Compilation

0368-3133 (Semester A, 2013/14)

Lecture 12: Abstract Interpretation

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv and Eran Yahav

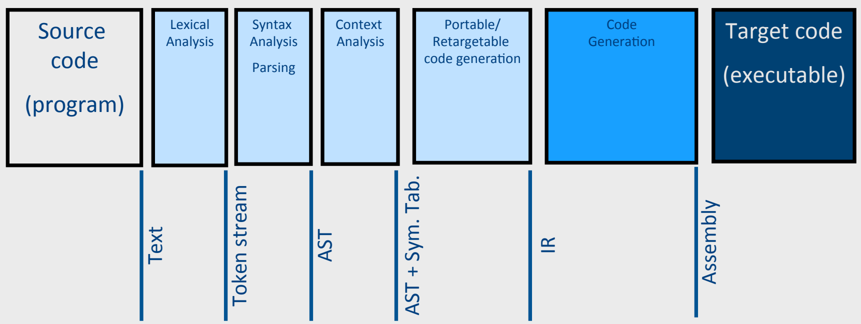
What is a compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

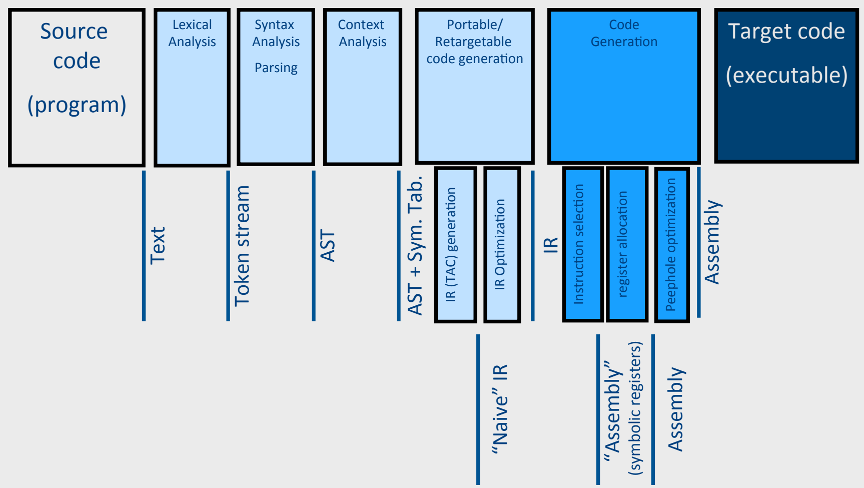
The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

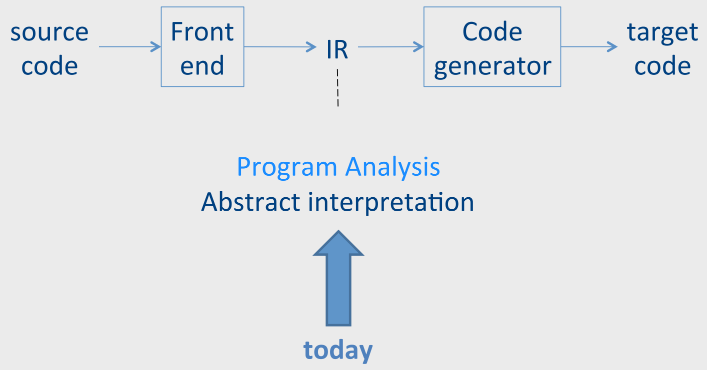
Stages of compilation



Stages of Compilation



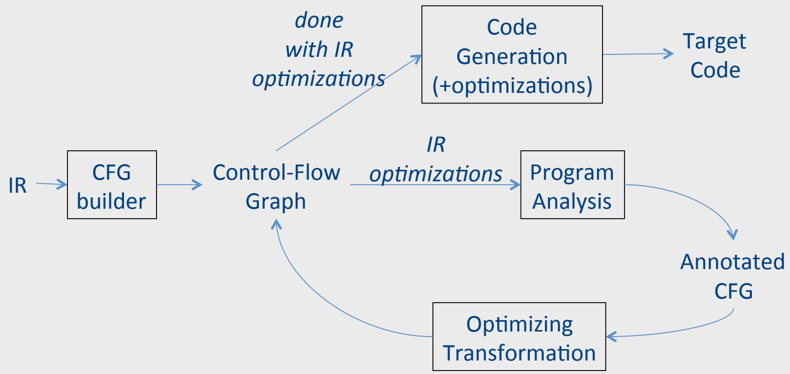
Optimization points



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?)*

Optimization path



Soundness

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

"At this point in the program, **x** holds some integer value"

Soundness

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

"At this point in the program, **x** is either 137 or 42"

(Un) Soundness

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

"At this point in the program, **x** is 137"

Soundness & Precision

```
int x;
int y;

if (y < 5)
    x = 137;
else
    x = 42;

Print(x);
```

"At this point in the program, **x** is either 137, 42, or 271"

Semantics-preserving optimizations

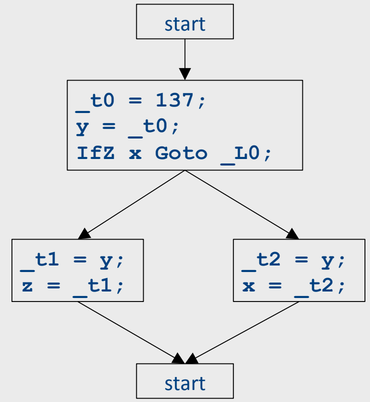
- An optimization is **semantics-preserving** if it does not alter the semantics (meaning) of the original program
  - ✓ Eliminating unnecessary temporary variables
  - ✓ Computing values that are known statically at compile-time instead of computing them at runtime
  - ✓ Evaluating iteration-independent expressions outside of a loop instead of inside
  - ✗ Replacing bubble sort with quicksort (why?)
- The optimizations we will consider in this class are all semantics-preserving

Types of optimizations

- An optimization is local if it works on just a single basic block
- An optimization is global if it works on an entire control-flow graph

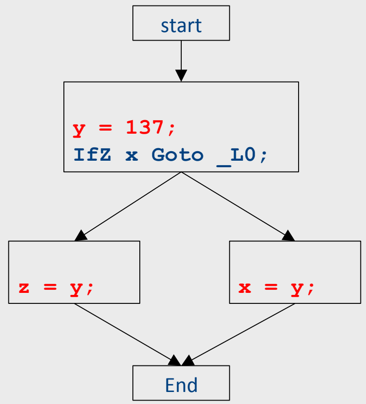
Local optimizations

```
int main() {
   int x;
   int y;
   int z;

   y = 137;
   if (x == 0)
      z = y;
   else
      x = y;
}
```



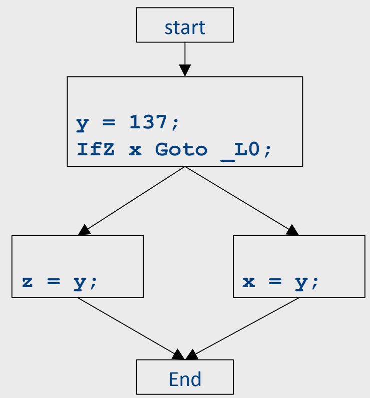
Local optimizations

```
int main() {
   int x;
   int y;
   int z;

   y = 137;
   if (x == 0)
      z = y;
   else
      x = y;
}
```



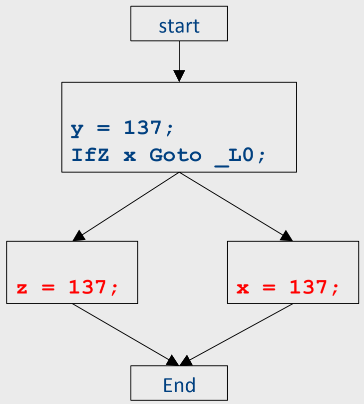
Global optimizations

```
int main() {
   int x;
   int y;
   int z;

   y = 137;
   if (x == 0)
      z = y;
   else
      x = y;
}
```



Global optimizations

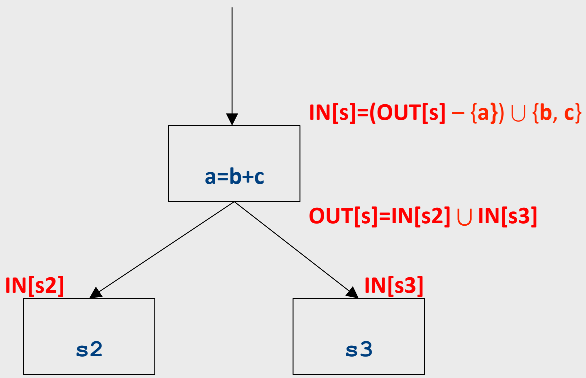
```
int main() {
   int x;
   int y;
   int z;

   y = 137;
   if (x == 0)
      z = y;
   else
      x = y;
}
```



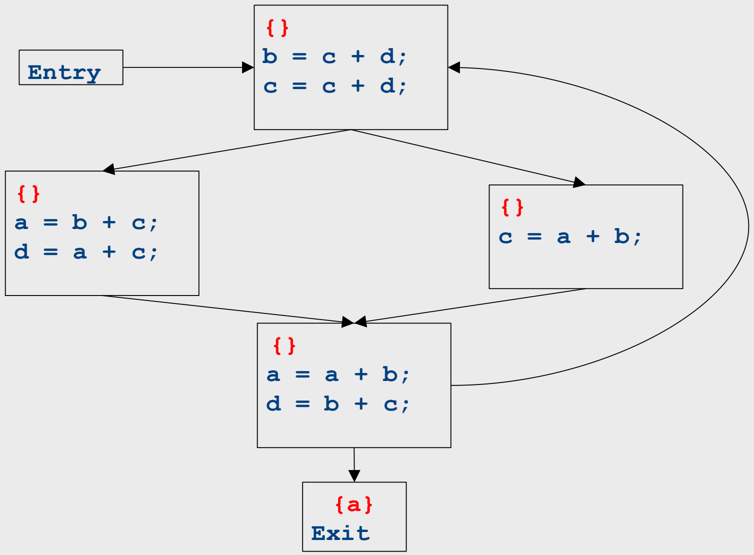
Global Analyses

- Common subexpression elimination
- Copy propagation
- Dead code elimination
- Live variables
  - Used for register allocation

