Code Optimization

Part I

Optimization

Basic Code Generation
    Produces functional but poor code.

Goal: Improve the code as much as possible.
    Dramatically improves code performance (e.g., 2X to 10X)

"*Optimization*" -- more likely "*Improvement*"

Machine-Independent v. Machine-Dependent Optimizations

Variety of techniques
    Add as many optimization algorithms as possible
        Some are VERY complex!
    Do testing w/ sample programs to evaluate
        which optimization strategies work best.

Different needs for different languages (FORTRAN)

Optimization: Requirements

Every optimization must be "safe"

Must not change the program output ... for any input.

Must not allow new errors or exceptions.

Goals of optimization:

- Runtime Execution Speed!!!
- Other (e.g., Code Size, Power Consumption)

Every optimization should improve the program

but may slow some programs!

Is optimization worth the effort?

Some algorithms may be difficult to implement.

Many programs run only once

Compiler used heavily during debugging.

Program is only run once or twice before being modified.

$\Rightarrow$ Compiler performance matters more.

But some programs are *computation-intensive*

More computation per time unit means more accurate results

The secret to getting programs to run faster?

Use a better algorithm!

Most optimizations done by a compiler are
"*constant-factor*" speed-ups
(e.g., 25% faster)

Optimizations by the programmer:

- Change the algorithm
  $$N^2 \rightarrow N \log N$$

- Profile the program and tweak the algorithm

- Misc. transformations

Types of Optimization

Machine Independent Optimizations

- Live Variable Analysis
- Common sub-expressions
- Eliminate unnecessary copying
- Loop transformations

...etc...

Optimization transforms IR Code

Machine Dependent Optimizations

- Effective Register Usage
- Select Best Target Instructions
- Select a schedule that executes quickly

... given the CPU idiosynchracies

(e.g., memory latencies, functional units, etc.)

Optimization transforms Target Code

Is Optimization Necessary…

...assuming the programmer writes good, efficient code?

Source Code:

```
A[i] := B[i] + C[i];
```

Translation:

```
t1 := i * 4
t2 := B[t1]
t3 := i * 4
t4 := C[t3]
t5 := t2 + t4
t6 := i * 4
A[t6] := t5
```

The compiler will insert many hidden operations
(often concerning pointers and address calculations)

Local vs Global

"Local Transformations"
   Within a single basic block
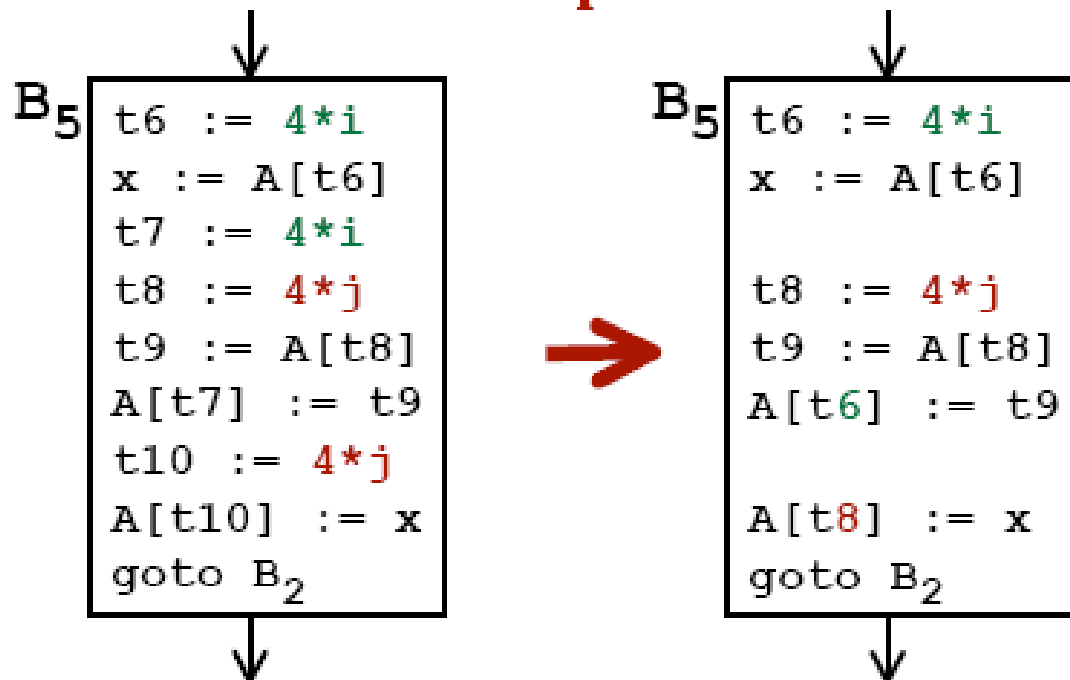
"Global Transformations"
   Concern several basic blocks
      (but typically within a single routine / control flow graph)

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
```

→

B₅
```
t6  := 4*i
x   := A[t6]

t8  := 4*j
t9  := A[t8]
A[t6]  := t9

A[t8]  := x
goto B₂
```

Quicksort

```
procedure quicksort (m,n: int) is
    var i,j,v,x: int := 0;
    if (n ≤ m) then return; end;
    i := m - 1;
    j := n;
    v := A[n];
    while true do
        repeat
            i := i + 1;
        until A[i] ≥ v;
        repeat
            j := j - 1;
        until A[j] ≤ v;
        if i ≥ j then exit; end;
        x := A[i];
        A[i] := A[j];
        A[j] := x;
    end;
    x := A[i];
    A[i] := A[n];
    A[n] := x;

