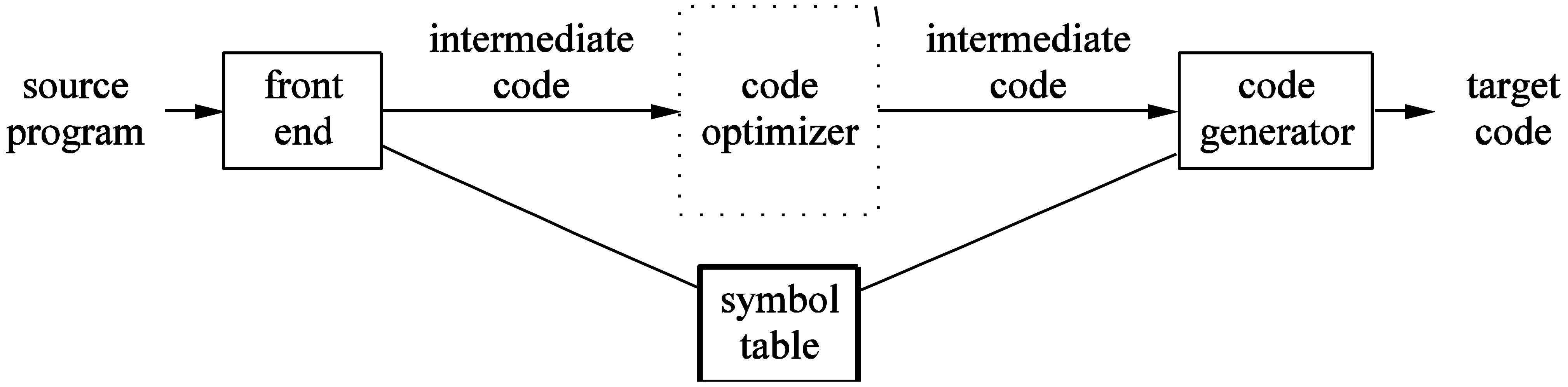
Code Optimization



Optimizations

[GCC/LLVM bugs: **1,622** (total) / **1,031** (fixed)]
[Reports: GCC (link1, link2, link3, link4, link5), LLVM (link1, link2, link3, link4, link5)]

[GCC/LLVM bugs: **1,634** (total) / **1,076** (fixed)]
[Reports: GCC (link1, link2, link3, link4, link5), LLVM (link1, link2, link3, link4, link5)]

[Recent CompCert bug reports: **31** (total) / **27** (fixed)]
[Reports: link]

[Recent Scala and Dotty bug reports: **42** (total) / **17** (fixed)]
[Reports: link]

[Recent ICC bug reports: **35** (total) / **unknown** (fixed)]
[Reports: link]

Optimizations

For languages like C and C++ there are three granularities of optimizations

Complexity

1. Local optimizations
   - Apply to a basic block in isolation
2. Global optimizations
   - Apply to a control-flow graph (method body) in isolation
3. Inter-procedural optimizations
   - Apply across method boundaries

powerful

Most compilers do (1), many do (2), few do (3)

Cost of Optimizations

- In practice, a conscious decision is made not to implement the fanciest optimization known

- Why?
  - Some optimizations are hard to implement
  - Some optimizations are costly in compilation time
  - Some optimizations have low benefit, no theoretic guarantee
  - Many fancy optimizations are all three!

Goal: Maximum benefit for minimum cost

Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
- Techniques
  1. Algebraic Simplification
  2. Constant Folding
  3. Dead Code Elimination
  4. Common Subexpression Elimination
  5. Copy Propagation
- Each local optimization does little by itself
- Typically optimizations interact, performing one optimization enables another
- Optimizing compilers repeat optimizations until no improvement is possible

Basic Blocks

- A basic block is a sequence of statements such that:
  - Flow of control enters at start
  - Flow of control leaves at end
  - No possibility of halting or branching except at end
- Each basic block has a first statement known as the "leader" of the basic block
- A name is "live" at a given point if its value will be used again in the program

- Useful for local optimization

Transformations on Basic Blocks

- A basic block computes a set of expressions
  - The expressions are the values of names that are live on exit from the block
  - Two basic blocks are equivalent if they compute the same set of expressions
- Certain transformations can be applied without changing the computed expressions of a block
  - An optimizer uses such transformations to improve running time or space requirements of a program

Algebraic Simplification

- Some statements can be deleted
  - **x := x + 0**
  - **x := x * 1**
- Some statements can be simplified
  - **x := x * 0** $\Rightarrow$ **x := 0**
  - **x := x * 2** $\Rightarrow$ **x := x + x**
  - **x := x ** 2** $\Rightarrow$ **x := x * x**
  - **x := x * 8** $\Rightarrow$ **x := x << 3**
  - **x := x * 15** $\Rightarrow$ **t := x << 4; x := t – x**

(on some machines << is faster than *, and + is faster than *; but not on all!)

