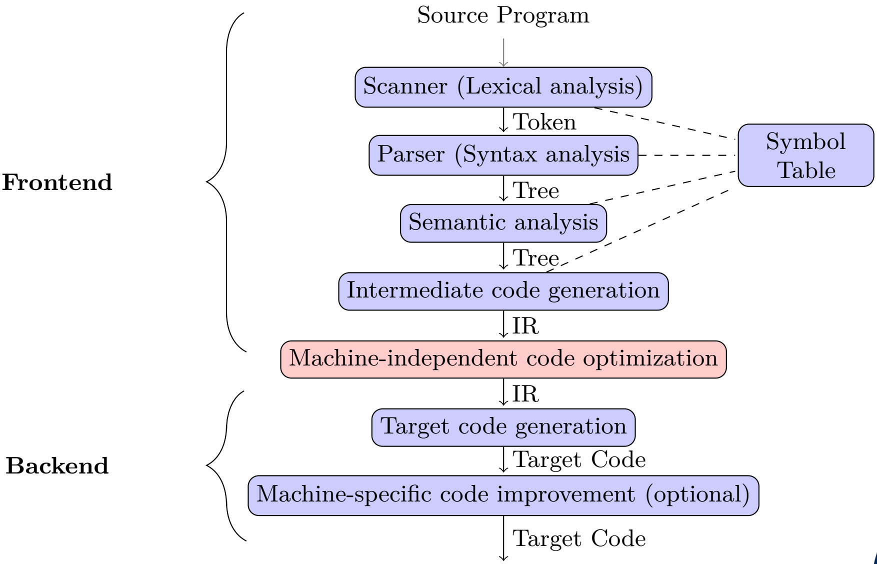
Optimizations

Lecture 21

Section 8.5, 9.1

Rokan Uddin Faruqui

Associate Professor
Dept of Computer Science and Engineering
University of Chittaong, Bangladesh
Email: *rokan@cu.ac.bd*


Source Program
Scanner (Lexical analysis)
Token
Parser (Syntax analysis
Tree
Semantic analysis
Tree
Intermediate code generation
IR
Machine-independent code optimization
IR
Target code generation
Target Code
Machine-specific code improvement (optional)
Target Code
Symbol Table
Frontend
Backend

Optimizations

local optimizations – within a basic block,

global optimizations – between basic blocks.

Outline

DAG for basic blocks

Three-address code to DAG structure:

- One leaf per initial value.
- One node per statement, with an edge to the node representing the operand value(s).
- Inner node is labelled by operator, and list of variables, which computes the value.
- Output nodes are those labelled with live exit variables.

DAG for basic blocks

Three-address code to DAG structure:

- One leaf per initial value.
- One node per statement, with an edge to the node representing the operand value(s).
- Inner node is labelled by operator, and list of variables, which computes the value.
- Output nodes are those labelled with live exit variables.

DAG for basic blocks

Three-address code to DAG structure:

- One leaf per initial value.
- One node per statement, with an edge to the node representing the operand value(s).
- Inner node is labelled by operator, and list of variables, which computes the value.
- Output nodes are those labelled with live exit variables.

DAG for basic blocks

Three-address code to DAG structure:

- One leaf per initial value.
- One node per statement, with an edge to the node representing the operand value(s).
- Inner node is labelled by operator, and list of variables, which computes the value.
- Output nodes are those labelled with live exit variables.

DAG optimization uses

- Local common subexpressions.
- Dead code elimination.
- Apply algebraic simplifications.
- Reorder statements that do not depend on one other.

Local common subexpressions

- Eliminate instructions that has already been computed.

$$
\begin{aligned}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= a - d
\end{aligned}
$$

Local common subexpressions

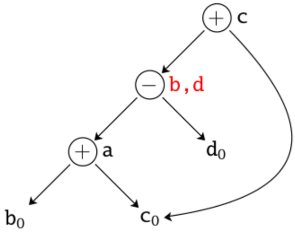
- Eliminate instructions that has already been computed.

$$
\begin{aligned}
{\color{blue} a} &{\color{blue} = b + c} \\
{\color{red} b} &{\color{red} = a - d} \\
{\color{blue} c} &{\color{blue} = b + c} \\
{\color{red} d} &{\color{red} = a - d}
\end{aligned}
$$

Local common subexpressions

- Eliminate instructions that has already been computed.

