# Code Optimization
## Module 5

*Prepared by*

*Jesna J S*

*Assistant Professor CSE, TKMCE*

# CODE OPTIMIZATION

- Code optimization refers to techniques used by a compiler to create a more efficient program from the given source code.
- The efficiency of a program is usually measured by its size or execution time.
- *For large computations, speed is crucial. On smaller computers, reducing program size can be just as important, or even more so.*
- *Need of optimization???*

## Why optimization is required?

Before Optimization

```
T1=b*c
T2=a+T1
T3=b*c
X=T3+c
Y=T2+d
```

After:

```
T1=b*c
T2=a+T1
X=T1+c
Y=T2+d
```

Before:

```
int i=5,j;
for(j=0;j<2*i+10;j++)
    printf("My value is %d",j);
```

```
T1 =b * c
T2 = a + T1
b = 20 * a
T3 = b * c
X = T3 + c
Y = T2 + d
```

## The three criteria used to select optimization techniques are:

1. The optimization should **provide significant improvements without requiring excessive effort** from the compiler developer or the compiler itself.
2. It must **preserve the original meaning of the source program**, ensuring that a correct program remains correct after optimization.
3. On average, the optimization should **reduce either the execution time or the memory usage** of the final program.

# Different stages of compilation where optimizations can be applied

- Source Code(User Level Optimizations).
- Front-End (Intermediate Code Optimization).
- Code Generator (Target Code Optimization).

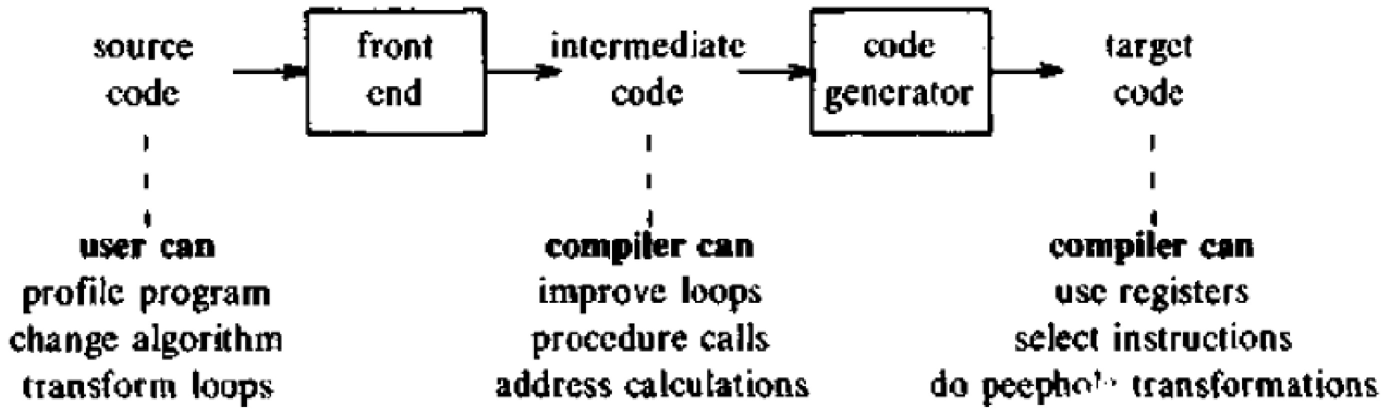


**Fig. 10.1.** Places for potential improvements by the user and the compiler.

## 1. Source Code (User Level Optimizations)

- At this stage, the user (programmer) can make optimizations before compilation.
- Possible improvements:
    - **Profile the program**: Analyze how the code runs to identify performance bottlenecks.
    - **Change the algorithm**: Use a more efficient algorithm to improve speed or reduce memory usage.
    - **Transform loops**: Optimize loops by techniques like loop unrolling or loop fusion.

## 2. Front-End (Intermediate Code Optimization)

- The compiler processes the source code and generates **intermediate code**.
- The compiler applies optimizations such as:
  - **Improving loops**: Enhancing loop structures to execute faster.
  - **Optimizing procedure calls**: Reducing overhead from function calls.
  - **Address calculations**: Optimizing memory address computations to minimize access time.

## 3. Code Generator (Target Code Optimization)

- This phase converts intermediate code into machine-specific **target code**.
- Further compiler optimizations include:
  - **Register allocation**: Efficiently using CPU registers to reduce memory access.
  - **Instruction selection**: Choosing the best machine instructions for execution.
  - **Peephole optimization**: Making small local changes to improve efficiency, such as removing redundant instructions.

## Organization of Code optimizer

- This diagram shows how a compiler's **code optimizer** improves program efficiency by analyzing control flow and data flow, applying transformations, and then passing the optimized code to the **code generator**.
- This step is crucial in reducing execution time and memory usage.
- The **front end** generates intermediate code, which is then optimized before being sent to the **code generator**.
- The optimization process involves **control-flow analysis** (examining execution sequences, eliminating dead code), **data-flow analysis** (tracking variable usage, removing redundant calculations), and **transformations** (applying improvements like loop optimizations and strength reduction).

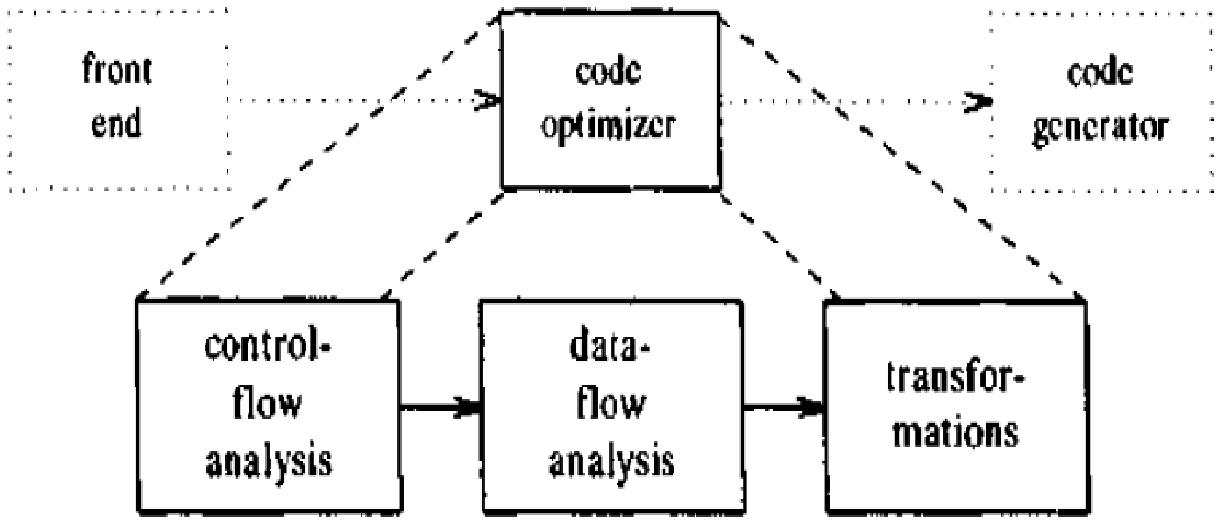


Fig. 10.3. Organization of the code optimizer.

## Types of Optimization

**1.Machine-Dependent Optimization**

- These optimizations are designed for a specific hardware architecture.
- They take advantage of the unique features of a particular machine, such as its processor, memory hierarchy, and instruction set.
- Examples include register allocation, instruction scheduling, and cache optimization.
- Since they are specific to a machine, they may not work efficiently on other systems.

## 2. Machine-Independent Optimization

- These optimizations apply to any machine, regardless of its hardware architecture.
- They focus on improving code efficiency at a higher level, before considering machine-specific details.
- Examples include common subexpression elimination, loop optimization, and dead code elimination.
- These techniques improve program performance without relying on hardware-specific features

***Optimization can be done in 2 phases:***

1. Local optimization:
   - Transformations are applied over a small segment of the program called basic block, in
   - which statements are executed in sequential order.
2. Global optimization
   - Transformations are applied over a large segment of the program like loop, procedures, functions etc.
   - Global optimization must be done before local optimization.

