Register transfer languages and static single assignment form

Matthew Might
University of Utah
matt.might.net
www.ucombinator.org

Administrivia

- Final project posted
- It's hard, but it's fun

Today

- Register-transfer languages
- Static single assignment form
- Tour of classical optimizations

RTL

A **register-transfer language** (RTL) is a flat-scoped intermediate form with an infinite number of registers.

Purpose

- Closer match to standard instruction sets
- Conversion to flat-scoped environments
- Apply classical intraprocedural optimization

RTL levels

- High-level
- Mid-level
- Low-level

High-level

```
s ∈ Stmt ::= v := exp ;
           |  v[exp] := exp ;
           |  return exp ;
           |  { s ... }
           |  if (exp) s else s
           |  while (exp) s
           |  v := exp(exp,...,exp)
```

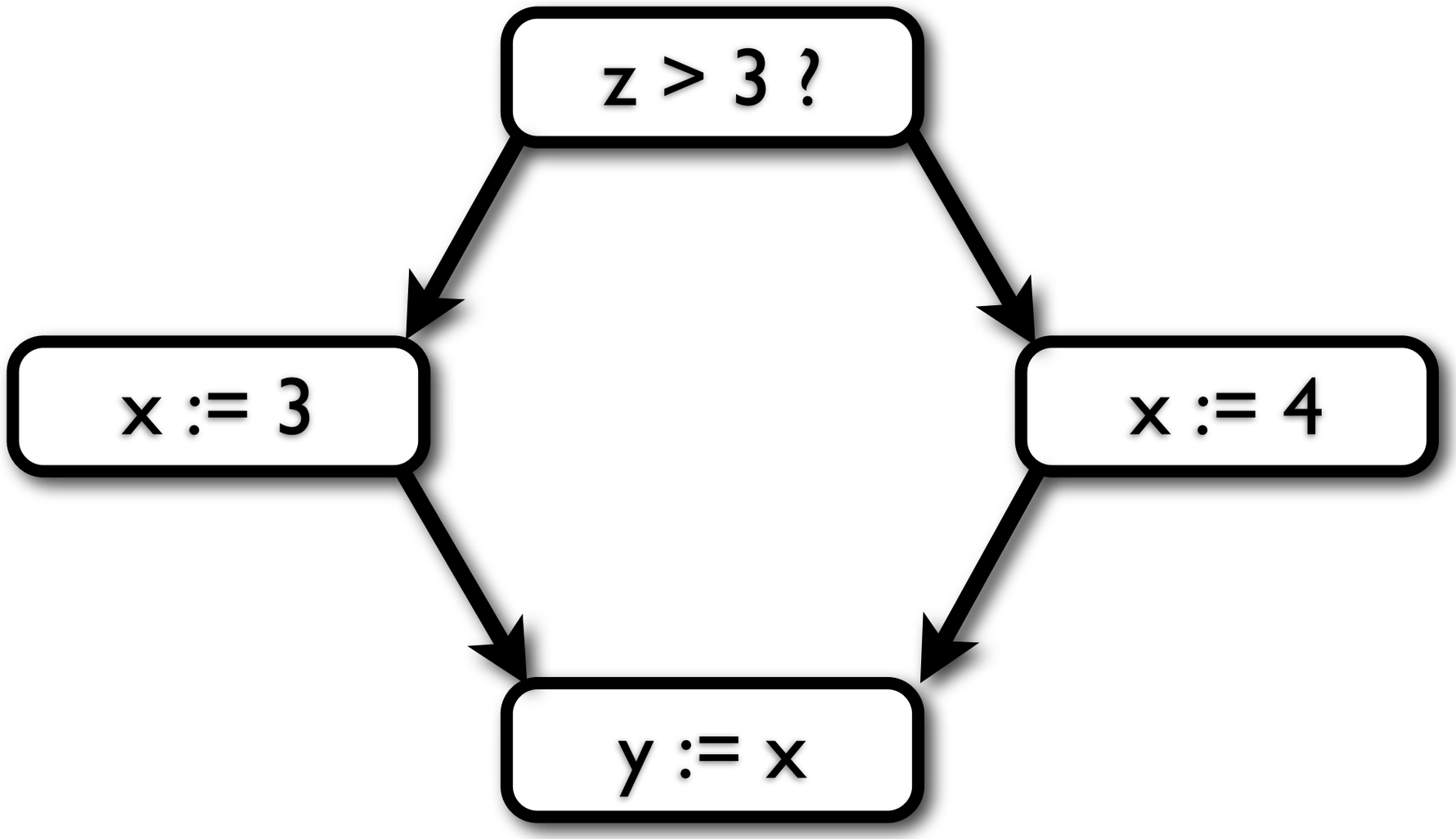
Low-level

```
s ∈ Stmt ::= v := exp ;
           |  v[exp] := exp ;
           |  label : s
           |  goto label ;
           |  if (exp) goto label ;
```

RTL => Flowchart

```
if (z > 3)
 x := 3 ;
else
 x := 4 ;
y := x ;
```

RTL => Flowchart



Static single assignment

Every variable has one point of assignment.

SSA example

```
x := 3
y := 10
```

SSA anti-example

```
x := 3
x := 10
```

SSA advantage

Turns assignments into “mathematical truths.”

Analysis of SSA

Some analyses faster; some flow-sensitive.

