# Code Optimization

Prepared By:

Prabhat Shukla

# Introduction

- Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

- Optimizations are classified into two categories:

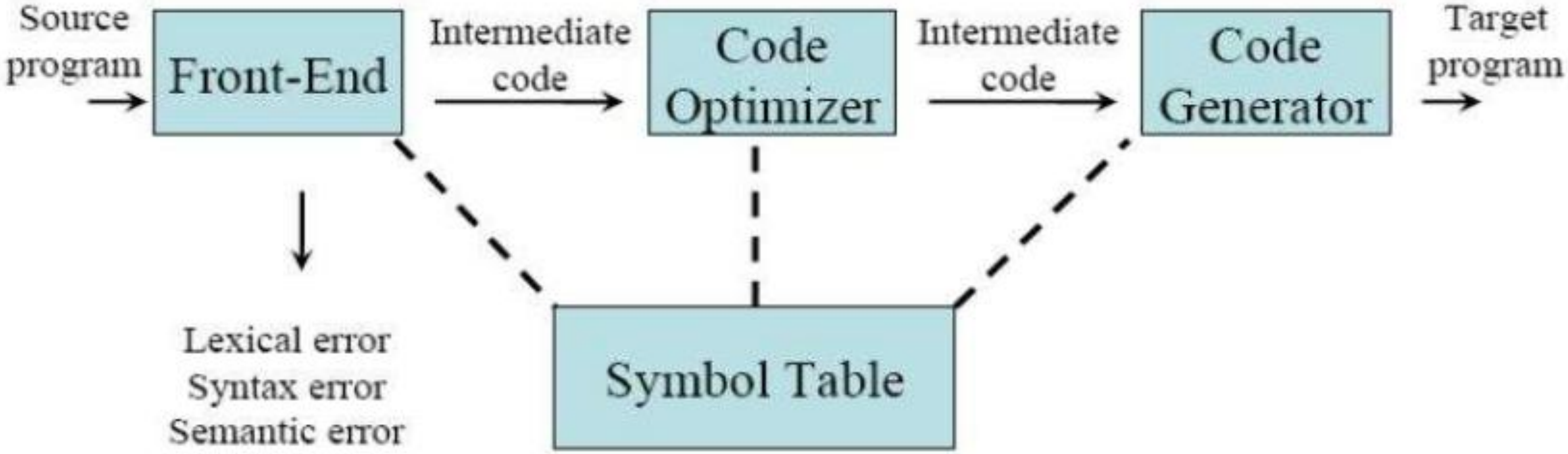
  a) Machine independent optimizations

  - improve target code without taking properties of target machine into consideration.