## BASIC BLOCK

- A basic block is a sequence of consecutive statements in which **flow of control enters at the beginning and leaves at the end without half or possibility of branching except at the end**.
- A basic block has no branching except at the end and no entry points except at the beginning.
- It helps in control-flow analysis and optimization by treating it as a single unit.
- Common optimizations performed on basic blocks include constant folding, dead code elimination, and common subexpression elimination.
- The following sequence of three-address statements forms a basic block:

```
t1 = a + a
t2 = a + b
t3 = 2 * t2
t4 = t1 + t3
t5 = b * b
t6 = t4 + t5
```

*The following algorithm can be used to partition a sequence of three address statements into basic blocks.*

**Input** : A sequence of three-address statements.

**Output** : A list of basic blocks with each three-address statement in exactly one block.

**Method** :

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are the following.
   1. The first statement is a leader.
   2. Any statement that is the target of a conditional or unconditional goto is a leader.
   3. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic blocks consists of the leader and all the statements up to but not including next leader or the end of the program.

```
k = 10 ;
m = k + j
if (m < 25)
 {
   m = 1000 ;
   j = j - 1 ;
   s = 0 ;
 }
p = m + j ;
```

```
1.  k = 10              ———— Leader (Rule1)
2.  t1 = k + j
3.  m = t1
4   if (m < 25)  goto (7)
5.  t2 = m + j          ———— Leader (R3)
6.  p = t2   goto (11)
7.  m = 1000            ———— Leader (R2/R3)
8.  t3 = j - 1
9.  j = t3
10. s = 0           goto 5
11.
```

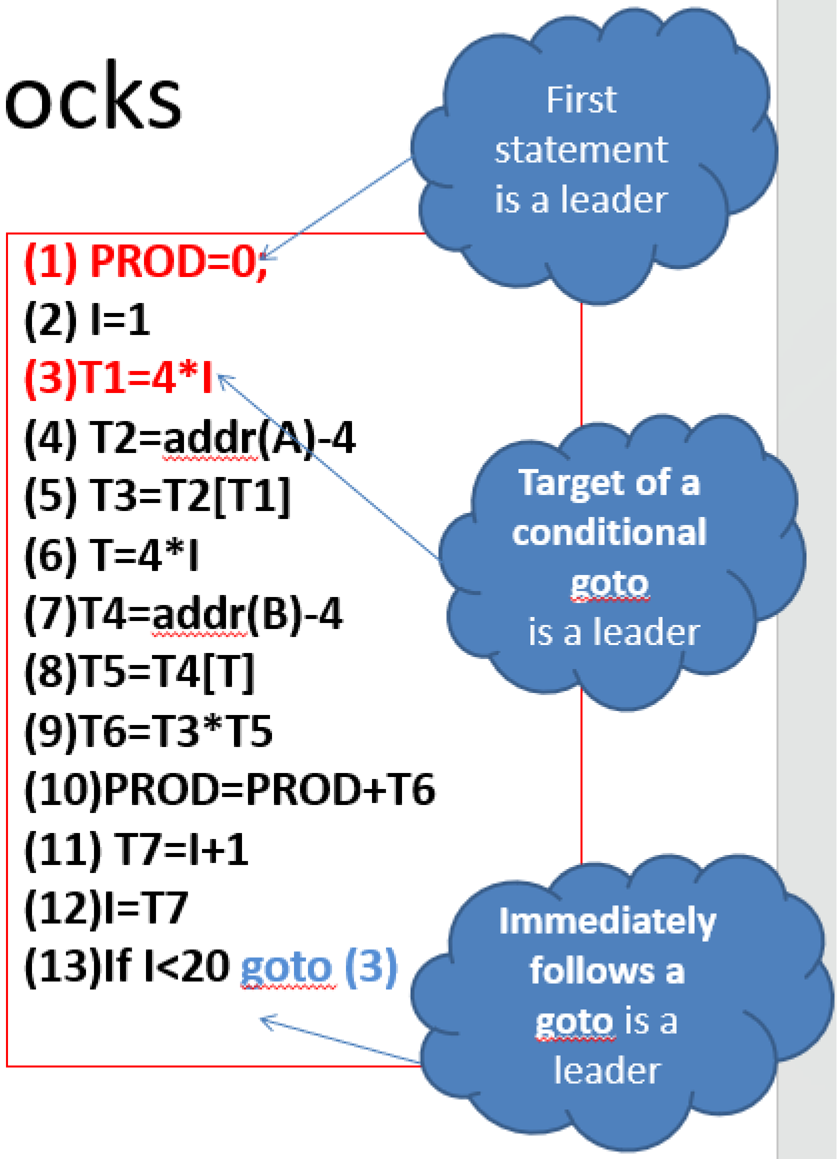
**Eg: Program for computing Dot product**

```
begin
       prod := 0;
       i := 1;
      do
            begin
                  prod := prod + a[i] * b[i];
                   i := i + 1;
            End
      while i <= 20
end
```

```
(1)PROD = 0
(2) i = 1
(3)T1 = 4 * I
(4)T2 = addr(A) – 4
(5)T3 = T2[T1]
(6)T4 = 4 * I
(7)T5 = addr(B) – 4
(8)T6 = T5[T4]
(9)T7 = T3 * T6
(10) PROD = PROD + T7
(11) T8 = i + 1
(12) i = T8
(13) if i < 20 goto (3)
```

# Basic Blocks

```
(1) PROD=0;
(2) I=1
(3)T1=4*I
(4) T2=addr(A)-4
(5) T3=T2[T1]
(6) T=4*I
(7)T4=addr(B)-4
(8)T5=T4[T]
(9)T6=T3*T5
(10)PROD=PROD+T6
(11) T7=I+1
(12)I=T7
(13)If I<20 goto (3)
```



```
(1) PROD=0;
(2) I=1
(3)T1=4*I
(4) T2=addr(A)-4
(5) T3=T2[T1]
(6) T=4*I
(7)T4=addr(B)-4
(8)T5=T4[T]
(9)T6=T3*T5
(10)PROD=PROD+T6
(11) T7=I+1
(12)I=T7
(13)If I<20 goto (3)
```

First statement is a leader
Target of a conditional goto is a leader
Immediately follows a goto is a leader

# FLOW GRAPH

- A flow graph is a directed graph representing the flow of control among basic blocks in a program.
- The nodes in the flow graph represent basic blocks.

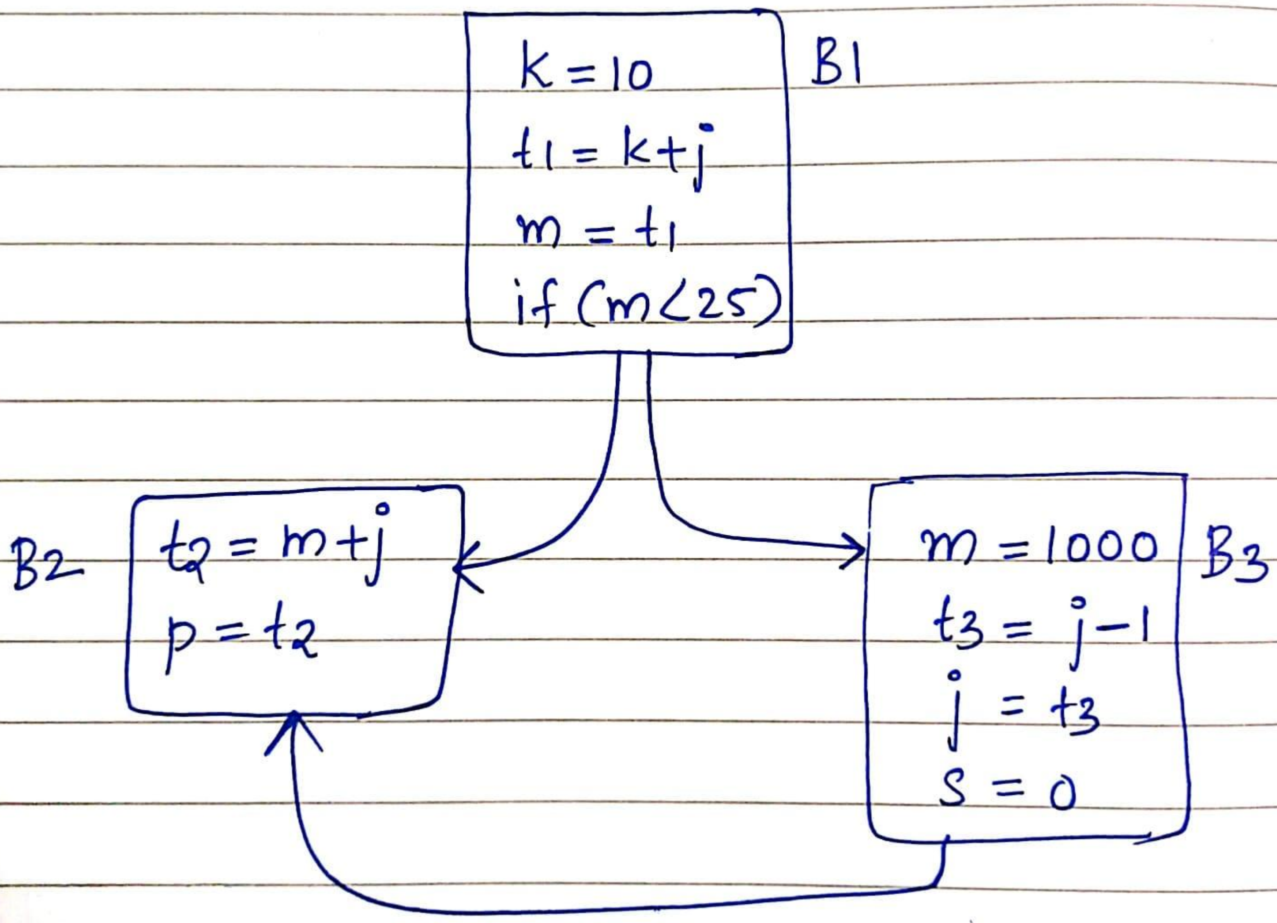
✓Initial Node: One node is marked as initial, representing the first basic block of the program.

