UNIT – IV

CODE OPTIMIZATION

Principle Sources of Optimization – Basic Blocks and flow graphs – Loop optimization and its types – DAG – Peephole Optimization – Dominators – Global Data Flow Analysis.

Principle Sources of Optimization:

Optimization:

* reduce the running time of code.
* transforms the code without changing its o/p.
* replaces longer sequence of operations or code by lesser and efficient sequence of operations which results in the same o/p.

Properties of efficient code transformation:

→ must be fast, it should not delay the overall compiling process.

→ must preserve the meaning of the code.

→ must increase the speed of the pgm.

Possible & easy transformations are:

* Rearranging the sequence of code.
* modifying intermediate code by address calculations and improving loops.

Types:

① Machine Dependant: Transformations take maximum advantage of memory hierarchy.

② Machine Independant: Transformations do not involve CPU registers and memory location.

→ principle sources of code optimization must preserve the semantics of the original program.

* Function preserving Transformations
  - Common sub-expression elimination
  - copy propagation
  - dead code elimination
  - constant folding

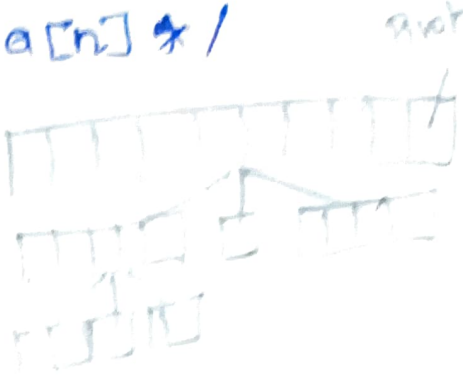
* Loop Optimizations
  - code motion
  - induction variable elimination
  - reduction in strength

Function preserving Transformation:

- improves a program without changing the function for computation.
- for each and every transformation, local and global optimizations will take place.

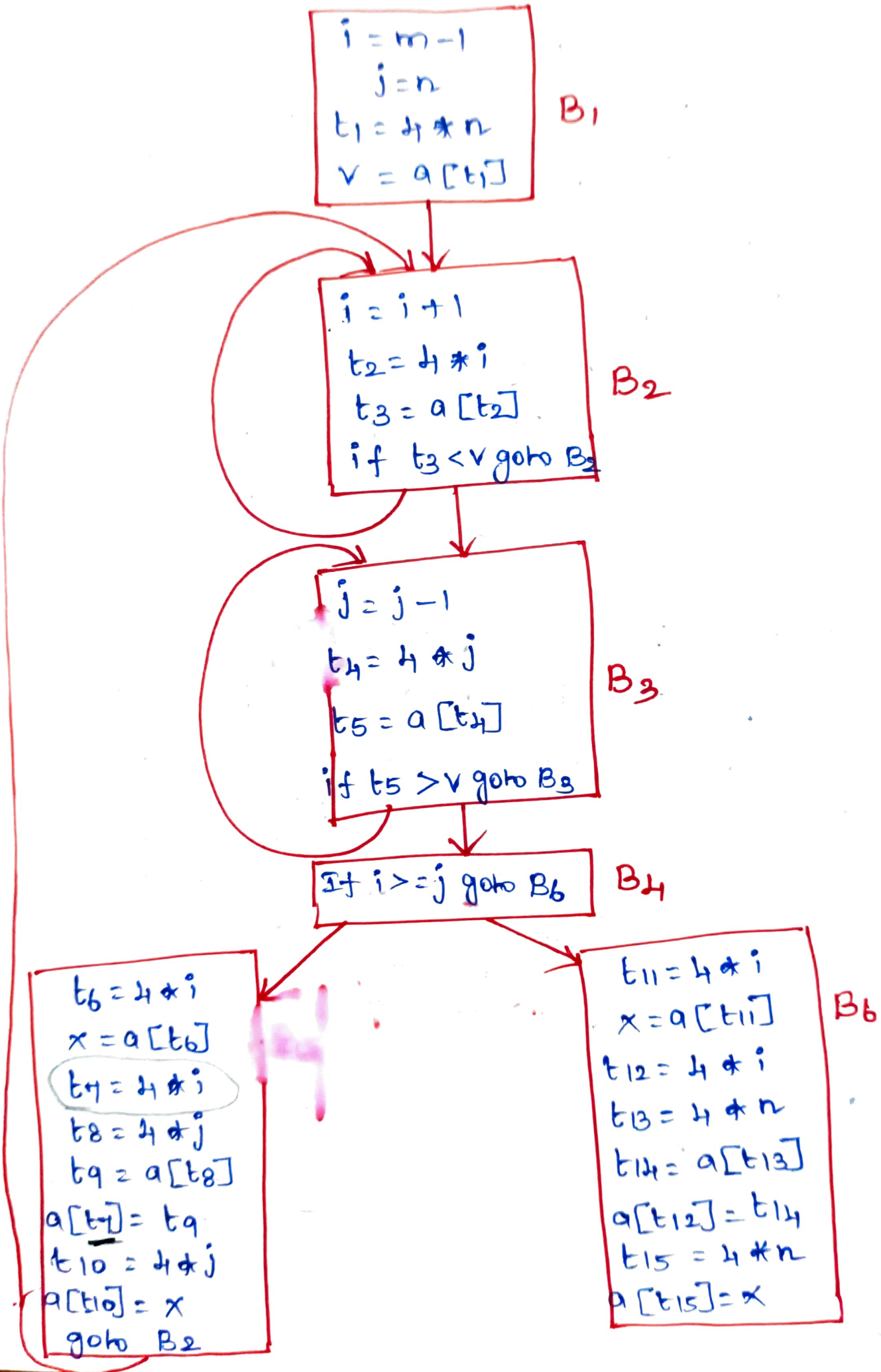
Eg:

```
void quicksort (int m, int n)
/* recursively sort a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return          // always big
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (i) {
        do i = i+1; while a[i] < v);   // left
        do j = j-1; while a[i] > v);   // right
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
        /* swap a[i], a[j] */
    }
    x = a[i]; a[j] = a[n]; a[n] = x;
    /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort (m, j);
    quicksort (i+1, n);
```



a[n] ⇒ b[t*n]
= a[t]

array
h*n

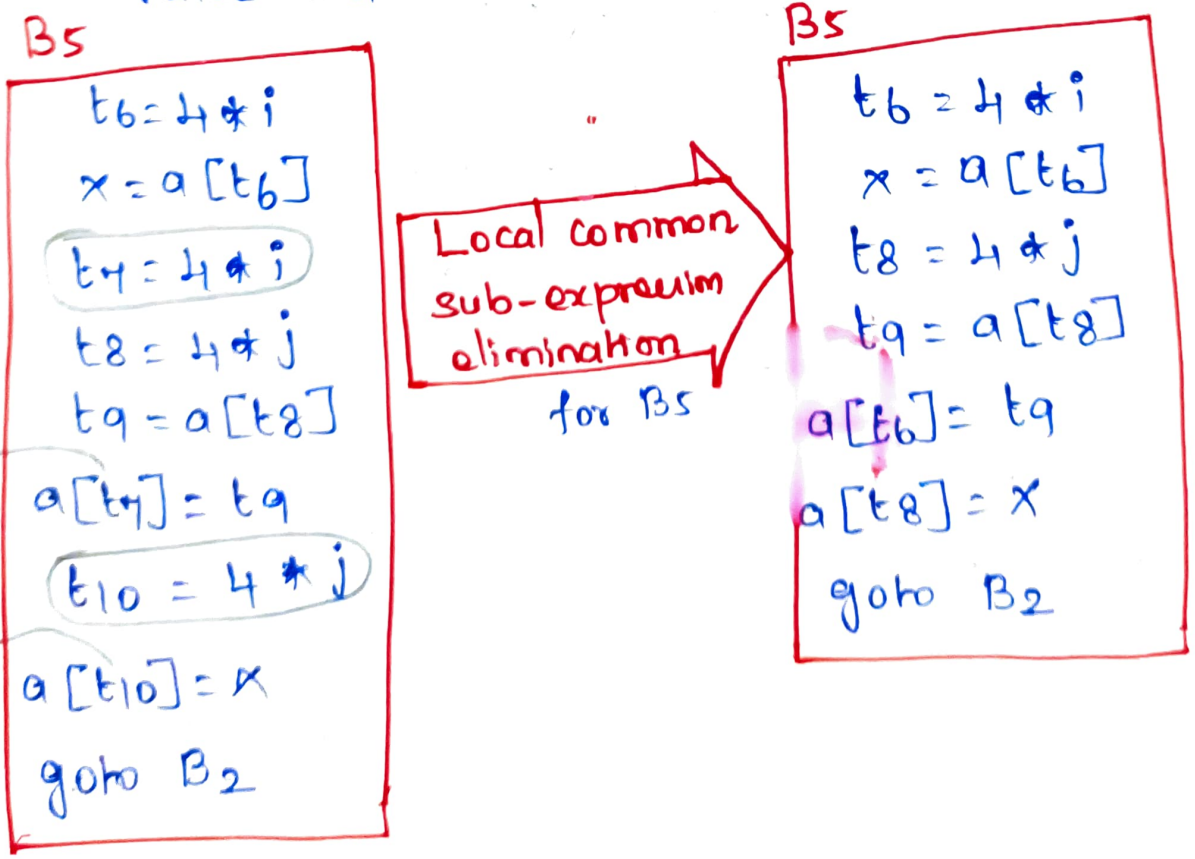

i = m - 1
j = n
t1 = 4 * n
v = a[t1]
B1
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B2
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B3
If i >= j goto B6
B4
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x
B6

Common sub-expression elimination:

* The expression which computes the same value that has already been computed in the common sub-expression.

