Unit IV
Code Optimization

- **Code Optimization**: Principal sources of optimization, Optimization of Basic blocks.

Code Optimization

- The term optimization in a compiler may be applied to a technique design to obtain more efficient object code than would be obtained by simple, straight forward code generators

- Criteria for Code-Improving Transformations

  1.They must be ensure that the transformed program is semantically equivalent to the original program.

  2. The improvement of the program efficiently (on the average, speed up program by measurable amount) must be achieved without changing the program code.

  3. A transformation must be worth in effort. (a compiler should not expand the effort and time to implement to code-improving transformation)

Code Optimization

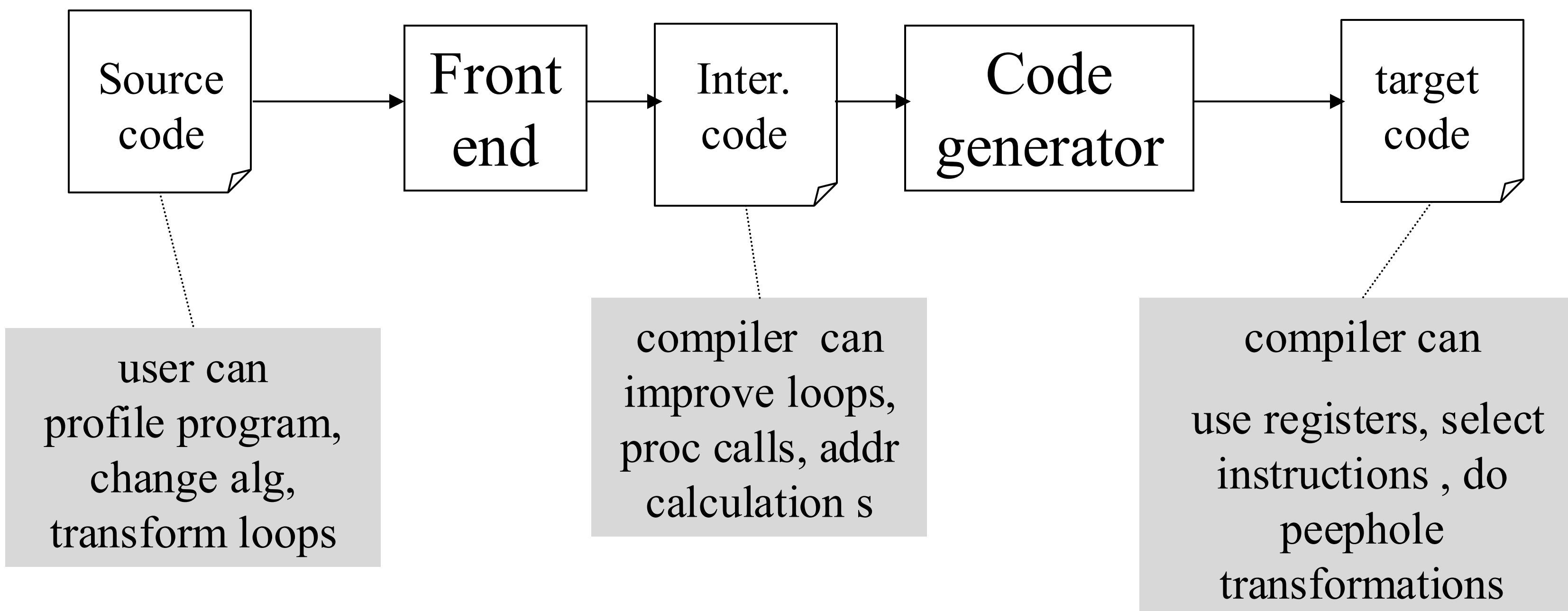
An optimization can be classified as machine dependent or machine independent

- **M/C dependent:** Exploit characteristics of the target machine , such as register allocation , addressing structures etc.

- **M/C independent:** Program transformations that improve the target code without taking into consideration any properties of the target machine.

Introduction

- Optimization can be done in almost all phases of compilation.
- Places for potential improvements by the user and the compiler.



