Unit - V

VIJAYSHRI
PAGE NO.:
DATE:

* **Code optimization** ⇒ It's fifth phase of compiler phases. It is required to produce an ~~eff~~ efficient code. The improvement over intermediate code by transformation is called optimization.

(OR)

Code optimization is a techniqure which tries to improve the code by eliminating unnecessary code lines & arranging the statements in such a sequence that speed up the program execution without wasting the resources.

Code optimization may be
- machine independent
- Machine dependent.

Machine independent :- अगर programming code machine के Hardware configuration से independent (free) है तो उसे machine independent code optimization कहते है

Machine dependent :- अगर programming code या optimization machine के Hardware configuration पर से dependent है तो उसे machine dependent code optimization कहते है

(use or ही)

By default use use machine independent Three address code (TAC) optimization techniques.]

Advantages :-

→ It provides faster execution (Executes faster)
→ It use efficient memory (Efficient memory usage)
→ It provides better performance (~~yield~~ Yield better performance)

Term used in code optimization ⇒

- Basic blocks
- Flow Graph (: control flow Graph)

→ **Basic blocks** ⇒ It breaks intermediate code (TAC) into blocks.

Basic blocks are sequences of intermediate code with a single entry & a single exist block.
Basic blocks are represented as "Direct acyclic graph (DAG)"

→ **Control flow graph** ⇒ It's Graphically representation of Basic block means it's show the flow of data or control flow of data.
control flow graphs shows the control flow among basic blocks..

Determine basic block in program ⇒ Determine leader in code & each leader corresponds to a unique basic block

Leader definition ⇒

→ First statement is a leader
→ Any statement which is the target of a conditional or unconditional goto
→ Any statement which immediately follows a conditional goto

Example:-

① $i=1$ — leader - B_1
② $i=0$
③ if $i<100$ goto 5 - leader - B_2
④ goto 13 — leader - B_3
⑤ $t_1 = 4*i$ — leader - B_4
⑥ $t_2 = A[t_1]$
⑦ if $t_2 = x$ goto 9
⑧ goto 10 — leader - B_5
⑨ $i = i$ — leader - B_6
⑩ $t_3 = i+1$ - leader - B_7
⑪ $i = t_3$
⑫ goto 3
⑬ ---- — leader - B_8

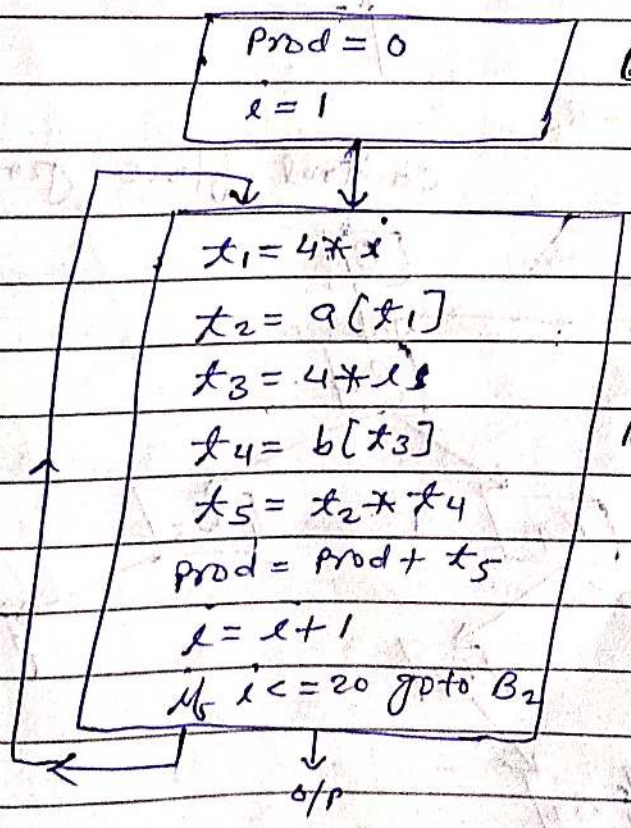
GIRIRAJ

PAGE NO.:
DATE:

Convert following program into the Three Address code(TAC) & identify the Basic blocks.

⇒ begin
prod = 0;
i = 1;
do
⇒ begin
prod = prod + a[i] * b[i]
i = i + 1;
⇒ end
while (i <= 20)
⇒ End

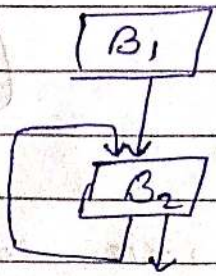
① Product or prod = 0 ⇒ leader - B_1
② i = 1 — B_1
③ t_1 = 4 * i; - leader - B_2
④ t_2 = a[t_1]
⑤ t_3 = 4 * i;
⑥ t_4 = b[t_3]
⑦ t_5 = t_2 * t_4
⑧ prod = prod + t_5
⑨ i = i + 1
⑩ if i <= 20 goto 3 — B_2



Basic block table

Blocks name	Statement no.
B_1	1, 2
B_2	3, 4, 5, 6, 7 8, 9, 10

Control flow ~~diag~~ graph



A flow graph is a directed graph in which the flow control of information is added to the basic blocks.

Q.2)

Convert following TAC into Basic blocks & draw flow graph

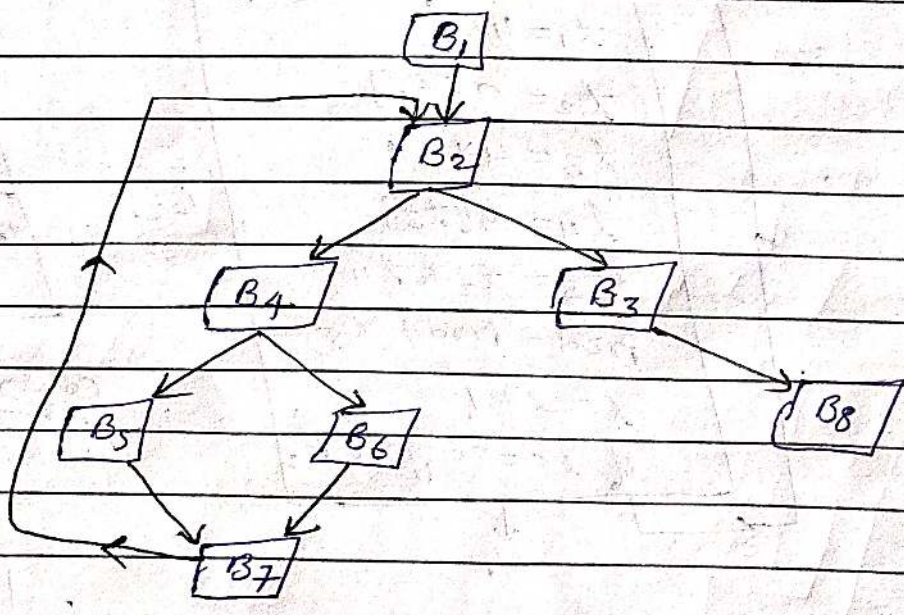
Soln

TAC

① $i = 1$ — leader - B_1
② $i = 0$
③ if $i < 100$ goto 5 - leader - B_2
④ goto 13 — leader - B_3
⑤ $t_1 = 4 * i$ — leader - B_4
⑥ $t_2 = A[t_1]$
⑦ if $t_2 = x$ goto 9
⑧ goto 10 — leader - B_5
⑨ $i = i$ — leader - B_6
⑩ $t_3 = i + 1$ — leader - B_7
⑪ $i = t_3$
⑫ goto 3
⑬ ---- — leader - B_8

Basic blocks	statement No.
B_1	1, 2
B_2	3
B_3	4
B_4	5, 6, 7
B_5	8
B_6	9
B_7	10, 11, 12
B_8	13

Control flow graph



The basic block is a sequence of consecutive statements which are always executed in sequence without halt or possibility of ~~branching~~ branching.

Note: (i) The basic blocks does not have any Jump statements among them.

(ii) When the first instruction is executed, all the instructions in the same basic block will be executed in these sequence of appearance without loosing the flow control of the program.

Example

$a = b + c + d$.

$t_1 = b + c$

$t_2 = t_1 + d$

$a = t_2$

Basic block

Jump

if (A < B) then 1 else 0

(1) if (A < B) goto 4

(2) $t_1 = 0$

(3) goto 5

(4) $t_1 = 1$

(5) ----

Jump

It's not basic block because Jump statement ~~goto~~ So it make a simple basic block

Rules for Partitioning into blocks :- After ~~an~~ an intermediate code is generated for the given code, we can use the following rules to partition into basic blocks

Rules-1 :- Determine the leaders :-

(i) First statement is a leader

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

(iii) Any statement that immediately follow a goto is a leader

(1) if (A < B) goto 4 — leader from (i) Rule (i)

(2) $t_1 = 0$

(3) goto 5 — leader from (ii) Rule ⇒ (ii)

(4) $t_1 = 1$ — leader from (iii) Rule (iii) ⇒ (iii)

(5) ----

Rule-2 ⇒ The basic block is formed starting at the leader statements & ending just before the next leader statement appearings

```
for (i=1; i<10; i++)
{
  a = a + b[i];
}
j = a
```

Q.3) Given TAC, identify the blocks & construct flow graph

① $i = 1$ ——— leader – B_1

② $t_1 = i * 4$ ——— leader – B_2

③ $t_2 = b[t_1]$

④ $t_3 = a + t_2$

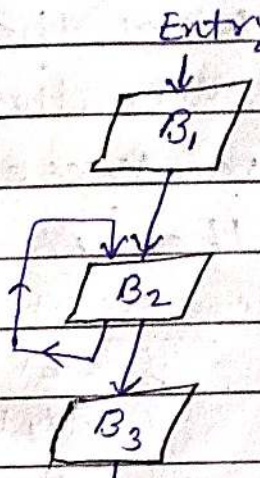
⑤ $a = t_3$

⑥ $i = i + 1$

⑦ if $i < 10$ goto 2

⑧ $j = a$ ——— leader – B_3

Flow graph



Q.4)

① $i = 1$ ——— leader – B_1

② $j = 1$ ——— leader – B_2

③ $t_1 = 10 * i$ ——— leader – B_3

④ $t_2 = t_1 + j$

⑤ $t_3 = 8 * t_2$

⑥ $t_4 = t_3 - 88$

⑦ $a[t_4] = 0$

⑧ $j = j + 1$

⑨ if $j < 10$ goto B_3

⑩ $i = i + 1$ ——— B_4

⑪ if $i < 10$ goto B_2

⑫ $i = 1$ ——— leader – B_5

⑬ $t_5 = i - 1$ ——— leader – B_6

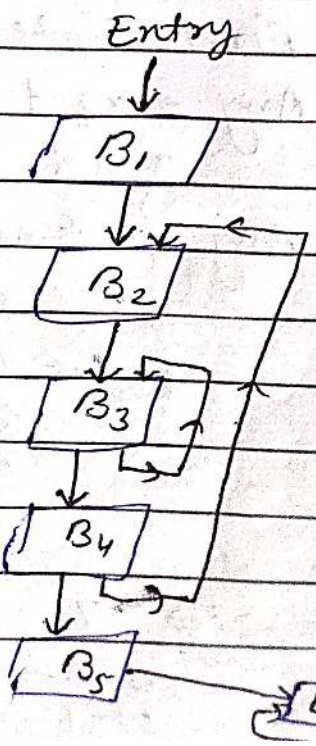
⑭ $t_6 = 88 * t_5$

⑮ $a[t_6] = 1$

⑯ $i = i + 1$

⑰ if $i < 10$ goto B_6

Blocks	Statement
B_1	1
B_2	2
B_3	3, 4, 5, 6, 7, 8, 9
B_4	10, 11
B_5	12
B_6	13, 14, 15, 16, 17



