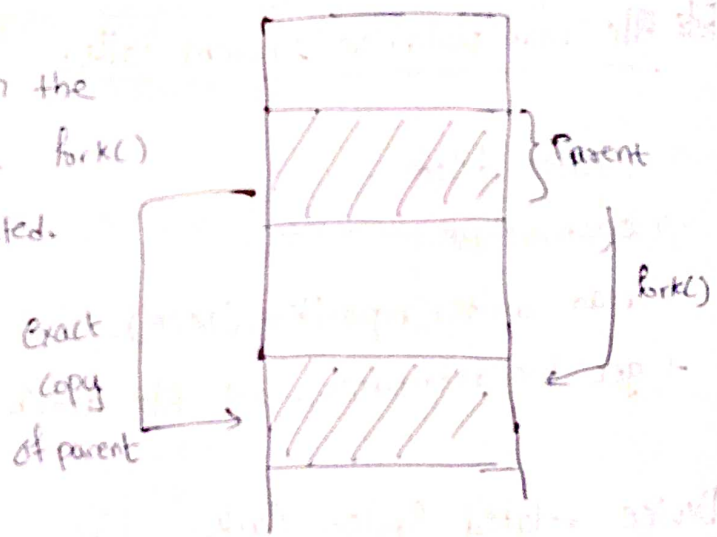
→ Transfer status information (acknowledgement etc.)

→ Attach or detach remote devices

Fork & execve:



→ Child process executes from the instn that is next to the fork() call from which child is created.



→ In server, to service multiple clients we use fork to create another server process.

→ But creating a new process requires context switch and also it has several overheads

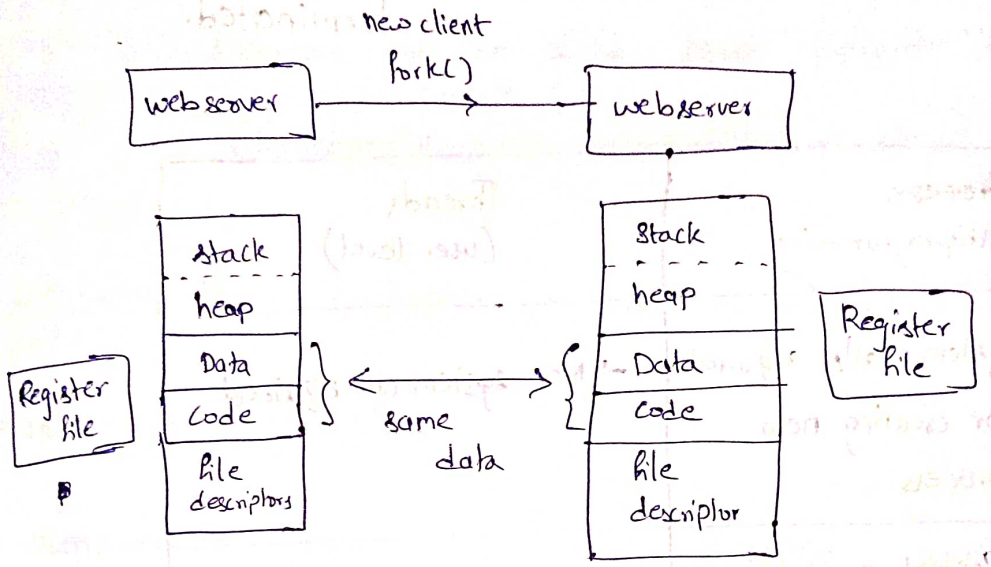
→ Return value:

fork() returns 0 to child process

child proc id to parent process

→ execve is used replace the child process created with fork call.

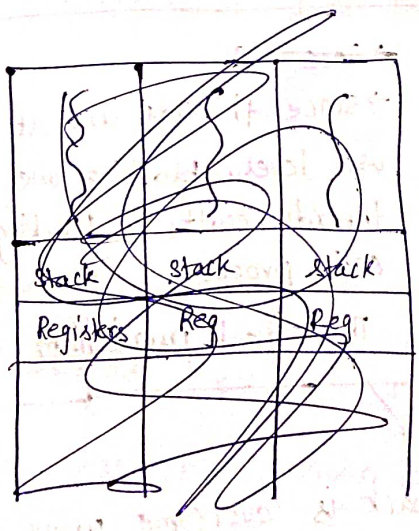
Disadvantage with processes:

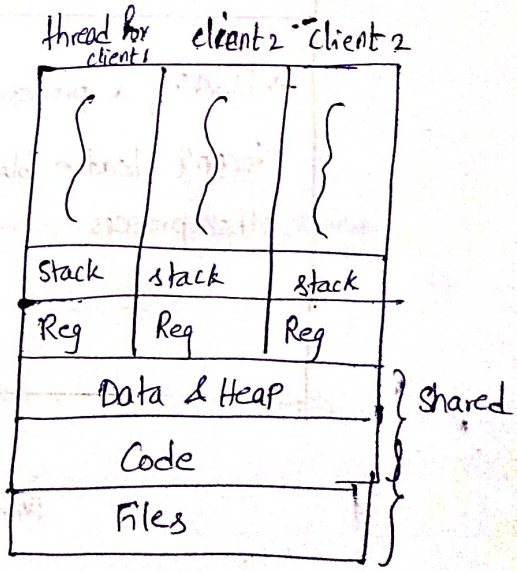


→ Same data & Code is stored multiple times

→ Also context switching is required.

Threads:





→ Threads can be created by user itself (if thread is user level thread)

→ Code, data, heap will shared among the threads.

→ Each threads has its own program counter.

→ Switching b/w threads doesn't need context switching

States in life cycle of thread: New, ready, running, blocked, terminated.

~~Process~~ Multiprogramming	Threads (user level)
→ System calls required for creating new process	→ No system call required.
→ Context switch	→ Register set switch
→ Different copies of code, data, heap	→ Same copy of code, data, heap
→ one ~~th~~ process can't access stack of other process	→ one thread can access stack of other thread
→ Blocking a process doesn't lead to block other process	~~→ Since H.~~ → Since threads are at user level, blocking one thread results in blocking entire process. This effects parallelism

This drawback is overcome using kernel level threads. So here ~~ke~~ OS creates the thread on behalf of the process.

Problems with user level threads:

→ Blocking system calls block entire task (process)

→ Unfair scheduling

i.e., a process with 100 threads and a process with 1 thread will be given same priority in scheduling (for example same time quantum in round robin scheduling)

To overcome these problems we use kernel level threads.

Kernel Level Threads:

→ whenever a thread has to be created, system call is made and kernel creates the thread.

→ Now kernel is aware of threads at user level and thus blocking a thread doesn't cause the entire task to block.

→ Here OS schedules threads rather than processes.

Thus fair scheduling is achieved.

Disadvantages:

→ Expensive compared to creation of user level threads ~~and maintaining~~ is ~~also expensive~~.

→ Switching a thread requires system call. However this switching is less costly compared to process switch.

AVL Tree Deletion:

6 cases: $R_0, R_1, R_{-1}, L_0, L_1, L_{-1}$

R:

→ R case occurs if deletion is present in the right side of imbalanced node.

→ L case occurs if deletion is present in the left side of imbalanced node.

→ In R case, if balance factor of left child of imbalanced node is i, we say it is R_i case.

→ In L case, if balance factor of right child of imbalanced node is i, we say it is L_i case.

Deletion:

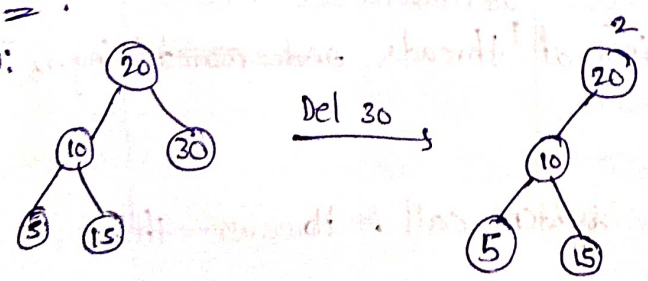
→ Delete node like we do in BST.

→ Now check for imbalances (in the path of deletion).

→ Based on the case under which the deletion falls, apply appropriate rotations.

R_0:

Ex:



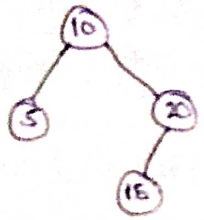
Imbalance at 20 & deletion is to right of 20. ∴ R case.