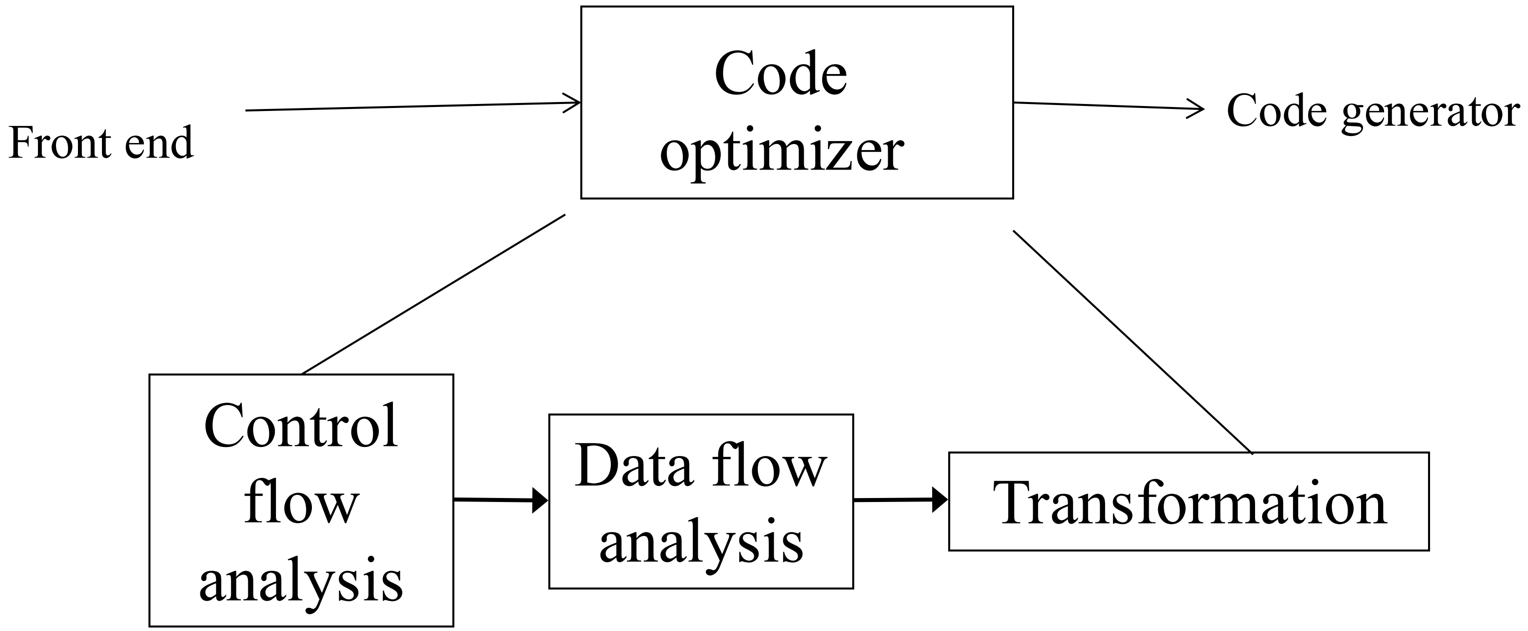
An organization for optimizing compiler

The code optimization techniques needed to analyze and transform a program code.

The code improvement phase consists of control flow and data flow analysis followed by the applications of transformations.

- A process of identifying loops is called control flow analysis
- A process of collecting information about the way variables are used in a program called data flow analysis
- The transformations are applied to intermediate code to generate efficient target code.

Organization of an code optimizer



Principal sources of optimization

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block otherwise it is global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

➢ Function-Preserving Transformations

➢ Loop optimizations

Function -Preserving Transformations are:

1. Common sub expression elimination
2. Copy propagation
3. Dead code elimination
4. Constant folding

Common sub expression elimination: An occurrence of an expression E is called a common sub expression if E was previously computed and the values of variables in E have not changed since the previous computation.

And the common sub expressions are eliminated.