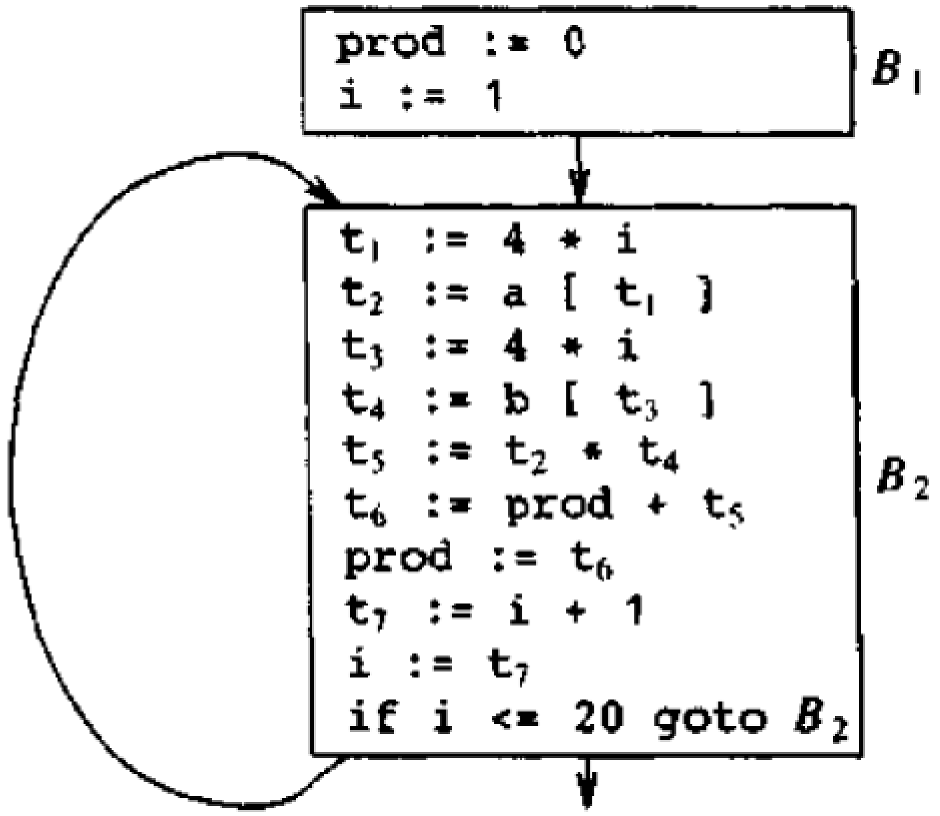
✓There is a directed edge from block $B_1$ to block $B_2$ can immediately follow B1 in come execution sequence; that is, if

1. ***There is a conditional or unconditional jump from the last statement of $B_1$ to the first statement of $B_2$, or***
2. ***$B_2$ immediately follows $B_1$ in the order of the program, and $B_1$ does not end in an unconditional jump.***

We say that $B_1$ is a predecessor of $B_2$, and $B_2$ is a successor of $B_1$.

1. $K = 10$ —— Leader Rule1
2. $t_1 = k + j$
3. $m = t_1$
4 if $(m < 25)$ goto 7
5. $t_2 = m + j$ —— leader R3
6. $p = t_2$ goto 11
7. $m = 1000$ —— Leader R2/R3
8. $t_3 = j - 1$
9. $j = t_3$
10. $s = 0$ **goto 5**
11.





```
prod := 0
i := 1
```

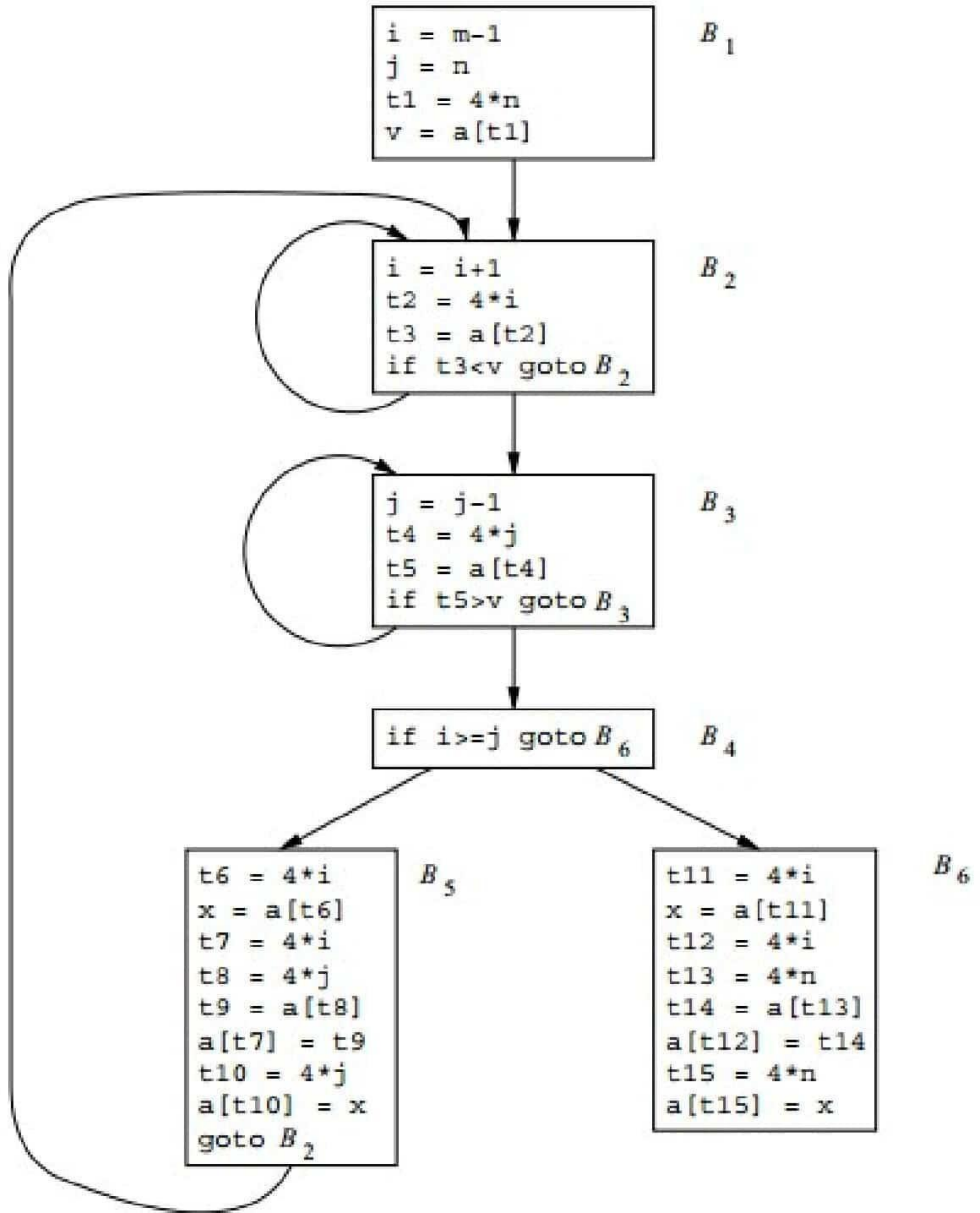
```
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
t4 := b [ t3 ]
t5 := t2 * t4
t6 := prod + t5
prod := t6
t7 := i + 1
i := t7
if i <= 20 goto B2
```