Q.5) Construct basic blocks & flow graph

```
For (i=1 to n)
{
    x = x+1;
}
```

Soln

① $i = 1$ —— leader - B_1

② If ($i \leq n$) goto 4 — leader - B_2

③ goto next 9 — leader - B_3

④ $t_1 = x+1$ —— leader - B_4

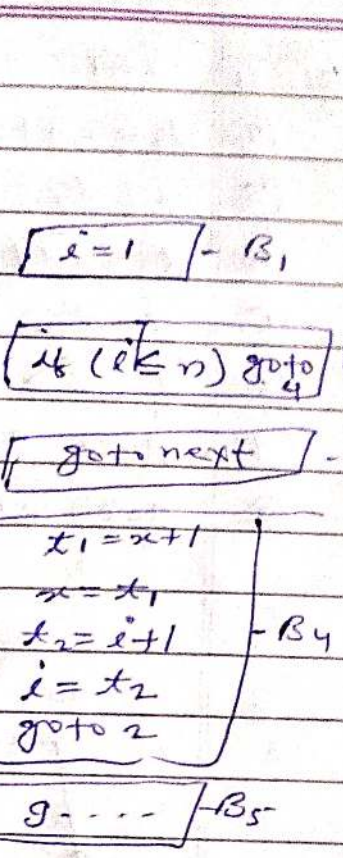
⑤ $x = t_1$

⑥ $t_2 = i+1$

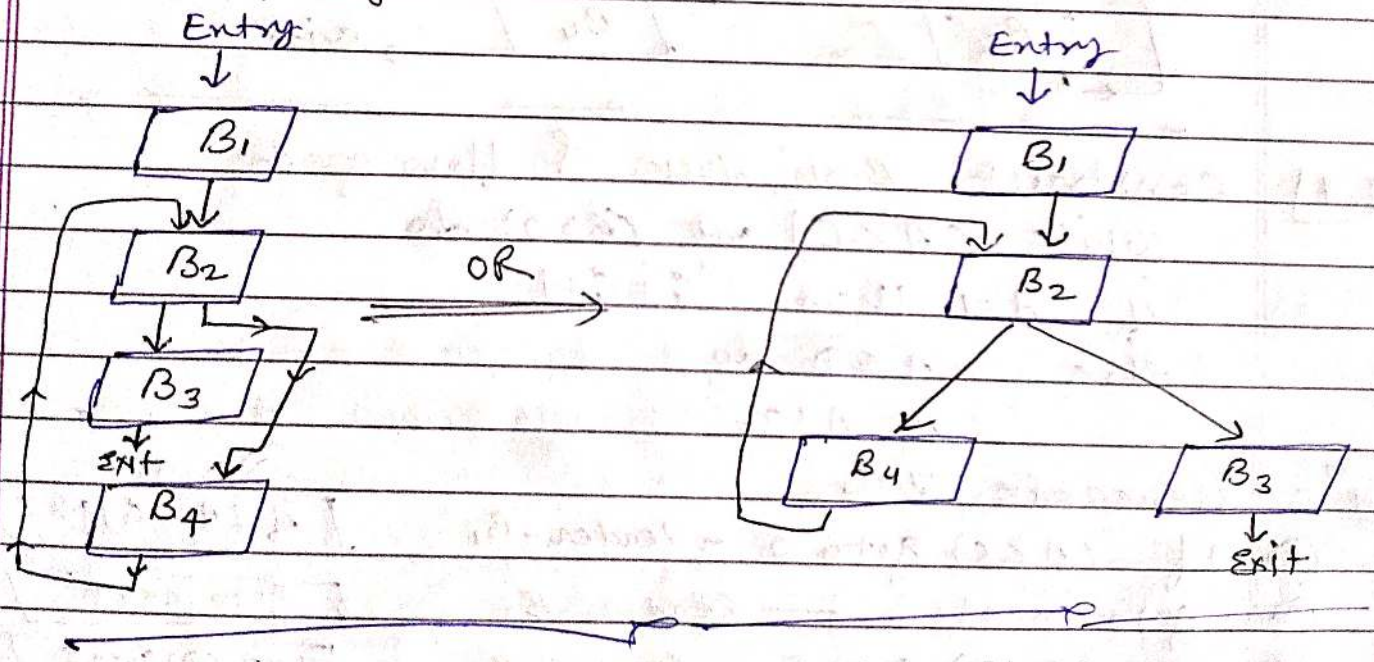
⑦ $i = t_2$

⑧ goto 2

⑨ ----- leader - B_5



Control flow graph



Q.6) Construct Basic blocks & flow graph

```
i = 1
Repeat
    x = x+1
    i = i+1
while (i ≥ n)
```

TAC

Soln (1) $i=1$ —— leader - B_1

(2) $t_1 = x+1$ —— leader - B_2

(3) $x = t_1$

(4) $t_2 = i+1$

(5) $i = t_2$

(6) if $(i \geq n)$ goto 8

(7) goto 2 —— leader - B_3

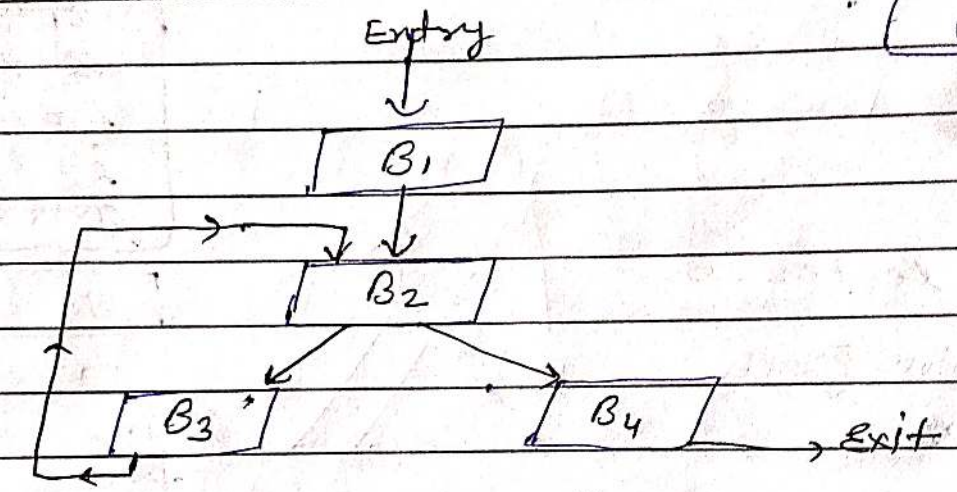
(8) goto next —— leader - B_4

$i=1$ — B_1

$t_1 = x+1$
$x = t_1$
$t_2 = i+1$
$i = t_2$
if $(i \geq n)$ goto 8/B_4 — B_2

goto 2/B_2 — B_3

goto next — B_4



Q.7) Construct Basic block & flow graph

while $(A < C)$ and $(B > D)$ do
if $A = 1$ then $C = C+1$
else $A \leq D$ do
$A = A+2$

