

CSC410

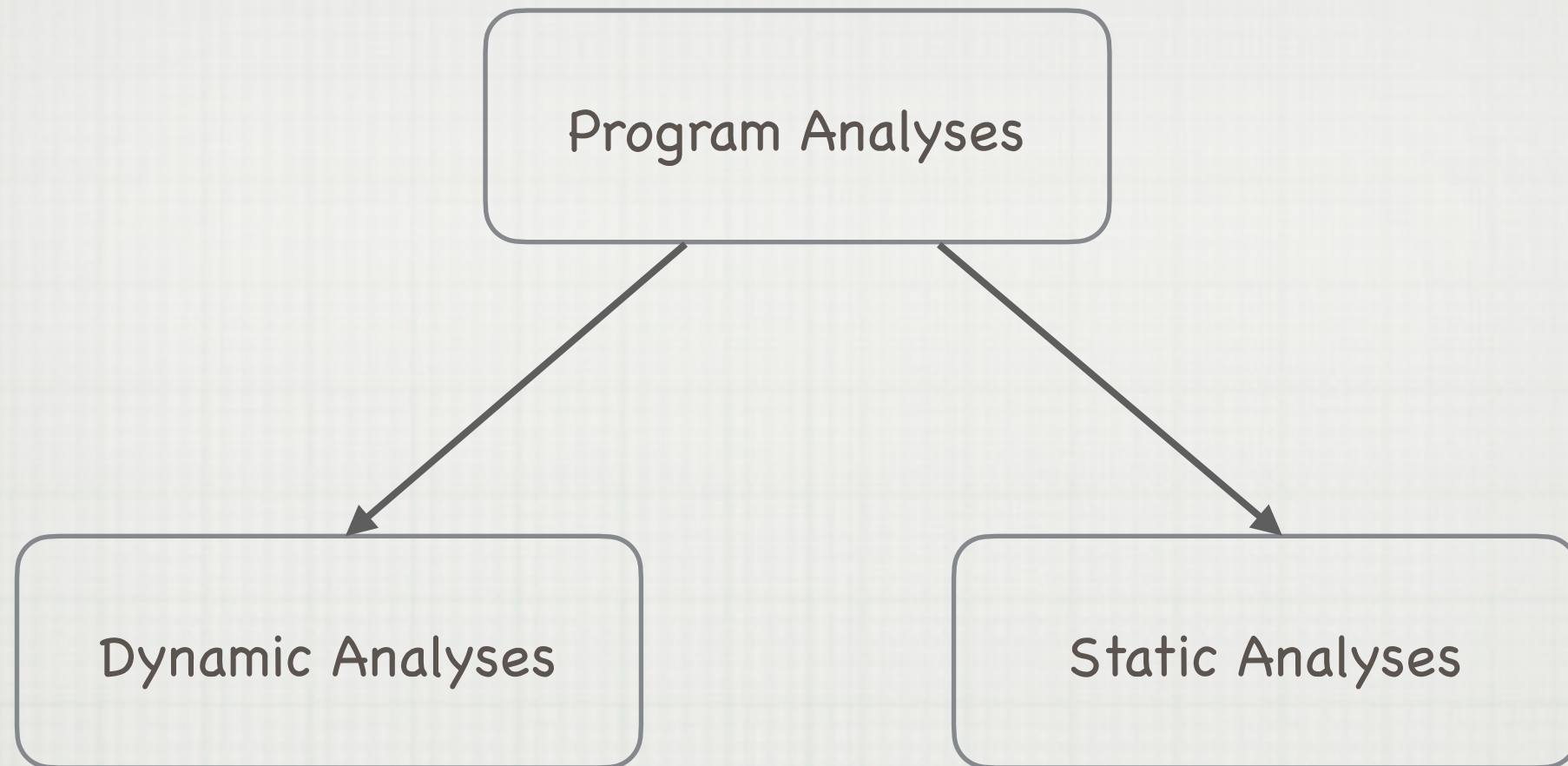
Data Flow Analyses

AZADEH FARZAN

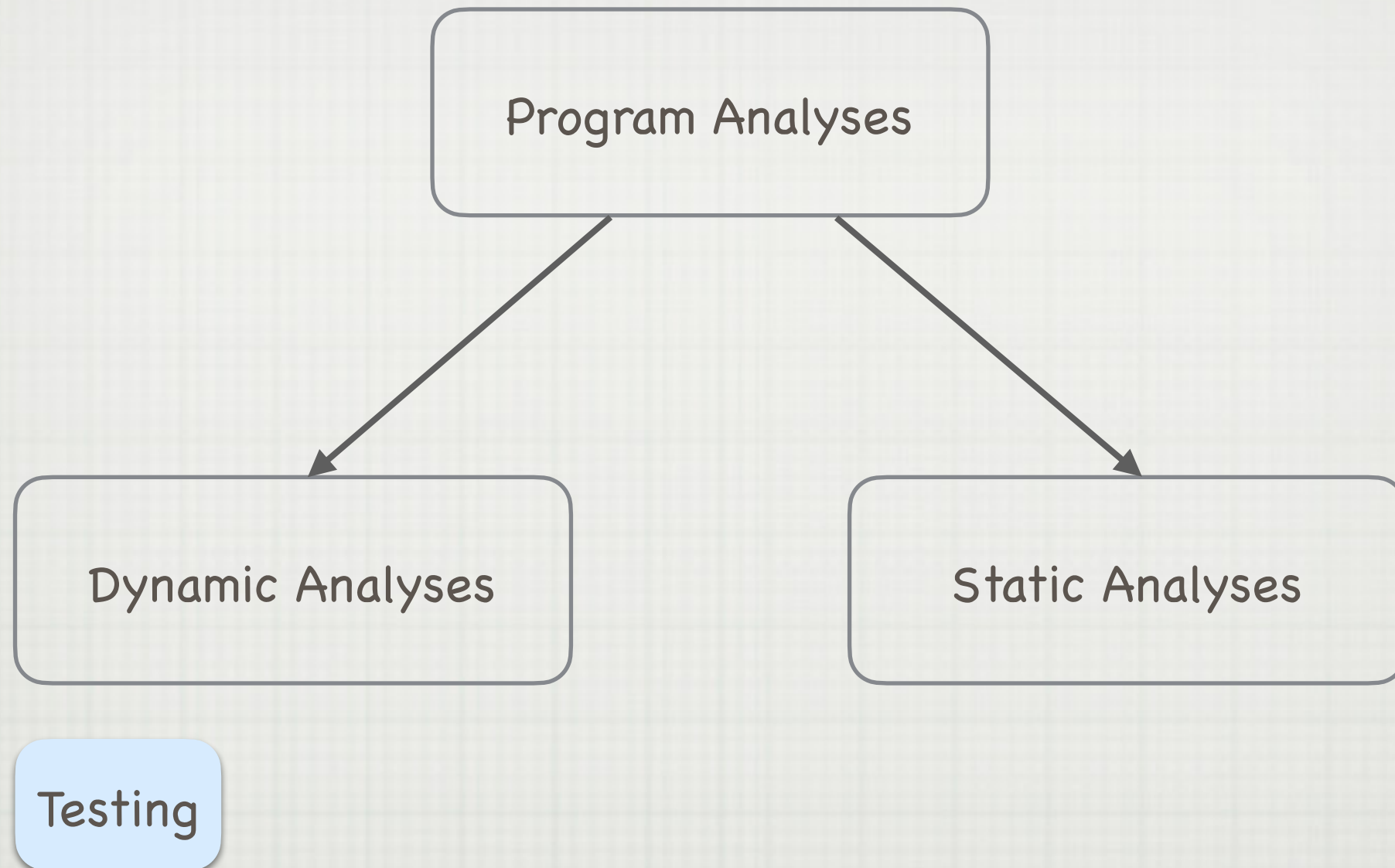
FALL 2023

What is program analysis?