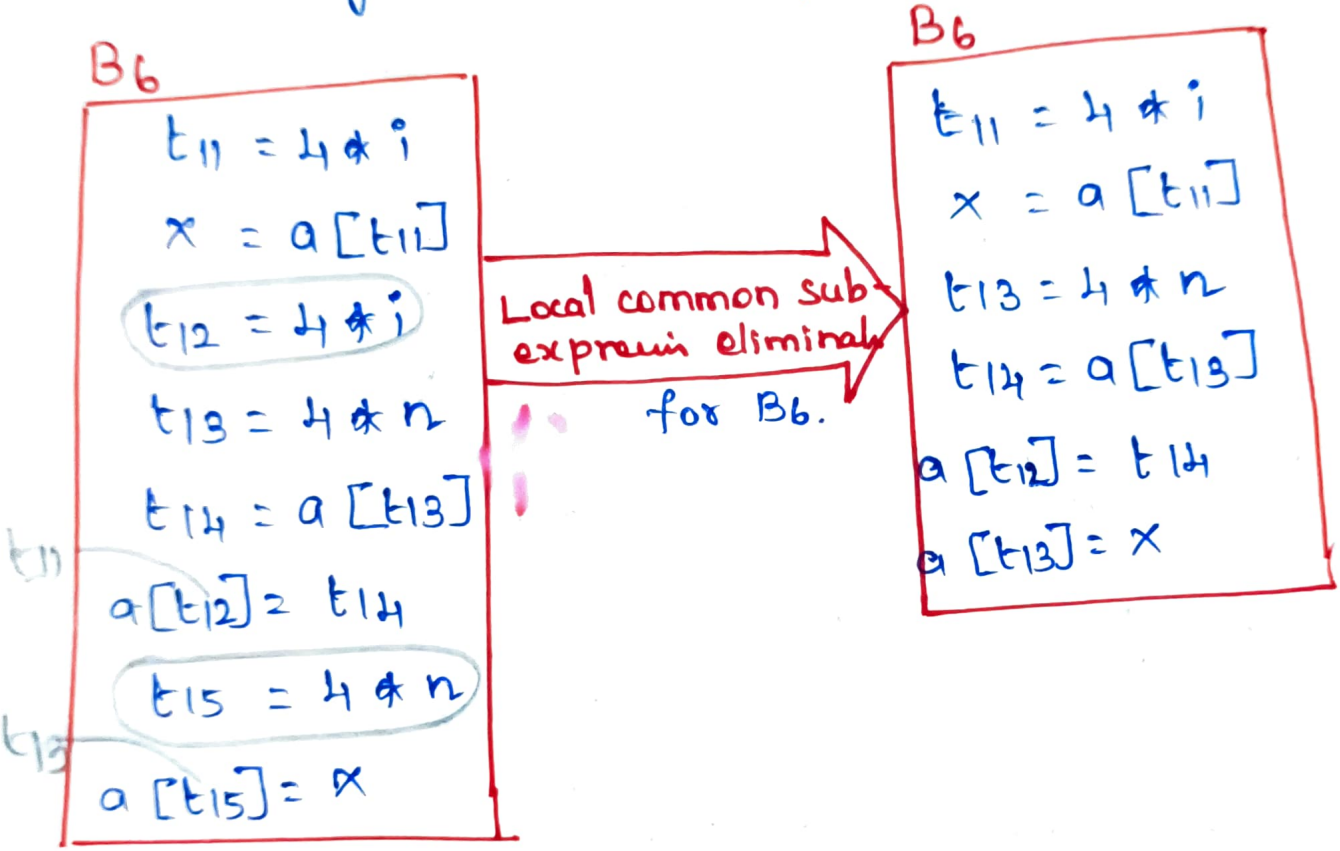
↳ an occurrence of an expression E is said to be a common sub-expression if E was previously computed and the values in E have not changed since the previous computation.

↳ it helps in replacing the common sub-expression by its previously computed value and hence reduces re-computation.



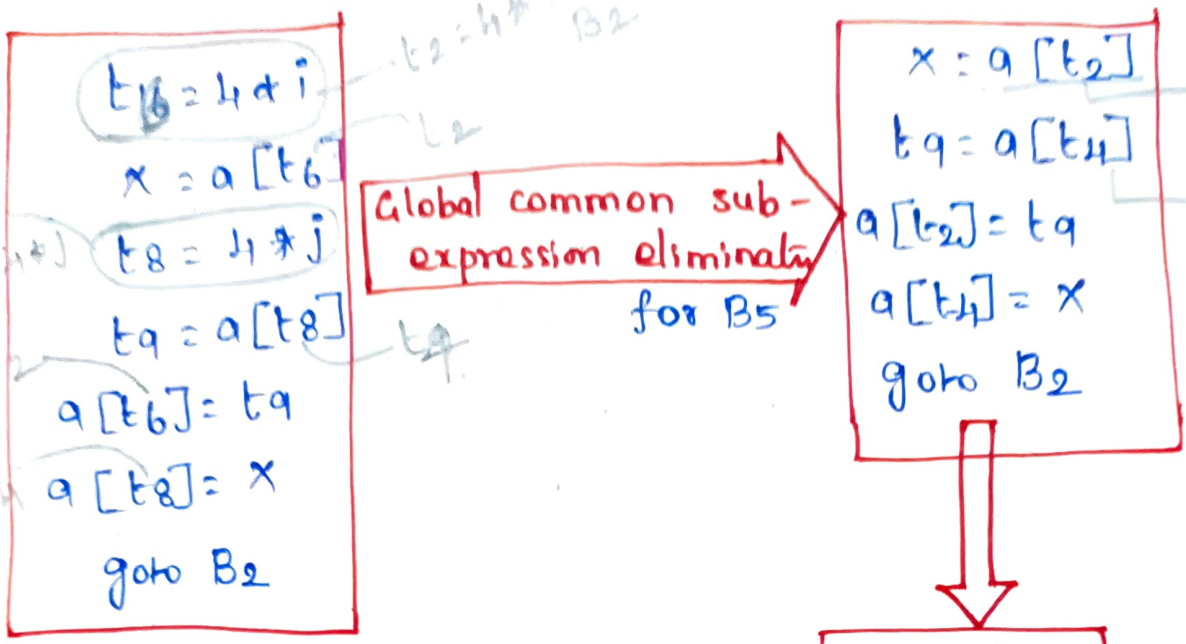
→ block B5, the assignments to t7 and t9 are found to be common sub-expression for t6 and t8 respectively and hence both are eliminated by replacing their occurrences with t6 and t8 respectively.