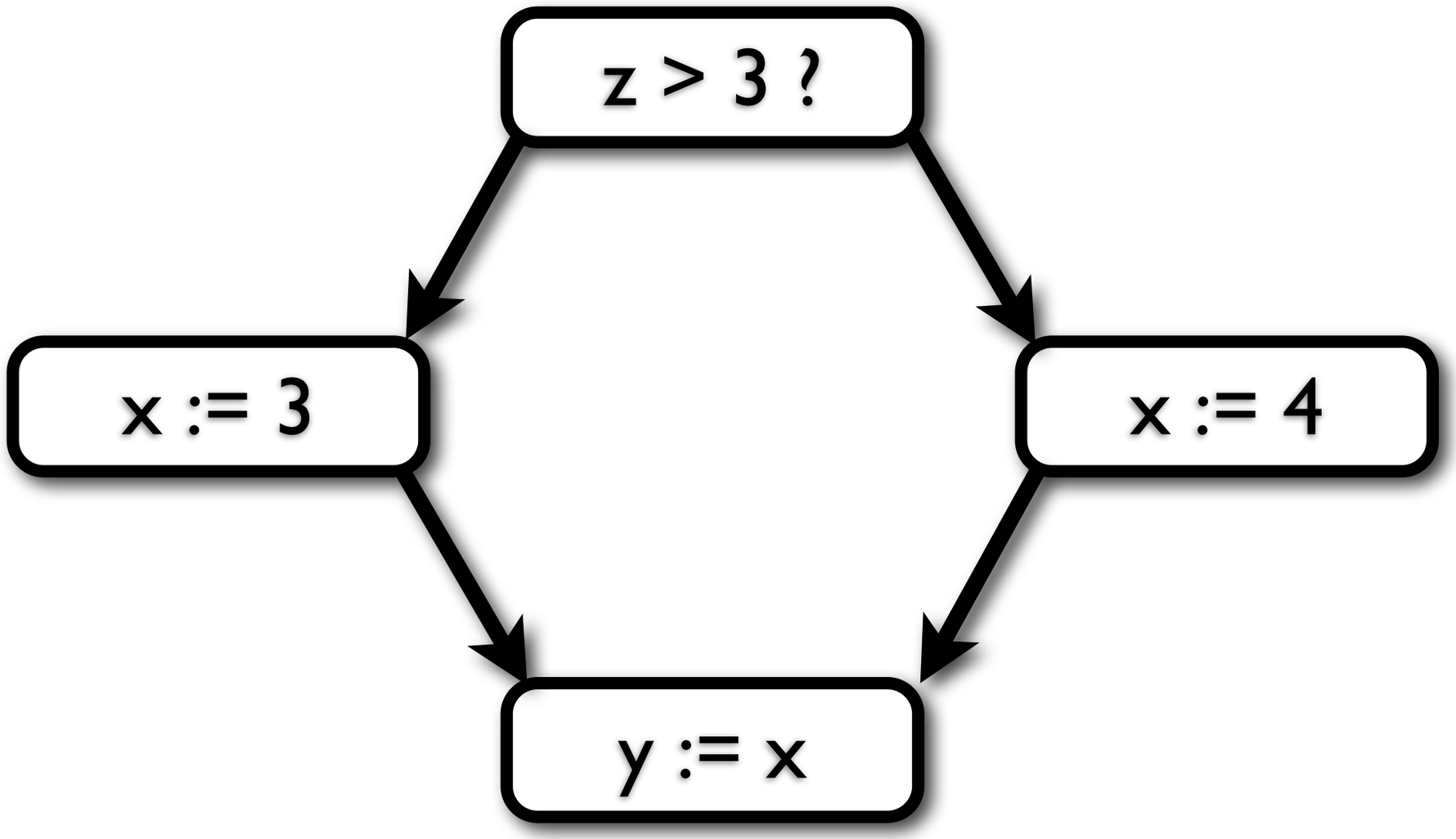
Converting to SSA

Rename variables; insert “phony” functions.