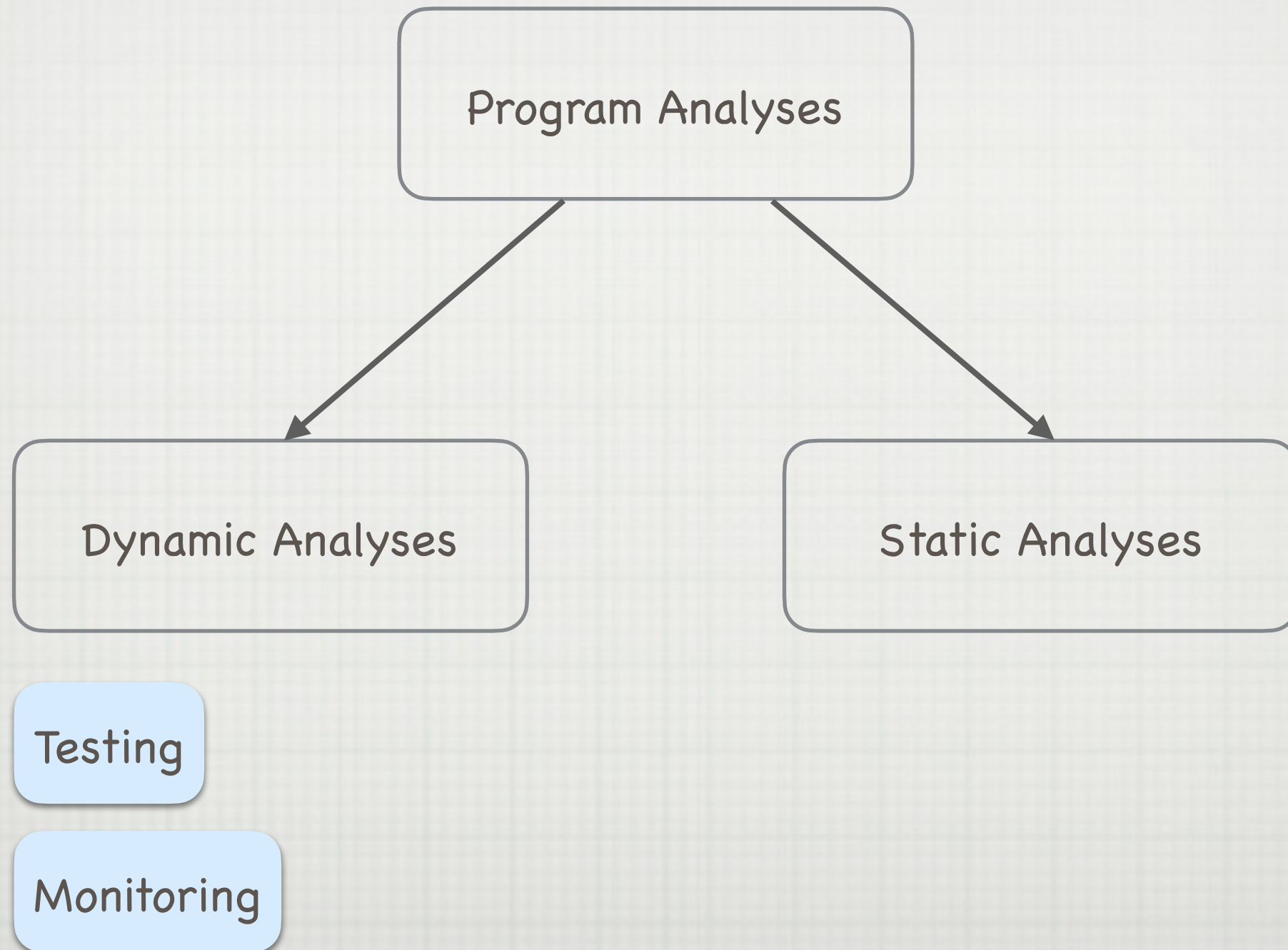
Program Analyses



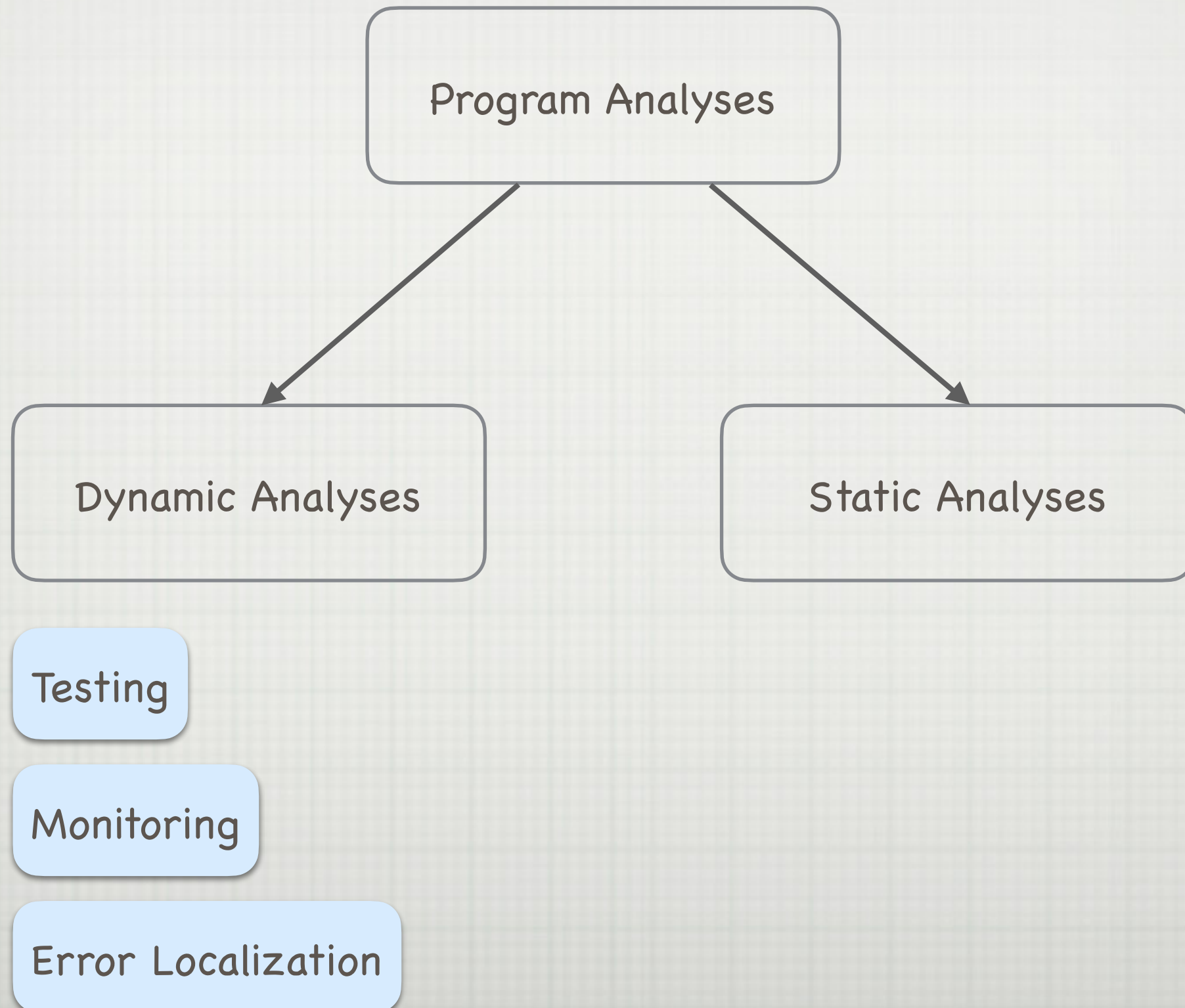
Program Analyses



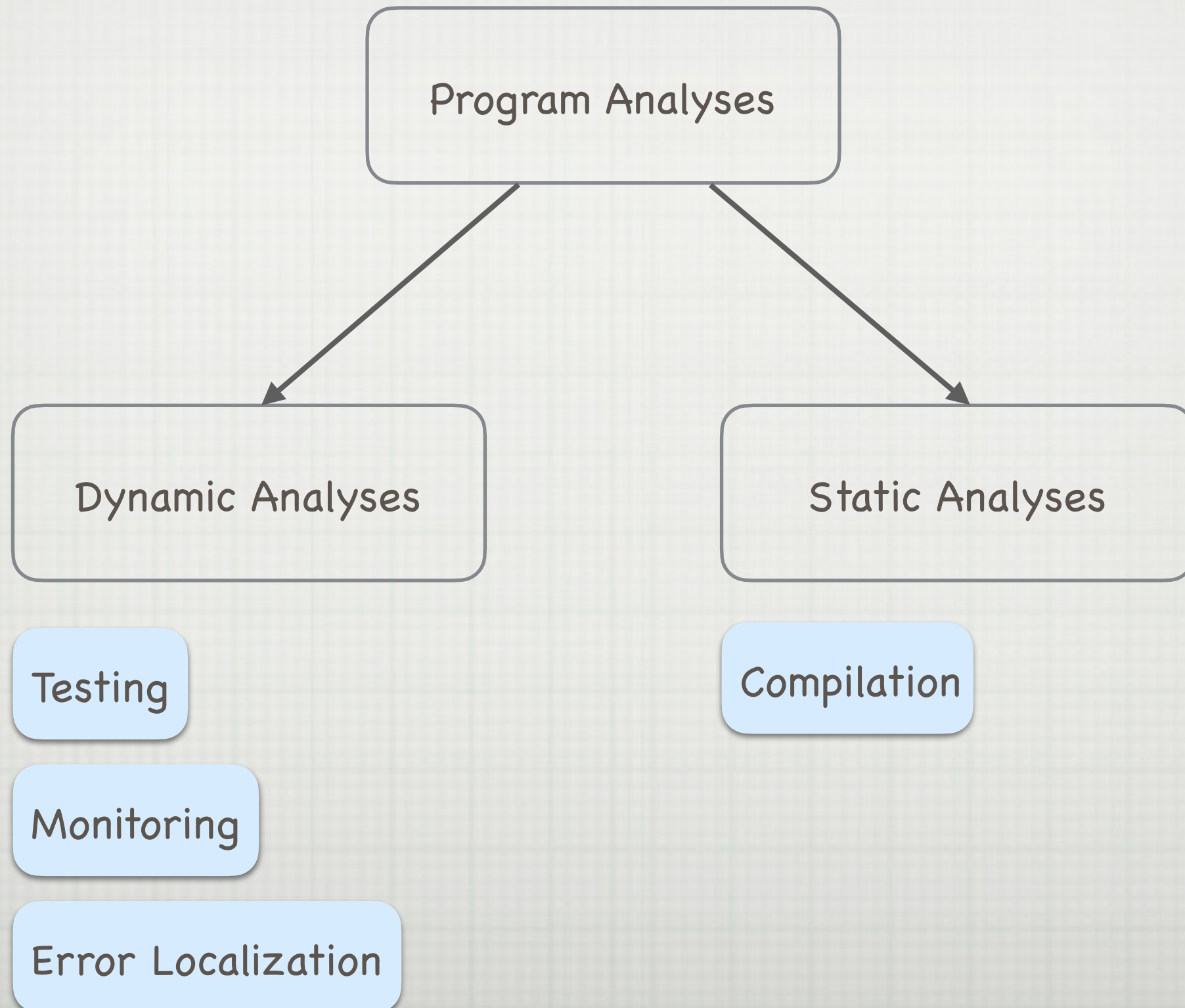
Program Analyses



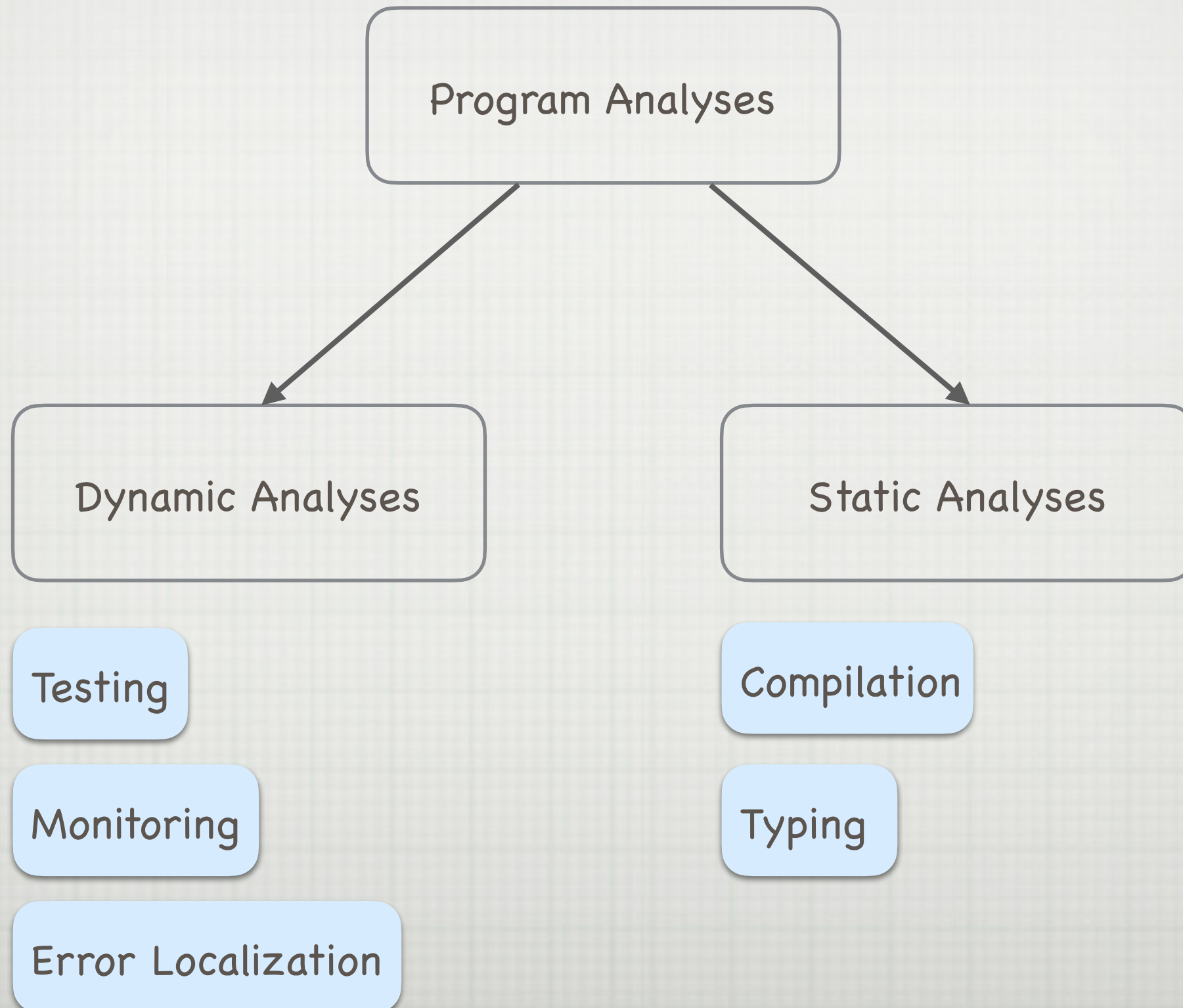
Program Analyses



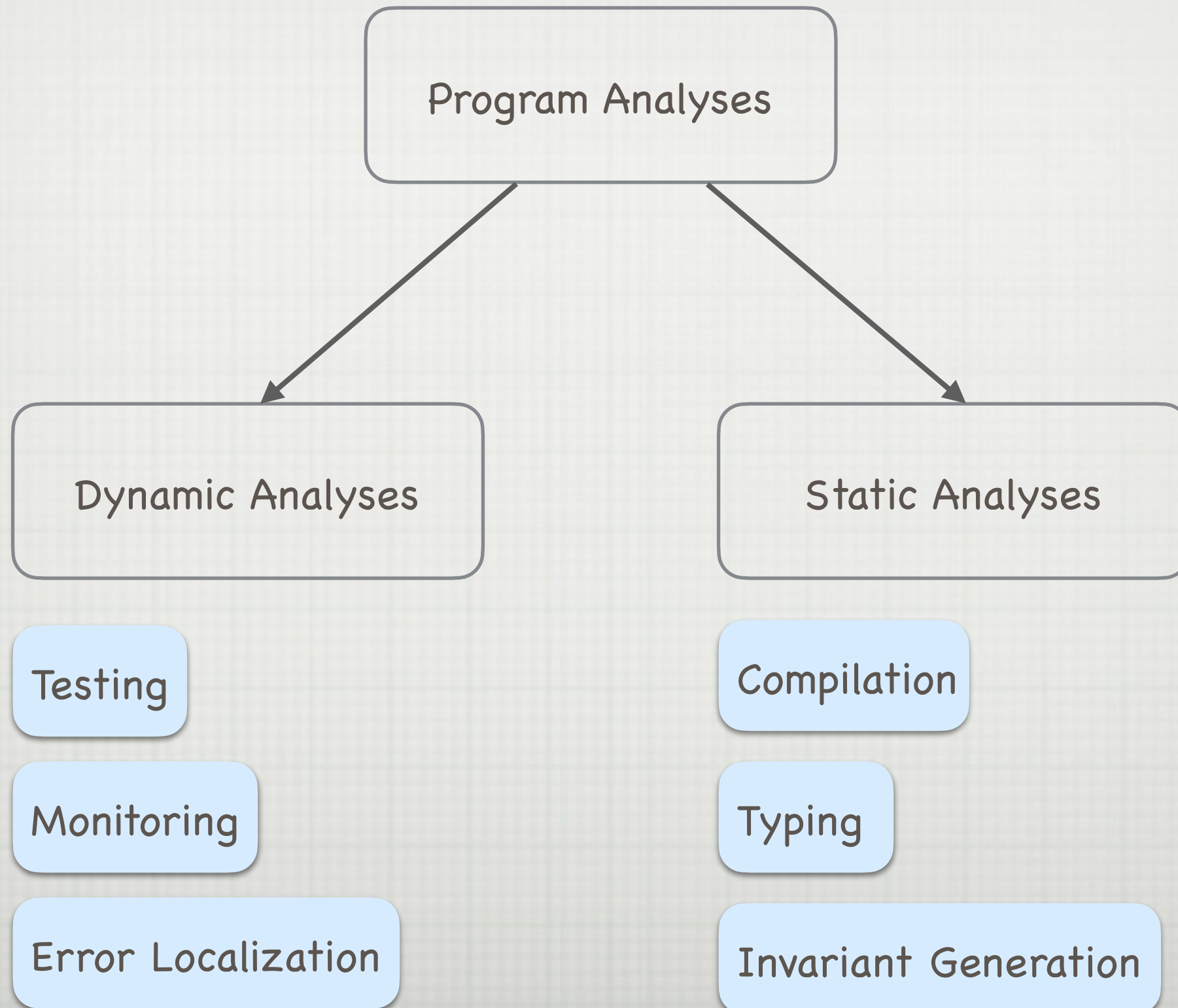
Program Analyses



Program Analyses



Program Analyses



Program Analyses

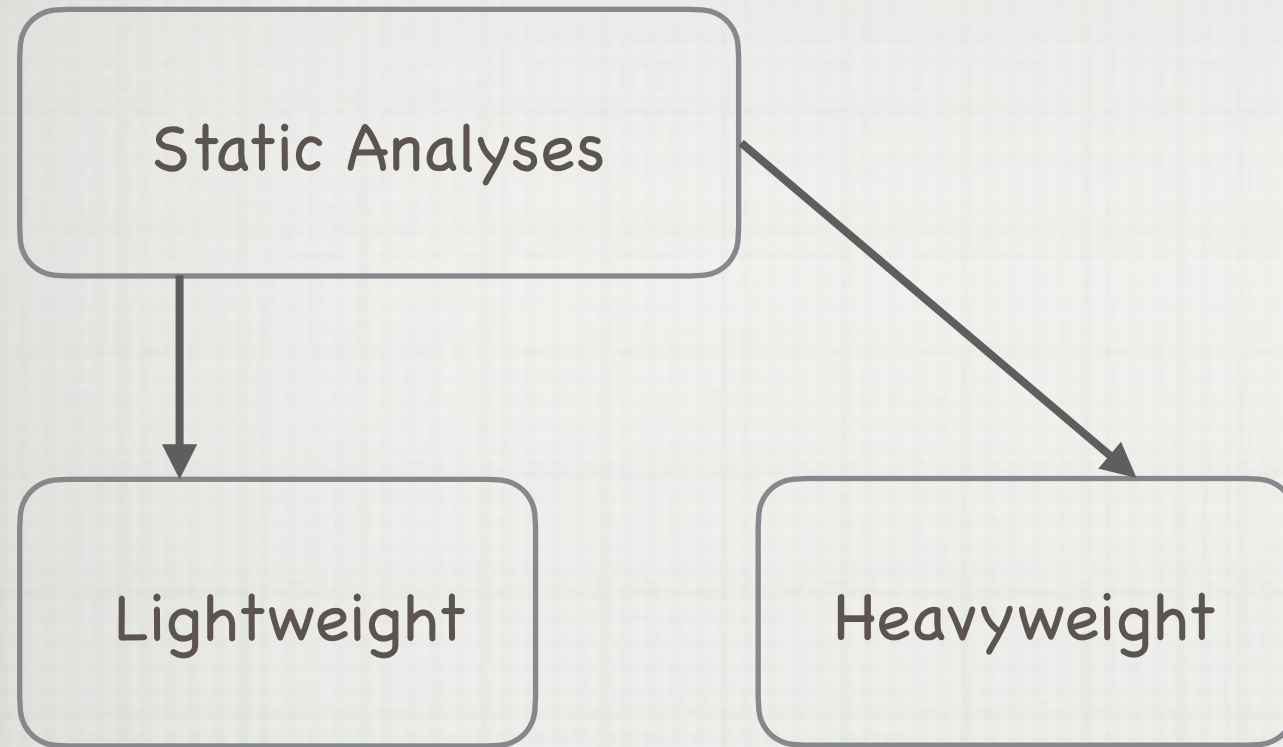
Static Analyses

Compilation

Typing

Invariant Generation

Program Analyses



Compilation

Typing

Invariant Generation

Program Analyses

Compilation

Typing

Invariant Generation

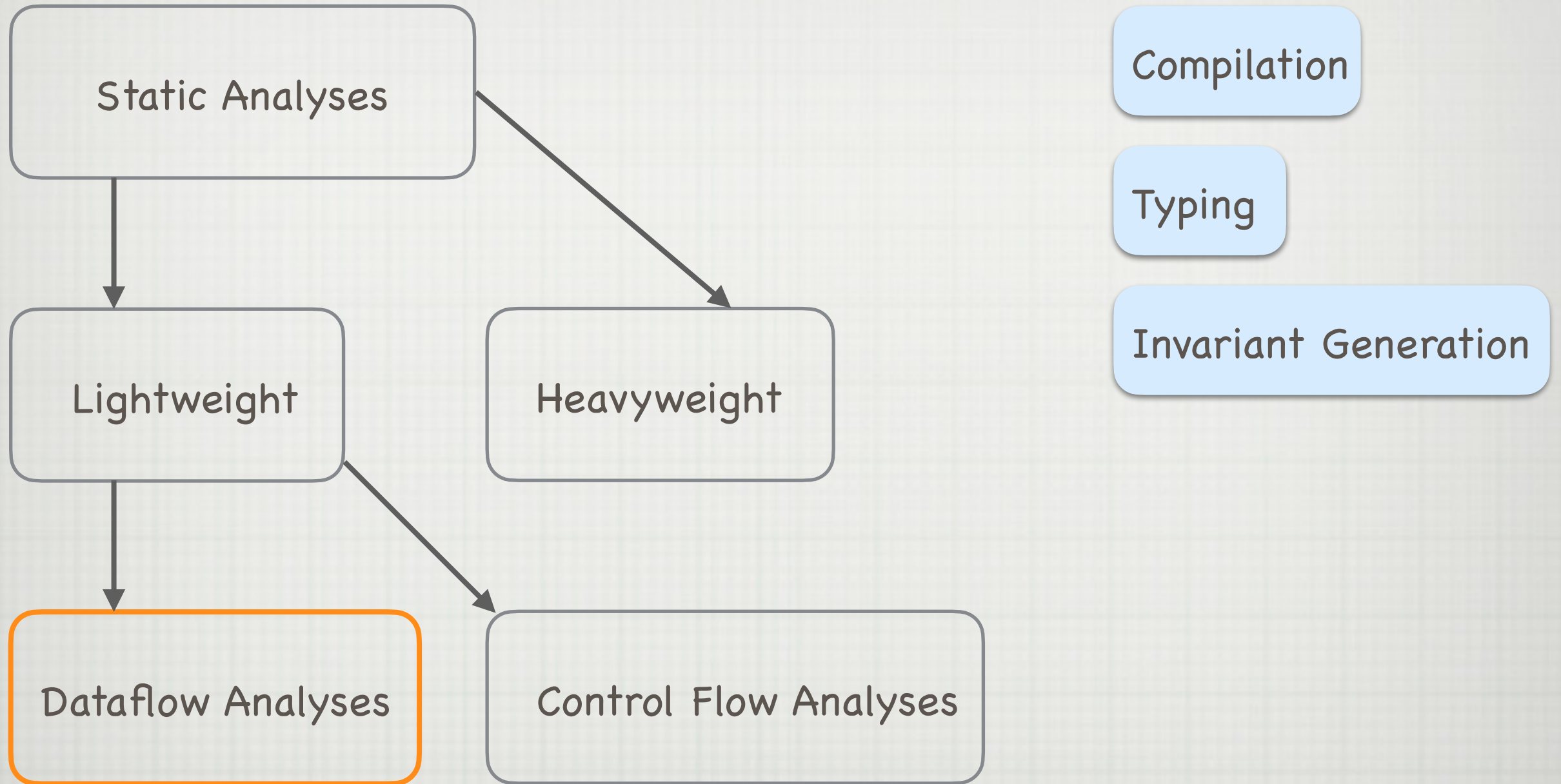
Static Analyses

Lightweight

Heavyweight

Dataflow Analyses

Control Flow Analyses



Program Analyses

Compilation

Typing

Invariant Generation

Static Analyses

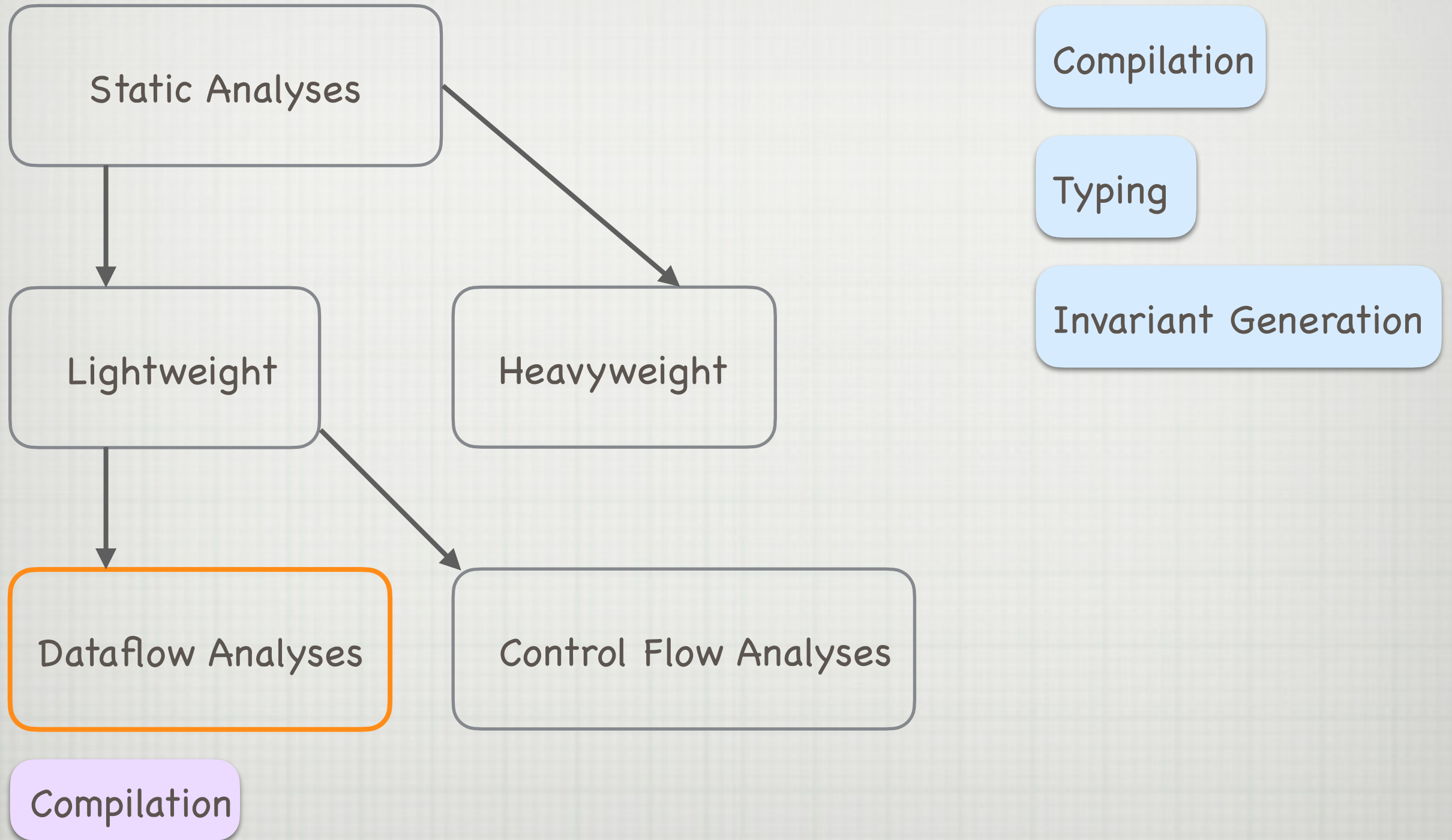
Lightweight

Heavyweight

Dataflow Analyses

Control Flow Analyses

Compilation



Program Analyses

Compilation

Typing

Invariant Generation

Static Analyses

Lightweight

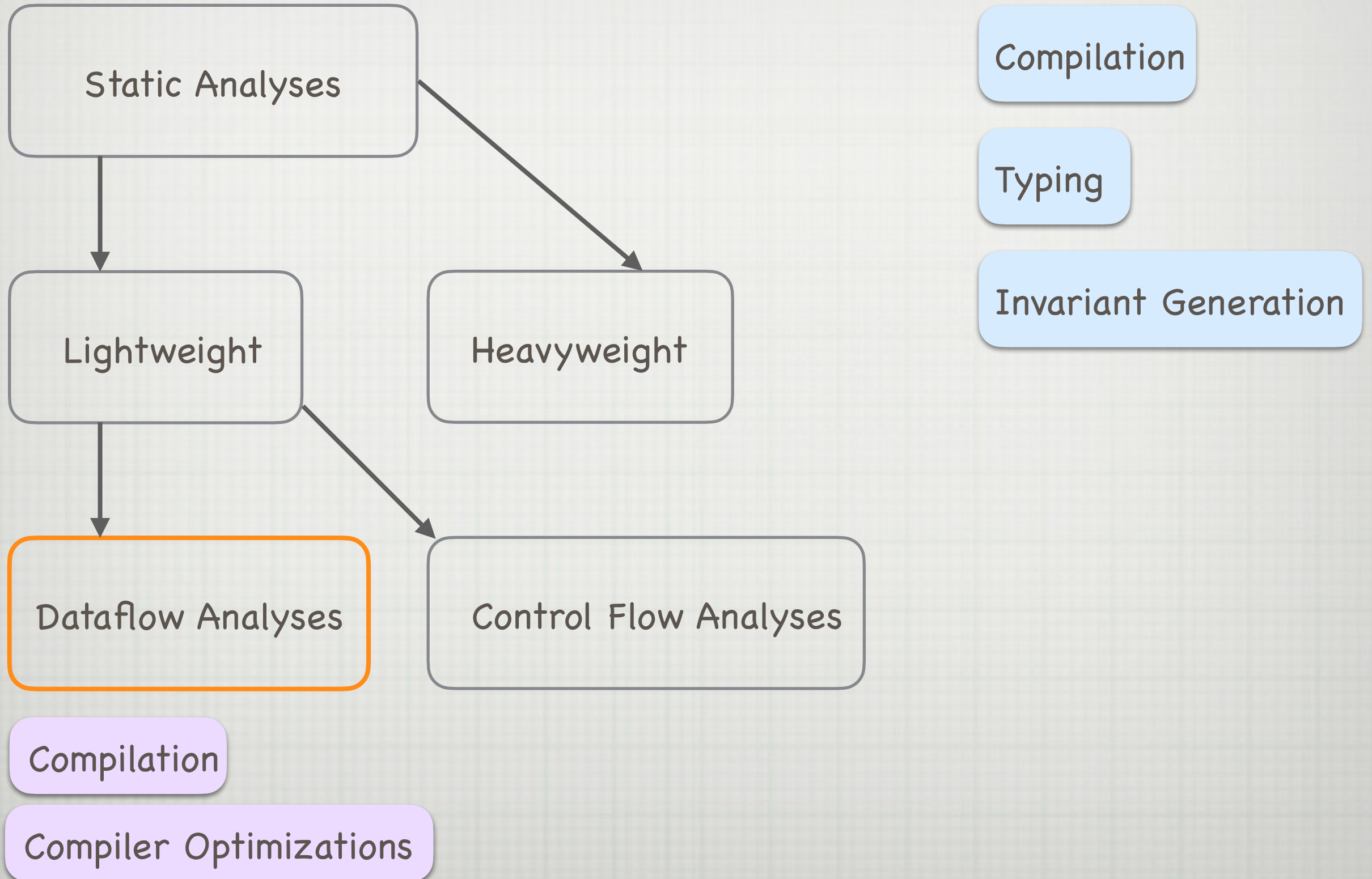
Heavyweight

Dataflow Analyses

Control Flow Analyses

Compilation

Compiler Optimizations



Program Analyses

Compilation

Typing

Invariant Generation

Static Analyses

Lightweight

Heavyweight

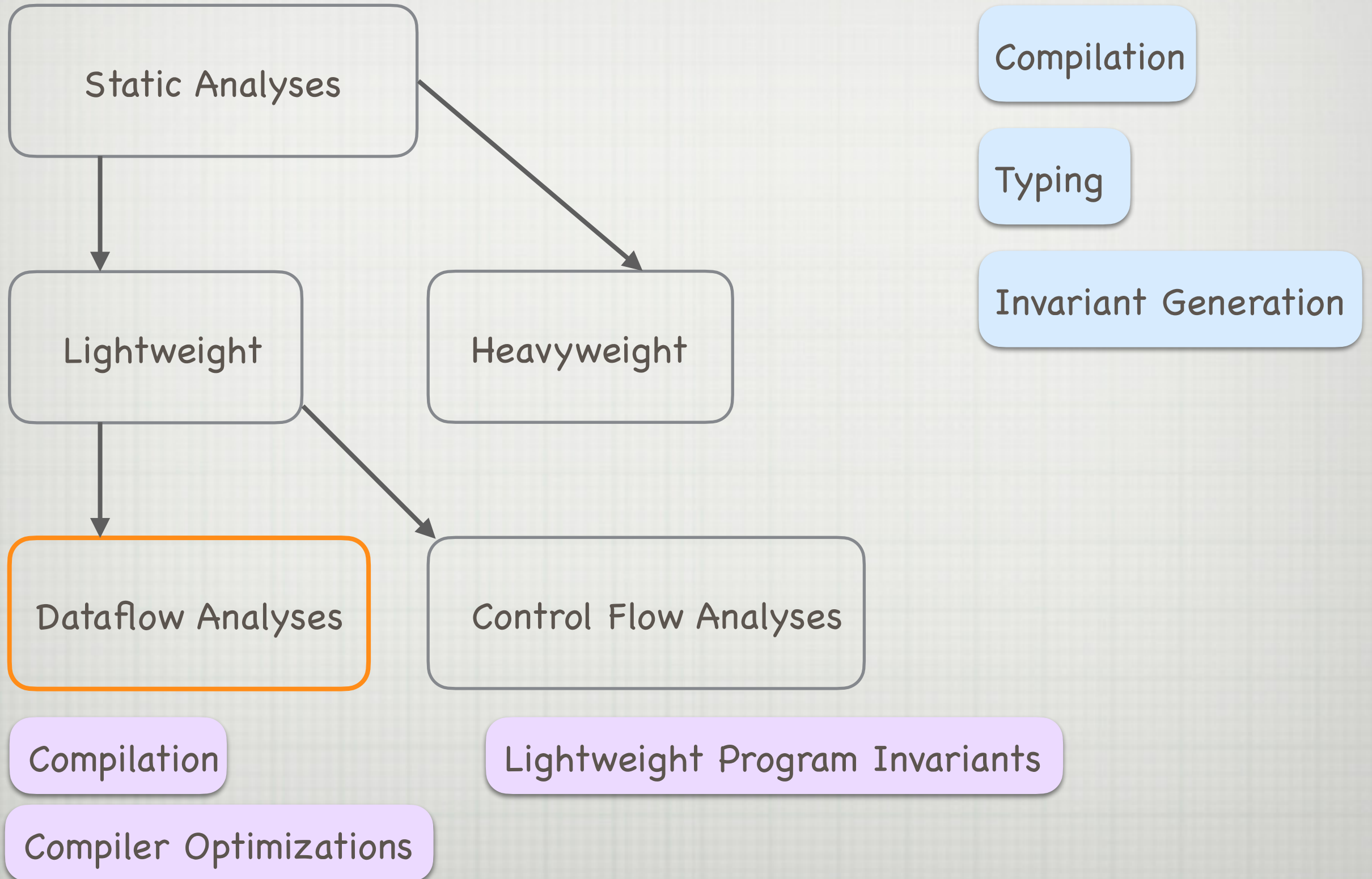
Dataflow Analyses

Control Flow Analyses

Compilation

Compiler Optimizations

Lightweight Program Invariants



Dataflow Analyses

Dataflow Analyses

Goal and scope:

- ☐ Calculate over-approximate information about the behaviours of the program.
- ☐ Do it efficiently!

Dataflow Analyses

Goal and scope:

- ☐ Calculate over-approximate information about the behaviours of the program.
- ☐ Do it efficiently!

Applications:

- ☐ Widely used in optimizing compilers
- ☐ Verification
- ☐ Testing

Let's start with an
example...

Dead Code Elimination

Dead Code Elimination

Can part of this program be removed without affecting program behaviour?

```
x := 2
y := 5
x := 1
z := 0
if (x <= 0) {
    z := x + 2
} else {
    z := y * y
}
x := z
```

Dead Code Elimination

Can part of this program be removed without affecting program behaviour?

```
x := 2  
y := 5  
x := 1  
z := 0  
if (x <= 0) {  
    z := x + 2  
}  
else {  
    z := y * y  
}  
x := z
```

Dead Code Elimination

Can part of this program be removed without affecting program behaviour?

```
x := 2  
y := 5  
x := 1  
z := 0  
if (x <= 0) {  
    z := x + 2  
}  
else {  
    z := y * y  
}  
x := z
```

Based on the principle that these assignments to **x** and **z** can never be read by any other statement.

Live Variable Analysis

Live Variable Analysis

Intuition: A variable v is **live** at a program point if its value **may** be read later in **some** execution of the program. Otherwise, it is **dead**.

Live Variable Analysis

Intuition: A variable v is **live** at a program point if its value **may** be read later in **some** execution of the program. Otherwise, it is **dead**.

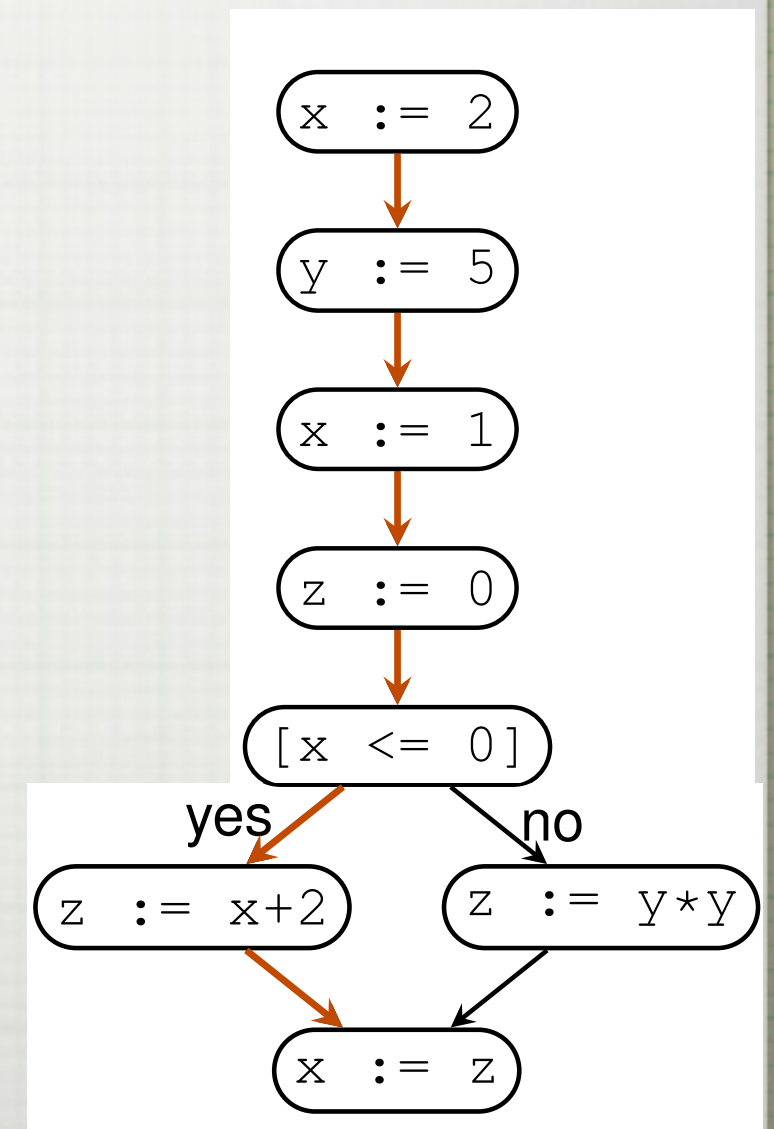
An assignment $x := \text{exp}$ can be safely removed from the program (i.e it is dead code) if x is dead (at the exit point of this statement).

How do we calculate this
information automatically
and efficiently?

Models

Control Flow Graphs

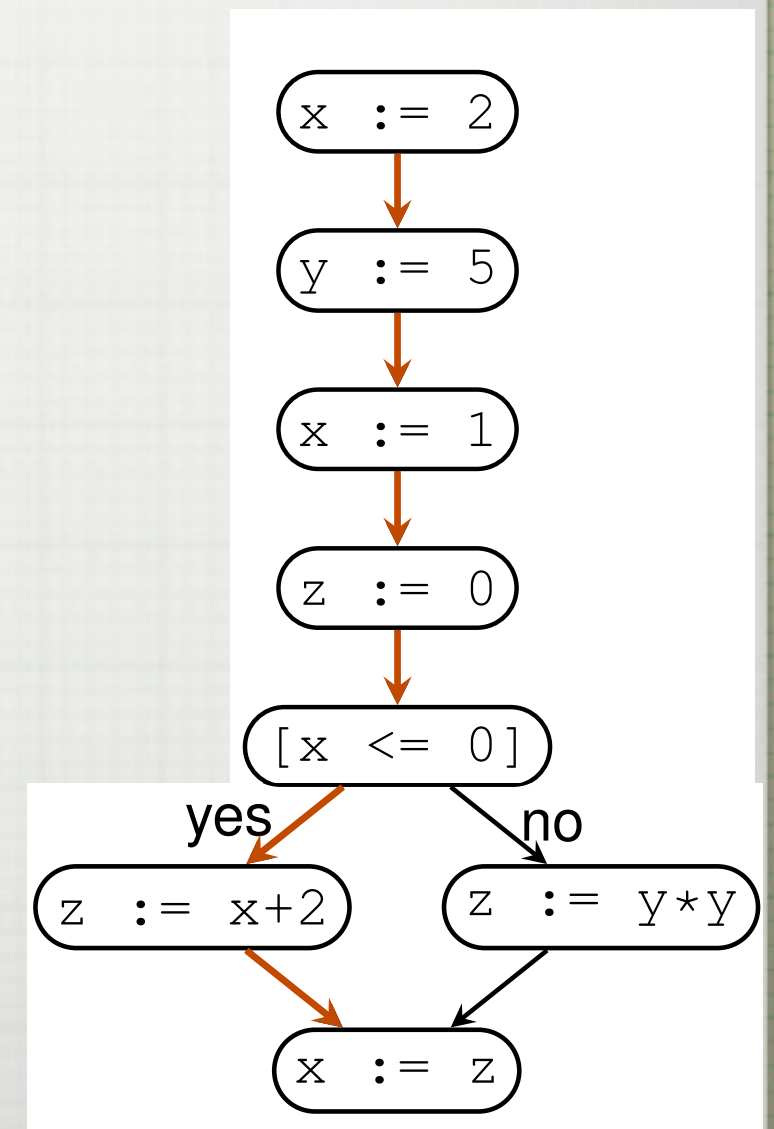
A **Control Flow Graph (CFG)** is a graphical representation of the executions of a program. Formally, a CFG is a directed graph $\langle V, E \rangle$, where:



Control Flow Graphs

A **Control Flow Graph (CFG)** is a graphical representation of the executions of a program. Formally, a CFG is a directed graph $\langle V, E \rangle$, where:

- V is a set of program labels (AKA locations, points)
- E is the control flow successor relation: if v follows u in some execution, then $u \rightarrow v \in E$



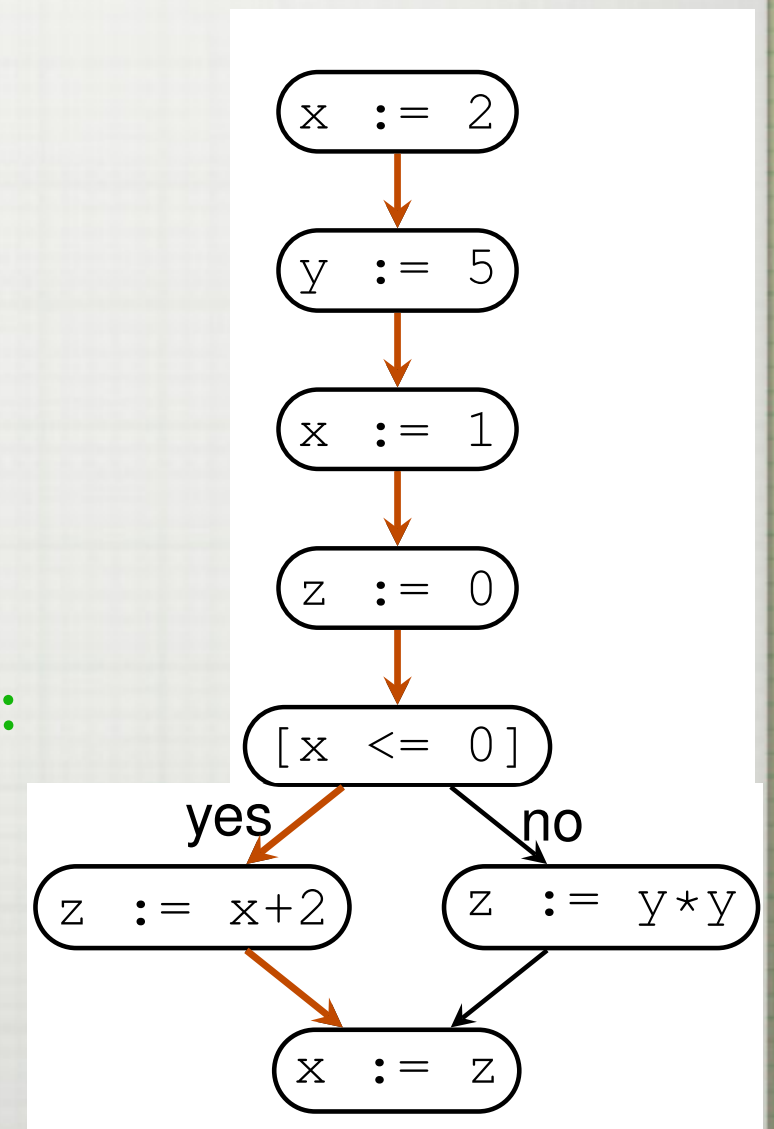
Control Flow Graphs

A **Control Flow Graph (CFG)** is a graphical representation of the executions of a program. Formally, a CFG is a directed graph $\langle V, E \rangle$, where:

- V is a set of program labels (AKA locations, points)
- E is the control flow successor relation: if v follows u in some execution, then $u \rightarrow v \in E$

Every execution corresponds to a path in the CFG:

- the converse does not hold!
- A CFG represents an over-approximation of the executions of a program



Formal Specification

Live Variable Analysis

Intuition: A variable v is **live** at a program point if its value may be read later in **some** execution of the program. Otherwise, it is **dead**.

Live Variable Analysis

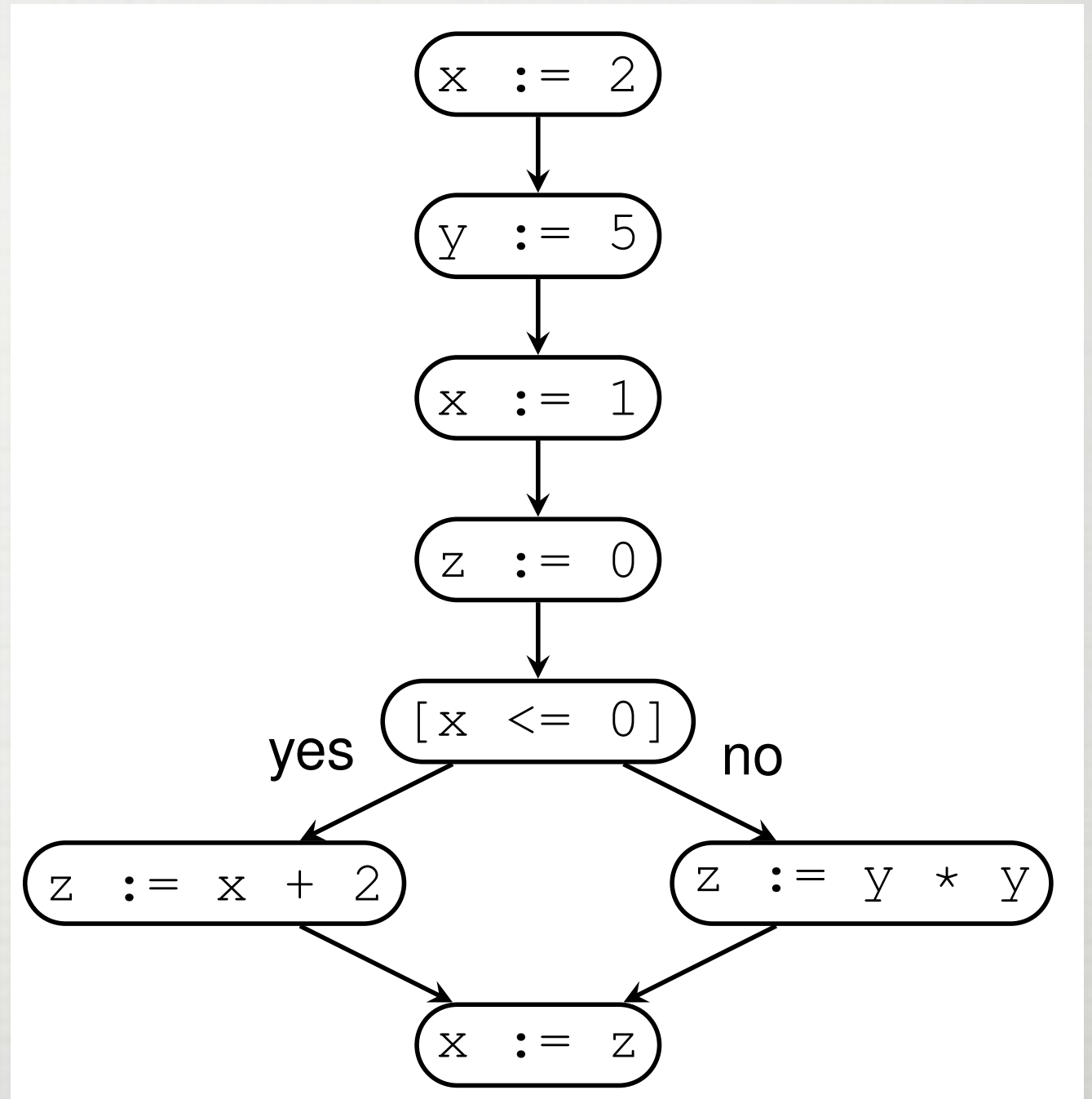
Intuition: A variable v is **live** at a program point if its value may be read later in **some** execution of the program. Otherwise, it is **dead**.

Formal definition: A variable var is **live** at a label l if there exists a CFG path from l to a label l' such that l' **reads** var , and no label along that path **writes** to var . Otherwise it is **dead**.

