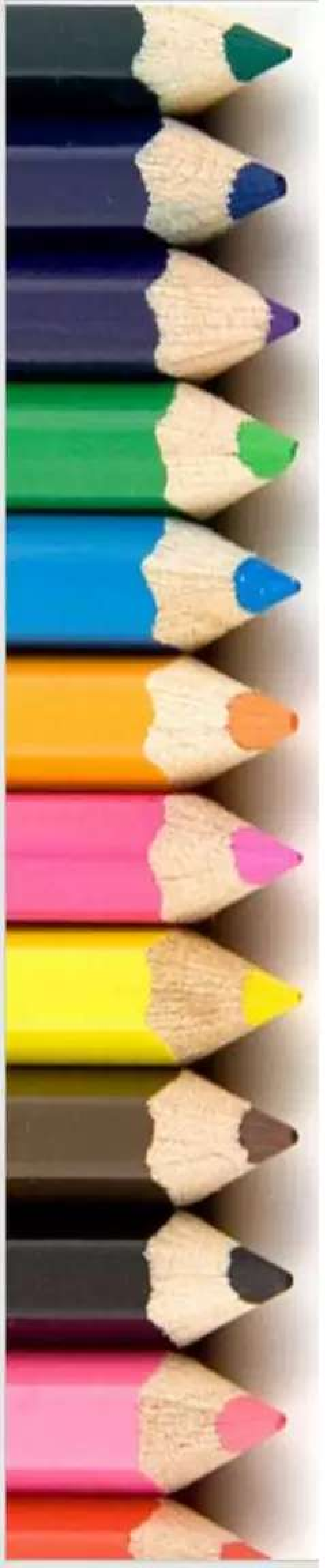


Optimization

- In computing, optimization is the process of modifying a system to make some aspect of it work more efficiently or use fewer resources.
- Ex: a computer program may be optimized so that it executes more rapidly or is capable of operating with less memory storage or other resources, or draw less power.
- The system may be a single computer program, a collection of computers or even an entire network such as the internet.

Code Optimization

- ***Optimization*** is a program transformation technique, which tries to improve the code that consume less resources (i.e. CPU, Memory) and deliver high speed.
- In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.
- Usually done at the end of the development stage since it reduces readability & adds code that is used to improve the performance.

Code Optimization

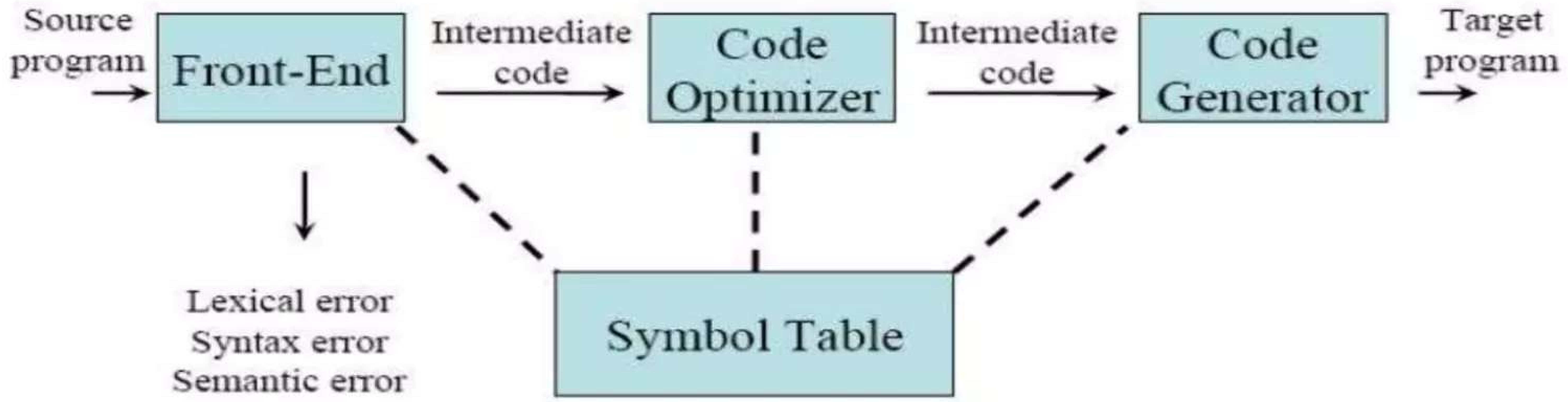
Optimized code’s features:-

- Executes faster
- Code size get reduced
- Efficient memory usage
- Yielding better performance
- Reduces the time and space complexity

