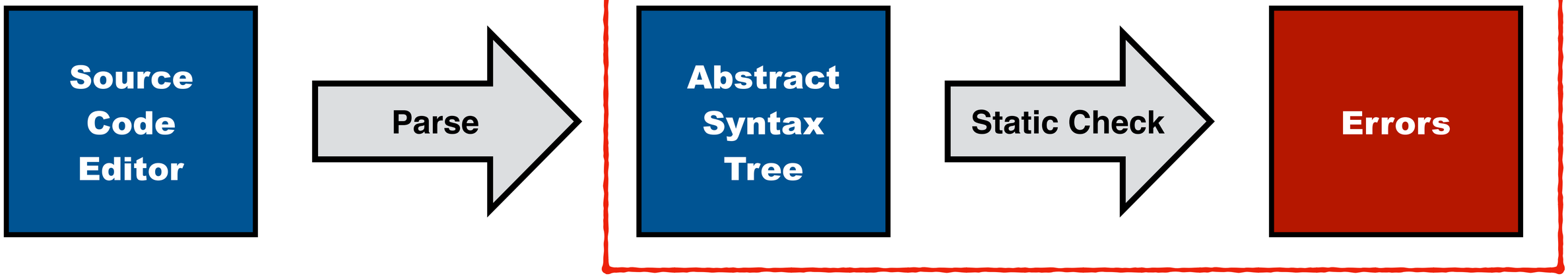
Declare Your Language

Chapter 10: Data-Flow Analysis

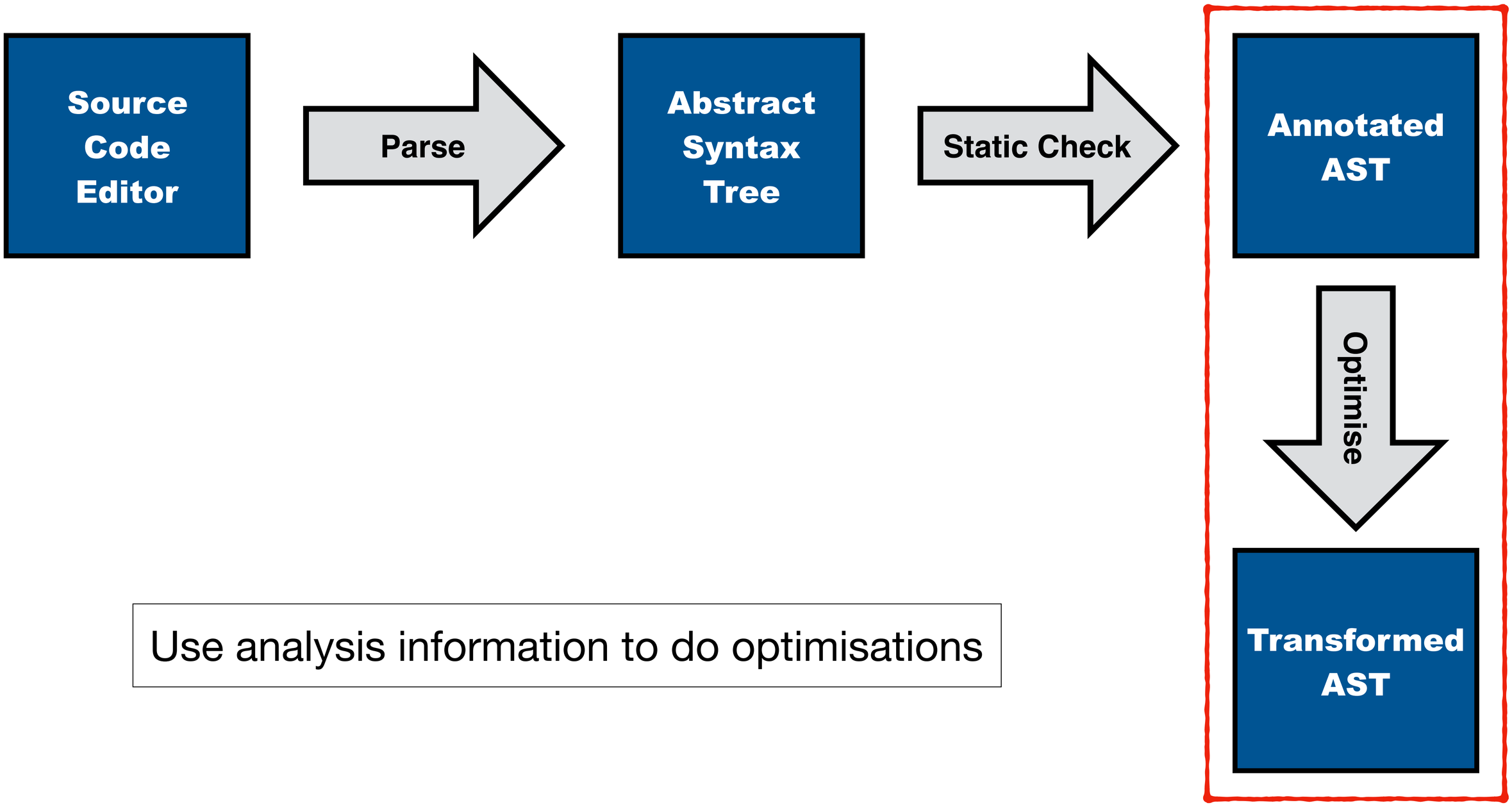
Jeff Smits

IN4303 Compiler Construction
TU Delft
November 2017

TU Delft



Check such things as final fields being initialised by a constructor



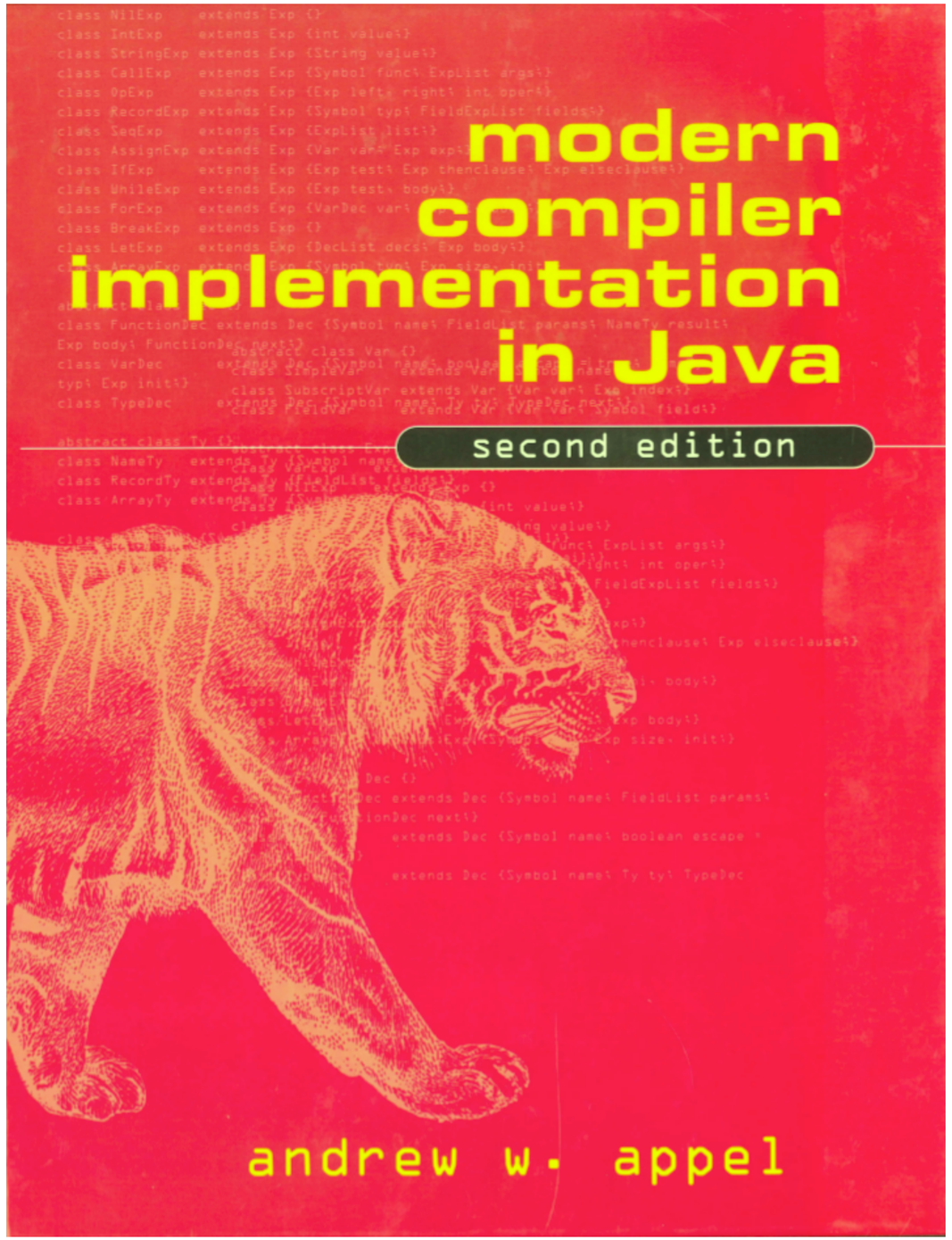
Use analysis information to do optimisations

Reading Material

Chapter 10: Liveness Analysis

Chapter 17: Dataflow Analysis

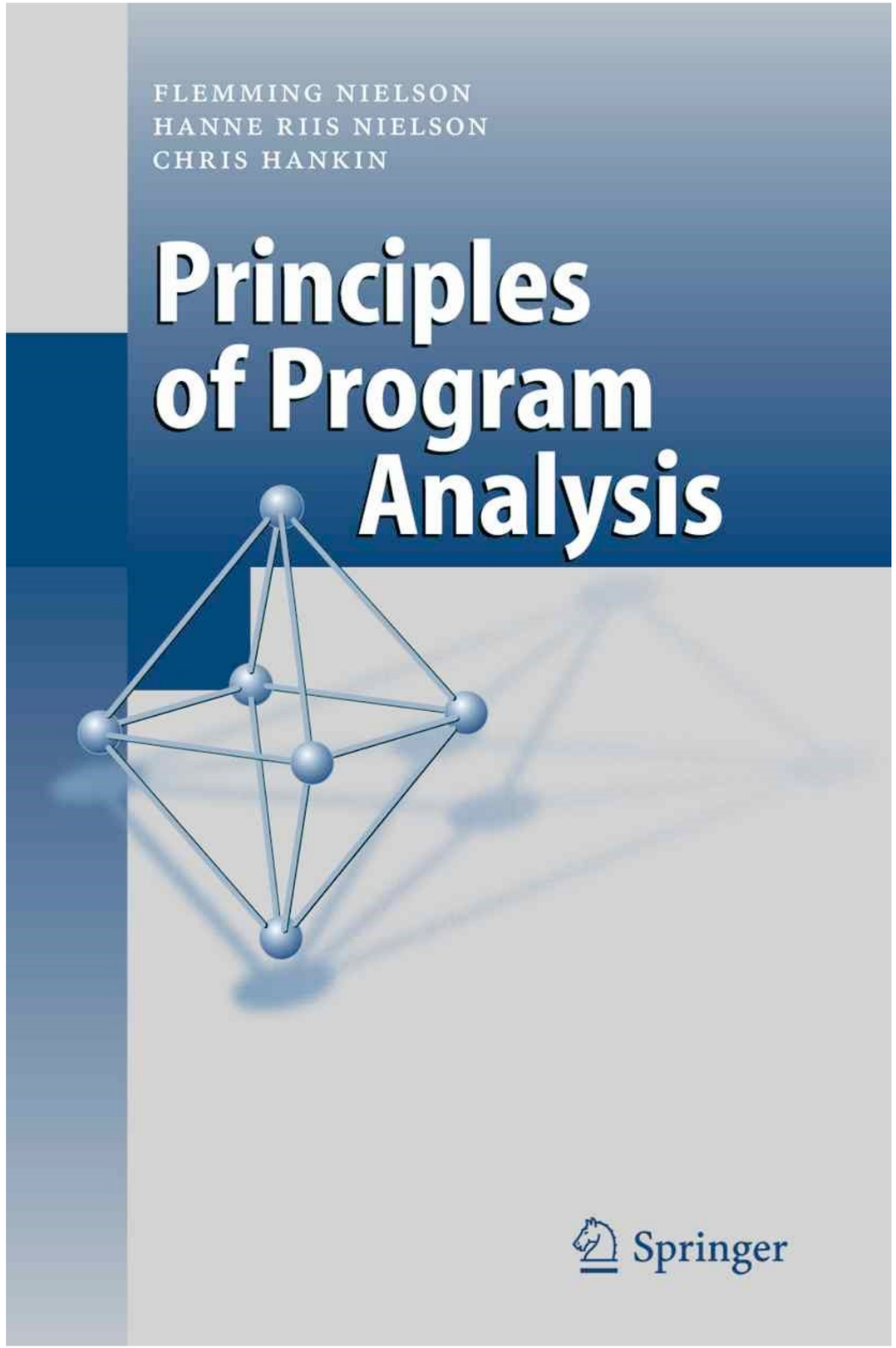
The *applied*/“cookbook” version



Chapter 1: Introduction

Chapter 2: Data Flow Analysis

The theoretical/“general” version



Data-Flow

Examples in Tiger

Available Expressions

```
let
  var x : int := a + b
  var y : int := a * b
 in
  while y > a + b do
    (
      a := a + 1;
      x := a + b
    )
end
```

Available Expressions

```
let
  var x : int := a + b
  var y : int := a * b
 in
  while y > a + b do
    (
      a := a + 1;
      x := a + b
    )
end
```

Available Expressions

```
let
  var x : int := a + b
  var y : int := a * b
 in
  while y > a + b do
    (
      a := a + 1;
      x := a + b
    )
end
```

Available Expressions

```
let
  var x : int := a + b
  var y : int := a * b
 in
  while y > a + b do
    (
      a := a + 1;
      x := a + b
    )
end
```

Available Expressions

Common subexpression elimination

```
let
  var x : int := a + b
  var y : int := a * b
 in
  while y > a + b do
    (
      a := a + 1;
      x := a + b
    )
end
```

Available Expressions

Common subexpression elimination

```
let
  var x : int := a + b
  var y : int := a * b
 in
  while y >    x    do
     (
        a := a + 1;
        x := a + b
     )
end
```

Available Expressions

```
let
  var x : int := a + b
  var y : int := a * b
 in
  while y > a + b do
    (
      a := a + 1;
      x := a + b
    )
end
```

a + b
a + b, a * b
a + b

Live Variables

```
x := 2;
y := 4;
x := 1;
if y > x then
  z := y
else
  z := y * y;
x := z
```

Live Variables

```
x := 2;
y := 4;
x := 1;
if y > x then
  z := y
else
  z := y * y;
x := z
```

Live Variables

Dead code elimination

```
y := 4;
x := 1;
if y > x then
  z := y
else
  z := y * y;
x := z
```

What is Data-Flow Analysis?