**Fig. 9.9.** Flow graph for program.

## Draw flow graph

```
(1)   i := m-1
(2)   j := n
(3)  t1 := 4*n
(4)   v := a[t1]
(5)   i := i+1
(6)  t2 := 4*i
(7)  t3 := a[t2]
(8)  if t3 < v goto (5)
(9)   j := j-1
(10) t4 := 4*j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15)  x := a[t6]
(16)        t7 := 4*i
(17)        t8 := 4*j
(18)        t9 := a[t8]
(19)   a[t7] := t9
(20)       t10 := 4*j
(21)  a[t10] := x
(22)      goto (5)
(23)       t11 := 4*i
(24)         x := a[t11]
(25)       t12 := 4*i
(26)       t13 := 4*n
(27)       t14 := a[t13]
(28)  a[t12] := t14
(29)       t15 := 4*n
(30)  a[t15] := x
```

**Fig. 10.4.** Three-address code for fragment in Fig. 10.2.



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

```
if i>=j goto B6
```

```
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
```

# Principle Sources of Optimization

- A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
- Improves the quality of the code.
- Two classes of local transformations that can be applied to basic blocks.
  - ✓Structure preserving transformations
  - ✓Algebraic transformations.

1. Structure preserving transformations.
   1. Common subexpression elimination
   2. Dead-code elimination
   3. Renaming of temporary variables
   4. Interchange of two independent adjacent statements

## 1. Common Sub expression elimination

- Identifies and eliminates duplicate expressions that compute the same value multiple times
- Optimizes Register and Memory Usage
- Two types:
    1. Local common sub expression elimination
    2. Global common sub expression elimination

**1. Local common sub expression elimination**

✓Within the same block

```
a = b + c
b = a – d
c = b + c
d = a - d
```

$\Longrightarrow$

```
a = b + c
b = a – d
c = b + c
d = c
```

## 2. Global common sub expression elimination

- Applied across multiple basic block
- More complex due to inter-block dependency checks.

------------------------------------------------------------------------

*Basic Block 1:*

```
    t1 = x * y
    a = t1 + z
    t2 = a + b
```

------------------------------------------------------------------------

*Basic Block 2:*

```
    t3 = x * y   ; Same expression computed again
    b = t3 – w
```

$\Longrightarrow$

*Basic Block 1:*

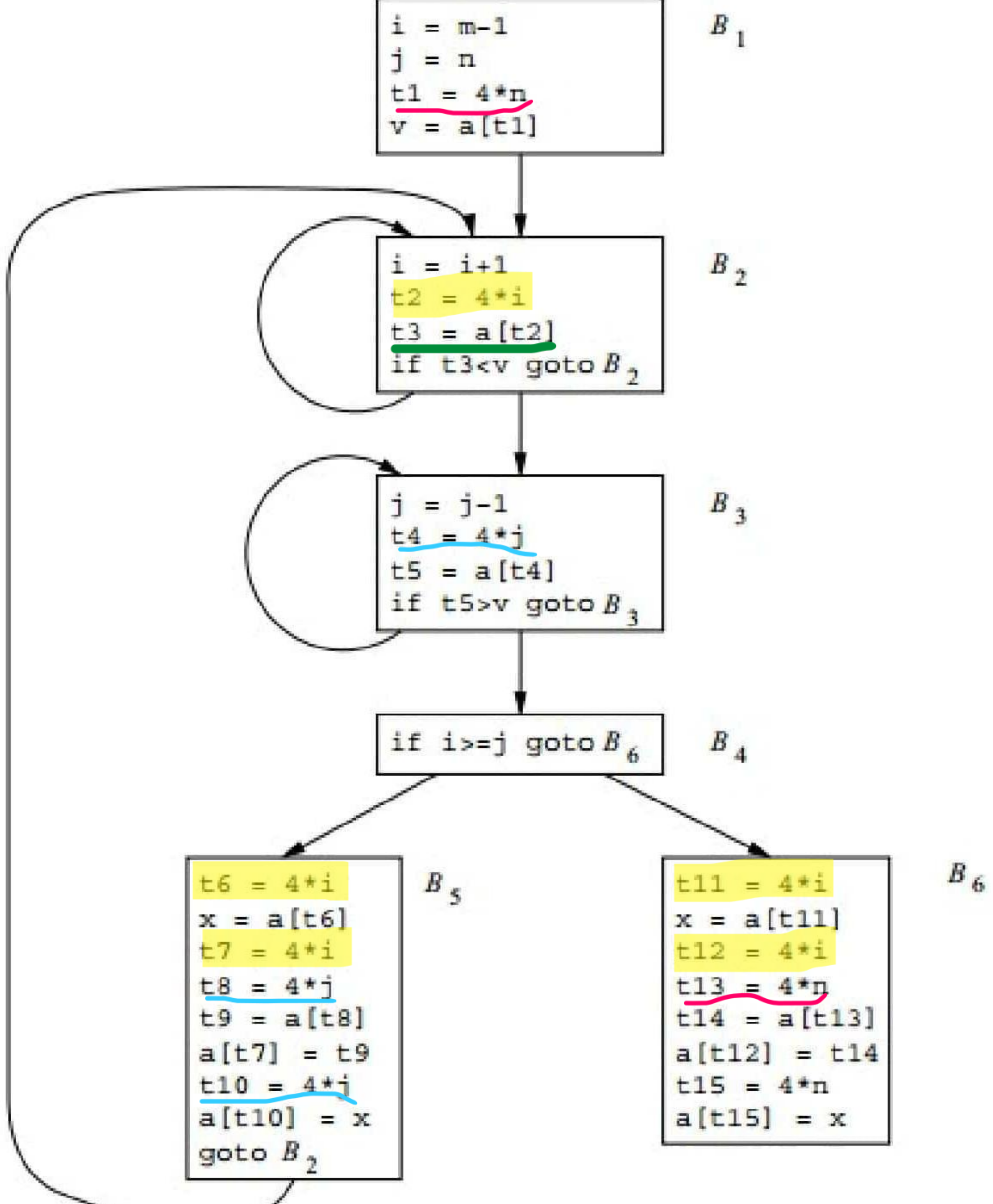
```
    t1 = x * y
    a = t1 + z
    t2 = a + b
```

--------------------

*Basic Block 2:*

```
    b = t1 - w
```

***The redundant computation t2 = x * y across blocks is removed, and t1 is reused.***



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

```
if i>=j goto B6
```

```
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
```

## B5

```
T6=4*I
X=a[T6]
T8=4*J
T9=a[T8]
a[T6]=T9
a[T8]=X
goto B2
```

```
X=a[T2]
T9=a[T4]
a[T2]=T9
a[T4]=X
goto B2
```