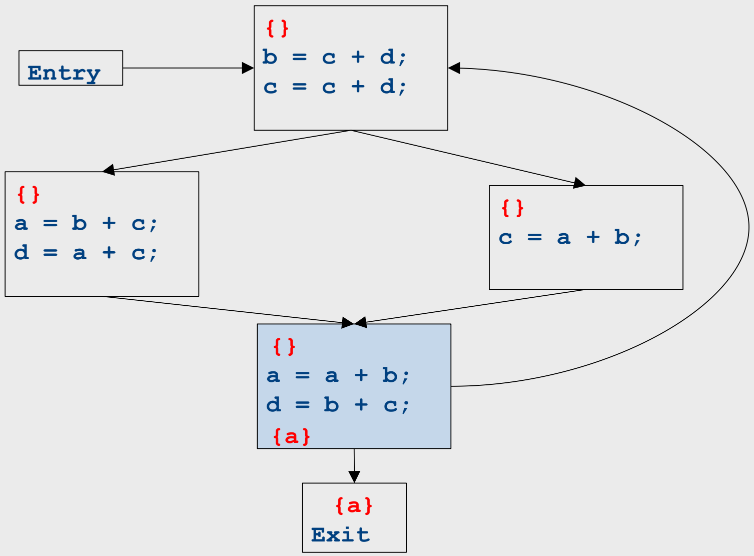
Global liveness analysis



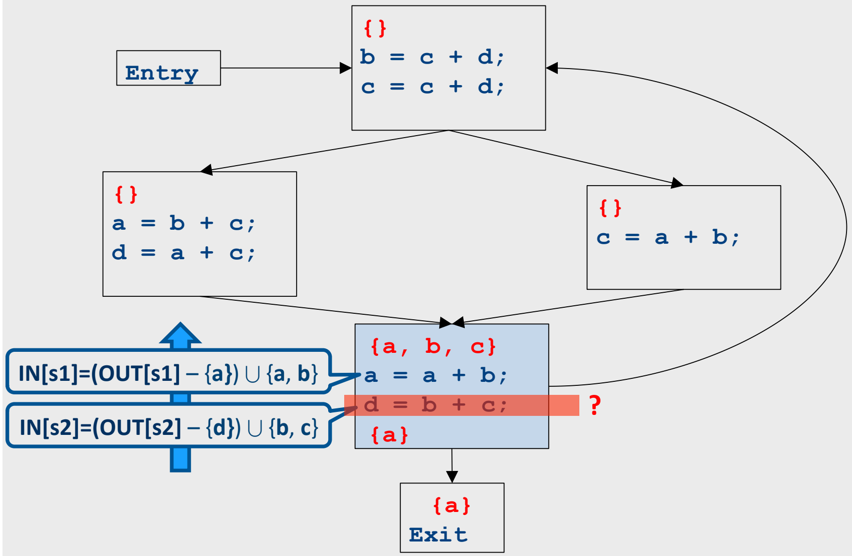
Live variable analysis- initialization



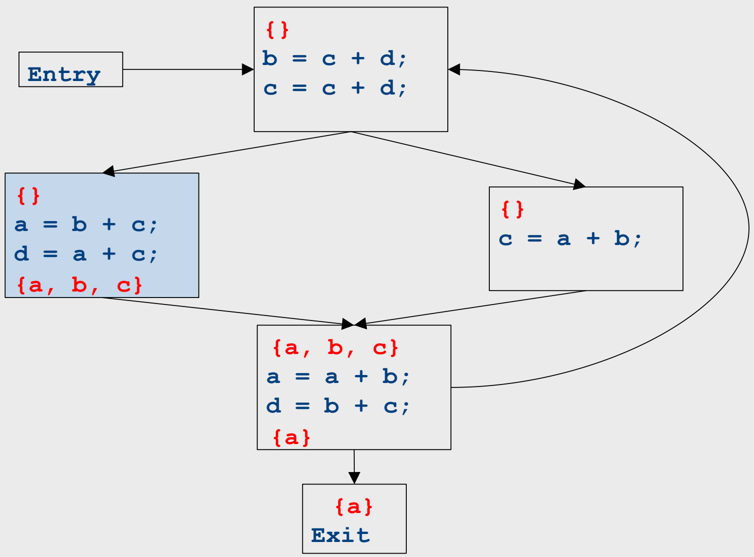
Live variable analysis



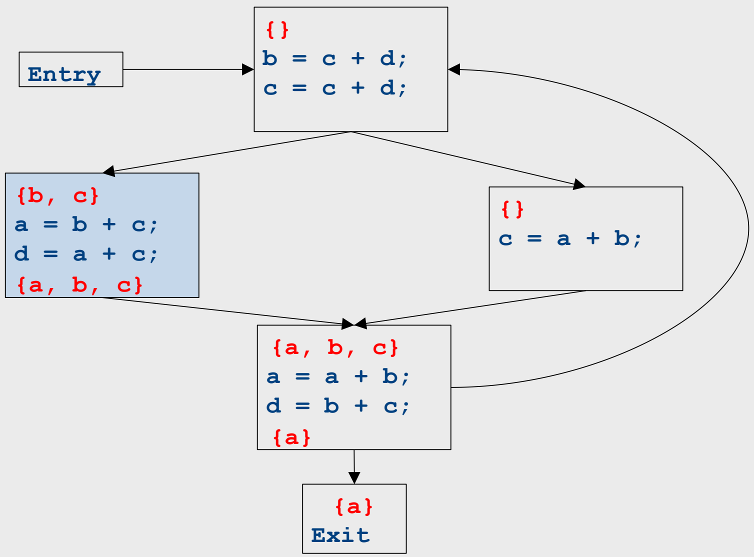
Live variable analysis



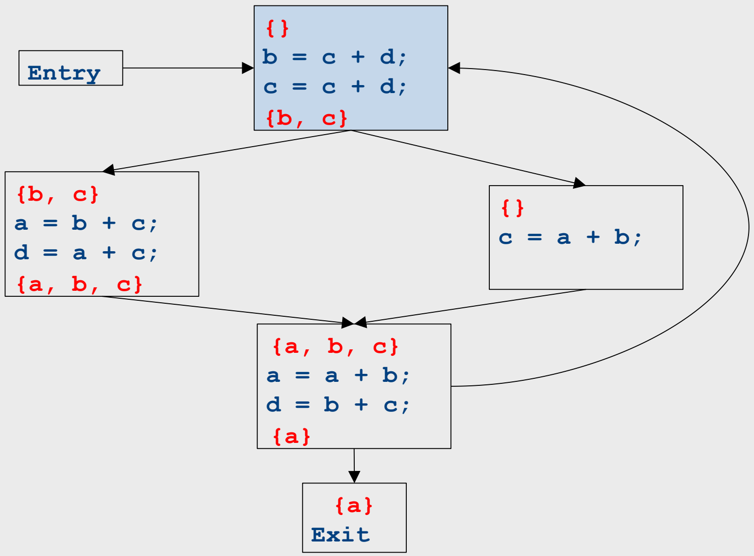
Live variable analysis



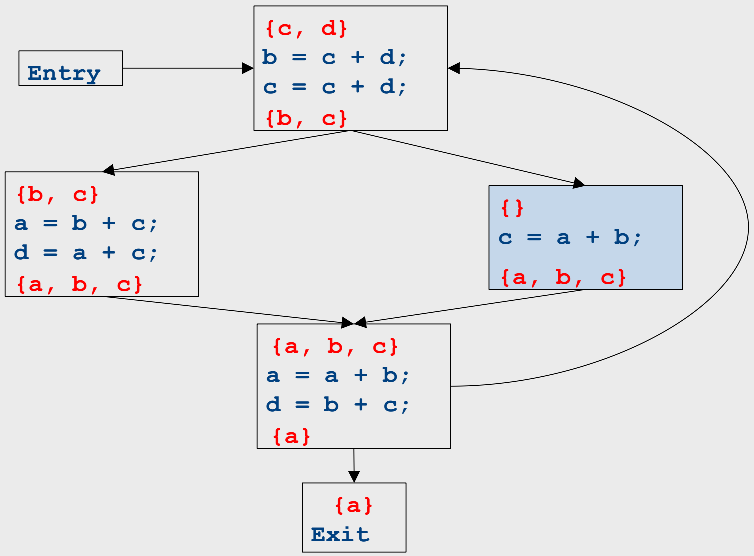
CFGs with loops - iteration



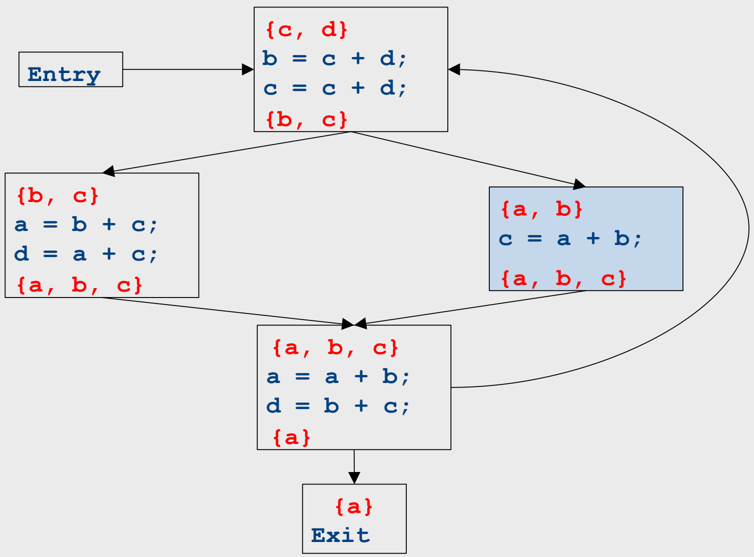
CFGs with loops - iteration



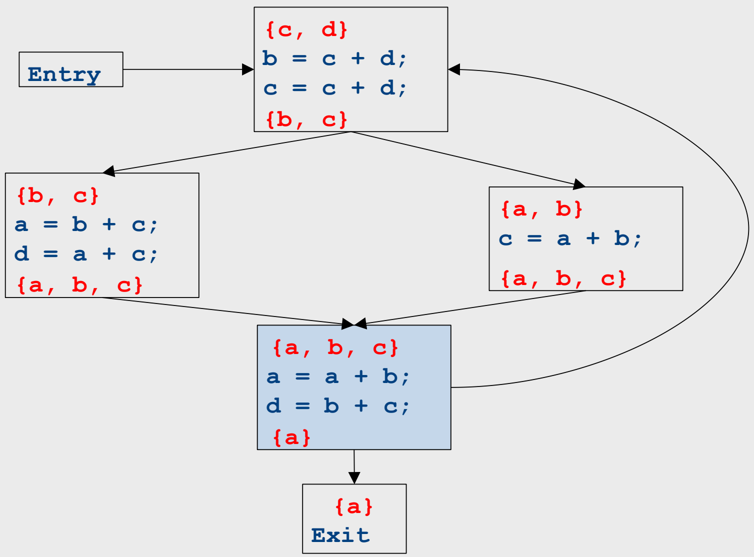
CFGs with loops - iteration



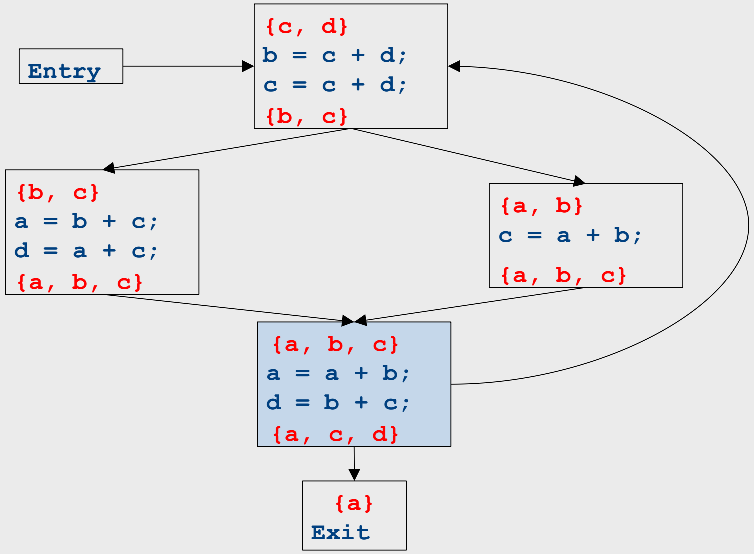
Live variable analysis



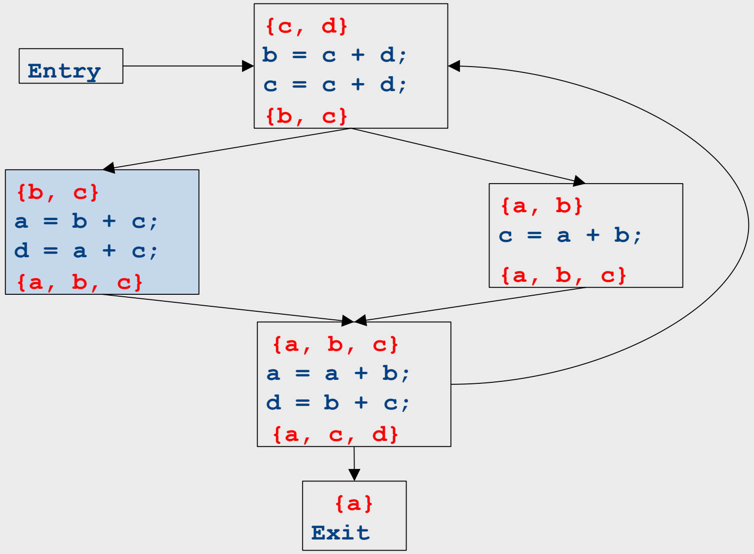
Live variable analysis



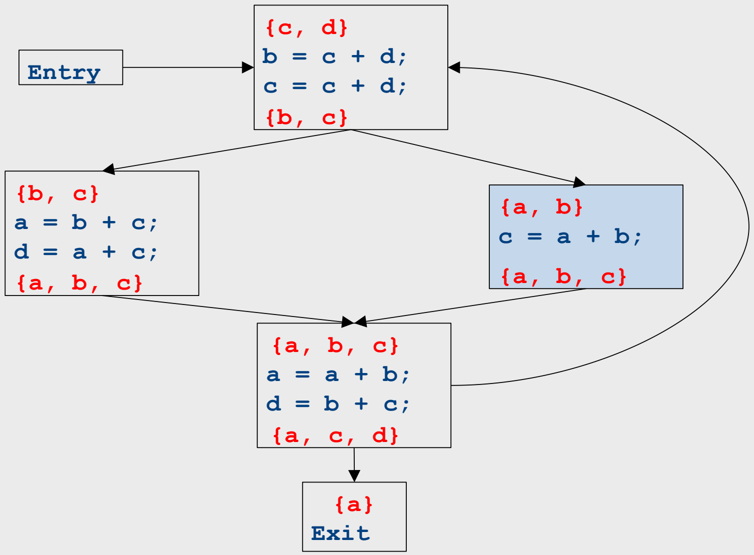
Live variable analysis



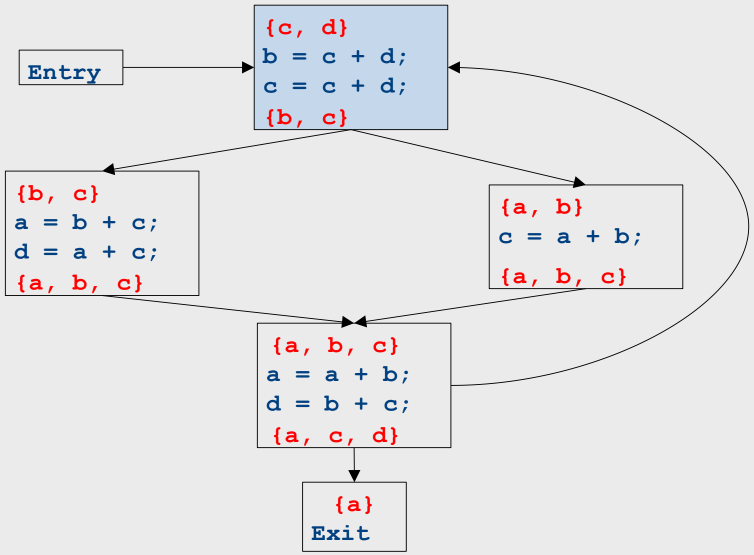
Live variable analysis



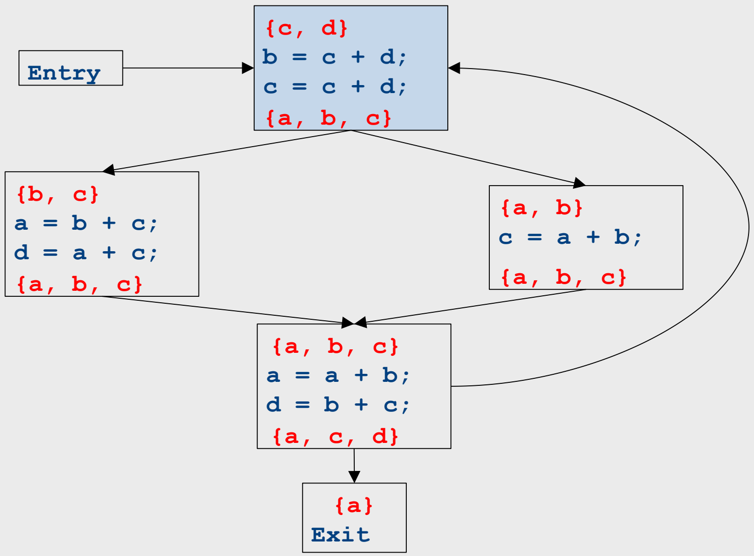
Live variable analysis



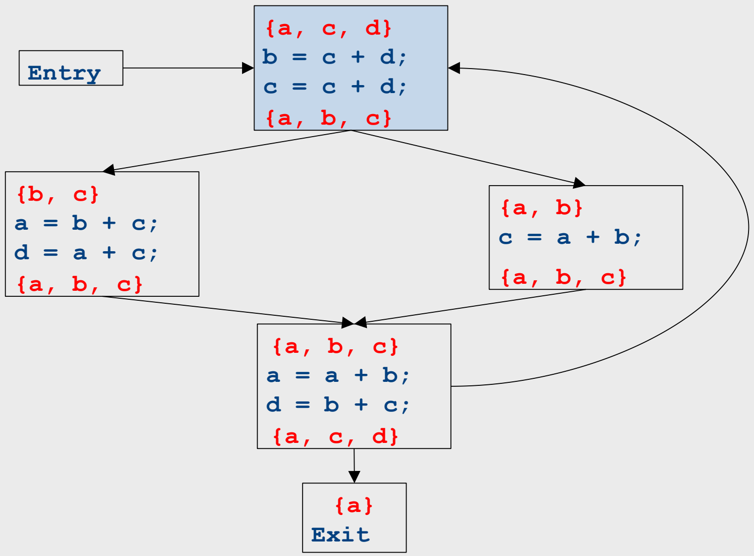
Live variable analysis



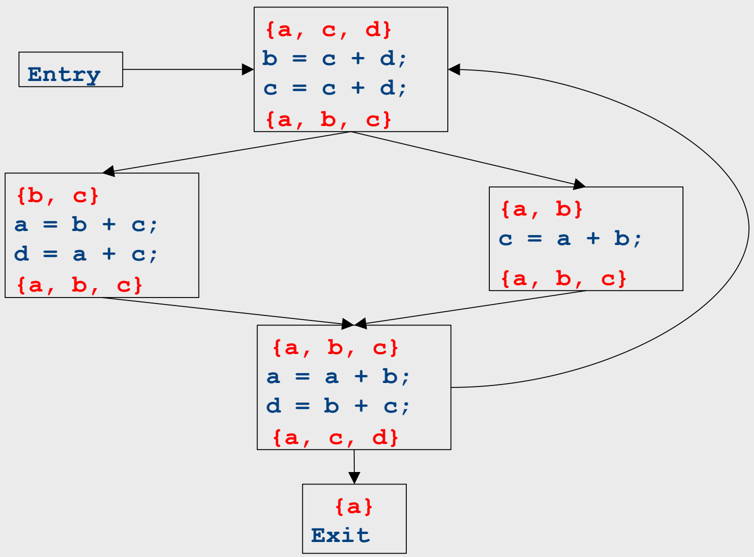
Live variable analysis



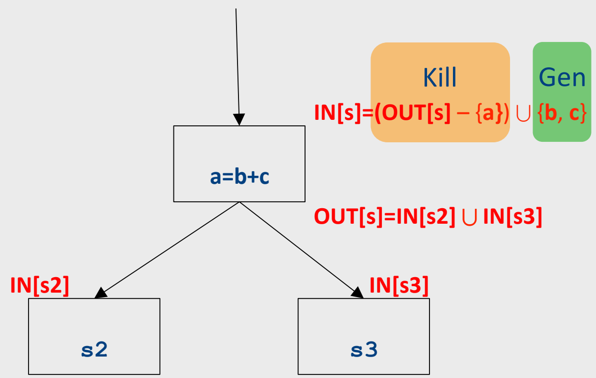
Live variable analysis



Live variable analysis



Global liveness analysis: Kill/Gen



Formalizing data-flow analyses

- Define an analysis of a basic block as a quadruple (D, V, ⊔, F, I) where
  - **D** is a direction (forwards or backwards)
  - **V** is a set of values the program can have at any point
  - ⊔ merging (joining) information
  - **F** is a family of transfer functions defining the meaning of any expression as a function $f : \mathbf{V} \rightarrow \mathbf{V}$
  - I is the initial information at the top (or bottom) of a basic block

Liveness Analysis

- **Direction:** Backward
- **Values:** Sets of variables
- **Transfer functions:** Given a set of variable assignments V and statement a = b + c:
- Remove a from V (any previous value of a is now dead.)
- Add b and c to V (any previous value of b or c is now live.)
- Formally: $V_{in} = (V_{out} \setminus \{\mathbf{a}\}) \cup \{\mathbf{b}, \mathbf{c}\}$
- **Initial value:** Depends on semantics of language
  - E.g., function arguments and return values (pushes)
  - Result of local analysis of other blocks as part of a global analysis
- **Merge:** ⊔ = ∪

Available Expressions

- **Direction:** Forward
- **Values:** Sets of expressions assigned to variables
- **Transfer functions:** Given a set of variable assignments V and statement a = b + c:
  - Remove from V any expression containing a as a subexpression
  - Add to V the expression a = b + c
  - Formally: $V_{out} = (V_{in} \setminus \{e \mid e \text{ contains } \mathbf{a}\}) \cup \{a = b + c\}$
- **Initial value:** Empty set of expressions
- **Merge:** ⊔ = ?

Why does this work? (Live Var.)

- To show correctness, we need to show that
  - The algorithm eventually terminates, and
  - When it terminates, it has a sound answer
- Termination argument:
  - Once a variable is discovered to be live during some point of the analysis, it always stays live
  - Only finitely many variables and finitely many places where a variable can become live
- Soundness argument (sketch):
  - Each individual rule, applied to some set, correctly updates liveness in that set
  - When computing the union of the set of live variables, a variable is only live if it was live on some path leaving the statement

Abstract Interpretation

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

Another view of program analysis

- 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

Another view of program analysis

- We want to reason about some property of the runtime behavior of the program
- Could we run the program and just watch what happens?

