Compiler Design

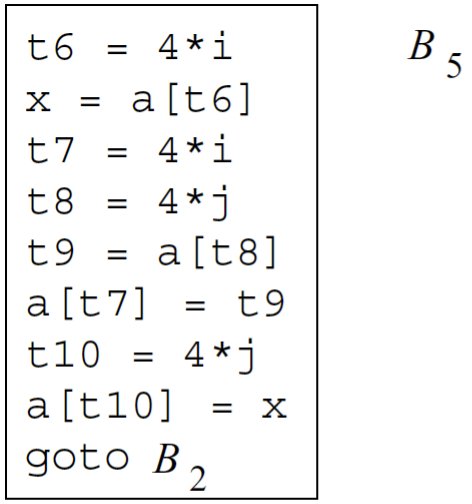
Fatemeh Deldar

Isfahan University of Technology

1403-1404

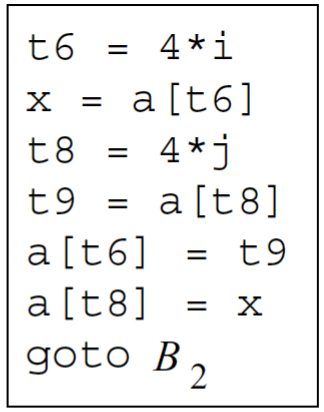
Global Optimization

- **Semantics-Preserving Transformations**
  - A program will include several calculations of the same value
  - **Example:** Local common-subexpression elimination



B_5

(a) Before.

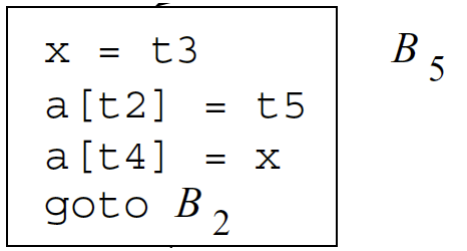


B_5

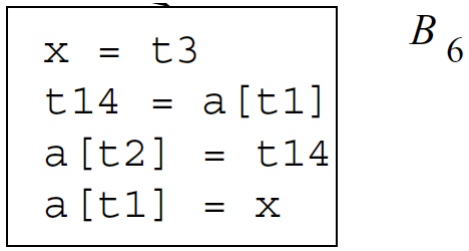
(b) After.

Global Optimization

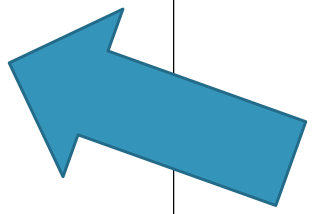
- **Example:** Eliminating both global and local common subexpressions from blocks B5 and B6