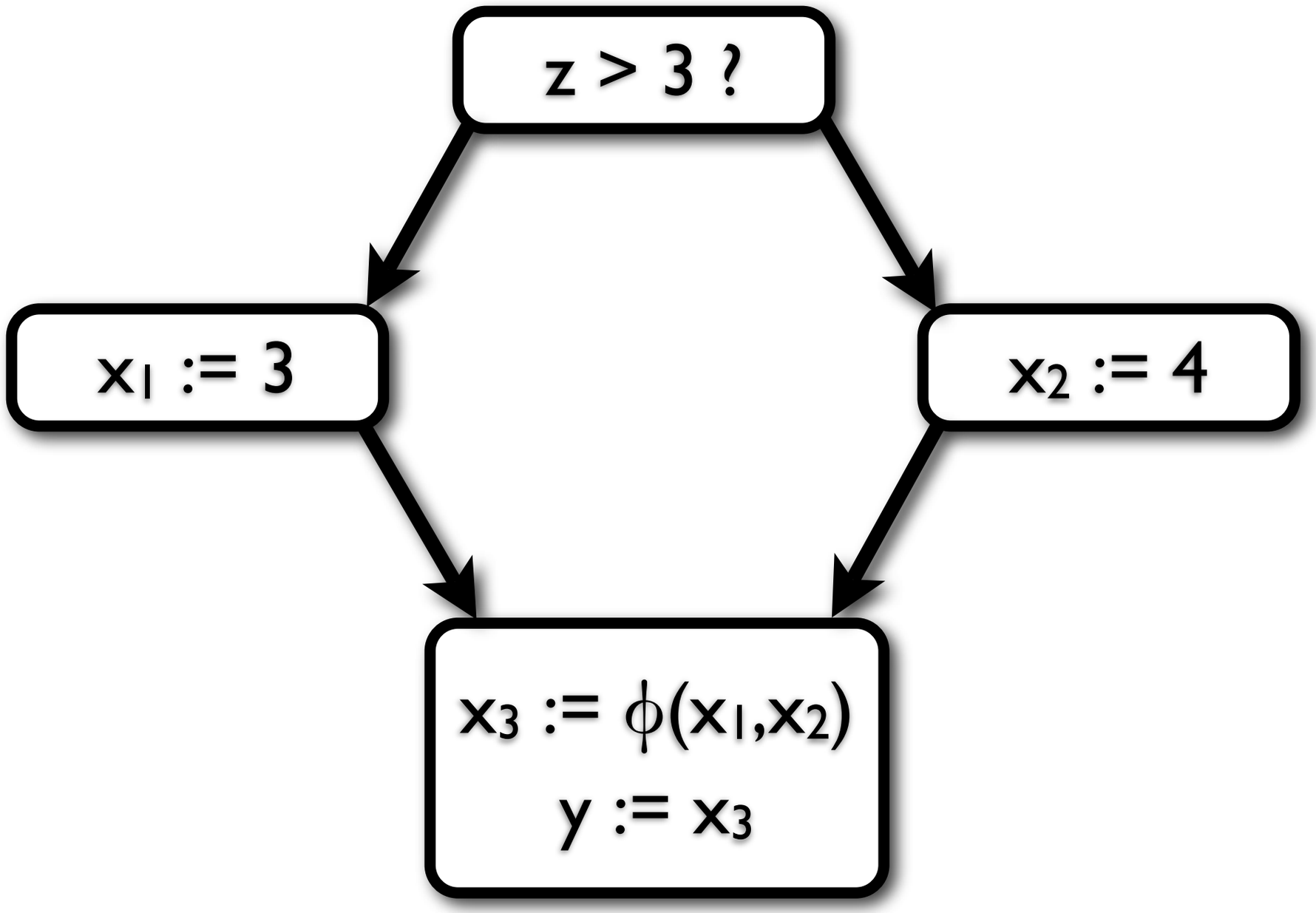
Meaning of ϕ

The expression $\phi(e_1,\ldots,e_n)$ evaluates nondeterministically to one of its arguments.

Example: RTL



Example: SSA



Meaning of SSA

Original execution is within nondeterministic execution.

Pointers in SSA

- In SSA, variables cannot have addresses
- Option 1: Stack-allocate locals (LLVM)
- Option 2: Scalarize locals (if possible)

Example

```
x    := 3 ;
p    := &x ;
*p   := 20 ;
z    := f(x) ;
```

Example: Stack-allocation

```
px  := alloca() ;
...
*px := 3 ;
p   := px ;
*p  := 20 ;
z   := f(*px) ;
```

Example: Scalarization

```
x1 := 3 ;
p  := ADDR_X ;
if (p == ADDR_X)
 x2 := 20 ; // *p := 20
else
 compiler_error() ;
x3 := ϕ(x1,x2)
z  := f(x3) ;
```

Optimizations

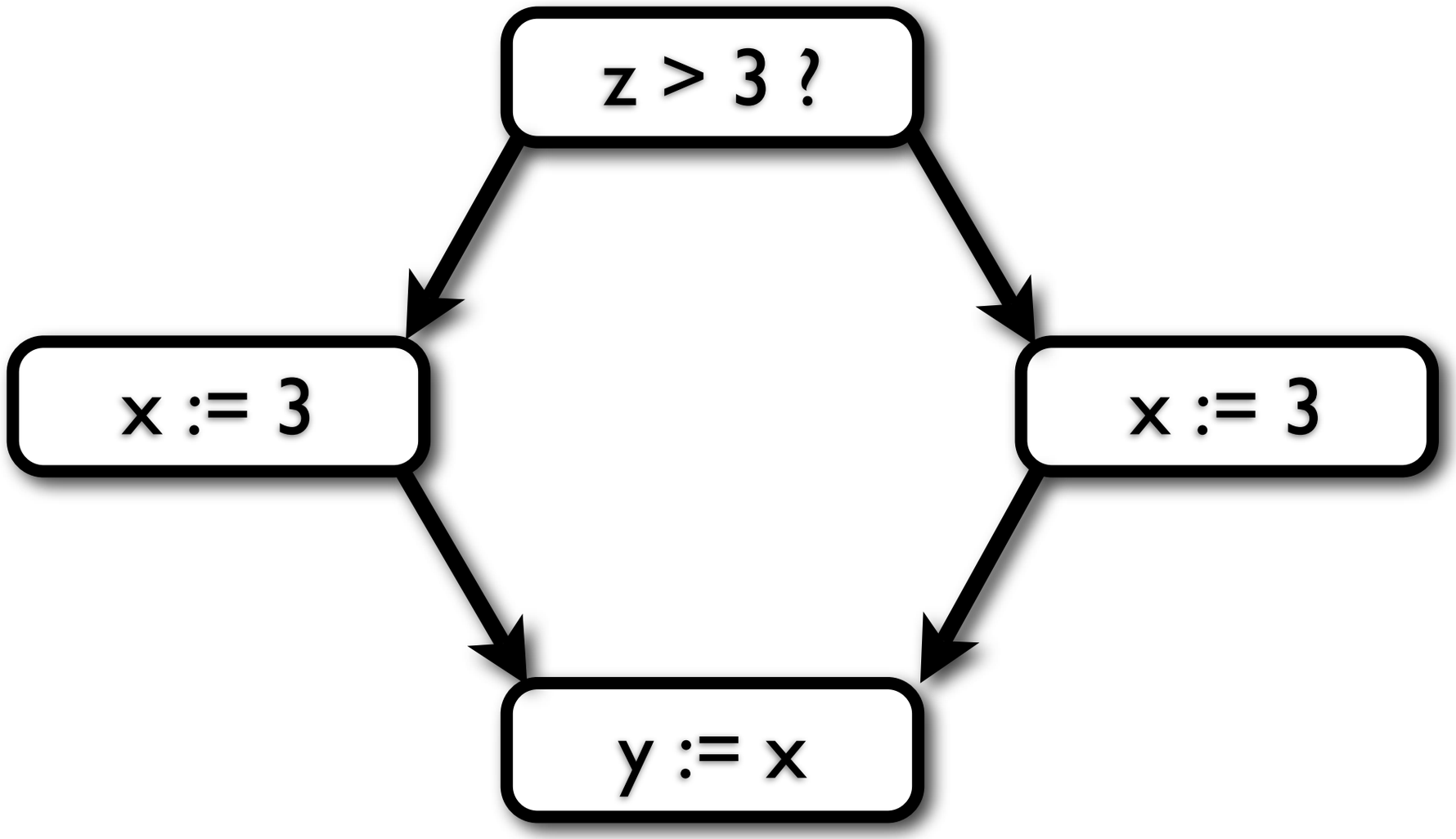
Classical optimization

- Constant folding
- Constant propagation
- Common subexpressions
- Very-busy expressions
- Register allocation