Constant Folding

- Operations on constants can be computed at compile time
  - If there is a statement x := y op z and y and z are constants
  - Then y op z can be computed at compile time

- Eg.
  - **x := 2+3** **⇒ x := 5**
  - **x := 2*3** **⇒ x := 6**
  - **if 2 < 0 jump L** **⇒ if false jump L** **⇒ delete**

Dead Code Elimination

- Eliminate unreachable basic blocks:
  - Code that is unreachable from the initial block
  - Is it possible?

- Removing unreachable code makes the program smaller
  - and sometimes also faster

Common Subexpression Elimination

- Common subexpression elimination (DAG):

```
a := b + c          a := b + c
b := a - d          b := a - d
c := b + c    ⇒     c := b + c
d := a - d          d := b
```

- SSA form basic block without DAG

x := y + z		x := y + z
…	⇒	… no change of x,y,z
w := y + z		w := x

- Take care of points, array, function calls

Copy Propagation

- If w := x appears in a block, replace subsequent uses of w with uses of x
  - Assumes SSA form
- Example:

b := z + y		b := z + y
a := b	⇒	a := b
x := 2 * a		x := 2 * b

- Only useful for enabling other optimizations
  - Constant folding
  - Dead code elimination

Examples

- Copy Propagation and Constant Folding

a := 5		a := 5
x := 2 * a	⇒	x := 10
y := x + 6		y := 16
t := x * y		t := x << 4

- Copy Propagation and Dead Code Elimination and Algebraic Simplification

x := z + y		x := z + y		x := z + y		x := z + y
a := x	⇒	a := x	⇒	x := 2 * x	⇒	x := x + x
x := 2 * a		x := 2 * x				

Assume (a is not used anywhere else)

Applying Local Optimizations

- Each local optimization does little by itself
- Typically optimizations interact, performing one optimization en
- Optimizing compilers repeat optimizations until no improvemen

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

Algebraic optimization

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

Copy propagation

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

Constant folding

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

Common subexpression elimination

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

Copy propagation

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

a := x * x
~~b := 3~~
~~c := x~~
~~d := a~~
~~e := 6~~
f := a + a
g := 6 * f

Dead code elimination

Review

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
- Techniques
  1. Algebraic Simplification
  2. Constant Folding
  3. Dead Code Elimination
  4. Common Subexpression Elimination
  5. Copy Propagation
- Each local optimization does little by itself
- Typically optimizations interact, performing one optimization enables another
- Optimizing compilers repeat optimizations until no improvement is possible

Peephole Optimizations on Assembly Code

- These optimizations work on intermediate code
  - Target code
  - But they can be applied on IR

- Peephole optimization is effective for improving assembly code
  - The "peephole" is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent one (but faster)

$$i_1, \ldots, i_n \rightarrow j_1, \ldots, j_m$$

Peephole Optimizations on Assembly Code

Eg. mov \$a \$b, mov \$b \$a → move \$a \$b

if move \$b \$a is not the target of a jump

- Flow-of-Control Optimizations

```
    goto L1
    …
L1: goto L2
```

⟹

```
    goto L2
    …
L1: goto L2
```

- If there are no other jumps to `L1` and `L1` is preceded by an unconditional jump, the statement at `L1` can be eliminated
- Many of the basic block optimizations can be cast as peephole optimizations
- As for local optimizations, peephole optimizations must be applied repeatedly for maximum effect

Local Optimizations: Notes

- Intermediate code is helpful for many optimizations

- Many simple optimizations can still be applied on assembly language

- “Program optimization” is grossly misnamed
  - Code produced by “optimizers” is not optimal in any reasonable sense
  - “Program improvement” is a more appropriate term