Another view of program analysis

- The only way to find out exactly what a program will actually do is to run it
- Problems:
  - The program might not terminate
  - The program might have some behavior we didn't see when we ran it on a particular input

Another view of program analysis

- The only way to find out exactly what a program will actually do is to run it
- Problems:
  - The program might not terminate
  - The program might have some behavior we didn't see when we ran it on a particular input
- Inside a basic block, it is simpler
  - Basic blocks contain no loops
  - There is only one path through the basic block

Another view of program analysis

- The only way to find out exactly what a program will actually do is to run it
- Problems:
  - The program might not terminate
  - The program might have some behavior we didn't see when we ran it on a particular input
- Inside a basic block, it is simpler
  - Basic blocks contain no loops
  - There is only one path through the basic block
  - But still …

Assigning new semantics (Local)

- 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

Assigning new semantics (Global)

- 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
- Merge information from different paths

Abstract Interpretation

- 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
- Merge information from different paths
- Treat the optimizer as an interpreter for these new semantics

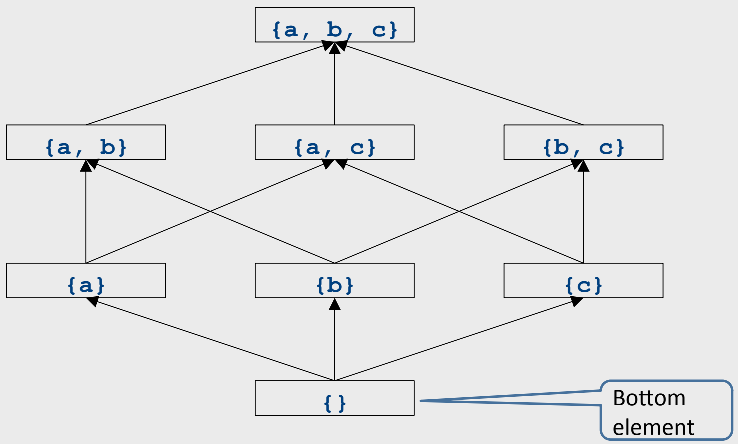
Theory of Program Analysis

- Building up all of the machinery to design this analysis was tricky
- The key ideas, however, are mostly independent of the analysis:
  - We need to be able to compute functions describing the behavior of each statement
  - We need to be able to merge several subcomputations together
  - We need an initial value for all of the basic blocks
- There is a beautiful formalism that captures many of these properties

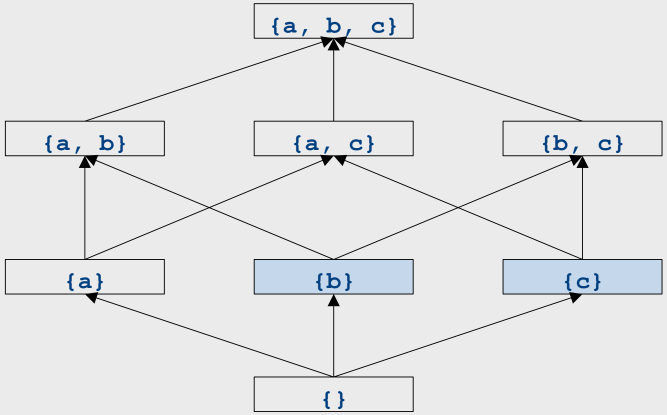
Join semilattices

- A join semilattice is a ordering defined on a set of elements
  - $0 \leq 1 \leq 2 \leq \ldots$
  - $\{\} \leq \{0\} \leq \{0,1\}$, $\{1,2\} \leq \{0,1,2\} \nleq \{1,2,3,4\}$
- 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
  - The join of two elements represents combining (merging) information from two elements by an overapproximation
- The bottom element represents "no information yet" or "the least conservative possible answer"

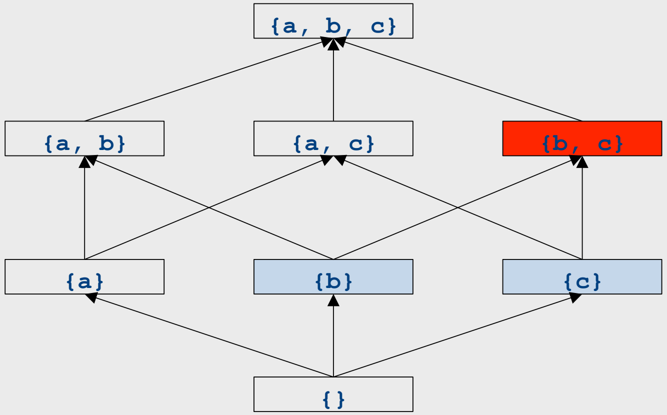
Join semilattice for liveness



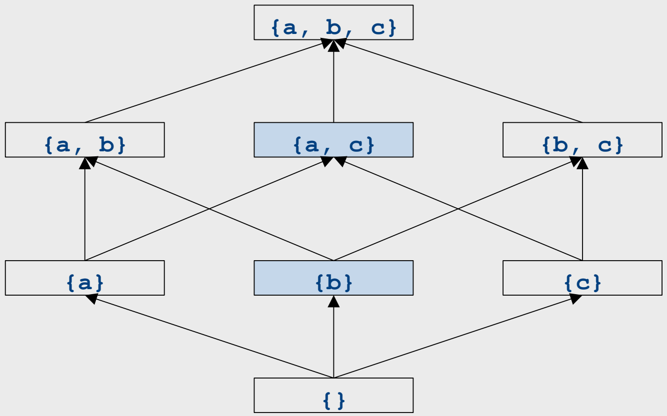
What is the join of {b} and {c}?



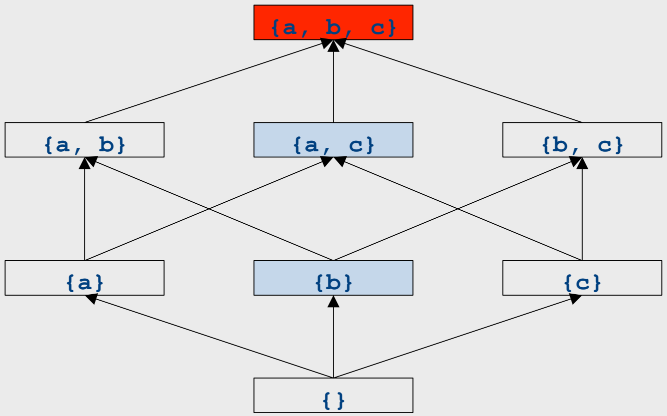
What is the join of {b} and {c}?



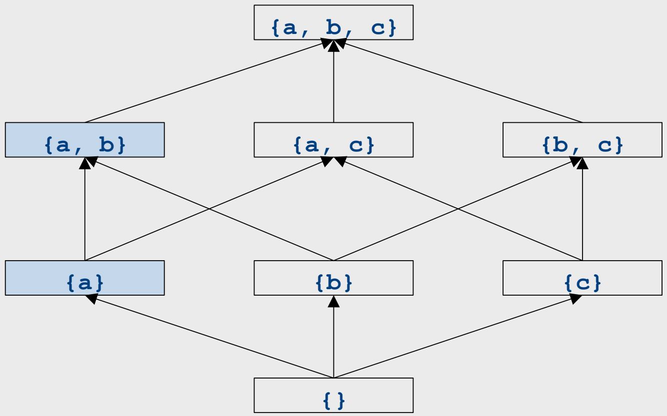
What is the join of {b} and {a,c}?



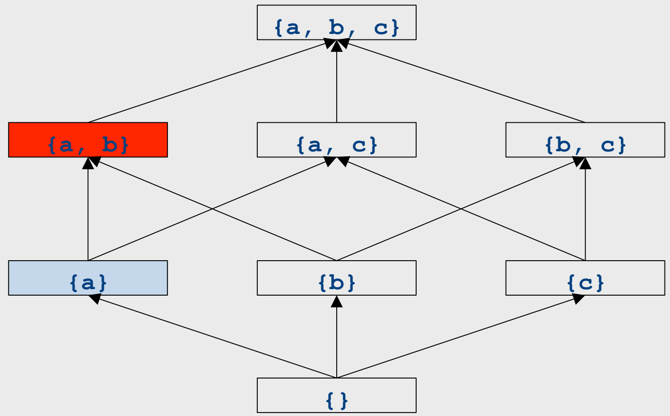
What is the join of {b} and {a,c}?



What is the join of {a} and {a,b}?



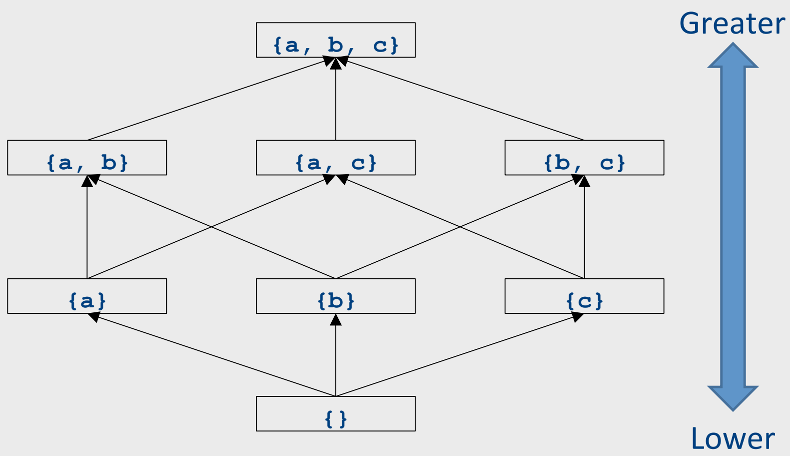
What is the join of {a} and {a,b}?



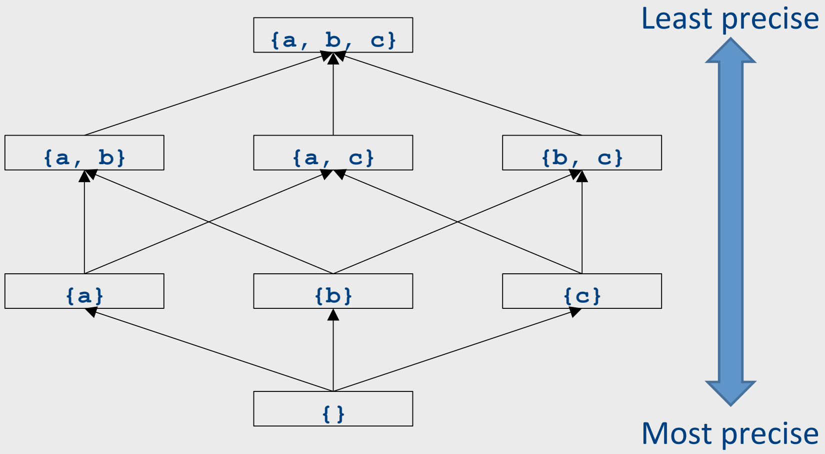
Formal definitions

- 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 ordering



Join semilattices and ordering



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 $x \sqsubseteq z$

An example join semilattice

- The set of natural numbers and the **max** function
- Idempotent
  - **max**{a, a} = a
- Commutative
  - **max**{a, b} = **max**{b, a}
- Associative
  - **max**{a, **max**{b, c}} = **max**{**max**{a, b}, c}
- Bottom element is 0:
  - **max**{0, a} = a
- What is the ordering over these elements?

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: $\emptyset \cup x = x$
- What is the ordering over these elements?

Semilattices and program analysis

- Semilattices naturally solve many of the problems we encounter in global analysis
- How do we combine information from multiple basic blocks?
- What value do we give to basic blocks we haven't seen yet?
- How do we know that the algorithm always terminates?

Semilattices and program analysis

- Semilattices naturally solve many of the problems we encounter in global analysis
- How do we combine information from multiple basic blocks?
  - Take the join of all information from those blocks
- What value do we give to basic blocks we haven't seen yet?
  - Use the bottom element
- How do we know that the algorithm always terminates?
  - Actually, we still don't! More on that later

A general framework

- A global analysis is a tuple (**D, V, ⊔, F, I**), where
  - **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
  - **V** is a set of values
  - ⊔ is a join operator over those values
  - **F** is a set of transfer functions $f : \mathbf{V} \rightarrow \mathbf{V}$
  - **I** is an initial value
- The only difference from local analysis is the introduction of the join operator

Running global analyses

- Assume that (**D, V, ⊔, F, I**) is a forward analysis
- Set OUT[**s**] = ⊥ for all statements **s**
- Set OUT[**entry**] = **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[$\mathbf{p}_1$] ⊔ OUT[$\mathbf{p}_2$] ⊔ … ⊔ 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

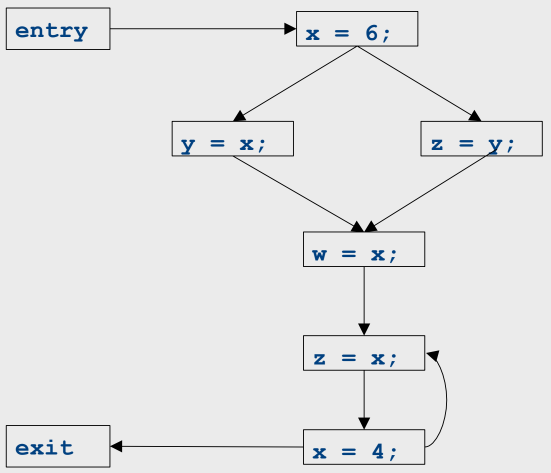
The dataflow framework

- This form of analysis is called the dataflow framework
- Can be used to easily prove an analysis is sound
- With certain restrictions, can be used to prove that an analysis eventually terminates
  - Again, more on that later

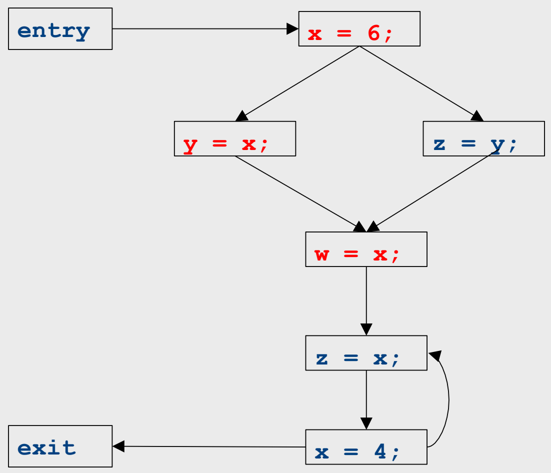
Global constant propagation

- 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

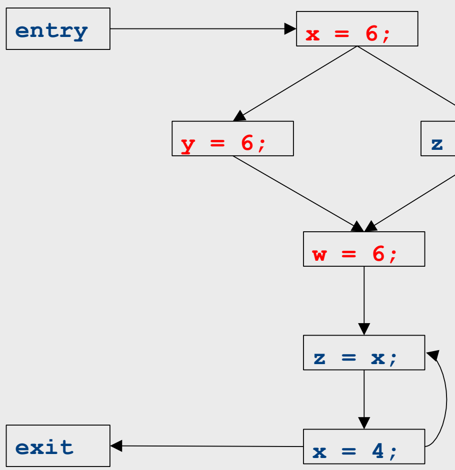
Global constant propagation



Global constant propagation



Global constant propagation



Constant propagation analysis

- In order to do a constant propagation, we need to track what values might be assigned to a variable at each program point
- Every variable will either
  - Never have a value assigned to it,
  - Have a single constant value assigned to it,
  - Have two or more constant values assigned to it, or
  - Have a known non-constant value.
  - Our analysis will propagate this information throughout a CFG to identify locations where a value is constant

Properties of constant propagation

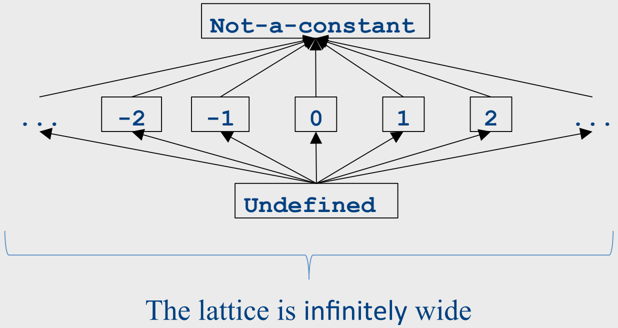
- For now, consider just some single variable **x**
- At each point in the program, we know one of three things about the value of **x**:
  - **x** is definitely not a constant, since it's been assigned two values or assigned a value that we know isn't a constant
  - **x** is definitely a constant and has value **k**
  - We have never seen a value for **x**
- Note that the first and last of these are **not** the same!
  - The first one means that there may be a way for **x** to have multiple values
  - The last one means that **x** never had a value at all

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)