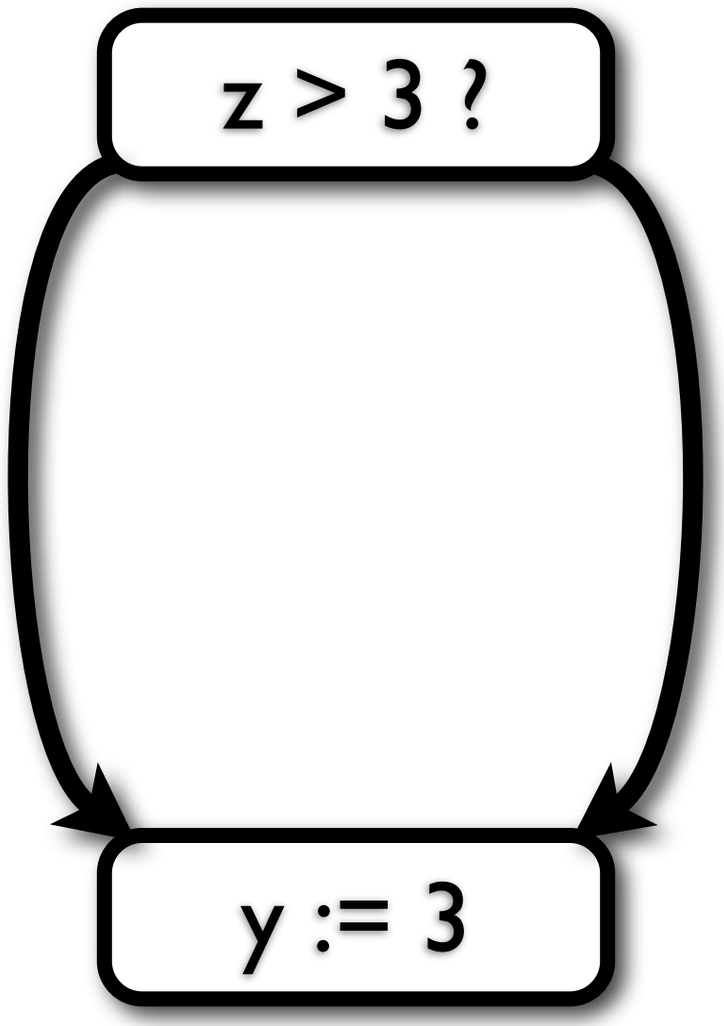
Constant folding

- Given v := e_1 op e_2
- If e_1, e_2 are constant,
- Evaluate e_1 op e_2

Constant propagation



Constant propagation



Common subexpressions

