Menoufia University

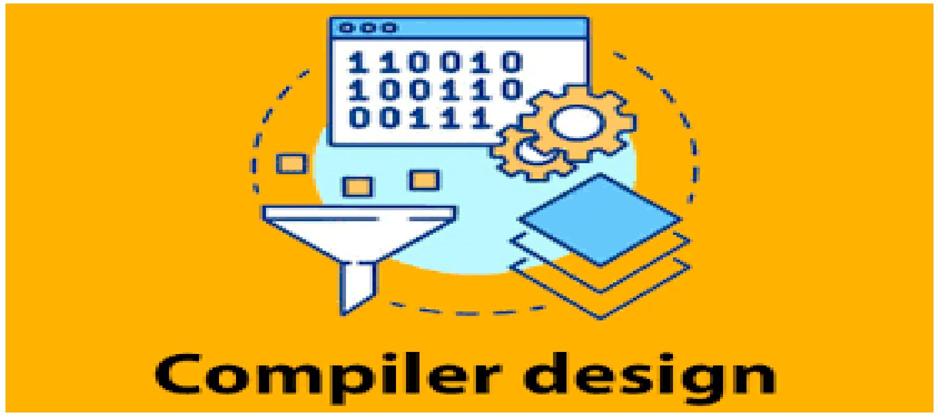
Faculty of computers & Information

Computer Science Department.

Compiler Design

4 Year – first Semester

Lecture 10



DR. Eman Meslhy Mohamed

Lecturer at Computer Science department

2023-2024

Optimization

- It is the process of improving generated code so as to reduce running time and/or reduce the space required to store it in memory.

- Software designers are often faced with decisions which involve a space-time tradeoff

- i.e., one method will result in a faster program, another method will result in a program which requires less memory.

Optimization

- However, many optimization techniques are capable of improving the object program in both time and space, which is why they are employed in most modern compilers.

- Optimization techniques can be separated into two general classes: (local and global)

- ***Local optimization*** techniques normally are concerned with transformations on small sections of code (involving only a few instructions) and generally operate on the **machine language instructions** which are produced by the code generator.

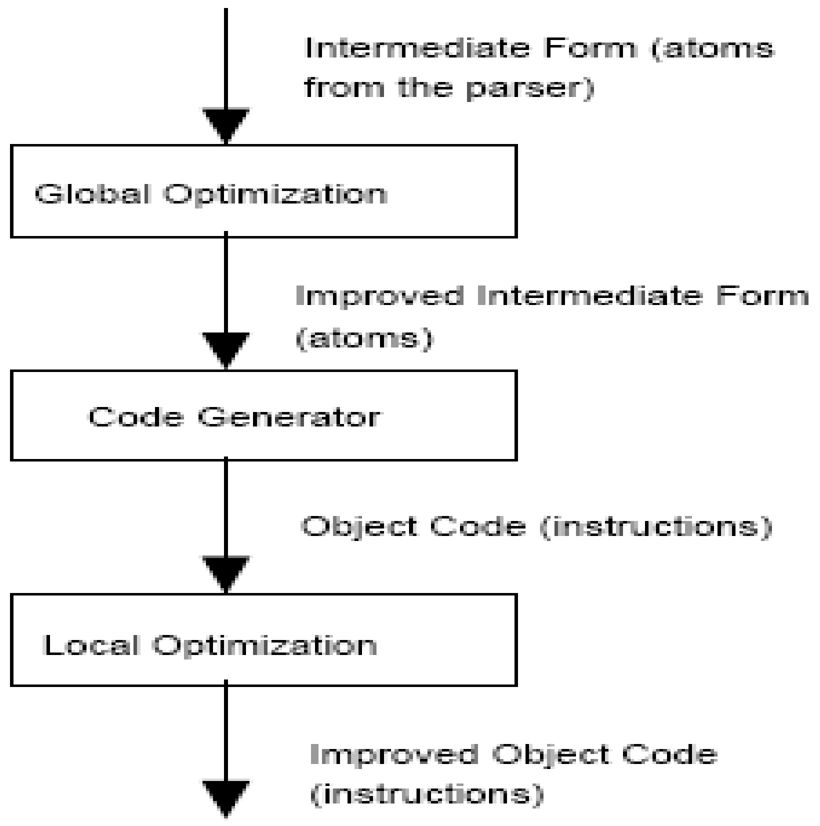
Optimization

- On the other hand, **global optimization** techniques are generally concerned with larger blocks of code, or even multiple blocks or modules