Soln Generate TAC

(1) If $(A < C)$ goto 3 — leader - B_1

(2) goto next —— leader - B_2

(3) If $(B < D)$ goto 5 — leader - B_3

(4) goto next —— leader - B_4

(5) if $(A = 1)$ goto 11 — leader - B_4

(6) if $(A \leq D)$ goto 8 — leader - B_5

(7) goto 1 —— leader - B_6

if $(A < C)$ goto 3 — B_1

goto next — B_2

if $(B < D)$ goto 5 — B_3

goto next — B_4

if $A = 1$ goto 11 — B_4

if $A \leq D$ goto 8 — B_5

goto 1 — B_6

(8) $t_1 = A+2$ — leader - B_7

(9) $A = t_1$

(10) goto 6 — leader - B_8

(11) $t_2 = G+1$ — leader - B_9

(12) $G = t_2$

(13) goto 1 — leader - B_{10}

(14) next

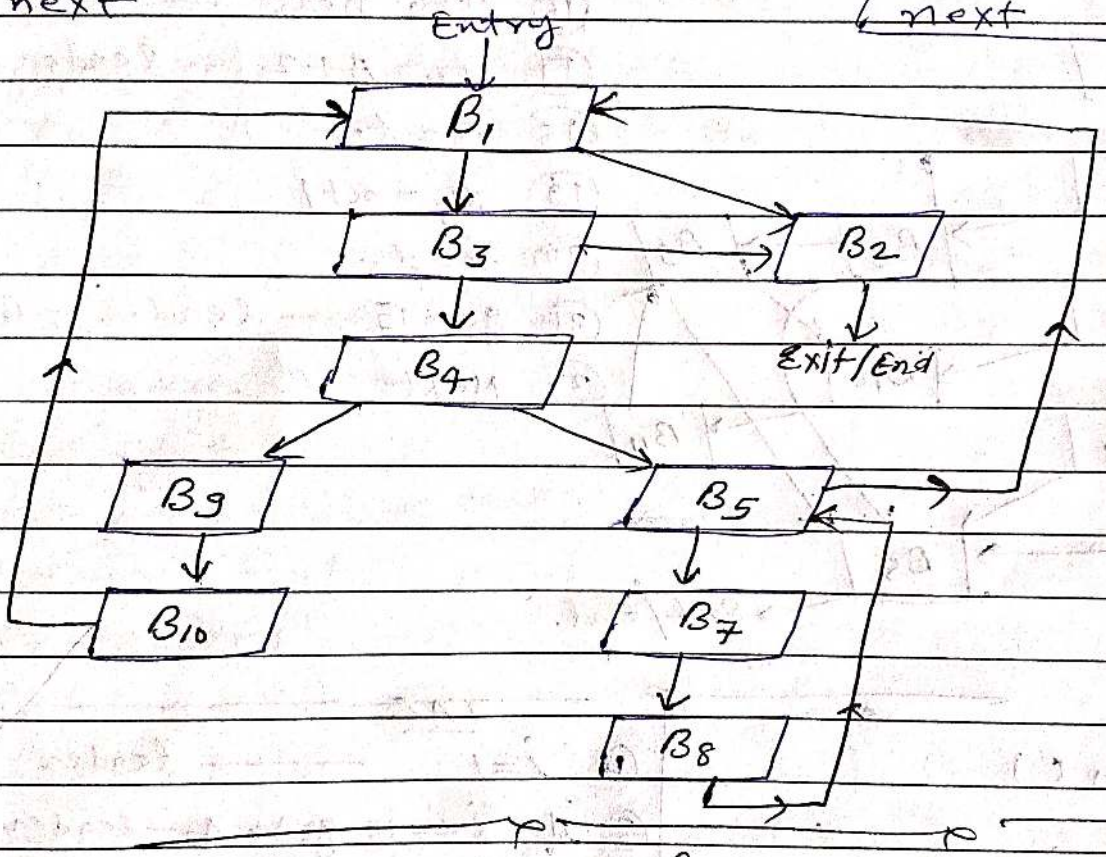
$t_1 = A+2$ - B_7

$A = t_1$
goto 6 - B_8

$t_2 = G+1$ - B_9

$G = t_2$
goto 1 - B_{10}

next

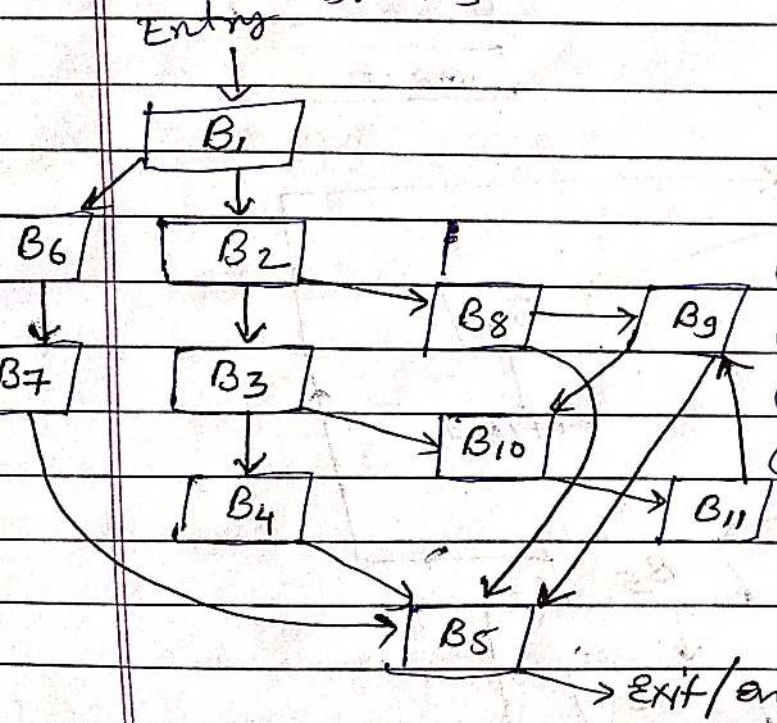


Q.8) Construct Basic block & flow graph

```
switch x
case 2: if (y=2) then
          z = z*2;
          break;
case 4: y = y*2;
          break;
case 6: while x<=n
        { A = A+2;
```

(1) if x=2 goto 7 - leader - B_1
(2) if x=4 goto 12 - leader - B_2
(3) if x=6 goto 17 - leader - B_3
(4) $t_1 = z+2$ — leader - B_4
(5) $z = t_1$
(6) goto next - leader - B_5
(7) if y=2 goto 9 - leader - B_6
(8) goto next - leader - B_5

```
        x = x+1;
    }
    break;
default:
        z = z+2;
        break;
```



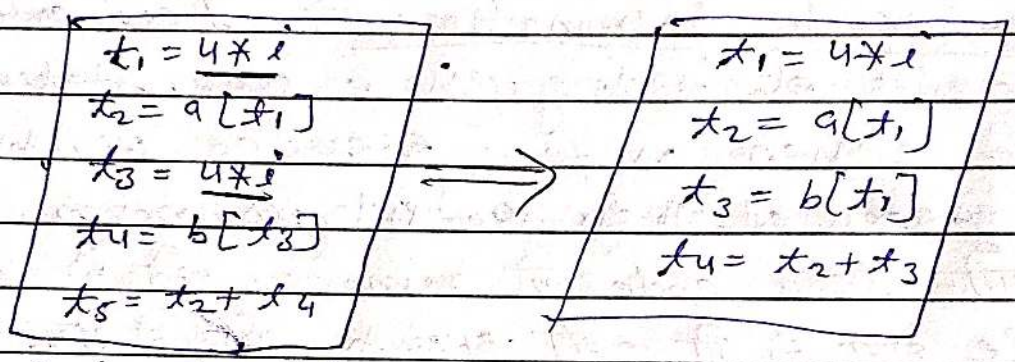
(9) $t_2 = z*2$ — leader - B7
(10) $z = t_2$
(11) goto next — leader - B5
(12) $t_3 = y*2$ — leader - B8
(13) $y = t_3$
(14) goto next — leader - B5
(15) if ($x < n$) goto 17 — leader - B9
(16) goto next — leader - B5
(17) $t_4 = A+2$ — leader - B10
(18) $a = t_4$
(19) $t_5 = x+1$
(20) $x = t_5$
(21) goto 15 — leader - B11
(22) Next

* Local optimization & Global optimization ⇒

Local optimization ⇒ These optimizations consider the statement within basic blocks.

Most commonly used optimizations are
- Variable propagation
- common subexpression elimination

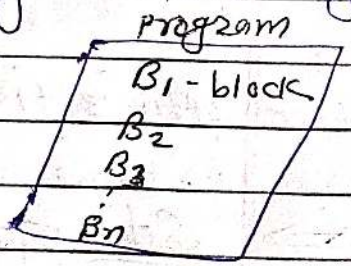
Example:-



Global optimization ⇒ The scope of global optimization is generally, a program unit, usually spanning over number of basic blocks.

→ It can achieved better transformation leading to more optimized.

→ Program flow graph normally undergoes two types analysis:-
- Control flow analysis
- Data flow analysis



Programs are divided into multiple blocks & Blocks divided into multiple statements.

Code optimization types

local (on a single block) | global (check in all blocks)

VIJAYSHRI PAGE NO.: DATE:

Code optimization techniques ⇒

- Compile time Evaluation
- Common sub expression elimination
- Strength reduction
- Code movement
- Dead code elimination.

⇒ Compile time Evaluation ⇒

- Constant Folding
- Constant Propagation

Constant Folding :- It refers to a technique of evaluating the expressions whose operands are known to be constant at compile time itself. [इसमे constant value को पहले से ही calculate करके compile time मे ही value को fixed कर देते हो ताकी run time उस code को evaluate करने का time save हो जाये] (बच)

Example :- length = $(\frac{22}{7}) * d$;

→ length = 3.14 * d.

Constant Propagation ⇒ In constant propagation, if a variable is assigned a constant value then subsequent use that variable can be replaced by a constant as long as no intervening assignment has changed the value of the variables.

[Constant value मे, program मे जहाँ-जहाँ constant name use किया है वहाँ-वहाँ program मे constant name का use न करते हुये उस constant value को लिख देते हो जिससे compiler का run time evaluate करने का time save हो जायेगा] (बच)

Example:- Pi = 3.14 ;
r = 5 ;
Area = Pi * r * r ;
→ Area = 3.14 * 5 * 5

Here the value of Pi is replaced by 3.14 & r by 5 then computation of 3.14 * 5 * 5 is done during compilation time.

⇒ Common Expression elimination ⇒ The common sub-expression is an expression appearing repeatedly in the code which is computated previously. This technique replaces redundant expression each time it is encountered. [जो code बार-बार (more than one) programming में use होता रहता है उसे common sub-expression कहते है।]

Example:-

$t_1 = 4 * i$
$t_2 = a[t_1]$
$t_3 = 4 * j$
$t_4 = 4 * i$
$t_5 = n$
$t_6 = b[t_4] + t_5$

Before optimization

same

$t_1 = 4 * i$
$t_2 = a[t_1]$
$t_3 = 4 * j$
$t_5 = n$
$t_6 = b[t_1] + t_5$

After optimization

⇒ Code movement ⇒ It is a technique of moving a block of code outside a loop. If it wait have any difference if it is executed outside or inside the loop.

Example:-

```
for (i=0; i<n; i++)
{
    x=y+z;
    a[i] = 6*i;
}
```

Before optimization

```
x=y+z;
for (i=0; i<n; i++)
{
    a[i] = 6*i;
}
```

After optimization

⇒ **Dead code Elimination** ⇒ Dead code elimination includes eliminating those code statements which are either never executed or unreachable or if executed their output is never used.
[इसमे dead code को remove करते है dead code वह code होता है जो कभी execute नही हो सकता / dead code या तो unreachable code होता है या execute होता भी है तो उसके result का program मे कोई use ही नही होता है उसे dead code कहते है इस type के code को remove करते है]

Example:-

```
i=0
if (i==1)
{
a=x+5;
}
```

Before optimization

```
i=0
```

इस code मे i=0 fixed value हो गई है & if (i==1) जो भी कभी भी i==1 नही होगी क्योकी i=0 है इसलिये इसे remove कर दिया है

After optimization

⇒ **Strength reduction** ⇒ It is the replacement of expressions that are expensive which cheaper & simple ones onces. [जो expression expensive है

DATE:

DAG (Directed acyclic Graph) Representation ⇒

In compiler design, A directed acyclic graph (DAG) is an abstract syntax tree (AST) with a unique node for each value. (OR)
A directed acyclic graph (DAG) is a directed graph that contains no cycles.

DAG are useful data structure for implementing transformation of basic block of three address code.
~~A DAG is represented by following rules:-~~

Use of DAG for optimizing basic block :-

→ DAG is a useful data structure for implementing transformations of basic blocks.
→ A basic blocks can be optimized by the construction of DAG
→ A DAG can be constructed for a block & certain transformations such as common sub expression elimination & dead code elimination can be applied for performing the local optimization.
→ To apply the transformations on basic block, a DAG is constructed from three address code.

Properties of DAG ⇒

→ The reachability relation in a DAG forms a partial order & any finite partial order may be represented by a DAG using reachability.
→ The transitive reduction & transitive closure are both uniquely defined for DAG.
→ Every DAG has a topological ordering

Application of DAG ⇒ DAG is used in :-

→ Determining the common sub expression (expression computed more than once)

→ Determining which names are used in the block & computed outside the block.

→ Determining which statements of the block could have their computed values outside the block.

→ Simplifying the list of quadruples by eliminating the common sub expressions & not performing the assignment of the form x := y until and useless it is a must.

Rules for DAG construction ⇒

Rule-1 ⇒ In a DAG,

→ All leaf nodes are represented by identifiers, variables (names) & constants.

→ All ~~interior~~ interior nodes represented by operators

Rule-2 ⇒

while constructing DAG, there is a check made to find if there is an existing node with the same children. A new node is created only when such a node does not exist. This action allows upto detect common sub-expressions & eliminate the recomputation of the same.

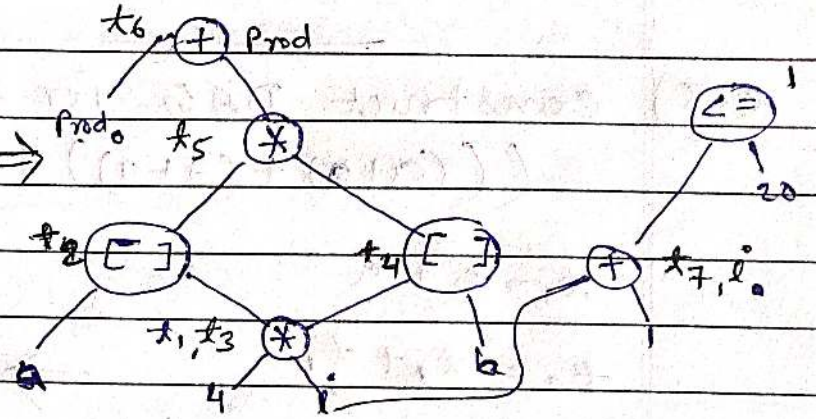
Rule-3 ⇒

The assignment of the form x := y must not be performed until and unless it is a must.

