Compilation

0368-3133 (Semester A, 2013/14)

Lecture 11: Data Flow Analysis & Optimizations

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv and Eran Yahav

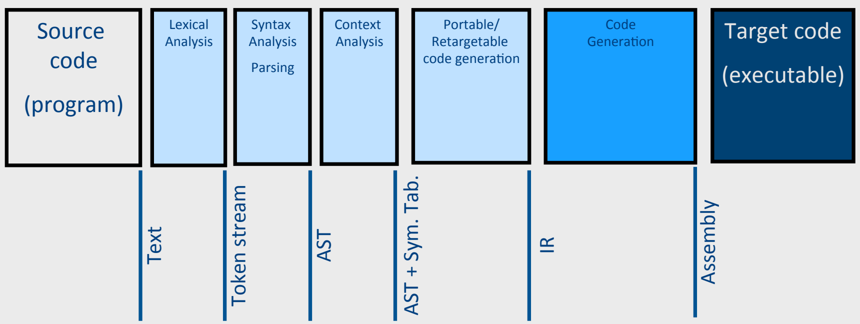
What is a compiler?

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

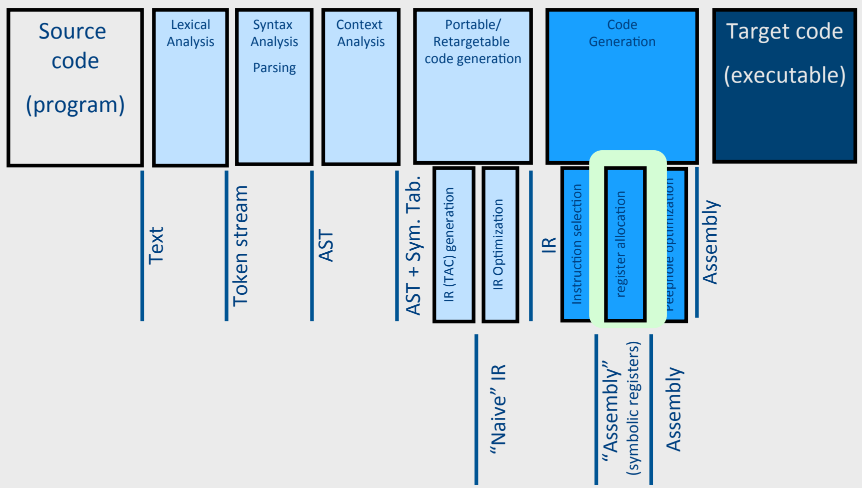
The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

Stages of compilation



Stages of Compilation



Registers

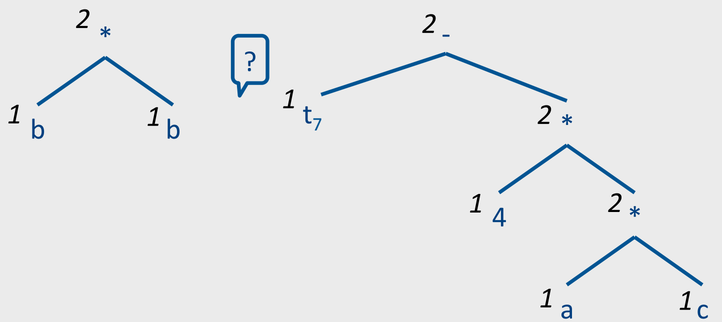
- Most machines have a set of registers, dedicated memory locations that
  - can be accessed quickly,
  - can have computations performed on them, and
  - are used for special purposes (e.g., parameter passing)
- Usages
  - Operands of instructions
  - Store temporary results
  - Can (should) be used as loop indexes due to frequent arithmetic operation
  - Used to manage administrative info
    - e.g., runtime stack

Register Allocation

- 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)

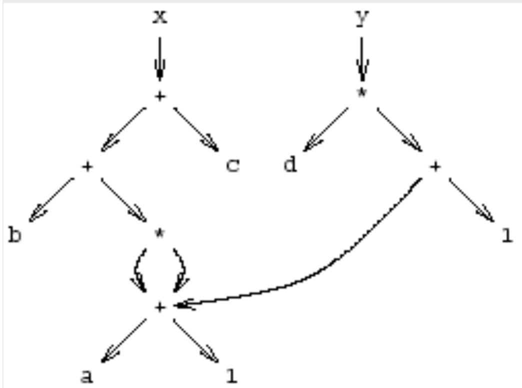
Register Allocation for Expression trees

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



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

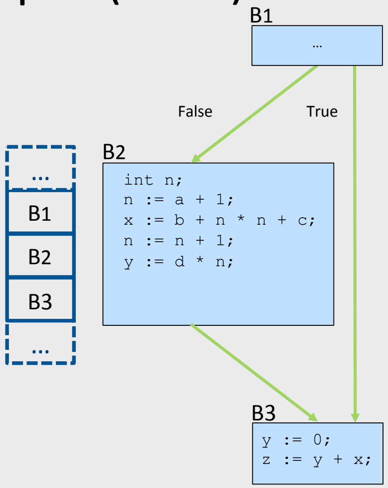
Register Allocation for Basic Blocks



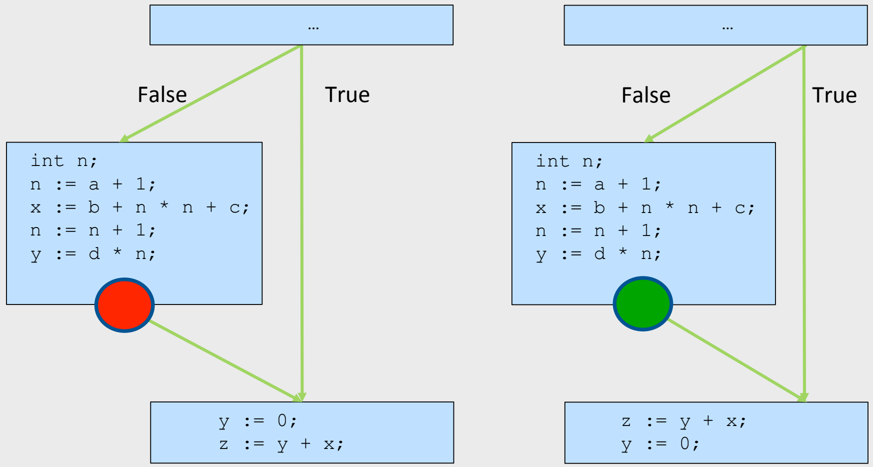

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,X1

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
```

Control Flow Graphs (CFGs)

- A directed graph G=(V,E)
- nodes V = basic blocks
- edges E = control flow
  - (B1,B2) ∈ E when control from B1 flows to B2
- Basic block = Sequence of instructions
  - Cannot jump *into* the middle of a BB
  - Cannot jump out of the middle of the BB
- Leader-based algorithm



y, dead or alive?



Variable Liveness

- A statement x = y + z
  - **defines** x
  - **uses** y and z
- A variable x is live at a program point if its value (at this point) is used at a later point

y = 42	x undef, y live, z undef
z = 73	x undef, y live, z live
x = y + z	x is live, y dead, z dead
print(x);	x is dead, y dead, z dead

(showing state after the statement)

Global Register Allocation using Liveness Information

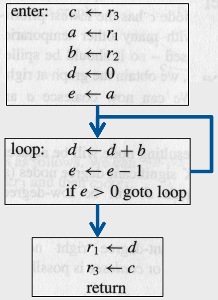
- For every node n in CFG, we have out[n]
  - Set of temporaries live out of n
- Two variables *interfere* if they appear in the same out[n] of any node n
  - **Cannot be allocated to the same register**
- Conversely, if two variables do not interfere with each other, they can be assigned the same register
  - We say they have disjoint live ranges
- How to assign registers to variables?

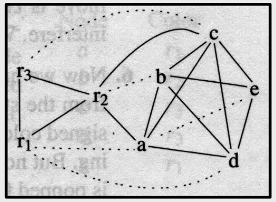
Interference graph

R_1,r_2 pass parameters
R_1 stores return value

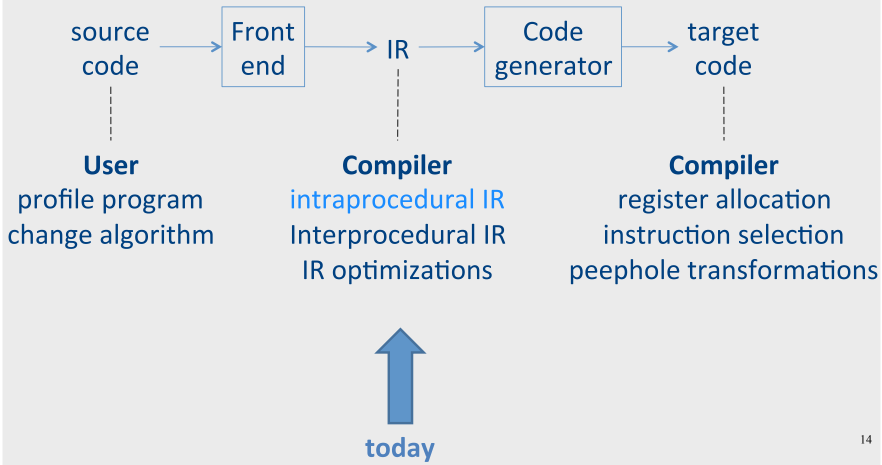
```
int f(int a, int b) {
   int d=0;
   int e=a;
   do {d = d+b;
      e = e-1;
   } while (e>0);
   return d;
}
```

enter: $c \leftarrow r_3$ Callee-saved registers
$a \leftarrow r_1$
$b \leftarrow r_2$ Caller-saved registers
$d \leftarrow 0$
$e \leftarrow a$
loop: $d \leftarrow d + b$
$e \leftarrow e - 1$
if $e > 0$ goto loop
$r_1 \leftarrow d$
$r_3 \leftarrow c$
return (r_1, r_3 *live out*)





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?)*

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

A formalism for IR optimization

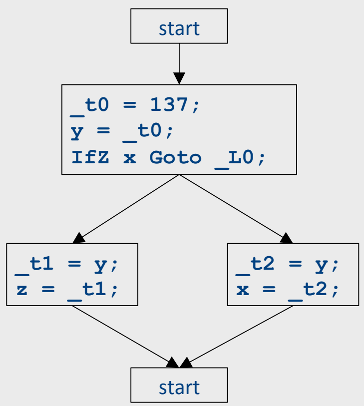
- Every phase of the compiler uses some new abstraction:
  - Scanning uses regular expressions
  - Parsing uses Context Free Grammars (CFGs)
  - Semantic analysis uses proof systems and symbol tables
  - IR generation uses ASTs
- In optimization, we need a formalism that captures the structure of a program in a way amenable to optimization
  - Control Flow Graphs (CFGs)

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
- An optimization is interprocedural if it works across the control-flow graphs of multiple functions
  - We won't talk about this in this course

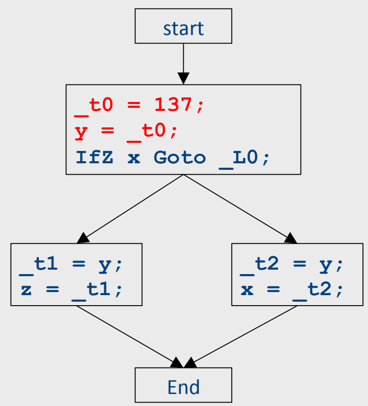
Local optimizations

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

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



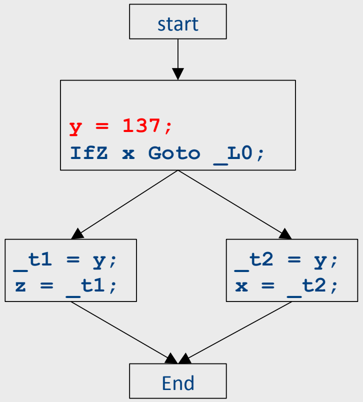
Local optimizations

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

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



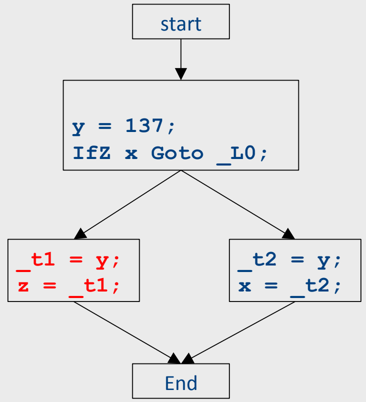
Local optimizations

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

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



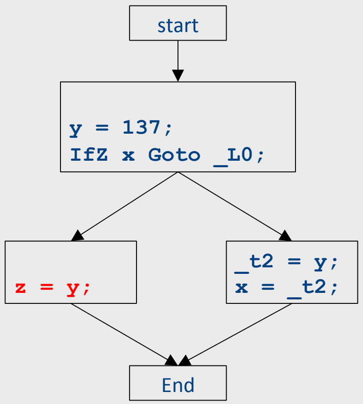
Local optimizations

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

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



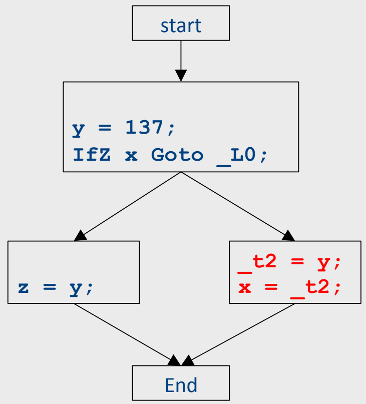
Local optimizations

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

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



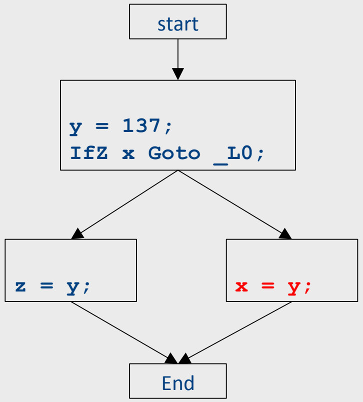
Local optimizations

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

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



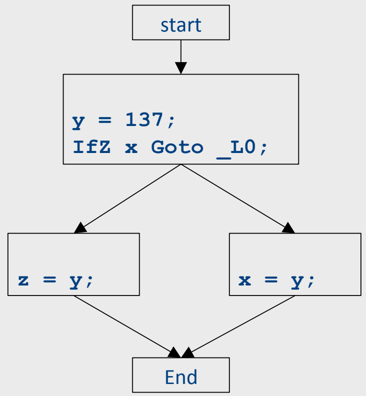
Local optimizations

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

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



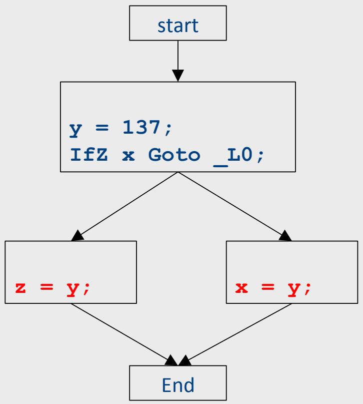
Global optimizations

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

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



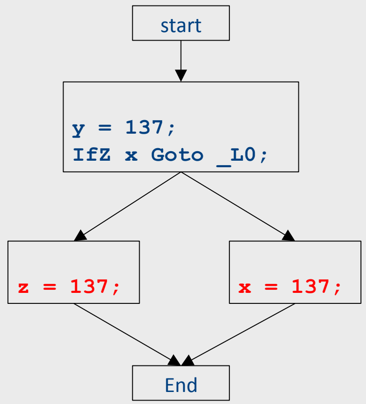
Global optimizations

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

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



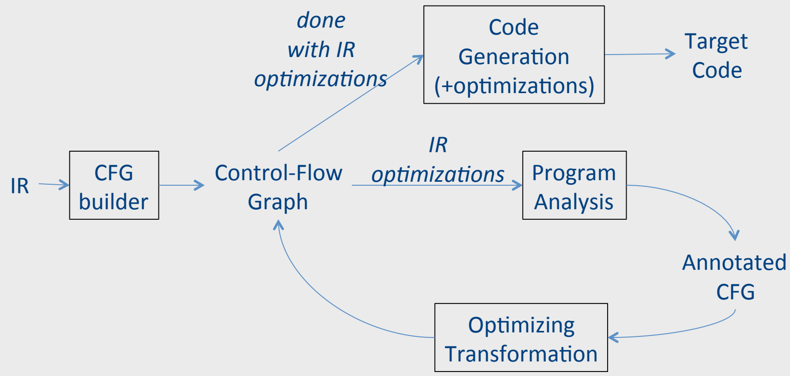
Global optimizations

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

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



Local Optimizations

Optimization path



Common subexpression elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = a + b;
_tmp6 = *(x);
_tmp7 = *(_tmp6);
Push _tmp5;
Push x;
Call _tmp7;
```


Common subexpression elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = a + b;
_tmp6 = *(x);
_tmp7 = *(_tmp6);
Push _tmp5;
Push x;
Call _tmp7;
```

Common subexpression elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = *(x);
_tmp7 = *(_tmp6);
Push _tmp5;
Push x;
Call _tmp7;
```


Common subexpression elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = *(x);
_tmp7 = *(_tmp6);
Push _tmp5;
Push x;
Call _tmp7;
```


Common subexpression elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = *(x);
_tmp7 = *(_tmp6);
Push _tmp5;
Push x;
Call _tmp7;
```


Common subexpression elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = *(x);
_tmp7 = *(_tmp6);
Push _tmp5;
Push x;
Call _tmp7;
```

Common subexpression elimination

```
Object x;
int a;
int b;
int c;

x = new
Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(x);
_tmp7 = *(_tmp6);
Push _tmp5;
Push x;
Call _tmp7;
```


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

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(x);
_tmp7 = *(_tmp6);
Push _tmp5;
Push x;
Call _tmp7;
```


Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(x);
_tmp7 = *(_tmp6);
Push _tmp5;
Push x;
Call _tmp7;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(_tmp1);
_tmp7 = *(_tmp6);
Push _tmp5;
Push _tmp1;
Call _tmp7;
```


Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(_tmp1);
_tmp7 = *(_tmp6);
Push _tmp5;
Push _tmp1;
Call _tmp7;
```


Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(_tmp1);
_tmp7 = *(_tmp6);
Push _tmp5;
Push _tmp1;
Call _tmp7;
```


Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(_tmp1);
_tmp7 = *(_tmp6);
Push _tmp5;
Push _tmp1;
Call _tmp7;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(_tmp1);
_tmp7 = *(_tmp6);
Push c;
Push _tmp1;
Call _tmp7;
```


Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new
Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(_tmp1);
_tmp7 = *(_tmp6);
Push c;
Push _tmp1;
Call _tmp7;
```


Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new
Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(_tmp6);
Push c;
Push _tmp1;
Call _tmp7;
```


Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(_tmp6);
Push c;
Push _tmp1;
Call _tmp7;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(_tmp2);
Push c;
Push _tmp1;
Call _tmp7;
```


Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(_tmp2);
Push c;
Push _tmp1;
Call _tmp7;
```


Copy Propagation

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp0;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(_tmp2);
Push c;
Push _tmp1;
Call _tmp7;
```


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

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp0;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(_tmp2);
Push c;
Push _tmp1;
Call _tmp7;
```


Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new
Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp0;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(_tmp2);
Push c;
Push _tmp1;
Call _tmp7;
```

values never read

values never read

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new
Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4;
Push _tmp0;
_tmp1 = Call _Alloc;
Pop tmp2;
*(_tmp1) = _tmp2;



_tmp4 = _tmp0 + b;
c = _tmp4;


_tmp7 = *(_tmp2);
Push c;
Push _tmp1;
Call _tmp7;
```


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

Applying local optimizations

- The different optimizations we've seen so far all take care of just a small piece of the optimization
- Common subexpression elimination eliminates unnecessary statements
- Copy propagation helps identify dead code
- Dead code elimination removes statements that are no longer needed
- To get maximum effect, we may have to apply these optimizations numerous times

Applying local optimizations example

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```


Applying local optimizations example

```
b = a * a;
c = a * a;
d = b + c;
e = b + b;
```

Which optimization should we apply here?

Applying local optimizations example

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Which optimization should we apply here?

Common sub-expression elimination

Applying local optimizations example

```
b = a * a;
c = b;
d = b + c;
e = b + b;
```

Which optimization should we apply here?

Applying local optimizations example

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Which optimization should we apply here?

Copy propagation

Applying local optimizations example

```
b = a * a;
c = b;
d = b + b;
e = b + b;
```

Which optimization should we apply here?

Applying local optimizations example

```
b = a * a;
c = b;
d = b + b;
e = d;
```

Which optimization should we apply here?

