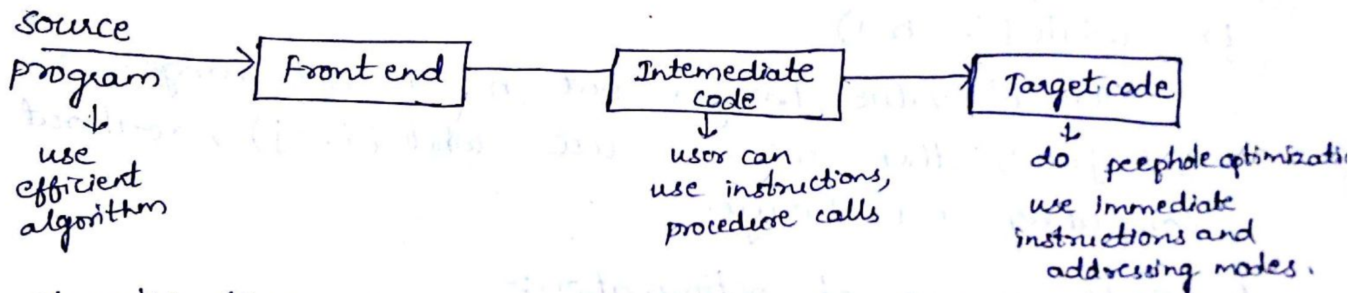
CODE OPTIMIZATION

Code Optimization :-

Code optimization is required to produce efficient target code. There are 2 important issues that need to be consider, while applying the code optimization,

i.) Semantic equivalence of source program, should not be changed

ii.) Program efficiency must be achieved without changing the algorithm of source program.



Classification of optimization :-

1.) Machine dependent optimization

2) Machine independent optimization

Machine dependent optimization :-

It is based on the characteristics of the target machine for the instruction set used and addressing modes used for the instruction to produce the efficient target code.

- Allocation of sufficient number of resources.
- Using immediate instruction whenever necessary.

Machine Independent Optimization:-

It is based on the characteristics of programming languages for a appropriate programming structure and usage of efficient arithmetic properties inorder to generate efficient code.

i.) From the source program eliminate unreachable code.

ii.) Use alternative equivalent sequence of instructions that will produce efficient target code.

iii.) Move two or more identical computations at one place and make use of the result instead of each time computing the expressions.

Ex:- while(i<=n-1)

Here 'i' value changes but n-1 doesnot change. So let j=n-1, then we can use while(i<=j), without computing n-1 always.

Principle Sources of optimization:-

1) Common Subexpression Elemination:-

An expression 'E' is called common sub expression if it was previously computed and the values of the variables in expression 'E' have not change since previous computation. In this we have to avoid recomputing of the expression, if its value is already being computed.

Ex:-

$$t_1 := 4\times i \qquad\qquad t_1 := 4\times i$$
$$n := a[t_1] \qquad\qquad n := a[t_1]$$
$$t_2 := 4\times i \quad \xrightarrow{\text{After elimination}} \quad t_3 := 4\times j$$
$$t_3 := 4\times j \qquad\qquad t_4 := a[t_3]$$
$$t_4 := a[t_3] \qquad\qquad a[t_1] := t_4$$
$$a[t_2] := t_4 \qquad\qquad a[t_3] := n$$
$$t_5 := 4\times j$$
$$a[t_5] := n$$

The result of both is same but difference is we eliminate repeated statements in RHS.

2) <u>Copy</u> <u>Propagation</u>:-

It means use of one variable instead of another variable. If $x:=y$ is a copy statement then copy propagation is a kind of transformation in which use 'y' for 'x', whenever possible, after copy statement $x:=y$

Ex:-

$$n := t_1 \qquad\qquad n := t_1$$
$$a[t_1] := t_2 \quad \longrightarrow \quad a[t_1] := t_2$$
$$a[t_3] := n \qquad\qquad a[t_3] := t_1$$

M

3) <u>Constant folding</u>:-

In which the constant expressions are calculated during the compilation. Here the value of expression is a constant and using the constant, do the constant folding.

Ex:- const $max := 3$

$$i := 2 + max \quad \longrightarrow \quad i := 5$$
$$j := 2\times 3 + a \qquad\qquad j := 6 + a$$

4) <u>Dead code elimination</u>:-

An variable is said to be live in the program if it's value can be used in the program otherwise it is said to be dead at that point. So, optimization can be performed by eliminating such a dead code.

Ex:-
```
t:=false
if (t)
{
  ==
  ―
}
```

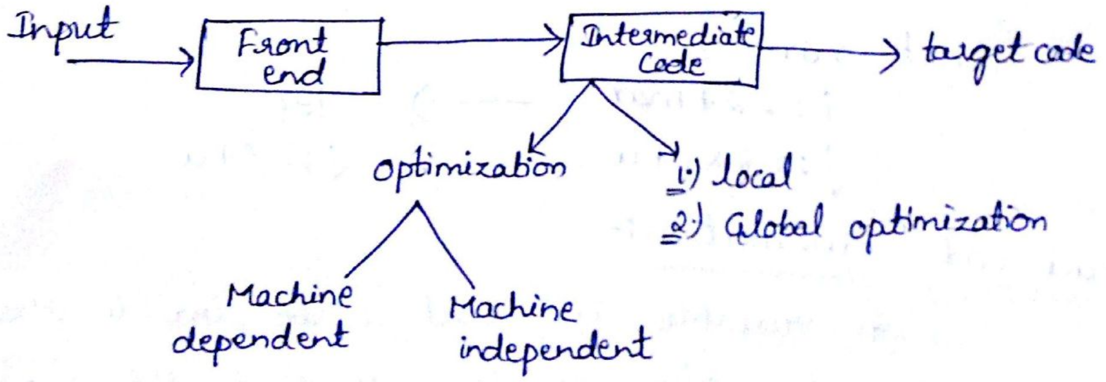
• One advantage of copy propagation is that it often turns copy statement into dead code. Hence after the copy propagation if dead-code elimination is done, then useless statements can be removed.