Q.1 Block of TAC is given below. construct DAG

① $t_1 = 4 * i$
② $t_2 = a[t_1]$
③ $t_3 = 4 * i$
④ $t_4 = b[t_3]$
⑤ $t_5 = t_2 * t_4$
⑥ $t_6 = prod + t_5$
⑦ $prod = t_6$
⑧ $t_7 = i + 1$
⑨ $i = t_7$
⑩ if $i <= 20$ goto 1

Basic block B_1 ⟹ DAG

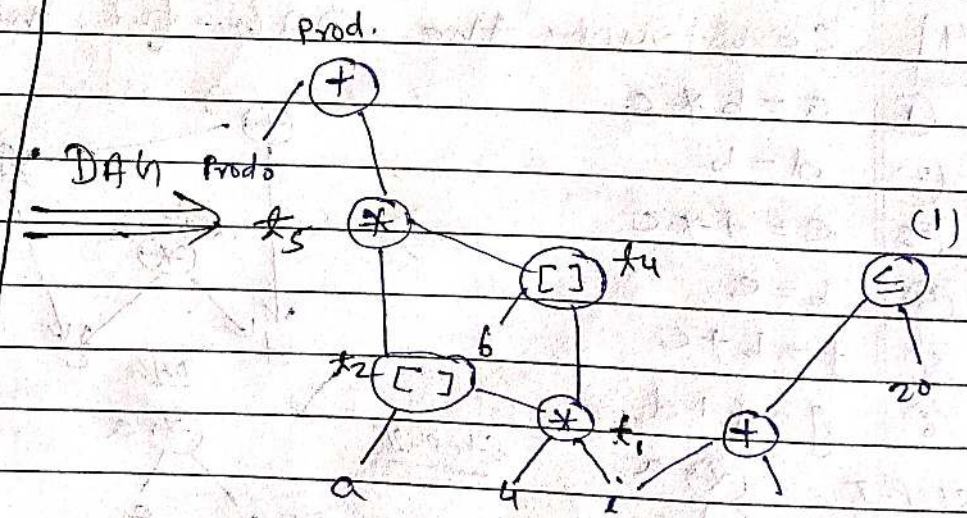


DAG

Reduced (optimized)

① $t_1 = 4 * i$
② $t_2 = a[t_1]$
③ $t_4 = b[t_1]$
④ $t_5 = t_2 * t_4$
⑤ ~~t_6~~ $prod = prod + t_5$
⑥ ~~t_7~~ $i = i + 1$
⑦ if $i <= 20$ goto 1

DAG



Q.2 Construct DAG for the given expression

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

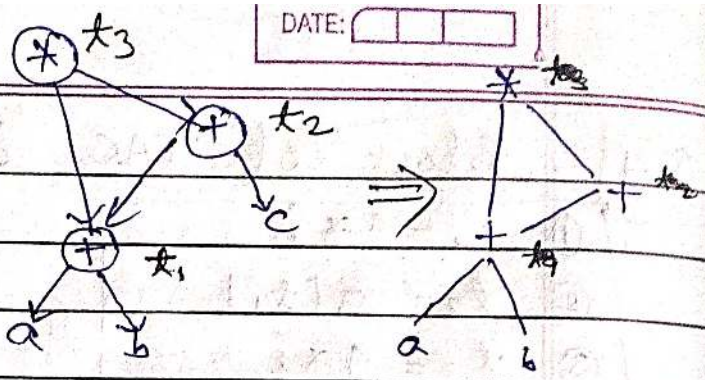
Soln: Convert in TAC (Three address code)

DATE:

$t_1 = a+b$

$t_2 = t_1 + c$

$t_3 = t_1 * t_2$



DAG — abstract DAG

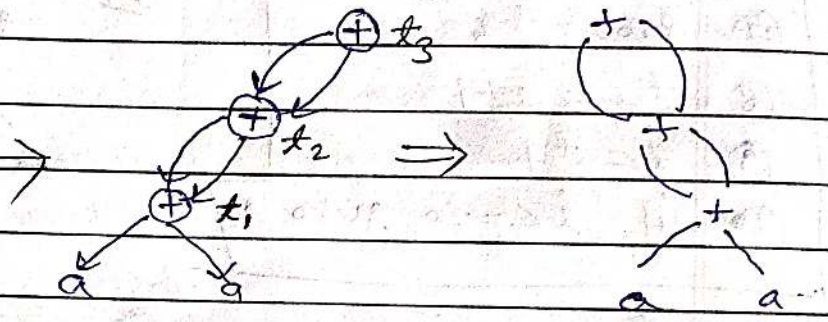
Q.3) Construct DAG for the given expression

$(((a+a)+(a+a))+((a+a)+(a+a)))$

$t_1 = a + a$

$t_2 = t_1 + t_1$

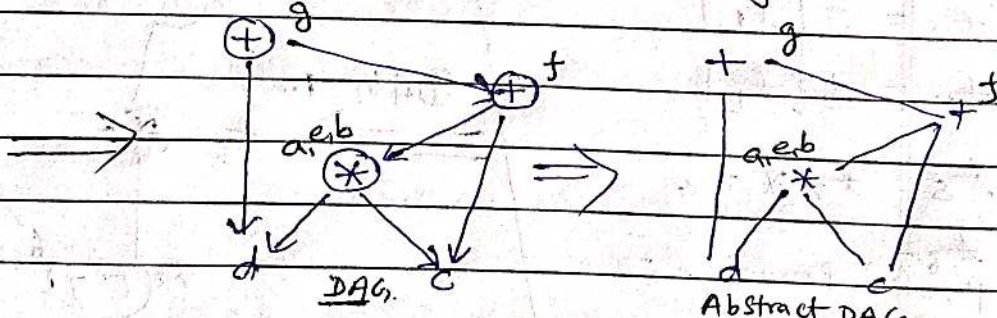
$t_3 = t_2 + t_2$



DAG — Abstract DAG

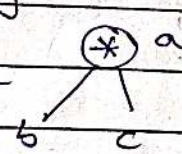
Q.4) Construct the DAG for the following blocks.

1. $a = b * c$
2. $d = b$
3. $e = d * c$
4. $b = e$
5. $f = b + c$
6. $g = f + d$

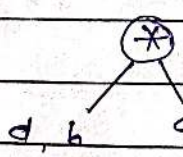


DAG — Abstract DAG

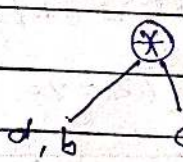
Step-I



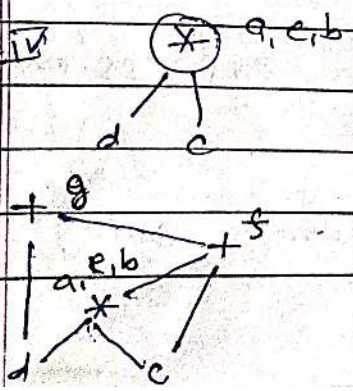
Step-II



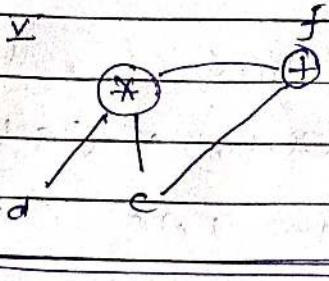
Step-III



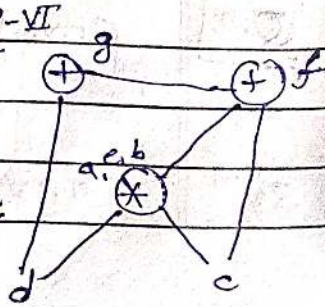
Step-IV



Step-V

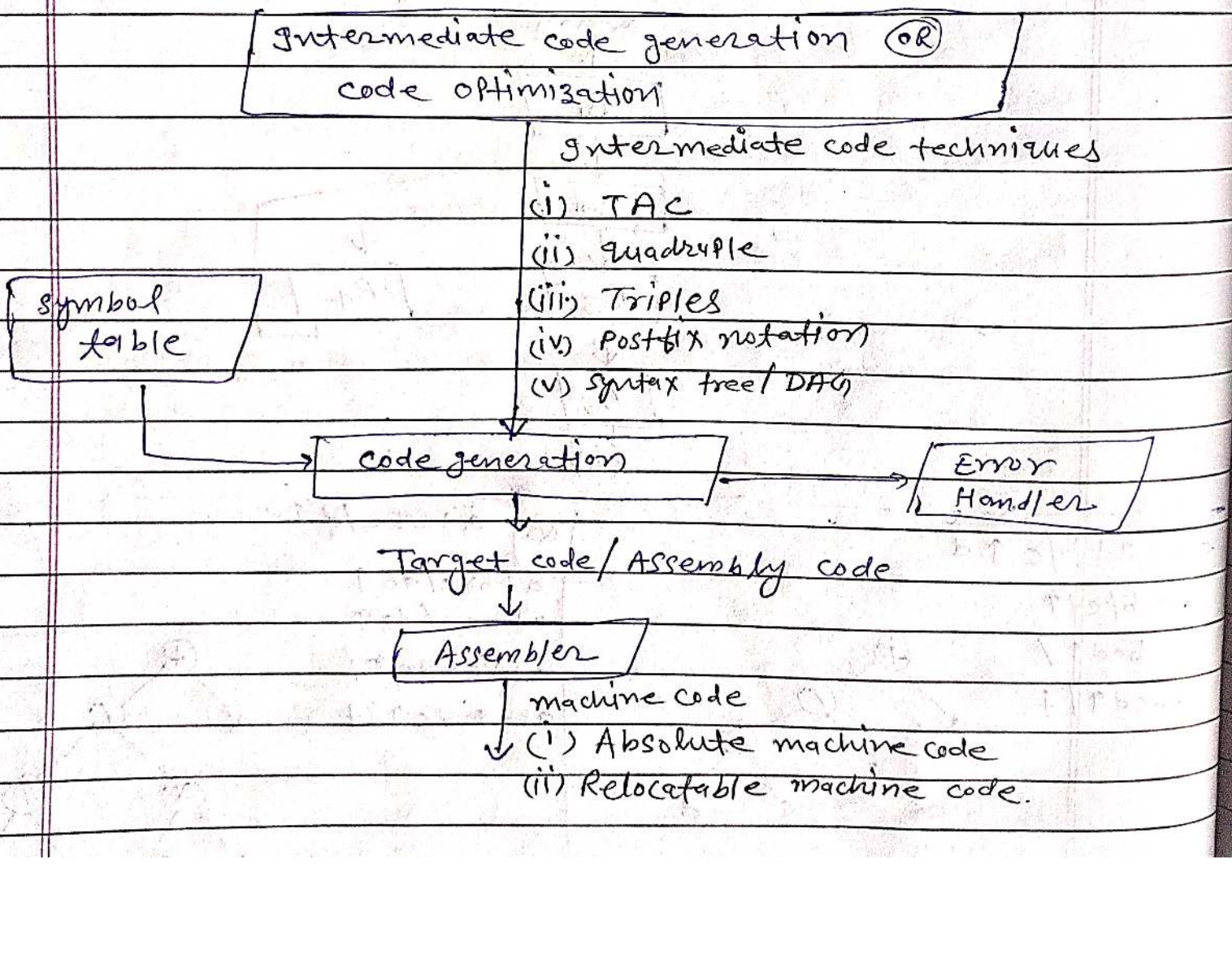


Step-VI



* Code generation ⇒ It's last phase of compiler phase code generation is the final activity of compiler & code generation is the process of creating Assembly/ machine language. There are some properties of code generation.

