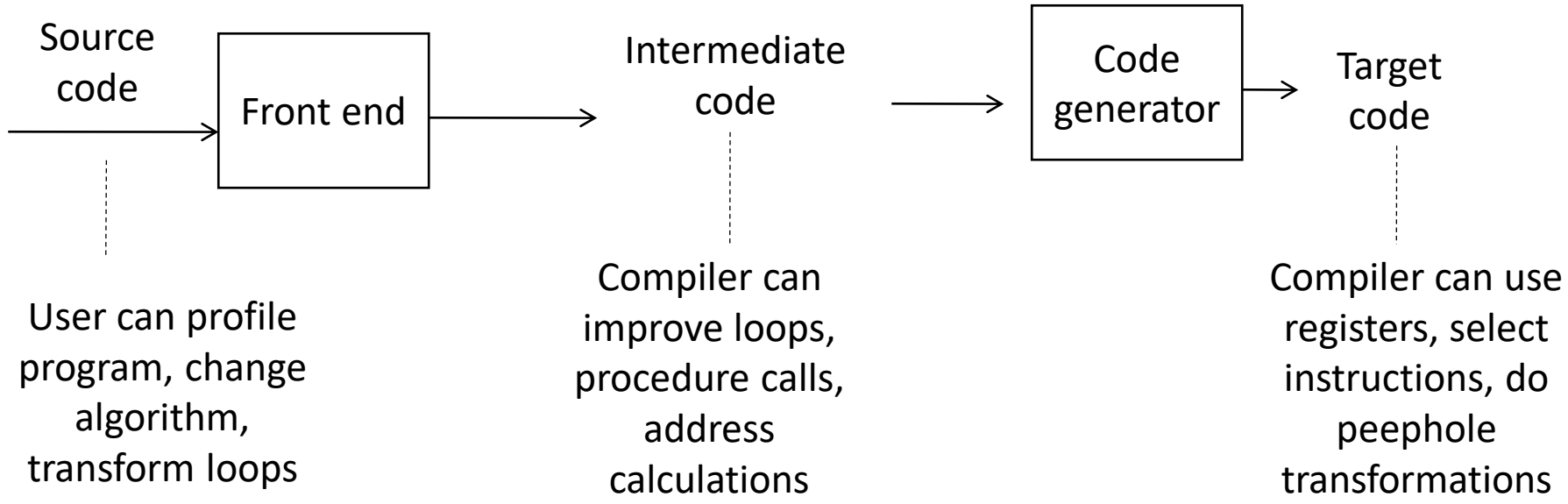
Code Optimization

Code optimization

- Compilers that apply code improving transformations are called *optimizing compilers.*

Getting better performance

- Places for potential improvement by user and compiler



Code Optimization

- Code improving transformations- improve the performance of a program
- Criteria for code improving transformations
  - Must preserve the meaning of the program
    - Optimization must not change the output of a program for a given input or cause an error such as division by zero that was not present in the original version of the program
  - Transformations must speed up the program by a measurable amount
    - Reduce the size of the code
  - Transformation must be worth the effort

Introduction

- Code optimization can be
  - Machine dependent
    - Requires knowledge of the target machine architecture
    - Better utilization of registers and other architecture characteristics possible
  - Machine independent
    - Performed independently of the target machine for which the compiler is generating code.

Basic Blocks and Flow Graph

- **Flow graph:** Graph representation of three address statements
  - Helpful for understanding code generation algorithms.
  - Nodes represent computation
  - Edges represent flow of control
- **Basic block** is a sequence of consecutive statements such that
  - Control enters at the beginning
  - Control leaves at the end
  - Control cannot halt or branch except at the end

Identification of Basic Block

Input: A sequence of three address statements

Output: A list of three address statements with each three address statement in exactly one block.

