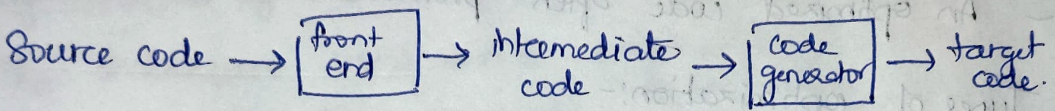
Code optimization:-

- The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code making it consume fewer resources (i.e. CPU, memory) so that faster-running machinecode will result.
- In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.



- A code optimization process must follow the three rules given below:
  - The output code must not, in any way, change the meaning of the program.
  - Optimization should increase the speed of the program and if possible, the program should demand less number of Resources.
  - Optimization should itself be fast and should not delay the overall compiling process.
- Efforts for an optimized code can be made at various levels of compiling the process.
  - At beginning, users can change /rearrange the code or use better algorithms to write the code.
  - After generating intermediate code, the compiler can ~~make~~ ~~use of~~ ~~memory~~ ~~hierarchy~~ modify the intermediate code by address calculations and improving loops.
  - While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Why optimize?

- It involves in reducing the size of the code
- Reduce the speed space consumed and increases the speed of compilation.
- Manually analyzing datasets involves a lot of time. Hence we make use of software like tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer.
- An optimized code often promotes re-usability.

Types of optimization:-

Optimization is broadly classified into two types:-

1. Machine Independent
2. Machine dependent.

Machine Independent optimization:-

- The compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolutely memory locations.
- Attempts to improve the intermediate code to get a better target code as output.

Ex:

```
do
{
item=10;
value= value + item;
}
while (value <100);
```

→ This code involves repeated assignment of the identifier item

⇓

```
item =10;
do
{
   value = value + item;
}
while (value <100);
```

→ not only save the CPU cycles, but can be used on any processor.

Machine - dependent optimization:-

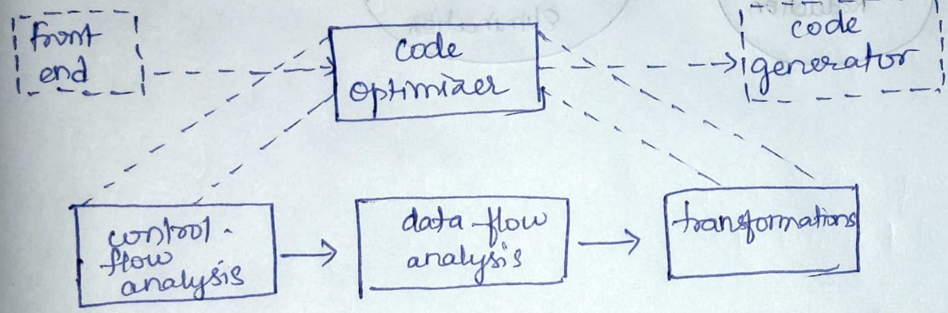
- Machine - dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture.
- It involves CPU registers and may have absolute memory references rather than relative references.
- Machine dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Organization of the code optimizer :-

The techniques used are a combination of

→ Control - Flow analysis.

→ Data - Flow analysis.



Control - Flow Analysis :- Identifiers loops in the flow graph of a program since such loops are usually good candidates for improvement.

Data - Flow Analysis:- Collects information about the way variable are used in a program.

Principle Sources of optimization:-

There are generally two phases of optimization:

1. Global optimization: Transformations are applied to large programsegments that includes functions, procedures and loops.
2. Local optimization: Transformations are applied to small block of statements. The local optimization is done prior to global optimization.

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. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a [t4] = 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. t5 = i - 1
14. t6 = 88 * t5
15. a [t6] = 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:

DAG representation for basic blocks

- A DAG for basic block is a **Directed Acyclic Graph** with the following labels on nodes:
  - The leaves of graph are labeled by unique identifier and that identifier can be variable names or constants.
  - Interior nodes of the graph is labeled by an operator symbol.
  - Nodes are also given a sequence of identifiers for labels to store the computed value.

- It does not contain any cycles in it, hence called **Acyclic**.

Optimization Of Basic Blocks

- DAGs are a type of data structure. It is used to implement transformations on basic blocks.

- A DAG is constructed for optimizing the basic block.

- A DAG is usually constructed using **Three Address Code.**

- DAG provides a good way to determine the common sub-expression.

- It gives a picture representation of how the value computed by the statement is used in subsequent statements.