→ similarly B6 can be optimised as



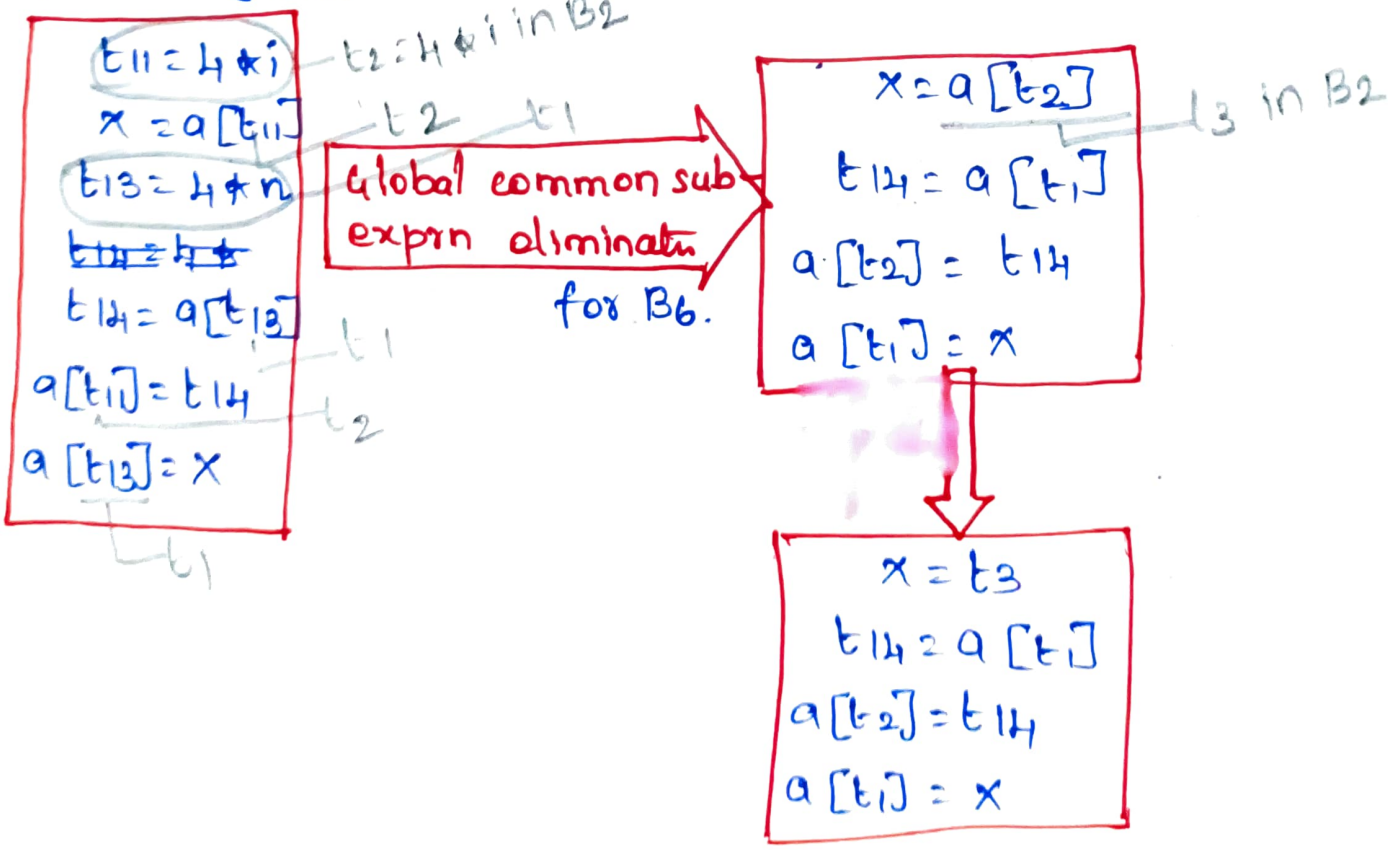
* In the transformed B5 and B6, common sub-expressions 4 * i and 4 * j are available.

* Compare B5 with B2 and B3 and the transformation can be done as follows:



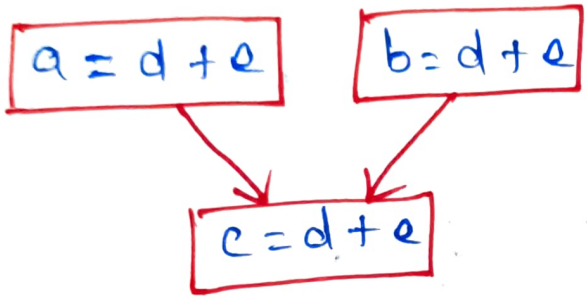
* In B2, t2 computes 4 * i and in B3, t4 computes 4 * j. Hence the computations of those two sub-expressions are eliminated in B5 and replacement occurs by the computed value.

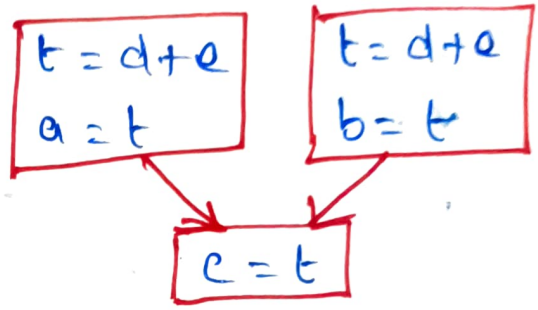
* Similarly, B6 can be optimized as,



Copy Propagation:

* copy statement are of the form $f = g$.
* done by replacing the occurrence of either g by f or f by g.

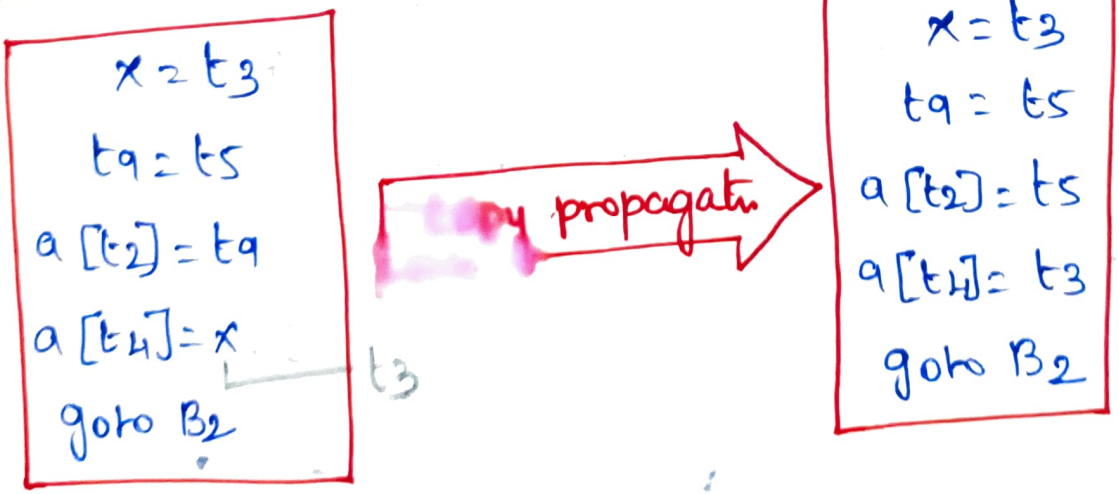




→ from global common sub-expr elimination for B_5 and B_6, it is clearly visible that $x = t_3$ and $t_4 = t_5$ are copy stmts.

→ Hence B_5 and B_6 can further be optimized as

B_5:



x = t3

t9 = t5

a[t2] = t9

a[t4] = x

goto B2

B6 :

```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
```

Copy Propagation

```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = t3
```

Dead code Elimination :

* A variable is said to be dead or useless if the value of the variable never gets used. otherwise it is said to be live variable.

* Dead code refers to the stmt which computes value that is never used.

```
if (debug)
print
.....
```

```
debug = FALSE
if (debug
Print.....
```

```
debug = FALSE
```

* The print stmt is executed if and only if the value of if condition holds (i.e., true).

* So, when the variable gets assigned with false, than for all cases of if conditional stmt the condition never satisfies which in turn make the print stmt not reachable (never gets exec).

→ hence the conditional stmt and its body becomes dead and are eliminated.

B5 :

```
x = t3
t9 = t5
a[t2] = t5
a[t4] = t3
goto B2
```

Dead code elimination for B5

```
a[t2] = t5
a[t4] = t3
goto B2
```

B6 :

```
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = t3
```

Dead code elimination for B6

```
t14 = a[t1]
a[t2] = t14
a[t1] = t3
```

Constant folding:

* process which replaces the computational statement that always computes a constant value by its value of constant.

Eg: The computation a = 2 * 3.14 can be replaced by a = 6.28.

Basic Blocks and flow graphs:

* Basic blocks are sequence of consecutive 3-addr stmts or instructions

Properties of basic block:

- Control can enter and exit only through the first and last stmt respectively in a basic block without any branching (or) halting.

Partitioning 3-addr stmts:

* 3-addr stmts can be partitioned into basic block as follows:

→ Finding the leader

- The first stmt of intermediate code is a leader
- Target of conditional and unconditional goto is a leader.
- Any inst immediately following conditional or unconditional jump is a leader.