Common sub-expression elimination (again)

Other types of local optimizations

- Arithmetic Simplification
  - Replace "hard" operations with easier ones
  - e.g. rewrite `x = 4 * a;` as `x = a << 2;`
- Constant Folding
  - Evaluate expressions at compile-time if they have a constant value.
  - e.g. rewrite `x = 4 * 5;` as `x = 20;`

Optimizations and analyses

- Most optimizations are only possible given some analysis of the program's behavior
- In order to implement an optimization, we will talk about the corresponding program analyses

Available expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the available expressions in a program
- An expression is called available if some variable in the program holds the value of that expression
- In common subexpression elimination, we replace an available expression by the variable holding its value
- In copy propagation, we replace the use of a variable by the available expression it holds

Finding available expressions

- Initially, no expressions are available
- Whenever we execute a statement **a = b *op* c**:
  - Any expression holding **a** is invalidated
  - The expression **a = b *op* c** becomes available
- **Idea:** Iterate across the basic block, beginning with the empty set of expressions and updating available expressions at each variable

Available expressions example

```
 {   }
a = b;
 { a = b }
c = b;
 { a = b, c = b }
d = a + b;
 { a = b, c = b, d = a + b }
e = a + b;
 { a = b, c = b, d = a + b, e = a + b }
d = b;
 { a = b, c = b, d = b, e = a + b }
f = a + b;
 { a = b, c = b, d = b, e = a + b, f = a + b }
```


Common sub-expression elimination

```
 {   }
a = b;
 { a = b }
c = b;
 { a = b, c = b }
d = a + b;
 { a = b, c = b, d = a + b }
e = a + b;
 { a = b, c = b, d = a + b, e = a + b }
d = b;
 { a = b, c = b, d = b, e = a + b }
f = a + b;
 { a = b, c = b, d = b, e = a + b, f = a + b }
```


Common sub-expression elimination

```
 {   }
a = b;
 { a = b }
c = b;
 { a = b, c = b }
d = a + b;
 { a = b, c = b, d = a + b }
e = a + b;
 { a = b, c = b, d = a + b, e = a + b }
d = b;
 { a = b, c = b, d = b, e = a + b }
f = a + b;
 { a = b, c = b, d = b, e = a + b, f = a + b }
```


Common sub-expression elimination

```
 {   }
a = b;
 { a = b }
c = a;
 { a = b, c = b }
d = a + b;
 { a = b, c = b, d = a + b }
e = d;
 { a = b, c = b, d = a + b, e = a + b }
d = a;
 { a = b, c = b, d = b, e = a + b }
f = e;
 { a = b, c = b, d = b, e = a + b, f = a + b }
```

Live variables

- The analysis corresponding to dead code elimination is called 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
- Dead code elimination works by computing liveness for each variable, then eliminating assignments to dead variables

Computing live variables

- To know if a variable will be used at some point, we iterate across the statements in a basic block in reverse order
- Initially, some small set of values are known to be live (which ones depends on the particular program)
- When we see the statement a = b op c:
  - Just before the statement, a is not alive, since its value is about to be overwritten
  - Just before the statement, both b and c are alive, since we're about to read their values
  - (what if we have a = a + b?)

Liveness analysis

```
 { b }
a = b;
 { a, b }
c = a;
 { a, b }
d = a + b;
 { a, b, d }
e = d;
 { a, b, e }
d = a;
 { b, d, e }
f = e;
 { b, d } - given
```

Which statements are dead?

Dead Code Elimination

```
 { b }
a = b;
 { a, b }
c = a;
 { a, b }
d = a + b;
 { a, b, d }
e = d;
 { a, b, e }
d = a;
 { b, d, e }
f = e;
 { b, d }
```

Which statements are dead?

Dead Code Elimination

```
 { b }
a = b;
 { a, b }

 { a, b }
d = a + b;
 { a, b, d }
e = d;
 { a, b, e }
d = a;
 { b, d, e }

 { b, d }
```


Liveness analysis II

```
 { b }
a = b;

 { a, b }
d = a + b;
 { a, b, d }
e = d;
 { a, b }
d = a;
 { b, d }
```

Which statements are dead?

Liveness analysis II

```
 { b }
a = b;

 { a, b }
d = a + b;
 { a, b, d }
e = d;
 { a, b }
d = a;
 { b, d }
```

Which statements are dead?

Dead code elimination

```
 { b }
a = b;

 { a, b }
d = a + b;
 { a, b, d }
e = d;
 { a, b }
d = a;
 { b, d }
```

Which statements are dead?

Dead code elimination

```
 { b }
a = b;


 { a, b }
d = a + b;
 { a, b, d }

 { a, b }
d = a;
 { b, d }
```


Liveness analysis III

Which statements are dead?

```
 { b }
a = b;


 { a, b }
d = a + b;


 { a, b }
d = a;
 { b, d }
```


Dead code elimination

Which statements are dead?

```
 { b }
a = b;


 { a, b }
d = a + b;


 { a, b }
d = a;
 { b, d }
```


Dead code elimination

```
 { b }
a = b;


 { a, b }



 { a, b }
d = a;
 { b, d }
```

Dead code elimination

```
a = b;
```

If we further apply copy propagation this statement can be eliminated too

```
d = a;
```


A combined algorithm

- Start with initial live variables at end of block
- Traverse statements from end to beginning
- For each statement
  - If assigns to dead variables – eliminate it
  - Otherwise, compute live variables before statement and continue in reverse

A combined algorithm

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
```


A combined algorithm

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
 { b, d }
```

A combined algorithm

```
a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
 { b, d }
```


A combined algorithm

```
a = b;

c = a;

d = a + b;

e = d;

d = a;


 { b, d }
```


A combined algorithm

```
a = b;

c = a;

d = a + b;

e = d;
 { a, b }
d = a;


 { b, d }
```


A combined algorithm

```
a = b;

c = a;

d = a + b;

e = d;
 { a, b }
d = a;


 { b, d }
```

A combined algorithm

```
a = b;

c = a;

d = a + b;

 { a, b }
d = a;

 { b, d }
```


A combined algorithm

```
a = b;

c = a;

d = a + b;

 { a, b }
d = a;

 { b, d }
```


A combined algorithm

```
a = b;

c = a;

 { a, b }
d = a;

 { b, d }
```


A combined algorithm

```
a = b;

c = a;

 { a, b }
d = a;

 { b, d }
```

A combined algorithm

```
a = b;

 { a, b }
d = a;

 { b, d }
```


A combined algorithm

```
 { b }
a = b;

 { a, b }
d = a;

 { b, d }
```

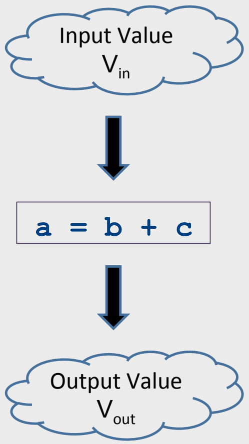

A combined algorithm

```
a = b;

d = a;
```


High-level goals

- Generalize analysis mechanism
  - Reuse common ingredients for many analyses
  - Reuse proofs of correctness
- Generalize from basic blocks to entire CFGs
  - Go from local optimizations to global optimizations

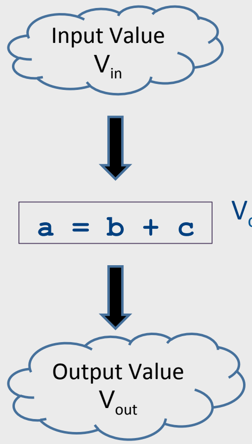
Formalizing local analyses



Transfer Function

$$V_{out} = f_{\mathtt{a=b+c}}(V_{in})$$

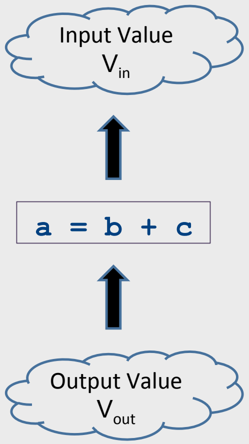
Available Expressions



$V_{out} = (V_{in} \setminus \{e \mid e \text{ contains } \mathbf{a}\}) \cup \{a=b+c\}$

Expressions of the forms
a=… and x=…**a**…

Live Variables



$$V_{in} = (V_{out} \setminus \{\mathtt{a}\}) \cup \{\mathtt{b,c}\}$$

Information for a local analysis

- What direction are we going?
  - Sometimes forward (available expressions)
  - Sometimes backward (liveness analysis)
- How do we update information after processing a statement?
  - What are the new semantics?
  - What information do we know initially?

Formalizing local 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
  - **F** is a family of transfer functions defining the meaning of any expression as a function f : **V** $\rightarrow$ **V**
  - I is the initial information at the top (or bottom) of a basic block

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

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

Running local analyses

- Given an analysis **(D, V, F, I)** for a basic block
- Assume that **D** is "forward;" analogous for the reverse case
- Initially, set OUT[**entry**] to **I**
- For each statement **s**, in order:
  - Set IN[**s**] to OUT[**prev**], where **prev** is the previous statement
  - Set OUT[**s**] to f_s(IN[**s**]), where f_s is the transfer function for statement **s**

Global Optimizations

Global analysis

- A global analysis is an analysis that works on a control-flow graph as a whole
- Substantially more powerful than a local analysis
  - (Why?)
- Substantially more complicated than a local analysis
  - (Why?)

Local vs. global analysis

- Many of the optimizations from local analysis can still be applied globally
  - Common sub-expression elimination
  - Copy propagation
  - Dead code elimination
- Certain optimizations are possible in global analysis that aren't possible locally:
  - e.g. code motion: Moving code from one basic block into another to avoid computing values unnecessarily
- Example global optimizations:
  - Global constant propagation
  - Partial redundancy elimination

Loop invariant code motion example

```
while (t < 120) {
  z = z + x - y;
}
```

⇨

```
w = x - y;
while (t < 120) {
  z = z + w;
}
```

value of expression x – y is not changed by loop body

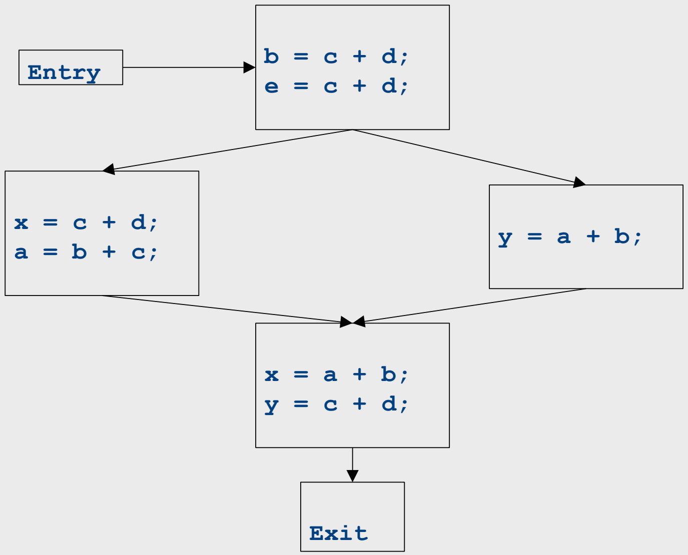
Why global analysis is hard

- Need to be able to handle multiple predecessors/successors for a basic block
- Need to be able to handle multiple paths through the control-flow graph, and may need to iterate multiple times to compute the final value (but the analysis still needs to terminate!)
- Need to be able to assign each basic block a reasonable default value for before we've analyzed it

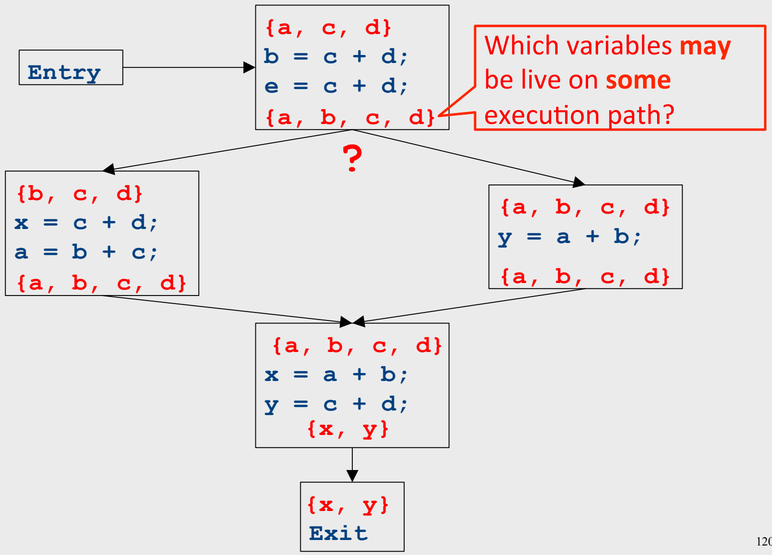
Global dead code elimination

- Local dead code elimination needed to know what variables were live on exit from a basic block
- This information can only be computed as part of a global analysis
- How do we modify our liveness analysis to handle a CFG?

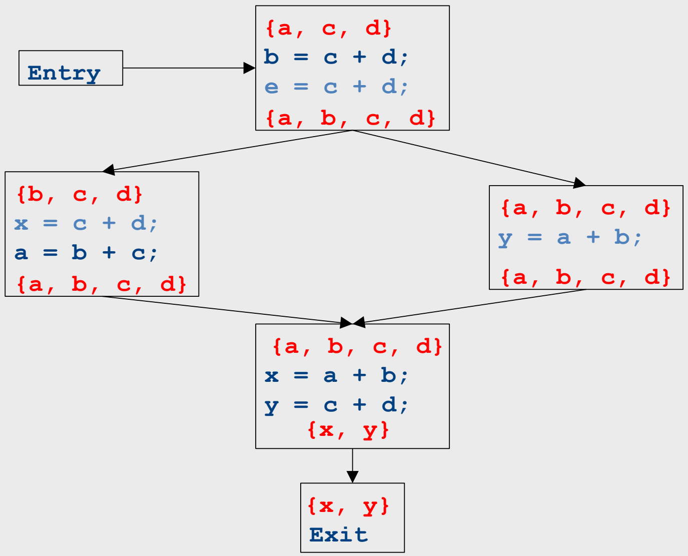
CFGs without loops



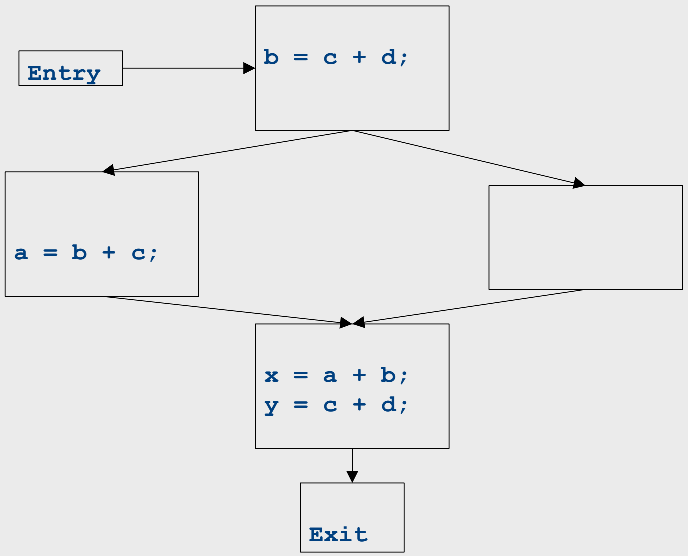
CFGs without loops



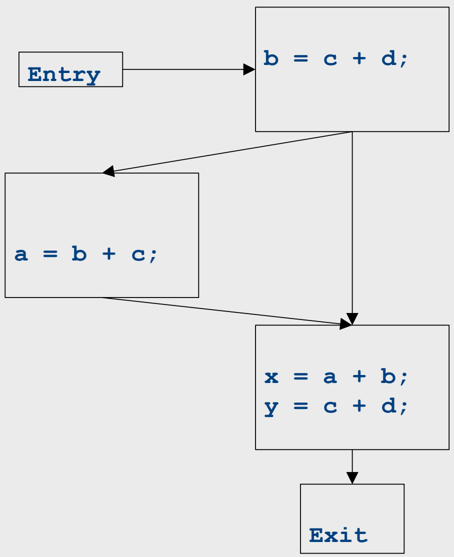
CFGs without loops



CFGs without loops



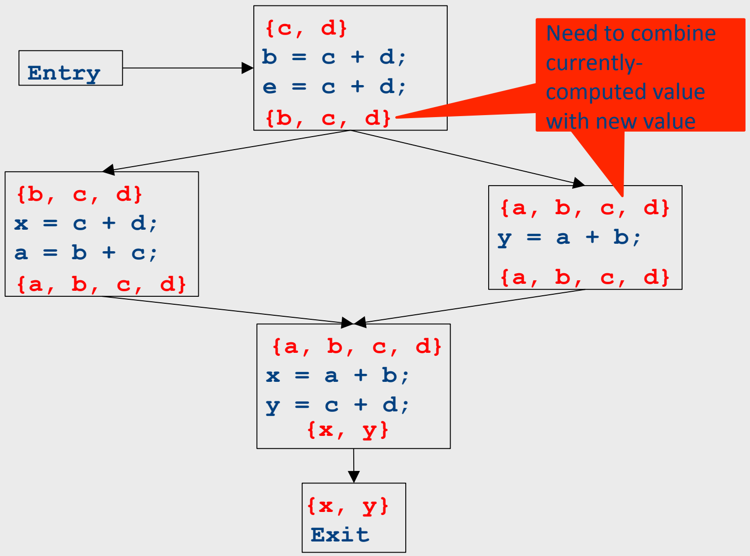
CFGs without loops



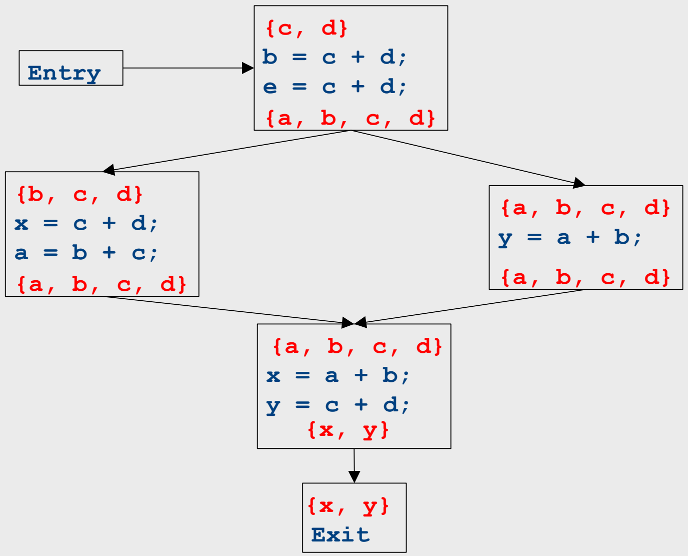
Major changes – part 1

- In a local analysis, each statement has exactly one predecessor
- In a global analysis, each statement may have **multiple** predecessors
- A global analysis must have some means of **combining information** from all predecessors of a basic block

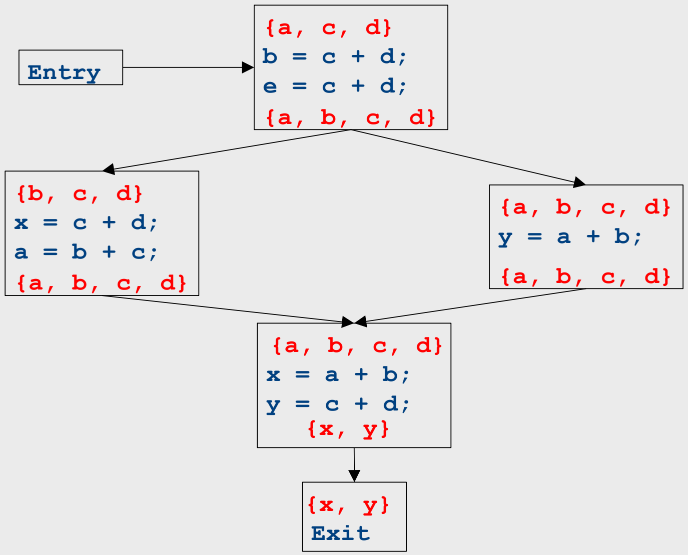
CFGs without loops



CFGs without loops



CFGs without loops

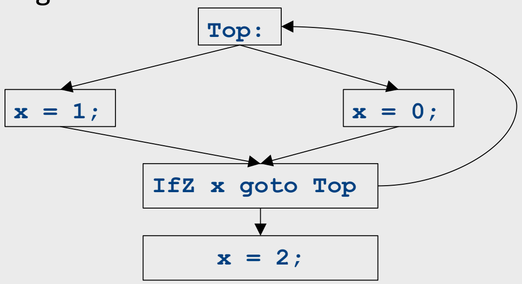


Major changes – part 2

- In a local analysis, there is only one possible path through a basic block
- In a global analysis, there may be **many** paths through a CFG
- May need to recompute values multiple times as more information becomes available
- Need to be careful when doing this not to loop infinitely!
  - (More on that later)