Code Optimization

Optimizations are classified into two categories:

- ***Machine independent optimizations*** - improve target code without taking properties of target machine into consideration.
- ***Machine dependent optimization*** - improve target code by checking properties of target machine.



Code Optimization

Criteria for Optimization:-

- An optimization must preserve the meaning of a program:

  -Cannot change the output produced for any input.

  -Can not introduce an error.
- Optimization should, on average, speed up programs.
- Optimization should itself be fast and should not delay the overall compiling process.

Code Optimization

Optimization can occur at several levels:

1. ***Design level:-*** At the highest level, the design may be optimized to make best use of the available resources.
   - The implementation of this design will benefit from the use of suitable efficient algorithms and the implementation of these algorithms will benefit from writing good quality code.
2. ***Compile level:-*** Use of an optimizing compiler tends to ensure that the executable program is optimized at least as much as the compiler can predict

Code Optimization

Improvements can be made at various phases:-

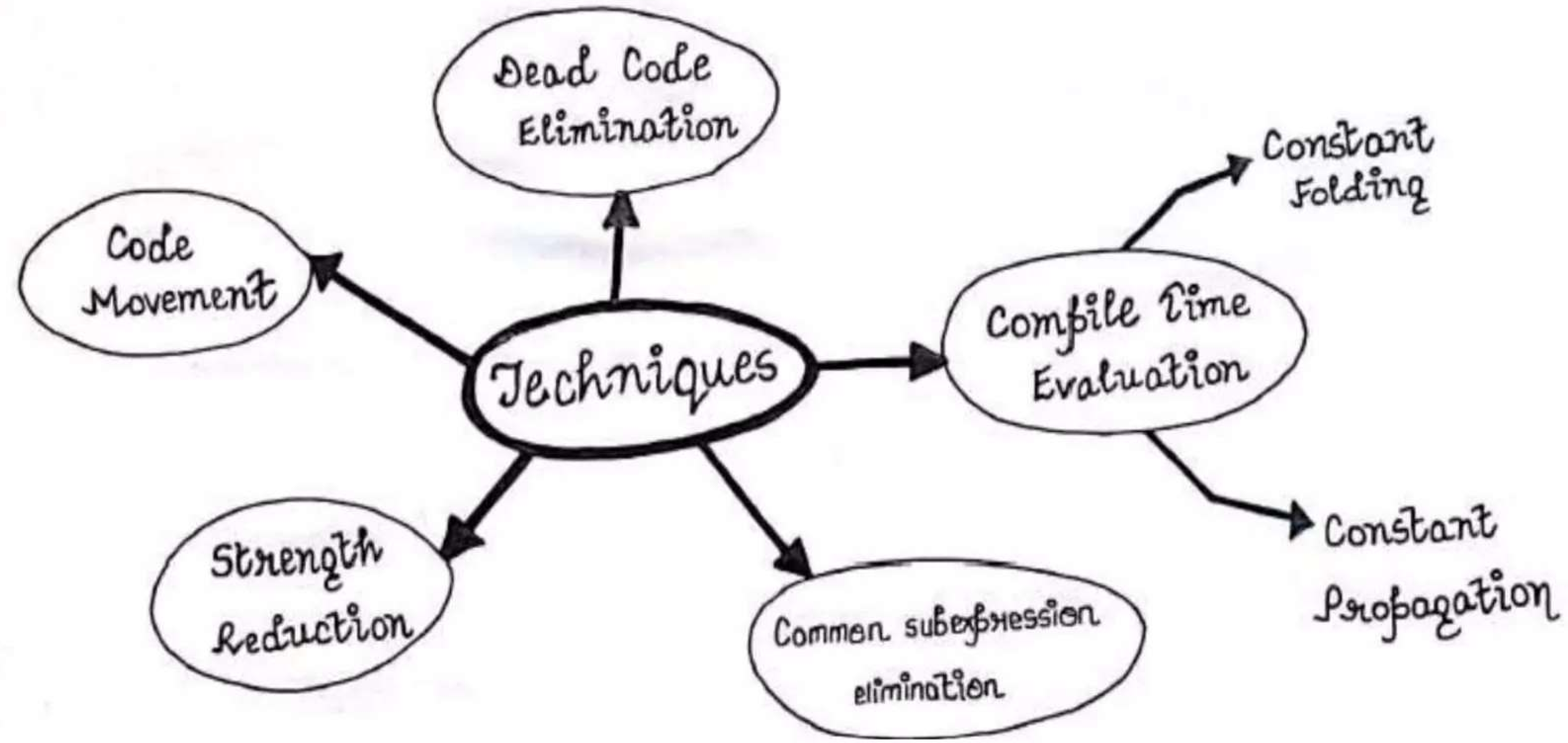
- ***Source Code***: Algorithm's transformation can produce spectacular improvements
- ***Intermediate Code***: Compiler can improve loops, procedure calls and address calculations.