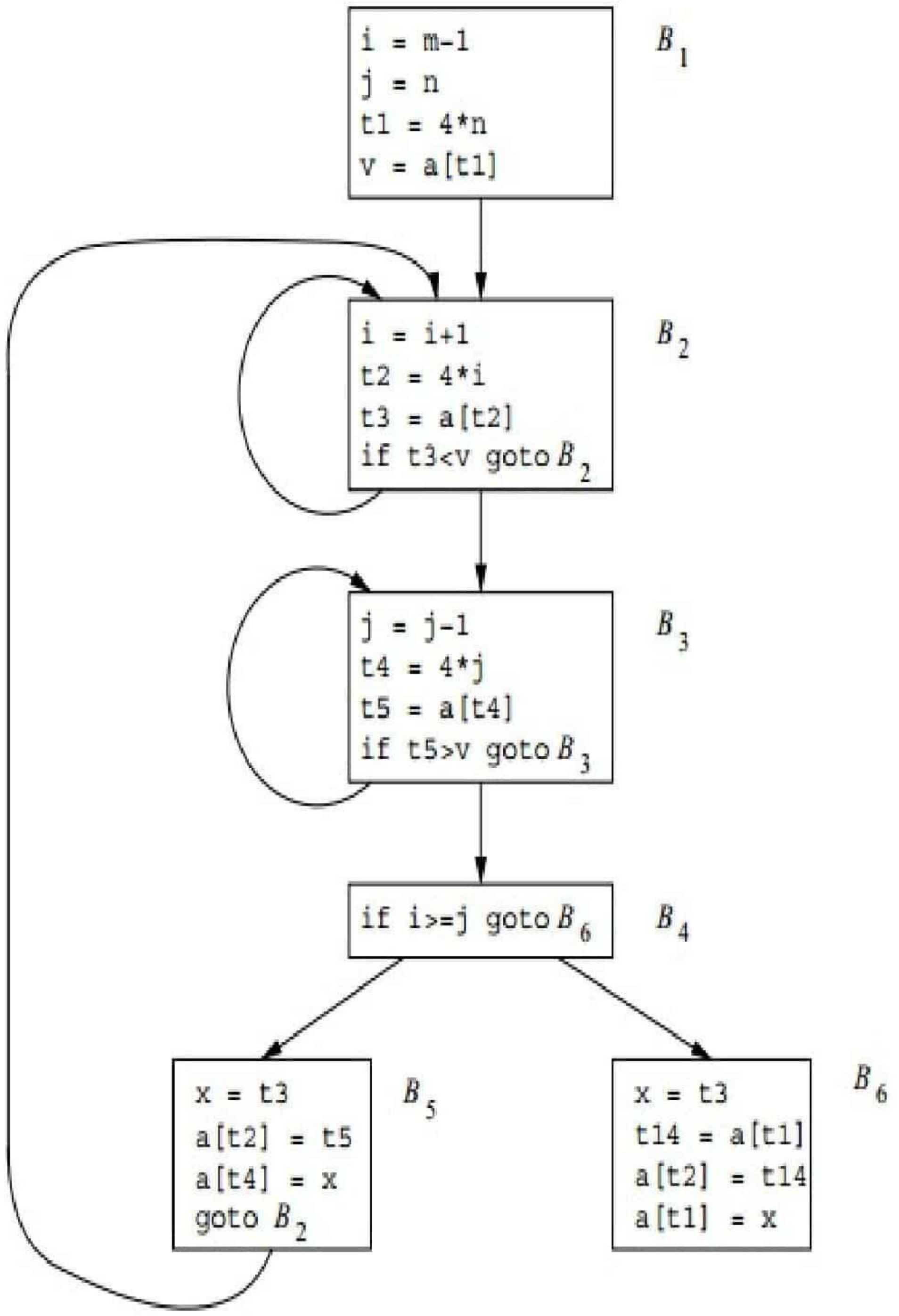
```
X=T3
a[T2]=T5
a[T4]=X
goto B2
```

## B6

```
T11=4*I
X=a[T11]
T13=4*N
T14=a[T13]
a[T11]=T14
a[T13]=X
```

```
X=a[T2]
T14=a[T1]
a[T2]=T14
a[T1]=X
```

```
X=T3
T14=a[T1]
a[T2]=T14
a[T1]=X
```



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

```
if i>=j goto B6
```

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

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

## 2. Copy Propagation

- Replaces redundant variable copies with their original values.
- This helps reduce unnecessary assignments and improves code efficiency.
- Improves Register Allocation.

*Before Optimization.*

**t1 = a**

**t2 = t1 + b**

**c = t2**

$\Longrightarrow$

*After Optimization.*

**t2 = a + b**

**c = t2**

## Dead Code Elimination

- Removes code statements that **do not affect the program's output**.
- This helps reduce program size, improve execution speed, and enhance readability.
- Code is considered dead when:
    - ✓It assigns a value to a variable that is never used later.
    - ✓It performs computations whose results are never used.
    - ✓It contains unreachable statements (code that can never execute).

*Before optimization*

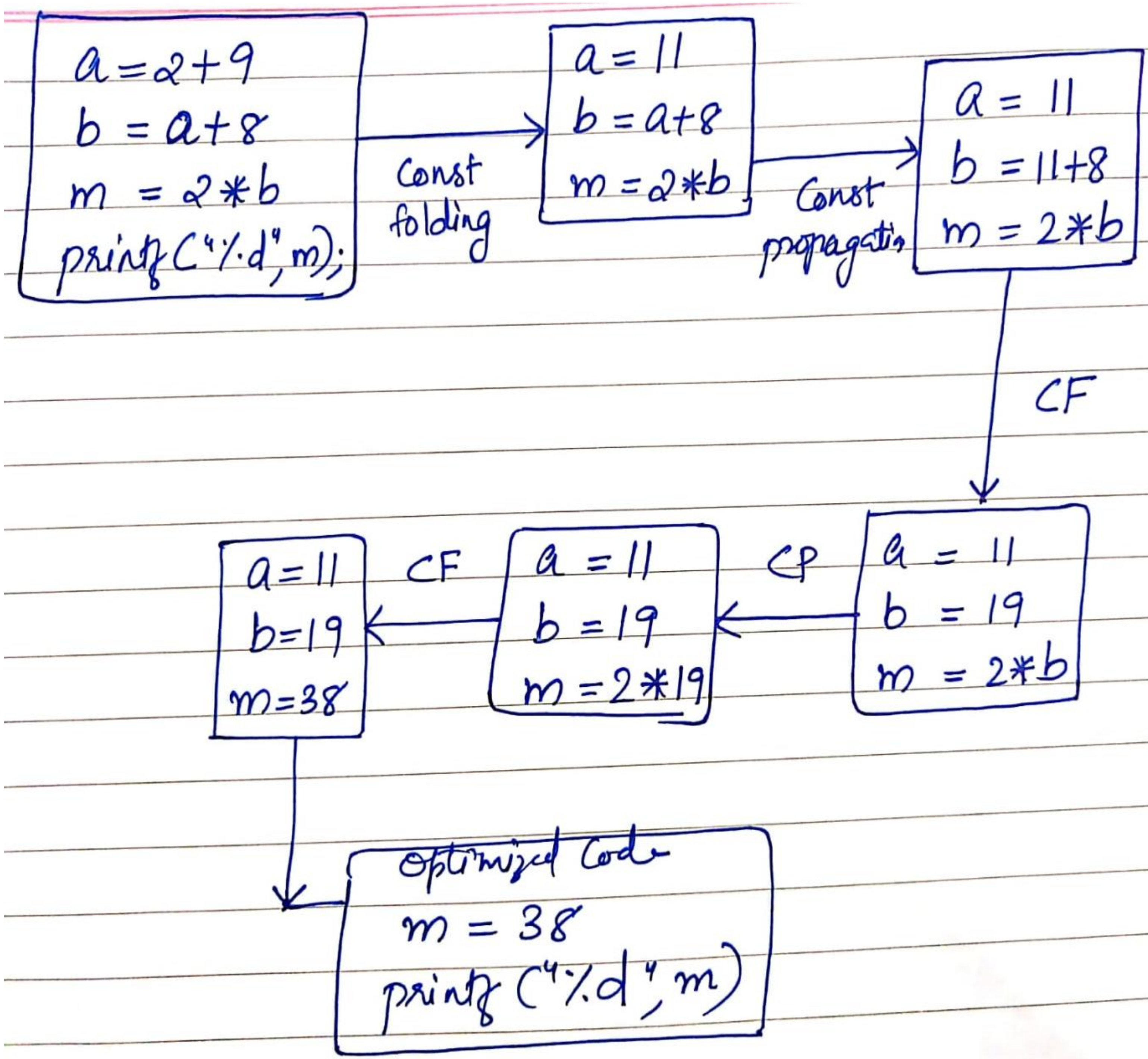
```
x = 5
y = x + 2
b = 10
z = y * 3
a = z - 4  ;  'b' is assigned 10 but never used
```

*After optimization*

```
x = 5
y = x + 2
z = y * 3
a = z - 4
```

## Constant folding & Constant Propagation

- Helps eliminate unnecessary calculations and improve efficiency.
- Constant Folding is an optimization where the compiler evaluates constant expressions at **compile time** instead of runtime.
- Constant Propagation replaces variables that have known constant values with those constants, reducing variable lookups and further enabling optimizations like **Dead Code Elimination (DCE)**.



```
a=2+9
b = a+8
m = 2*b
printf("%d", m);
```

```
a=11
b=a+8
m=2*b
```

```
a = 11
b = 11+8
m = 2*b
```

```
a = 11
b = 19
m = 2*b
```

```
a = 11
b = 19
m = 2*19
```

```
a=11
b=19
m=38
```