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

https://en.wikipedia.org/wiki/Data_flow_analysis

What is Data-Flow Analysis?

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

Gathering information about

https://en.wikipedia.org/wiki/Data_flow_analysis

What is Data-Flow Analysis?

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

Gathering information about

- The *possible* set of values at various point in a program

https://en.wikipedia.org/wiki/Data_flow_analysis

What is Data-Flow Analysis?

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

Gathering information about

- The *possible* set of values at various point in a program
- The program's *control flow graph* is used to …

https://en.wikipedia.org/wiki/Data_flow_analysis

Control Flow Graphs

What is a Control Flow Graph?

A **control flow graph** (**CFG**) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

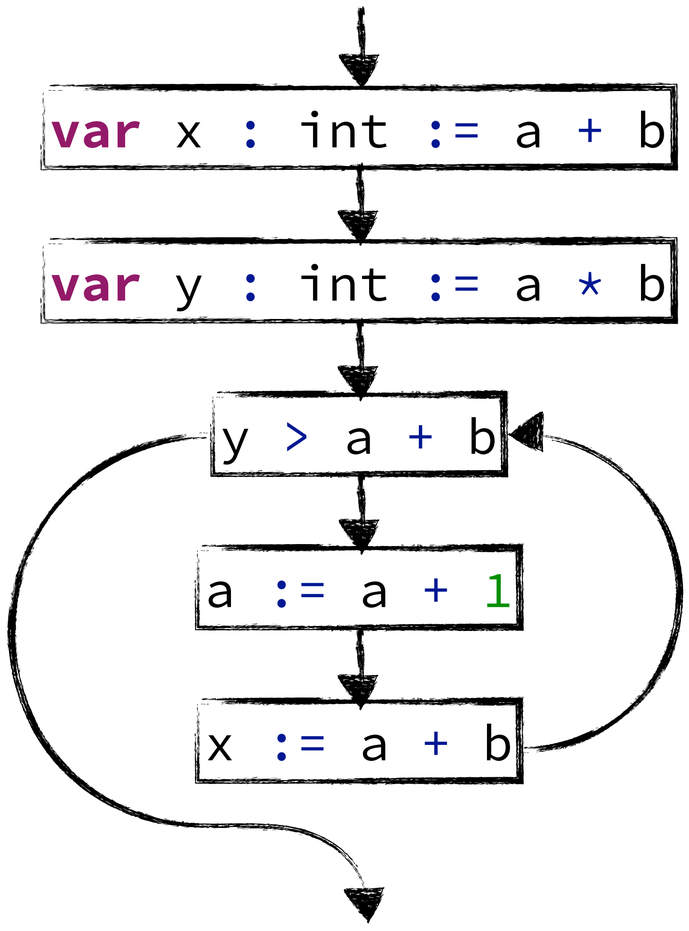
https://en.wikipedia.org/wiki/Control_flow_graph

Control Flow Graphs

```
let
  var x : int := a + b
  var y : int := a * b
 in
  while y > a + b do
    (
      a := a + 1;
      x := a + b
    )
end
```

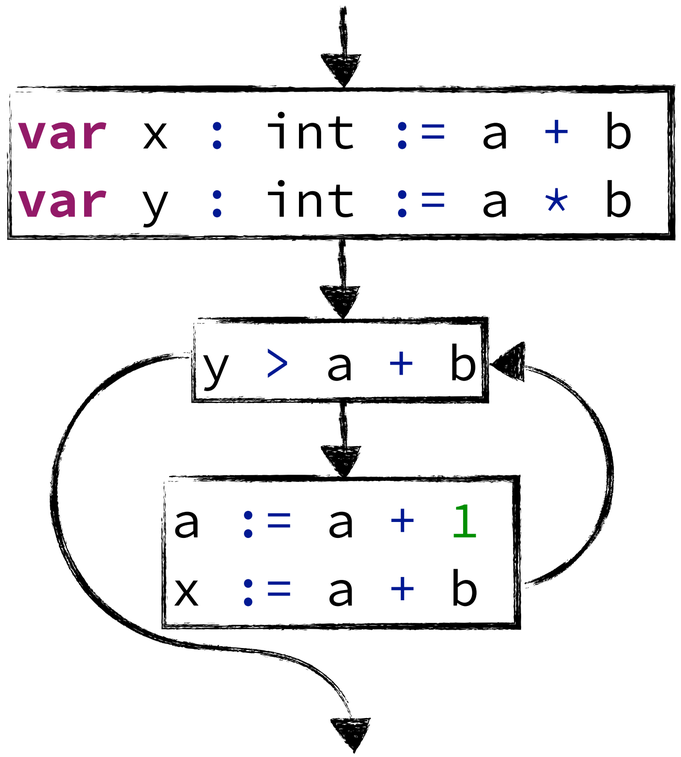
Control Flow Graphs

Full flow graph



Control Flow Graphs

Basic Blocks



Control Flow Graphs

Ingredients

Control Nodes

- Usually outermost statements and expressions
- Or blocks for consecutive statements

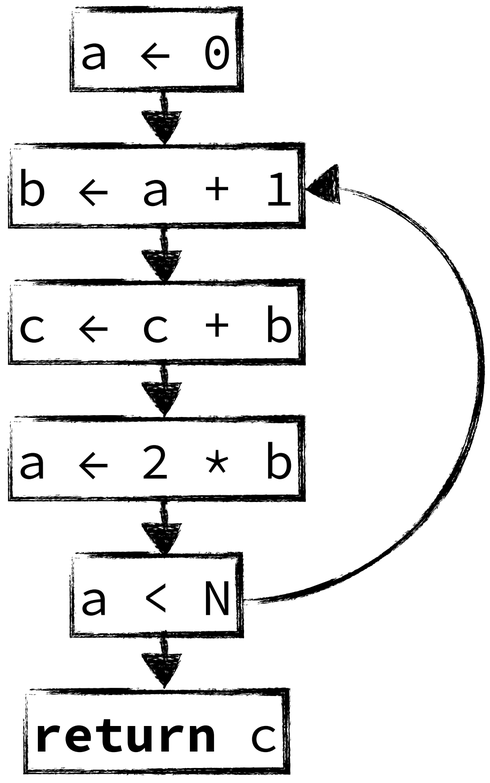
Control Edges

- Back edges: show loops
- Splits: conditionally split the control flow
- Merges: combine previously split control flow

Control Flow Graphs

Supports unstructured control flow

```
    a ← 0
L1: b ← a + 1
    c ← c + b
    a ← 2 * b
    if a < N goto L1
    return c
```



What is Data-Flow Analysis?

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

https://en.wikipedia.org/wiki/Data_flow_analysis

What is Data-Flow Analysis?

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint. This general approach was developed by Gary Kildall while teaching at the Naval Postgraduate School.[1]

https://en.wikipedia.org/wiki/Data_flow_analysis

What is Data-Flow Analysis?

Possible set of values

- at various points in a computer program

Data-flow equations

- for each node of the control flow graph

Solve them by repetition

- i.e., it reaches a fixpoint

What is Data-Flow Analysis?

What is Data-Flow Analysis?

Some information about

What is Data-Flow Analysis?

Some information about

- What may or must happen, in the previous or next part of the program

What is Data-Flow Analysis?

Some information about

- What may or must happen, in the previous or next part of the program
- Usually some approximation of values that the program computes

What is Data-Flow Analysis?

Some information about

- What may or must happen, in the previous or next part of the program
- Usually some approximation of values that the program computes

Transfer functions

What is Data-Flow Analysis?

Some information about

- What may or must happen, in the previous or next part of the program
- Usually some approximation of values that the program computes

Transfer functions

- for each node of the control flow graph

What is Data-Flow Analysis?

Some information about

- What may or must happen, in the previous or next part of the program
- Usually some approximation of values that the program computes

Transfer functions

- for each node of the control flow graph
- express the ‘effect’ of a control node

What is Data-Flow Analysis?

Some information about

- What may or must happen, in the previous or next part of the program
- Usually some approximation of values that the program computes

Transfer functions

- for each node of the control flow graph
- express the ‘effect’ of a control node

Calculate a fixpoint

What is Data-Flow Analysis?

Some information about

- What may or must happen, in the previous or next part of the program
- Usually some approximation of values that the program computes

Transfer functions

- for each node of the control flow graph
- express the ‘effect’ of a control node

Calculate a fixpoint

- because loops

Traditional Gen/Kill Sets

Traditional set base analysis

Traditional set base analysis

Traditional set based analysis has

Traditional set base analysis

Traditional set based analysis has

- Sets as the type of information that’s calculated

Traditional set base analysis

Traditional set based analysis has

- Sets as the type of information that’s calculated
- A transfer function of:

Traditional set base analysis

Traditional set based analysis has

- Sets as the type of information that's calculated
- A transfer function of:
  - previousSet ∖ kill(currentNode) ∪ gen(currentNode)

Traditional set base analysis

Traditional set based analysis has

- Sets as the type of information that's calculated
- A transfer function of:
  - previousSet ∖ kill(currentNode) ∪ gen(currentNode)
- Usually the initial set is empty

Traditional set base analysis

Traditional set based analysis has

- Sets as the type of information that's calculated
- A transfer function of:
  - previousSet ∖ kill(currentNode) ∪ gen(currentNode)
- Usually the initial set is empty

Traditional set base analysis

Traditional set based analysis has

- Sets as the type of information that's calculated
- A transfer function of:
  - previousSet ∖ kill(currentNode) ∪ gen(currentNode)
- Usually the initial set is empty

Depends on the direction of the analysis

Available Expressions

"An **expression** is **available** if it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```

AllAE

- All Available Expressions in the program

FV

- Free Variables of the argument

SE

- Subexpressions of the argument

Available Expressions

"An **expression** is **available** if it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∉ FV(e2) }
```

AllAE

- All Available Expressions in the program

FV

- Free Variables of the argument

SE

- Subexpressions of the argument

Available Expressions

"An **expression** is **available** if it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∉ FV(e2) }
```

AllAE

- All Available Expressions in the program

FV

- Free Variables of the argument

SE

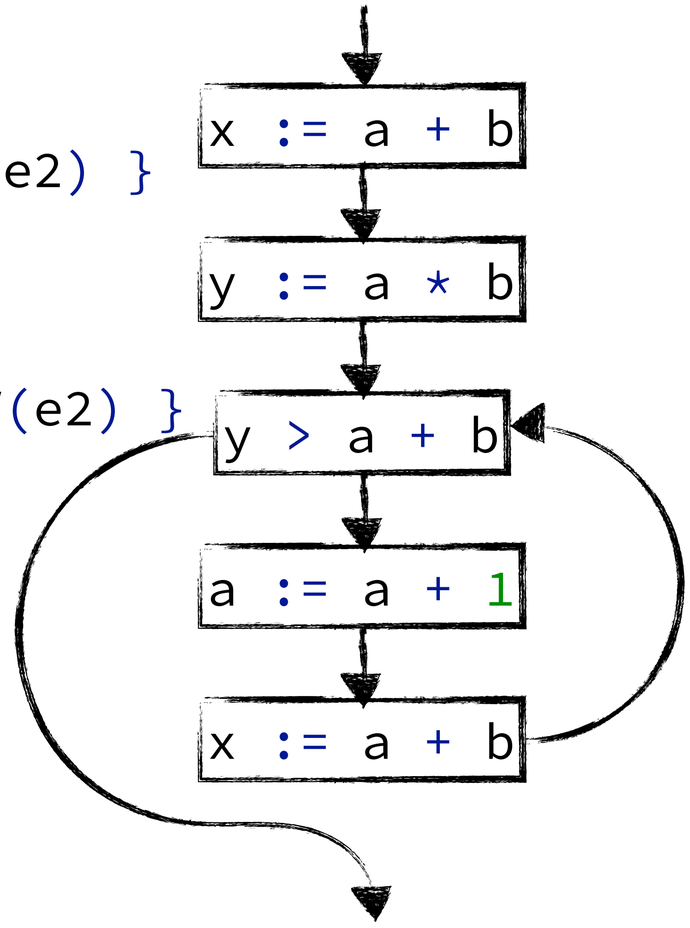
- Subexpressions of the argument

This mistake can be prevented by making the control flow more explicit in the graph: First we execute the right-hand side, then we do the assignment.

Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }

gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```

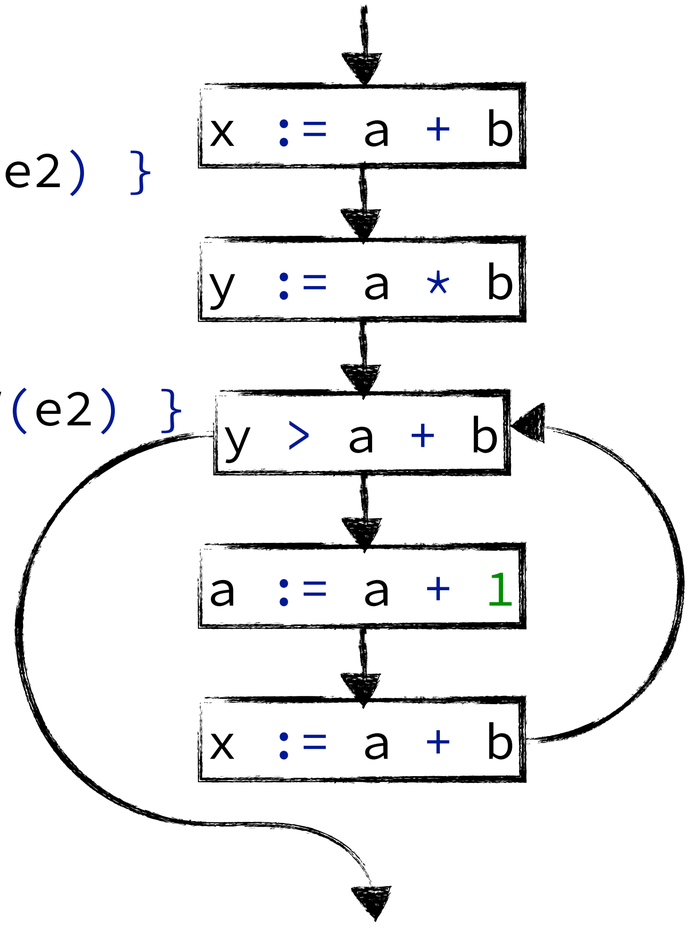


Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```

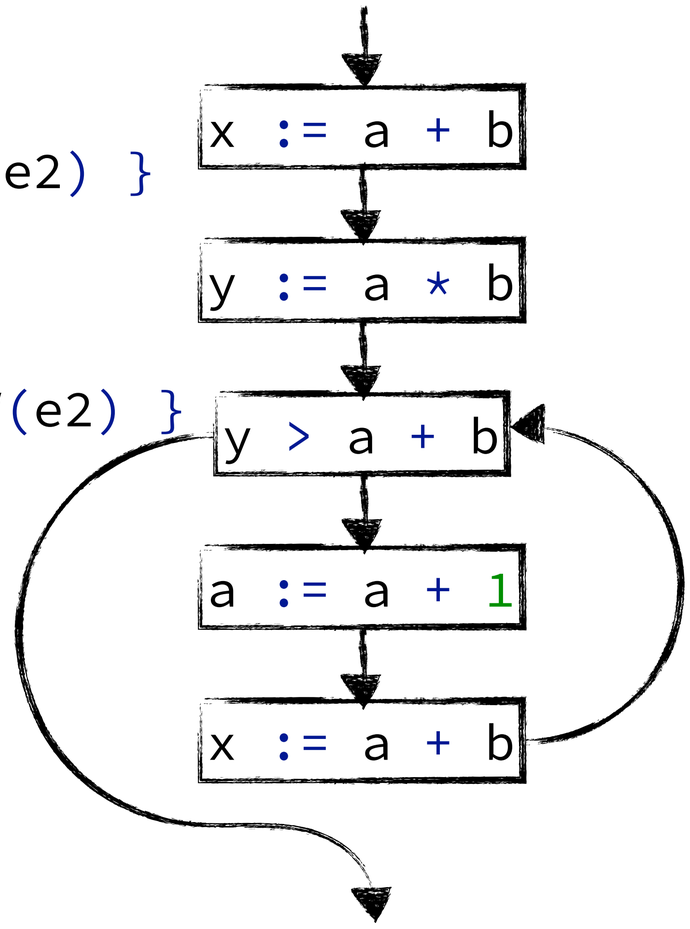


Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```



{}

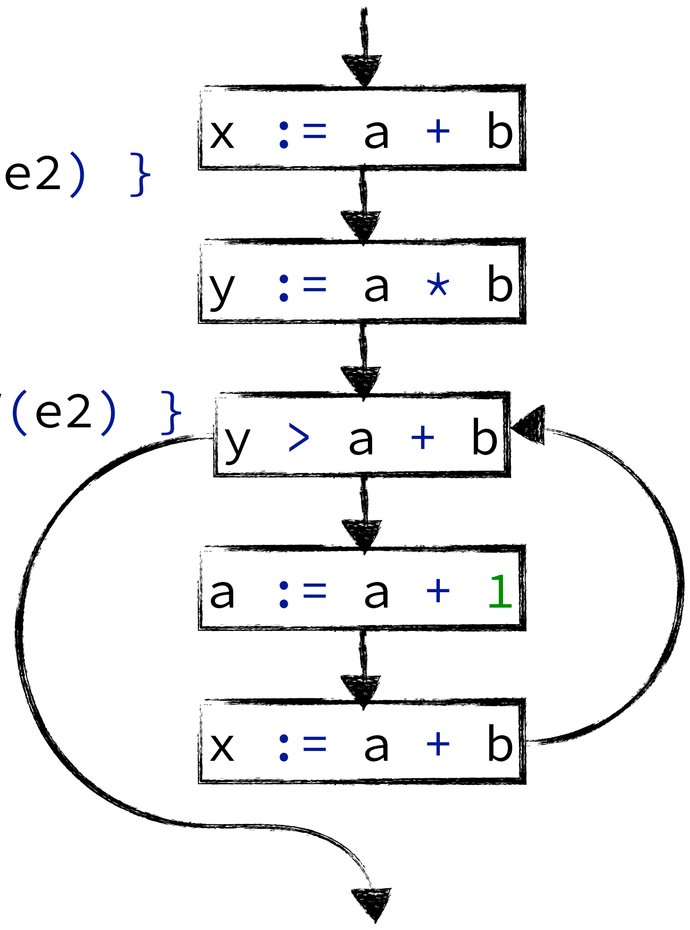
{a + b}

Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```



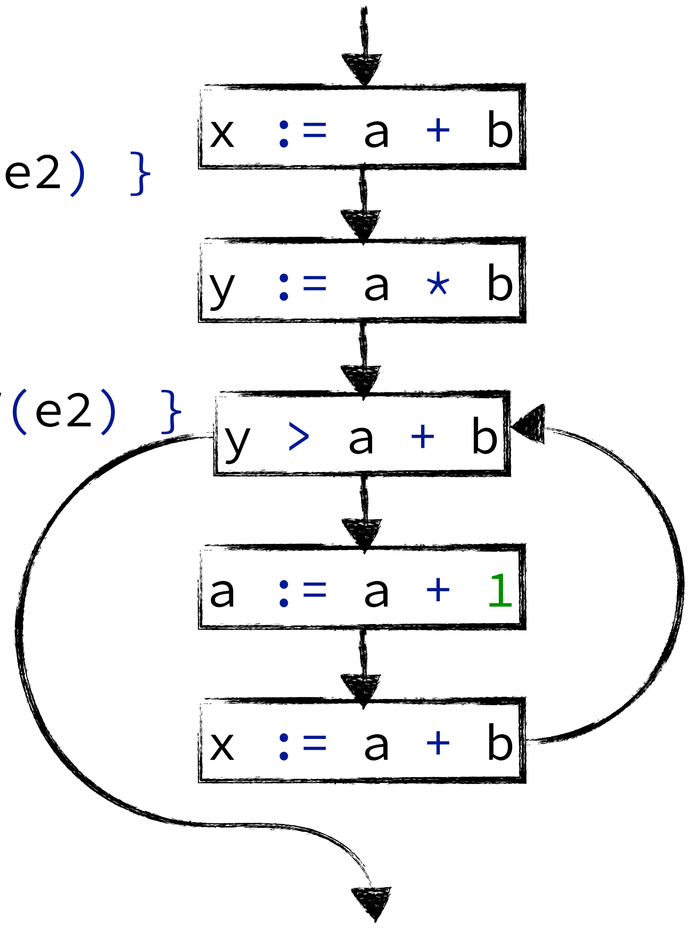

```
{}

{a + b}

{a + b, a * b}
```

Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```



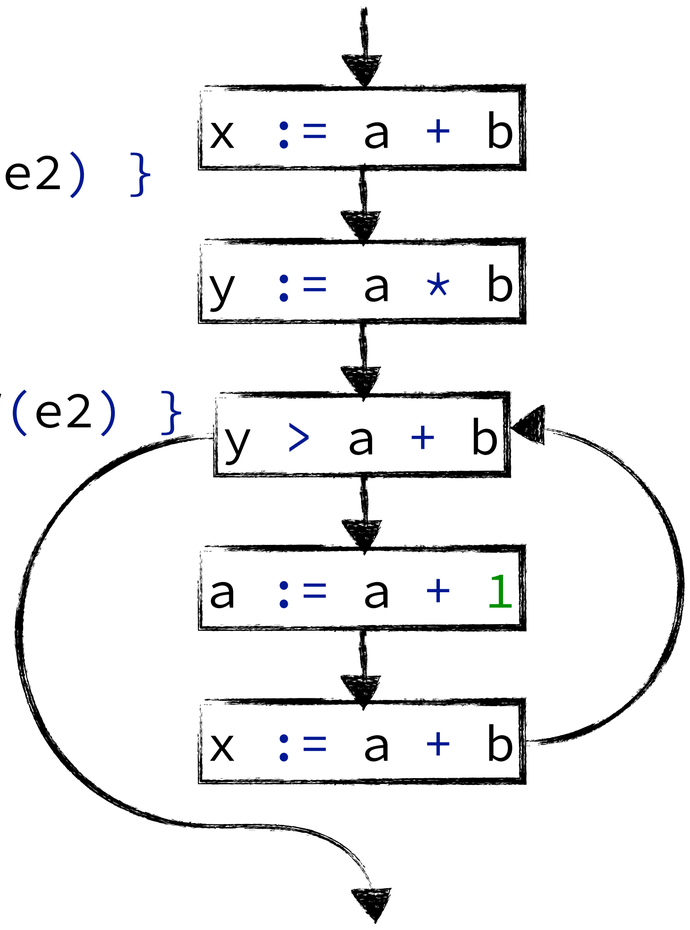

```
{}

{a + b}

{a + b, a * b}

{a + b, a * b}
```

Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```



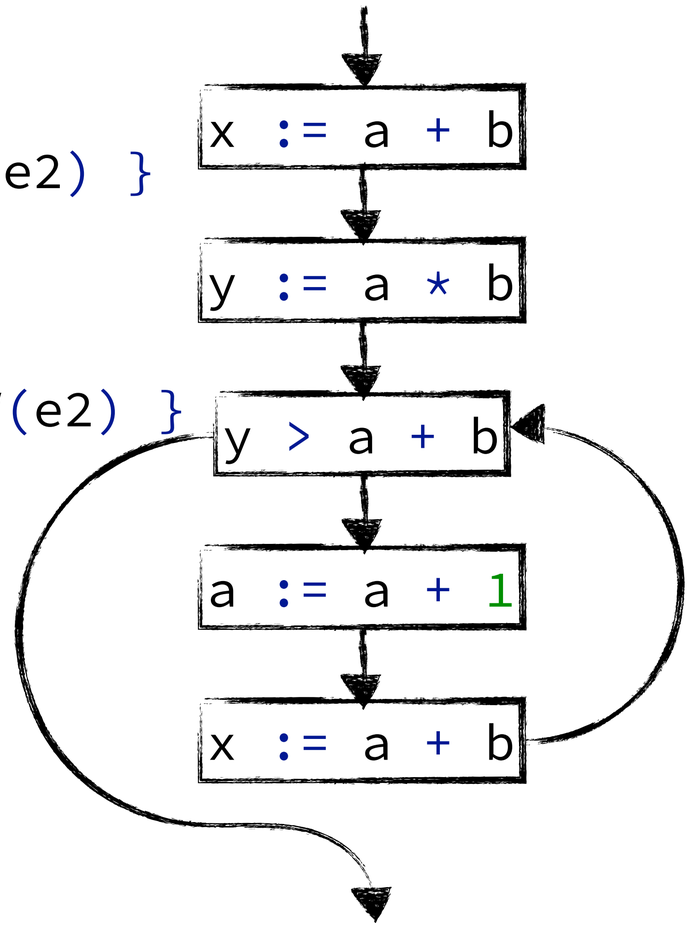

```
{}

{a + b}

{a + b, a * b}

{a + b, a * b}

{}
```

Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```



{}

{a + b}

{a + b, a * b}

{a + b, a * b}

{}

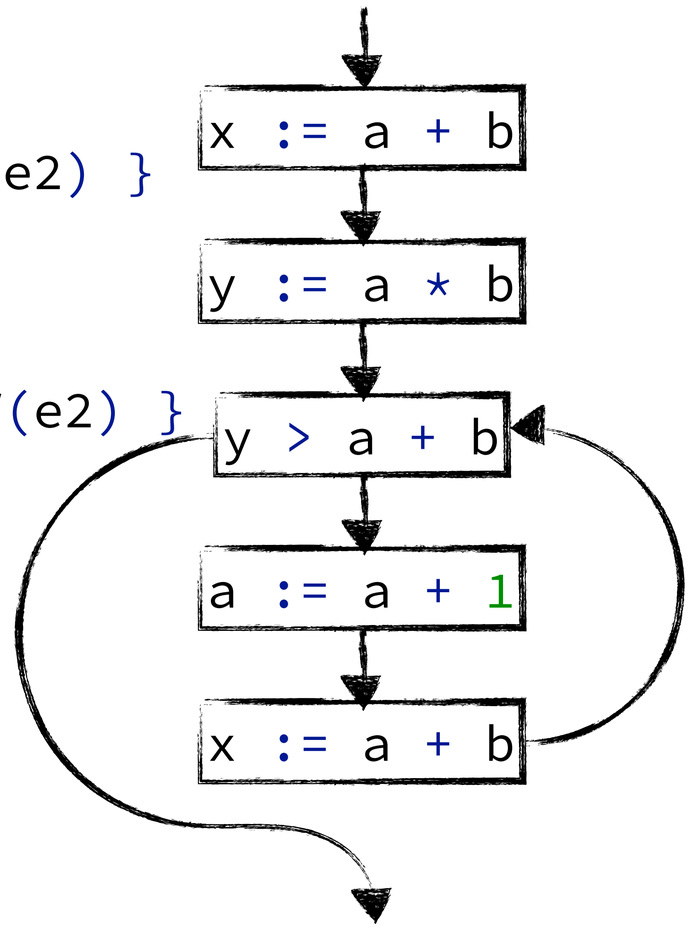
{a + b}

Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```



{}

{a + b}

{a + b, a * b}

{a + b, a * b}

{}

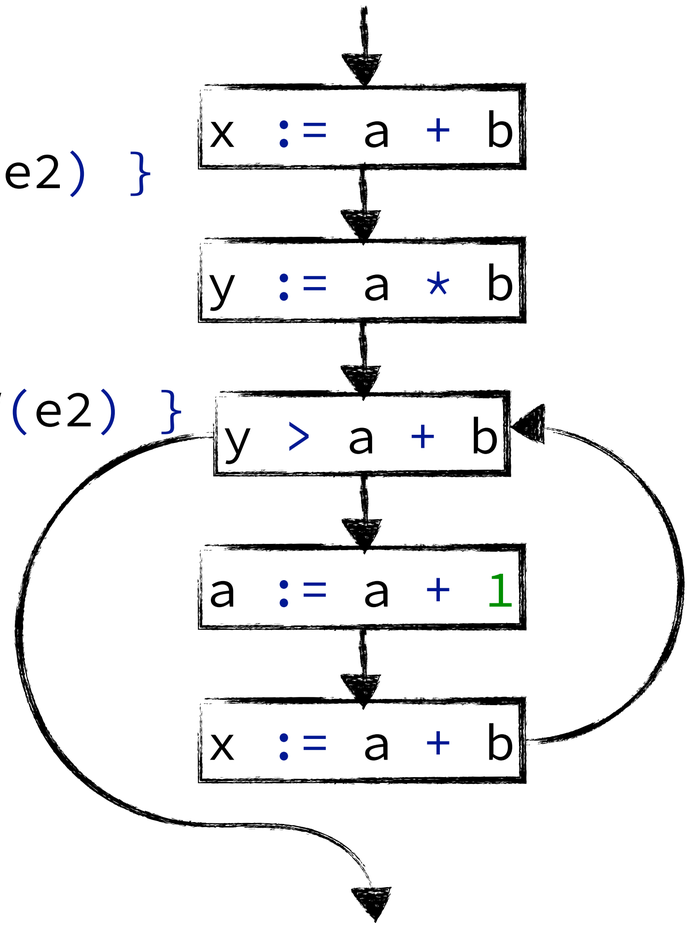
{a + b}

Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```



{}

{a + b}

{a + b, a * b}

{a + b}

{}

{a + b}

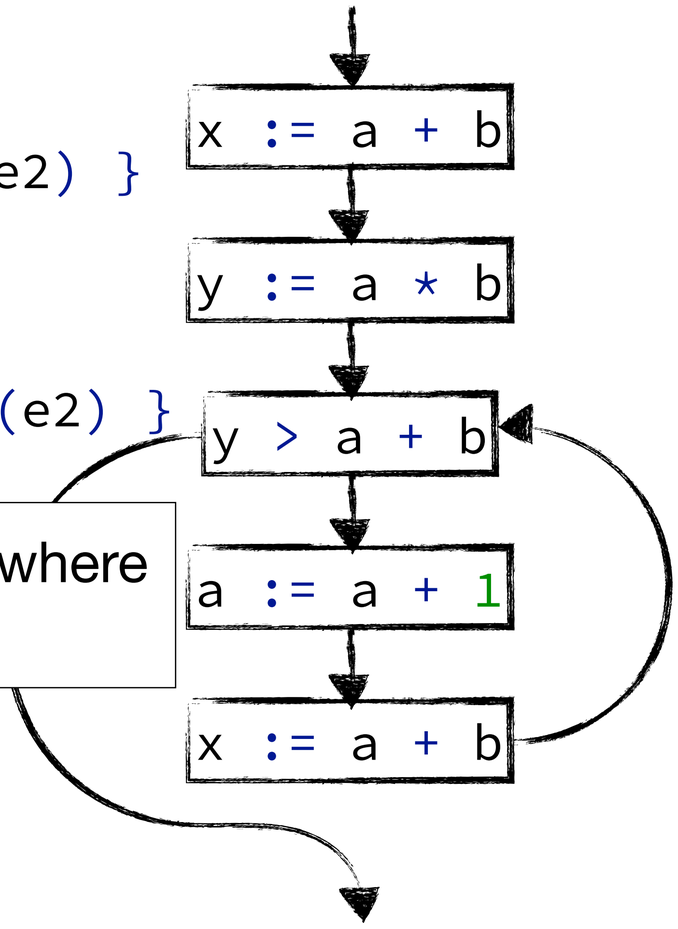
Available Expressions

"An **expression** is **available** is it *must* have already been computed, and not later modified, on all paths to the program point"

```
kill(Assign(var, e1)) :=
  { e2 ∈ AllAE | var ∈ FV(e2) }
```

```
gen(Assign(var, e1)) :=
  { e2 ∈ SE(e1) | var ∈ FV(e2) }
```

A *must* analysis intersects sets where control flow merges




```
{}

{a + b}

{a + b, a * b}

{a + b}

{}

{a + b}
```

Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

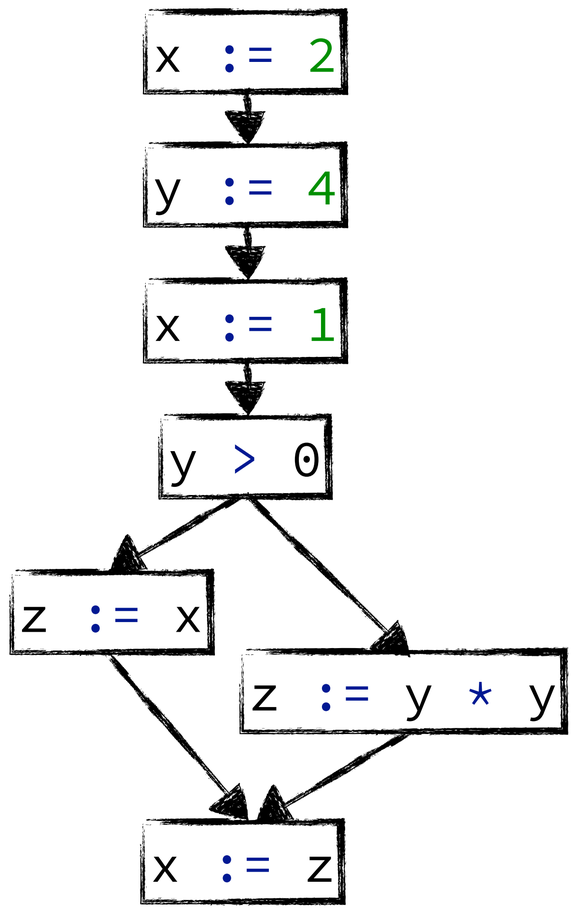
```
kill(Assign(var, e1)) :=
  { var }
```

```
gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```

Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

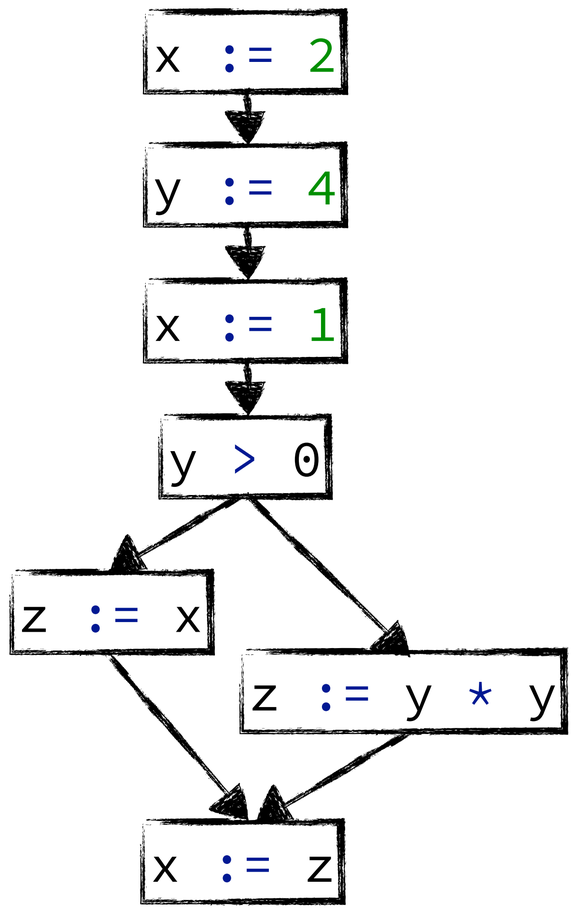
```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

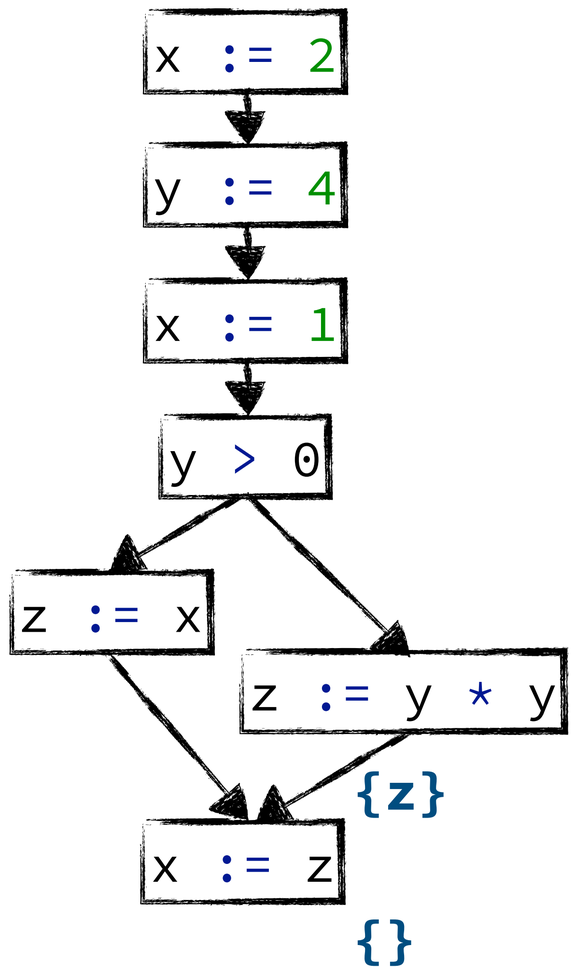
```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

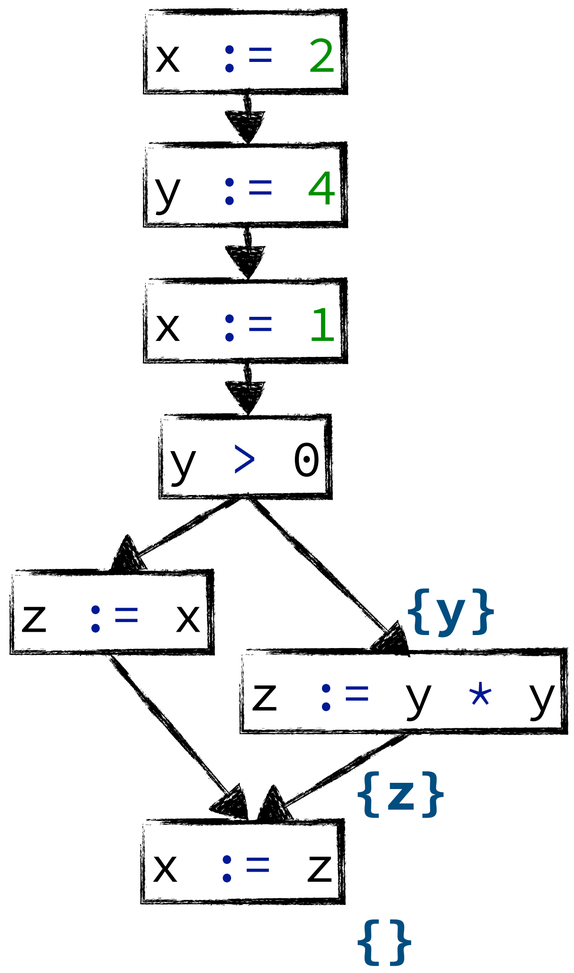
```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

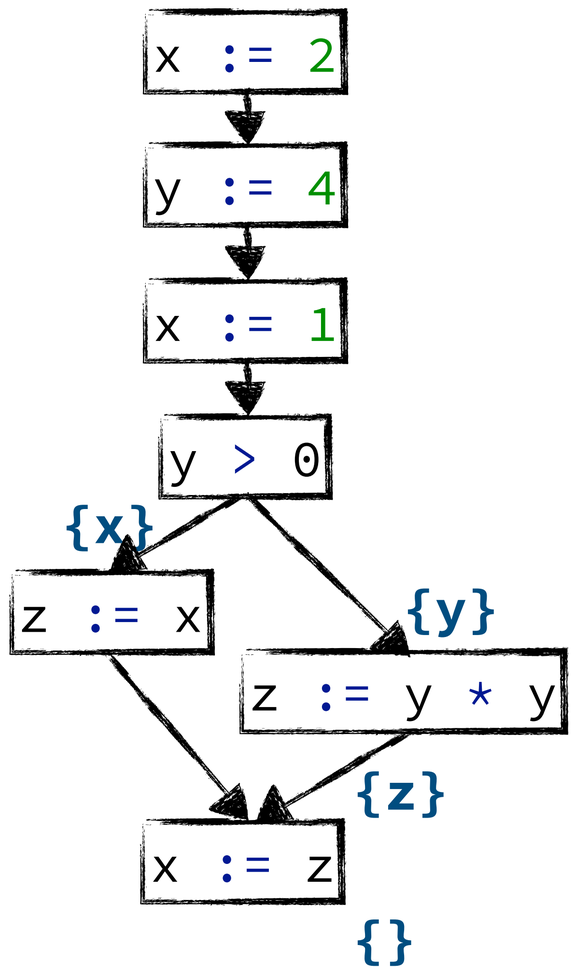
```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

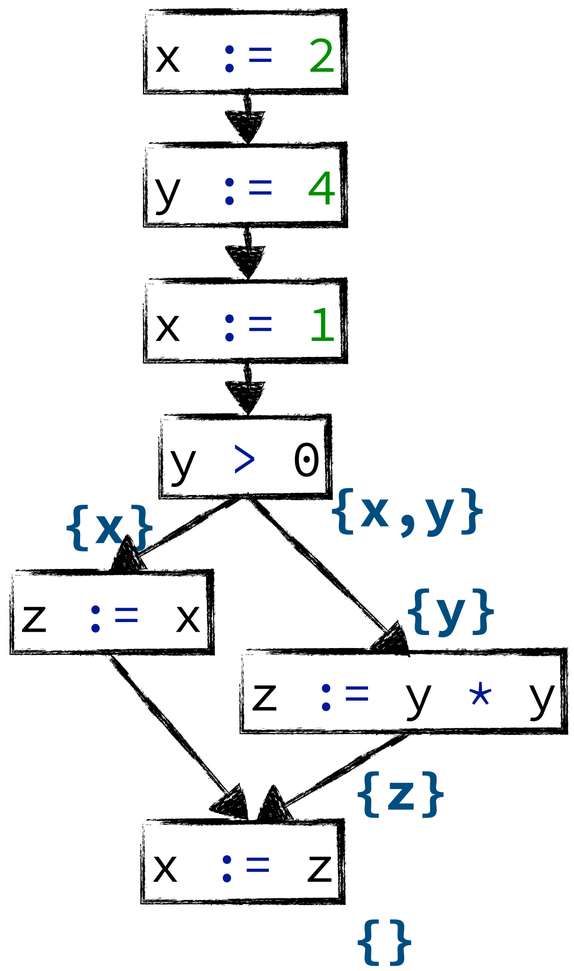
```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