Method:

1. Determine the set of leaders, the first statement of basic blocks.
   a) The first statement is a leader.
   b) Any statement that is the target of a conditional or unconditional goto is a leader.
   c) Any statement that immediately follows a conditional or unconditional goto is a leader

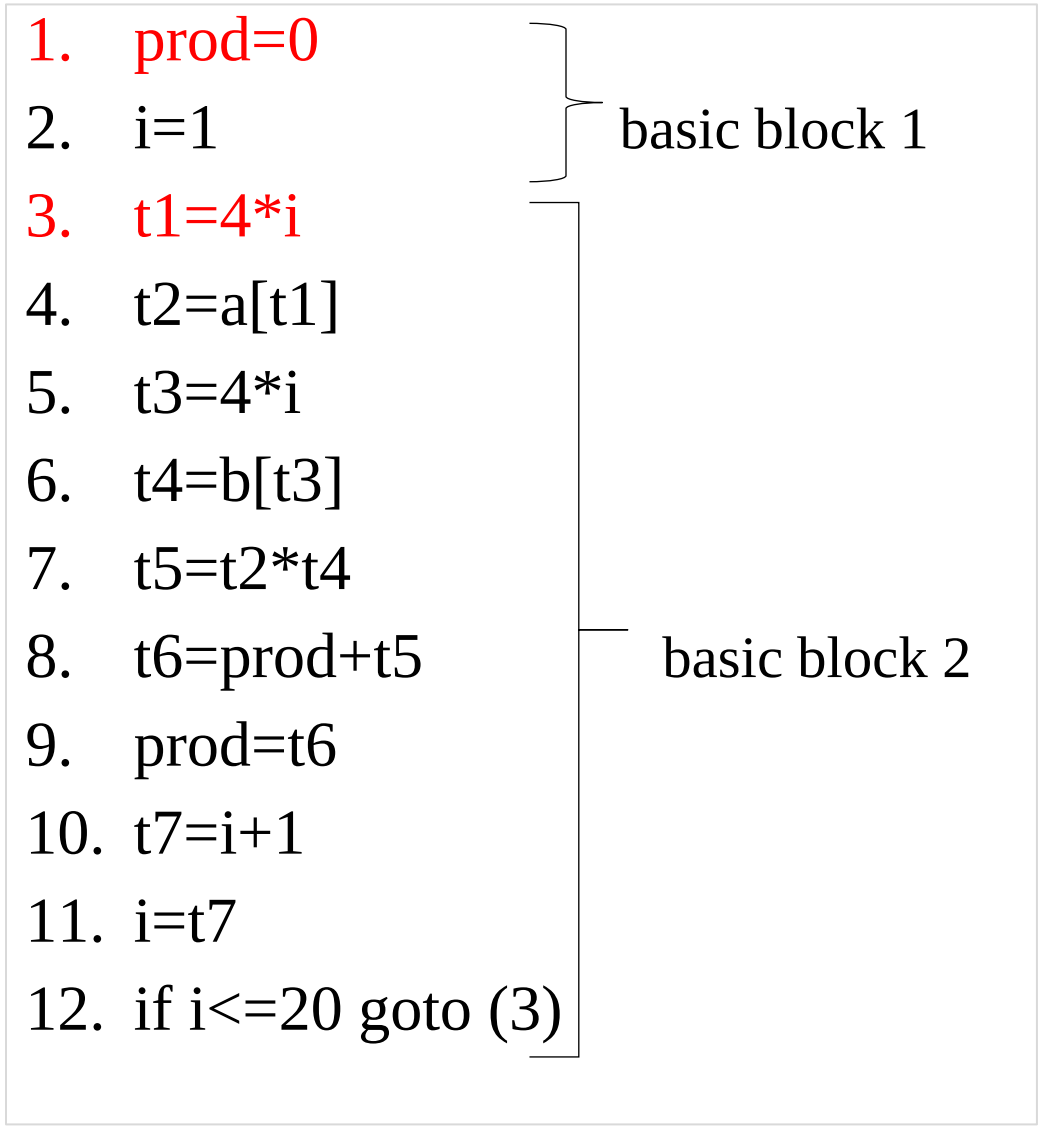
2. For each leader the basic block consists of the leader and all statements upto, but not including the next leader or end of the program.