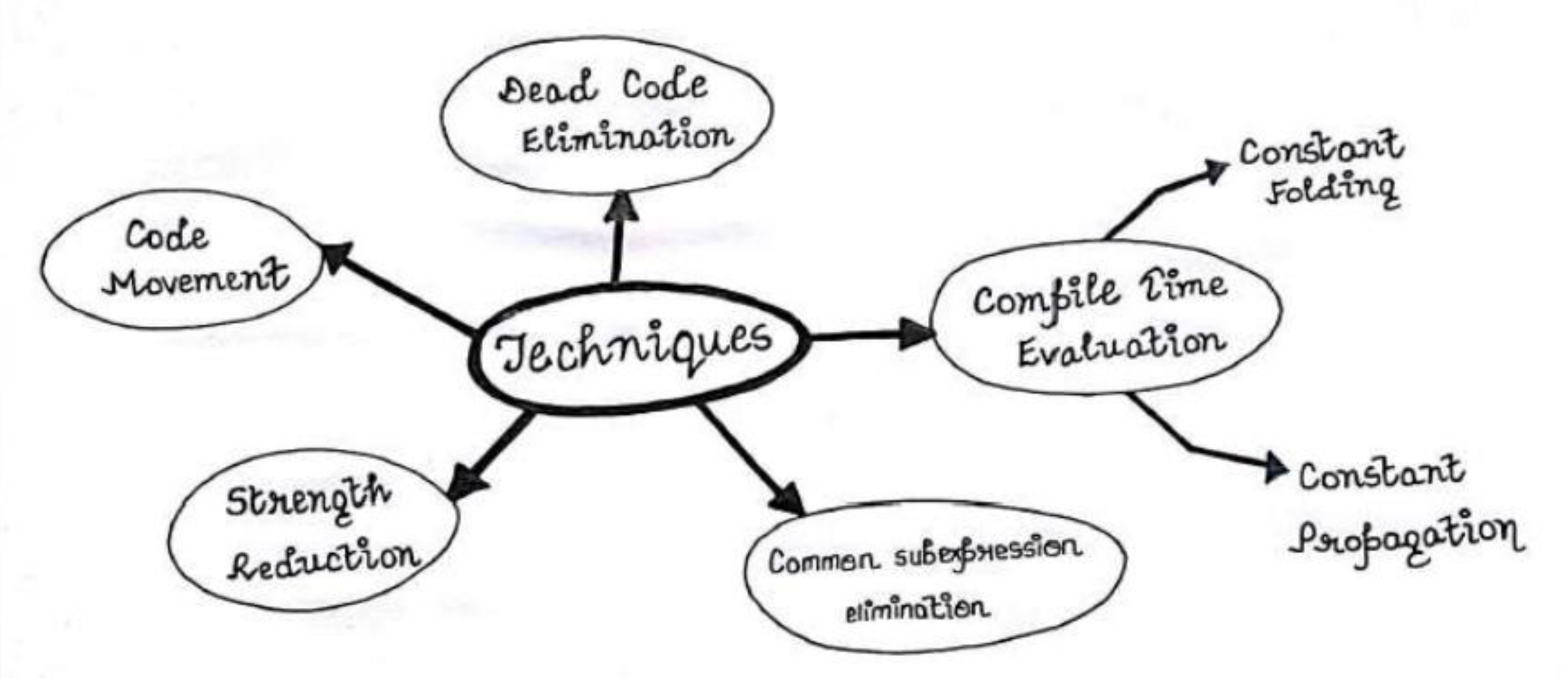
  b) Machine dependent optimization

# Criteria for Optimization

- The transformation must preserve the meaning of programs.
- A transformation must, on the average, speed up programs by a measurable amount.
- The transformation must be worth the effort.



# Techniques for Optimization



# 1) Compile time evaluation

**A) Constant folding-**

- As the name suggests, this technique involves folding the constants by evaluating the expressions that involves the operands having constant values at the compile time.

**Example-**

Circumference of circle = (22/7) x Diameter

- Replace (22/7) with 3.14 during compile time and save the execution time.

# 1) Compile time evaluation

## B) Constant Propagation-

- In this technique, if some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program wherever it has been used during compilation, provided that its value does not get alter in between.

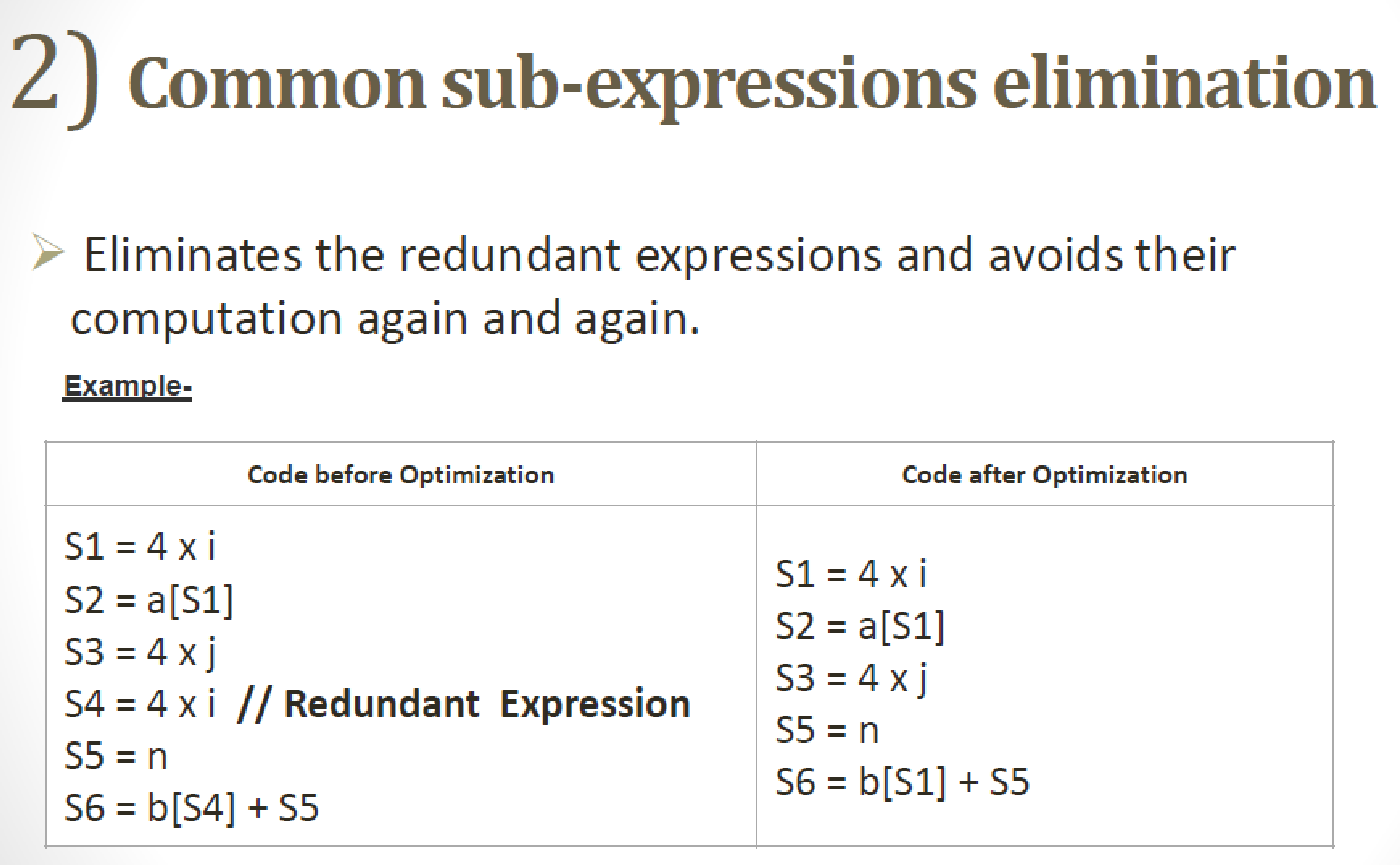
**Example-**

pi = 3.14

radius = 10

Area of circle = pi x radius x radius

- Substitutes constants value during compile time and saves the execution time.



