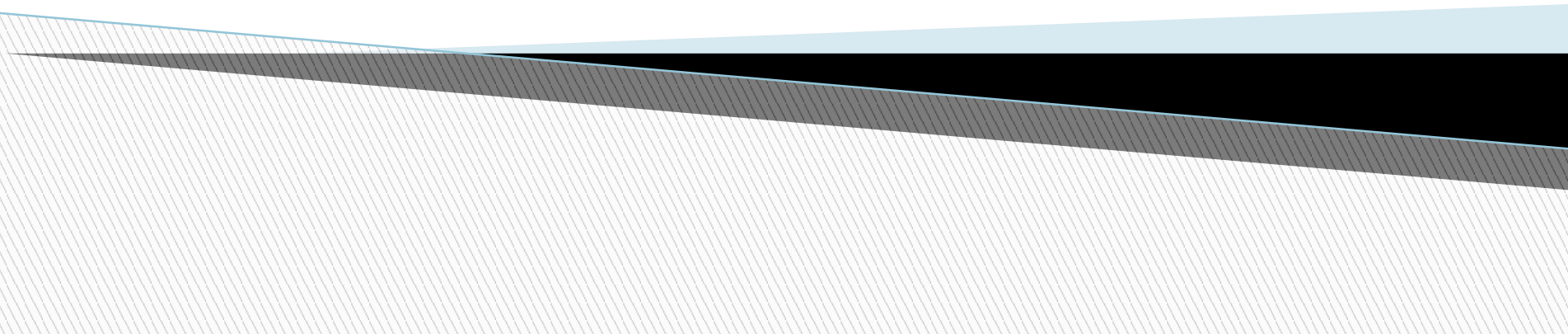
Code Optimization



- 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.
- In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.

- code optimizing process must follow the three rules given below:
- 1. The output code must not, in any way, change the meaning of the program.
- 2. Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- 3. Optimization should itself be fast and should not delay the overall compiling process.

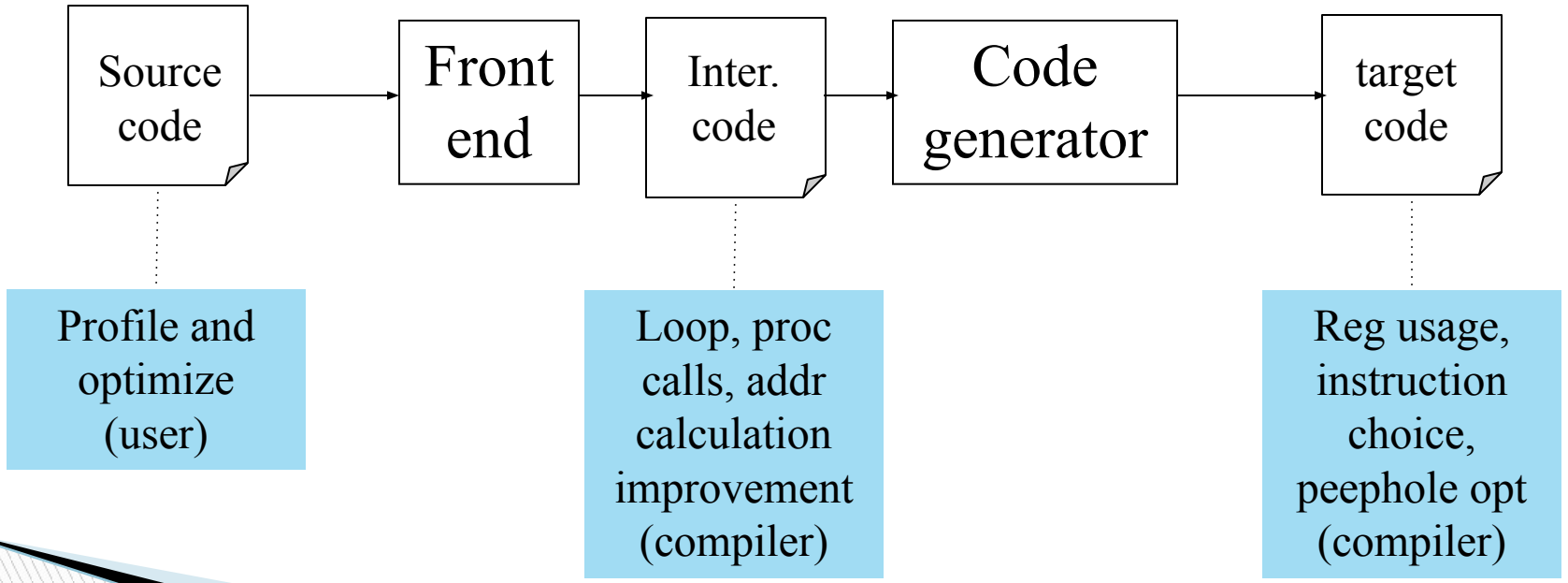
- Efforts for an optimized code can be made at various levels of compiling the process.
- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Basic Blocks