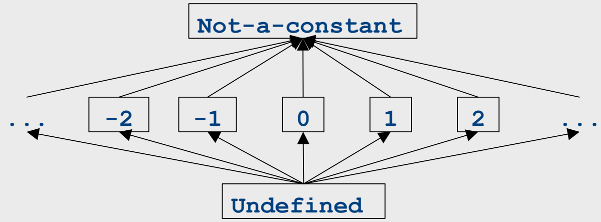
A semilattice for constant propagation

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



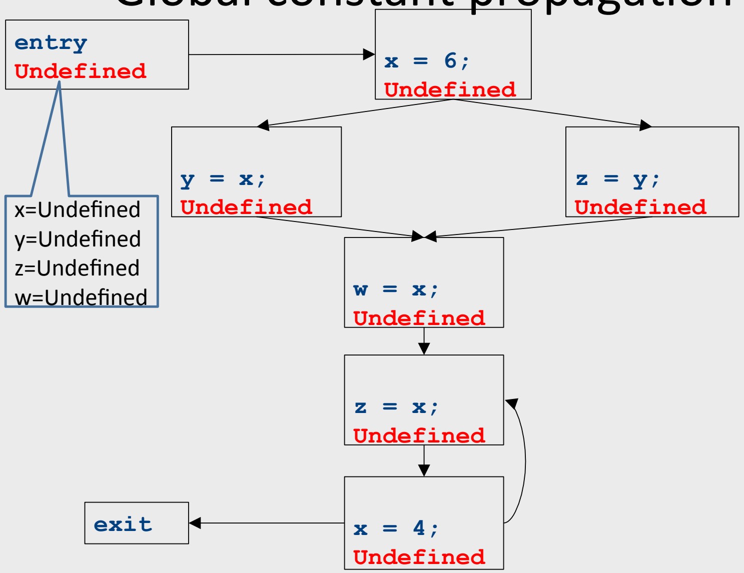
A semilattice for constant propagation

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

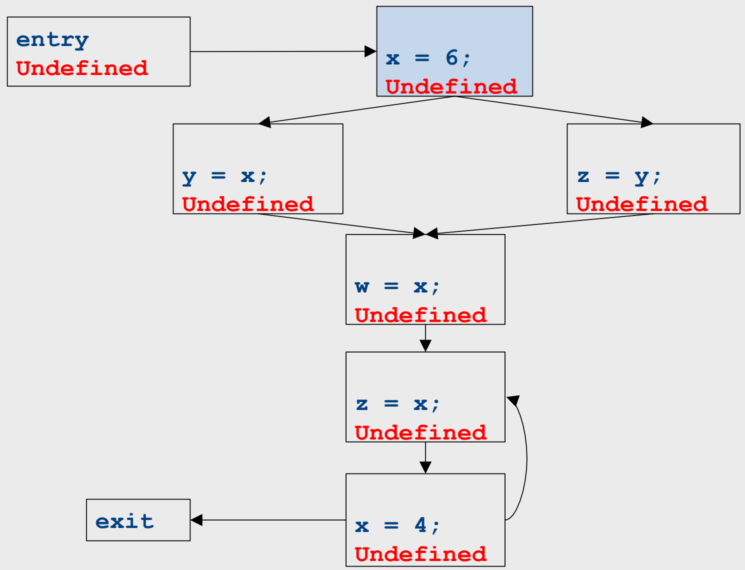


- Note:
  - The join of any two different constants is **Not-a-Constant**
  - The join of **Not a Constant** and any other value is **Not-a-Constant**
  - The join of **Undefined** and any other value is that other value