CFGs with loops

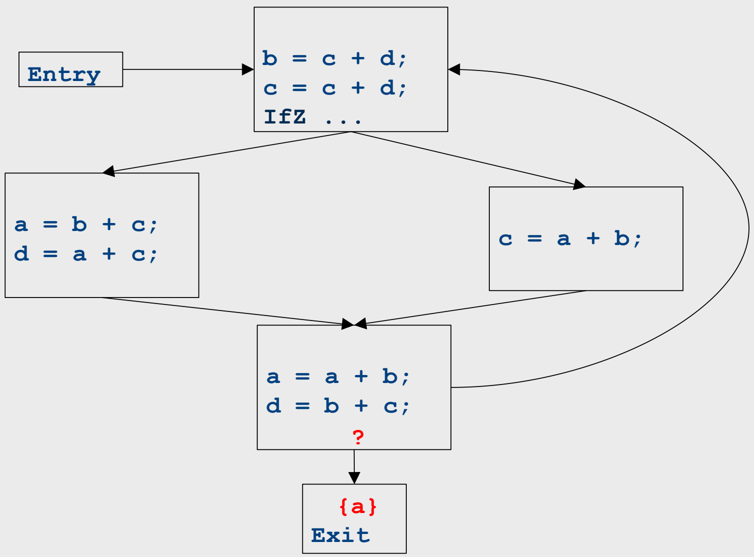
- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths
- When we add loops into the picture, this is no longer true
- Not all possible loops in a CFG can be realized in the actual program



CFGs with loops

- Up to this point, we've considered loop-free CFGs, which have only finitely many possible paths
- When we add loops into the picture, this is no longer true
- Not all possible loops in a CFG can be realized in the actual program
- **Sound approximation:** Assume that every possible path through the CFG corresponds to a valid execution
  - Includes all realizable paths, but some additional paths as well
  - May make our analysis less precise (but still sound)
  - Makes the analysis feasible; we'll see how later

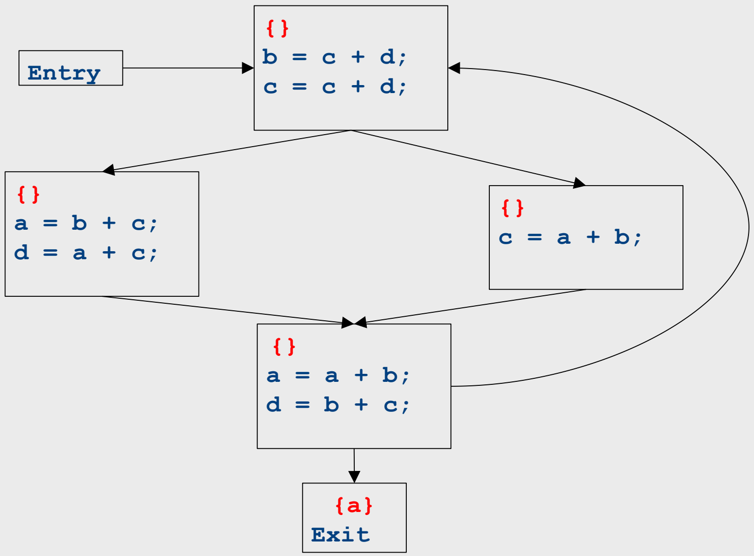
CFGs with loops



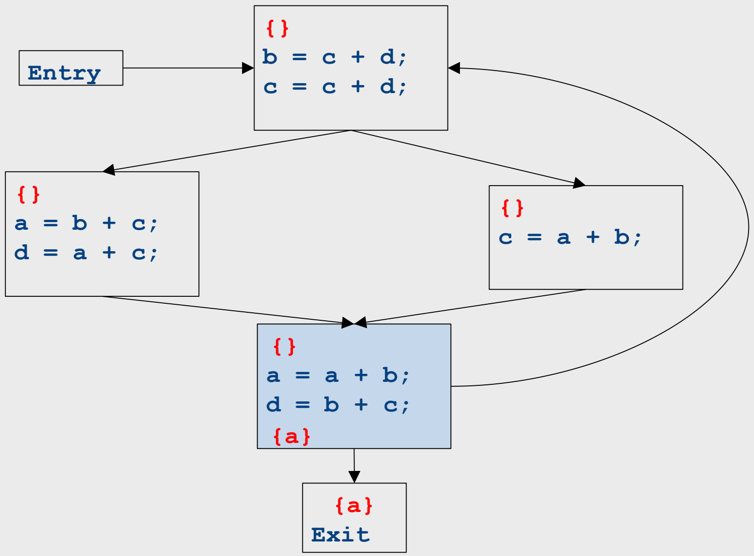
Major changes – part 3

- In a local analysis, there is always a well defined "first" statement to begin processing
- In a global analysis with loops, every basic block might depend on every other basic block
- To fix this, we need to assign initial values to all of the blocks in the CFG