Local Common sub expression elimination

B5

```
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
```

Before

B5

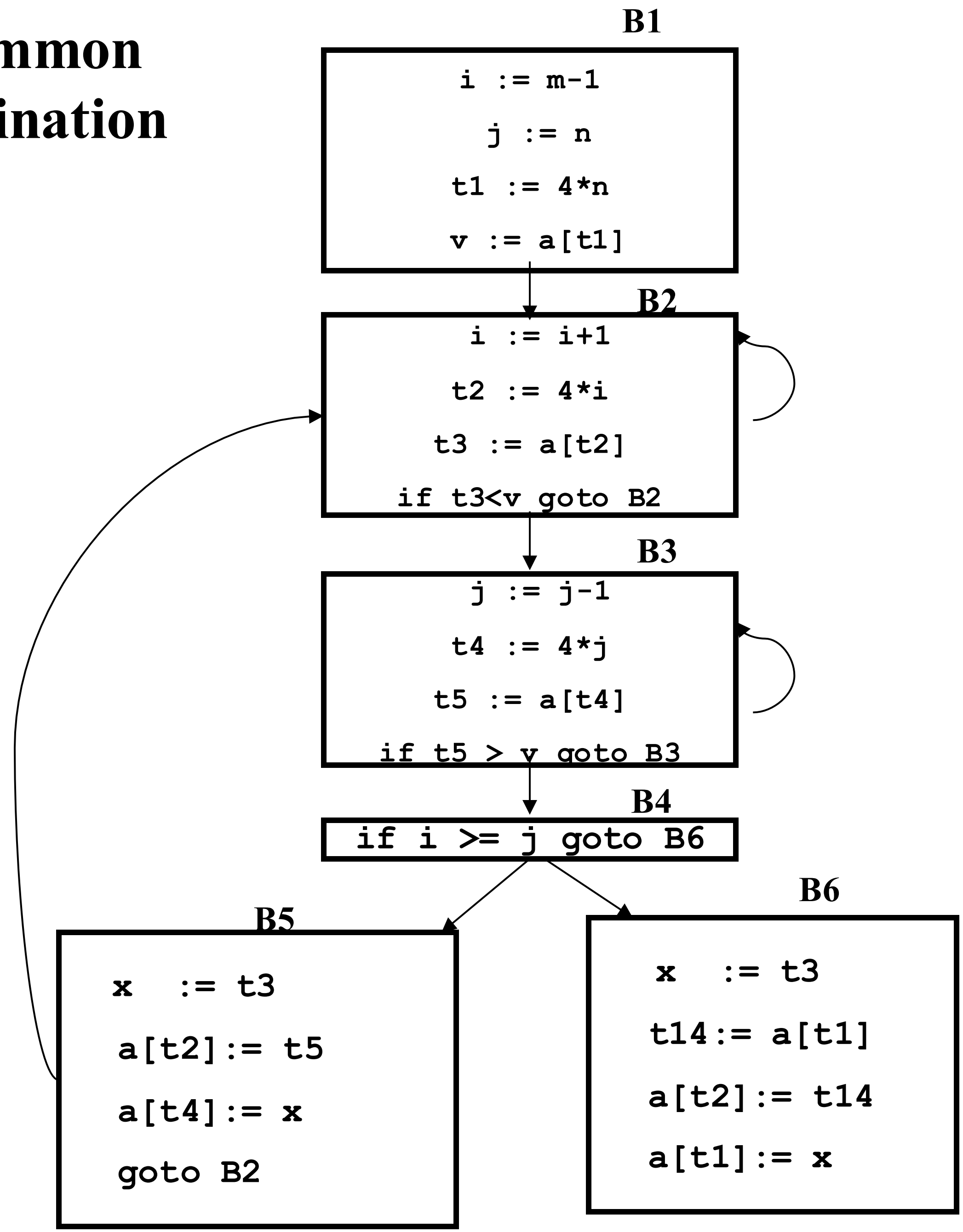
```
t6   := 4 * i
 x   := a[t6]
t8   := 4*j
t9   := a[t8]
a[t6]:= t9
a[t8]:= x
goto B2
```

After

Global and local common sub expression elimination

B5

```
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
```



Optimization Techniques (contd..)