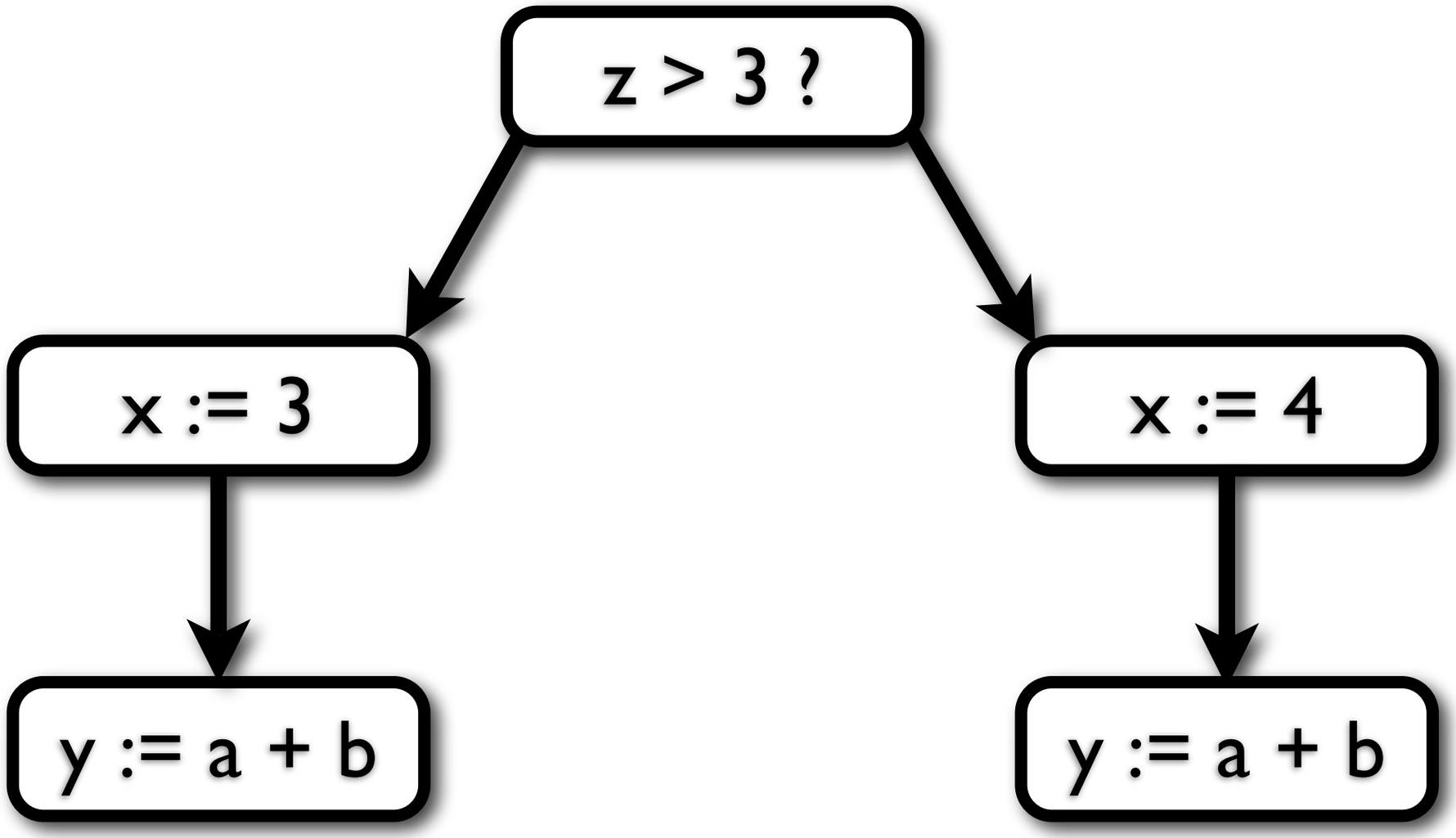
```
y := a + b + d
x := a + b + c
```

Common subexpressions

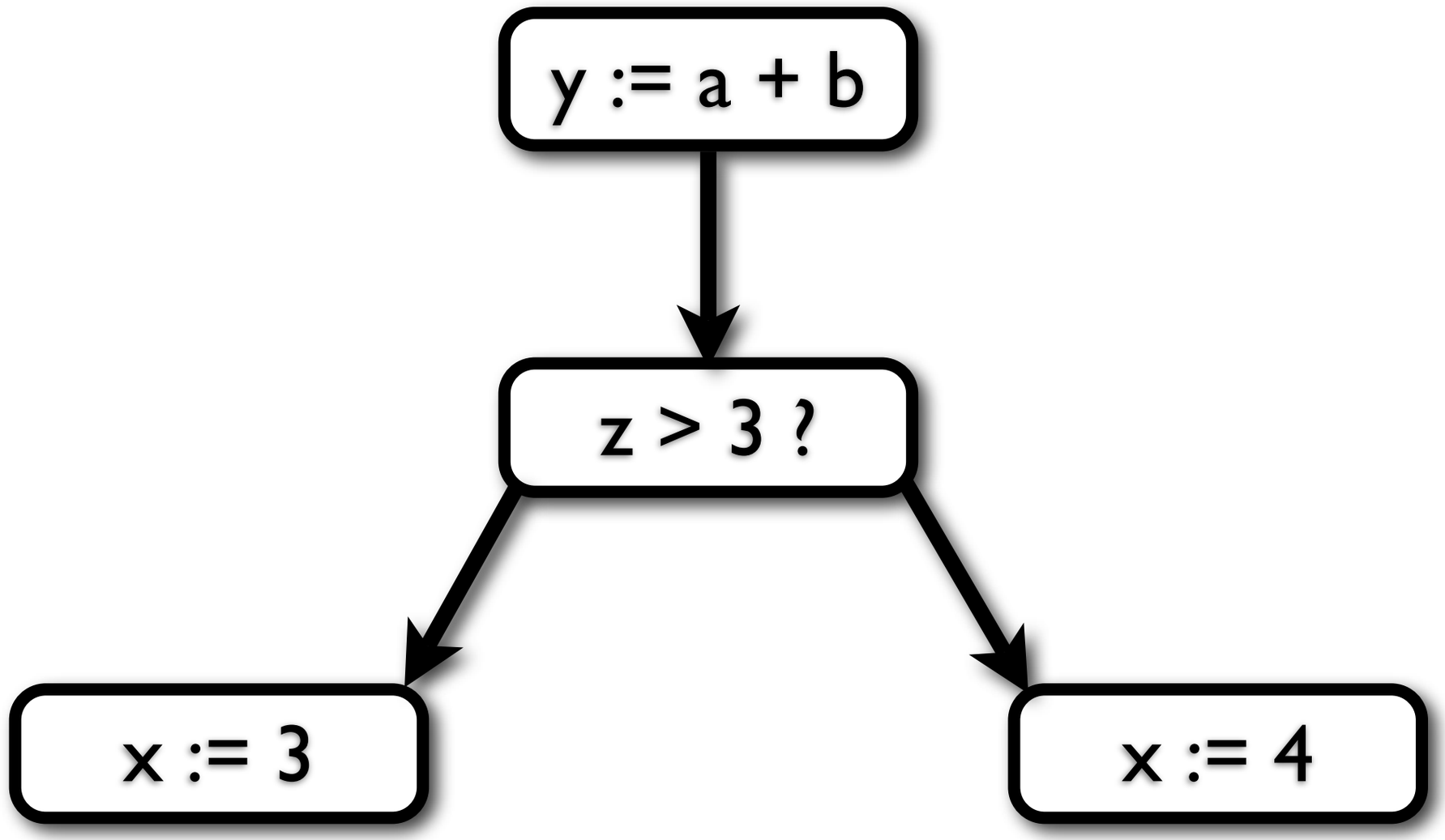
```
y := a + b + d
x := a + b + c
```

```
t := a + b
y := t + d
x := t + c
```

