Code Optimization

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance

Objectives:

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Types of Code Optimization

1. **Machine Independent Optimization:** This code optimization phase attempts to improve the intermediate code to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations. depends on programming language
2. **Machine Dependent Optimization**: Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. depends on architecture of machine , number of cpu registers involved etc

Phases of Optimization:

There are generally two phases of optimization:

1. **Global Optimization:**
   Transformations are applied to large program segments that include functions, procedures and loops.
2. **Local Optimization:**
   Transformations are applied to small blocks of statements. The local optimization is done prior to global optimization.

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 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.

Examples:

- Common sub expression elimination

- Copy propagation,
- Dead-code elimination
- Constant folding

1. 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 re computing the expression if we can use the previously computed value.

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.

2. 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 f.

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

3. 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.

Example:

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

Here, 'if' statement is dead code because this condition will never get satisfied.

4. 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.

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:

- Code motion, which moves code outside a loop;
- Induction-variable elimination, which we apply to replace variables from inner loop.
- Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

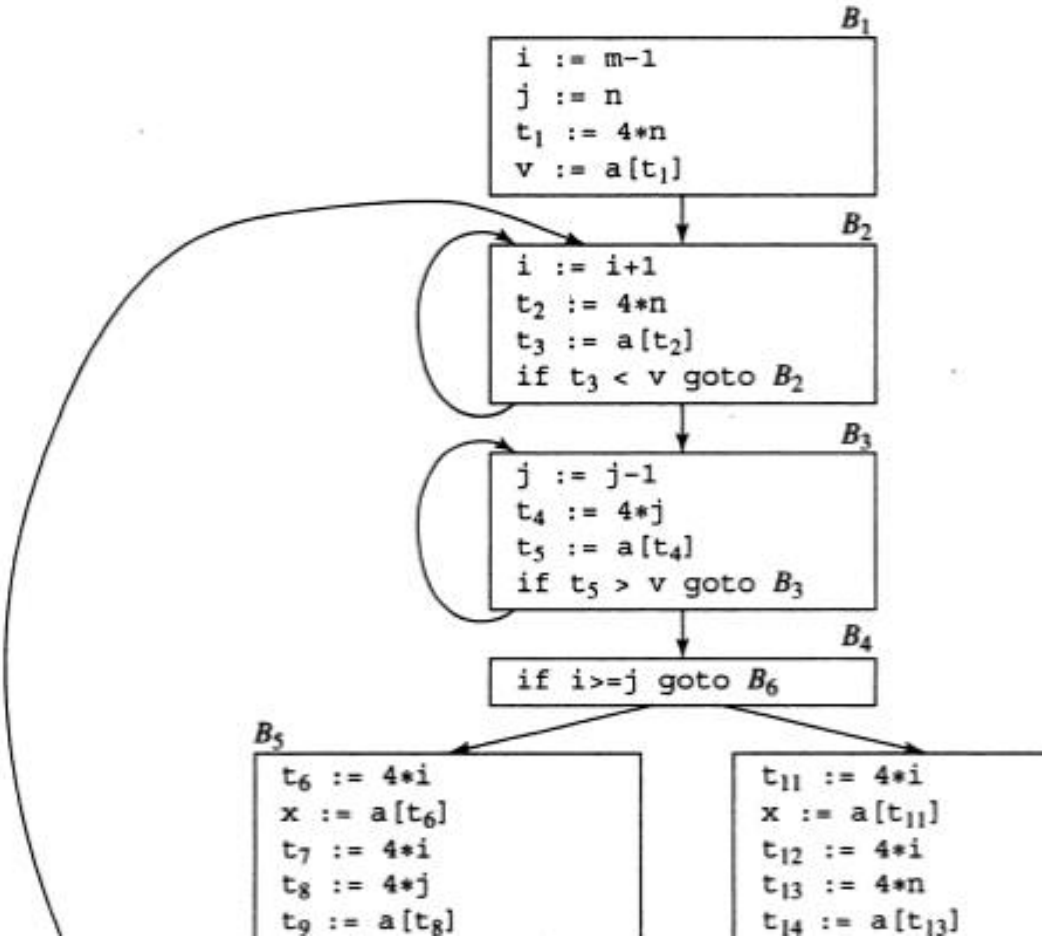


Fig. 5.2 Flow graph

- 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:

Example:

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 */

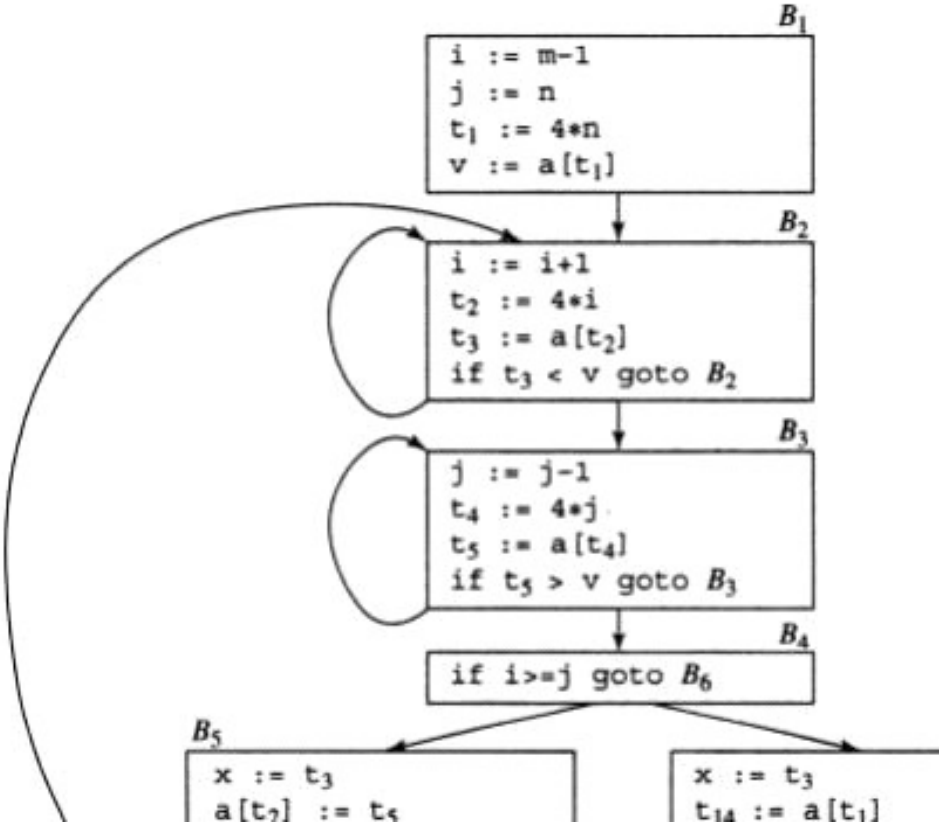
- **Induction Variables :**

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.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

- **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. For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine.



Code Generation

Issues in the Design of a Code Generator:

1. Input to the Code Generator
2. The Target Program
3. Instruction Selection
4. Register Allocation
5. Evaluation Order

While the details are dependent on the specifics of the intermediate representation, the target language, and the run-time system, tasks such as instruction selection, register allocation and assignment, and instruction ordering are en-countered in the design of almost all code generators.

The most important criterion for a code generator is that it produce cor-rect code. Correctness takes on special significance because of the number of special cases that a code generator might face. Given the premium on correct-ness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

1. **Input to the Code Generator:**

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the Intermediate Representation (IR).

The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as byte codes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's. The front end has scanned, parsed, and translated the source program into a relatively low-level IR, so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate, such as integers and floating-point numbers. All syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type-conversion operators have been inserted wherever necessary. The code generator can therefore proceed on the assumption that its input is free of these kinds of errors.