→ The sequence of stmts from a leader to the stmt before the next leader constitutes a basic block, i.e., no two leaders are in same basic block.

Basic Block partitioning Algorithm:

Input: A sequence of three-address stmts.

Output: A list of basic blocks, with each stmt in exactly one block.

Method:

1. Determine set of leaders (first stmts)
   (i) First stmt of seq is a leader.
   (ii) Any target of a goto (conditional or unconditional) is a leader.
   (iii) Any stmt immediately following a goto (conditional or unconditional) is a leader.
2. For each leader, its basic block consists of the leader and all stmts upto but not including the next leader or the end of the program.

Eg

```
for i from 1 to 10 do
for j from 1 to 10 do
    a[i,j] = 0.0;
for i from 1 to 10 do
    a[i,i] = 1.0;
```

Convert the aforementioned source code into three-address stmts as follows

1. i = 1
2. j = 1
3. $t_1 = 10 * i$
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. $t_5 = i - 1$
14. $t_6 = 88 * t_5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

Constructing 3-addr stmt from source code.

Leaders are,

1 – First stmt is a Leader.

2 – Target of conditional or unconditional goto is a Leader.

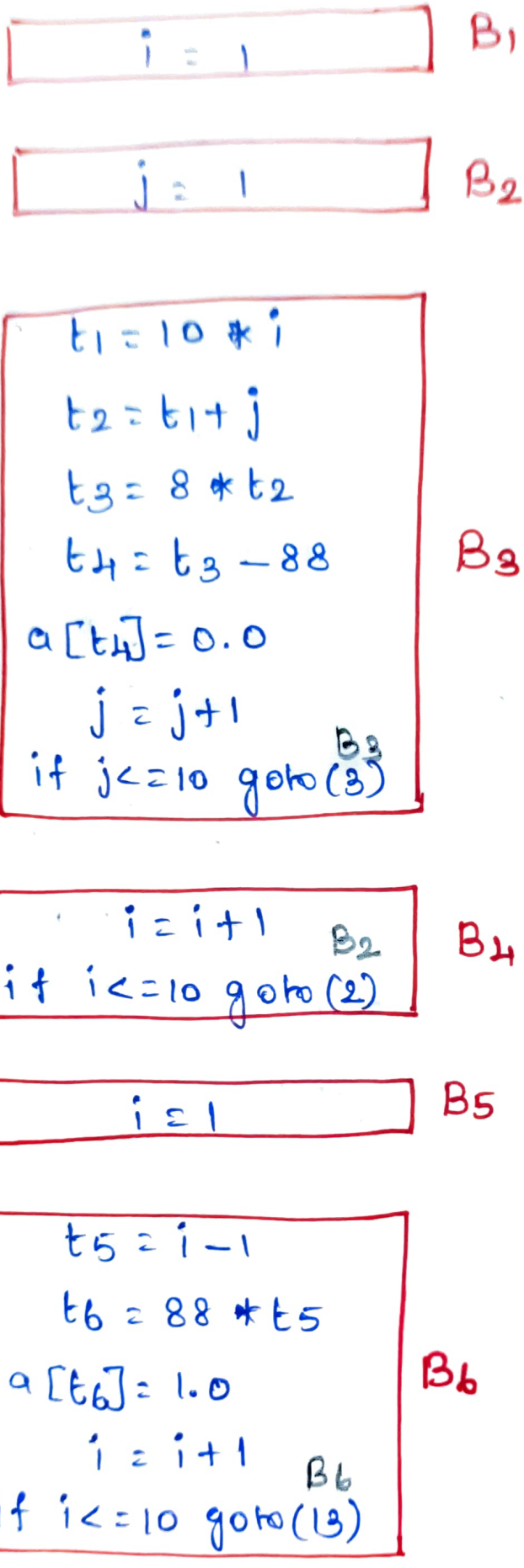
3 – Target of conditional or unconditional goto is a Leader.

10 – statement immediately following conditional goto is a Leader.

12 – statement immediately following conditional goto is a Leader.

13 – Target of conditional or unconditional goto is a Leader.

⟹ Hence, basic blocks are formed by having no two leaders in same basic block. Since six leaders are in transformed 3-address stmts, it leads to emergence of six basic blocks as follows:


i = 1
B1
j = 1
B2
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
a[t4] = 0.0
j = j + 1
if j <= 10 goto (3)
B3
i = i + 1
if i <= 10 goto (2)
B2
B4
i = 1
B5
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto (13)
B6
B6

Transformation on Basic Blocks:

- should not change the set of expn computed by it.
- A basic block can undergo the following transformations.
  - * structure preserving transformation.
  - * Algebraic transformation.

Structure Preserving transformation:

- → Common sub-expn elimination.
- → Dead code elimination
- → interchanging independent stmts

Algebraic transformation:

Next-use information

Flow Graph

Loops

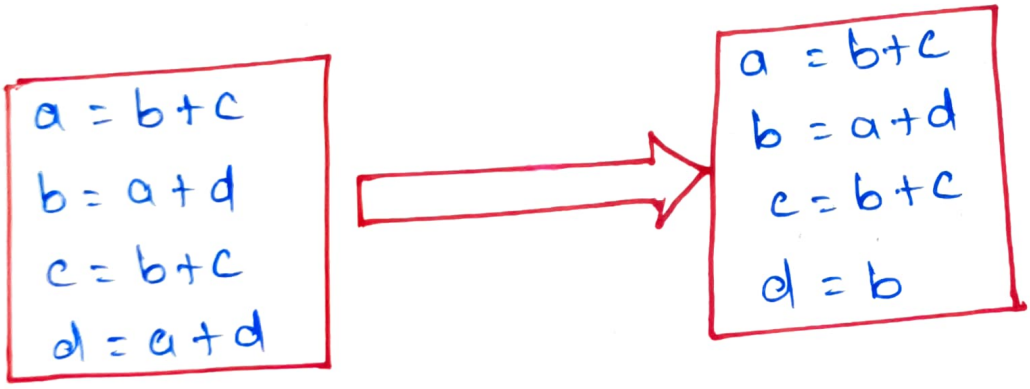
- identification of loop

Structure Preserving Transformations:

Common sub-expn elimination:

→ A expn is said to be a common sub-expn if it was previously computed and the value of variables involved in it have not changed since the previous computation.

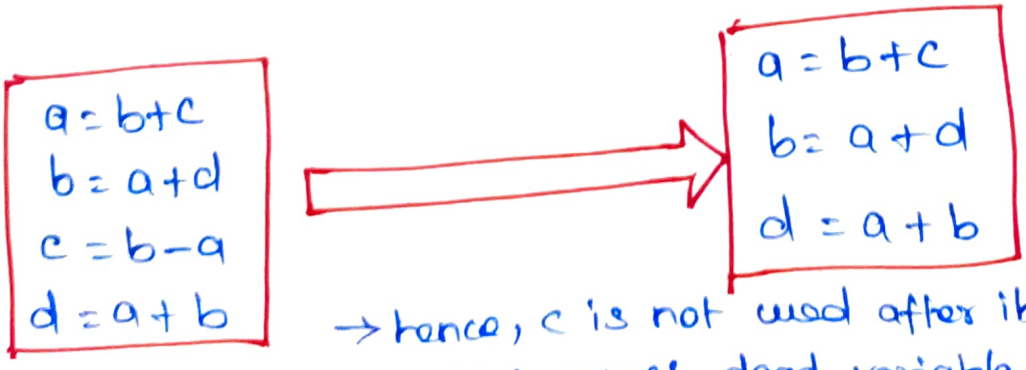
→ b and d computes the same expn and hence the computation on d is replaced by b.



Dead code elimination:

→ A variable is live, if the value is used subsequently otherwise it is dead.

→ A stmt involving dead variable can be removed from the basic block.



→ hence, c is not used after its computation so it becomes dead variable and it is termed as dead code.

Renaming of temporary variable:

* If a computation involves a temporary variable, then the variable can be removed.

Eg $t = x + y$ ⟶ $u = x + y$

* After renaming, all the uses of t will be replaced by u so that value of the basic block does not get affected.

Interchanging independent stmts:

- stmts can be interchanged if they are independent.
- interchanging of stmt should not affect the value of the basic block not even the operands.

$t = x + y$
$u = a + b$

⟶

$u = a + b$
$t = x + y$

Algebraic transformations:

- Algebraic transformations perform its computation based on rules of algebraic identities.
- It aims at simplifying the calculations.
- It also reduces the computational cost of statement.

