Code Generation and optimization

Basic Block:-

Introduction

- The basic block is a sequence of consecutive statements which are always executed in sequence without halt or possibility of branching.
- The basic blocks does not have any jump statements among them.
- When the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

Examples

→ $a = b + c + d$

Three address code-

$t_1 = b + c$

$t2 = t1 + d$

$a = t2$

✓

→ If A<B then 1 else 0

(1) if (A<B) goto (4)

(2) T1 = 0

(3) goto (5)

(4) T1 = 1

(5)

✗

Rules for partitioning into blocks

After an intermediate code is generated for the given code, we can use the following rules to partition into Basic Blocks -

Rule-1: Determine the leaders -

a) The first statement is a leader.

b) Any target statement of conditional or unconditional goto is a leader.

c) Any statement that immediately follow a goto is a leader.

Rule-2: The Basic Block is formed starting at the leader statement and ending just before the next leader statement appearing.

Problem:- Consider the following three address code statements -

(1) PROD = 0
(2) I = 1
(3) T2 = addr(A) - 4
(4) T4 = addr(B) - 4
(5) T1 = 4 * I
(6) T3 = T2[T1]
(7) T5 = T4[T1]
(8) T6 = T3 * T5
(9) PROD = PROD + T6
(10) I = I + 1
(11) IF I <= 20 GOTO (5)

Compute the Basic Blocks.

Solution:

- Because first statement is a leader, so-

  PROD=0 is a leader

- Because the target statement of conditional or unconditional goto is a leader, so-

  T1 = 4*I is also a leader

So, the given code can be partitioned into 2 blocks as-

```
PROD = 0
I = 1
T2 = addr(A) - 4
T4 = addr(B) - 4
```
B_1

```
T1 = 4*I
T3 = T2[T1]
T5 = T4[T1]
T6 = T3*T5
PROD = PROD + T6
I = I+1
IF I<=20 GOTO B2
```
B_2

Flow Graph

Definition:-

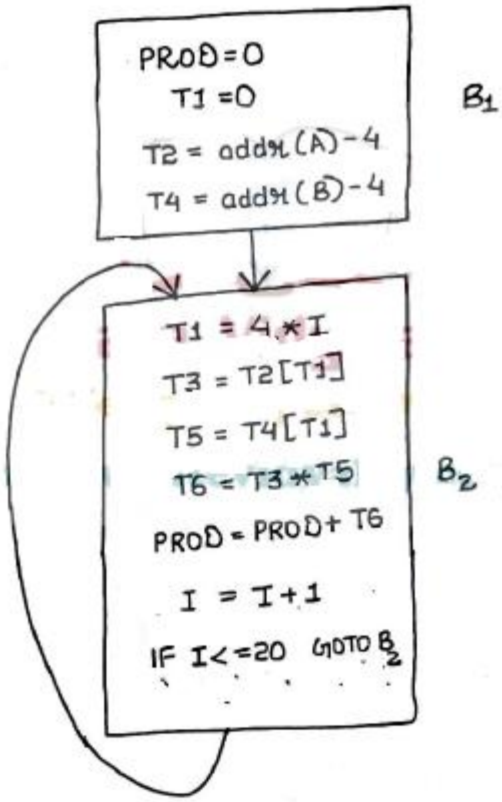
A flow graph is a directed graph in which the flow control information is added to the basic blocks.

Rules:-

- The basic blocks are the nodes to the flow graph.
- The block whose leader is the 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 the given sequence, we can say that B1 is a predecessor of B_2.

Problem: Draw the flow graph for the three address code given in the last question.

Soln:



Directed Acyclic Group:-

Definitions:

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.

Use of DAG for optimizing basic blocks :-

- DAG is a useful data structure for implementing transformations on basic blocks.
- A basic block can be optimized by the construction of DAG.
- A DAG can be constructed for a block and certain transformations such as common subexpression elimination and 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 statement.