    quicksort (m,j);
    quicksort(i+1,n);
endProc;
```
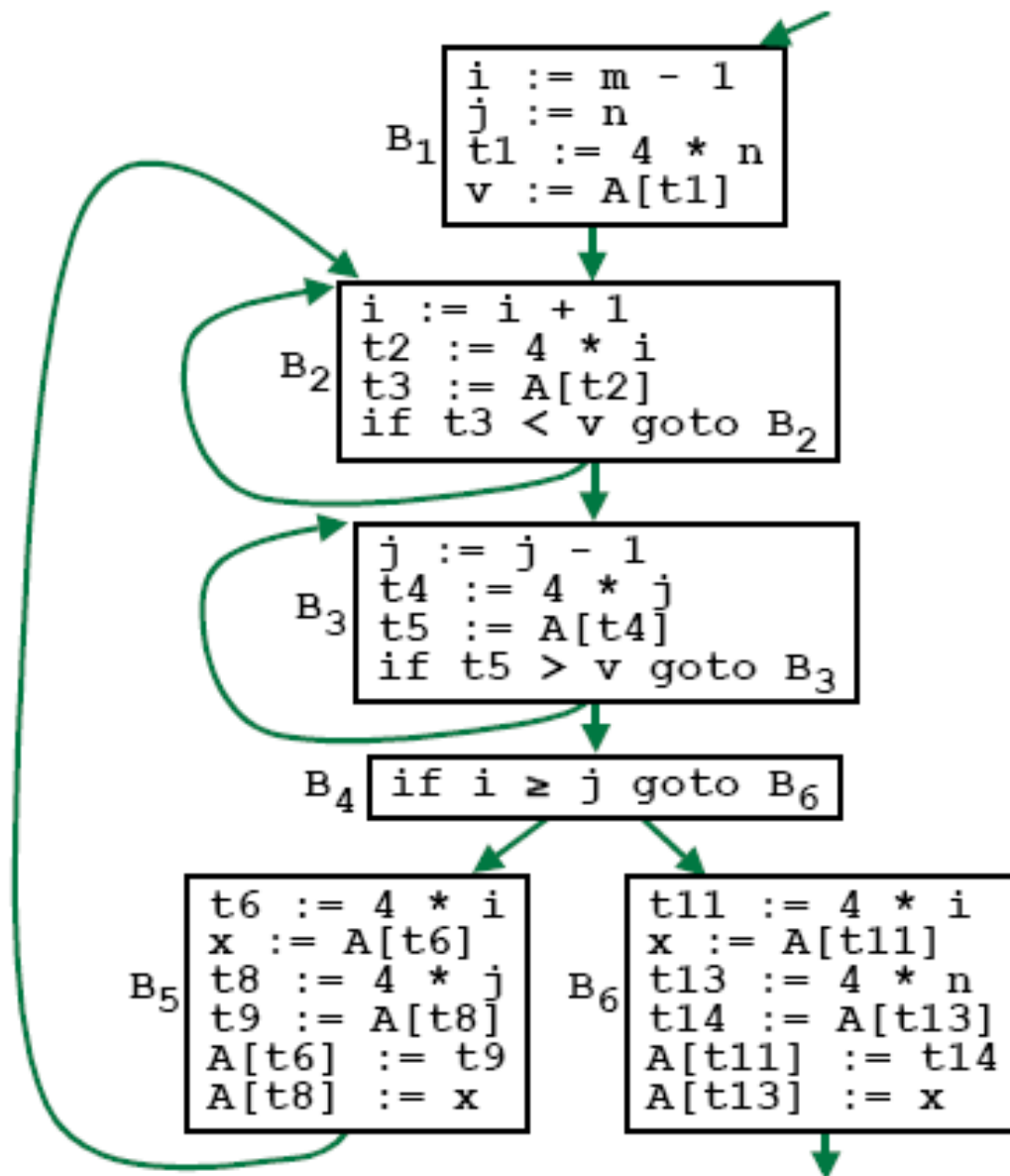
Quicksort

```
procedure quicksort (m,n: int) is
    var i,j,v,x: int := 0;
    if (n ≤ m) then return; end;
    i := m - 1;
    j := n;
    v := A[n];
    while true do
        repeat
            i := i + 1;
        until A[i] ≥ v;
        repeat
            j := j - 1;
        until A[j] ≤ v;
        if i ≥ j then exit; end;
        x := A[i];
        A[i] := A[j];
        A[j] := x;
    end;
    x := A[i];
    A[i] := A[n];
    A[n] := x;
    quicksort (m,j);
    quicksort(i+1,n);
endProc;
```
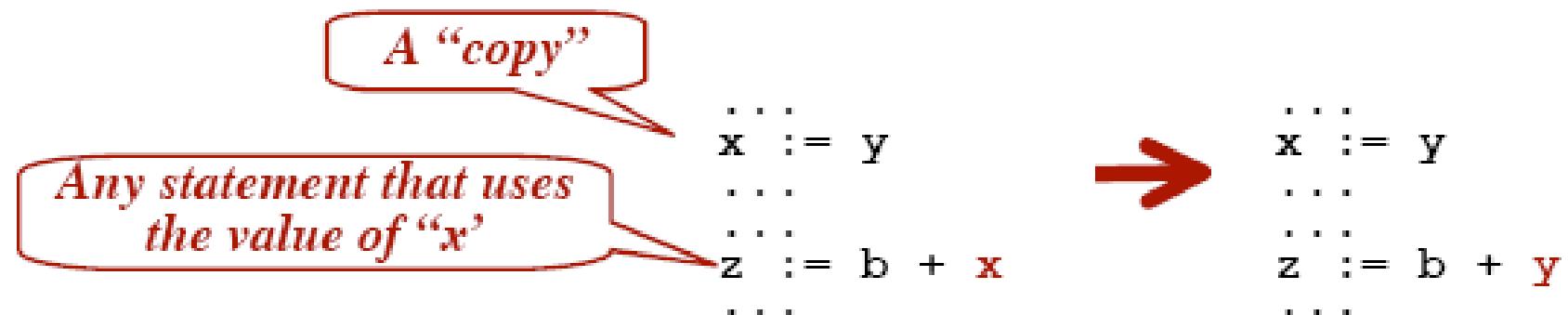
Flowgraph


```
         ┌──────────────────┐
      B₁ │ i  := m - 1      │
         │ j  := n          │
         │ t1 := 4 * n      │
         │ v  := A[t1]      │
         └──────────────────┘

         ┌──────────────────┐
      B₂ │ i  := i + 1      │
         │ t2 := 4 * i      │
         │ 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      │ B₆│ t11 := 4 * i     │
  │ x  := A[t6]      │   │ x   := A[t11]    │
  │ t8 := 4 * j      │   │ t13 := 4 * n     │
  │ t9 := A[t8]      │   │ t14 := A[t13]    │
  │ A[t6] := t9      │   │ A[t11] := t14    │
  │ A[t8] := x       │   │ A[t13] := x      │
  └──────────────────┘   └──────────────────┘
```

Copy Propagation

A "copy"

Any statement that uses
the value of "x"