• Optimization can be done in 2 places in the program.
i) Local optimization
ii) Global optimization.

i) Local optimization :-
Local optimization can be done within the block in the program.

ii) Global optimization :-
Global optimization can be done throughout the program.

Loop Optimization:-



• The code optimization can be significantly done in loops.

• The innerloops where the program spend bulk amount of time, so the running time of the program may be improve, if we decrease the number of instructions in inner loop, even if we increase the amount of instructions outside the loop.

· Loop optimization can be carried out by the following methods:

1.) Code motion.
2) Induction variable
3) Strength Reduction
4.) Loop invariant method
5.) Loop unrolling
6.) Loop fusion.

1. <u>Code motion</u>:-

code motion is a technique, which moves the code outside the loop.

<u>Ex</u>:-

```
while( i<= limit-2)            t= limit-2;
{                       ⟶      while(i<=t)
---                            {
---                            ----
}                              ---
                               }
```

2. <u>Induction variable</u>:-

A variable 'x' is called Induction variable of the loop 'l' if the value of the variable gets change in every time. It is either increment or decrement by some constant

<u>Ex</u>:-

```
i:=i+1
t1:= 4*i.                      t1:= t1+4
t2 := a[t1]             ⟶      t2:= a[t1]
if t2<10 goto B1               if t2 < 10 goto B1
```

Here 'i' and 't1' variables are induction variables.

3. **Strength Reduction:-**

Here the strength of certain operators is higher than the others. For example, the strength of * operator is higher than + operator. In this we can replace * with + operator.

Ex:-

```
for(i=1; i<=100; i++)                  temp = 7;
{                                      for(i=1; i<=100; i++)
count := i*7;          ⟶              {
}                                      count= temp;
                                       temp= temp + 7;
                                       }
```

4. **Loop invariant method:-**

Here the computation of instructions inside the loop is avoided and thereby the computation overhead of the compiler is avoided.

Ex:-

```
while (i<=50)                          t= a/b
{                      ⟶              while (i<=50)
i := i+a/b;                            {
}                                      i= i+t;
                                       }
```

5. **Loop unrolling:-**

Here the number of testing conditions and jumping instructions can be reduced by using writing the code 2 times.

Ex:-

```
while (i<=50)                          while(i<=50)
{                      ⟶              {
a[i]= b[i];                            a[i] = b[i];
i++;                                   i++;
}                                      a[i] = b[i];
                                       i++;
                                       }
```

6. **Loop fusion:-**

Here several loops in program is merged into single loop.

Ex:-

```
for i= 1 to n do                       for i=1 to n*m do
    j = 1 to m do      ⟶                  a[i] := 0;
a[i,j] =0;
```

Basic Block:-

Basic block is a sequence of consequent statements in which flow of controls entered at begin and ending at last without halting the program or possibility of branching is called Basic block.

Ex:- Find out the basic blocks on 3-address code.

```
(1)  | prod := 0. |  --> leader
(2)  |   i := 1   |  --> (Block B1)

(3)  | t1 := 4*i             |  --> leader
(4)  | t2 := a[t1]           |
(5)  | t3 := 4*i             |
(6)  | t4 := b[t3]           |  --> (Block B2)
(7)  | t5 := t2*t4           |
(8)  | t6 := prod+t5         |
(9)  | t7 := i+1             |
(10) |   i := t7             |
(11) | if(i<=10) goto (3)    |
```

M
T

Optimizing basic blocks:-

1) structure preserving transformation (by using DAG).
2) Use algebraic identities.

Terminologies use in basic blocks:-

1. Define and use
2. Live and Dead variables.

Algorithm for partitioning into Basic blocks:-

1. At first, we are determining leaders from the basic block.

2. The first statement or the first instruction is

the intermediate code or basic block is a 'leader'.

3. Any statement that immediately follows 'goto' or unconditional goto is called a leader.

4. The basic block formed by set of instructions from the starting leader to the next leader in the intermediate code.

Optimization of Basic blocks:-

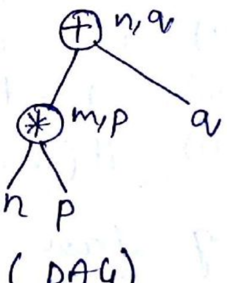
1. **Structure preserving transformations:-**

Structure preserving transformation is a DAG based transformation that means DAG is constructed from the basic block and applying the principal sou[illegible]es of optimization techniques like common sub expression elimination, constant folding, copy propagation, dead code elimination.

Ex:-

```
m:= n*p
n:= m+q
p:= n*p
q:= m+q
```

$\longrightarrow$



(DAG)

• Here the common subexpressions are m:= n*p and n:= m[illegible] that are identified from the DAG and eliminate the common subexpressions.

2. **Use of Algebraic Identities:-**

i.) The algebraic transformations can be obtained using strength reduction technique.

Ex:- Instead of 2*a we can use a+a and instead of a/2 we can use 0.5*a.

ii.) Now the constant folding technique can be applied to achieve the algebraic transformations.

