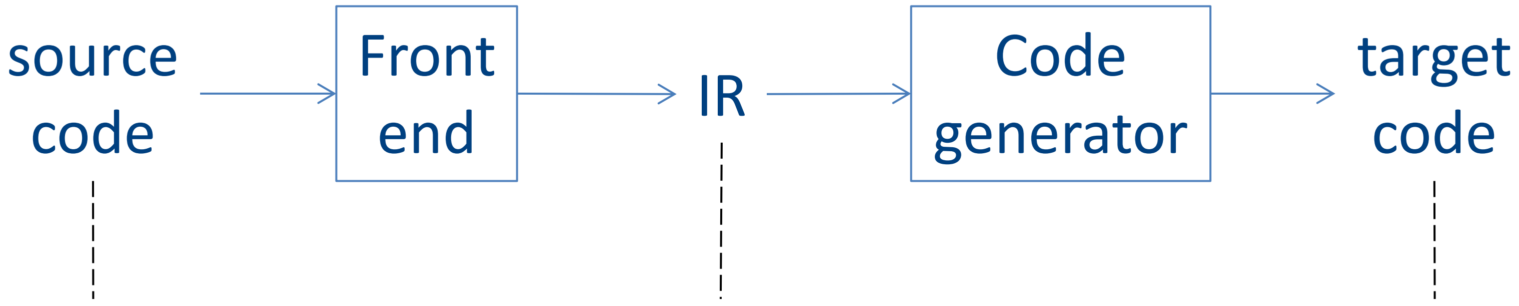
Compilation

Lecture 10a

Abstract Interpretation

Noam Rinetzky

Optimization points



User
profile program
change algorithm

Compiler
intraprocedural IR
Interprocedural IR
IR optimizations

Compiler
register allocation
instruction selection
peephole transformations

now

IR Optimization

- Making code “better”

Overview of IR optimization

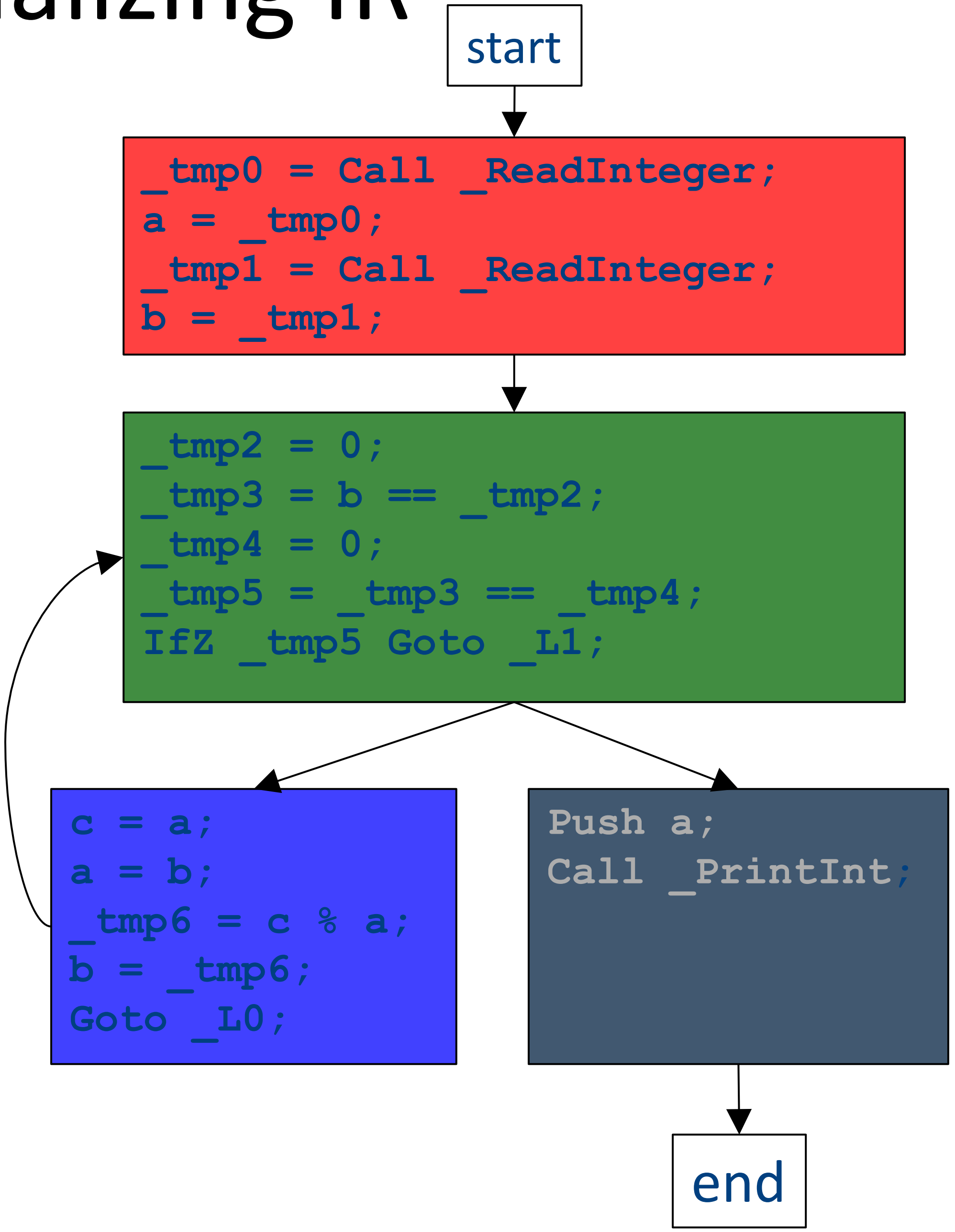
- Formalisms and Terminology
  - Control-flow graphs
  - Basic blocks
- Local optimizations
  - Speeding up small pieces of a procedure
- Global optimizations
  - Speeding up procedure as a whole
- The dataflow framework
  - Defining and implementing a wide class of optimizations

Program Analysis

- In order to optimize a program, the compiler has to be able to reason about the properties of that program
- An analysis is called **sound** if it never asserts an incorrect fact about a program
- All the analyses we will discuss in this class are sound
  - *(Why?)*

Visualizing IR

```
main:
    _tmp0 = Call _ReadInteger;
    a = _tmp0;
    _tmp1 = Call _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    Push a;
    Call _PrintInt;
```



Common Subexpression Elimination

- If we have two variable assignments
  ```
  v1 = a op b
  …
  v2 = a op b
  ```
- and the values of v1, a, and b have not changed between the assignments, rewrite the code as
  ```
  v1 = a op b
  …
  v2 = v1
  ```
- Eliminates useless recalculation
- Paves the way for later optimizations

Common Subexpression Elimination

- If we have two variable assignments
  v1 = a op b   [or: v1 = a]
  …
  v2 = a op b   [or: v2 = a]
- and the values of v1, a, and b have not changed between the assignments, rewrite the code as
  v1 = a op b   [or: v1 = a]
  …
  v2 = v1
- Eliminates useless recalculation
- Paves the way for later optimizations

Copy Propagation

- If we have a variable assignment
  v1 = v2
  then as long as v1 and v2 are not reassigned, we can rewrite expressions of the form
  a = … v1 …
  as
  a = … v2 …
  provided that such a rewrite is legal

Dead Code Elimination

- An assignment to a variable v is called dead if the value of that assignment is never read anywhere
- Dead code elimination removes dead assignments from IR
- Determining whether an assignment is dead depends on what variable is being assigned to and when it's being assigned

Abstract Interpretation

- Theoretical foundations of program analysis
- Cousot and Cousot 1977
- Abstract meaning of programs
  - Executed at compile time

Another view of local optimization

- In local optimization, we want to reason about some property of the runtime behavior of the program
- Could we run the program and just watch what happens?
- **Idea:** Redefine the semantics of our programming language to give us information about our analysis

Assigning new semantics

- Example: Available Expressions
- Redefine the statement **a = b + c to** mean "**a now holds the value of b + c,** and any variable holding the value **a is** now invalid"
- Run the program assuming these new semantics
- Treat the optimizer as an interpreter for these new semantics

Join semilattices

- A join semilattice is a ordering defined on a set of elements
- Any two elements have some join that is the smallest element larger than both elements
- There is a unique bottom element, which is smaller than all other elements
- Intuitively:
  - The join of two elements represents combining information from two elements by an overapproximation
- The bottom element represents “no information yet” or “the least conservative possible answer”

Join semilattices and ordering

$$\begin{tikzcd}
 & \{a, b, c\} & \\
\{a, b\} \arrow[ur] & \{a, c\} \arrow[u] & \{b, c\} \arrow[ul] \\
\{a\} \arrow[u] \arrow[ur] & \{b\} \arrow[ul] \arrow[ur] & \{c\} \arrow[ul] \arrow[u] \\
 & \{\} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Least precise

$$\updownarrow$$

Most precise

Formal definitions

- A join semilattice is a pair (V, $\sqcup$), where
- V is a domain of elements
- $\sqcup$ is a join operator that is
  - commutative: $x \sqcup y = y \sqcup x$
  - associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
  - idempotent: $x \sqcup x = x$
- If $x \sqcup y = z$, we say that z is the join or (**l**east **u**pper **b**ound) of x and y
- Every join semilattice has a bottom element denoted $\perp$ such that $\perp \sqcup x = x$ for all x

Join semilattices and orderings

- Every join semilattice $(V, \sqcup)$ induces an ordering relationship $\sqsubseteq$ over its elements
- Define $x \sqsubseteq y$ iff $x \sqcup y = y$
- Need to prove
  - Reflexivity: $x \sqsubseteq x$
  - Antisymmetry: If $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$
  - Transitivity: If $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$

A general framework

- A global analysis is a tuple $(\mathtt{D}, \mathtt{V}, \sqsubseteq, \mathtt{F}, \mathtt{I})$, where
  - $\mathtt{D}$ is a direction (forward or backward)
    - The order to visit statements within a basic block, not the order in which to visit the basic blocks
  - $\mathtt{V}$ is a set of values
  - $\sqcup$ is a join operator over those values
  - $\mathtt{F}$ is a set of transfer functions $f : \mathbf{V} \rightarrow \mathbf{V}$
  - $\mathtt{I}$ is an initial value
- The only difference from local analysis is the introduction of the join operator

Running global analyses

- Assume that $(\mathbb{D}, \mathbb{V}, \sqcup, \mathbb{F}, \mathbb{I})$ is a forward analysis
- Set OUT[**s**] = $\bot$ for all statements **s**
- Set OUT[**entry**] = $\mathbb{I}$
- Repeat until no values change:
  - For each statement **s** with predecessors $\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n}$:
    - Set IN[**s**] = OUT[$\mathbf{p_1}$] $\sqcup$ OUT[$\mathbf{p_2}$] $\sqcup$ … $\sqcup$ OUT[$\mathbf{p_n}$]
    - Set OUT[**s**] = f_s (IN[**s**])
- The order of this iteration does not matter
  - This is sometimes called chaotic iteration

Global constant propagation

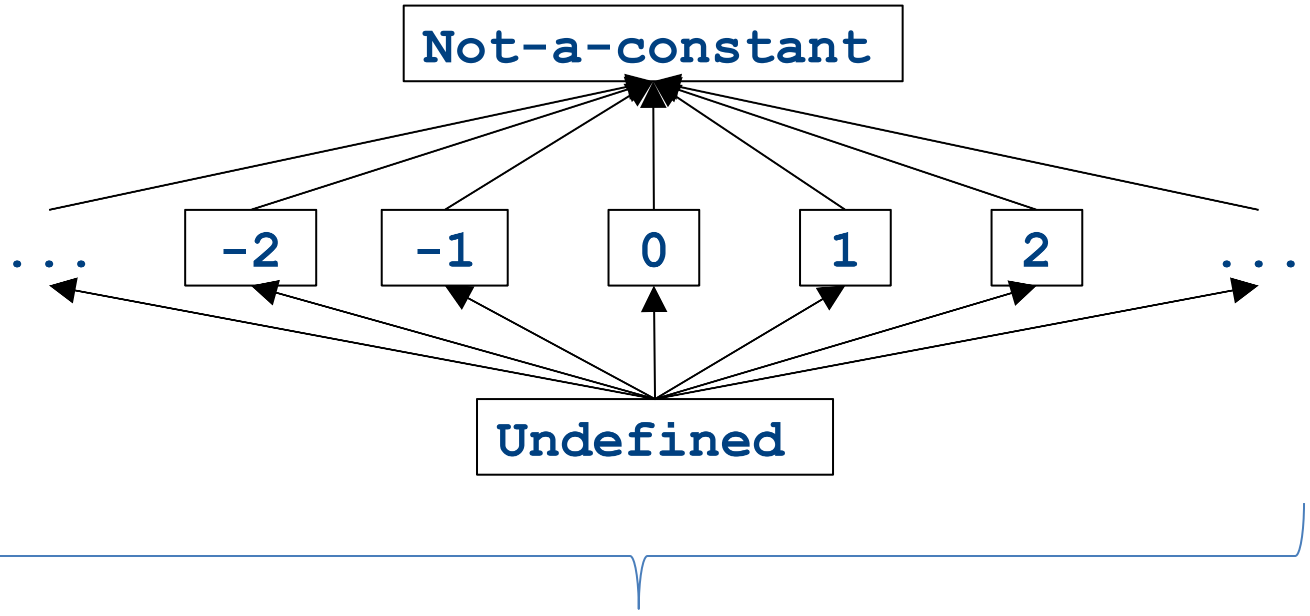
- Constant propagation is an optimization that replaces each variable that is known to be a constant value with that constant
- An elegant example of the dataflow framework

Defining a join operator

- The join of any two different constants is **Not-a-Constant**
  - (If the variable might have two different values on entry to a statement, it cannot be a constant)
- The join of **Not a Constant** and any other value is **Not-a-Constant**
  - (If on some path the value is known not to be a constant, then on entry to a statement its value can't possibly be a constant)
- The join of **Undefined** and any other value is that other value
  - (If **x** has no value on some path and does have a value on some other path, we can just pretend it always had the assigned value)

A semilattice for constant propagation

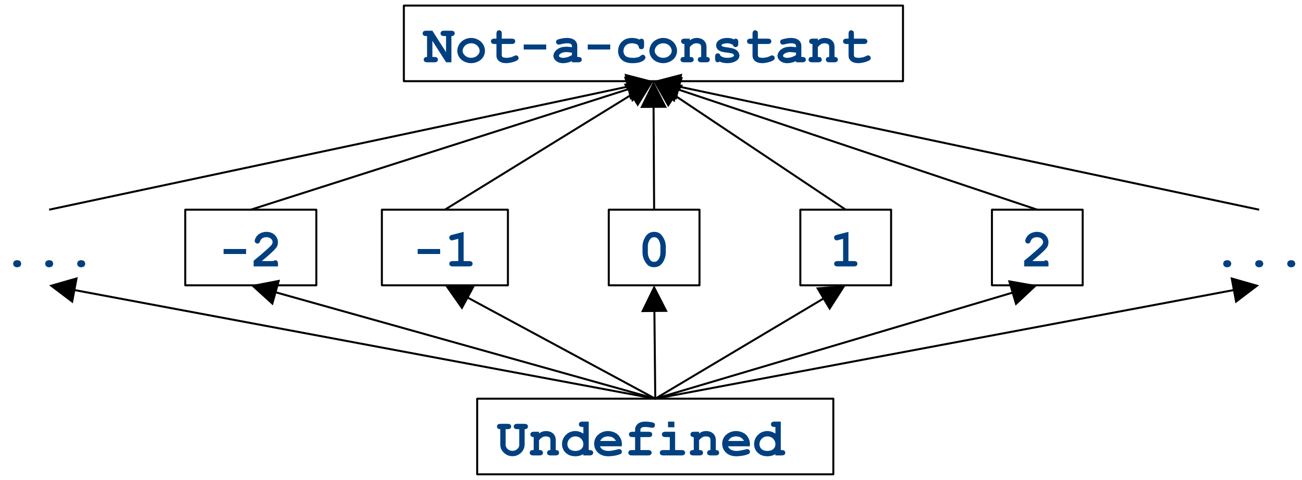
- One possible semilattice for this analysis is shown here (for each variable):



The lattice is infinitely wide

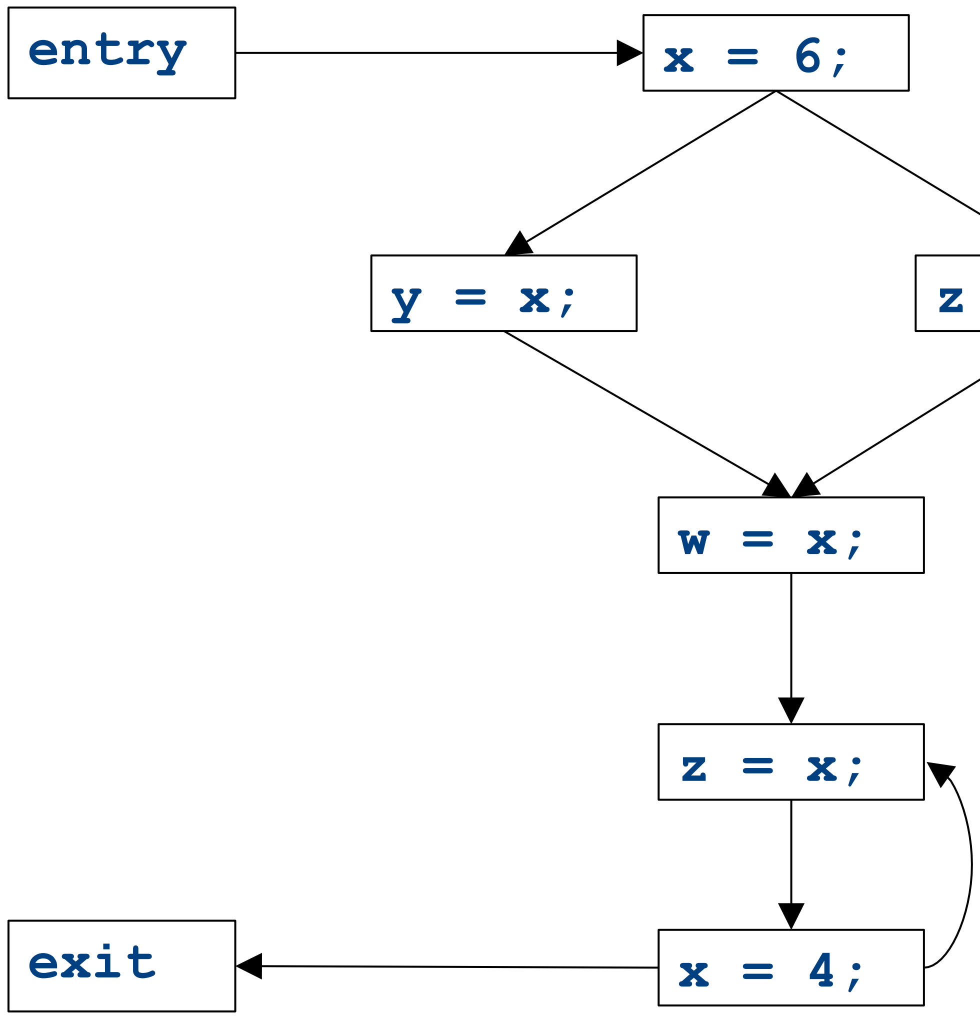
A semilattice for constant propagation

- One possible semilattice for this analysis is shown here (for each variable):

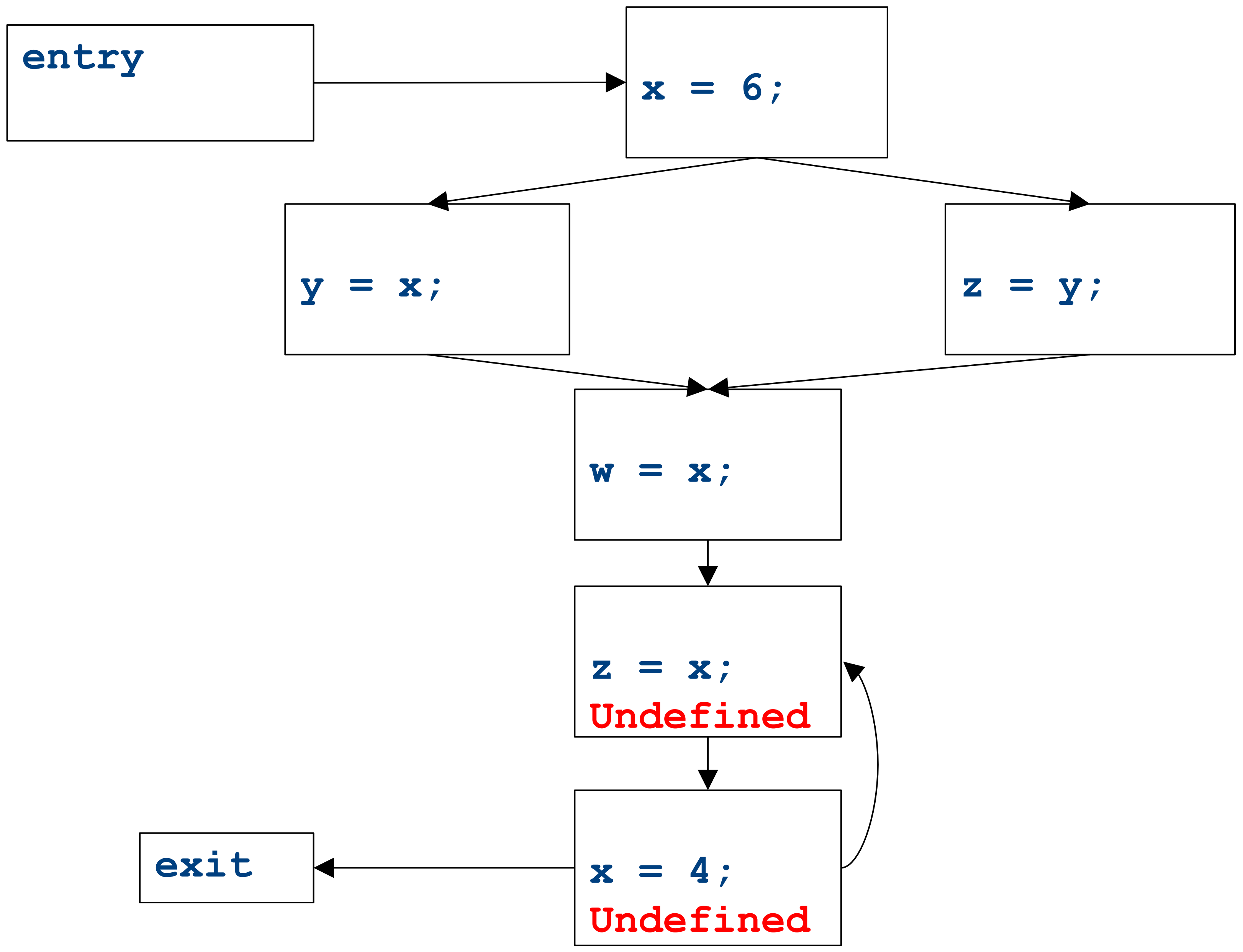


- Note:
  - The join of any two different constants is **Not-a-Constant**
  - The join of **Not a Constant** and any other value is **Not-a-Constant**
  - The join of **Undefined** and any other value is that other value

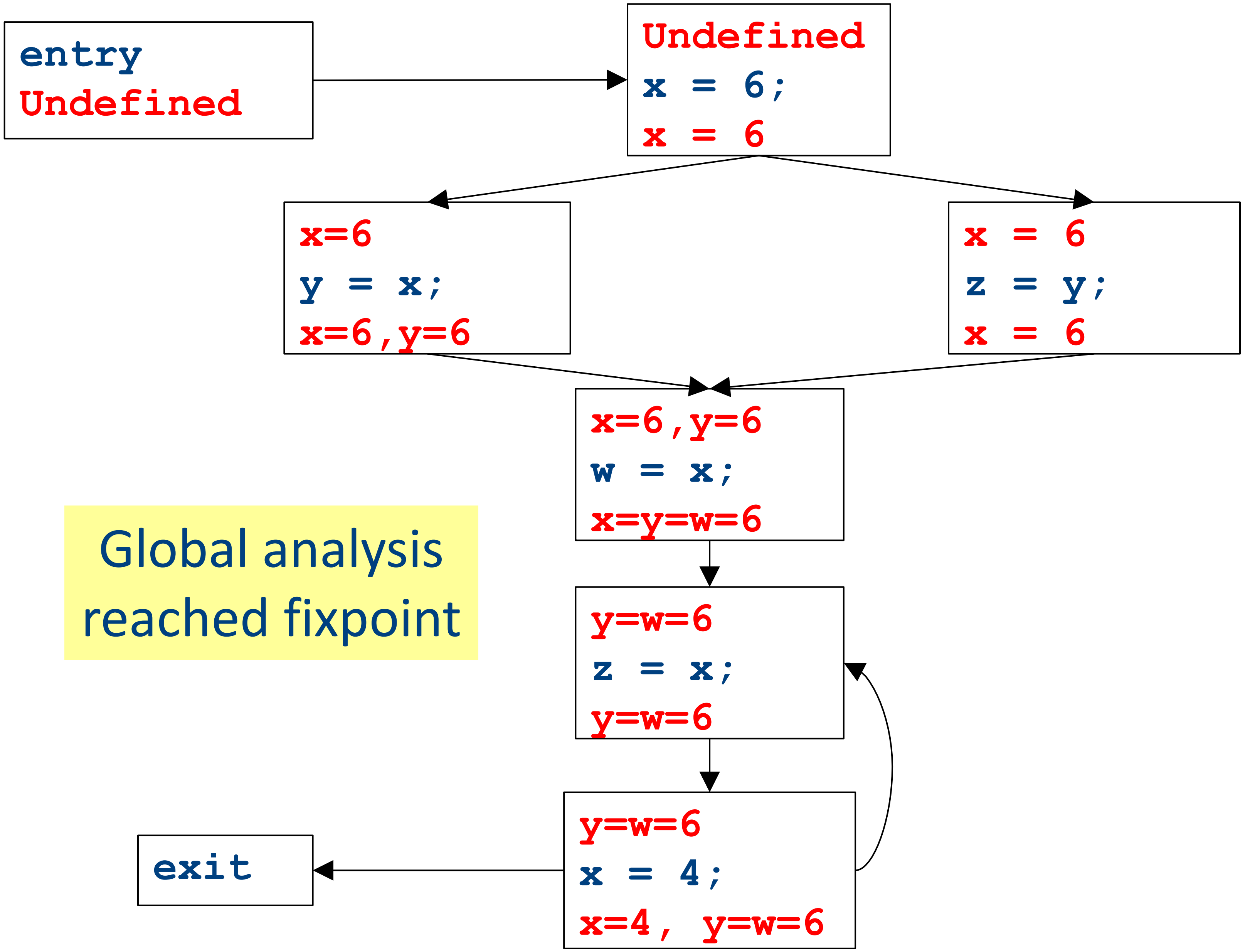
Global constant propagation



Global constant propagation

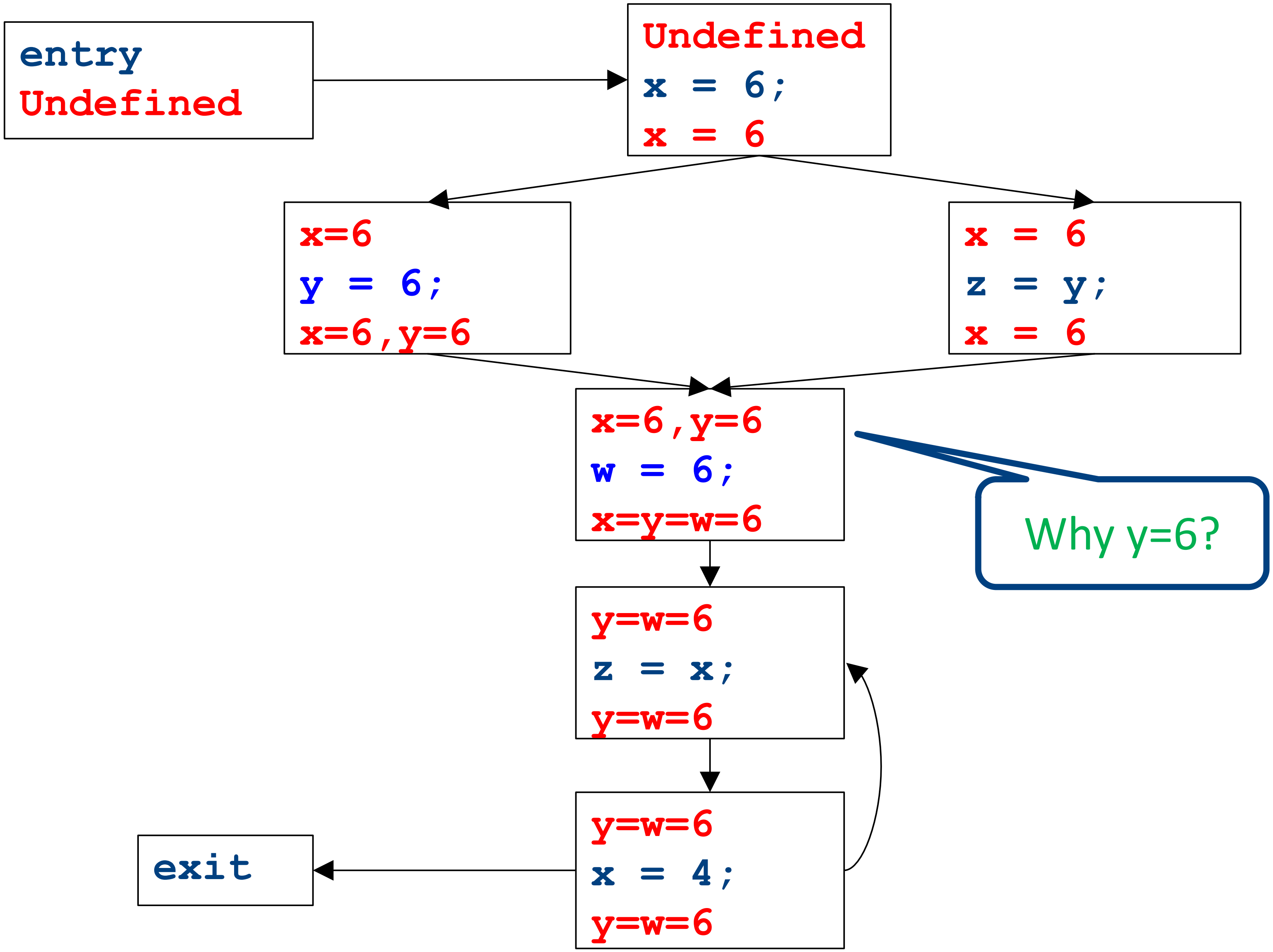


Global constant propagation



Global analysis reached fixpoint

Global constant propagation



Dataflow for constant propagation

- Direction: **Forward**
- Semilattice: Vars→ {Undefined, 0, 1, -1, 2, -2, …, Not-a-Constant}
  - Join mapping for variables point-wise
    $\{x \mapsto 1, y \mapsto 1, z \mapsto 1\} \sqcup \{x \mapsto 1, y \mapsto 2, z \mapsto \text{Not-a-Constant}\} =$
    $\{x \mapsto 1, y \mapsto \text{Not-a-Constant}, z \mapsto \text{Not-a-Constant}\}$
- Transfer functions:
  - $f_{\mathtt{x=k}}(V) = V|_{x \mapsto k}$ *(update V by mapping x to k)*
  - $f_{\mathtt{x=a+b}}(V) = V|_{x \mapsto \text{Not-a-Constant}}$ *(assign Not-a-Constant)*
- Initial value: **x is Undefined**
  - (When might we use some other value?)

Proving termination

- Our algorithm for running these analyses continuously loops until no changes are detected
- Given this, how do we know the analyses will eventually terminate?
  - In general, **we don‘t**

Terminates?

Liveness Analysis

- A variable is live at a point in a program if later in the program its value will be read before it is written to again

Join semilattice definition

- A join semilattice is a pair $(V, \sqcup)$, where
- V is a domain of elements
- $\sqcup$ is a join operator that is
  - commutative: $x \sqcup y = y \sqcup x$
  - associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
  - idempotent: $x \sqcup x = x$
- If $x \sqcup y = z$, we say that z is the join
  or (**L**east **U**pper **B**ound) of x and y
- Every join semilattice has a bottom element
  denoted $\perp$ such that $\perp \sqcup x = x$ for all x

Partial ordering induced by join

- Every join semilattice $(V, \sqcup)$ induces an ordering relationship $\sqsubseteq$ over its elements
- Define $x \sqsubseteq y$ iff x ◀◀ y = y
- Need to prove
  - Reflexivity: $x \sqsubseteq x$
  - Antisymmetry: If $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$
  - Transitivity: If $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$

A join semilattice for liveness

- Sets of live variables and the set union operation
- Idempotent:
  - $x \cup x = x$
- Commutative:
  - $x \cup y = y \cup x$
- Associative:
  - $(x \cup y) \cup z = x \cup (y \cup z)$
- Bottom element:
  - The empty set: $\emptyset \cup x = x$
- Ordering over elements = subset relation

Join semilattice example for liveness

$$\begin{tikzcd}
 & \boxed{\{a,\ b,\ c\}} & \\
\boxed{\{a,\ b\}} \arrow[ur] & \boxed{\{a,\ c\}} \arrow[u] & \boxed{\{b,\ c\}} \arrow[ul] \\
\boxed{\{a\}} \arrow[u] \arrow[ur] & \boxed{\{b\}} \arrow[ul] \arrow[ur] & \boxed{\{c\}} \arrow[ul] \arrow[u] \\
 & \boxed{\{\}} \arrow[ul] \arrow[u] \arrow[ur] & \text{Bottom element} \arrow[l, dash]
\end{tikzcd}$$

Dataflow framework

- A global analysis is a tuple $(\mathtt{D}, \mathtt{V}, \sqcup, \mathtt{F}, \mathtt{I})$, where
  - $\mathtt{D}$ is a direction (forward or backward)
    - The order to visit statements within a basic block, **NOT** the order in which to visit the basic blocks
  - $\mathtt{V}$ is a set of values (sometimes called domain)
  - $\sqcup$ is a join operator over those values
  - $\mathtt{F}$ is a set of transfer functions f_s : **V → V** (for every statement s)
  - $\mathtt{I}$ is an initial value

Running global analyses

- Assume that $(\mathbb{D}, \mathbb{V}, \sqcup, \mathbb{F}, \mathbb{I})$ is a forward analysis
- For every statement s maintain values before - IN[s] - and after - OUT[s]
- Set OUT[**s**] = $\perp$ for all statements **s**
- Set OUT[**entry**] = $\mathbb{I}$
- Repeat until no values change:
  - For each statement **s** with predecessors PRED[s]={$\mathbf{p}_1$, $\mathbf{p}_2$, … , $\mathbf{p}_n$}
    - Set IN[**s**] = OUT[$\mathbf{p}_1$] $\sqcup$ OUT[$\mathbf{p}_2$] $\sqcup$ … $\sqcup$ OUT[$\mathbf{p}_n$]
    - Set OUT[**s**] = f_s(IN[**s**])
- The order of this iteration does not matter
  - Chaotic iteration

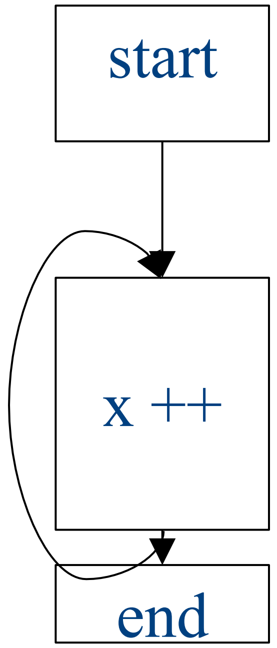
Proving termination

- Our algorithm for running these analyses continuously loops until no changes are detected
- Problem: how do we know the analyses will eventually terminate?