Example

At $x := 1$:

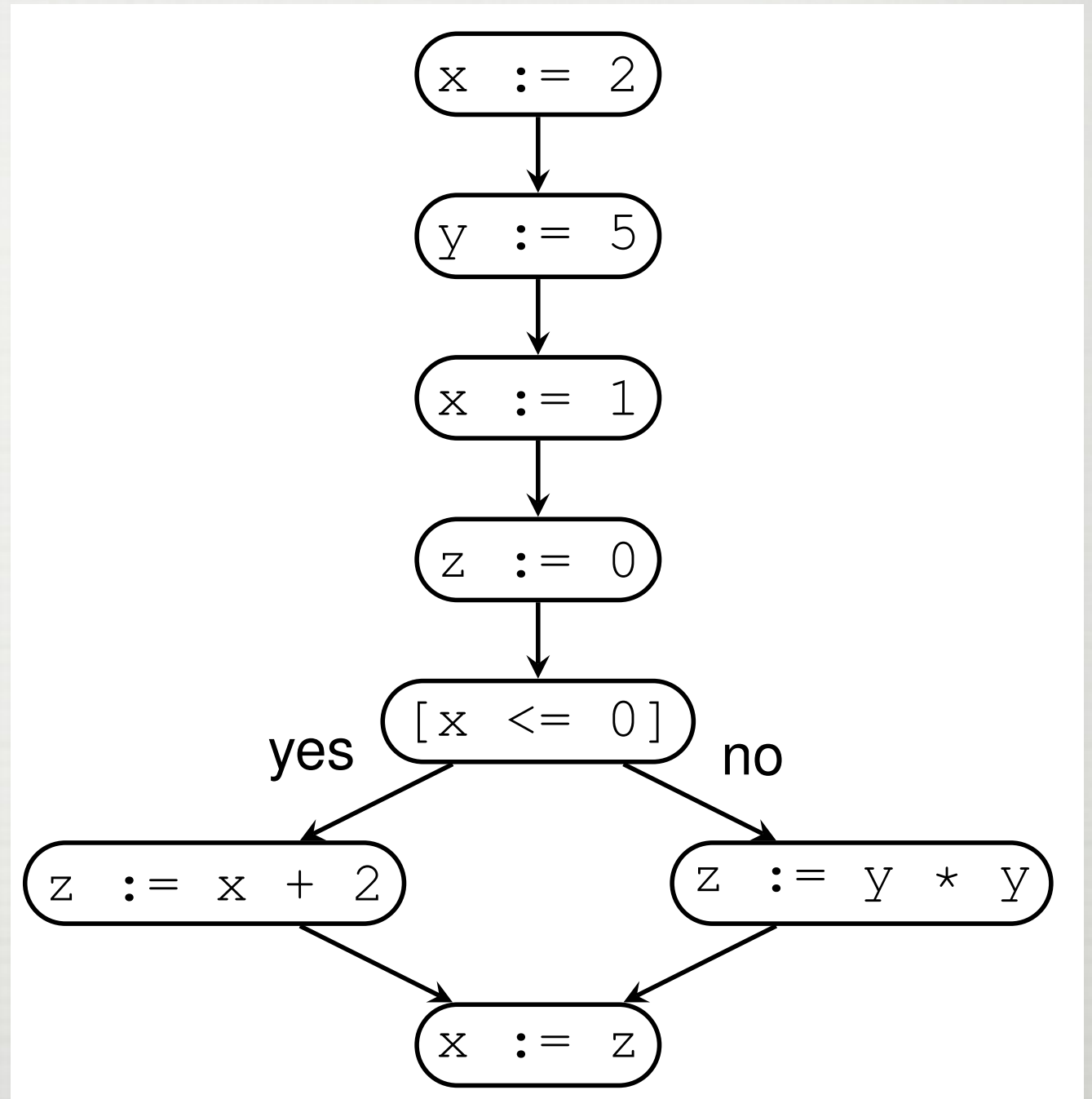
- ☐ x
- ☐ y
- ☐ z



Example

At $x := 1$:

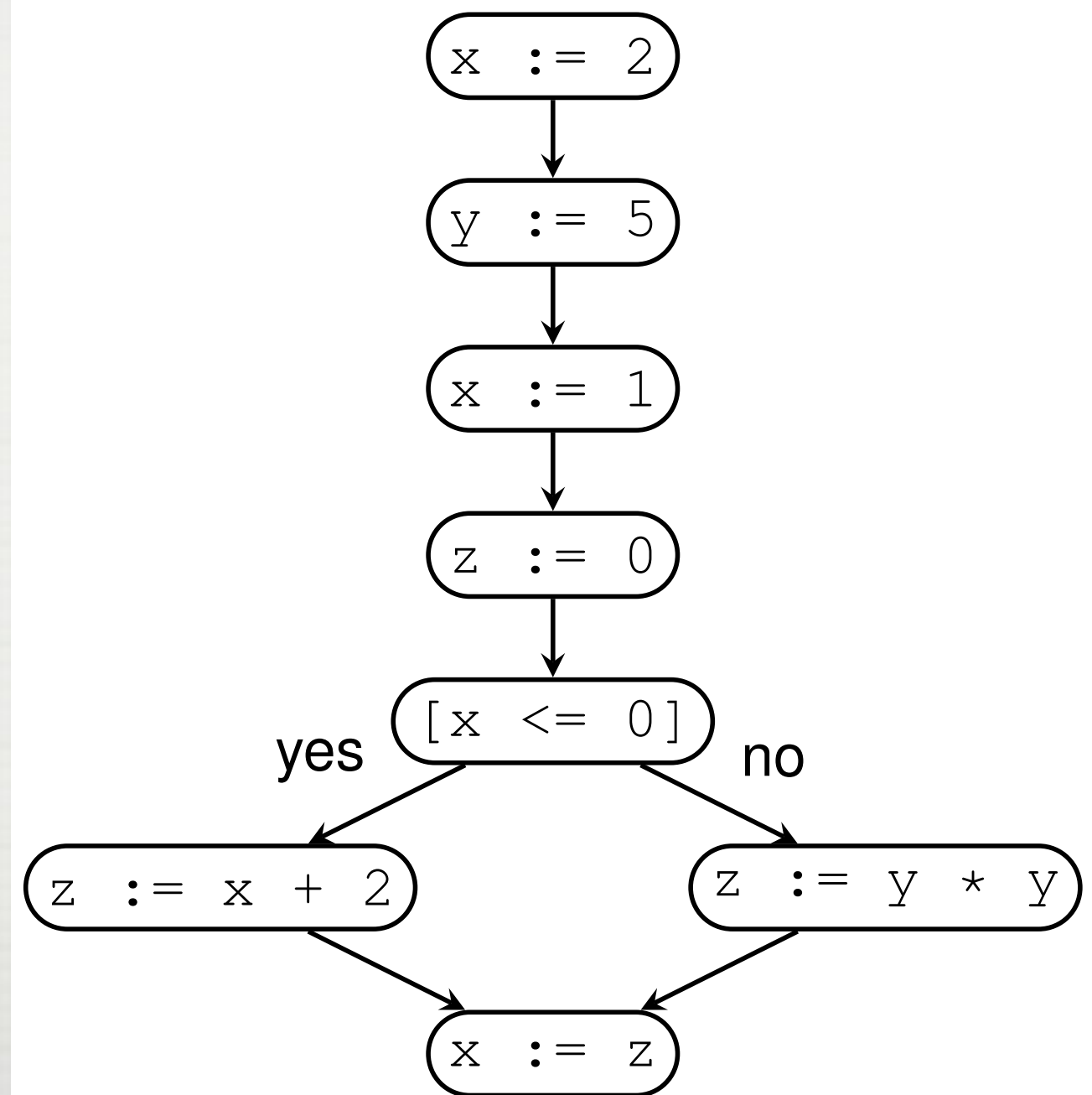
- ☐ x is live
- ☐ y
- ☐ z



Example

At $x := 1$:

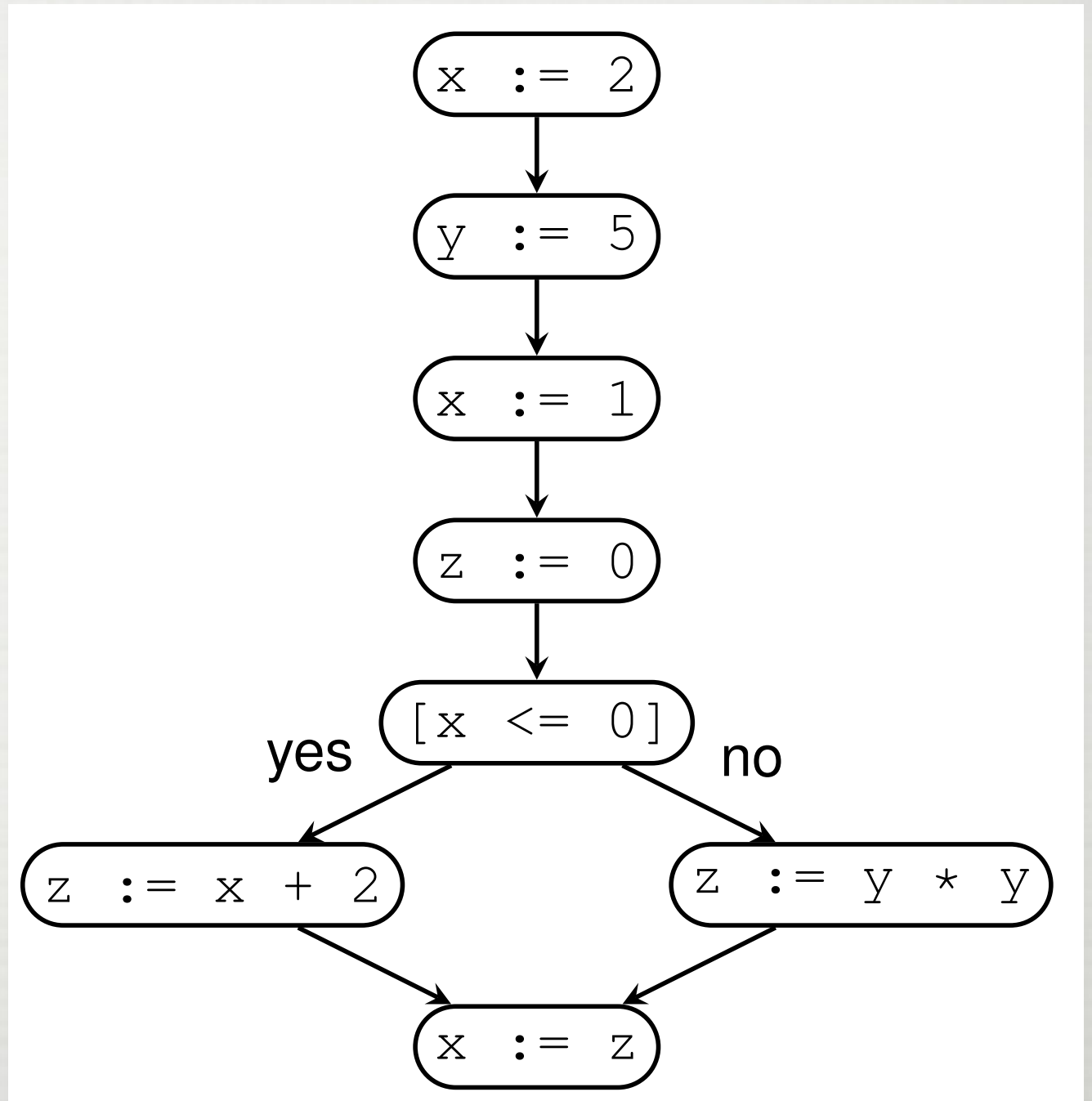
- ☐ x is live
- ☐ y is live
- ☐ z



Example

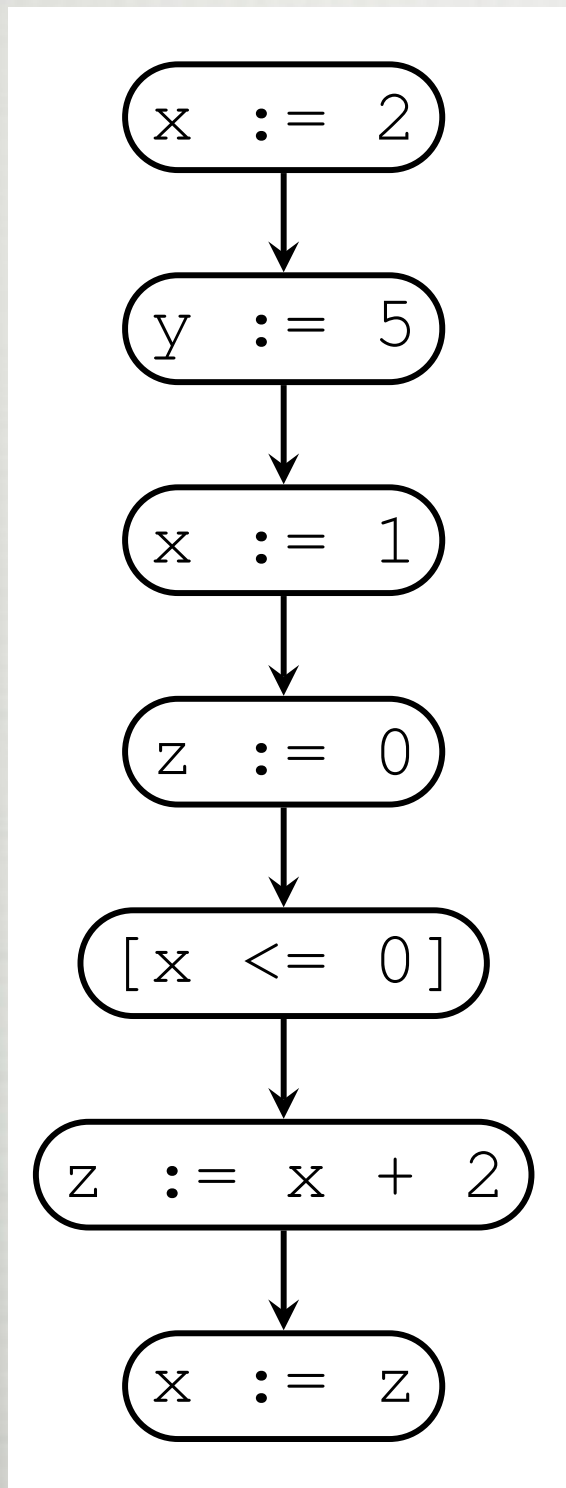
At $x := 1$:

- ☐ x is live
- ☐ y is live
- ☐ z is dead

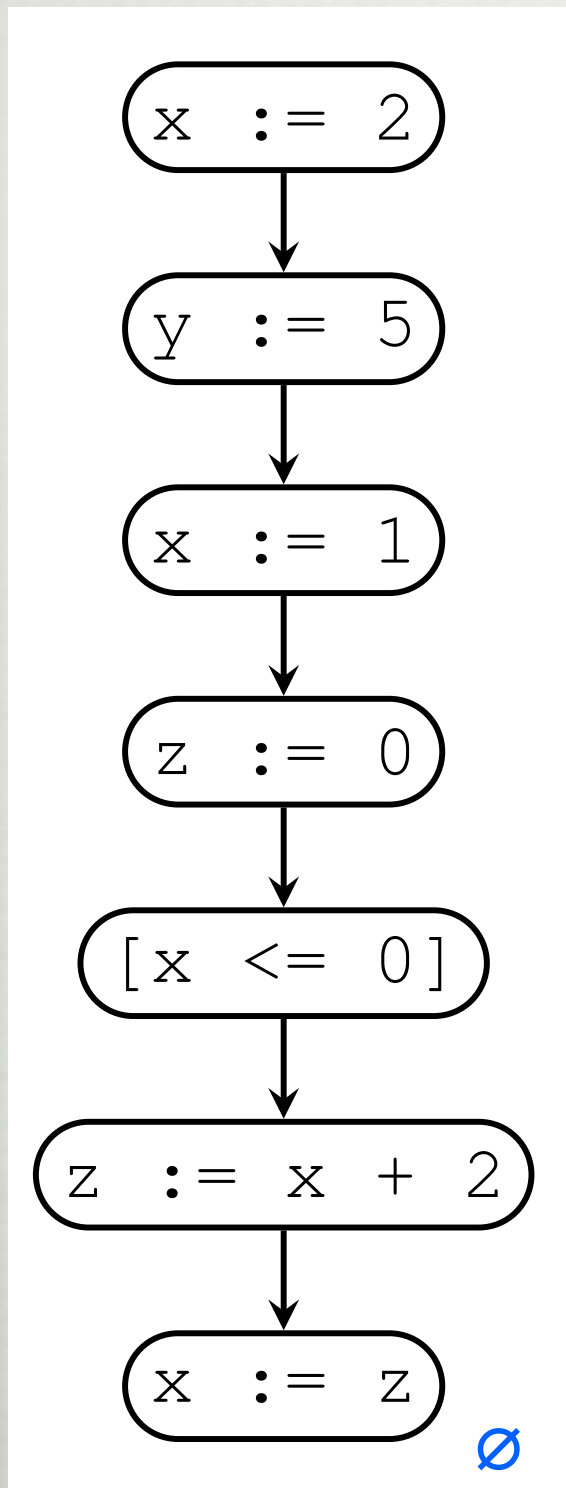


Let's start automating this
in the simplest context ...

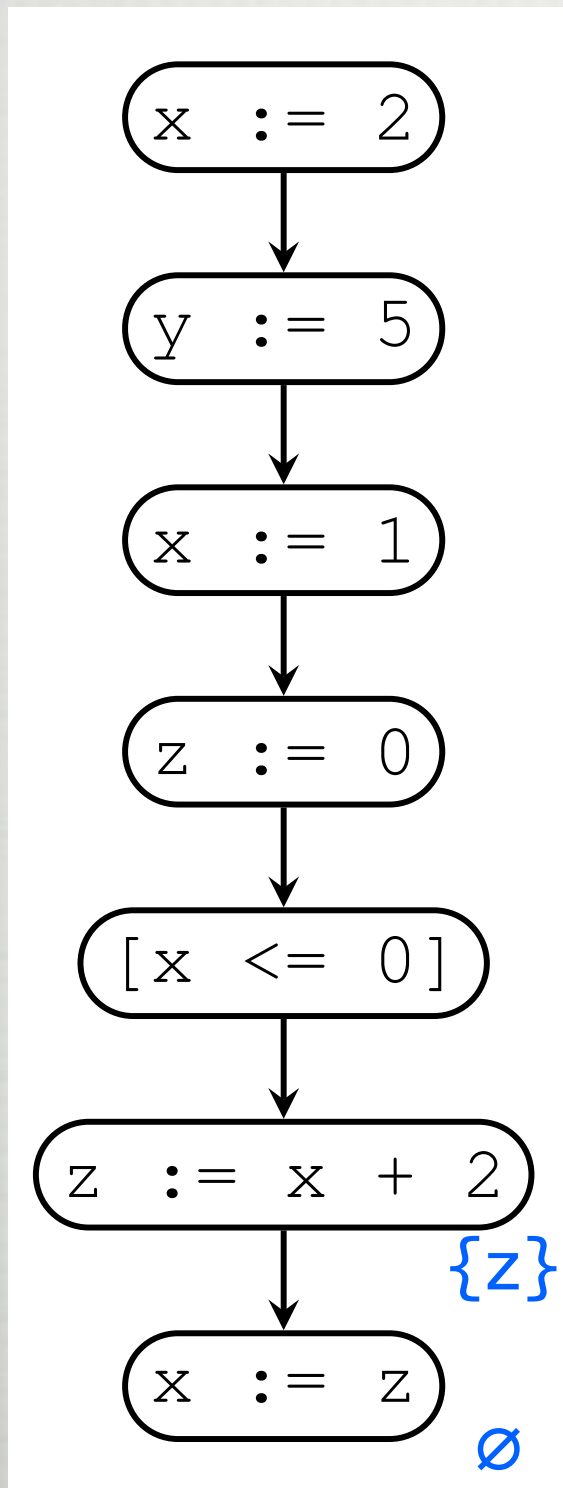
Live Variable Analysis on Paths



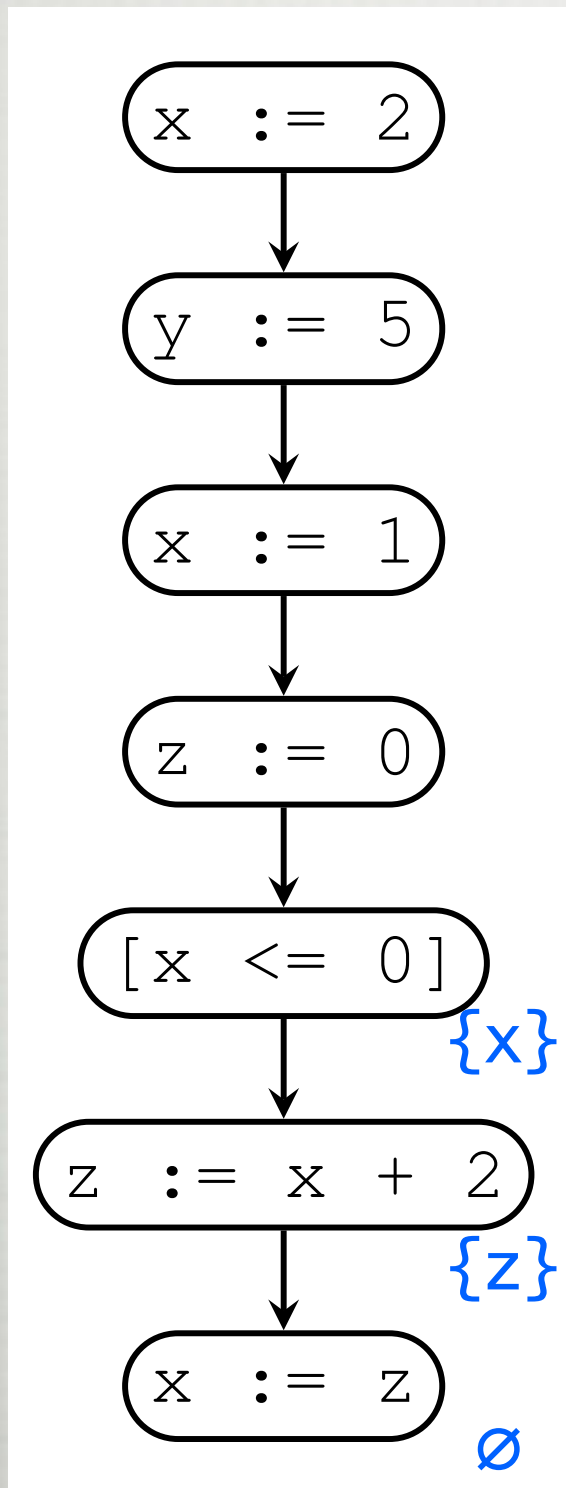
Live Variable Analysis on Paths



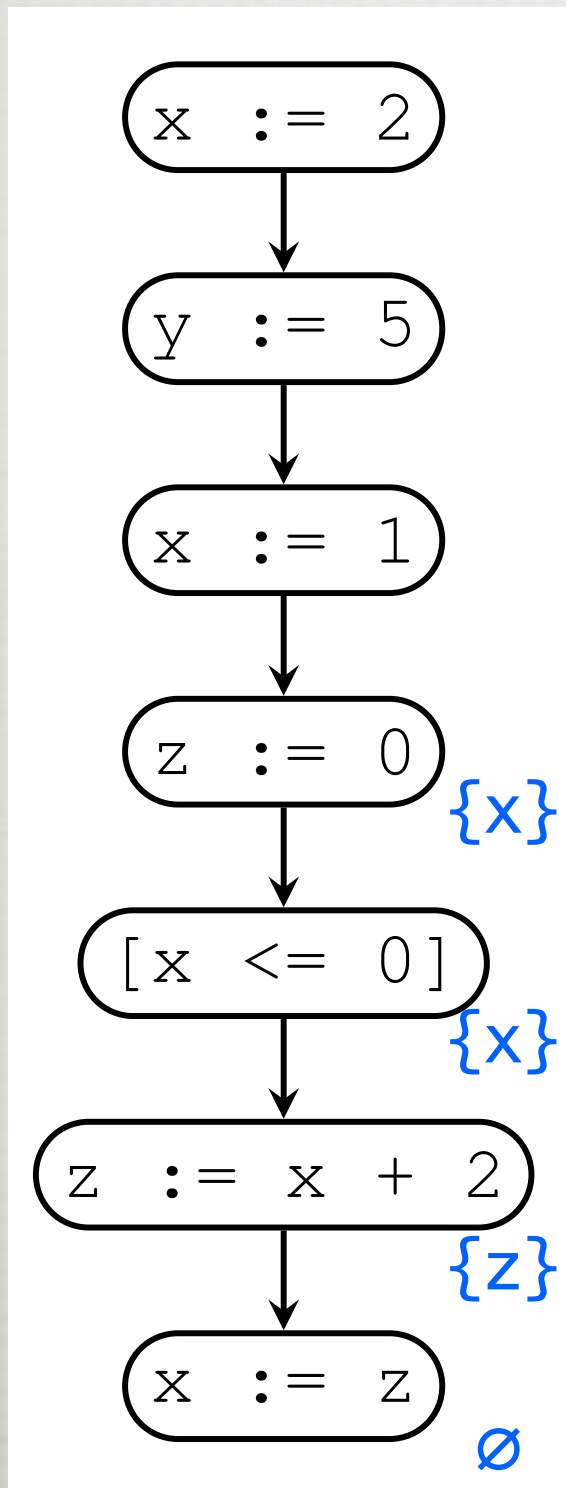
Live Variable Analysis on Paths



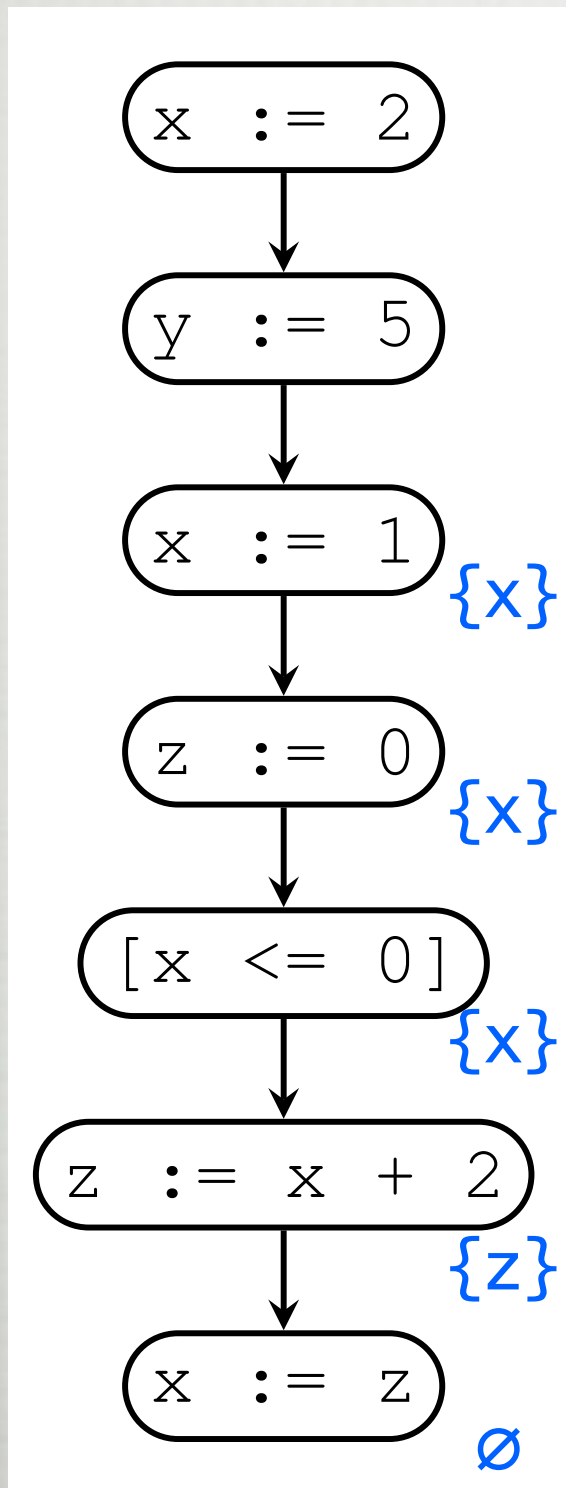
Live Variable Analysis on Paths



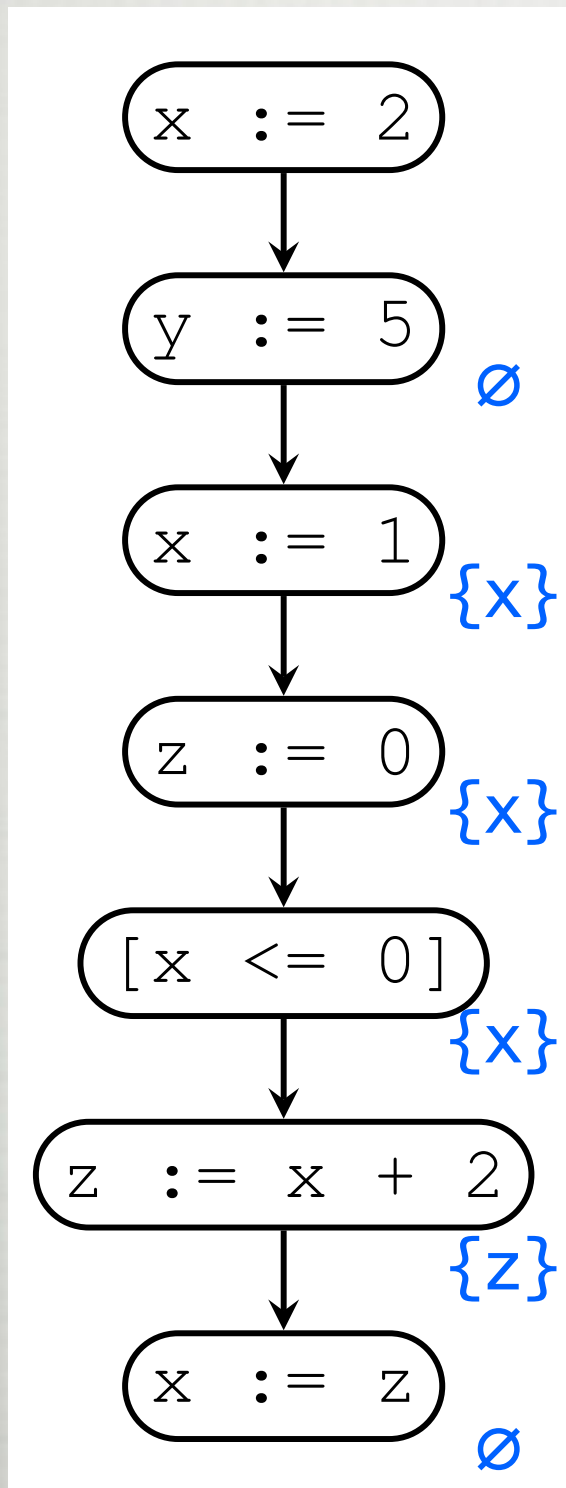
Live Variable Analysis on Paths



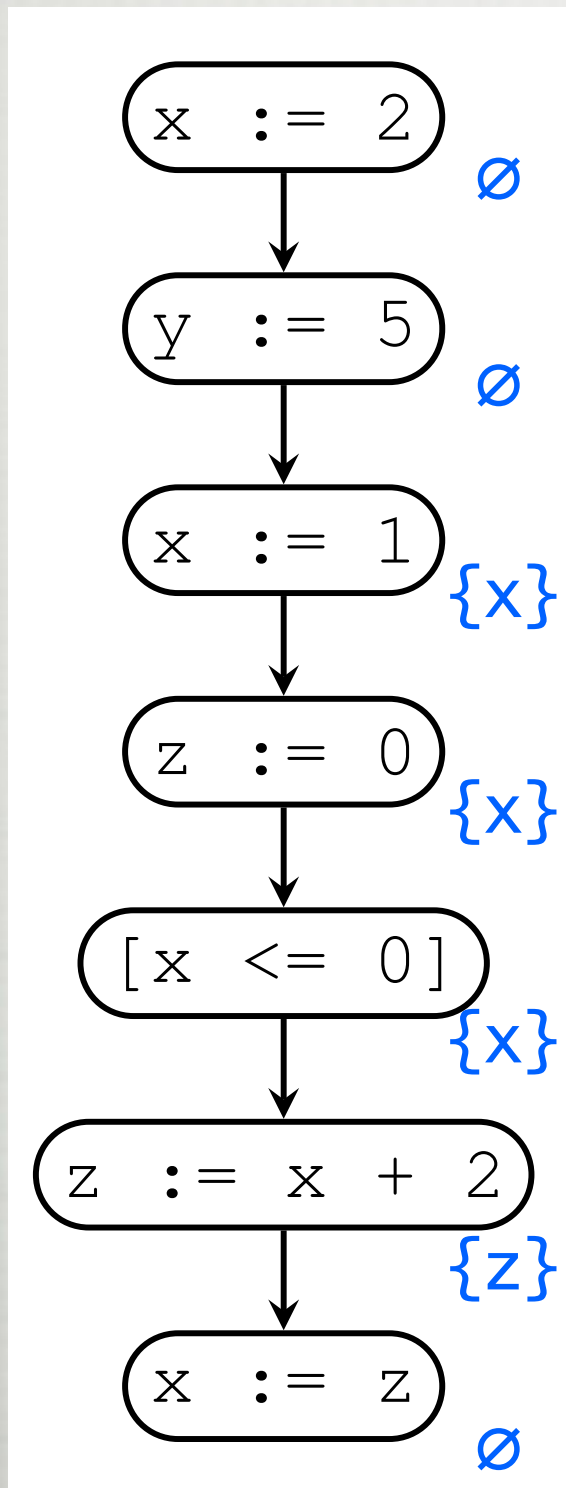
Live Variable Analysis on Paths



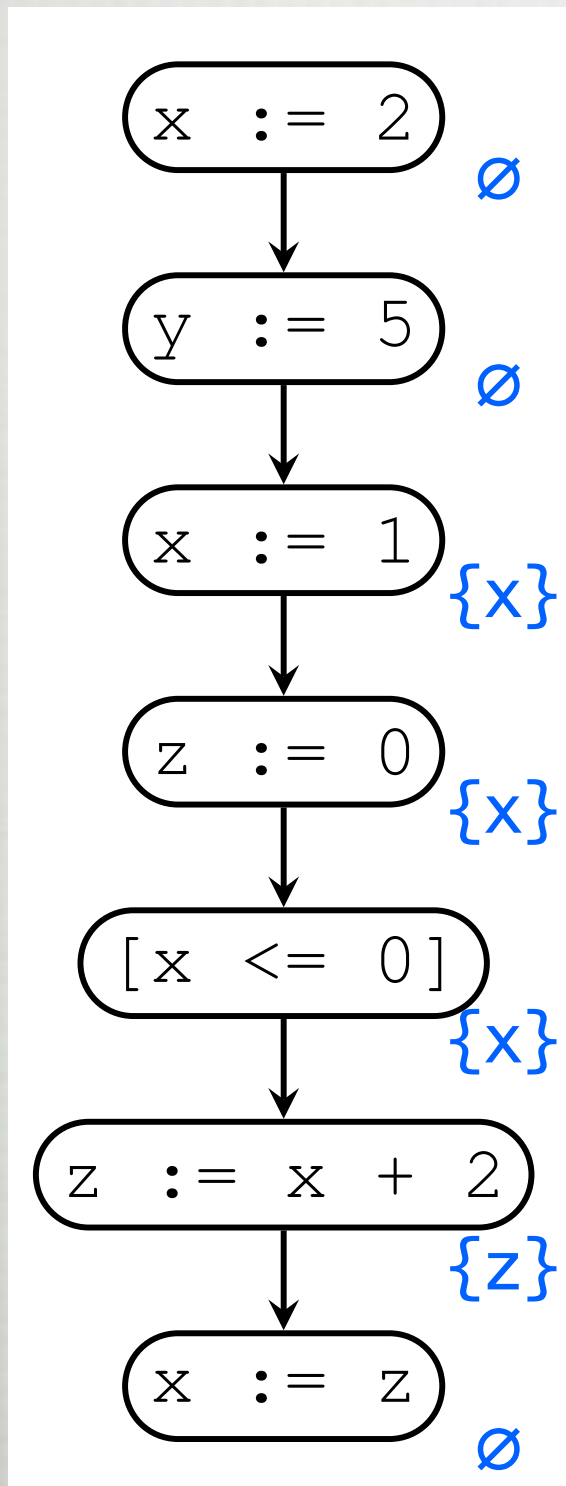
Live Variable Analysis on Paths



Live Variable Analysis on Paths



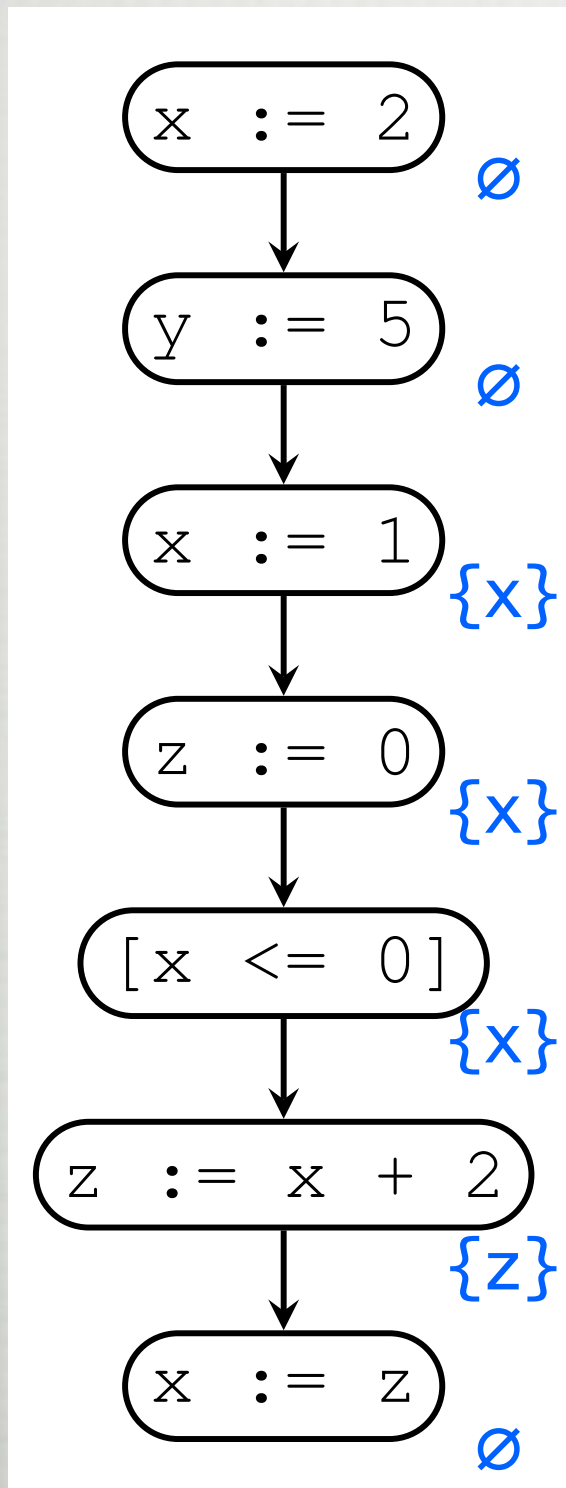
Live Variable Analysis on Paths



Define for each label l :

- ☐ $\text{read}(l)$: the set of variables l reads from.
- ☐ $\text{write}(l)$: the set of variables l writes to.