```
. . .              . . .
x := y             x := y
. . .              . . .
. . .              . . .
z := b + x    →    z := b + y
. . .              . . .
```
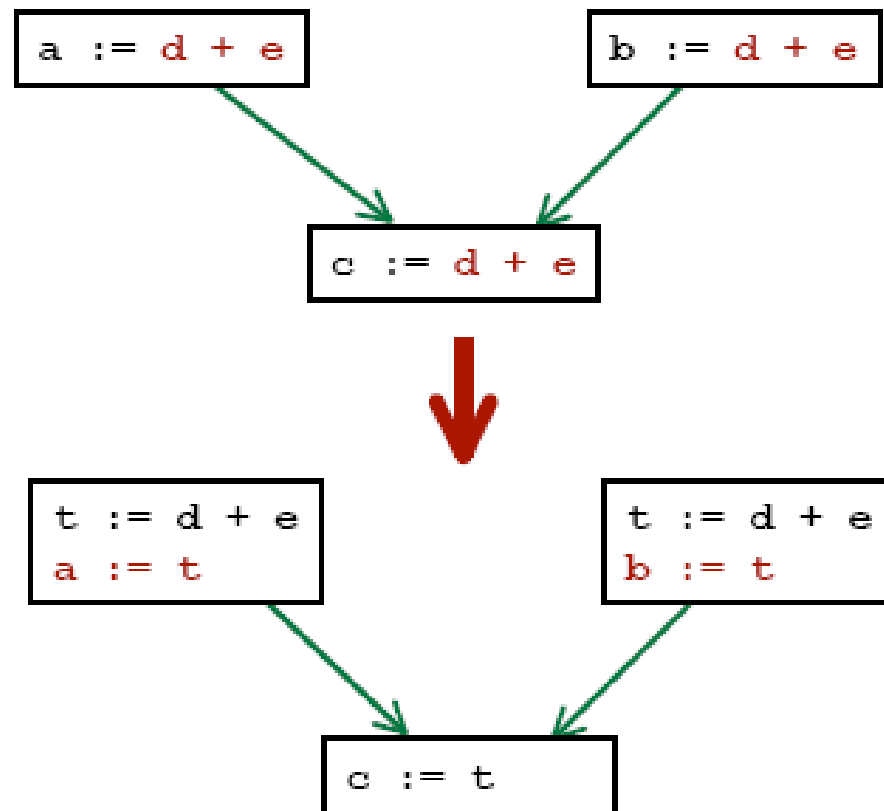
Why perform this optimization?

The copy may become DEAD CODE.

We may delete the copy later!

Copy Propagation

An expression...
    Simple computation
    Computed in several places

```
a := d + e          b := d + e
```

```
c := d + e
```

```
t := d + e          t := d + e
a := t              b := t
```

```
c := t
```

Copies will be introduced during sub-expression elimination
    ...but they may be DEAD CODE!

Example

B1:
```
i  := m - 1
j  := n
t1 :=  4 * n
v  := A[t1]
```

B2:
```
i  := i + 1
t2 :=  4 * i
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]
t8 := 4 * j
t9 := A[t8]
A[t6] := t9
A[t8] := x
```

B6:
```
t11 := 4 * i
x  := A[t11]
t13 := 4 * n
t14 := A[t13]
A[t11] := t14
A[t13] := x
```

Example

Global Common Sub-Expression Elimination

B1
```
i  := m - 1
j  := n
t1 := 4 * n
v  := A[t1]
```

B2
```
i  := i + 1
t2 := 4 * i
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]
t8 := 4 * j
t9 := A[t8]
A[t6] := t9
A[t8] := x
```

B6
```
t11 := 4 * i
x   := A[t11]
t13 := 4 * n
t14 := A[t13]
A[t11] := t14
A[t13] := x
```

Example

Global Common
Sub-Expression Elimination

B1
```
i  := m - 1
j  := n
t1 := 4 * n
v  := A[t1]
```

B2
```
i  := i + 1
t2 := 4 * i
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 := t2
x  := A[t6]
t8 := t4
t9 := A[t8]
A[t6] := t9
A[t8] := x
```

B6
```
t11 := t2
x   := A[t11]
t13 := t1
t14 := A[t13]
A[t11] := t14
A[t13] := x
```

Example

Copy Propagation

B₁
```
i  := m - 1
j  := n
t1 := 4 * n
v  := A[t1]
```

B₂
```
i  := i + 1
t2 := 4 * i
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 := t2
x  := A[t6]
t8 := t4
t9 := A[t8]
A[t6] := t9
A[t8] := x
```

B₆
```
t11 := t2
x   := A[t11]
t13 := t1
t14 := A[t13]
A[t11] := t14
A[t13] := x
```

Example

Copy Propagation

```
      i  := m - 1
      j  := n
B1    t1 := 4 * n
      v  := A[t1]
```

```
      i  := i + 1
      t2 := 4 * i
B2    t3 := A[t2]
      if t3 < v goto B2
```

```
      j  := j - 1
      t4 := 4 * j
B3    t5 := A[t4]
      if t5 > v goto B3
```

```
B4    if i ≥ j goto B6
```

```
      t6 := t2
      x  := A[t2]
B5    t8 := t4
      t9 := A[t4]
      A[t2] := t9
      A[t4] := x
```

```
      t11 := t2
      x   := A[t2 ]
B6    t13 := t1
      t14 := A[t1 ]
      A[t2 ] := t14
      A[t1 ] := x
```

Example

Global Common Sub-Expression Elimination

```
        i  := m - 1
        j  := n
     B1 t1 := 4 * n
        v  := A[t1]

        i  := i + 1
        t2 := 4 * i
     B2 t3 := A[t2]
        if t3 < v goto B2

        j  := j - 1
        t4 := 4 * j
     B3 t5 := A[t4]
        if t5 > v goto B3

     B4 if i ≥ j goto B6

     t6 := t2              t11 := t2
     x  := A[t2]           x   := A[t2 ]
  B5 t8 := t4          B6 t13 := t1
     t9 := A[t4]          t14 := A[t1 ]
     A[t2] := t9          A[t2 ] := t14
     A[t4] := x           A[t1 ] := x
```

Example

Global Common
Sub-Expression Elimination

```
       ┌─────────────────┐
       │ i  := m - 1     │
       │ j  := n         │
   B1  │ t1 := 4 * n     │
       │ v  := A[t1]     │
       └─────────────────┘

       ┌─────────────────┐
       │ i  := i + 1     │
       │ t2 := 4 * i     │
   B2  │ t3 := A[t2]     │
       │ if t3 < v goto B2│
       └─────────────────┘

       ┌─────────────────┐
       │ j  := j - 1     │
       │ t4 := 4 * j     │
   B3  │ t5 := A[t4]     │
       │ if t5 > v goto B3│
       └─────────────────┘

   B4  │ if i ≥ j goto B6 │

   ┌─────────────────┐   ┌─────────────────┐
   │ t6 := t2        │   │ t11 := t2       │
   │ x  := t3        │   │ x   := t3       │
B5 │ t8 := t4        │ B6│ t13 := t1       │
   │ t9 := t5        │   │ t14 := A[t1 ]   │
   │ A[t2]  := t9    │   │ A[t2 ] := t14   │
   │ A[t4]  := x     │   │ A[t1 ] := x     │
   └─────────────────┘   └─────────────────┘
```

Example

Copy Propagation