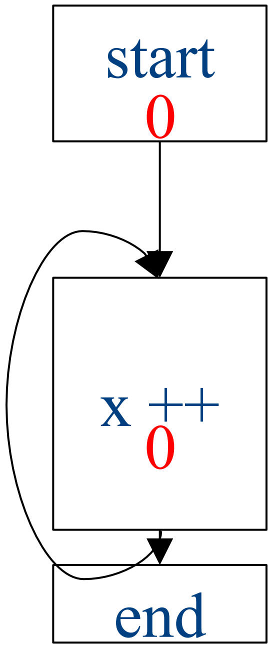
A non-terminating analysis

- The following analysis will loop infinitely on any CFG containing a loop:
- Direction: Forward
- Domain: $\mathbb{N}$
- Join operator: **max**
- Transfer function: $f(n) = n + 1$
- Initial value: 0

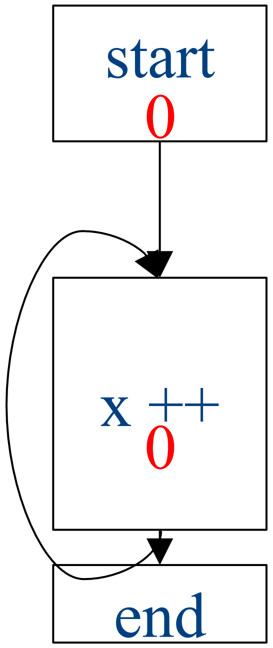
A non-terminating analysis



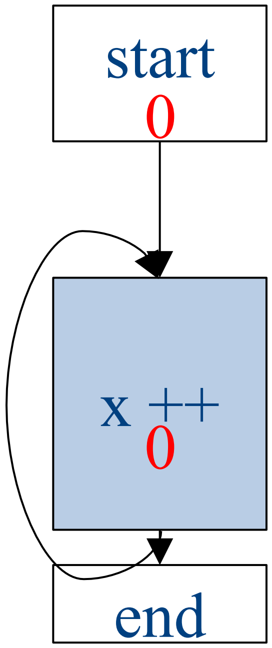
Initialization



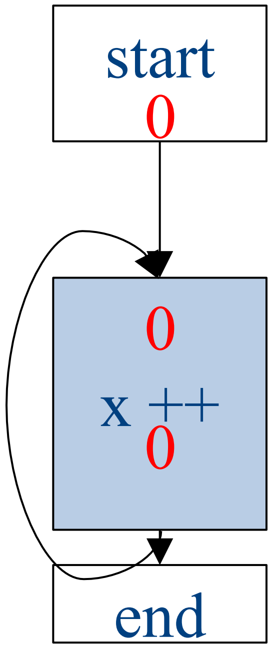
Fixed-point iteration



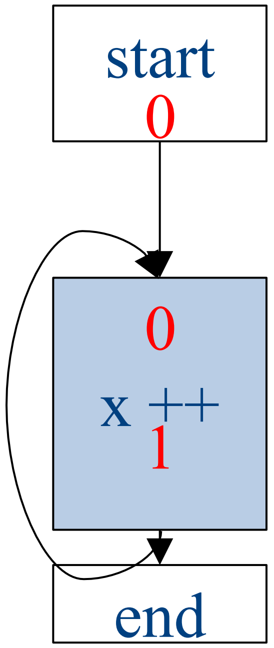
Choose a block



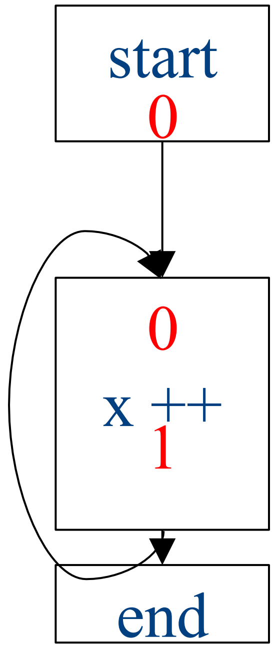
Iteration 1



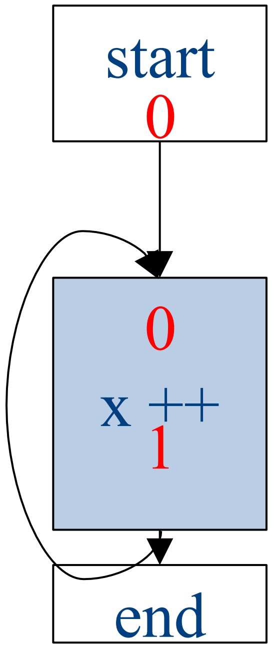
Iteration 1



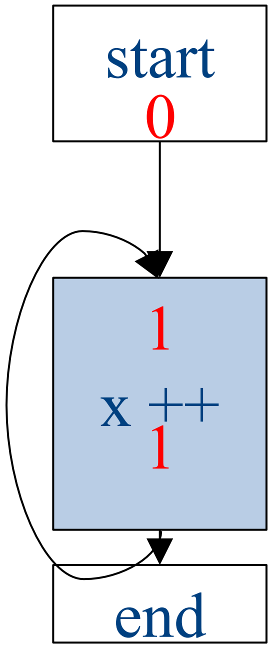
Choose a block



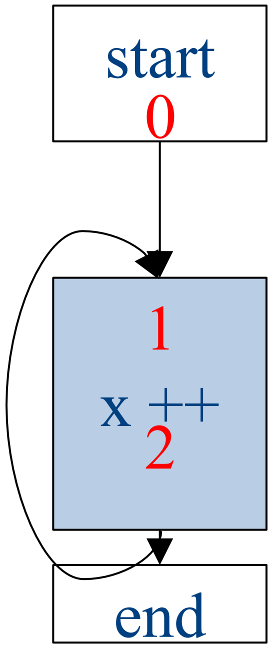
Iteration 2



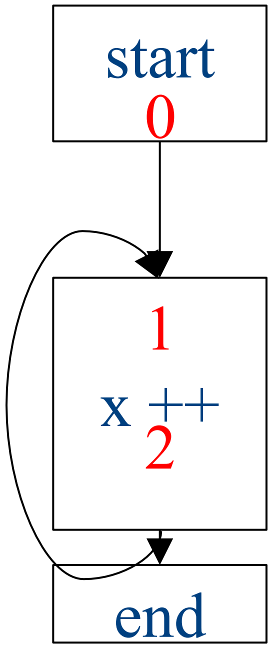
Iteration 2



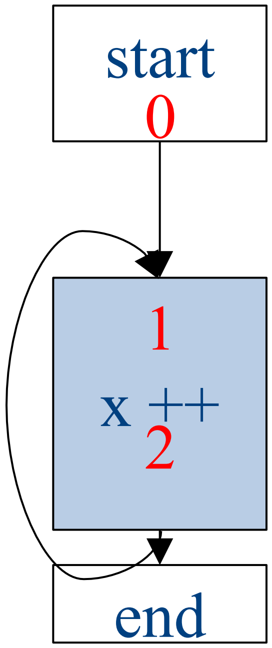
Iteration 2



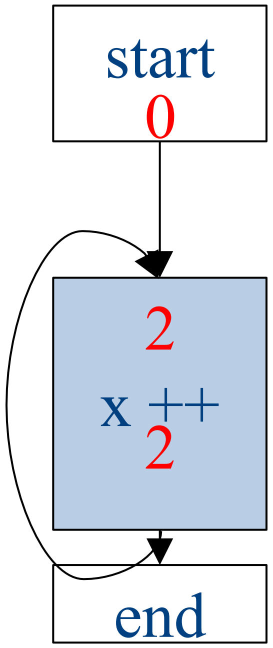
Choose a block



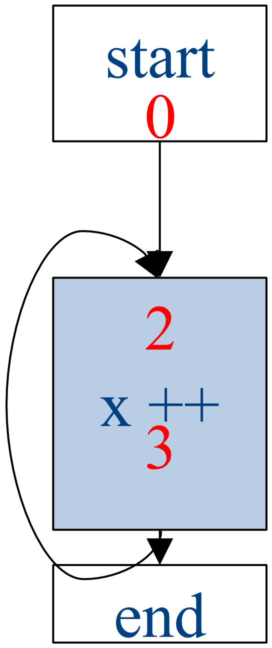
Iteration 3



Iteration 3

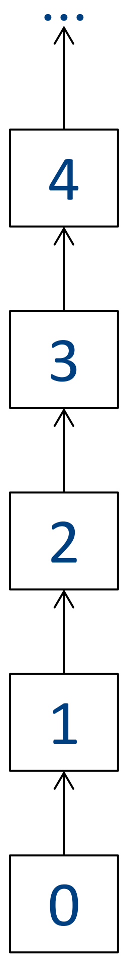


Iteration 3



Why doesn't this terminate?

- Values can increase without bound
- Note that "increase" refers to the lattice ordering, not the ordering on the natural numbers
- The height of a semilattice is the length of the longest increasing sequence in that semilattice
- The dataflow framework is not guaranteed to terminate for semilattices of infinite height
- Note that a semilattice can be infinitely large but have finite height
  - e.g. constant propagation



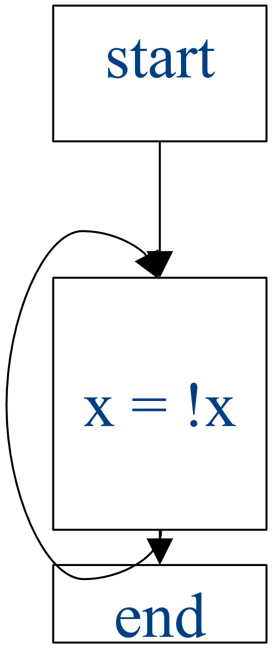
Height of a lattice

- An increasing chain is a sequence of elements $\perp \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \ldots \sqsubseteq a_k$
  - The length of such a chain is k
- The height of a lattice is the length of the maximal increasing chain
- For liveness with *n* program variables:
  - $\{\} \subseteq \{v_1\} \subseteq \{v_1,v_2\} \subseteq \ldots \subseteq \{v_1,\ldots,v_n\}$
- For available expressions it is the number of expressions of the form a=b op c
  - For *n* program variables and *m* operator types:mn^3

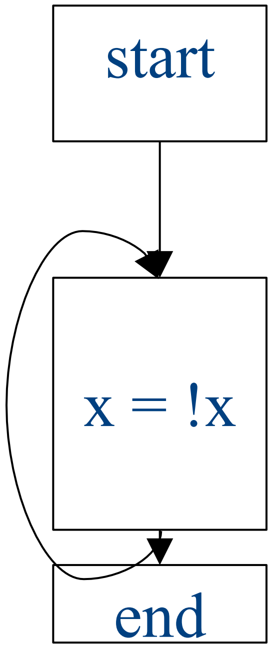
Another non-terminating analysis

- This analysis works on a finite-height semilattice, but will not terminate on certain CFGs:
- Direction: Forward
- Domain: Boolean values **`true`** and **`false`**
- Join operator: Logical OR
- Transfer function: Logical NOT
- Initial value: **`false`**

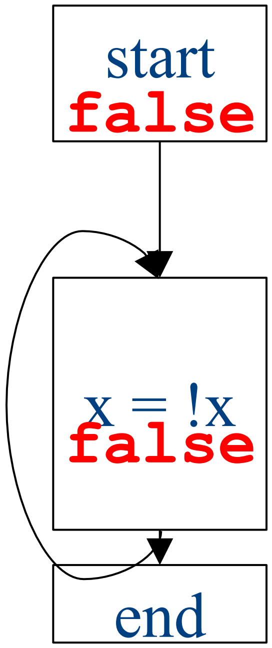
A non-terminating analysis



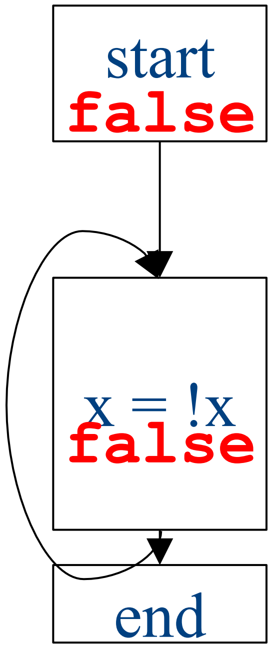
A non-terminating analysis



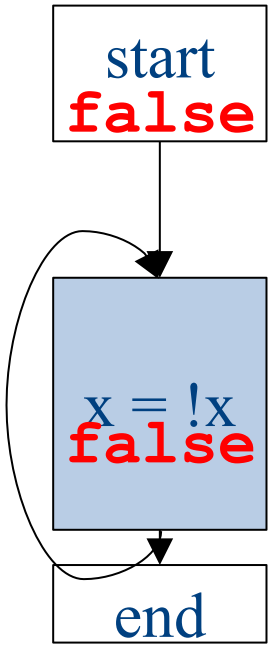
Initialization



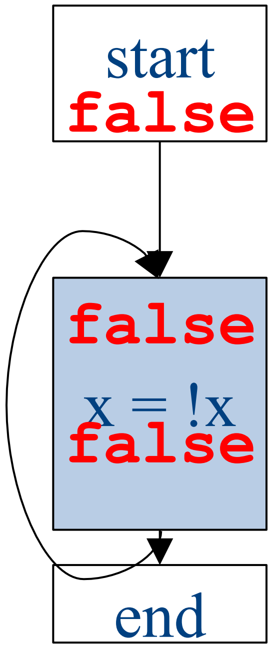
Fixed-point iteration



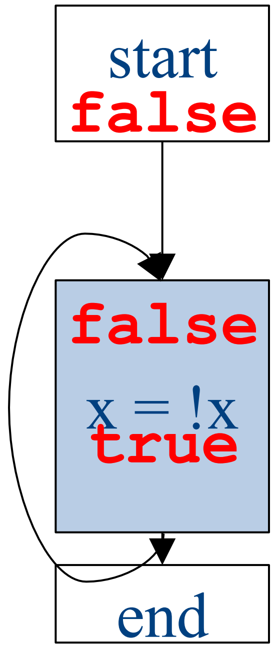
Choose a block



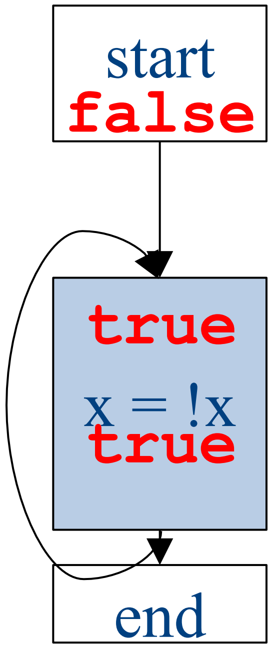
Iteration 1



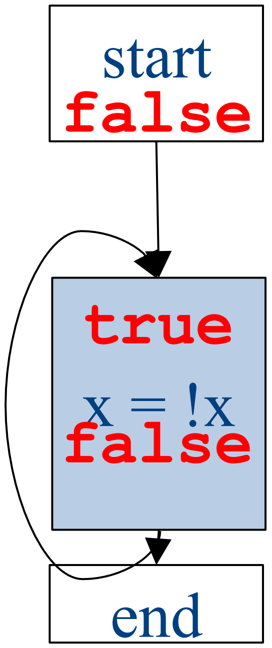
Iteration 1



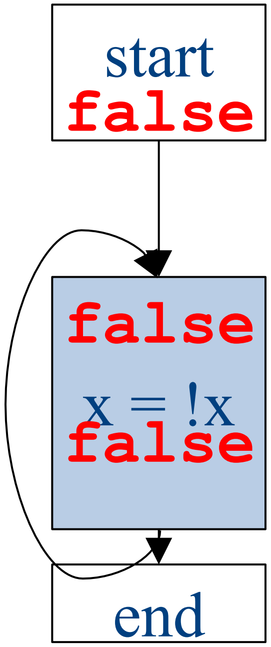
Iteration 2



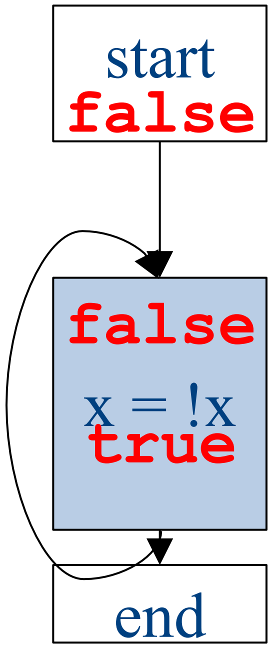
Iteration 2



Iteration 3

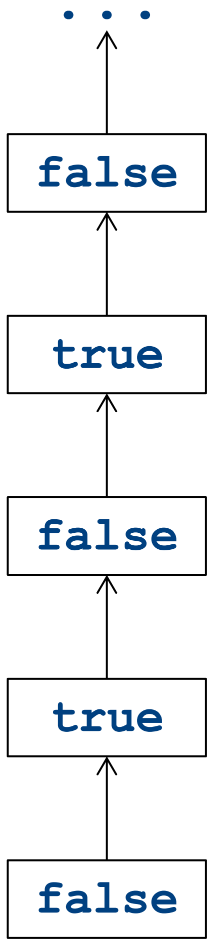


Iteration 3



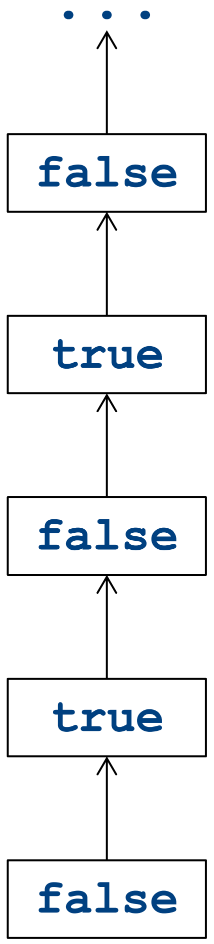
Why doesn't it terminate?

- Values can loop indefinitely
- Intuitively, the join operator keeps pulling values up
- If the transfer function can keep pushing values back down again, then the values might cycle forever



Why doesn't it terminate?

- Values can loop indefinitely
- Intuitively, the join operator keeps pulling values up
- If the transfer function can keep pushing values back down again, then the values might cycle forever
- How can we fix this?



Monotone transfer functions

- A transfer function *f* is monotone iff
  if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$
- Intuitively, if you know less information about a program point, you can't “gain back” more information about that program point
- Many transfer functions are monotone, including those for liveness and constant propagation
- Note: Monotonicity does **not** mean that $x \sqsubseteq f(x)$
  - (This is a different property called extensivity)

Liveness and monotonicity

- A transfer function f is monotone iff

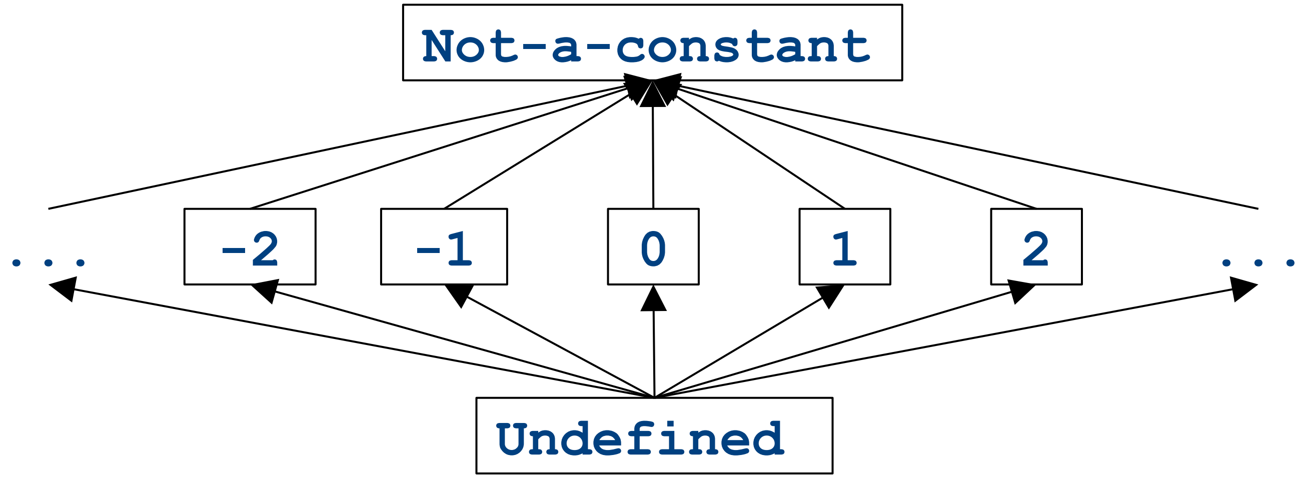
  if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$
- Recall our transfer function for **a = b + c** is
  - $f_{a = b + c}(V) = (V - \{a\}) \cup \{b, c\}$
- Recall that our join operator is set union and induces an ordering relationship

  $X \sqsubseteq Y$ iff $X \subseteq Y$
- Is this monotone?

Is constant propagation monotone?

- A transfer function *f* is monotone iff

  if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$
- Recall our transfer functions
  - $f_{\mathtt{x=k}}(V) = V[x \mapsto k]$ *(update V by mapping x to k)*
  - $f_{\mathtt{x=a+b}}(V) = V[x \mapsto \text{Not-a-Constant}]$ *(assign Not-a-Constant)*
- Is this monotone?



The grand result

- Theorem: A dataflow analysis with a **finite-height semilattice** and family of **monotone transfer functions** always terminates
- Proof sketch:
  - The join operator can only bring values up
  - Transfer functions can never lower values back down below where they were in the past (monotonicity)
  - Values cannot increase indefinitely (finite height)

An “optimality” result

- A transfer function *f* is distributive if
  $$f(a \sqcup b) = f(a) \sqcup f(\mathrm{b})$$
  for every domain elements *a* and *b*
- If all transfer functions are distributive then the fixed-point solution is the solution that would be computed by joining results from all (potentially infinite) control-flow paths
  - Join over all paths
- Optimal if we ignore program conditions

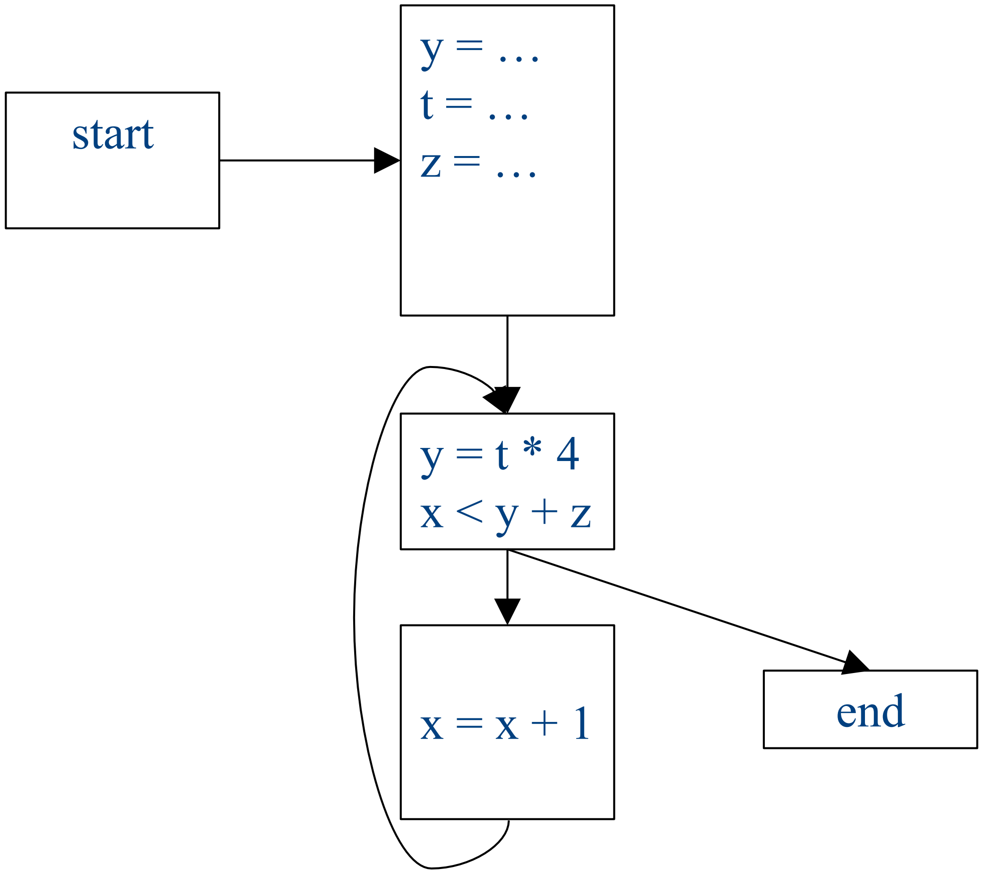
An “optimality” result

- A transfer function *f* is distributive if
  $$f(a \sqcup b) = f(a) \sqcup f(b)$$
  for every domain elements *a* and *b*
- If all transfer functions are distributive then the fixed-point solution is equal to the solution computed by joining results from all (potentially infinite) control-flow paths
  - Join over all paths
- Optimal if we pretend all control-flow paths can be executed by the program
- Which analyses use distributive functions?

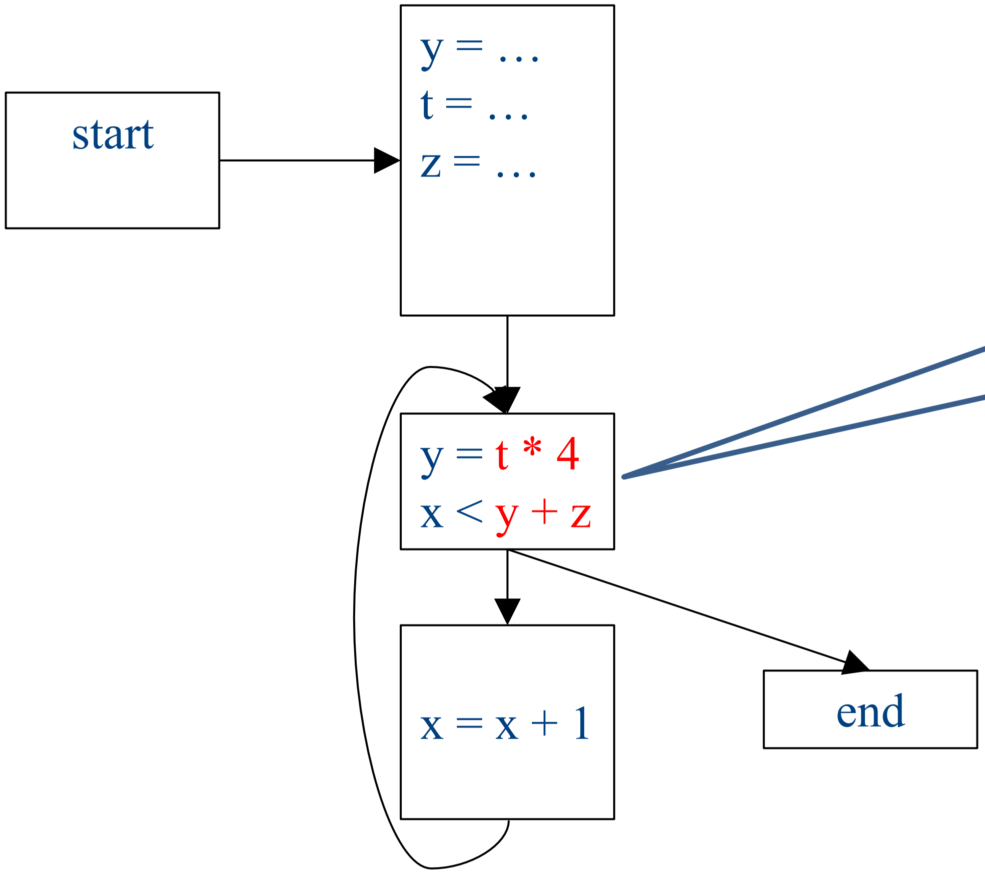
Loop optimizations

- Most of a program’s computations are done inside loops
  - Focus optimizations effort on loops
- The optimizations we’ve seen so far are independent of the control structure
- Some optimizations are specialized to loops
  - Loop-invariant code motion
  - (Strength reduction via induction variables)
- Require another type of analysis to find out where expressions get their values from
  - Reaching definitions
    - (Also useful for improving register allocation)

Loop invariant computation

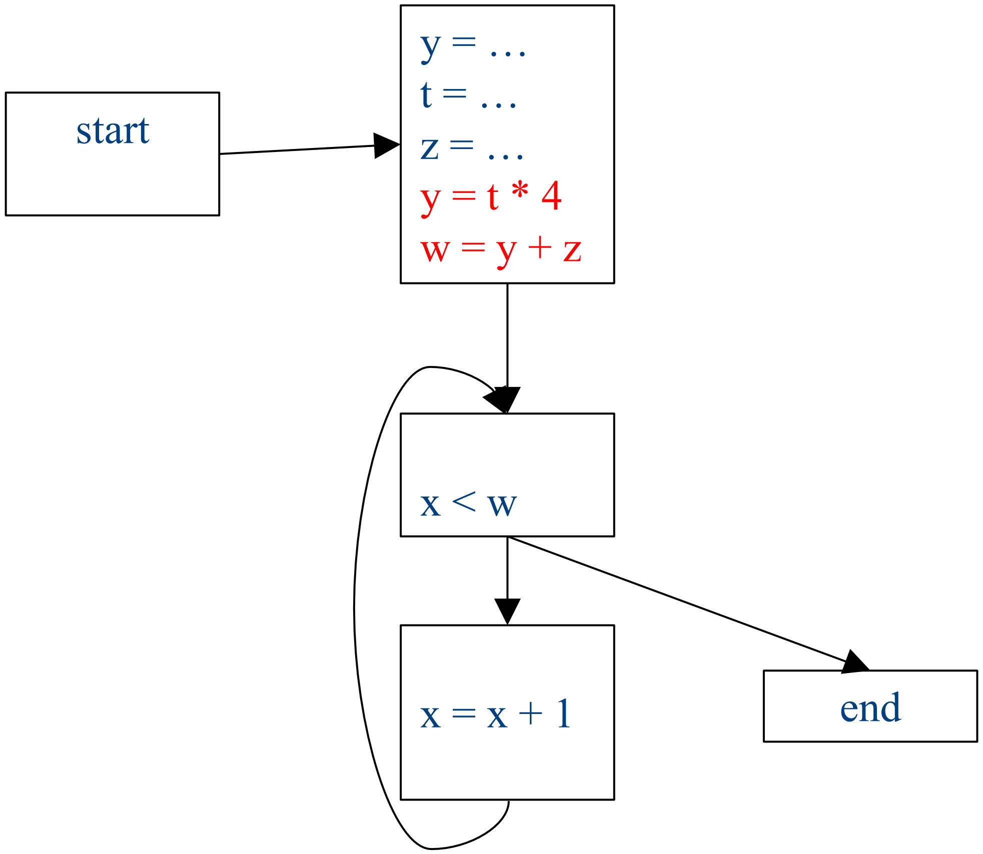


Loop invariant computation

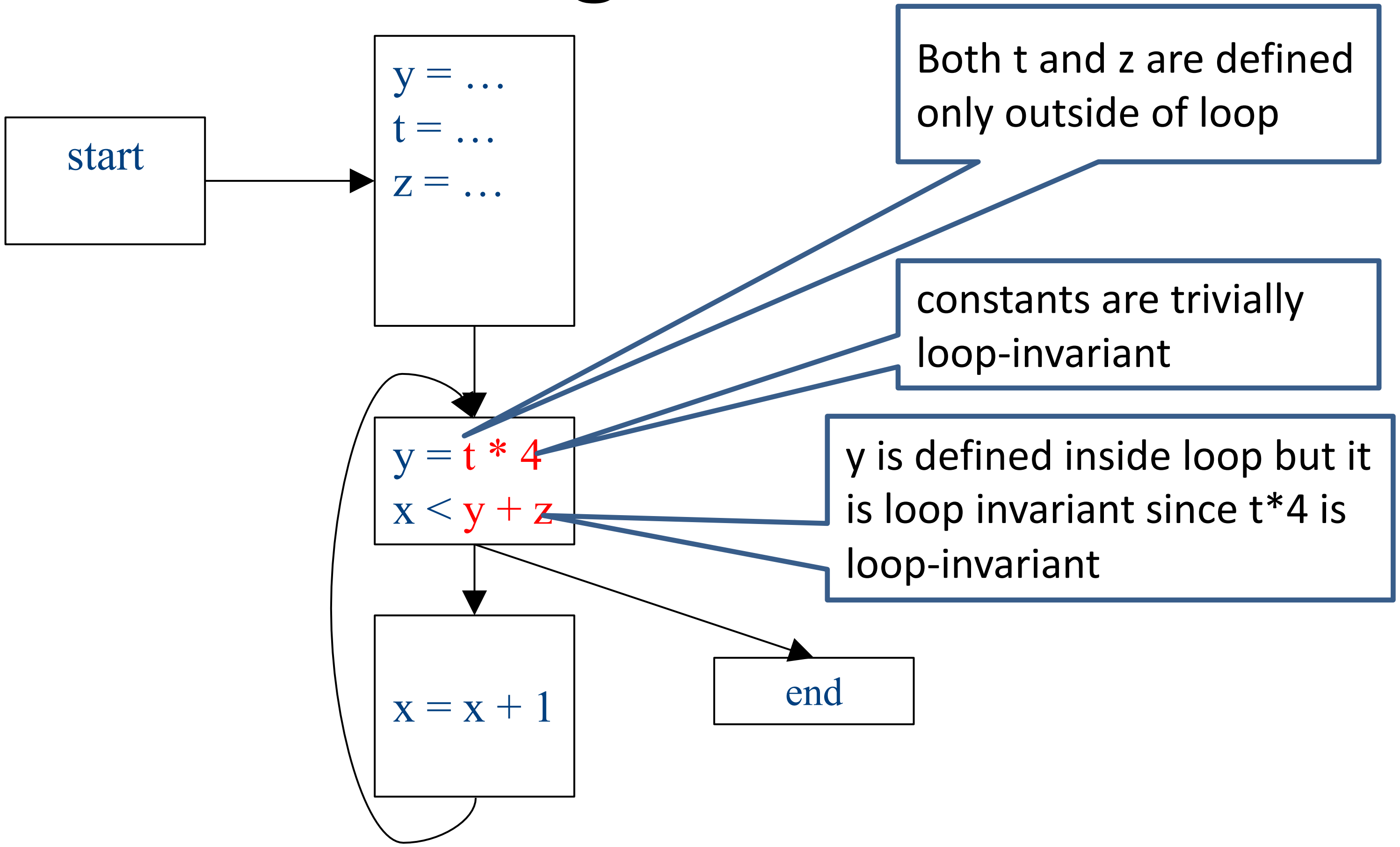


t*4 and y+z have same value on each iteration

Code hoisting

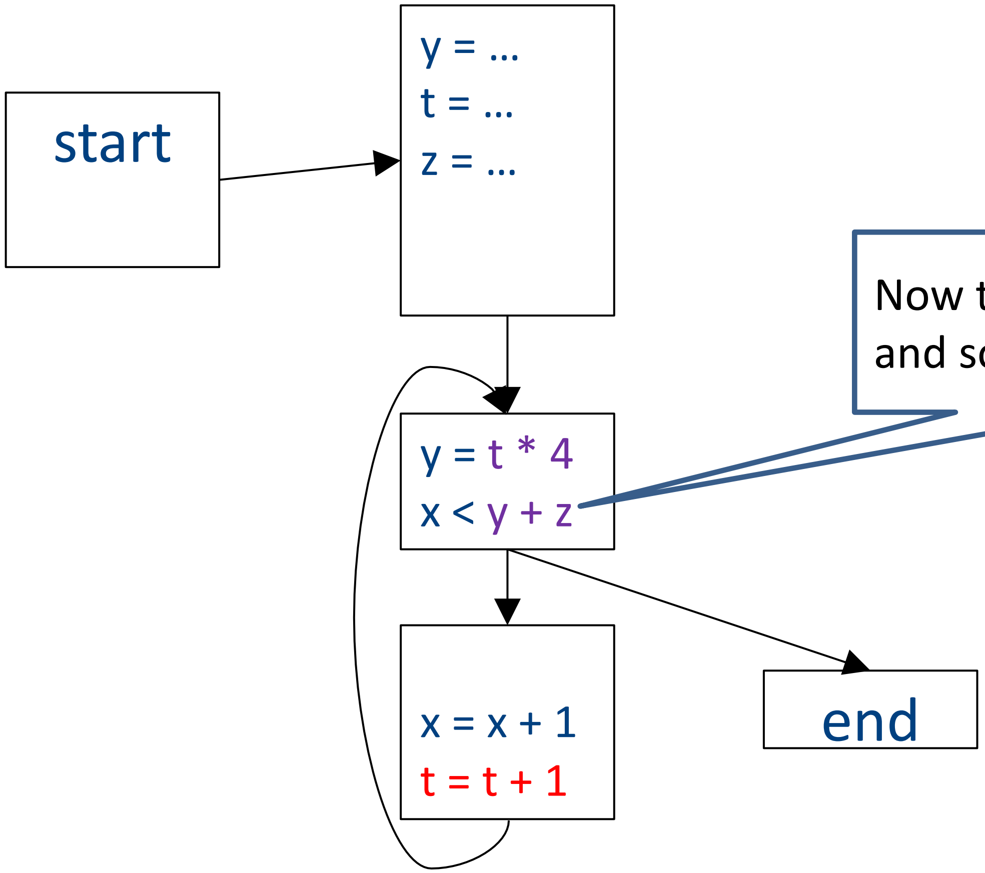


What reasoning did we use?



start
y = …
t = …
z = …
y = t * 4
x < y + z
x = x + 1
end
Both t and z are defined only outside of loop
constants are trivially loop-invariant
y is defined inside loop but it is loop invariant since t*4 is loop-invariant

What about now?