  -Typically only optimizing compilers include this phase

- ***Target Code***: Compilers can use registers efficiently

Code Optimization

- Main techniques of Optimization:-



Code Optimization

There are primarily 3 types of optimizations:-

(1) Local optimization - Apply to a basic block in isolation

(2) Global optimization - Apply across basic blocks

(3) Loop optimization – Apply on loops

Another significant optimization can be:-

(4) peep-hole optimization - Apply across boundaries

Code Optimization

- Optimization performed within a basic block.
- This is simplest form of optimizations
- No need to analyze the whole procedure body.
- Just analyze the basic blocks of the procedure.
- The local optimization techniques include:
  - Constant Folding
  - Constant Propagation
  - Algebraic Simplification
  - Operator Strength Reduction
  - Dead Code Elimination

Local Code Optimization

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.
- Ex: Circumference of circle = (22/7) x Diameter
- Replace (22/7) with 3.14 during compile time and save the execution time.

Local Code Optimization

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.
- Ex: pi = 3.14, radius = 10
- Area of circle = pi x radius x radius
- Substitutes constants value during compile time and saves the execution time.

Local Code Optimization

C) Algebraic Simplification-

- Use algebraic properties to simplify expressions
- Some expressions can be simplified by replacing them with an equivalent expression that is more efficient.
- Ex: `-(-i)` $\rightarrow$ `i`

```
void f (int i)
{
  a[0] = i + 0;
  a[1] = i * 0;
  a[2] = i - i;
  a[3] = 1 + i + 1;
}
```

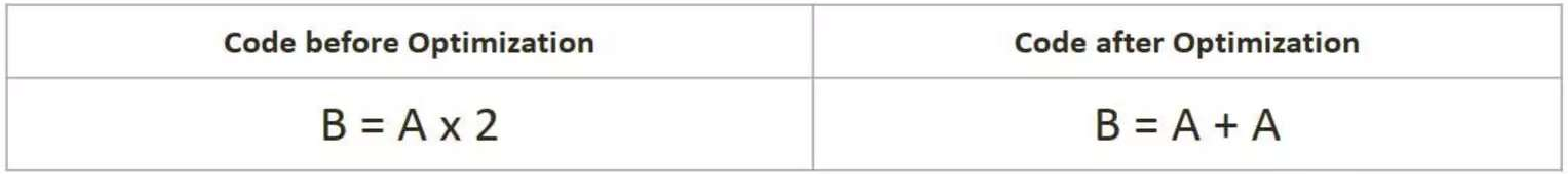
$\Rightarrow$

```
void f (int i)
{
  a[0] = i;
  a[1] = 0;
  a[2] = 0;
  a[3] = 2 + i;
}
```