Balance factor of left child of 20 (i.e., 10) is 0

∴ R_0 case.

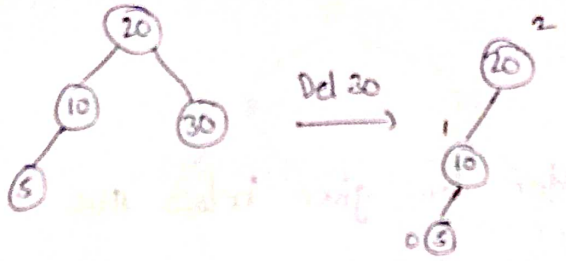
Soln to R_0 case is LL rotation.
i.e., single right rotation.

∴



For R_0, LR can also be applied.
(But we generally use LL cuz its easy)

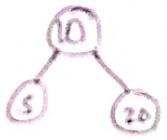
R_1 :



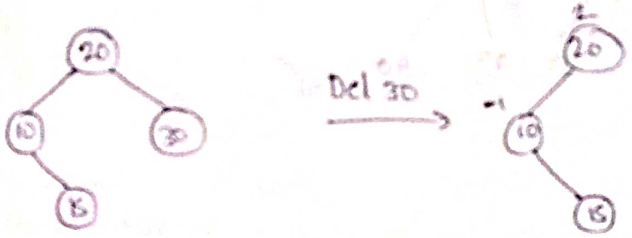
∴ R_1 case

soln to R_1 : LL rotation at imbalanced node.

∴

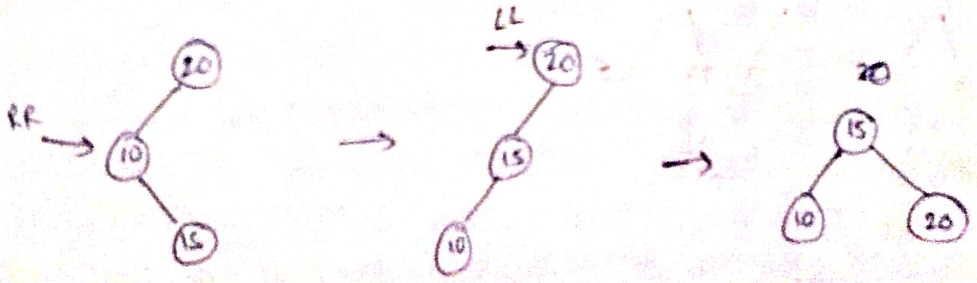


R_{-1} :



∴ R_{-1} case

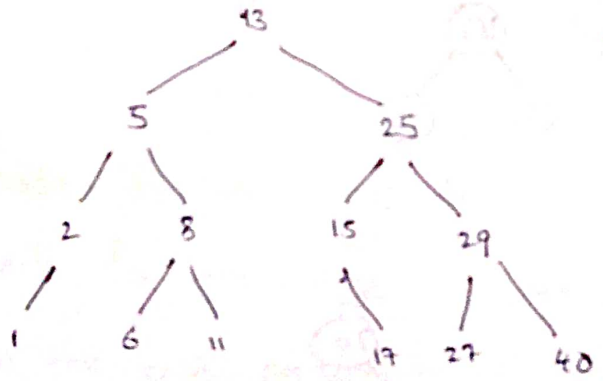
Soln to R_{-1} : LR rotation.



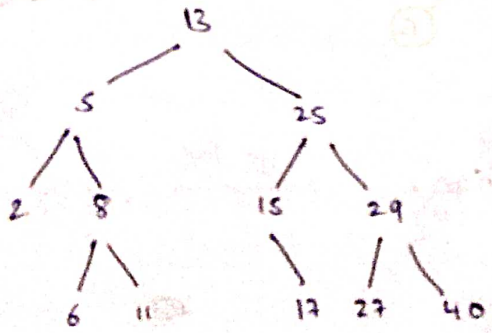
Case	Soln
R_0	LL
R_1	LL
R_{-1}	LR
L_0	RR
L_1	RL
L_{-1}	RR

Eg : Delete keys in the given order for given below AVL tree

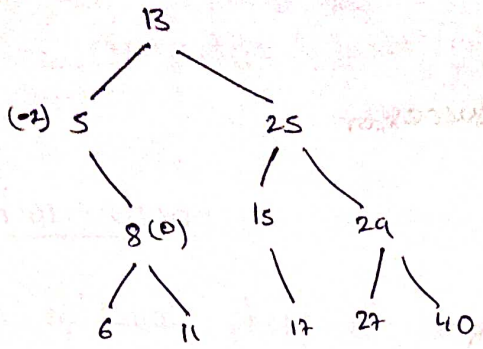
1, 2, 11, 17, 5, 8, 13, 6, 15



:

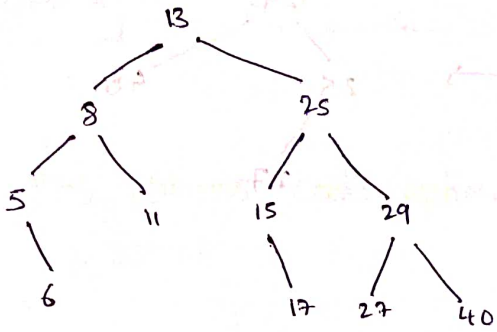


2 :

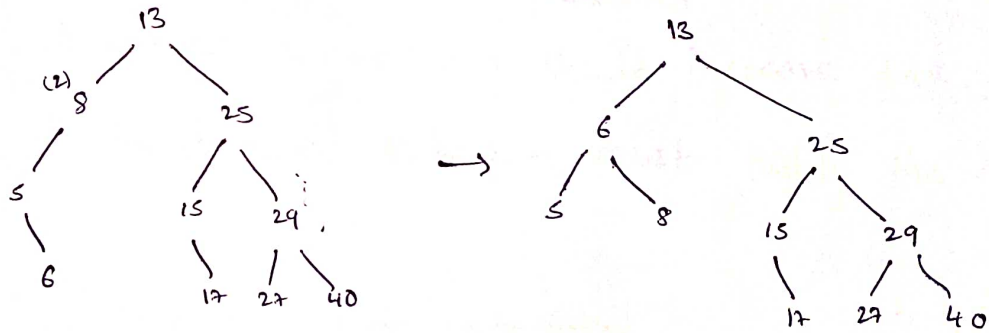


∴ L_0 case

∴ Apply RR rotation



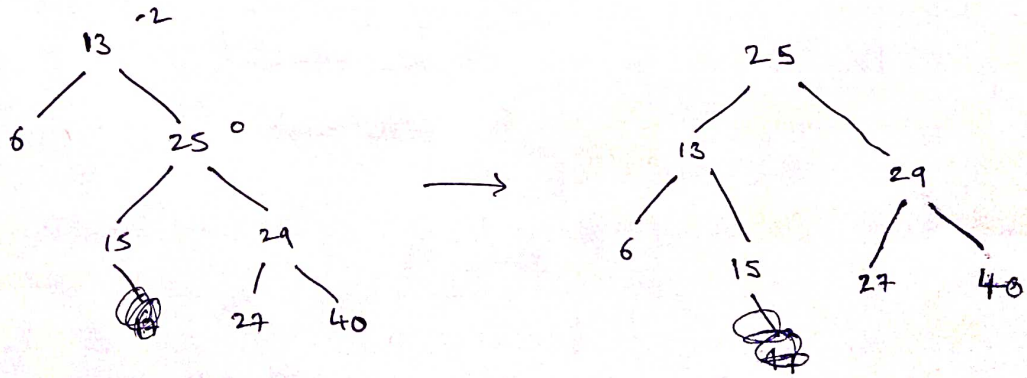
11 :



17 : Direct deletion

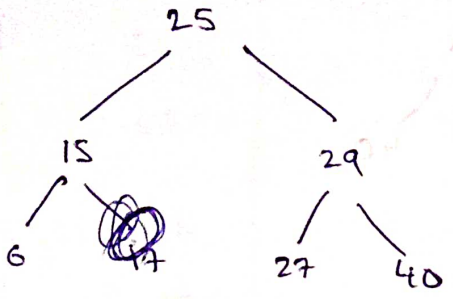
5 : Direct deletion

8 :