Now t is not loop-invariant and so are t*4 and y

Loop-invariant code motion

- *d*: t = a_1 op a_2
  - *d* is a program location
- a_1 op a_2 loop-invariant (for a loop *L*) if computes the same value in each iteration
  - Hard to know in general
- Conservative approximation
  - Each a_i is a constant, or
  - All definitions of a_i that reach *d* are outside *L*, or
  - Only one definition of of a_i reaches *d*, and is loop-invariant itself
- Transformation: hoist the loop-invariant code outside of the loop

Reaching definitions analysis

- A definition *d*: t = … reaches a program location if there is a path from the definition to the program location, along which the defined variable is never redefined

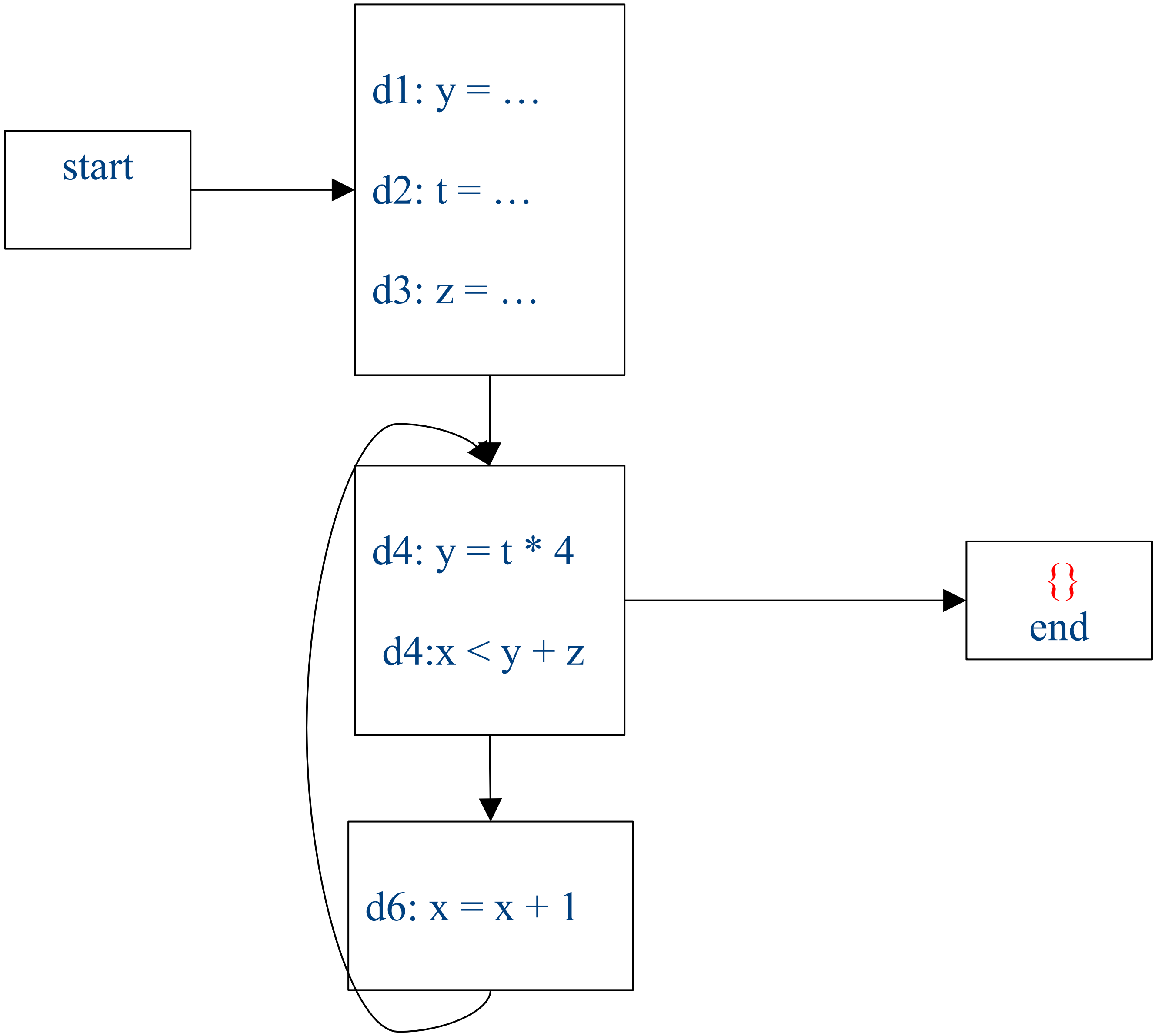
Reaching definitions analysis

- A definition *d*: t = … reaches a program location if there is a path from the definition to the program location, along which the defined variable is never redefined
- Direction: Forward
- Domain: sets of program locations that are definitions `
- Join operator: union
- Transfer function:

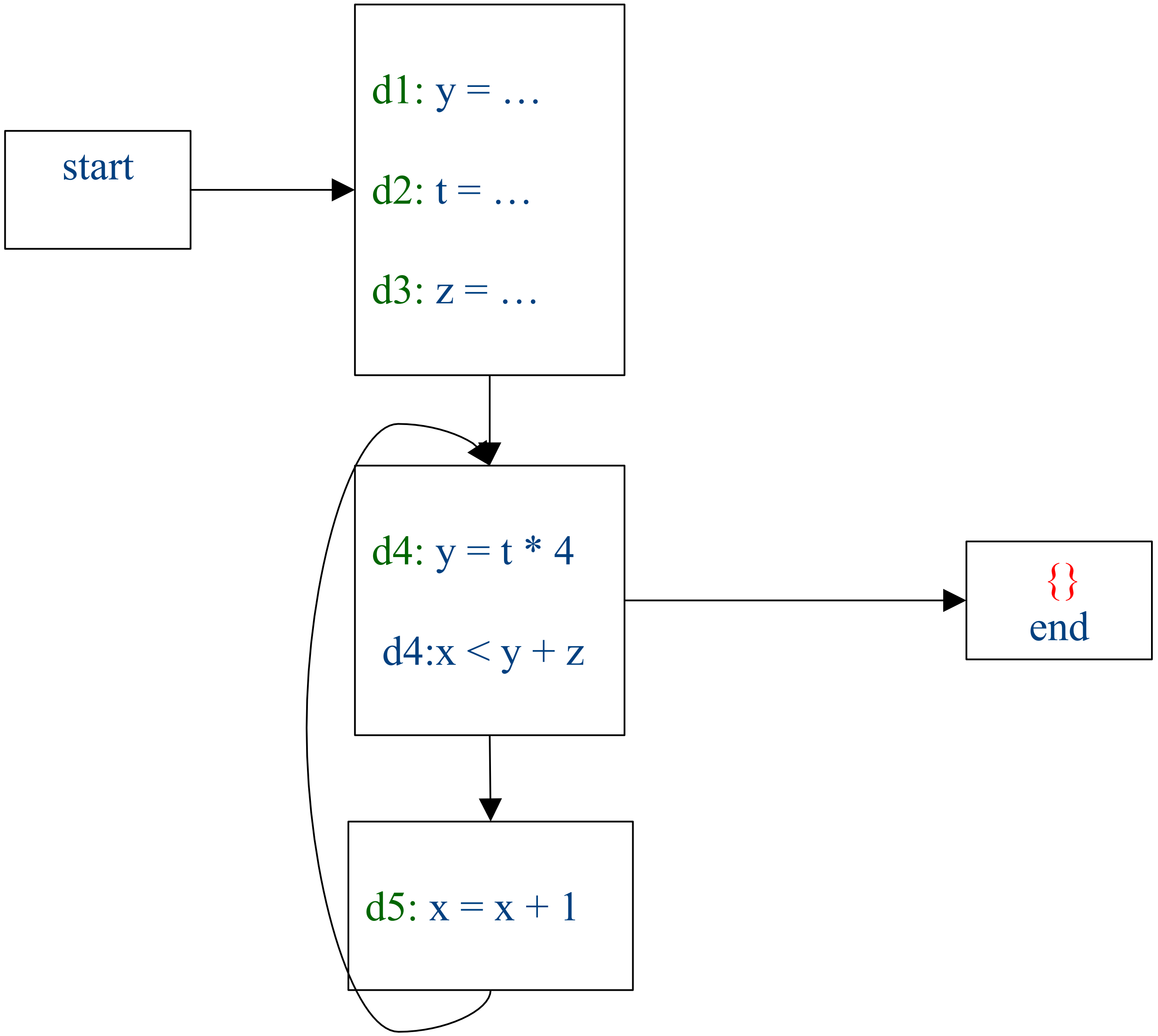
  $$f_{d:\ a=b\ op\ c}(RD) = (RD - defs(a)) \cup \{d\}$$
  $$f_{d:\ not\text{-}a\text{-}def}(RD) = RD$$

  – Where *defs*(*a*) is the set of locations defining *a* (statements of the form *a*=...)
- Initial value: {}