Identification of Basic Block

```
begin
   prod=0;
   i=1;
   do begin
      prod=prod+a[i]*b[i];
        i=i+1;
   end
   while (i<=20)
end
```

1. prod=0
2. i=1
3. t1=4*i
4. t2=a[t1]
5. t3=4*i
6. t4=b[t3]
7. t5=t2*t4
8. t6=prod+t5
9. prod=t6
10. t7=i+1
11. i=t7
12. if i<=20 goto (3)

Identification of Basic Block



Flow Graph

- Flow graph can be constructed as follows
  - Identify the basic blocks of the function.
  - The initial node is the block whose leader is the first instruction of the function
  - There is a directed edge from block B1 to B2 if
    - there is a conditional or unconditional jump from the last statement of B1 to the first statement of B2
    - B2 immediately follows B1 in the textual order of the program and B1 does not end in an unconditional jump

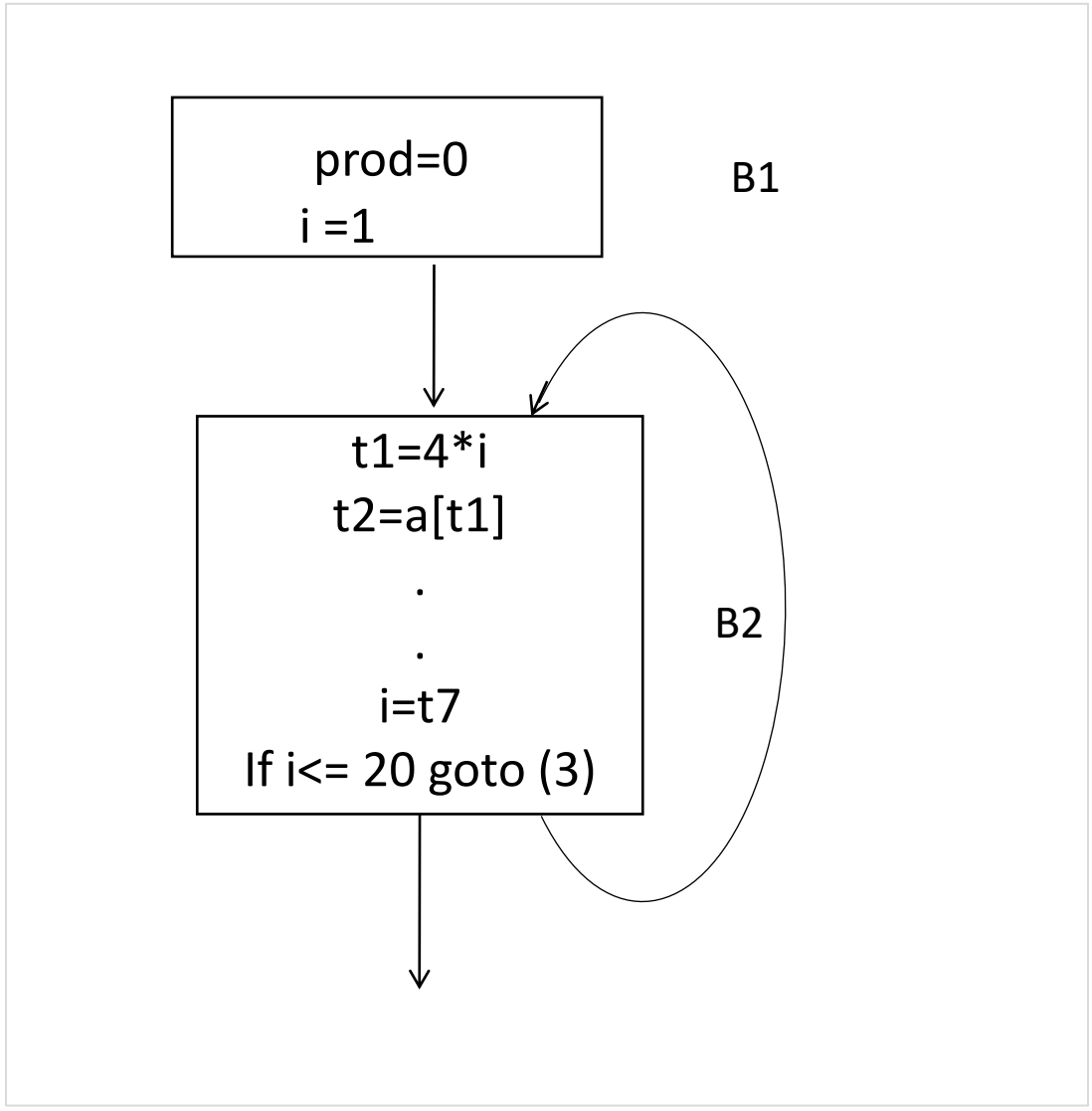
- We say that B1 is the *predecessor* of B2 and B2 is the *successor* of B1