# 2) Common sub-expressions elimination

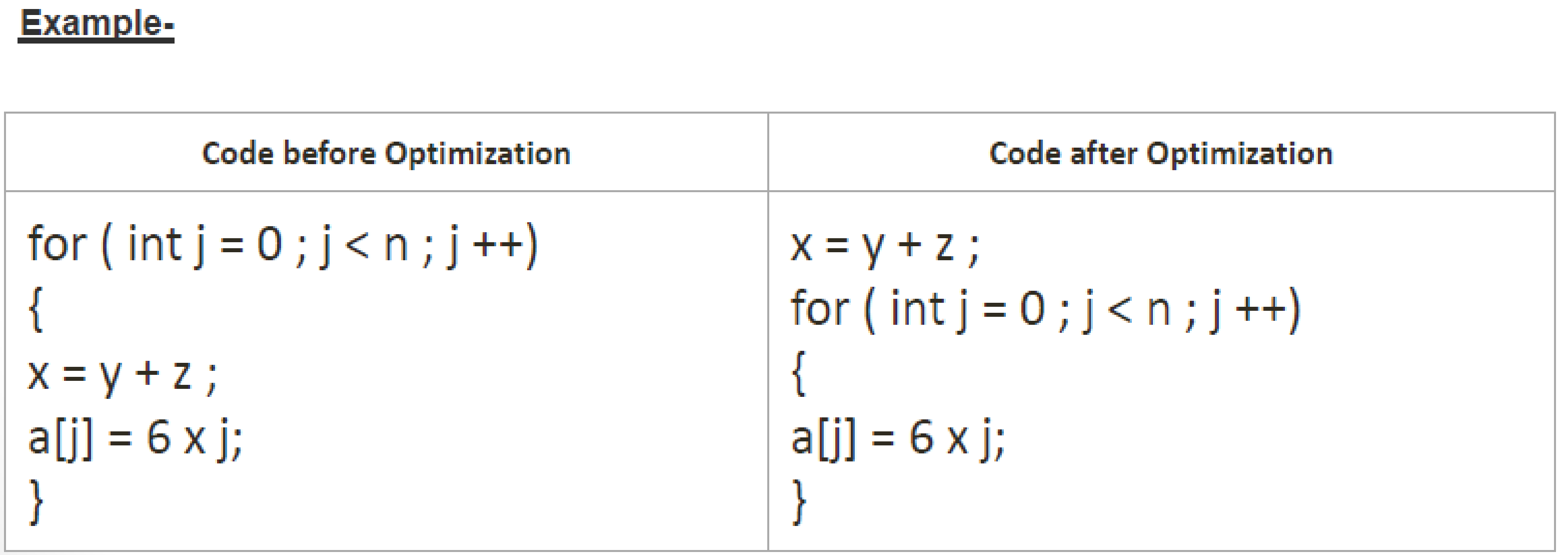
- Eliminates the redundant expressions and avoids their computation again and again.

**Example-**

| Code before Optimization | Code after Optimization |
|---|---|
| S1 = 4 x i<br>S2 = a[S1]<br>S3 = 4 x j<br>S4 = 4 x i **// Redundant Expression**<br>S5 = n<br>S6 = b[S4] + S5 | S1 = 4 x i<br>S2 = a[S1]<br>S3 = 4 x j<br>S5 = n<br>S6 = b[S1] + S5 |

# 3) Code movement

- The code is moved out of the loop if it does not matter whether it is present inside the loop or it is present outside the loop.