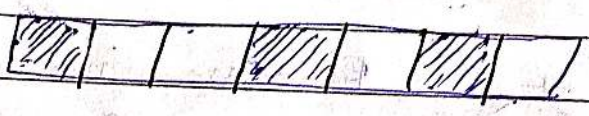
- → correctness (code in correct form & not change in meaning (purpose))
- → High quality (No error, No ambiguous)
- → Efficient use of resource of target machine.
- → Quick code generation (जल्दी से compilation process complete हो 3# ~~error~~ code generate हो)



⇒ Absolute machine code ⇒ It used fixed memory location in memory & immidiatly evaluated. It useful for small program.

⇒ Relocatable machine code ⇒ It's not used fixed memory Not a fixed location in memory & code can be placed whereever linker find the from in RAM It useful for commercial compiler.

It's dynamic memory allocation based.



Design issues in code generation :-

① Input to code generation [Phase से जो इस ~~Phase~~ ~~code~~ I/P आयेगा वह intermediate code techniques से ही होना चाहिये]

② Target program [ऐसा Target/Assembly code होना चाहिये जिसे Assembler Absolute/Relocatable code में convert करे]

④ Instruction Selection [

⑤ Register Allocation

③ Memory management [Symbol table/Error Handler data structure में add हो / information retrieve हो]

⇒ Instruction selection ⇒ example $a = y + z$

```
Mov y, R0   - Register R0
ADD z, R0
MOV R0, x
```

```
a = b + c
d = a + e
```

Transform →

```
Mov b, R0
ADD c, R0
Mov R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

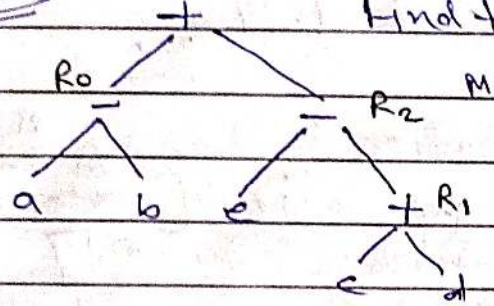
```
Mov b, R0
ADD c, R0
ADD e, R0
Mov R0, d
```

Ambiguous code because Mov R0, a & Mov a, R0

⇒ Register Allocation ⇒ पिछले example में एक ही Register R_0 use हुआ है इसी वजह More than one Register use हो रहा है तो

$t_1 = a + b$ → { MOV a, R_0
ADD b, R_0
$t_2 = t_1 * c$ → MULT c, R_0
$t_3 = t_2 / d$ → DIV d, R_0
MOV R_0, t

Example



Find the minimum Register used in given tree

Syntax

MOV $R_0 \leftarrow a$
$R_1 \leftarrow b$
SUB $R_0 \leftarrow R_0 - R_1$
$R_1 \leftarrow c$
$R_2 \leftarrow d$
ADD $R_1 \leftarrow R_1 + R_2$
$R_2 \leftarrow e$
SUB $R_2 \leftarrow R_2 - R_1$
ADD $R_0 \leftarrow R_0 + R_2$

So minimum Register used = 3 (R_0, R_1, R_2)

* Peephole optimization ⇒ यह Assembly code को optimize करने की technique है

A statement by statement code generation strategy generate target code that contain redundent instruction to optimize such target code we use peephole optimization technique.

→ Optimization can be directly applied on assembly language.
→ A simple but efficient effective technique for locally improving the target code.
→ It is done by examining a sliding window of target instructions (called peephole) & replacing instruction sequenced within the peephole by a shorter or faster sequence.

→ Repeated passes over the target code are necessary to get maximum benefits.

~~Examples:- types of optimization required in peephole~~

~~(i) Redundant instruction~~ :- Example :-

(i) Mov R0, a
(ii) Mov a, R0
} These are redundant instruction so eliminate (ii) instruction

∴ Mov R0, a & remove mov a, R0

(ii) Example-(2)

Mov &a, &b
Mov &b, &a
} ⟹ Mov &a, &b

Characteristics of Peephole optimization ⇒

① Redundant-instruction elimination ⇒

load
store

LD R0, a
ST a, R0
} ⟹ LD R0, a

These are redundant instruction

Here LD = load a into R0
ST = store R0 into a

② Eliminating Unreachable code ⇒

if debug == 1 goto L1
goto L2
L1: Print "Hi"
L2: - - - - -

} if debug != 1 goto L2
Print "Hi"
L2: - - - - -

③ Flow of control optimization ⇒

goto test
- - -
test: goto done
- - -
done

⟹

goto done
- - -
test goto done
- - -
done

goto - test - done → done

④ Algebric simplification ⇒

$x = x + 0$ (or) $x = x * 1$ } Eliminate such instructions

⑤ Reduction strength ⇒ Addition is cheaper than multiplication
Subtraction is cheaper than division

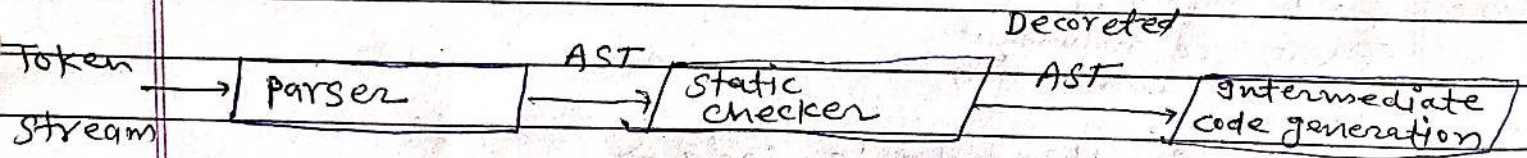
⑥ ~~Machin~~ machine ~~idems~~ idioms ⇒

$x = x + 1$ ⟹ Removed & ~~mach~~ use auto increment

$x = x - 1$ ⟹ Removed & use auto decrement

---

* Type checking ⇒ AST (abstract syntax tree)



Type :- Notation varies from language to language
→ Goal of type checking is to ensures that operations are used with correct type.
Type checking ⇒ it's may be two types
- Static type checking
- Dynamic type checking

Static type checking :- It performs on compile time
Dynamic type checking :- It performs on run time.

Static (semantic) check ⇒

→ **Type check :-** Operator applied to an incompatible operand

Example :- 2 % 4.5 generate error because modules operator work b/w two ~~integers~~ integer values (operand)

→ **Flow control :-** Statement that cause flow of control to leave a construct must have some place to which to transfer flow of control.

Example :-

```
while ()
{
break;
}
```

(or)

```
while ()
{
=
}
break;
```

अगर Break statement & jump statement का use सही जगह नही किया गया तो error आ जायेगी

→ **Uniqueness check ⇒** There are situation where object must be defined exactly onces

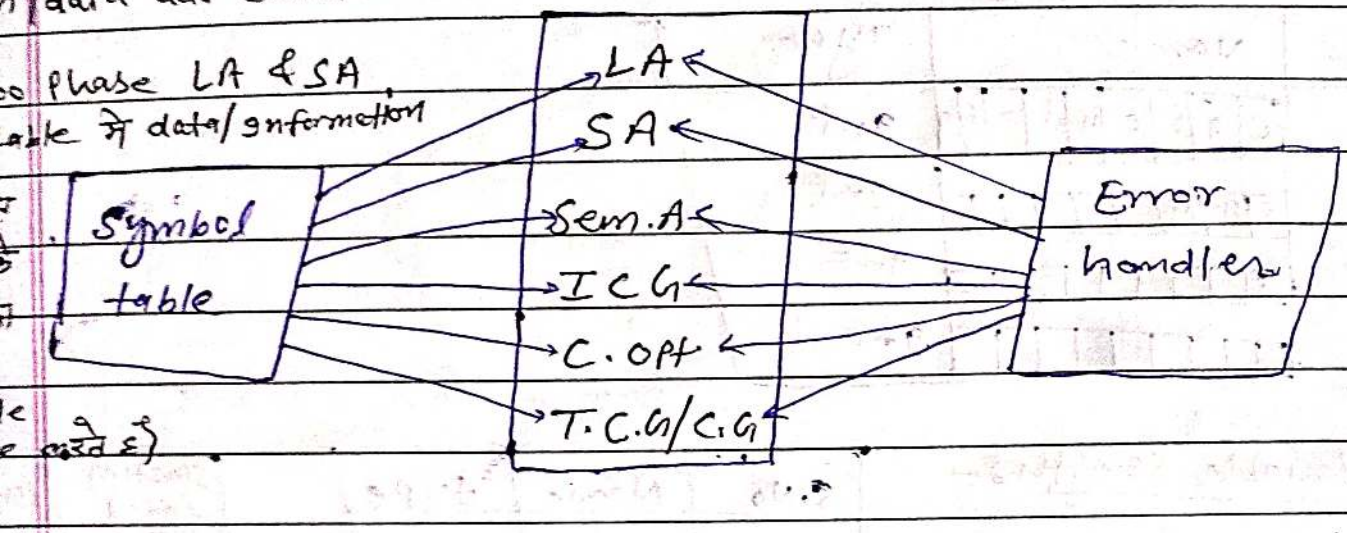
Example :- goto label a अगर multiple label use कर रहे है तो every label को uniquelly identified होना चाहीये नही तो goto not properly work.

→ **Name related check :-** Some times, the same name must appear two or more times.

Example :-

```
main()
{
add (x,y);
}
add (a,b);
```

Symbol table ⇒ Symbol table is a data structure created and maintained by compiler in order to store information about variables, functions, class, object etc. symbol table create & infor- data add compiler के LA & SA phase में करते हैं)



Formate of symbol table ⇒ Compiler uses following type information in symbol table

- Data type, Name, Scope (local/global), address, other attributes

Example:- static int a;
float b;

S.No.	Name	type	Attribute	Scope
1	a	int	static	global
2	b	float	–	local

Representation of symbol table:- It's represented by mainly two ways

- Fixed length :- symble table size is fixed length
- Variable length :- Symbol table size is variable length

⇒ Fixed length :- Symbol table size is fixed length

Example :- int calculate;
int sum;
int a, b;

S.No.	Name	Type
1	calculate	int
2	sum	int
3	a	int
4	b	int

⇒ Variable length :-

S.No.	Name	type
1	calculate	int
2	sum	int
3	a	int
4	a	int

⇒

starting index	length	type
0	10	int
10	4	int
14	2	int
16	2	int

array

c	a	l	c	u	l	a	t	e	$	s	u	m	$	a	$	b	$				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17				

$ से string का कि part होता है

Operation on symbol table ⇒

- insert (); // value/information insert in symbol table
- LookUp (); // LookUp symbol in symbol table (check करना)
- Delete (); // value/information delete from symbol table
- Scope mgt (); // check scope of variable (local/global)

insert (Variablename, type) (or) insert (a, int)
LookUp (symbol) (or) LookUp (a) a से related all information show
Delete (symbol) (or) delete (a)
Scope management : local & global variable management.

Example:- int value ⇔; // global data