```
      ┌─────────────────┐
   B₁ │ i  := m - 1     │
      │ j  := n         │
      │ t1 := 4 * n     │
      │ v  := A[t1]     │
      └─────────────────┘

      ┌─────────────────┐
   B₂ │ i  := i + 1     │
      │ t2 := 4 * i     │
      │ 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 := t2        │    B₆ │ t11 := t2        │
      │ x  := t3        │       │ x  := t3         │
      │ t8 := t4        │       │ t13 := t1        │
      │ t9 := t5        │       │ t14 := A[t1 ]    │
      │ A[t2] := t9     │       │ A[t2 ] := t14    │
      │ A[t4] := x      │       │ A[t1 ] := x      │
```

Example

Copy Propagation

$$B_1 \quad \begin{array}{l} \texttt{i := m - 1} \\ \texttt{j := n} \\ \texttt{t1 := 4 * n} \\ \texttt{v := A[t1]} \end{array}$$

$$B_2 \quad \begin{array}{l} \texttt{i := i + 1} \\ \texttt{t2 := 4 * i} \\ \texttt{t3 := A[t2]} \\ \texttt{if t3 < v goto } B_2 \end{array}$$

$$B_3 \quad \begin{array}{l} \texttt{j := j - 1} \\ \texttt{t4 := 4 * j} \\ \texttt{t5 := A[t4]} \\ \texttt{if t5 > v goto } B_3 \end{array}$$

$$B_4 \quad \texttt{if i } \geq \texttt{ j goto } B_6$$

$$B_5 \quad \begin{array}{l} \texttt{t6 := t2} \\ \texttt{x := t3} \\ \texttt{t8 := t4} \\ \texttt{t9 := t5} \\ \texttt{A[t2] := t5} \\ \texttt{A[t4] := t3} \end{array}$$

$$B_6 \quad \begin{array}{l} \texttt{t11 := t2} \\ \texttt{x := t3} \\ \texttt{t13 := t1} \\ \texttt{t14 := A[t1]} \\ \texttt{A[t2] := t14} \\ \texttt{A[t1] := t3} \end{array}$$

Example

Dead Code Elimination

B_1
```
i  := m - 1
j  := n
t1 := 4 * n
v  := A[t1]
```

B_2
```
i  := i + 1
t2 := 4 * i
t3 := A[t2]
if t3 < v goto B2
```

B_3
```
j  := j - 1
t4 := 4 * j
t5 := A[t4]
if t5 > v goto B3
```

B_4
```
if i ≥ j goto B6
```

B_5
```
t6 := t2
x  := t3
t8 := t4
t9 := t5
A[t2] := t5
A[t4] := t3
```

B_6
```
t11 := t2
x   := t3
t13 := t1
t14 := A[t1 ]
A[t2 ] := t14
A[t1 ] := t3
```

Example

B₁
```
i  := m - 1
j  := n
t1 := 4 * n
v  := A[t1]
```

B₂
```
i  := i + 1
t2 := 4 * i
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
```
A[t2] := t5
A[t4] := t3
```

B₆
```
t14 := A[t1 ]
A[t2 ] := t14
A[t1 ] := t3
```

Comparison to Original

```
t6 := 4 * i
x  := A[t6]
t8 := 4 * j
t9 := A[t8]
A[t6] := t9
A[t8] := x
```

```
t11 := 4 * i
x   := A[t11]
t13 := 4 * n
t14 := A[t13]
A[t11] := t14
A[t13] := x
```

Consider this Example

```
debug := FALSE;
...
if (debug) then print (...) endif
```

> *Unreachable? Can this code be eliminated?*

Data Flow Analysis

"Which computations can reach which points"

Only one DEFINITION of "debug" can reach this USE.

Must have the value "false", so okay to optimize the IF statement.

Global Common Sub-Expression Elimination.

Copy Propagation.

Live-Variable Anaylsis.

Constant Folding

*"If we know the value of a variable at compile-time,
we may go ahead and perform the computation."*

Dead-Code Elimination

"Eliminate code that is unreachable."

"Eliminate code that compute DEAD variables."

Optimizing Loops

The 90-10 Rule

"90% of execution time is spent in 10% of the code."

Try to move code out of loops

Identify Loops

Nesting of loops

"inner loops"

"outer loops"

GOAL:

Move Code "Outward"

Make Loops Run Faster

Loop Invariant Computation

```
while i <= MAX-1 do
        ...
        j := i * (MIN+1);
        ...
end;
```

These computations are "loop-invariant"

If an expression is computed within a loop

  and

It does not depend on variables that change in the loop

  then

Move it to just before the loop!

```
t1 := MAX-1;
t2 := MIN-1;
while i <= t1 do
        ...
        j := i * t2;
        ...
end;
```

Induction Variables

Definition:

Loop counters that move together (in "lock-step") during loop execution.

Example Source:

```
loop
   . . .
   i := i + 1;
   ... A[i]...
   . . .
endloop
```

Translation to IR:

```
. . .
i := i + 1
t := i * 4
... A[t]...
. . .
```

Relationship:

$t = i*4$

Note:

```
t := i*4
```

Establishes the relationship.

Also: We know this relationship exists directly after

```
t := i*4
```

is executed.

i	t
1	4
2	8
3	12
4	16
5	20
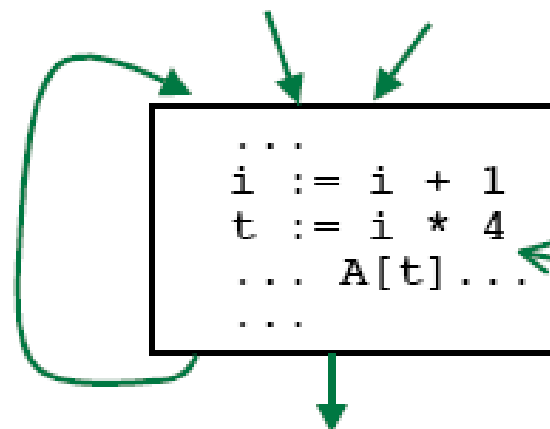
Some Reasoning

Assume $t = i*4$ here.