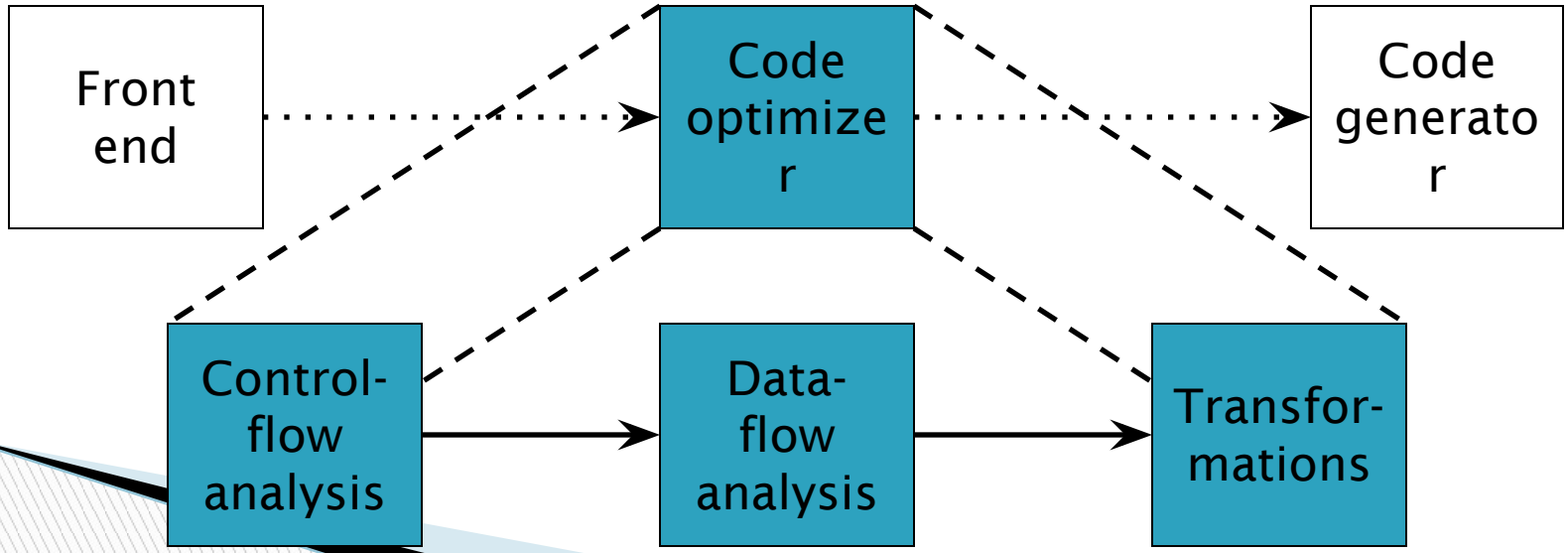
- Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code.
- These basic blocks do not have any jump statements among them,
- i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

Places for potential improvements by the user and compiler

- Optimization can be done in almost all phases of compilation.