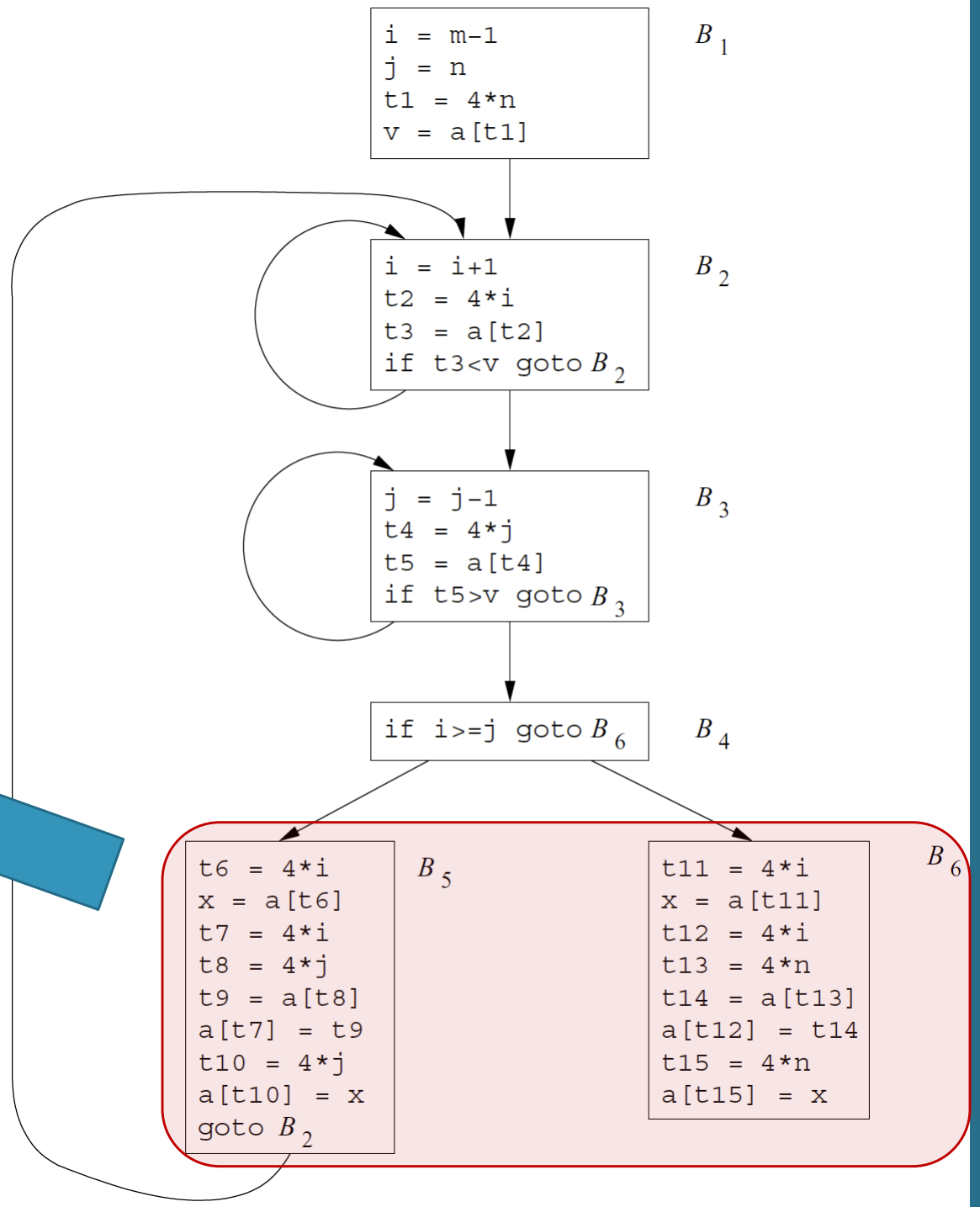


B_5



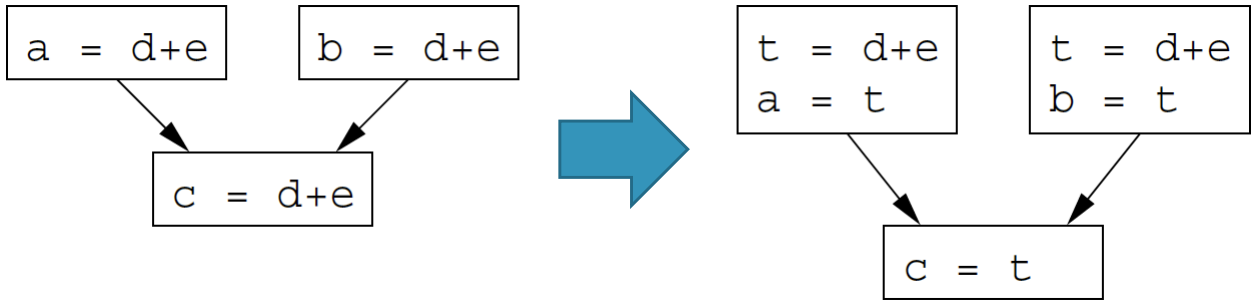
B_6





Global Optimization

- **Copy Propagation**
  - The idea behind the copy-propagation transformation is to use v for u, wherever possible after the copy statement u = v
  - **Example**
    - Since control may reach c = d+e either after the assignment to a or after the assignment to b, it would be incorrect to replace c = d+e by either c = a or by c = b



Global Optimization

- **Copy Propagation**

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```


```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Global Optimization

- **Dead-Code Elimination**
  - A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point
  - **Example:**
    - It may be possible for the compiler to deduce that each time the program reaches this statement, the value of debug is FALSE
    - We can eliminate both the test and the print operation from the object code