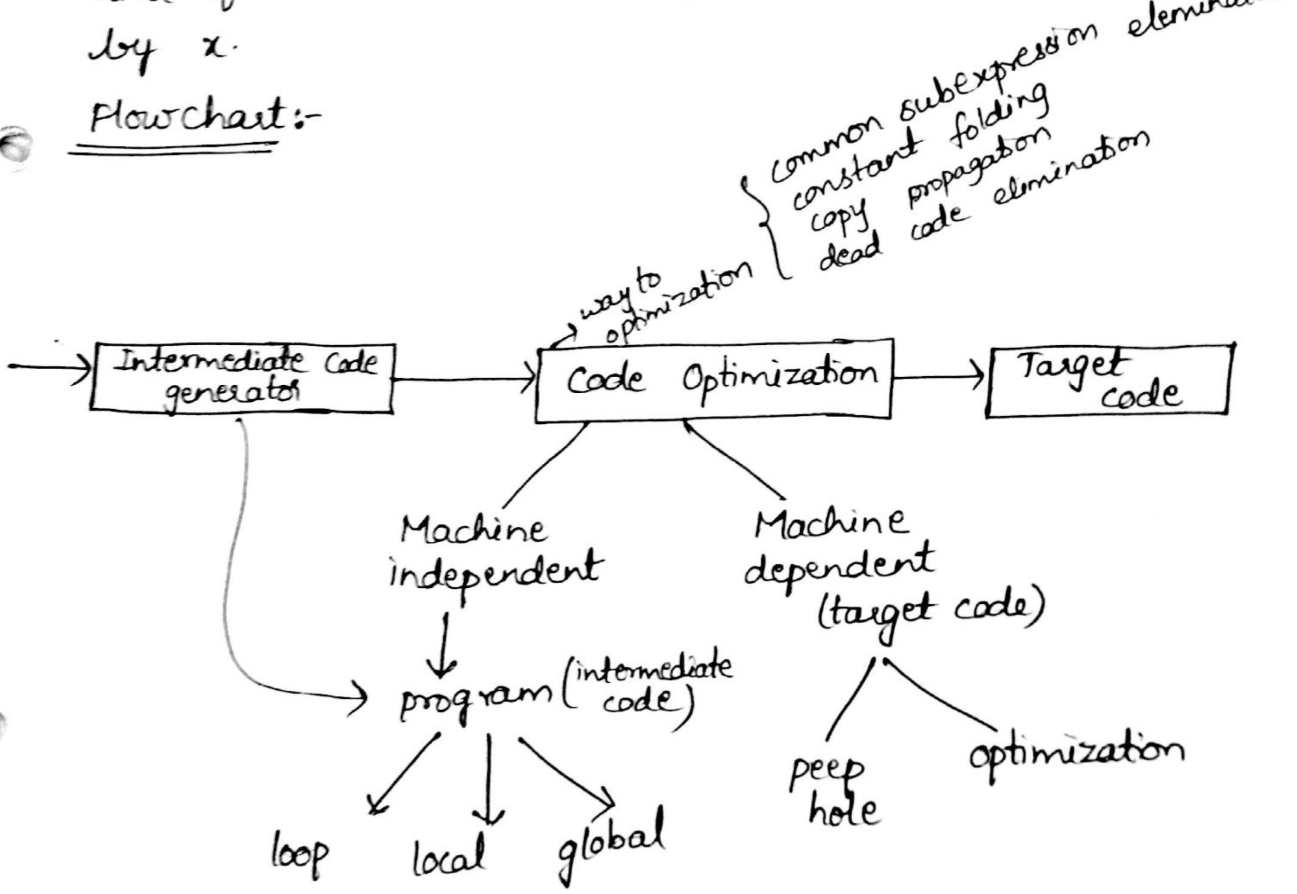
Ex:- Instead of using $x := 2 * 3.14$ we can use $x := 6.28$

iii.) The use of common subexpression elimination, we use The use of associativity and commutativity is to apply algebraic transformations on basic blocks.

Ex:- $x := y * z$
$t := z * r * y$

• Here commutative law can be applied as $y*z = z*y$ and from the second expression we can replace $z*y$ by x.

Flowchart:-



Peephole Optimization:-

Here optimization can be done on target progra which is generated by the compiler.

Characteristics of ~~peeple~~ peephole optimization:-

1. Redundant instruction elimination
2. Flow of control optimization
3. Algebraic simplification

4. Reduction in strength
5. Use of machine Idioms.

- Peephole is nothing but it is a simple window, is moved over the target program and applied the optimization.

1. <u>Redundant instruction elimination</u>:-
Here redundant instructions are the redundant loads and redundant stores can be eliminated in this type of transformation.

Ex:-
```
MOV  R0, X
MOV  X, R0
```

Here MOV X, R0 is a redundant instruction because the value of x is already in register R0.

2. <u>Flow of control optimization</u>:-
Here unnecessary jumps can be eliminated using peephole optimization unnecessary jumps on jumps can be eliminated.

Ex:-
```
goto test                      goto done
                   →
test: goto done                done:

done :
```

3. <u>Algebraic Simplification</u>:-
Peephole optimization is an effective technique for algebraic simplifications. The statements such as $x=x+0$ and $x=x*1$ can be eliminated in target program by using peephole optimization.

4. <u>Reduction in Strength</u>:-
Here certain operators or certain machine instructors

are cheaper than the other in order to improve efficiency of target program, we can replace these instructions by cheaper instructions.

Ex:- x^2 is cheaper than $x*x$, and addition, subtraction instructions are cheaper than multiplication and division. So, we can effectively use equivalent addition and subtraction for multiplication and division.

5) Use of Machine Idioms:-

Here target instructions have equivalent machine instructions for performing some of the operations, hence we can replace these target instructions by equivalent machine instructions inorder to improve efficiency of target program.

Ex:- Some machines having auto increment and auto decrement instructions or addressing modes are used to perform addition and subtraction operations. ~~Eg~~- prgm counte

Flow Graph:-

- Flow graph is a directed graph in which flow of contr information is added to the basic blocks, the nodes to the flowgraph are represented by basic block.
- So, the block whose leader is first statement is called Initial block. There is a directed edge from block B_1 to block B_2, if B_2 immediately follows B_1 in sequence, we can say that B_1 is predecessor of block B_2.