Eg:

$x + 0 = 0 + x = x$

$x - 0 = x$

$x * 1 = 1 * x = x$

$x = y ** 2 \longleftrightarrow x = y * y$

Next-use Information:

- used to know when the value of the variable will be used next.
- the variables will reside in register based on the referencing frequency.
- if the variable is not referenced subsequently, the register is assigned to another variable.

Eg
```
i : # x = 5
j : # a = x + 3
```

→ where, i, j are stmts in which variable x is assigned a value and is used respectively.

→ from the above example, it is inferred that the stmt j uses the value of x computed at i as there is ~~not~~ no stmt for x that intervenes along the path from i to j.

Flow Graph:

* Intermediate code can be represented by a graph called as flow graph.
* Flow graph contains node and edges in which nodes represent the basic block and edges represent the flow of ctrl or information among basic blocks.
* Edge from block B to block C indicates that the first instruction of C immediately follows the last statement of B.

Loop Optimization:

- m/c independant optimization.
- most execution time of a pgm is spent on loops.
- decreasing the no.of instruction in an inner loop improves the running time of a program.

Types:

1) Code motion
2) Induction variable Elimination
3) Reduction in strength.

Code motion:

- moves the code outside the loop.

Eg:

```
while (i<100)
{ a = limit/2 + i;
  i++;
}
```

loop invariant computation

```
t = limit/2;
while (i<100)
{ a = t + i;
  i++;
}
```

- if an expression gives the same result independant of number of times the loop is executed, then the expression can be placed outside the loop.

Induction variable Elimination:

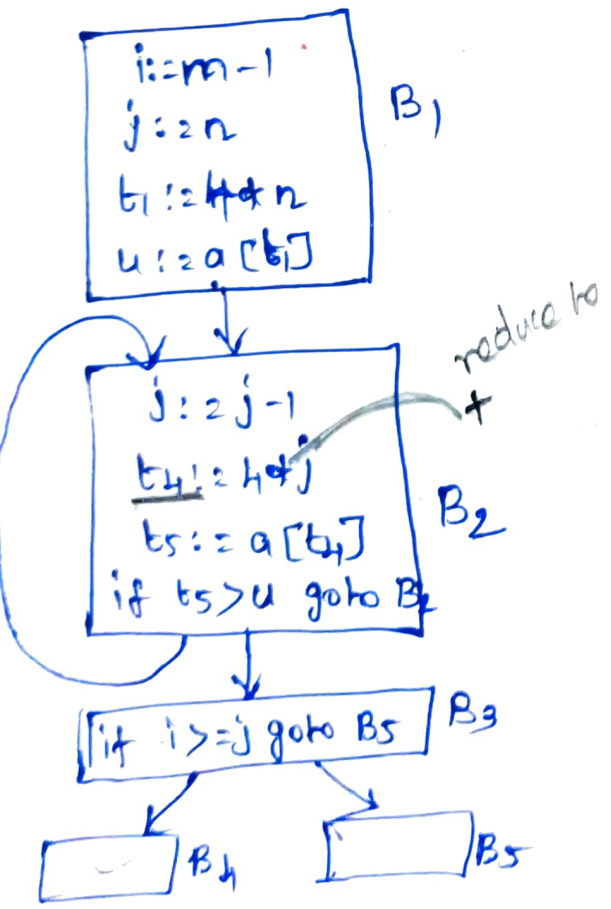
- An induction variable of the form

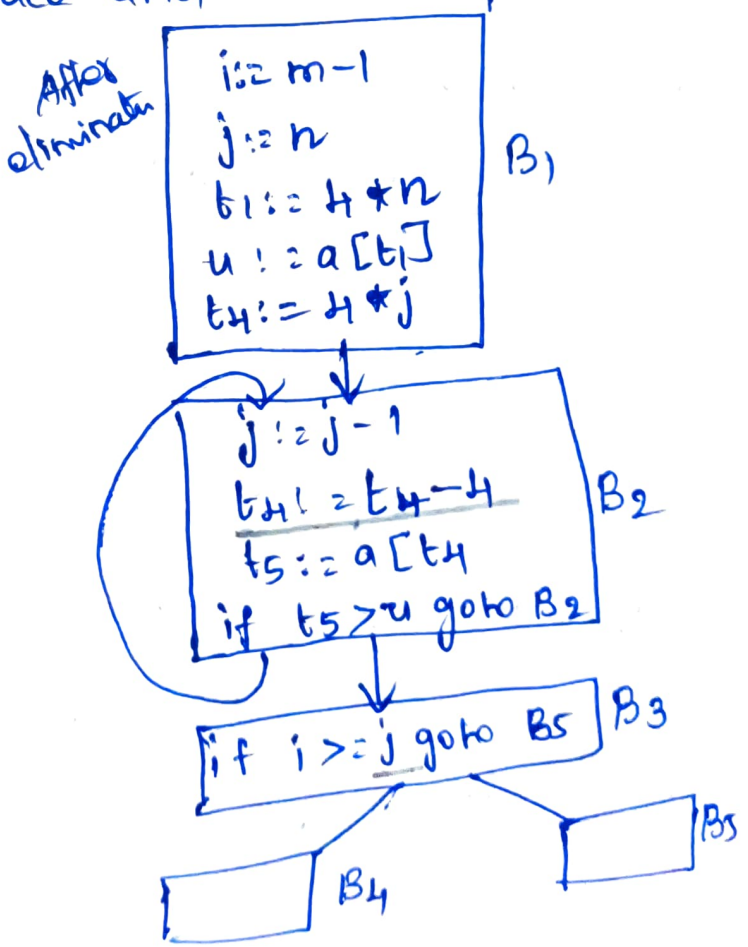
  $x = x + c$ (or) $x = x - c$.

- That is, a variable x is inductive if there is a constant c (either +ve (or) -ve), such that for each assignment of x, its value increases (or decreases) by c.

* It is used to replace the variable from inner loop.

* It can reduce the no. of additions in a loop. It improves both code space and runtime performance.





```
i = 1
while (i<10)
{
  y = i*4;
  i = i+1;
}
```

⟹

```
t = 4
while (t<40)
{
  y = t;
  t = t+4;
}
```

y=4 y=8 y=12 ... y=16 y=36

Reduction in strength:

→ strength reduction is used to replace the expensive operation by the cheaper one on the target m/c.

→ addition of a constant is cheaper than a multiplication. so we can replace multiplication with an addition with the loop.

→ multiplication is cheaper than exponent. So we can replace exponent with multiplication within the loop.

Exp > mul > add

Eg:

```
while (i<10)
{
  j = 3*i+1;
  a[i] = a[j]-2;
  i = i+2;
}
```

```
s = 3*i+1;
while (i<10)
{
  j = s;
  a[j] = a[j]-2
  i = i+2;
  s = s+6;
}
```

It is cheaper to compute s = s+6 than j = 3*i

DAG Representation of Basic Block:

→ Determines common sub expression.

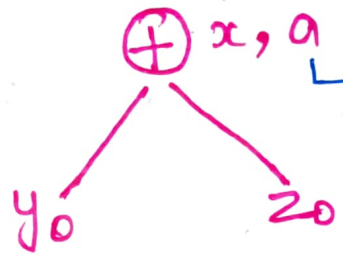
1) Leaves are labeled by variable names or Constant
   Initial values are subscribed with 0.

2) Interior nodes are operators.

3) Internal nodes also represent result of the expression.

$x := y+z$

$a := y+z$



Eg:

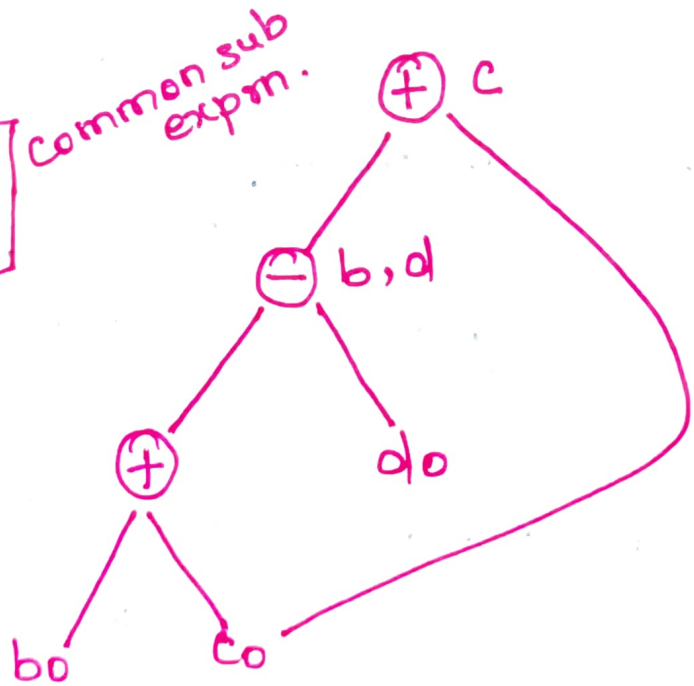
$a := b+c$

$b := a-d$

$c := b+c$

$d := a-d$

common sub expn.