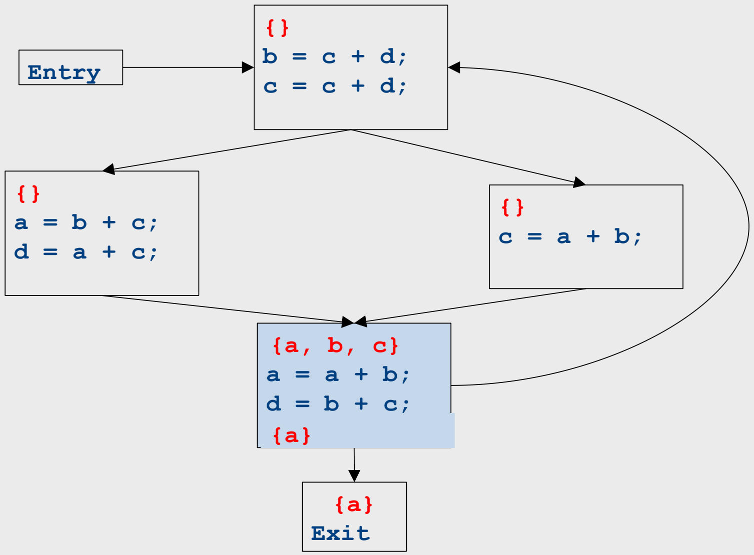
CFGs with loops - initialization



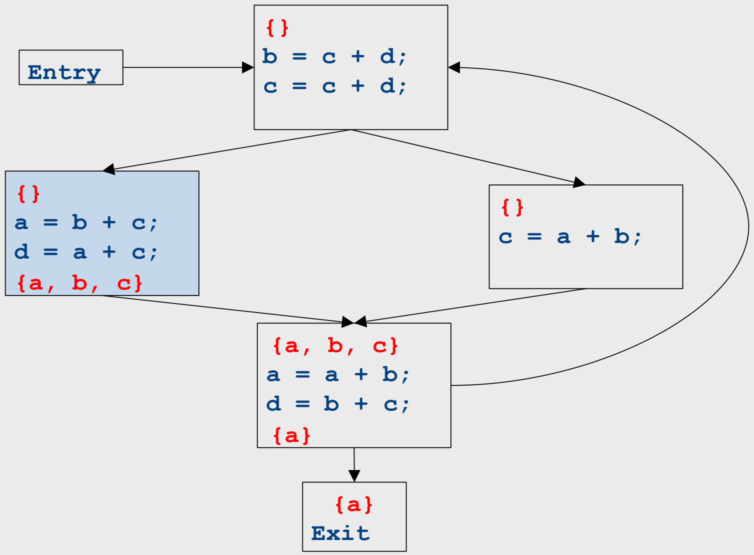
CFGs with loops - iteration



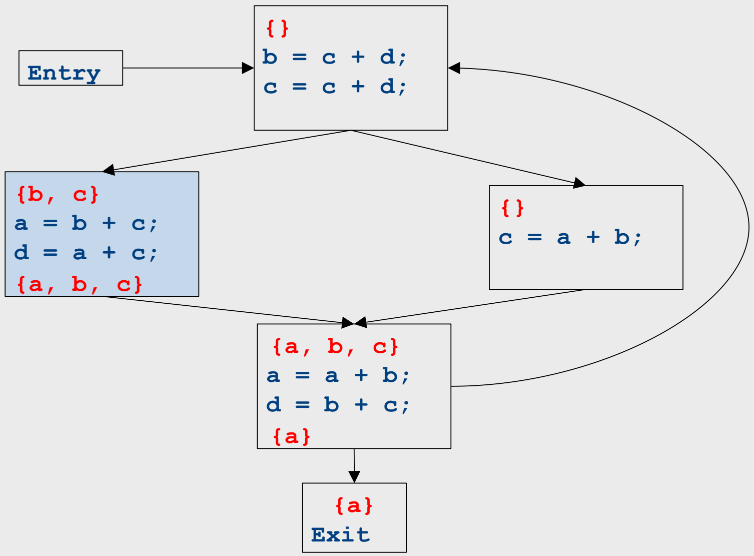
CFGs with loops - iteration



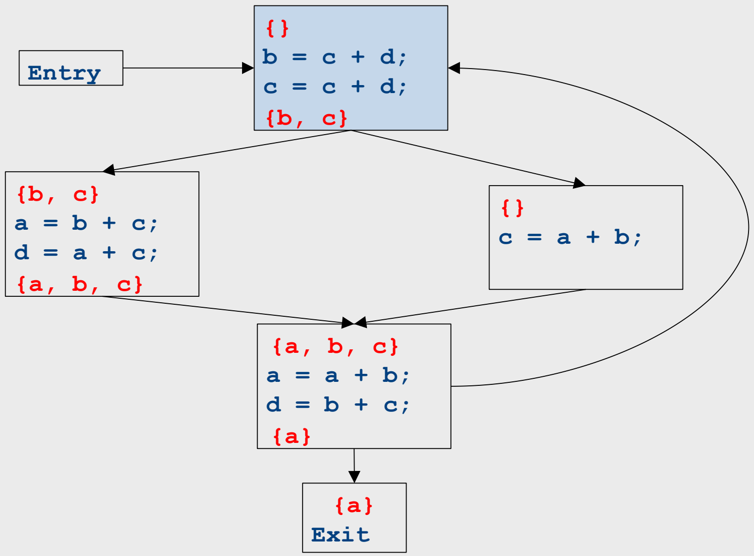
CFGs with loops - iteration



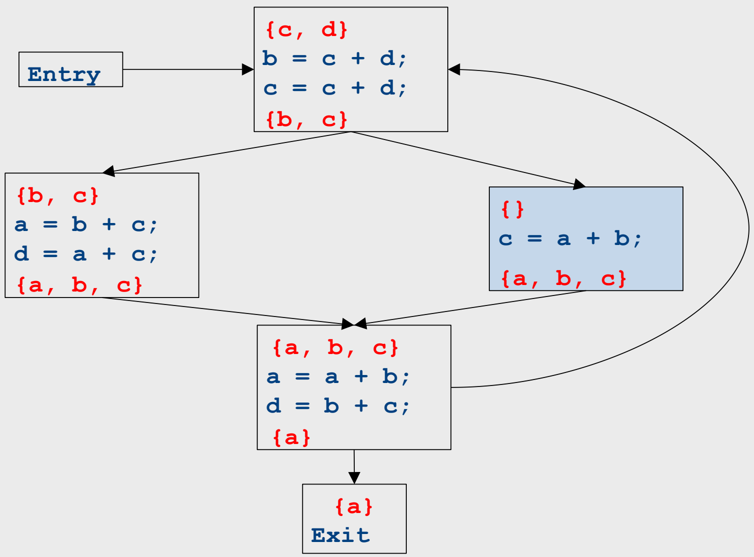
CFGs with loops - iteration



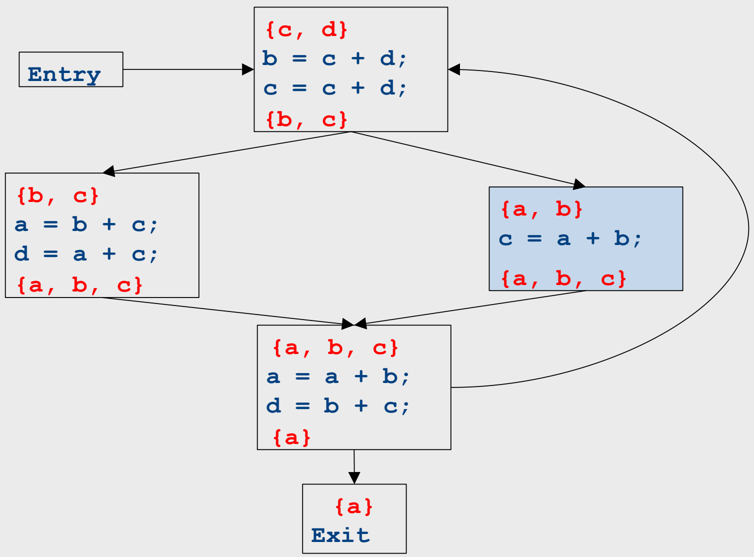
CFGs with loops - iteration



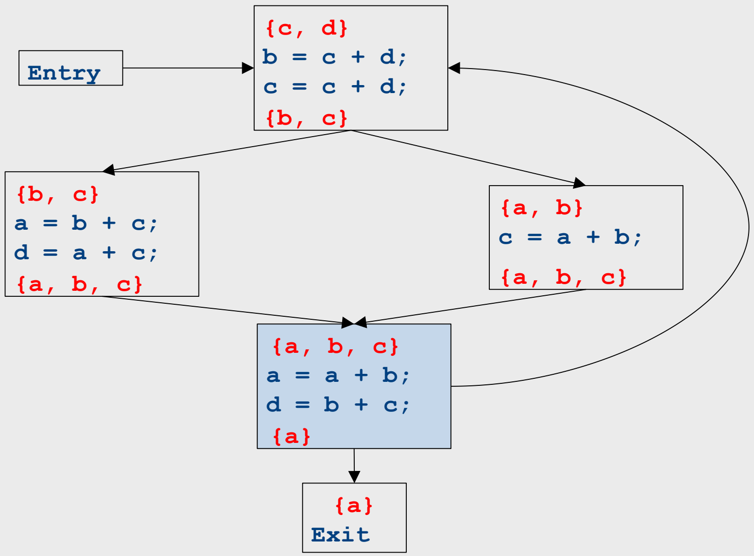
CFGs with loops - iteration



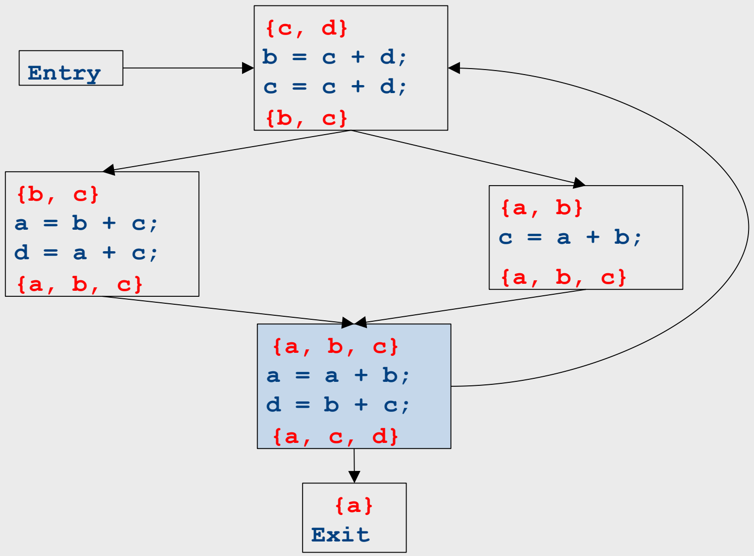
CFGs with loops - iteration



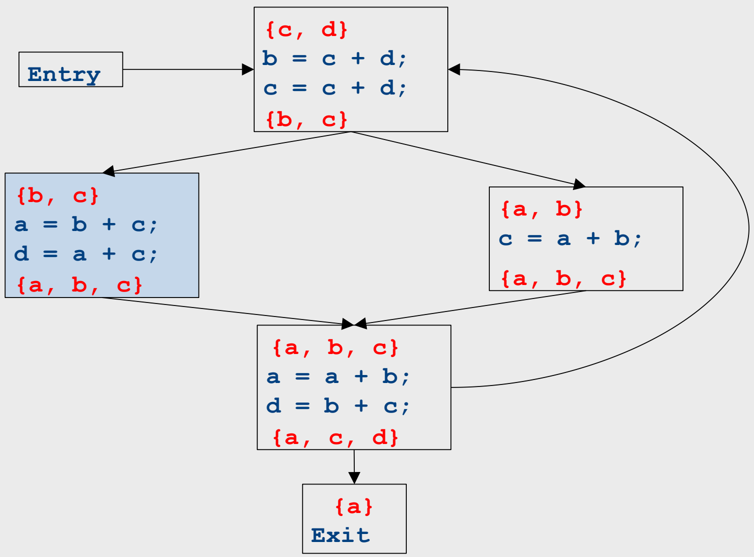
CFGs with loops - iteration



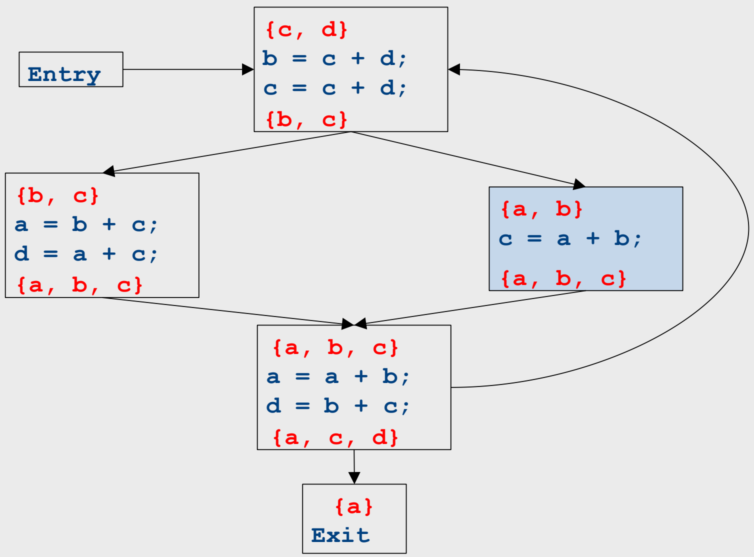
CFGs with loops - iteration



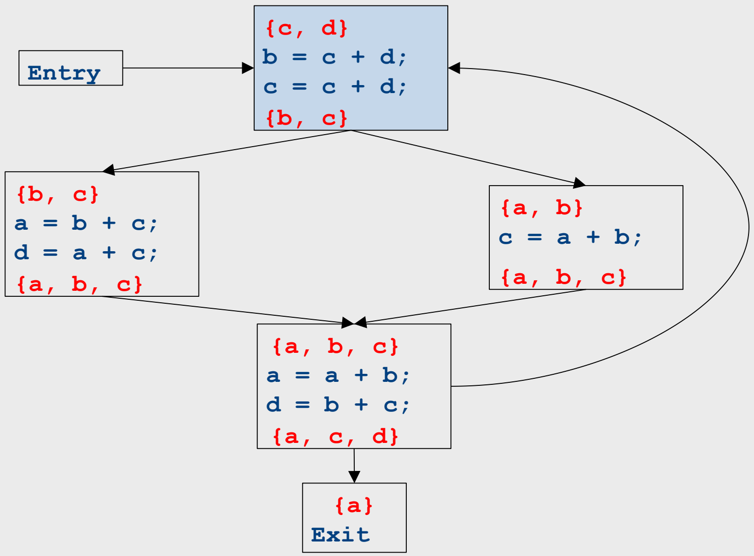
CFGs with loops - iteration



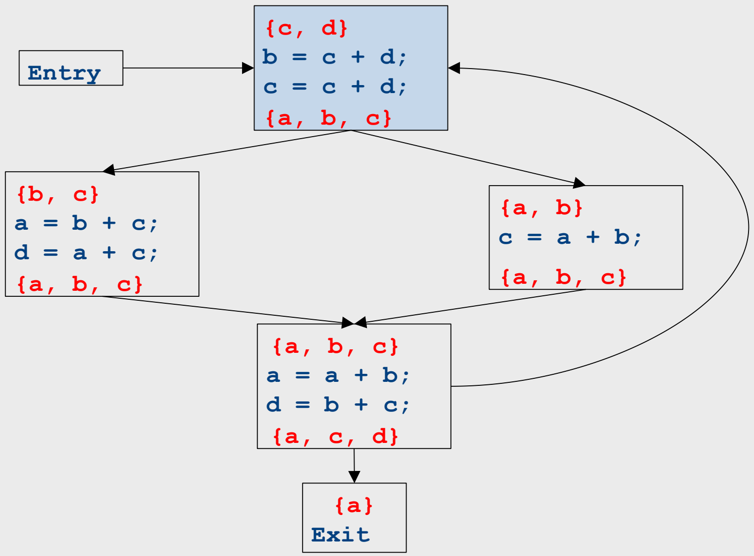
CFGs with loops - iteration



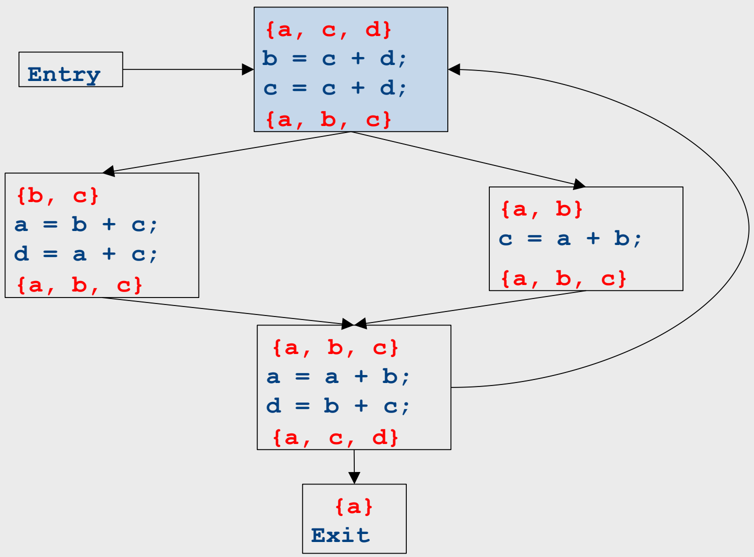
CFGs with loops - iteration



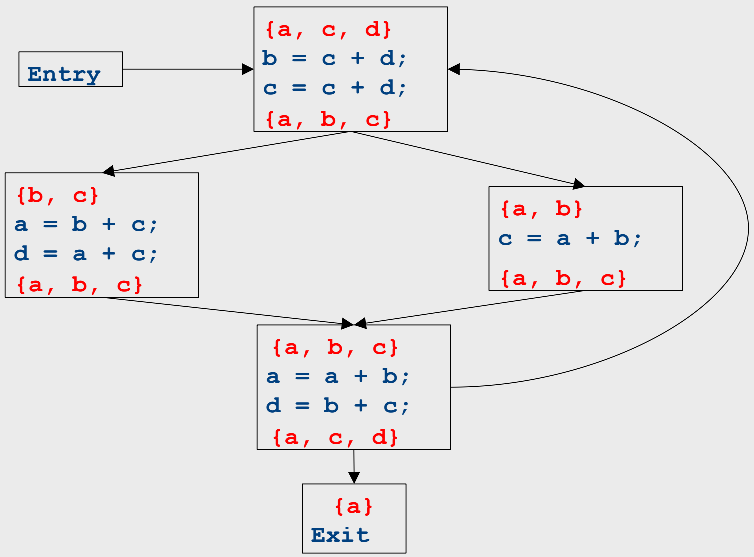
CFGs with loops - iteration



CFGs with loops - iteration



CFGs with loops - iteration



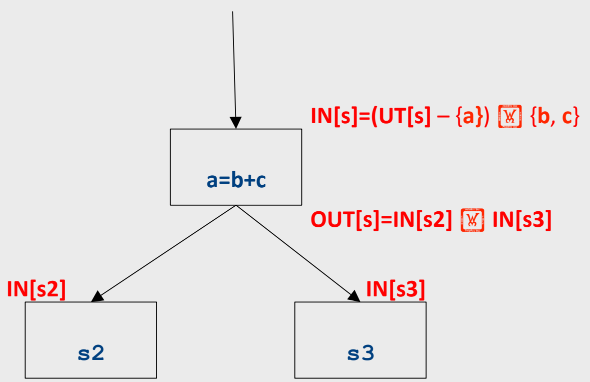
Summary of differences

- Need to be able to handle multiple predecessors/successors for a basic block
- Need to be able to handle multiple paths through the control-flow graph, and may need to iterate multiple times to compute the final value
  - But the analysis still needs to terminate!
- Need to be able to assign each basic block a reasonable default value for before we've analyzed it

Global liveness analysis

- Initially, set IN[**s**] = { } for each statement **s**
- Set IN[**exit**] to the set of variables known to be live on exit (language-specific knowledge)
- Repeat until no changes occur:
  - For each statement **s** of the form **a = b + c**, in any order you'd like:
    - Set OUT[**s**] to set union of IN[**p**] for each successor **p** of **s**
    - Set IN[**s**] to (OUT[**s**] – **a**) ∪ {**b**, **c**}.
- Yet another fixed-point iteration!

Global liveness analysis



Why does this work?

- 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

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

Properties of local analysis

- The only way to find out 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
- However, this is not a problem inside a basic block
  - Basic blocks contain no loops
  - There is only one path through the basic block

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

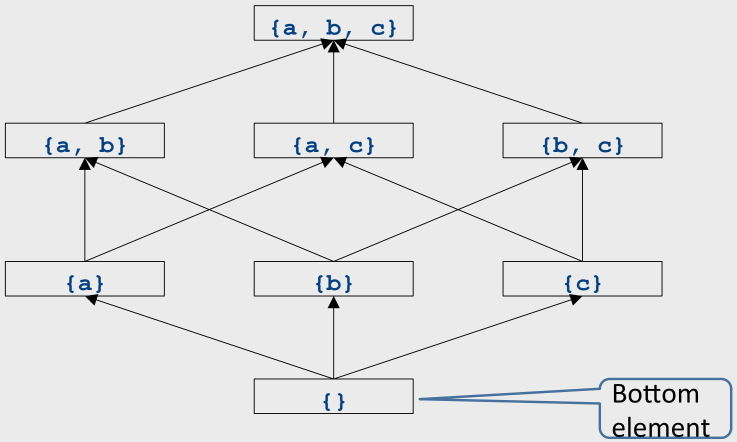
Theory to the rescue

- 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

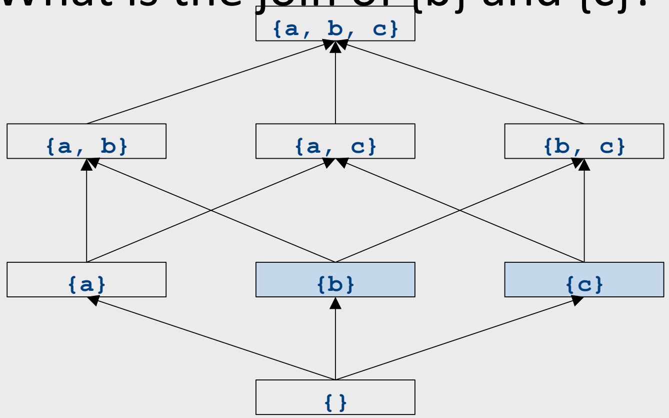
Join semilattices

- A join semilattice is a ordering defined on a set of elements
- 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”

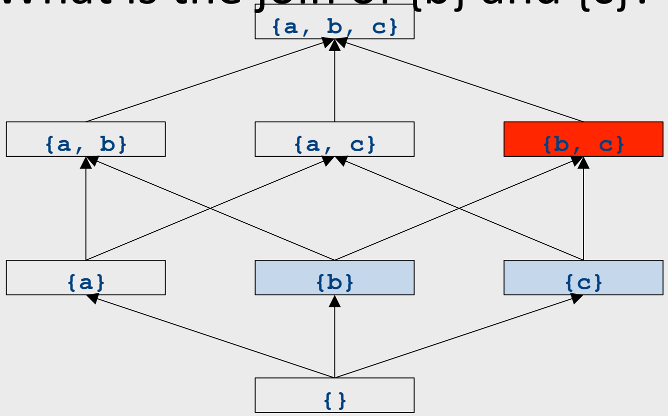
Join semilattice for liveness



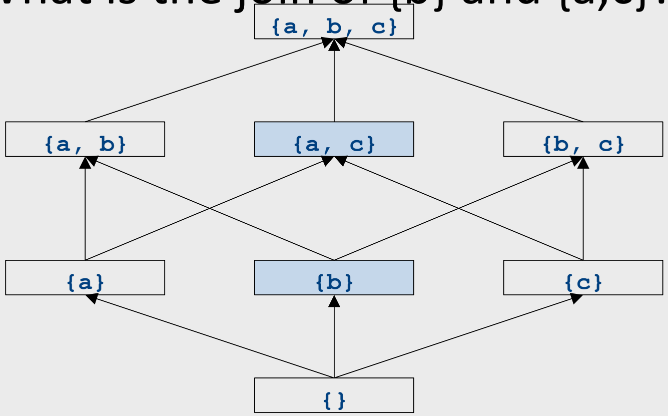
What is the join of {b} and {c}?



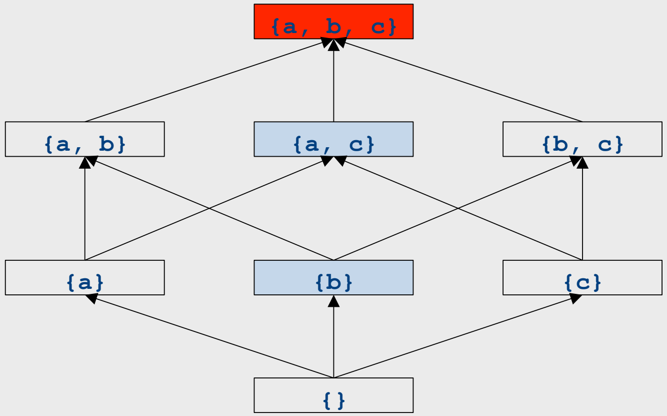
What is the join of {b} and {c}?



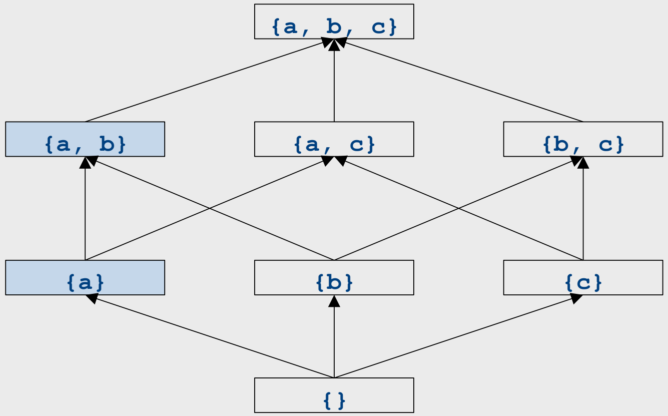
What is the join of {b} and {a,c}?



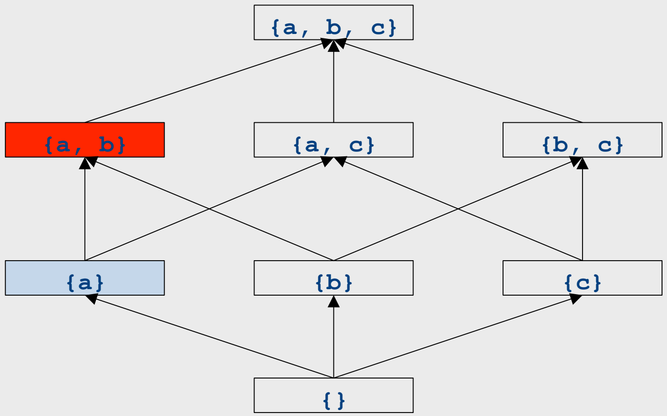
What is the join of {b} and {a,c}?



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



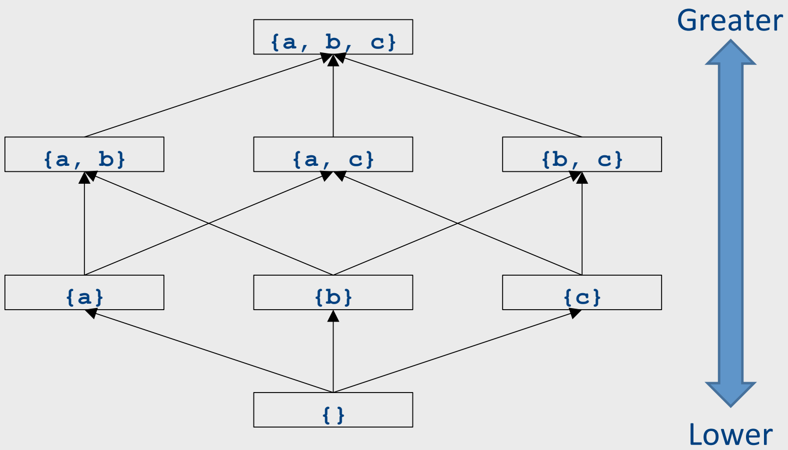
What is the join of {a} and {a,b}?



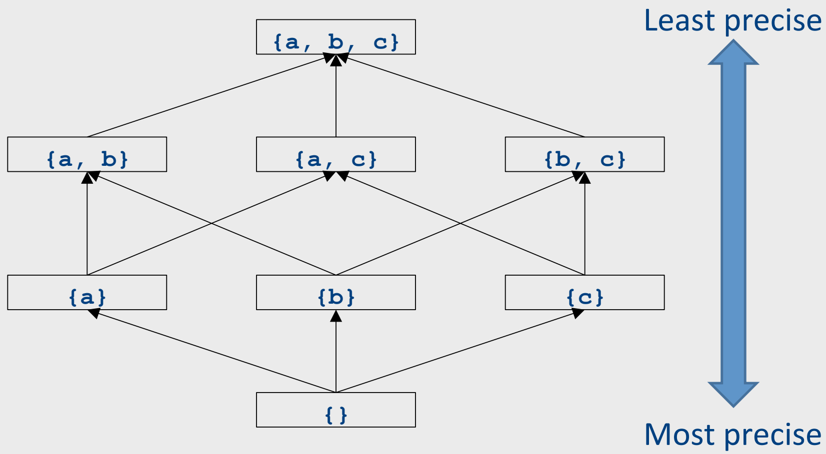
Formal definitions

- A join semilattice is a pair (V, ⊠), where
- V is a domain of elements
- ⊠ is a join operator that is
  - commutative: x ⊠ y = y ⊠ x
  - associative: (x ⊠ y) ⊠ z = x ⊠ (y ⊠ z)
  - idempotent: x ⊠ x = x
- If x ⊠ y = z, we say that z is the join or (least upper bound) of x and y
- Every join semilattice has a bottom element denoted ⊠ such that ⊠ ⊠ 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 $\sqcup$ x = x
- Commutative:
  - x $\sqcup$ y = y $\sqcup$ x
- Associative:
  - (x $\sqcup$ y) $\sqcup$ z = x $\sqcup$ (y $\sqcup$ z)
- Bottom element:
  - The empty set: Ø $\sqcup$ 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

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 : V* ☒ **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 $\mathbf{p_1, p_2, \ldots, p_n}$:
    - 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

For comparison

- 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**])

- Set IN[**s**] = {} for all statements **s**
- Set OUT[**exit**] = the set of variables known to be live on exit
- Repeat until no values change:
  - For each statement **s** of the form **a=b+c**:
    - Set OUT[**s**] = set union of IN[**x**] for each successor **x** of **s**
    - Set IN[**s**] = (OUT[**s**]-{a}) ⊠ {b,c}

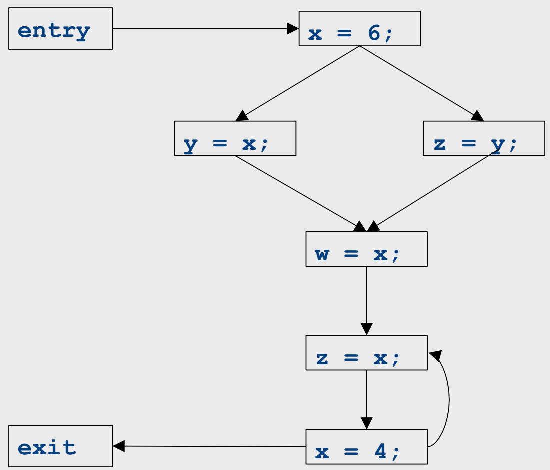
The dataflow framework

- 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

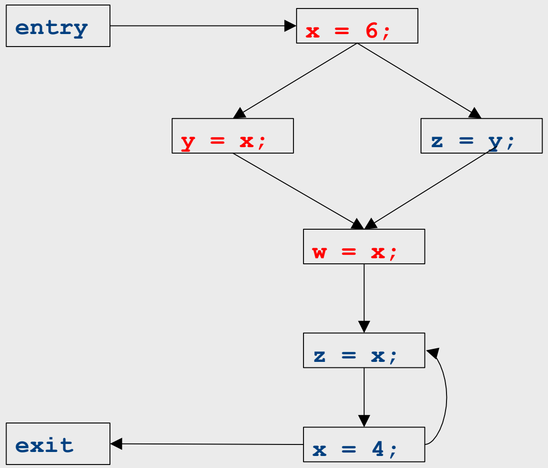
Global constant propagation

- 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

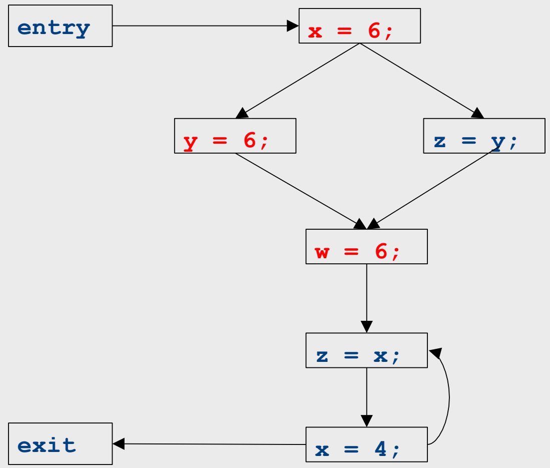
Global constant propagation



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

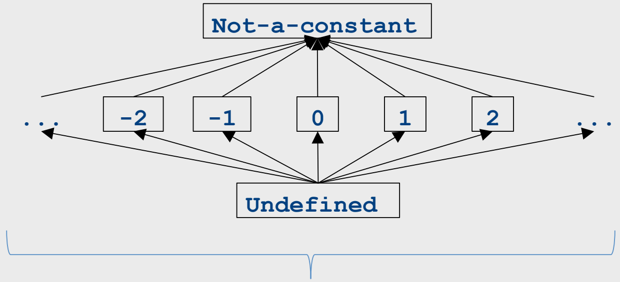
- For now, consider just some single variable **x**
- At each point in the program, we know one of three things about the value of **x**:
  - **x** is definitely not a constant, since it's been assigned two values or assigned a value that we know isn't a constant
  - **x** is definitely a constant and 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

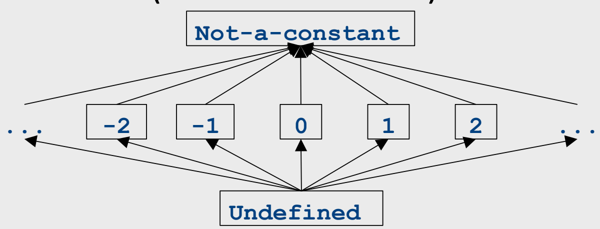
- One possible semilattice for this analysis is shown here (for each variable):