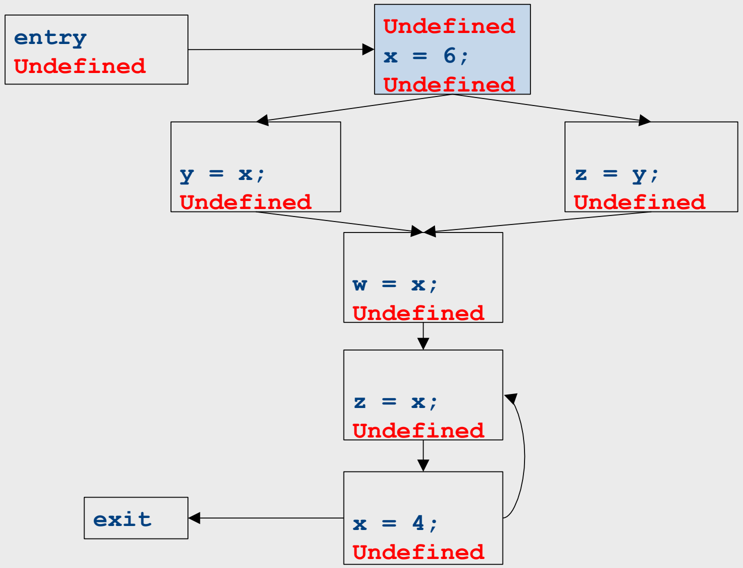
Global constant propagation



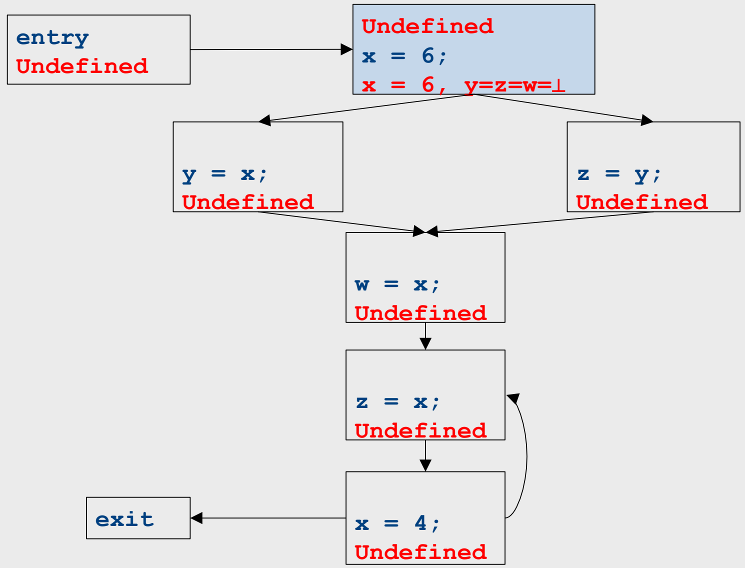
Global constant propagation



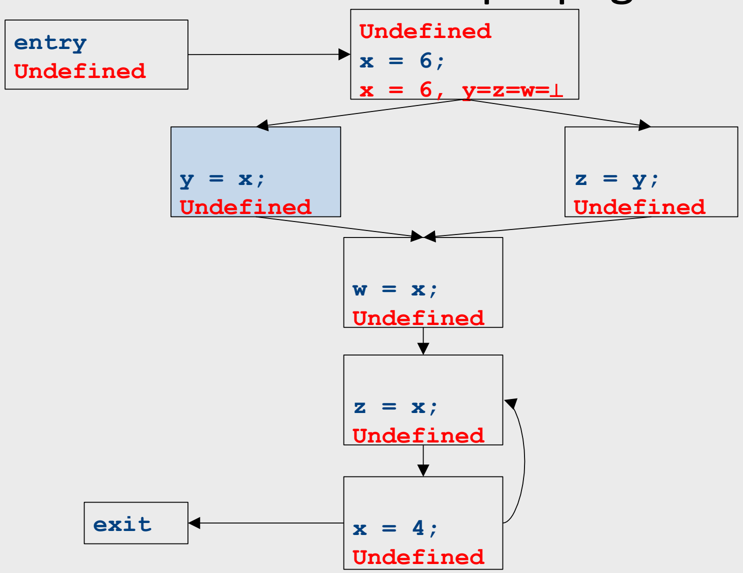
Global constant propagation



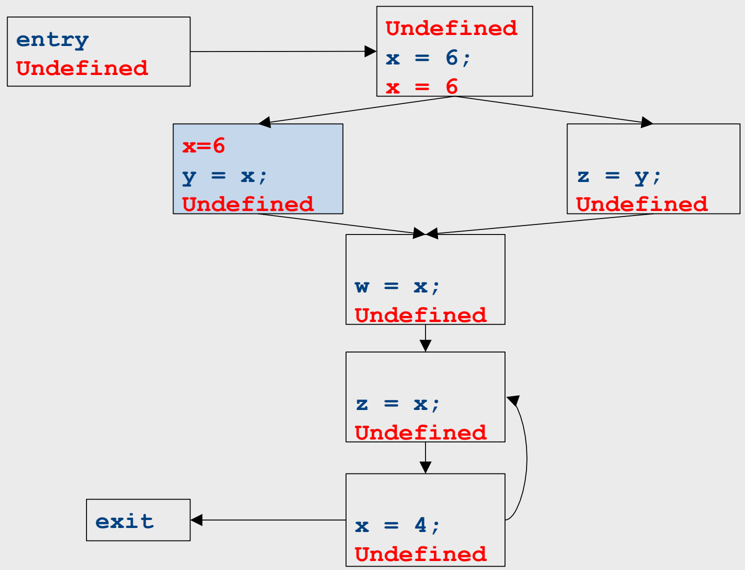
Global constant propagation



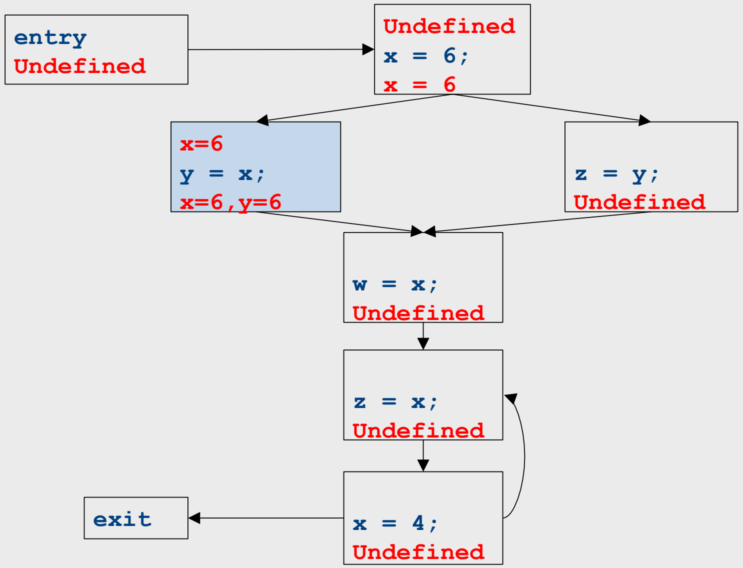
Global constant propagation



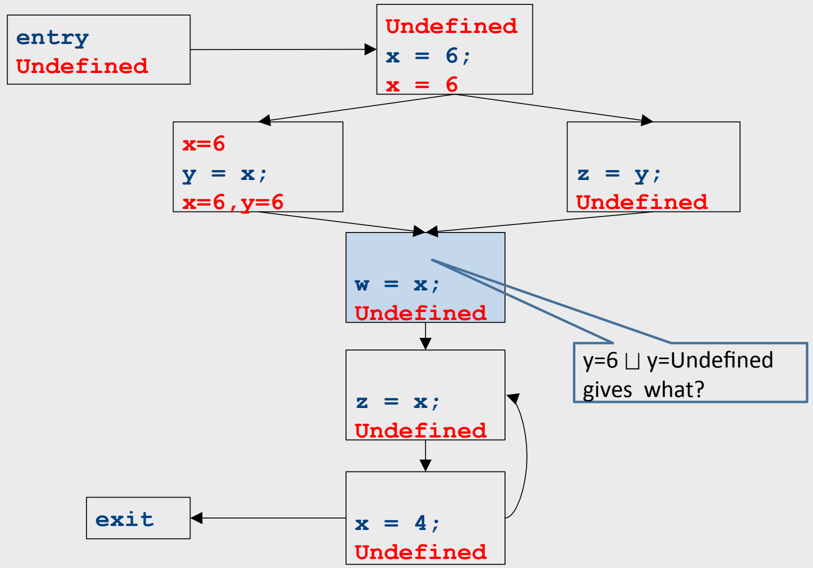
Global constant propagation



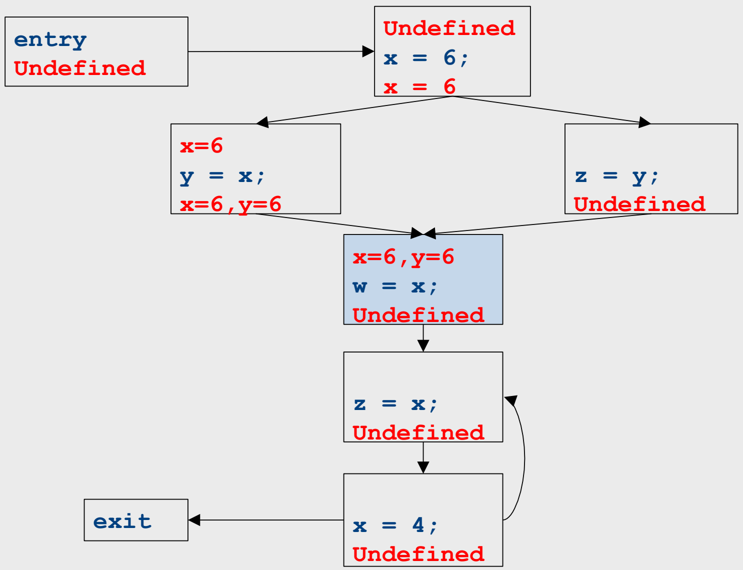
Global constant propagation



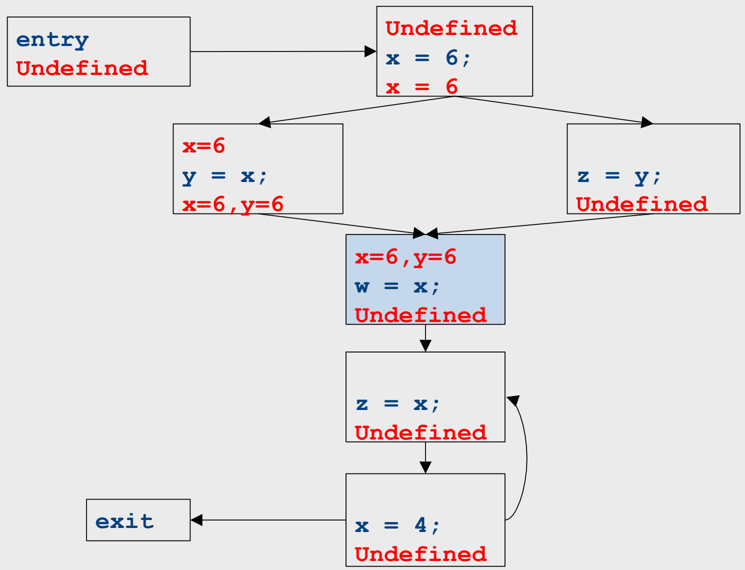
Global constant propagation



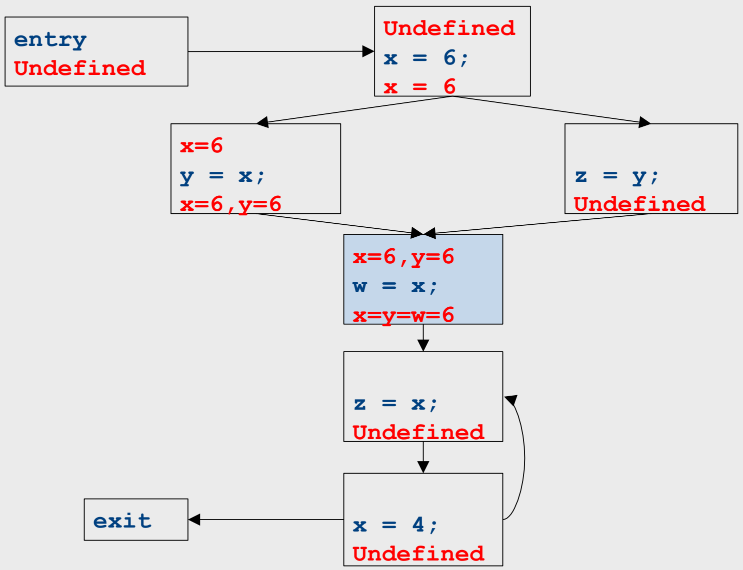
Global constant propagation



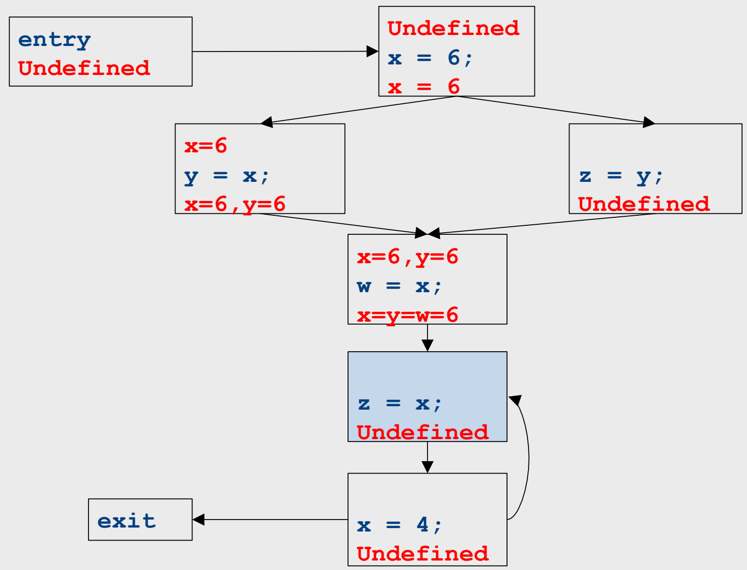
Global constant propagation



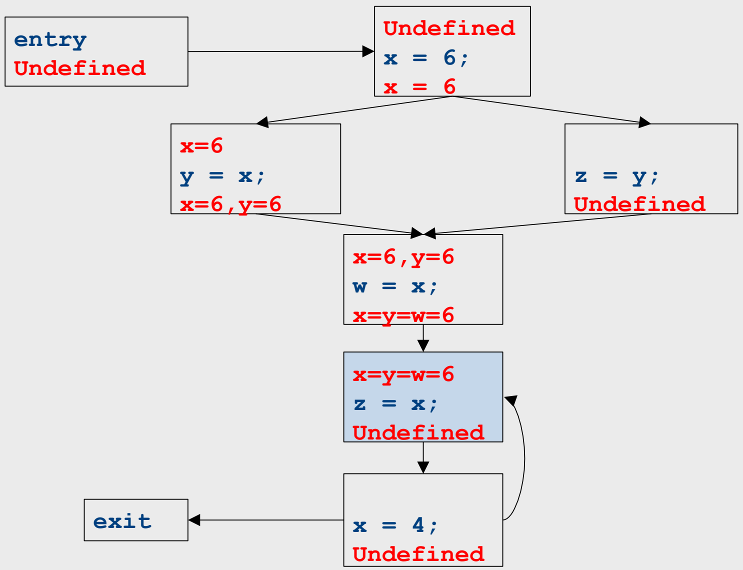
Global constant propagation



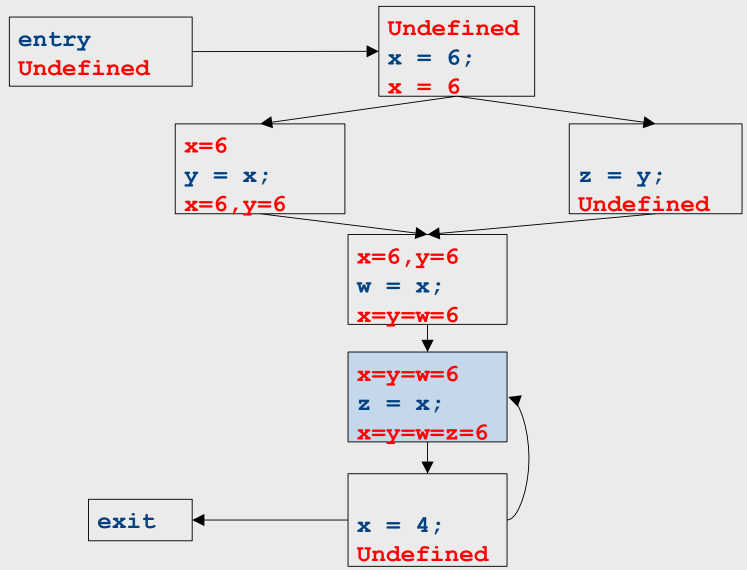
Global constant propagation



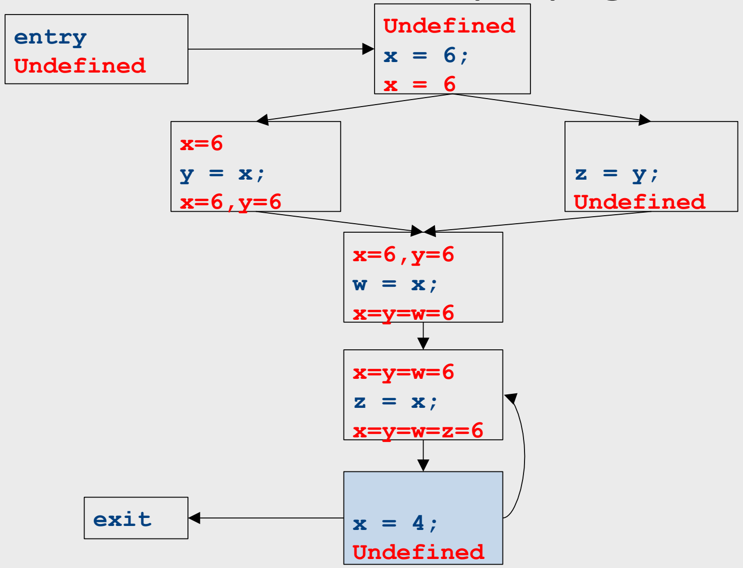
Global constant propagation



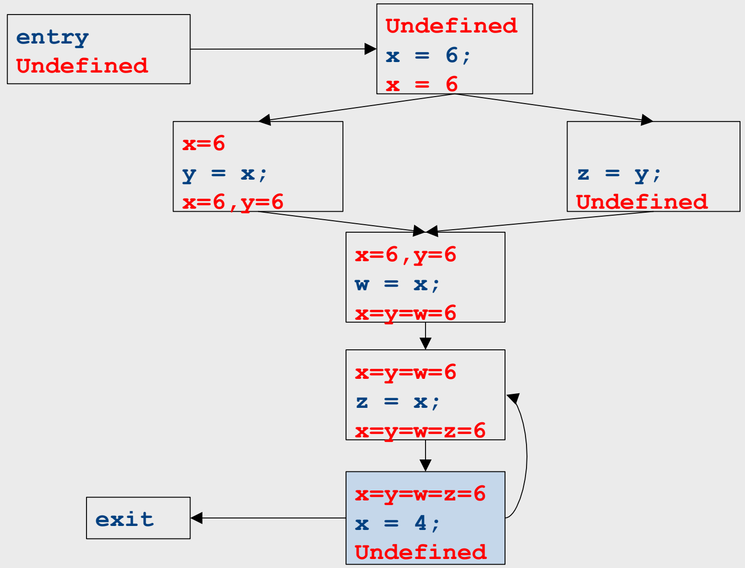
Global constant propagation



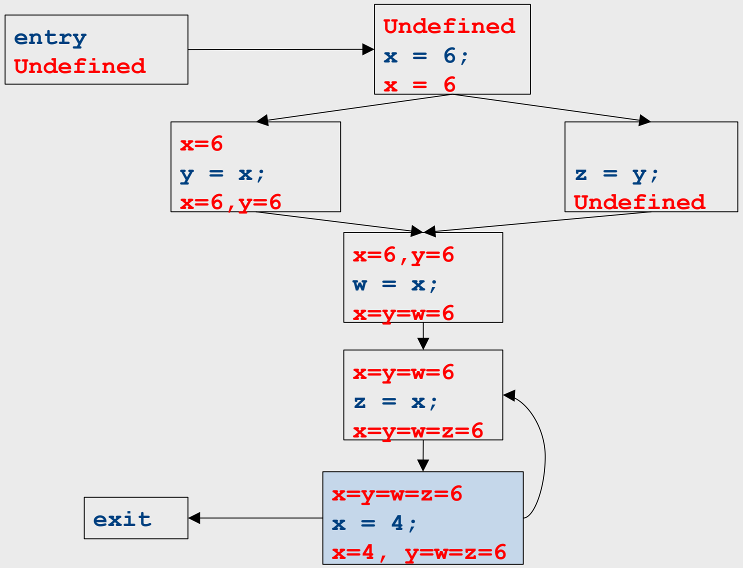
Global constant propagation



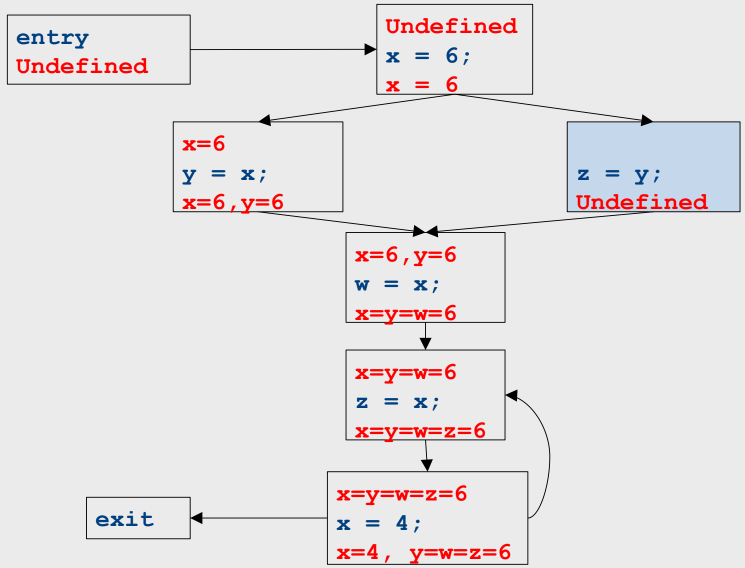
Global constant propagation



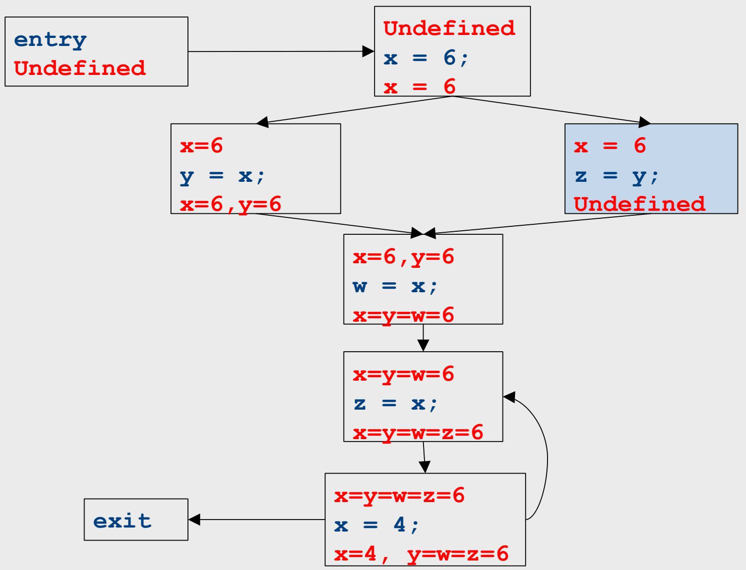
Global constant propagation



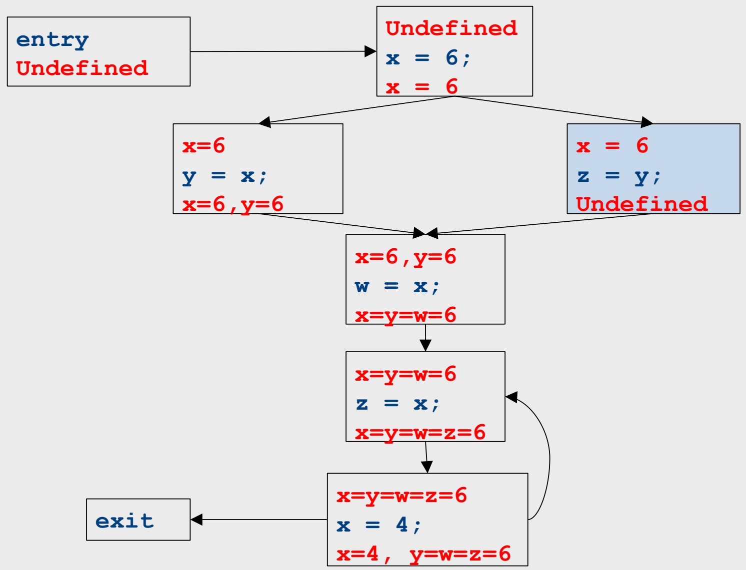
Global constant propagation



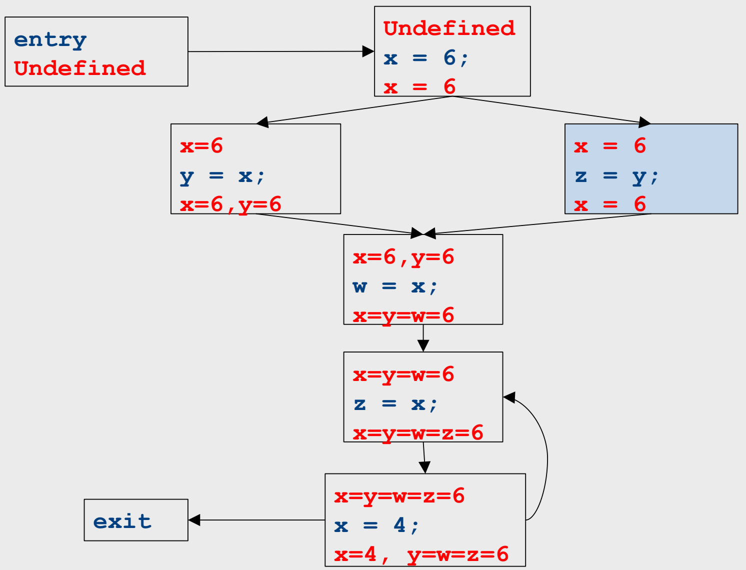
Global constant propagation



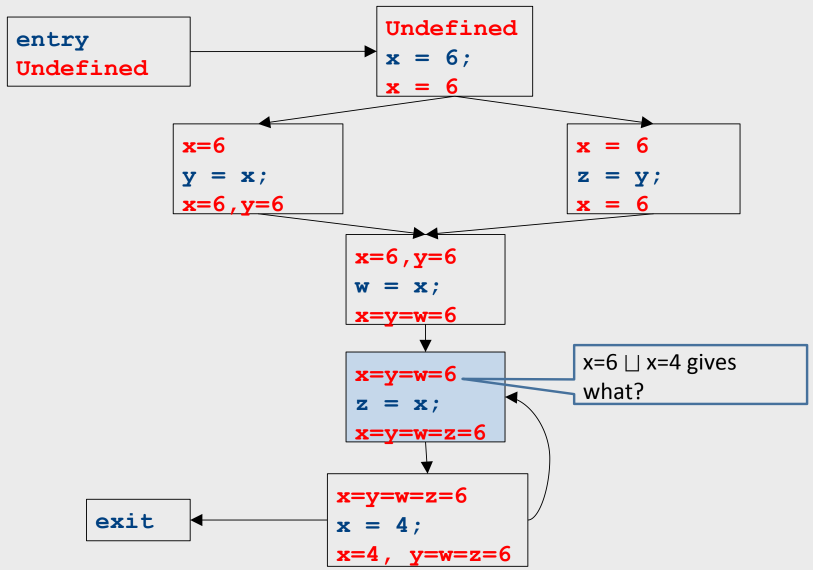
Global constant propagation



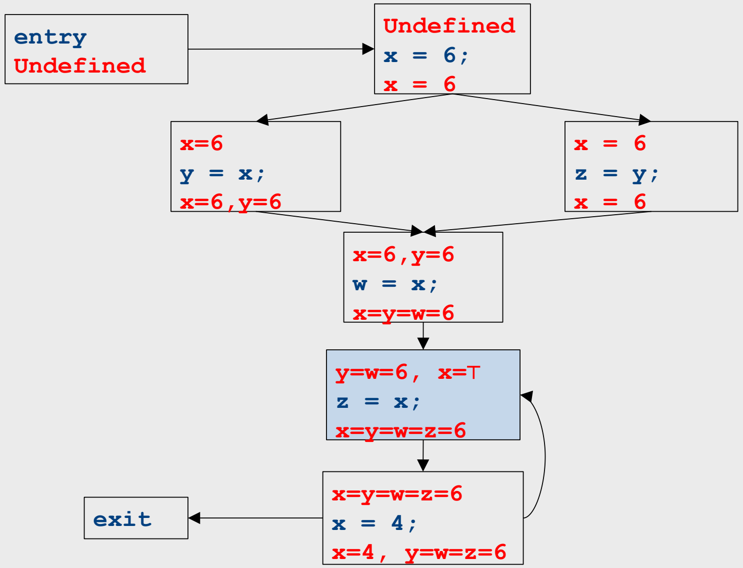
Global constant propagation



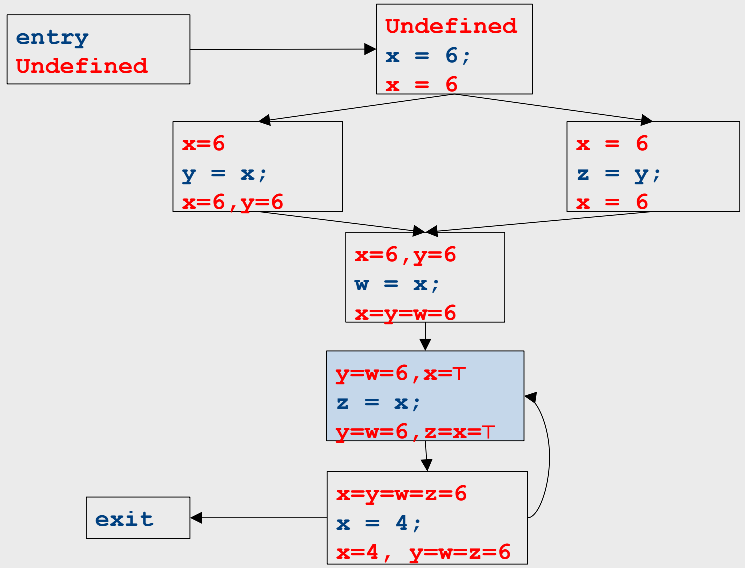
Global constant propagation



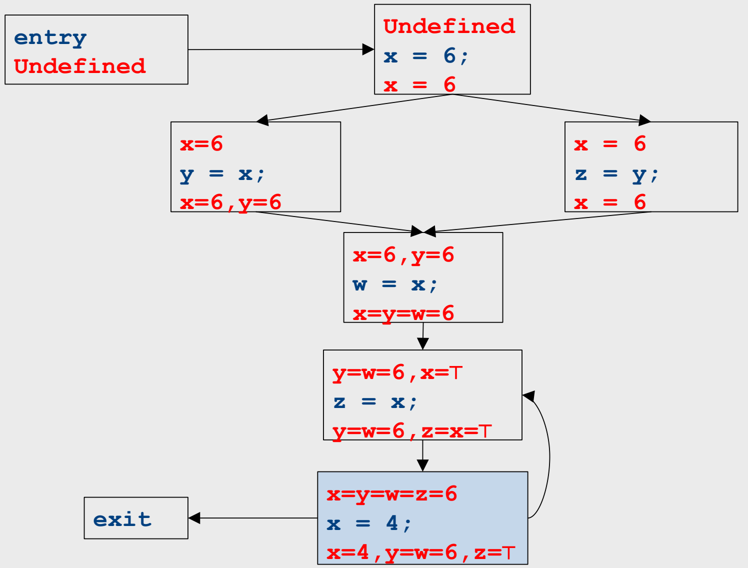
Global constant propagation



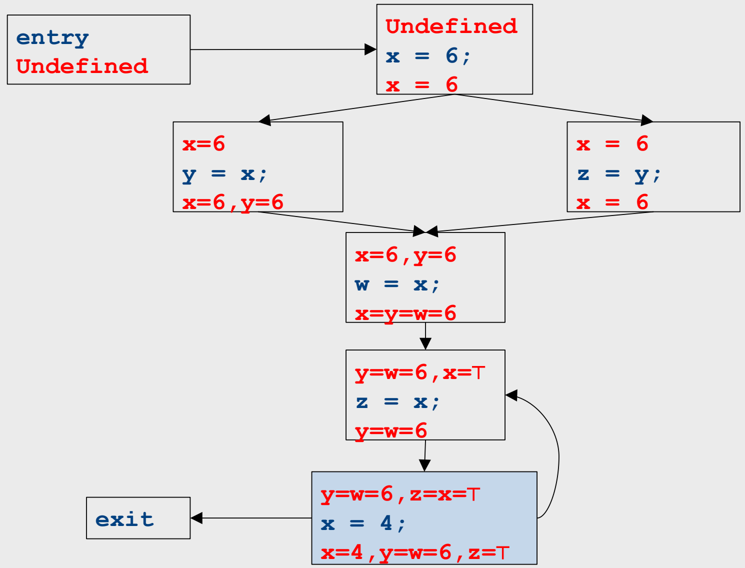
Global constant propagation



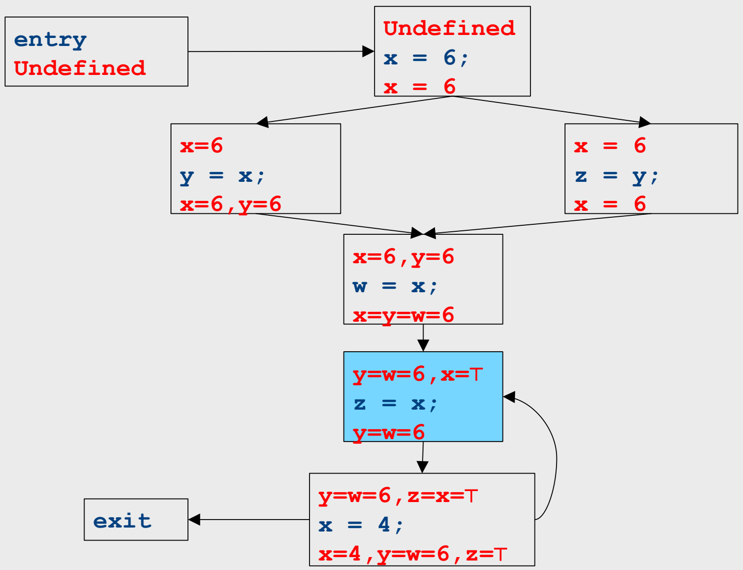
Global constant propagation



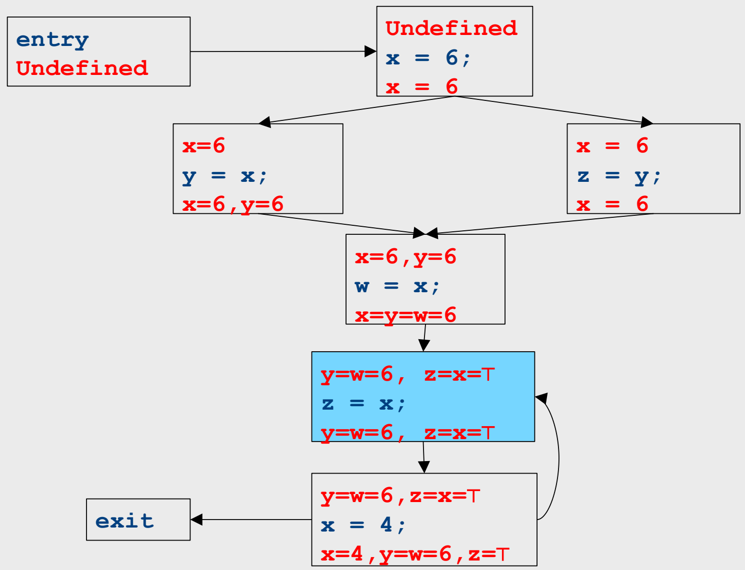
Global constant propagation



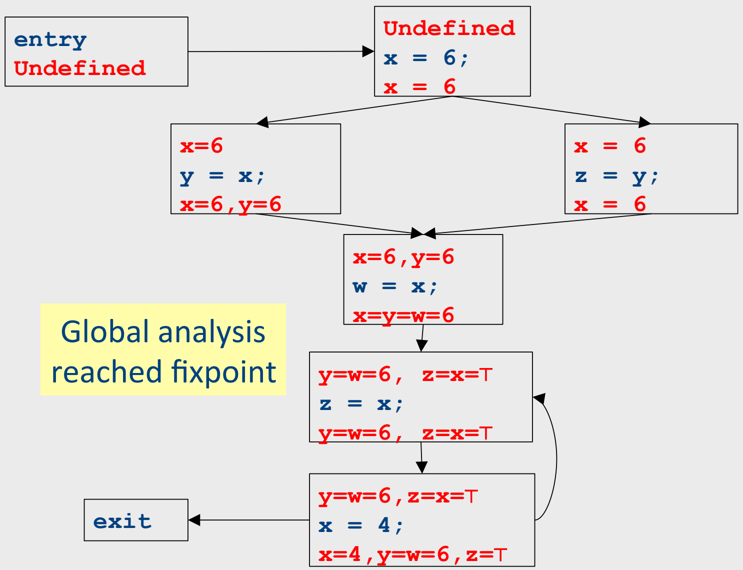
Global constant propagation



Global constant propagation



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↦1,y↦1,z↦1} ⊔ {x↦1,y↦2,z↦Not-a-Constant} =
    {x↦1,y↦Not-a-Constant,z↦Not-a-Constant}
- Transfer functions:
  - $f_{\mathbf{x=k}}(V) = V|_{x\mapsto k}$ *(update V by mapping x to k)*
  - $f_{\mathbf{x=a+b}}(V) = V|_{x\mapsto ?}$
- Initial value: **x is Undefined**
  - (When might we use some other value?)

Dataflow for constant propagation

- Direction: **Forward**
- Semilattice: Vars$\rightarrow$ {Undefined, 0, 1, -1, 2, -2, …, Not-a-Constant}
  - Join mapping for variables point-wise
    {x↦1,y↦1,z↦1} ⊔ {x↦1,y↦2,z↦Not-a-Constant} = {x↦1,y↦Not-a-Constant,z↦Not-a-Constant}
- Transfer functions:
  - $f_{x=k}(V) = V|_{x \mapsto k}$ *(update V by mapping x to k)*
  - $f_{x=a+b}(V) = V|_{x \mapsto ?}$
- 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 $\bot$ such that $\bot \sqcup x = x$ for all x

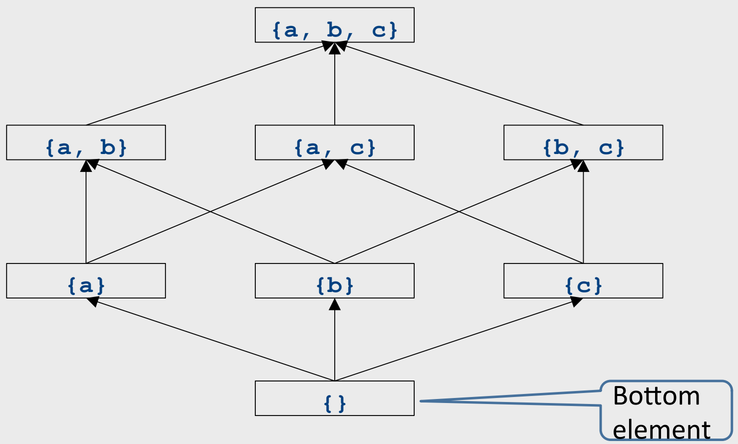
Partial ordering induced by join

- Every join semilattice $(V, \sqcup)$ induces an ordering relationship $\sqsubseteq$ over its elements
- Define $x \sqsubseteq y$ iff $x \sqcup y = y$
- 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 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



Dataflow framework

- A global analysis is a tuple (**D, V, ⊔, F, I**), where
  - **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
  - **V** is a set of values (sometimes called domain)
  - ⊔ is a join operator over those values
  - **F** is a set of transfer functions $f_s : \mathbf{V} \to \mathbf{V}$ (for every statement s)
  - **I** is an initial value

Running global analyses

- Assume that (**D, V, ⊔, F, I**) is a forward analysis
- For every statement s maintain values before - IN[s] - and after - OUT[s]
- Set OUT[**s**] = ⊥ for all statements **s**