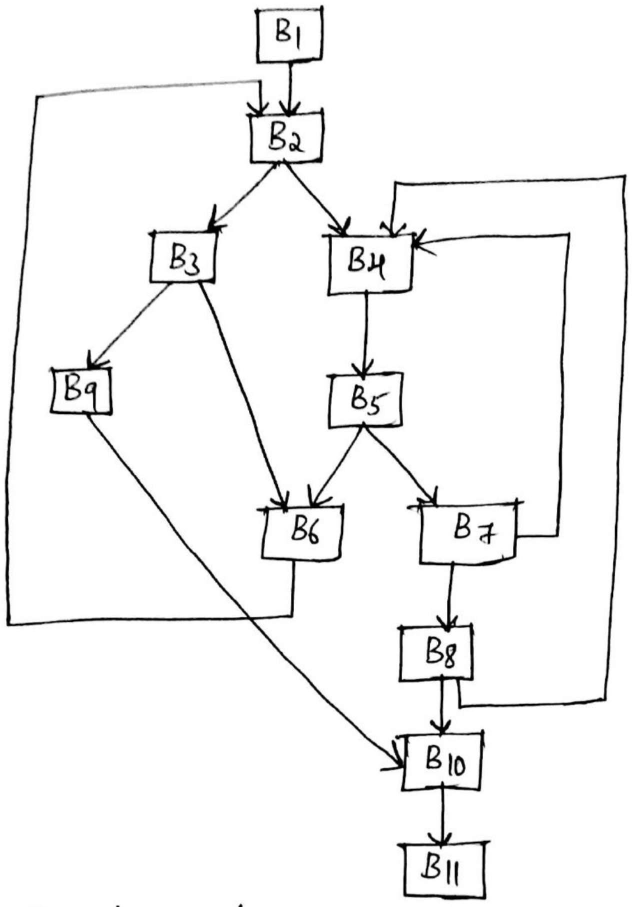
~~Loops in the Flow Graph:-~~

Ex:- Consider the following flow graph and find

a. Denominators for each basic block.

b. Detect all loops in the graph.

For each loop find the back edge and header block information.



Sol:- a.)

B1:	dominates all block
B2:	itself
B3:	B9
B4:	B4, B7, B8
B5:	B7, B8
B6:	itself
B7:	B8
B8:	itself
B9:	itself
B10:	B11
B11:	

Backedge B6 – B2
header : B2
Backedge B7 – B4
header : B4
Backedge B8 - B4
header : B4

Loops in the Flow Graph:-

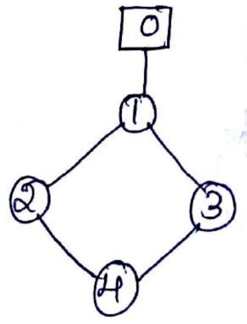
Loop is a collection of nodes in the flow graph.

1. All such nodes are strongly connected that means always there is a path from any node to any another node within that loop.
2. The collection of nodes have unique entry, that means there is only one path from a node outside the loop to a node inside the loop.

Basic Terminologies:-

1. Denominators:- In a flow graph a node 'D' denominates 'n' if every path to node 'n' from initial node goes to 'D' only. This can be denoted as 'D dom n'. Every initial node denominates all the nodes in the flow graph.

Ex:-

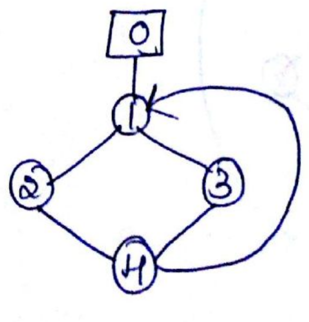


Here block '0' dominates all nodes in flow graph.

2. Natural loops:-

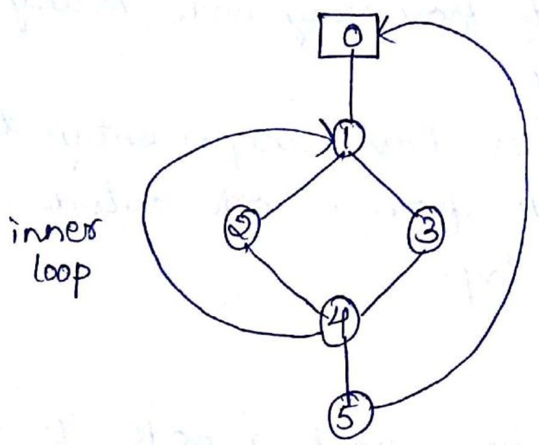
Loop in a flow graph can be denoted by n→d such that 'd' dominates 'n'. These edges are called back edges and for a loop there can be more than one back edges. If there is P→Q then 'Q' is head and 'P' is tail and head dominates tail.

Ex:-



3) Inner Loops:-
The inner loop is a loop that contains no another loop.

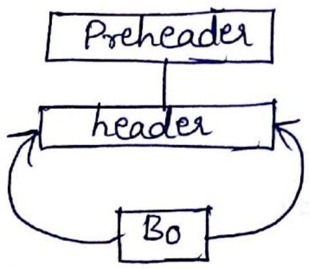
Ex:-



1-4 (inner loop)
4-1 (backedge)

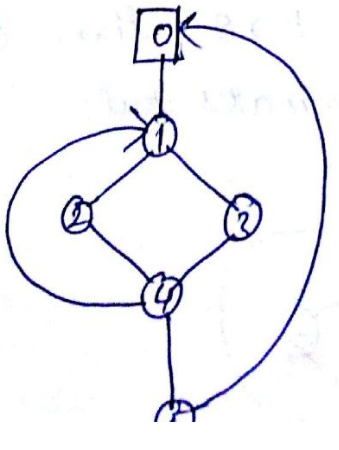
4. Preheaders:-
Preheader is a new block created such that the successor of this block is header and the all the computations that can be made before the header block.

Ex:-



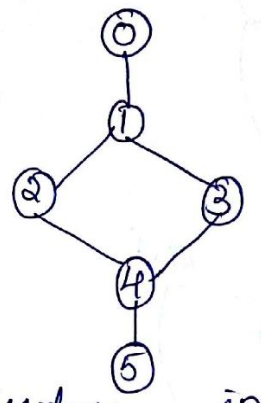
5. Reduceable Flow graph:-
There are 2 types of flow graphs, forward edges and backward edges. These edges have following properties:
i) Forward edges from a acyclic graph.
ii) Backward edges are edges whose head dominates their tail

Ex:-



Scanned by CamScanner

Backward edges are 5→0 and 4→1 arde reduce or eliminate backward edges from the flow graph. Now the flow graph can be represented as



• The program structure, in which there is exclusive use of if-else statement, while loop or goto statements generates flowgraph which is always reduceable.

Local optimization :-

The optimization done using DAG.