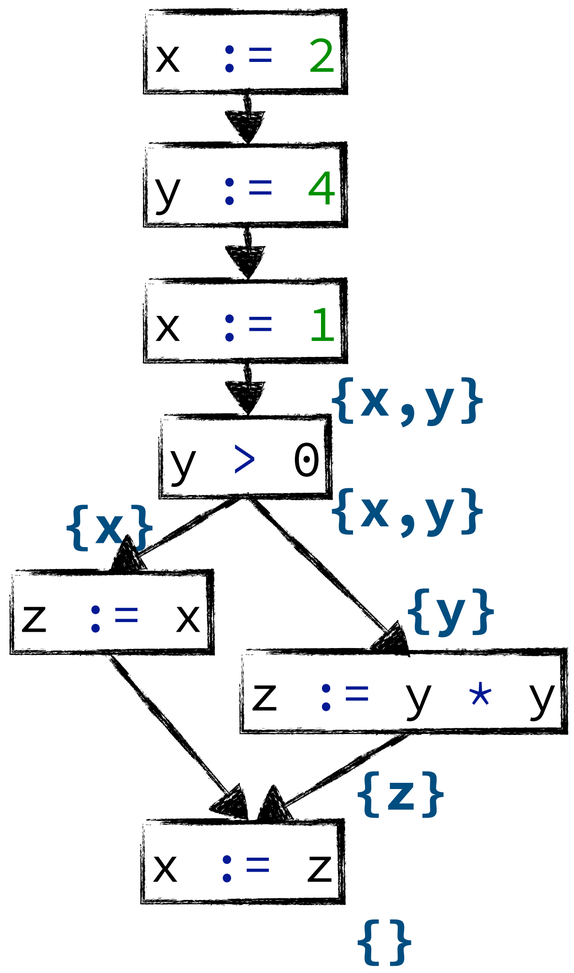
```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

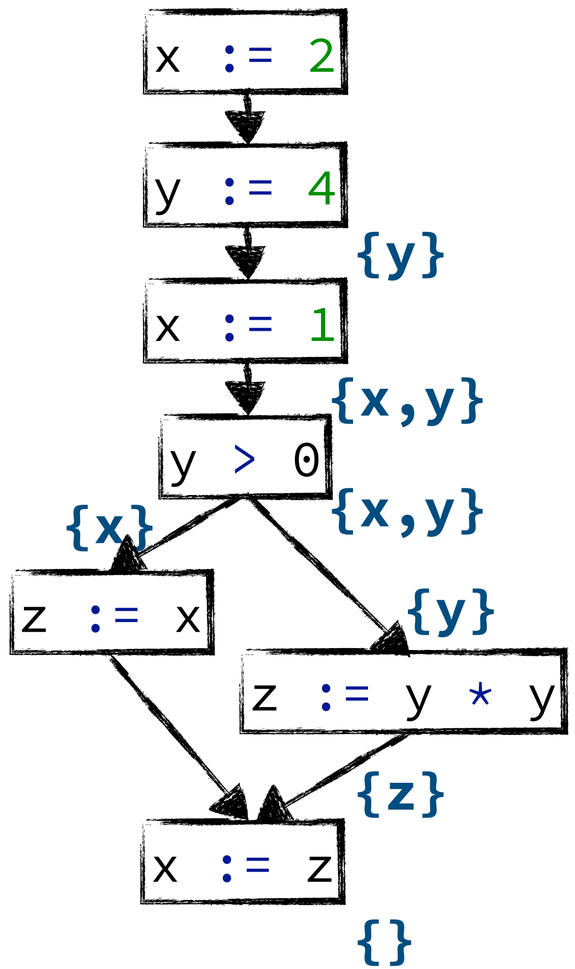
```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

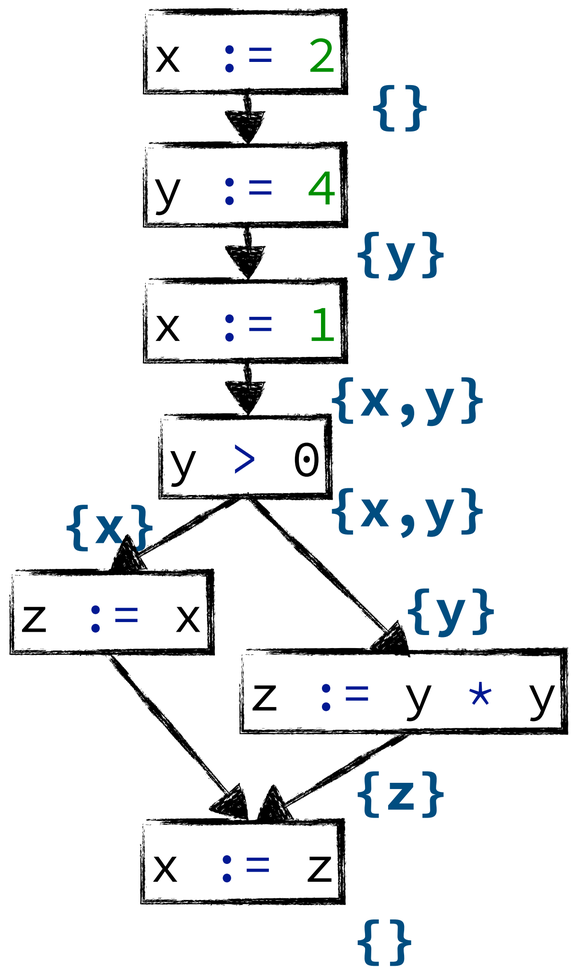
```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

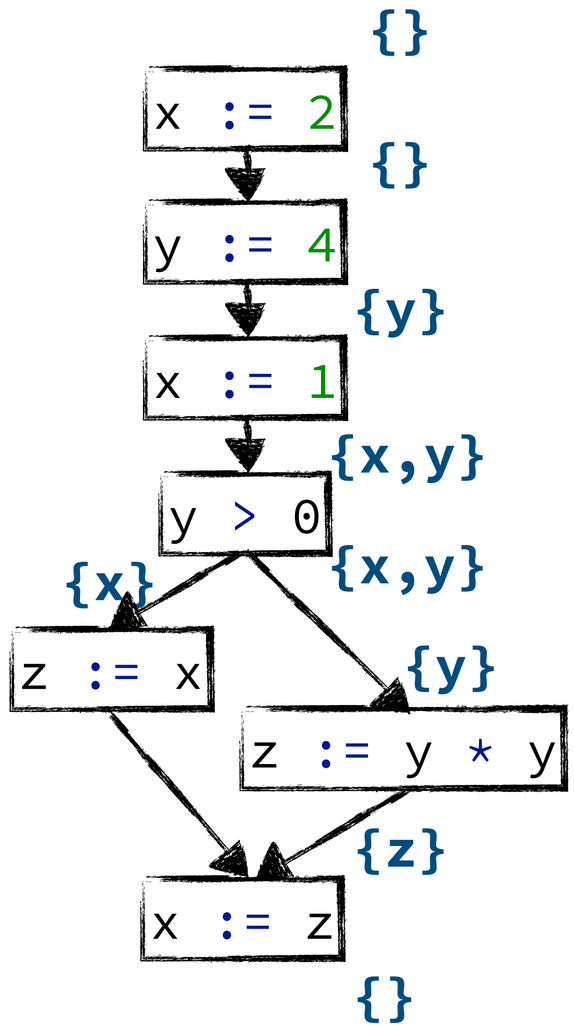
```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Live Variables

"A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path. "

```
kill(Assign(var, e1)) :=
  { var }

gen(Assign(var, e1)) :=
  { FV(e1) }

gen(b@BinOp(_, _, _)) :=
  { FV(b) }

gen(u@UnOp(_, _)) :=
  { FV(u) }
```



Traditional set based analysis

Sets as analysis information

Kill and **gen** sets per control node type

- previousSet ∖ **kill**(currentNode) ∪ **gen**(currentNode)

Can propagate either **forward** or **backward**

Can merge information with either **union** or **intersection**

- Respectively called **may** and **must** analyses

Beyond Sets

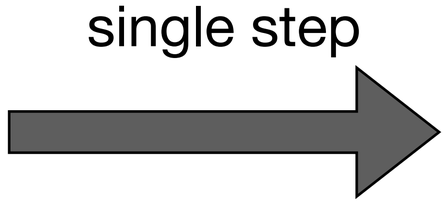
Constant propagation and folding

```
let
  var a : int := 0
  var b : int := a + 1
 in
    c := c + b;
    a := 2 * b
end
```

Constant propagation and folding

```
let
  var a : int := 0
  var b : int := a + 1
 in
    c := c + b;
    a := 2 * b
end
```

single step



```
let
  var a : int := 0
  var b : int := 0 + 1
 in
    c := c + b;
    a := 2 * b
end
```

Constant propagation and folding

```
let
  var a : int := 0
  var b : int := a + 1
 in
    c := c + b;
    a := 2 * b
end
```

single step

```
let
  var a : int := 0
  var b : int := 0 + 1
 in
    c := c + b;
    a := 2 * b
end
```

full propagation

```
let
  var a : int := 0
  var b : int := 0 + 1
 in
    c := c + 1;
    a := 2 * 1
end
```

Constant propagation and folding

```
let
  var a : int := 0
  var b : int := a + 1
 in
    c := c + b;
    a := 2 * b
end
```

Constant propagation and folding

```
let
  var a : int := 0
  var b : int := a + 1
 in
    c := c + b;
    a := 2 * b
end
```

$a \mapsto 0$

Constant propagation and folding

```
let
  var a : int := 0
  var b : int := a + 1
 in
    c := c + b;
    a := 2 * b
end
```

$a \mapsto 0$

$a \mapsto 0, b \mapsto 1$

Constant propagation and folding

```
let
  var a : int := 0
  var b : int := a + 1
 in
    c := c + b;
    a := 2 * b
end
```

$a \mapsto 0$

$a \mapsto 0,\ b \mapsto 1$

$a \mapsto 0,\ b \mapsto 1,\ c \mapsto ?$

Constant propagation and folding

```
let
  var a : int := 0
  var b : int := a + 1
 in
    c := c + b;
    a := 2 * b
end
```

$a \mapsto 0$

$a \mapsto 0,\ b \mapsto 1$

$a \mapsto 0,\ b \mapsto 1,\ c \mapsto ?$

$a \mapsto 2,\ b \mapsto 1,\ c \mapsto ?$

Constant propagation and folding

The type of the analysis information

- Variables bound to either a particular *constant* or a *marker for non-constants*

The transfer functions per control node

- Basically an interpreter implementation for constants
- Needs to propagate markers when found

Monotone Frameworks

Termination

Termination

Data-Flow Analysis needs fixpoint computation

- Because of loops

Termination

Data-Flow Analysis needs fixpoint computation

- Because of loops

To terminate, there needs to *be* a fixpoint

- **And** we need to actually *get* to that fix point
- How can we check this?

Lattice Theory

Lattice Theory

A set X is totally ordered under ≤ if for a, b, c ∈ X

Lattice Theory

A set X is totally ordered under ≤ if for a, b, c ∈ X

- $a \leq b \wedge b \leq a \Rightarrow a = b$ (antisymmetry)

Lattice Theory

A set X is totally ordered under ≤ if for a, b, c ∈ X

- $a \leq b \wedge b \leq a \Rightarrow a = b$ (antisymmetry)
- $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (transitivity)

Lattice Theory

A set X is totally ordered under ≤ if for a, b, c ∈ X

- $a \leq b \wedge b \leq a \Rightarrow a = b$ (antisymmetry)
- $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (transitivity)
- $a \leq b \vee b \leq a$ (totality)

Lattice Theory

A set X is totally ordered under ≤ if for a, b, c ∈ X

- $a \leq b \wedge b \leq a \Rightarrow a = b$ (antisymmetry)
- $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (transitivity)
- $a \leq b \vee b \leq a$ (totality)

A partial ordering drop the totality constraint

Lattice Theory

A set X is totally ordered under ≤ if for a, b, c ∈ X

- $a \leq b \wedge b \leq a \Rightarrow a = b$ (antisymmetry)
- $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (transitivity)
- $a \leq b \vee b \leq a$ (totality)

A partial ordering drop the totality constraint

- e.g. subset inclusion:

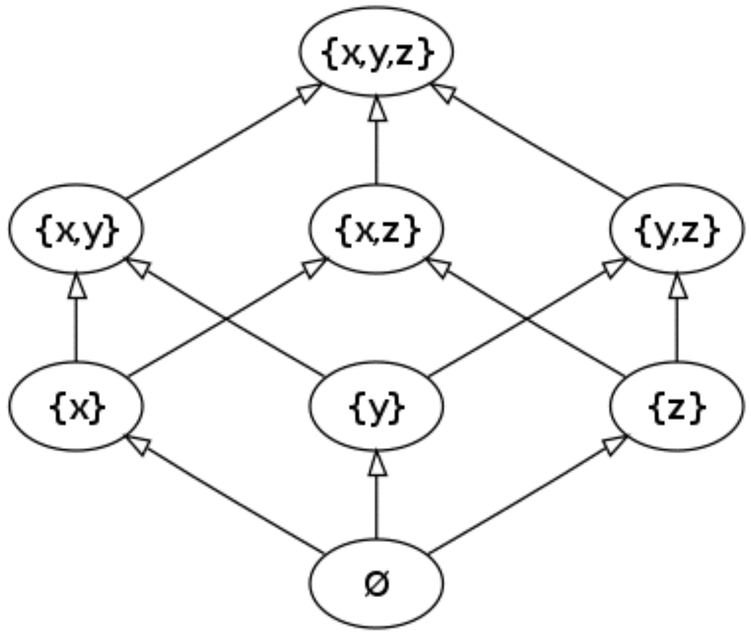
Lattice Theory

A set X is totally ordered under ≤ if for a, b, c ∈ X

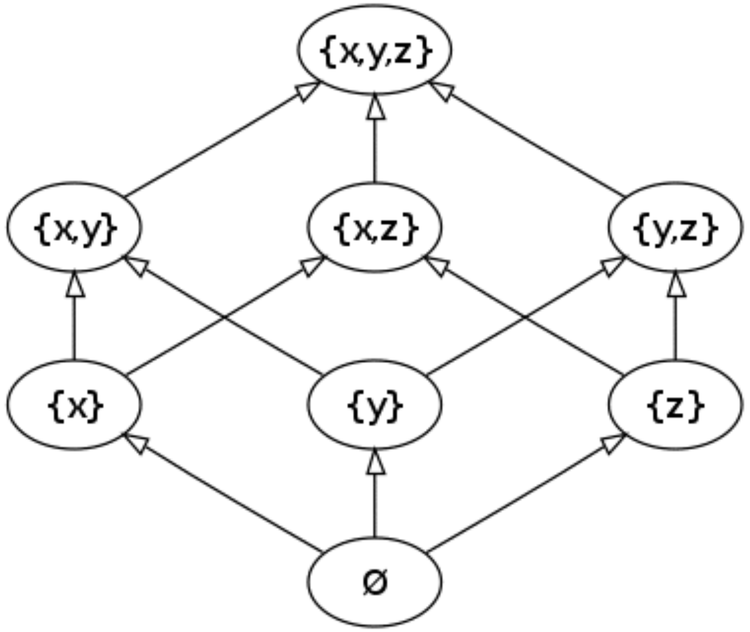
- $a \leq b \wedge b \leq a \Rightarrow a = b$ (antisymmetry)
- $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (transitivity)
- $a \leq b \vee b \leq a$ (totality)