Then, after "i" is incremented, the relationship will be

$t = (i-1) * 4$

Rewriting:

$t = i*4 - 4$

Or:

$i = (t + 4) / 4$

Use this value of "i" to compute the new "t" as a function of the old "t".

```
t  :=  i  *  4
t  :=  [  (t + 4) / 4 ]  *  4
```

Rewriting:

```
t  :=  t + 4
```

Conclusion:

It is okay to replace   `t := i * 4`

by: `t := t + 4`

But don't forget to establish $t = i*4$ before the loop begins!

```
    .
    .
    .
i  :=  i  +  1
t  :=  i  *  4
    .
    .
    .
```

The Transformation

Before:

```
...
i := i + 1
t := i * 4
... A[t]...
...
```
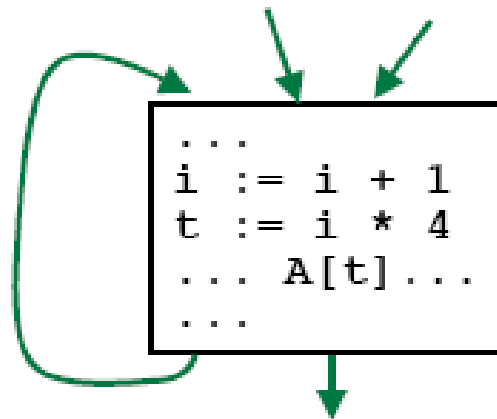
Must also check:
"t" and "i" are not changed
    elsewhere in the loop

Benefit:
The definition of i may
            become DEAD.
... *Eliminate it altogether!*

After:

```
t := i * 4
```

```
...
i := i + 1
t := t + 4
... A[t]...
...
```
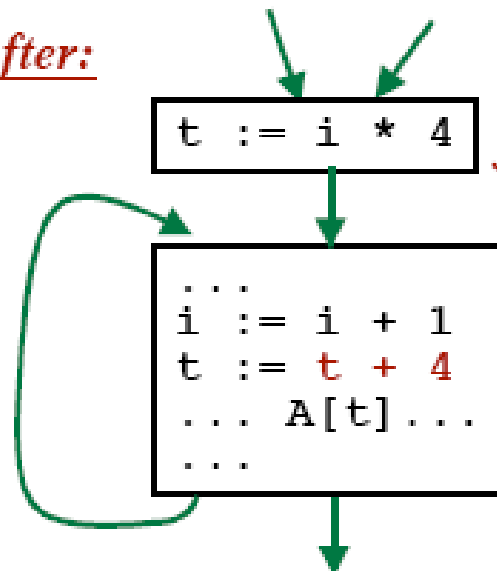
"Preheader"
A new block added
to "just before" the loop.

Definitions

"*Reduction in Strength*"

A costly operation is replaced by a cheaper operation.

```
t := i * 4
```

⬇

```
t := t + 4
```

"*Constant Folding*"

If all operands to an operator are constants...
evaluate the operator at compile-time.

```
t := 100 * 4
```

⬇

```
t := 400
```

Example

Our Example, so far...

$$B_1 \quad \begin{array}{l} \text{i} \; := \; \text{m} \; - \; 1 \\ \text{j} \; := \; \text{n} \\ \text{t1} \; := \; 4 \; * \; \text{n} \\ \text{v} \; := \; \text{A[t1]} \end{array}$$

$$B_2 \quad \begin{array}{l} \text{i} \; := \; \text{i} \; + \; 1 \\ \text{t2} \; := \; 4 \; * \; \text{i} \\ \text{t3} \; := \; \text{A[t2]} \\ \text{if} \; \text{t3} \; < \; \text{v} \; \text{goto} \; B_2 \end{array}$$

$$B_3 \quad \begin{array}{l} \text{j} \; := \; \text{j} \; - \; 1 \\ \text{t4} \; := \; 4 \; * \; \text{j} \\ \text{t5} \; := \; \text{A[t4]} \\ \text{if} \; \text{t5} \; > \; \text{v} \; \text{goto} \; B_3 \end{array}$$

$$B_4 \quad \text{if} \; \text{i} \; \geq \; \text{j} \; \text{goto} \; B_6$$

$$B_5 \quad \begin{array}{l} \text{A[t2]} \; := \; \text{t5} \\ \text{A[t4]} \; := \; \text{t3} \end{array}$$

$$B_6 \quad \begin{array}{l} \text{t14} \; := \; \text{A[t1]} \\ \text{A[t2]} \; := \; \text{t14} \\ \text{A[t1]} \; := \; \text{t3} \end{array}$$

Example


```
        ┌─────────────────┐
        │ i  := m - 1     │
        │ j  := n         │                    ┌ t2  :=  4 * i
     B₁ │ t1 := 4 * n     │                    { t4  :=  4 * j
        │ v  := A[t1]     │◄──────             └
        └─────────────────┘

                                    Induction Variables
        ┌─────────────────────┐
        │ i  := i + 1         │
        │ t2 := 4 * i         │
     B₂ │ t3 := A[t2]         │
        │ if t3 < v goto B₂   │
        └─────────────────────┘

        ┌─────────────────────┐
        │ j  := j - 1         │
        │ t4 := 4 * j         │
     B₃ │ t5 := A[t4]         │
        │ if t5 > v goto B₃   │
        └─────────────────────┘

     B₄ │ if i  ≥ j  goto B₆ │

     B₅                          B₆
        ┌──────────────┐            ┌──────────────────┐
        │              │            │ t14 := A[t1 ]    │
        │ A[t2] := t5  │            │ A[t2 ] := t14    │
        │ A[t4] := t3  │            │ A[t1 ] := t3     │
        └──────────────┘            └──────────────────┘
```

B_1: i := m - 1; j := n; t1 := 4 * n; v := A[t1]

B_2: i := i + 1; t2 := 4 * i; t3 := A[t2]; if t3 < v goto B_2

B_3: j := j - 1; t4 := 4 * j; t5 := A[t4]; if t5 > v goto B_3

B_4: if i ≥ j goto B_6

B_5: A[t2] := t5; A[t4] := t3

B_6: t14 := A[t1]; A[t2] := t14; A[t1] := t3

t2 := 4 * i
t4 := 4 * j

Induction Variables

Example

Induction Variables

```
B1
    i  := m - 1
    j  := n
    t1 := 4 * n
    v  := A[t1]

B2
    i  := i + 1
    t2 := t2 + 4
    t3 := A[t2]
    if t3 < v goto B2

B3
    j  := j - 1
    t4 := t4 - 4
    t5 := A[t4]
    if t5 > v goto B3

B4
    if t2 ≥ t4 goto B6

B5
    A[t2] := t5
    A[t4] := t3

B6
    t14 := A[t1 ]
    A[t2 ] := t14
    A[t1 ] := t3
```

```
t2 := 4 * i
t4 := 4 * j
```

Example

Example

B_1
```
i  := m - 1
j  := n
t1 := 4 * n
v  := A[t1]
```

```
t2 := 4 * i
t4 := 4 * j
```

**Dead Code
Elimination**

B_2
```
t2 := t2 + 4
t3 := A[t2]
if t3 < v goto B2
```

B_3
```
t4 := t4 - 4
t5 := A[t4]
if t5 > v goto B3
```

B_4 `if t2 ≥ t4 goto B6`

B_5
```
A[t2] := t5
A[t4] := t3
```

B_6
```
t14 := A[t1 ]
A[t2 ] := t14
A[t1 ] := t3
```

Where are the loops

Given:

A Control Flow Graph
(Each node is a Basic Block)

Goal:

Locate the loops.