Copy propagation: An assignment statement in the form f:=g called copy statement.

The idea behind the copy propagation is to use g for f wherever possible after the copy statement f:=g

Ex: In block B5 x:=t3 is a copy statement

After applying copy propagation to B5 yields

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

Optimization Techniques (contd..)

Dead code elimination: The code is said to be dead when the statements that compute values that never get used

Ex: After copy propagation in Block B5 the x value is never used

x:=t3 is dead code and eliminated

Constant Folding :

X=32;

X=X+32;

These statements are replaced by

X=64;

Loop Optimizations

- Three techniques are important for loop optimization

    i) Code motion

    ii) induction variable elimination

    iii) reduction in strength

Code motion: Which moves code outside a loop. This transformation takes an expression that yields the same result independent of the number times a loop is executed and places the expression before the loop. Such expression is called loop –invariant computation.

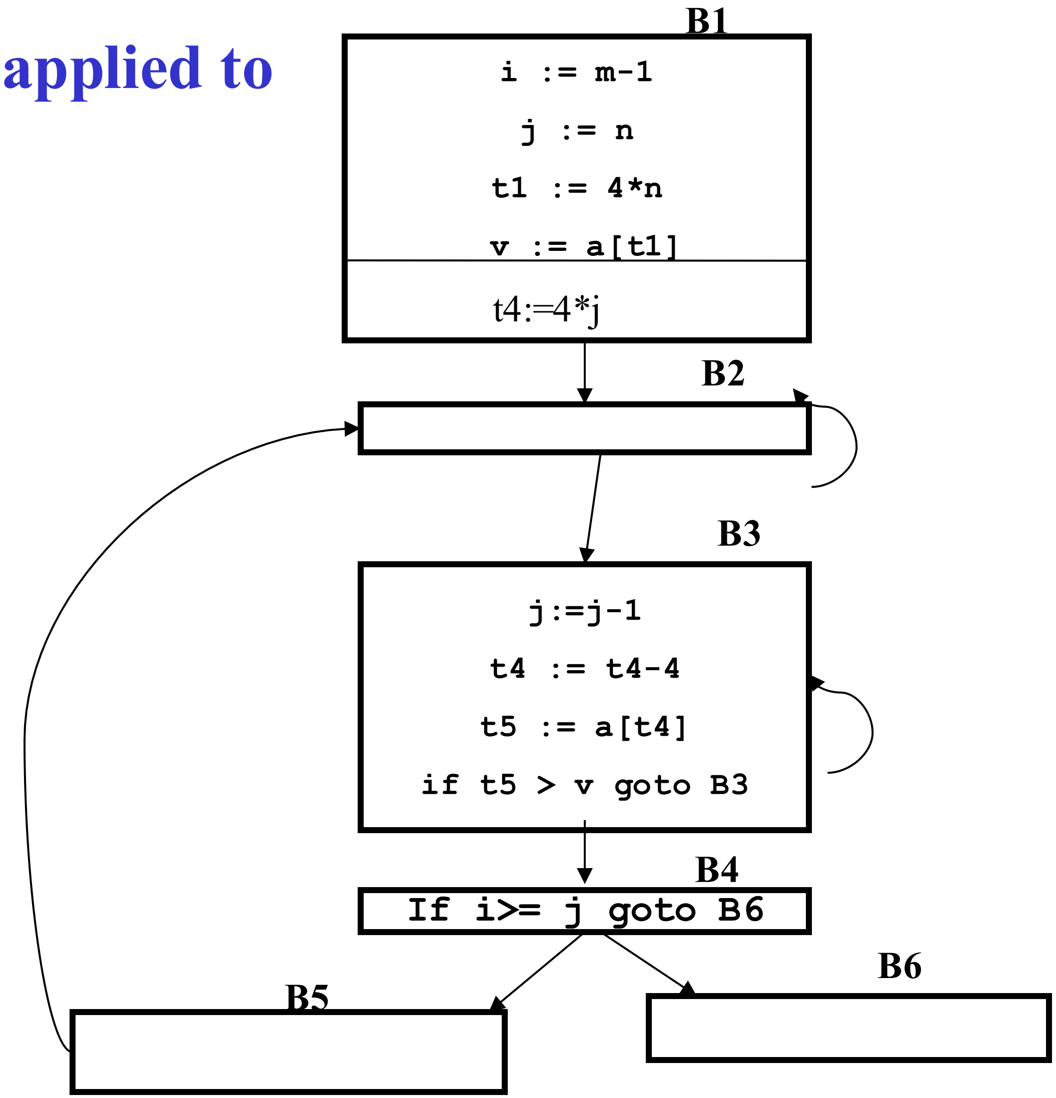
Ex: while (i<=limit-2) is replaced with

```
t= limit-2
while(i<=t)
```