Algorithm for construction of DAG

Input: A basic block

Output: It contains the following information:

- Each node contains a label. For leaves, the label is an identifier/constant.
- Each node contains a list of attached identifiers to hold the computed values.
- Consider the following formats of three-address:
  - Case (i) x:= y op z
  - Case (ii) x:= op y
  - Case (iii) x:= y

Method:

Step 1:

- If y operand is undefined then create **node(y)**.
- If z operand is undefined then for case(i) create **node(z)**.

Step 2:

- For case(i), create **node(op)** whose right child is node(z) and left child is node(y).
- For case(ii), check whether there is **node(op)** with one child node(y).
- For case(iii), node x will be node(y).

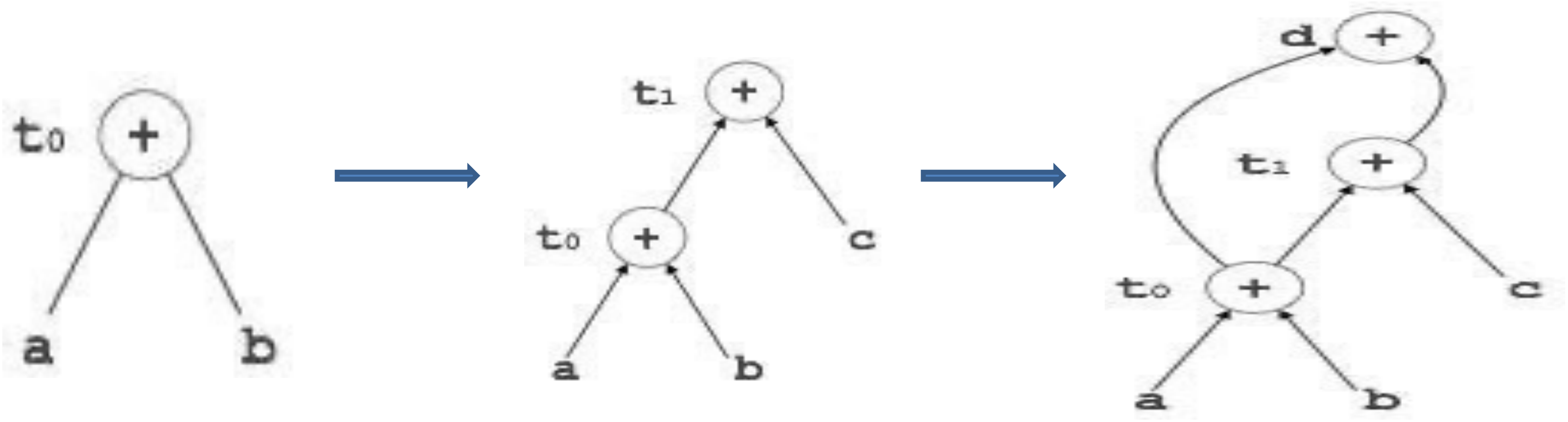
Example-1

Consider the three address code:

$t_0 = a + b$

$t_1 = t_0 + c$

$d = t_0 + t_1$



Example-2

Consider the following block and construct a DAG for it:

(1) a = b* c

(2) d = b

(3) e = d * c

(4) b = e

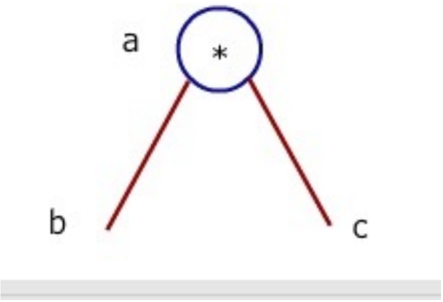
(5) f = b + c

(6) g = f + d

Directed Acyclic Graph for the given block is:

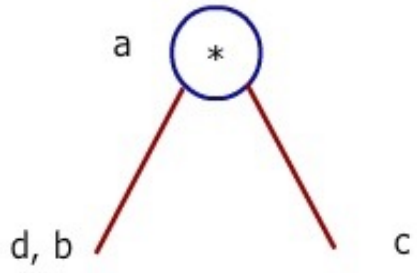
Step 1

- Consider the first statement, i.e., **a = b * c.**
- Create a leaf node with label **b** and **c** as left and right child respectively and parent of it will be *****.
- Append resultant variable **a** to the node *****.



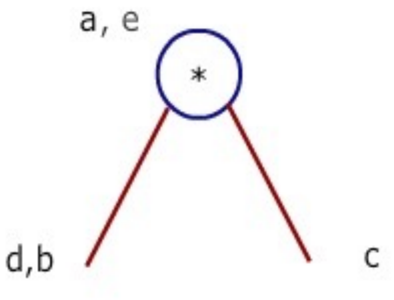
Step 2

- For second statement, i.e., **d = b**, node **b** is already created.
- So, append **d** to this node.