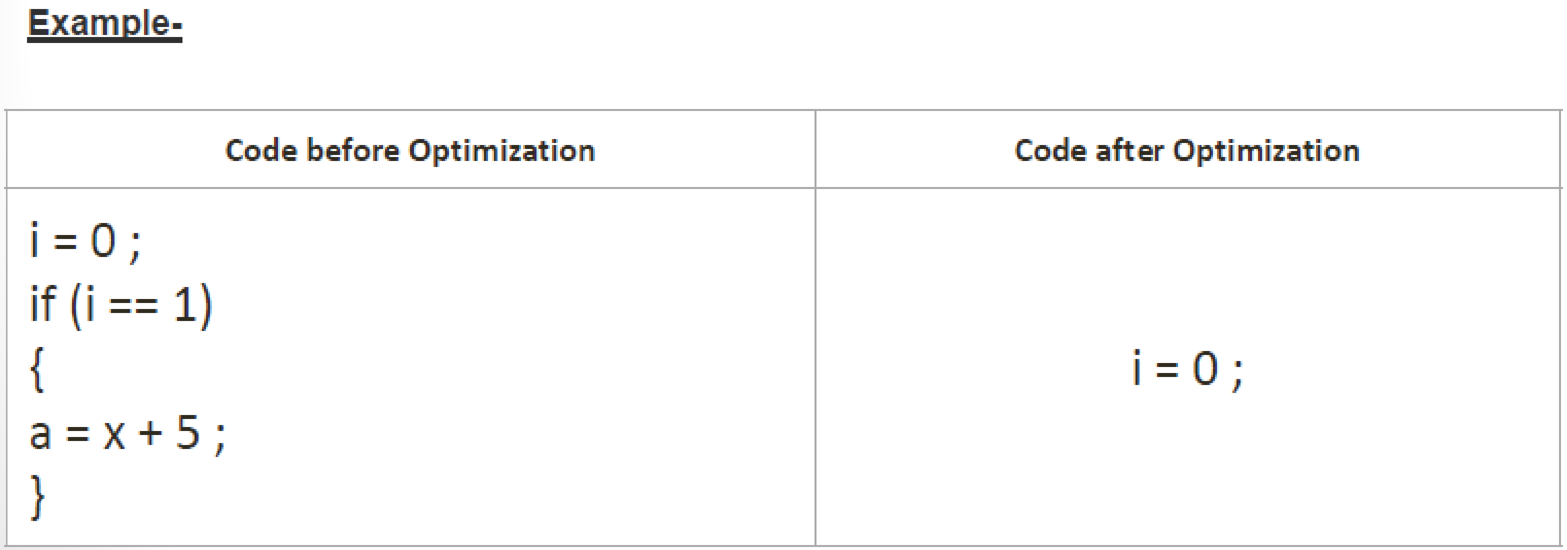


**Example-**

| Code before Optimization | Code after Optimization |
|---|---|
| for ( int j = 0 ; j < n ; j ++)<br>{<br>x = y + z ;<br>a[j] = 6 x j;<br>} | x = y + z ;<br>for ( int j = 0 ; j < n ; j ++)<br>{<br>a[j] = 6 x j;<br>} |

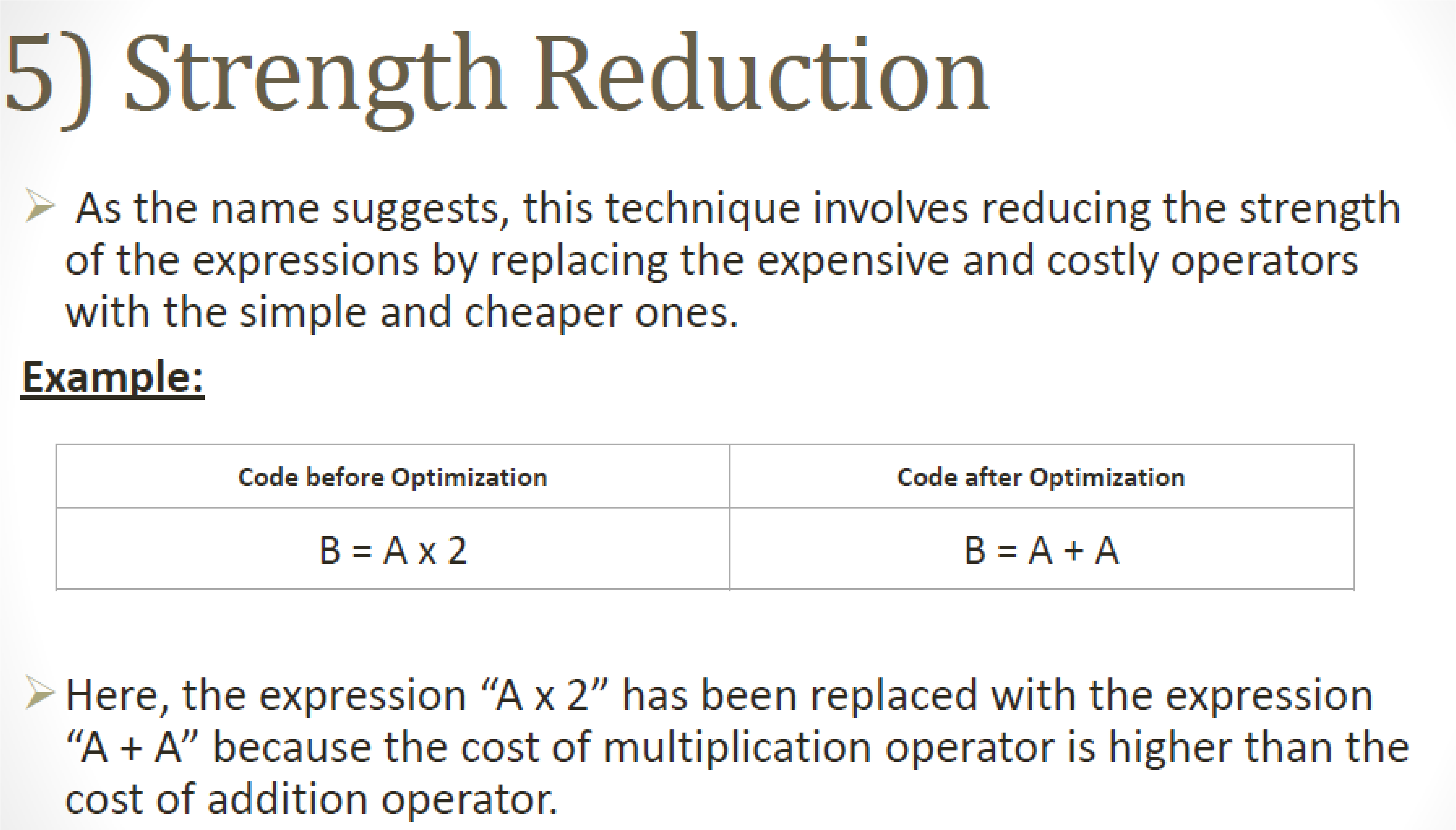
# 4) Dead code elimination

- Those code are eliminated which either never executes or are not reachable or even if they get execute, their output is never utilized.