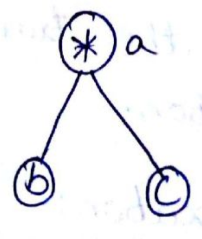
Ex:-

```
a:=b*c
d:=b
e:=d*c
b:= e
f := b+c
g := f+d
```

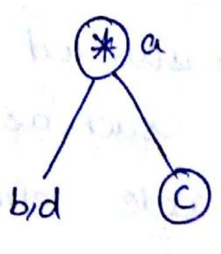
Block B_1

Sol:- The DAG for first instruction in block is as follows

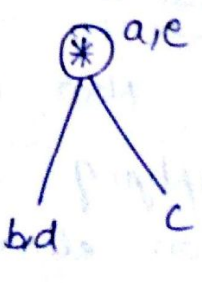
(1)



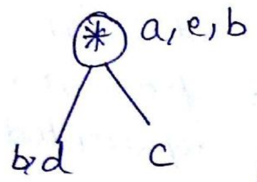
(2)



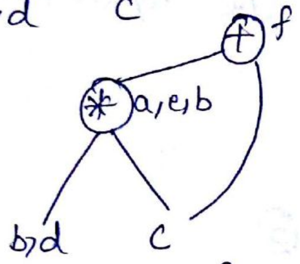
(3)



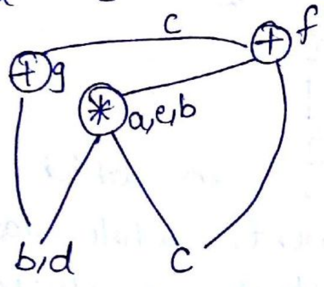
(4)



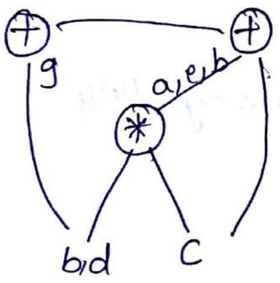
(5)



(6)



(7)



Local optimization :-

The local optimization is done at a restricted scope. Local optimization is a kind of optimization that can be done on some sequence of statements involving or available in basic block. Now DAG is constructed fo basic block and applying the transformation techniques for doing local optimization.

DAG based local optimization :-

Now DAG can be constructed for each block and applying transformations such as common sub expression elimination, dead code elimination, copy propagation, constant folding; FOR performing the

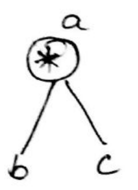
• local optimization, now optimization can be generated by using DAG.

i.) Common subexpression e:=d*c which is actually b*c is eliminated since d:=b.

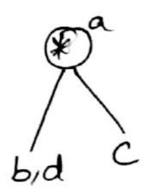
ii.) Deadcode statements like b:=e is eliminated.

Ex:-

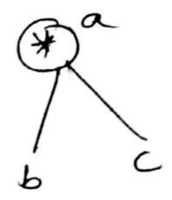
(1)



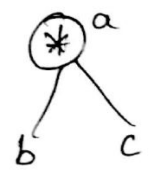
(2)



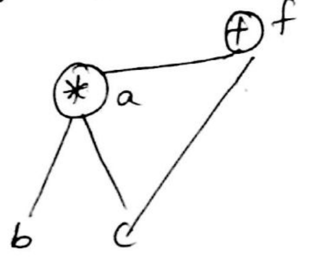
(3)



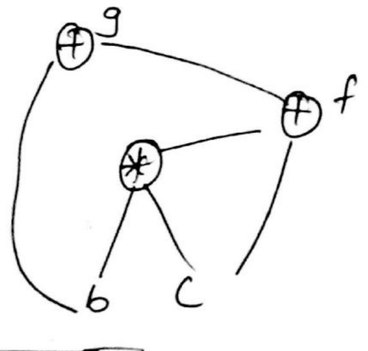
(4)



(5)



(6)