Flow Graph

1. prod=0
2. i=1
3. t1=4*i
4. t2=a[t1]
5. t3=4*i
6. t4=b[t3]
7. t5=t2*t4
8. t6=prod+t5
9. prod=t6
10. t7=i+1
11. i=t7
12. if i<=20 goto (3)

basic block 1

basic block 2



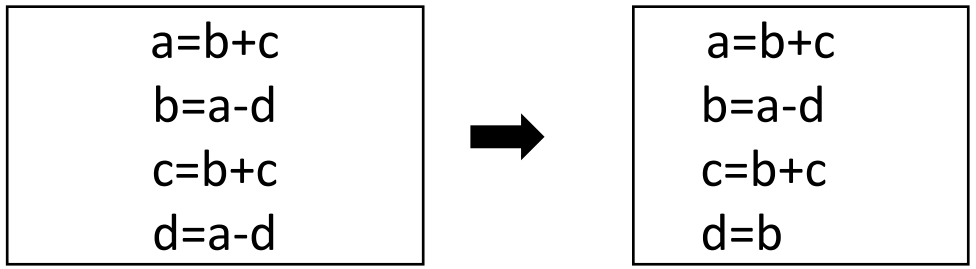
Transformations on Basic Blocks

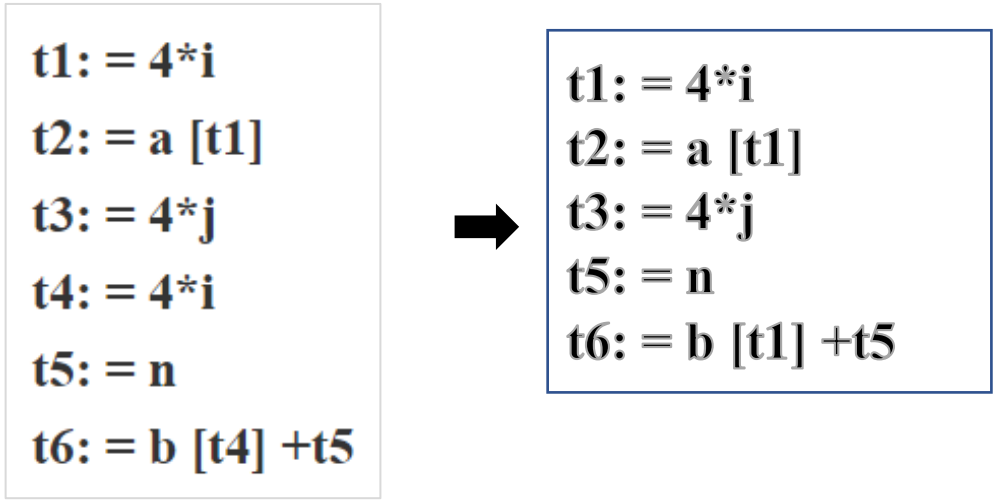
- Two important classes of local transformations:
  - **Structure preserving transformations**
    - Common subexpression elimination
    - Dead code elimination
    - Renaming of temporary variables
    - Interchange of two independent adjacent statements
  - **Algebraic Transformations**
    - Operations that simplify expressions
    - Replacing expensive operations by cheaper ones