**Example-**

| Code before Optimization | Code after Optimization |
|---|---|
| i = 0 ;<br>if (i == 1)<br>{<br>a = x + 5 ;<br>} | i = 0 ; |



# 5) Strength Reduction

- As the name suggests, this technique involves reducing the strength of the expressions by replacing the expensive and costly operators with the simple and cheaper ones.

**Example:**

| Code before Optimization | Code after Optimization |
|---|---|
| B = A x 2 | B = A + A |

- Here, the expression “A x 2” has been replaced with the expression “A + A” because the cost of multiplication operator is higher than the cost of addition operator.

# Basic Blocks

**Introduction:**

- ■ A *basic block is a sequence of consecutive instructions which are* always executed in sequence without halt or possibilty of branching.

- ■ *The basic block does not have any jump statements among them.*

# Basic Blocks(Contd..)

example

**Right**

1. **t1=b + c**
2. **t2=t1+d**
3. **a=t2**

**Wrong**

1. **if A<B goto(4)**
2. **t1=0**
3. **goto(5)**
4. **t1=1**

# Algorithms

1. **Determine the set of *leaders*,**

   **a**. The first statement is the leader.

   **b**. Any statement that is the target of a conditional or goto is a leader.

   **c**. Any statement that immediately follows conditional or goto is a leader.

**2**. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

**Consider the following three address code statements .Compute the basic blocks.**

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)

So the given code can be portioned in to 2 blocks as :

B1:

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

B2:

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

# Example on Basic Blocks

```
1. a=0;
2. if(a<10) goto(4)
3. goto(9)
4. t1=x+1
5. x=t1
6. t2=a+1
7. a=t2
8. goto (2)
9. exit
```

1. **Determine the set of *leaders*,**

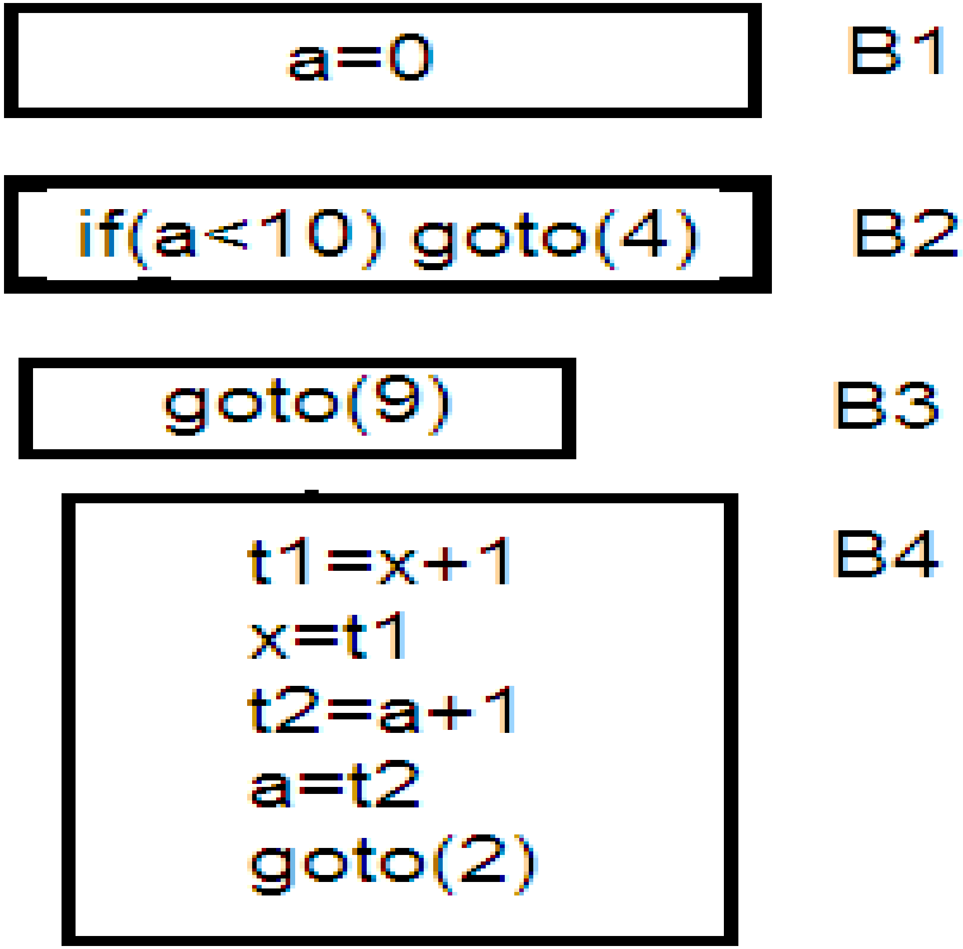
   **a**. The first statement is the leader.