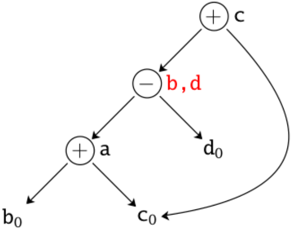
$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d$$



Local common subexpressions

- Eliminate instructions that has already been computed.

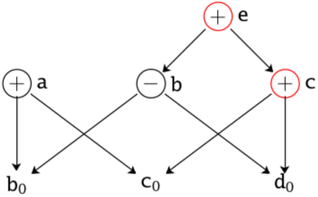
$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d$$



Dead code elimination

- Eliminate instructions that compute values that are never used

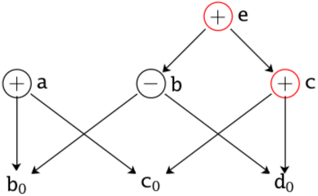
$$a = b + c$$
$$b = b - d$$
$$c = c + d$$
$$e = b + c$$



Dead code elimination

- Eliminate instructions that compute values that are never used

$$a = b + c$$
$$b = b - d$$
$$c = c + d$$
$$e = b + c$$



Algebraic Identities

- Apply algebraic simplification rules.

$$x + 0 = 0 + x = x$$

$$x - 0 = x$$

$$x \times 1 = 1 \times x = x$$

$$x/1 = x$$

$$2 + 2 = 4 (\text{constant folding})$$

$$\text{Expensive} \text{Cheaper}$$

$$x^2 = x \times x (\text{lib function})$$

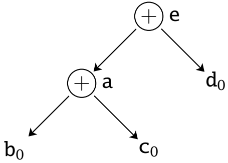
$$2 \times x = x + x$$

$$x/2 = x \times 0.5$$

Algebraic Identities

- **Commutativity** with local common subexpressions.
- **Associativity** with composite expressions....

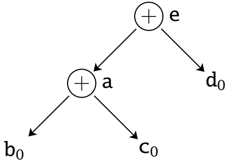
$$a = c + b$$
$$e = c + d + b$$



Algebraic Identities

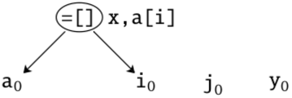
- **Commutativity** with local common subexpressions.
- **Associativity** with composite expressions....

$$a = c + b$$
$$e = c + d + b$$



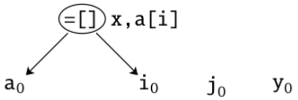
Array References

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$



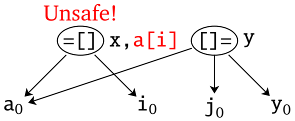
Array References

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$



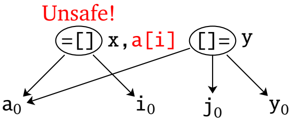
Array References

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$



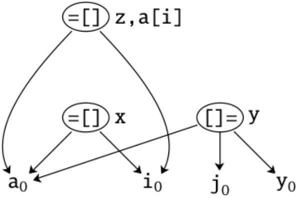
Array References

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$



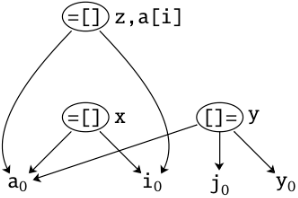
Array References

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$



Array References

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$



Remarks

- The entire memory is a single array!
- In C: no check of bounds of arrays, so memory can be arbitrarily updated.