- and will be applied to the intermediate form, **atom strings, or syntax trees** put out by the parser.

Optimization

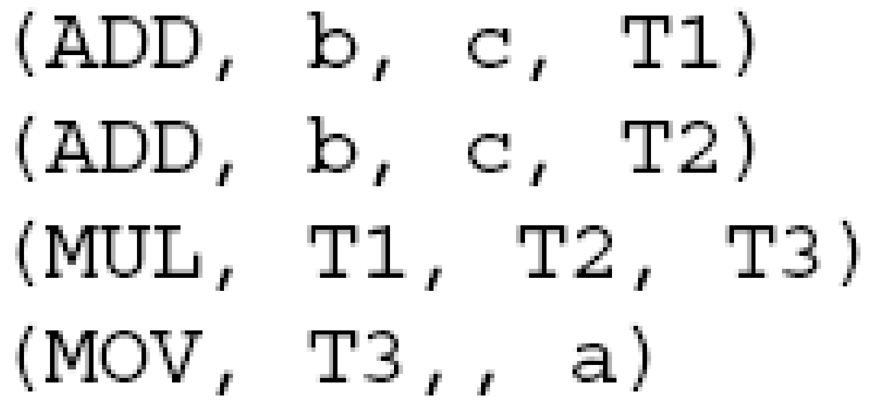
- Both local and global optimization phases are optional, but may be included in the compiler


Intermediate Form (atoms from the parser)
Global Optimization
Improved Intermediate Form (atoms)
Code Generator
Object Code (instructions)
Local Optimization
Improved Object Code (instructions)

Basic Blocks and DAGs

- The sequence of atoms put out by the parser is clearly not an optimal sequence;
  - there are many unnecessary and redundant atoms.

For example, consider the statement:
a = (b + c) ∗ (b + c) ;



```
(ADD, b, c, T1)
(ADD, b, c, T2)
(MUL, T1, T2, T3)
(MOV, T3,, a)
```

Basic Blocks and DAGs

- In the above example, it is clearly not necessary to evaluate the sum b + c twice.

- In addition, the MOV atom is not necessary because the MUL atom could store its result directly into the variable a.

```
(ADD, b, c, T1)
(ADD, b, c, T2)
(MUL, T1, T2, T3)
(MOV, T3,, a)
```

```
(ADD, b, c, T1)
(MUL, T1, T1, a)
```

Basic blocks

A basic block is a section of atoms which contains **no** Label or branch atoms (i.e., LBL, TST, JMP).

```
(ADD, b, c, T1)
(ADD, b, c, T2)
(MUL, T1, T2, T3)
(MOV, T3,, a)
```

```
(ADD, b, c, T1)                 Block 1
```

```
(LBL, L1)
```