Structure Preserving Transformations

Common Subexpression Elimination

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





Structure Preserving Transformations

- **Dead Code Elimination**
- A variable is **live** at a point if its value can be used subsequently, otherwise it is **dead**
- Dead code may appear as a result of previous transformations
- Example:

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

- Here, *if statement* is dead code because this condition will never get satisfied.
- Suppose ‘x’ is dead, i.e never subsequently used at the point where the statement x= y + z appears in a basic block.
- This statement can be removed without changing the value of the basic block.

Structure Preserving Transformations

- **Renaming Temporary Variables**

•We can transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary.

• Such a basic block is called a ***normal form block***.

$$t=b+c$$

•If we change this statement to

$$u= b+c$$

where u is a new temporary variable and change all instances of **t** to **u** the value of this basic block is not changed.

Structure Preserving Transformations

- **Interchange of statements**

  t1 = b + c

  t2 = x + y

- We can interchange the two statements without affecting the value of the block if and only if **neither x nor y is t1** and **neither b nor c is t2**.
- A normal form basic block permits all interchanges that are possible.

Algebraic Transformations

- **Simplify expressions**

  x=x+0 or x=x*1

Can be eliminated from the basic block

- **Replacing expensive operations by cheaper ones**

  x=y**2

Can be replaced by

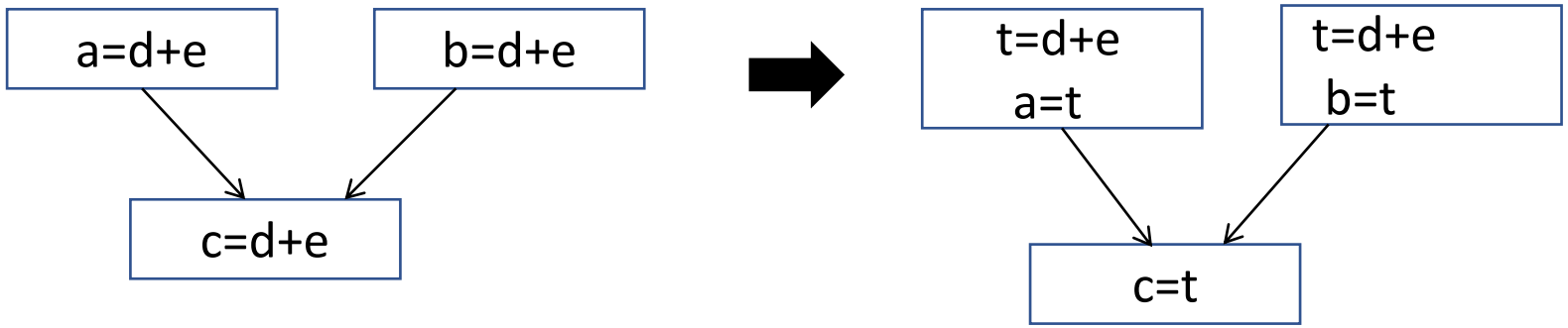
  x=y*y

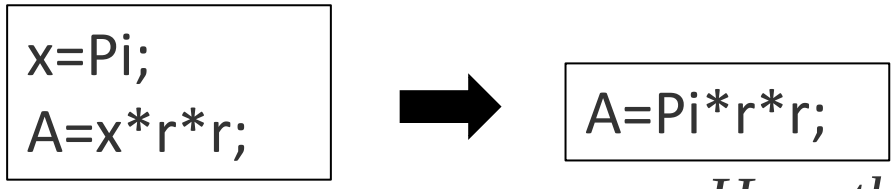
The Principal Sources of Optimization

- A transformation of a program is called **local** if it can be performed by looking only at the statements in the basic block, otherwise it is called **global**.
- Many transformations can be performed at local and global levels
- Local transformations are performed first
- **Function-Preserving Transformation**
  Improves performance without changing the function it computes
- Common subexpression elimination
- Copy propagation
- Dead-code elimination
- Constant folding

Copy Propagation

- Concerns assignments of the form f=g called copy statements or copies.
- 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.