a:= b*c
d:=b
f:=a+c
g: f+d

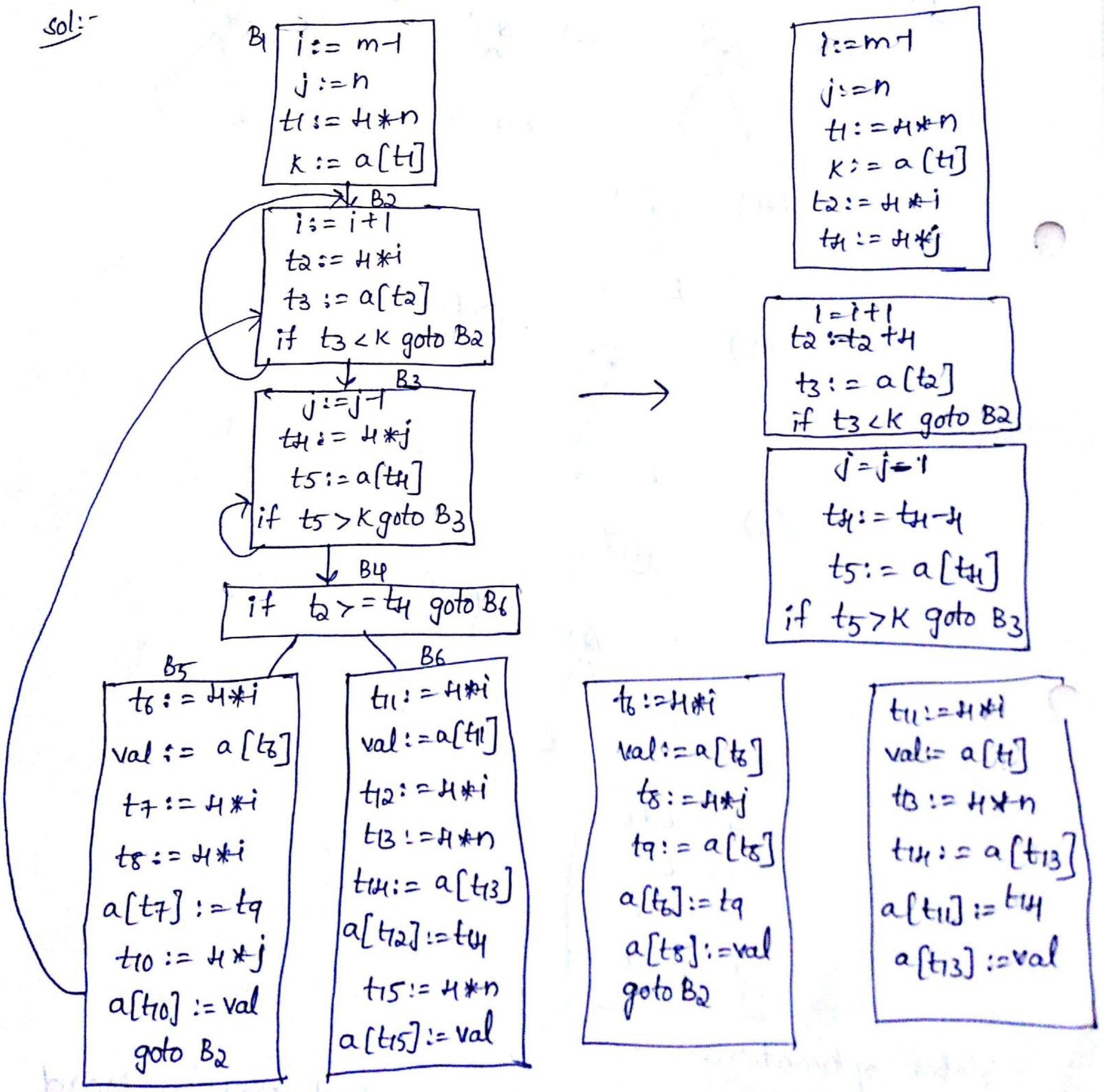
Global optimation:-

Global optimization is applied over a broad scope such as function (or) procedure (or) flowgraph

• For a global optimization, program is represented in the form of program flow graph. So, the program flow graph is a graphical representation in which each node is a block and edges represents flow of control from one block to another block.

Ex:- Compare the local optimization with global optimization. Give some example.

Sol:-



```
B1
i := m-1
j := n
t1 := 4*n
k := a[t1]

B2
i := i+1
t2 := 4*i
t3 := a[t2]
if t3<k goto B2

B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5>k goto B3

B4
if t2>=t4 goto B6

B5
t6 := 4*i
val := a[t6]
t7 := 4*i
t8 := 4*i
a[t7] := t9
t10 := 4*j
a[t10] := val
goto B2

B6
t11 := 4*i
val := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := val
```

```
i := m-1
j := n
t1 := 4*n
k := a[t1]
t2 := 4*i
t4 := 4*j

i = i+1
t2 := t2 +4
t3 := a[t2]
if t3<k goto B2

j = j-1
t4 := t4-4
t5 := a[t4]
if t5>k goto B3

t6 := 4*i
val := a[t6]
t8 := 4*j
t9 := a[t8]
a[t6] := t9
a[t8] := val
goto B2

t11 := 4*i
val := a[t1]
t13 := 4*n
t14 := a[t13]
a[t11] := t14
a[t13] := val
```

B5

```
val:= a[t2]
t9 := a[t4]
a[t2] := t9
a[t4] := val
```

B6

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

• Two types of global optimization analysis:-

1. Control flow analysis
2. Data flow analysis

1. Control Flow Analysis:-

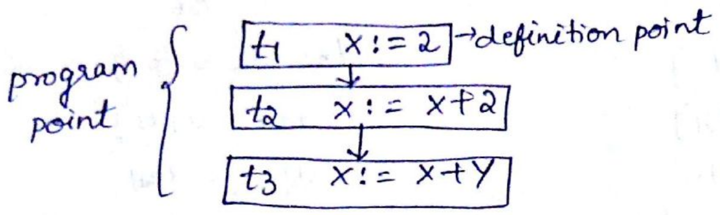
The control flow analysis determines the information regarding arrangement of graph nodes, presence of loops and nesting of loop and nodes visited before execution of specific node. So, thus in control flow analysis the analysis made on flow of control by carefully examining the program in flow graph.

2. Data flow analysis:-

Data flow analysis is a analysis made of the data flow. So, that is the data flow analysis, determir the information regarding definition and use of the dat in program. So, the data flow analysis is basically a process in which the values are computed using d flow properties. Here the properties are

i. Available expression
ii. Reaching Definition
iii. Live and Dead variables
iv. Busy expression.

• Some of the common terminologies in the progra are definition point, reference point and evaluation point.

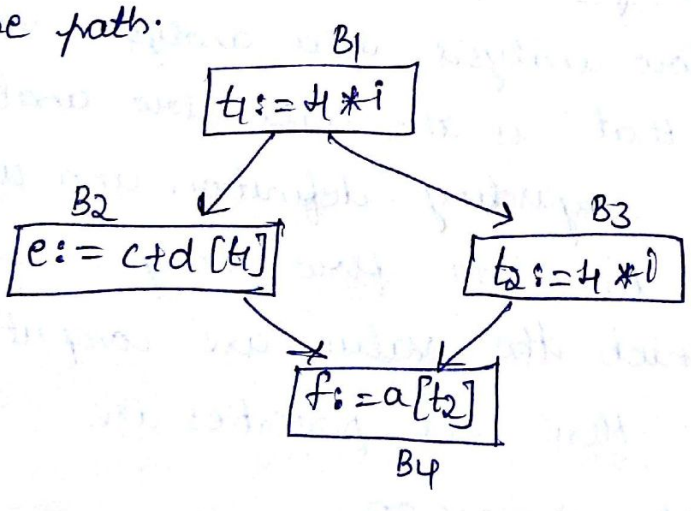


- A program point containing definition is called definition point
- A program point at which reference to a data item is called reference point.
- A program point at which some evaluating expression is given is called as evaluation point.

1.) Available Expression:-

A expression x+Y is available at program point 'w' if and only if along all the paths reaching to 'w'.

- The expression x+Y is said to be available at its evaluation point.
- The expression x+Y is said to be available if no definitio of any operand of expression follows its last evaluation along the path.



- Here expression 4*i is available expression for block B2, B3, B4, because this expression is not been changed by any of block before appearing in block B4

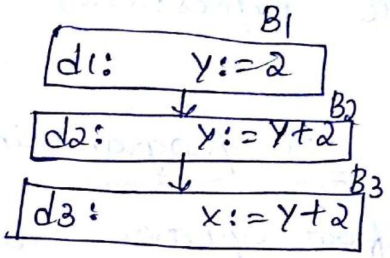
Advantage:-

- Easy to eliminate common sub expressions.

ii.) <u>Reaching</u> <u>Definitions</u>:-

The definition 'd' reaches at point 'p' if there is a path from ~~not~~ D to P, along which 'D' is not kill.