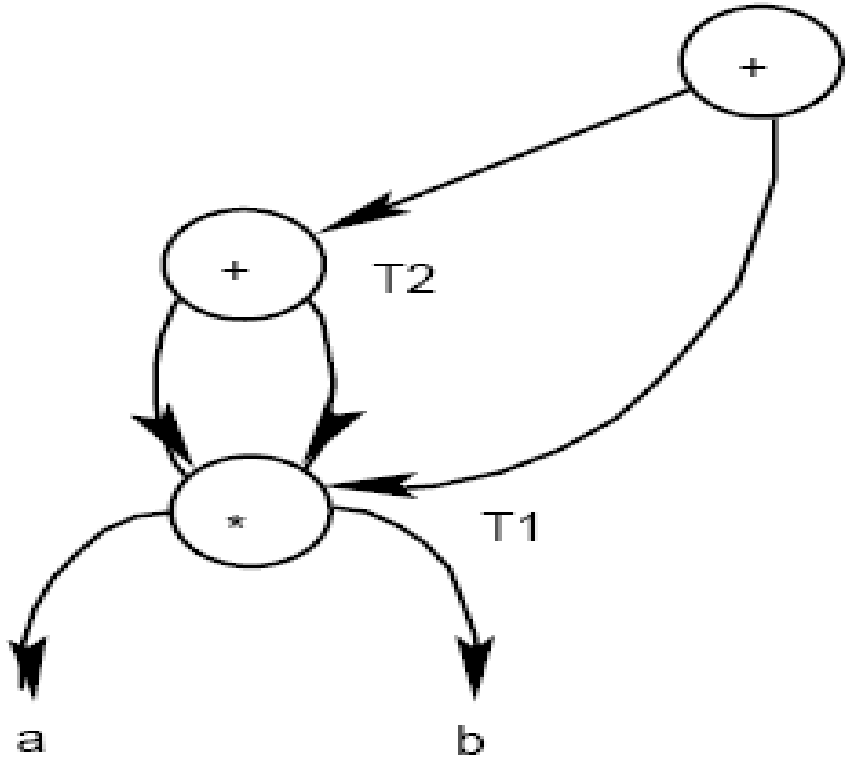
```
(ADD, b, c, T2)                 Block 2
(MUL, T1, T2, T3)
```

```
(TST, b, c,, 1, L3)
```

```
(MOV, T3,, a)                   Block 3
```

Directed Acyclic Graph (DAG)

- The DAG is *directed* because the arcs have arrows indicating the direction of the arcs,
  - it is *acyclic* because there is no path leading from a node back to itself.



DAG

- To build a DAG, given a sequence of atoms representing an arithmetic expression with binary operations, we use the following algorithm:

1. Read an atom.
2. If the operation and operands match part of the existing DAG, then add the result Label to the list of Labels on the parent and repeat from Step 1.
3. Otherwise, allocate a new node for each operand that is not already in the DAG, and a node for the operation. Label the operation node with the name of the result of the operation.
4. Connect the operation node to the two operands with directed arcs, so that it is clear which operand is the left and which is the right.
5. Repeat from Step 1.