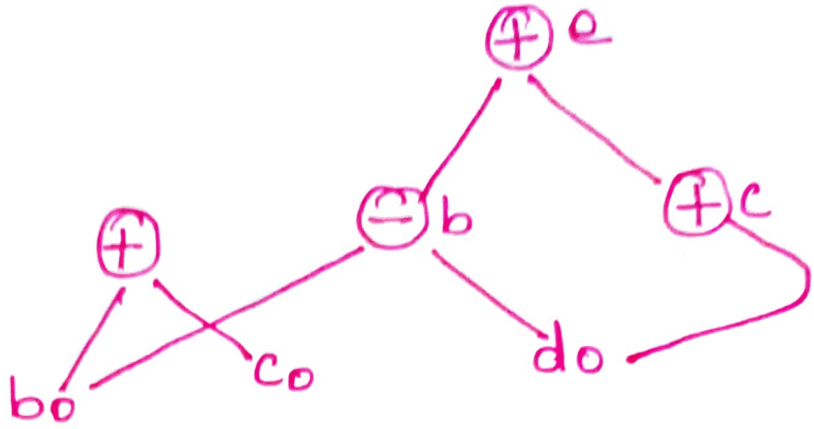
Eg:

a := b + c

b := b - d

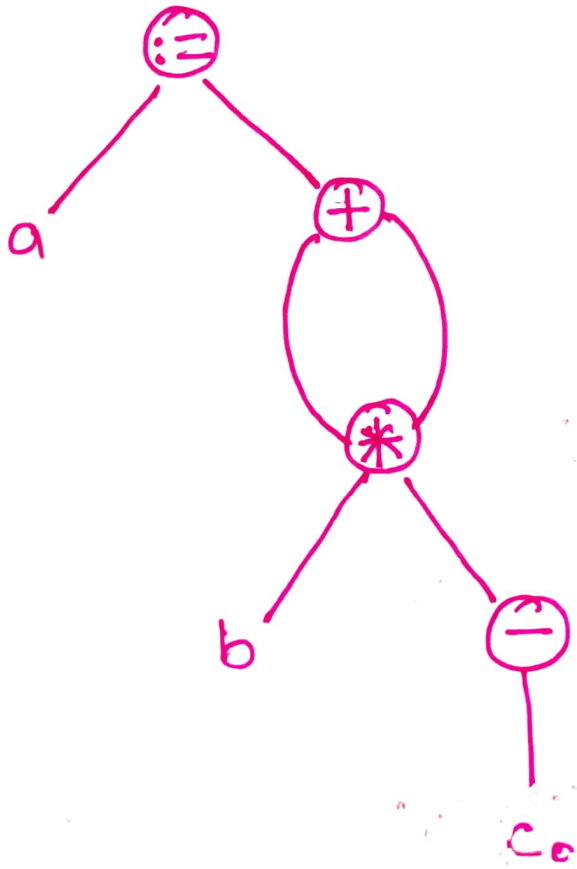
c := c + d

e := b + c



Eg:

a := b * -c + b * -c



Eg:

(1) $t_1 = 4 * i$

(2) $t_2 = a[t_1]$

(3) $t_3 = 4 * i$

(4) $t_4 = b[t_3]$

(5) $t_5 = t_2 * t_4$

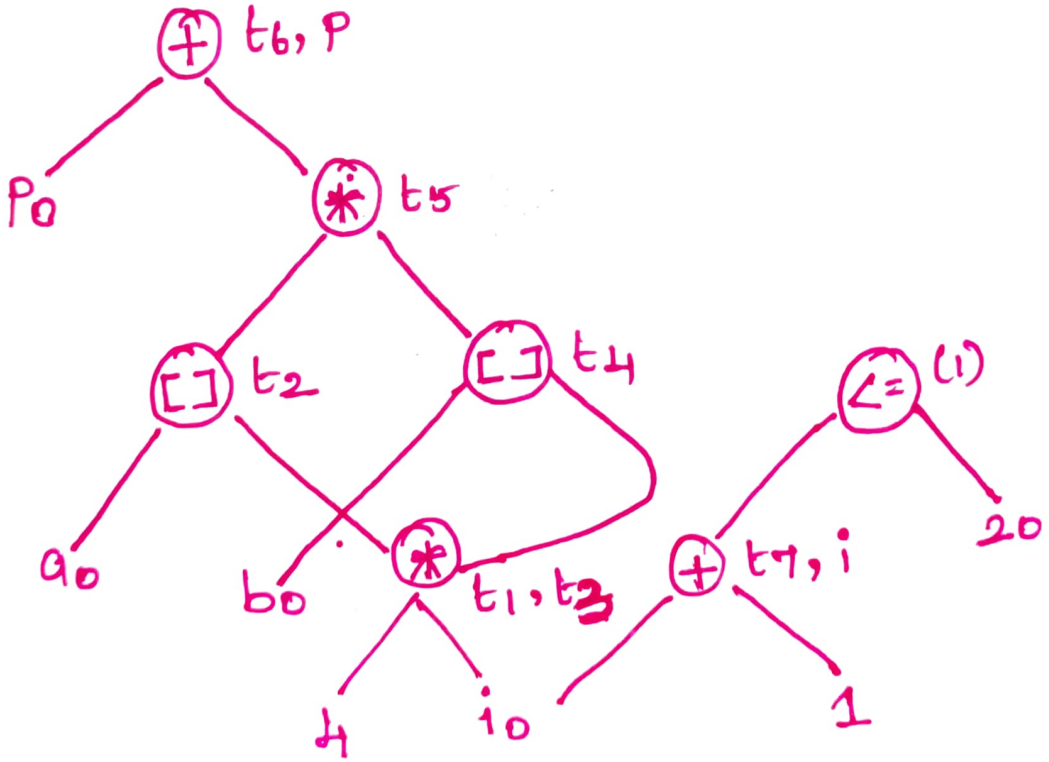
(6) $t_6 = P + t_5$

(7) $P = t_6$ — no need to create a new Label

(8) $t_7 = i + 1$

(9) $i = t_7$

(10) if $i <= 20$ goto (1)



Peephole Optimization:

Peephole → small moving window on the target program.

Peephole optimization → Replace long sequence of target instruction by a shorter or faster sequence.

→ can be implement to intermediate or target code

Techniques:

1) Redundant Instruction Elimination.
2) unreachable code Elimination.
3) Flow-of-ctrl optimization.
4) Algebraic simplifications.
5) Use of machine idioms.

Redundant Instruction Elimination:

1) MOV R0, x → load instruction
2) MOV x, R0 → x moved to R0

↓

redundant load and store.

↓

[if procoded by any label we can't eliminate.]

Unreachable code Elimination:

```
x = 0;
if (x == 1) — never become true
{
  a = b; — unreachable.
}
```

~~x = 0;~~
~~if x ≠ 1 goto L1~~

Intermediate code.

```
x = 0
if x = 1 goto L1
goto L2
L1: a = b;
L2:
```

⟹

```
x = 0
if x ≠ 1 goto L2
(a = b) x
L2:
```

Flow of ctrl optimization

```
goto L1
...
L1: goto L2
...
L2: a = b
```

⟹

```
goto L2
...
L1: goto L2
...
L2: a = b
```

if any stmt is not target of other stmt, that can be removed.

Algebraic simplification:

$x = x + 0$
$x = x * 1$
} — x not changed

↓

Reduction in strength.

$x \uparrow 2 = x * x$
$2 * x = x + x$
$x * 2 = x << 1$ — left shift
$x / 2 = x >> 1$ — right shift.

Use of machine idioms:

$i = i + 1$
↳ INC i

$i = i - 1$
↳ DEC i

Dominators:

* A node d dominates n → d dom n — node d of a flow graph dominates node n if every path from the initial node of the flow graph to n goes through d.