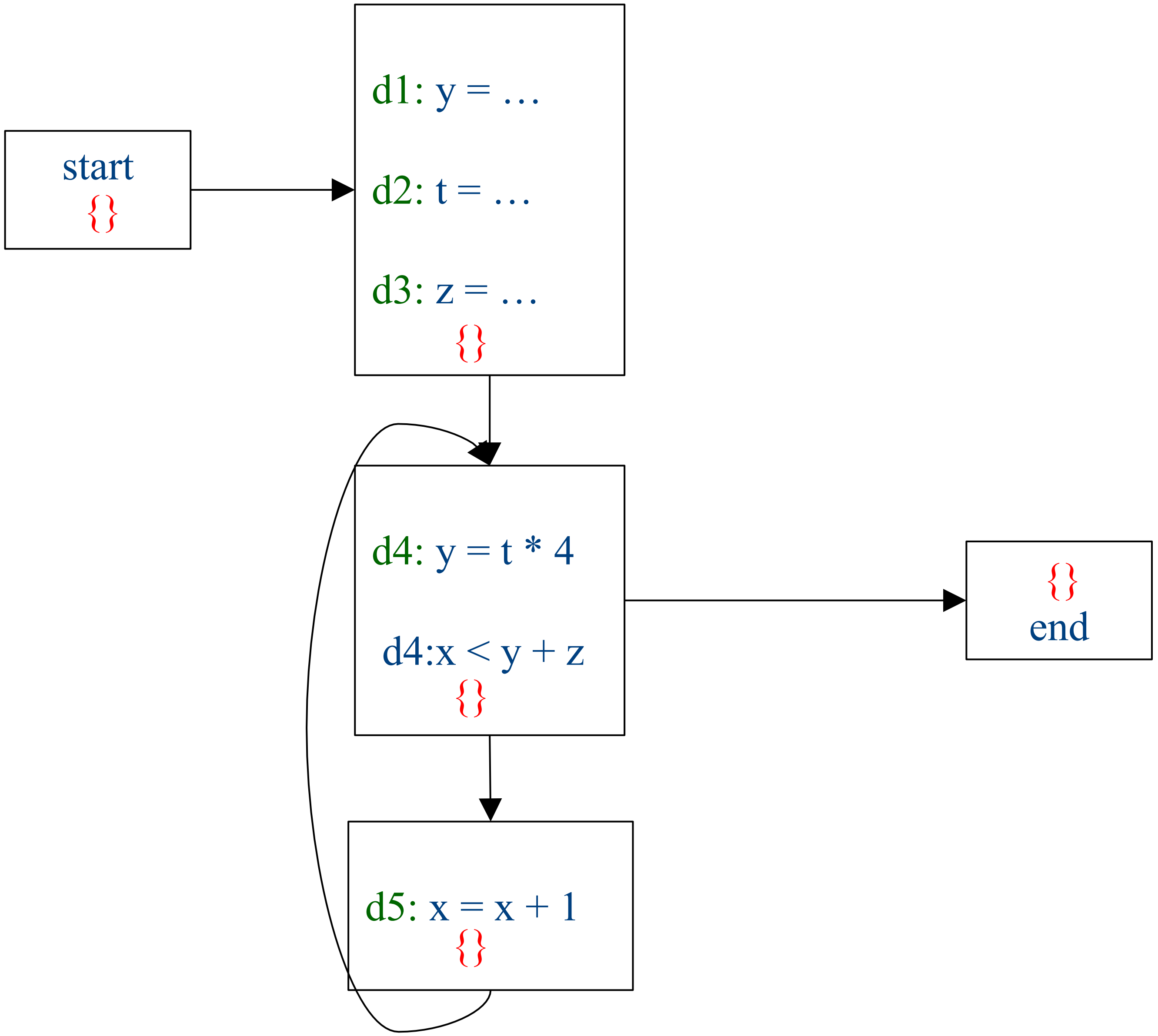
Reaching definitions analysis



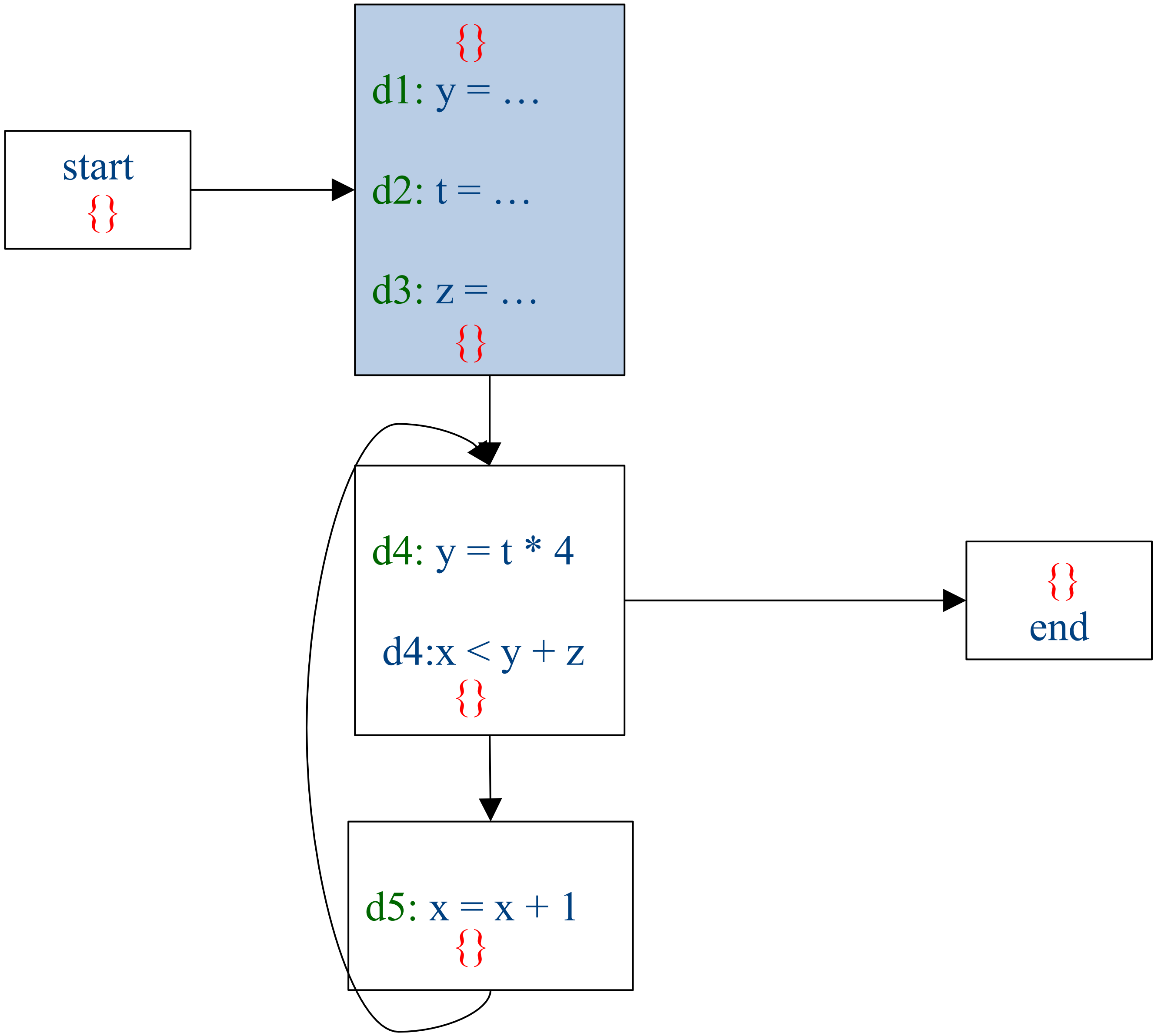
Reaching definitions analysis



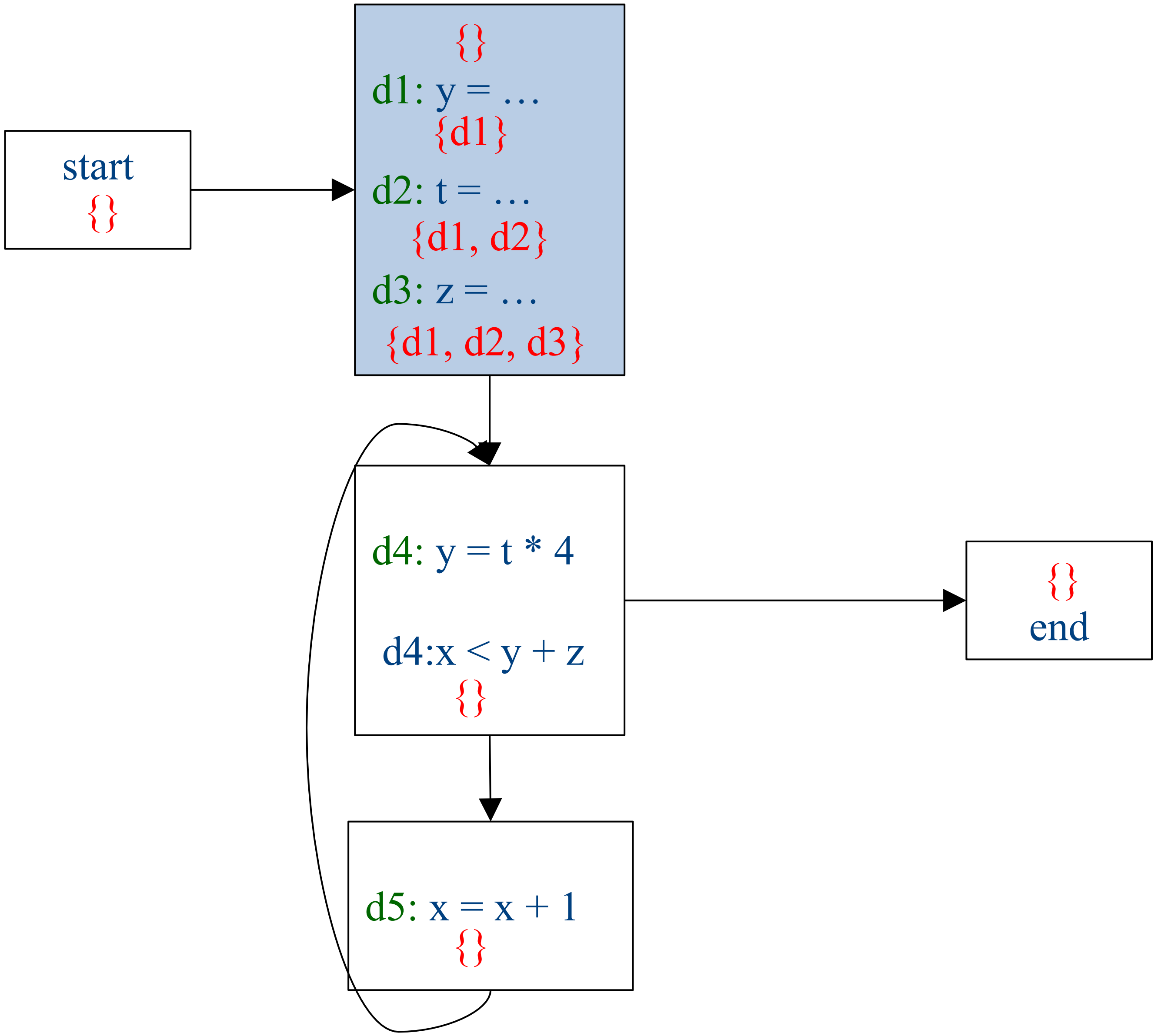
Initialization



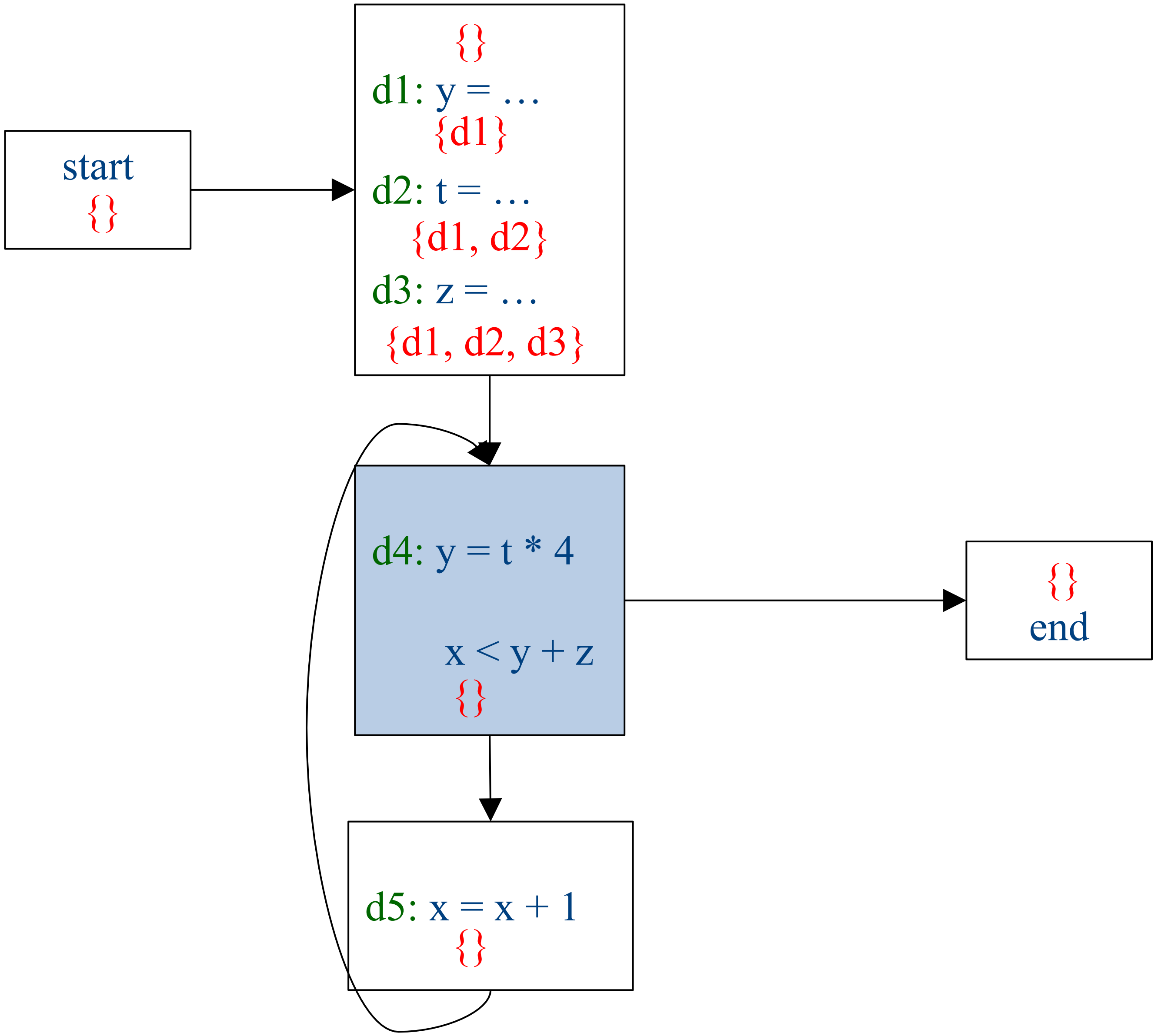
Iteration 1



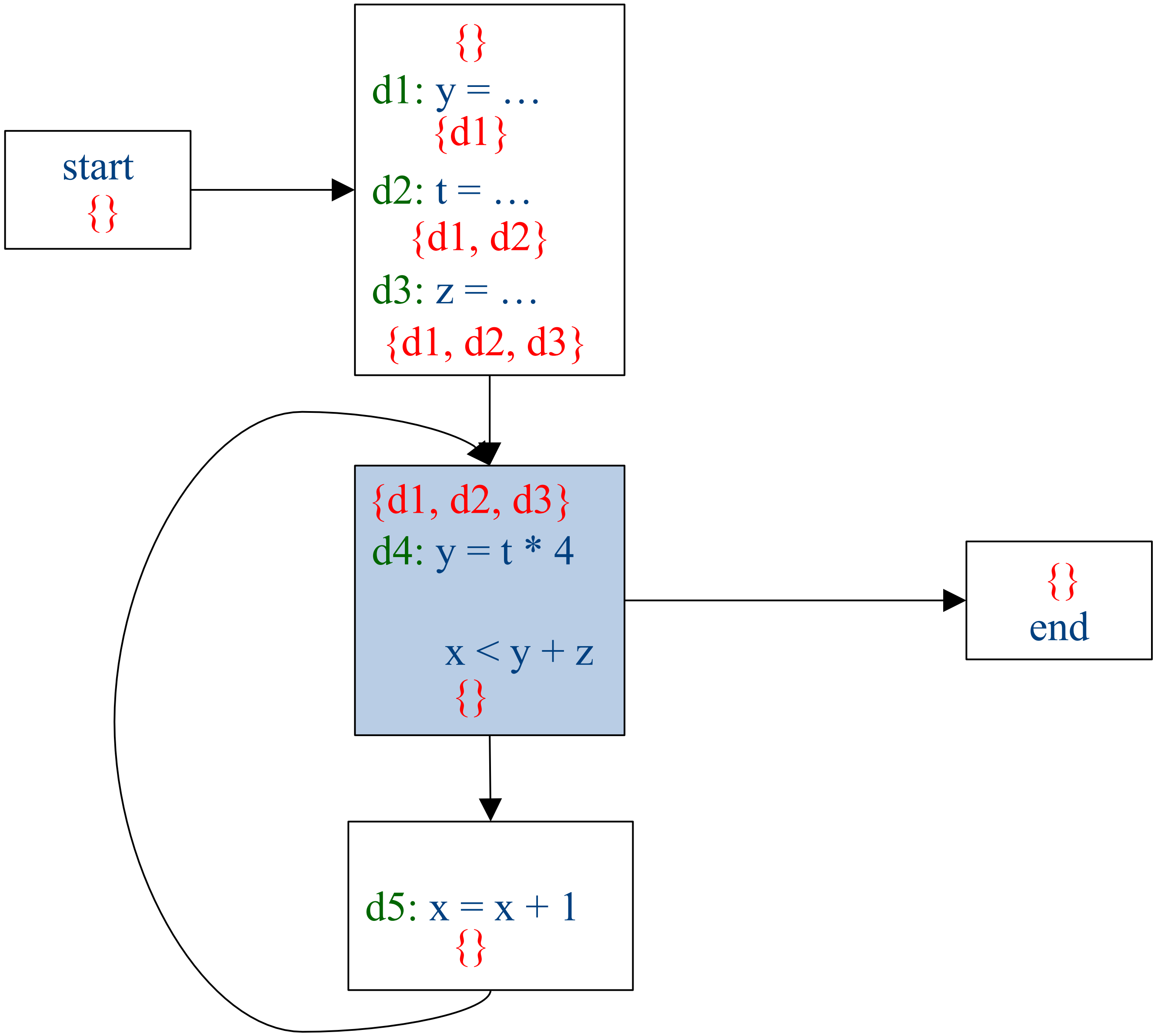
Iteration 1



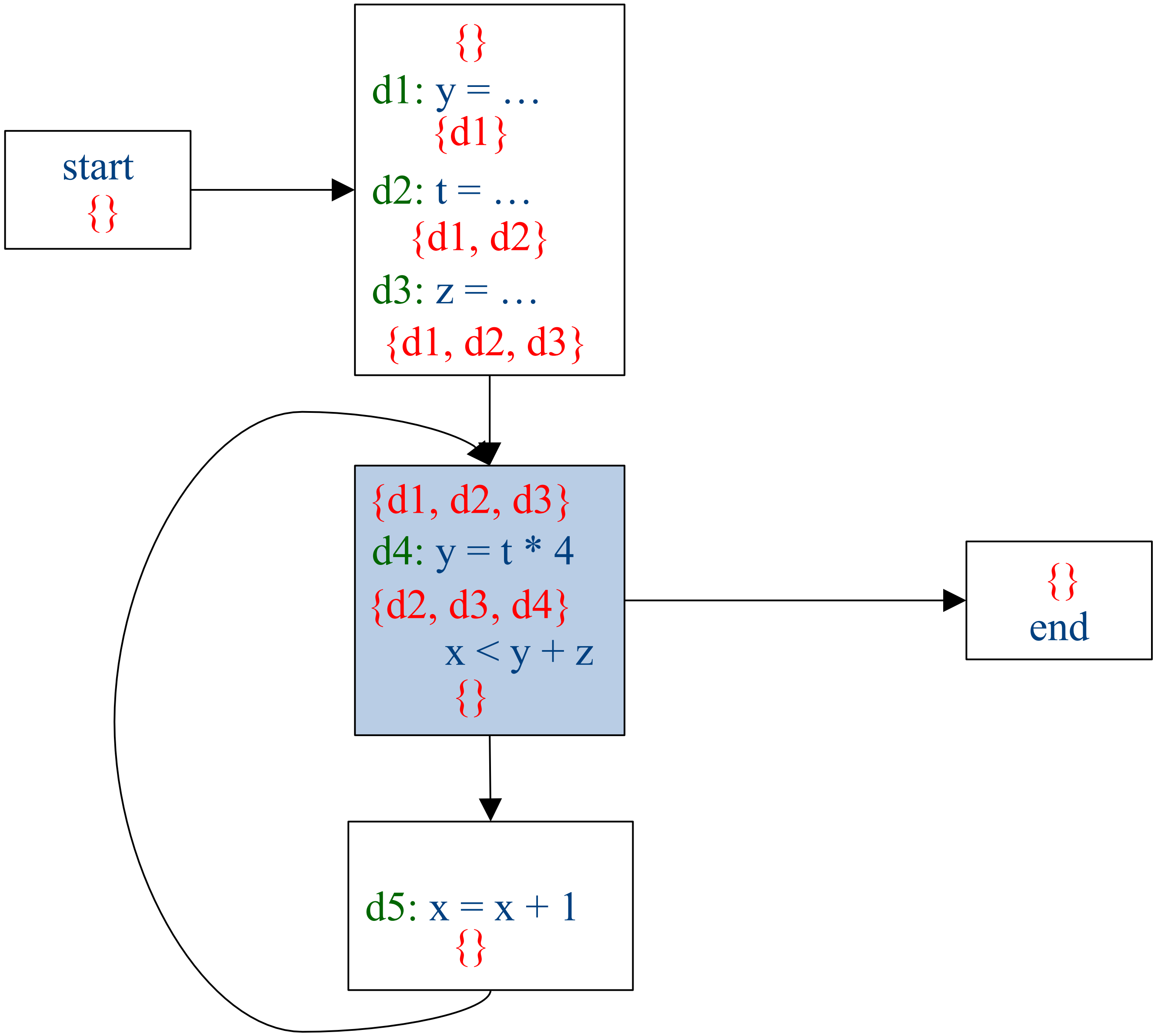
Iteration 2



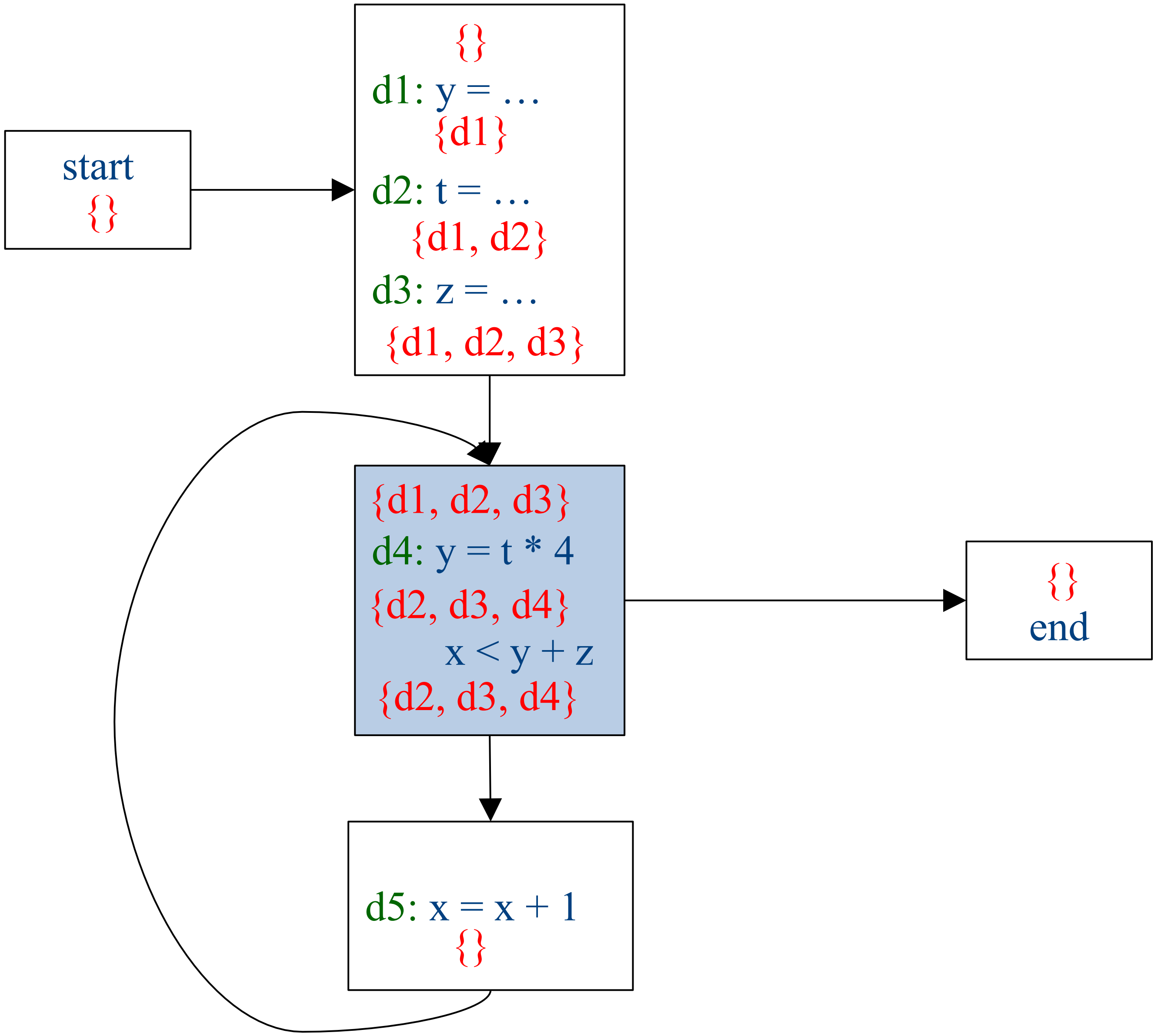
Iteration 2



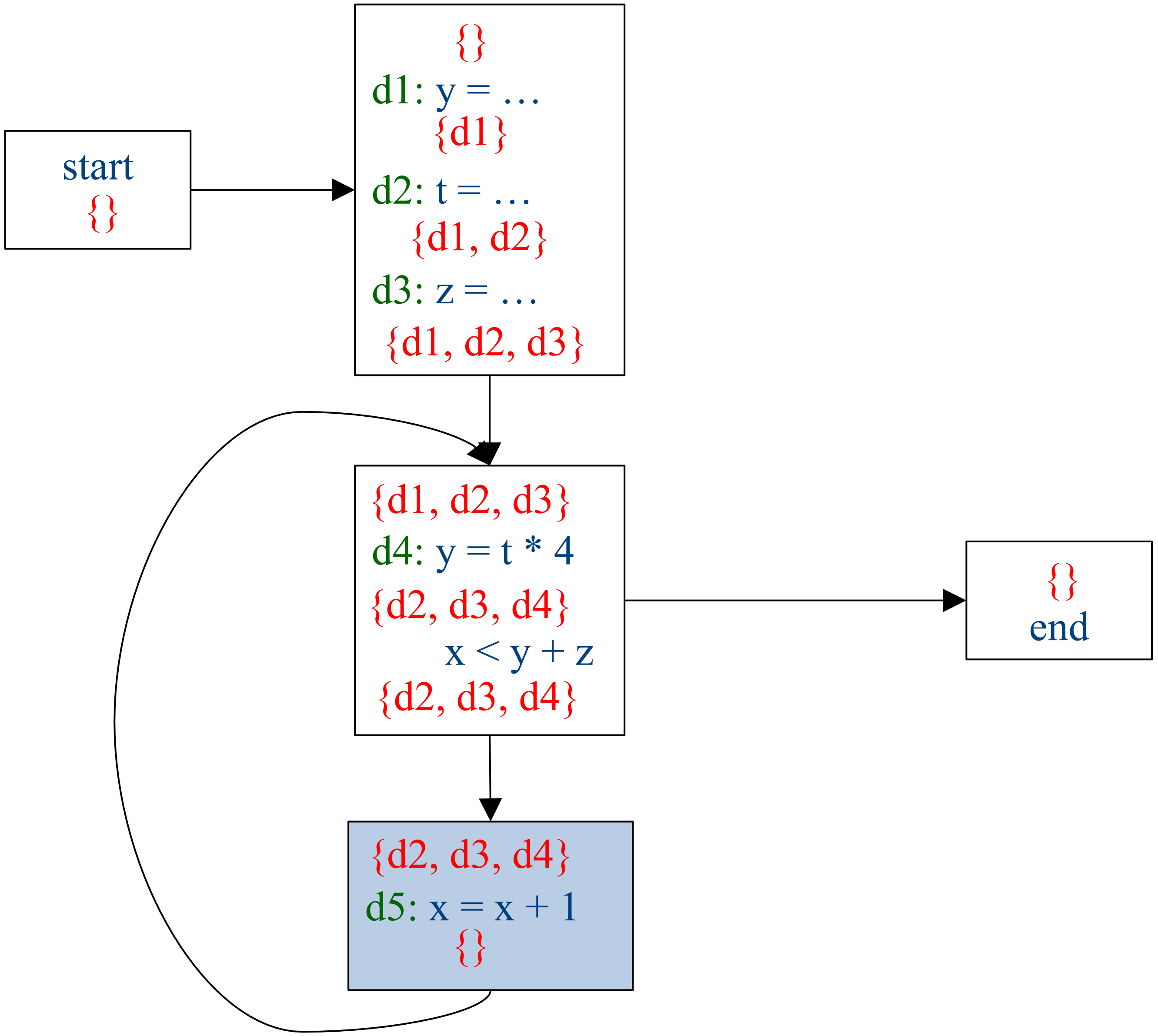
Iteration 2


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{}
{}
end
d5: x = x + 1
{}

Iteration 2

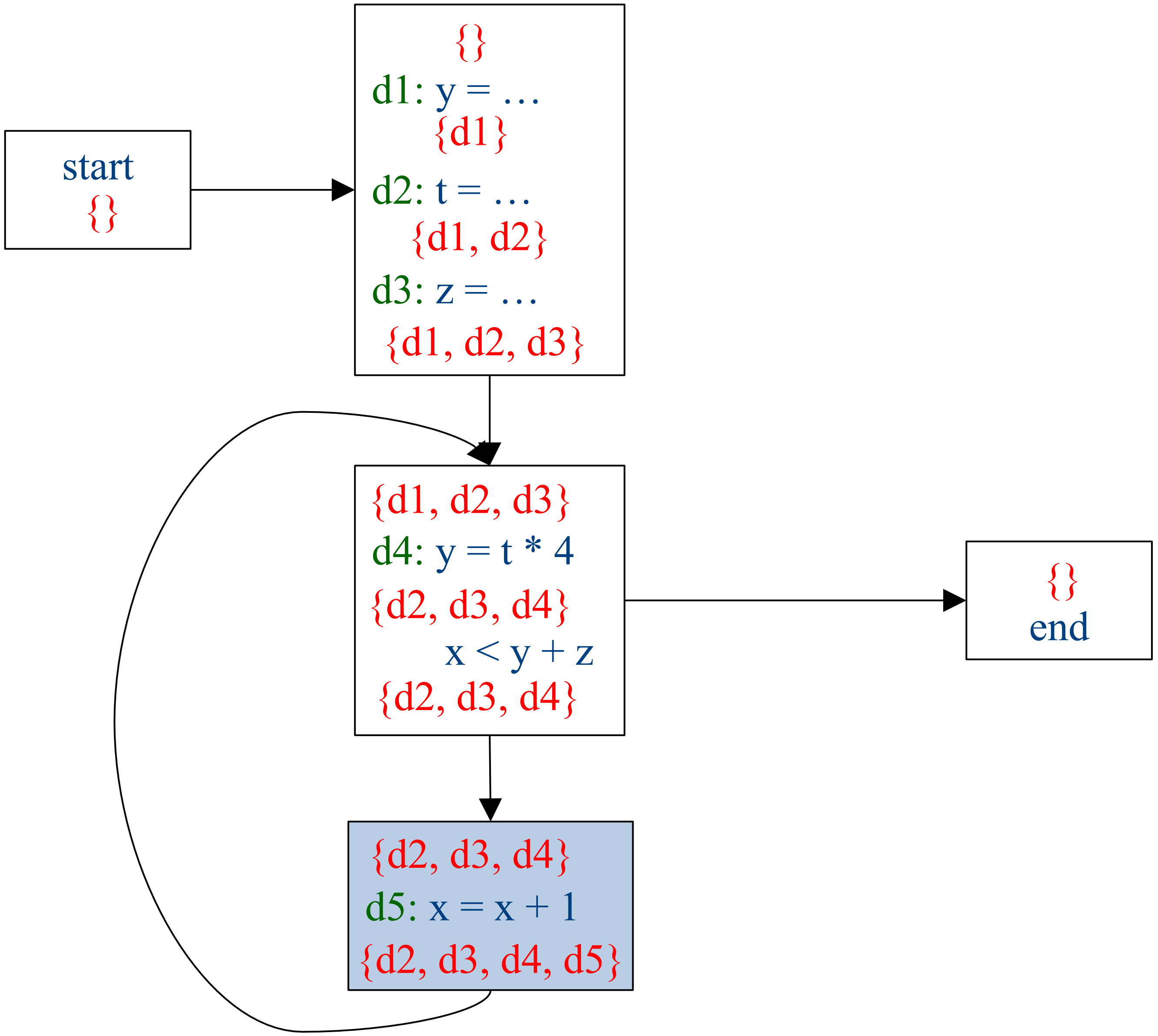

start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
d5: x = x + 1
{}

Iteration 3

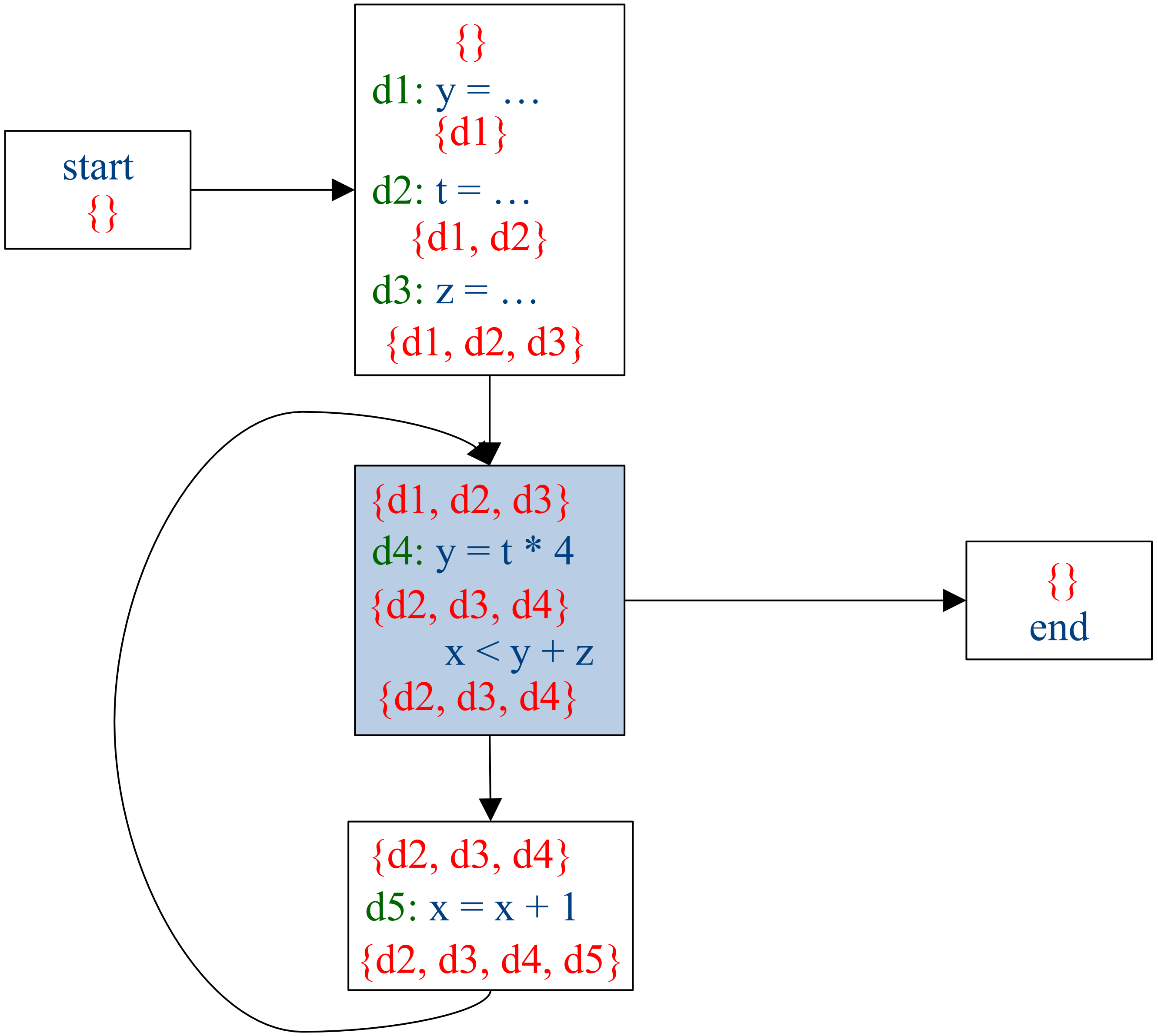


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
{d2, d3, d4}
d5: x = x + 1
{}

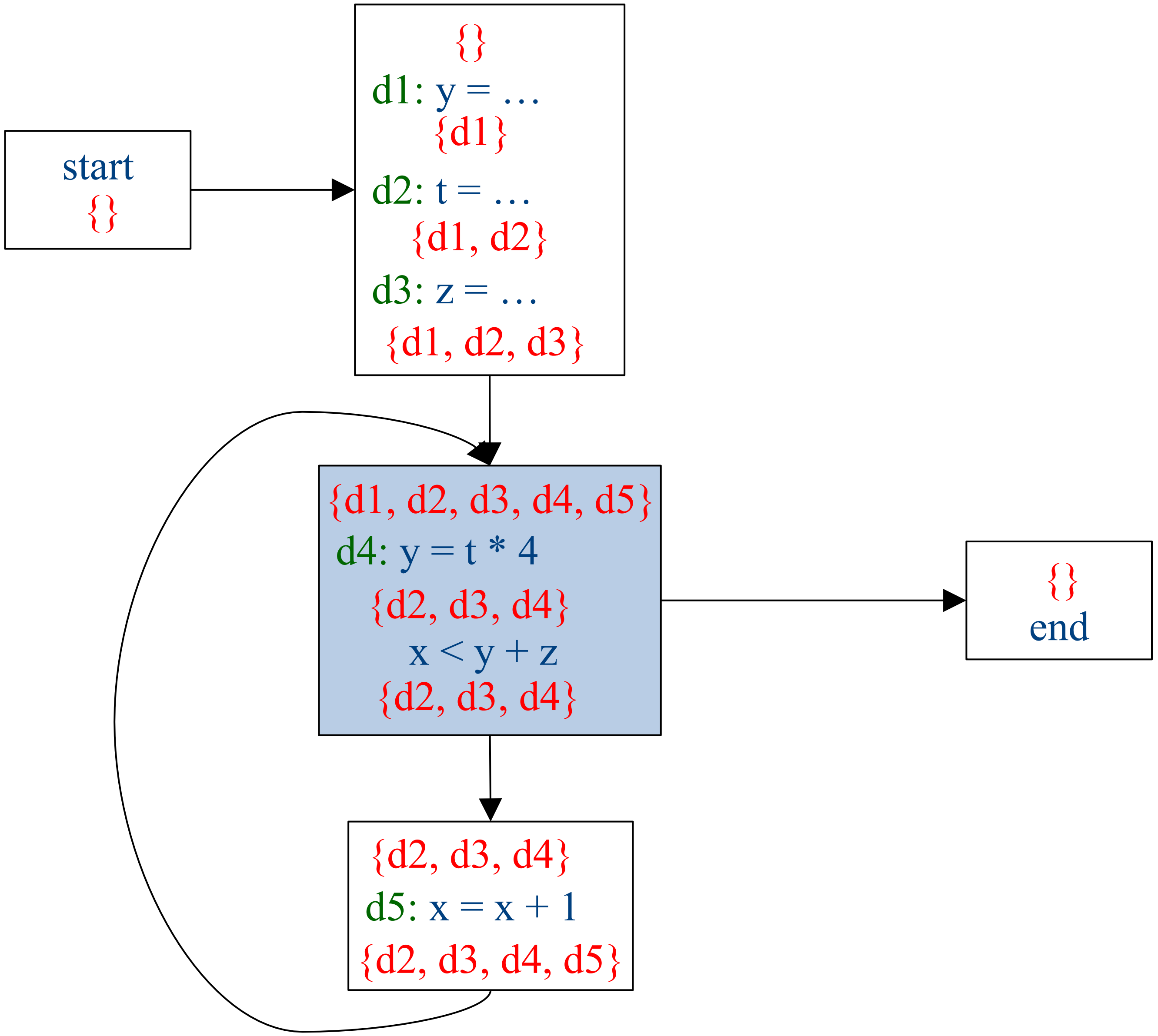
Iteration 3


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}

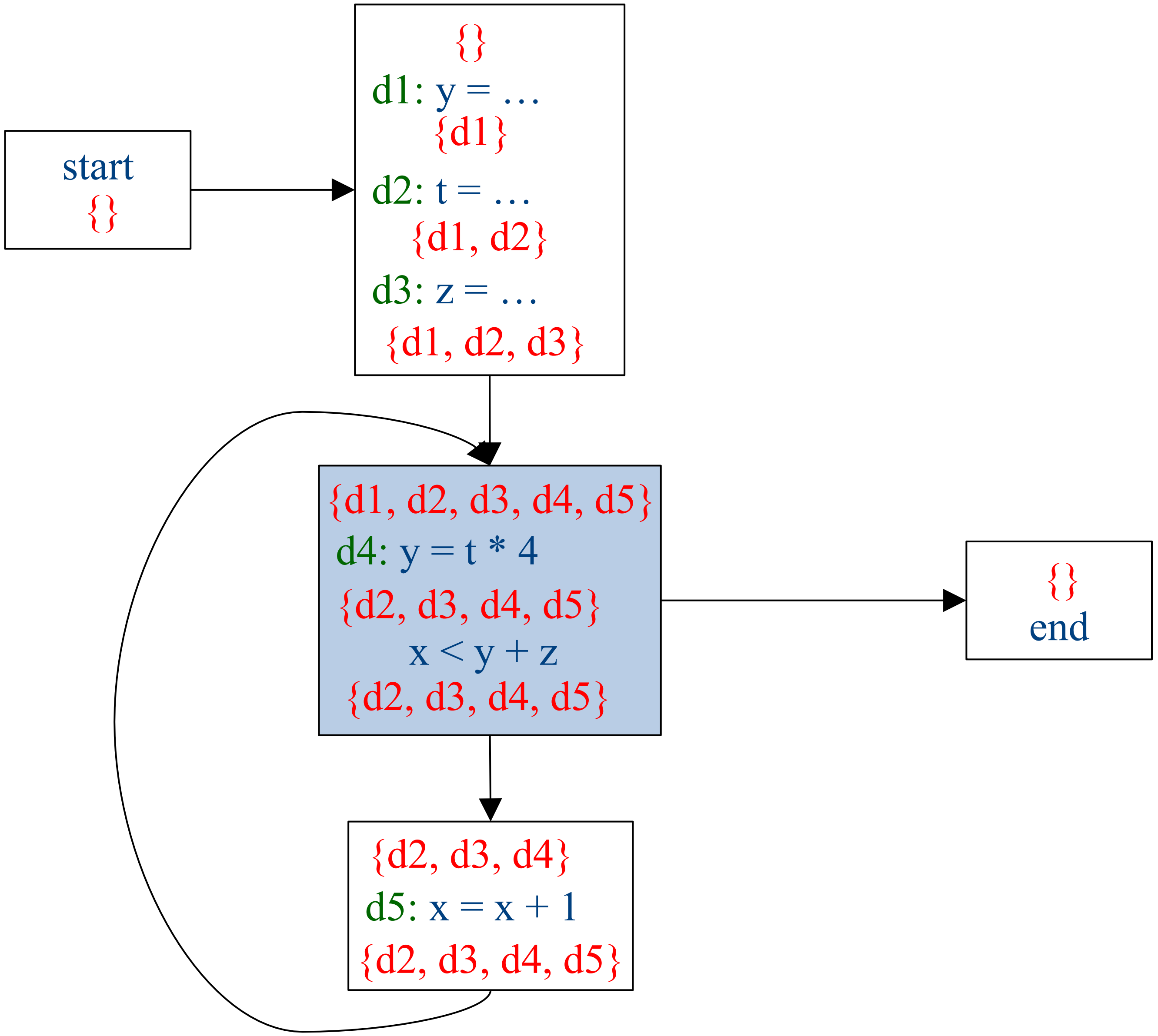
Iteration 4


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}

Iteration 4

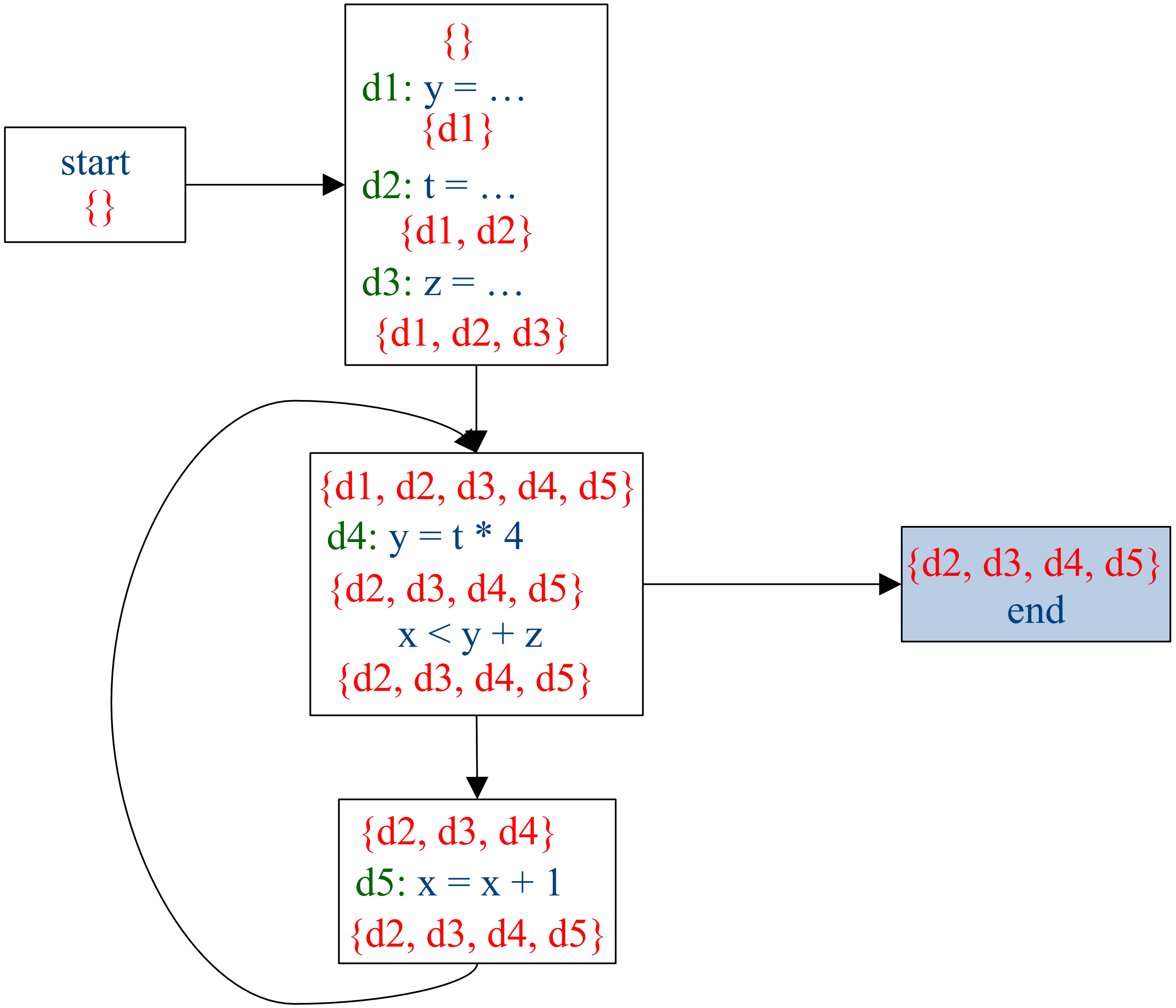

start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3, d4, d5}
d4: y = t * 4
{d2, d3, d4}
x < y + z
{d2, d3, d4}
{}
end
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}

Iteration 4

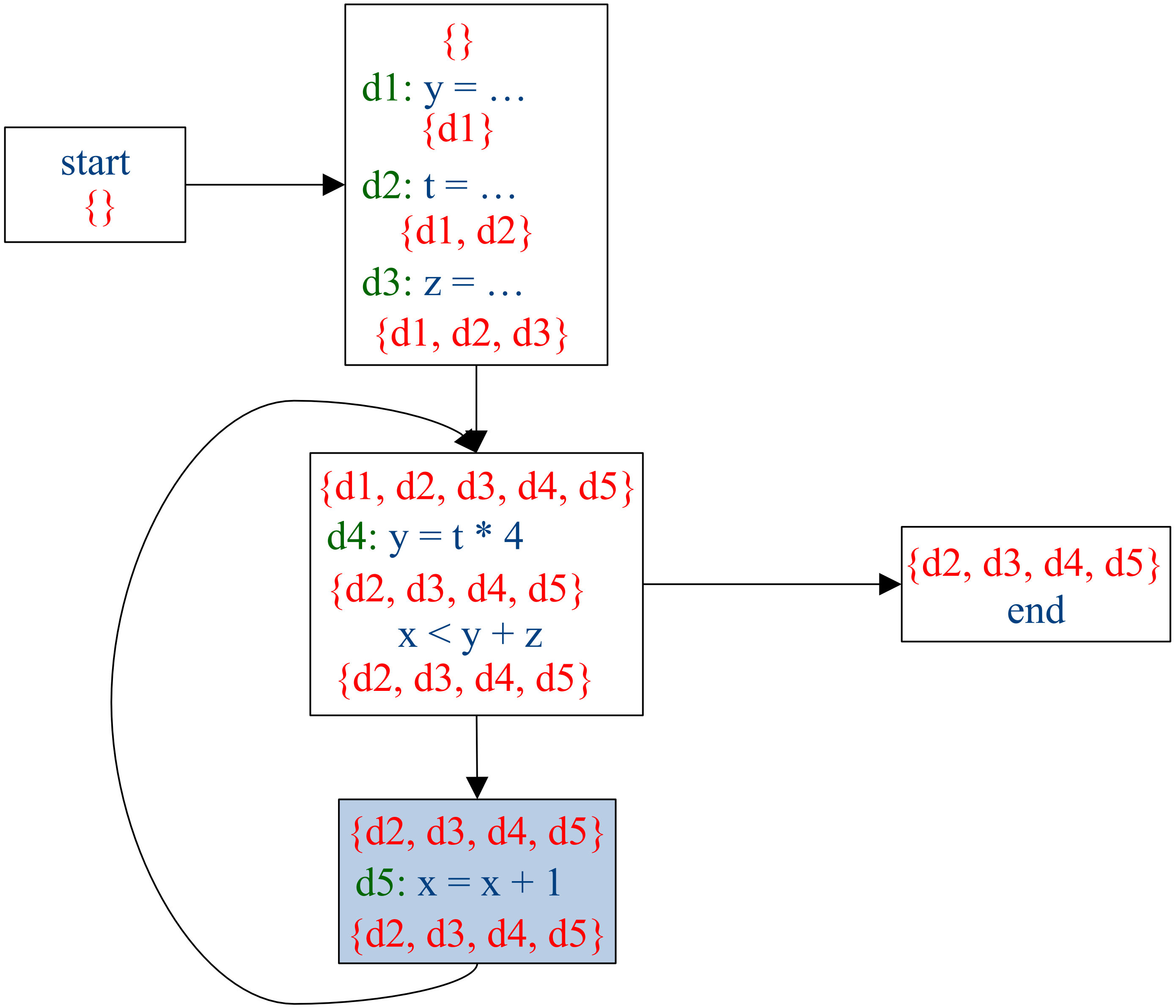


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3, d4, d5}
d4: y = t * 4
{d2, d3, d4, d5}
x < y + z
{d2, d3, d4, d5}
{}
end
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}

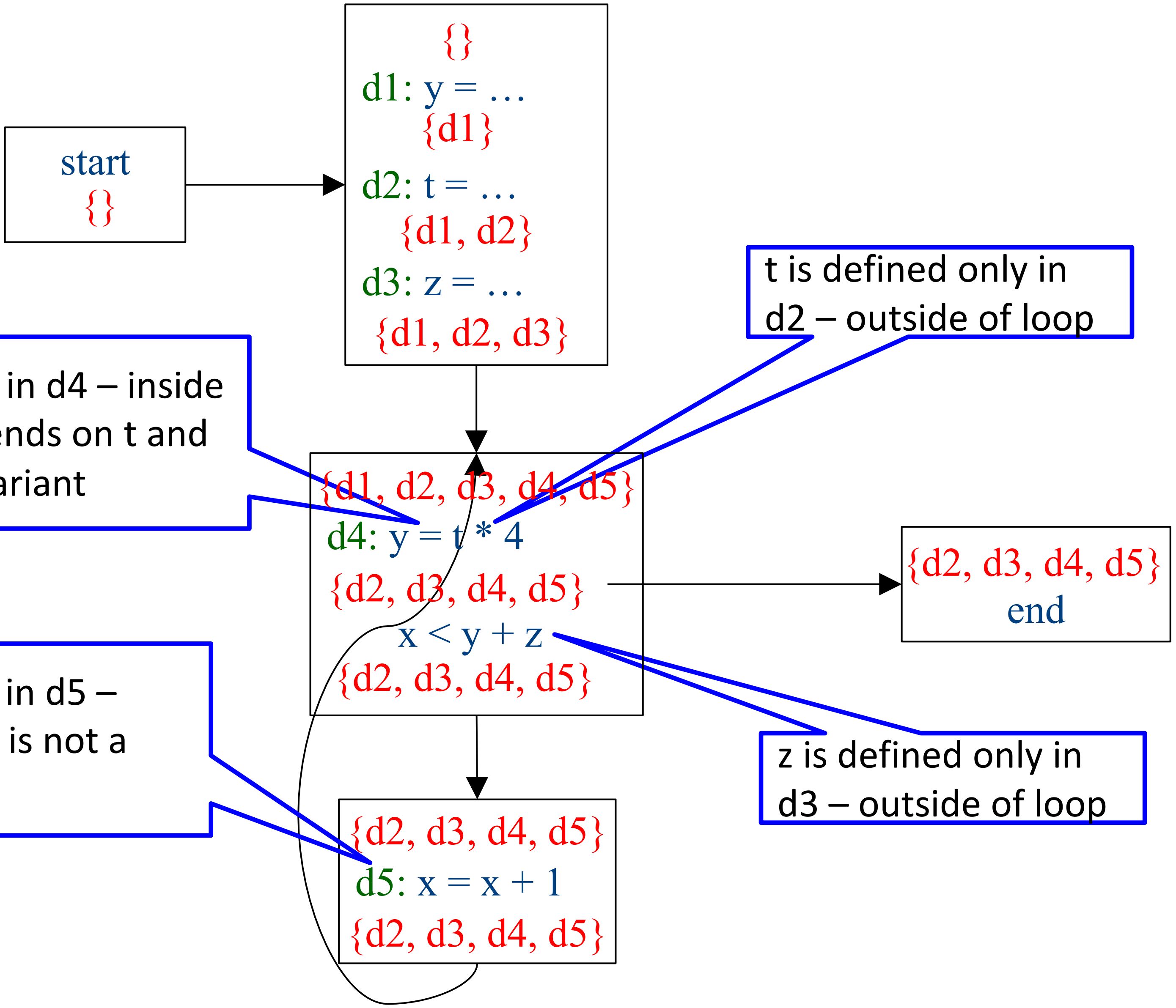
Iteration 5


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3, d4, d5}
d4: y = t * 4
{d2, d3, d4, d5}
x < y + z
{d2, d3, d4, d5}
{d2, d3, d4}
d5: x = x + 1
{d2, d3, d4, d5}
{d2, d3, d4, d5}
end

Iteration 6


start
{}
{}
d1: y = …
{d1}
d2: t = …
{d1, d2}
d3: z = …
{d1, d2, d3}
{d1, d2, d3, d4, d5}
d4: y = t * 4
{d2, d3, d4, d5}
x < y + z
{d2, d3, d4, d5}
{d2, d3, d4, d5}
end
{d2, d3, d4, d5}
d5: x = x + 1
{d2, d3, d4, d5}

Which expressions are loop invariant?