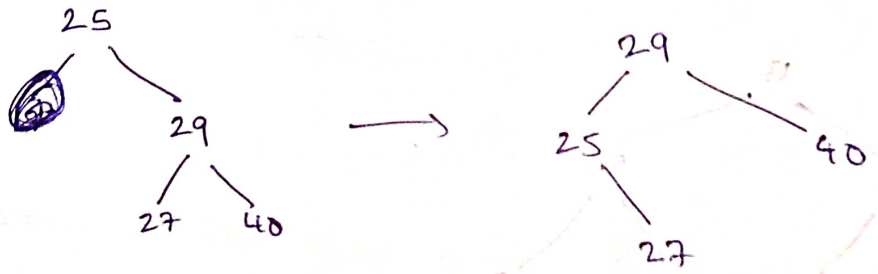


13:

replace 13 with lowest successor



6, 15:

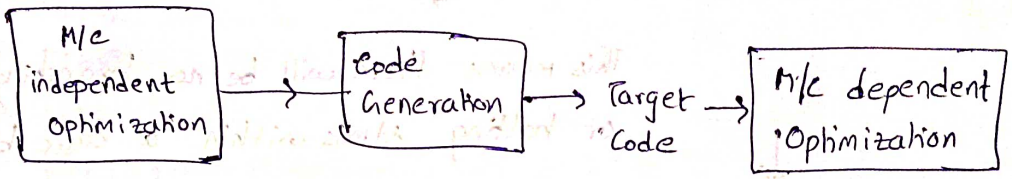


16/01/21

Compiler Design

Code Optimization:

→ It is an optional phase of compilation.



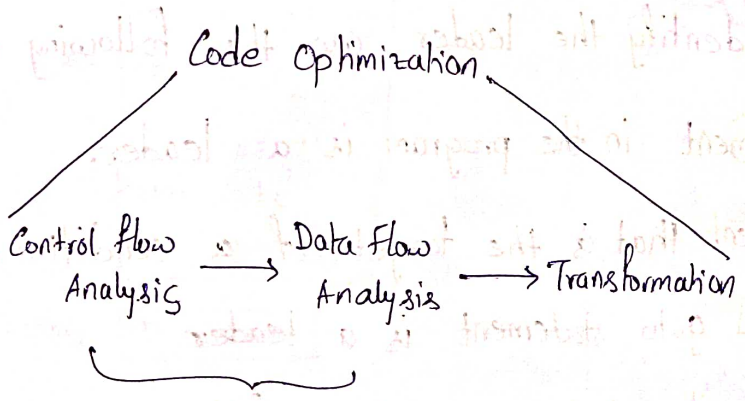
→ A compiler that performs code optimization is called optimizing compiler.

→ Properties of optimization:

(i) Functionality must be preserved.

(ii) ~~Optimat~~ optimization should increase the speed.

(iii) Transformation techniques must justify the effort.



- done using flow graph

Basic block and Flowgraph:

→ A graphical representation of three address code is called flow graph.

→ A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end

This means there will be no branching or halting stmts within a basic block.

Eg:

```
t1 = c*d
t2 = t1/e
t3 = b+t2
a = t3
```

Procedure to identify the basic block:

Given a TAC, first identify the "leader" statements and group the leader statement with statements upto the next leader. To identify the leader, use the following rules:

(i) First statement in the program is a leader.

(ii) Any statement that is the target of a conditional or unconditional goto statement is a leader

(iii) Any statement that immediately follows a conditional / unconditional goto statement is a leader.

Ex :

1. i = 0 → leader
2. n = 10
3. $t_1 = n-1$ → leader
4. if $i > t_1$ goto (12)
5. $t_2 = i * i$ → leader
6. $t_3 = 4 * i$
7. $t_4 = a[t_3]$
8. $t_4 = t_2$
9. $t_5 = i+1$
10. $i = t_5$
11. goto (3)
12. return → leader

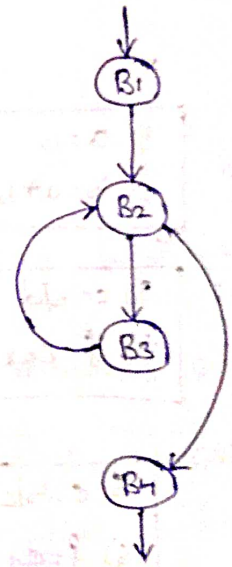
B1
```
1. i = 0
2. n = 10
```

B2
```
3. t1 = n-1
4. if i > t1 goto (12)
```

B3
```
5. t2 = i * i
6. t3 = 4 * i
7. t4 = a[t3]
8. t4 = t2
9. t5 = i+1
10. i = t5
11. goto (3)
```

B4
```
12. return
```

Flow graph.



~~Flow~~

Note :

→ Cycle in Control flow graph implies the existence of loop in the program.

Q: Consider below TAC.

```
     1. a=0          ---------> leader
     2. b=a*b
L1:  3. c=b/d        ---------> leader
     4. if c<x goto L2
     5. e=b/c        ---------> leader
     6. f=e+1
L2:  7. g=f          ---------> leader
     8. h=t-3
     9. if e>0 goto L3
     10. goto L1     ---------> leader
L3:  11. return      ---------> leader
```

Find no of leaders, no of blocks, no of edges in flow graph.

B1
```
1) a=0
2) b=a*b
```

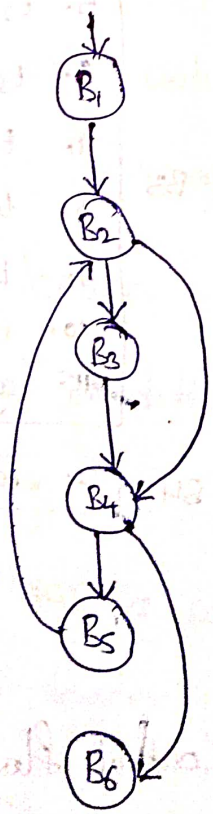
B2
```
L1: 3) c=b/d
    4) if c<x goto L2
```

B3
```
5) e=b/c
6) f=e+1
```

B4
```
L2: 7) g=f
    8) h=t-3
    9) if e>0 goto L3
```

B5
```
10) goto L1
```

B6
```
L3: return
```



∴ no of leaders = no of blocks = 6

no of edges in flow graph = 7

Classification of code optimization:

M/C Dependent

→ optimization based on target machine. i.e., depending on the ISA, addressing modes etc.

→ This optimization is performed on target code.

M/C Independent

→ Here optimation is based only on program stmts & program logic.

→ This optimization is performed on intermediate code.

Scope of Optimization:

(i) Local Scope: Considers only the statements of the basic block that is being optimized. This is called local optimization.

(ii) Global Scope: Considers the statements of blocks other than the block that is being optimized. This is called global optimization.

Local Optimization:

1. Common Subexpression Elimination
2. Copy Propagation
3. Dead code elimination
4. Constant Folding

} Function preserving transformations

1. Common Subexpression Elimination:

An expression E is called a common subexpression if it was previously computed and the values of variables in E have not changed since the previous computation.

Eg:

$t_1 = 4 * i$
$n = a[t_1]$
$t_2 = 4 * i$ → replace with t_1
$t_3 = 4 * j$
$t_4 = a[t_3]$
$a[t_2] = t_4$
$t_5 = 4 * j$ → replace with t_3
$a[t_5] = n$

⇓ Common Subexpression Elimination

$t_1 = 4 * i$
$n = a[t_1]$
$t_2 = t_1$
$t_3 = 4 * j$
$t_4 = a[t_3]$
$a[t_2] = t_4$
$t_5 = t_3$
$a[t_5] = n$

Eg: Consider following basic block

$a = b+c$		$a = b+c$
$c = a+d$	CSE →	$c = a+d$
$e = b+c$		$e = b+c$
$f = a+d$		$f = c$

Here $b+c$ is not a Common Subexpression

2. Copy Propagation:

It means use of one variable instead of another

Eg: $x=y$ is called copy statement

In copy ~~proga~~ propagation, we use y inplace of x whenever possible.