- A definition 'D' of variable 'x' is killed when there is a redefinition of 'x'.
- So the definition d_1 said to be 'reaching definition' for block B_2 but definition d_1 is said to be not reaching definition for block B_3 because the value 'y' variable is killed in block B2.



<u>Advantage</u>:-

- Used in constant and variable propagation.

iii.) <u>Live</u> <u>Variables</u>:-

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

<u>Advantages</u>:-

- Live variables are used to deadcode elimination and register allocation.

iv.) <u>Busy</u> <u>Expression</u>:-

A expression 'E' is said to be a busy expression along some path P_i to P_j iff and only if an evolution of 'E' exists along some path p_i to p_j and no definition of any operand exists before its evolution along the path.

Advantage:-

• Useful in performing code movement optimization.

Data Flow Equations:-

Data flow Equations are the equations that are representing the expressions that are appearing in flow graph. These equations are useful for computing live variable, available expressions, and reaching definitions.

Data Flow Equations for programming constructs:-

The data flow equation written in a form of equation such that,

$$\underset{\substack{\downarrow \\ \text{synthesized} \\ \text{attributes}}}{out[s]} = gen[s] \cup (\underset{\substack{\downarrow \\ \text{inherited} \\ \text{attributes}}}{in[s]} - kill[s])$$

• Here 'S' stands for the statement for which the data flow equation can be written.

• 'out' represents output i.e., the information genera at the end of the statement.

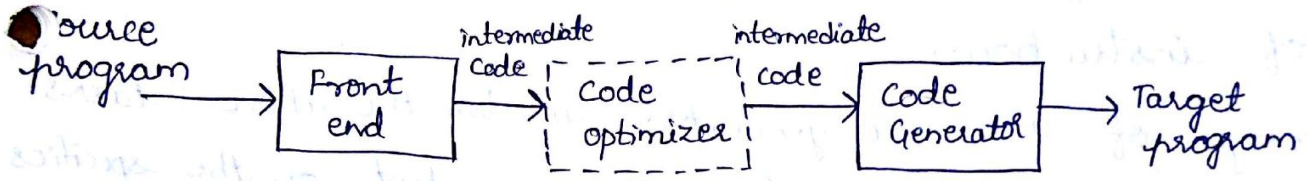
• 'gen' stands for generating the information within the statement and

• 'in' represents gathered information before enterin into the loop,

CODE GENERATION

Code Generation Phase:-

- Final phase in compiler model
- Takes as input intermediate representation (IR) code symbol table information.
- produces output semantically equivalent target program.



- Compiler needs to produce efficient target programs.
- Includes an optimization phase prior to code generation.
- May make multiple passes over the IR before generating the target program.

Target Program Code (or) Object code forms:-

The back end code generator of a compiler may generate different forms of code depending on the requirements.

- Absolute machine code (executable code) - for small programs generate this code.
- Relocatable machine code (object files for linker) - using linker, compiler links subroutines.
- Assembly language (facilitates debugging) - in form of opcodes and addressing codes.
- Byte code forms for interpreters (Ex:- JVM)

Code generation phase:-

Code generation has 3 primary tasks:-

- Instruction selection:-
  Choose appropriate target machine instructions to implement the IR (intermediate representation) statements.
- Register allocation and assignment:-
  Decide what values to keep in which registers.
- Instruction ordering:-
  Decide in what order to schedule the execution of instructions.
- Design of all code generators involve the above 3 tasks. Details of code generation are dependent on the specifics of IR, target language and run-time system.

The target Machine:-

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set.
- Our machine is byte-addressable (word = 4 bytes), has n-general purpose registers $R_0, R_1, ---- R_{n-1}$.
- 2-address instructions of form instruction, opcode (source) destination data fields.

Ex:- opcodes (op)

MOV (move content from destination)

ADD

SUB

The Target Machine: Address modes:-

mode	form	Address	Added cost
Absolute	M	M	1
Register	R	R	0
Indexed	(CR)	C+contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*(CR)	contents(C+contents(CR))	1
literal	#C	N/A	1

Instruction costs:-

- Machine is a simple processor with fixed instruction costs.
- cost of instruction = 1 + cost(source - mode) + cost(destination - mode).

Ex:-

Instruction	Operation	Cost
MOV R0, R1	store content (R0) into register R1 (1+0+0)	1
MOV R0, M	store content (R0) into memory location M (1+0+1)	2
MOV M, R0	store content(M) into register R0 (1+1+0)	2
MOV 4(R0), M	store content (4+contents(R0)) into M (1+1+1)	3
MOV *4(R0), M	store content(contents(4+ contents(R0))) into M →(1+1+1)	3
MOV #1, R0	store content store 1 into R0 (1+1+0)	2
ADD 4(R0), *12(R1)	store content into (R0) add contents (4+ contents(R0)) to value at location contents(12 + contents(R1)) → (1+1+1)	3

Instruction selection :-

- Instruction selection is important to obtain efficient code.
- Suppose we translate 3-address code.

Ex :-

```
x := y+z
to :  MOV  y, Ro
      ADD  z, Ro
      MOV  Ro, x
```

```
a:=a+1  =>  MOV  a, Ro (1+1+0)    2
            ADD  #1, Ro           3
            MOV  Ro, a (1+0+1)    2
                        cost  =   7
```

Better ⇓ (1+1+) ADD #1,a cost=3

Best ⇓ INC a cost=2

Therefore in above example replace ADD #1, Ro with INC Ro, so that cost can be reduced.

```
a:= a+1  =>  MOV  a, Ro (1+1+0)   2
             INC  Ro              2
             MOV  Ro,a (1+0+1)    2
                         cost =   6
```

Need for Global Machine-specific code optimization :-

- Suppose we translate 3-address code

```
x := y+z
to :  MOV  y, Ro
      ADD  z, Ro
      MOV  Ro, x
```

- Inises of storage in processor :-

Then we translate,

```
a:= b+c                 MOV  a, R0
d:=a+e       ⟶          ADD  b, R0
                        MOV  R0, a  ] redundant
                        MOV  a, R0
                        ADD  e, R0
                        MOV  R0, d
```

Register Allocation and Assignment:-

- Efficient utilization of the limited set of registers is important to generate good code.
- Registers are assigned by
  - Register allocation to select the set of variables that will reside in registers at a point in the code.
  - Registers assignment to pick the specific register that a variable will reside in.
- Finding an optimal register assignment in general is NP-complete.