The lattice is infinitely wide

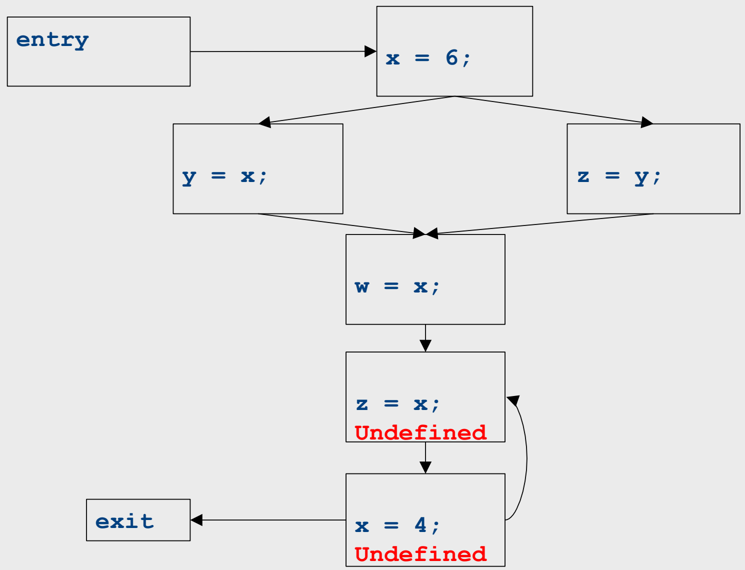
A semilattice for constant propagation

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

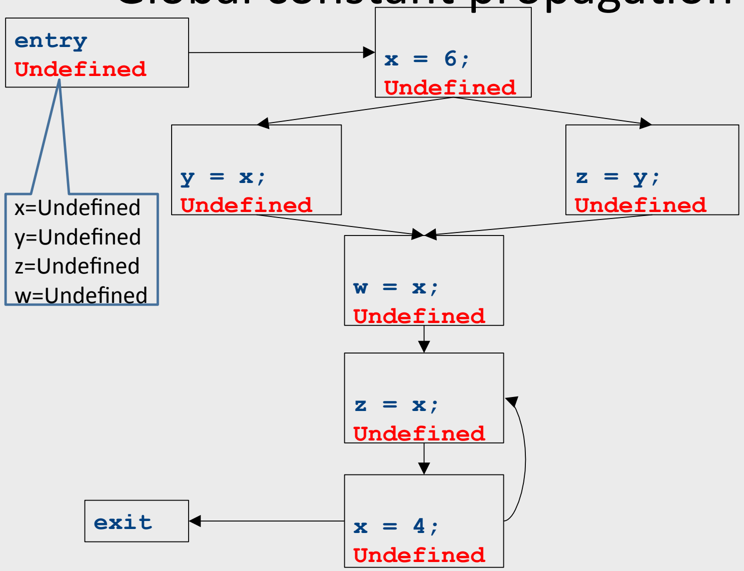


- Note:
  - The join of any two different constants is **Not-a-Constant**
  - The join of **Not a Constant** and any other value is **Not-a-Constant**
  - The join of **Undefined** and any other value is that other value