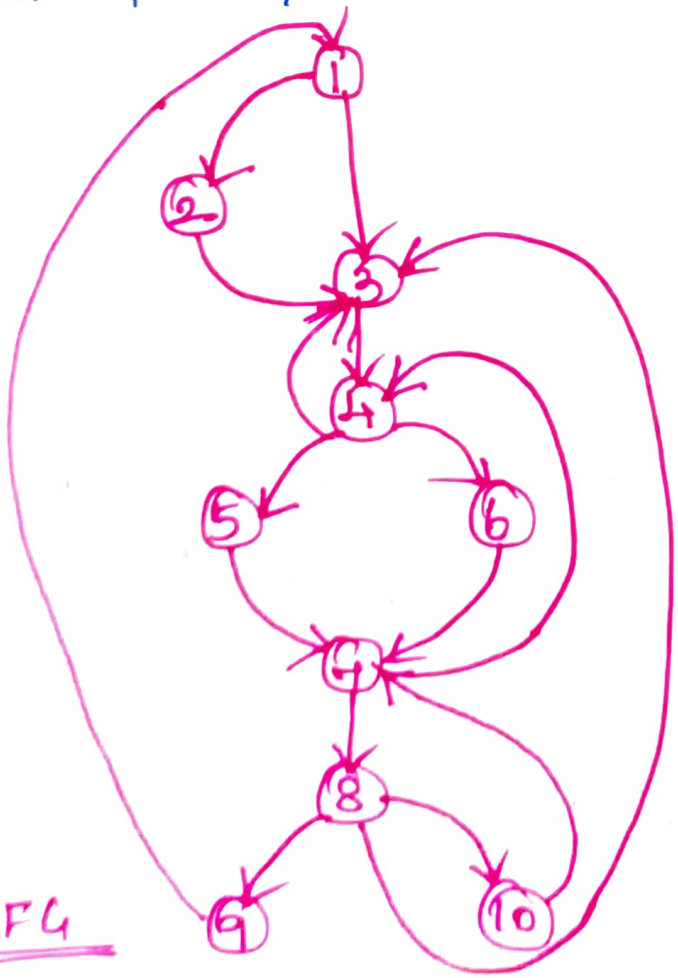
* Every node dominates itself and the entry of a loop dominates all nodes in the loop.

* Useful way of presenting dominator informat is in a tree., called the dominator tree.

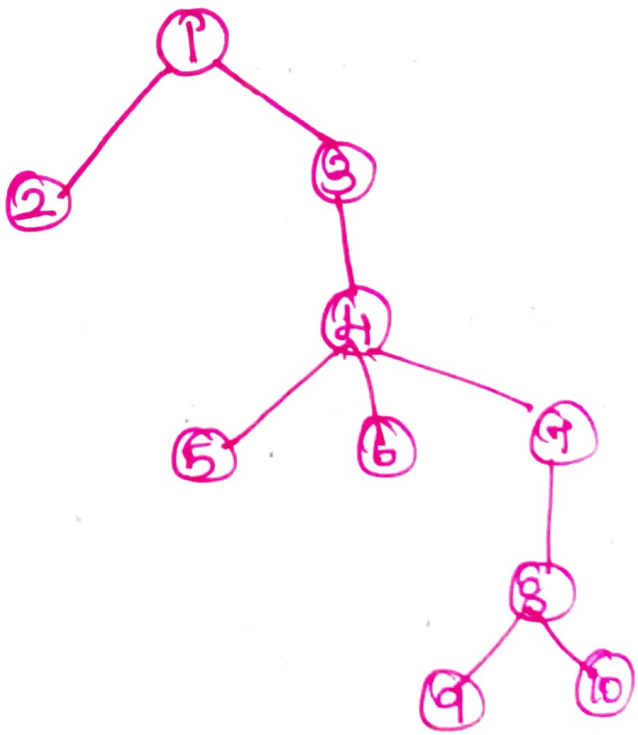
* The dominator tree is in which the initial node is the root and each node d dominates only its descendants in the tree.

* The immediate dominator m of a node n is the last dominator on the path from the initial node to n (m dom n).

* The immediate dominator m has the property that if $d \neq n$ and d dom n then d dom m.



CFG



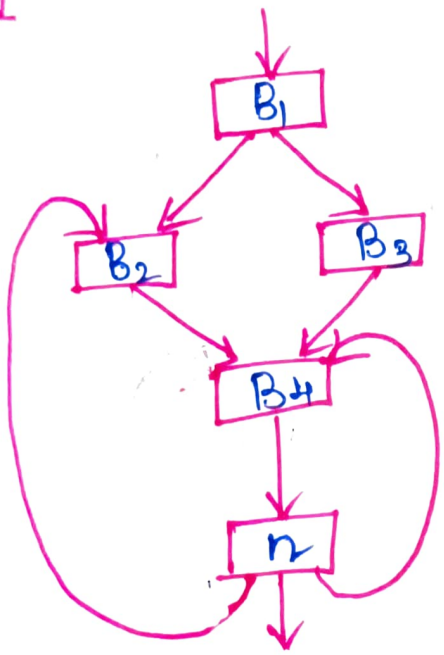
Dominator tree

→ Node **d** dominates node **n** in a graph

d dom n iff

- Every path from the start node to node **n** goes through node **d**.
- A node dominates itself.

Eg



Dominators of n?

B_1 dominates n

B_2 does not dominate n

B_3 does not dominate n

B_4 dominates n

n dominates n

Global Data flow Analysis:

→ it collects the information about the entire pgm and distributed the information to each block in the flow graph.

→ A typical Data flow eqn:

$$out[S] = gen[S] \cup [in[S] - kill[S]]$$

→ The current value of the variable can be identified using data-flow analysis.

The information at the end of the stmt is either generated within the stmt or enters at the beginning and is not killed as ctrl flow through stmt.

— This eqn is called data flow equation.

* The data flow eqns can be solved with 3 factors

(i) information abt generating and killing depends on the data-flow analysis.

(ii) data-flow analysis is affected by the ctrl constructs in the pgm because data flow along ctrl paths.

(iii) some changes will be there while doing through stmts like procedure calls, assignments through pointer variables and assignments to array variables.

— while performing global data flow analysis, there is a change of encountering several points and paths.

Points and paths:

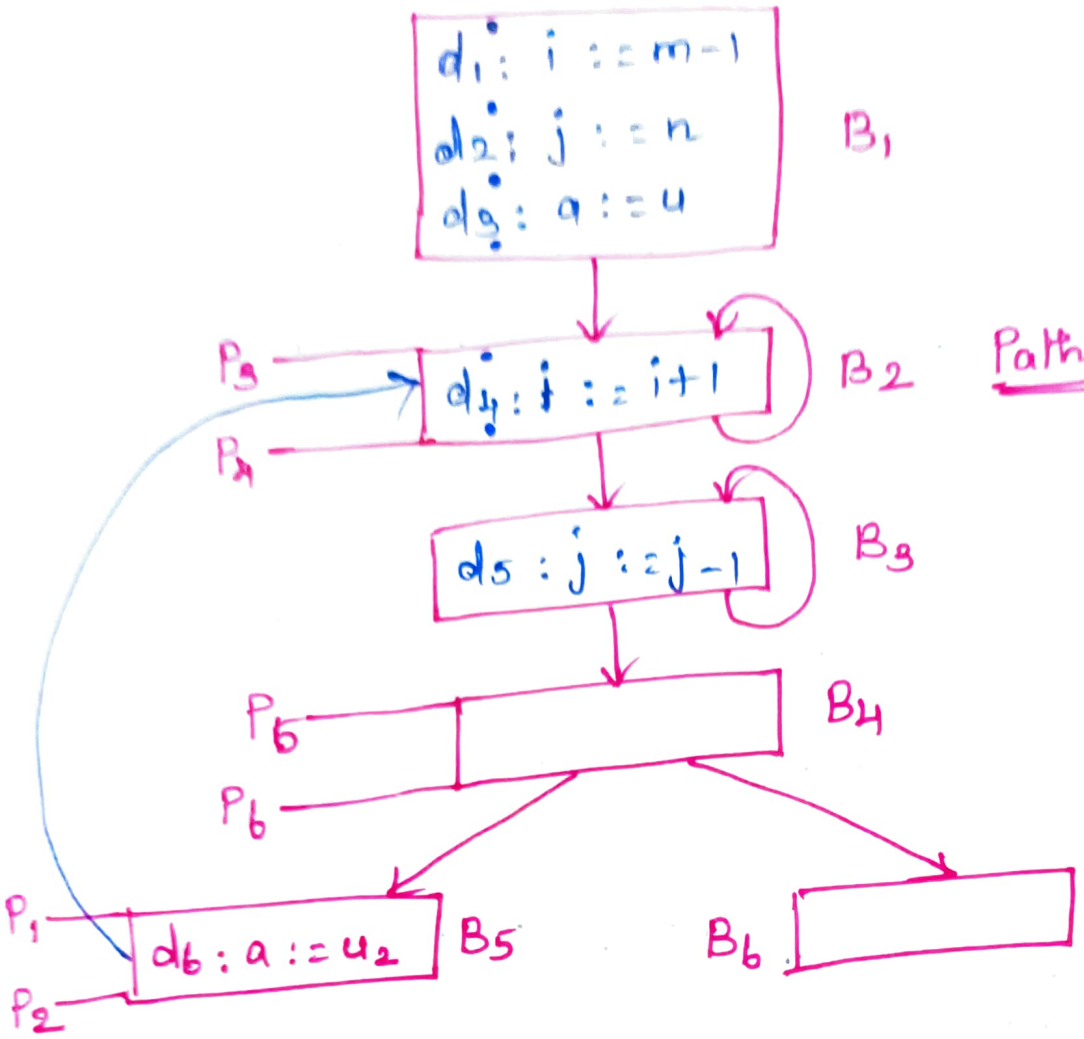
* A point is the instance at which the state of the variable is defined. In a basic block, a point is defined b/w two adjacent stmt.