Outline

1. Local Optimization

2. Global Optimization

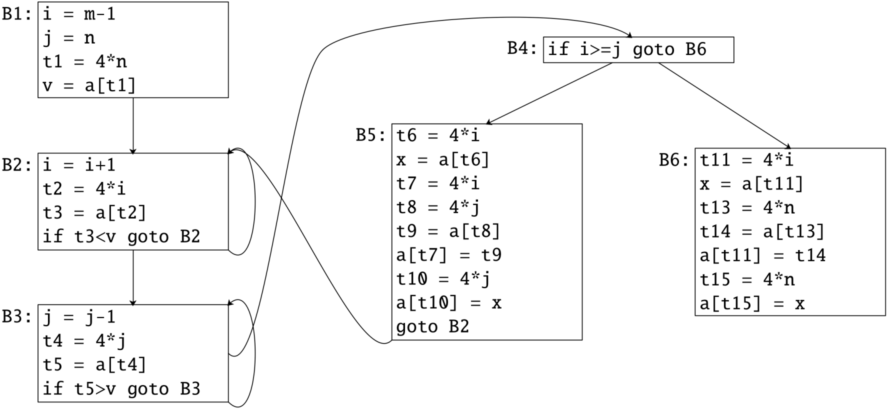
Example: quicksort

```
void quicksort(int a[], int m, int n) {
int i, j, v, x; if (n <= m) return;
/*fragment starts here*/
i = m-1; j = n; v = a[n];
while (1) {
  do i = i+1; while (a[i] < v);
  do j = j-1; while (a[j] > v);
  if (i >= j) break;
     x = a[i]; a[i] = a[j]; a[j] = x;
   }
  x = a[i]; a[i] = a[n]; a[n] = x;
/*fragment end here*/
  quicksort(a,m,j); quicksort(a,i+1,n);
}
```

Three-address code for fragments of qsort

```
(1)   i  = m-1
(2)   j  = n
(3)   t1  = 4*n
(4)   v  = a[t1]
(5)   i  = i+1
(6)   t2  = 4*i
(7)   t3  = a[t2]
(8)   if t3<v goto (5)
(9)   j  = j-1
(10)  t4  = 4*j
(11)  t5  = a[t4]
(12)  if t5>v goto (9)
(13)  if  i>=j  goto  (23)
(14)  t6  = 4*i
(15)  x  = a[t6]
(16)  t7  = 4*i
(17)  t8  = 4*j
(18)  t9  = a[t8]
(19)  a[t7]  = t9
(20)  t10  = 4*j
(21)  a[t10]  = x
(22)  goto (5)
(23)  t11  = 4*i
(24)  x  = a[t11]
(25)  t12  = 4*i
(26)  t13  = 4*n
(27)  t14  = a[t13]
(28)  a[t12]  = t14
(29)  t15  = 4*n
(30)  a[t15]  = x
```

Basic blocks for quick sort: first try



Local Common Subexpression 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
```

Local Common Subexpression 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
```

$\Rightarrow$

B5:
```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

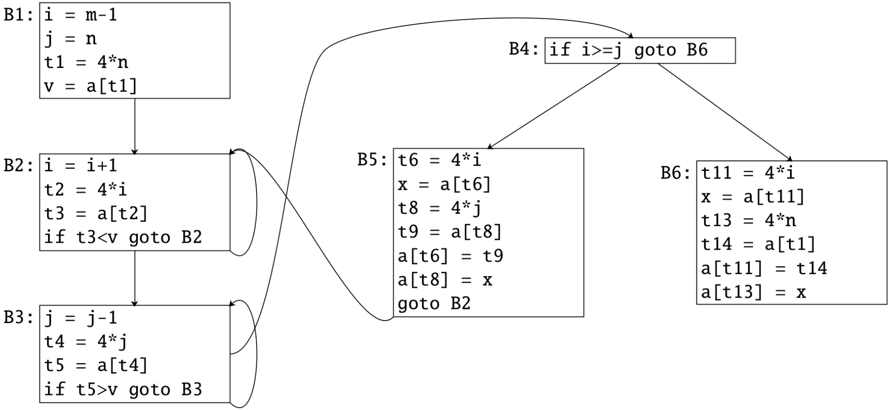
Local Common Subexpression Elimination

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
```