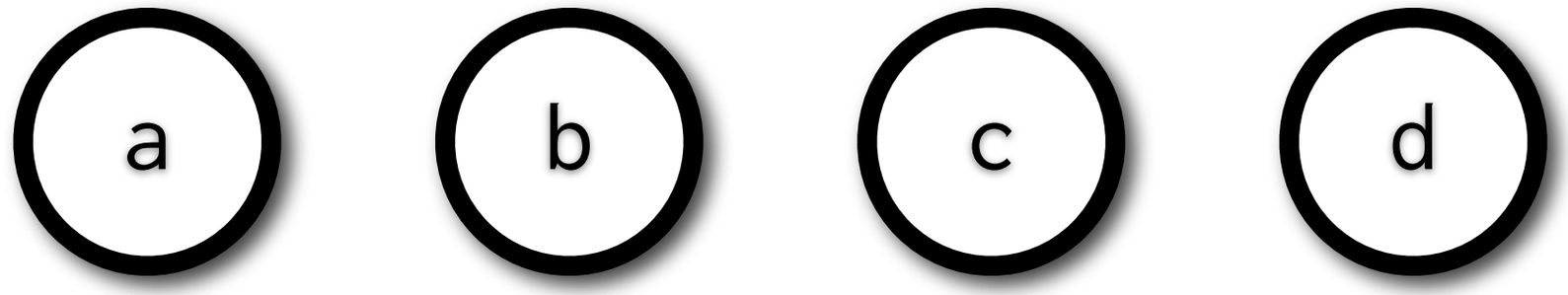
Very busy expressions



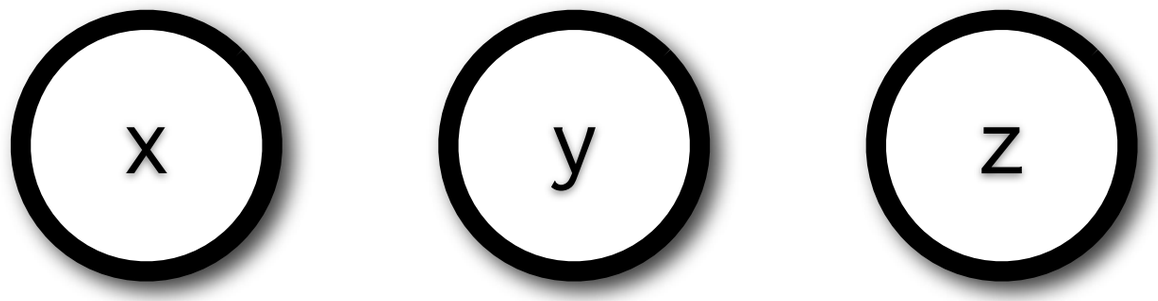
Very busy expressions



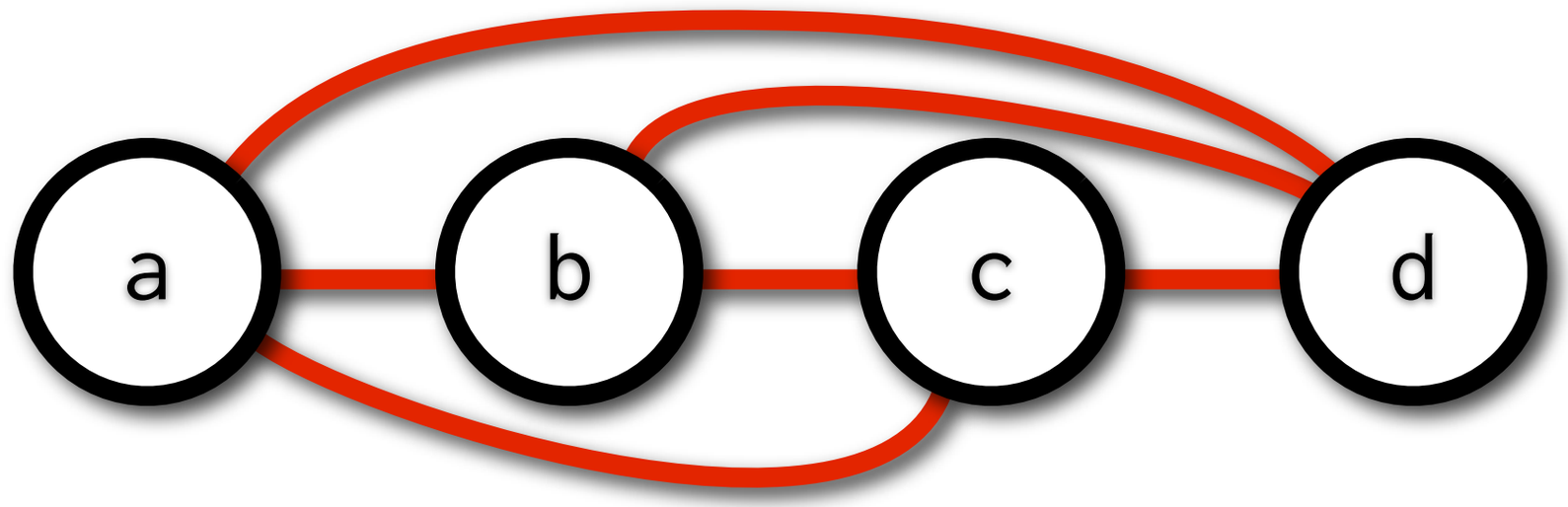
Register allocation



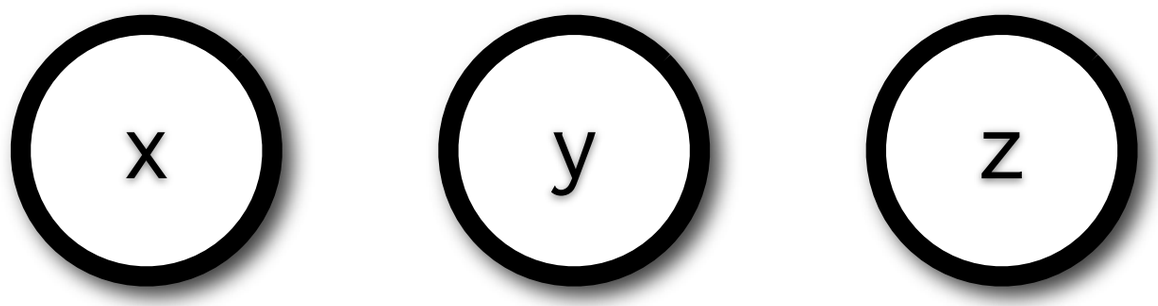

```
x := a + b
y := c + d
z := x + y
```