y is defined only in d4 – inside of loop but depends on t and 4, both loop-invariant

t is defined only in d2 – outside of loop

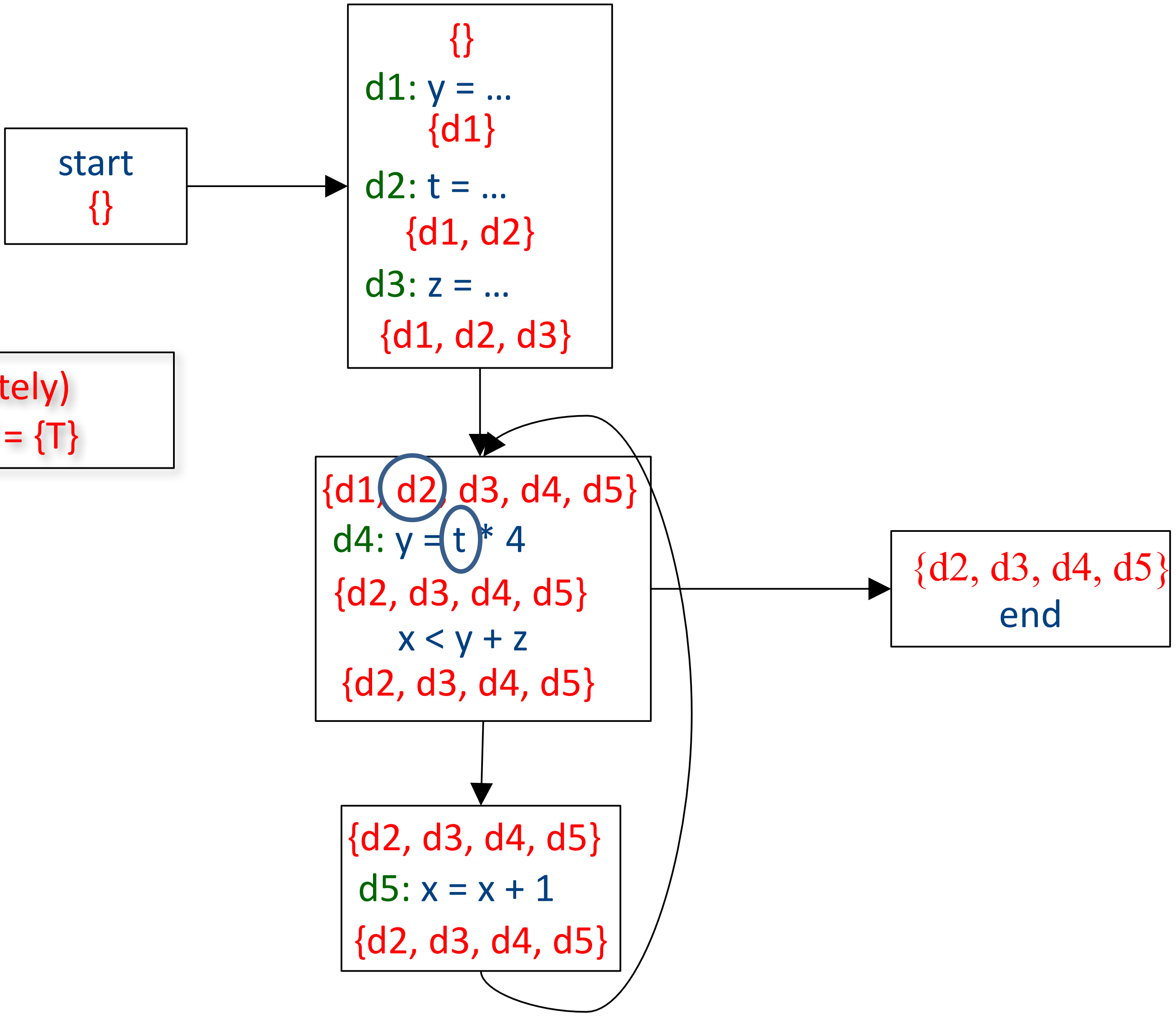
x is defined only in d5 – inside of loop so is not a loop-invariant

z is defined only in d3 – outside of loop

Inferring loop-invariant expressions

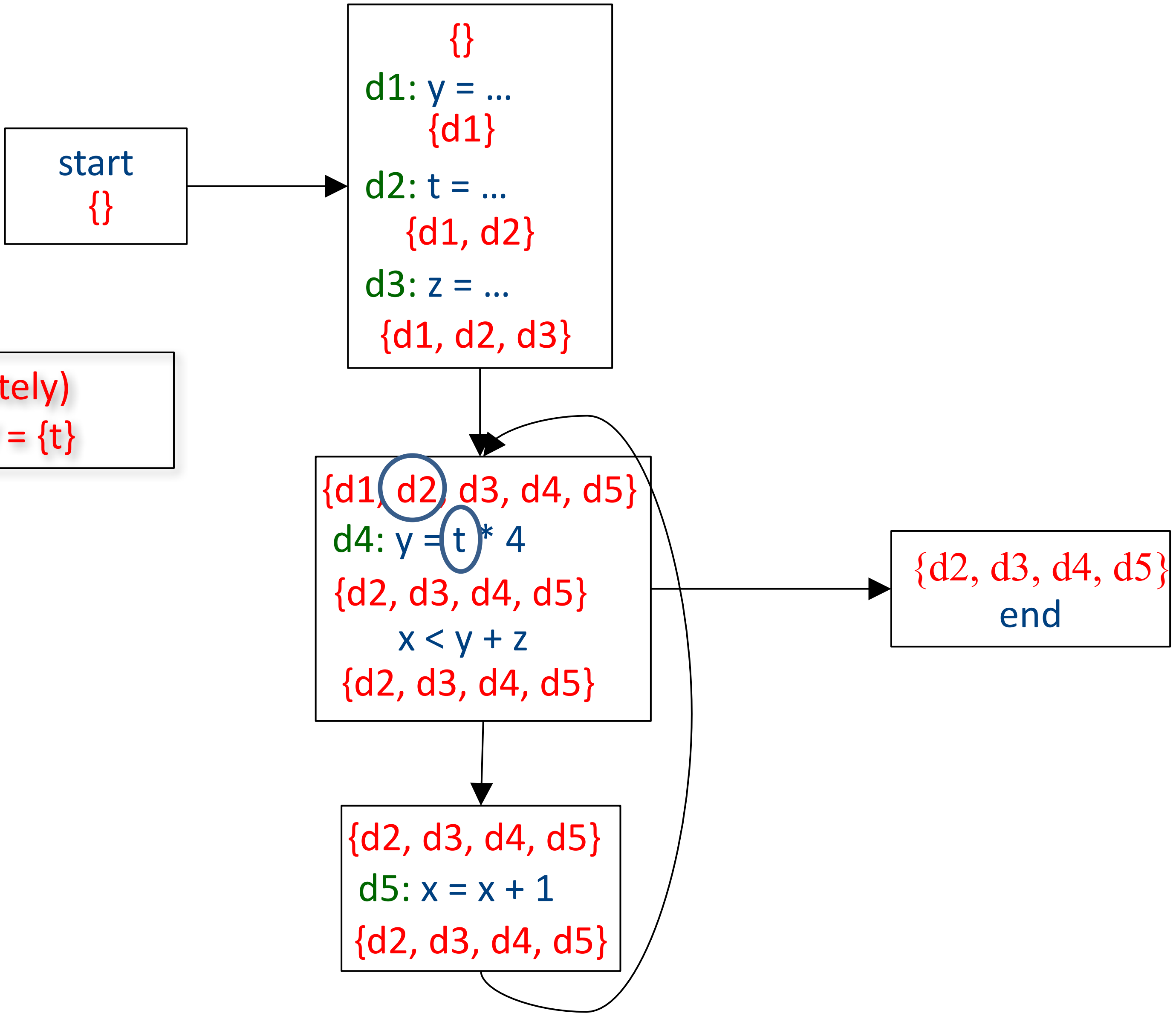
- For a statement *s* of the form t = a_1 op a_2
- A variable a_i is immediately loop-invariant if all reaching definitions IN[*s*]={d_1,…,d_k} for a_i are outside of the loop
- LOOP-INV = immediately loop-invariant variables and constants
  LOOP-INV = LOOP-INV ▶ {x | d: x = a_1 op a_2, d is in the loop, and both a_1 and a_2 are in LOOP-INV}
  - Iterate until fixed-point
- An expression is loop-invariant if all operands are loop-invariants

Computing LOOP-INV



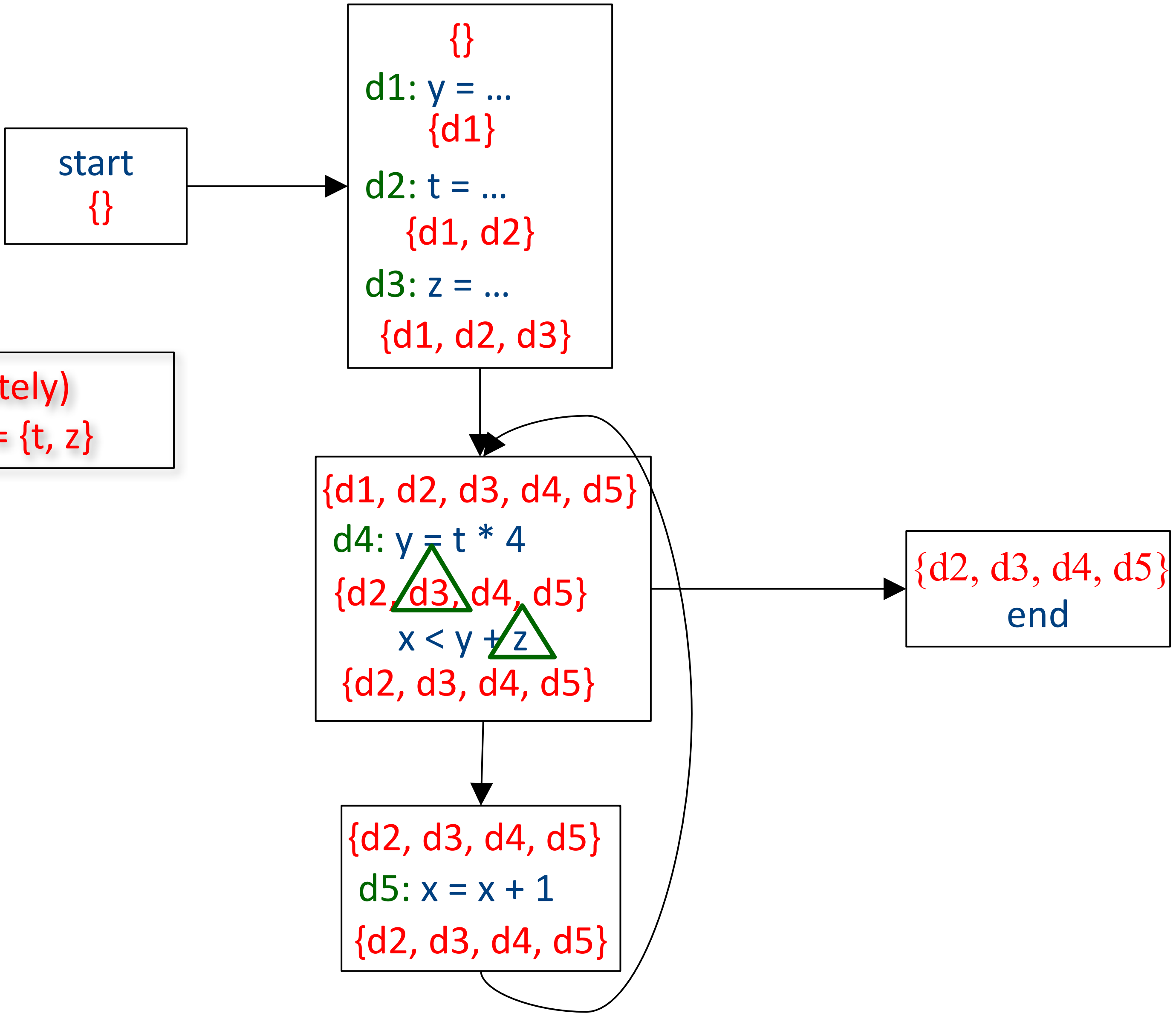
(immediately)
LOOP-INV = {T}

Computing LOOP-INV



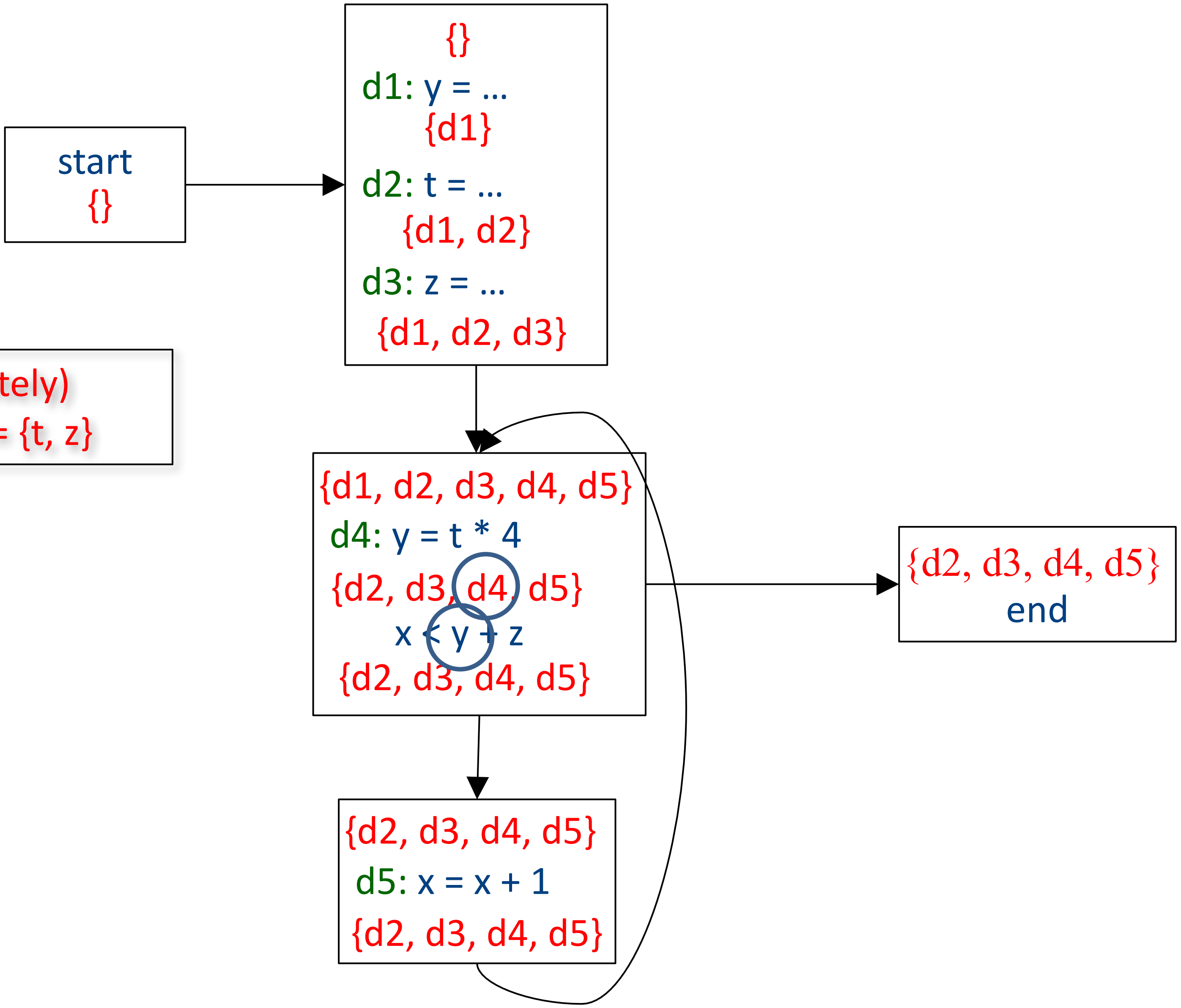
(immediately)
LOOP-INV = {t}

Computing LOOP-INV



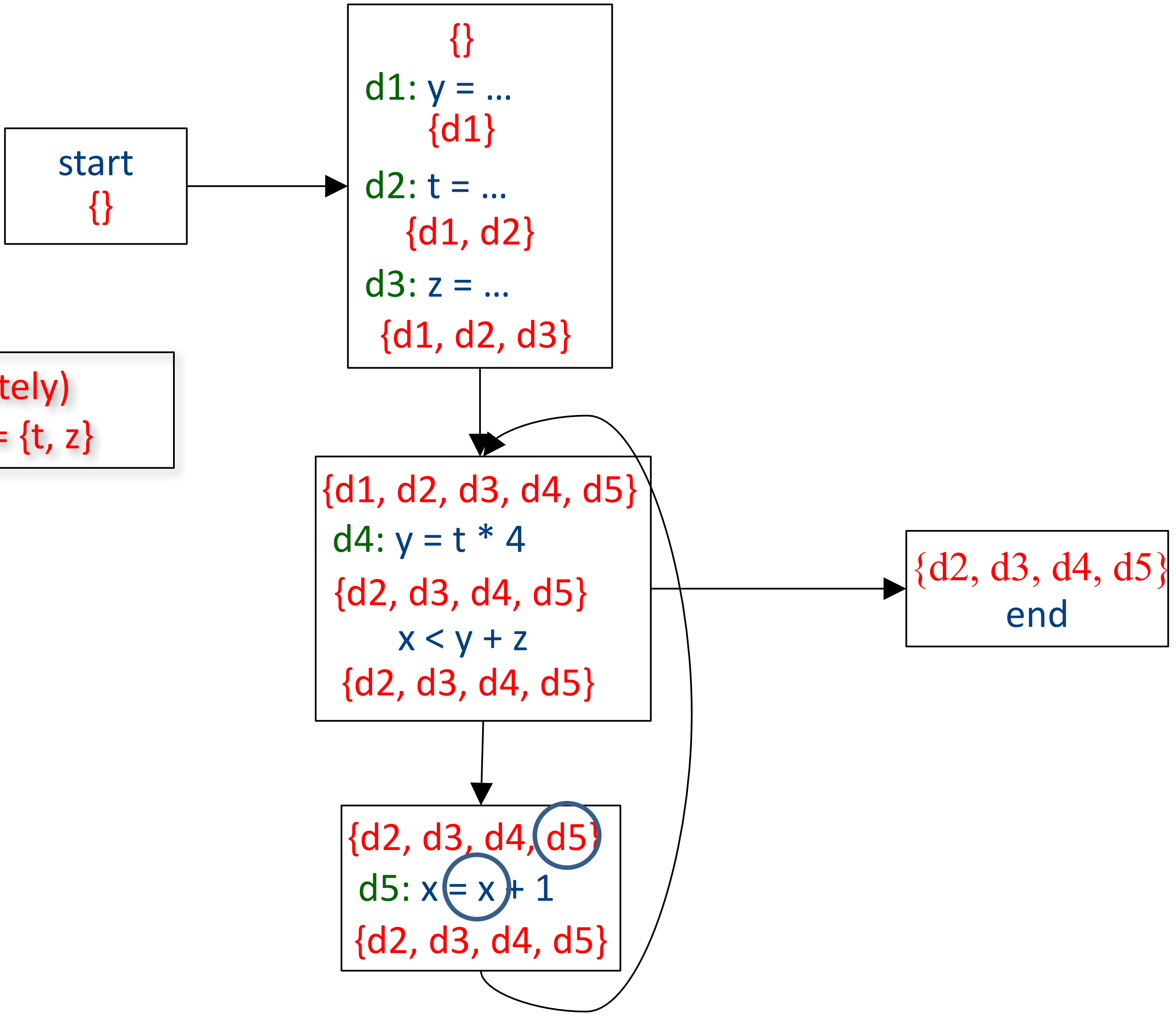
(immediately)
LOOP-INV = {t, z}

Computing LOOP-INV



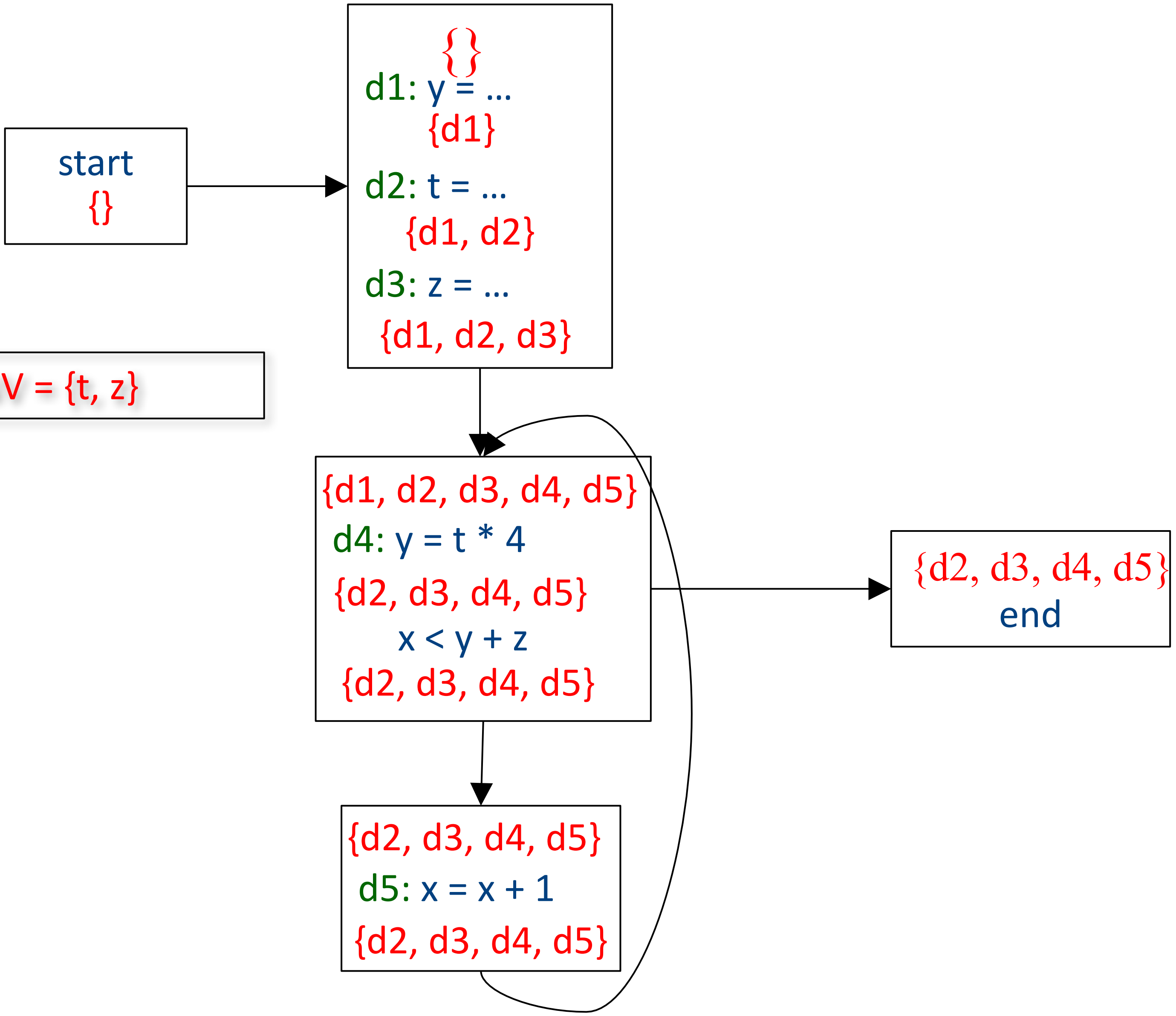
(immediately)
LOOP-INV = {t, z}

Computing LOOP-INV



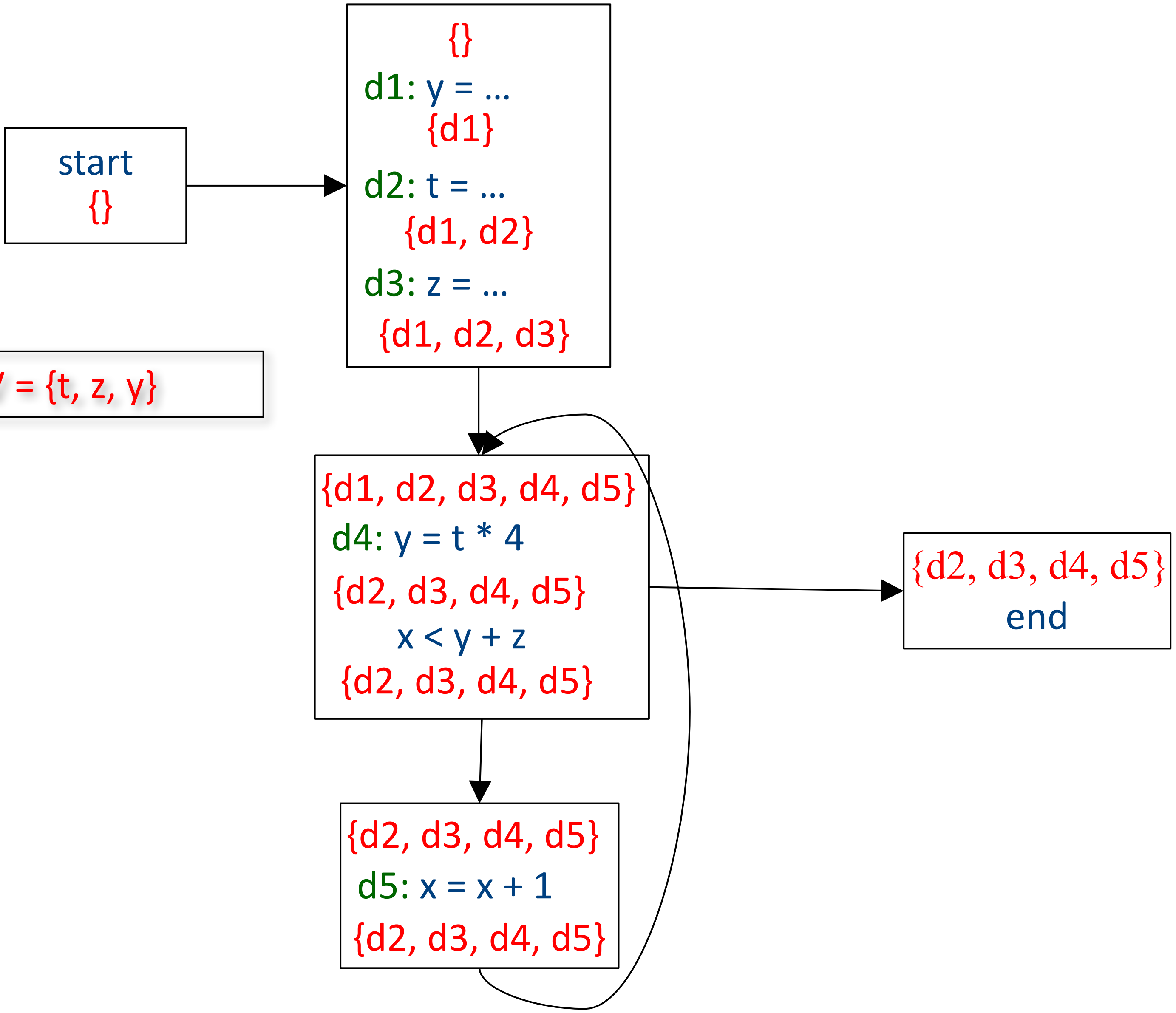
(immediately)
LOOP-INV = {t, z}

Computing LOOP-INV



LOOP-INV = {t, z}

Computing LOOP-INV



LOOP-INV = {t, z, y}

Induction variables

j is a linear function of the induction variable with multiplier 4

```
while (i < x) {
    j = a + 4 * i
    a[j] = j
    i = i + 1
}
```

i is incremented by a loop-invariant expression on each iteration – this is called an induction variable

Strength-reduction

Prepare initial value

```
j = a + 4 * i
while (i < x) {
    j = j + 4
    a[j] = j
    i = i + 1
}
```

Increment by multiplier

Compilation

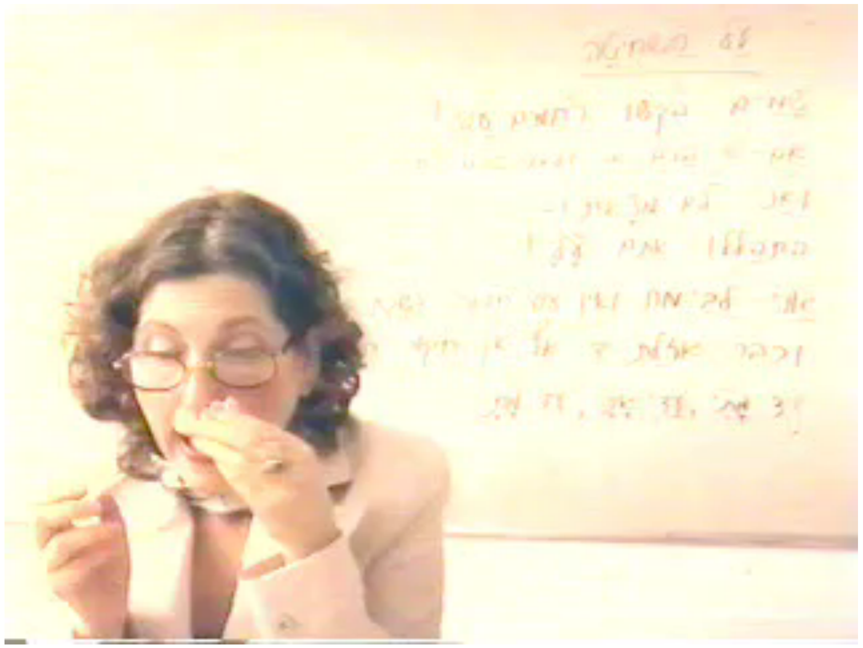
0368-3133

Lecture 10b

Register Allocation

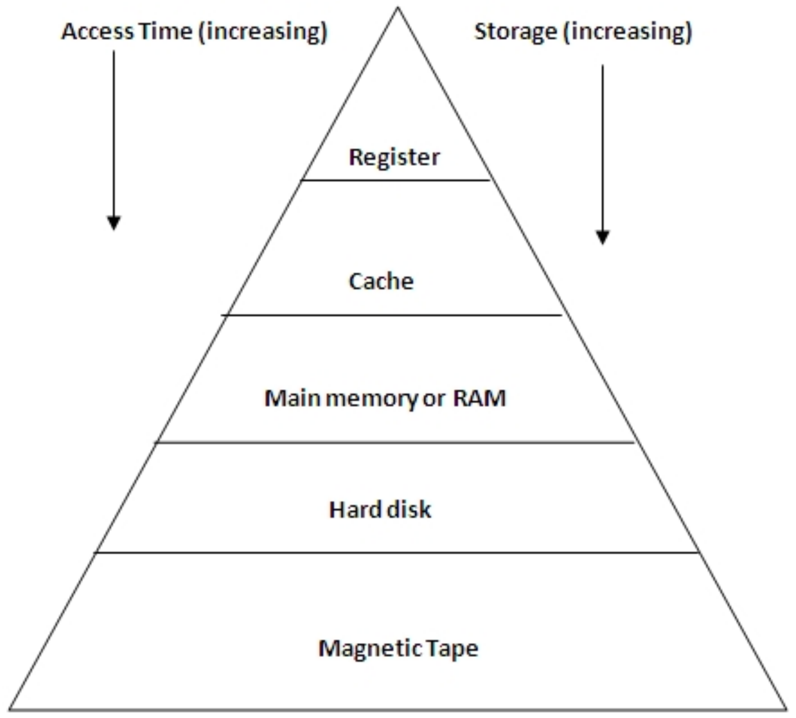
Noam Rinetzky

What is a Compiler?



Registers

- **Dedicated memory** locations that
  - can be accessed quickly,
  - can have computations performed on them, and



Registers

- **Dedicated memory** locations that
  - can be accessed quickly,
  - can have computations performed on them, and

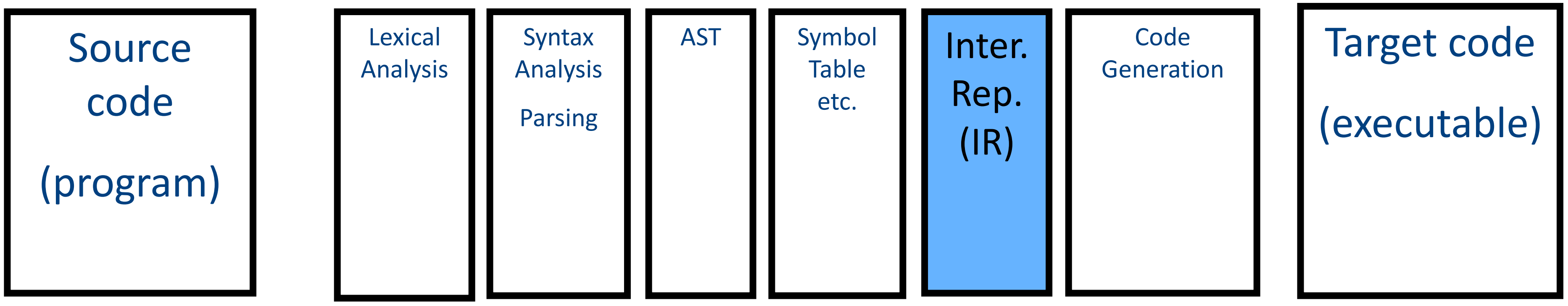
- Usages
  - Operands of instructions
  - Store temporary results
  - Can (should) be used as loop indexes due to frequent arithmetic operation
  - Used to manage administrative info
    - e.g., runtime stack

Register allocation

- Number of registers is **limited**

- Need to **allocate** them in a clever way
  - Using registers intelligently is a critical step in any compiler
    - A good register allocator can generate code orders of magnitude better than a bad register allocator

Register Allocation: IR



Simple approach

- Straightforward solution:
  - Allocate each variable in activation record
  - At each instruction, bring values needed into registers, perform operation, then store result to memory

```
x = y + z
```

```
mov 16(%ebp), %eax
mov 20(%ebp), %ebx
add %ebx, %eax
mov %eax, 24(%ebp)
```

- Problem: program execution very inefficient– moving data back and forth between memory and registers

Simple code generation

- assume machine instructions of the form
- `LD reg, mem`
- `ST mem, reg`
- `OP reg,reg,reg` (*)

- assume that we have all registers available for our use
  - Ignore registers allocated for stack management
  - Treat all registers as general-purpose

Simple code generation

- assume machine instructions of the form
- `LD reg, mem`
- `ST mem, reg`
- `OP reg,reg,reg` (*)

Fixed number of Registers!

Register allocation

- In **TAC**, there is an unlimited number of variables (temporaries)
- On a physical machine there is a small number of registers:
  - **x86** has **4** general-purpose registers and a number of specialized registers
  - **MIPS** has **24** general-purpose registers and **8** special-purpose registers

- Register allocation is the process of assigning variables to registers and managing data transfer in and out of registers

simple code generation

- assume machine instructions of the form
- `LD reg, mem`
- `ST mem, reg`
- `OP reg,reg,reg` (*)

Fixed number of Registers!