Strength reduction

- Reduction in strength: replace expensive operations by cheaper ones
  - $x^2 \Rightarrow x * x$
  - fixed-point multiplication and division by a power of 2 $\Rightarrow$ shift
  - floating-point division by a constant $\Rightarrow$ floating-point multiplication by a constant

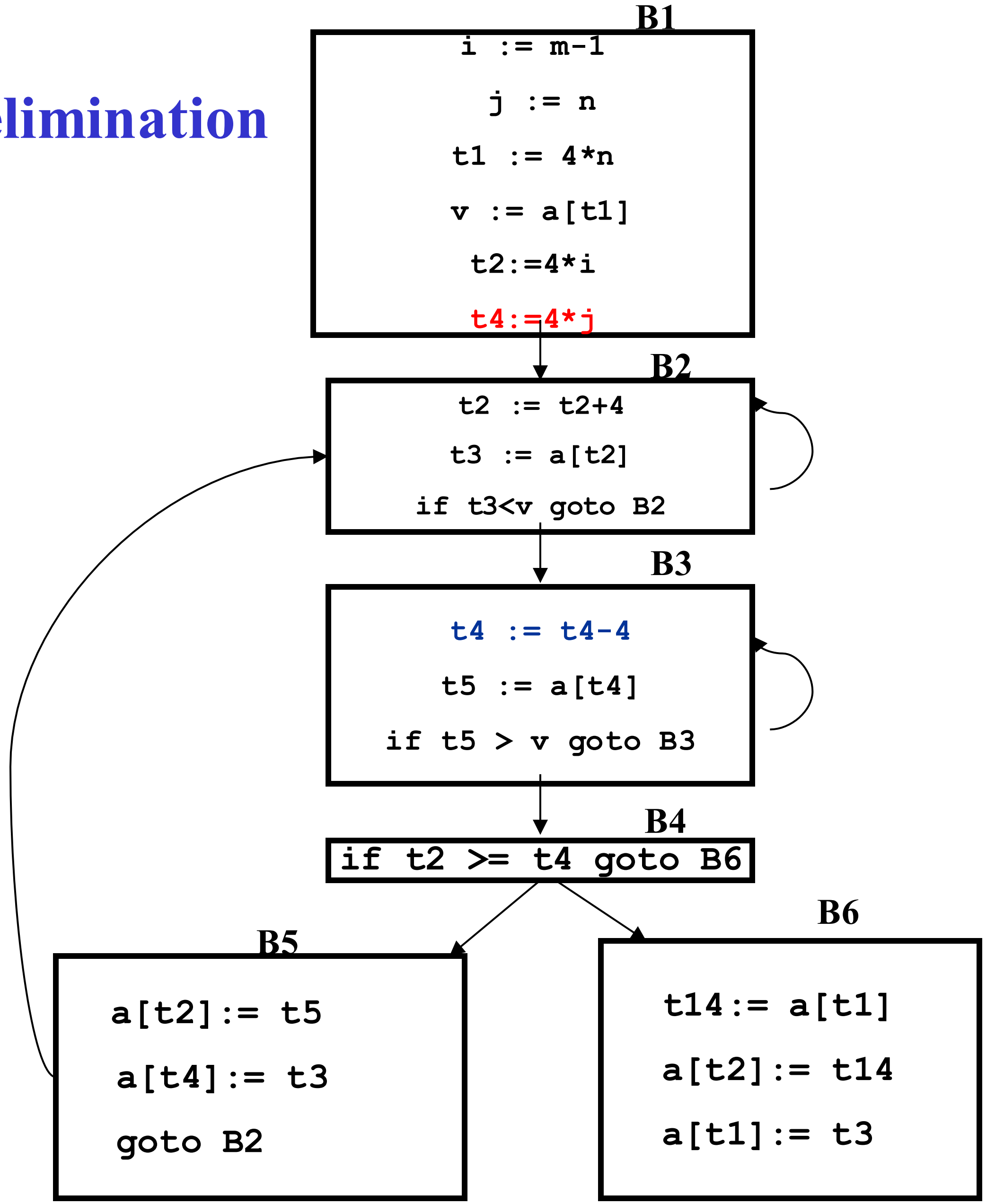
Strength reduction applied to 4*j in block B3