Live Variable Analysis on Paths



Define for each label l :

- ☐ $\text{read}(l)$: the set of variables l reads from.
- ☐ $\text{write}(l)$: the set of variables l writes to.

For a path $l_1 \dots l_n$:

$$\begin{aligned} \text{live}(\ell_n) &= \emptyset \\ \text{live}(\ell_{i-1}) &= \text{live}(\ell_i) \setminus \text{write}(\ell_i) \cup \text{read}(\ell_i) \end{aligned}$$

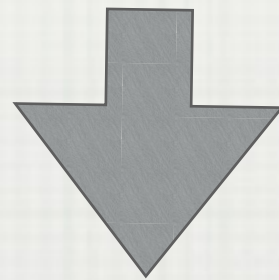
Live Variable Analysis **on Paths**

$$live(\ell_n) = \emptyset$$

$$live(\ell_{i-1}) = live(\ell_i) \setminus write(\ell_i) \cup read(\ell_i)$$

Live Variable Analysis **on Paths**

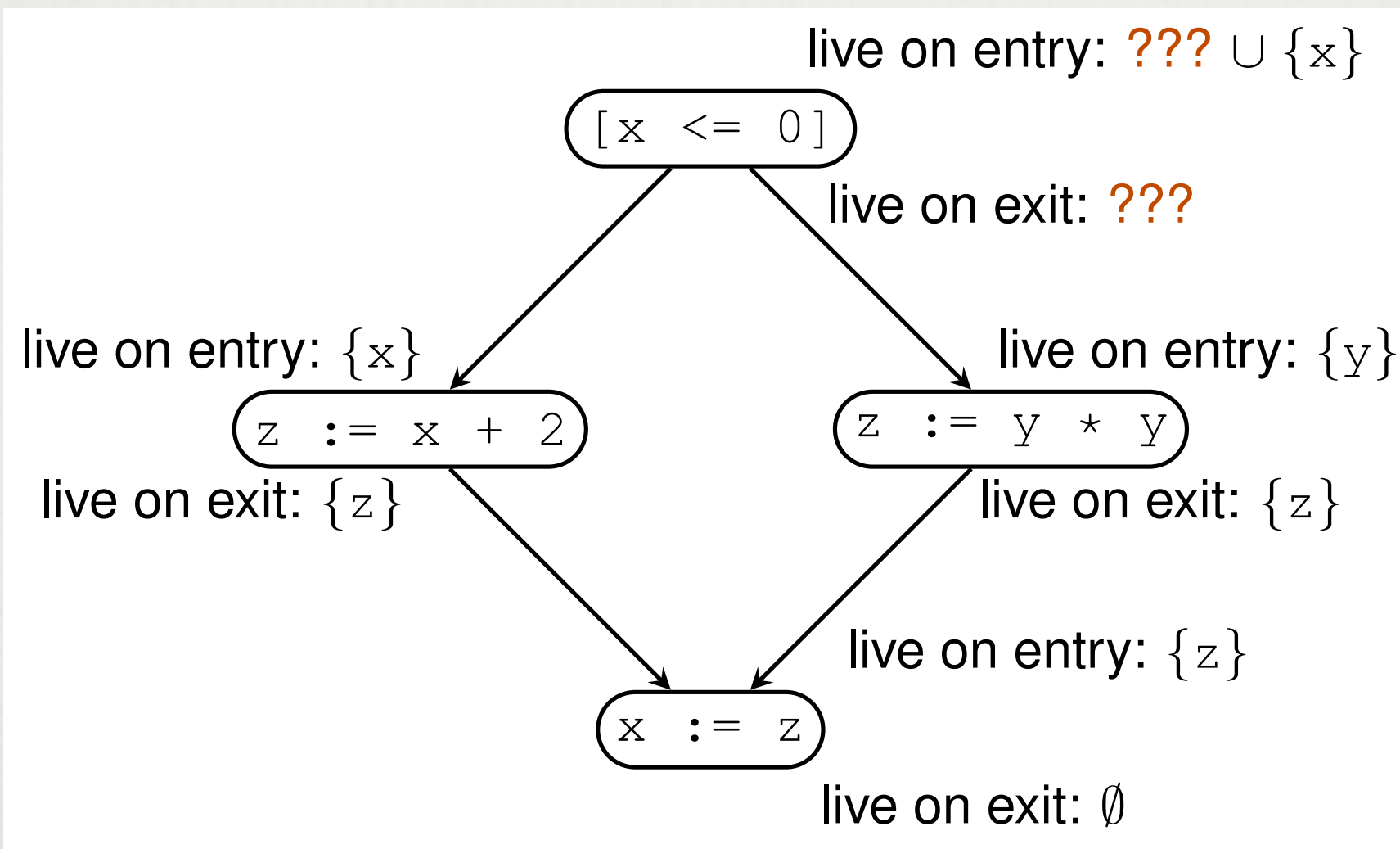
$$\begin{aligned} \textit{live}(\ell_n) &= \emptyset \\ \textit{live}(\ell_{i-1}) &= \textit{live}(\ell_i) \setminus \textit{write}(\ell_i) \cup \textit{read}(\ell_i) \end{aligned}$$



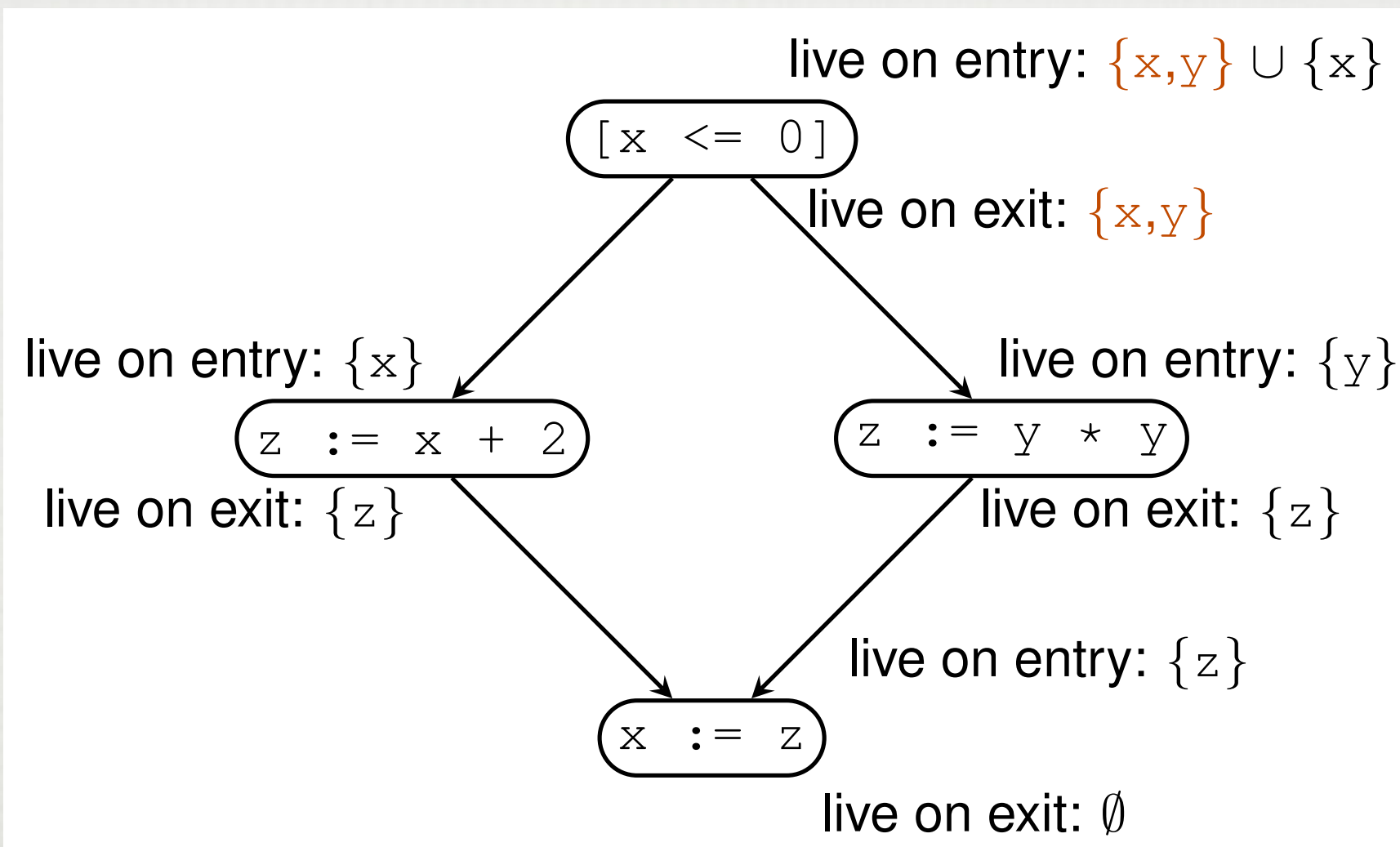
$$\begin{aligned} \textit{live}_{\textit{exit}}(\ell_n) &= \emptyset \\ \textit{live}_{\textit{entry}}(\ell) &= \textit{live}_{\textit{exit}}(\ell) \setminus \textit{write}(\ell) \cup \textit{read}(\ell) \\ \textit{live}_{\textit{exit}}(\ell_{i-1}) &= \textit{live}_{\textit{entry}}(\ell_i) \end{aligned}$$

What if we have branches
along the way?

Live Variable Analysis on DAGs



Live Variable Analysis on DAGs



Live Variable Analysis on DAGs

$$live_{exit}(\ell_n) = \emptyset$$

$$live_{entry}(\ell) = live_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

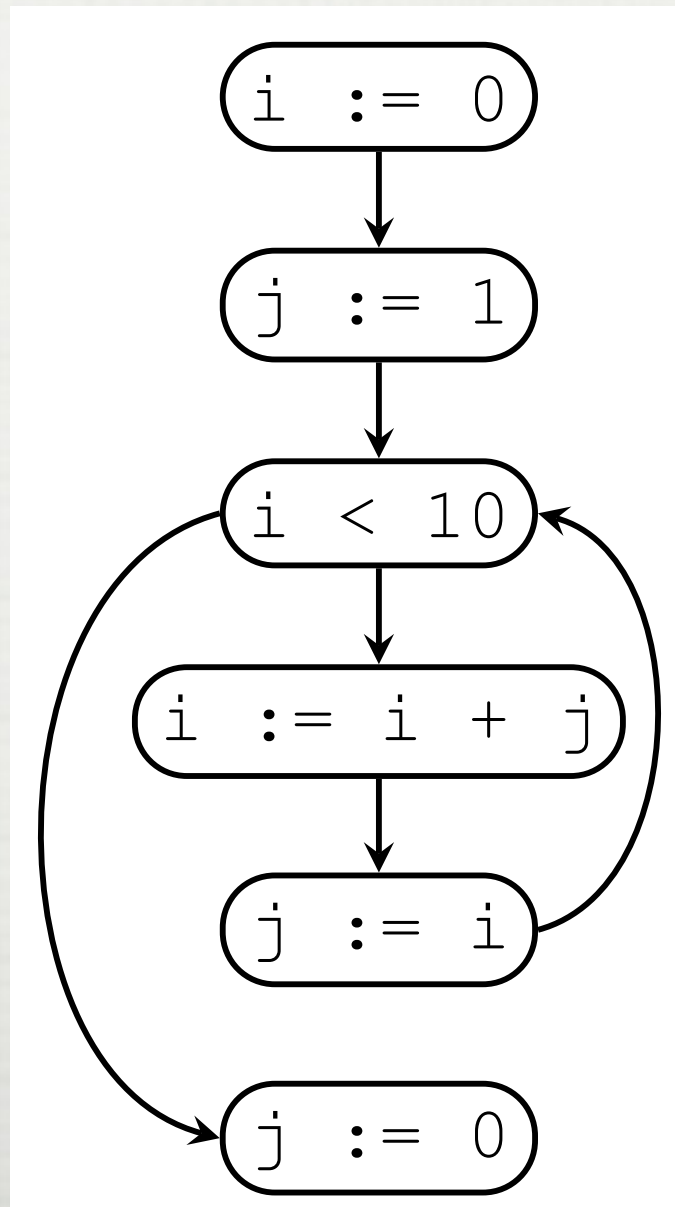
$$live_{exit}(\ell_{i-1}) = live_{entry}(\ell_i)$$

Live Variable Analysis on DAGs

$$\begin{aligned}live_{entry}(\ell) &= live_{exit}(\ell) \setminus write(\ell) \cup read(\ell) \\live_{exit}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} live_{entry}(\ell')\end{aligned}$$

What if there are cycles in
the graph?

What if there are cycles in the graph?



Live Variable Analysis on Graphs

Consider the equations below as **a system of constraints**. A solution is any assignment of live sets that satisfies the equations below:

$$\begin{aligned} LV_{entry}(\ell) &= LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell) \\ LV_{exit}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell') \end{aligned}$$

Live Variable Analysis on Graphs

Consider the equations below as **a system of constraints**. A solution is any assignment of live sets that satisfies the equations below:

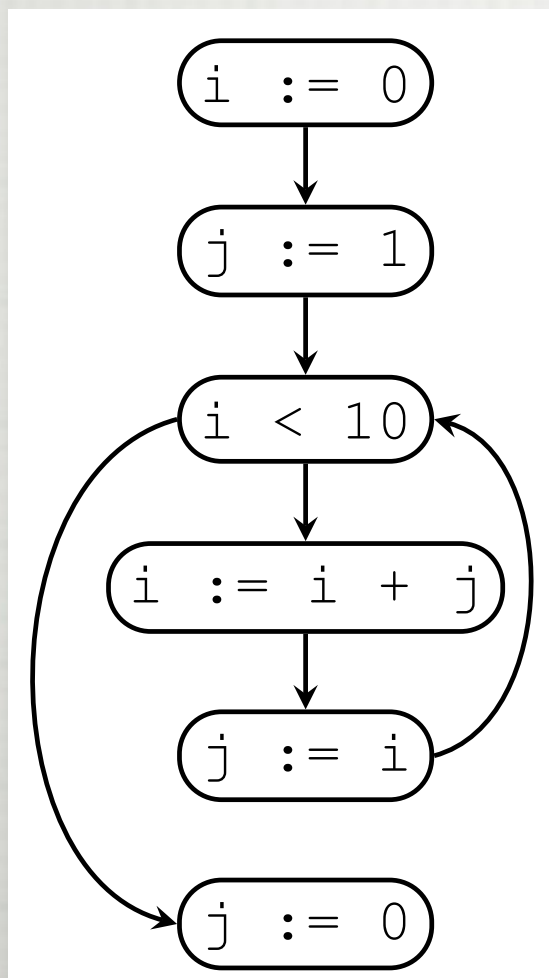
$$\begin{aligned}LV_{entry}(\ell) &= LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell) \\ LV_{exit}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')\end{aligned}$$

Our desired live variables information certainly is **a** solution to this constraint system.

Solution not necessarily unique

$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

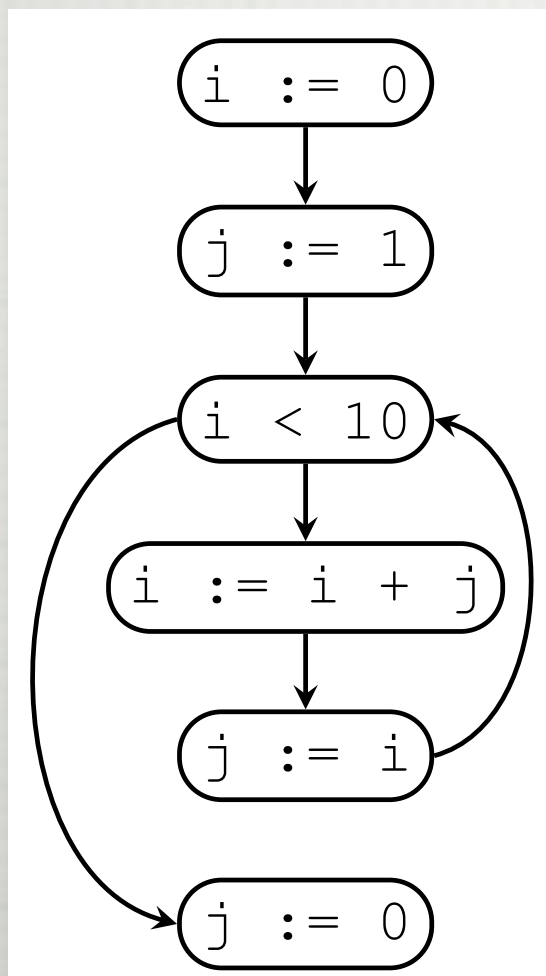
$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$



Solution not necessarily unique

$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

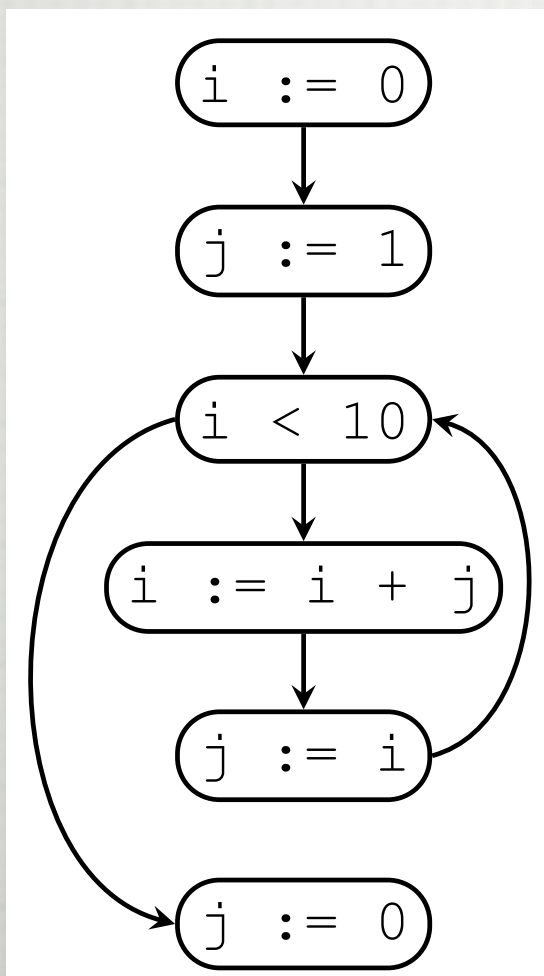


Label	LV_{exit}	LV_{entry}
$i := 0$	$\{i\}$	$\emptyset$
$j := 1$	$\{i, j\}$	$\{i\}$
$i < 10$	$\{i, j\}$	$\{i, j\}$
$i := i + j$	$\{i\}$	$\{i, j\}$
$j := i$	$\{i, j\}$	$\{i\}$
$j := 0$	$\emptyset$	$\emptyset$

Solution not necessarily unique

$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$



Label	LV_{exit}	LV_{entry}
$i := 0$	$\{i, k\}$	$\{k\}$
$j := 1$	$\{i, j, k\}$	$\{i, k\}$
$i < 10$	$\{i, j, k\}$	$\{i, j, k\}$
$i := i + j$	$\{i, k\}$	$\{i, j, k\}$
$j := i$	$\{i, j, k\}$	$\{i, k\}$
$j := 0$	$\emptyset$	$\emptyset$

Key Observation

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

Key Observation

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

$$x \text{ is live at } l \Rightarrow x \in \text{Live}_{\text{exit}}(l)$$

Key Observation

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

$$x \text{ is live at } l \Rightarrow x \in \text{Live}_{\text{exit}}(l)$$

The desired solution is the **least** solution!

Key Observation

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

$$x \text{ is live at } l \Rightarrow x \in \text{Live}_{\text{exit}}(l)$$

The desired solution is the **least** solution!

Our comparator is the subset relation!

Can we compute the least
solution?

The Worklist Algorithm

Algorithm 1 Worklist method

$worklist \leftarrow empty$
for each $\ell \in V$ **do**
 $LV_{exit}(\ell) \leftarrow \emptyset$
 $LV_{entry}(\ell) \leftarrow LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$
 add ℓ to $worklist$
end for
while $worklist$ not empty **do**
 pick ℓ from $worklist$
 $old \leftarrow LV_{entry}(\ell)$
 $LV_{exit}(\ell) \leftarrow \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$
 $LV_{entry}(\ell) \leftarrow LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$
 if $old \neq LV_{entry}(\ell)$ **then**
 Add predecessors of ℓ to $worklist$
 end if
end while

The Worklist Algorithm

Algorithm 1 Worklist method

worklist $\leftarrow$ *empty*

for each $\ell \in V$ **do**

$LV_{exit}(\ell) \leftarrow \emptyset$

$LV_{entry}(\ell) \leftarrow LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$

add ℓ to *worklist*

end for

while *worklist* not empty **do**

pick ℓ from *worklist*

old $\leftarrow LV_{entry}(\ell)$

$LV_{exit}(\ell) \leftarrow \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$

$LV_{entry}(\ell) \leftarrow LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$

if *old* $\neq LV_{entry}(\ell)$ **then**

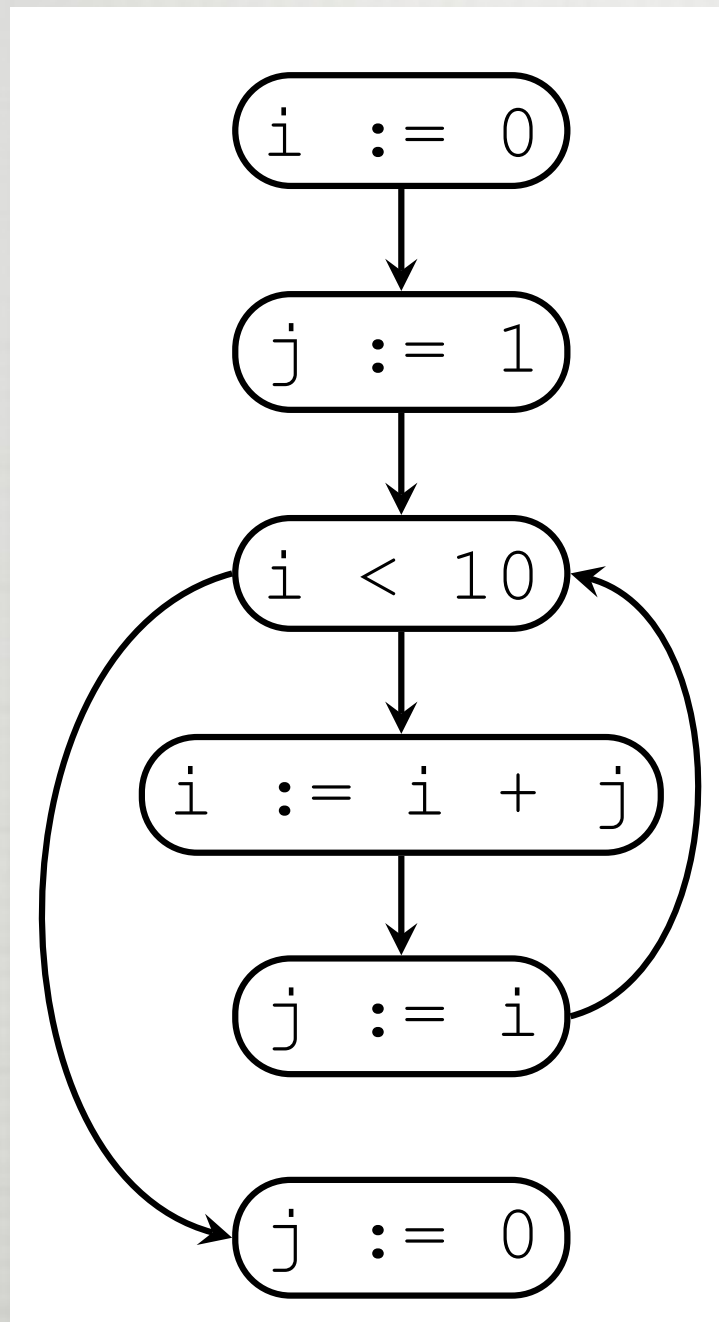
 Add predecessors of ℓ to *worklist*

end if

end while

$$\begin{aligned} \forall \ell. LV_{entry}(\ell) &= LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell) \\ LV_{exit}(\ell) &\neq \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell') \Rightarrow \ell \in worklist \end{aligned}$$

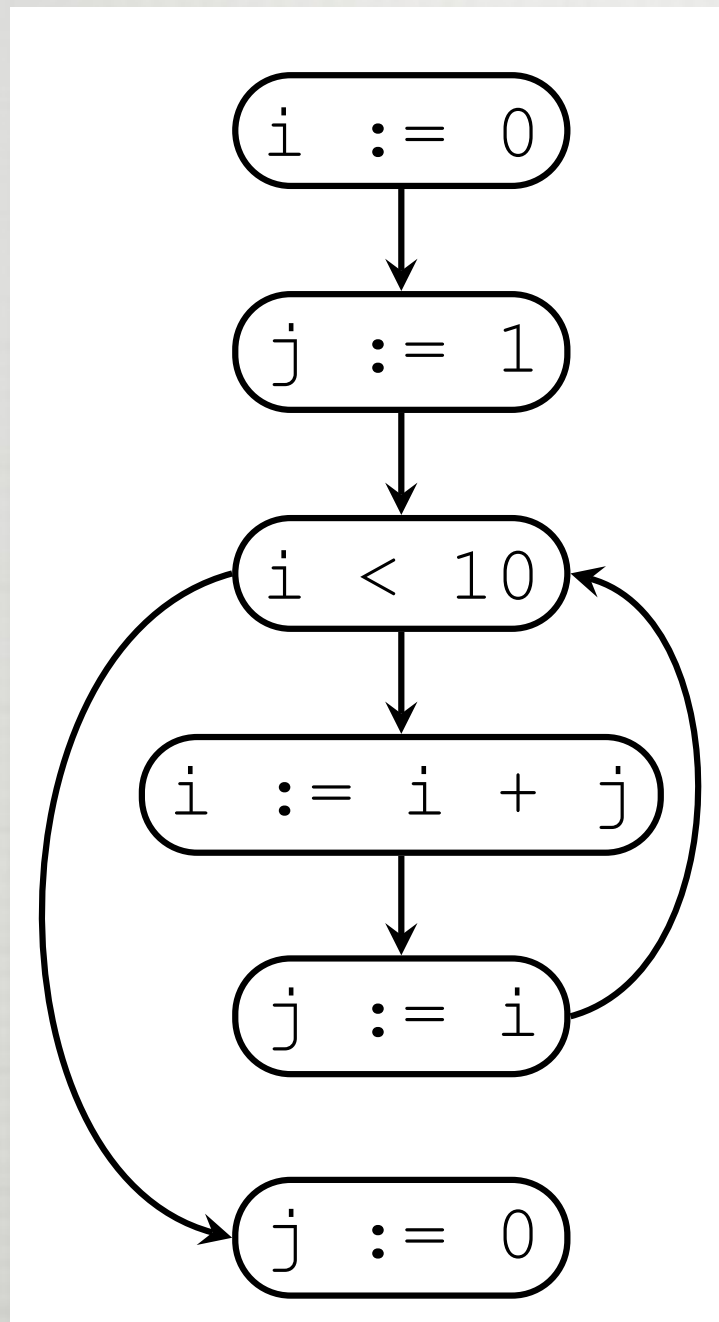
Worklist Algorithm Example



$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Worklist Algorithm Example

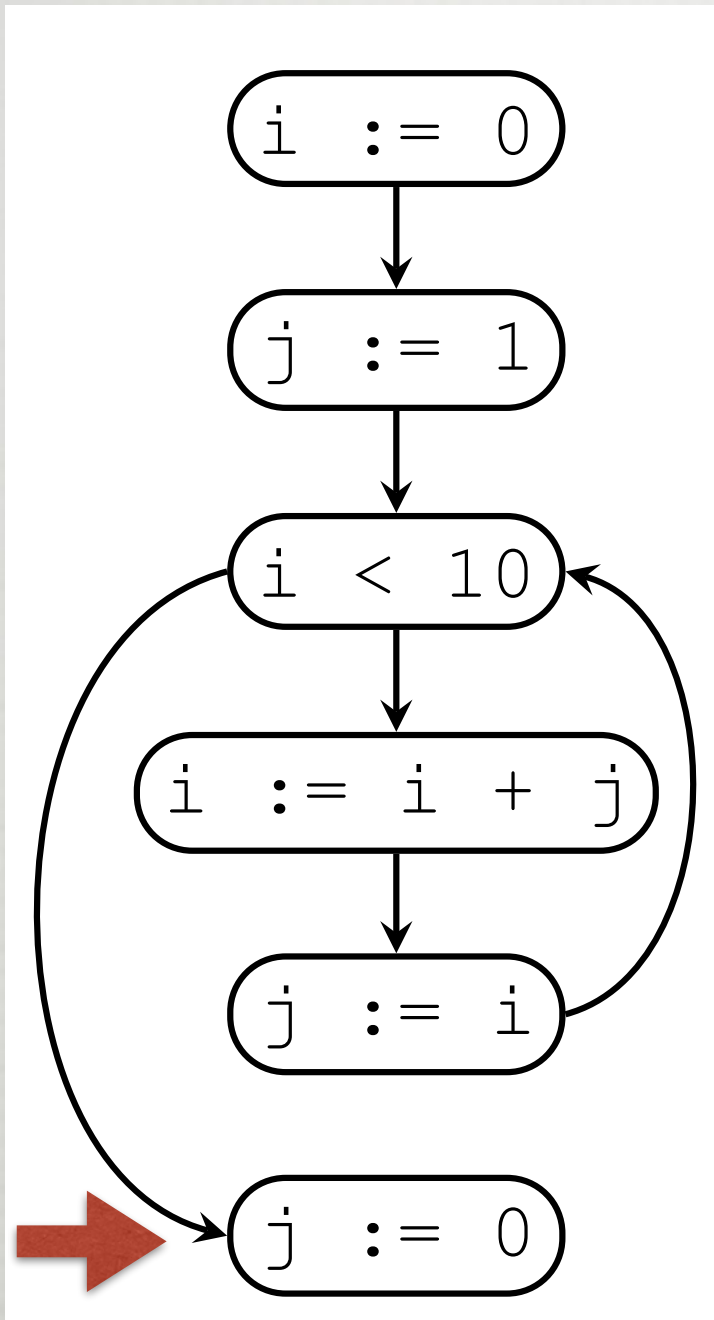


$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Label	LV_{exit}	LV_{entry}	In WL
$i := 0$	$\emptyset$	$\emptyset$	✓
$j := 1$	$\emptyset$	$\emptyset$	✓
$i < 10$	$\emptyset$	$\{i\}$	✓
$i := i + j$	$\emptyset$	$\{i, j\}$	✓
$j := i$	$\emptyset$	$\{i\}$	✓
$j := 0$	$\emptyset$	$\emptyset$	✓