- We will assume that we have all registers available for any usage
  - Ignore registers allocated for stack management
  - Treat all registers as general-purpose

Plan

- Goal: Reduce number of temporaries (registers)
  - Machine-agnostic optimizations
    - Assume unbounded number of registers
  - Machine-dependent optimization
    - Use at most K registers
    - K is machine dependent

Sethi-Ullman translation

- Algorithm by Ravi Sethi and Jeffrey D. Ullman to emit optimal TAC
  - Minimizes number of temporaries for a **single expression**

Generating Compound Expressions

- Use registers to store temporaries
  - Why can we do it?

- Maintain a counter for temporaries in c
- Initially: c = 0
- **cgen**(e_1 *op* e_2) = {

```
    Let A = cgen(e_1)
    c = c + 1
    Let B = cgen(e_2)
    c = c + 1
    Emit( _tc = A op B; ) // _tc is a register
    Return _tc
}
```

Improving **cgen** for expressions

- Observation – naïve translation needlessly generates temporaries for leaf expressions
- Observation – temporaries used exactly once
  - Once a temporary has been read it can be reused for another sub-expression
- **cgen**(e_1 *op* e_2) = {

```
  Let _t1 = cgen(e_1)
  Let _t2 = cgen(e_2)
  Emit( _t1 =_t1 op _t2; )
  Return _t1
}
```

- Temporaries **cgen**(e_1) can be reused in **cgen**(e_2)

Register Allocation

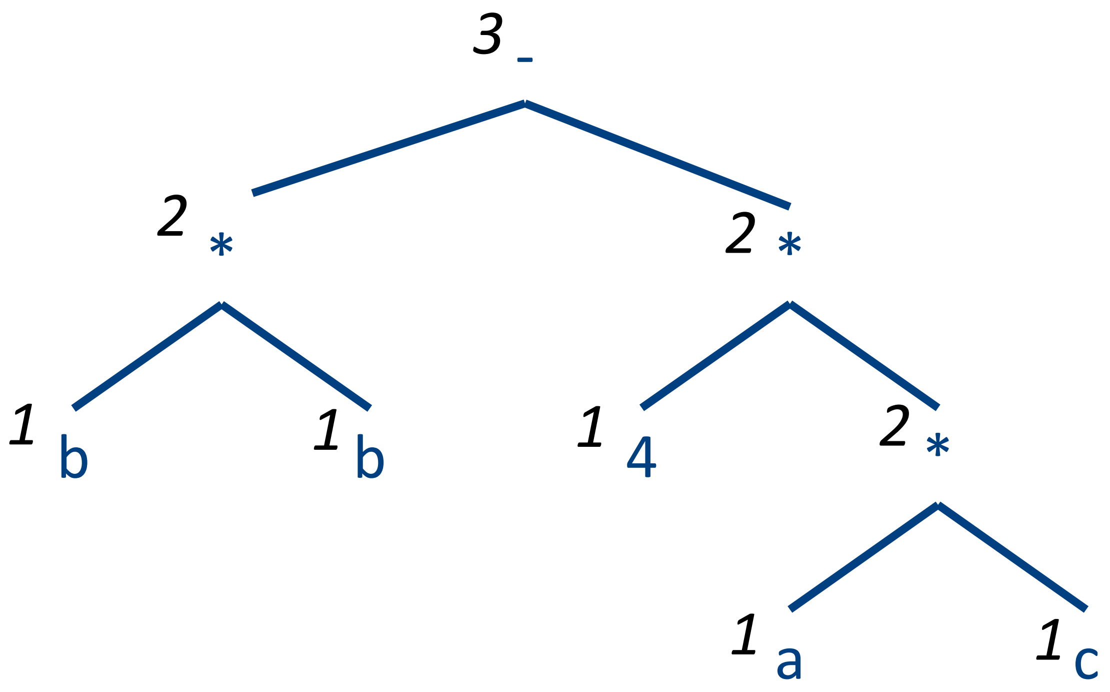
- Machine-agnostic optimizations
    - Assume unbounded number of registers
  - Expression trees
  - Basic blocks

- Machine-dependent optimization
    - K registers
    - Some have special purposes
  - Control flow graphs (whole program)

Sethi-Ullman translation

- Algorithm by Ravi Sethi and Jeffrey D. Ullman to emit optimal TAC
  - Minimizes number of temporaries for a **single expression**

Example (optimized): b*b-4*a*c



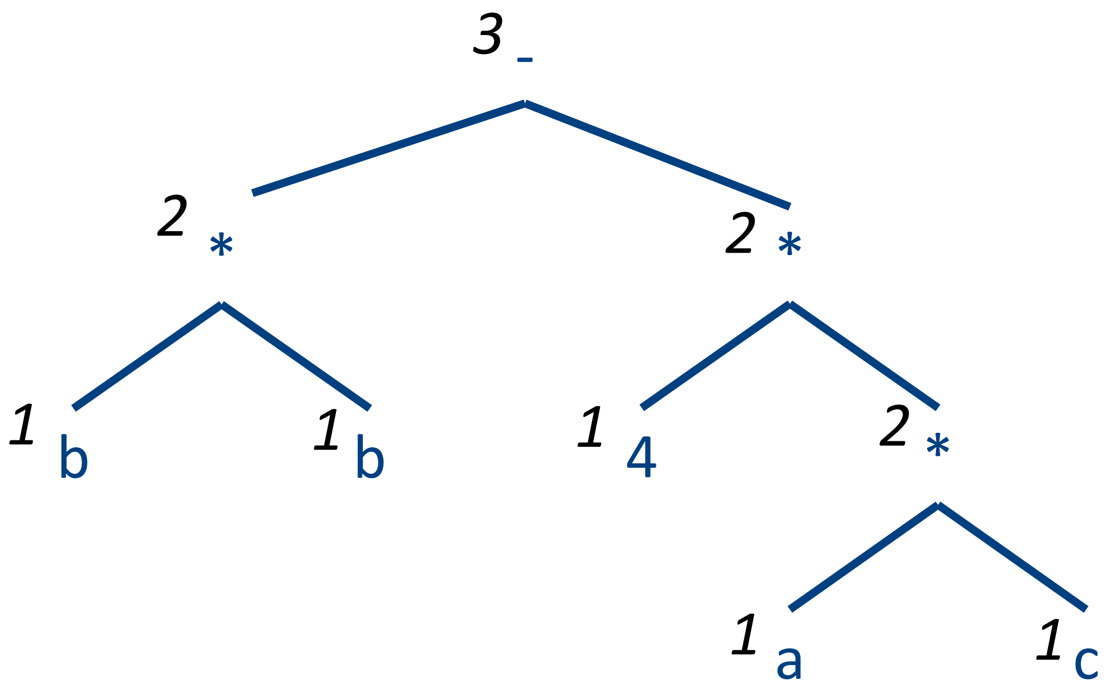
Generalizations

- More than two arguments for operators
  - Function calls
- Multiple effected registers
  - Multiplication
- Spilling
  - Need more registers than available
- Register/memory operations

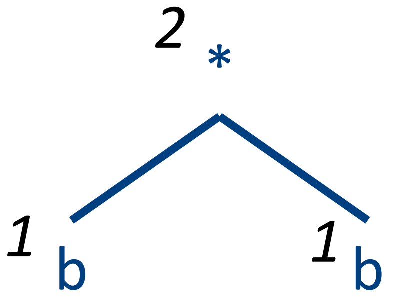
Simple **Spilling** Method

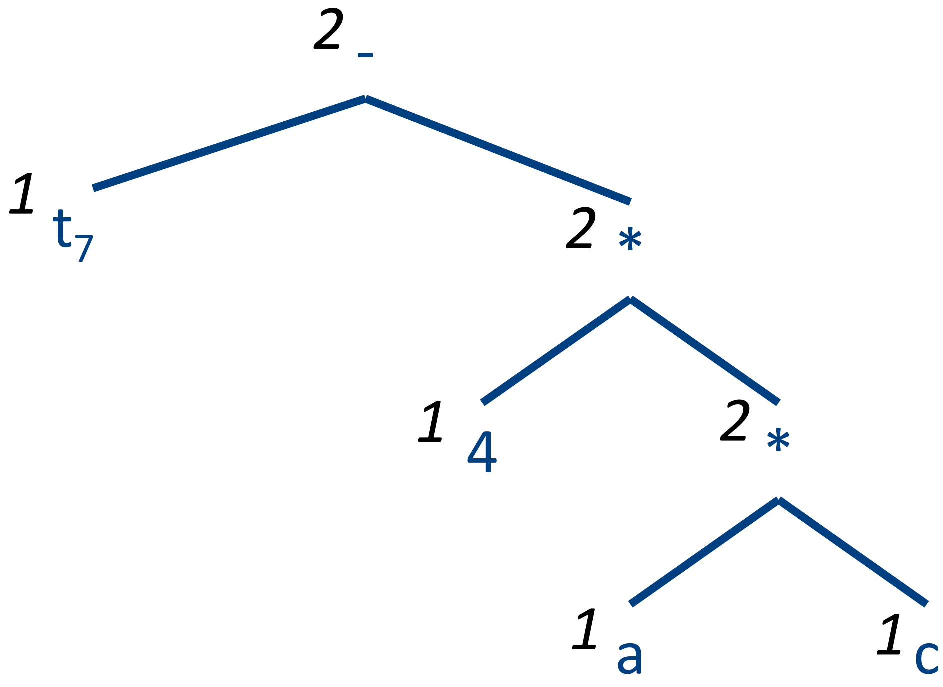
- Heavy tree – Needs more registers than available
- A “heavy” tree contains a “heavy” subtree whose dependents are “light”
- Simple spilling
  - Generate code for the light tree
  - Spill the content into memory and replace subtree by temporary
  - Generate code for the resultant tree

Example (optimized): x:=b*b-4*a*c



Example (spilled): x := b*b-4*a*c

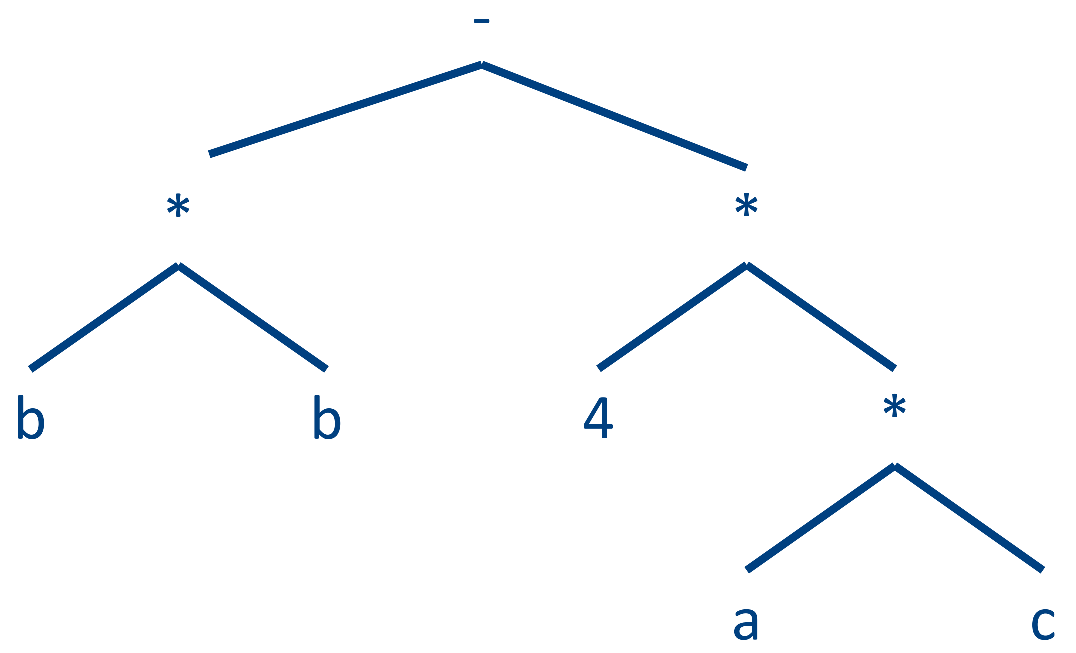




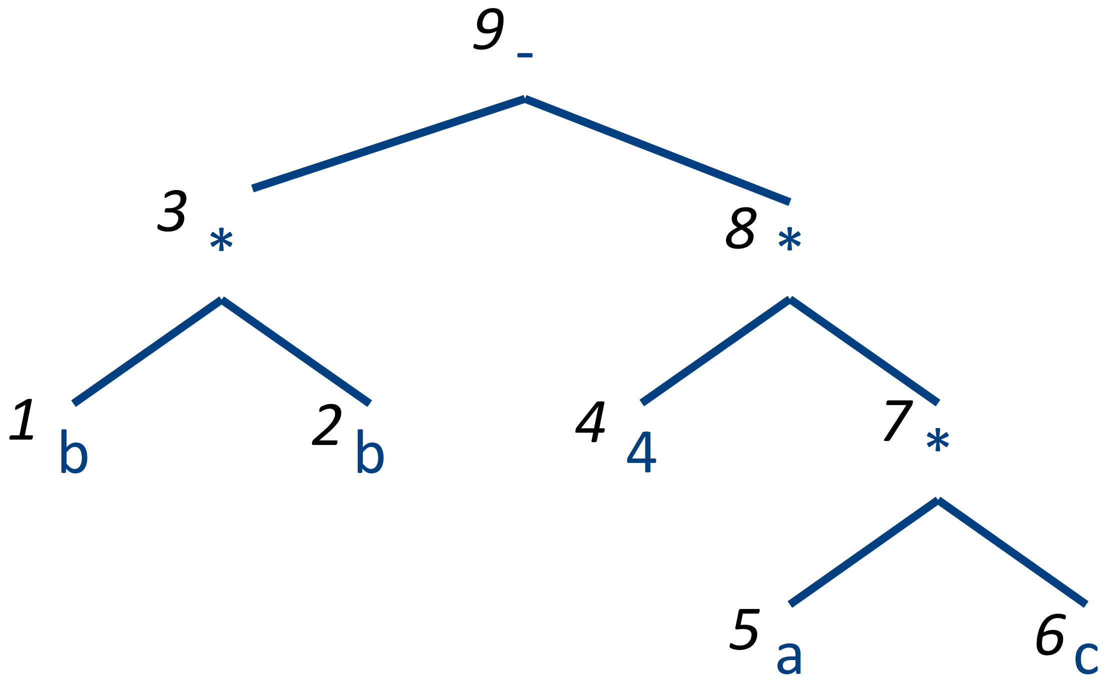
```
t7 := b * b
```

```
x := t7 - 4 * a * c
```

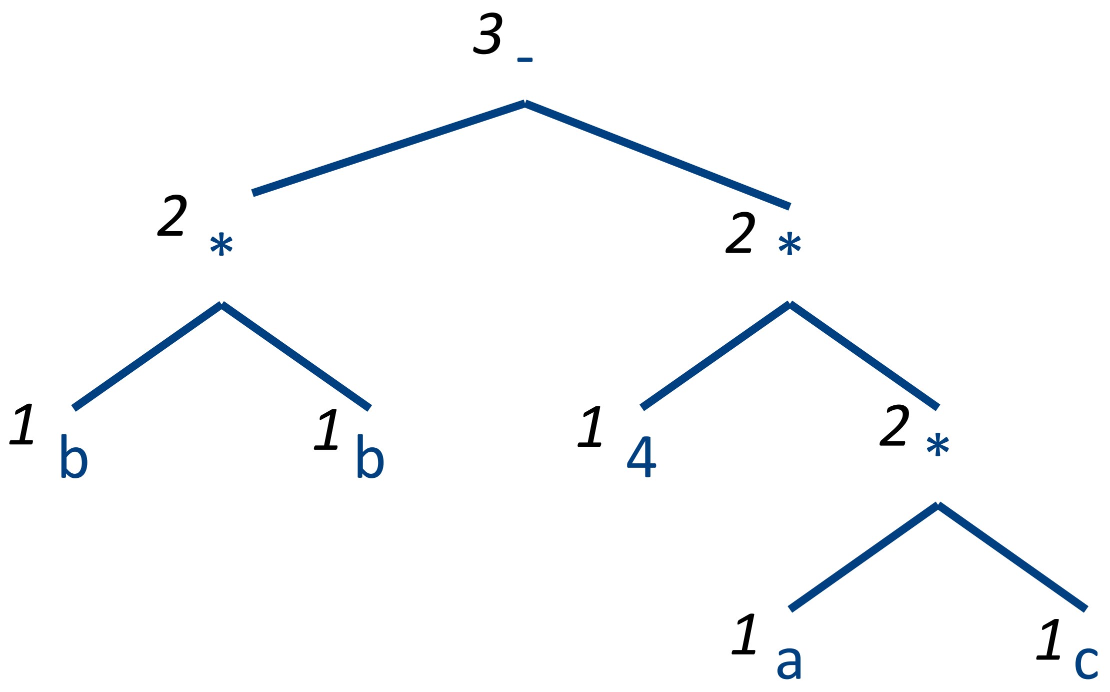
Example: b*b-4*a*c



Example (simple): b*b-4*a*c



Example (optimized): b*b-4*a*c



Spilling

- Even an optimal register allocator can require more registers than available
- Need to generate code for every correct program
- The compiler can save temporary results
  - Spill registers into temporaries
  - Load when needed
- Many heuristics exist

Simple Spilling Method

- Heavy tree – Needs more registers than available
- A `heavy’ tree contains a `heavy’ subtree whose dependents are ‘light’
- Generate code for the light tree
- Spill the content into memory and replace subtree by temporary
- Generate code for the resultant tree

Spilling

- Even an optimal register allocator can require more registers than available
- Need to generate code for every correct program
- The compiler can save temporary results
  - Spill registers into temporaries
  - Load when needed
- Many heuristics exist

Simple approach

- Straightforward solution:
  - Allocate each variable in activation record
  - At each instruction, bring values needed into registers, perform operation, then store result to memory

```
x = y + z
```

$\Rightarrow$

```
mov 16(%ebp), %eax
mov 20(%ebp), %ebx
add %ebx, %eax
mov %eax, 24(%ebx)
```

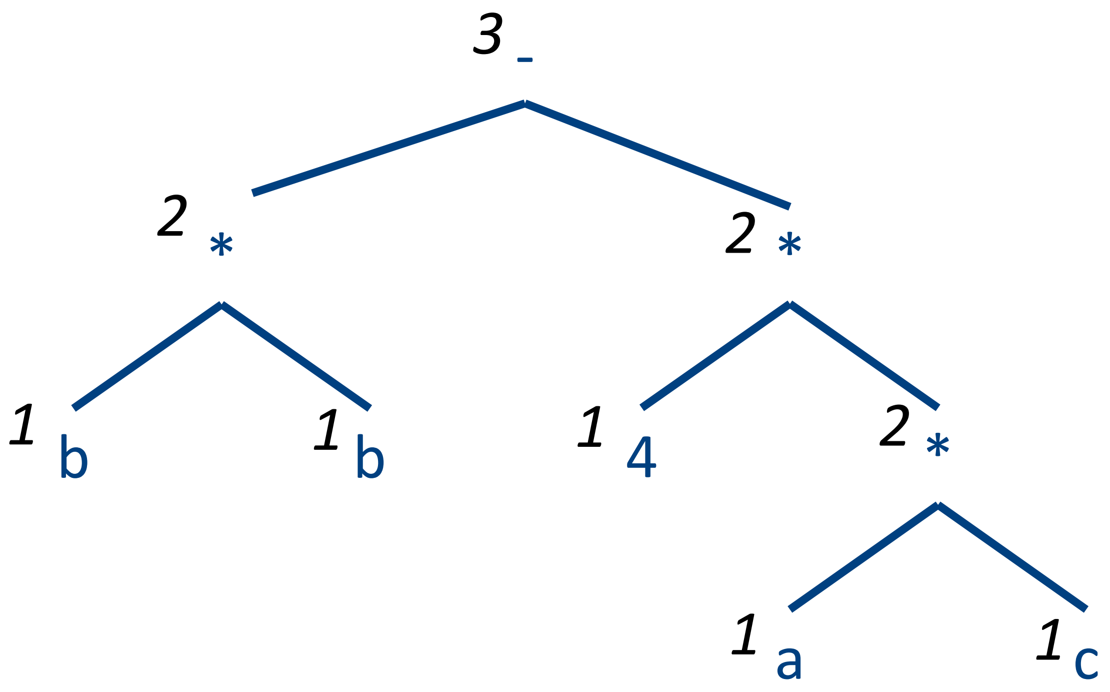
- Problem: program execution very inefficient– moving data back and forth between memory and registers

Register Allocation

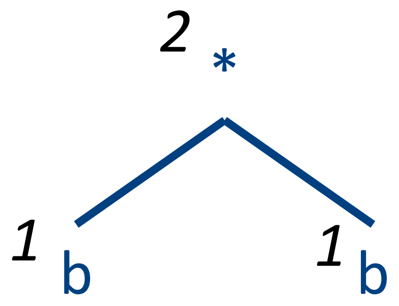
- Machine-agnostic optimizations
    - Assume unbounded number of registers
  - Expression trees (tree-local)
  - Basic blocks (block-local)

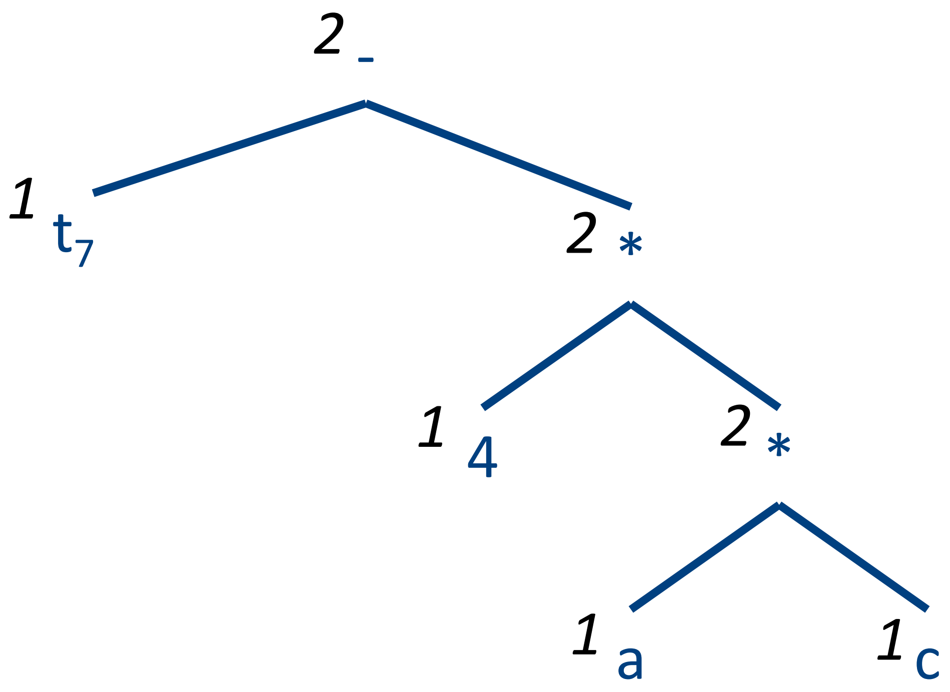
- Machine-dependent optimization
    - K registers
    - Some have special purposes
  - Control flow graphs (global register allocation)

Example (optimized): b*b-4*a*c



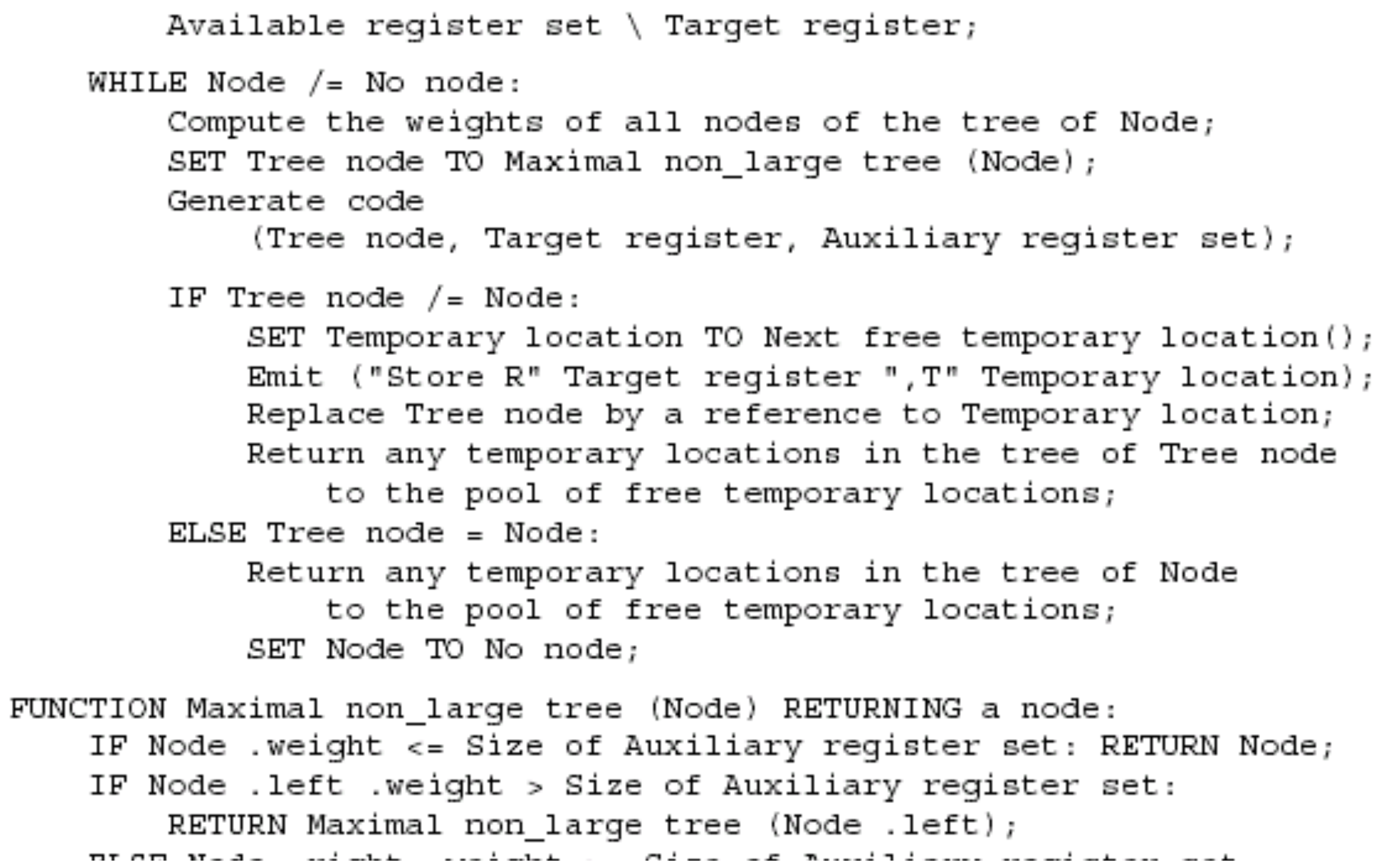
Example (spilled): x := b*b-4*a*c





```
t7 := b * b                x := t7 - 4 * a * c
```

Simple Spilling Method



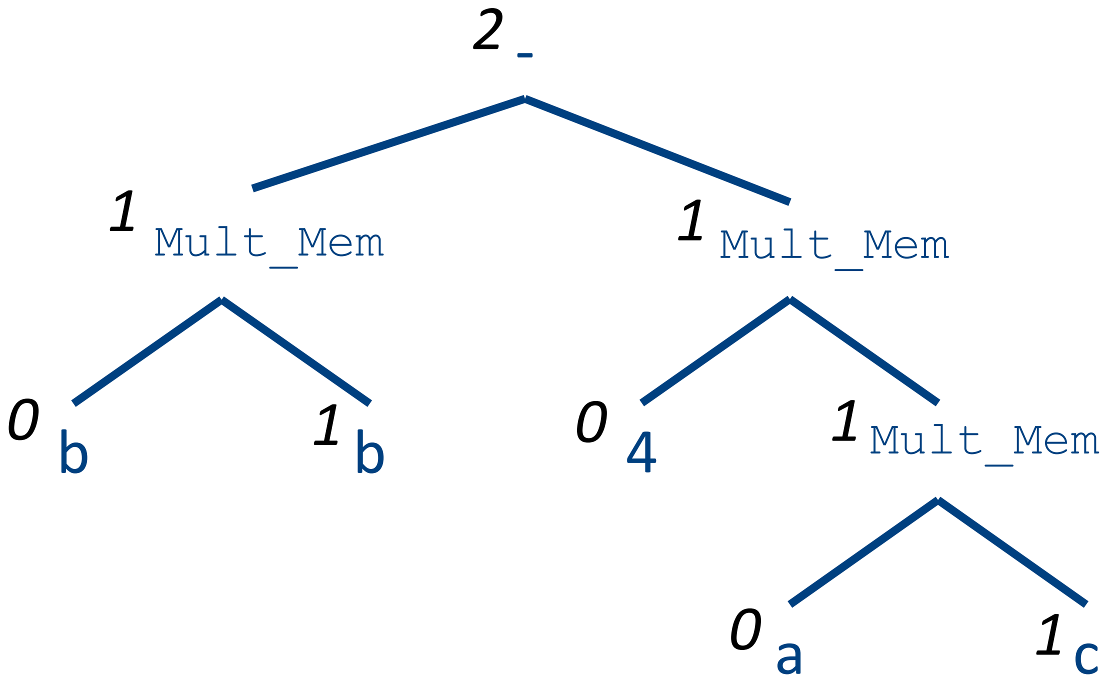
```
        Available register set \ Target register;

    WHILE Node /= No node:
        Compute the weights of all nodes of the tree of Node;
        SET Tree node TO Maximal non_large tree (Node);
        Generate code
            (Tree node, Target register, Auxiliary register set);

        IF Tree node /= Node:
            SET Temporary location TO Next free temporary location();
            Emit ("Store R" Target register ",T" Temporary location);
            Replace Tree node by a reference to Temporary location;
            Return any temporary locations in the tree of Tree node
                to the pool of free temporary locations;
        ELSE Tree node = Node:
            Return any temporary locations in the tree of Node
                to the pool of free temporary locations;
            SET Node TO No node;

FUNCTION Maximal non_large tree (Node) RETURNING a node:
    IF Node .weight <= Size of Auxiliary register set: RETURN Node;
    IF Node .left .weight > Size of Auxiliary register set:
        RETURN Maximal non_large tree (Node .left);
    ELSE Node .right .weight >= Size of Auxiliary register set
```

Register Memory Operations

- Add_Mem X, R1
- Mult_Mem X, R1
- No need for registers to store right operands

Hidden Registers

Example: b*b-4*a*c



Can We do Better?