Ex:-

```
t := a*b
t := t+a
t := t/d
   ⇩ {R1=t}
MOV   a, R1
MUL   b, R1
ADD   a, R1
DIV   d, R1
MOV   R1, t
```

Here we use only one register and only 5 instructions.
So, it is efficient.

Ex 2:-

$t := a * b$
$t := t + a$
$t := t / d$

$\Downarrow$

$\{R_0 = a, R_1 = t\}$

```
MOV  a, R0
MOV  R0, R1
MUL  b, R1
ADD  R0, R1
DIV  d, R1
MOV  R1, t
```

Here, we use 2 registers, and 6 instructions.
So, it is not efficient.

- Number of registers usage should be less to have an efficient output (target program).

Choice of Evaluation order:-

- When instructions are independent, their evaluation order can be changed.

Ex:-

$a+b-(c+d)*e \Rightarrow$

$t_1 := a+b$
$t_2 := c+d$
$t_3 := e*t_2$
$t_4 := t_1-t_3$

$\Rightarrow$

```
MOV  a, R0
ADD  b, R0
MOV  R0, t1
MOV  c, R1
ADD  d, R1
MOV  e, R0
MUL  R1, R0
MOV  R1, R0
SUB  R0, R1
MOV  R1, t4
```

reorder $\Downarrow$

$t_2 := c+d$
$t_3 := e*t_2$
$t_1 := a+b$
$t_4 := t_1-t_3$

$\Longrightarrow$

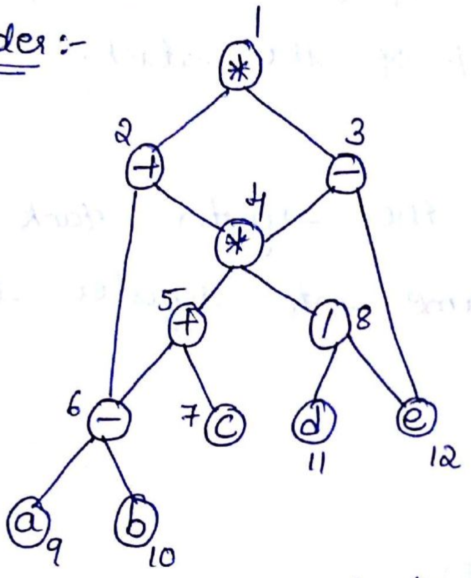
```
MOV  c, R0
ADD  d, R0
MOV  e, R1
MUL  R0, R1
MOV  a, R0
ADD  b, R0
SUB  R1, R0
MOV  R0, t4
```

Two classes of storage in processor :-

1. Registers :-
   - Fast access, but only few of them
   - Address space, not visible to programmer
   - Doesnot support pointer access.

2. Memory :-
   - slow access, but large
   - supports pointers.

Heuristic order :-



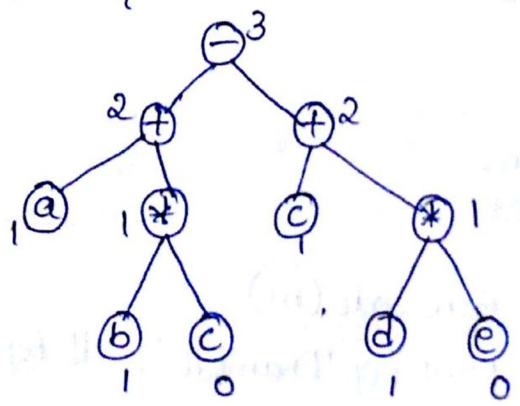
1,2,3,4,5,6,8 (except leaf nodes)

- By using this order of evaluation, efficiency of code generation is done maximum.

Labelling Algorithm :- (bottom up passion)

$$label(n) = \begin{cases} \max(L_1, L_2) & \text{if } L_1 \neq L_2 \\ L_1 + 1 & \text{if } L_1 = L_2 \end{cases}$$

Ex:- consider a DAG

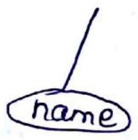


left → 1
right → 0

- Here we use data structure register stack, temporary stack. It contains general purpose registers $R_0, R_1, \ldots R_{n-1}$ used while evaluation of expression, we push values to stack. The top of stack contains R_0 only. It contains temporary variables.
- It contains a function cod_gen(n). Its purpose to generate a target code using information present in top of stack.
- Another function is swap functions. It purpose is to swap registers in top of the stack.

Case :1

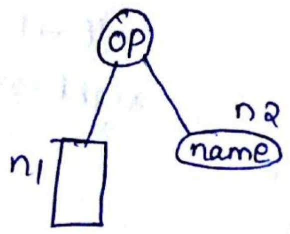
If the top of the register stack contains name, means load the name into register that can be obtained by



print(MOV ‖ name ‖, top (reg.stack))

Case 2:

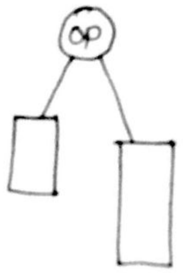
If the nodes right child is leaf n2, then, the target code will be, generate code for n1 and then gen_code(n2).



Gen_code(n1)
print (op ‖ name ‖ ',' ‖ top(regstack))

Case - 3:

If the left child ~~of~~ n ie, n1 requires less number ~~of~~ of registers than n2, then we swap the top two registers on top ~~of~~ stack.



Case - 4:

If the both child requires same number ~~of~~ registers, then we evaluate first right sub-tree and store in temporary stack and then evaluate left sub-tree.