```
Void one()
{
  int a;
  int b;
  {
    int c;
    int d;
  }
  int e;
  {
    int f;
    int g;
  }
}
Void two()
{
  int x;
  int y;
  {
    int p;
    int q;
  }
  int r;
}
```

Symbol table

S.No.	Name	work as	Data type
1	value	Variable	type
2	one	Procedure (function)	Void.
3	two	Procedure	Void

Symbol table ⟹

Name	work as	Data type
a	Variable	int
b	Variable	int
e	Variable	int

Name	work as	Data type
x	Variable	int
y	Variable	int
r	Variable	int

Name	work as	Data type
c	Var.	int
d	Var.	int

Name	work as	Data type
f	Var.	int
g	Var.	int

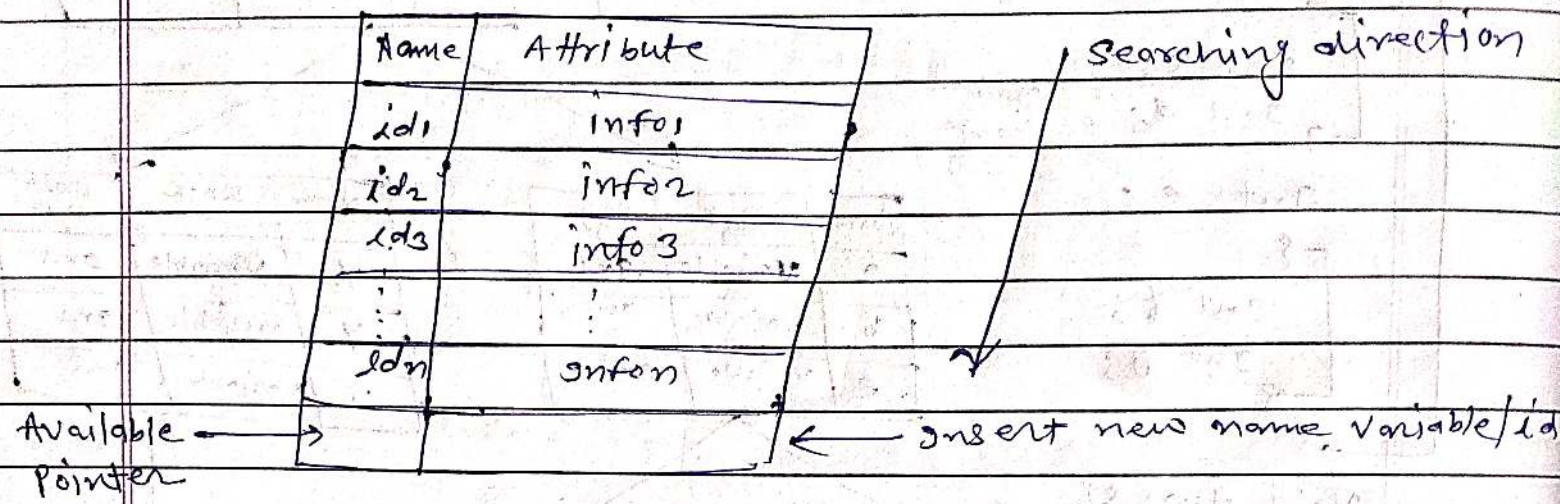
Name	work as	Data type
p	Var.	int
q	Var.	int

Implementation of Symbol table ⇒

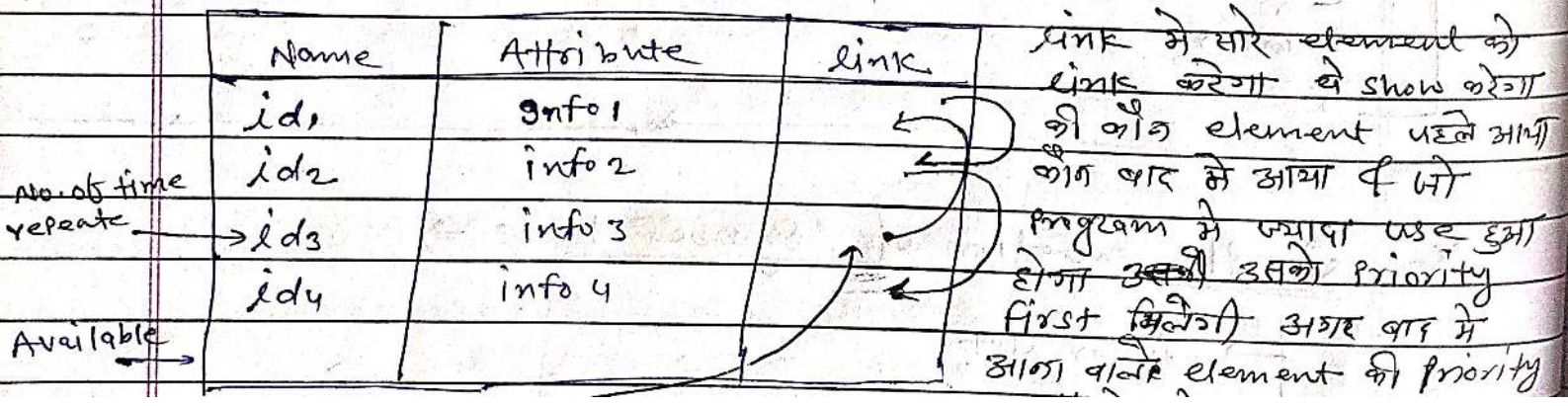
- Linear list
- Self organizing list
- Binary search list
- Hash table

⇒ **Linear list** :- It's simplest method for implementing the symbol table.

Linear list is the simplest way to implement symbol table. An array used to stor information

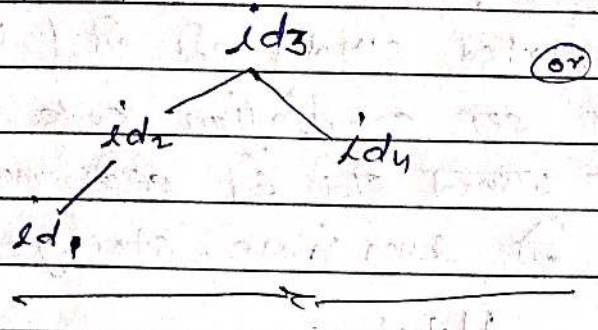


⇒ **self organising list** ⇒ This symbol table implemented using linked list. A link list field added to each record.

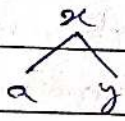


⇒ Binary search tree ⇒ Binary search tree में left value root node से less होती है & right value root node से greater होती है।

int a, x, y;

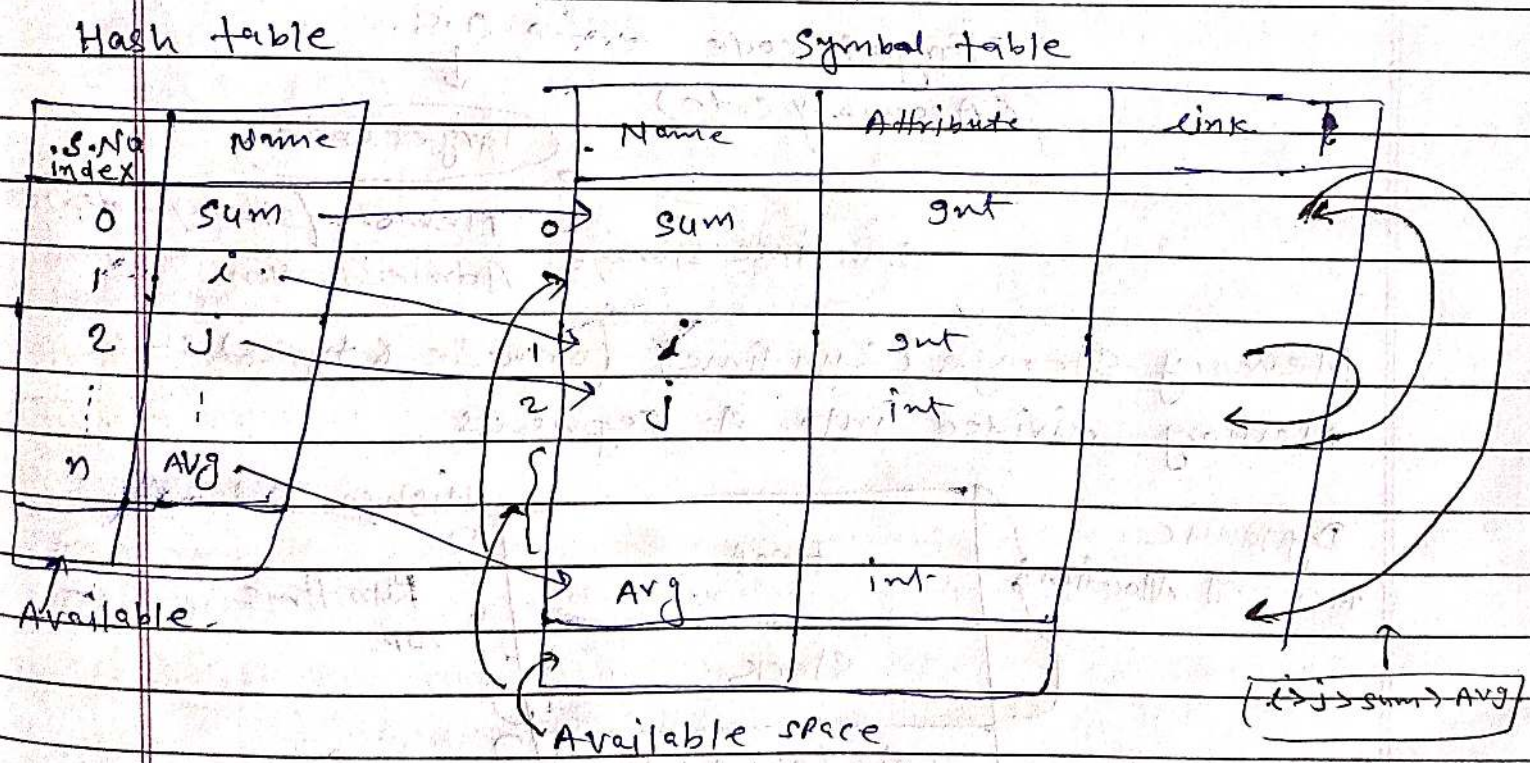


(or)



Alphabetic order

⇒ Hash table ⇒ Hashing is the most powerful implementation technique in Symbol table. Hashing technique use two tables are maintained.



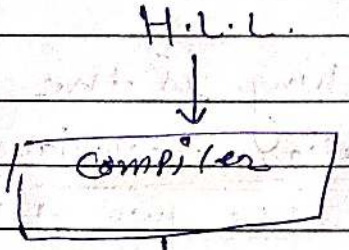
~~Hash~~ Hash function h(name) return. integer value (index number) 0, 1, 2, 3, ---n

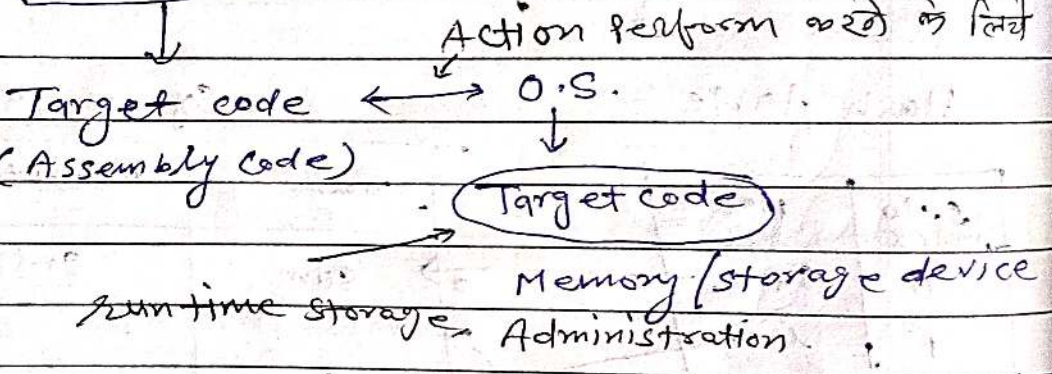
~~Example~~ ~~h(j) = return 2 Jump 2 location~~

Symbol table

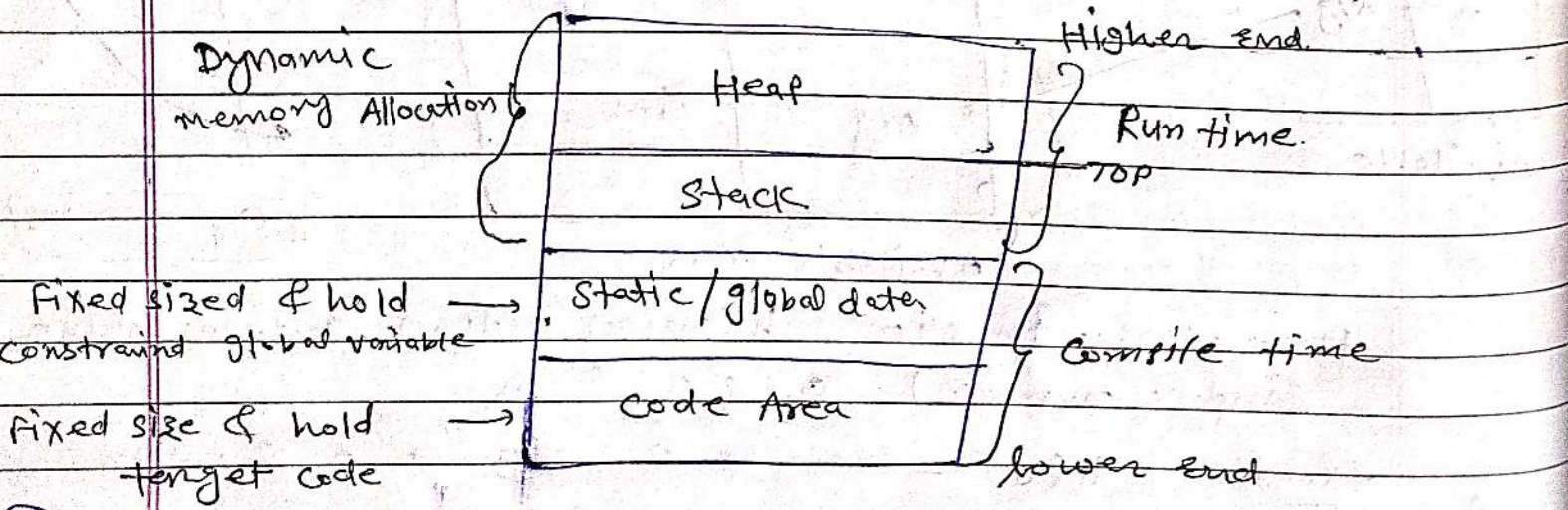
Searching speed :- Hash table > Binary search tree > self organizing list > linear list

* **Run time storage Administration** → जब program बनाते हैं तो program High level lang. बना होता है फिर इस program को compile करेंगे लेकिन compilation process में कहीं भी memory से related study नहीं की। जब program को run करते हैं तो इस compilation code को run होने के लिये memory की जरूरत होती है उसको management करने की technique को run time storage Administration कहते





Memory storage (run time) formate (types) :-
Memory divided into 4 segments



Stack used for local data (activation recode) & Heap used for local & non local data

* Run time Storage Administration ~~के~~ type :- It divided into two categories

→ Implementation of a simple stack allocation scheme
→ Implementation of block structured language (Heap allocation)

⇒ Simple stack allocation scheme ⇒