DAG

- As an example, we will build a DAG for the expression

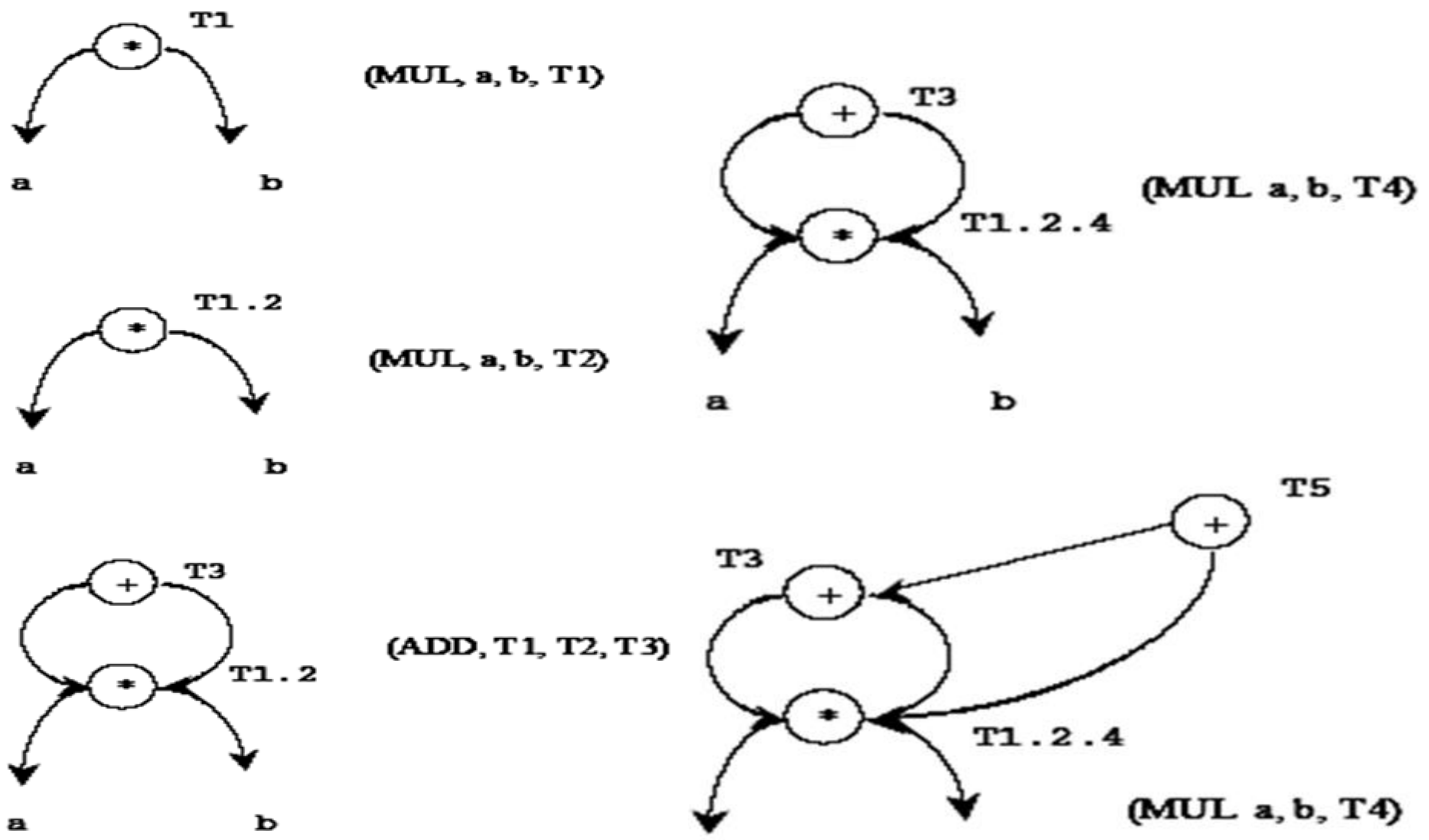
  $(a * b + a * b) + a * b.$

- The atom sequence as put out by the parser would be:

  (MUL, a, b, T1)
  (MUL, a, b, T2)
  (ADD, T1, T2, T3)
  (MUL, a, b, T4)
  (ADD, T3, T4, T5)

DAG

(MUL, a, b, T1)
(MUL, a, b, T2)
(ADD, T1, T2, T3)
(MUL, a, b, T4)
(ADD, T3, T4, T5)

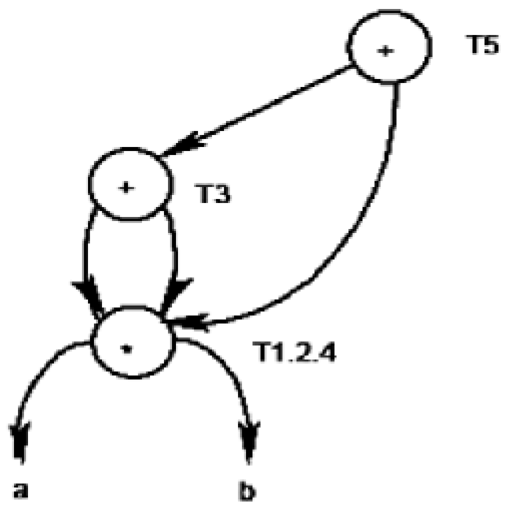


Convert the DAG to a basic block of atoms

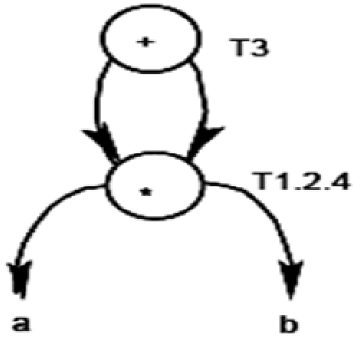
- The algorithm given below will generate atoms from DAG