Register allocation

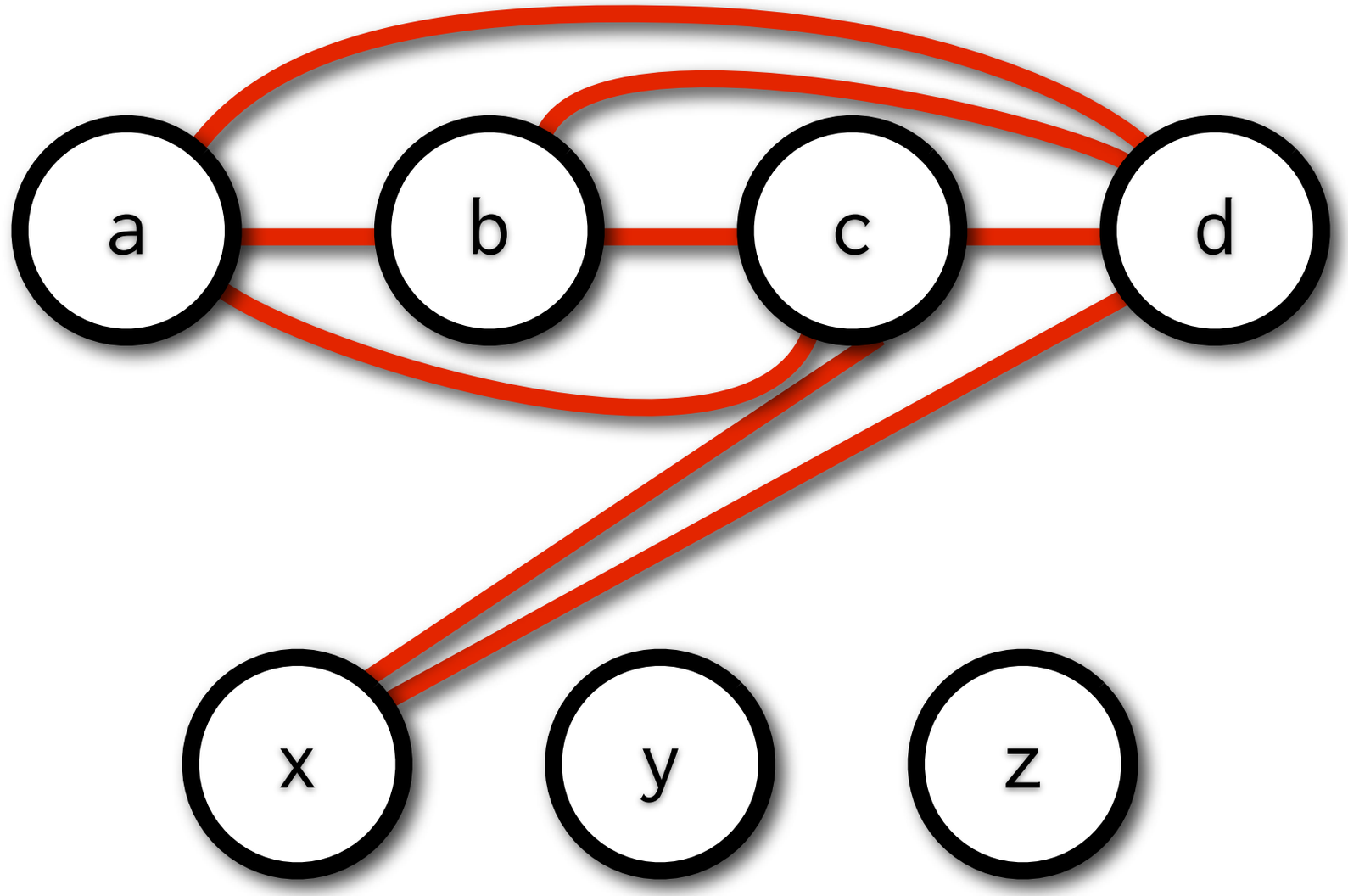



```
x := a + b
y := c + d
z := x + y
```



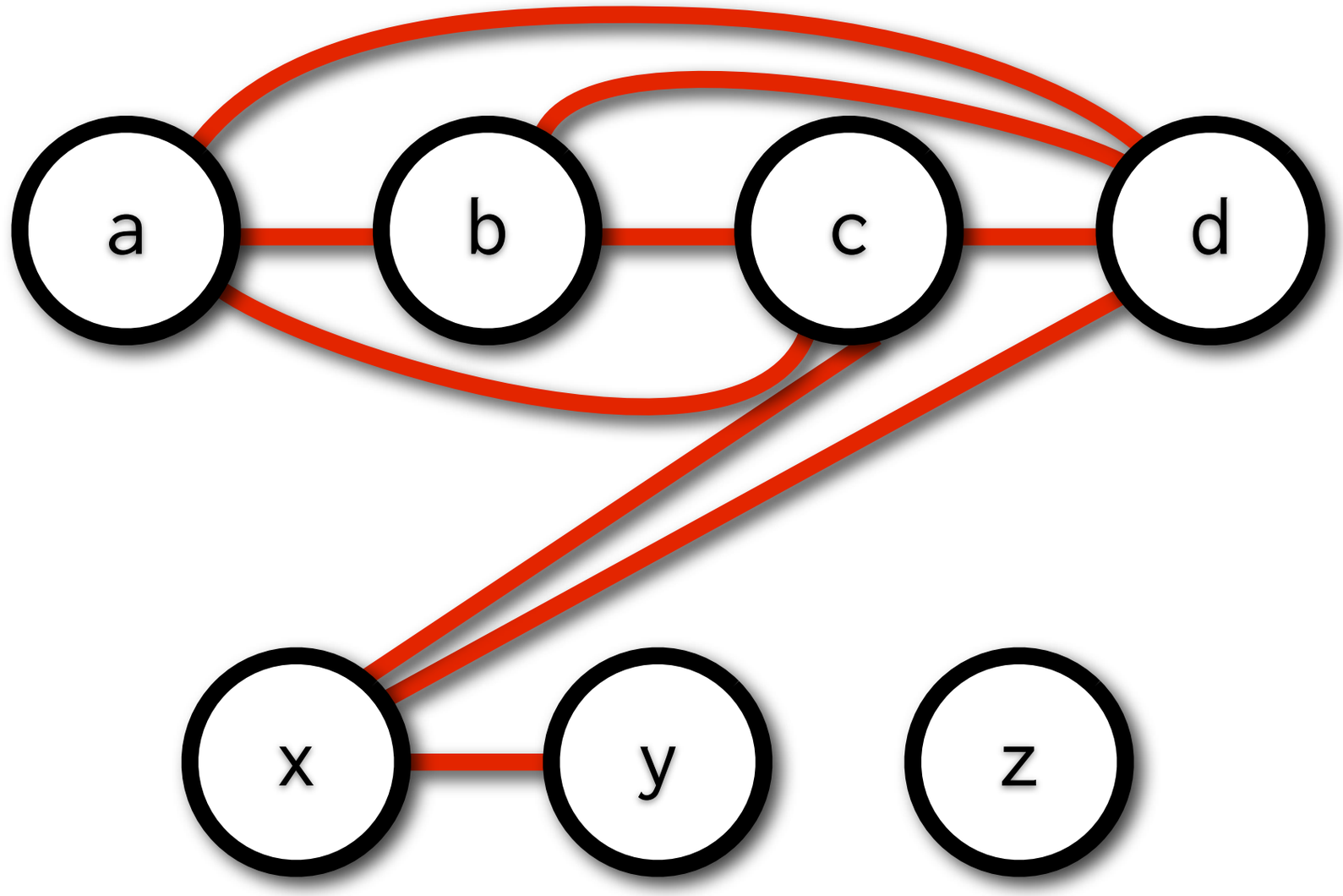
Register allocation

```
x := a + b
y := c + d
z := x + y
```