```
main()
{
  one();
}
one()
{
  two();
}
two()
{
  —
  —
}
```

जो function उसको Activation record
Activation record main()
Activation record one()
Activation record two()

Activation record.

जो Activation Record पहले create होगा वह stack में पहले insert (push) करेगा
जो last में Activation record बनेगा वह stack के TOP में आयेगा & जो stack के TOP पर होगा पहले वही execute होगा

Activation record.

Return value
Actual Parameter
control link (Dynamic link)
Access link (static link)
Local data
Temporary variable
(optional)

Terms Details :-

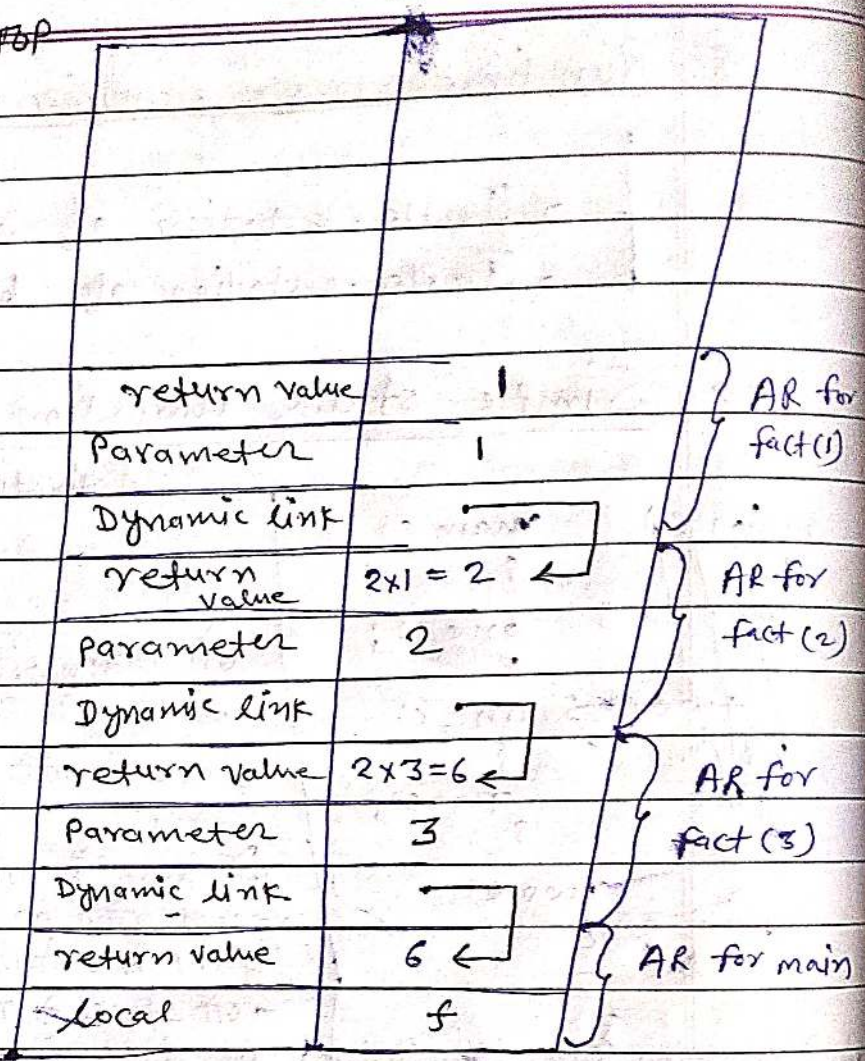
Temp. variable :- If any temporary variable in program [store Here]
local data :- If any local data (variable) available in program [store Here]
Access link :- अगर program में non local data होगा तो store यहां करेंगे
ये optional हैं क्योंकी stack allocation में non-local data नहीं होगा
control link :- जो Activation Record को call कर रहा है उसकी link होगी
Actual Parameter :- Actual Parameter store Here
Return value :- If any function return any value.

Example :-

```
main()
{
int f;
f = fact(3);
}
int fact(int n)
{
if(n==1)
return 1;
else
return (n*fact(n-1));
}
```

Total Activation record. (AR)

AR for main
AR for fact(3)
AR for fact(2)
AR for fact(1)



Stack allocation scheme

⇒ Block structured language [Heap allocation scheme]

Storage

local
(Handling by AR)

Non-local
(Handling by scope information)

Static scope information
(Block structure storage)

Dynamic scope information
(Non block structure storage)

DATE:

Example :-

Local data ⇒

```
A()
{
int a; // local data
}
```

Activation record (AR)

return	
local variable	a

Non-local data ⇒

```
Test()
{
int a, b; // local data
{
int X, Y; // non local data
}
{ int c, d; // non local data
}
```

for for for

Example :-

Static scope rule (lexical scope) ⇒

```
Scope.test()
{
int P, Q;
{
int P;
{
B3 int r;
}
}
{
B4 int Q, S, t;
}
}
```

B1 B2

Step-I

Y (B3)
P (B2)
Q (B1)
P (B1)

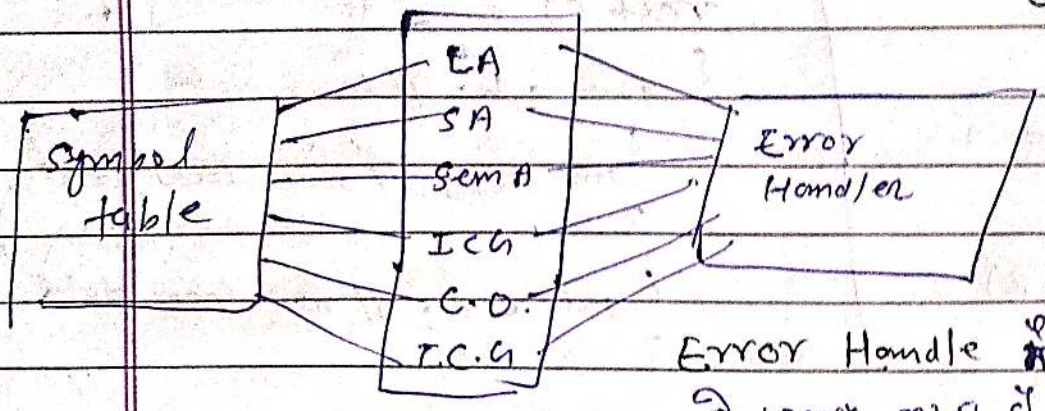
Access करेंगे तो ये दोनों access होने के बाद बच जायेगा

⇒

Step-II

t (B4)
S (B4)
Q (B4)
Q (B1)
P (B1)

Error Detection & Recovery ⇒



Error Handle =
Error Detection +
Error reporting +
Error recovery

Error Handle ही LA-SA-Sem A Phase मे work करती है बाकी phase मे error आने पर Handle करता है

Error

- Compile time error
  - Lexical Phase Error
  - Syntax Phase Error
  - Semantic Phase Error
- Run time error

main error generated in three phases

Error Recovery method ⇒

Recovery Method / Error	Lexical Phase Error	Syntax/Syntactic Phase Error	Semantic Phase Error
Panic mode	✓	✓	X
Phrase-level	X	✓	X
Error Production	X	✓	X
Global production	X	✓	X
Using symbol table	X	X	✓

⇒ Lexical Phase Error ⇒

- → exceding length of identifier :- example int sum.
  (define length करोगे length से identif
- → Appearance of illegal character :- Example int a $
  illegal char
  not allowing
- → Unmatched string or comment :- Example "Hell or
  / @ /* comment

ample :-