Worklist Algorithm Example

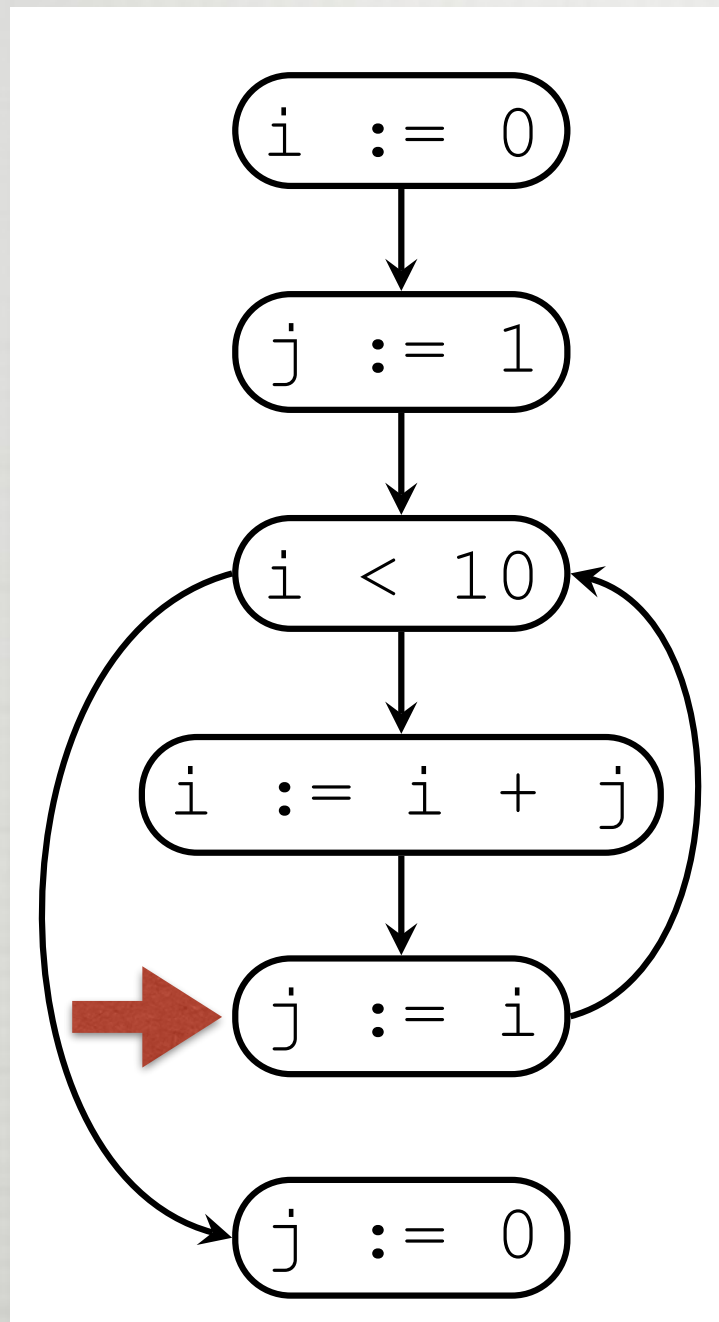


$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Label	LV_{exit}	LV_{entry}	In WL
$i := 0$	$\emptyset$	$\emptyset$	✓
$j := 1$	$\emptyset$	$\emptyset$	✓
$i < 10$	$\emptyset$	$\{i\}$	✓
$i := i + j$	$\emptyset$	$\{i, j\}$	✓
$j := i$	$\emptyset$	$\{i\}$	✓
$j := 0$	$\emptyset$	$\emptyset$	✓

Worklist Algorithm Example

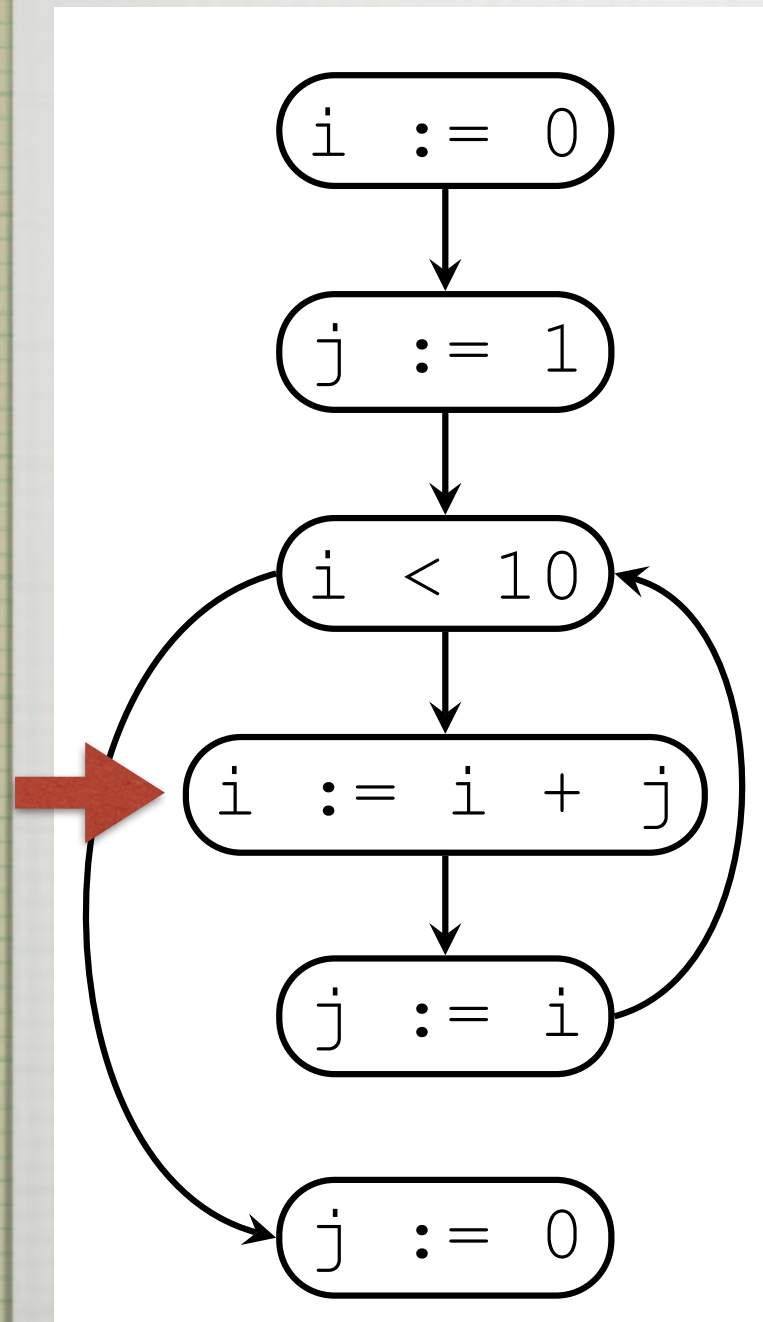


$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Label	LV_{exit}	LV_{entry}	In WL
$i := 0$	$\emptyset$	$\emptyset$	✓
$j := 1$	$\emptyset$	$\emptyset$	✓
$i < 10$	$\emptyset$	$\{i\}$	✓
$i := i + j$	$\emptyset$	$\{i, j\}$	✓
$j := i$	$\emptyset$	$\{i\}$	✓
$j := 0$	$\emptyset$	$\emptyset$	

Worklist Algorithm Example

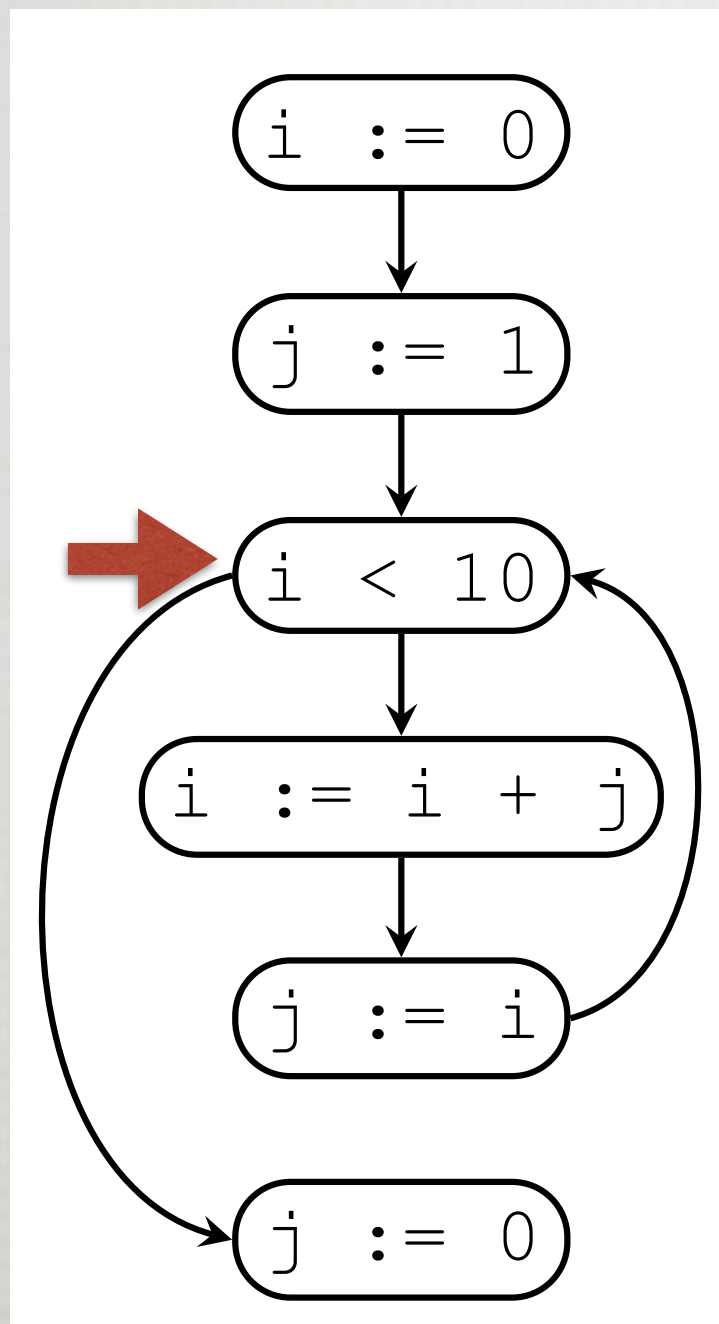


$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Label	LV_{exit}	LV_{entry}	In WL
i := 0	$\emptyset$	$\emptyset$	✓
j := 1	$\emptyset$	$\emptyset$	✓
i < 10	$\emptyset$	{i}	✓
i := i + j	$\emptyset$	{i, j}	✓
j := i	{i}	{i}	
j := 0	$\emptyset$	$\emptyset$	

Worklist Algorithm Example

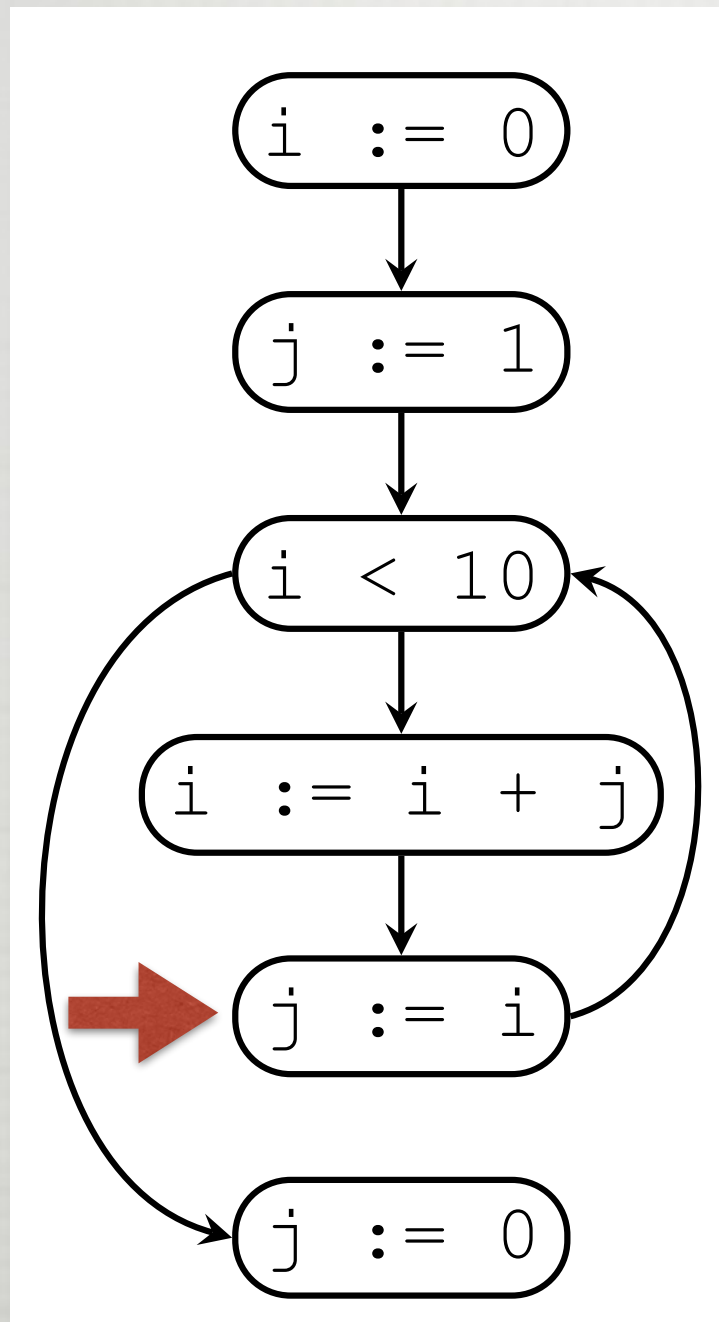


$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Label	LV_{exit}	LV_{entry}	In WL
$i := 0$	$\emptyset$	$\emptyset$	✓
$j := 1$	$\emptyset$	$\emptyset$	✓
$i < 10$	$\emptyset$	$\{i\}$	✓
$i := i + j$	$\{i\}$	$\{i, j\}$	
$j := i$	$\{i\}$	$\{i\}$	
$j := 0$	$\emptyset$	$\emptyset$	

Worklist Algorithm Example

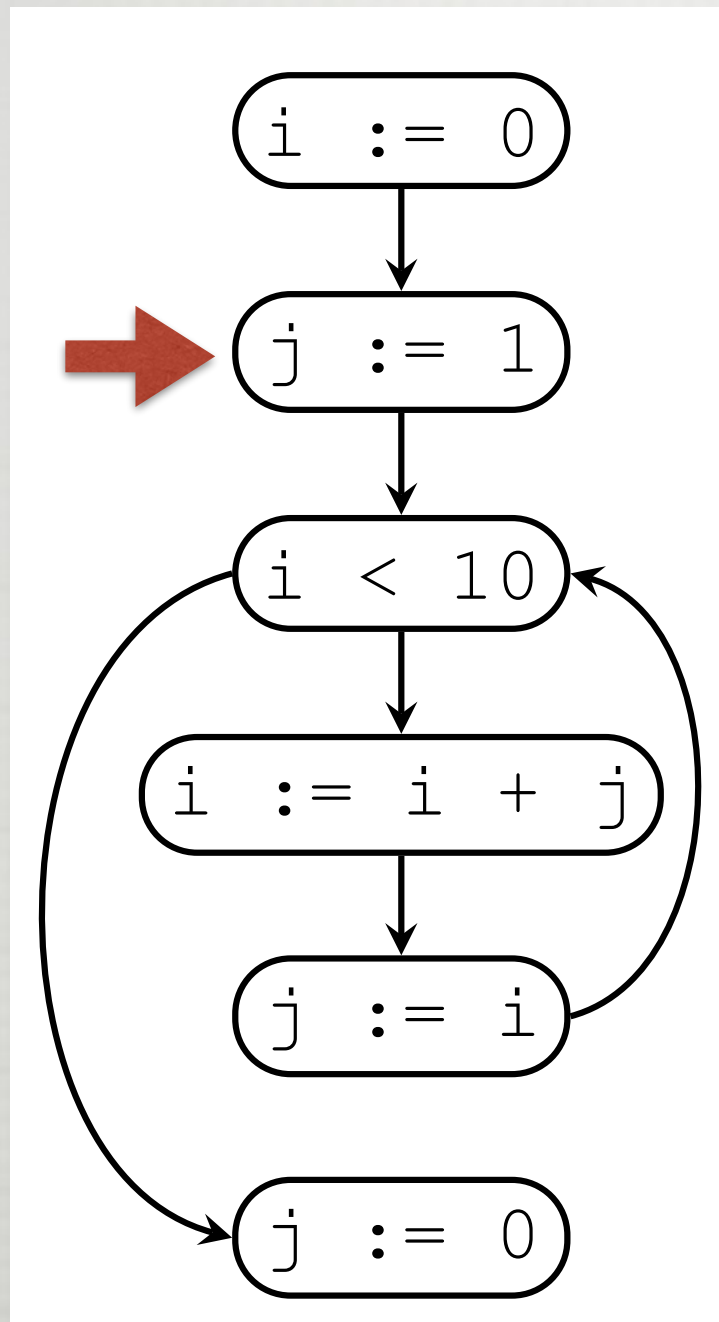


$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Label	LV_{exit}	LV_{entry}	In WL
$i := 0$	$\emptyset$	$\emptyset$	✓
$j := 1$	$\emptyset$	$\emptyset$	✓
$i < 10$	$\{i, j\}$	$\{i, j\}$	
$i := i + j$	$\{i\}$	$\{i, j\}$	
$j := i$	$\{i\}$	$\{i\}$	✓
$j := 0$	$\emptyset$	$\emptyset$	

Worklist Algorithm Example

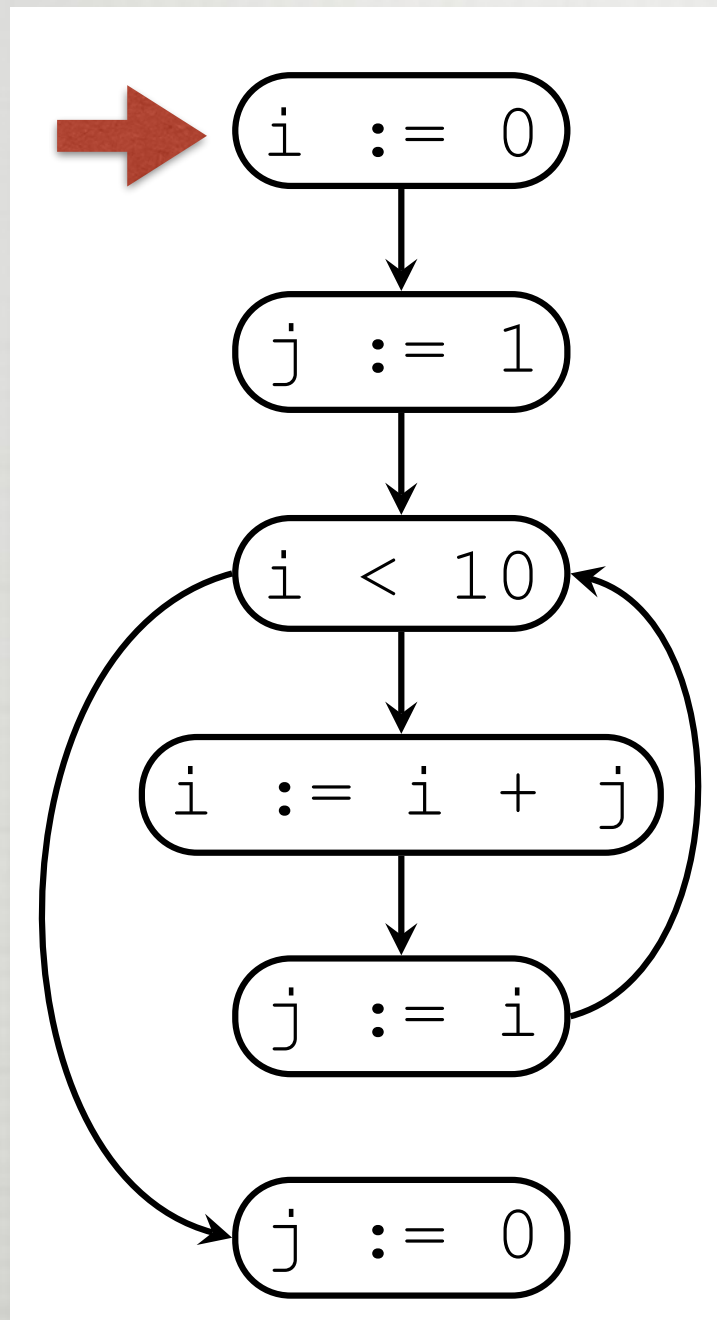


$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Label	LV_{exit}	LV_{entry}	In WL
$i := 0$	$\emptyset$	$\emptyset$	✓
$j := 1$	$\emptyset$	$\emptyset$	✓
$i < 10$	$\{i, j\}$	$\{i, j\}$	
$i := i + j$	$\{i\}$	$\{i, j\}$	
$j := i$	$\{i, j\}$	$\{i\}$	
$j := 0$	$\emptyset$	$\emptyset$	

Worklist Algorithm Example

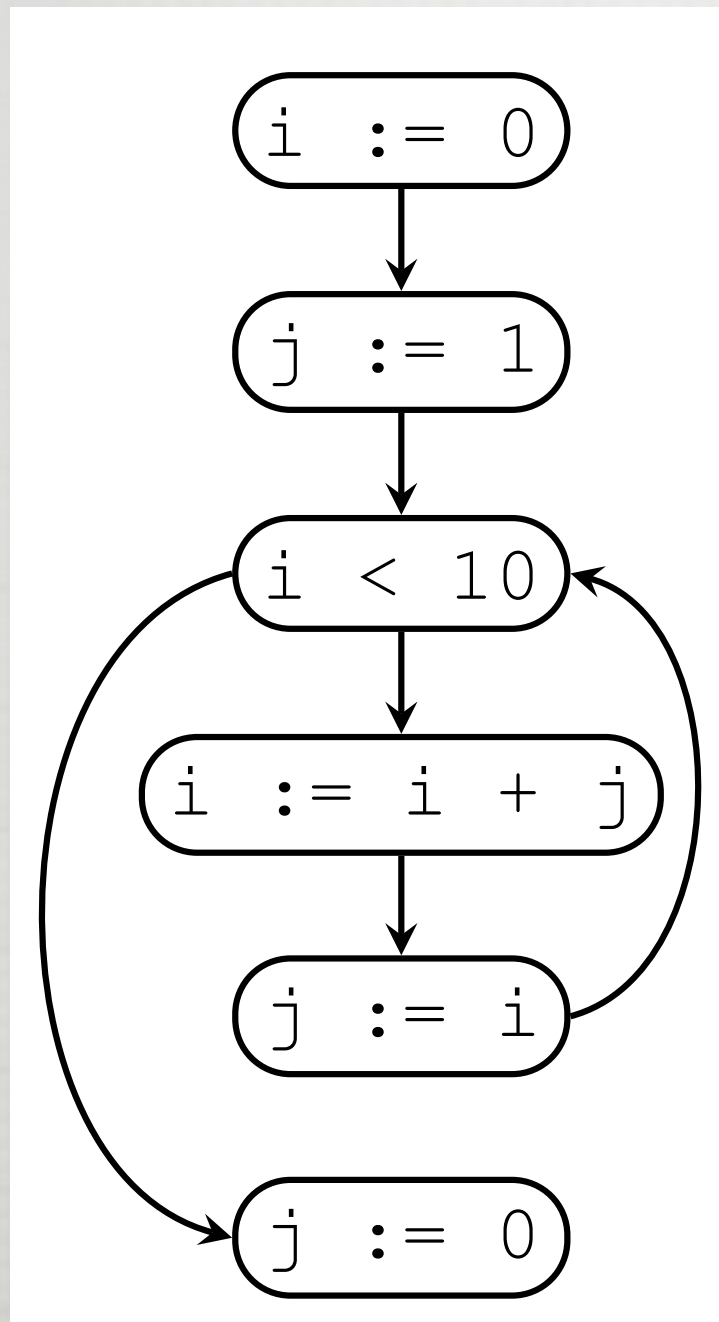


$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Label	LV_{exit}	LV_{entry}	In WL
$i := 0$	$\emptyset$	$\emptyset$	$\checkmark$
$j := 1$	$\{i, j\}$	$\{i\}$	
$i < 10$	$\{i, j\}$	$\{i, j\}$	
$i := i + j$	$\{i\}$	$\{i, j\}$	
$j := i$	$\{i, j\}$	$\{i\}$	
$j := 0$	$\emptyset$	$\emptyset$	

Worklist Algorithm Example



$$LV_{entry}(\ell) = LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell)$$

$$LV_{exit}(\ell) = \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell')$$

Label	LV_{exit}	LV_{entry}	In WL
$i := 0$	$\{i\}$	$\emptyset$	
$j := 1$	$\{i, j\}$	$\{i\}$	
$i < 10$	$\{i, j\}$	$\{i, j\}$	
$i := i + j$	$\{i\}$	$\{i, j\}$	
$j := i$	$\{i, j\}$	$\{i\}$	
$j := 0$	$\emptyset$	$\emptyset$	

Correctness

Worklist Algorithm Correctness

Worklist Algorithm Correctness

Why does the worklist algorithm always terminate?

Worklist Algorithm Correctness

Why does the worklist algorithm always terminate?

- $LV_{\text{entry}}(l)/LV_{\text{exit}}(l)$: always increase.

Worklist Algorithm Correctness

Why does the worklist algorithm always terminate?

- $LV_{\text{entry}}(l)/LV_{\text{exit}}(l)$: always increase.
- chains in $\mathcal{P}(\text{Vars})$ are finite.

Worklist Algorithm Correctness

Why does the worklist algorithm always terminate?

- ☐ $LV_{\text{entry}}(l)/LV_{\text{exit}}(l)$: always increase.
- ☐ chains in $\mathcal{P}(\text{Vars})$ are finite.
- ☐ Either the sets increase, or the worklist decreases.

Worklist Algorithm Correctness

Why does the worklist algorithm always terminate?

- ☐ $LV_{\text{entry}}(l)/LV_{\text{exit}}(l)$: always increase.
- ☐ chains in $\mathcal{P}(\text{Vars})$ are finite.
- ☐ Either the sets increase, or the worklist decreases.

Why does the worklist algorithm compute the least solution?

Worklist Algorithm Correctness

Why does the worklist algorithm always terminate?

- ☐ $LV_{\text{entry}}(l)/LV_{\text{exit}}(l)$: always increase.
- ☐ chains in $\mathcal{P}(\text{Vars})$ are finite.
- ☐ Either the sets increase, or the worklist decreases.

Why does the worklist algorithm compute the least solution?

- ☐ Loop invariant for the algorithm:

Worklist Algorithm Correctness

Why does the worklist algorithm always terminate?

- ☐ $LV_{\text{entry}}(\ell)/LV_{\text{exit}}(\ell)$: always increase.
- ☐ chains in $\mathcal{P}(\text{Vars})$ are finite.
- ☐ Either the sets increase, or the worklist decreases.

Why does the worklist algorithm compute the least solution?

- ☐ Loop invariant for the algorithm:

$$\begin{aligned} LV_{\text{exit}}(\ell) &\subseteq \text{live}_{\text{exit}}(\ell) \\ LV_{\text{entry}}(\ell) &\subseteq \text{live}_{\text{entry}}(\ell) \end{aligned}$$

Worklist Algorithm Correctness

Why does the worklist algorithm always terminate?

- ☐ $LV_{\text{entry}}(\ell)/LV_{\text{exit}}(\ell)$: always increase.
- ☐ chains in $\mathcal{P}(\text{Vars})$ are finite.
- ☐ Either the sets increase, or the worklist decreases.

Why does the worklist algorithm compute the least solution?

- ☐ Loop invariant for the algorithm:
- ☐ sets are increasing

$$\begin{aligned} LV_{\text{exit}}(\ell) &\subseteq \text{live}_{\text{exit}}(\ell) \\ LV_{\text{entry}}(\ell) &\subseteq \text{live}_{\text{entry}}(\ell) \end{aligned}$$

What about other
analyses?

Let us use a bit of basic
algebra!

First Structure

Partially Ordered Sets

$(S, \sqsubseteq)$: a set S and a (partial) order relation $\sqsubseteq$

- $\sqsubseteq$ is reflexive, transitive, and anti-symmetric

Partially Ordered Sets

Information Order

$(S, \sqsubseteq)$: a set S and a (partial) order relation $\sqsubseteq$

- $\sqsubseteq$ is reflexive, transitive, and anti-symmetric

Partially Ordered Sets

Information Order

$(S, \sqsubseteq)$: a set S and a (partial) order relation $\sqsubseteq$

- $\sqsubseteq$ is reflexive, transitive, and anti-symmetric

$x \sqsubseteq x$ (reflexivity).

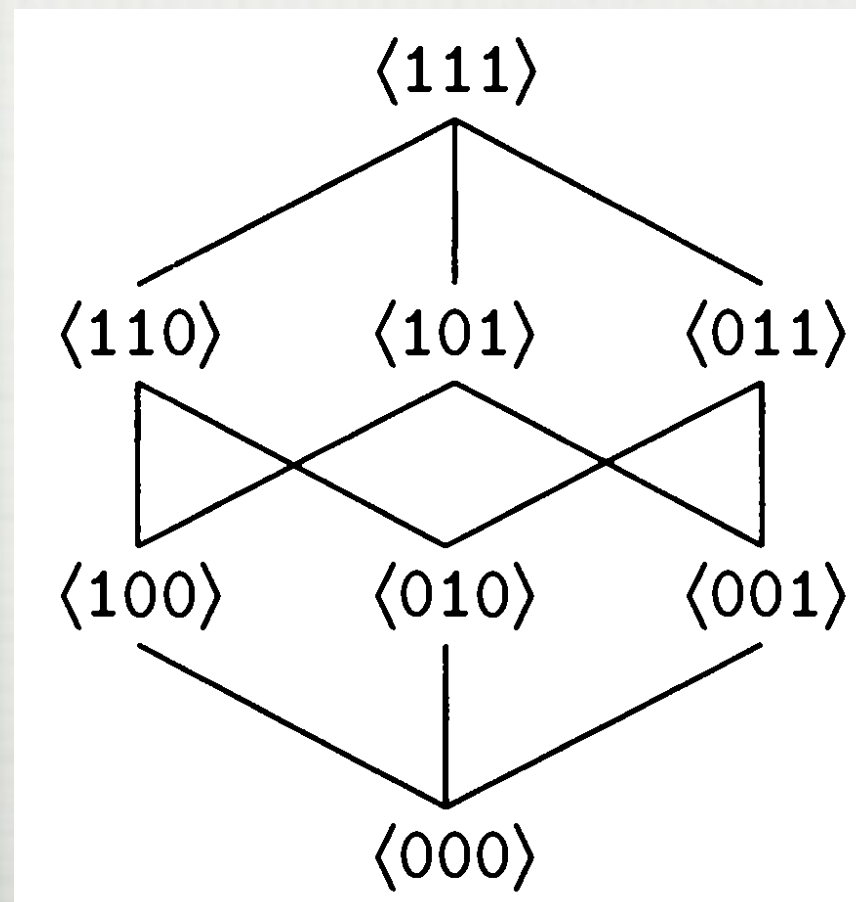
If $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$ (transitivity).

If $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$ (antisymmetry).

Partially Ordered Sets

$(S, \sqsubseteq)$: a set S and a (partial) order relation $\sqsubseteq$

- $\sqsubseteq$ is reflexive, transitive, and anti-symmetric



Second Structure

Semi-Lattices

A (meet) *semi-lattice* $\mathbf{L} = (S, \sqcap)$ is a set S with a binary operation, called *meet* ($\sqcap$), that has the following properties:

(1) For all $x, y \in S$, there exist a **unique** $z \in S$ such that $x \sqcap y = z$ (**CLOSURE**).