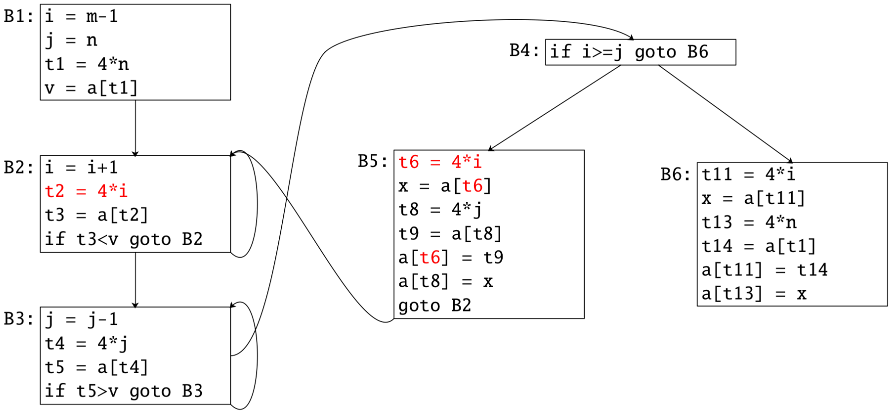
$\Rightarrow$

B5:
```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

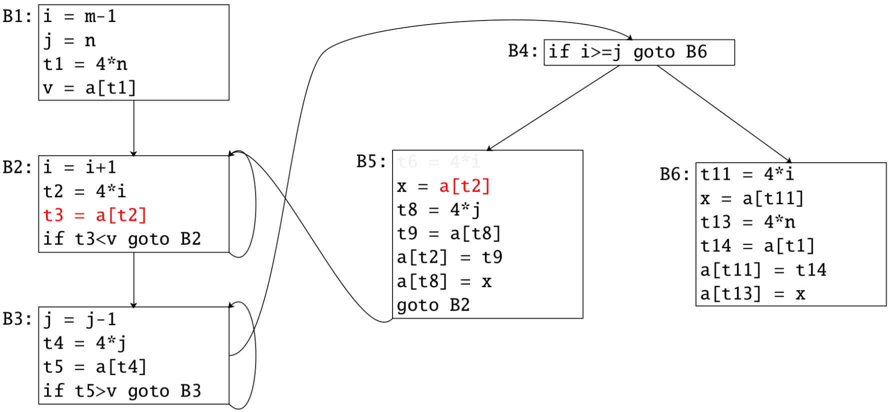
Global Common Subexpression Elimination



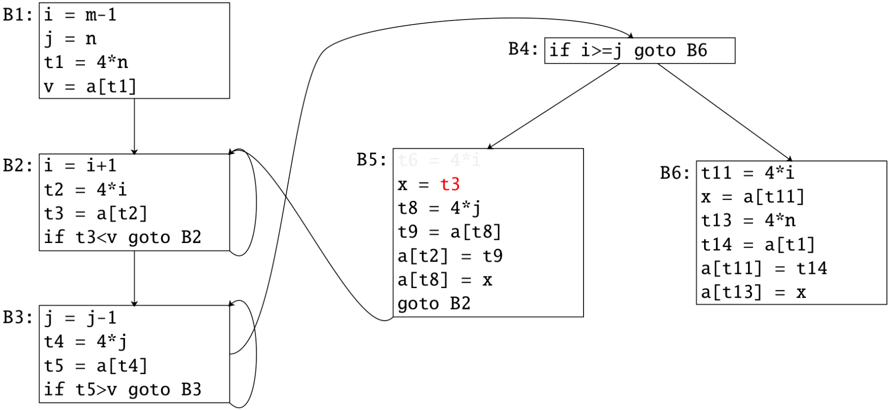
Global Common Subexpression Elimination



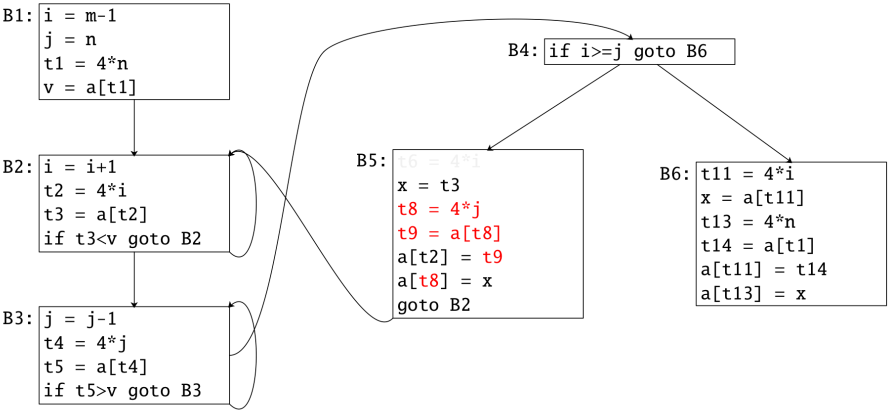
Global Common Subexpression Elimination



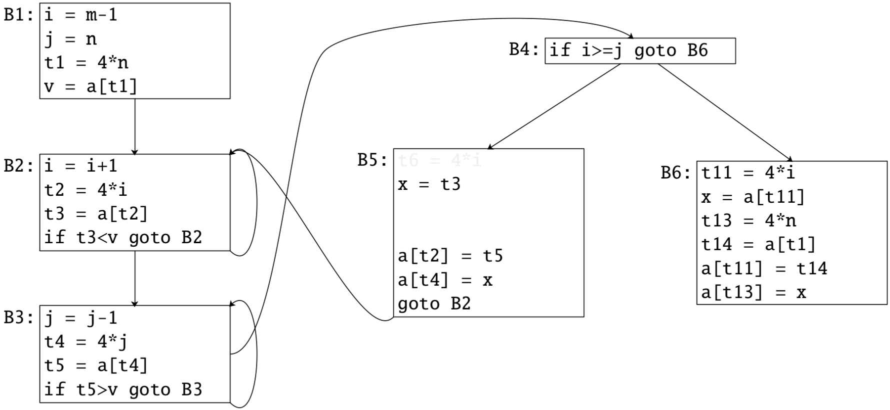
Global Common Subexpression Elimination