- Set OUT[**entry**] = **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[$\mathbf{p_1}$] ⊔ OUT[$\mathbf{p_2}$] ⊔ … ⊔ OUT[$\mathbf{p_n}$]
    - Set OUT[**s**] = f_s(IN[**s**])
- The order of this iteration does not matter
  - Chaotic iteration

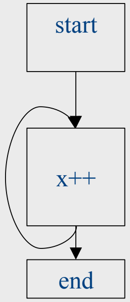
Proving termination

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

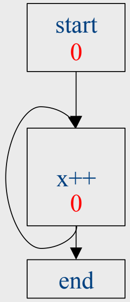
A non-terminating analysis

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

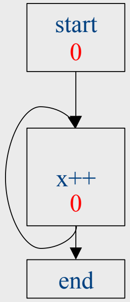
A non-terminating analysis



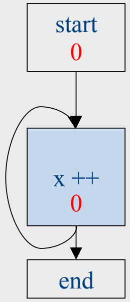
Initialization



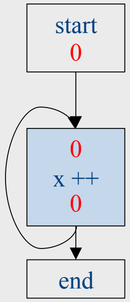
Fixed-point iteration



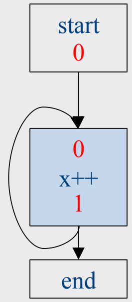
Choose a block



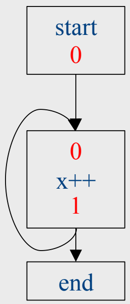
Iteration 1



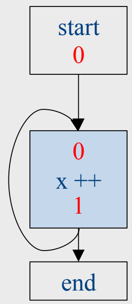
Iteration 1



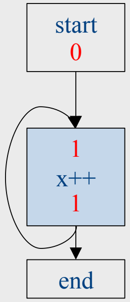
Choose a block



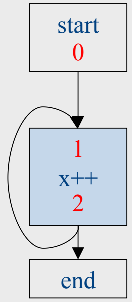
Iteration 2



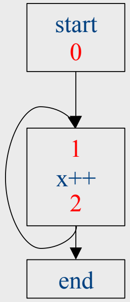
Iteration 2



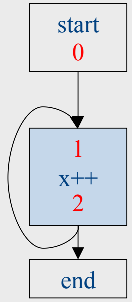
Iteration 2



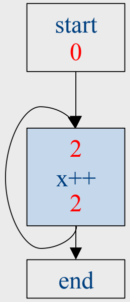
Choose a block



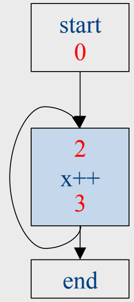
Iteration 3



Iteration 3

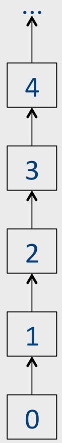


Iteration 3



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



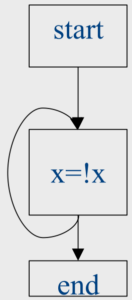
Height of a lattice

- An increasing chain is a sequence of elements $\bot \sqsubset a_1 \sqsubset a_2 \sqsubset \ldots \sqsubset a_k$
  - 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:
  - $\{\} \subset \{v_1\} \subset \{v_1,v_2\} \subset \ldots \subset \{v_1,\ldots,v_n\}$
- For available expressions it is the number of expressions of the form a=b op c
  - For *n* program variables and *m* operator types: $m \cdot n^3$

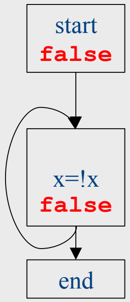
Another non-terminating analysis

- 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`**

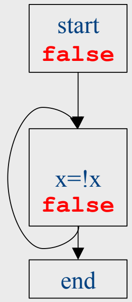
A non-terminating analysis



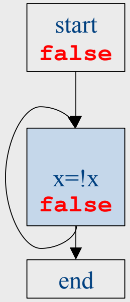
Initialization



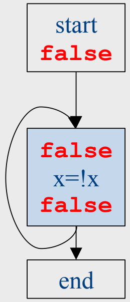
Fixed-point iteration



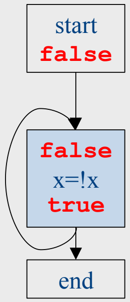
Choose a block



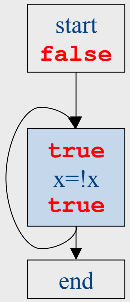
Iteration 1



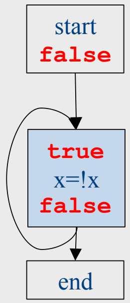
Iteration 1



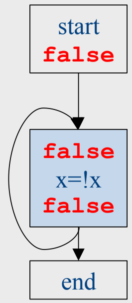
Iteration 2



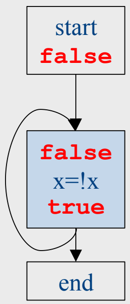
Iteration 2



Iteration 3

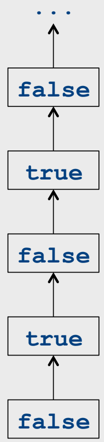


Iteration 3



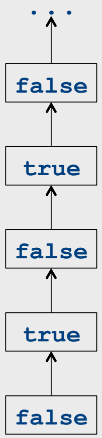
Why doesn't it terminate?

- 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

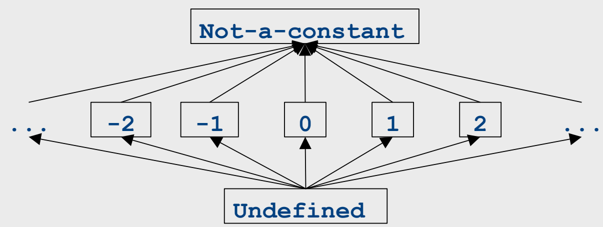
- A transfer function f is monotone iff
  if $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$
- Intuitively, if you know less information about a program point, you can't "gain back" more information about that program point
- Many transfer functions are monotone, including those for liveness and constant 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
  - $f_{\mathbf{x=k}}(V) = V|_{x \mapsto k}$ *(update V by mapping x to k)*
  - $f_{\mathbf{x=a+b}}(V) = V|_{x \mapsto \text{Not-a-Constant}}$ *(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)

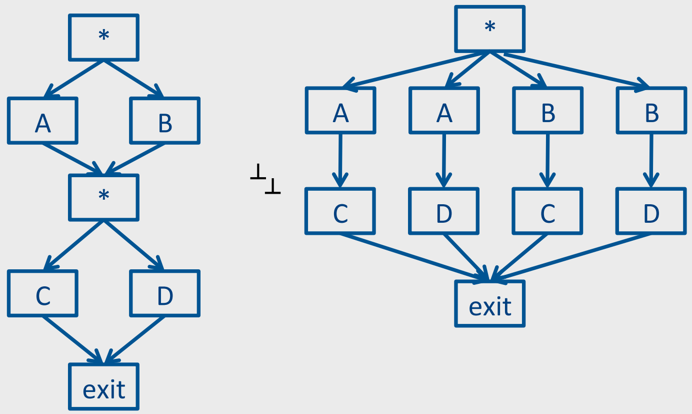
An "optimality" result

- A transfer function *f* is distributive if
  $f(a \sqcup b) = f(a) \sqcup f(b)$
  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

If(*)
A
else
B;

If(*)
C
else
D;



$$f_C(f_A(\bot) \sqcup f_B(\bot)) \sqcup f_D(f_A(\bot) \sqcup f_B(\bot)) \overset{?}{=} f_C(f_A(\bot)) \sqcup f_D(f_A(\bot)) \sqcup f_C(f_B(\bot)) \sqcup f_D(f_B(\bot))$$

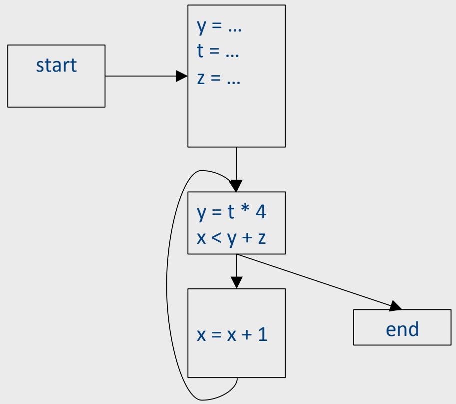
An "optimality" result

- 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?