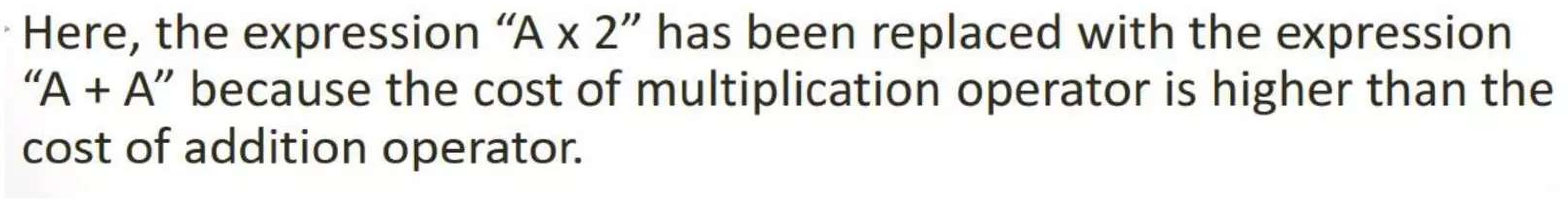
Code Optimization

D) 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



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.

Code Optimization

E) 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.

Code before Optimization	Code after Optimization

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

Code Optimization

F) Common sub-expressions elimination:

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

Code before Optimization	Code after Optimization

S1 = 4 x i
S2 = a[S1]
S3 = 4 x j
S4 = 4 x i // Redundant Expression
S5 = n
S6 = b[S4] + S5
	S1 = 4 x i
	S2 = a[S1]
	S3 = 4 x j
	S5 = n
	S6 = b[S1] + S5

Global Code Optimization

- Optimization across basic blocks within a procedure /function
- Could be restricted to a smaller scope, Example: a loop
- Data-flow analysis is done to perform optimization across basic blocks
- Each basic block is a node in the flow graph of the program.
- These optimizations can be extended to an entire control - flow graph
- Most of compiler implement global optimizations with well founded theory and practical gains

Peephole Optimization

- Optimization technique that operates on the one or few instructions at a time.
- Performs machine dependent improvements
- Peeps into a single or sequence of two to three instructions (peephole) and replaces it by most efficient alternative (shorter or faster) instructions.
- Peephole is a small moving window on the target systems

Peephole Optimization

Characteristics of peep-hole optimizations:-

- Redundant-instruction (loads and stores)elimination
- Flow-of-control optimizations - - Elimination of multiple jumps (i.e. goto statements)
- Elimination of unreachable code
- Algebraic simplifications
- Reducing operator strength
- Use of machine idioms