$n=t_1$
$a[t_1]=t_2$
$a[t_3]=n$

$\xrightarrow{\text{copy propagation}}$

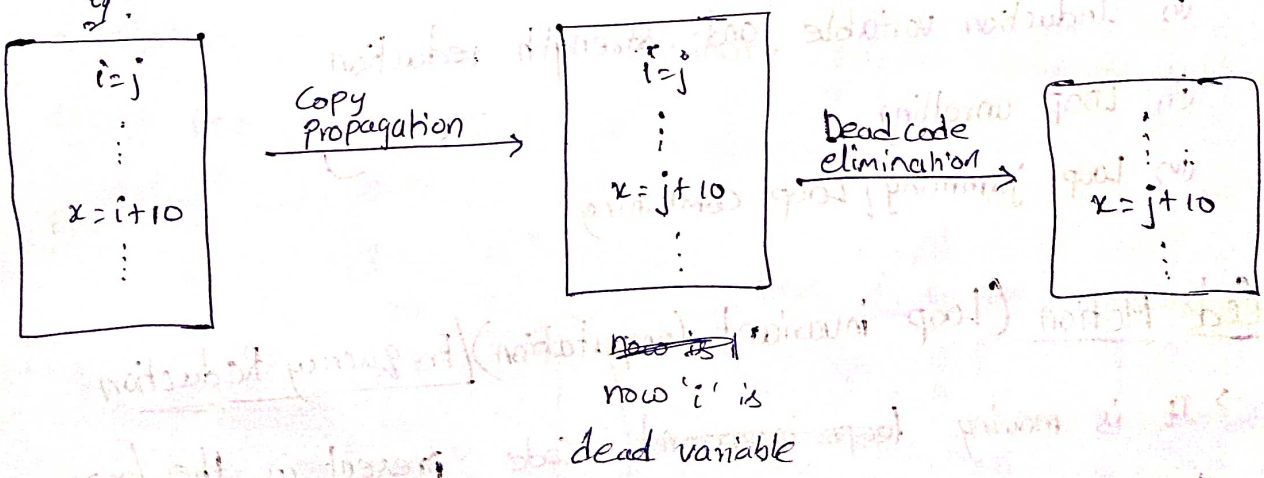
$n=t_1$
$a[t_1]=t_2$
$a[t_3]=t_1$

→ Copy propagation alone may not give any benefits. But it helps ~~for~~ other techniques like dead code elimination.

3. Dead code Elimination:

A variable is said to be live in the program, if its value can be used subsequently. otherwise it is dead at that point.

Eg:



The code that has no impact on result is called dead code

Eg:

```
i=0
if (i==1) ---- always false
{
    ≡ } → Dead Code
}
```

4. <u>Constant Folding</u>:

Replacing expression consisting of constants with their final value.

Eg: $i = 5*6+a \xrightarrow{\text{Constant folding}} i = 30+a$

→ Constant folding replaces runtime computation with compile time computation.

Loop Optimization:

i) Code motion (Loop invariant computation) / Frequency Reduction

ii) Induction variable and strength reduction

iii) Loop unrolling

iv) Loop jamming / Loop combining

i) <u>Code Motion</u> (Loop invariant Computation) / <u>Frequency Reduction</u>

→ It is moving ~~loop invariant~~ code present in the loop to outside of loop.

Eg:

```
while (i<= max-1)
{
    sum = sum+a[i];
    i+=1;
}
```

Here value of max-1 is not changed, but still max-1 is evaluated for every iteration

⇓ Code motion

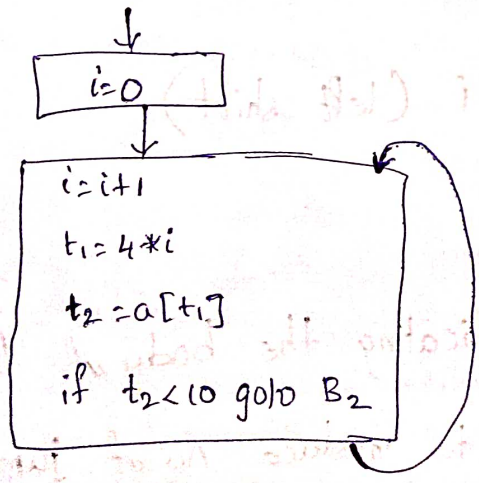
```
n=max-1;
while( i<=n)
{
    sum = sum+a[i];
    i+=1;
}
```

ii) Induction variable & Reduction in strength

A variable x is called an induction variable of loop, if the value of variable gets changed every time.

→ If there is more than one induction variable, we can find a relation b/w them and eliminate all the induction variables except one.

Eg:



Here i, t_1 are induction variables

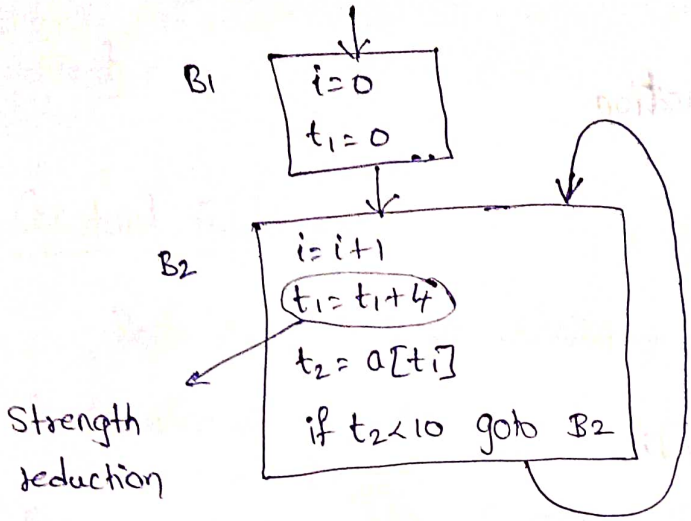
$i = 1, 2, 3, 4, \ldots$

$t_1 = 4, 8, 12, 16, \ldots$

So here we can replace $t_1 = 4*i$ with $t_1 = t_1 + 4$

along with this we need to initialize t_1 to 0.

∴ The optimized code is



→ The process of replacing high strength operation (Eg: *) with low strength operation (Eg: +) is called strength reduction.

Eg: $a*2 \xrightarrow{\text{strength reduction}} a+a$

$a^2 \longrightarrow a*a$

$a/2 \longrightarrow a*0.5$

(or)

right shift (integer division)

$a*2^i \longrightarrow a<<i$ (left shift).

(iii) Loop unrolling:

It is the process of duplicating the body of loop multiple times. This is dones to reduce no of jumps and no of tests dones.

```
i=1
while (i<=100)
{
    a[i]=b[i];
    i=i+1;
}
```

no of jumps = 100

no of tests = 101 (i=1 to i=101)

⇓ Loop unrolling

```
i=1
while (i<=100)
{
    a[i]=b[i];
    i=i+1;
    a[i]=b[i];
    i=i+1;
}
```

no of jumps = 50

no of tests = 51

However a care must be taken while duplication loop body

For example if we duplicate 3 times

```
i=1
while(i<=100) .. i= 1,4,7,... 100
{
    a[i]=b[i];
    i=i+1;
    a[i]=b[i];
    i=i+1;
    a[i]=b[i];
    i=i+1;
}
```

when i=100, condition evaluates to true then

```
a[100]=b[100]
i=100+1
a[101]=b[101]
i=101+1
a[102]=b[102]
i=102
```

These 2 lines affect functionality

∴ This optimization is wrong.

(iv) Loop Jamming / Loop Combining

In this method, several loops are merged to one loop.

Eg: we do this when two or more loops run for same amount of times and the loops are independent (i.e., ~~using~~ not having any common variables)

Eg.:

```
for (i=0; i<n; i++)
    a[i]=0;
for (j=0; j<n; j++)
    b[j]=1;
```

no of jumps = 2n
no of tests = 2(n+1)

⇓ Code Jamming

```
for(k=0; k<n; k++)
{
    a[k]=0;
    b[k]=1;
}
```

no of jumps = n
no of tests = n+1

Global Optimization:

→ Global optimization requires control flow analysis and data flow analysis. Based on these analysis we apply program transformation.

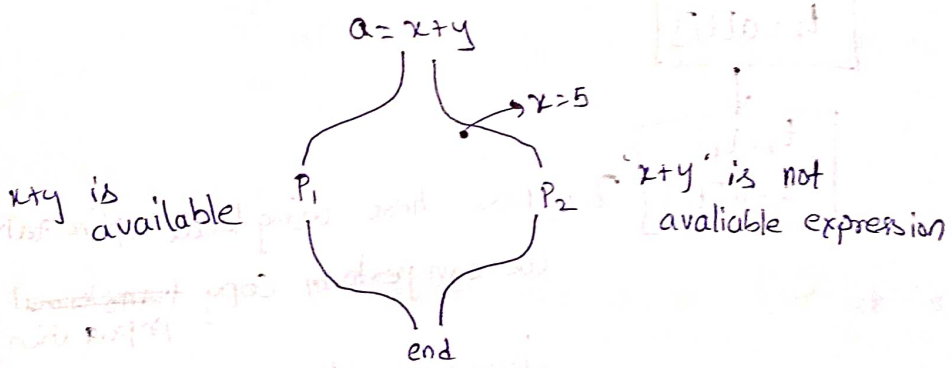
Dataflow Analysis: It is a process in which values are computed using dataflow properties.