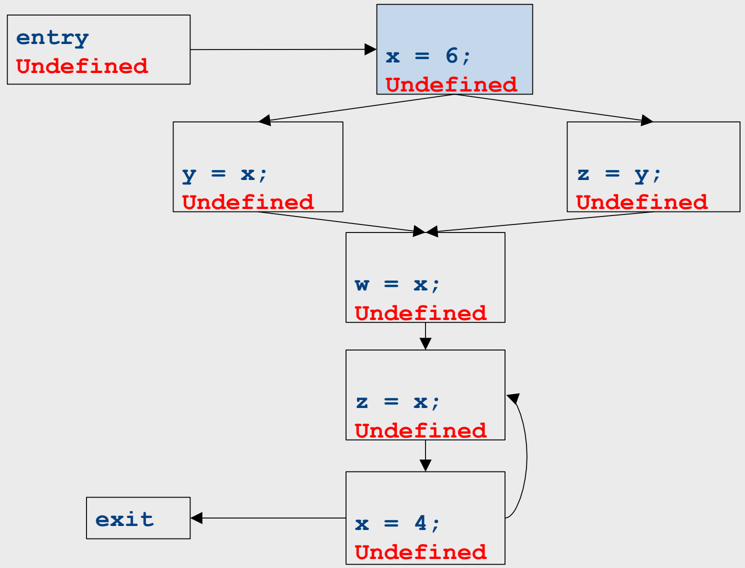
Global constant propagation



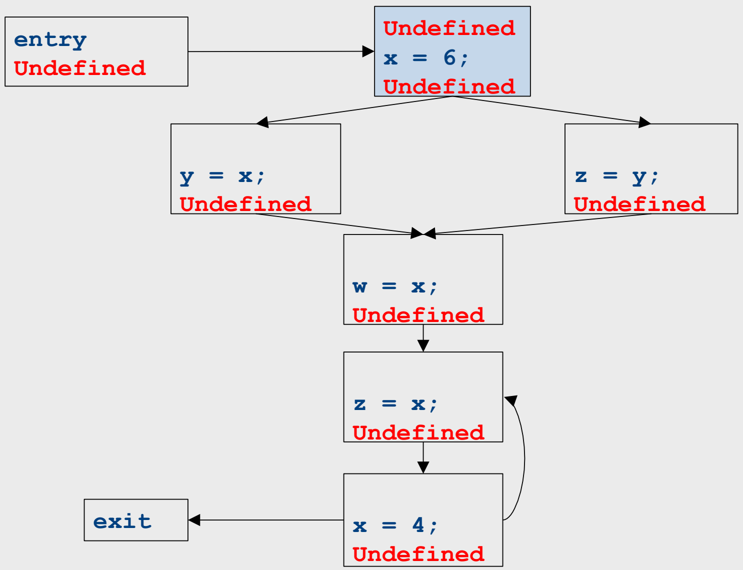
Global constant propagation



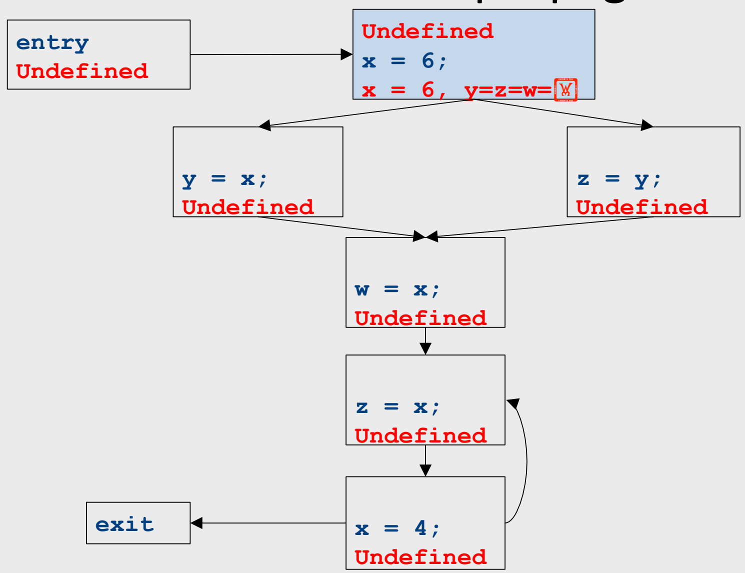
Global constant propagation



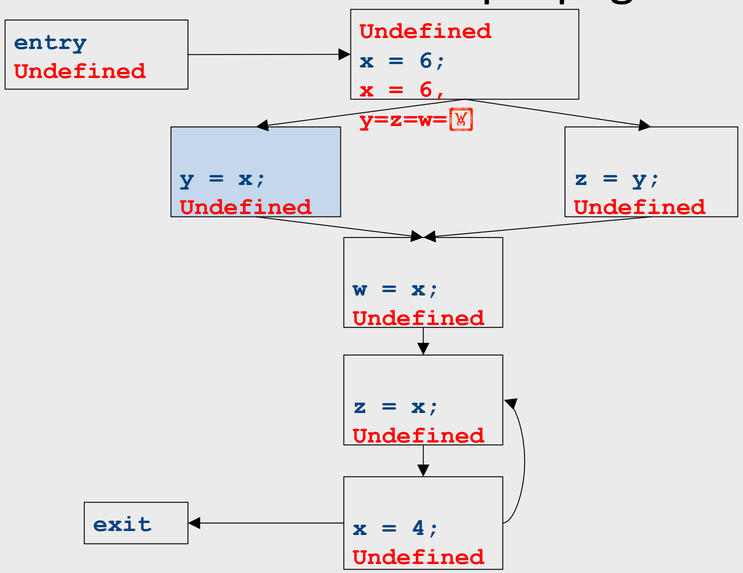
Global constant propagation



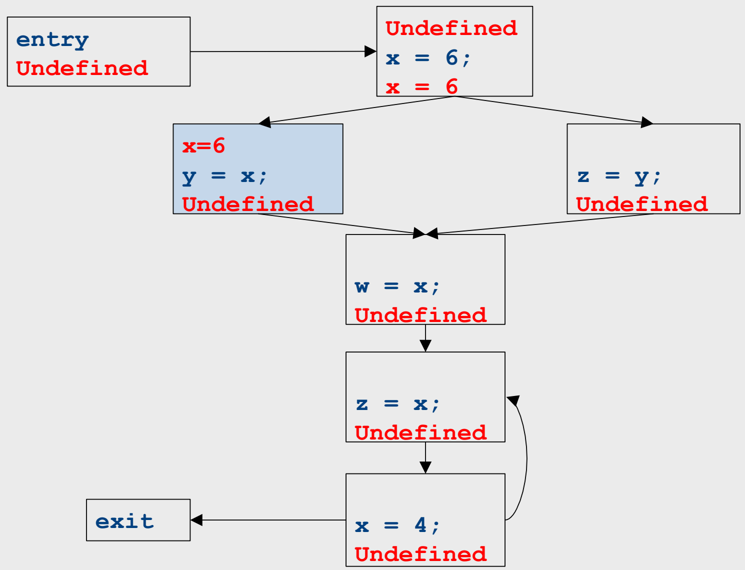
Global constant propagation



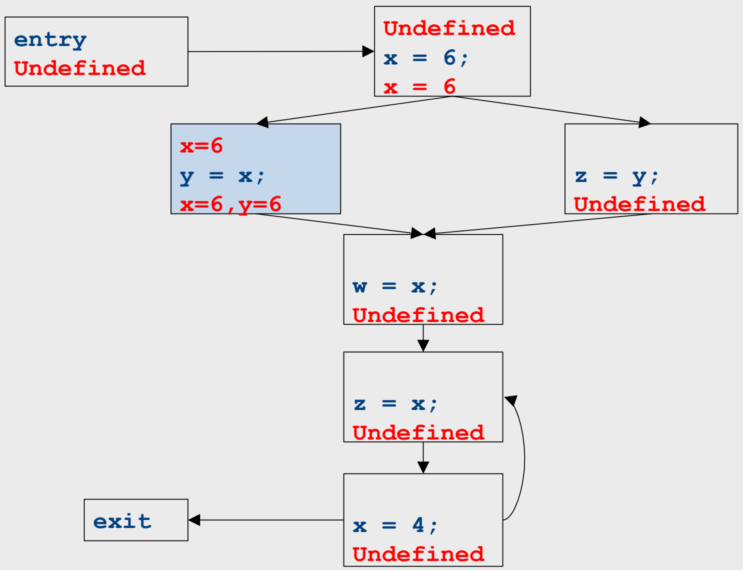
Global constant propagation



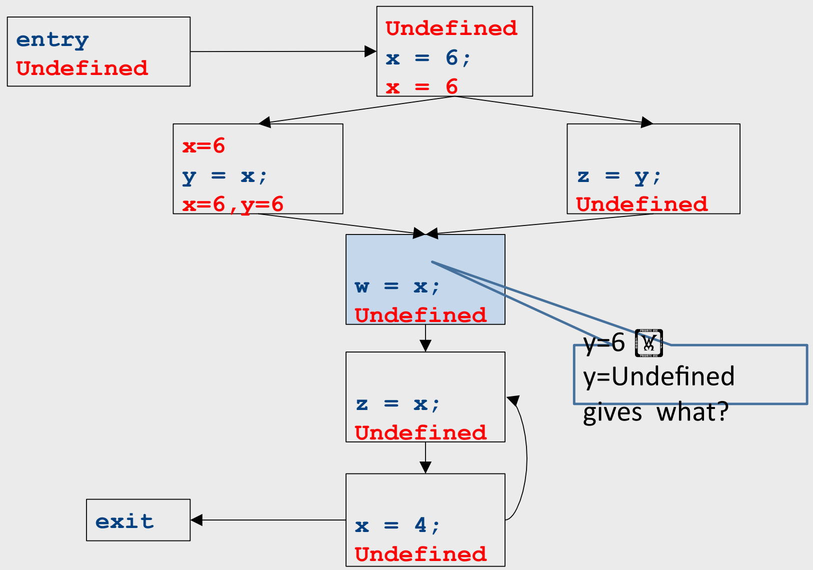
Global constant propagation



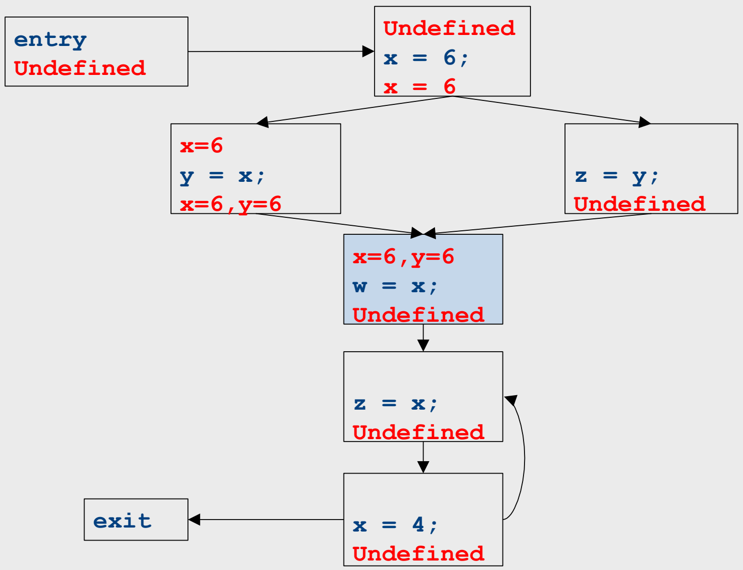
Global constant propagation



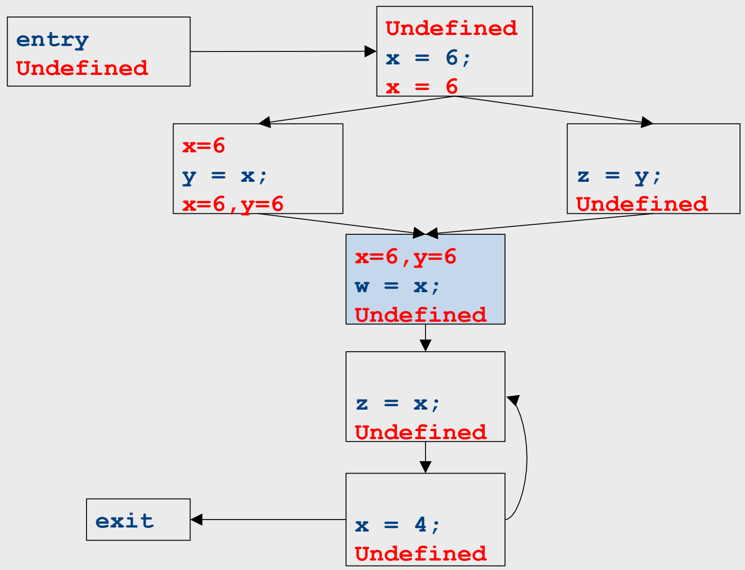
Global constant propagation



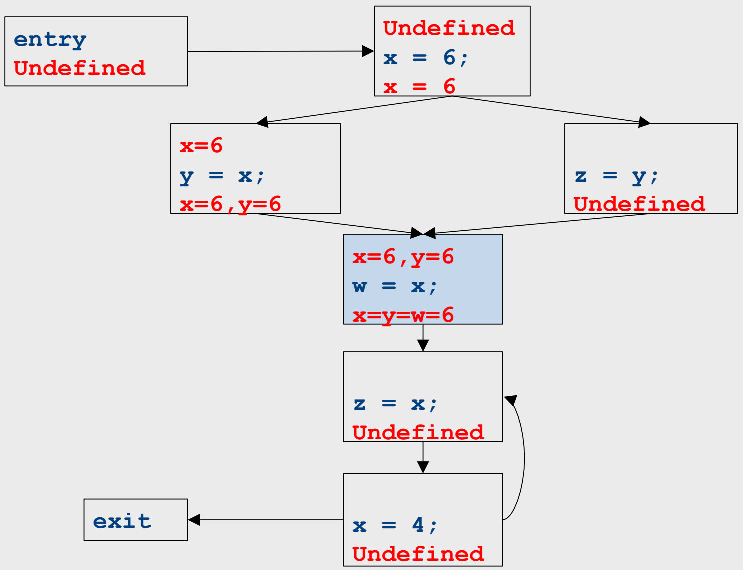
Global constant propagation



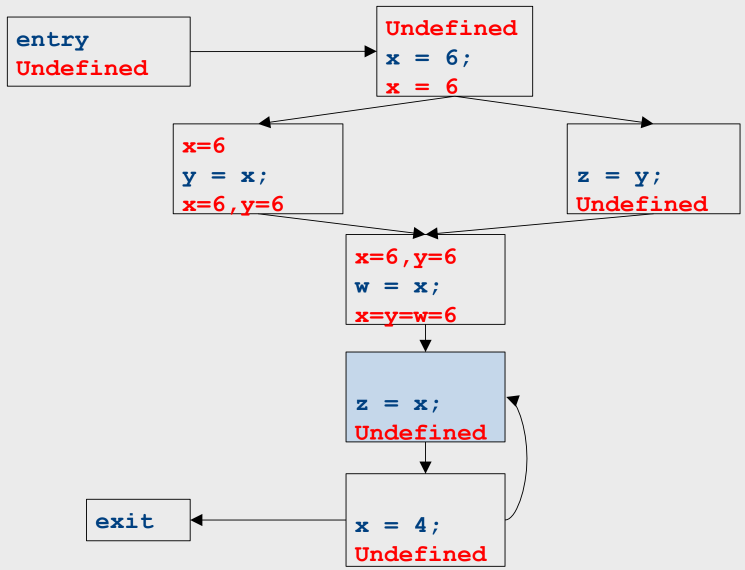
Global constant propagation



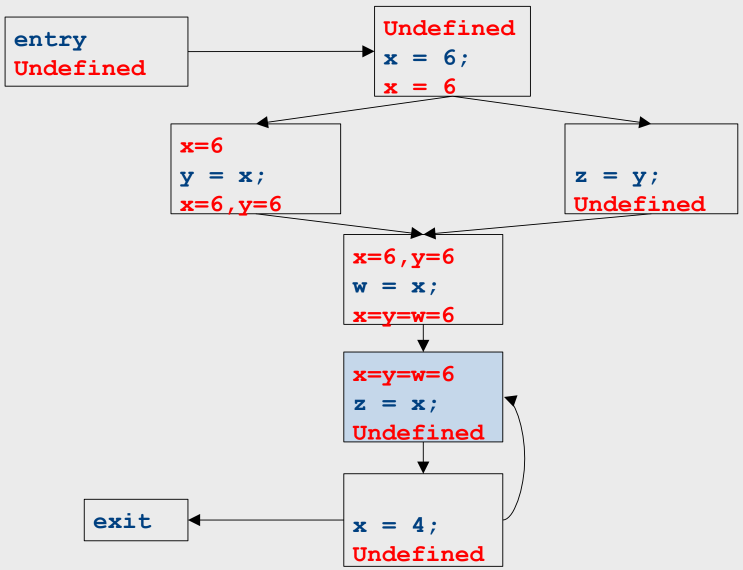
Global constant propagation



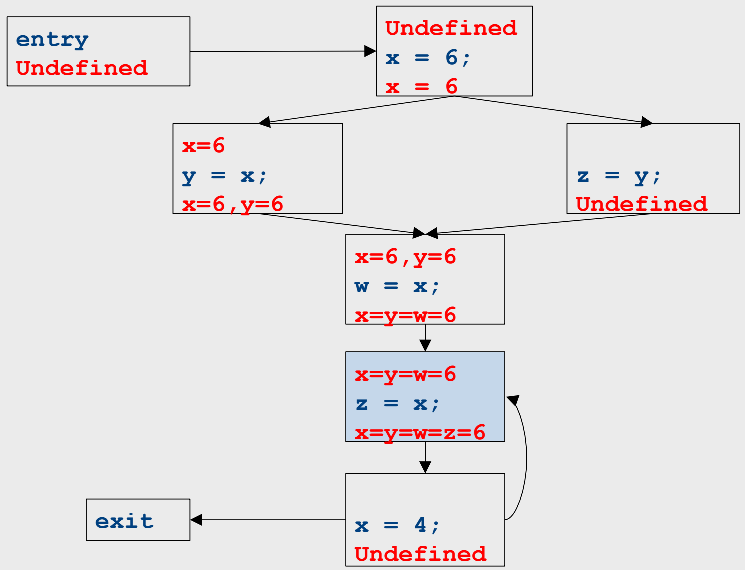
Global constant propagation



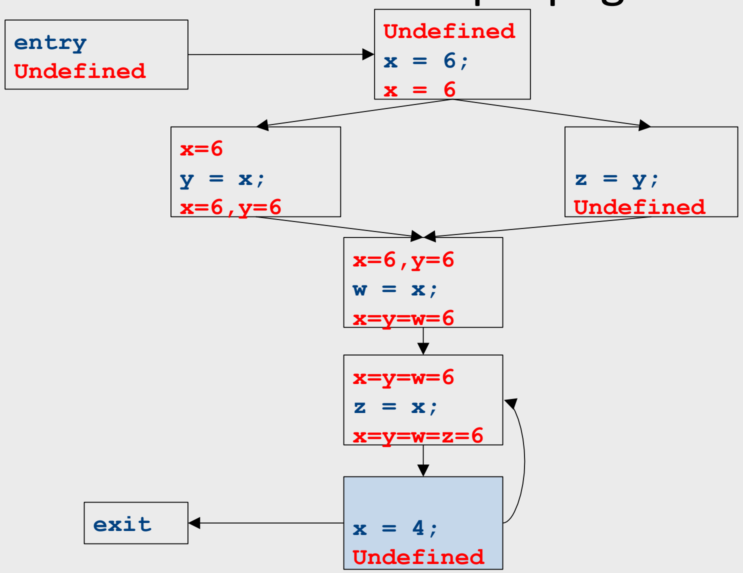
Global constant propagation



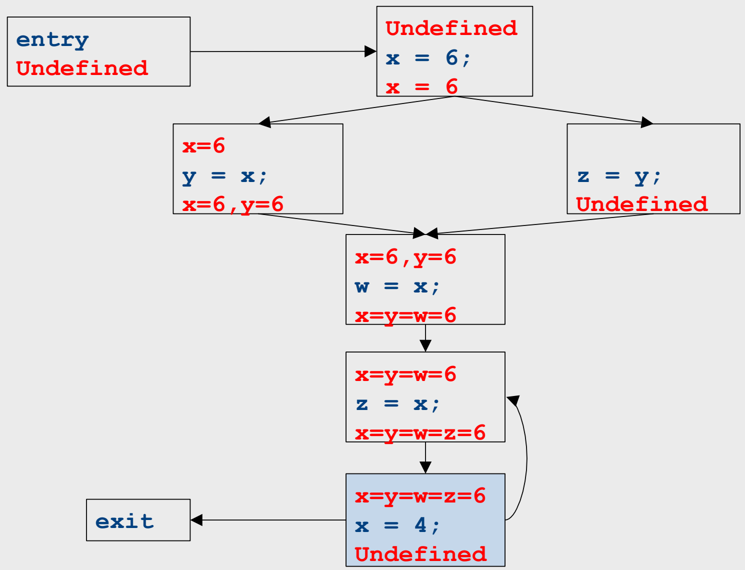
Global constant propagation



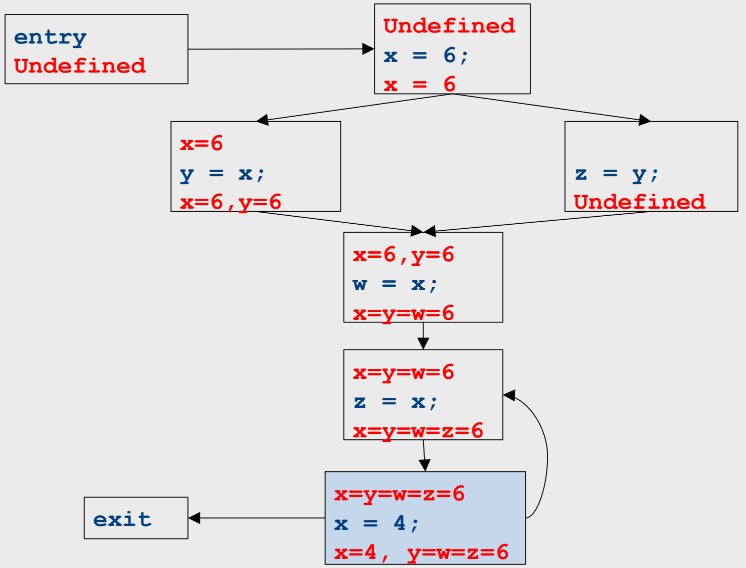
Global constant propagation



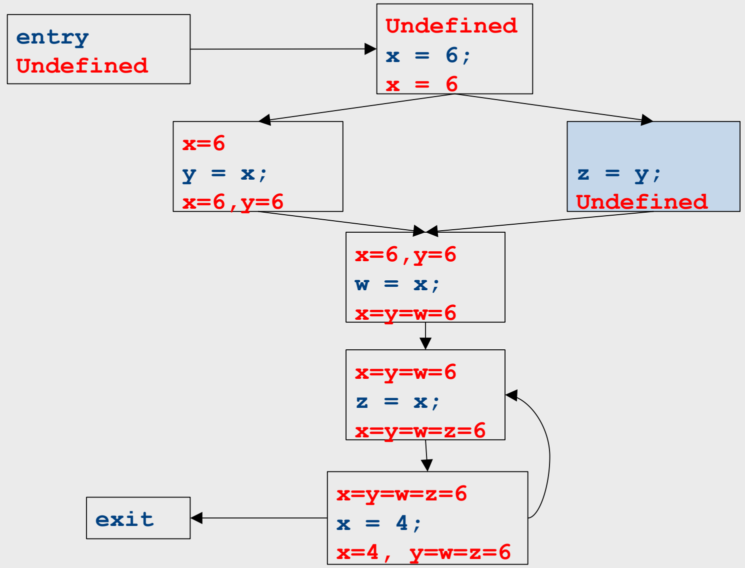
Global constant propagation



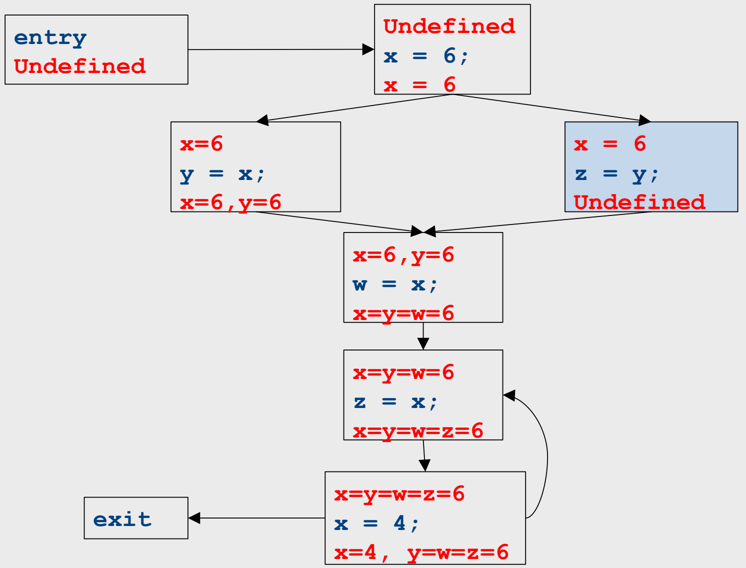
Global constant propagation



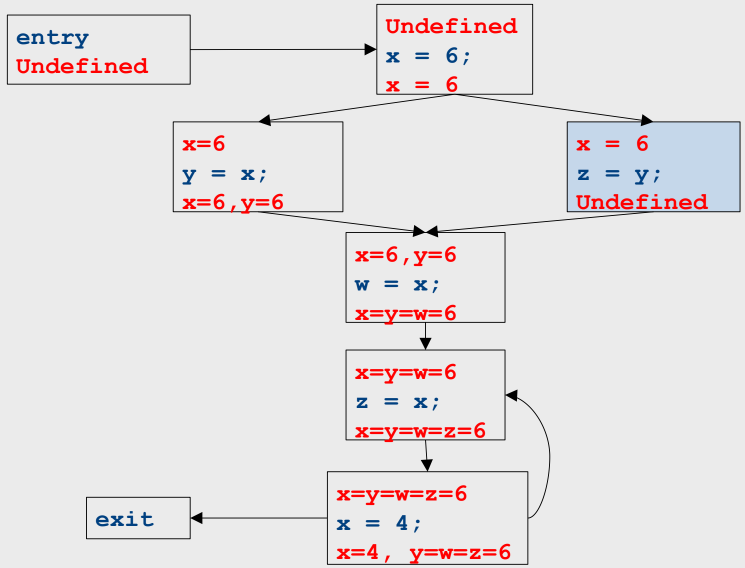
Global constant propagation



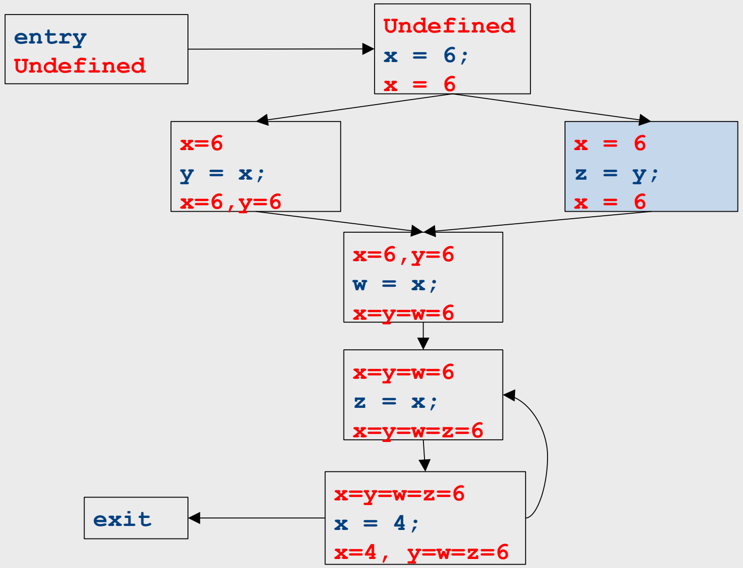
Global constant propagation



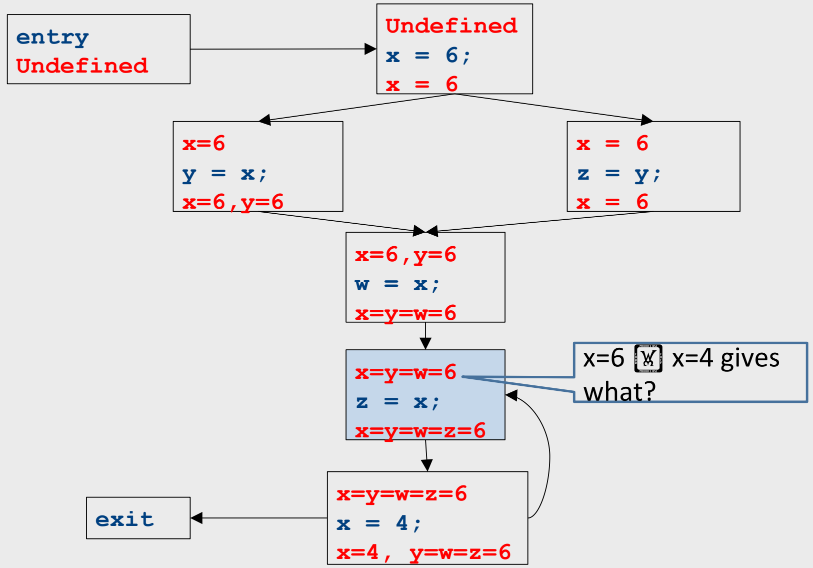
Global constant propagation



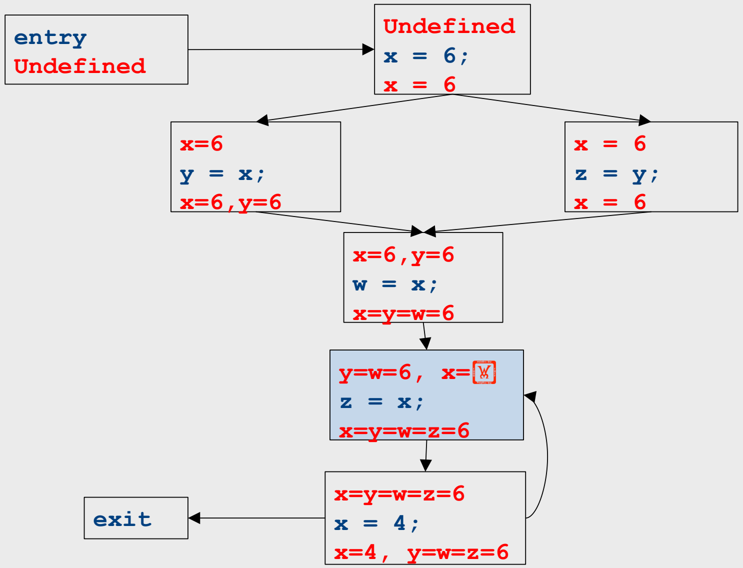
Global constant propagation



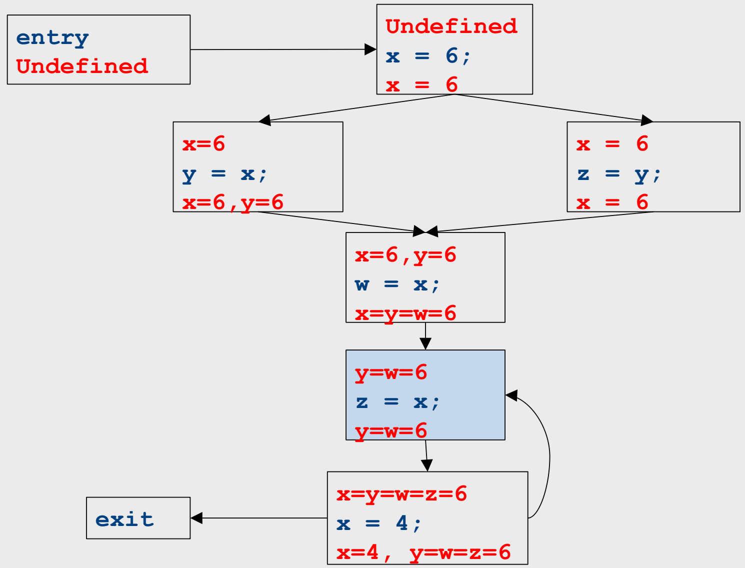
Global constant propagation



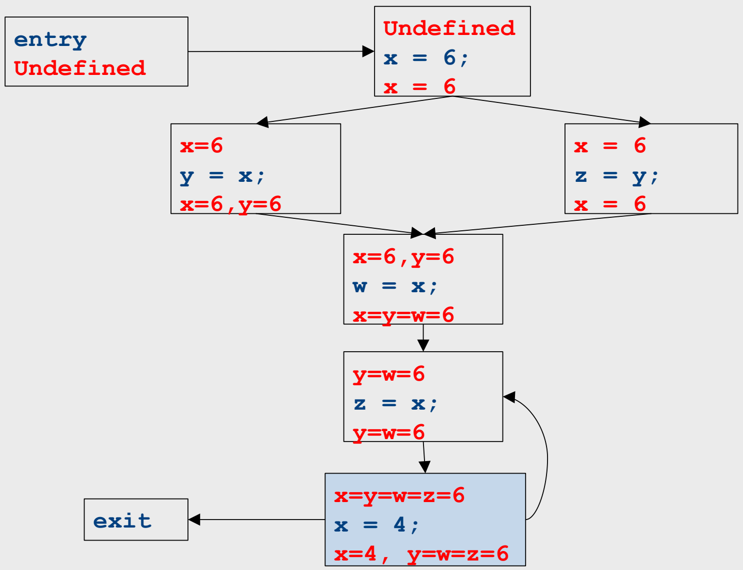
Global constant propagation



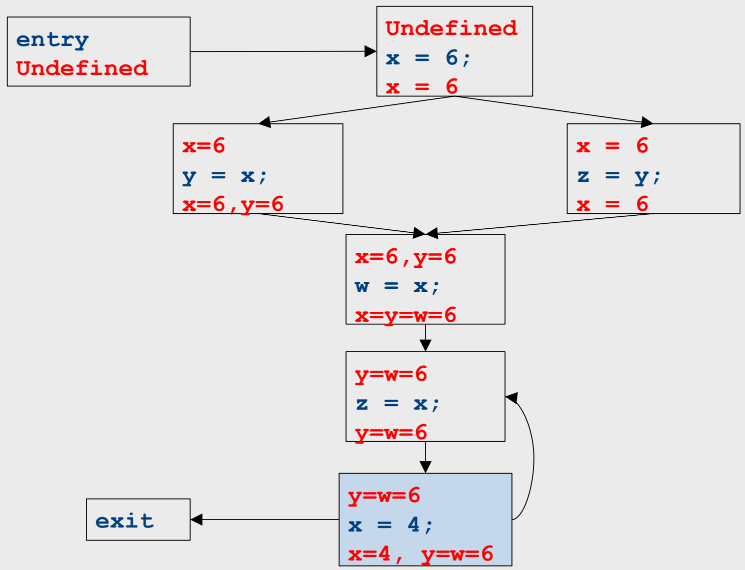
Global constant propagation



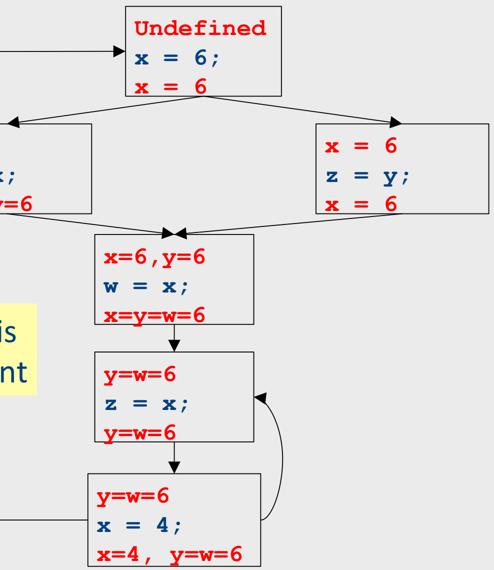
Global constant propagation



Global constant propagation

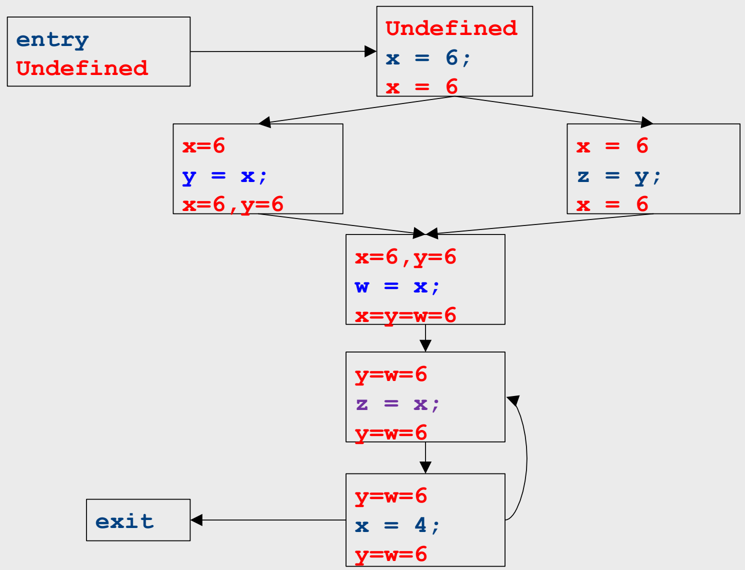


Global constant propagation

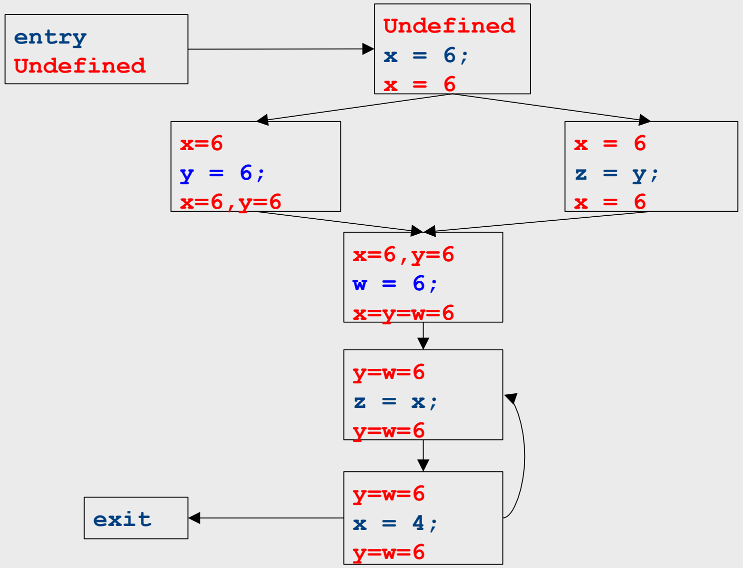


Global analysis reached fixpoint

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:
  - $f_{\mathbf{x=k}}(V) = V|_{x▯k}$ *(update V by mapping x to k)*
  - $f_{\mathbf{x=a+b}}(V) = V|_{x▯\text{Not-a-Constant}}$ *(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, ⊠), where
- V is a domain of elements
- ⊠ is a join operator that is
  - commutative: x ⊠ y = y ⊠ x
  - associative: (x ⊠ y) ⊠ z = x ⊠ (y ⊠ z)
  - idempotent: x ⊠ x = x
- If x ⊠ 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 ⊠ such that ⊠ ⊠ x = x for all x