1. Choose any node having no incoming arcs (root node).
2. Put out an atom for its operation and its operands.
3. Delete this node and its outgoing arcs from the DAG.
4. Repeat from Step 1 as long as there are still operation nodes remaining in the DAG.

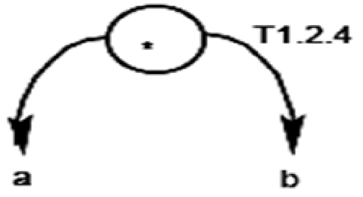
Convert the DAG to a basic block of atoms



(ADD, T3, T1.2.4, T5)



(ADD, T1.2.4, T1.2.4, T3)



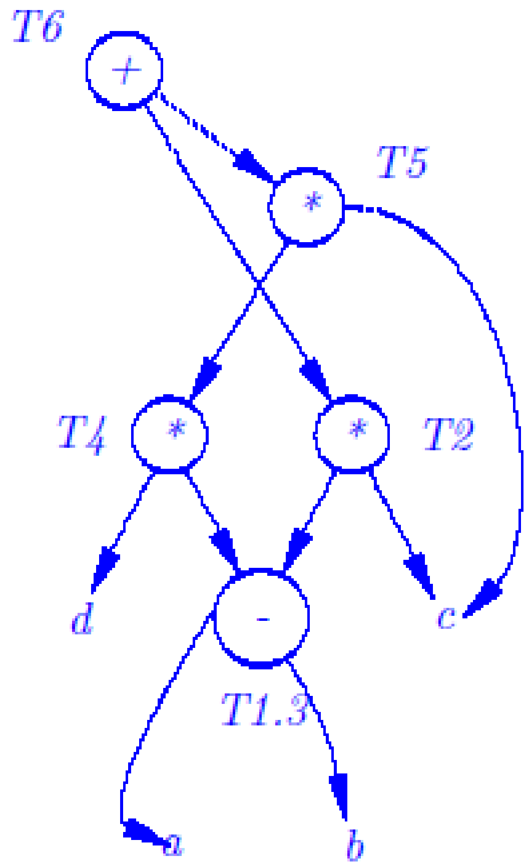
(MUL, a,b, T1.2.4)

(MUL, a, b, T1)
(MUL, a, b, T2)
(ADD, T1, T2, T3)
(MUL, a, b, T4)
(ADD, T3, T4, T5)

Example

Construct the DAG and show the optimized sequence of atoms for the Java expression ((a - b) * c) + (d * (a - b)) * c. The atoms produced by the parser are shown below:

(SUB, a, b, T1)
(MUL, T1, c, T2)
(SUB, a, b, T3)
(MUL, d, T3, T4)
(MUL, T4, c, T5)
(ADD, T2, T5, T6)



(SUB, a, b, T1.3)
(MUL, d, T1.3, T4)
(MUL, T4, c, T5)
(MUL, T1.3, c, T2)
(ADD, T2, T5, T6)