Domination
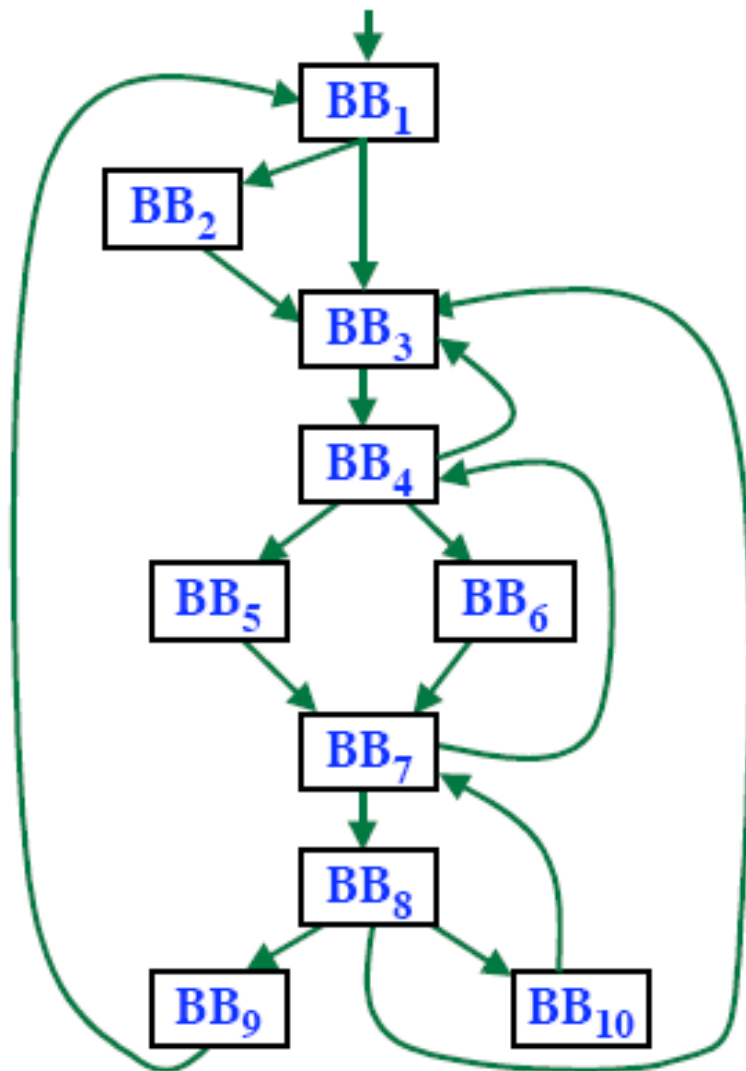
Domination
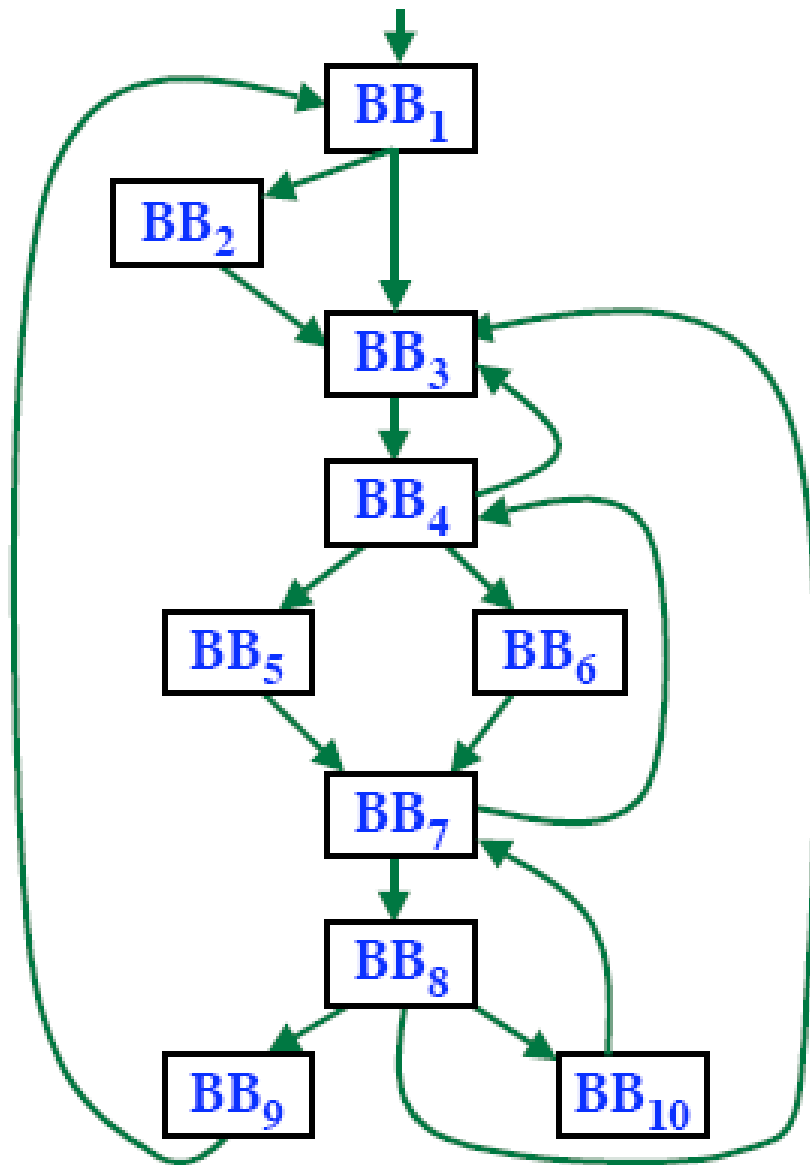
Node	Dominates
1	All
2	2
3	3 - 10
4	4 - 10
5	5
6	6
7	7,8,9,10
8	8,9,10
9	9
10	10

Dominator Tree

The Definition of Natural Loops

What is a loop anyway?

- **Must have a single entry point**

    The Header Node

    (The Header dominates all nodes in the loop.)

- **Must be a path back to the header.**

A loop is defined by an edge B→A such that A dom B.