The dataflow properties are:

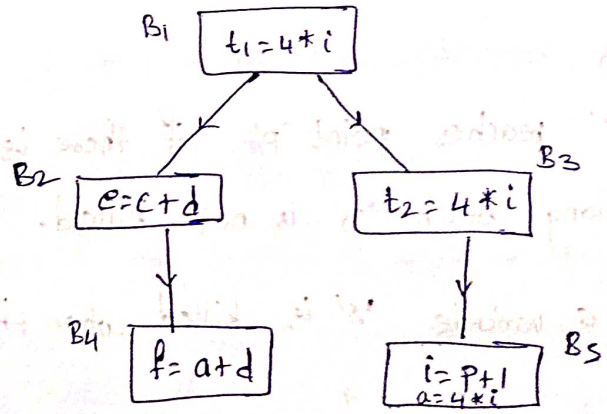
(i) Available Expression
(ii) Reaching Definition
(iii) Live variable
(iv) Busy Expression

(i) Available Expression:

An expression $x+y$ is available at point 'p' if every path from the node to 'p' evaluates '$x+y$' and after the last such evaluation prior to reaching p, there are not subsequent assignments to x or y.



Eg:

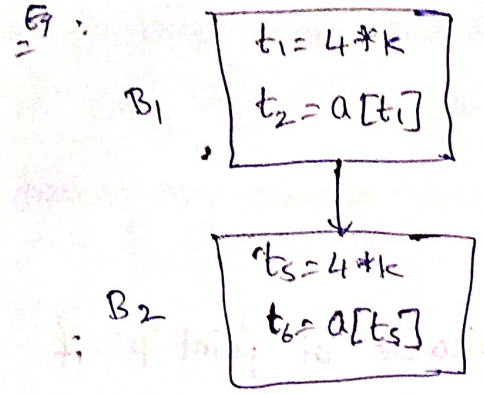


Here the expression $4*i$ is available to blocks B_2, B_3, B_4 but not to the block B_5

Adv:

→ available expressions can be used to eliminate common subexpressions present across different blocks.

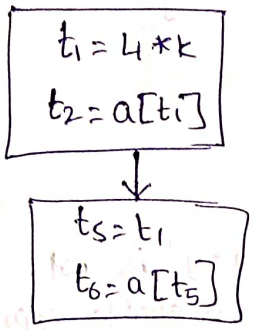
Eg:



4*k is available expressin to B_2.

∴ 4*k in B_2 is Common subexpression

⇓ CSE based on available expression



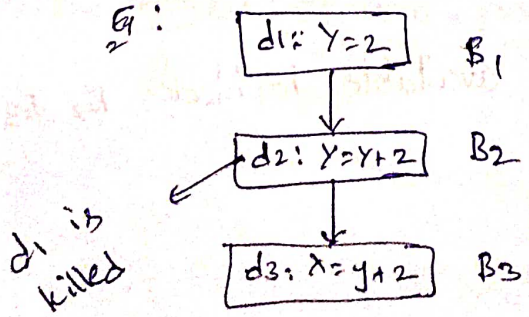
→ Now here using local optimization we can perform copy ~~transformal~~ Propagation and dead code elimination if possible.

(ii) Reaching Definition:

A definition 'D' reaches point 'p' if there is a path from D to P. along which 'D' is not killed.

A definition 'D' of a variable 'x' is killed when there is a redefinition of x.

Eg:



→ d_1 is ~~reacha~~ reachable, to B_2 but not to B_3

→ d_2 is reaching definition to B_3

Adv: Reaching definition is used in Copy propagation.

Eg: d_1 ← $x = t_3$ → Definition of x (copy statement)

B_1: $x = t_3$, $a[t_1] = t_2$, $a[t_4] = x$

d_1 is ~~available~~ reaching defn to B_2

B_2: $y = x + 3$, $a[t_5] = 4$

copy Prop. ⟹

$x = t_3$, $a[t_1] = t_2$, $a[t_4] = t_3$

$y = t_3 + 3$, $a[t_5] = 4$

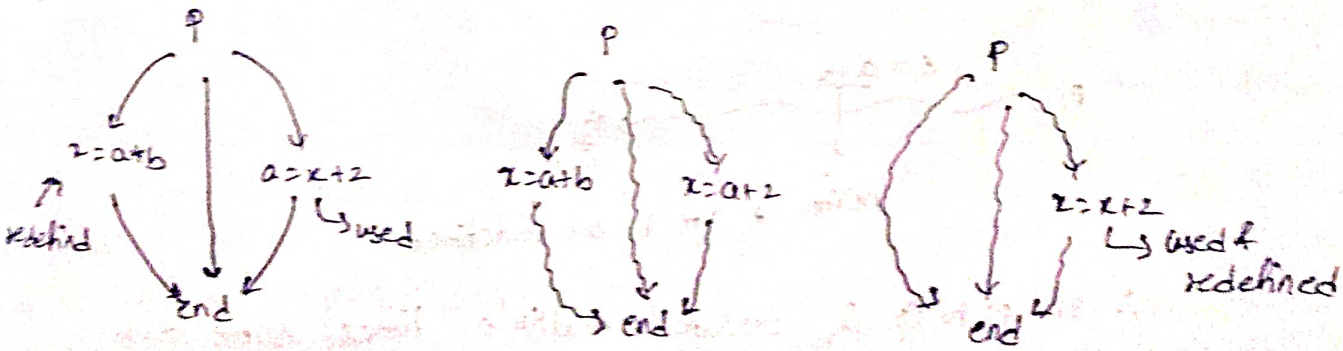
Now we can even perform dead code elimination.

(iii) **Live Variable:**

A variable 'x' is ~~called~~ said to be live at a point 'p', if there is some path from 'p' to the exit, along which the value of x is used before it is redefined. Otherwise a variable is said to be dead at the point.

→ The concept of live variable is used in register allocation.

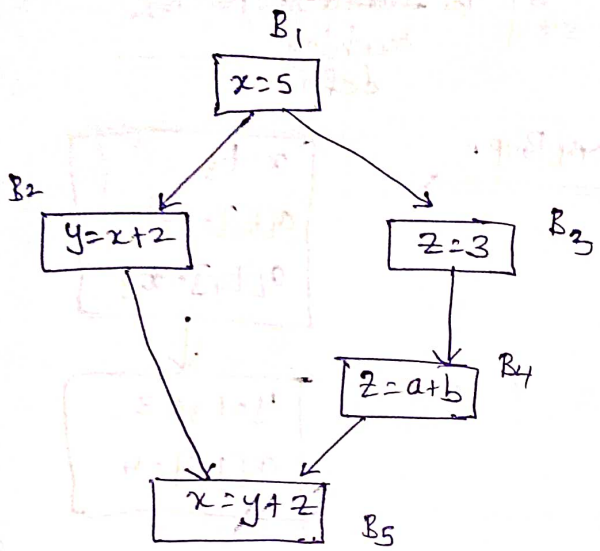
Eg:



∴ Here x is live at P Here x is dead at P Here x is live at P.

→ If a variable is not live ~~and~~ at some point and if a register is ~~allocated~~ allocated, then the register can be deallocated.