- Yes: Increase view of code
  - Simultaneously allocate registers for multiple expressions

- But: Lose per expression optimality
  - Works well in practice

Register Allocation

- Machine-agnostic optimizations
    - Assume unbounded number of registers
  - – Expression trees
  - – Basic blocks

- Machine-dependent optimization
    - K registers
    - Some have special purposes
  - – Control flow graphs (whole program)

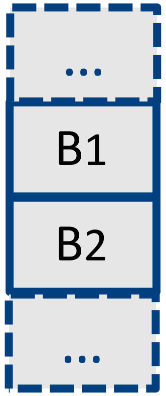
Basic Blocks

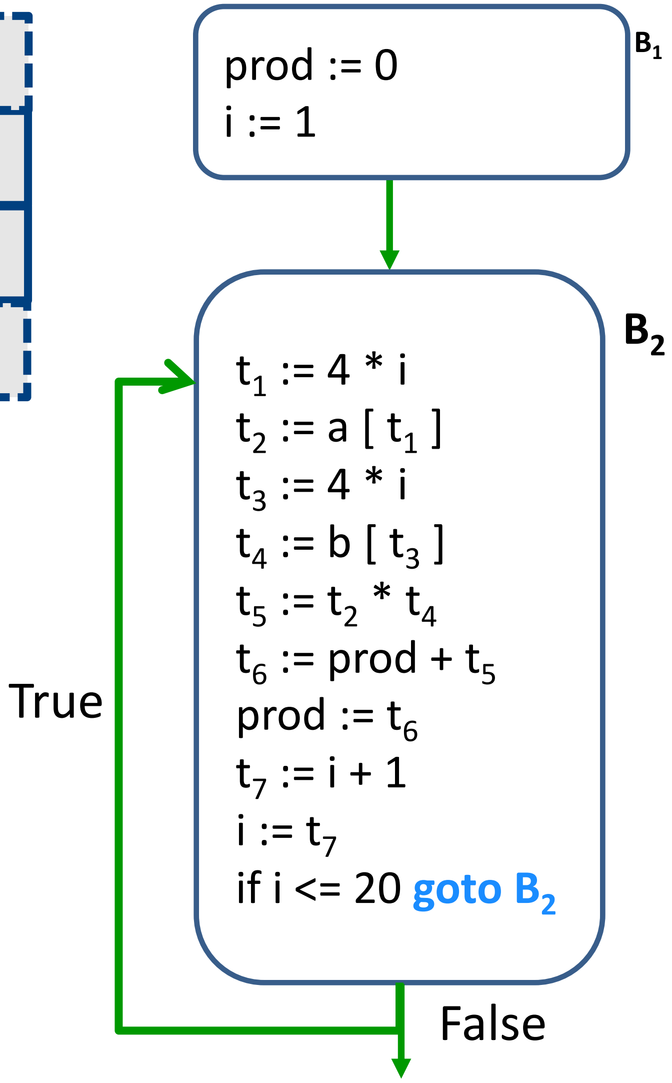
- **basic block** is a sequence of instructions with
  - **single entry** (to first instruction), no jumps to the middle of the block
  - **single exit** (last instruction)
  - code execute as a sequence from first instruction to last instruction without any jumps
- edge from one basic block B1 to another block B2 when the last statement of B1 may jump to B2

control flow graph

- A directed graph G=(V,E)
- nodes V = basic blocks
- edges E = control flow
  - (B1,B2) $\in$ E when control from B1 flows to B2

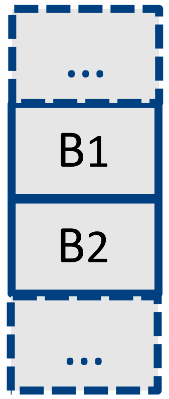
- Leaders-based construction
  - Target of jump instructions
  - Instructions following jumps

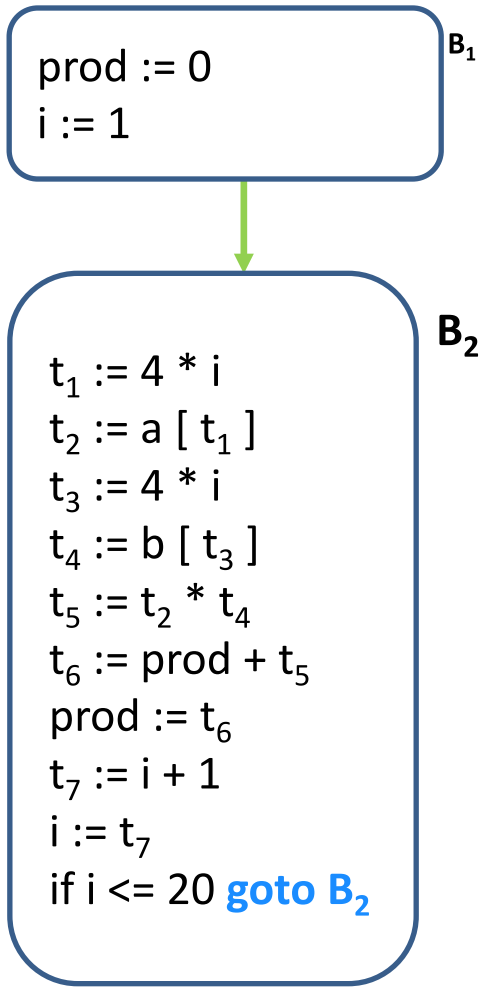




control flow graph

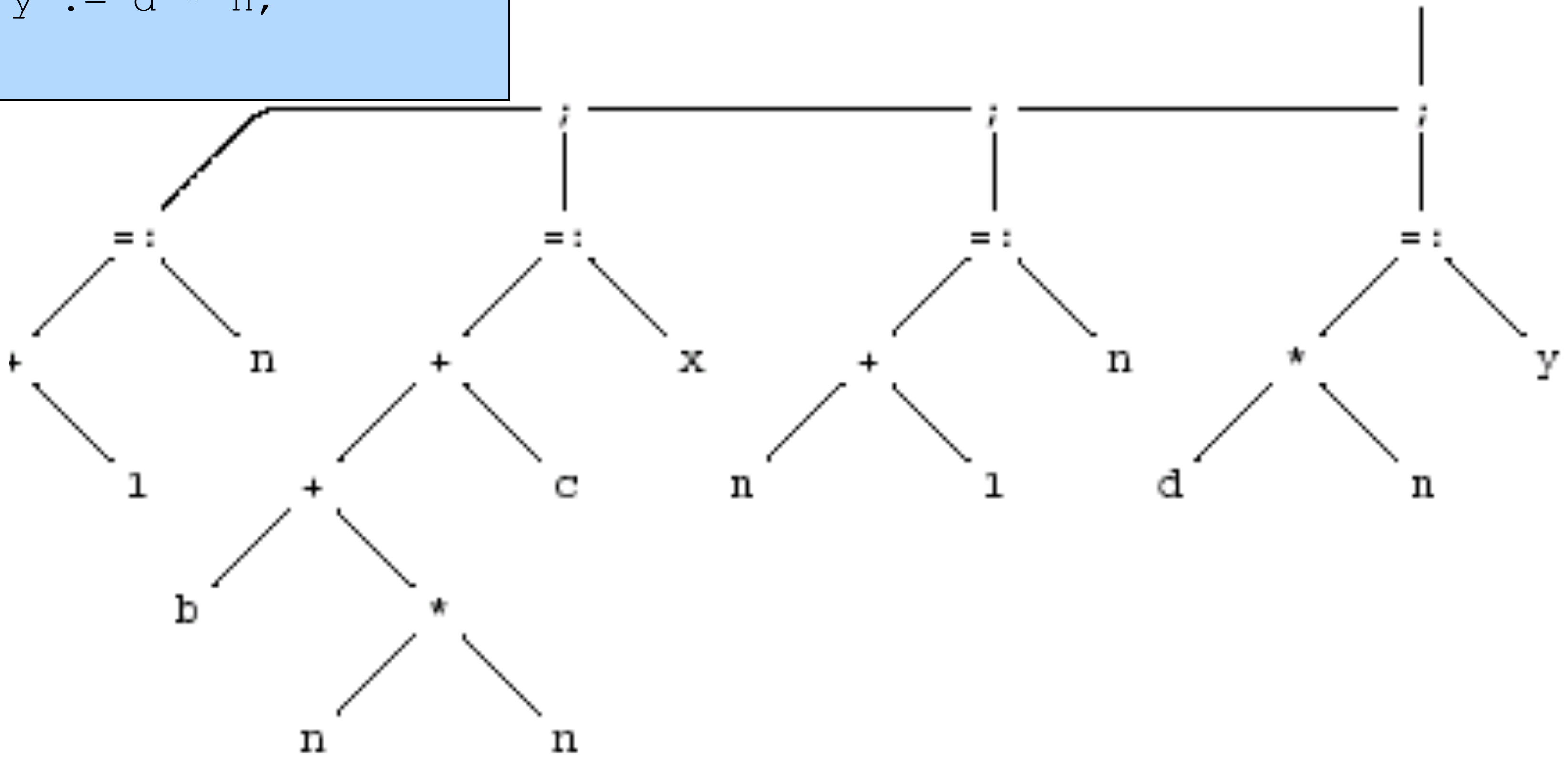
- A directed graph G=(V,E)
- nodes V = basic blocks
- edges E = control flow
  - (B1,B2) $\in$ E when control from B1 flows to B2





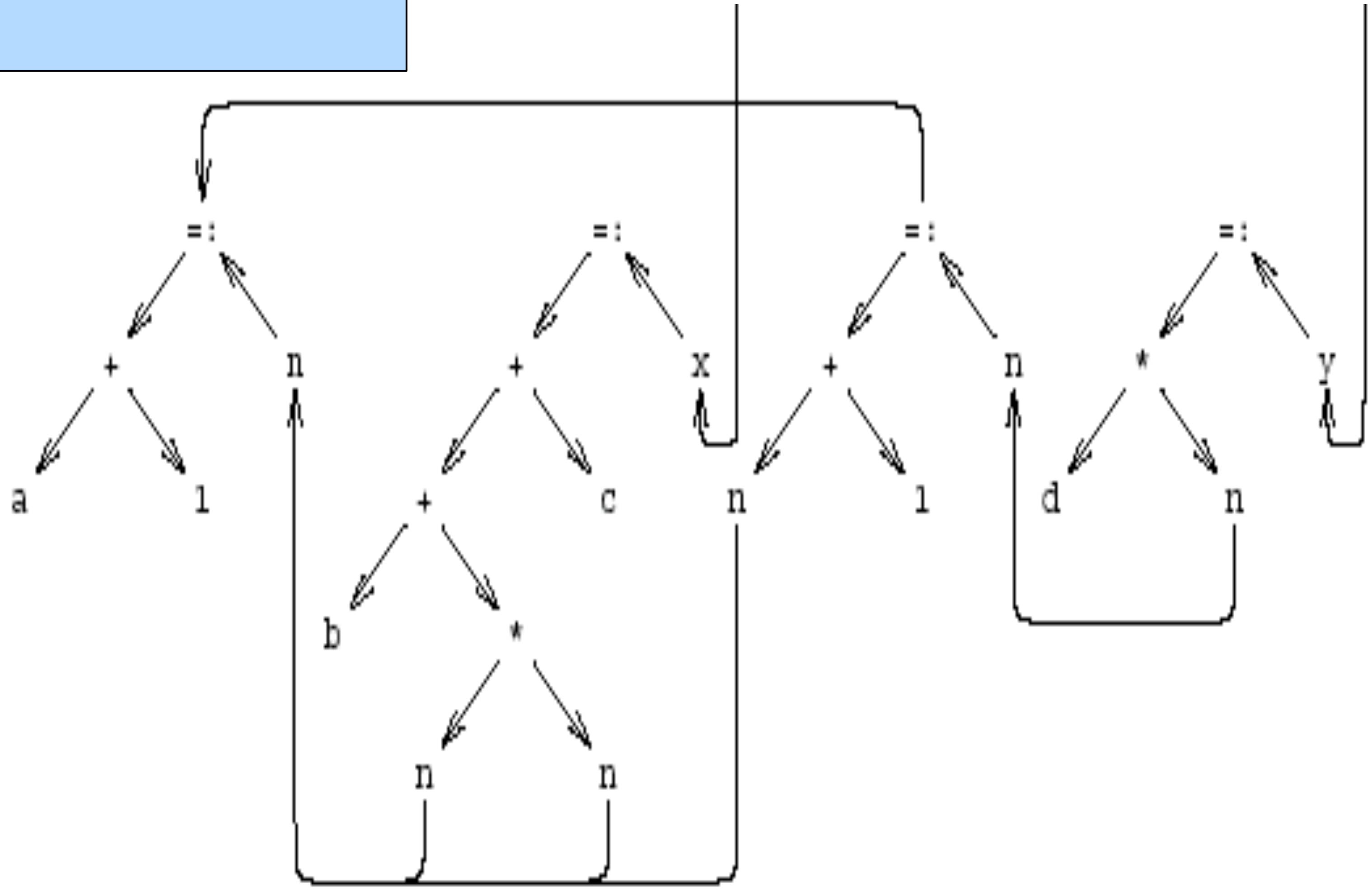
AST for a Basic Block

```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```



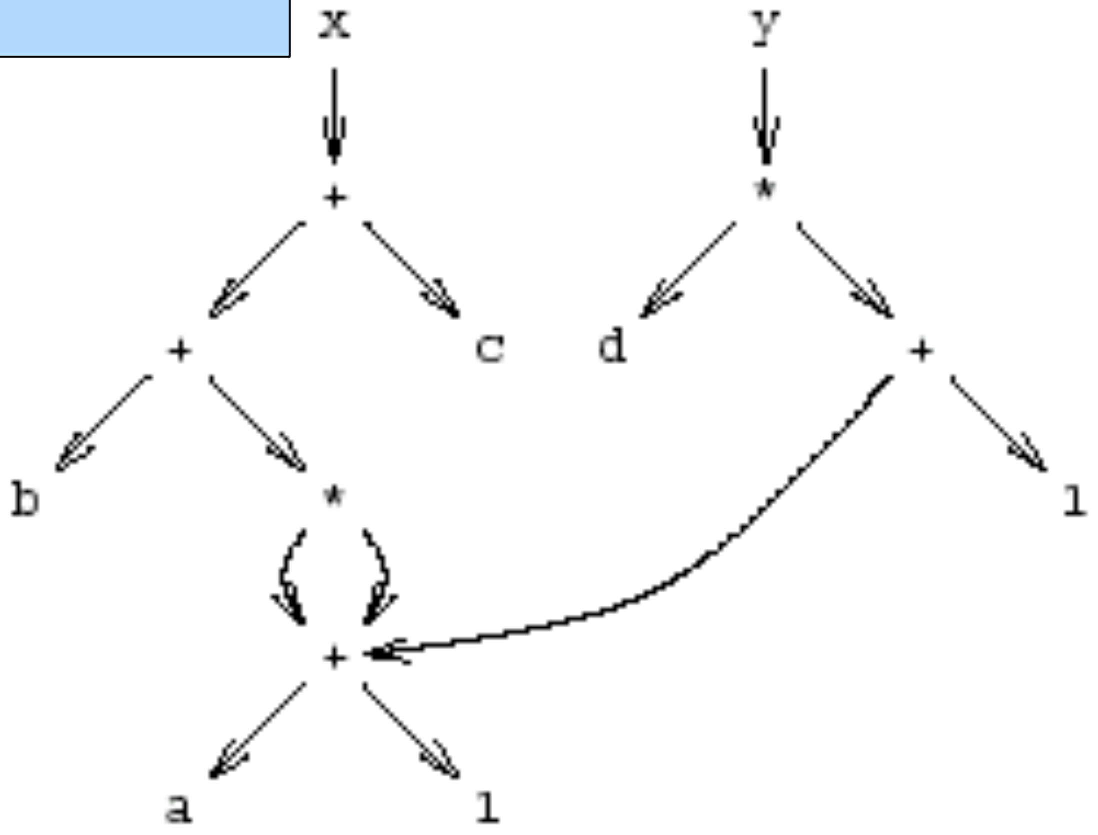
Dependency graph

```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

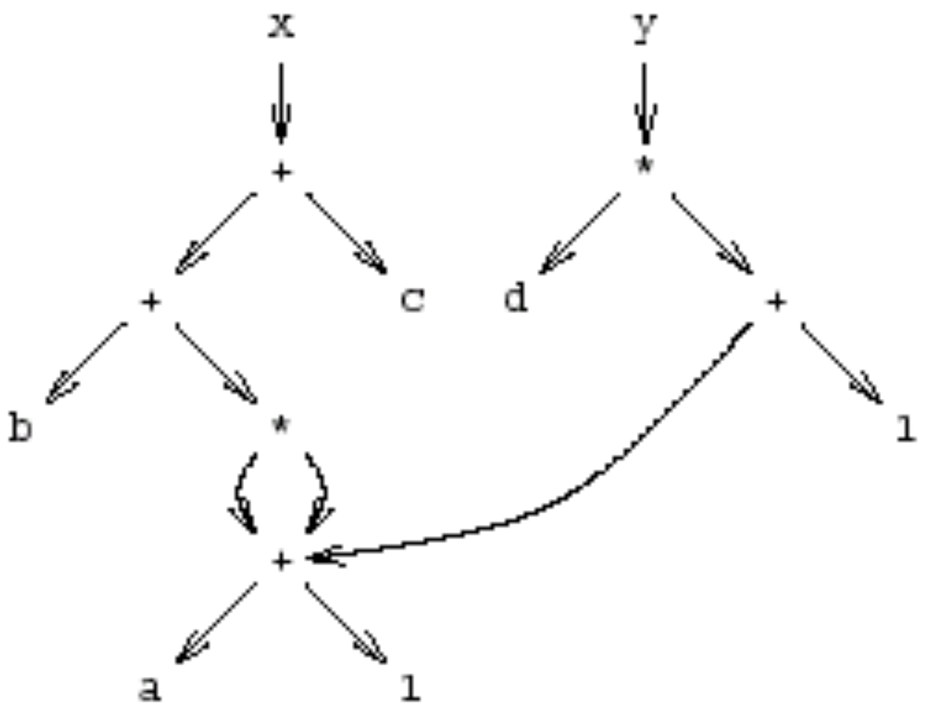


```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

Simplified Data Dependency Graph



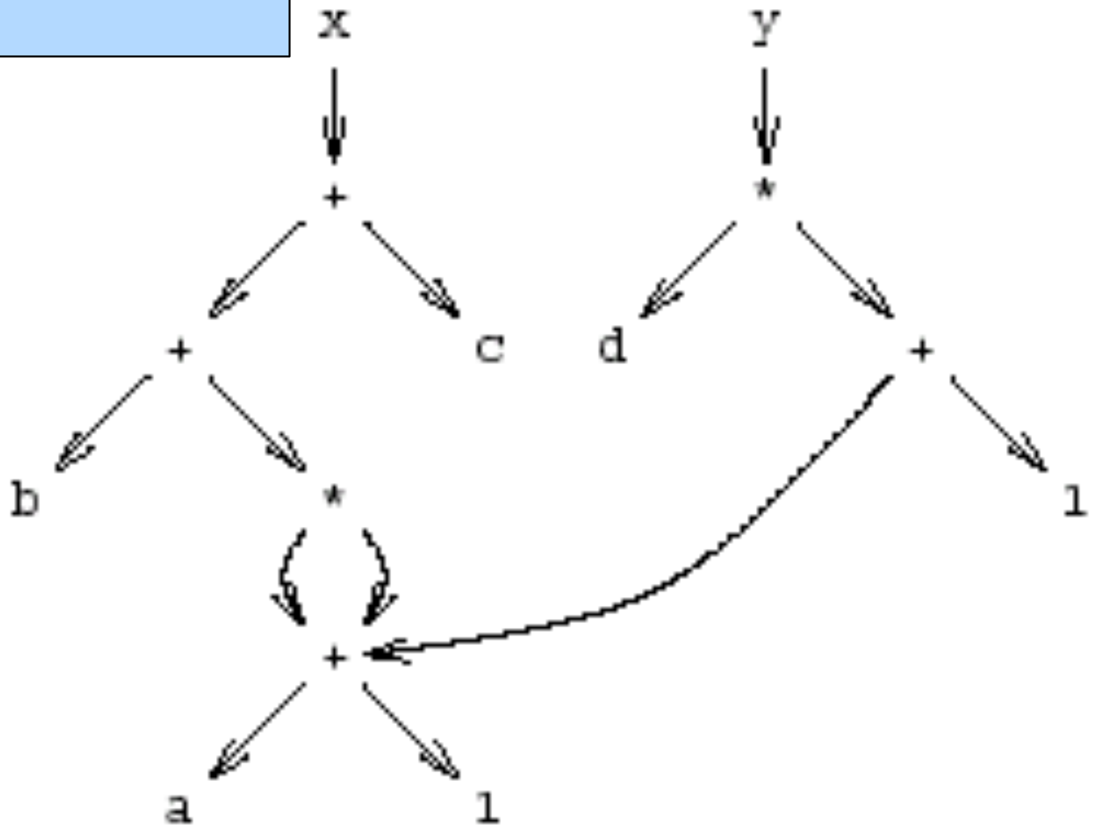
Pseudo Register Target Code



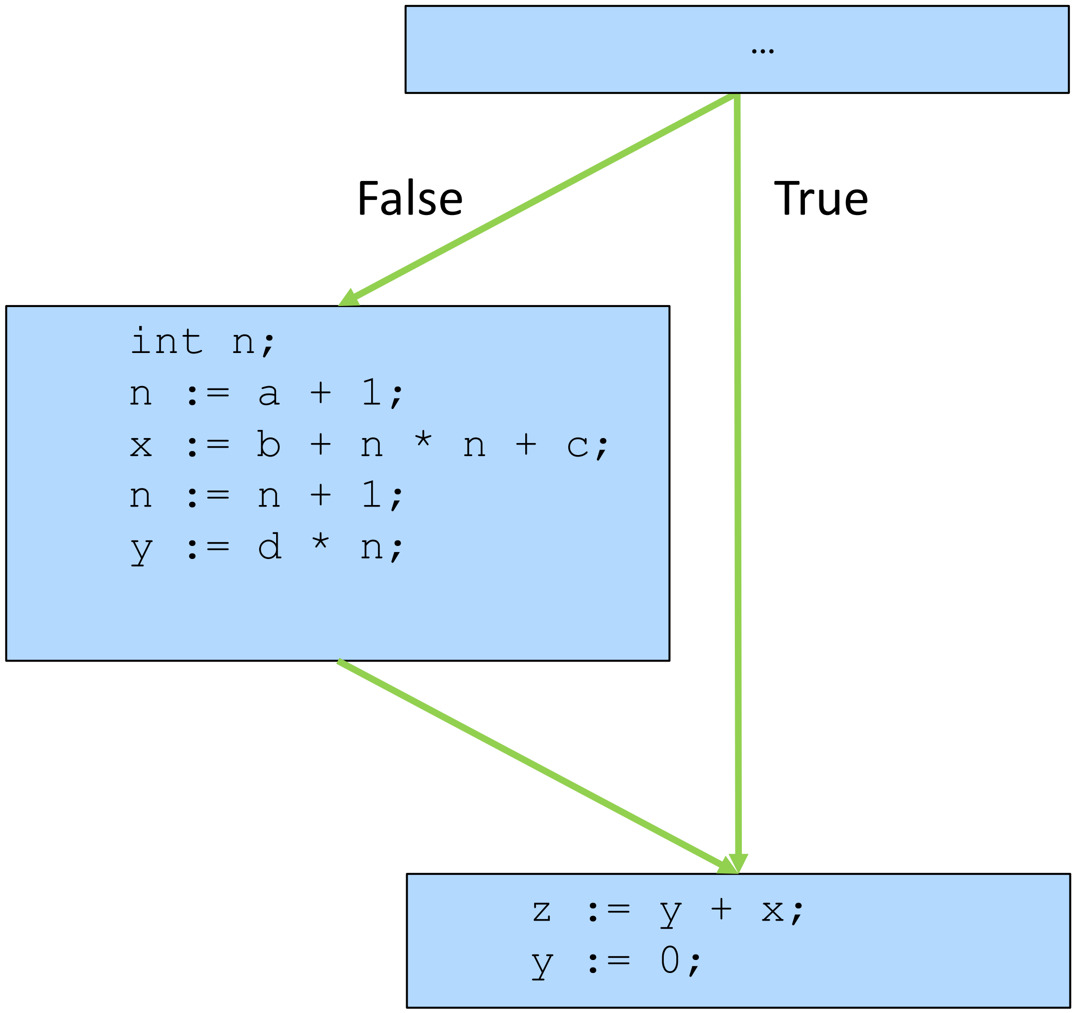
```
Load_Mem     a,R1
Add_Const    1,R1
Load_Reg     R1,X1

Load_Reg     X1,R1
Mult_Reg     X1,R1
Add_Mem      b,R1
Add_Mem      c,R1
Store_Reg    R1,x

Load_Reg     X1,R1
Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg    R1,y
```

```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

Question: Why "`y`"?



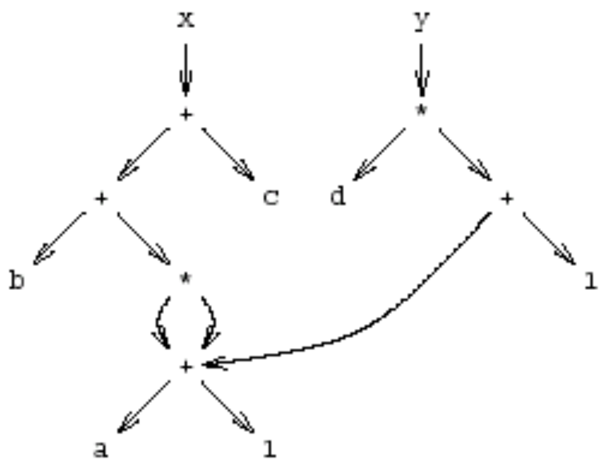
Question: Why "`y`"?



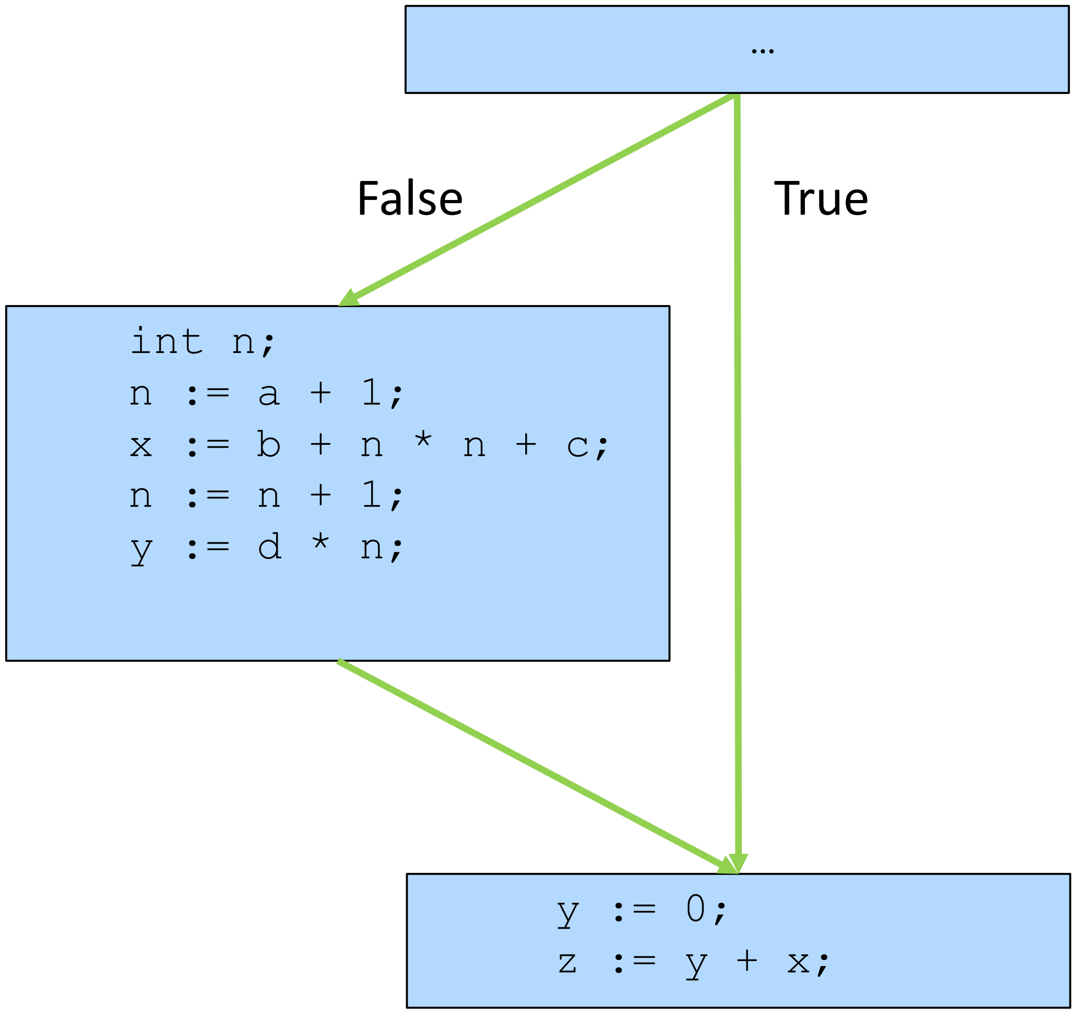
```
…
```

```
int n;
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```

```
z := y + x;
y := 0;
```

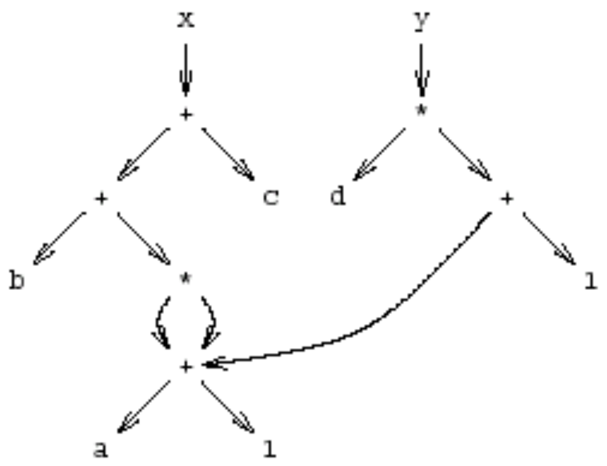


Question: Why "y"?

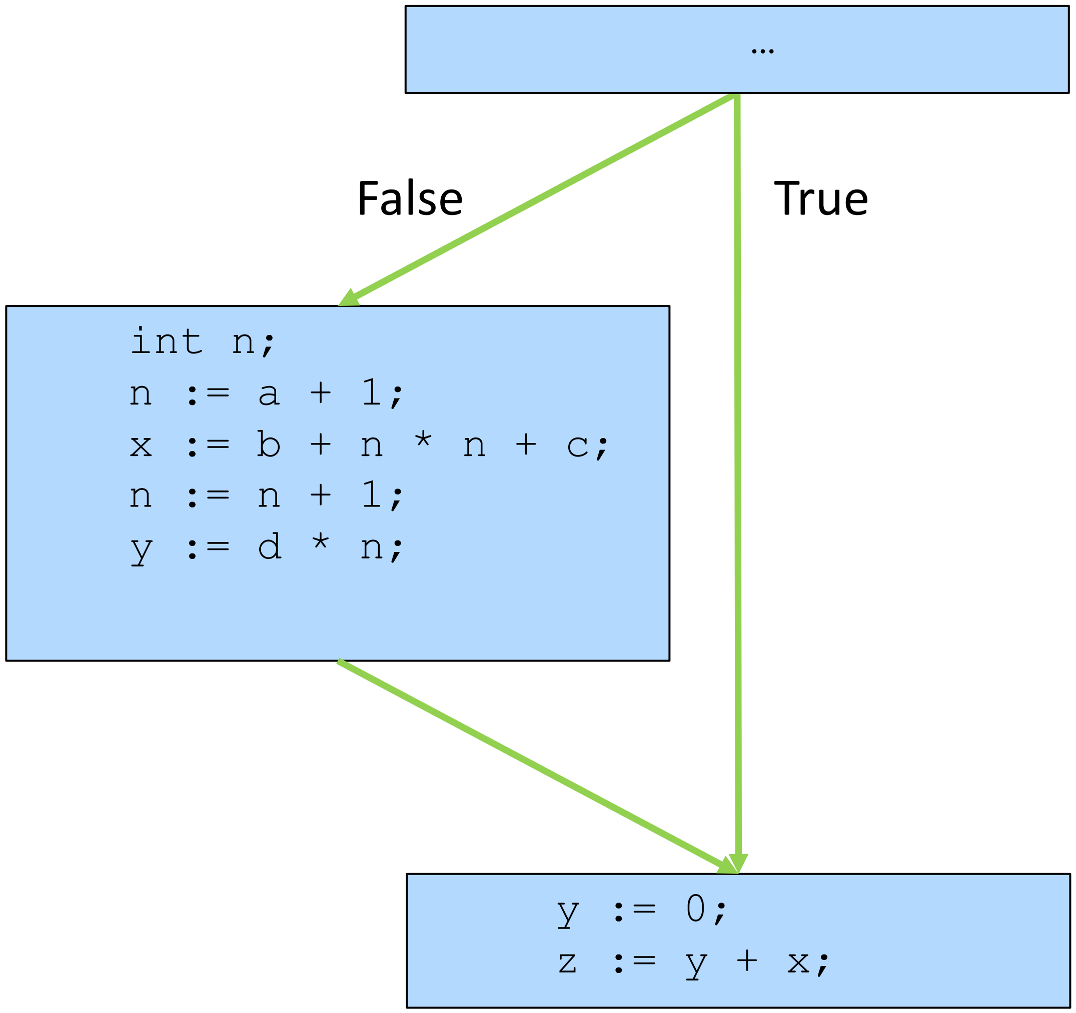


```
int n;
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```

```
y := 0;
z := y + x;
```

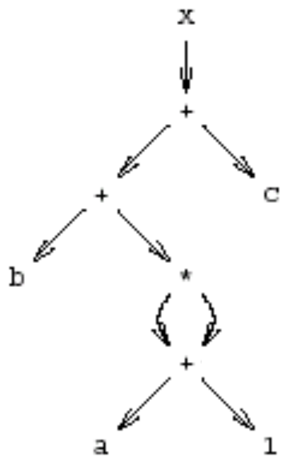


Question: Why "y"?

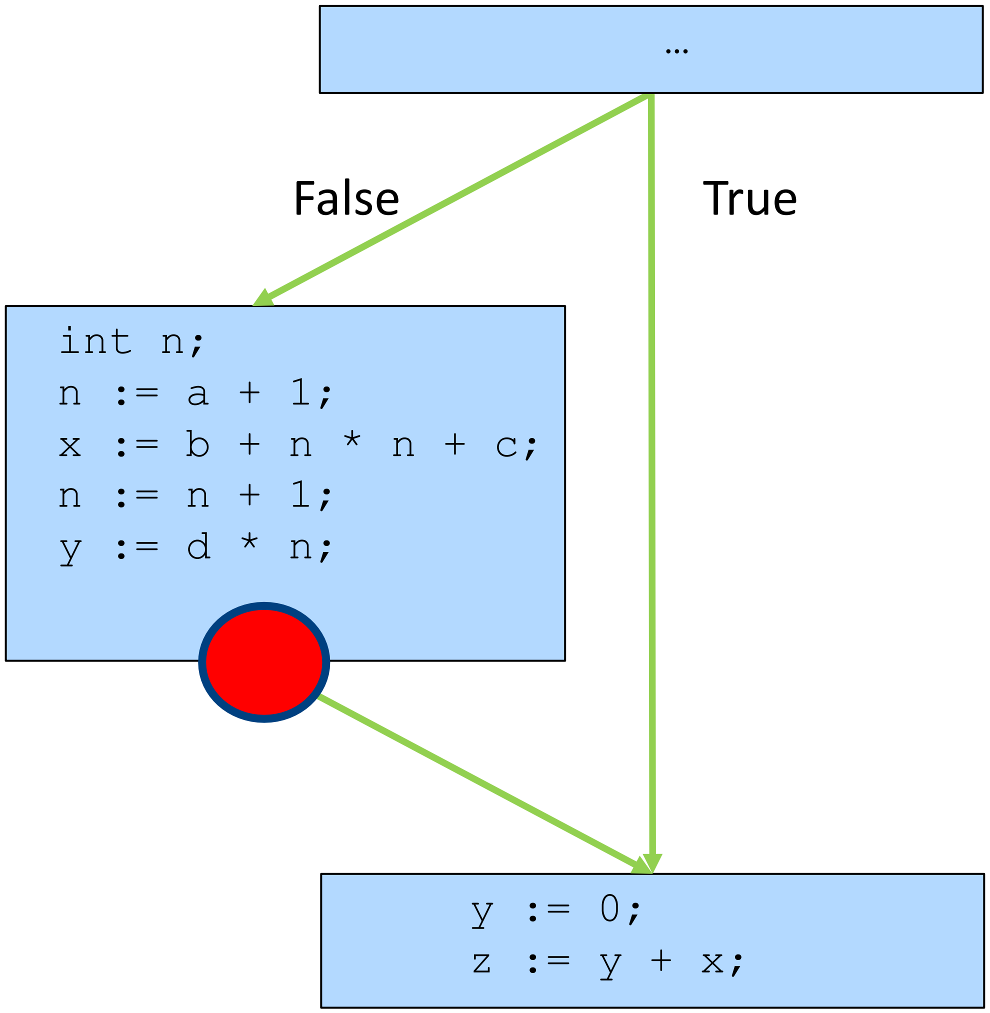


```
int n;
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```

```
y := 0;
z := y + x;
```

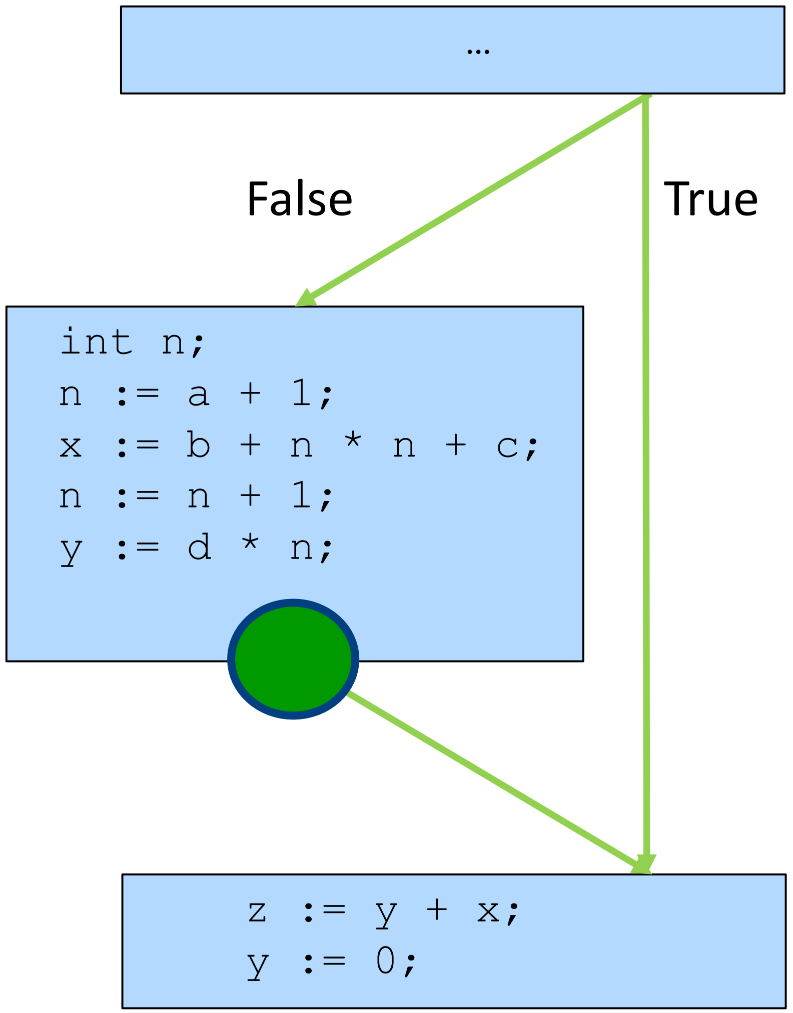


y, dead or alive?