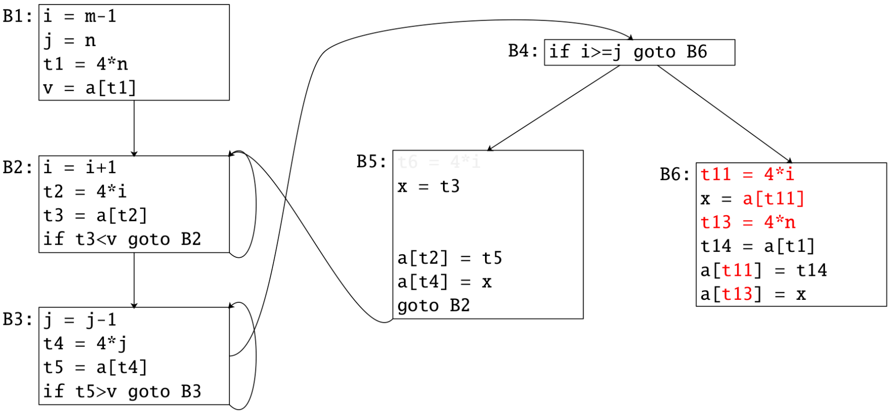
Global Common Subexpression Elimination



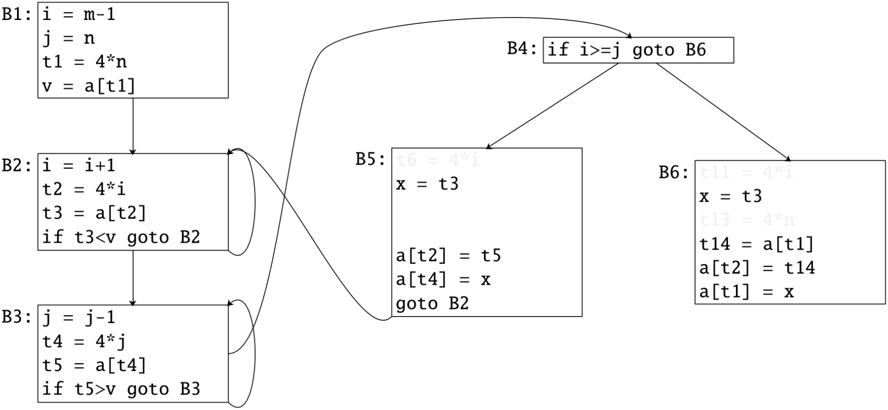
Global Common Subexpression Elimination



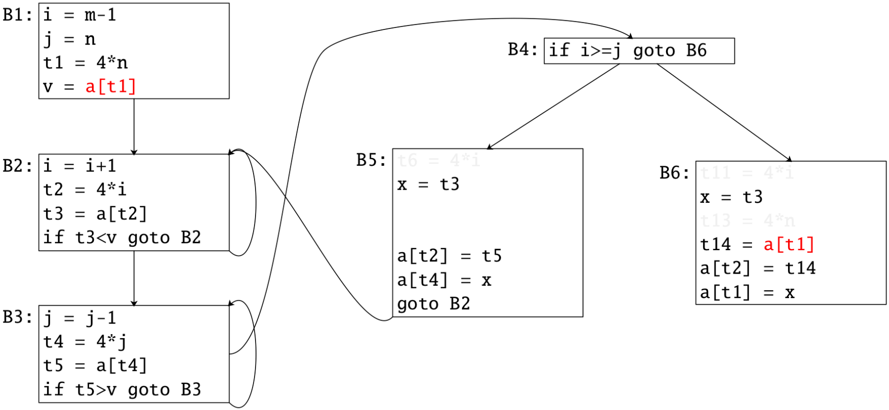
Global Common Subexpression Elimination



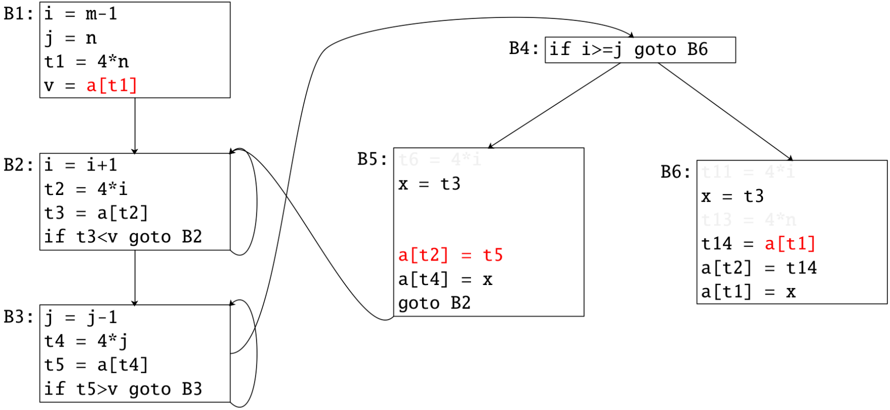
Global Common Subexpression Elimination



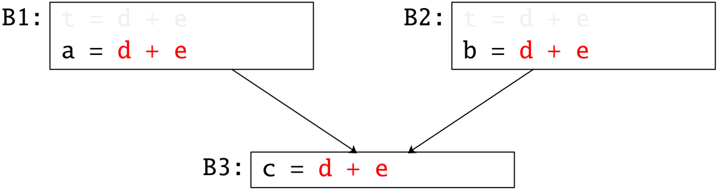
Global Common Subexpression Elimination



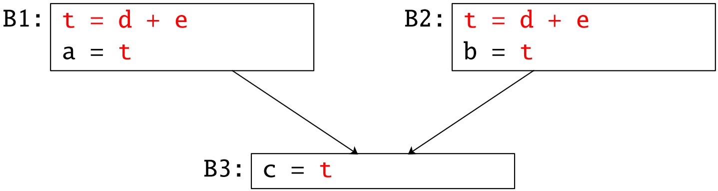
Global Common Subexpression Elimination



Copy Propagation



Copy Propagation



Dead-code Elimination

if (debug) print

debug = FALSE

Code Motion

```
while (i <= limit-2) /*not changing limit*/
```

becomes

```
t = limit-2;
while (i <= t) /*not changing limit or t*/
```

Summary

Reduce redundancy but preserve semantics!

- Global Common Subexpressions.
- Copy Propagation.
- Dead-code Elimination.
- Code Motion.