(2) For all $x, y, z \in S$, we have

$$x \sqcap x = x \quad (\text{idempotence})$$

$$x \sqcap y = y \sqcap x \quad (\text{commutativity})$$

$$x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z \quad (\text{associativity})$$

Complete Semi-Lattices

The **unit** for $\sqcap$ is $\top$:

$$\forall x : x \sqcap \top = \top \sqcap x = x$$

Complete Semi-Lattices

The **unit** for $\sqcap$ is $\top$:

$$\forall x : x \sqcap \top = \top \sqcap x = x$$

The meet semi-lattice is called **complete** if $\top \in \mathbb{L}$

The Connection

The Connection Between The Structures

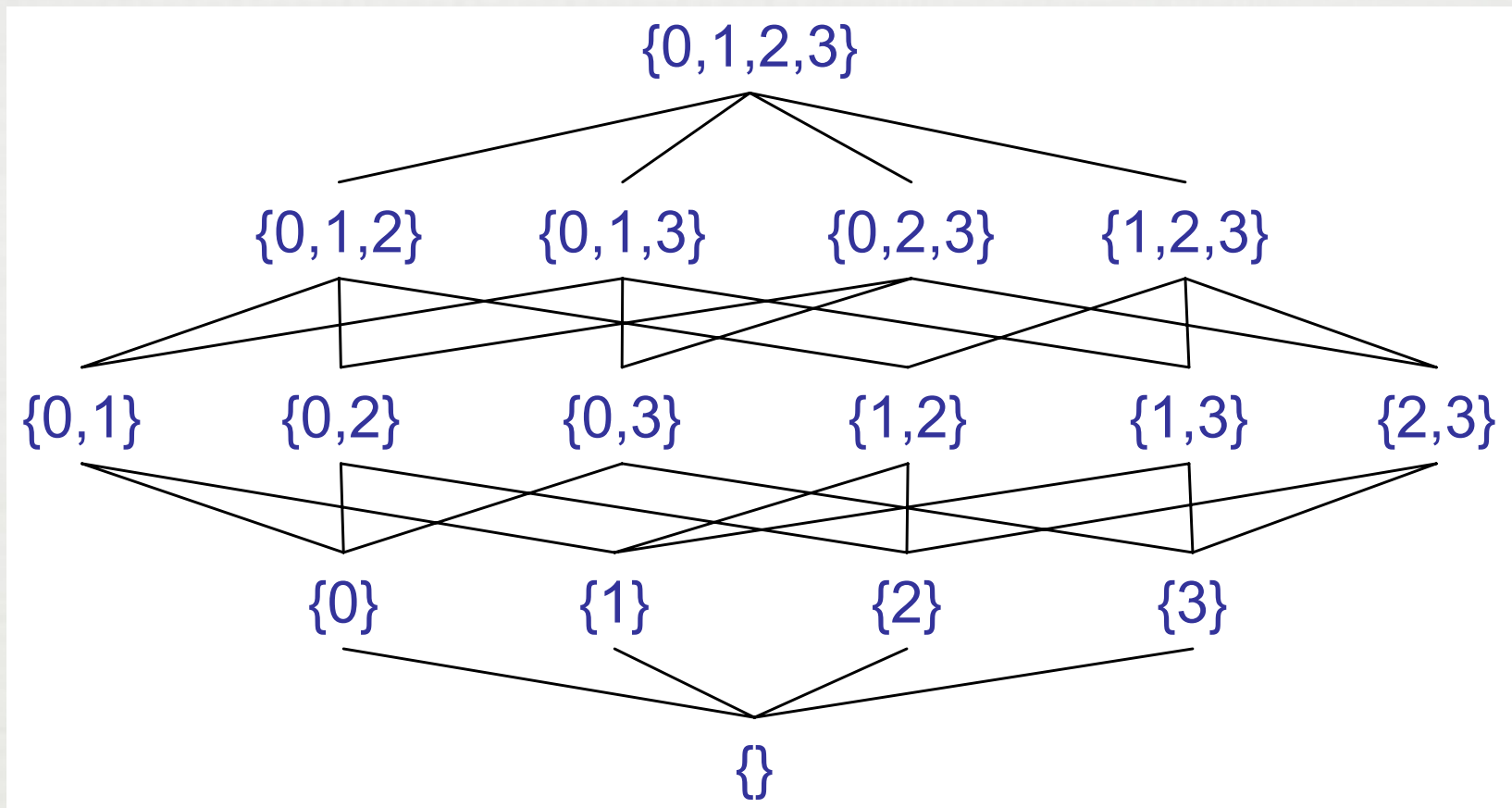
Given a semi-lattice and define a binary operation $\sqsubseteq$:

$$x \sqsubseteq y \text{ if and only if } x \sqcap y = x$$

$\sqsubseteq$ is provably a **partial order relation**.

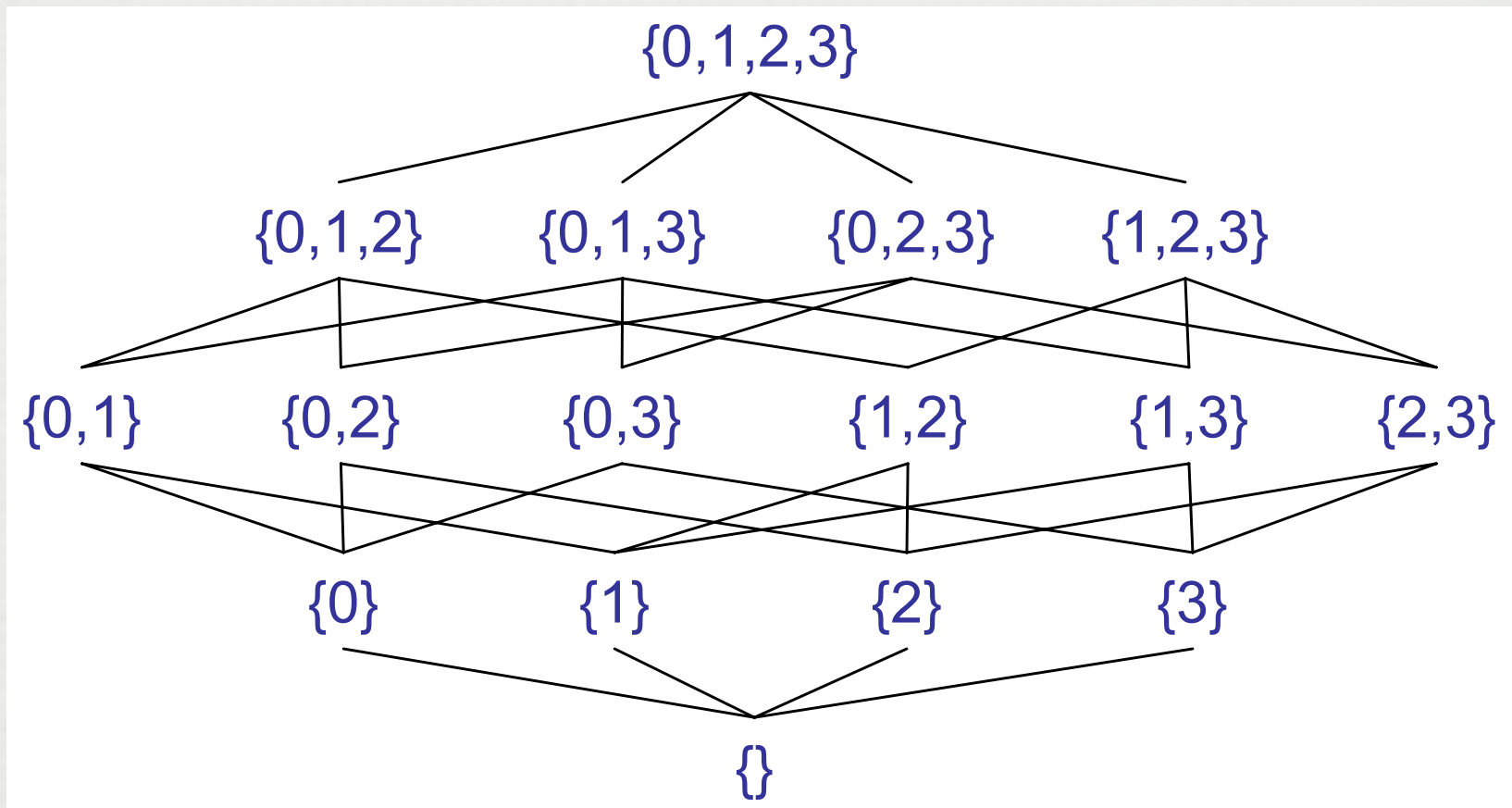
$\sqcap$ is provably the **greatest lower bound** defined based on $\sqsubseteq$.

Example: Subset SemiLattice



$(\mathcal{P}(\mathcal{S}), \cap)$ is a complete meet semi-lattice.

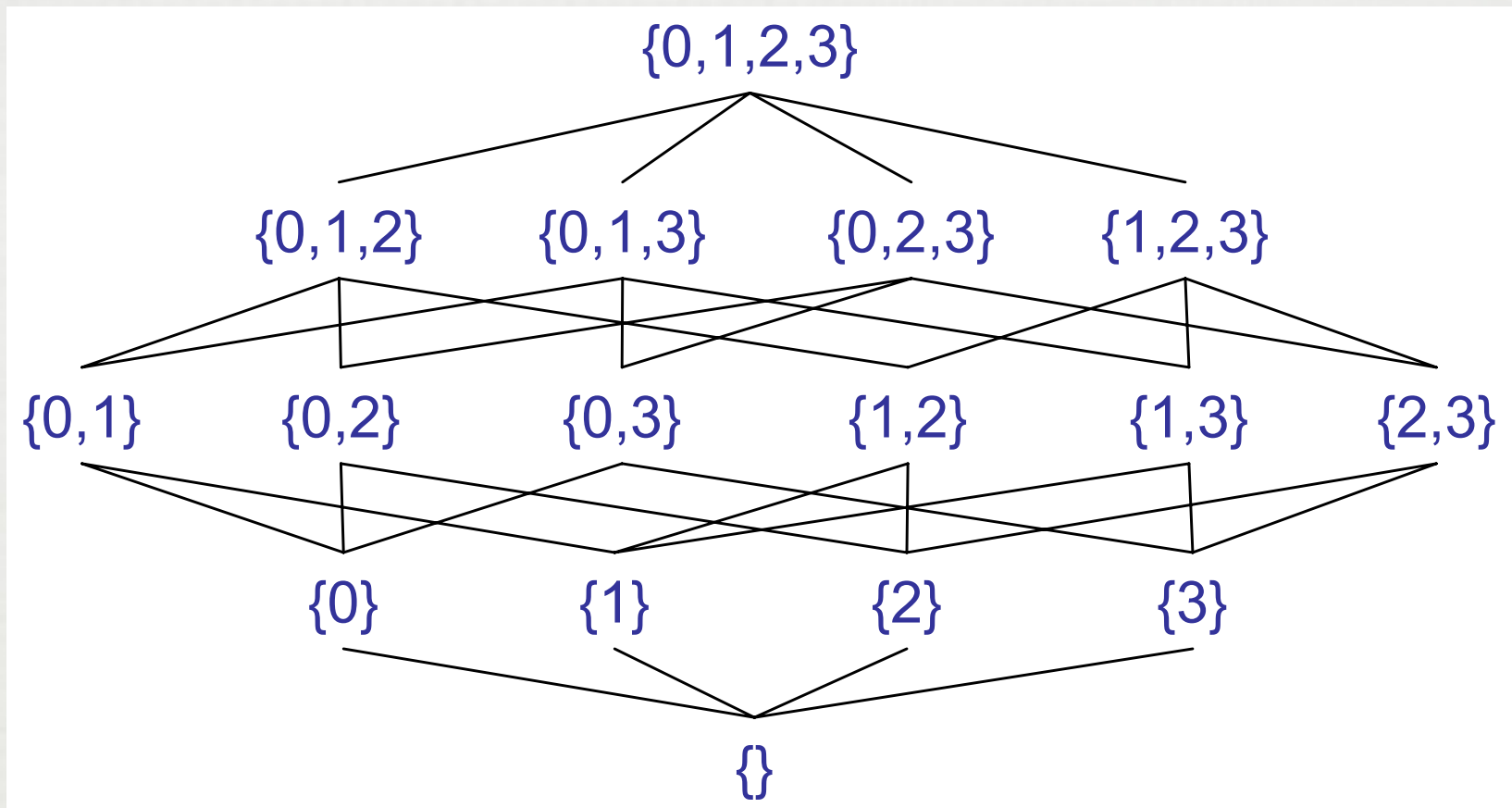
Example: Subset SemiLattice



$(\mathcal{P}(\mathcal{S}), \cap)$ is a complete meet semi-lattice.

$$\sqsubseteq = \subseteq$$

Example: Subset SemiLattice



$(\mathcal{P}(S), \cap)$ is a complete meet semi-lattice.

$$\sqsubseteq = \subseteq$$

$$\top = S$$

The Converse

Given a partially ordered set $(S, \sqsubseteq)$, where the greatest lower bound of every pair of elements is defined, let:

$x \sqcap y =$ the greatest lower bound according to $\sqsubseteq$

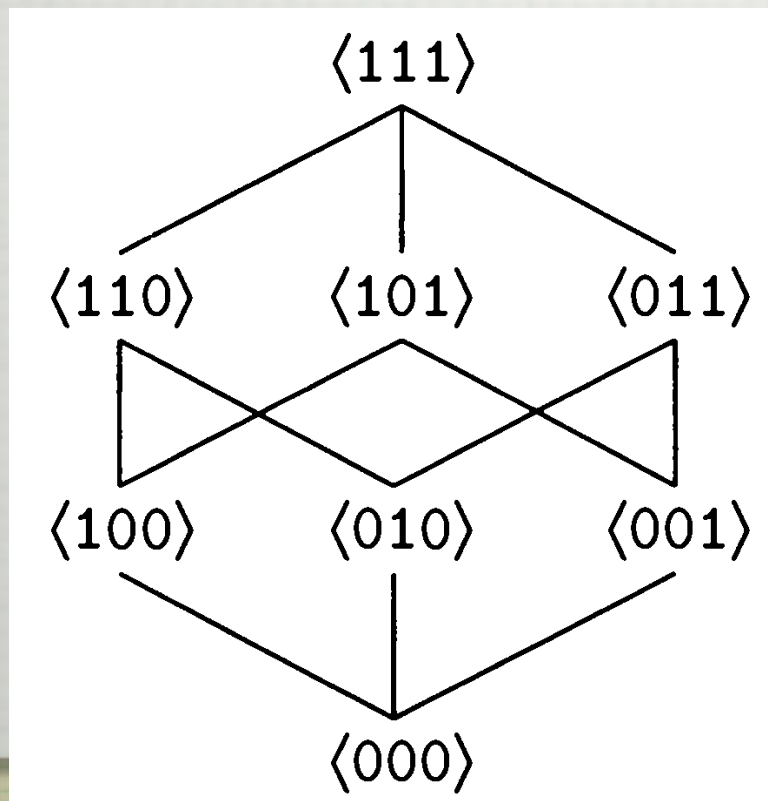
$(S, \sqcap)$ is provably a **semi-lattice**.

The Converse

Given a partially ordered set $(S, \sqsubseteq)$, where the greatest lower bound of every pair of elements is defined, let:

$x \sqcap y =$ the greatest lower bound according to $\sqsubseteq$

$(S, \sqcap)$ is provably a **semi-lattice**.

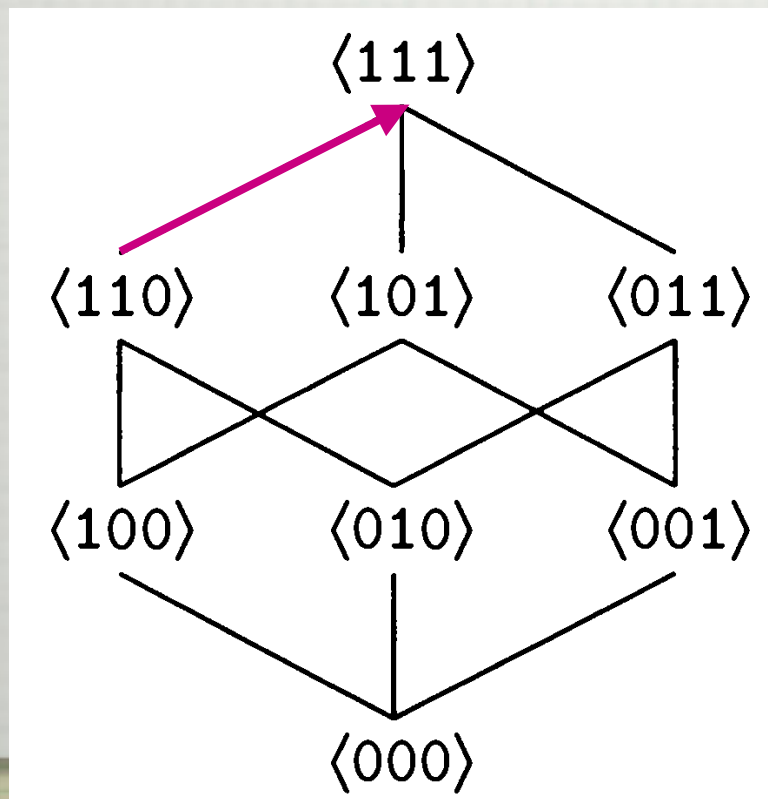


The Converse

Given a partially ordered set $(S, \sqsubseteq)$, where the greatest lower bound of every pair of elements is defined, let:

$x \sqcap y =$ the greatest lower bound according to $\sqsubseteq$

$(S, \sqcap)$ is provably a **semi-lattice**.

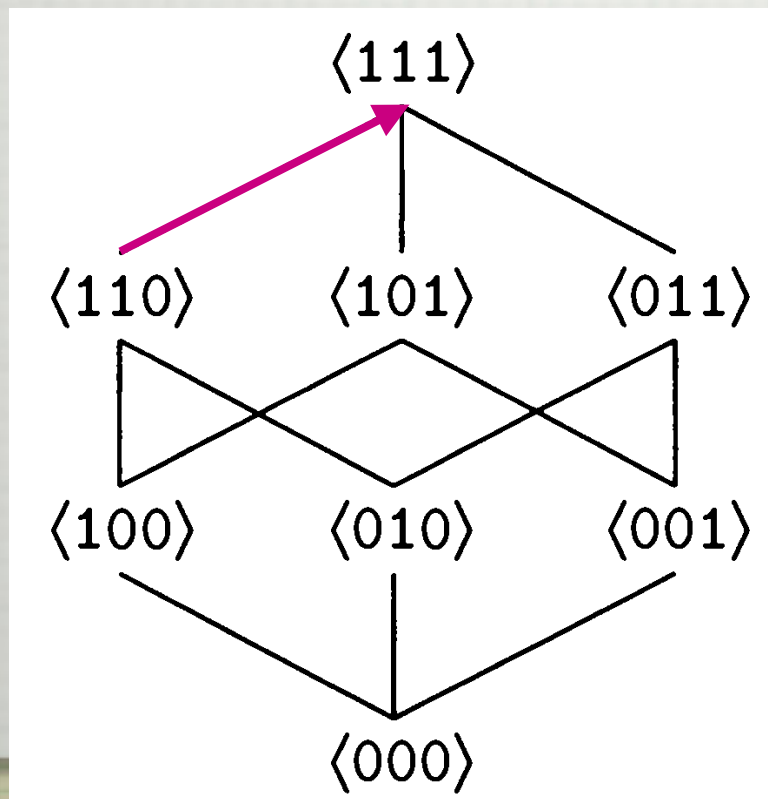


The Converse

Given a partially ordered set $(S, \sqsubseteq)$, where the greatest lower bound of every pair of elements is defined, let:

$x \sqcap y =$ the greatest lower bound according to $\sqsubseteq$

$(S, \sqcap)$ is provably a **semi-lattice**.



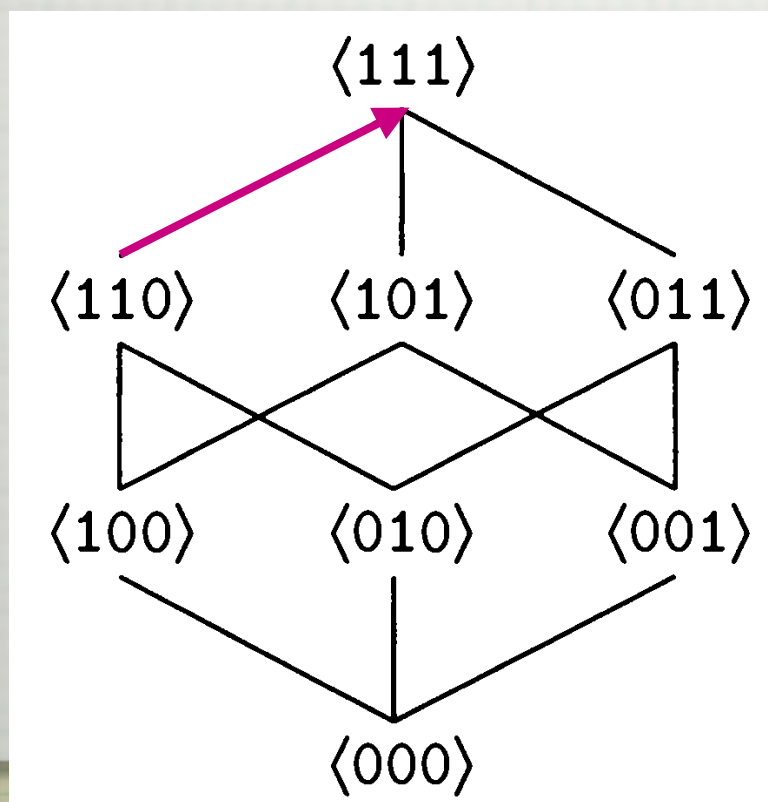
$$\sqcap = \wedge$$

The Converse

Given a partially ordered set $(S, \sqsubseteq)$, where the greatest lower bound of every pair of elements is defined, let:

$x \sqcap y =$ the greatest lower bound according to $\sqsubseteq$

$(S, \sqcap)$ is provably a **semi-lattice**.



$$\sqcap = \wedge$$

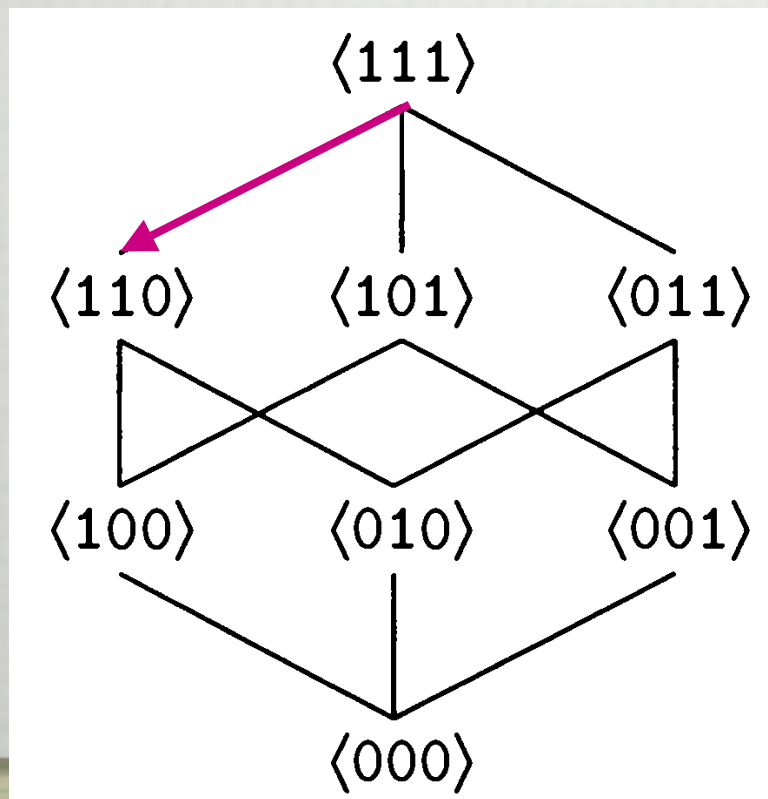
$$\langle 110 \rangle \sqcap \langle 011 \rangle = \langle 110 \rangle \wedge \langle 011 \rangle = \langle 010 \rangle$$

The Converse

Given a partially ordered set $(S, \sqsubseteq)$, where the greatest lower bound of every pair of elements is defined, let:

$x \sqcap y =$ the greatest lower bound according to $\sqsubseteq$

$(S, \sqcap)$ is provably a **semi-lattice**.

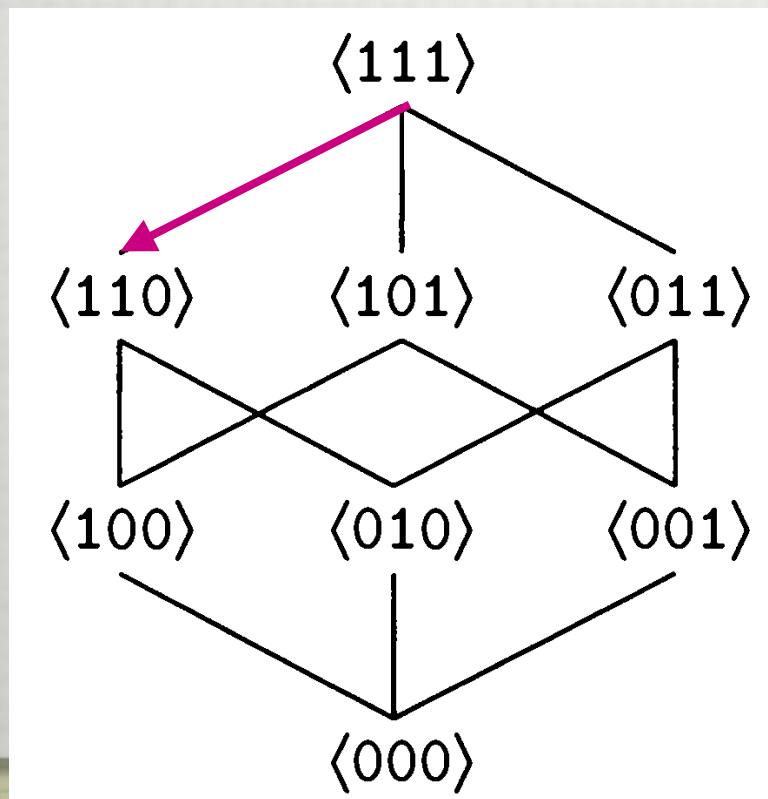


The Converse

Given a partially ordered set $(S, \sqsubseteq)$, where the greatest lower bound of every pair of elements is defined, let:

$x \sqcap y =$ the greatest lower bound according to $\sqsubseteq$

$(S, \sqcap)$ is provably a **semi-lattice**.



$$\sqcap = \vee$$

$$\langle 110 \rangle \sqcap \langle 011 \rangle = \langle 110 \rangle \vee \langle 011 \rangle = \langle 111 \rangle$$

Semi-Lattices

A (join) *semi-lattice* $\mathbf{L} = (S, \sqcup)$ is a set S with a binary operation, called *join* ($\sqcup$), that has the following properties:

(1) For all $x, y \in S$, there exist a **unique** $z \in S$ such that $x \sqcup y = z$ **(CLOSURE)**.

(2) For all $x, y, z \in S$, we have

$$x \sqcup x = x \quad \text{(idempotence)}$$

$$x \sqcup y = y \sqcup x \quad \text{(commutativity)}$$

$$x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z \quad \text{(associativity)}$$

Complete Join Semi-Lattice

The **unit** for $\sqcup$ is $\perp$:

$$\forall x : x \sqcup \perp = \perp \sqcup x = x$$

The join semi-lattice is called **complete** if $\perp \in \mathbb{L}$

Defining The Order Based on Join

Given a semi-lattice and define a binary operation $\sqsubseteq$:

$$x \sqsubseteq y \text{ if and only if } x \sqcup y = y$$

$\sqsubseteq$ is provably a **partial order relation**.

$\sqcup$ is provably the **least upper bound** defined based on $\sqsubseteq$.

Lattices

A *lattice* $\mathbf{L} = (S, \sqcap, \sqcup)$ is a set S with two binary operations, a *meet* ($\sqcap$) and a *join* ($\sqcup$), such that:

Lattices

A *lattice* $\mathbf{L} = (S, \sqcap, \sqcup)$ is a set S with two binary operations, a *meet* ($\sqcap$) and a *join* ($\sqcup$), such that:

- (1) $(S, \sqcap)$ is a *meet* semi-lattice.

Lattices

A *lattice* $\mathbf{L} = (S, \sqcap, \sqcup)$ is a set S with two binary operations, a *meet* ($\sqcap$) and a *join* ($\sqcup$), such that:

- (1) $(S, \sqcap)$ is a *meet* semi-lattice.
- (2) $(S, \sqcup)$ is a *join* semi-lattice.

Lattices

A *lattice* $\mathbf{L} = (S, \sqcap, \sqcup)$ is a set S with two binary operations, a *meet* ($\sqcap$) and a *join* ($\sqcup$), such that:

- (1) $(S, \sqcap)$ is a *meet* semi-lattice.
- (2) $(S, \sqcup)$ is a *join* semi-lattice.
- (3) The following *absorption law* holds:

$$x \sqcup (x \sqcap y) = x \sqcap (x \sqcup y) = x$$

Lattices

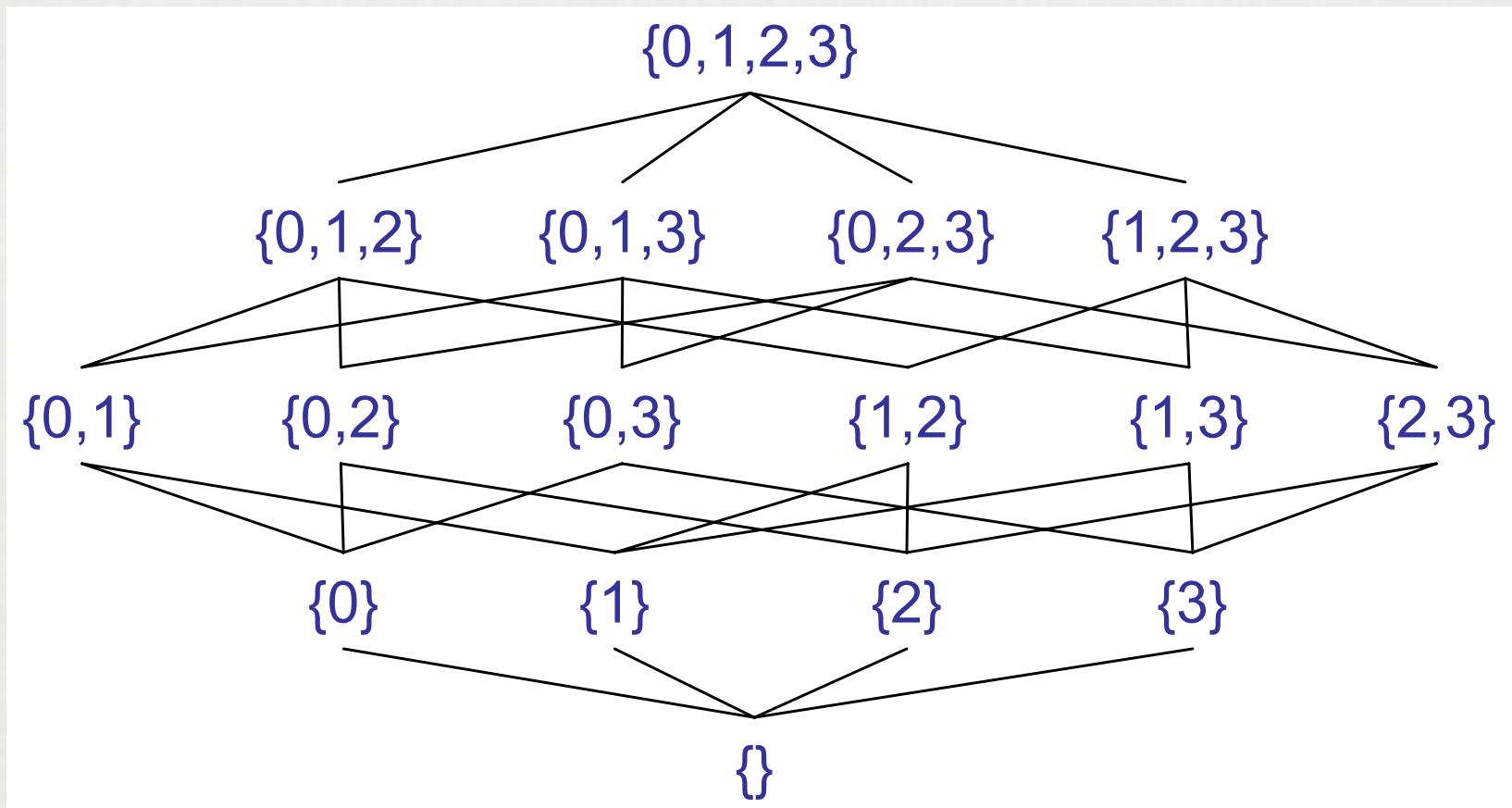
A *lattice* $\mathbf{L} = (S, \sqcap, \sqcup)$ is a set S with two binary operations, a *meet* ($\sqcap$) and a *join* ($\sqcup$), such that:

- (1) $(S, \sqcap)$ is a *meet* semi-lattice.
- (2) $(S, \sqcup)$ is a *join* semi-lattice.
- (3) The following *absorption law* holds:

$$x \sqcup (x \sqcap y) = x \sqcap (x \sqcup y) = x$$

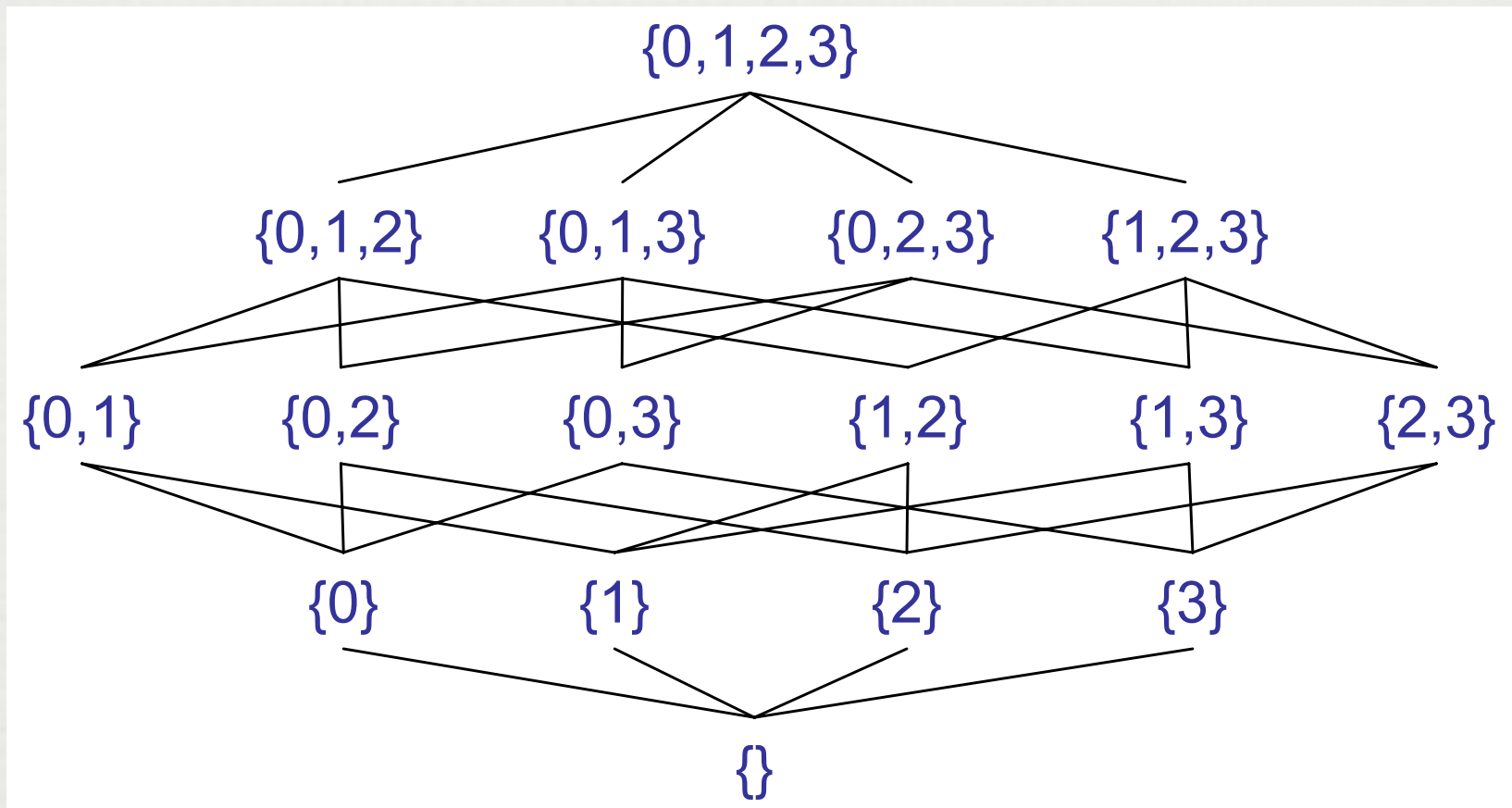
If both $\top$ and $\perp$ exist then the lattice is called *complete*.