Definition: Given such an edge B→A, A "*natural loop*" is the set of nodes...

- **Node A, and**
- **All nodes that can reach B without going through A.**
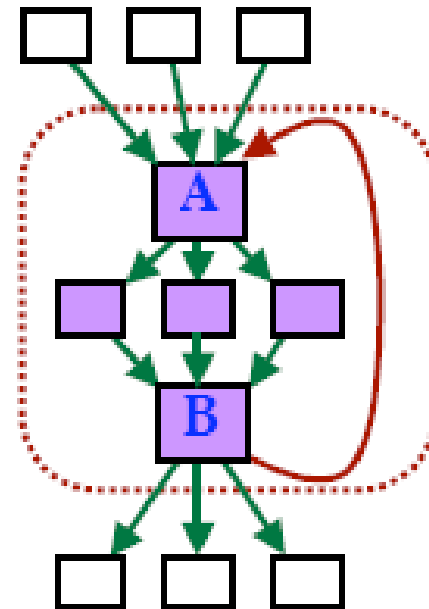
Where Are The Loops
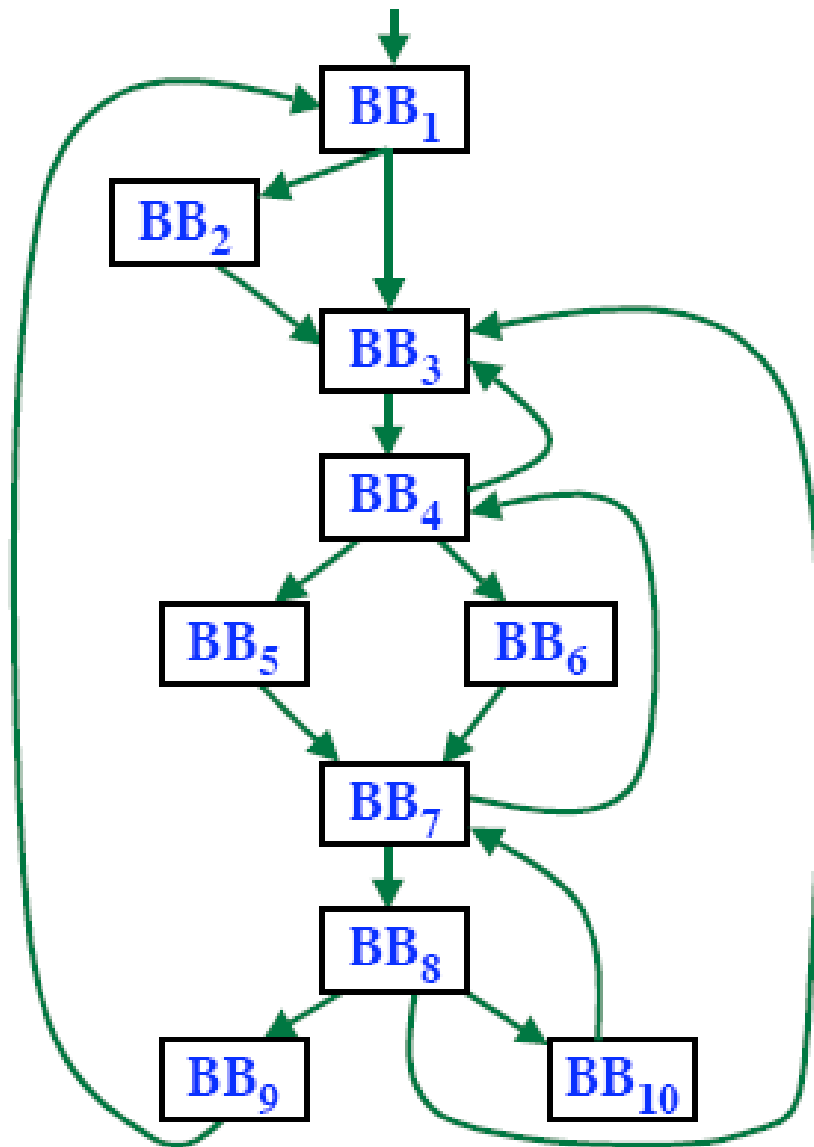
Where Are The Loops
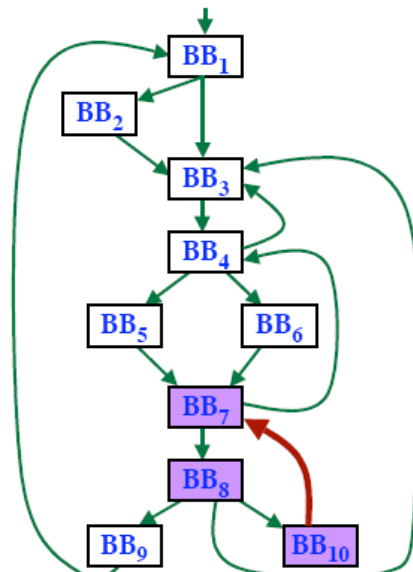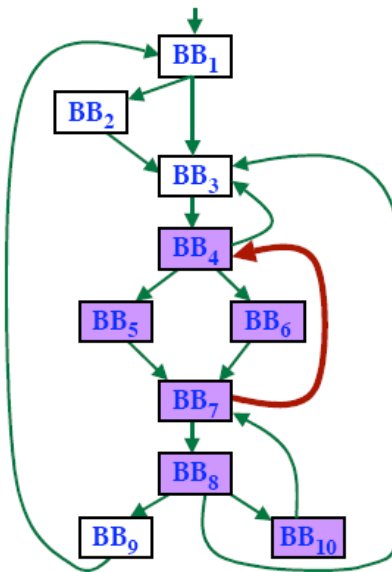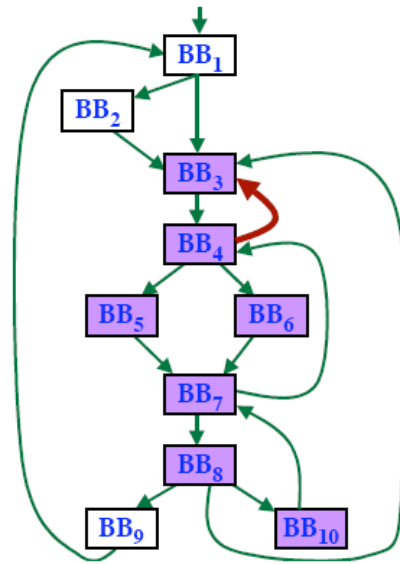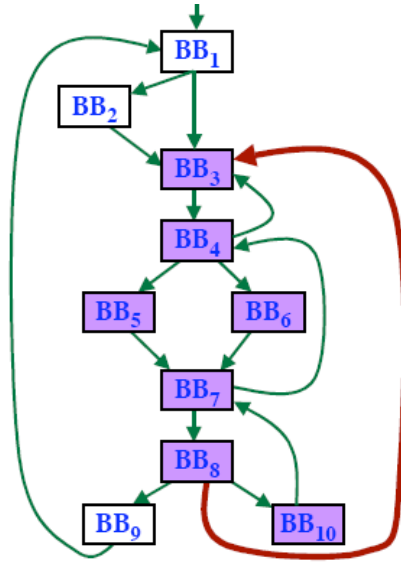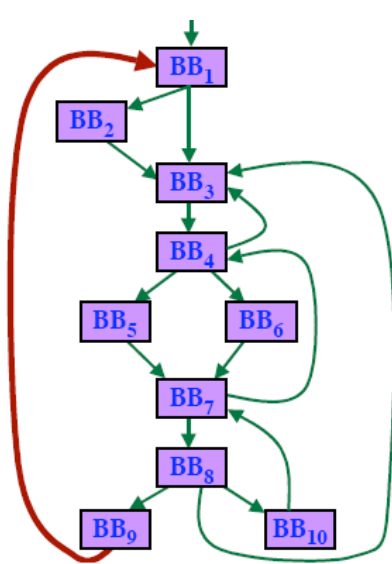
An Algorithm to Find a Natural Loop