   **b**. Any statement that is the target of a conditional or goto is a leader.

   **c**. Any statement that immediately follows conditional or goto is a leader.

**2**. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

# Example on Basic Blocks



B1

```
a=0
```

B2

```
if(a<10) goto(4)
```

B3

```
goto(9)
```

B4

```
t1=x+1
x=t1
t2=a+1
a=t2
goto(2)
```

# Flow Graph

**Introduction:**

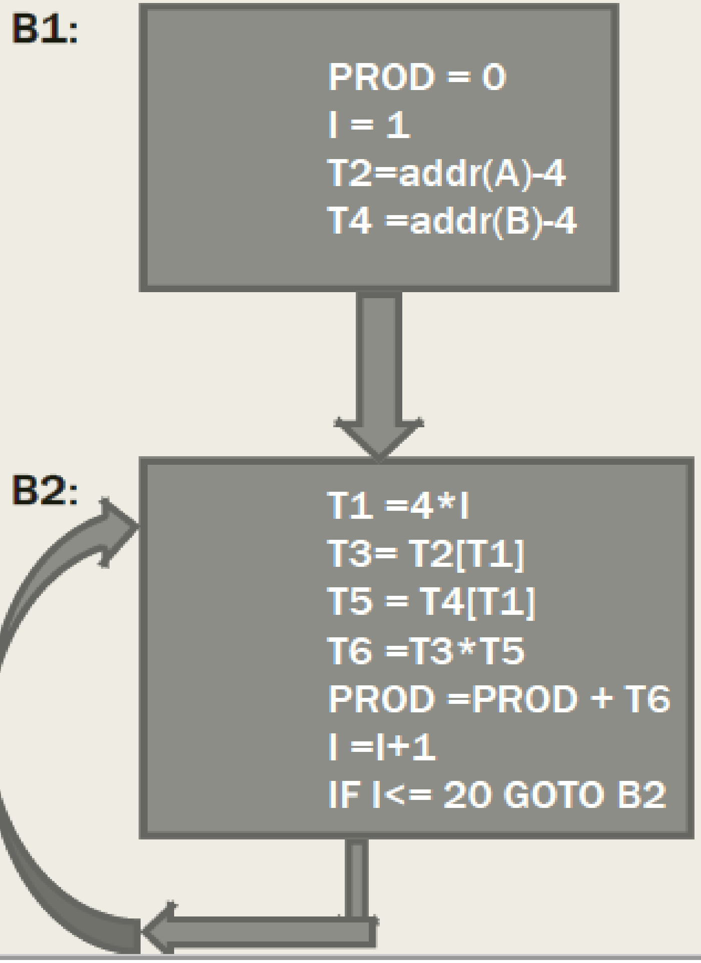
- A *flow graph is a graphical representation of a sequence of instructions with* control flow edges.
- A flow graph can be defined at the intermediate code level or target code level.
- The nodes of flow graphs are the basic blocks and flow-of-control to immediately follow node connected by directed arrow.

# 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 B1 to B2 if B2 immediately follows B1 in the given sequence
- Then we say that B1 is the predecessor of B2.

# Flow Graphs

The flow graph for the above three address code is given below:



B1:

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

B2:

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

# Directed Acyclic Graph(DAG)

**Definition:-**

In compiler design, a directed acyclic graph is an abstract syntax tree with a unique node for each value

# 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

# Application of DAG

- Determining common subexpression
- Determining which name are used in the block and computed outside the block
- Determining which statements of the block could have their computed value outside the block

# Rules for construction of DAG

1. In a DAG

   a. Leaf nodes represent identifiers, name or constants.

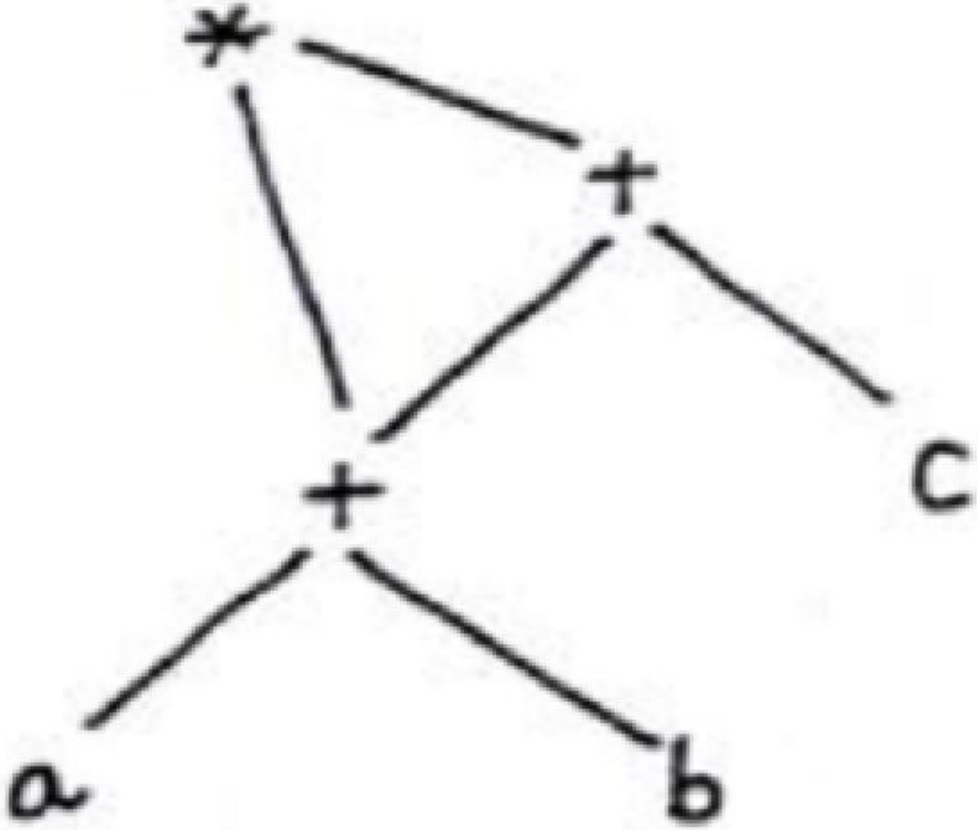
   b. Interior nodes represent operators

2. while constructing DAG, there is a check made to find if there is an existing node with same children. A new node is created only when such a node does not exist. This action allows us to detect common sub-expression .