```
int n;
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```

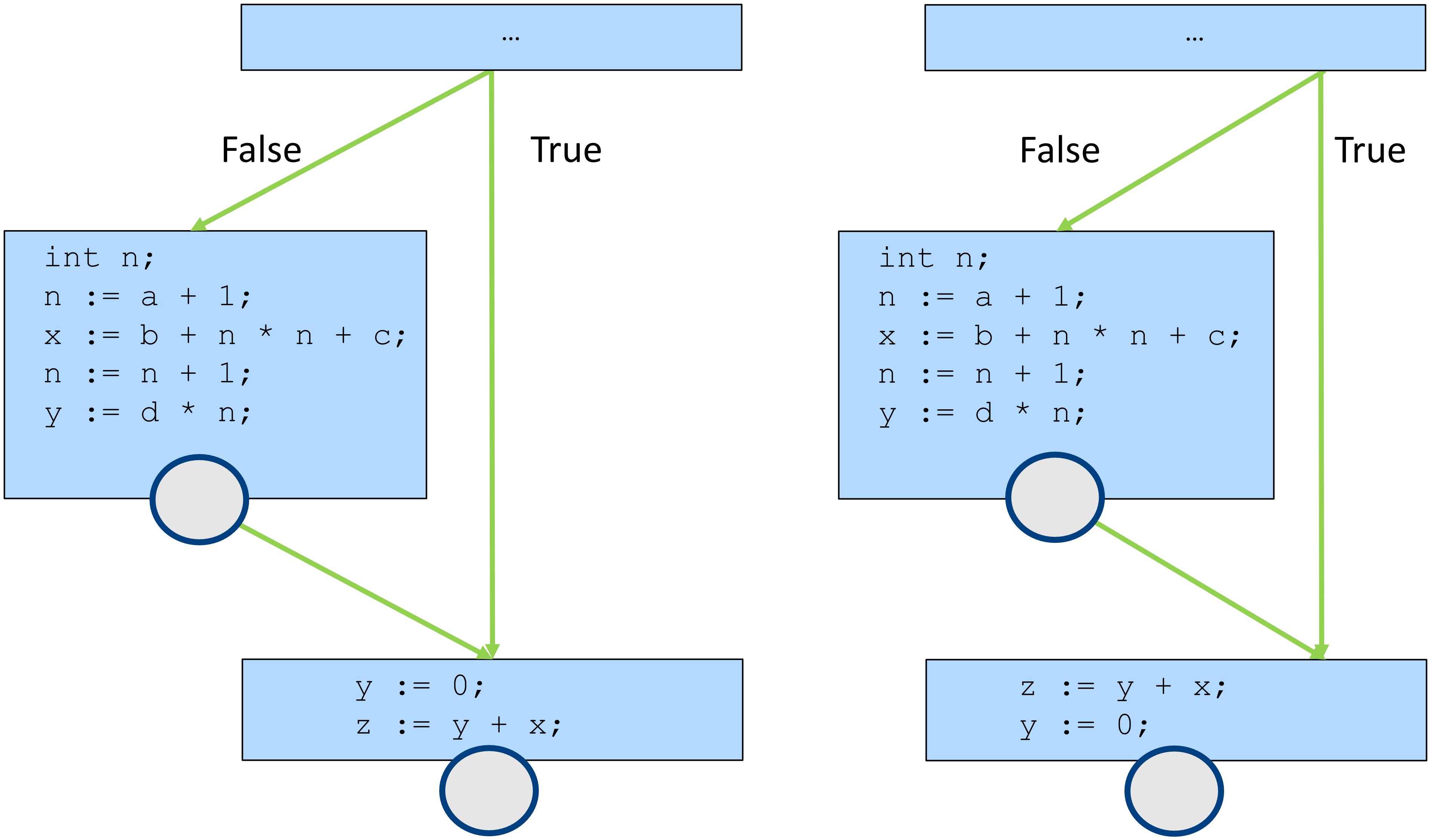
```
y := 0;
z := y + x;
```



```
int n;
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```

```
z := y + x;
y := 0;
```

`x`, dead or alive?



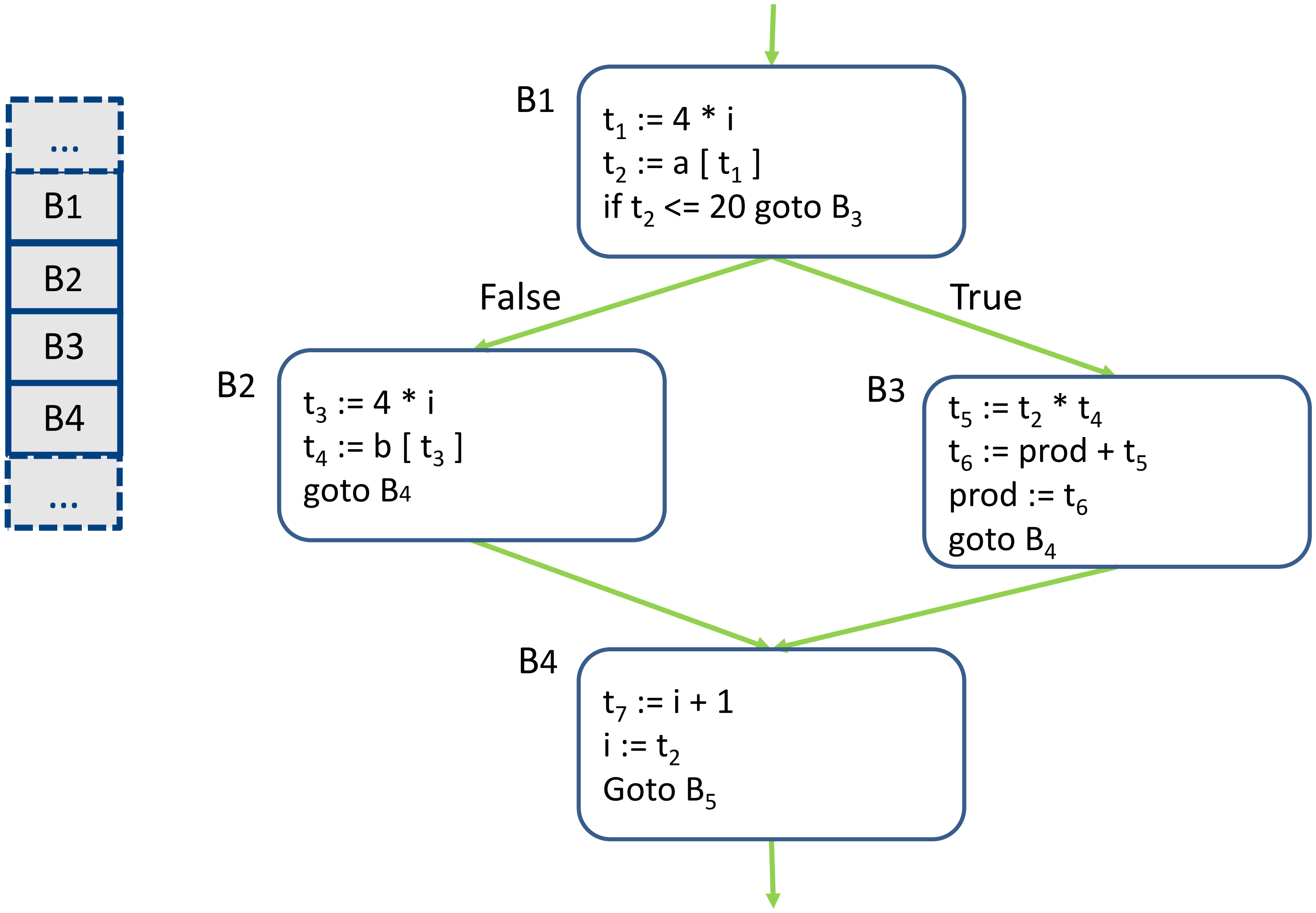
```
int n;
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```

```
y := 0;
z := y + x;
```

```
int n;
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```

```
z := y + x;
y := 0;
```

Another Example



Creating Basic Blocks

- **Input**: A sequence of three-address statements
- **Output**: A list of basic blocks with each three-address statement in exactly one block
- **Method**
  - Determine the set of **leaders** (first statement of a block)
    - The first statement is a leader
    - Any statement that is the target of a jump is a leader
    - Any statement that immediately follows a jump is a leader
  - For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

example

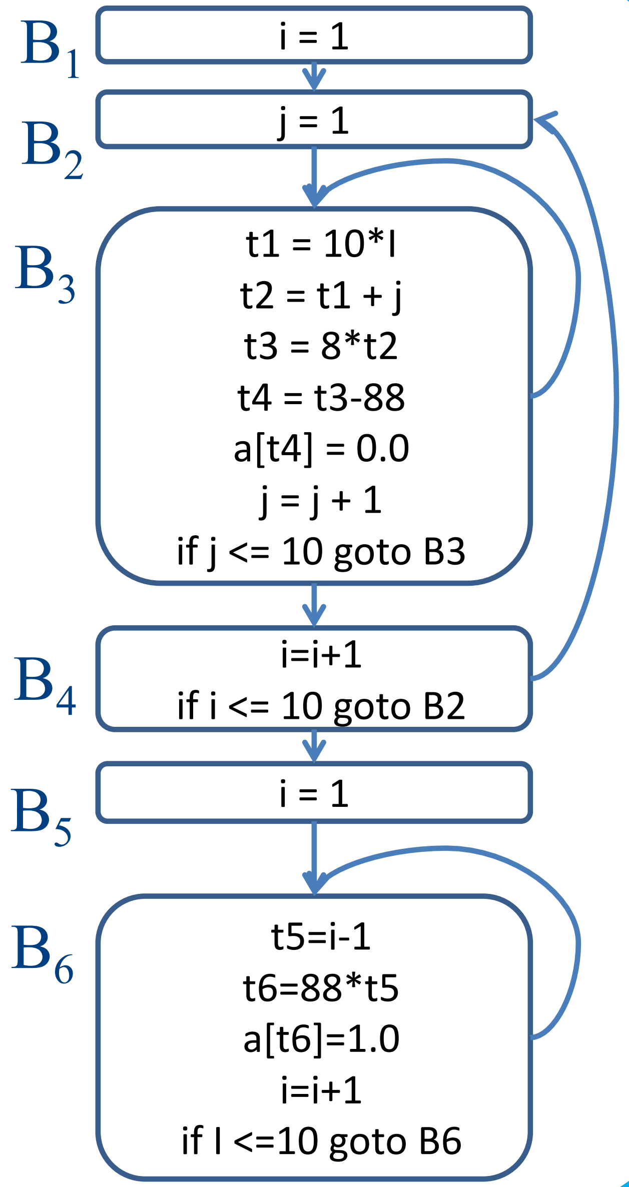
source

```
for i from 1 to 10
do
 for j from 1 to 10
 do
   a[i, j] = 0.0;
for i from 1 to 10
do
 a[i, i] = 1.0;
```

IR

```
1)    i = 1
2)    j =1
3)    t1 = 10*I
4)    t2 = t1 + j
5)    t3 = 8*t2
6)    t4 = t3-88
7)    a[t4] = 0.0
8)    j = j + 1
9)    if j <= 10 goto (3)
10)   i=i+1
11)   if i <= 10 goto (2)
12)   i=1
13)   t5=i-1
14)   t6=88*t5
15)   a[t6]=1.0
16)   i=i+1
17)   if I <=10 goto (13)
```

CFG



Example: Code Block

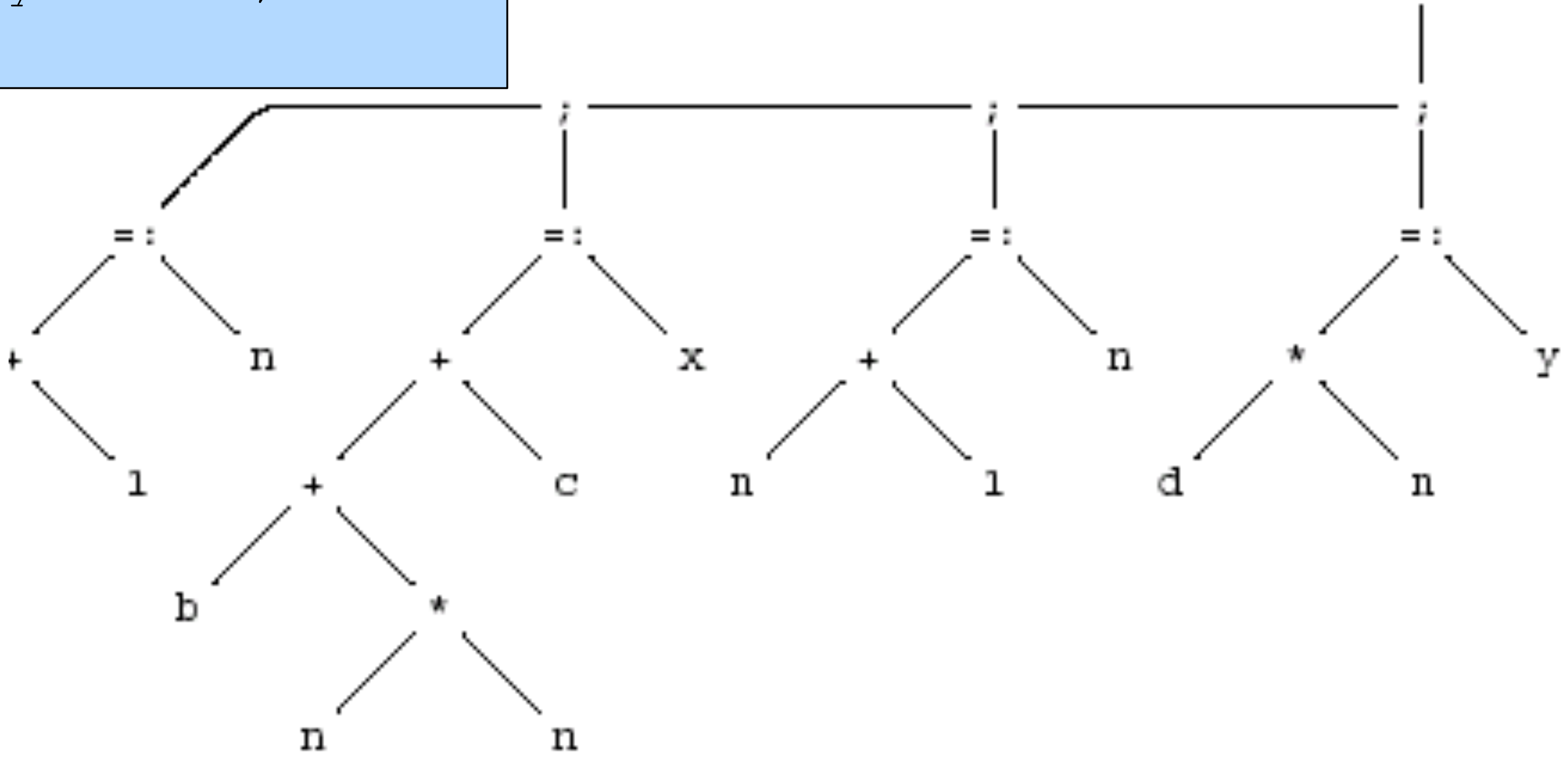
```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

Example: Basic Block

```
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
```

AST of the Example

```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```



Optimized Code (gcc)

```
{
  int n;
  n := a + 1;
  x := b + n * n + c;
  n := n + 1;
  y := d * n;
}
```

```
Load_Mem     a,R1
Add_Const    1,R1
Load_Reg     R1,R2

Mult_Reg     R1,R2
Add_Mem      b,R2
Add_Mem      c,R2
Store_Reg    R2,x

Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg    R1,y
```

Register Allocation for B.B.

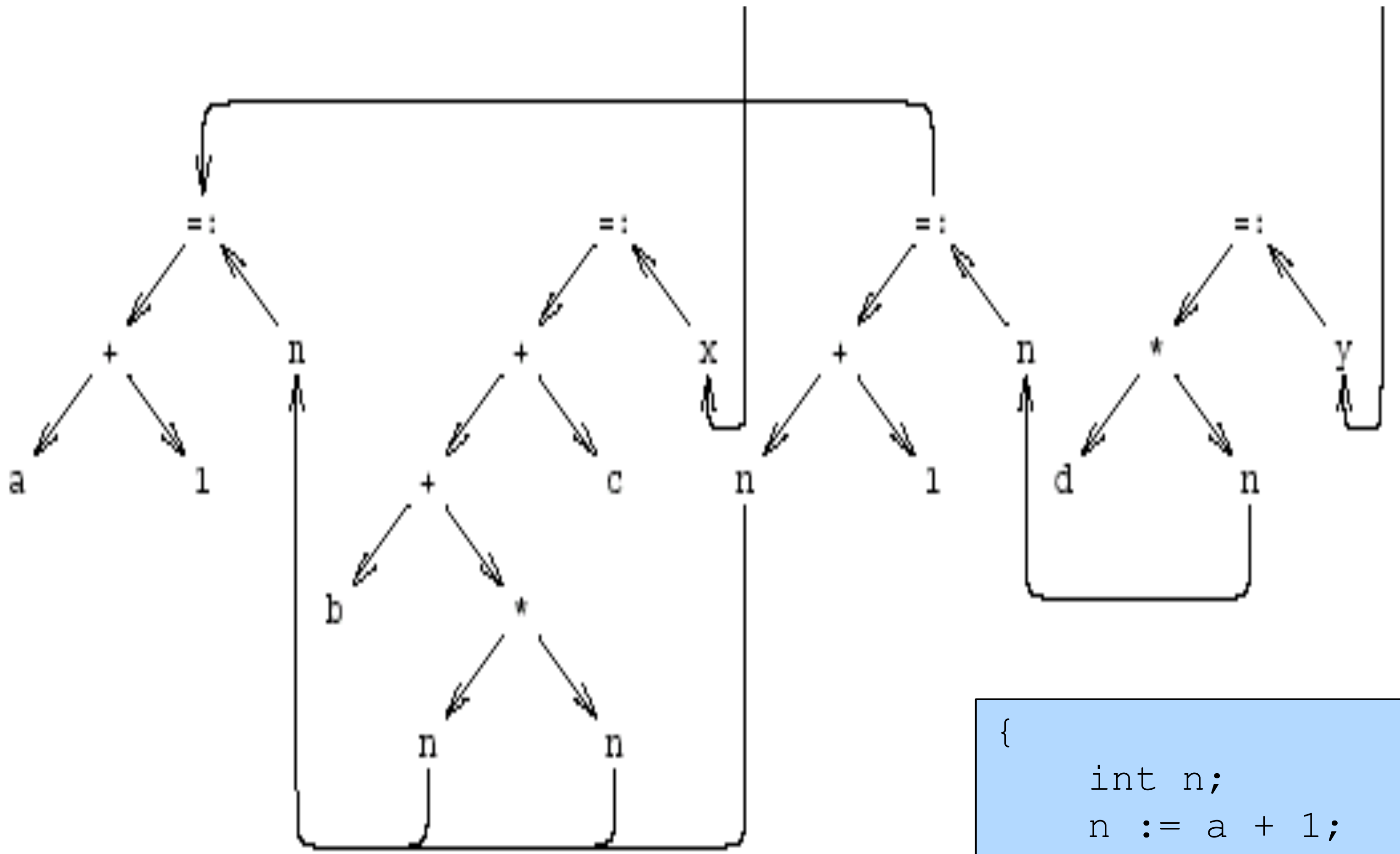
- Dependency graphs for basic blocks
- Transformations on dependency graphs
- From dependency graphs into code
  - Instruction selection
    - linearizations of dependency graphs
  - Register allocation
    - At the basic block level

Dependency graphs

- TAC imposes an order of execution
  - But the compiler can reorder assignments as long as the program results are not changed

- Define a partial order on assignments
  - $a < b \Leftrightarrow$ a must be executed before b
  - Represented as a directed graph
    - Nodes are assignments
    - Edges represent dependency
  - Acyclic for basic blocks

Running Example



```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

Sources of dependency

- Data flow inside expressions
  - Operator depends on operands
  - Assignment depends on assigned expressions
- Data flow between statements
  - From assignments to their use
  - Pointers complicate dependencies

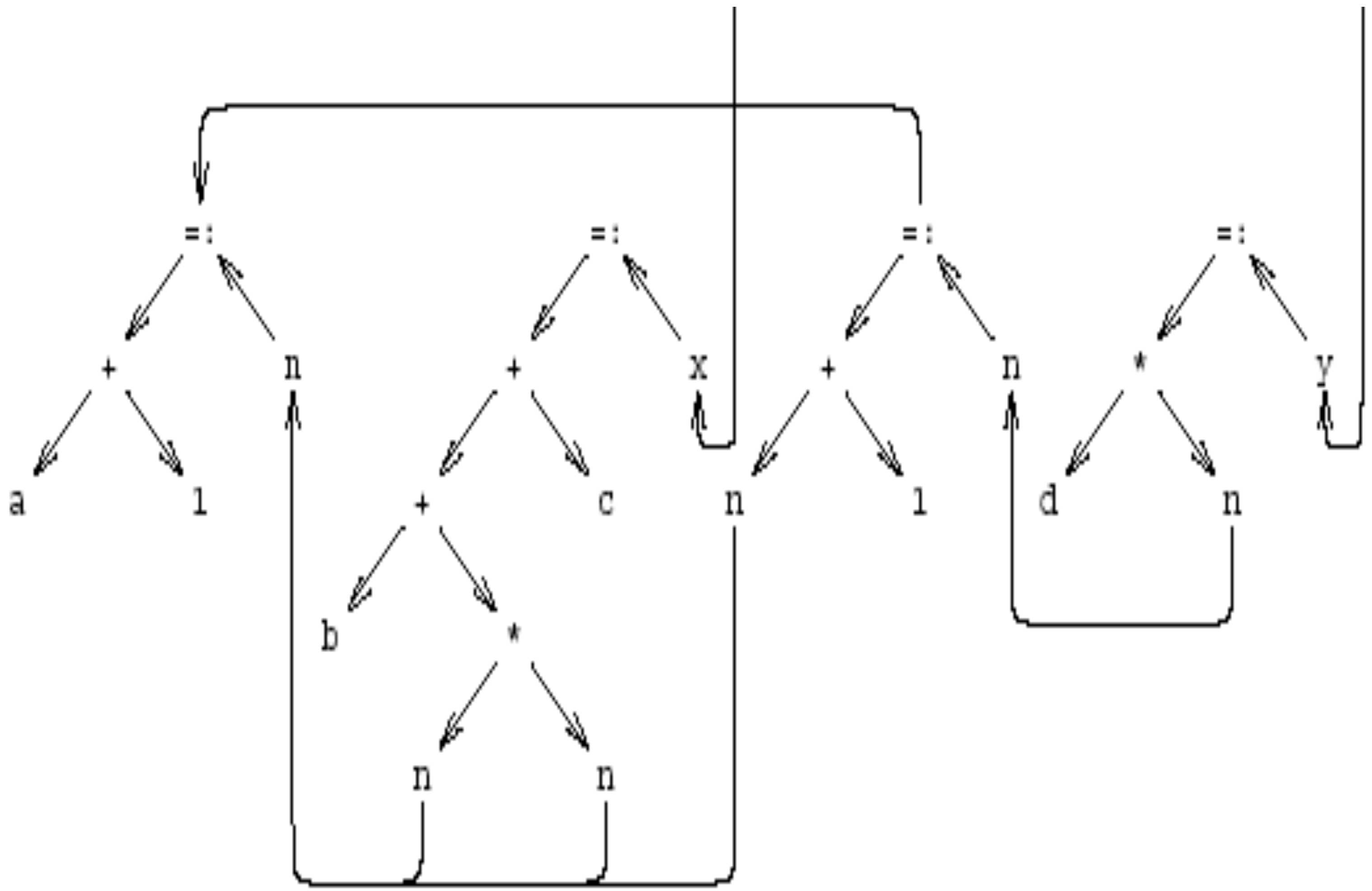
Sources of dependency

- Order of subexpresion evaluation is immaterial
  - As long as inside dependencies are respected
- The order of uses of a variable X are immaterial as long as:
  - X is used between dependent assignments
  - Before next assignment to X

Creating Dependency Graph from AST

- Nodes AST becomes nodes of the graph
- Replaces arcs of AST by dependency arrows
  - Operator $\rightarrow$ Operand
  - Create arcs from assignments to uses
  - Create arcs between assignments of the same variable
- Select output variables (roots)
- Remove ; nodes and their arrows

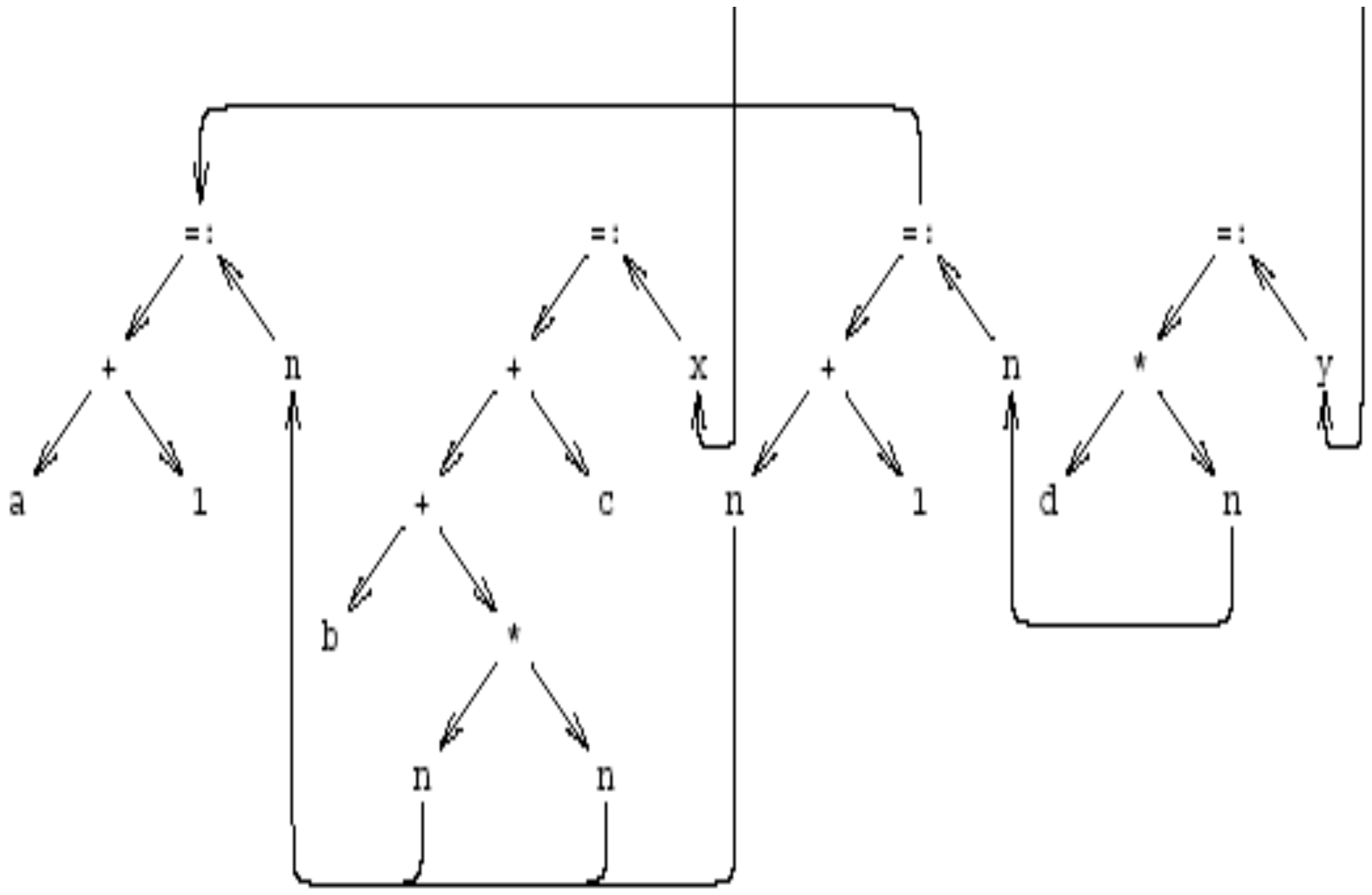
Running Example



Dependency Graph Simplifications

- Short-circuit assignments
  - Connect variables to assigned expressions
  - Connect expression to uses
- Eliminate nodes not reachable from roots

Running Example



Cleaned-Up Data Dependency Graph

$$\begin{tikzcd}
 & & x \arrow[d] & & & & y \arrow[d] & & \\
 & & + \arrow[dl] \arrow[dr] & & & & * \arrow[dl] \arrow[dr] & & \\
 & + \arrow[dl] \arrow[dr] & & c & & d & & + \arrow[dr] \arrow[ddlllll, bend left] & \\
b & & * \arrow[d, bend right] \arrow[d, bend left] & & & & & & 1 \\
 & & + \arrow[dl] \arrow[dr] & & & & & & \\
 & a & & 1 & & & & &
\end{tikzcd}$$

Common Subexpressions

- Repeated subexpressions
- Examples

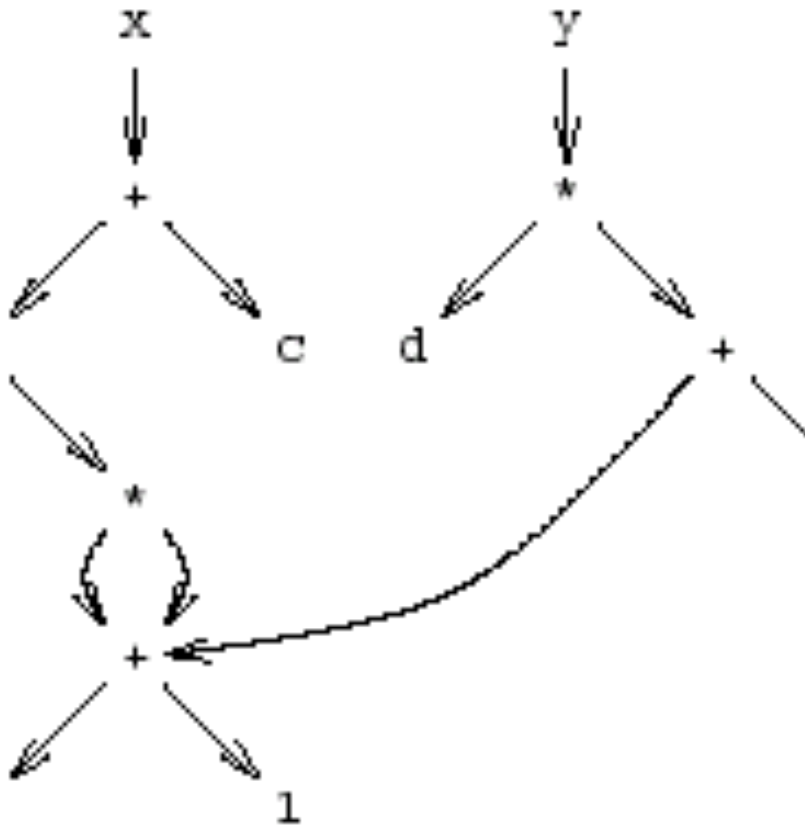
```
x = a * a  +   2 * a * b + b * b;
y = a * a  –   2 * a * b + b * b;
n[i] := n[i] +m[i]
```

- Can be eliminated by the compiler
  - In the case of basic blocks rewrite the DAG

From Dependency Graph into Code

- Linearize the dependency graph
  - Instructions must follow dependency
- Many solutions exist
- Select the one with small runtime cost
- Assume infinite number of registers
  - Symbolic registers
  - Assign registers later
    - May need additional spill
  - Possible Heuristics
    - Late evaluation
    - Ladders

Pseudo Register Target Code



```
Load_Mem     a,R1
Add_Const    1,R1
Load_Reg     R1,X1

Load_Reg     X1,R1
Mult_Reg     X1,R1
Add_Mem      b,R1
Add_Mem      c,R1
Store_Reg    R1,x

Load_Reg     X1,R1
Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg    R1,y
```

Non optimized vs Optimized Code

```
Load_Mem     a,R1
Add_Const    1,R1
Load_Reg     R1,X1

Load_Reg     X1,R1
Mult_Reg     X1,R1
Add_Mem      b,R1
Add_Mem      c,R1
Store_Reg    R1,x

Load_Reg     X1,R1
Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg    R1,y
```

```
Load_Mem     a,R1
Add_Const    1,R1
Load_Reg     R1,R2

Load_Reg     R2,R1
Mult_Reg     R2,R1
Add_Mem      b,R1
Add_Mem      c,R1
Store_Reg    R1,x

Load_Reg     R2,R1
Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg    R1,y
```

```
ıd_Mem      a,R1
l_Const     1,R1
ıd_Reg      R1,R2

.t_Reg      R1,R2
l_Mem       b,R2
l_Mem       c,R2
ɔre_Reg     R2,x

l_Const     1,R1
.t_Mem      d,R1
ɔre_Reg     R1,y
```

```
{
  int n;
  n := a + 1;
  x := b + n * n + c;
  n := n + 1;
  y := d * n;
}
```

Register Allocation

- Maps symbolic registers into physical registers
  - Reuse registers as much as possible
  - Graph coloring (next)
    - Undirected graph
    - Nodes = Registers (Symbolic and real)
    - Edges = Interference
    - May require spilling

Register Allocation for Basic Blocks

- Heuristics for code generation of basic blocks
- Works well in practice
- Fits modern machine architecture
- Can be extended to perform other tasks
  - Common subexpression elimination
- But basic blocks are small
- Can be generalized to a procedure

Problem	Technique	Quality
Expression trees, using register-register or memory-register instructions	Weighted trees; Figure 4.30	
with sufficient registers:		Optimal
with insufficient registers:		Optimal
Dependency graphs, using register-register or memory-register instructions	Ladder sequences; Section 4.2.5.2	Heuristic
Expression trees, using any instructions with cost function	Bottom-up tree rewriting; Section 4.2.6	
with sufficient registers:		Optimal
with insufficient registers:		Heuristic
Register allocation when all interferences are known	Graph coloring; Section 4.2.7	Heuristic

The End