2. **The Target Program:**

The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.
A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

3. **Instruction Selection:**

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by factors such as the level of the IR, nature of the instruction-set architecture and the desired quality of the generated code.

For example, every three-address statement of the form x = y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD  R0, y        // R0 = y        (load y into regis
ADD R0, R0, z    // R0 = R0 + z   (add z to R0)
ST  x, R0        // x = R0        (store R0 into x)
```

This strategy often produces redundant loads and stores. F
sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

```
LD  R0, b        // R0 = b
ADD R0, R0, c    // R0 = R0 + c
ST  a, R0        // a = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if a is not subsequently used.

4. **Register Allocation:**

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

5. **Evaluation Order:**

The order in which computations are performed can affect the efficiency of the target code. As we shall see, some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator

The Target Language:

- A Simple Target Machine Model
- Program and Instruction Costs

1. A Simple Target Machine Model

Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte-addressable machine with n general-purpose registers, R0,R1, ... ,Rn - 1. Most instructions consist of an operator, followed by a target, followed by a list of source operands. A label may precede an instruction.

Instructions:

• **Load operations**: The instruction LD dst, addr loads the value in location addr into location dst This instruction denotes the assignment dst = addr. The most common form of this instruction is LD r, x which loads the value in location x into register r.

• **Store operations:** The instruction ST x, r stores the value in register r into the location x. This instruction denotes the assignment x = r.

• **Computation operations:** Computation operations of the form OP dst, src1,src2, where OP is an operator like ADD or SUB, and dst, src1, and src2 are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by OP to the values in locations src1 and src2, and place the result of this operation in location dst. For example, SUB n,r2,r3 computes n = r2 - r3. .

• **Unconditional jumps**: The instruction BR L causes control to branch to the machine instruction with label L. (BR stands for branch.)

• **Conditional jumps**: Conditional jumps of the form Bcond r, L, where r is a register, L is a label, and cond stands for any of the common tests on values in the register r. For example, BLTZ r, L causes a jump to label L if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

Example: The three-address statement x = y - z can be implemented by the machine instructions:

```
LD    Rl,    Y      // Rl = y
LD    R2,    Z      // R2 = z
SUB Rl, Rl, R2   // Rl = Rl - R2
ST    x,     Rl     / /  x  =  Rl
```

Basic Blocks and Flow Graphs:

This section introduces a graph representation of intermediate code that is helpful for discussing code generation even if the graph is not constructed explicitly by a code-generation algorithm. The representation is constructed as follows:

Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the properties that the flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block. Control will leave the block without halting or branching, except possibly at the last instruction in the block.

The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

1. Basic Blocks:

Our first job is to partition a sequence of three-address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction. In the absence of jumps and labels, control proceeds sequentially from one instruction to the next.

Partitioning three-address instructions into basic blocks:

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are leaders, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
   Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
```

First, instruction 1 is a leader by rule (1) of Algorithm 8.5. To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17. By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively. Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12. Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

We conclude that the leaders are instructions 1, 2, 3, 10, 12, and 13. The basic block of each leader contains all the instructions from itself until just before the next leader. Thus, the basic

block of 1 is just 1, for leader 2 the block is just 2. Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12, and instruction 13's block is 13 through 17.

2. Flow Graphs:

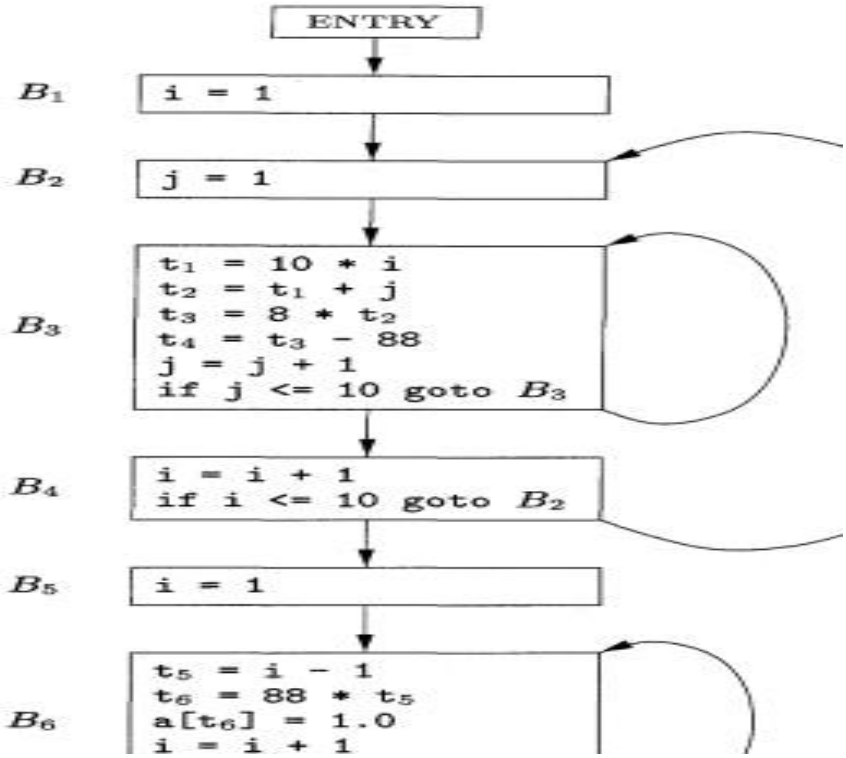
Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B.

Example:

The entry points to basic block B1, since B1 contains the first instruction of the program. The only successor of B1 is B2, because B1 does not end in an unconditional jump, and the leader of B2 immediately follows the end of B1.

Block B3 has two successors. One is itself, because the leader of B3, instruction 3, is the target of the conditional jump at the end of 3, instruction 9. The other successor is B4, because control can fall through the conditional jump at the end of B3 and next enter the leader of B4.

Only B6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B6.



A Simple Code Generator:

Here we shall consider an algorithm that generates code for a single basic block. It considers each three-address instruction in turn, and keeps track of what values are in what registers so it can avoid generating unnecessary loads and stores.

1. Register and Address Descriptors:

Our code-generation algorithm considers each three-address instruction in turn and decides what loads are necessary to get the needed operands into registers. After generating the loads, it generates the operation itself. Then, if there is a need to store the result into a memory location, it also generates that store.

In order to make the needed decisions, we require a data structure that tells us what program variables currently have their value in a register, and which register or registers, if so. We also need to know whether the memory location for a given variable currently has the proper value for that variable, since a new value for the variable may have been computed in a register and not yet stored. The desired data structure has the following descriptors:

For each available register, a register descriptor keeps track of the variable names whose current value is in that register. Since we shall use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty. As the code generation progresses, each register will hold the value of zero or more names.

For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location, or some set of more than one of these. The information can be stored in the symbol-table entry for that variable name.

2. The Code-Generation Algorithm:

An essential part of the algorithm is a function getReg(I), which selects registers for each memory location associated with the three-address instruction I. Function getReg has access to the register and address descriptors for all the variables of the basic block, and may also have access to certain useful data-flow information such as the variables that are live on exit from the block.

In a three-address instruction such as x = y + z, we shall treat + as a generic operator and ADD as the equivalent machine instruction. We do not, therefore, take advantage of commutativity of +. Thus, when we implement the operation, the value of y must be in the second register mentioned in the ADD instruction, never the third. A possible improvement to the algorithm is to generate code for both x = y + z and x = z + y whenever + is a commutative operator, and pick the better code sequence.

Peephole Optimization:

While most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program. The term "optimizing" is somewhat misleading because there is no guarantee that the resulting code is optimal under any mathematical measure. Nevertheless, many simple transformations can significantly improve the running time or space requirement of the target program.

A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

The peephole is a small, sliding window on a program. The code in the peephole need not be contiguous, although some implementations do require this.

Characteristics of peephole optimizations:

- Eliminating Redundant Loads and Stores
- Eliminating Unreachable Code
- Flow-of-Control Optimizations
- Algebraic Simplification and Reduction in Strength
- Use of Machine Idioms

1. Eliminating Redundant Loads and Stores:

Example:

```
LD a, RO
ST RO,  a
```

in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been loaded into register RO. Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. Put another way, the two instructions have to be in the same basic block for this transformation to be safe.

2. Eliminating Unreachable Code:

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain code fragments that are executed only if a variable debug is equal to 1. In the intermediate representation, this code may look like

```
 if debug == 1 goto LI
goto L2

L1: print debugging information
L2:
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of debug, the code sequence above can be replaced by

```
        if debug != 1 goto L2
        print debugging informat
L2:
```

3. Flow-of-Control Optimizations:

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

```
        goto L1
        ...
L1:     goto L2
```

by the sequence

```
        goto L2
```

If there are now no jumps to L1, then it may be possible to eliminate the statement L1: goto L2 provided it is preceded by an unconditional jump.

Similarly, the sequence

if a < b goto LI

LI: goto L2

Can be replaced by the sequence

if a < b goto L2

4. Algebraic Simplification and Reduction in Strength:

The algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as

x = x + 0

or

x = x * 1

Similarly, reduction-in-strength transformations can be applied in the peep-hole to replace 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.

5. 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 the 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 x = x + 1.

Optimization of Basic Blocks:
Optimization process can be applied on a basic block. While optimization, we don't need to change the set of expressions computed by the block.

There are two type of basic block optimization. These are as follows:
- Structure-Preserving Transformations
- Algebraic Transformations

1. Structure preserving transformations:
The primary Structure-Preserving Transformation on basic blocks is as follows:
- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements

(a) Common sub-expression elimination:
In the common sub-expression, you don't need to be computed it over and over again. Instead of this you can compute it once and kept in store from where it's referenced when encountered again.
Example:
a: = b + c
b : = a - d
c : = b + c
d : = a - d
In the above expression, the second and forth expression computed the same expression. So the block can be transformed as follows:
a : = b + c
b : = a - d
c : = b + c
d : = b

(b) Dead-code elimination:
It is possible that a program contains a large amount of dead code. This can be caused when once declared and defined once and forget to remove them in this case they serve no purpose.
Suppose the statement x:= y + z appears in a block and x is dead symbol that means it will never subsequently used. Then without changing the value of the basic block you can safely remove this statement.

(c) Renaming temporary variables:
A statement t:= b + c can be changed to u:= b + c where t is a temporary variable and u is a new temporary variable. All the instance of t can be replaced with the u without changing the basic block value.
(d) Interchange of statement:
Suppose a block has the following two adjacent statements:
t1 : = b + c
t2 : = x + y
These two statements can be interchanged without affecting the value of block when value of t1 does not affect the value of t2.

2. Algebraic transformations:
In the algebraic transformation, we can change the set of expression into an algebraically equivalent set. Thus the expression x:= x + 0 or x:= x *1 can be eliminated from a basic block without changing the set of expression.
Constant folding is a class of related optimization. Here at compile time, we evaluate constant expressions and replace the constant expression by their values. Thus the expression 5*2.7 would be replaced by13.5.
Sometimes the unexpected common sub expression is generated by the relational operators like <=, >=, <, >, +, = etc.
Sometimes associative expression is applied to expose common sub expression without changing the basic block value. if the source code has the assignments
a:= b + c
e:= c +d +b
The following intermediate code may be generated:
a:= b + c
t:= c +d
e:= t + b