A partial ordering drop the totality constraint

- e.g. subset inclusion:



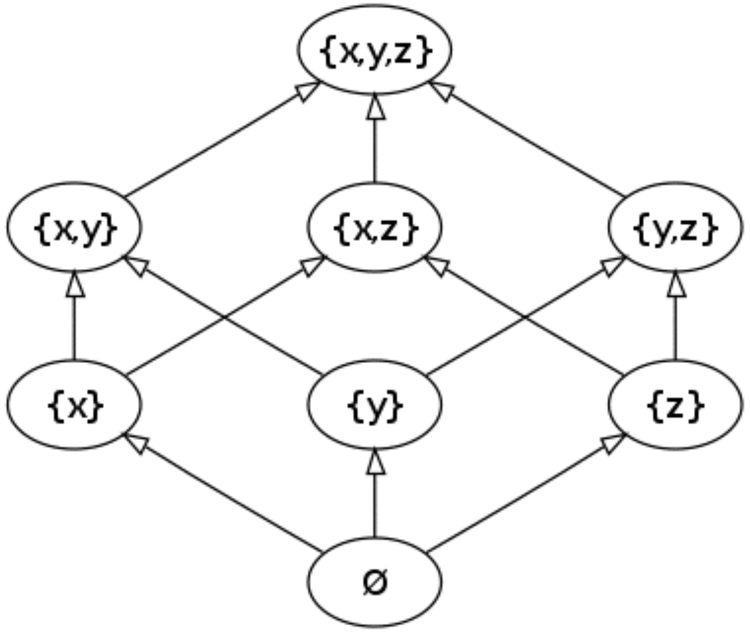
Lattice Theory



Lattice Theory

A *Lattice* is a partially ordered set where

- every two elements have a *unique* least upper bound (or supremum or join)
- every two elements have a *unique* greatest lower bound (or infimum or meet)



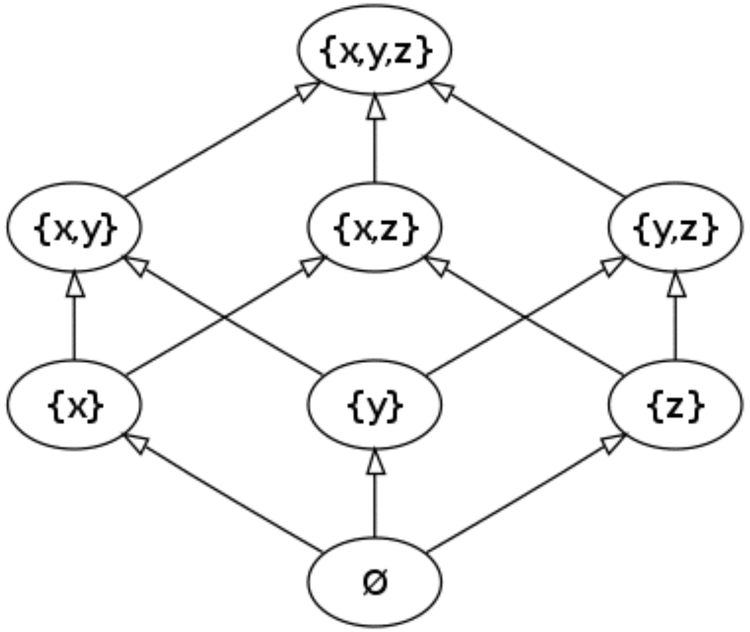
Lattice Theory

A *Lattice* is a partially ordered set where

- every two elements have a *unique* least upper bound (or supremum or join)
- every two elements have a *unique* greatest lower bound (or infimum or meet)

Least upper bound (LUB)

- $a \sqsubseteq b \Leftrightarrow a \sqcup b = b$
- $a \sqcup b = c \Rightarrow a \sqsubseteq c \wedge b \sqsubseteq c$



Lattice Theory

A *Lattice* is a partially ordered set where

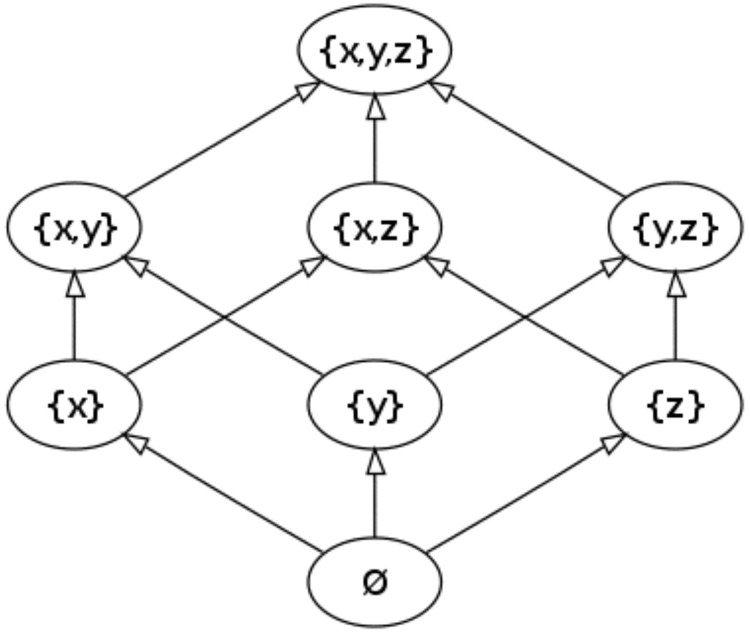
- every two elements have a *unique* least upper bound (or supremum or join)
- every two elements have a *unique* greatest lower bound (or infimum or meet)

Least upper bound (LUB)

- $a \sqsubseteq b \Leftrightarrow a \sqcup b = b$
- $a \sqcup b = c \Rightarrow a \sqsubseteq c \wedge b \sqsubseteq c$

Greatest lower bound (GLB)

- $a \sqsubseteq b \Leftrightarrow a \sqcap b = a$
- $a \sqcap b = c \Rightarrow c \sqsubseteq a \wedge c \sqsubseteq b$



Lattice Theory

A *Lattice* is a partially ordered set where

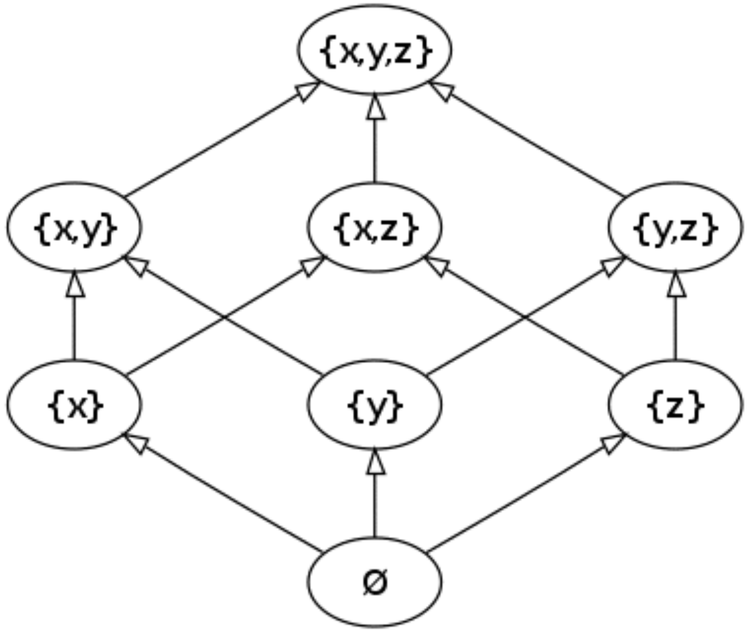
- every two elements have a *unique* least upper bound (or supremum or join)
- every two elements have a *unique* greatest lower bound (or infimum or meet)

Least upper bound (LUB)

- $a \sqsubseteq b \Leftrightarrow a \sqcup b = b$
- $a \sqcup b = c \Rightarrow a \sqsubseteq c \wedge b \sqsubseteq c$

Greatest lower bound (GLB)

- $a \sqsubseteq b \Leftrightarrow a \sqcap b = a$
- $a \sqcap b = c \Rightarrow c \sqsubseteq a \wedge c \sqsubseteq b$



A *bounded* lattice has a top and bottom

- These are $\top$ and $\bot$ respectively

Lattices for data-flow analysis

Consider $\top$ as the coarsest approximation

- It's a safe approximation, because it says we're not sure of anything

Then we can combine data-flow information with $\sqcup$

- It is the most information preserving combination of information

Lattices for data-flow analysis

Lattices for data-flow analysis

Transfer functions should be monotone increasing

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f, $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f, $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- This includes the identity function

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f, $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- This includes the identity function

Monotone transfer functions give a termination guarantee

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f, $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- This includes the identity function

Monotone transfer functions give a termination guarantee

- In a loop we reach a fixpoint if the functions start returning the same thing

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f, $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- This includes the identity function

Monotone transfer functions give a termination guarantee

- In a loop we reach a fixpoint if the functions start returning the same thing
- Worst case scenario: the loop reaches $\top$

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f, $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- This includes the identity function

Monotone transfer functions give a termination guarantee

- In a loop we reach a fixpoint if the functions start returning the same thing
- Worst case scenario: the loop reaches $\top$
- IMPORTANT: This only works if the lattice is *finite*

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f, $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- This includes the identity function

Monotone transfer functions give a termination guarantee

- In a loop we reach a fixpoint if the functions start returning the same thing
- Worst case scenario: the loop reaches $\top$
- IMPORTANT: This only works if the lattice is *finite*

General interval analysis has an infinite lattice

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f, $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- This includes the identity function

Monotone transfer functions give a termination guarantee

- In a loop we reach a fixpoint if the functions start returning the same thing
- Worst case scenario: the loop reaches $\top$
- IMPORTANT: This only works if the lattice is *finite*

General interval analysis has an infinite lattice

- $\top = [-\infty,\infty]$

Lattices for data-flow analysis

Transfer functions should be monotone increasing

- i.e. for transfer function f, $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$
- This includes the identity function

Monotone transfer functions give a termination guarantee

- In a loop we reach a fixpoint if the functions start returning the same thing
- Worst case scenario: the loop reaches $\top$
- IMPORTANT: This only works if the lattice is *finite*

General interval analysis has an infinite lattice

- $\top = [-\infty,\infty]$
- If a loop adds a finite number to a variable, you never get to ∞

Recap

An analysis consists of

- The type of the analysis information
- The *transfer functions* that express the ‘effect’ of a control node
- The initial analysis information

Recap

An analysis consists of

- The type of the analysis information, **and the lattice instance for that type**
- The *transfer functions* that express the ‘effect’ of a control node
  - **These should be monotone with respect to the lattice**
- The initial analysis information

Data-Flow Languages

How to define the data-flow rules of a language

?

Data-Flow specification approaches

Languages used for specifying data-flow analysis

- Attribute grammars: JastAdd [Söderberg'13]
- Datalog [Smaragdakis'15]: Doop [Kasterinis'13], Flix [Madsen'16]
- $INCA_L$ / MPS-DF [Szabó'18 / '16]
- Temporal Logic: Silver [VanWyk'10]

Separation of Concerns

Representation

- **Control Flow Graphs**
- **Before/After data-flow information on CFG nodes**

Declarative Rules

- To define control-flow rules of a language
- To define data-flow rules of a language

Language-Independent Tooling

- Analysis
- Code completion
- Refactoring
- Optimisation
- …

Control-flow graphs in FlowSpec

FlowSpec

Example program

```
x := 1;
if y > x then
  z := y;
else
  z := y * y;
y := a * b;
while y > a + b do
 (a := a + 1;
  x := a + b)
```

Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end
```

Example program

```
x := 1;
if y > x then
  z := y;
else
  z := y * y;
y := a * b;
while y > a + b do
 (a := a + 1;
  x := a + b)
```

Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end
```

Example program

```
start

x := 1;

if y > x then

  z := y;

else

  z := y * y;

y := a * b;

while y > a + b do

 (a := a + 1;

  x := a + b)

end
```

Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit
```

Example program

```
start

x := 1;

if y > x then

  z := y;

else

  z := y * y;

y := a * b;

while y > a + b do

 (a := a + 1;

  x := a + b)

end
```

Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit
```

Example program

```
start

x := 1;

if y > x then

  z := y;

else

  z := y * y;

y := a * b;

while y > a + b do

  (a := a + 1;

  x := a + b)

end
```

Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit
```

Example program

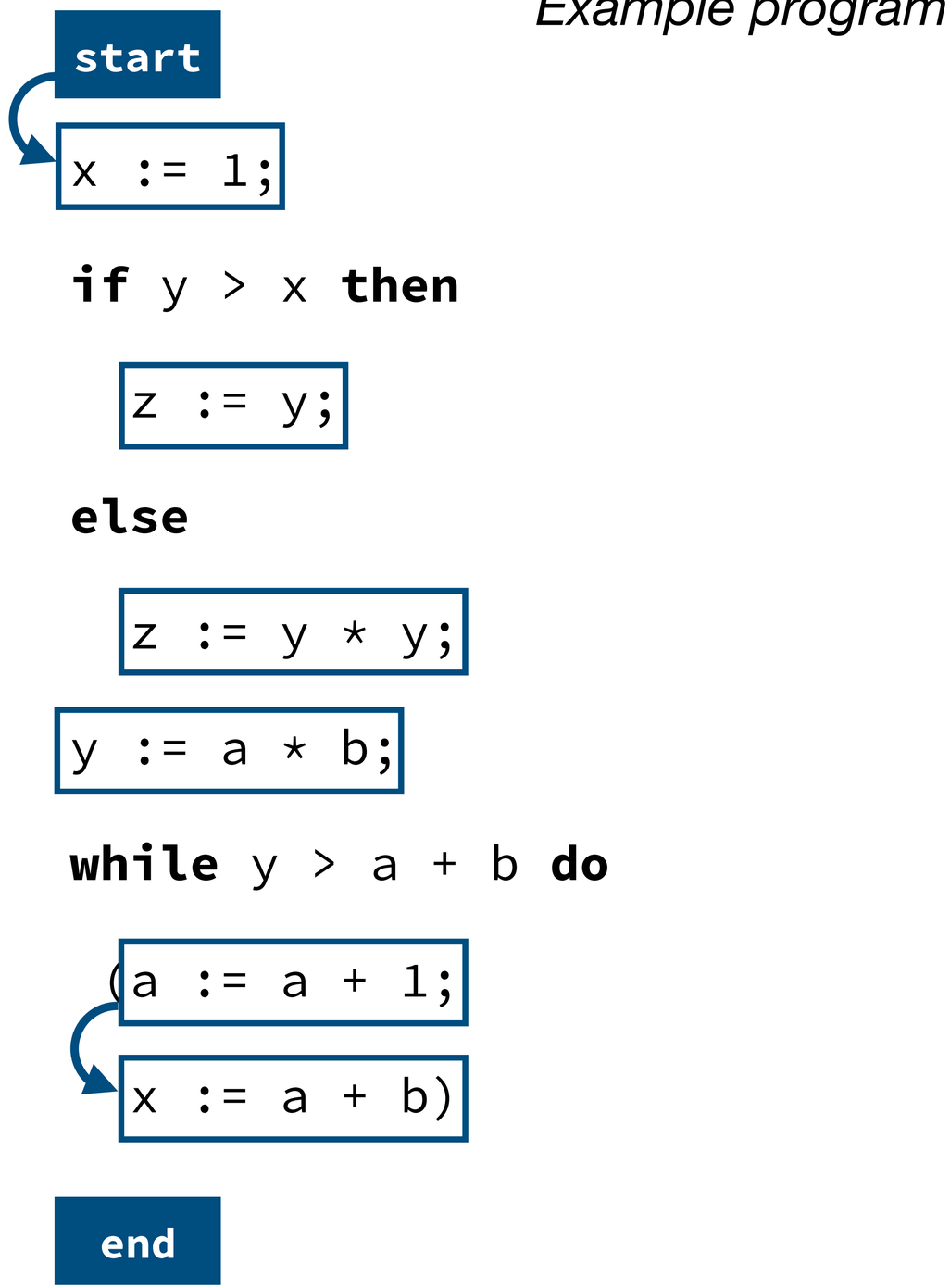
```
start
x := 1;
if y > x then
  z := y;
else
  z := y * y;
y := a * b;
while y > a + b do
  (a := a + 1;
  x := a + b)
end
```

Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit
```

Example program

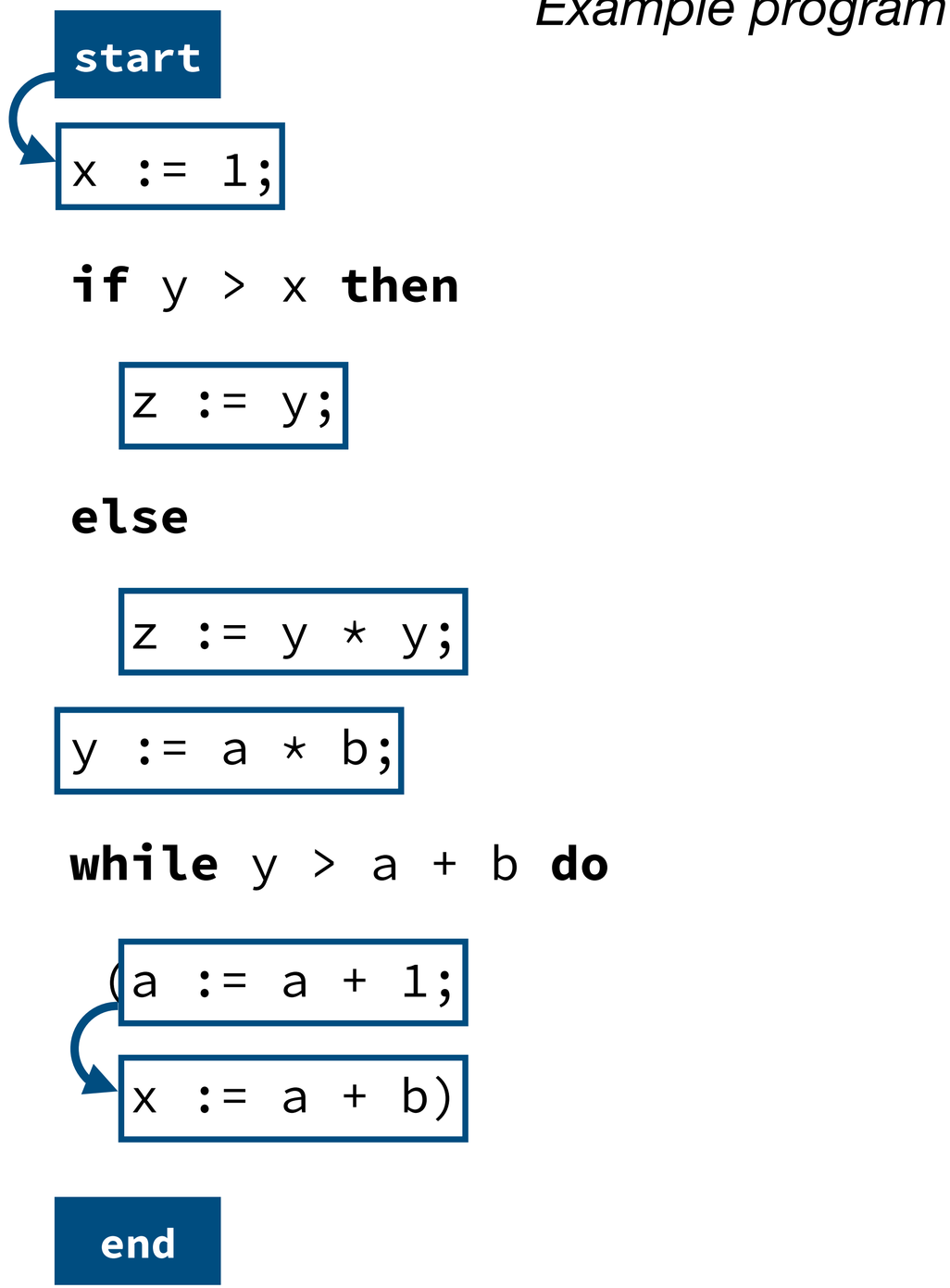


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
            c -> cfg e -> exit
```

Example program

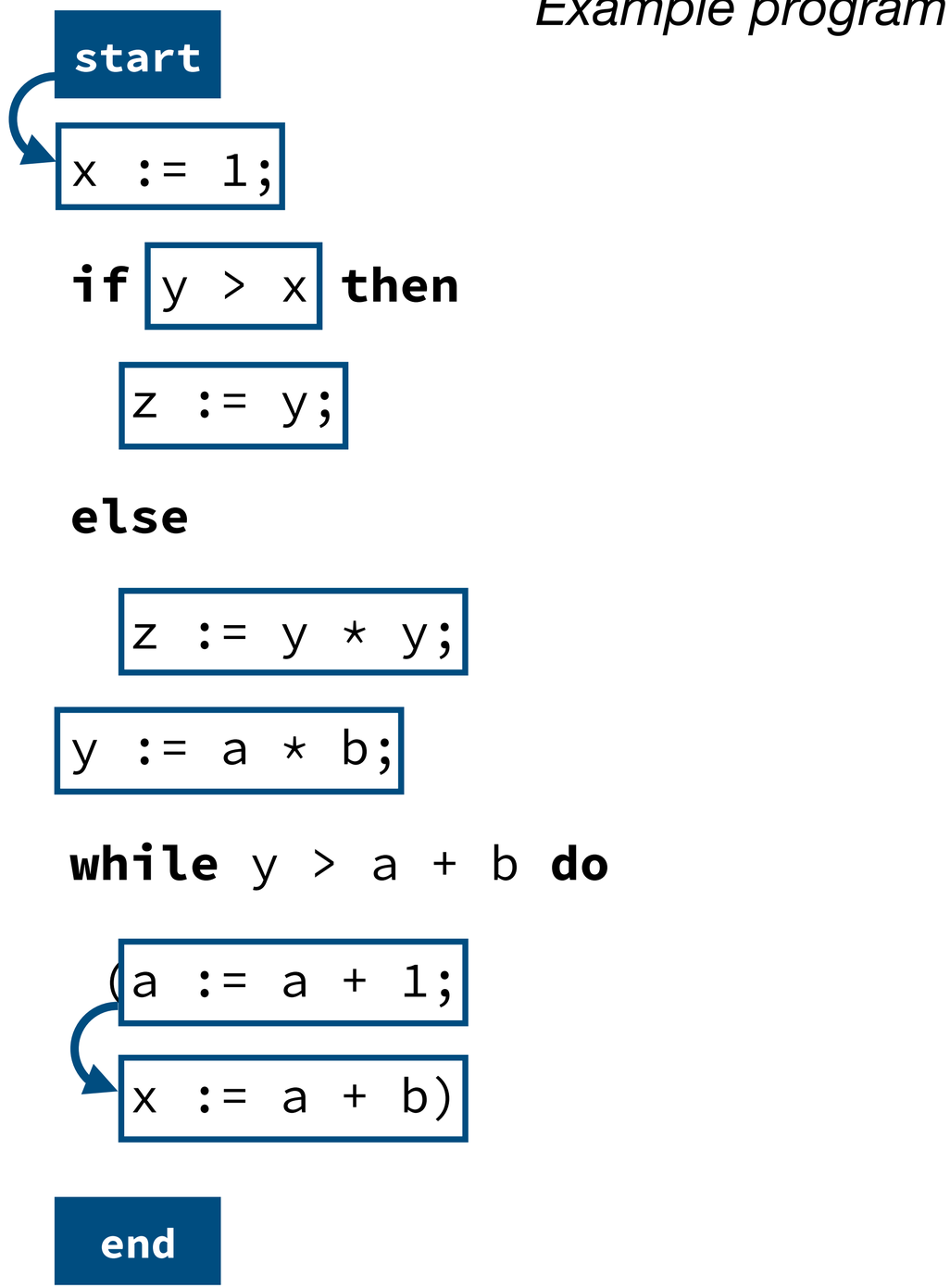


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
            c -> cfg e -> exit
```