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

# Construct DAG for the given expression (a+b)*(a+b+c)

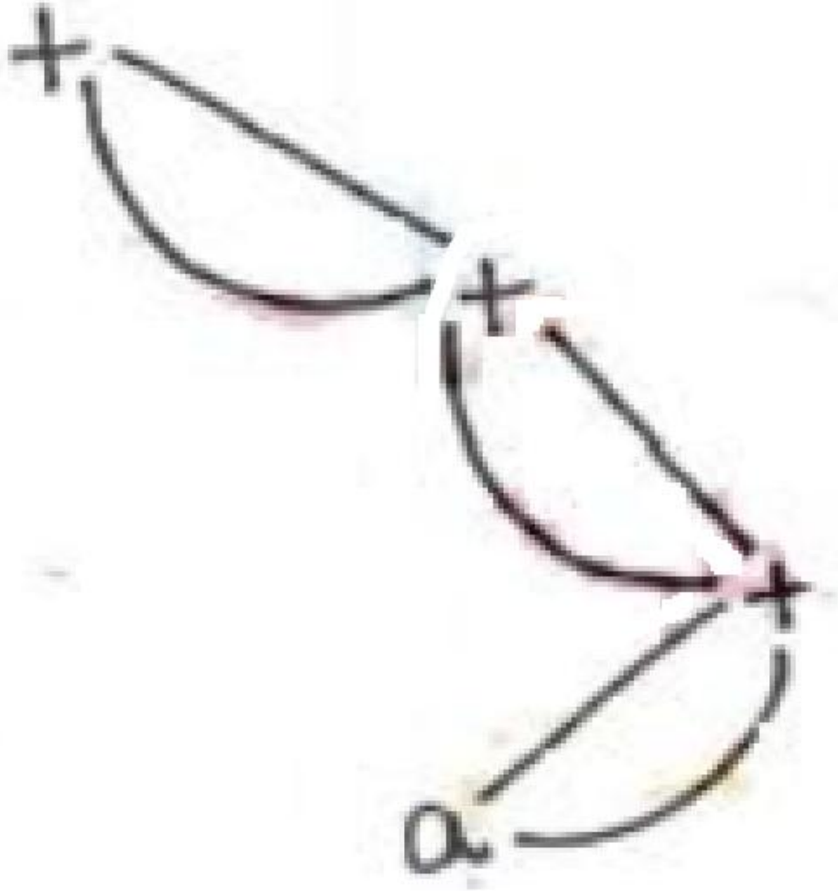
# Result



# Construct DAG for the given expression

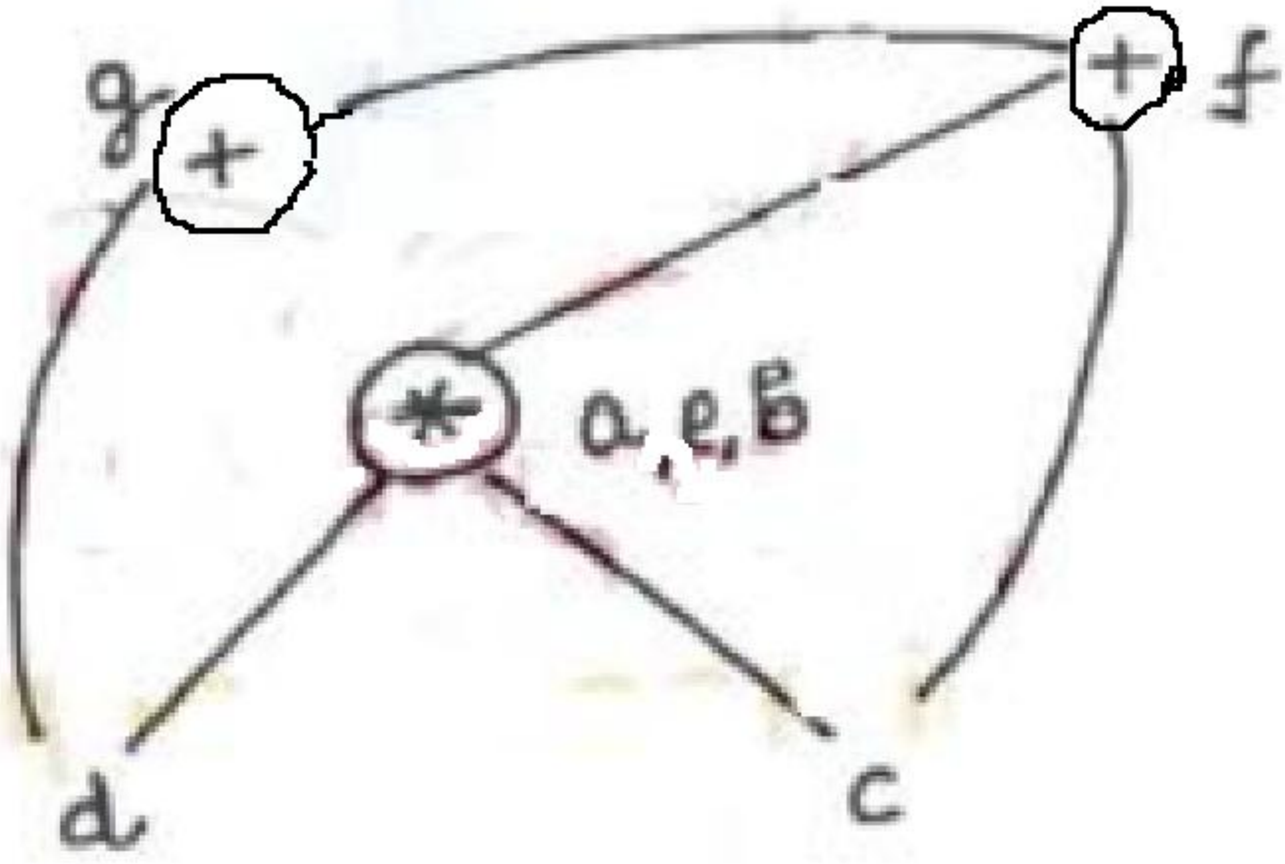
# (((a+a)+(a+a))+((a+a)+(a+a)))