```
Void main ()
{
int a, @, $ ; /* Variable declaration */
a=10;
Printf("%d", a); $
}
```

error

Error Recovery method ⇒

Panic mode method :-

int a, 5b, sum, $z ;

error

error recovery

Tokens :- int, a, [5b], sum, [$z], ;

remove

Scan करेगा जो valid token होंगे उनको रहने देगा। जो invalid token हैं उनको common या semicolon तक value से remove कर देगा।

⇒ Syntax or Syntactic Phase Error ⇒

- → Missing ~~Parameter~~ Parenthesis :- Printf("Hello" ;
- → Missing operator :- a+b c
  space but ~~not~~ missing operator

→ misspelled keyword :- ~~switch~~ swicth(ch)

→ colon in place of semicolon :- a=1 :

→ Extra blank space :- /* comment * /
space

Recovery ⇒

Panic mode :- Similar as in lexical Phase Error

Example printf("Hell ;
Parsing से
swicth(ch) .
remove
remove { semicolon/comma तक
सभी को remove करेगा
semicolon/comma पर रुकेगा

Phrase level recovery method :- When parcer encounter an Error it perform necessary action on remaining ~~input~~ input and parse rest of input.

Error production ⇒ Add extra grammar production & make an Augumented grammar & parse the input.

Global correction ⇒ The Parser Examine the whole program & tries to find out ~~closest~~ closest match for it which is error free due to high space & time complexity it is not implemented practically.

⇒ Semantic Phase Error ⇒

→ Incompatible type of operands :- int a; float b;
~~int a * float b~~ a * b;

→ Undeclared variable :- int a;
a * b ← undeclared.

→ Not matching actual argument with format argument

Example :- int a[10], b;

a = b;
↑ Error because it's Array type

Recovery :- int a
a * b

By default b को int integer consider
~~are~~ ~~symbol table~~ में
add कर देगा

using Symbol table ⇒

Global Data flow analysis ⇒ The global data flow analysis is a process of gathering the information about the whole program & distribute these information amongs each block of the flow graph.

→ The gathered information helps to ~~acti~~ achieve a number of optimization.

→ Global data flow analysis is used to solve a specific problem. ("user definition (ud) chaining");

Example:- Given that the identifier A is used at a point P at what point could the value of A used at P have been defined.

Reaching Definition:- The reaching definition implies the ~~detec~~ determination of definitions that apply at a point P in a flow graph. It follows the given steps

(i) Assign a distinct number of each definition such as $d_1, d_2, d_3, \ldots d_n$

(2) for each variable i, make a list of all definition in entire program where it is used.

(3) for each basic block B ~~compute~~ compute the following

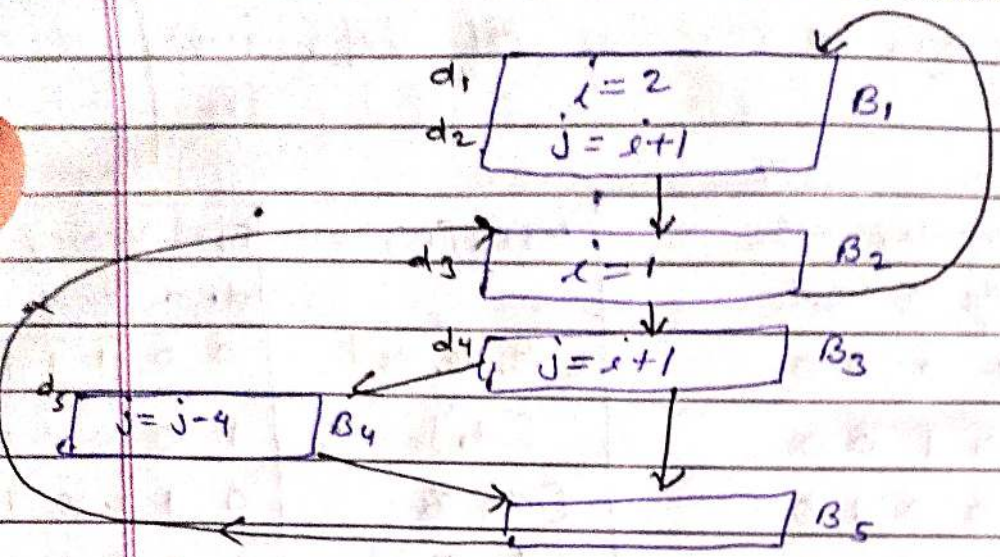
(a) GEN[B] {generate B}:- The set GEN(B) consists of all the definition generated in block B.

(b) KILL[B]:- The set of all the definition outside block B that defines the same variable having definitions in block B also.

(4) for all the basic blocks B, compute the following

(a) IN[B]: The set of all the definitions reading the point just before the first statement of block B.

⑥ OUT[B]:- The set of all the definitions reaching the point just after the last statement of basic block B



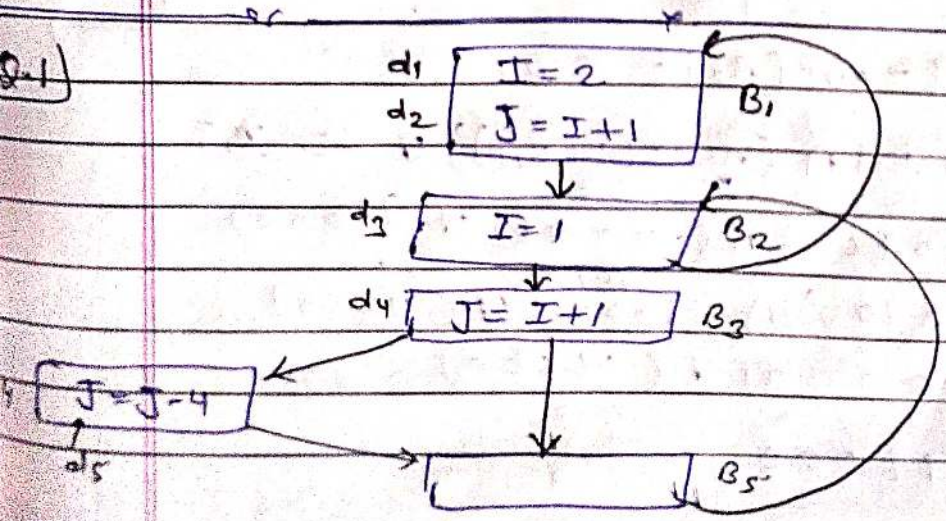
flow graph

Data-flow Equations :- There are two set of Equations are called data-flow Equations

① out[B] = IN[B] - KILL[B] U GEN[B]
= (IN[B] AND (~ KILL[B])) U GEN[B] }

② In[B] = ∪ OUT[B]
P is Precedesor of B

Q.1



Consider the following graph of given figure [flow graph]

P.T.O.

Find

① GEN and KILL for each block

② IN and OUT for reaching definitions

Sol:

①

Block B	GEN [B]	Bit vector d_1, d_2, d_3, d_4, d_5	KILL [B]	Bit vector d_1, d_2, d_3, d_4, d_5
B_1	$[d_1, d_2]$	1 1 0 0 0	$[d_3, d_4, d_5]$	0 0 1 1 1
B_2	$[d_3]$	0 0 1 0 0	$[d_1]$	1 0 0 0 0
B_3	$[d_4]$	0 0 0 1 0	$[d_2, d_5]$	0 1 0 0 1
B_4	$[d_5]$	0 0 0 0 1	$[d_2, d_4]$	0 1 0 1 0
B_5	ϕ	0 0 0 0 0	ϕ	0 0 0 0 0

② Initially $IB[B] = \phi$ $\quad OUT[B] = GEN[B]$

Block	IN [B]	OUT [B]
B_1	0 0 0 0 0	1 1 0 0 0 [out [5] = GEN[5]
B_2	0 0 0 0 0	0 0 1 0 0
B_3	0 0 0 0 0	0 0 0 1 0
B_4	0 0 0 0 0	0 0 0 0 1
B_5	0 0 0 0 0	0 0 0 0 0

For Pass-I ⇒ $IN[B_1] = OUT[B_2] = 00100$

$OUT[B_1] = (IN[B_1] \cap (\sim KILL[B_1])) \cup GEN[B_1]$

$= (00100) \cap (\sim 00111) \cup (11000)$

$= (00100) \cap (11000) \cup (11000)$ // 1's complement

$= (00000) \cup (11000)$

$OUT[B_1] = 11000$

$IN[B_2] = OUT[B_1] \cup OUT[B_5] = (11000) \cup (00000) = 11000$

$OUT[B_2] = IN[B_2] \cap (\sim KILL[B_2]) \cup GEN[B_2]$

$= 11000 \cap (\sim 10000) \cup (00100)$

1's complement ⇓

$= ((11000) \cap (01111)) \cup (00100) = 01100$

$IN[B_3] = OUT[B_2] = 01100$

$OUT[B_3] = IN[B_3] \cap [\sim KILL[B_3]) \cup GEN[B_3]$

$= ((01100) \cap (\sim 01001)) \cup (00010)$

⇓ 1's complement

$= ((01100) \cap (10110)) \cup (00010)$

$= (00100) \cup (00010) = 00110$

$IN[B_4] = OUT[B_3] = 00110$

$OUT[B_4] = (IN[B_4] \cap (\sim KILL[B_4])) \cup GEN[B_4]$

$= ((00110) \cap (\sim 01010)) \cup (00001)$

⇓ 1's complement

$= ((00110) \cap (10101)) \cup (00001)$

$= (00100) \cup (00001) = 00101$

$IN[B_5] = OUT[B_3] \cup OUT[B_4]$

$= (00110) \cup (00101) = 00111$

$OUT[B_5] = (IN[B_5] \cap (\sim KILL[B_5])) \cup GEN[B_5]$

$= ((00111) \cap (\sim 00000)) \cup (00000)$

$= ((00111) \cap (11111)) \cup (00000)$

$= (00111) \cup (00000) = 00111$

Similarly

$\cap$ = AND GATE $\cup$ = OR GATE

Pass-IV 01111 11000
11111 01111
01111 00110
00110 00101
00111 00111

VIJAYSHRI
PAGE NO.:
DATE:

Blocks	Pass-I		Pass-II		Pass-III	
	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]
B_1	00100	11000	01100	11000	01111	11000
B_2	11000	01100	11111	01111	11111	01111
B_3	01100	01100	01111	00110	01111	00110
B_4	00110	00101	00110	00101	00110	00101
B_5	00111	00111	00111	00111	00111	00111

Similarly Value calculated for ~~Pass~~ Pass-II, Pass-III, Pass-IV, ----

[Pass-II, Pass-III, Pass-IV, ---- calculate करने के GEN & KILL value उस table से use करेंगे लेकिन IN & out की value उससे पहले वाले Pass की value का use करेंगे]

Pass की value तब तक calculate करते जाते है जब तक all blocks की IN & out value same नही हो जाती

For Pass-II :

$IN[B_1] = OUT[B_2] = 01100$

$OUT[B_1] = (IN[B_1] \cap (\sim KILL(B_1])) \cup GEN[B_1]$

$= ((01100) \cap (-00111)) \cup (11000)$

$= ((01100) \cap (11000)) \cup (11000)$ (1's)

$= (01000) \cup (11000) = (11000)$

$IN[B_2] = OUT[B_1] \cup OUT[B_5] = (11000) \cup (00111) = 11111$

$OUT[B_2] = (IN[B_2] \cap (\sim KILL[B_2])) \cup GEN[B_2]$

$= ((11111) \cap (-10000)) \cup (00100)$

$= ((11111) \cap (01111)) \cup (00100) \Rightarrow (01111) \cup (00100)$

$= (01111)$

$IN[B_3] = OUT[B_2] = (01111)$