The Code Optimizer



PRINCIPAL SOURCES OF OPTIMIZATION(UQ 15 marks)

- 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 called **global**.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- Function preserving transformations examples:
- Common sub expression elimination
- Copy propagation,
- Dead-code elimination
- Constant folding

Common Sub expressions elimination:

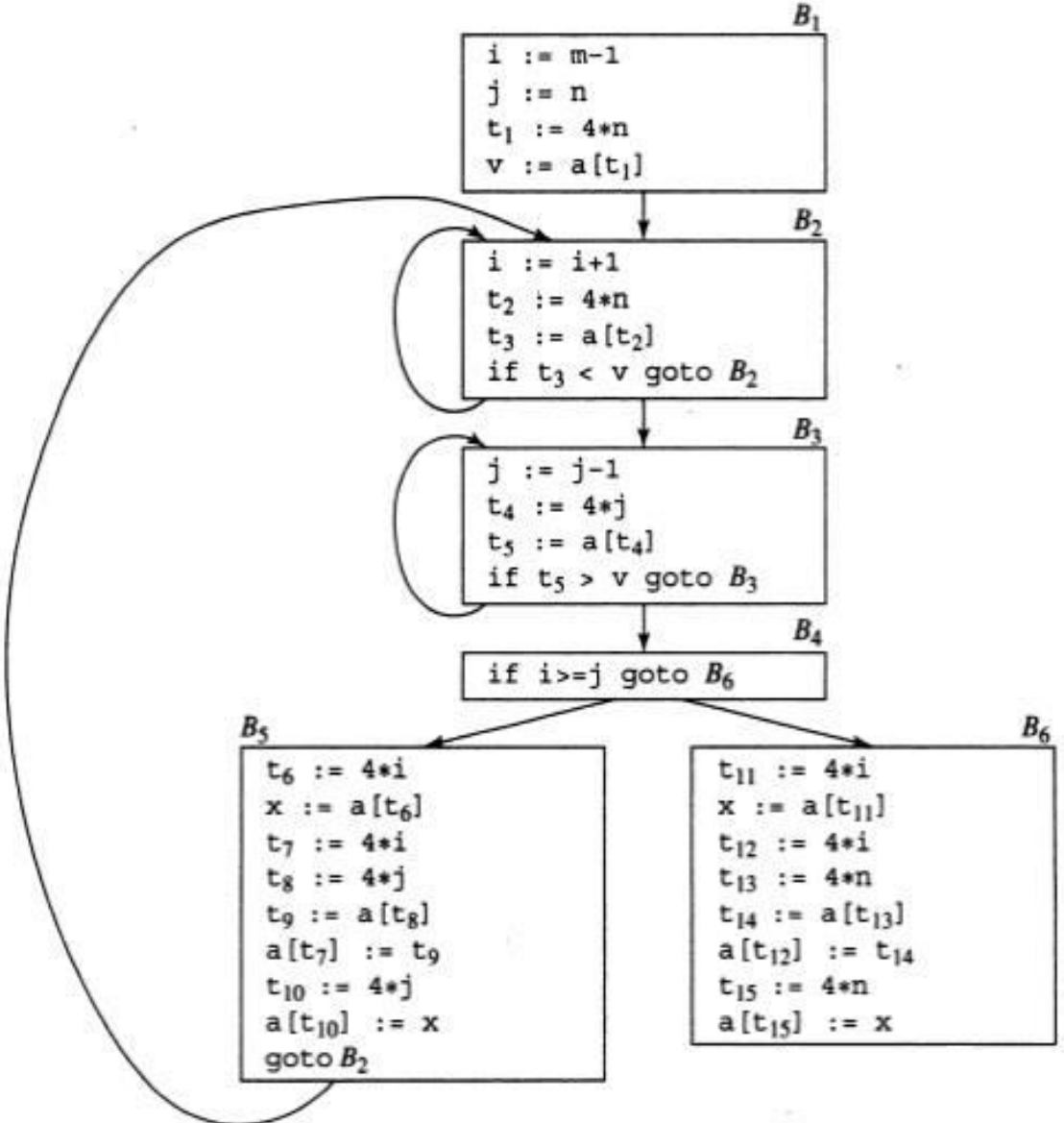
- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.
- We can avoid recomputing the expression if we can use the previously computed value.