Properties of a DAG

① The reachability relation in a DAG forms a partial order and any finite partial order may be represented by a DAG using reachability.

② The transitive reduction and transitive closure are both uniquely defined for DAGs.

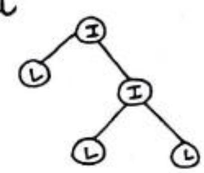
③ Every DAG has a topological ordering

Applications of a DAG

The DAG is used in-

1. determining the common subexpression (expressions computed more than once).
2. determining which names are used in the block and computed outside the block.
3. determining which statements of the block could have their computed value outside the block.
4. Simplifying the list of quadruples by eliminating the common subexpressions and not performing the assignment of the form $x:=y$ until and unless it is a must.

Rules for the construction of DAG :-



Rule-1:- In a DAG,

→ Leaf nodes represent identifiers, names or constants.

→ Interior nodes represent 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 us to detect common sub-expressions and eliminate the re-computation of the same.

Rule-3:- The assignment of the form $x:=y$ must not be performed until and unless it is a must.

Problems

Problem-1: Construct DAG for the given expression:-

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

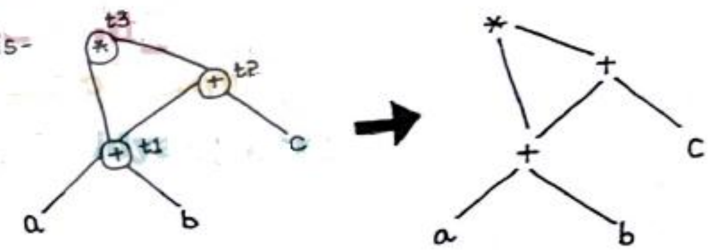
Soln:- Three address code for the given expression is-

$t1 = a + b$

$t2 = t1 + c$

$t3 = t1 * t2$

The DAG is-



Explanation:-

From the constructed DAG, we observe that the common subexpression (a+b) is translated into a single node in the DAG. The computation is carried out only once and stored in the identifier t1 and reused later.

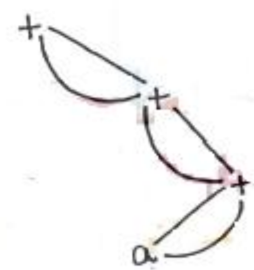
This illustrates how the DAG construction scheme identifies the common sub-expression & helps in eliminating its re-computation later.

Problem-02:- Construct DAG for the given expression -

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

Soln:-

DAG for the given expression is-



Problem-03:- Construct the DAG for the following block -

$a = b * c$

$d = b$

$e = d * c$

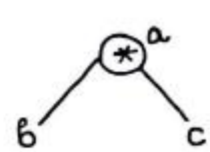
$b = e$

$f = b + c$

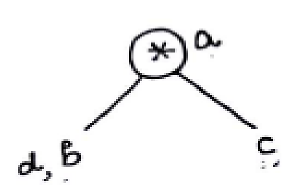
$g = f + d$

Soln:-

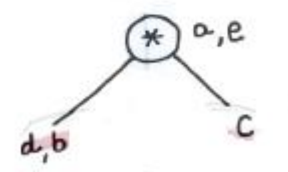
Step-1:



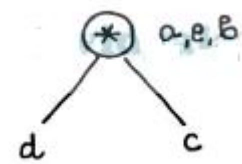
Step-2:-



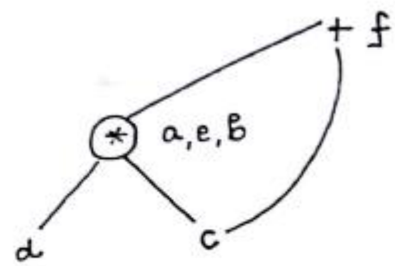
Step-3:-



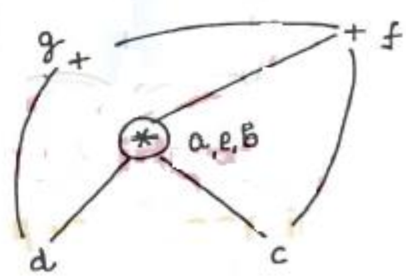
Step-4:-



Step-5:-



Step-6:-



Problem-4 :- optimitize the Block given in problem-3.