Optimized Code

```
m = 38
printf("%d", m)
```

## LOOP OPTIMIZATION TECHNIQUES

- Loop optimization techniques enhance the performance of loops by reducing execution time and memory overhead.
- Three key techniques are
    1. Induction Variable Elimination
    2. Code Motion
    3. Reduction in Strength

# CODE MOTION

- Moves computations that **do not change within a loop to outside loop** or repeated expressions to reduce redundant calculations and improve execution efficiency.
- Improves Execution Speed and optimizes Memory Usage.

```
for (int i = 0; i < n; i++)
{
    int x = a + b;
    arr[i] = x * i;
}
```

$\longrightarrow$

**Optimized Code**

```
int x = a + b;
for (int i = 0; i < n; i++)
{
        arr[i] = x * i;
}
```

**Example 9.5:** Evaluation of *limit* − 2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit */
```

Code motion will result in the equivalent code

```
t = limit-2
while (i <= t)  /* statement does not change limit or t */
```

## Induction Variable Elimination & Strength Reduction

- Induction variables are variables in a loop that change systematically (e.g., incrementing by a fixed value).
- Reduces redundant induction variables and simplifies loop indexing.

```
for (int i = 0, j = 0; i < n; i++, j = j + 2)
{
    A[j] = i;
}   // Here, j (j = j + 2) is an induction variable dependent on i.
```

**After Optimization**

```
for (int i = 0; i < n; i++)
 {
    A[2 * i] = i;  // Eliminating redundant induction variable j
}
```

- **Replace expensive operations** (such as multiplication, division, or modulus) with cheaper and computationally **efficient alternatives** (such as addition, subtraction, or bitwise shifts).

*Before optimization*

```
for (int i = 0; i < n; i++) {
    int x = 4 * i;  // Expensive multiplication inside loop
    arr[i] = x + 5;
}
```

*After Optimization*

```
int x = 0;  // Stores 4 * i without multiplication
for (int i = 0; i < n; i++) {
    arr[i] = x + 5;
    x = x + 4;  // Replaces 4 * i with addition
}
```

**Strength Reduction**:

- Replace multiplication (4 * i) with an equivalent addition.

**Eliminate Induction Variable**:

- Instead of computing 4 * i in every iteration, use a separate variable (x) to store the value and update it using addition.

# Loop Jamming

- **C**ombines multiple loops that have the same iteration range into a single loop.
- Helps in reducing loop overhead, improving cache locality, and enhancing overall performance.

```
for (int i = 0; i < n; i++)
{
   A[i] = B[i] + C[i];
}
for (int i = 0; i < n; i++)
{
   D[i] = E[i] * F[i];
}
```

*After Loop Jamming*

```
for (int i = 0; i < n; i++)
{
   A[i] = B[i] + C[i];
   D[i] = E[i] * F[i];
}
```

## Loop Unrolling

- The number of iterations in a loop is reduced by **executing multiple loop iterations within a single iteration**.
- This minimizes the overhead of loop control (like incrementing loop variables and checking conditions), improving performance.

```
for (int i = 0; i < n; i++)
{
    A[i] = A[i] + 5;
}
```

Unrolled by a factor 2

```
for (int i = 0; i < n; i += 2)
{
    A[i] = A[i] + 5;
    A[i + 1] = A[i + 1] + 5;
}// optimized code
```

Unrolled by a factor 4