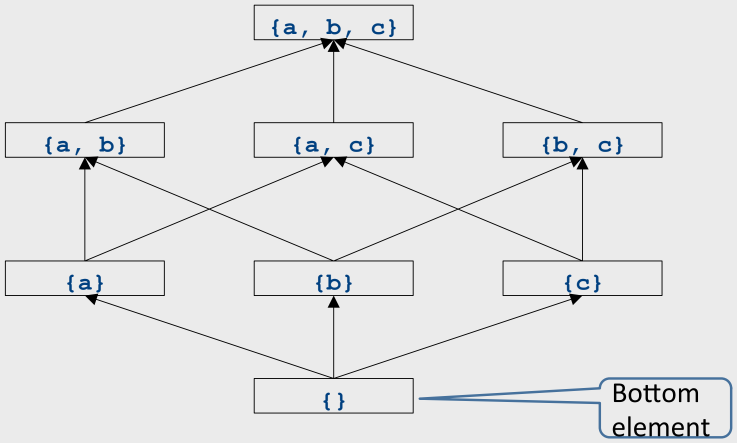
Partial ordering induced by join

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

A join semilattice for liveness

- Sets of live variables and the set union operation
- Idempotent:
  - x ⊠ x = x
- Commutative:
  - x ⊠ y = y ⊠ x
- Associative:
  - (x ⊠ y) ⊠ z = x ⊠ (y ⊠ z)
- Bottom element:
  - The empty set: Ø ⊠ 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 : **V** ⊠ **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

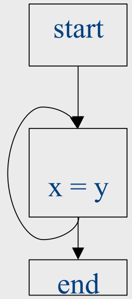
Proving termination

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

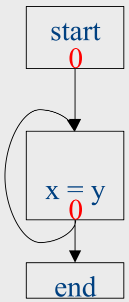
A non-terminating analysis

- 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

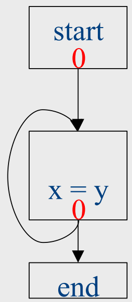
A non-terminating analysis



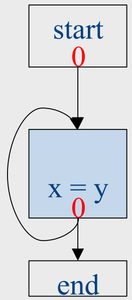
Initialization



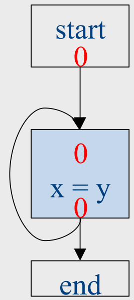
Fixed-point iteration



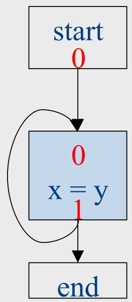
Choose a block



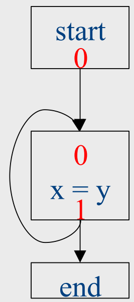
Iteration 1



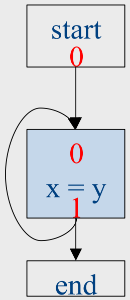
Iteration 1



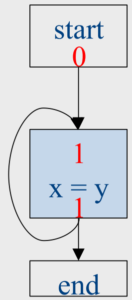
Choose a block



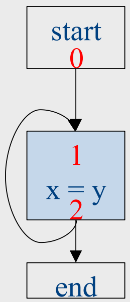
Iteration 2



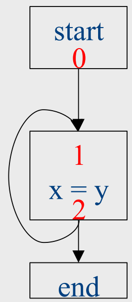
Iteration 2



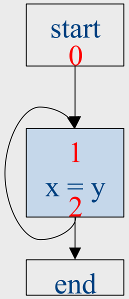
Iteration 2



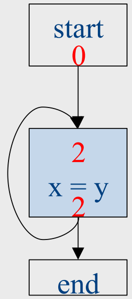
Choose a block



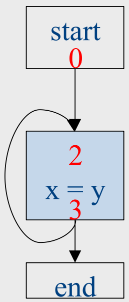
Iteration 3



Iteration 3

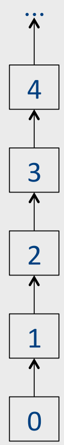


Iteration 3



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



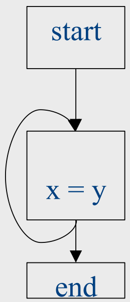
Height of a lattice

- An increasing chain is a sequence of elements ⊠ ⊠ a_1 ⊠ a_2 ⊠ … ⊠ 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:
  - {} ⊠ $\{v_1\}$ ⊠ $\{v_1,v_2\}$ ⊠ … ⊠ $\{v_1,…,v_n\}$
- 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*⊠n^3

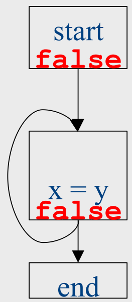
Another non-terminating analysis

- 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`**

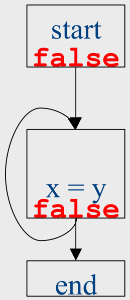
A non-terminating analysis



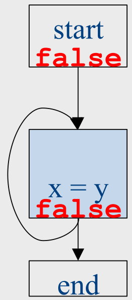
Initialization



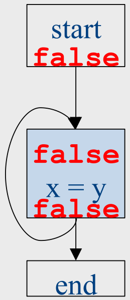
Fixed-point iteration



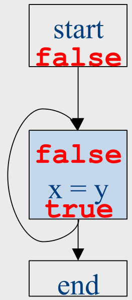
Choose a block



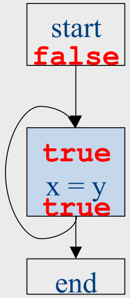
Iteration 1



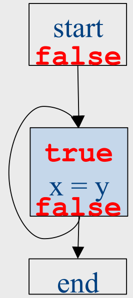
Iteration 1



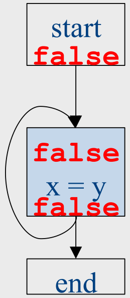
Iteration 2



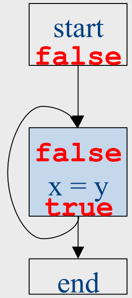
Iteration 2



Iteration 3

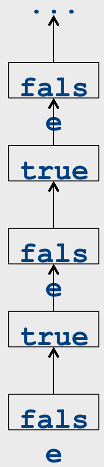


Iteration 3



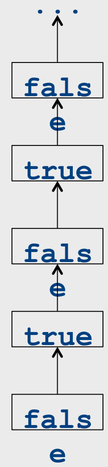
Why doesn't it terminate?

- 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 □ y, then *f*(x) □ *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 □ *f*(x)
  - (This is a different property called extensivity)

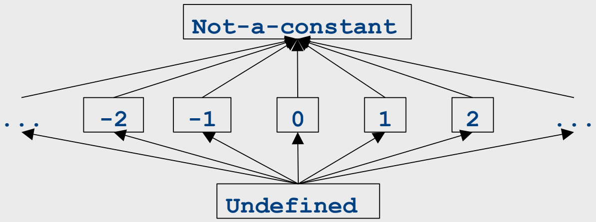
Liveness and monotonicity

- A transfer function *f* is monotone iff
  if x □ y, then *f*(x) □ *f*(y)
- Recall our transfer function for **a = b + c** is
  - $f_{a = b + c}$(V) = (V – {a}) □ {b, c}
- Recall that our join operator is set union and induces an ordering relationship
  X □ Y iff X □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
  - $f_{x=k}(V) = V|_{x \mapsto k}$ *(update V by mapping x to k)*
  - $f_{x=a+b}(V) = V|_{x \mapsto \text{Not-a-Constant}}$ *(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

- A transfer function *f* is distributive if

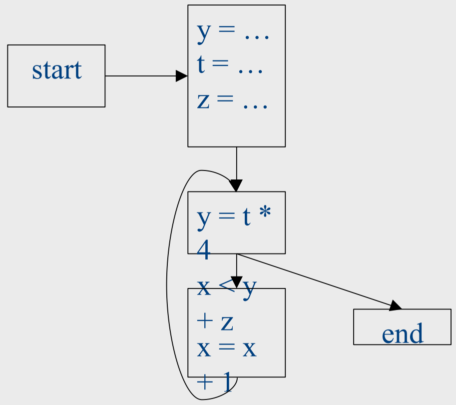
  f(*a* $\sqcup$ *b*) = *f*(*a*) $\sqcup$ *f*(b)

  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?

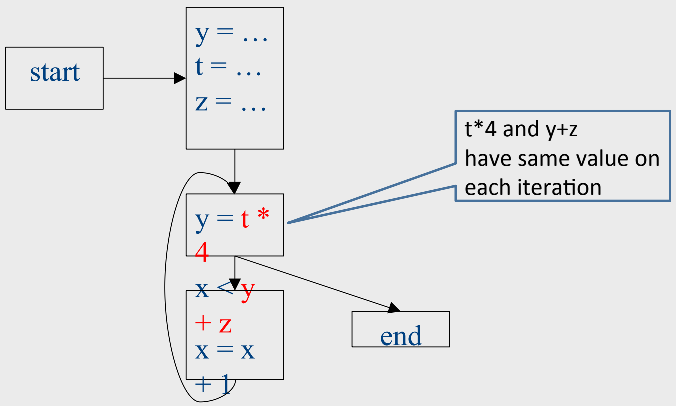
Loop optimizations

- 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)

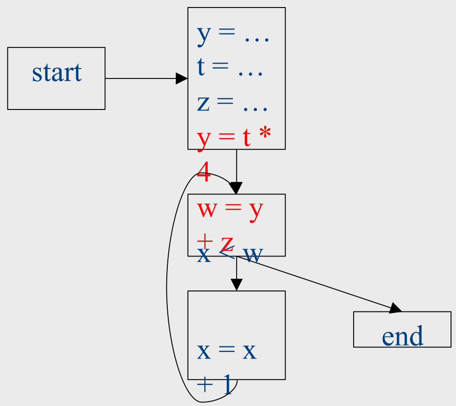
Loop invariant computation



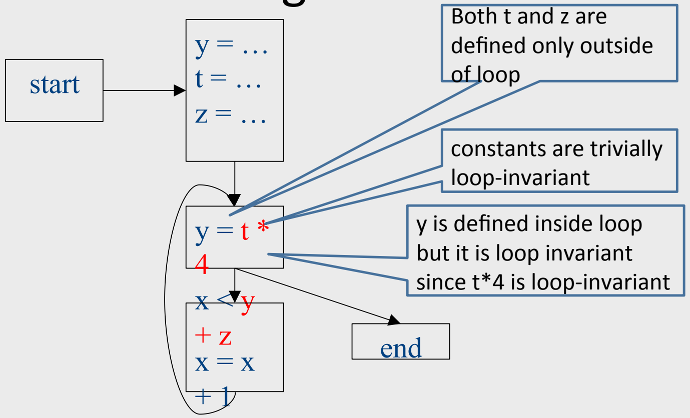
Loop invariant computation



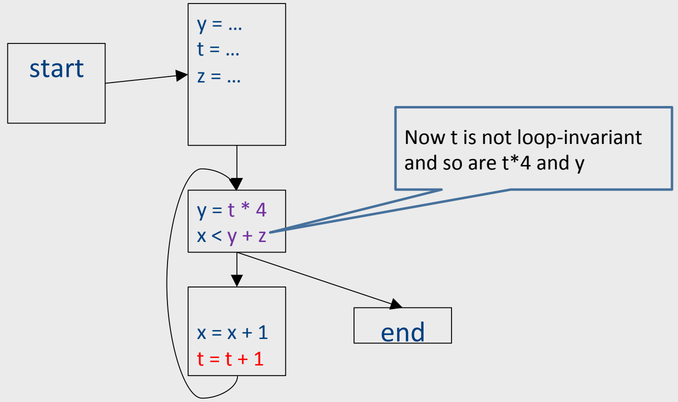
Code hoisting



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

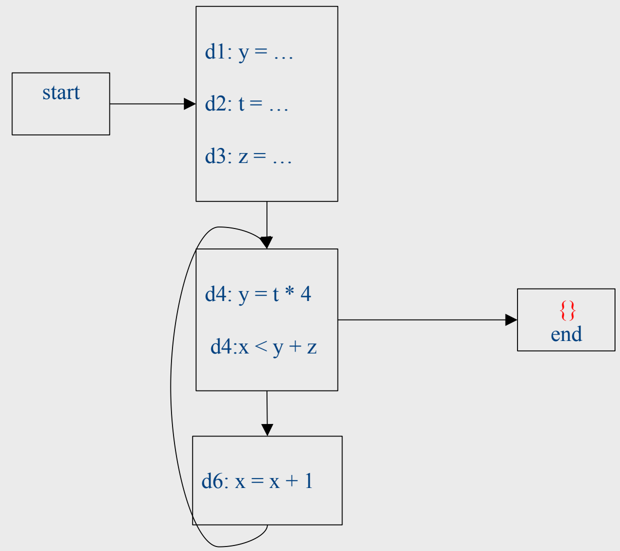
- A definition *d*: t = … reaches a program location if there is a path from the definition to the program location, along which the defined variable is never redefined
- Direction: Forward
- Domain: sets of program locations that are definitions `
- Join operator: union
- Transfer function:

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

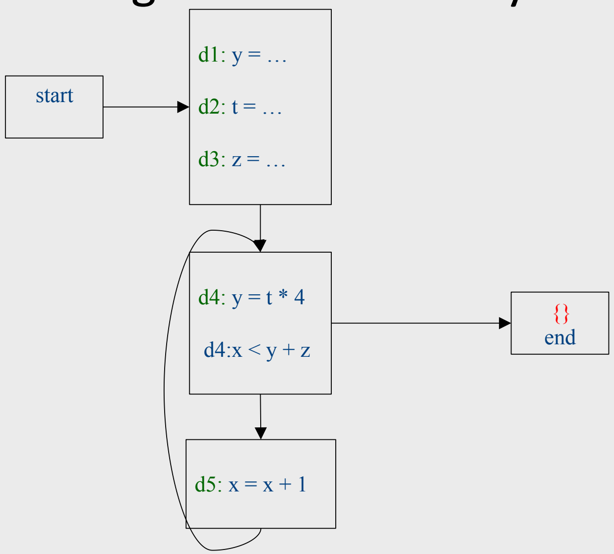
  $f_{d:\ not\text{-}a\text{-}def}(RD) = RD$