Eg:



At any stmt
a=b+c,
a is dead and
b,c becomes live
if they are not

At stmt
a=a+b,
~~a is~~
a,b are live

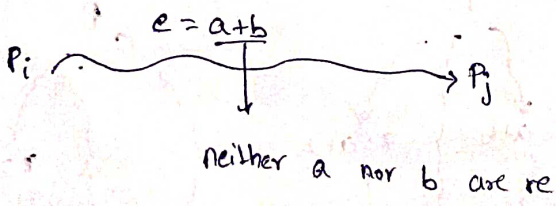
x is live at B1, B2

x is not live at B3, B4, B5

→ Live variables are useful in register allocation & dead code elimination.

ii) Busy Expression:

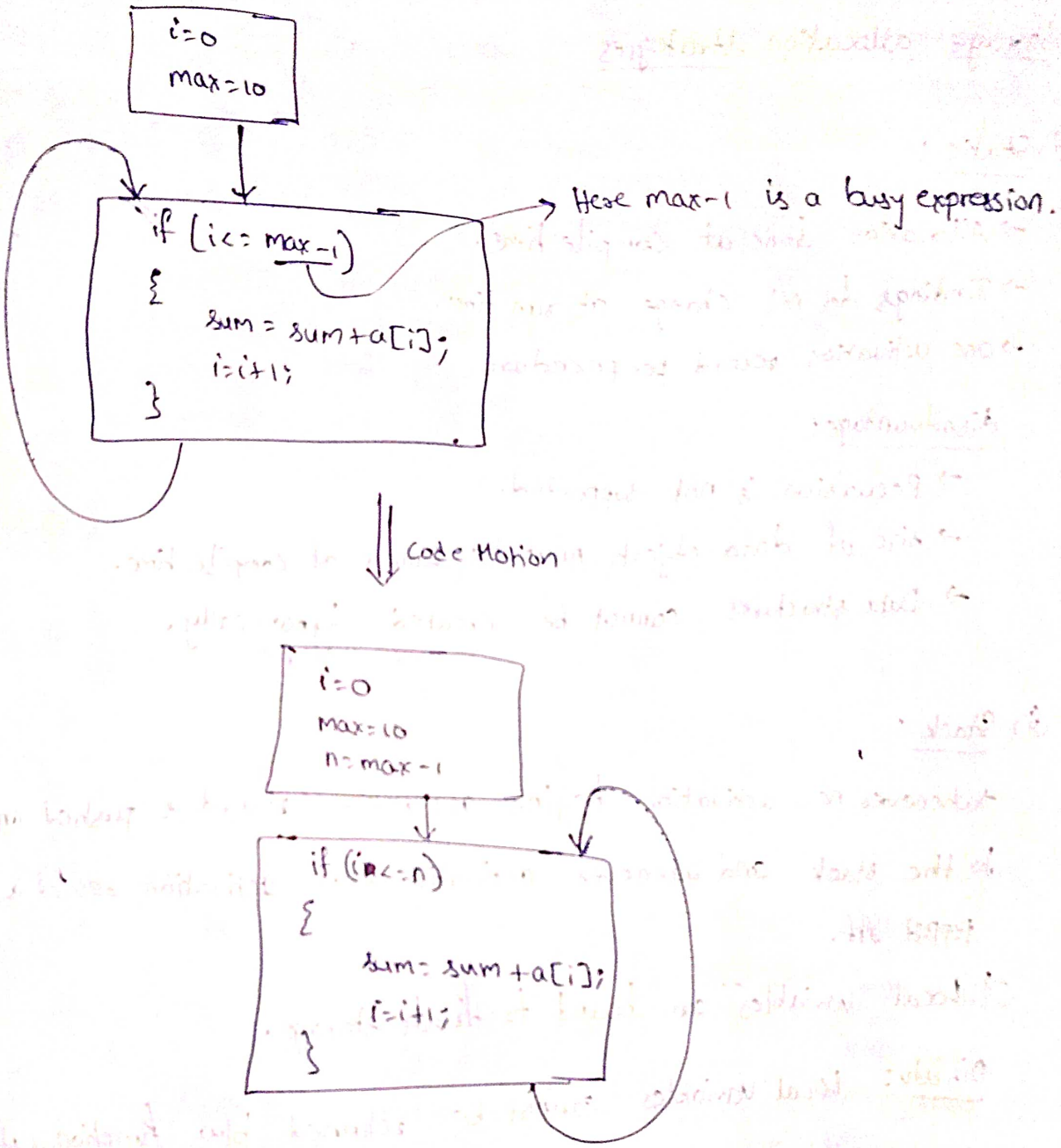
→ An ~~expt~~ expression 'e' is said to be busy along some path P_i to P_j iff an evaluation of e exists along some path from P_i to P_j and no definition of any operand exists before its evaluation along the path.



∴ If P_i to P_j is executed multiple times, same a+b is evaluated multiple time. So we can put a+b outside the path.

∴ Busy expressions are used in code motion.

Eg:



Runtime Environments

Storage allocation strategies

(i) Static :

→ Allocation done at compile time.
→ Bindings do not change at run time.
→ one activation record per procedure.

disadvantage:

→ Recursion is not supported.
→ size of data objects must be known at compile time.
→ Data structures cannot be created dynamically.

(ii) Stack :

→ whenever new activation begins, activation record is pushed on to the stack and whenever activation ends activation record is popped off.

→ Local variables are bound to fresh storage.

Disadv: Local variables cannot be retained b/w function calls.

(iii) Heaps :

Allocation & Deallocation can be done in any order.

→ Heap Managements is an overhead.
→ Lifetime of activation is arbitrary.
→ C used first fit for heap management.

Machine Dependent Optimization

- → Register Allocation.
- → Use of addressing modes.
- → Peephole optimization
  - Redundant load/store
  - Flow of Control optimization
  - Strength reduction
  - Use of M/C idioms.

Redundant load & store elimination:

```
x = y+z  ---->  mov y,R0
                add z,R0
                mov R0,x
```

```
a = b+c  ----------  mov b,R0
                     add c,R0
                     mov R0,a
d = a+e              mov a,R0
                     add e,R0
                     mov R0,d
```

→ redundant load & store

These 2 lines can be eliminated.

and we add mov R0,a at the end.

Flow of Control optimization

Avoid jumps on jump

```
jump L1
 :
L1: jump L2
 :
L2: jump L3
 :
L3:
```

→ jump L1

Dead Code Elimination

```
#define x 0
if(x)
{
  ==========
}
```

} Dead code

Use of M/C idioms:

$i = i + 1$

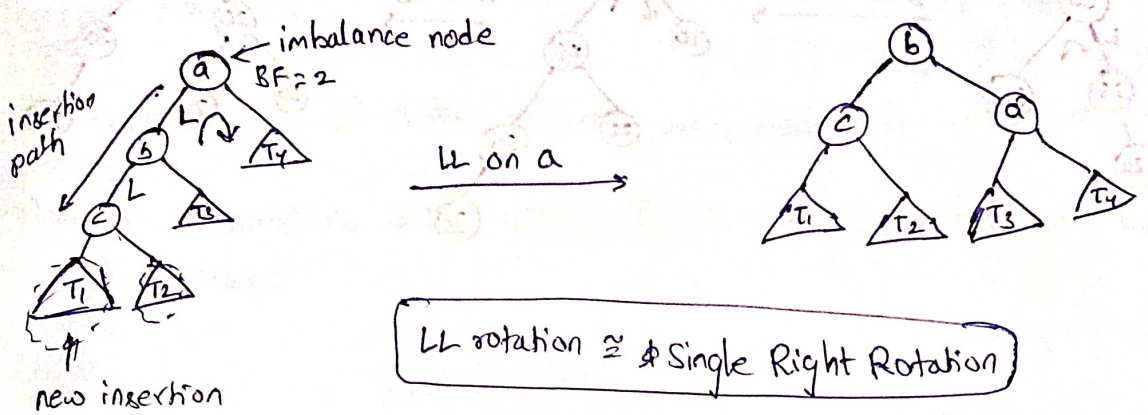
```
mov R0, i
add R0, 1
mov i, R0
```

→ Can be replaced with 'inc i' if the processor is capable

AVL Tree Insertion

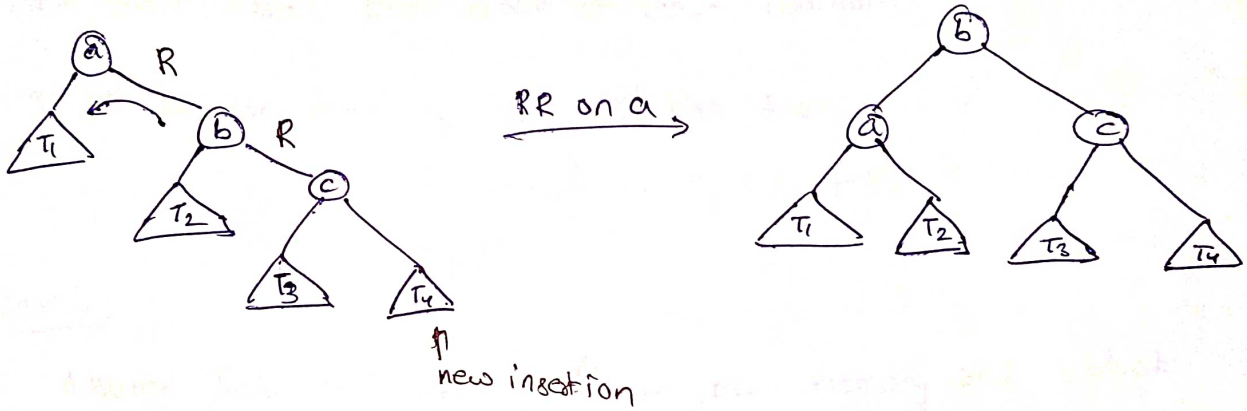
Types of imbalances:

LL imbalance:



LL rotation ≅ Single Right Rotation

RR Imbalance:

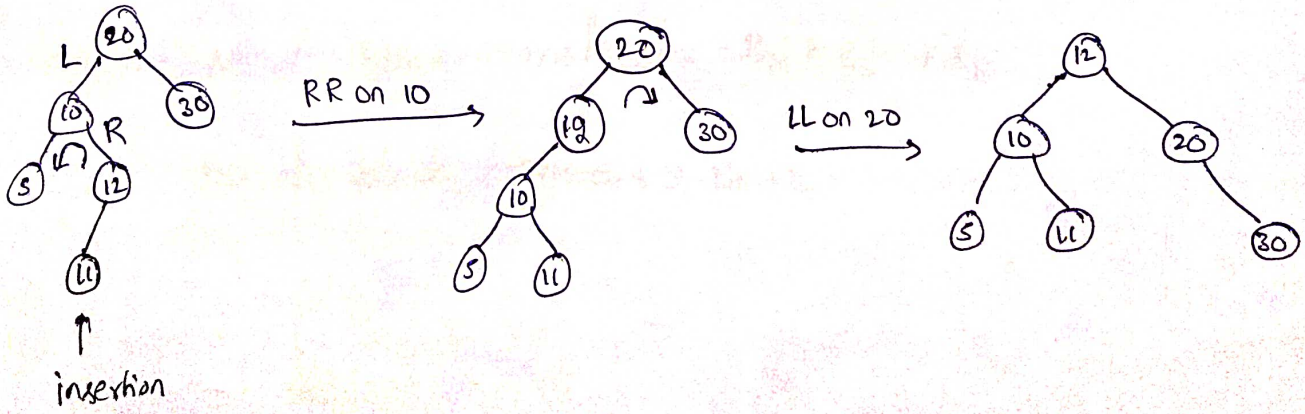


RR – single left rotation

LR rotation:

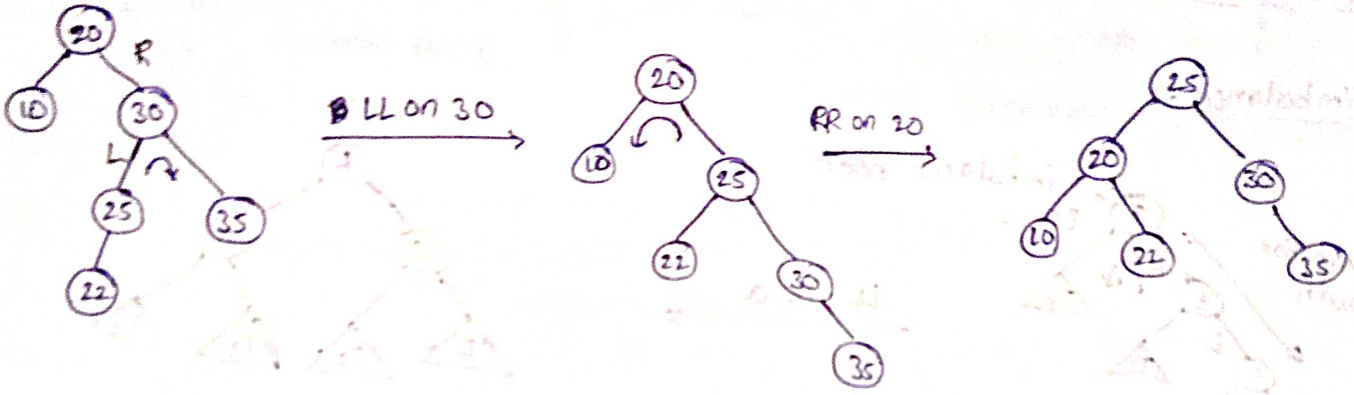
↳ RR on left child of imbalanced followed by LL on imbalanced node

↓ single left ↓ single right



RL Rotation:

↳ LL rotation on right child of imbalanced node followed by RR on imbalanced.



on imbalanced.

02/02/21

DBMS

Join Algorithms :

i) Nested Loop Join:

For each tuple of outer relation, we compare it with inner relation.

Consider 2 relations S & R with T_S, T_R no of tuples and B_S, B_R no of blocks.

Case (i) :

Assume only two blocks of main memory are available such that each block stores one block of each relation.

If 'R' is considered as outer relation then,

no of block transfers = $T_R * B_S + B_R$

Case (ii) :

Assume Relation S entirely fits in main memory and 1 block is available for relation R.

no of block transfers = $B_R + B_S$

(ii) Block Nested Loop Join

For each block of outer relation, all the tuples of a block of inner join are compared.

Case (i) : No of block transfers = $B_R * B_S + B_R$

Case (ii) : no of block transfers = $B_R + B_S$

Log based recovery in DBMS

Log & log record:

→ Log is collection of log records.

→ Any operation performed on database is stored in logs.

→ There are 4 types of log records

(i) Update log record: $\langle T_i, D, O, N \rangle$

- T_i → transaction identifier
- D → Data item
- O → old value
- N → New value

(ii) $\langle T_i$ start$\rangle$

(iii) $\langle T_i$ commit$\rangle$

(iv) $\langle T_i$ abort$\rangle$

Checkpoint:

→ It is difficult to examine entire log for recovery. So we use checkpoints.

→ A record of form $\langle$checkpoint $L\rangle$ is kept in the log where L is list of transactions active at that time.

→ For all the transactions that are commited before checkpoint, their modifications are written into database either before checkpoint or as part of checkpoint.

Recovery:

→ Recovery is done from last checkpoint

Redo list: If a transaction has $\langle T_i$ start$\rangle$ and $\langle T_i$ commit$\rangle$ or $\langle T_i$ abort$\rangle$ / and active transactions at checkpoint.

Undo list: If a transaction has only $\langle T_i$ start$\rangle$ then we add it to undo list.

→ For recovery we include all the transactions in list 'L' and transactions that started later.

Note:

Database can be modified in 2 ways:

i) Deferred modification technique: modify database only after transaction ~~par~~ commits partially.

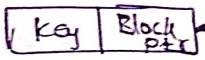
ii) Immediate Modification technique: Modify immediately.

Clustered Index:

→ On non key, ordered

→ sparse index.

→ Index record:



→ points to block containing the key and key has to be searched within the block.

B-Tree Deletion:

Case (i): Deletion from leaf

- More than min no of key → Delete directly
- Min no of keys → Borrow from immediate left/right sibling
- sibling don't have sufficient key
  - (i) Delete
  - (ii) Merge with immediate left of right sibling.
  - (iii) Bring the common parent down

If bringing common parent down reduces no of keys to less than minimum no of key req in parent node, then parent node should also borrow key from siblings or from it parent.

Case (ii): Deletion from internal node

- Replace with inorder succeccory if it has more than min no of keys
- Replace with inorder predecessor if it has more than min no of keys
- If neither of successor and predec has sufficient keys the replace with predec or succ and recursively apply delete on predec or succ.

→ At any point of time root can have any no of keys ≥ 1.

06/02/21

Optimal Cost Binary Search Tree

- Let internal nodes be $a_1, a_2, \ldots a_n$ and external nodes be $q_0, q_1, q_2 \ldots q_n$. If 'x' is the element searched,

  q_0 corresponds to $x < a_1$

  q_1 corresponds to $a_1 < x < a_2$

  $\vdots$

  q_i corresponds to $a_i < x < a_{i+1}$

  $\vdots$

  q_n corresponds to $a_n < x$

- Required cost is denoted as $C[0,n]$ where 0 and n represents lowest and highest external nodes.