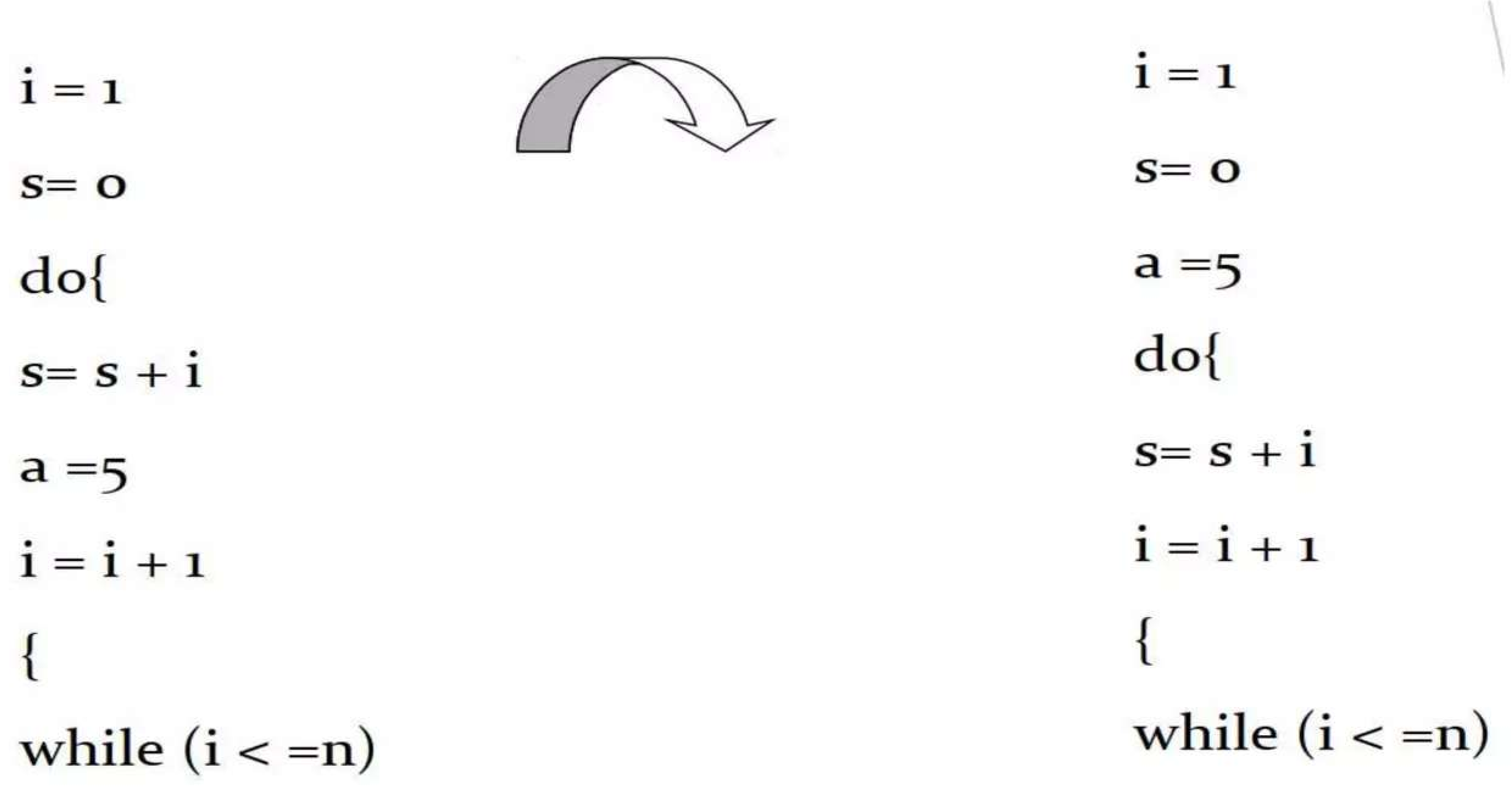
replace **Add #1,R**
by **Inc R**

Loop Optimization

- Loop optimization plays an important role in improving the performance of the source code by reducing overheads associated with executing loops.
- Loop Optimization can be done by removing:
  - Loop invariant
  - Induction variables

Loop Optimization

- Removal of Loop Invariant data:-



```
i = 1
s= 0
do{
s= s + i
a =5
i = i + 1
{
while (i < =n)
```

```
i = 1
s= 0
a =5
do{
s= s + i
i = i + 1
{
while (i < =n)
```

Bringing a=5 outside the do while loop, is called code motion.

Loop Optimization

- Induction variables:-

```
i = 1
s= 0
S1=0
S2=0
while (i < =n)
{
s= s + a[ i ]
t1 = i * 4
s= s + b[ t1 ]
t2 = t1 +2
s2= s2 + c[ t2 ]
i = i + 1
}
```

```
i = 1
s= 0
S1=0
S2=0
t2=0
while (i < =n)
{
s= s + a[ i ]
t1 = t1+ 4
s= s + b[ t1 ]
s2= s2 + c[t1 +2 ]
i = i + 1
}
```

"+" replaced " * ", t1 was made independent of i

t1,t2 are induction variables. i is inducing t1 and t1 is inducing t2