Optimizations

For languages like C and C++ there are three granularities of optimizations

Complexity

1. Local optimizations
   - Apply to a basic block in isolation
2. Global optimizations
   - Apply to a control-flow graph (method body) in isolation
3. Inter-procedural optimizations
   - Apply across method boundaries

powerful

Most compilers do (1), many do (2), few do (3)

Local vs. Global

- Local optimization involve statements within a single basic block
- All other optimizations are called global optimizations, e.g., peephole
- Local transformations are generally performed first
- Many types of transformations can be performed either locally or globally
- Global optimizations
  - ✓ Data-flow analysis
  - ✓ Intra-procedural analysis: across basic blocks, but not procedures
  - ✓ Inter-procedural analysis: across procedures

Global Optimization

- Global optimizations
  1. Global common subexpressions
  2. Copy Propagation
  3. Dead-code Elimination
  4. Code motion
  5. Induction Variables and Reduction in Strength

Quicksort in C

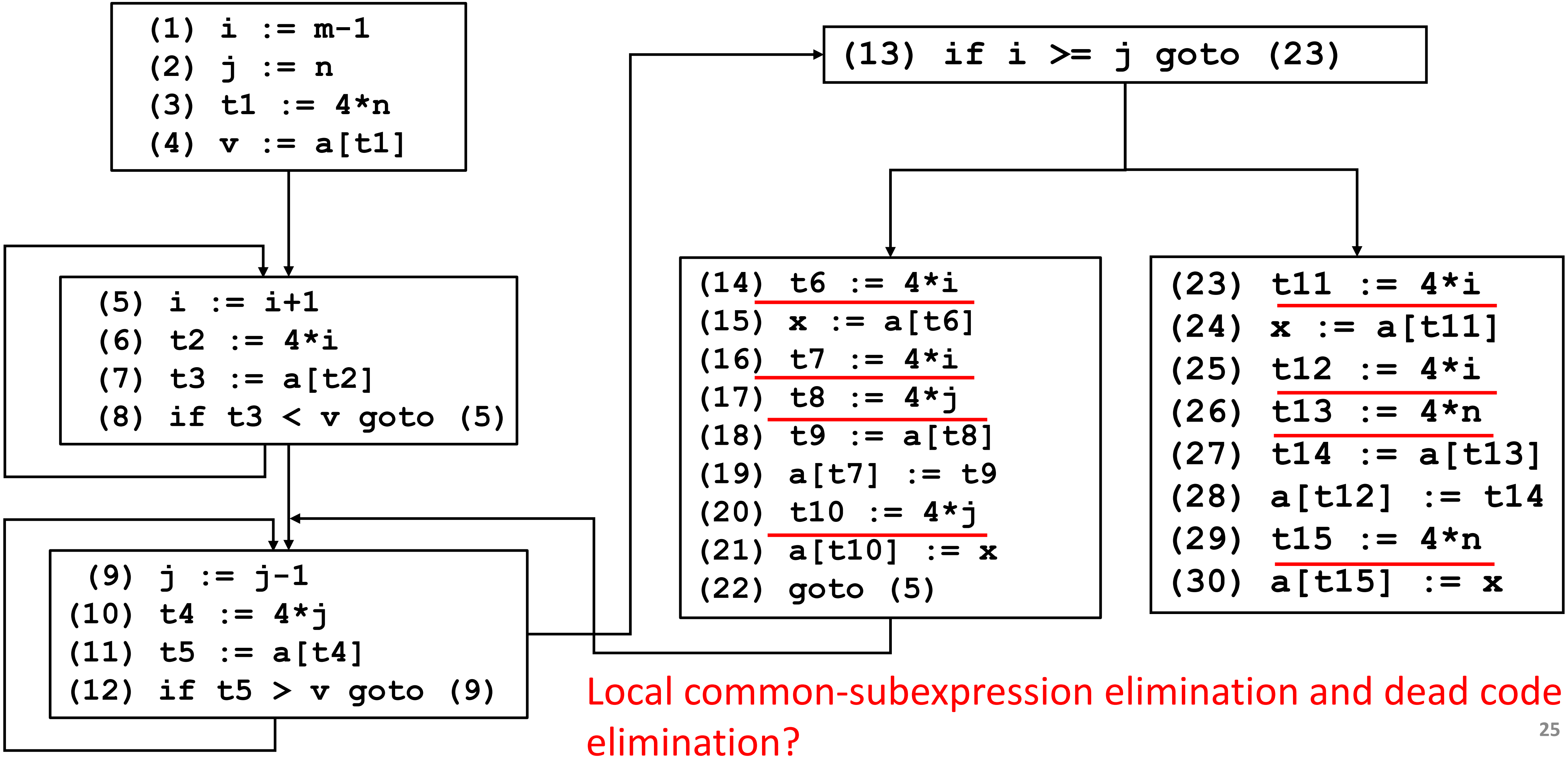
```
void quicksort(int m, int n) {
     int i, j, v, x;
     if (n <= m) return;
     /* Start of partition code */
     i = m-1; j = n; v =a[n];
     while (1) {
       do i = i+1; while (a[i] < v);
       do j = j-1; while (a[j] > v);
       if (i >= j) break;
       x = a[i]; a[i] = a[j]; a[j] = x;
     }
     x = a[i]; a[i] = a[n]; a[n] = x;
     /* End of partition code */
     quicksort(m, j); quicksort(i+1, n);
}
```