Here the variable x is eliminated

Constant folding

- Deducing at compile time that the value of an expression is a constant and using the constant instead

- Example:

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

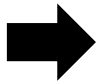
Loop Optimization

- Running time of a program can be improved by decreasing the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Techniques used are
  - Code motion which moves code outside the loop
  - Loop Unrolling
  - Induction variable elimination
  - Reduction in strength
    - Replacing an expensive operation by a cheaper one
    - Eg: replacing multiplication by addition.

Code Motion

- Reduces the amount of code in a loop
- Loop invariant: An expression that yields the same result independent of the no. of times the loop is executed
- Loop invariant computation is placed before the loop

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



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

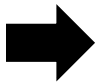
```
a = 200;
 while(a>0)
 {
    b = x + y;
    if (a % b == 0}
    printf("%d", a);
}
```

```
a = 200;
 b = x + y;
 while(a>0)
 {
    if (a % b == 0}
    printf("%d", a);
  }
```

Loop Unrolling

- **Loop Unrolling:**
- It helps in optimizing the execution time of the program by reducing the iterations.
- It increases the program's speed by eliminating the loop control and test instructions.

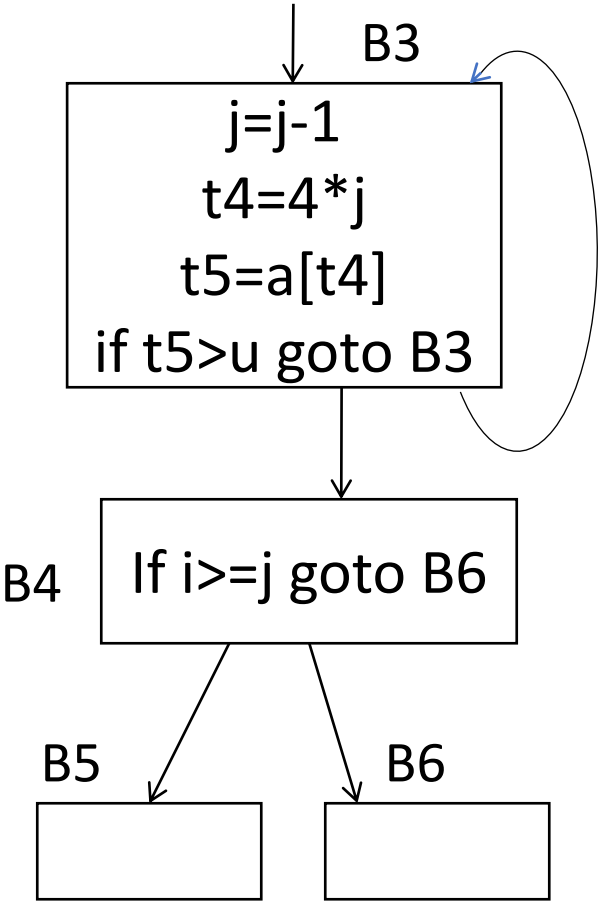
```
for(int i=0;i<2;i++)
{
  printf("Hello");
}
```



```
printf("Hello");
printf("Hello");
```

Induction Variables and Reduction in Strength

- In block B3 value of j and t4 remain in lockstep, every time the value of j decreases by 1 the value of t4 decreases by 4. Such identifiers are called *induction variables.*
- A variable x is an Induction Variable of a loop if every time the variable x changes values, it is incremented or decremented by some constant
- When there are two or more induction variables, it is possible to get rid of all but one by the process of *induction variable elimination.*
- Replace t4=4*j by t4=t4 – 4
- Place an initialization for t4 at the end of the block where j is initialized.