- For example
-
- **t1: = 4*i**
- **t2: = a [t1]**
- **t3: = 4*j**
- **t4: = 4*i**
- **t5: = n**
- **t6: = b [t4] +t5**

- The above code can be optimized using the common sub-expression elimination as
- **t1: = 4*i**
- **t2: = a [t1]**
- **t3: = 4*j**
- **t5: = n**
- **t6: = b [t1] +t5**

- The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

Flow Graph


B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
B2
i := i+1
t2 := 4*n
t3 := a[t2]
if t3 < v goto B2
B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
B4
if i>=j goto B6
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
B6
t11 := 4*i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x

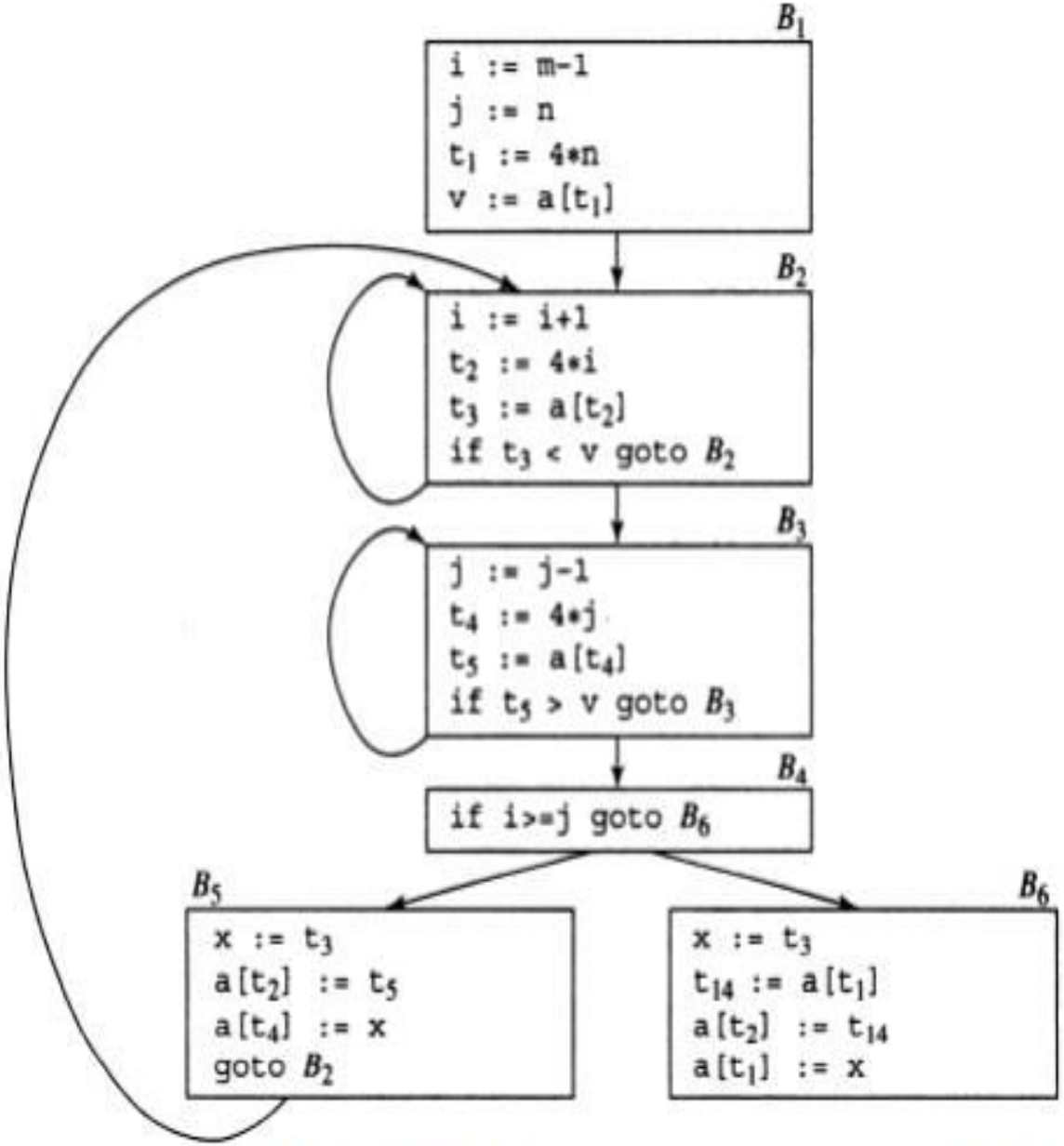


Fig. 5.3 B5 and B6 after common subexpression elimination

Copy Propagation:

- Assignments of the form f : = g called copy statements, or copies for short.
- The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g.
- Copy propagation means use of one variable instead of another.
- This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

Copy Propagation:

- · For example:
- x=Pi;
- A=x*r*r;
- The optimization using copy propagation can be done as follows: A=Pi*r*r;
- Here the variable x is eliminated

Dead-Code Eliminations:

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
- A related idea is dead or useless code, statements that compute values that never get used.
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

Dead-Code Eliminations:

- Example:

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

- Here, ‘if’ statement is dead code because this condition will never get satisfied.

Constant folding:

- Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.
- For example,
- a=3.14157/2 can be replaced by
- a=1.570 there by eliminating a division operation.

Loop Optimizations:

- In loops, especially in the inner loops, programs tend to spend the bulk of their time.
- The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

- Three techniques are important for loop optimization:
- 1. Code motion, which moves code outside a loop;
- 2. Induction-variable elimination, which we apply to replace variables from inner loop.
- 3. Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion:

- An important modification that decreases the amount of code in a loop is code motion.
- This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.
- Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

Code Motion:

- while (i <= limit-2) /* statement does not change limit*/
- Code motion will result in the equivalent of
- t= limit-2;
- while (i<=t) /* statement does not change limit or t */

Induction Variables :

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t4 remain in lock-step;
- every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4.