- If we consider $C[i,j]$

  ~~the key are~~ a_i

  the external nodes are $q_i, q_{i+1}, q_{i+2}, \ldots q_j$

  internal nodes are $a_{i+1}, a_{i+2}, \ldots a_j$

$$\therefore C[i,j] = \min_{\substack{i \le k \le j \\ \downarrow \\ \text{root}}} \left\{ C[i,k-1] + C[k,j] \right\} + w[i,j]$$

where $w[i,j] = \sum_{r=i}^{j} q_r + \sum_{r=i+1}^{j} p_r$

$\Rightarrow w[i,j] = w[i,j-1] + p_j + q_j$

$\Rightarrow w[i,i] = q_i$

$* \; C[i,i] = 0$

Eg: keys: $\{10, 20, 30, 40\}$

$p_i : \{3, 3, 1, 1\}$ → These are frequencies

$q_i : \{2, 3, 1, 1, 1\}$

Now we need to find $C[0,4]$

$j-i=0$	$w_{00}=2$ $C_{00}=0$ $r_{00}=0$	$w_{11}=3$ $C_{11}=0$ $r_{11}=0$	$w_{22}=1$ $C_{22}=0$ $r_{22}=0$	$w_{33}=1$ $C_{33}=0$ $r_{33}=0$	$w_{44}=1$ $C_{44}=0$ $r_{44}=0$
$j-i=1$	$w_{01}=8$ $C_{01}=8$ $r_{01}=1$	$w_{12}=7$ $C_{12}=7$ $r_{12}=2$	$w_{23}=3$ $C_{23}=3$ $r_{23}=3$	$w_{34}=3$ $C_{34}=3$ $r_{34}=4$	
$j-i=2$	$w_{02}=12$ $C_{02}=19$ $r_{02}=1$	$w_{13}=9$ $C_{13}=12$ $r_{13}=2$	$w_{24}=5$ $C_{24}=8$ $r_{24}=3$		
$j-i=3$	$w_{03}=14$ $C_{03}=25$ $r_{03}=2$	$w_{14}=11$ $C_{14}=19$ $r_{14}=2$			
$j-i=4$	$w_{04}=16$ $C_{04}=32$ $r_{04}=2$				

~~wfo~~ $w_{01} = w_{00} + p_1 + q_1 = 2+3+3 = 8$

$w_{12} = w_{11} + p_2 + q_2 = 3+3+1$

$w_{23} = w_{22} + p_3 + q_3 = 1+1+1 = 3$

$\vdots$

$w_{02} = w_{01} + p_2 + q_2 = 8+3+1 = 12$

$\forall i \; C[i,i] = 0$

$\forall i \; r[i,i] = 0$

$\rightarrow C_{01} = \min\{C_{00} + C_{11}\} + w_{01}$

$= (0+0) + 8 = 8 \quad \Rightarrow r_{01} = 1$

$C_{12} = \min\{C_{11} + C_{22}\} + w_{12}$

$= 0 + 0 + 7$

$= 7 \quad \Rightarrow r_{12} = 2$

$\vdots$

$c[i, i+1] = w_{i,i+1}$

$r[i, i+1] = i+1$

$\rightarrow C_{02} = \min\limits_{0 < k \le 2}\{C_{00} + C_{12},\ C_{01} + C_{22}\} + w_{02}$

$= \min\{\underset{(k=1)}{0+7},\ \underset{(k=2)}{8+0}\} + 12$

$= 7 + 12 = 19$

$\Rightarrow r_{02} = 1$

$\rightarrow C_{13} = \min\{\overset{(k=2)}{C_{11} + C_{23}},\ \overset{(k=3)}{C_{12} + C_{33}}\} + w_{13}$

$= \min\{0 + 3,\ 7 + 0\} + 9$

$= 9 + 3 = 12$

$r_{13} = 2$

$\vdots$

$\rightarrow C_{03} = \min\{C_{00} + C_{13},\ C_{01} + C_{23},\ C_{02} + C_{33}\} + w_{03}$

$= \min\{\underset{(k=1)}{0 + 12},\ \underset{(k=2)}{8 + 3},\ \underset{(k=3)}{19 + 0}\} + 14$

$= 11 + 14 = 25$

$r_{03} = 2$

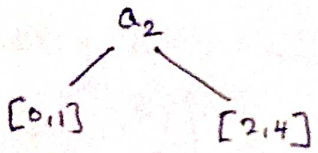
$\rightarrow C_{04} = \min\{C_{00} + C_{14},\ C_{01} + C_{24},\ C_{02} + C_{34},\ C_{03} + C_{44}\} + w_{04}$

$= \min\{0 + 19,\ 8 + 8,\ 19 + 3,\ 25 + 0\} + 16$

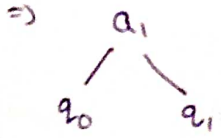
$= 16 + 16 = 32$

$r_{04} = 2$

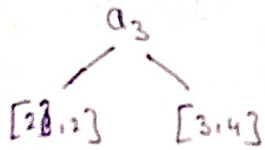
Since $r_{04} = 2$, the root is 2



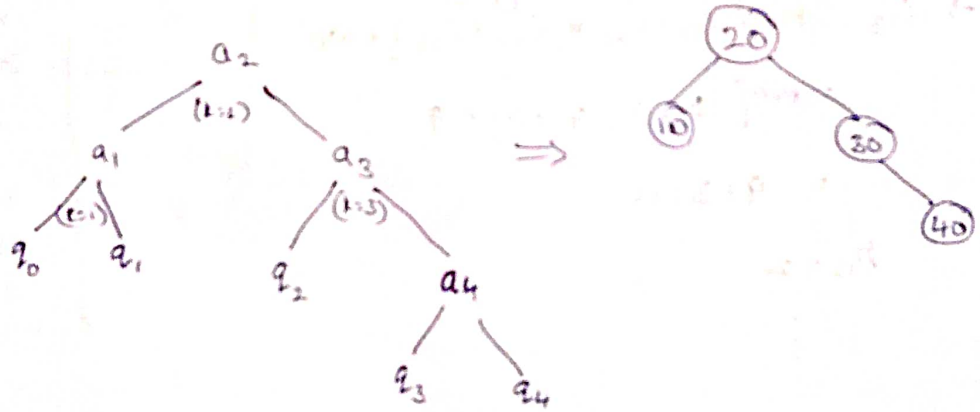
→ In [0,1] element, root is 1



→ In [2,4] element, root is 3



∴ The resultant tree is



Also minimum cost is $\frac{32}{16} = 2$

↳ (∵ we are considering frequencies)

a_2

~~1 — 0.33~~

1 1/3.

2 (live)

1 1/3 (doubt)

2

2

2

2 2/3

~~12+13~~

13 + 4

17

opt — 2

More — 2

2

TC — 1

Spa'Apt — 2

Algo graph — ~~2~~ (2)

prog — 2

~~Cache~~

prob — 2

DBMS blocks — 2

rank — 1

goto — 1

passage — 2

Algo graph — 2

Cach incle — 2

OS sync — 2

OS sync — 2

12+7 = 19

215

1sec ⟶ 2×10^6

~~c=1 f=9~~

$f=9$ $g=81$ $t=9^3$

$c=1$ $d=9$ $e=81$

$a=9$ $b=9$

$(01^*0)^*$

~~001~~

$(0+1(01^*0)^*1)^*$

$(0+1($

[0,1] element, root is 1 [2,4]

$$\frac{2^{31}}{2^{12}} = 2^{19} \times 8 = 2^{22} \rightarrow 2^{12}$$

$$\frac{2^{22}}{2^{12}} = 2^{10} \times 8 = 2^{13} \rightarrow 2^{3}$$

$$\frac{2^{13}}{2^{12}} = 2^{1} \times 8 = 2^{4}$$

1 1/3

2

2 2/3

2

2

2

2

2 2/3

2

2

22

23 → 2

28 → 2

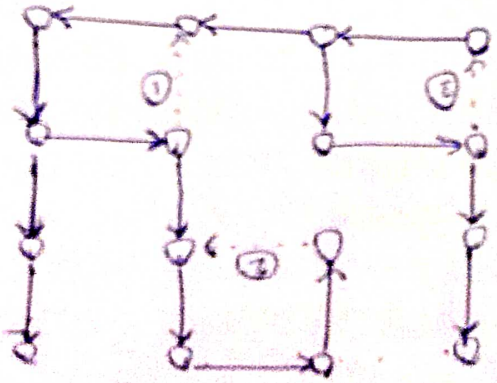
42 → 2

45

46

18 + 1 + 2

21 + 6

30

40