Other Global Optimization Techniques

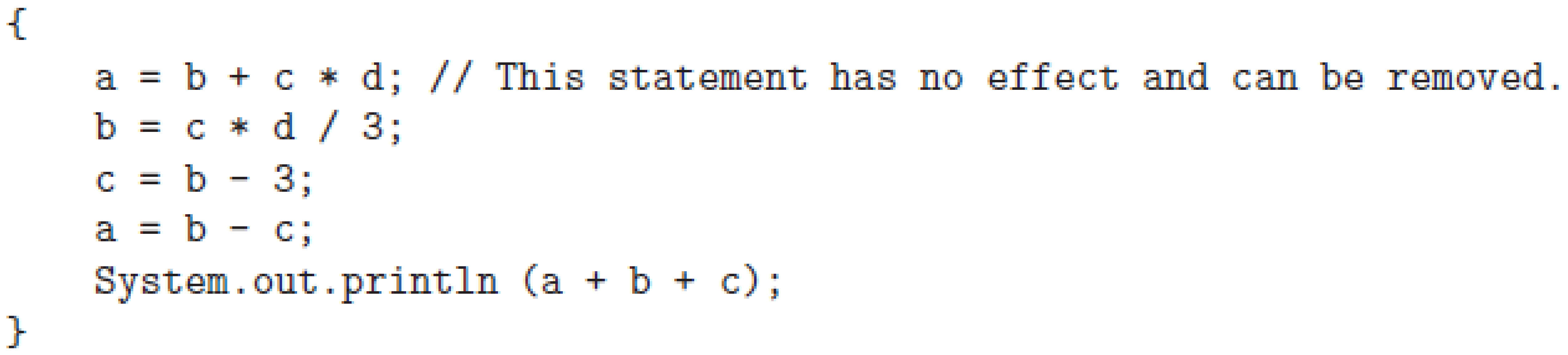
- **We will now examine a few other common global optimization techniques, however, we will not go into the implementation of these techniques.**

- ***Unreachable code*** **is an atom or sequence of atoms which cannot be executed because there is no way for the flow of control to reach that sequence of atoms.**

```
(JMP, L1)
(MUL, a, b, T1)
(SUB, T1, c, T2)
(ADD, T2, d, T3)
(LBL, L1)
```

Other Global Optimization Techniques

- **One such optimization technique is elimination of dead code, which involves determining whether computations specified in the source program are actually used and affect the program’s output.**



```
{
    a = b + c * d; // This statement has no effect and can be removed.
    b = c * d / 3;
    c = b - 3;
    a = b - c;
    System.out.println (a + b + c);
}
```

Other Global Optimization Techniques

A ***loop invariant*** is code within a loop which deals with data values that remain constant as the loop repeats.
Such code can be moved outside the loop, causing improved run time without changing the program’s semantics.

```
{
   for (int i=0; i<1000; i++)
      { a = sqrt (x);            // loop invariant
        vector[i] = i * a;
      }
}

{
   a = sqrt (x);                 // loop invariant
   for (int i=0; i<1000; i++)
      {
        vector[i] = i * a;
      }
}
```

Local Optimization

Local optimization techniques are often called *peephole* optimization, since they generally involve transformations on instructions which are close together in the object program.

There are three types of local optimization techniques

1. load/store optimization,
2. jump over jump optimization,
3. simple algebraic optimization.

a + b - c

```
LOD     R1,a
ADD     R1,b
STO     R1,T1
LOD     R1,T1
SUB     R1,c
STO     R1,T2
```

```
LOD     R1,a
ADD     R1,b
SUB     R1,c
STO     R1,T2
```

jump over jump optimization

```
(TST, a, b,, 3, L1)  if a > b  then goto L1
(JMP, L2)
(LBL, L1)
(MOV, b,, a)
(LBL, L2)
```

If(a>b)
a=b

The instructions generated by the code generator from this atom stream would be:

```
    LOD R1,a
    CMP R1,b,3          //Is R1 > b?
    JMP L1
    CMP 0,0,0           // Unconditional Jump
    JMP L2
L1:
    LOD R1,b
    STO R1,a
L2:
```

```
LOD R1,a
CMP R1,b,4              // Is R1 ≤b?
JMP L1
LOD R1,b
STO R1,a
L1:
```

Simple algebraic optimization

- The final example of local optimization techniques involves simple algebraic transformations which are machine dependent and are called ***simple algebraic optimizations***.

- For example, the following instructions can be eliminated:

```
MUL R1, 1
ADD R1, 0
```

Thanks