Soln:-

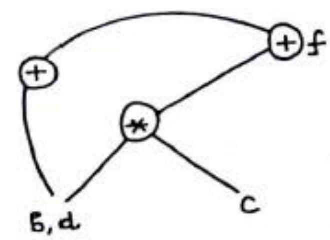
Step-1:- First construct the DAG for the given Block.

Step-2:- Now, the optimized code can be generated by traversing the DAG.

1. The common subexpression $e = d * c$ which is actually $b * c$ ($\because d = b$) is eliminated.
2. The dead code $b = e$ is eliminated.

The optimized basic Block is-

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

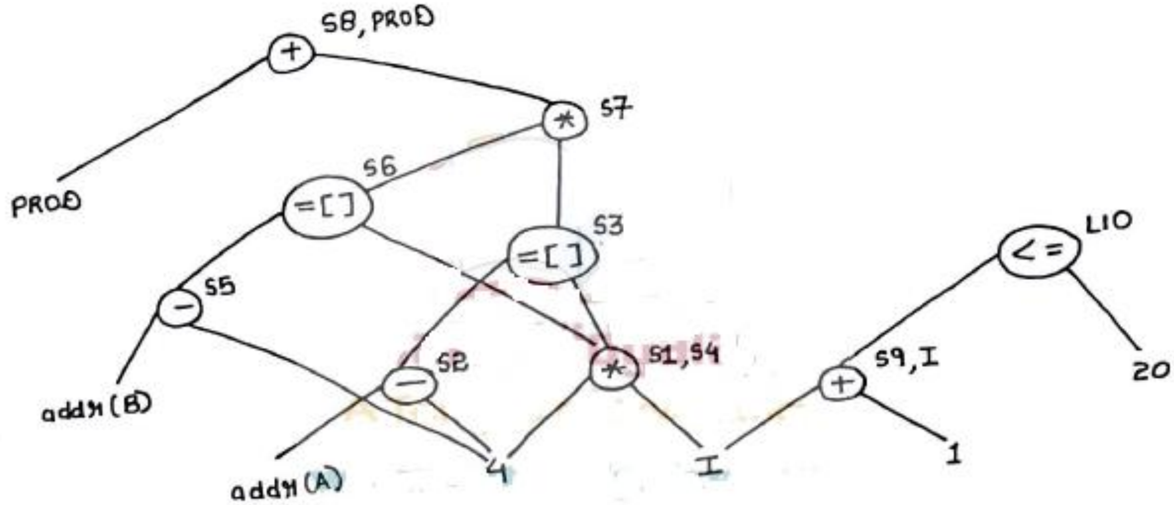


Problem-5 :- Consider the following Basic Block. Draw the DAG representation of the Block and identify local common sub-expressions. Eliminate the common expressions and rewrite the basic Block.

```
L10 :  S1 = 4 * I
       S2 = addr(A) - 4
       S3 = S2[S1]
       S4 = 4 * I
       S5 = addr(B) - 4
       S6 = S5[S4]
       S7 = S3 * S6
       S8 = PROD + S7
       PROD = S8
       S9 = I + 1
       I = S9
       If I <= 20 goto L10
```

Solution:-

DAG representation for the Block is :-



In this code fragment,

- $4 * I$ is a common subexpression. Hence, we can eliminate S4 because $S1 = S4$.