Flow Graph after induction-variable elimination

Induction variable elimination is used to replace variable from inner loop.
It can reduce the number of additions in a loop. It improves both code space and run time performance.

In this figure, we can replace the assignment t4:=4*j by t4:=t4-4. The only problem which will be arose that t4 does not have a value when we enter block B2 for the first time. So we place a relation t4=4*j on entry to the block B2.



Loop Unrolling: -

- For Example: -

```
int i = 1;
While ( i <=100 )
{
        a[i] = b[i];
        i++;
}
```

- Can be written as

```
int i = 1;
While ( i <=100 )
{
        a[i] = b[i];
        i++;
        a[i] = b[i];
        i++;
}
```

Loop Fusion or Loop Distribution

- **Loop fission** (or **loop distribution**) is a compiler optimization in which a loop is broken into multiple loops over the same index range with each taking only a part of the original loop's body. The goal is to break down a large loop body into smaller ones to achieve better utilization of locality of reference. This optimization is most efficient in multi-core processors that can split a task into multiple tasks for each processor.

```c
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
    a[i] = 1;
    b[i] = 2;
}
```

is equivalent to

```c
int i, a[100], b[100];
for (i = 0; i < 100; i++) {
    a[i] = 1;
}
for (i = 0; i < 100; i++) {
    b[i] = 2;
}
```

Loop Fusion or Loop Jamming

It is a compiler optimization and loop transformation which replaces multiple loops with a single one.

Loop Fusion: -

- For Example: -

  for i := 1 to n do

  for j := 1 to m do

  a[i, j] := 10

- Can be written as: -

  for i := 1 to n*m do

  a[i] := 10

Loop Invariant Method: -

- In this optimization technique the computation inside the loop is avoided and there by the computation overhead on compiler is avoided.
- This ultimately optimizes code generation.

Loop Invariant Method: -

- For Example: -

```
for i := 0 to 10 do begin
K = i + (a/b);
...
...
end;
```

- Can be written as

```
t := a/b;
for i := 0 to 10 do begin
K = i + t;
...
...
end;
```

Optimization of Basic Block

- A basic block computes a set of expressions.
- Transformations are useful for improving the quality of code.
- Two important classes of local optimizations that can be applied to a basic blocks
  - Structure Preserving Transformations
    1. Common sub expression elimination
    2. Dead-code elimination
    3. Renaming Temporary Variables
    4. Interchange of Statements
  - Algebraic Transformations

Structure Preserving Transformations

- **Common sub-expression elimination**

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

same

b is redefined

Structure Preserving Transformations contd.

- **Dead – Code Elimination**

  Say, x is dead, that is never subsequently used, at the point where the statement $x = y + z$ appears in a block.

  We can safely remove x

- **Renaming Temporary Variables**
  - say, $t = b+c$ where t is a temporary var.
  - If we change $u = b+c$, then change all instances of t to u.

- **Interchange of Statements**
  - $t_1 = b + c$
  - $t_2 = x + y$
  - We can interchange if neither x nor y is t_1 and neither b nor c is t_2

Algebraic Transformations

- Replace expensive expressions by cheaper one
  - $X = X + 0$ eliminate
  - $X = X * 1$ eliminate
  - $X = y**2$ (why expensive? Answer: Normally implemented by function call)
    - by $X = y * y$