* A location b/w two consecutive stmts.
* A location before first stmt of the Basic block.
* A location after the last stmt of the Basic block.

→ A path from a point p_1 to p_n is a sequence of points.

$p_1, p_2, \ldots p_n$ such that for each $i : 1 \leq i \leq n$

- p_i is a point immediately preceding a stmt and p_{i+1} is the point immediately following that statement in the same block. (or)
- p_i is the last point of some block and p_{i+1} is first point in the successor block.



Reaching Definition:

* Defn of a variable x is a stmt that assign a value to x
* Unambiguous Defn: The stmt that certainly assigns a value to x.

  Eg: Assignments to x, read a value from I/O device to x.
* Ambiguous Defn: stmt that may assign a value to x.

* call to procedure with x as parameter (call by ref).

* call to a procedure can access x (x being in the scope of the procedure)

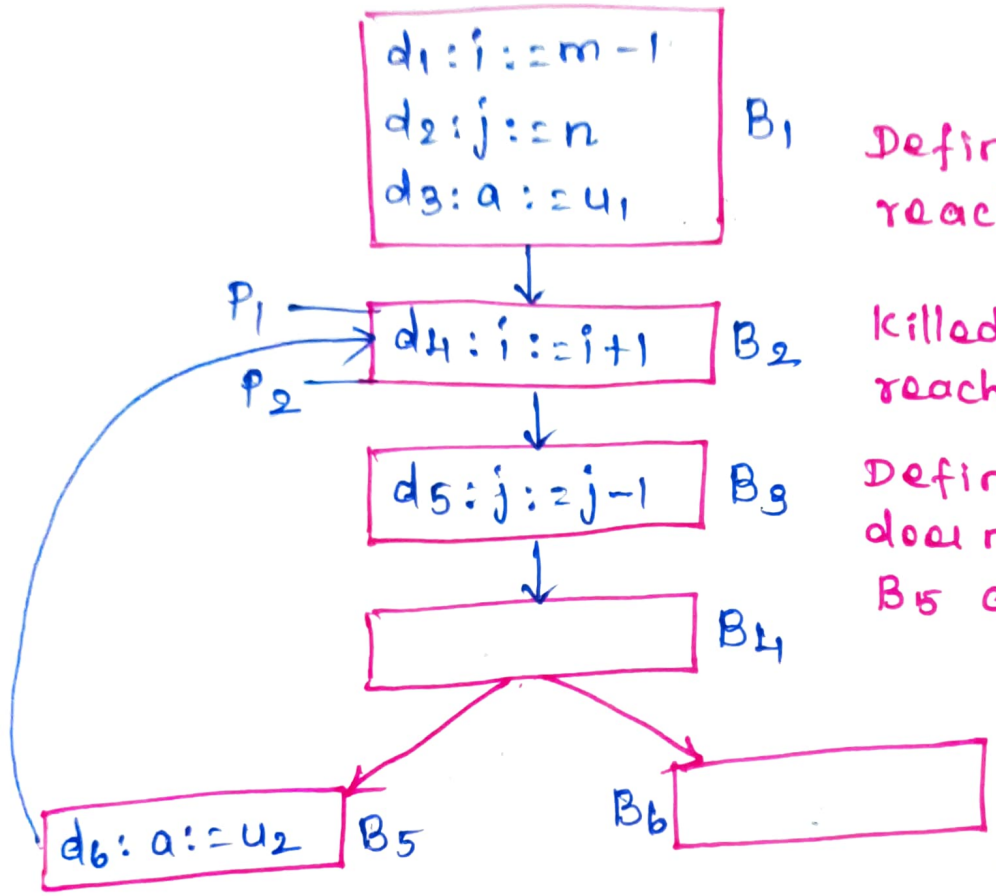
* x is an alias for some other variable (aliasing).

* Assignment through a pointer that could refer x.

Proc:

1. A definition d reaches a point P.
   - there is a path from the point immediately following d to P and,
   - d is not killed along the path (ie. there is not redefinition of the same variable in the path)
2. A definition of a variable is killed b/w two points when there is another definition of that variable along the path.

Eg: Reaching Definition.



Definition of i (d_1) reaches P_1

killed as d_4, does not reach P_2

Definition of i (d_1) does not reach B_3, B_4, B_5 and B_6

* Data flow analysis of a structured pgm.
* structured pgms have well defined loop constructs.
  — the resultant flow graph is always reducible.
* without loss of generality we only consider while-do and if-then-else control constructs.

$S \rightarrow$ id := E | S ; S
| if E then S else S | do S while E

$E \rightarrow$ id + id | id

The NT represent regions.

Data Flow Equations:

* Each region (or NT) has four Attributes:

1. gen[S]: set of definitions generated by the block S.
   - if a definition d is in gen[S], then d reaches the end of blocks.

2. kill[S]: set of definitions killed by block S.
   - if d is in kill[S], d never reaches the end of block S.
   - every path from the beginning of S, to end of S must have a definition for a. (where a is defined by d).

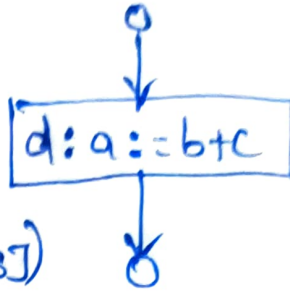
3. in[S]: The set of definition those are live at the entry point of block S.

4. out[S]: The set of defn those are live at the exit point of block S.

* The data flow eqns are inductive or syntax directed.
   - gen and kill are synthesized attributes.
   - in is an inherited attribute.

gen [S] = {d}

kill [S] = Da − {d}



out [S] = gen[S] ∪ (in[S] − kill[S])

Region :

A graph $G' = (N', E')$ which is portion of the control flow graph G.

- The set of nodes N' is in G' such that
  - N' includes the header h.
  - h dominates all node in N'.
- The set of edges E' is in G' such that
  - All edges $a \rightarrow b$ such that a, b are in N'.

Components :

* Region consisting of a stmts :
  - Ctrl ~~flow~~ can flow to only one block outside the region.
* loop is a special case of a region that is strongly connected and includes all its back edges.
* Dummy blocks with no stmts are used as technical convenience (indicates as open circles).

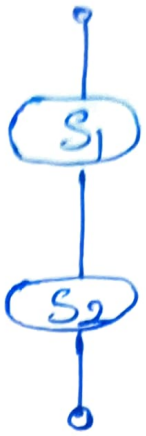
Composition of Regions:

$S \rightarrow id := E \mid S ; S$

$\mid$ if E then S else S

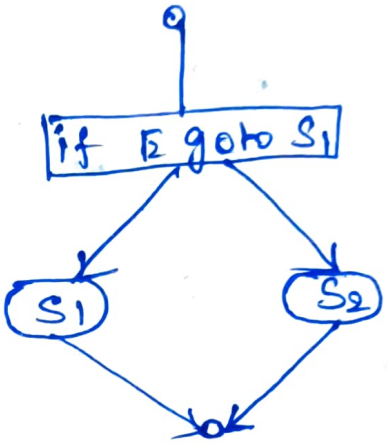
$\mid$ do S while E

$E \rightarrow id + id \mid id$



$S \rightarrow S_1 ; S_2$

$S \rightarrow$ if E then S_1 else S_2



$S \rightarrow$ do S_1 while E