Example: Subset Lattice



For a set S , $(\mathcal{P}(S), \cap, \cup)$ forms a complete lattice.

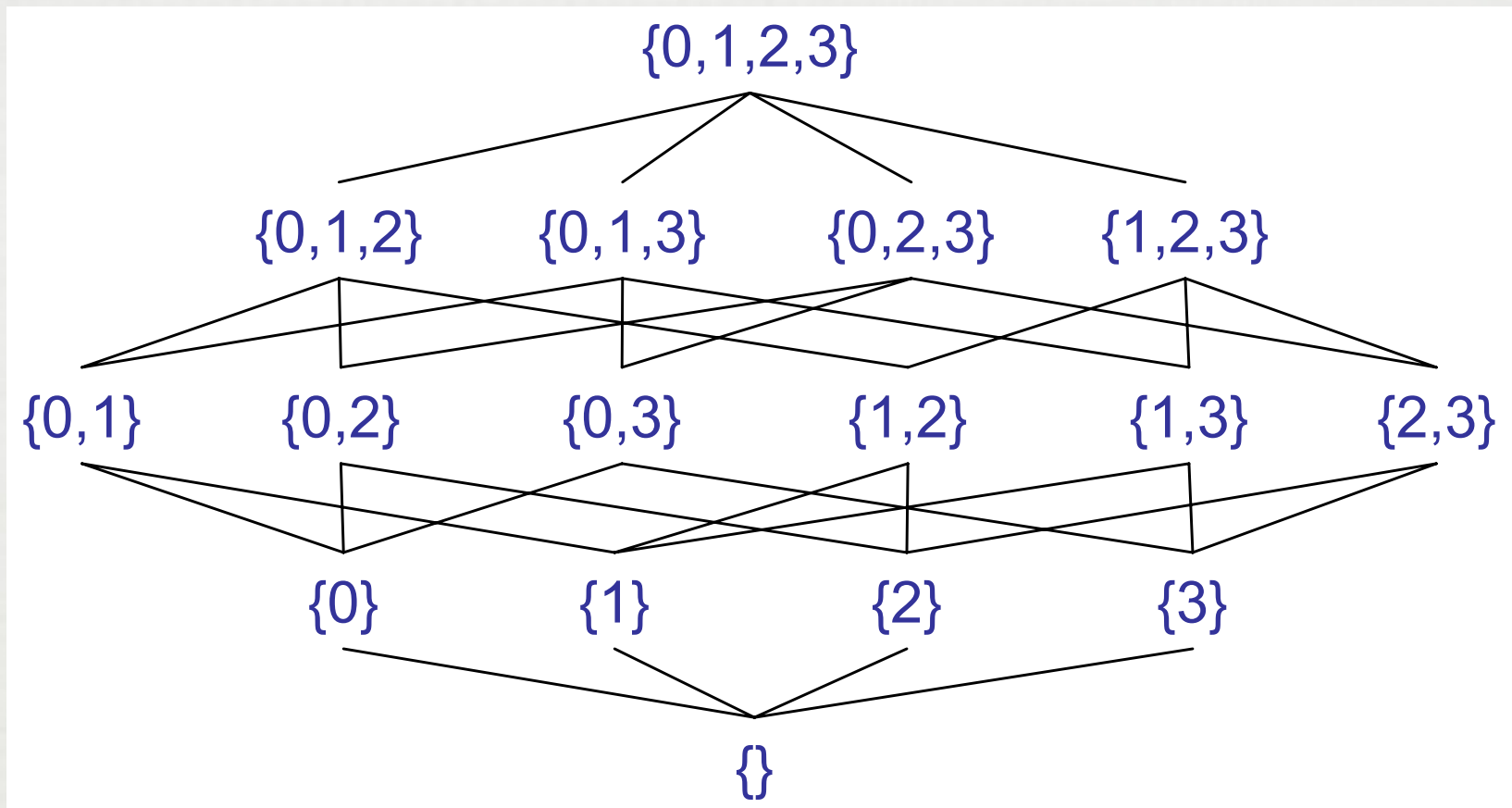
Example: Subset Lattice



For a set S , $(\mathcal{P}(S), \cap, \cup)$ forms a complete lattice.

$$\sqsubseteq = \subseteq$$

Example: Subset Lattice

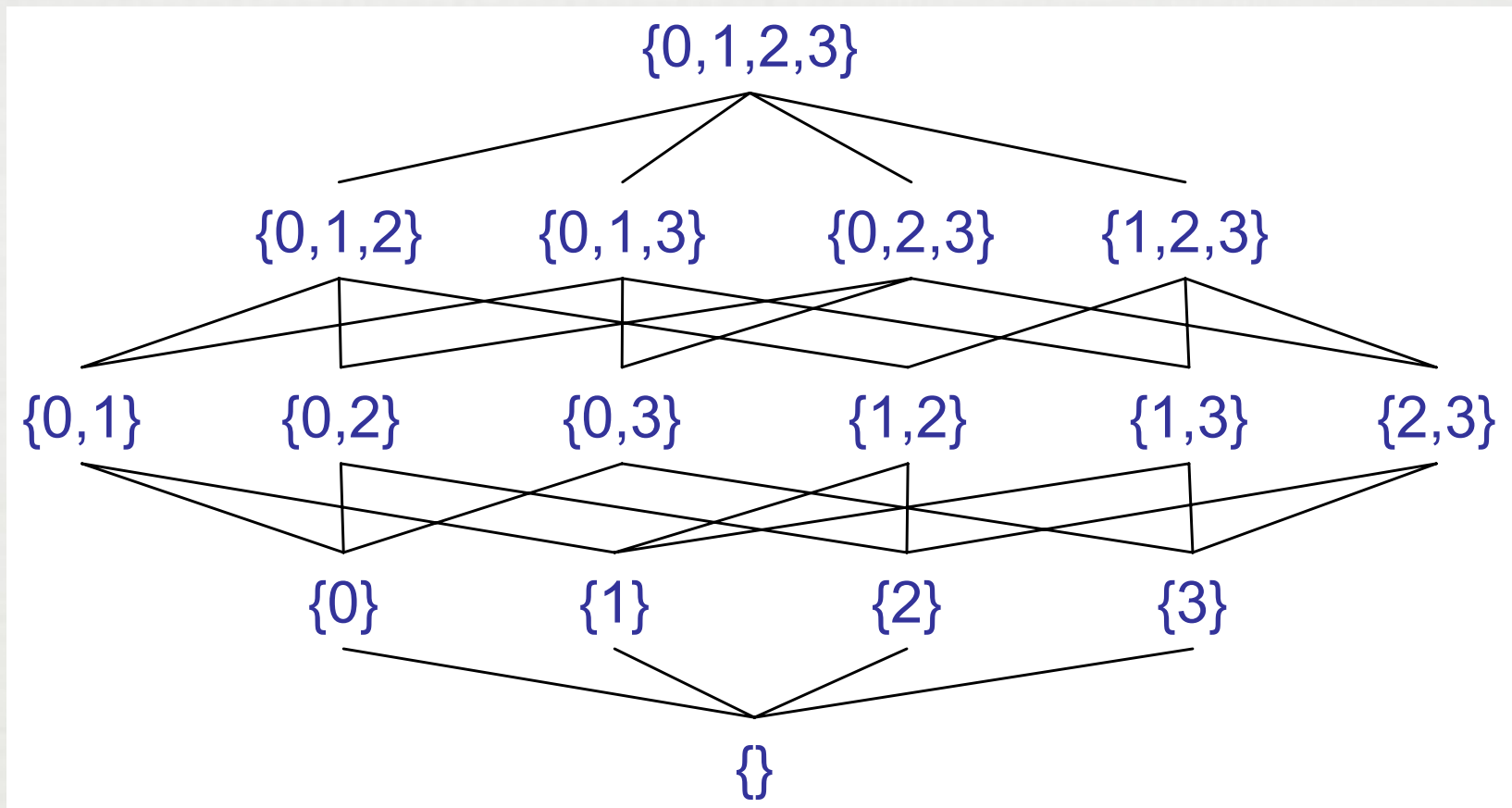


For a set S , $(\mathcal{P}(S), \cap, \cup)$ forms a complete lattice.

$$\sqsubseteq = \subseteq$$

$$\perp = \emptyset$$

Example: Subset Lattice



For a set S , $(\mathcal{P}(S), \cap, \cup)$ forms a complete lattice.

$$\sqsubseteq = \subseteq$$

$$\perp = \emptyset$$

$$\top = S$$

Recall Live Variable Analysis

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

Recall Live Variable Analysis

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

$$x \text{ is live at } l \Rightarrow x \in \text{Live}_{\text{exit}}(l)$$

Recall Live Variable Analysis

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

$$x \text{ is live at } l \Rightarrow x \in \text{Live}_{\text{exit}}(l)$$

Our **domain** is a **subset lattice** where S is the set of all variables!

Descending Chains

A **descending chain** is a sequence of elements related by the order:

$$x_1 \supseteq x_2 \supseteq \cdots \supseteq x_n$$

Descending Chains

A **descending chain** is a sequence of elements related by the order:

$$x_1 \supseteq x_2 \supseteq \cdots \supseteq x_n$$

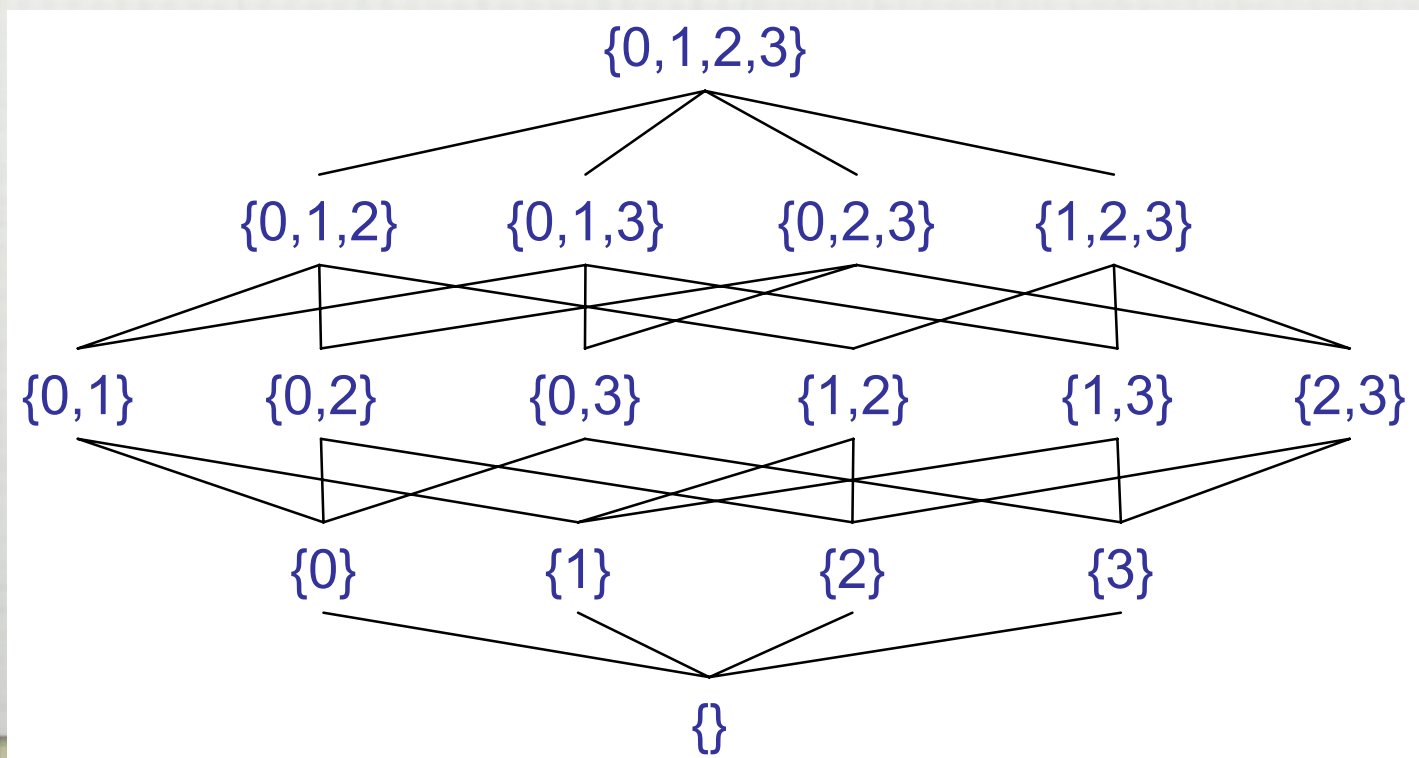
The **height** of a lattice (or semi-lattice) is the length of its longest descending chain.

Descending Chains

A **descending chain** is a sequence of elements related by the order:

$$x_1 \supseteq x_2 \supseteq \cdots \supseteq x_n$$

The **height** of a lattice (or semi-lattice) is the length of its longest descending chain.



Descending Chains

A **descending chain** is a sequence of elements related by the order:

$$x_1 \supseteq x_2 \supseteq \cdots \supseteq x_n$$

The **height** of a lattice (or semi-lattice) is the length of its longest descending chain.

Useful for Algorithmic convergence: a finite height!

Recall Live Variable Analysis

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

$$x \text{ is live at } l \Rightarrow x \in \text{LV}_{\text{exit}}(l)$$

Our **domain** is a **subset lattice** where S is the set of all variables!

Recall Live Variable Analysis

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

$$x \text{ is live at } l \Rightarrow x \in \text{LV}_{\text{exit}}(l)$$

Our **domain** is a **subset lattice** where S is the set of all variables!

What is the height of this lattice?

Recall Live Variable Analysis

Any solution $\text{Live}_{\text{entry}}$, $\text{Live}_{\text{exit}}$ to this system of equations **overapproximates** the true set of live variables.

$$x \text{ is live at } l \Rightarrow x \in \text{LV}_{\text{exit}}(l)$$

Our **domain** is a **subset lattice** where S is the set of all variables!

What is the height of this lattice?

The worklist algorithm **terminates** because this lattice has a **finite height**!

An Infinite Lattice

The aim of the *Constant Propagation Analysis* is to determine

For each program point, whether or not a variable has a constant value whenever execution reaches that point.

An Infinite Lattice

The aim of the *Constant Propagation Analysis* is to determine

For each program point, whether or not a variable has a constant value whenever execution reaches that point.

```
x := 2
y := 5
x := 1
z := 0
if (x <= 0) {
    z := x + 2
} else {
    z := y * y
}
x := z
```

An Infinite Lattice

The aim of the *Constant Propagation Analysis* is to determine

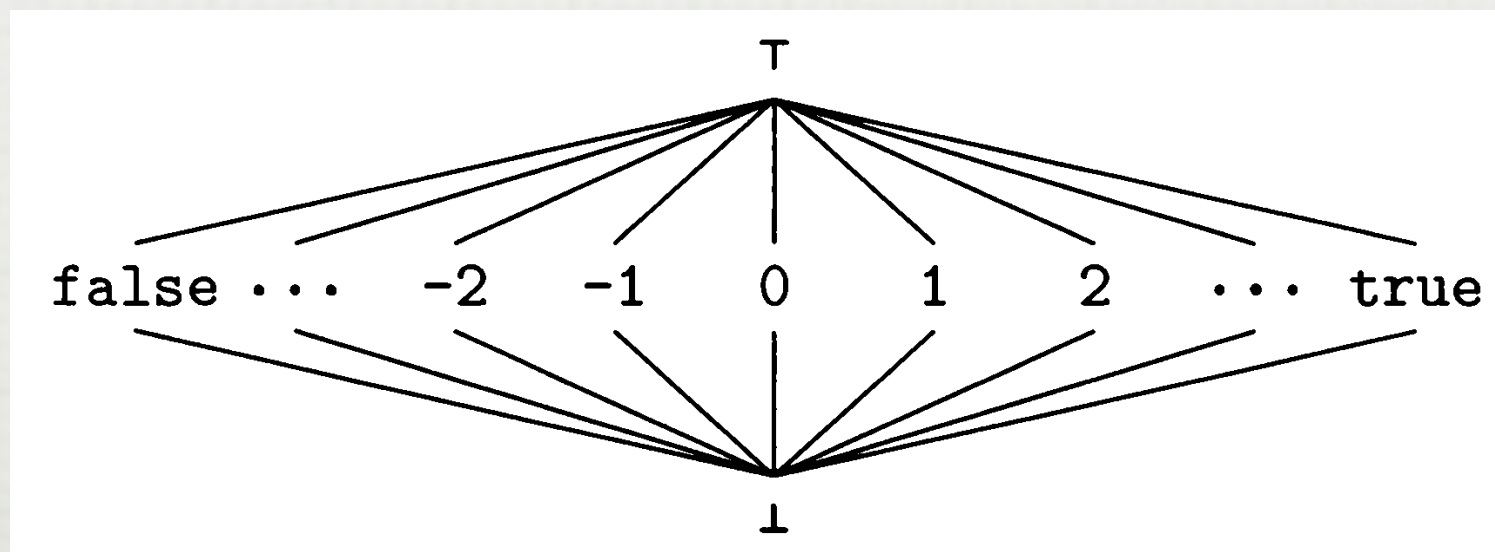
For each program point, whether or not a variable has a constant value whenever execution reaches that point.

An Infinite Lattice

The aim of the *Constant Propagation Analysis* is to determine

For each program point, whether or not a variable has a constant value whenever execution reaches that point.

Integer Constant Propagation Lattice

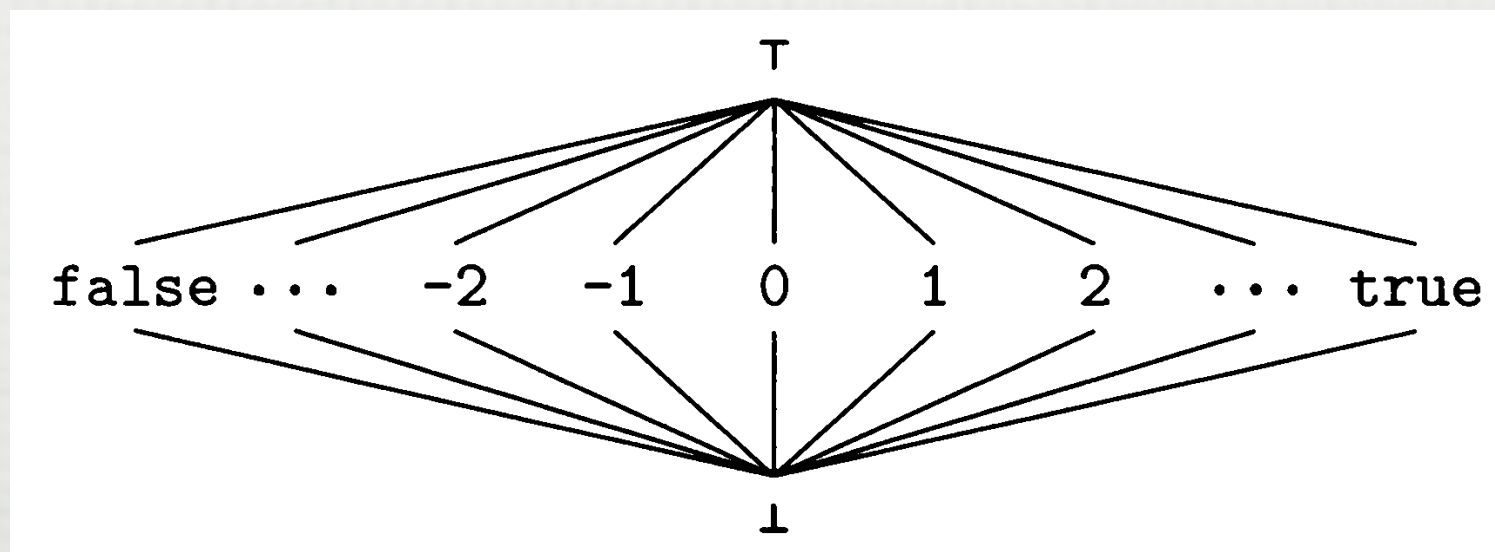


An Infinite Lattice

The aim of the *Constant Propagation Analysis* is to determine

For each program point, whether or not a variable has a constant value whenever execution reaches that point.

Integer Constant Propagation Lattice



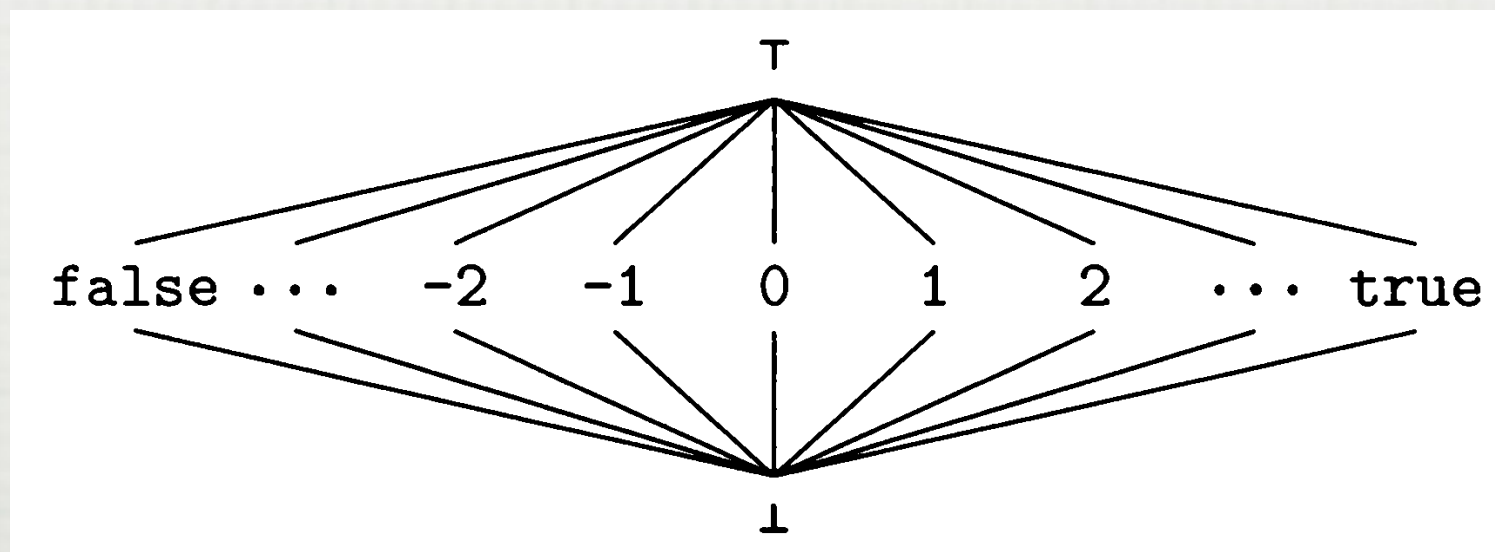
T: non-constant

An Infinite Lattice

The aim of the *Constant Propagation Analysis* is to determine

For each program point, whether or not a variable has a constant value whenever execution reaches that point.

Integer Constant Propagation Lattice



T: non-constant

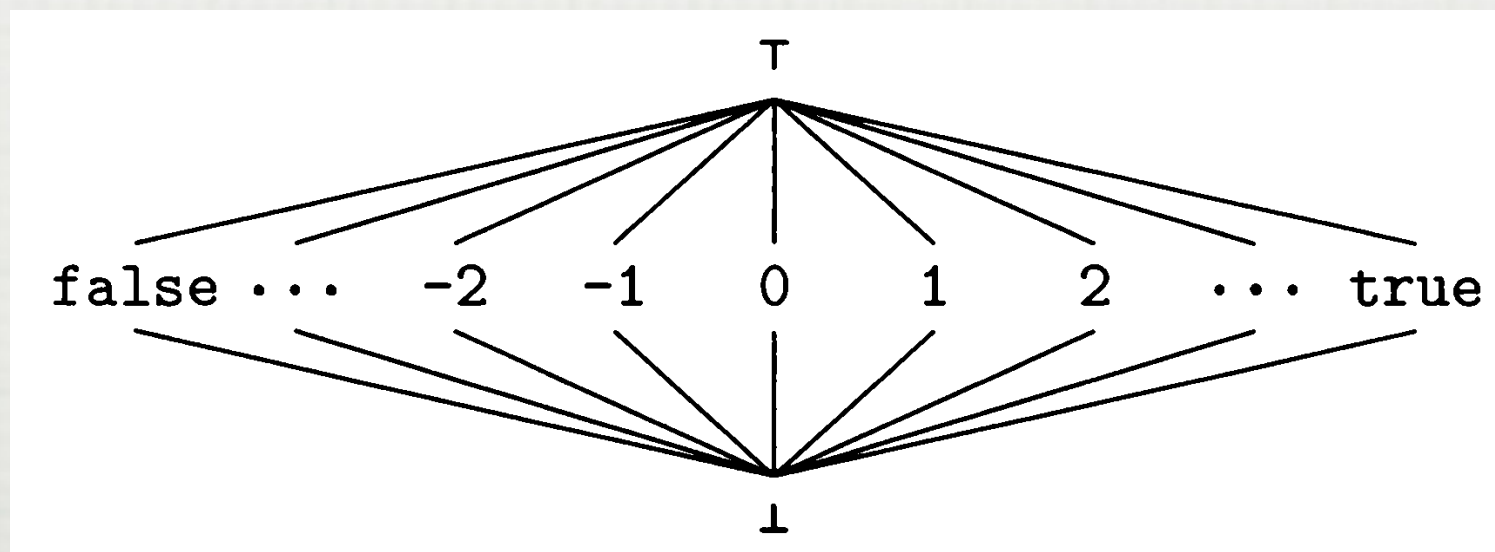
⊥: no-information

An Infinite Lattice

The aim of the *Constant Propagation Analysis* is to determine

For each program point, whether or not a variable has a constant value whenever execution reaches that point.

Integer Constant Propagation Lattice



$\top$: non-constant

$\perp$: no-information

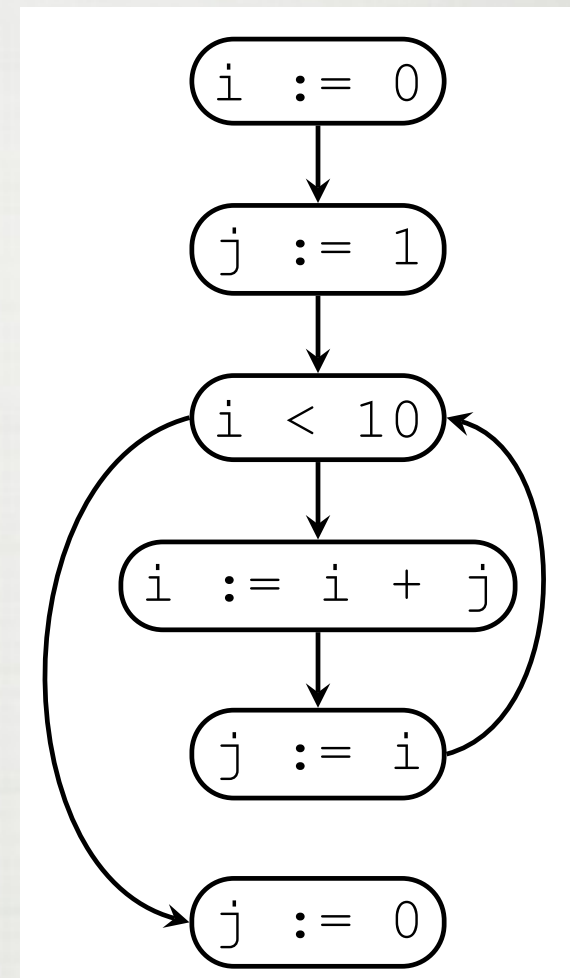
$$v_1 \sqcup v_2 = \begin{cases} v_1 & \text{if } v_1 = v_2 \\ \top & \text{if } v_1 \neq v_2 \end{cases}$$

A General Framework for
Dataflow Analyses
based on
Basic Lattice Theory

Component 1:
Domains are (semi)-lattices
of finite height!

Example: Live Variable Analysis

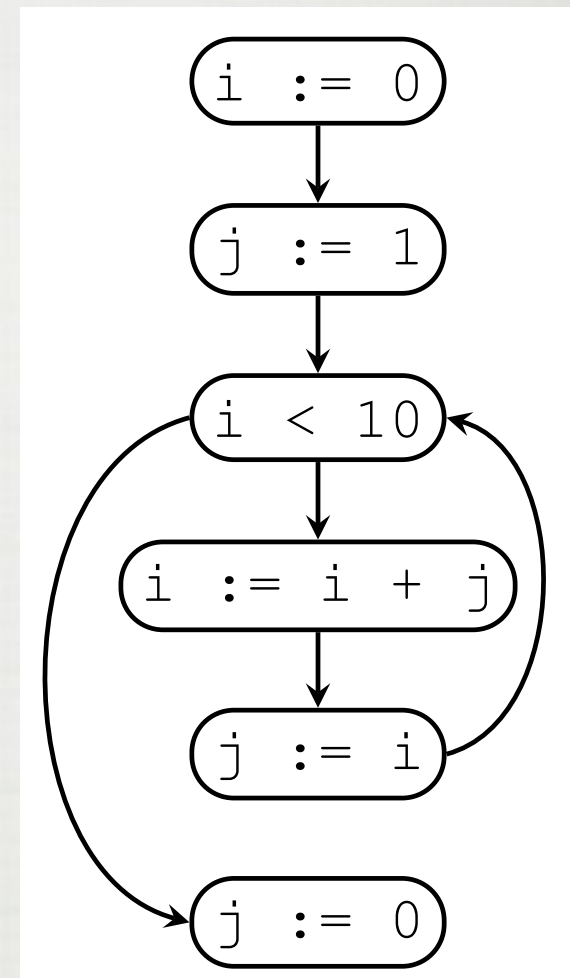
The domain is a (complete) **subset lattice** $(\mathcal{P}(S), \cup, \emptyset, S)$ where S is the set of all program variables.