```
if (debug) print ...
```

  - One advantage of copy propagation is that it often turns the copy statement into dead code

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

```
a[t2] = t5
a[t4] = t3
goto B2
```

Global Optimization

- **Code Motion**
  - Loops are a very important place for optimizations
    - Especially the inner loops where programs tend to spend the bulk of their time
    - The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop
  - An important modification that decreases the amount of code in a loop is code motion
    - This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and evaluates the expression before the loop
  - **Example:** Evaluation of *limit* – 2 is a loop-invariant computation

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

n+1 calculations of limit-2

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

1 calculations of limit-2

Global Optimization

- **Induction Variables and Reduction in Strength**
  - Another important optimization is to find induction variables in loops and optimize their computation
  - A variable x is said to be an **"induction variable"** if there is a positive or negative constant c such that each time x is assigned, its value increases by c
  - **Example:** i and t2 are induction variables in the loop containing B2

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

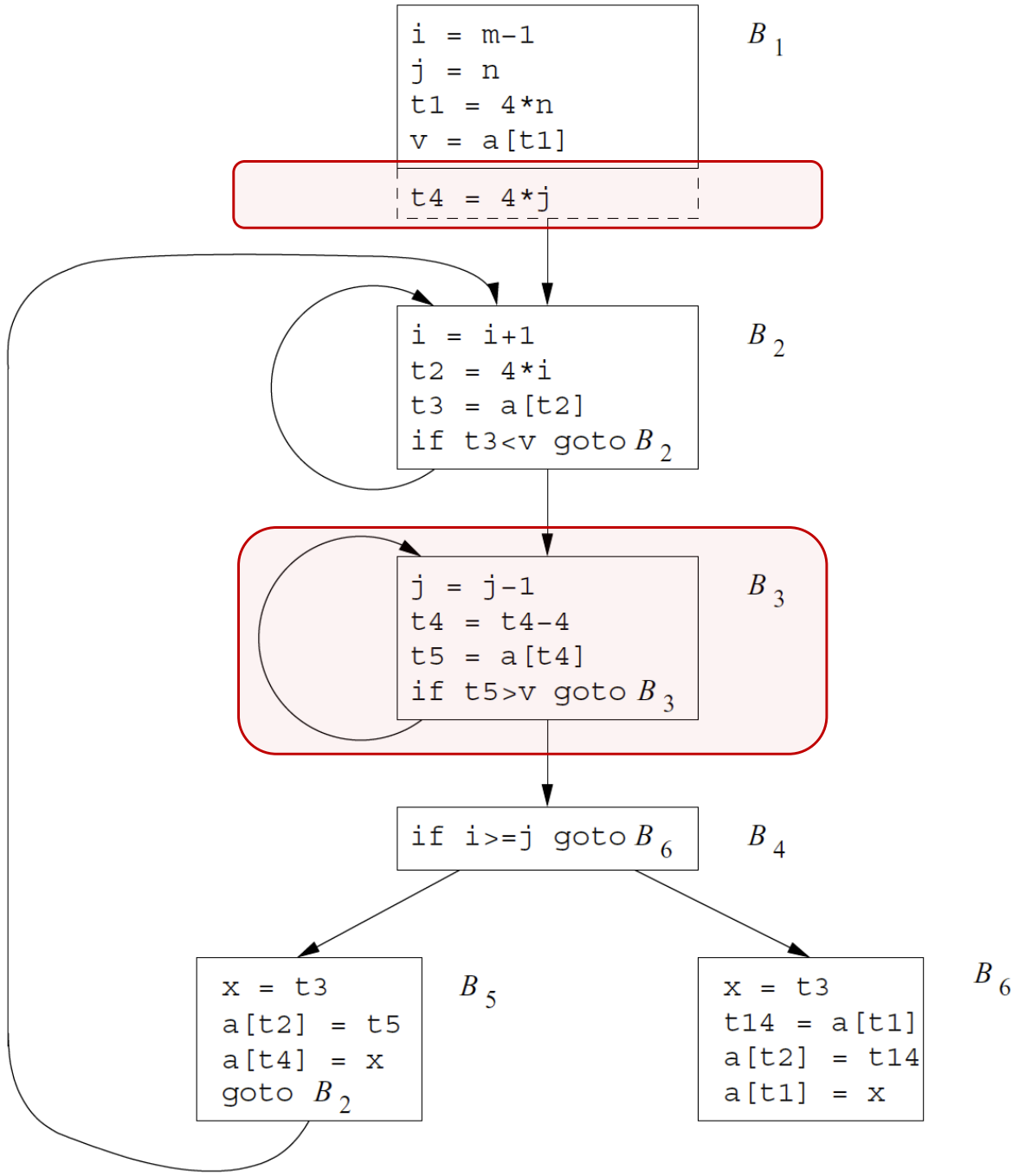
  - The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as **"strength reduction"**

Global Optimization

- **Induction Variables and Reduction in Strength**
  - When processing loops, it is useful to work **"inside-out"**; that is, we shall start with the inner loops and proceed to progressively larger, surrounding loops
  - When there are two or more induction variables in a loop, it may be possible to get rid of all but one
  - However, we can illustrate reduction in strength

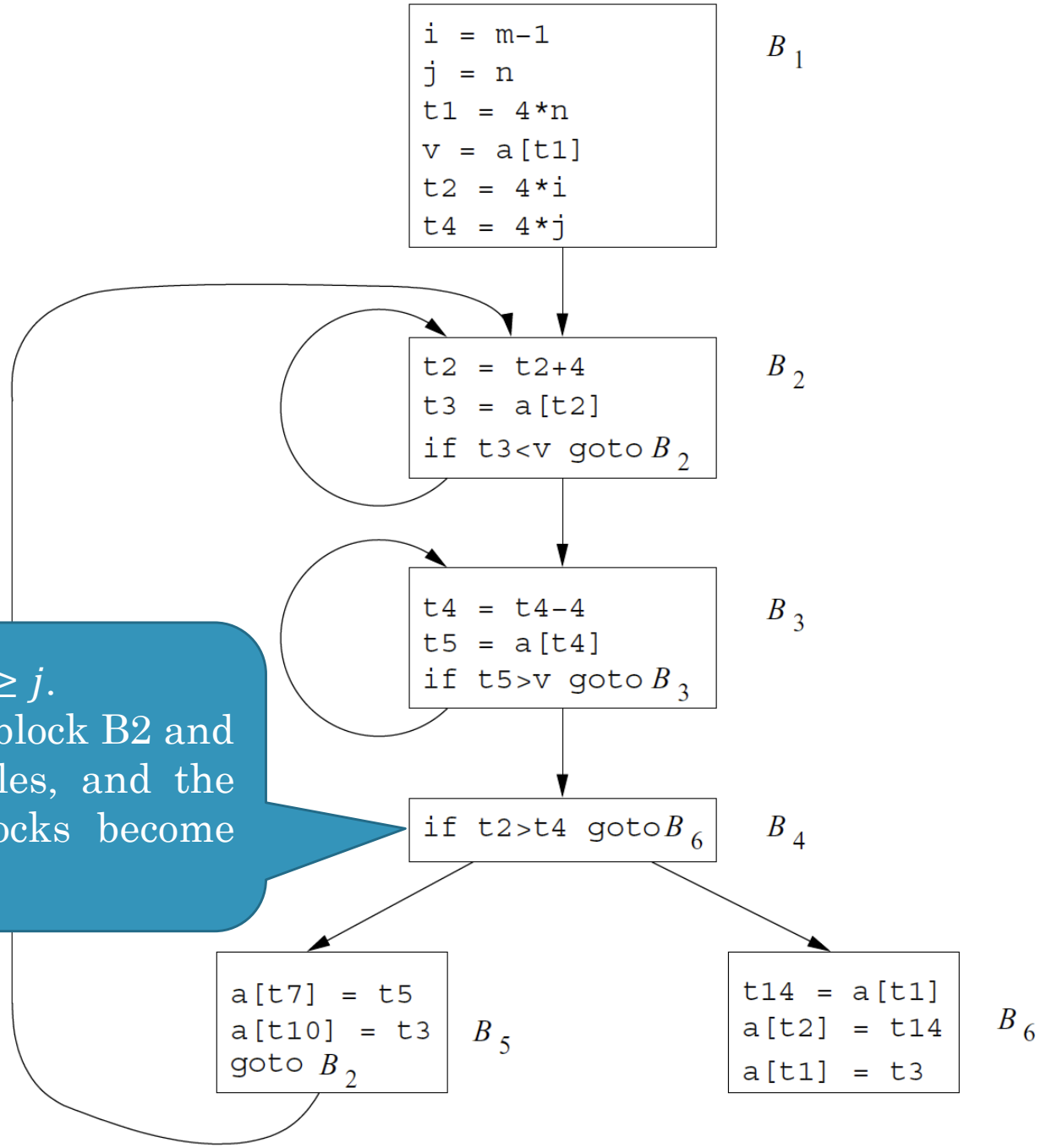
Global Optimization

- **Induction Variables and Reduction in Strength**
  - **Example:** Strength reduction applied to 4*j in block B3



Global Optimization

- **Induction Variables and Reduction in Strength**
  - **Example:** Flow graph after induction-variable elimination



- The test $t2 \geq t4$ can substitute for $i \geq j$.
- Once this replacement is made, i in block B2 and j in block B3 become dead variables, and the assignments to them in these blocks become dead code that can be eliminated

Code Optimization

- **Example:**

Initial code

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

Strength reduction