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

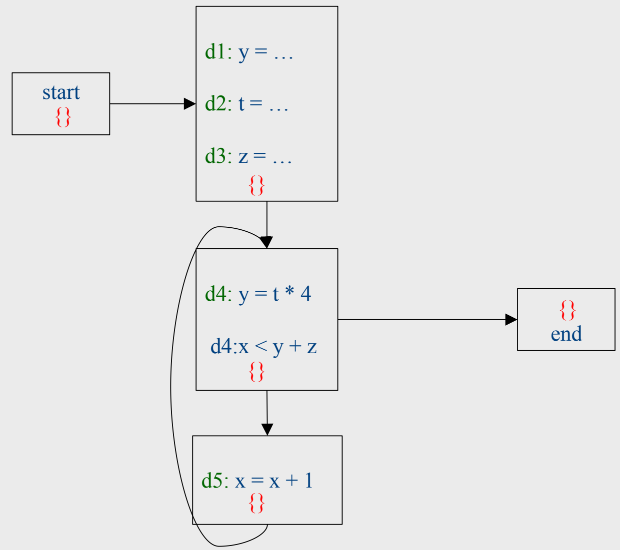
Reaching definitions analysis



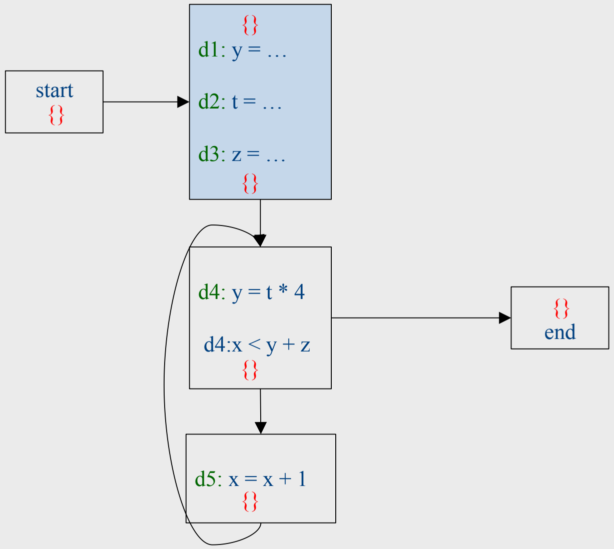
Reaching definitions analysis



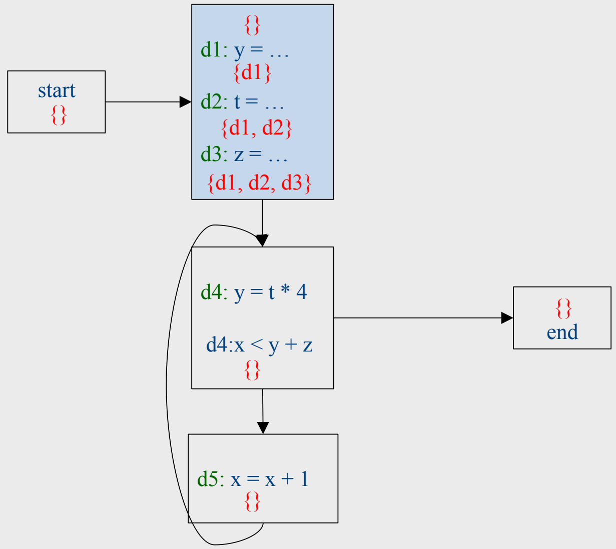
Initialization



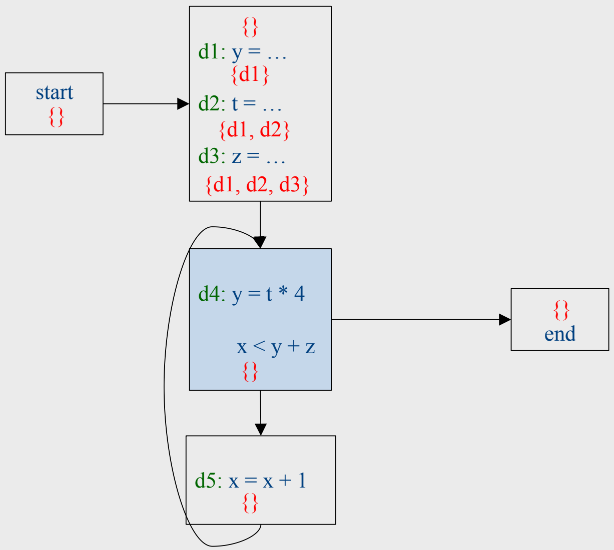
Iteration 1



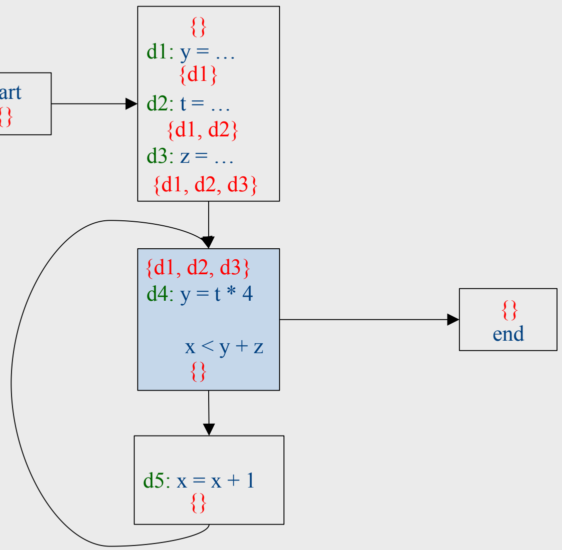
Iteration 1



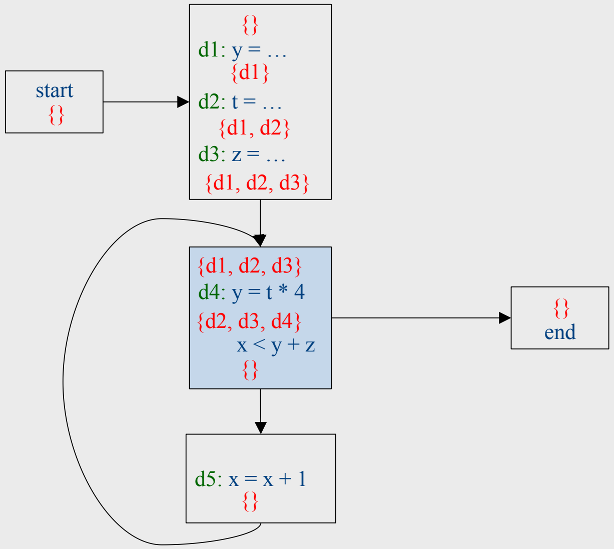
Iteration 2



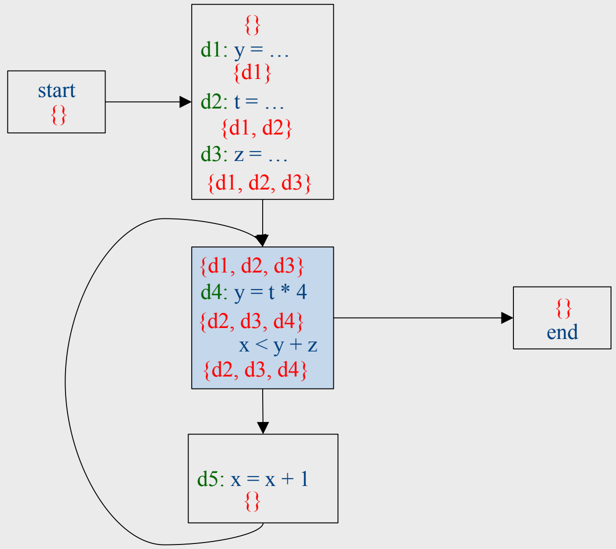
Iteration 2



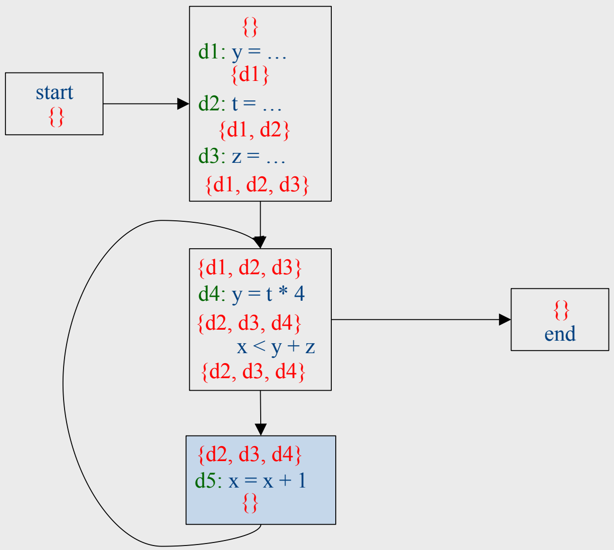
Iteration 2



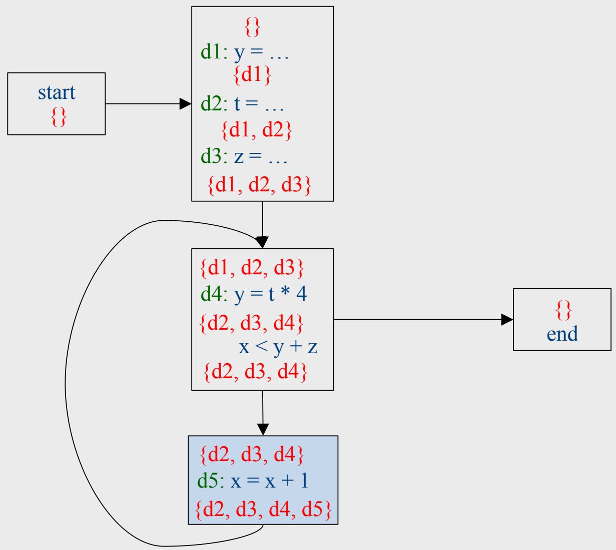
Iteration 2



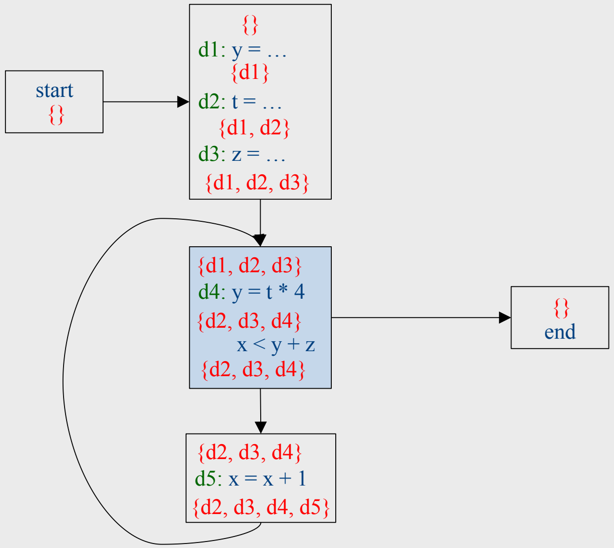
Iteration 3



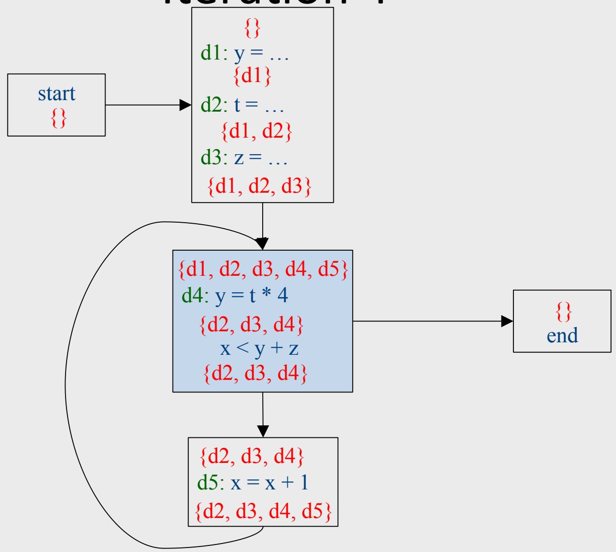
Iteration 3



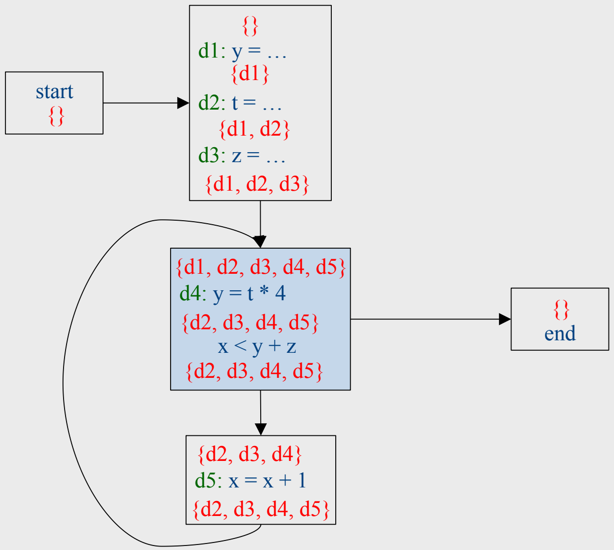
Iteration 4



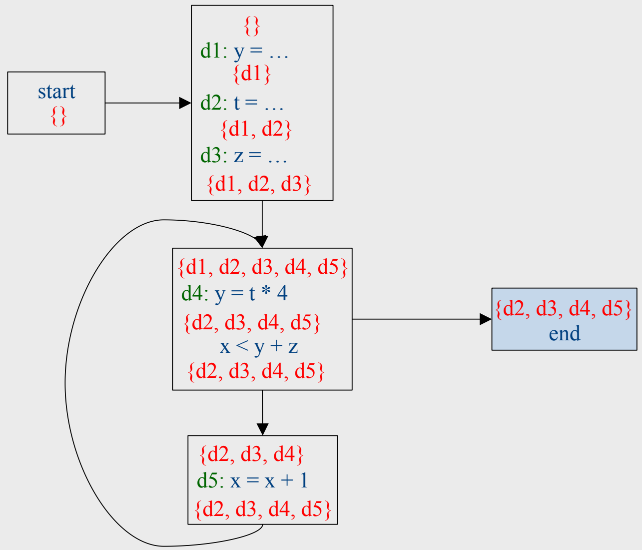
Iteration 4



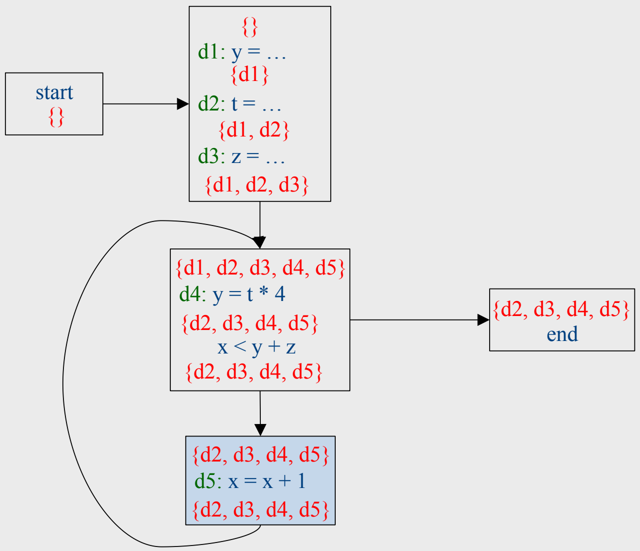
Iteration 4



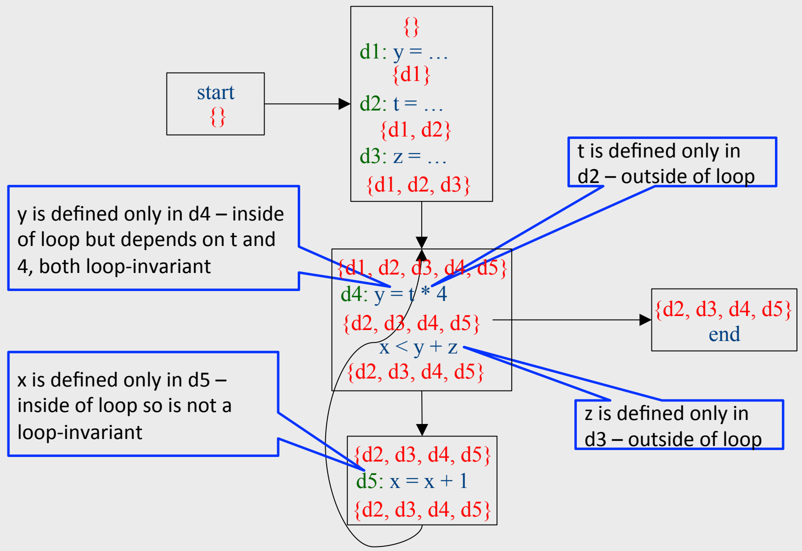
Iteration 5



Iteration 6



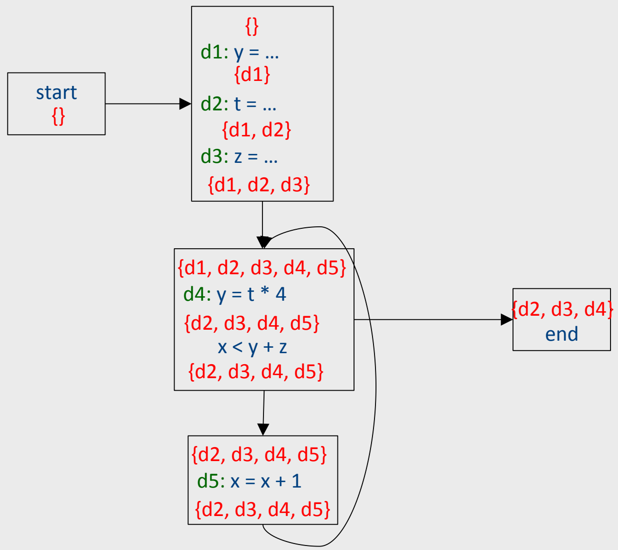
Which expressions are loop invariant?



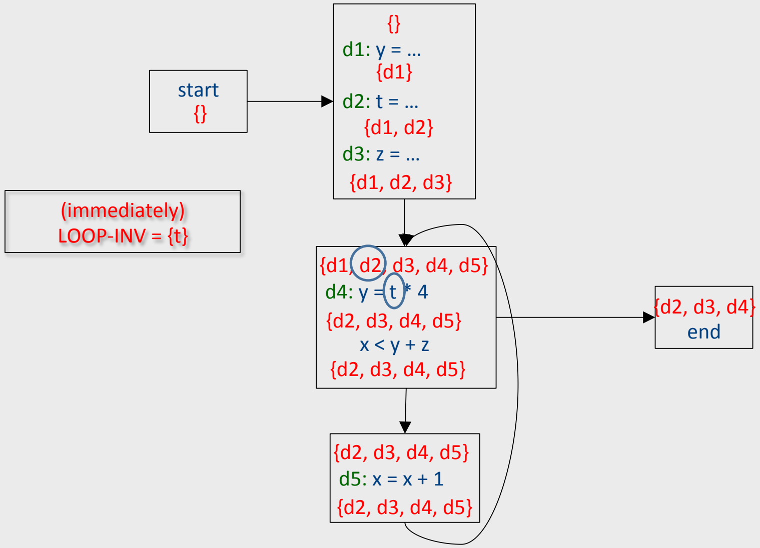
Inferring loop-invariant expressions

- 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

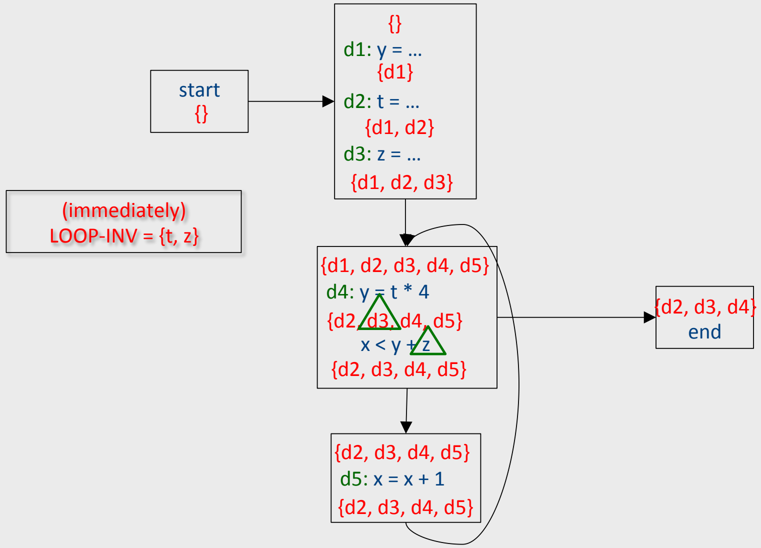
Computing LOOP-INV



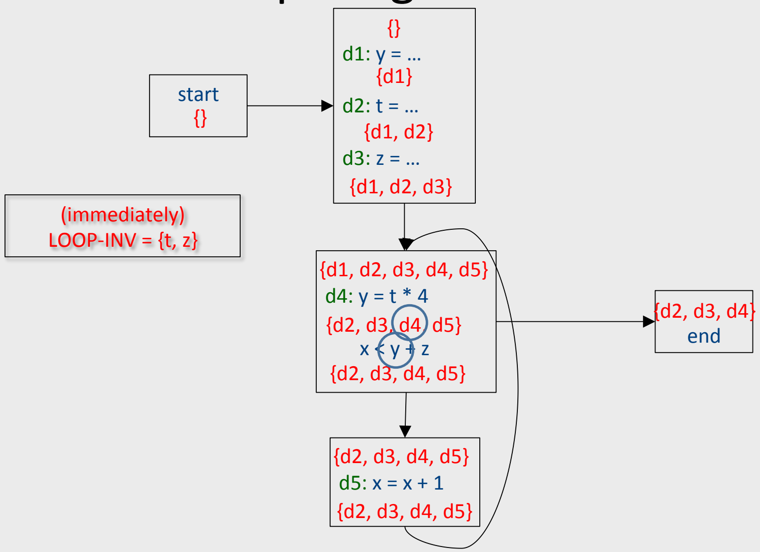
Computing LOOP-INV



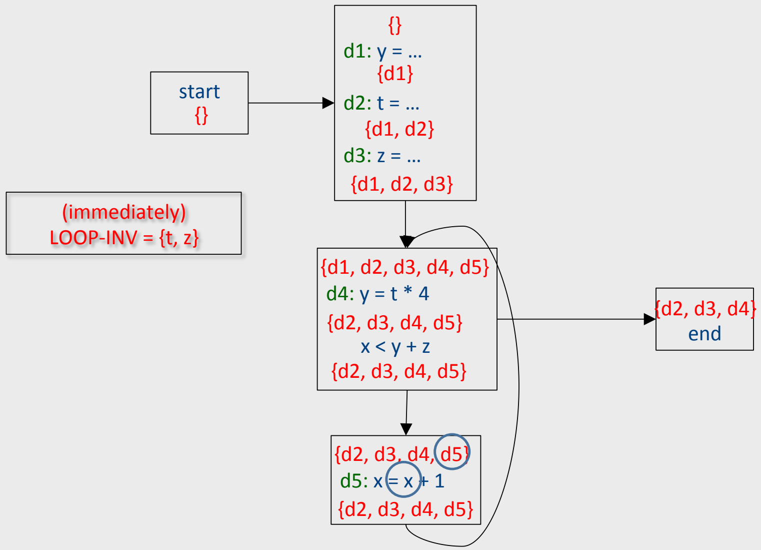
Computing LOOP-INV



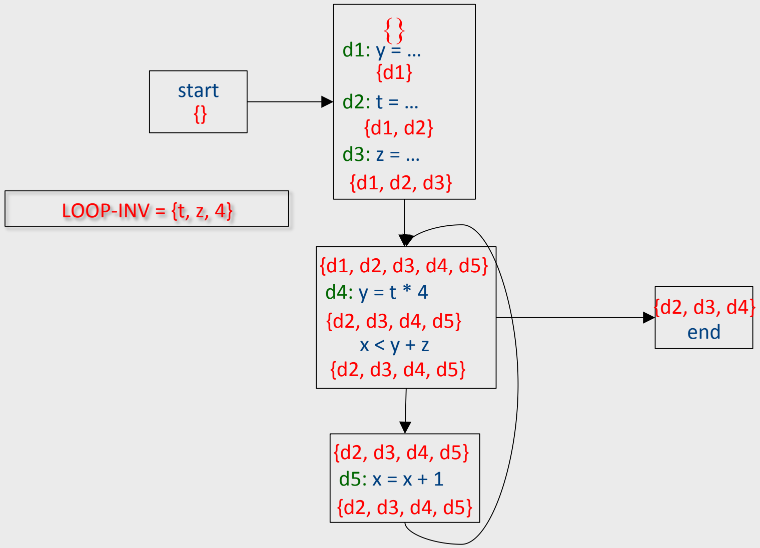
Computing LOOP-INV



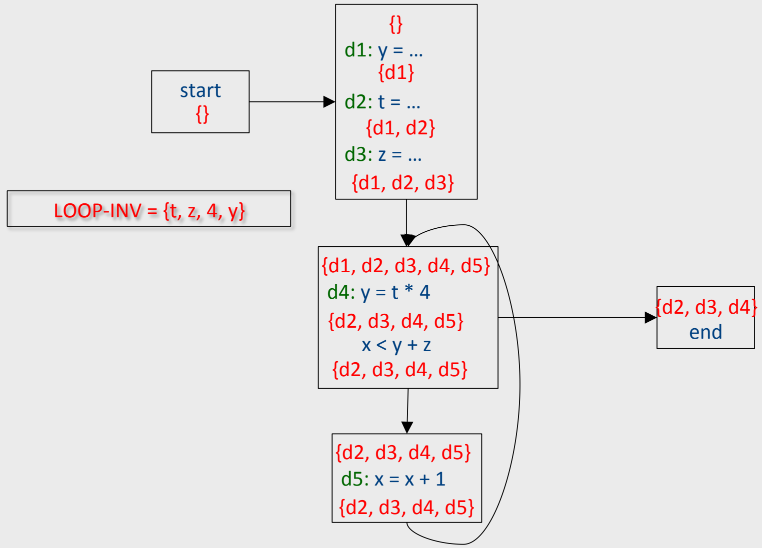
Computing LOOP-INV



Computing LOOP-INV



Computing LOOP-INV



Induction variables

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

Summary of optimizations

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