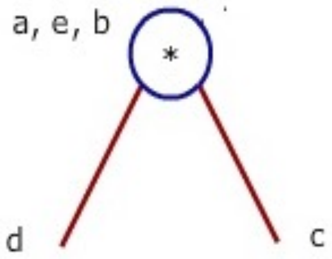
Step 3

- For third statement **e = d * c**, the nodes for **d, c** and ***** are already created.
- Node **e** is not created, so append node **e** to node *****.



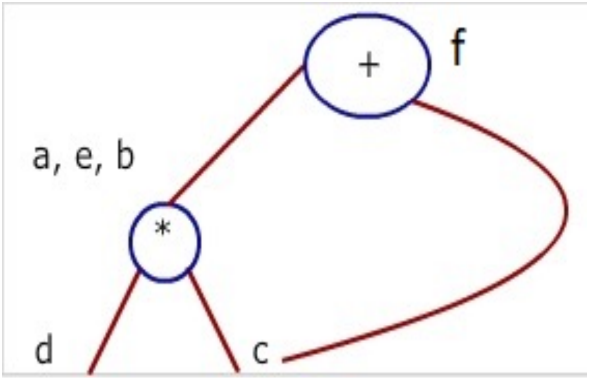
Step 4

- For fourth statement **b = e**, append **b** to node **e**.



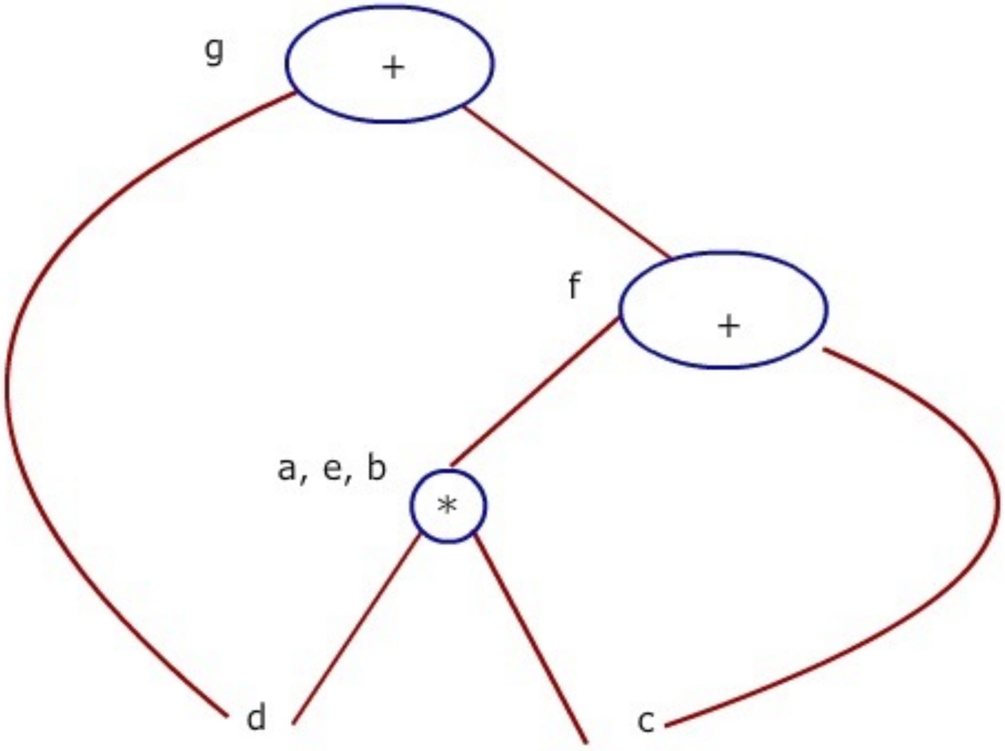
Step 5

- For fifth statement **f = b + c.**
- create a node for operator **+** whose left child **b** and right child **c** and append **f** to newly created node **+.**



Step 6

For last statement **g = f + d**, create a node for operator **+** whose left child **d** and right child **f** and append **g** to newly created node **+**.



- Now, the optimized block can be generated by traversing the DAG.
- The common sub-expression e = d * c which is actually b * c (since d = b) is eliminated.
- The dead code b = e is eliminated.

```
(1) a = b* c
(2) d = b
(3) e = d * c
(4) b = e
(5) f = b + c
(6) g = f + d
```

Optimized code

```
(1) a = b * c
(2) d = b
(3) f = a + c
(4) g = f + d
```