```
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
```

Code Optimization

- **Example (cont.):**

```
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
```

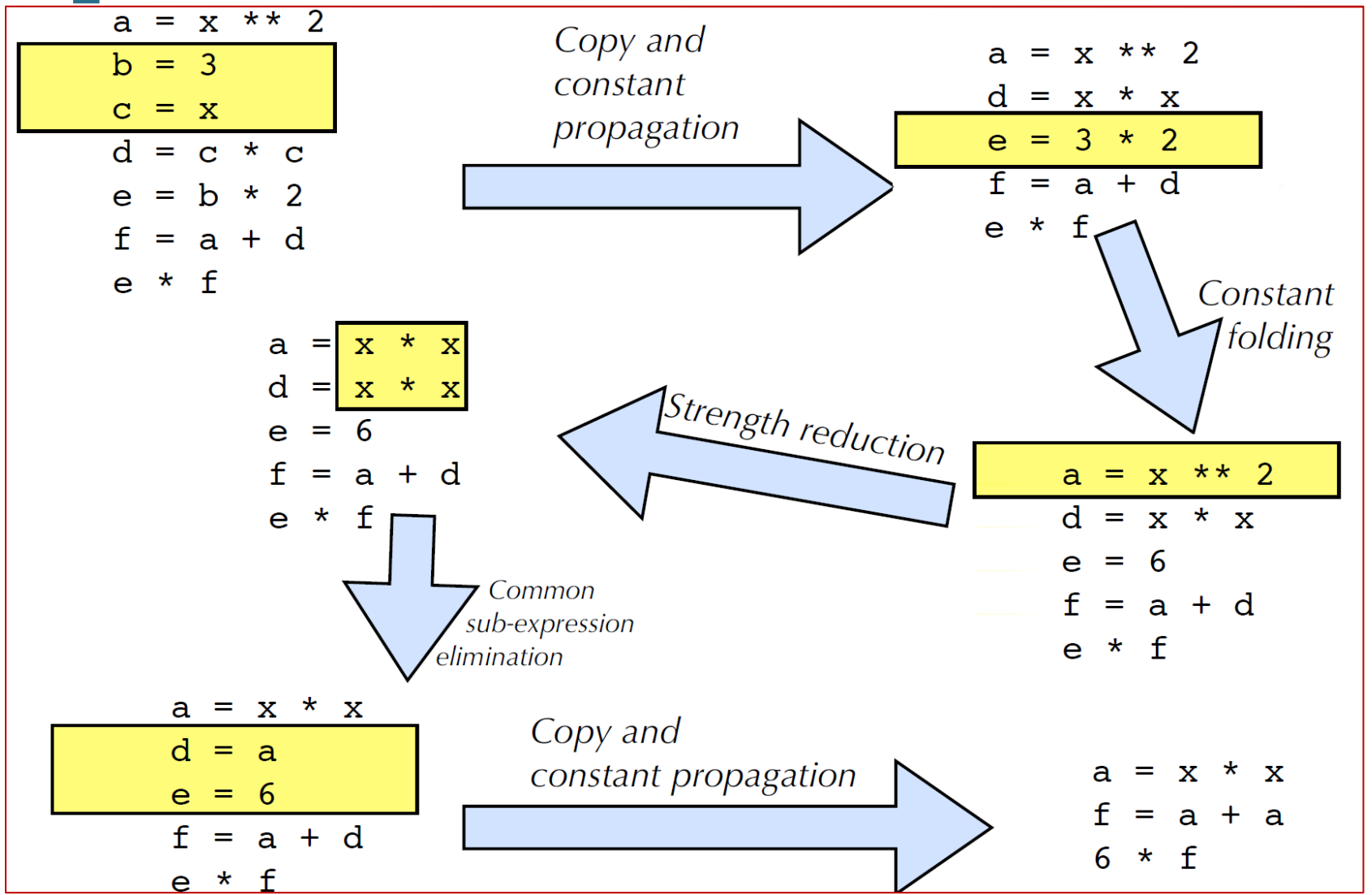
Code Optimization

- **Example (cont.):**

	Copy propagation	Dead code elimination	Final
a := x * x	a := x * x	a := x * x	a := x * x
b := 3	b := 3	b := 3	
c := x	c := x	c := x	
d := a	d := a	d := a	
e := 6	e := 6	e := 6	
f := a + d	f := a + a	f := a + a	f := a + a
g := e * f	g := 6 * f	g := 6 * f	g := 6 * f

Code Optimization

- **Example:**



Run-Time Environments

Introduction

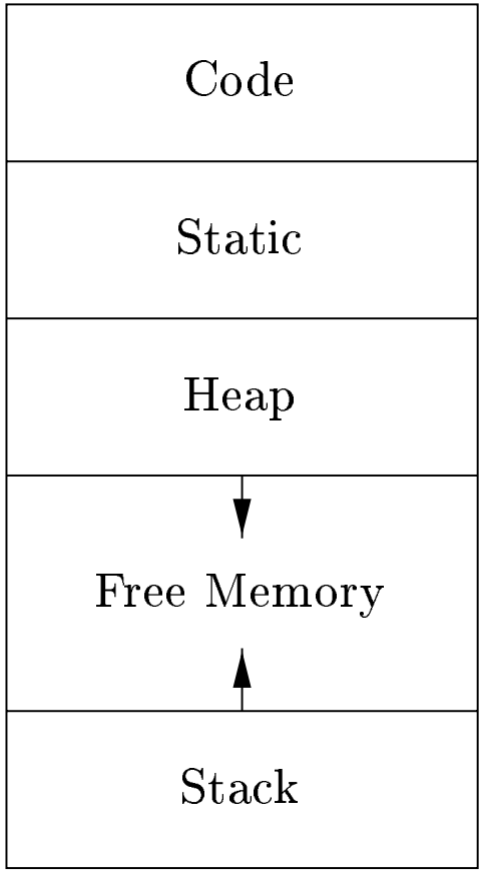
- **The compiler creates and manages a run-time environment in which it assumes its target programs are being executed**
- **This environment deals with a variety of issues such as:**
  - **The layout and allocation of storage locations for the objects named in the source program**
  - **The mechanisms used by the target program to access variables**
  - **The linkages between procedures**
  - **The mechanisms for passing parameters**
  - **The interfaces to the operating system, input/output devices, and other programs**

Storage Organization

- From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location
- The management and organization of this logical address space is shared between the compiler, operating system, and target machine
- The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory

Storage Organization

- **Typical subdivision of run-time memory into code and data areas**
  - **The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area Code, usually in the low end of memory**
  - **Similarly, the size of some program data objects, such as global constants, and data generated by the compiler, may be known at compile time, and these data objects can be placed in another statically determined area called Static**
  - **One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code.**



Storage Organization

- **Typical subdivision of run-time memory into code and data areas**
  - **The other two areas, Stack and Heap, are at the opposite ends of the remainder of the address space**
  - **These areas are dynamic; their size can change as the program executes**
    - **The *stack* is used to store data structures called activation records that get generated during procedure calls**
    - **Some programming languages allow the programmer to allocate and deallocate data under program control**
    - **The *heap* is used to manage this kind of data**
  - **In practice, the stack grows towards lower addresses, the heap towards higher**