Input: A Control Flow Graph
A Back-Edge, B→A

Output: Result = Set of nodes in the natural loop

```
Stack := empty
ResultSet := {A}
Insert (B)
while NotEmpty (Stack) do
   M := Pop (Stack)
   for each predecessor of P of M do
      Insert (P)
   endfor
endwhile
```

```
procedure Insert(X)
   if X is not in ResultSet then
      Add X to ResultSet
      Push X onto Stack
   endif
```

Inner/Outer Loops

A loop is a set of nodes.

Given two Natural Loops...
    Either...
- The loops are disjoint, or
- One loop is contained in (i.e., nested)
  within the other, or
- Both loops have the same header.

If two loops have the same header...
    They will be the same loop (same set of nodes)
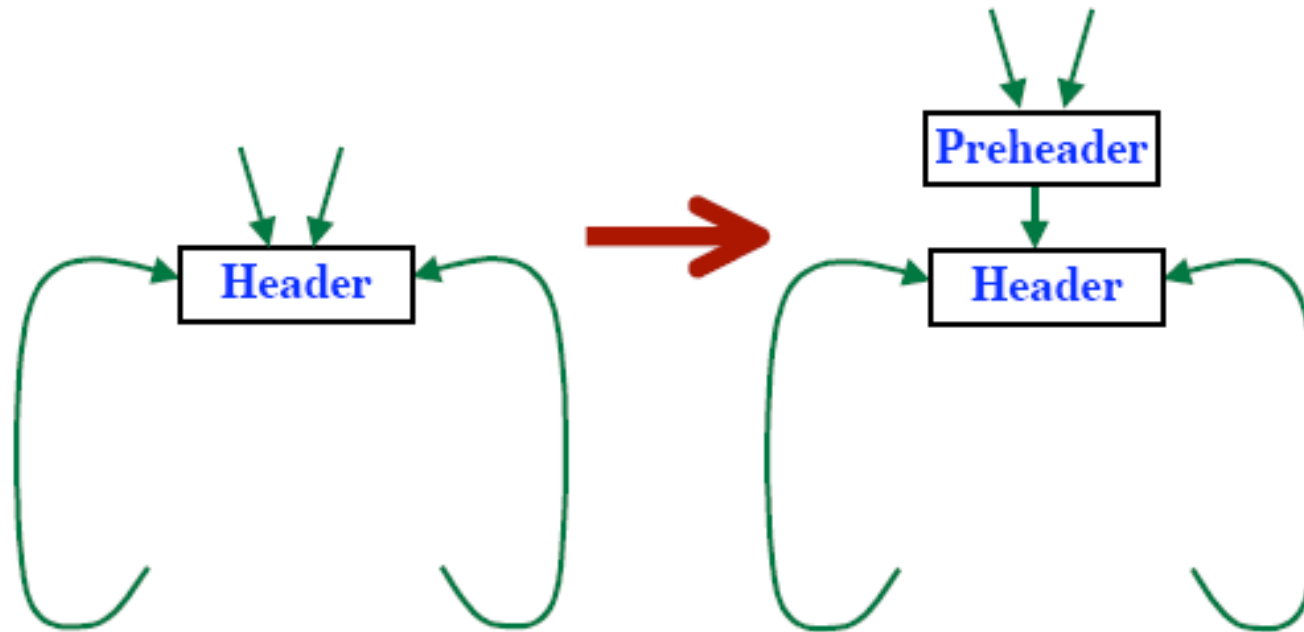
Loops with Multiple Back-Edge
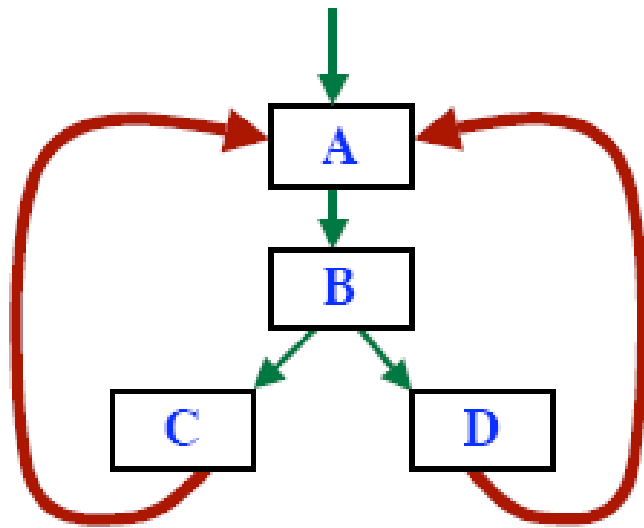

```
while (...) do
    ...A...
    ...B...
    if ... then
        ...C...
    else
        ...D...
    endif
endwhile
```

Which path is traversed most frequently?
  Undecideable...

Must treat as equally probable.

Loop Preheader

We can place loop-invariant computations in the preheader.

Reducible Control Flow Graphs

Definition:

In a reducible control flow graph, all loops have a single entry point.

Structured programming constructs
⇒ The control flow graph is reducible.
⇒ All loops are natural.

In a reducible flow graph...
We have only...
- **Forward Edges**
  These form an acyclic graph.
  All nodes can be reached via
  forward edges from initial node.
- **Back Edges**
  The HEAD dominates the TAIL
- **No "Cross Edges"**