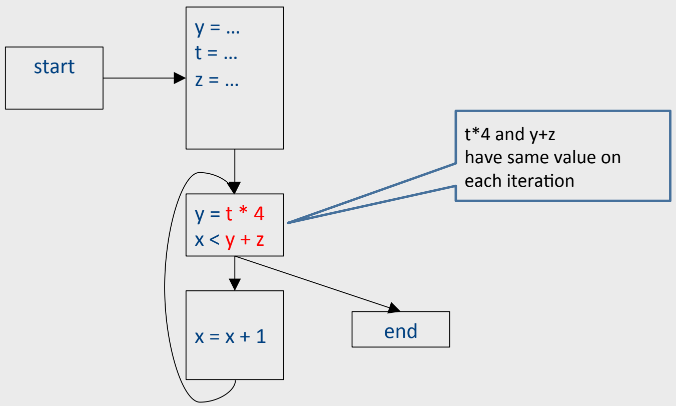
Loop optimizations

- 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)

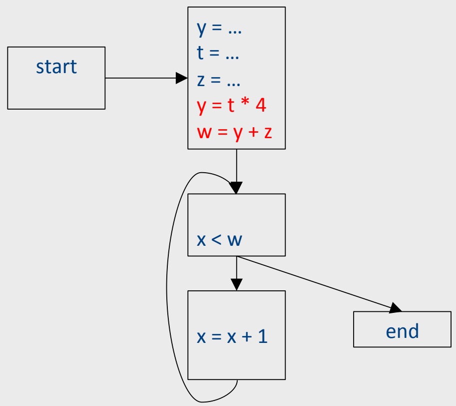
Loop invariant computation



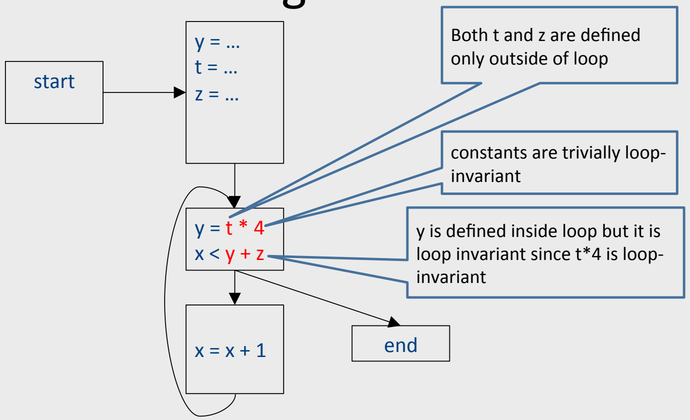
Loop invariant computation



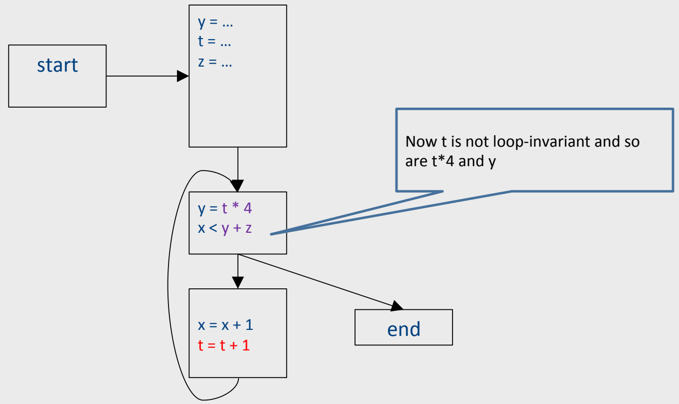
Code hoisting



What reasoning did we use?



What about now?



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

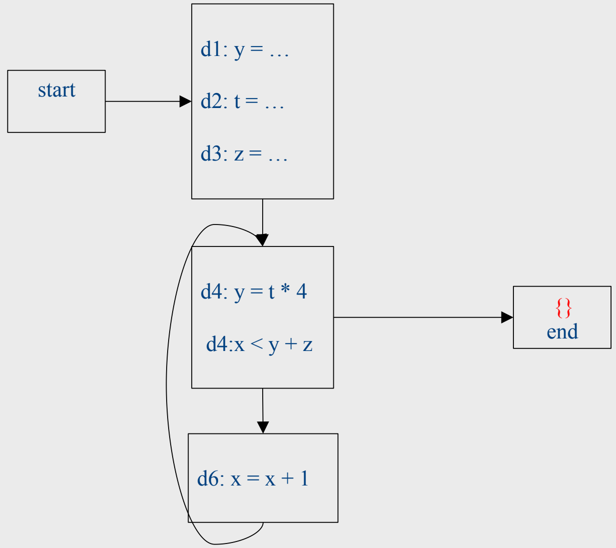
- A definition *d*: t = … reaches a program location if there is a path from the definition to the program location, along which the defined variable is never redefined
- Direction: Forward
- Domain: sets of program locations that are definitions `
- Join operator: union
- Transfer function:

  $f_{d:\ a=b\ op\ c}(\mathrm{RD}) = (\mathrm{RD} - \mathit{defs}(a)) \cup \{d\}$

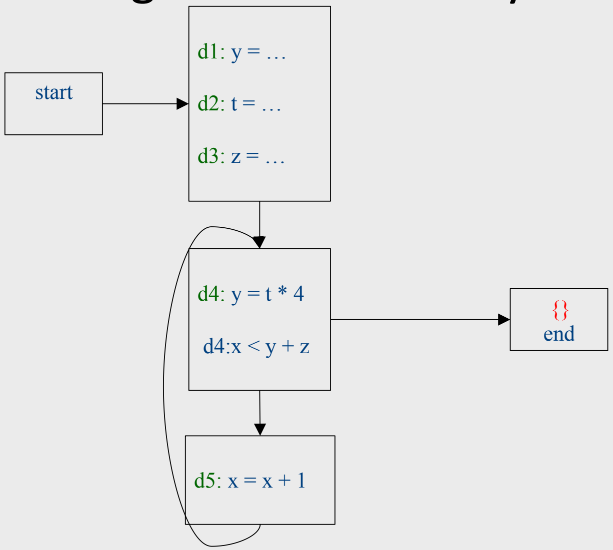
  $f_{d:\ not\text{-}a\text{-}def}(\mathrm{RD}) = \mathrm{RD}$