Partition in Three-Address Code

```
 (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]
```

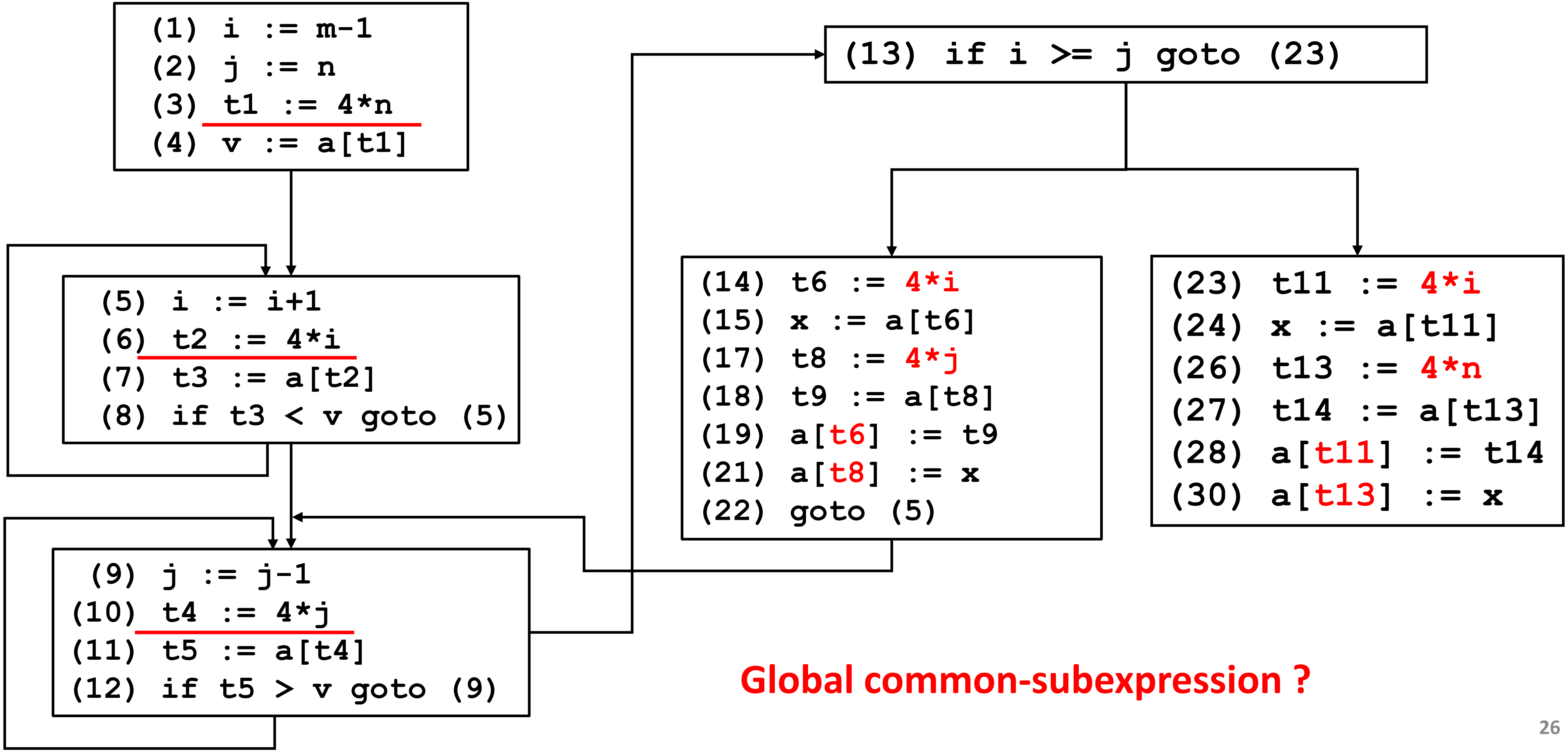
```
(16) t7 := 4*i
(17) t8 := 4*j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4*j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*n
(30) a[t15] := x
```