Optimization of Basic Blocks

- The code improving transformations for basic blocks included
- **Structure preserving transformations**
  - common sub expression elimination
  - dead-code elimination
- **Algebraic transformation**
  - reduction in strength

The Use of Algebraic Identities

- Algebraic identities another important class of optimization on basic blocks.
- For examples apply arithmetic identities, such as

  $x+0 = 0+x = x$

  $x-0 = x$

  $x*1 = 1*x = x$

Reduction in Strength and Constant Folding

- Another class of algebraic optimization includes reduction in strength, that is, replacing a more expensive operator by a cheaper one as in

  x**2 = x*x

  2.0 *x = x+x

- Evaluate constant expressions at compile time and replace the constant expressions by their values.

  2*3.14 = 6.28

A Simple Code Generator

- A good code generation must do the following things:
  - Produce correct code
  - Make use of machine architecture
  - Run efficiently
- Input to code generator
  - Intermediate code program
  - Symbol table
- Output of the code generator is the target program
  - Assembly language program
  - Absolute machine language program
  - Relocatable machine language program

Simple code generation algorithm

- **Input:** a sequence of three address statements partitioned into basic blocks
- **Output**: code for each three address statement

Algorithm takes advantage of any three address statement that are in register as long as possible.

- To avoid error the algorithm stores everything across basic block boundaries.

Register Descriptor keeps track of what is currently in each register. It is consulted whenever a new register is needed. Initially all registers are empty. As the code generation progresses each register will hold the values of one or more variables.

Address Descriptor keeps track of the location where the current value of the variable can be found at run time. Location can be a register, a stack location, a memory address. This information can be stored in symbol table.

Code Generation Algorithm

x := y op z

1. Invoke a function ***'getreg'*** to determine the location L where the result of computation of *y op z* should be stored. L will usually be a register

2. Consult the address descriptor for y to determine the y', the current location of y. If the value of y is currently both in register and memory location, prefer register y. If the value of y is not in L, generate instruction ***MOV y', L*** to place a copy of y in L

3. Now x and y are in L

4. Generate the instruction **op z', L** where z' is the current location of z. Update address descriptor of x to indicate that x is in L. If L is a register, update its descriptor to indicate that it contains the value of x and remove x from all other register descriptors.

5. If the current values of y and/or z are in register and have no next uses and they are not alive at the end of the block, alter the register descriptors to indicate that after the execution of x:=y op z, those registers no longer contains y and/or z respectively.

The function getreg

The function **getreg** returns the location L to hold the value of x for the assignment x:=y *op* z

The algorithm for getreg:

1) It searches for a register containing the name y. If such a register exists, and it holds the value of no other names (eg: x:=y, reg. for y holds the names of x and y), and y is not live and has no next use after the execution of x = y *op* z, then return L.

Update the address descriptor of y to indicate that y is no longer in L.

2) Failing (1), return an empty register for L if there is one.

3) Failing (2), if x has a next use in the block, or if *op* requires a register then

a. find an occupied register R.

b. Store the value of R into a memory location (MOV(R,M)) if value of R is not in proper M and update the address descriptor for M and return R.

If R holds value of many variables, generate a MOV for each of the variables.

4) Failing (3), select the memory location of x as L.