Register allocation

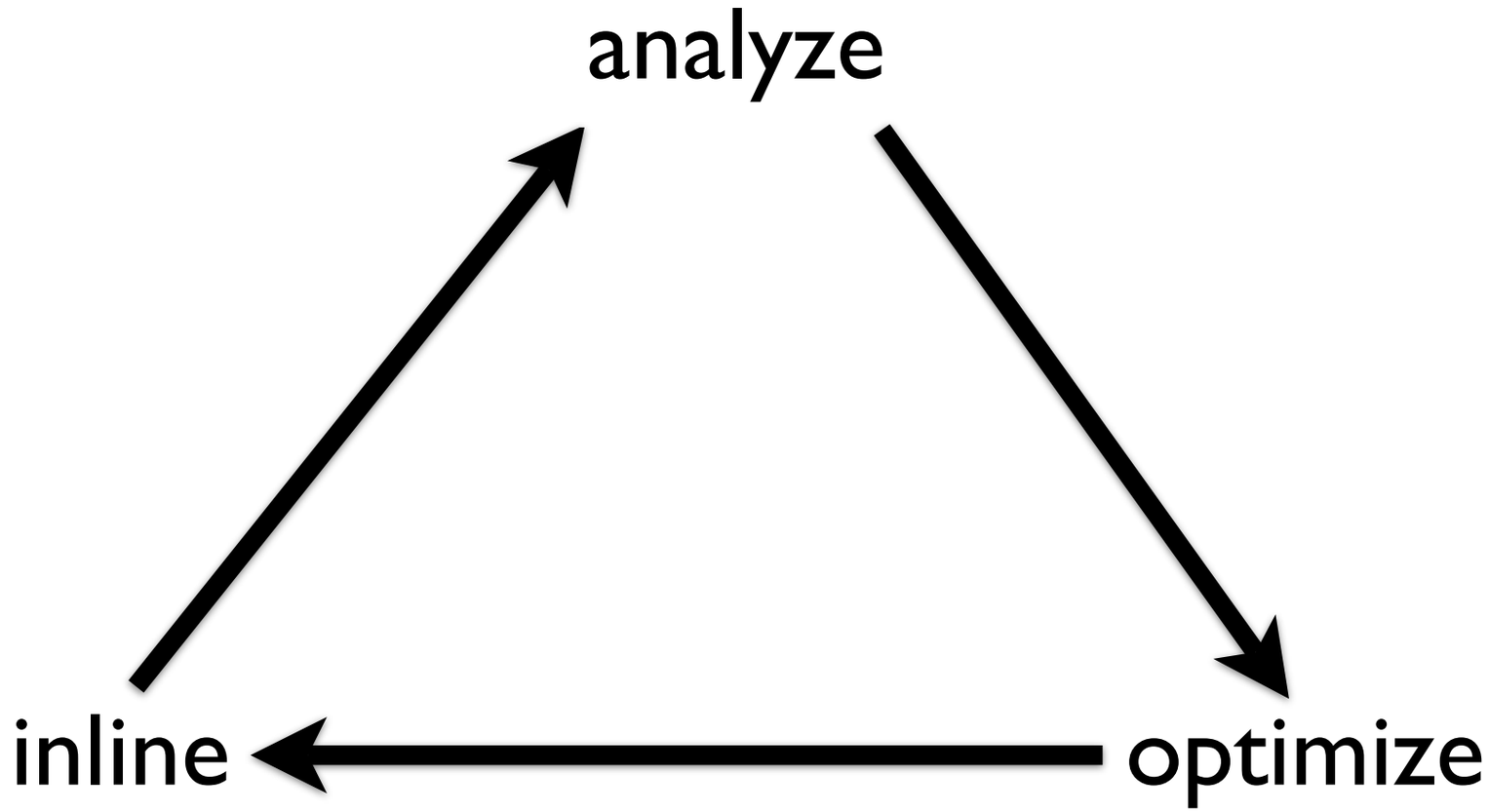
```
x := a + b
y := c + d
z := x + y
```



Why optimize these?

- Programmer freedom
- Machine-generated code
- Compact function inlines

Optimization cyle



80/20 rule

- Constant propagation
- Good register allocation
- Good procedure inlines