- Such identifiers are called induction variables.

- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination.
- For the inner loop around B3 in Fig. we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

- However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

- Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4.

- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction In Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

Reduction In Strength:

- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine.
- Fixed-point multiplication or division by a power of two is cheaper to implement as a shift.
- Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

PEEPHOLE OPTIMIZATION(6 marks UQ)

- A statement-by-statement code-generations strategy often produces target code that contains redundant instructions and suboptimal constructs.
- The quality of such target code can be improved by applying “optimizing” transformations to the target program.

- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

- The peephole is a small, moving window on the target program.
- The code in the peephole need not be contiguous, although some implementations do require this.
- It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

- **Characteristics of peephole optimizations:**
- Redundant-instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable code

OPTIMIZATION OF BASIC BLOCKS

- There are two types of basic block optimizations. They are :
  1. Structure-Preserving Transformations
  2. Algebraic Transformations

1.Structure-Preserving Transformations:

- The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

Common sub-expression elimination:

- Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced.

- Example:

- **a: =b+c**
- **b: =a-d**
- **c: =b+c**
- **d: =a-d**

- The 2nd and 4th statements compute the same expression: b+c and a-d
- Basic block can be transformed to
- **a: = b+c**
- **b: = a-d**
- **c: = a**
- **d: = b**

Dead code elimination:

- It is possible that a large amount of dead (useless) code may exist in the program.
- This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program - once declared and defined, one forgets to remove them in case they serve no purpose.
- Eliminating these will definitely optimize the code.

Renaming of temporary variables:

- A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
- In this a basic block is transformed to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

- Two statements
- **t1:=b+c**
- **t2:=x+y**
- can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

2. Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks.
- This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values.
- Thus the expression 2*3.14 would be replaced by 6.28.