Control-flow graph



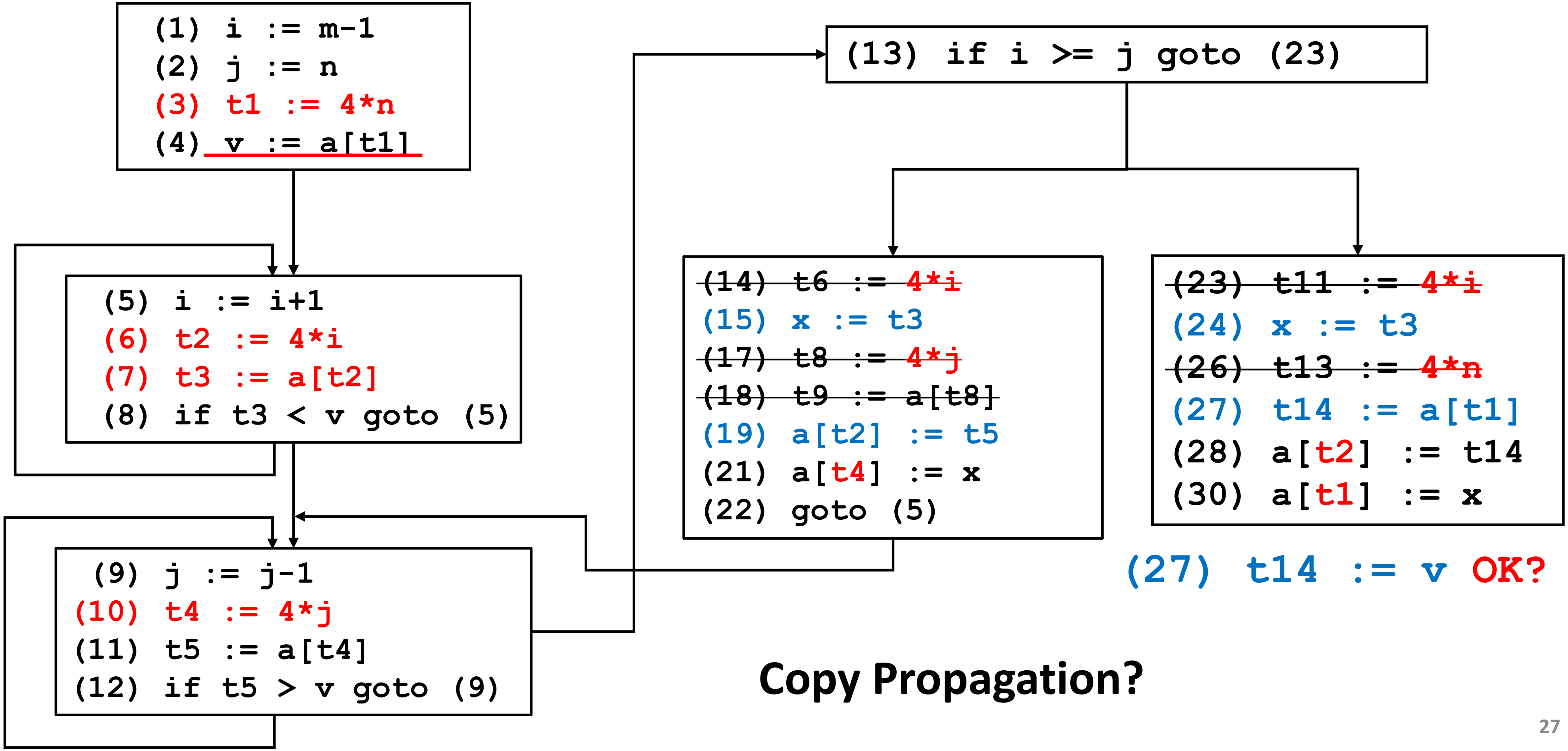
Local common-subexpression elimination and dead code elimination?