```
for (int i = 0; i < n; i += 4)
{
    A[i] = A[i] + 5;
    A[i + 1] = A[i + 1] + 5;
    A[i + 2] = A[i + 2] + 5;
    A[i + 3] = A[i + 3] + 5;
}// optimized code
```

## Optimization of basic blocks

1. Directed Acyclic graphs
2. Use of algebraic identities

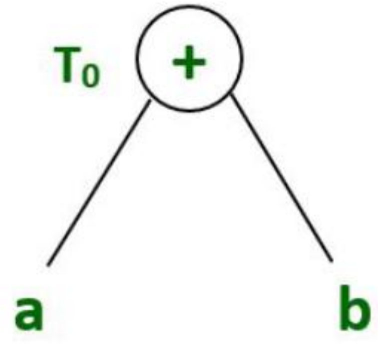
## DAG (DIRECTED ACYCLIC GRAPH)

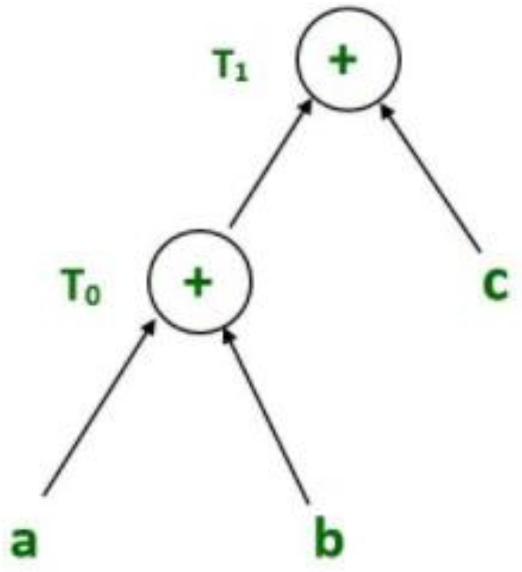
- A **Directed Acyclic Graph (DAG)** is a data structure used in compiler optimization to represent expressions in a way that **eliminates common subexpressions and optimizes redundant computations**.
- It helps in generating efficient intermediate code and optimizing computations.
- DAG Creation:
  1. Identify Expressions and Variables:
  2. Create Nodes for Operands : *If an operand (variable or constant) is encountered for the first time, create a node for it*.
  3. Create Nodes for Operators:
  4. Eliminate Common Subexpressions:
  5. Mark Final Computation Results:

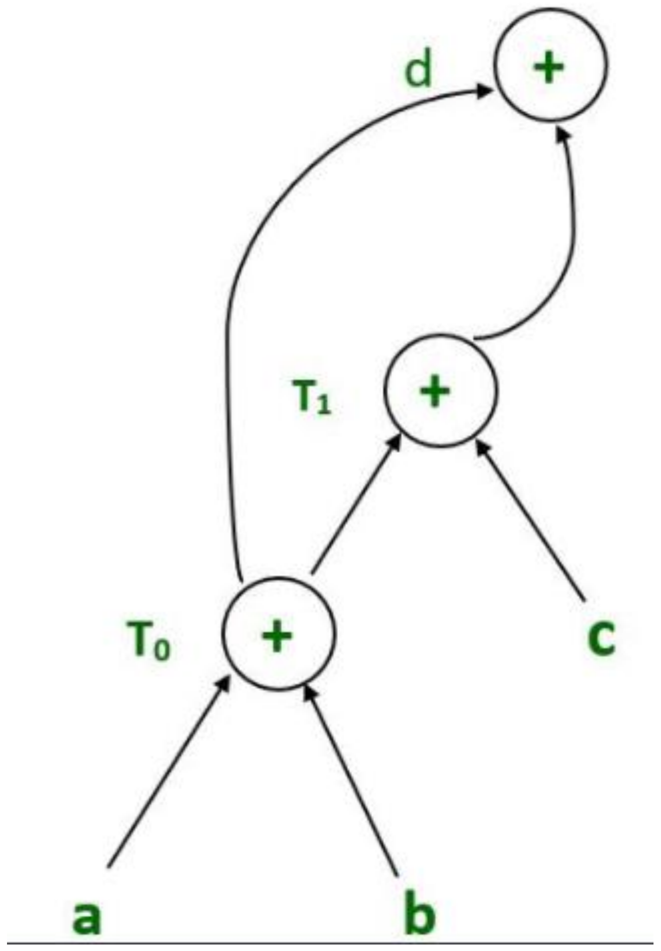
## Applications of DAG

- ✓Common Subexpression Elimination
- ✓Dead Code Elimination
- ✓Code Motion (Loop-Invariant Code Motion)
- ✓DAGs help **rearrange computations** to minimize CPU cycles and improve instruction scheduling.

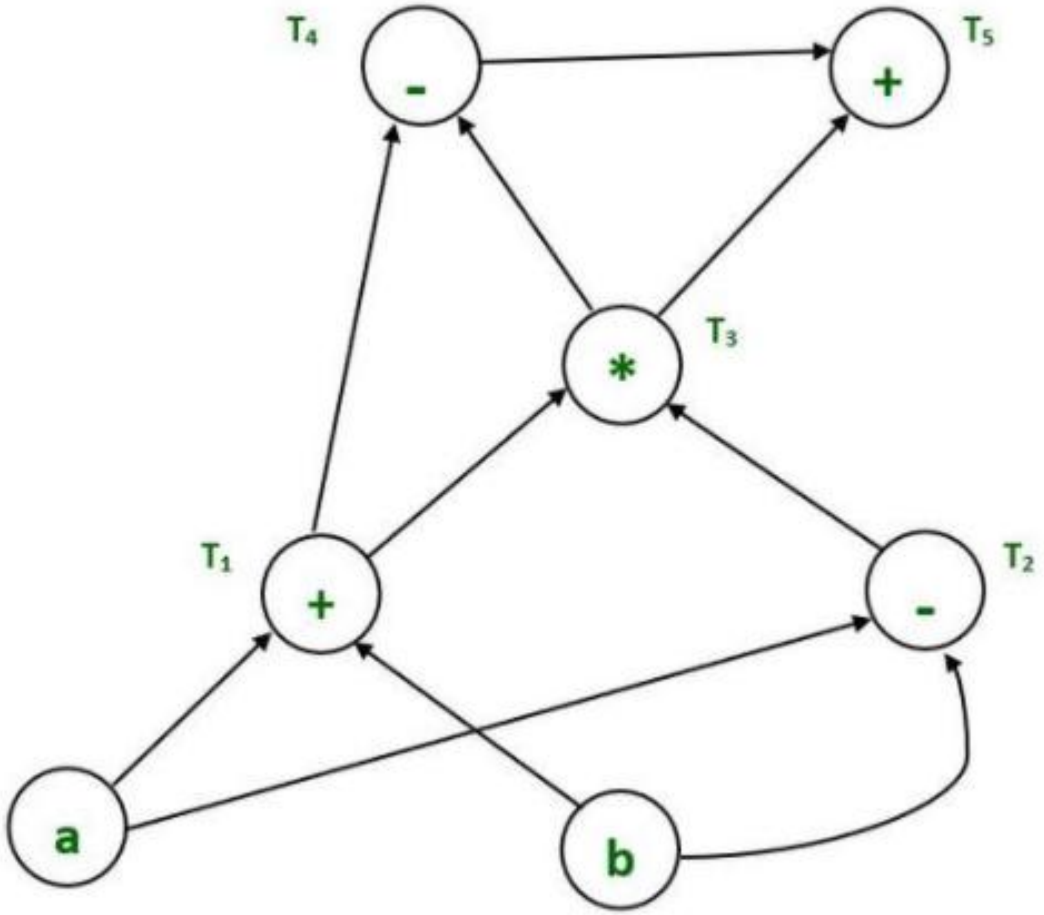
$T_0 = a + b$ —*Expression 1*
$T_1 = T_0 + c$ —-*Expression 2*
$d = T_0 + T_1$ ——*Expression 3*



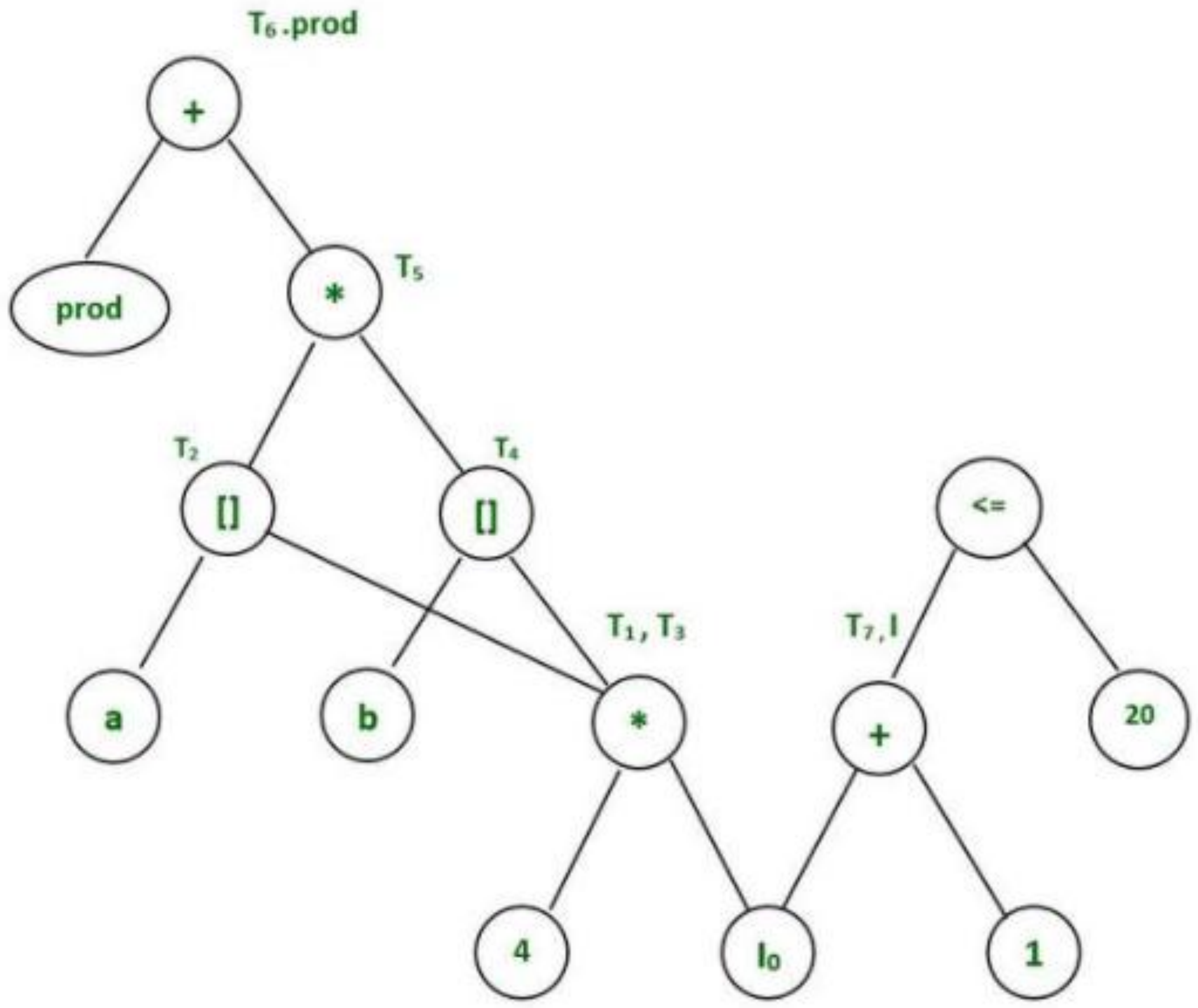




T1 = a + b
T2 = a – b
T3 = T1 * T2
T4 = T1 – T3
T5 = T4 + T3



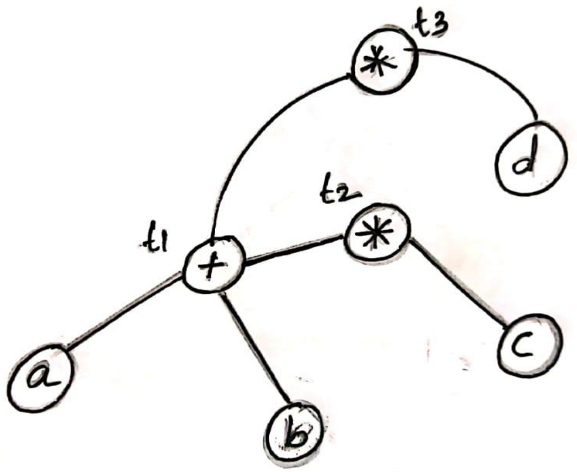
1. $T_1 := 4 * I_0$
2. $T_2 := a[T_1]$
3. $T_3 := 4 * I_0$
4. $T_4 := b[T_3]$
5. $T_5 := T_2 * T_4$
6. $T_6 := prod + T_5$
7. $prod := T_6$
8. $T_7 := I_0 + 1$
9. $I_0 := T_7$
10. if $I_0 <= 20$ goto 1



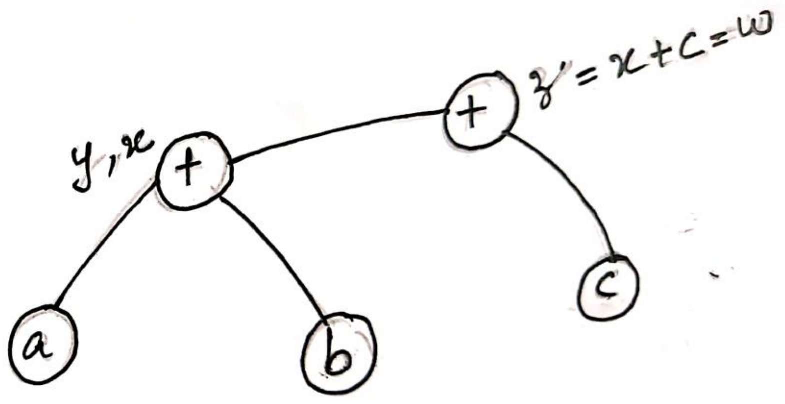
Q1. $t_1 = a+b$
$t_2 = a+b$ These two are common sub expressions.
$t_3 = t_1 * c$
$t_4 = t_2 * d$

---

$t_1 = a+b$
$t_2 = t_1 * c$
$t_3 = t_1 * d$

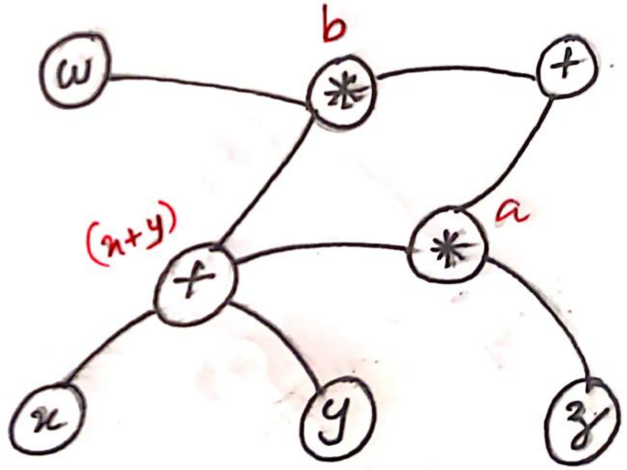


Q2 $x = a+b$
$y = a+b$
$z = x+c$
$w = y+c$



(a+b) is computed once
x+c is also computed once.

Q3. $a = (x+y) * z$
$b = (x+y) * w$
$c = a+b$



Q4. $t_1 = a * b$

$t_2 = c + d$