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

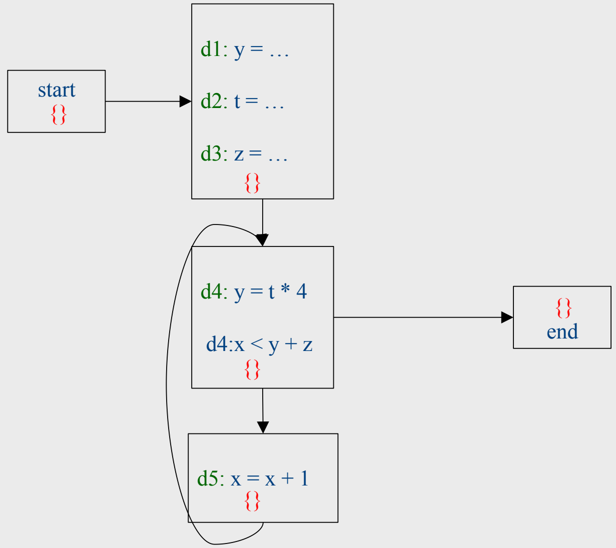
Reaching definitions analysis



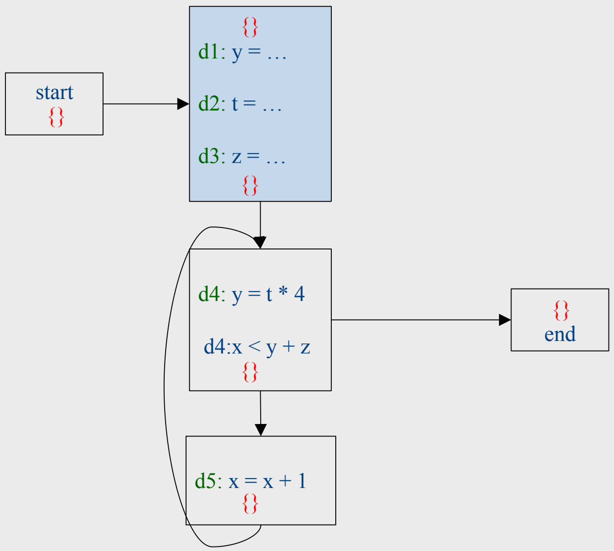
Reaching definitions analysis



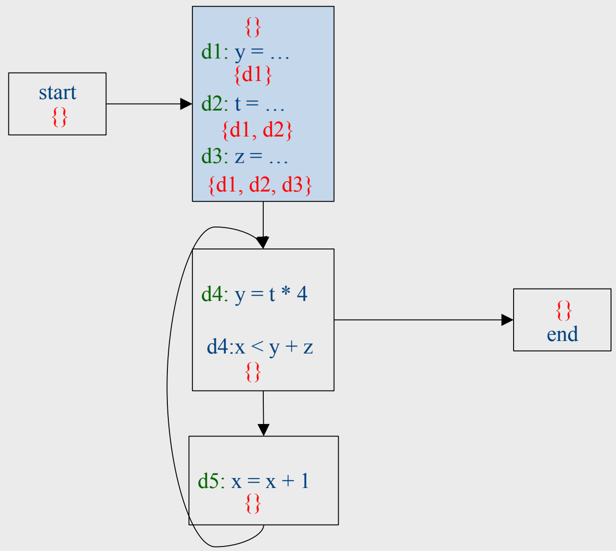
Initialization



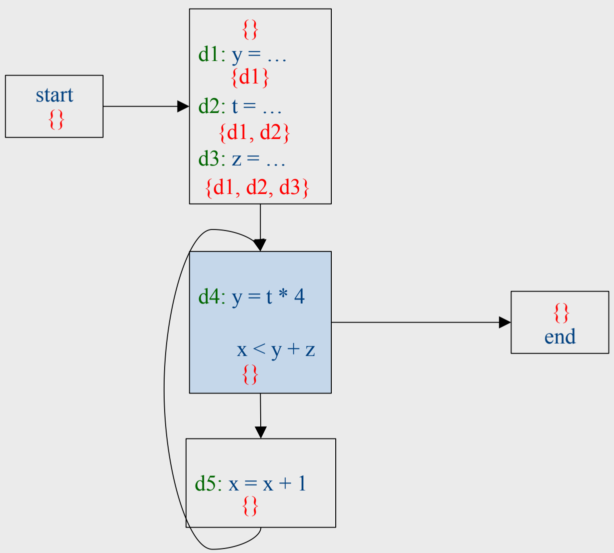
Iteration 1



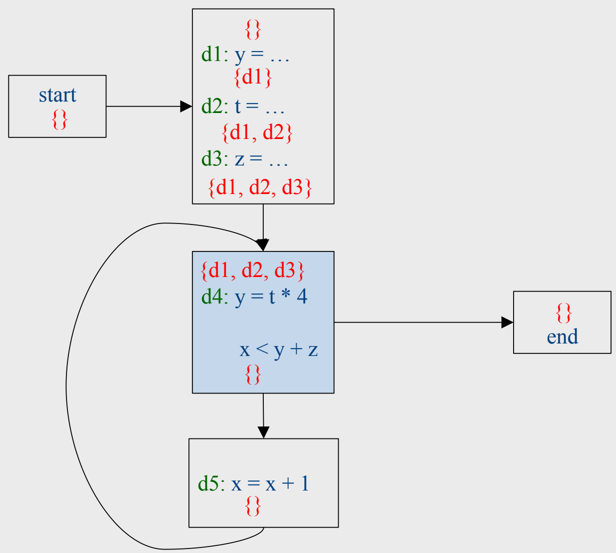
Iteration 1



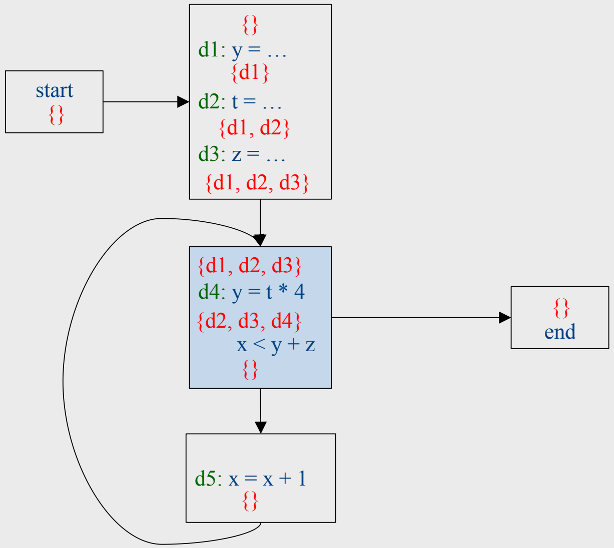
Iteration 2



Iteration 2

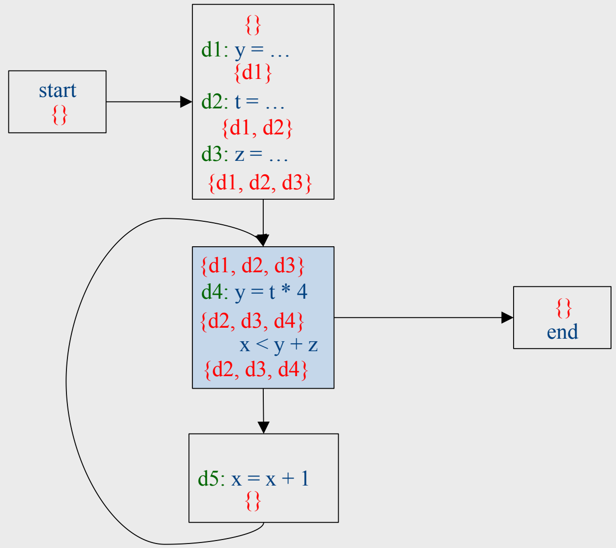


Iteration 2



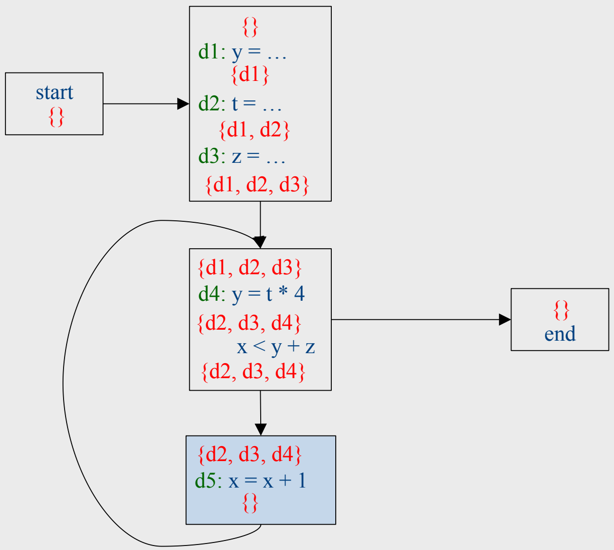
173

Iteration 2



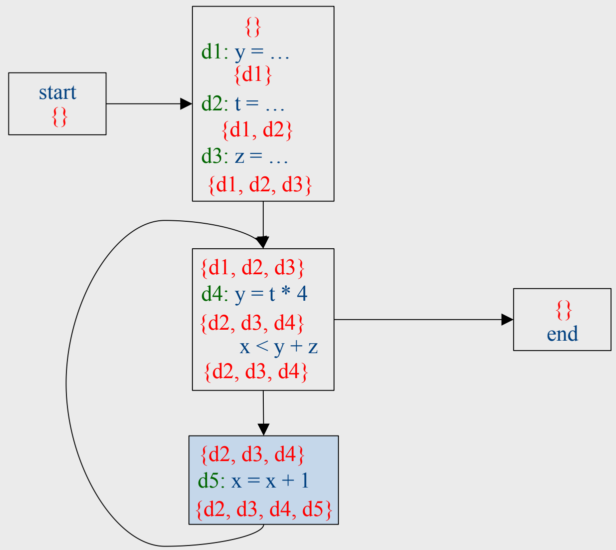
174

Iteration 3



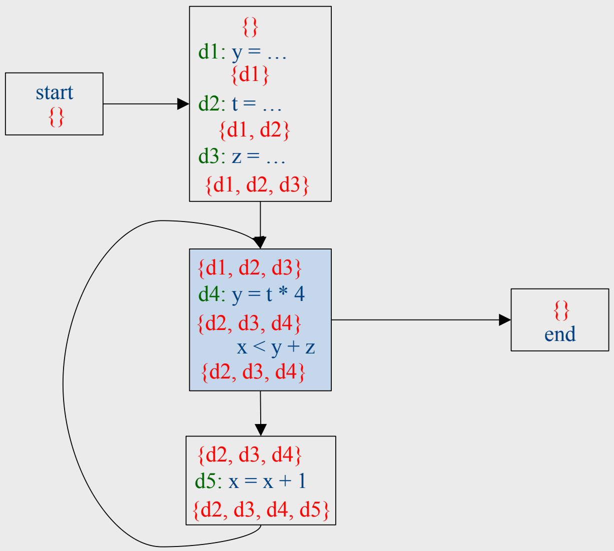
175

Iteration 3

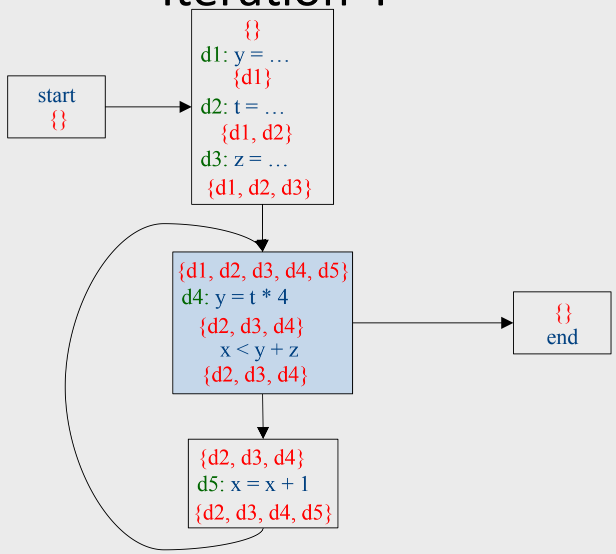


176

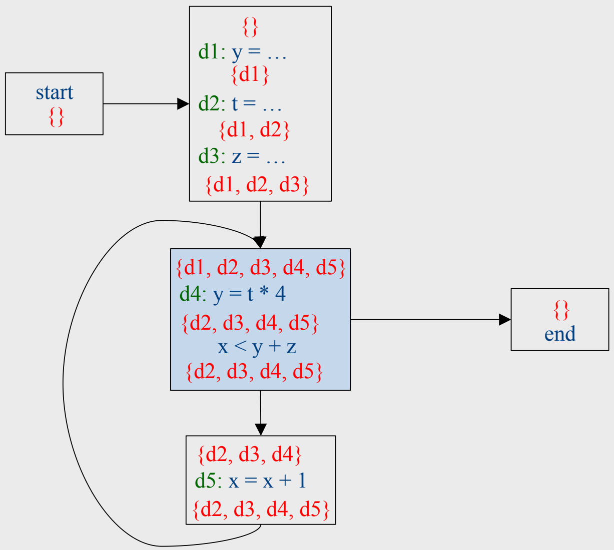
Iteration 4



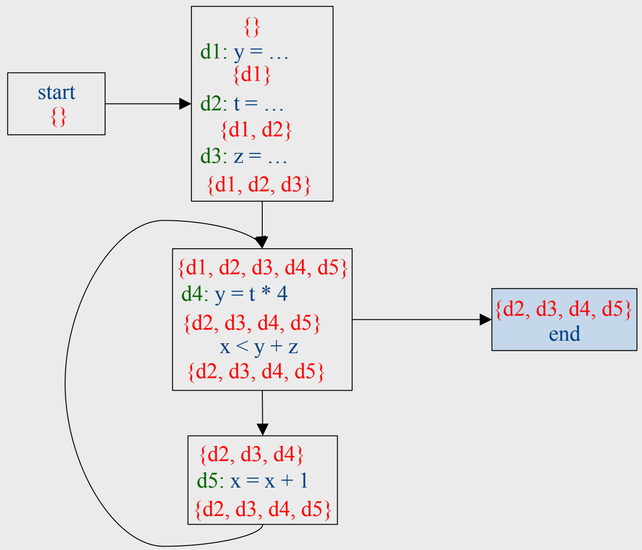
Iteration 4



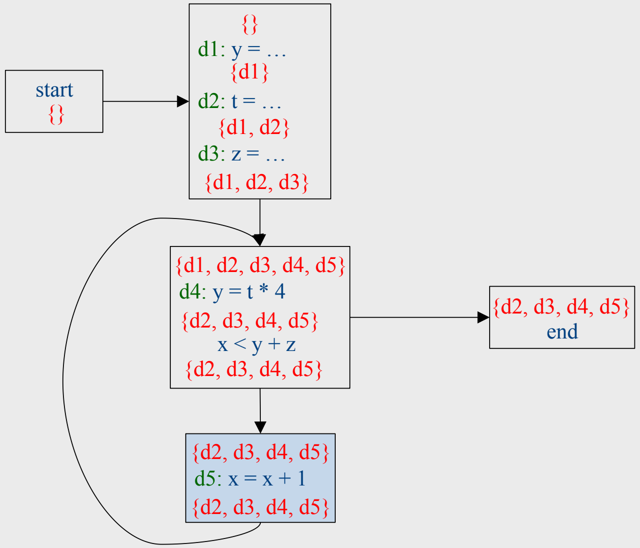
Iteration 4



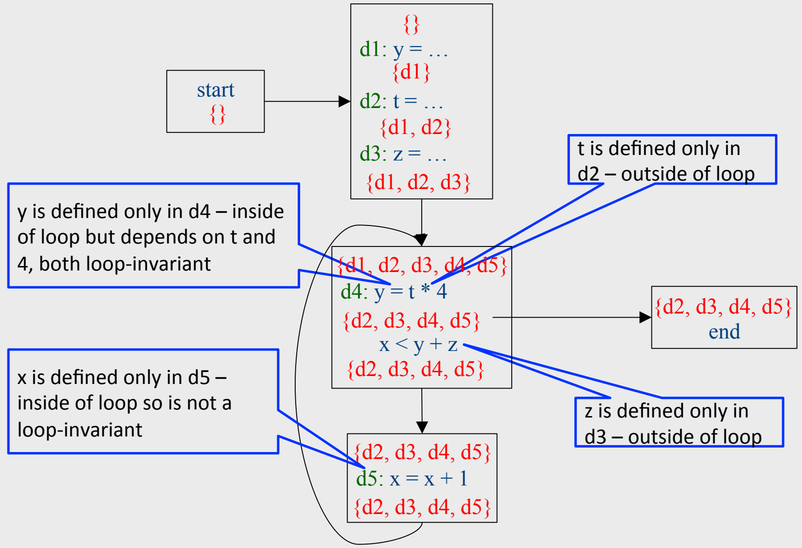
Iteration 5



Iteration 6



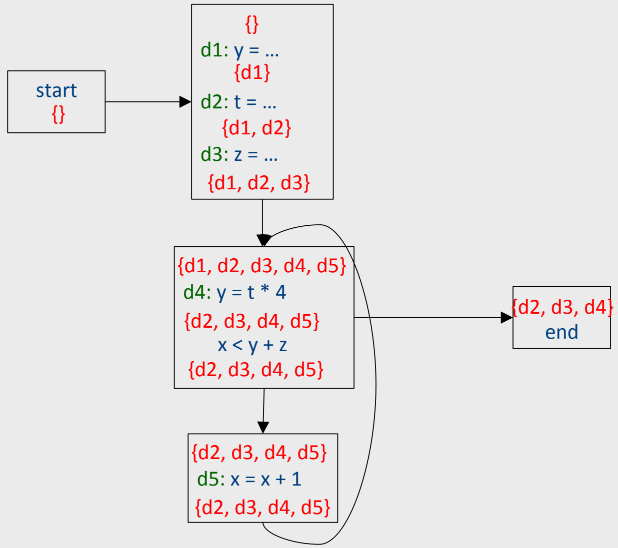
Which expressions are loop invariant?



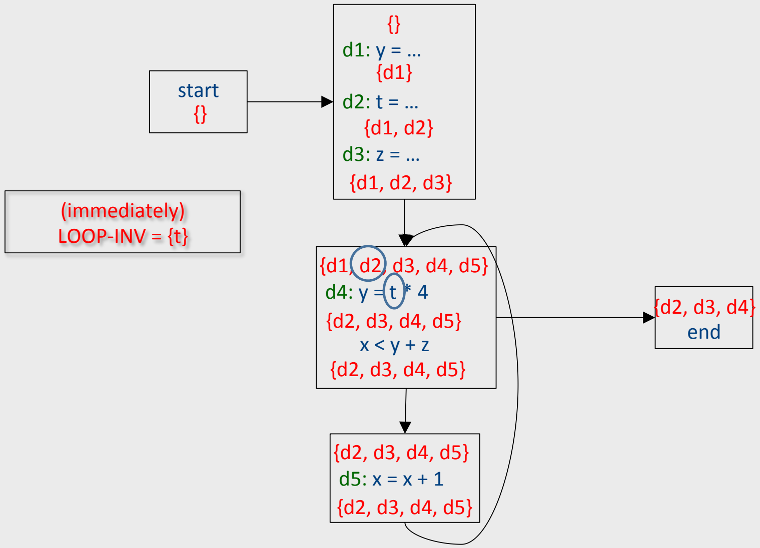
Inferring loop-invariant expressions

- 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

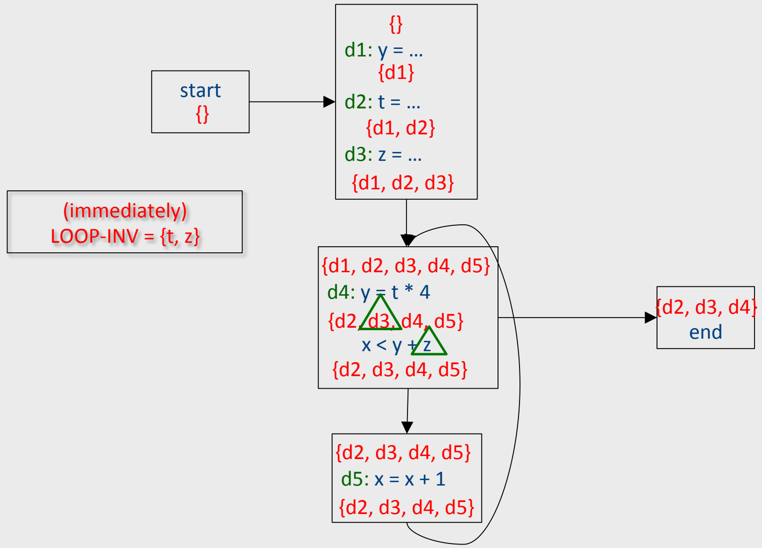
Computing LOOP-INV



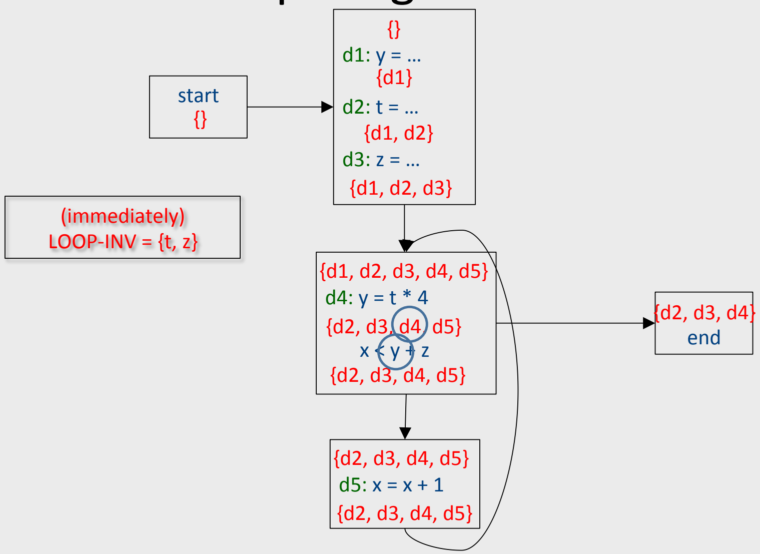
Computing LOOP-INV



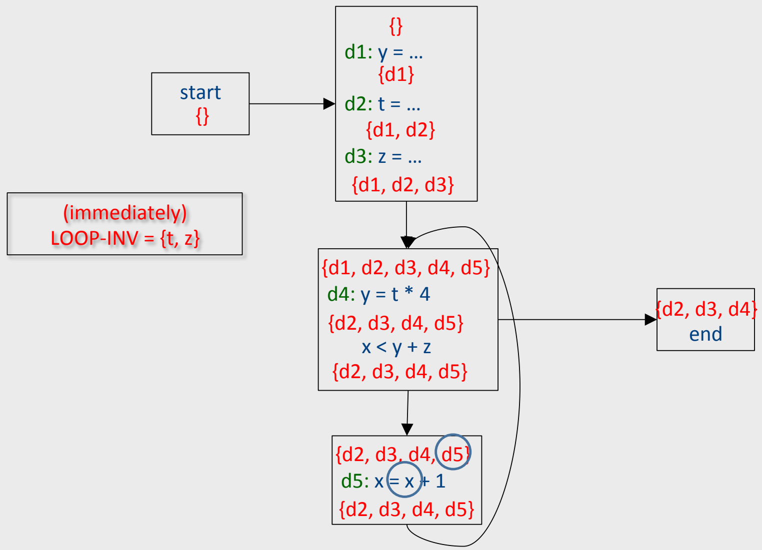
Computing LOOP-INV



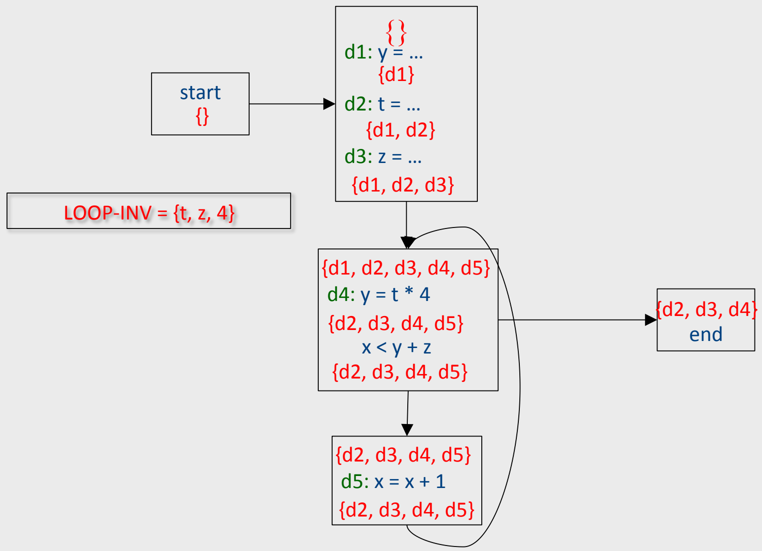
Computing LOOP-INV



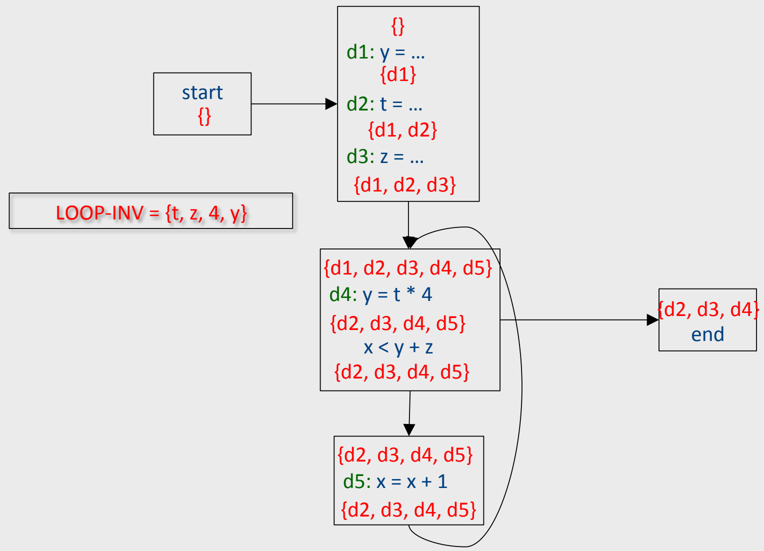
Computing LOOP-INV



Computing LOOP-INV



Computing LOOP-INV



Induction variables

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

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

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

Summary of optimizations

Analysis	Enabled Optimizations
Available Expressions	Common-subexpression elimination Copy Propagation
Constant Propagation	Constant folding
Live Variables	Dead code elimination Register allocation
Reaching Definitions	Loop-invariant code motion

Ad

- Advanced course on program analysis and verification
- Workshop on compile time techniques for detecting malicious JavaScripts
  - With Trusteer
    - Now IBM