S8 = PROD + S7
PROD = S8

  can be optimized as

PROD = PROD + S7

S9 = I + 1
I = S9

  can be optimized as

I = I + 1

After eliminating S4, S8 and S9, we get -

```
L10 :
        S1 = 4*I
        S2 = addr(A)-4
        S3 = S2[S1]
        S5 = addr (B)-4
        S6 = S5[S1]
        S7 = S3 * S6
        PROD = PROD+ S7
        I = I+1
        IF  I <=20  GOTO  L10
```

Code Optimization:-

Definition

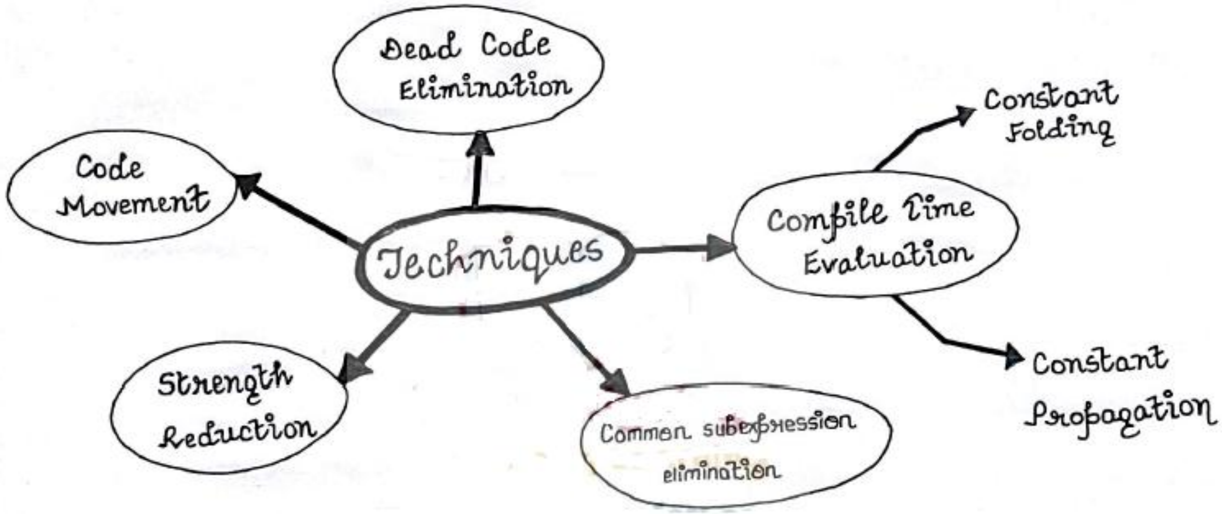
Code Optimization is a technique which tries to improve the code by eliminating unnecessary code lines and arranging the statements in such a sequence that speed up the program execution without wasting the resources.

Advantages:-

→ Executes faster

→ Efficient memory usage

→ Yields better performance



① Compile Time Evaluation

i) Constant Folding:-

It refers to a technique of evaluating the expressions whose operands are known to be constant at compile time itself.

Example:- length = (22/7) * d

ii) Constant Propagation:-

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

Example:-

pi = 3.14
r = 5
Area = pi * r * r

Here, the value of pi is replaced by 3.14 and r by 5, then computation of 3.14 * r * r is done during compilation.

(ii) Common sub-expression elimination :-

The common sub-expression is an expression appearing repeatedly in the code which is computed previously. This technique replaces redundant expression each time it is encountered.

Example:-

```
T1 = 4*i
T2 = a[T1]
T3 = 4*j
T4 = 4*i
T5 = n
T6 = b[T4] + T5
```

Before Optimization

```
T1 = 4*i
T2 = a[T1]
T3 = 4*j
T5 = n
T6 = b[T4] + T5
```

After Optimization

(iii) Code Movement :-

It is a technique of moving a block of code outside a loop if it won't have any difference if it is executed outside or inside the loop.

Example:-

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

Before Optimization

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

After Optimization

(IV) 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.

Example:-

Before Optimization	After Optimization

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

(V) Strength Reduction:-

It is the replacement of expressions that are expensive with cheaper and simple ones.

Example:-

Before Optimization	After Optimization
B = A * 2	B = A + A