$t_3 = a * b$

$t_4 = t_1 + t_2$

$t_5 = t_3 + t_2$

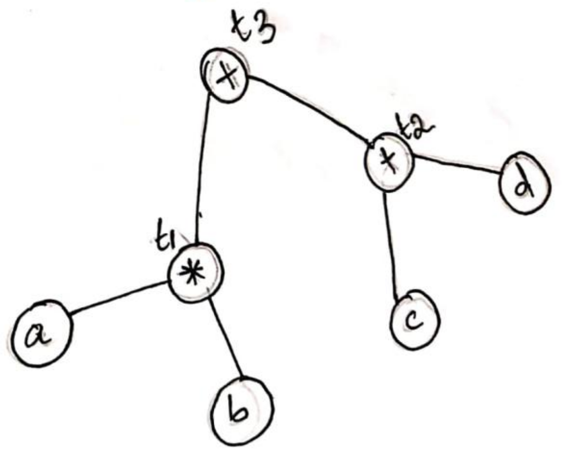
common subexpression eliminating.

$t_1 = a * b$

$t_2 = c + d$

$t_3 = t_1 + t_2$

$\longrightarrow$



Q5. $t_1 = a + b$

$t_2 = t_1 * c$

$t_3 = t_1 * d$ dead code. (is never used) Thus we can omit it.

$t_4 = t_2 + e$

$t_5 = t_1 * c$

$t_2$ & $t_5$ are common subexp.

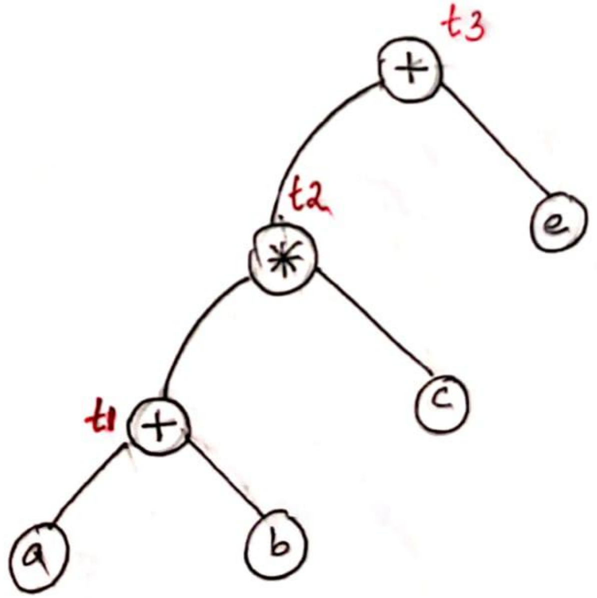
Thus :

$t_1 = a + b$

$t_2 = t_1 * c$

✓ $t_3 = t_2 + e$

~~t4~~



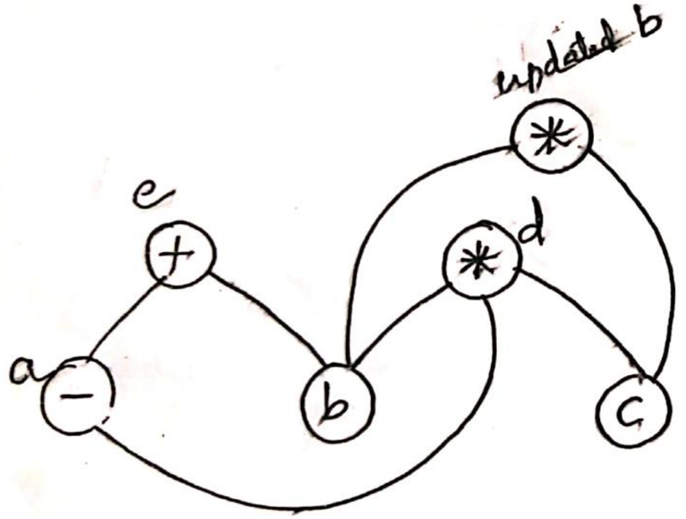
6). $d = b * c$

$e = a + b$

$b = b * c$

$a = e - d$

In the third stmt `b' gets updated . ∴ the stmts 1 & 3 are not common sub-expressions.



## Use of Algebraic identities

1. x + 0 = 0 + x = x
2. x – 0 = x
3. x * 1 = 1 * x = x
4. x / 1 = x
5. x * * 2 = x * x
6. 2 . 0 * x = x + x
7. x / 2 = x * 0 . 5

**Algebraic Transformations:**

- It represents another important class of optimizations on basic blocks.

  $x + 0 = 0 + x = x$

  $x - 0 = x$

  $x * 1 = 1 * x = x$

  $x / 1 = x$

- Another class of algebraic optimization includes reduction in strength.

  $x ** 2 = x * x$

  $2 * x = x + x$

  $x / 2 = x * 0.5$

- associative laws may also be applied to expose common subexpression.

  $a = b + c$

  $e = c + d + b$

- With the intermediate code might be

  $a = b + c$

  $t = c + d$

  $e = t + b$

- If t is not needed outside this block, the sequence can be

  $a = b + c$

  $e = a + d$

## PEEPHOLE OPTIMIZATION

- **Peephole Optimization** is a **local code optimization technique** that improves the efficiency of generated machine code by analyzing small sections of code (a "peephole") and making replacements.
- The techniques employed are:
    1. Redundant Loads and Stores
    2. Unreachable code elimination
    3. Multiple jumps
    4. Algebraic Simplification
    5. Reduction in Strength
    6. Use of machine idioms

## 1. Redundant Loads and Stores

- Removes unnecessary instructions that do not affect the program's outcome.

```
MOV  R1,  A
MOV  A,  R1
```

*Optimized code*: **MOV  R1 ,  A**

## Use of Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like `i:=i+1`.