Local common-subexpression elimination and dead code elimination



Global common-subexpression ?

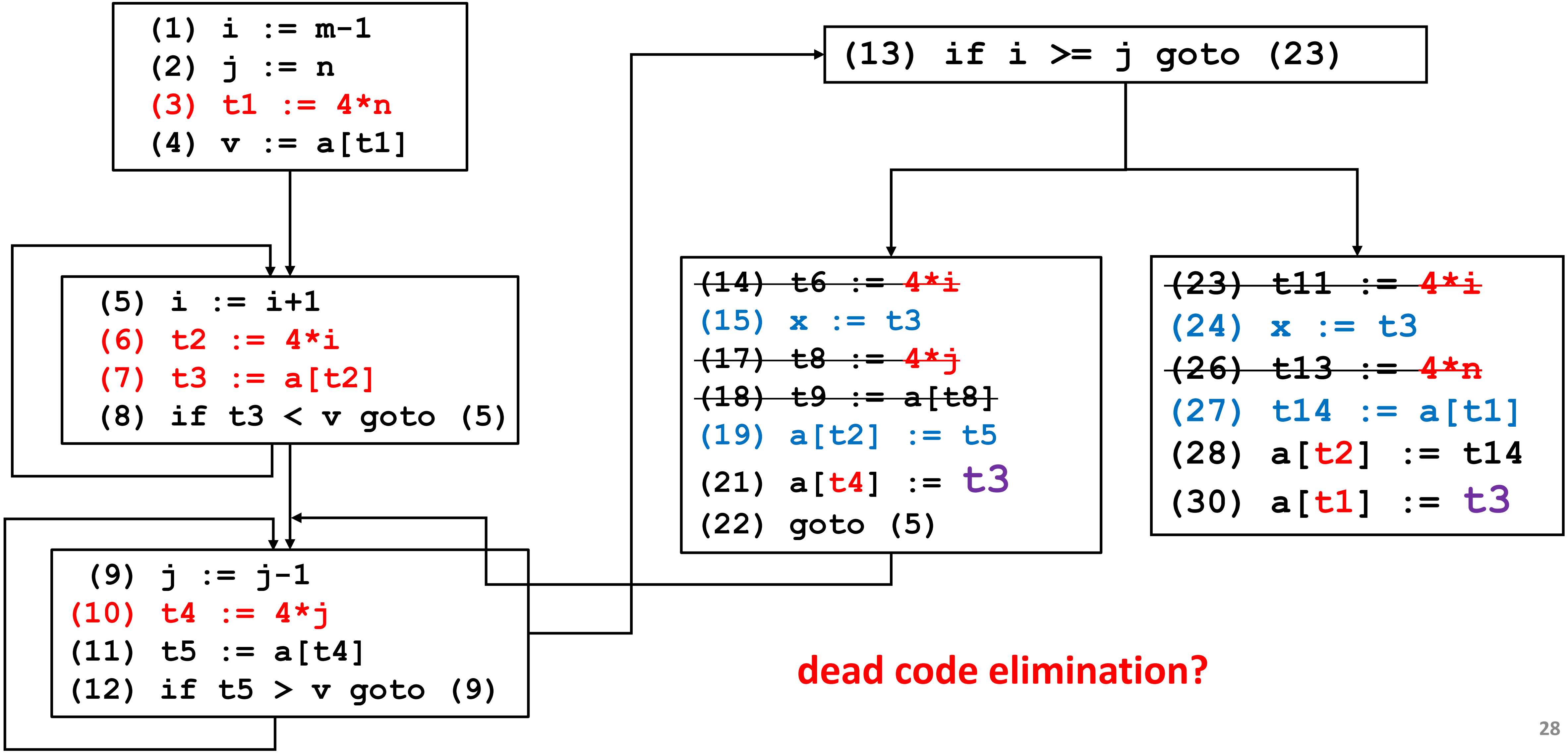
Global common-subexpression



(27) t14 := v OK?