Example

d := (a-b) + (a-c) + (a-c)

t=a-b
u=a-c
v=t+u
d=v+u

Statements	**Code generated**	**Register descriptor**	**Address descriptor**
		All registers are initially empty	
t = a-b	MOV a, R0 SUB b,R0	R0 contains t	t in R0
u= a-c	MOV a, R1 SUB c,R1	R0 contains t R1 contains u	t in R0 u in R1
v= t+u	ADD R1,R0	R0 contains v R1 contains u	u in R1 v in R0
d= v+u	ADD R1,R0 MOV R0,d	R0 contains d	d in R0 d in R0 and memory

Machine Dependent Optimization

- Machine dependent optimization involves transformations that take into consideration, the properties of the target machine like registers and special machine instruction sequences available, etc.
- These techniques are applied on generated target code.
- 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
- Many simple transformations can significantly improve the running time or space requirement of the target program.

Peephole Optimization

- Peephole optimization is a simple but effective technique for locally improving the target code
- 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
- 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.
- Each improvement may spawn opportunities for additional improvements.
- In general, repeated passes over the target code are necessary to get the maximum benefit
- Peephole optimization can be applied on intermediate codes as well as on target codes.

Peephole Optimization

- Transformations done in Peephole Optimization
  - Redundant instruction elimination
  - Unreachable code elimination
  - Flow of control optimization
  - Algebraic simplification
  - Reduction in strength
  - Use of machine idioms

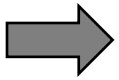
Peephole Optimization

- **Eliminating redundant load and stores**
- Consider the instruction sequence in the target program
  - LD R0, a
  - ST a, R0
- 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 **R0**.

Peephole Optimization

- **Eliminating Unreachable Code**
- 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.
- One obvious peephole optimization is to eliminate jumps over jumps.

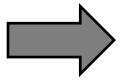
```
if debug == 1 goto L1
goto L2
L1: print debugging information
L2:
```



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

- If debug is set to 0 at the beginning of the program, the argument of the first statement always evaluates to true, so the statement can be replaced by goto L2.
- Then all statements that print debugging information are unreachable and can be eliminated one at a time.

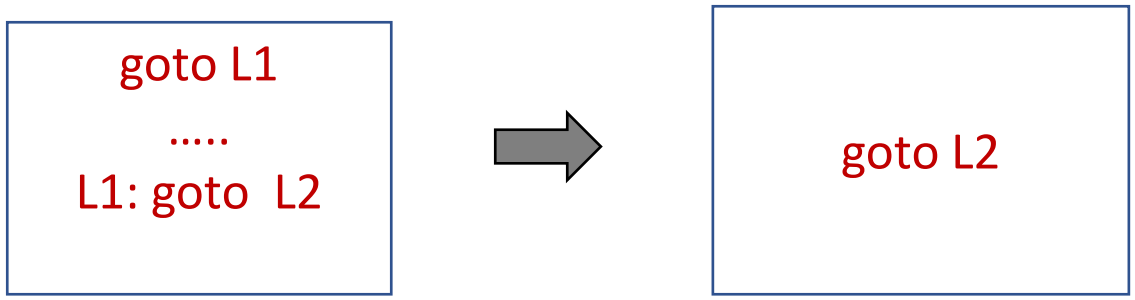
```
if 0 != 1 goto L2
print debugging information
L2:
```



```
goto L2
```

Peephole Optimization

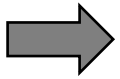
- **Eliminating Unreachable Code**
- An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions.



Peephole Optimization

- **Flow of Control Optimization**
- 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

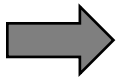
```
goto L1
L1: goto L2
```



```
goto L2
L1: 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.
- The sequence

```
if a < b goto L1
L1: goto L2
```



```
if a < b goto L2
L1: goto L2
```

Peephole Optimization

- **Algebraic Simplification and Reduction in Strength**
- Algebraic identities can also be used by a peephole optimizer to eliminate three-address statements
  - x = x + 0 or x = x * 1 ← No need
- Reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones on the target machine.
  - 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 approximated as multiplication by a constant, which may be cheaper.

Peephole Optimizations

- **Use of Machine Idioms**
  - The target machine may have hardware instructions to implement certain specific operations efficiently.
  - 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 these 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

    - i:=i+1 → i++
    - i:=i-1 → i--