# Result



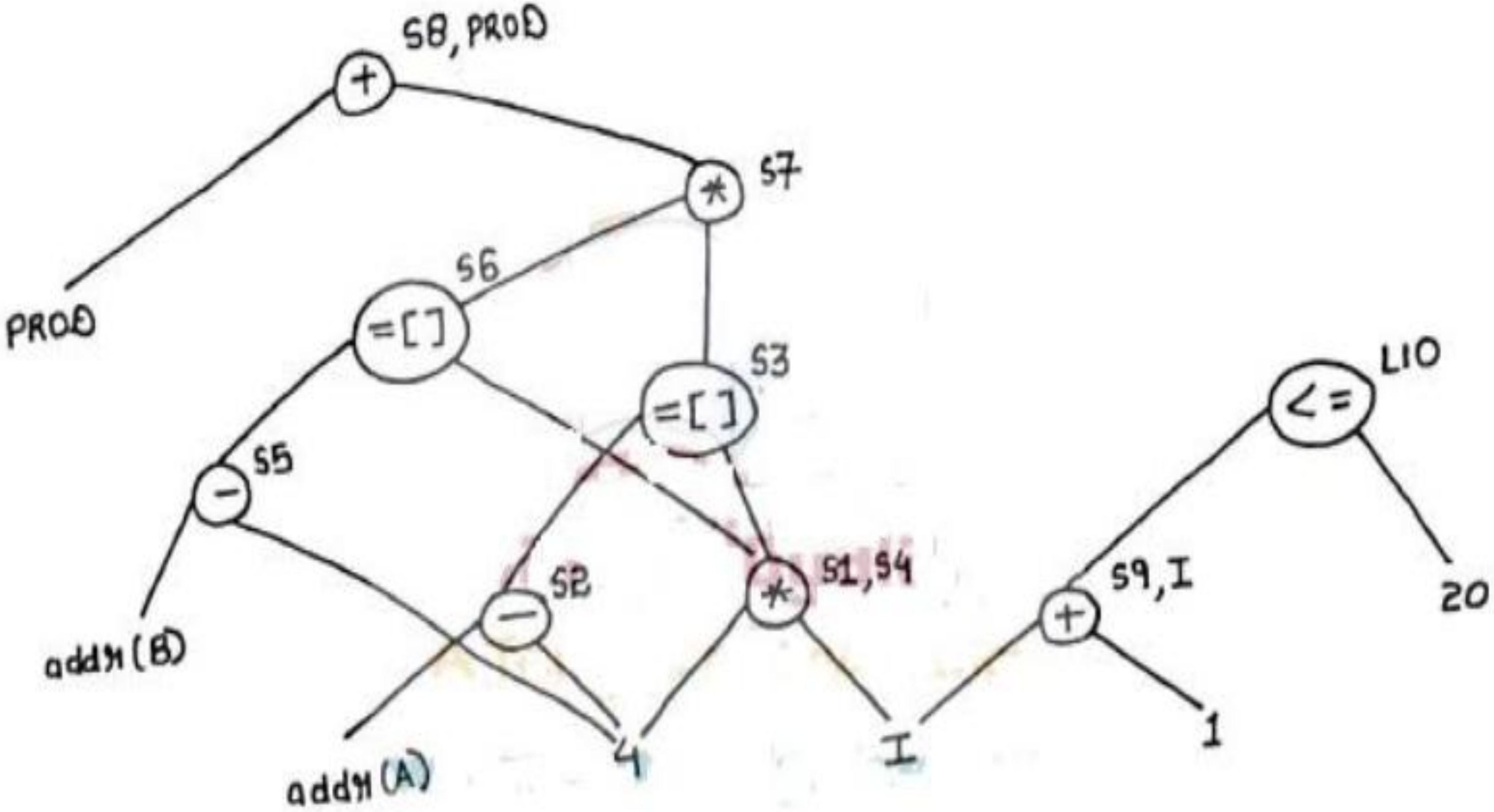
Construct DAG for the following basic blocks

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


g
+
+
f
*
a, e, b
d
c

# Draw the DAG representation of following code

```
1. s1=4*I
2. s2=addr(A)-4
3. s3=s2[s1]
4. s4=4*I
5. s5=addr(B)-4
6. s6=s5[s4]
7. s7=s3*s6
8. s8= PROD+s7
9. PROD=s8
10. s9=I+1
11. I=s9
12. if I<=20 goto(1)
```


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