Copy Propagation?

Copy Propagation



dead code elimination?

Dead code elimination

(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 := t3~~
~~(17) t8 := 4*j~~
~~(18) t9 := a[t8]~~
(19) a[t2] := t5
(21) a[t4] := t3
(22) goto (5)

~~(23) t11 := 4*i~~
~~(24) x := t3~~
~~(26) t13 := 4*n~~
(27) t14 := a[t1]
(28) a[t2] := t14
(30) a[t1] := t3

Code motion?
Move invariant to outside of loop

Induction variables and Reduction in strength

```
(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  := t3~~
~~(17) t8  := 4*j~~
~~(18) t9  := a[t8]~~
(19) a[t2]  := t5
(21) a[t4]  := t3
(22) goto (5)
```

```
~~(23) t11  := 4*i~~
~~(24) x  := t3~~
~~(26) t13  := 4*n~~
(27) t14  := a[t1]
(28) a[t2]  := t14
(30) a[t1]  := t3
```

```
(9) j  := j-1
(10) t4  := 4*j   => t4  :=t4-4

(5) i  := i+1
(6) t2  := 4*i    => t2  :=t2+4
```

Induction variables and Reduction in strength

(1) i := m-1
(2) j := n
(3) t1 := 4*n
(4) v := a[t1]

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

(5) i := i+1
(6) t2 := t2+4
(7) t3 := a[t2]
(8) if t3 < v goto (5)

(9) j := j-1
(10) t4 := t4-4
(11) t5 := a[t4]
(12) if t5 > v goto (9)

(13) if i >= j goto (23)

~~(14) t6 := 4*i~~
~~(15) x := t3~~
~~(17) t8 := 4*j~~
~~(18) t9 := a[t8]~~
(19) a[t2] := t5
(21) a[t4] := t3
(22) goto (5)

~~(23) t11 := 4*i~~
~~(24) x := t3~~
~~(26) t13 := 4*n~~
(27) t14 := a[t1]
(28) a[t2] := t14
(30) a[t1] := t3

(9) j := j-1
(10) t4 := 4*j => t4 :=t4-4

(5) i := i+1
(6) t2 := 4*i => t2 :=t2+4

Induction variables and Reduction in strength

(1) i := m-1
(2) j := n
(3) t1 := 4*n
(4) v := a[t1]

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

(5) i := i+1
(6) t2 := t2+4
(7) t3 := a[t2]
(8) if t3 < v goto (5)

(9) j := j-1
(10) t4 := t4-4
(11) t5 := a[t4]
(12) if t5 > v goto (9)

(13) if i >= j goto (23)

~~(14) t6 := 4*i~~
~~(15) x := t3~~
~~(17) t8 := 4*j~~
~~(18) t9 := a[t8]~~
(19) a[t2] := t5
(21) a[t4] := t3
(22) goto (5)

~~(23) t11 := 4*i~~
~~(24) x := t3~~
~~(26) t13 := 4*n~~
(27) t14 := a[t1]
(28) a[t2] := t14
(30) a[t1] := t3

(10) t4 := 4*j => t4:=t4-4

(6) t2 := 4*i => t2:=t2+4

i>=j iff t2>=t4

Induction variables and Reduction in strength

(1) i := m-1
(2) j := n
(3) t1 := 4*n
(4) v := a[t1]

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

~~(5) i := i+1~~
(6) t2 := t2+4
(7) t3 := a[t2]
(8) if t3 < v goto (5)

~~(9) j := j-1~~
(10) t4 := t4-4
(11) t5 := a[t4]
(12) if t5 > v goto (9)

(13) if t2 >= t4 goto (23)

~~(14) t6 := 4*i~~
~~(15) x := t3~~
~~(17) t8 := 4*j~~
~~(18) t9 := a[t8]~~
(19) a[t2] := t5
(21) a[t4] := t3
(22) goto (5)

~~(23) t11 := 4*i~~
~~(24) x := t3~~
~~(26) t13 := 4*n~~
(27) t14 := a[t1]
(28) a[t2] := t14
(30) a[t1] := t3

(10) t4 := 4*j => t4:=t4-4

(6) t2 := 4*i => t2:=t2+4

i>=j iff t2>=t4