Example: Live Variable Analysis

The domain is a (complete) **subset lattice** $(\mathcal{P}(S), \cup, \emptyset, S)$ where S is the set of all program variables.

Why $\cup$? Because a variable is live if it is live on **some** path from the node.

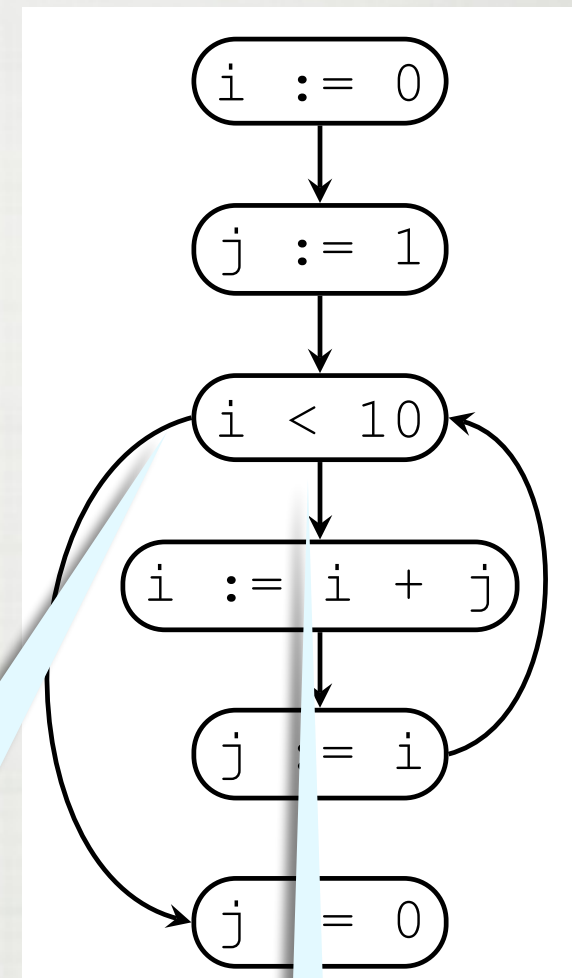


Example: Live Variable Analysis

The domain is a (complete) **subset lattice** $(\mathcal{P}(S), \cup, \emptyset, S)$ where S is the set of all program variables.

Why $\cup$? Because a variable is live if it is live on **some** path from the node.

i **live** on **all** paths this way?



i **live** on **all** paths this way?

Example: Live Variable Analysis

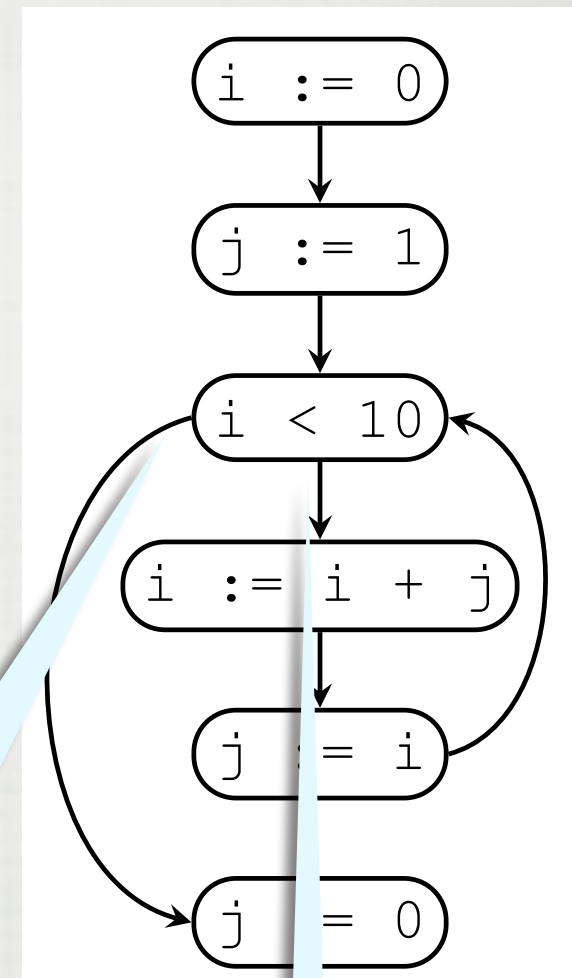
The domain is a (complete) **subset lattice** $(\mathcal{P}(S), \cup, \emptyset, S)$ where S is the set of all program variables.

Why $\cup$? Because a variable is live if it is live on **some** path from the node.

i **live** on **all** paths this way?

By choosing to call variables **live** at **exit**, we also decide that this is a **backward** dataflow problem.

i **live** on **all** paths this way?



Example: Live Variable Analysis

To fully define the **domain**:

Example: Live Variable Analysis

To fully define the **domain**:

- Define the (semi) lattice: dataflow facts and how to combine them!

Example: Live Variable Analysis

To fully define the **domain**:

- Define the (semi) lattice: dataflow facts and how to combine them!
- Decide on the direction of the analysis: forward or backward!

Component 2: Transfer Functions

Transfer Functions

A *transfer function* models, for a particular data flow analysis problem, the effect of the programming language constructs as a mapping from the lattice (used in the analysis) to itself).

$$\forall st \in \text{Statements}, f_{st} : \mathbf{L} \mapsto \mathbf{L}$$

Transfer Functions

A *transfer function* models, for a particular data flow analysis problem, the effect of the programming language constructs as a mapping from the lattice (used in the analysis) to itself).

$$\forall st \in \text{Statements}, f_{st} : \mathbf{L} \mapsto \mathbf{L}$$

Example:

$$\begin{aligned} LV_{entry}(\ell) &= LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell) \\ LV_{exit}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell') \end{aligned}$$

Transfer Functions

A *transfer function* models, for a particular data flow analysis problem, the effect of the programming language constructs as a mapping from the lattice (used in the analysis) to itself).

$$\forall st \in \text{Statements}, f_{st} : \mathbf{L} \mapsto \mathbf{L}$$

Example:

backward: would reverse for forward!

$$\begin{aligned} LV_{entry}(\ell) &= LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell) \\ LV_{exit}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell') \end{aligned}$$

Properties of Transfer Functions

Monotonicity: $\forall x, y \in \mathbf{L} : x \sqsubseteq y \implies f(x) \sqsubseteq f(y).$

Properties of Transfer Functions

Monotonicity: $\forall x, y \in \mathbf{L} : x \sqsubseteq y \implies f(x) \sqsubseteq f(y).$

Distributivity: $\forall x, y \in \mathbf{L} : f(x \sqcap y) = f(x) \sqcap f(y).$

Properties of Transfer Functions

Monotonicity: $\forall x, y \in \mathbf{L} : x \sqsubseteq y \implies f(x) \sqsubseteq f(y).$

Distributivity: $\forall x, y \in \mathbf{L} : f(x \sqcap y) = f(x) \sqcap f(y).$

Example:

$$\begin{aligned} LV_{entry}(\ell) &= LV_{exit}(\ell) \setminus write(\ell) \cup read(\ell) \\ LV_{exit}(\ell) &= \bigcup_{\ell \rightarrow \ell' \in E} LV_{entry}(\ell') \end{aligned}$$

Properties of Transfer Functions

Monotonicity: $\forall x, y \in \mathbf{L} : x \sqsubseteq y \implies f(x) \sqsubseteq f(y).$

Transfer functions in a dataflow analysis must be monotone!

Properties of Transfer Functions

Monotonicity: $\forall x, y \in \mathbf{L} : x \sqsubseteq y \implies f(x) \sqsubseteq f(y).$

Transfer functions in a dataflow analysis must be monotone!

Distributivity: $\forall x, y \in \mathbf{L} : f(x \sqcap y) = f(x) \sqcap f(y).$

Properties of Transfer Functions

Monotonicity: $\forall x, y \in \mathbf{L} : x \sqsubseteq y \implies f(x) \sqsubseteq f(y).$

Transfer functions in a dataflow analysis must be monotone!

Distributivity: $\forall x, y \in \mathbf{L} : f(x \sqcap y) = f(x) \sqcap f(y).$

But not necessarily distributive!

Properties of Transfer Functions

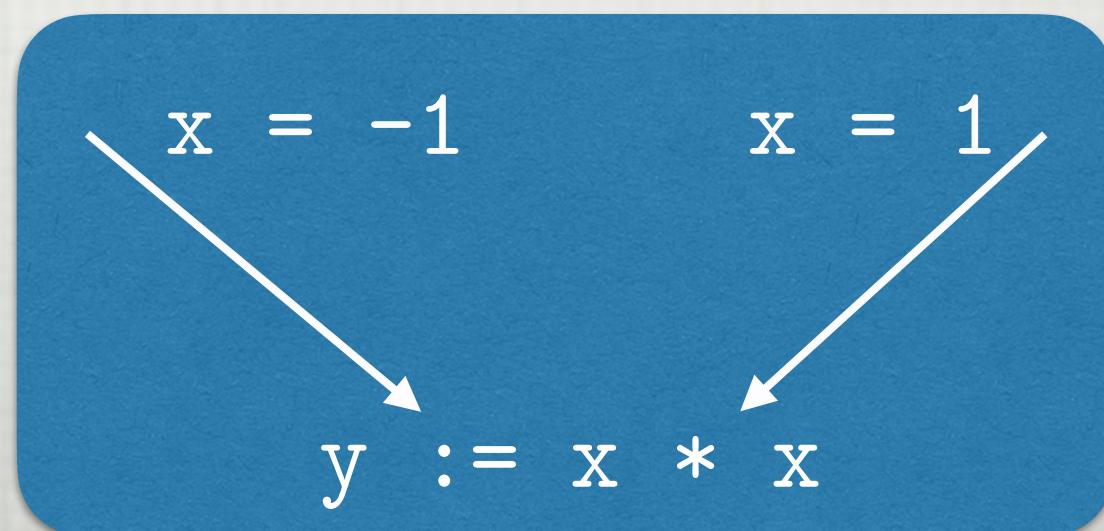
Monotonicity: $\forall x, y \in \mathbf{L} : x \sqsubseteq y \implies f(x) \sqsubseteq f(y).$

Transfer functions in a dataflow analysis must be monotone!

Distributivity: $\forall x, y \in \mathbf{L} : f(x \sqcap y) = f(x) \sqcap f(y).$

But not necessarily distributive!

Constant Propagation:



Component 3: The Computation

The Goal

What is the *goal* of dataflow analyses?

The Goal

What is the *goal* of dataflow analyses?

meet-over-all-paths (MOP) solutions.

The Goal

What is the *goal* of dataflow analyses?

meet-over-all-paths (MOP) solutions.

- Start from the beginning (entry node, or exist note for backward flow problems) with some *initial information*.

The Goal

What is the *goal* of dataflow analyses?

meet-over-all-paths (MOP) solutions.

- Start from the beginning (entry node, or exist note for backward flow problems) with some *initial information*.
- Walk down a path and *apply transfer functions along these paths* to each node in the flow graph.

The Goal

What is the *goal* of dataflow analyses?

meet-over-all-paths (MOP) solutions.

- Start from the beginning (entry node, or exist note for backward flow problems) with some *initial information*.
- Walk down a path and *apply transfer functions along these paths* to each node in the flow graph.
- For each node, compute the *meet* of all paths to this point.

Formally

For a path $\pi = init \dots l$

$$f_{\pi} = f_{init} \circ \dots \circ f_l$$

$$MOP_{\circ}(l) = \bigcap_{\pi \in Path(l)} f_{\pi}(\iota).$$

$$MOP_{\bullet}(l) = f_l(MOP_{\circ}(l)).$$

Formally

For a path $\pi = init \dots l$

Transfer Function for location l

$$f_{\pi} = f_{init} \circ \dots \circ f_l$$

Set of all paths to l

Initial information at “init”

$$MOP_{\circ}(l) = \bigcap_{\pi \in Path(l)} f_{\pi}(I).$$

$$MOP_{\bullet}(l) = f_l(MOP_{\circ}(l)).$$

Can this solution be
computed effectively?

Bad News!

For an arbitrary data flow analysis problem where **transfer functions** are only monotone, one can show that there may be **no algorithm** to compute the *MOP* solution.

Bad News!

For an arbitrary data flow analysis problem where **transfer functions** are only monotone, one can show that there may be **no algorithm** to compute the *MOP* solution.

Lemma

The MOP solution for Constant Propagation is undecidable.

Proof: Let $u_1, \dots, u_n$ and $v_1, \dots, v_n$ be strings over the alphabet $\{1, \dots, 9\}$; let $|u|$ denote the length of u ; let $\llbracket u \rrbracket$ be the natural number denoted.

The Modified Post Correspondence Problem is to determine whether or not $u_{i_1} \dots u_{i_m} = v_{i_1} \dots v_{i_m}$ for some sequence $i_1, \dots, i_m$ with $i_1 = 1$.

```
x :=  $\llbracket u_1 \rrbracket$ ; y :=  $\llbracket v_1 \rrbracket$ ;
while [...] do
  (if [...] then x := x *  $10^{|u_1|}$  +  $\llbracket u_1 \rrbracket$ ; y := y *  $10^{|v_1|}$  +  $\llbracket v_1 \rrbracket$  else
  :
  if [...] then x := x *  $10^{|u_n|}$  +  $\llbracket u_n \rrbracket$ ; y := y *  $10^{|v_n|}$  +  $\llbracket v_n \rrbracket$  else skip)
[z := abs((x-y)*(x-y))]  $^\ell$ 
```

Then **MOP** $_{\bullet}(\ell)$ will map z to 1 if and only if the Modified Post Correspondence Problem has no solution. This is undecidable.

So, what do we do?

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Good News

Instead, compute the *maximal fixed point solution* (MFP).

in the *meet lattice*: the dual for the *join lattice* is the least fixed point!

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Consider the **set of constraints** below:

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Consider the **set of constraints** below:

$$MFP_{\circ}(l) = \begin{cases} \iota & l = init \\ \bigcap_{(l',l) \in flow} MFP_{\bullet}(l') & \text{otherwise} \end{cases}$$

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Consider the **set of constraints** below:

$$MFP_{\circ}(l) = \begin{cases} \iota & l = init \\ \bigcap_{(l',l) \in flow} MFP_{\bullet}(l') & \text{otherwise} \end{cases}$$

$$MFP_{\bullet}(l) = f_l(MFP_{\circ}(l))$$

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Consider the **set of constraints** below:

$$MFP_{\circ}(l) = \begin{cases} \iota & l = init \\ \bigcap_{(l',l) \in flow} MFP_{\bullet}(l') & \text{otherwise} \end{cases}$$

$$MFP_{\bullet}(l) = f_l(MFP_{\circ}(l))$$

forward

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Consider the **set of constraints** below:

$$MFP_{\circ}(l) = \begin{cases} \iota & l = init \\ \bigcap_{(l',l) \in flow} MFP_{\bullet}(l') & \text{otherwise} \end{cases}$$

entry

$$MFP_{\bullet}(l) = f_l(MFP_{\circ}(l))$$

forward

exit

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Consider the **set of constraints** below:

$$MFP_{\circ}(l) = \begin{cases} \iota & l = init \\ \bigcap_{(l',l) \in flow} MFP_{\bullet}(l') & \text{otherwise} \end{cases}$$

exit

$$MFP_{\bullet}(l) = f_l(MFP_{\circ}(l))$$

entry

backward

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Consider the **set of constraints** below:

$$MFP_{\circ}(l) = \begin{cases} \iota & l = \textit{init} \\ \bigcap_{(l',l) \in \textit{flow}} MFP_{\bullet}(l') & \text{otherwise} \end{cases}$$

$$MFP_{\bullet}(l) = f_l(MFP_{\circ}(l))$$

What is a **solution** to these set of constraints?

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Consider the **set of constraints** below:

$$MFP_{\circ}(l) = \begin{cases} \iota & l = \textit{init} \\ \bigcap_{(l',l) \in \textit{flow}} MFP_{\bullet}(l') & \text{otherwise} \end{cases}$$

$$MFP_{\bullet}(l) = f_l(MFP_{\circ}(l))$$

What is a **solution** to these set of constraints?

A **solution** is a fixed point!

Good News

Instead, compute the *maximal fixed point solution* (MFP).

Consider the **set of constraints** below:

$$MFP_{\circ}(l) = \begin{cases} \iota & l = \textit{init} \\ \bigcap_{(l',l) \in \textit{flow}} MFP_{\bullet}(l') & \text{otherwise} \end{cases}$$

$$MFP_{\bullet}(l) = f_l(MFP_{\circ}(l))$$

What is a **solution** to these set of constraints?

A **solution** is a fixed point!

Is it **unique**?

Algebra brings it all
together!

Fixed Point Solutions

Theorem: (Knaster-Tarski Fixpoint Theorem)

Fixed Point Solutions

Theorem: (Knaster-Tarski Fixpoint Theorem)

Let $\mathbf{L}$ be a **complete lattice** and $F : \mathbf{L} \rightarrow \mathbf{L}$ be a **monotone function**. Then, the set of fixpoints of F in $\mathbf{L}$ is also a complete lattice.

Fixed Point Solutions

Theorem: (Knaster-Tarski Fixpoint Theorem)

Let $\mathbf{L}$ be a **complete lattice** and $F : \mathbf{L} \rightarrow \mathbf{L}$ be a **monotone function**. Then, the set of fixpoints of F in $\mathbf{L}$ is also a complete lattice.

Corollary: We have a set of solutions (fixed points), with a guarantee for the existence of a **maximal** (also minimal) solution.

How do we compute it?

Fixed Point Solutions

Theorem: (Kleene Fixpoint Theorem)

Fixed Point Solutions

Theorem: (Kleene Fixpoint Theorem)

Let $\mathbf{L}$ be a **complete lattice** and $F : \mathbf{L} \rightarrow \mathbf{L}$ be a **monotone function**. The maximal fixpoint of $\mathbf{L}$ is the infimum of the descending chain $\top \sqsupseteq F(\top) \sqsupseteq F(F(\top)) \sqsupseteq \dots$.

Fixed Point Solutions

Theorem: (Kleene Fixpoint Theorem)

Let $\mathbf{L}$ be a **complete lattice** and $F : \mathbf{L} \rightarrow \mathbf{L}$ be a **monotone function**. The maximal fixpoint of $\mathbf{L}$ is the infimum of the descending chain $\top \sqsupseteq F(\top) \sqsupseteq F(F(\top)) \sqsupseteq \dots$.

We can start with the solution $\top$ and continually apply F to its own result until we eventually reach a fixed point which will be maximal.

Fixed Point Solutions

Theorem: (Kleene Fixpoint Theorem)

Let $\mathbf{L}$ be a **complete lattice** and $F : \mathbf{L} \rightarrow \mathbf{L}$ be a **monotone function**. The maximal fixpoint of $\mathbf{L}$ is the infimum of the descending chain $\top \sqsupseteq F(\top) \sqsupseteq F(F(\top)) \sqsupseteq \dots$.

We can start with the solution $\top$ and continually apply F to its own result until we eventually reach a fixed point which will be maximal.

DFA Algorithm

```
 $\forall k \in N. \text{IN}_k = \text{OUT}_k = \top$   
repeat  
  foreach  $k \in N$  do {  
     $\text{IN}_k = \sqcap \{ \text{OUT}_p \mid p \in \text{pred}(k) \}$   
     $\text{OUT}_k = F_k(\text{IN}_k)$   
  }  
while solution changes
```

The Coincidence

If transfer functions are **monotone**:

$$MOP_{\circ}(l) \sqsupseteq MFP_{\circ}(l) \qquad MOP_{\bullet}(l) \sqsupseteq MFP_{\bullet}(l)$$

Less Precise!

The Coincidence

If transfer functions are **monotone**:

$$MOP_{\circ}(l) \sqsupseteq MFP_{\circ}(l) \qquad MOP_{\bullet}(l) \sqsupseteq MFP_{\bullet}(l)$$

The Coincidence

If transfer functions are **monotone**:

$$MOP_{\circ}(l) \supseteq MFP_{\circ}(l) \qquad MOP_{\bullet}(l) \supseteq MFP_{\bullet}(l)$$

The fixpoint solution **over-approximates** the result!

The Coincidence

If transfer functions are **monotone**:

$$MOP_{\circ}(l) \supseteq MFP_{\circ}(l) \qquad MOP_{\bullet}(l) \supseteq MFP_{\bullet}(l)$$

The fixpoint solution **over-approximates** the result!

If transfer functions are **distributive**:

$$MOP_{\circ}(l) = MFP_{\circ}(l) \qquad MOP_{\bullet}(l) = MFP_{\bullet}(l)$$

The Coincidence

If transfer functions are **monotone**:

$$MOP_{\circ}(l) \supseteq MFP_{\circ}(l) \qquad MOP_{\bullet}(l) \supseteq MFP_{\bullet}(l)$$

Less Precise!

The fixpoint solution **over-approximates** the result!

If transfer functions are **distributive**:

$$MOP_{\circ}(l) = MFP_{\circ}(l) \qquad MOP_{\bullet}(l) = MFP_{\bullet}(l)$$

The two solutions **coincide**!

Let's make another
instance of our
framework!

Very Busy Expressions

if $[a > b]^1$ then $([x := b - a]^2; [y := a - b]^3)$ else $([y := b - a]^4; [x := a - b]^5)$

Very Busy Expressions

if $[a > b]^1$ then $([x := b - a]^2; [y := a - b]^3)$ else $([y := b - a]^4; [x := a - b]^5)$



$[t1 := b - a]^A; [t2 := a - b]^B;$
if $[a > b]^1$ then $([x := t1]^2; [y := t2]^3)$ else $([y := t1]^4; [x := t2]^5)$

Very Busy Expressions

if $[a > b]^1$ then $([x := b - a]^2; [y := a - b]^3)$ else $([y := b - a]^4; [x := a - b]^5)$



$[t1 := b - a]^A; [t2 := a - b]^B;$
if $[a > b]^1$ then $([x := t1]^2; [y := t2]^3)$ else $([y := t1]^4; [x := t2]^5)$

An expression is **very busy** at the **exit** from a label if, no matter what path is taken from the label, the expression is always used before any of the variables occurring in it are redefined.

Check List

Check List

- Define the semi-lattice: dataflow facts and how to combine them!

Check List

- Define the semi-lattice: dataflow facts and how to combine them!
- Decide on the direction of the analysis: forward vs backward.

Check List

- Define the semi-lattice: dataflow facts and how to combine them!
 - Decide on the direction of the analysis: forward vs backward.
 - Sanity check: the corresponding order should make sense!

Check List

- Define the semi-lattice: dataflow facts and how to combine them!
 - Decide on the direction of the analysis: forward vs backward.
 - Sanity check: the corresponding order should make sense!
 - Decide on the initial values.

Check List

- Define the semi-lattice: dataflow facts and how to combine them!
 - Decide on the direction of the analysis: forward vs backward.
 - Sanity check: the corresponding order should make sense!
 - Decide on the initial values.
- Design the transfer functions:

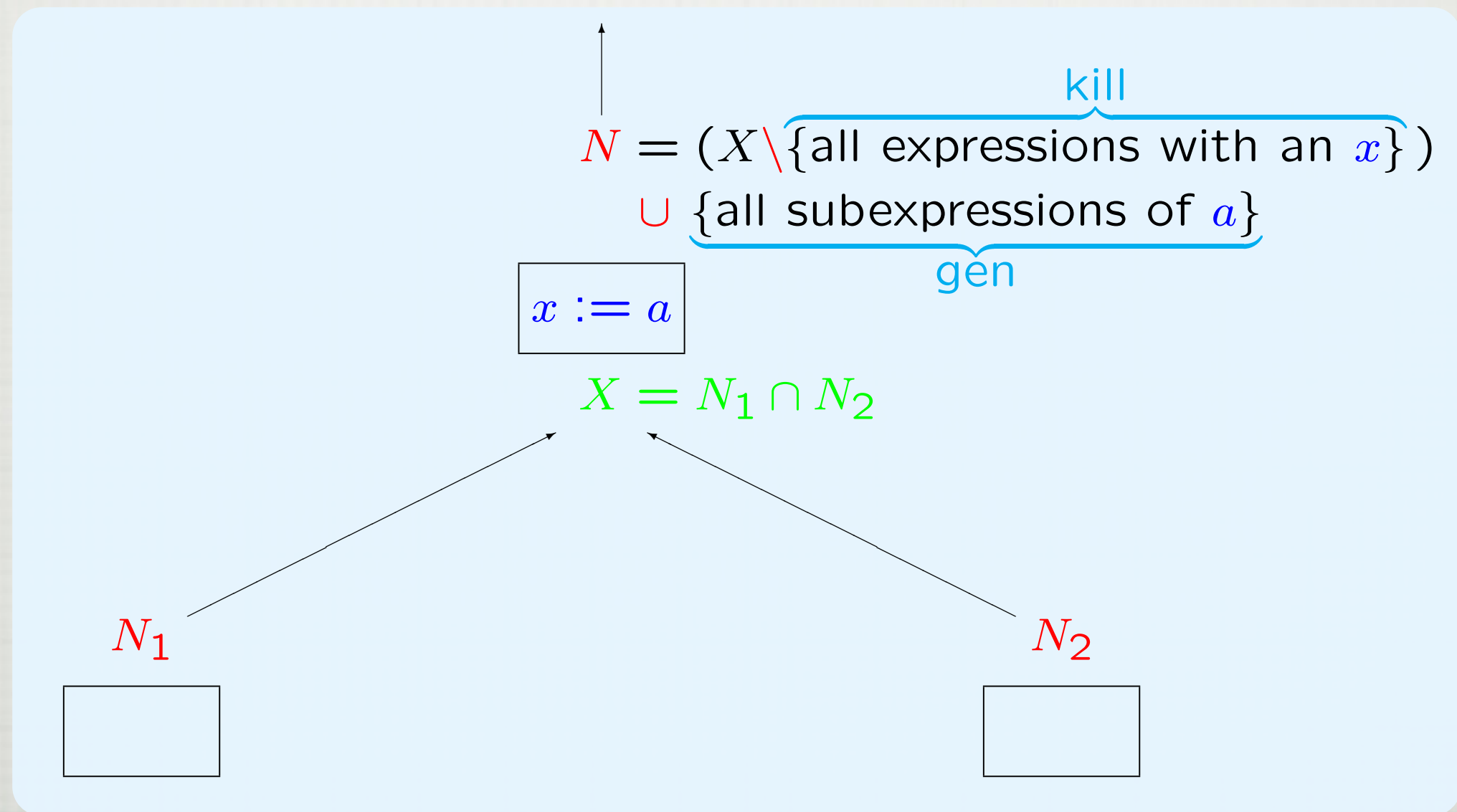
Check List

- Define the semi-lattice: dataflow facts and how to combine them!
 - Decide on the direction of the analysis: forward vs backward.
 - Sanity check: the corresponding order should make sense!
 - Decide on the initial values.
- Design the transfer functions:
 - How does each statement affect the dataflow facts?

Check List

- Define the semi-lattice: dataflow facts and how to combine them!
 - Decide on the direction of the analysis: forward vs backward.
 - Sanity check: the corresponding order should make sense!
 - Decide on the initial values.
- Design the transfer functions:
 - How does each statement affect the dataflow facts?
 - Sanity check: Monotonicity!

The Design Process



Very Busy Expressions: Formal Setup

Dataflow Facts: $D = \mathcal{P}(Exp)$

Very Busy Expressions: Formal Setup

Dataflow Facts: $D = \mathcal{P}(Exp)$

Domain: complete meet semi-lattice $(D, \cap, D)$

Very Busy Expressions: Formal Setup

Dataflow Facts: $D = \mathcal{P}(Exp)$

Domain: complete meet semi-lattice $(D, \cap, D)$

Direction: Backward

Very Busy Expressions: Formal Setup

Dataflow Facts: $D = \mathcal{P}(Exp)$

Domain: complete meet semi-lattice $(D, \cap, D)$

Direction: **Backward**

Transfer Functions:

Very Busy Expressions: Formal Setup

Dataflow Facts: $D = \mathcal{P}(Exp)$

Domain: complete meet semi-lattice $(D, \cap, D)$

Direction: **Backward**

Transfer Functions:

$$VB_{\bullet}(l) = \begin{cases} \emptyset & l = exit \\ \bigcap_{(l,l') \in flow} VB_{\bullet}(l') & \text{otherwise} \end{cases}$$

Very Busy Expressions: Formal Setup

Dataflow Facts: $D = \mathcal{P}(\text{Exp})$

Domain: complete meet semi-lattice $(D, \cap, D)$

Direction: **Backward**

Transfer Functions:

$$VB_{\bullet}(l) = \begin{cases} \emptyset & l = \text{exit} \\ \bigcap_{(l,l') \in \text{flow}} VB_{\bullet}(l') & \text{otherwise} \end{cases}$$

$$VB_{\circ}(l) = RD_{\bullet}(l) \setminus \{ \text{exp} \mid \text{var}(\text{exp}) \cap \text{write}(l) \neq \emptyset \} \\ \cup \text{computed}(l)$$

Very Busy Expressions: Formal Setup

Dataflow Facts: $D = \mathcal{P}(Exp)$

Domain: complete meet semi-lattice $(D, \cap, D)$

unlike **live variables**: here we want the greatest fixed point!

Direction: **Backward**

Transfer Functions:

$$VB_{\bullet}(l) = \begin{cases} \emptyset & l = exit \\ \bigcap_{(l,l') \in flow} VB_{\bullet}(l') & \text{otherwise} \end{cases}$$

$$VB_{\circ}(l) = RD_{\bullet}(l) \setminus \{exp \mid var(exp) \cap write(l) \neq \emptyset\} \\ \cup computed(l)$$

Reaching Definitions

	Live Variables	Reaching Definitions	Very Busy Expressions
L	$\mathcal{P}(Var)$	$\mathcal{P}(Var \times Loc)$	$\mathcal{P}(Exp)$
$\sqcap$	$\cup$	$\cup$	$\cap$
$\sqsubseteq$	$\supseteq$	$\supseteq$	$\subseteq$
$\perp$	$\emptyset$	$\emptyset$	Exp
	Backward	Forward	Backward

Reaching Definitions

	←	$\{(x, ?), (y, ?), (z, ?)\}$
$[y := x]^1;$	←	$\{(x, ?), (y, 1), (z, ?)\}$
$[z := 1]^2;$	←	$\{(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)\}$
while $[y > 0]^3$ do	←	$\{(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)\}$
$[z := z * y]^4;$	←	$\{(x, ?), (y, 1), (y, 5), (z, 4)\}$
$[y := y - 1]^5$	←	$\{(x, ?), (y, 5), (z, 4)\}$
od;	←	$\{(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)\}$
$[y := 0]^6$	←	$\{(x, ?), (y, 6), (z, 2), (z, 4)\}$

Reaching Definitions: Formal Setup

Reaching Definitions: Formal Setup

Dataflow Facts: $D = Vars \times Locs$

Reaching Definitions: Formal Setup

Dataflow Facts: $D = Vars \times Locs$

Domain: $(D, \cup, \emptyset, D)$

Reaching Definitions: Formal Setup

Dataflow Facts: $D = Vars \times Locs$

Domain: $(D, \cup, \emptyset, D)$

Direction: **Forward**

Reaching Definitions: Formal Setup

Dataflow Facts: $D = Vars \times Locs$

Domain: $(D, \cup, \emptyset, D)$

Direction: **Forward**

Transfer Functions:

Reaching Definitions: Formal Setup

Dataflow Facts: $D = Vars \times Locs$

Domain: $(D, \cup, \emptyset, D)$

Direction: **Forward**

Transfer Functions:

$$RD_{\circ}(l) = \begin{cases} Var \times \{?\} & l = init \\ \bigcup_{(l', l) \in flow} RD_{\bullet}(l') & \text{otherwise} \end{cases}$$

Reaching Definitions: Formal Setup

Dataflow Facts: $D = Vars \times Locs$

Domain: $(D, \cup, \emptyset, D)$

Direction: **Forward**

Transfer Functions:

$$RD_{\circ}(l) = \begin{cases} Var \times \{?\} & l = init \\ \bigcup_{(l', l) \in flow} RD_{\bullet}(l') & \text{otherwise} \end{cases}$$

$$RD_{\bullet}(l) = RD_{\circ}(l) \setminus write(l) \times Locs \cup write(l) \times \{l\}$$