Example program

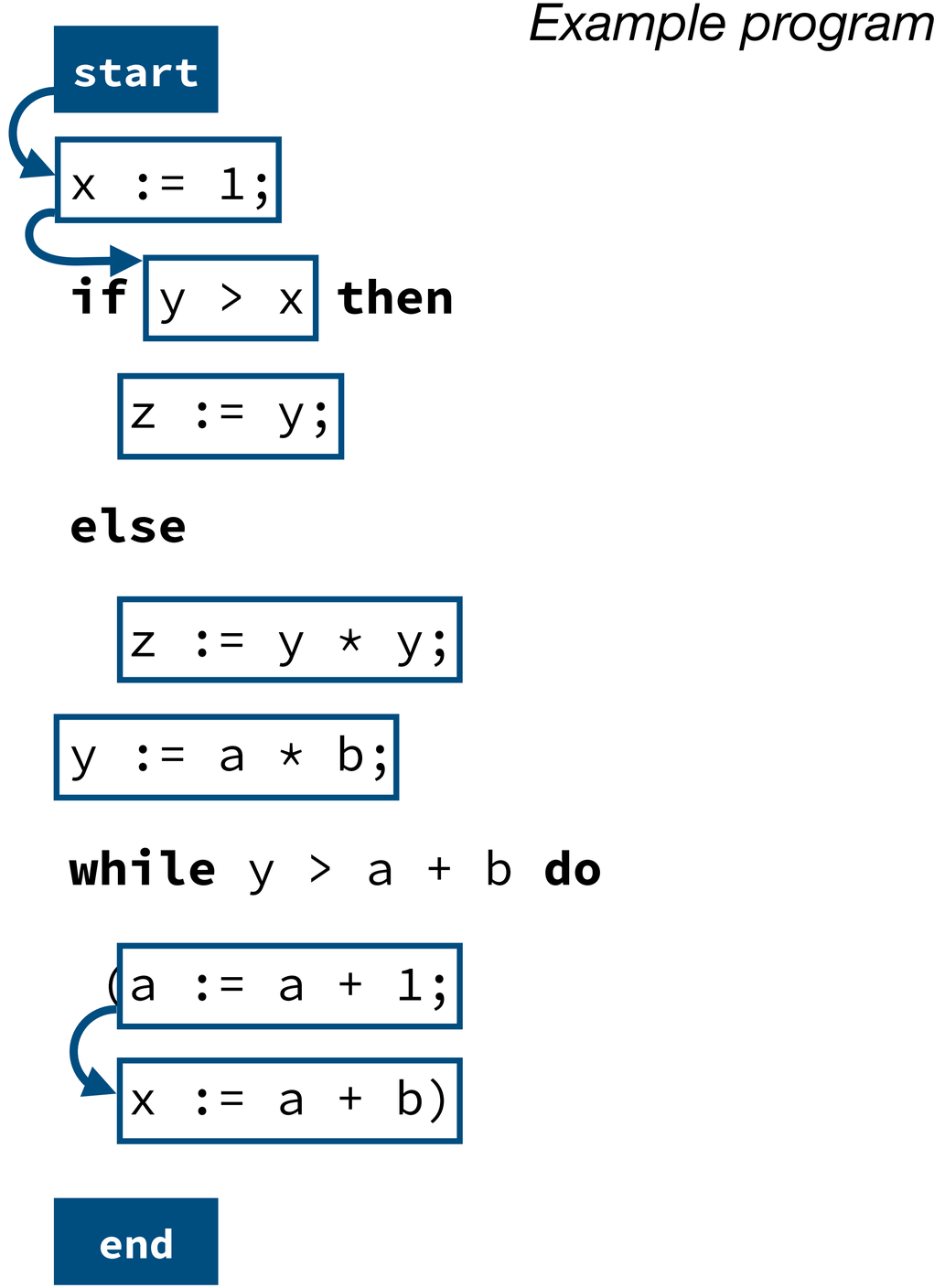


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
            c -> cfg e -> exit
```

Example program

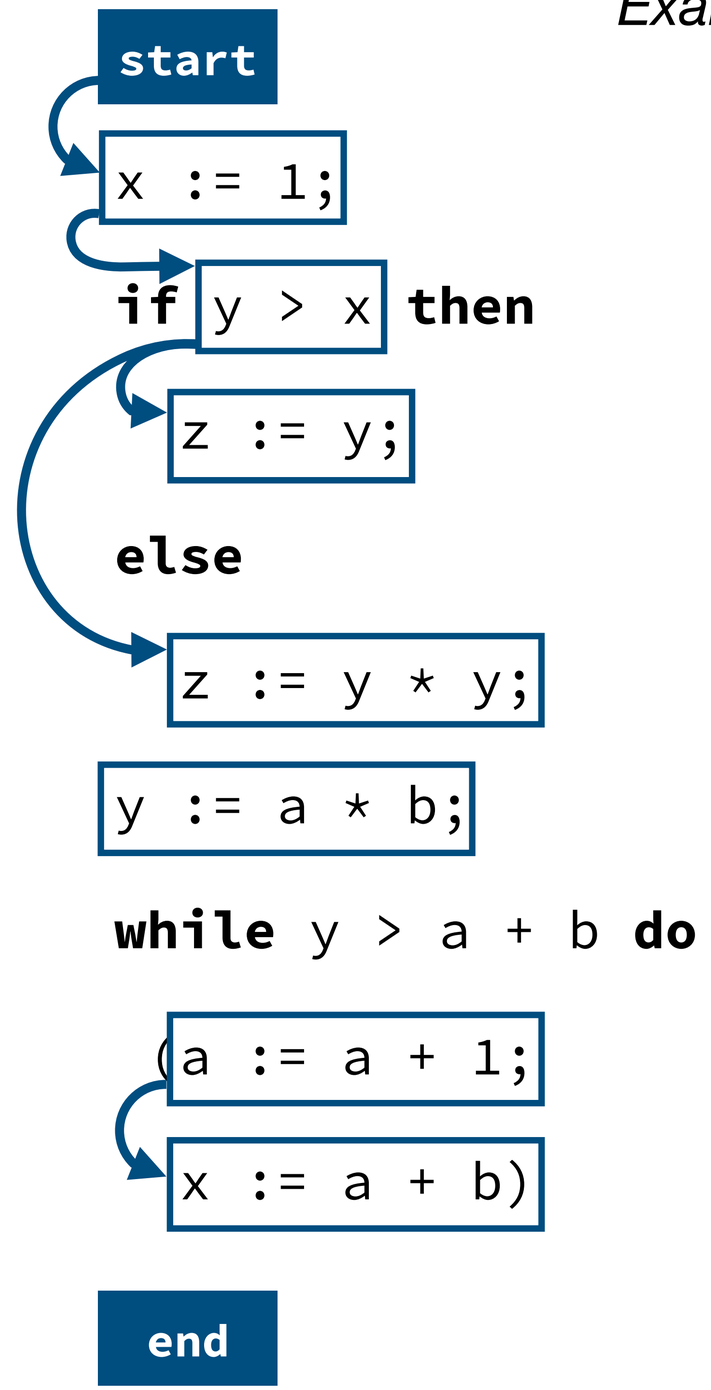


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
           c -> cfg e -> exit
```

Example program

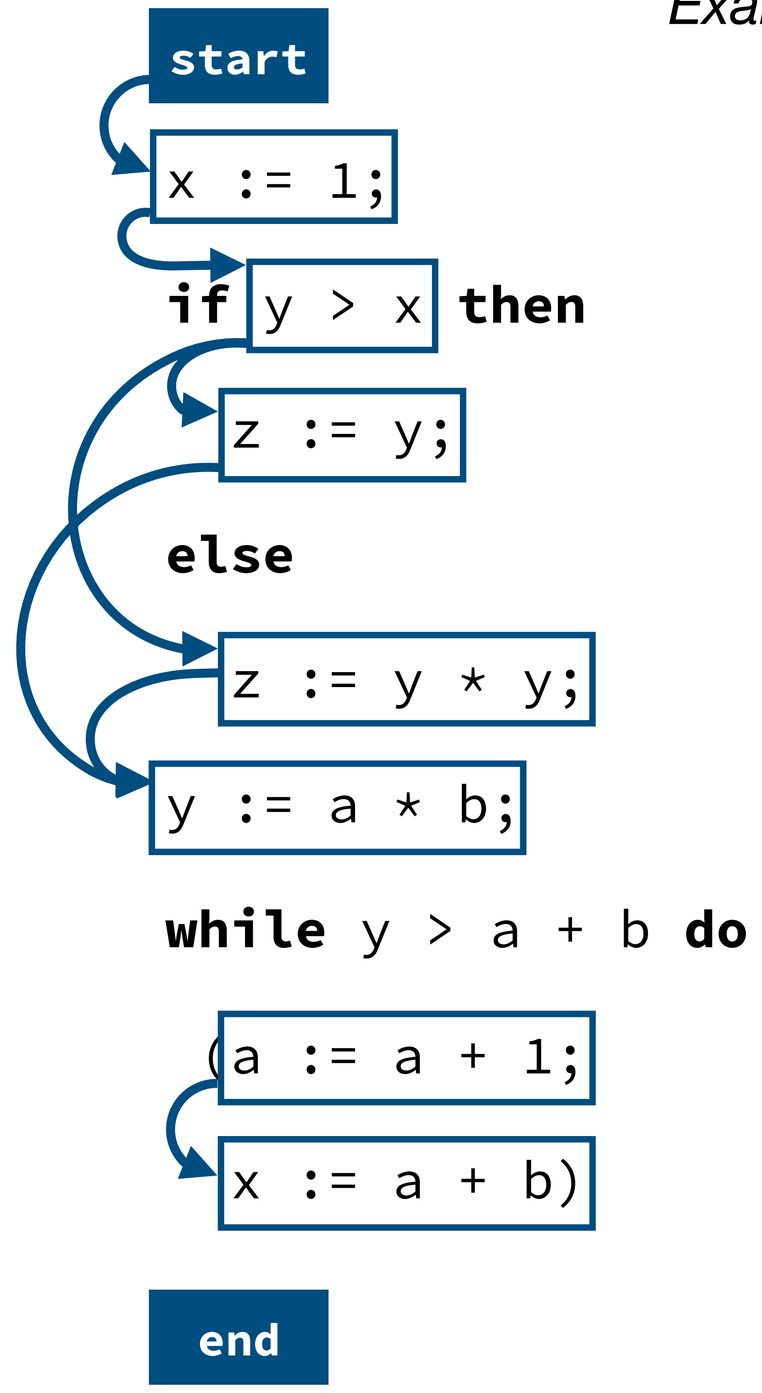


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
           c -> cfg e -> exit
```

Example program

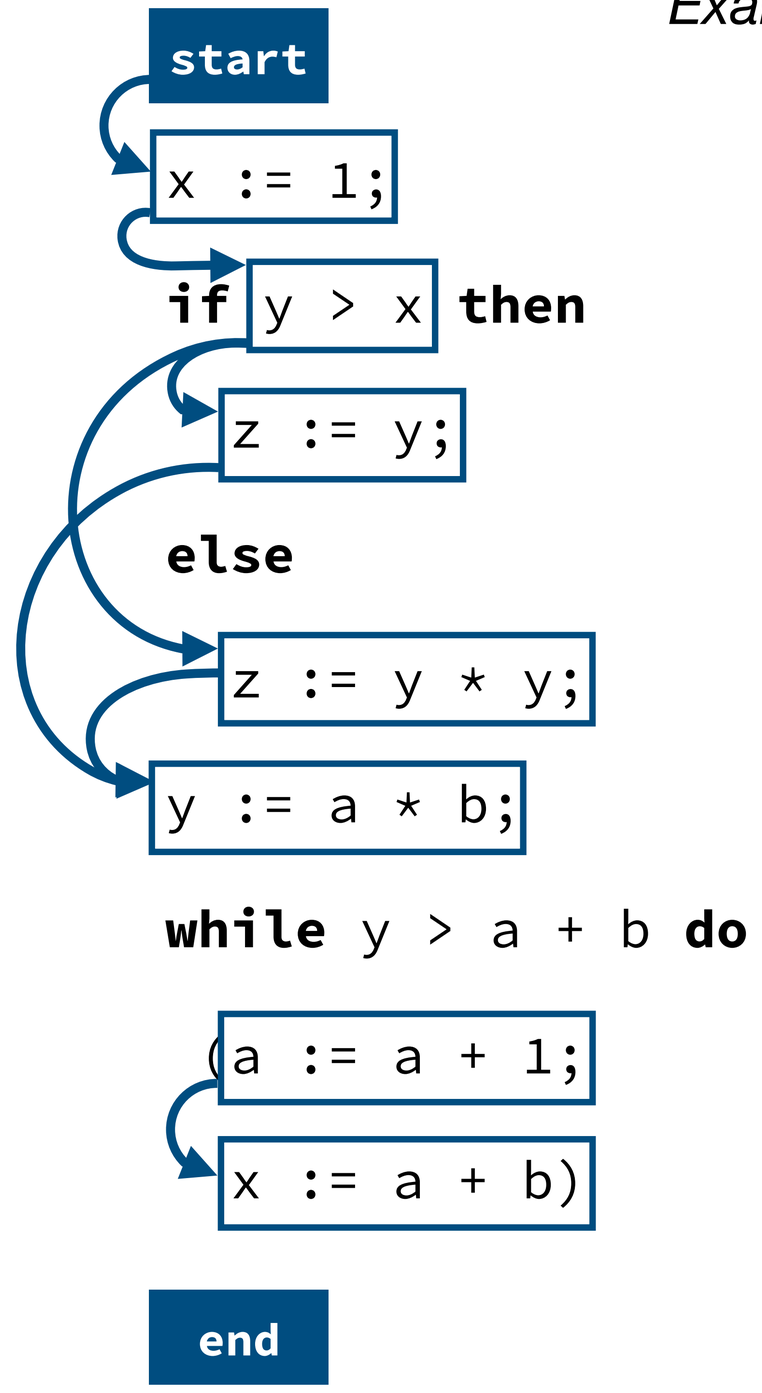


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
           c -> cfg e -> exit

cfg While(c, b) =
  entry -> c -> cfg b -> c -> exit
```

Example program

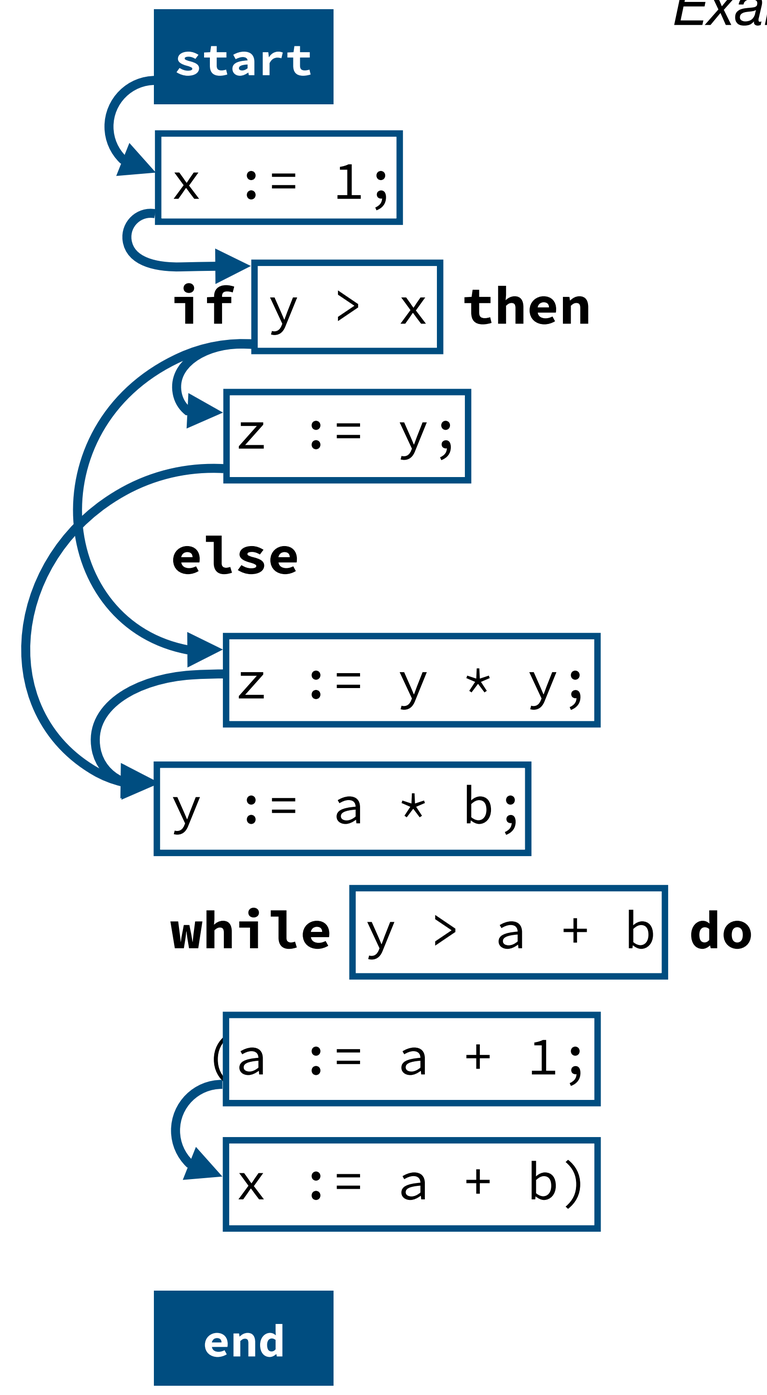


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
           c -> cfg e -> exit

cfg While(c, b) =
  entry -> c -> cfg b -> c -> exit
```

Example program

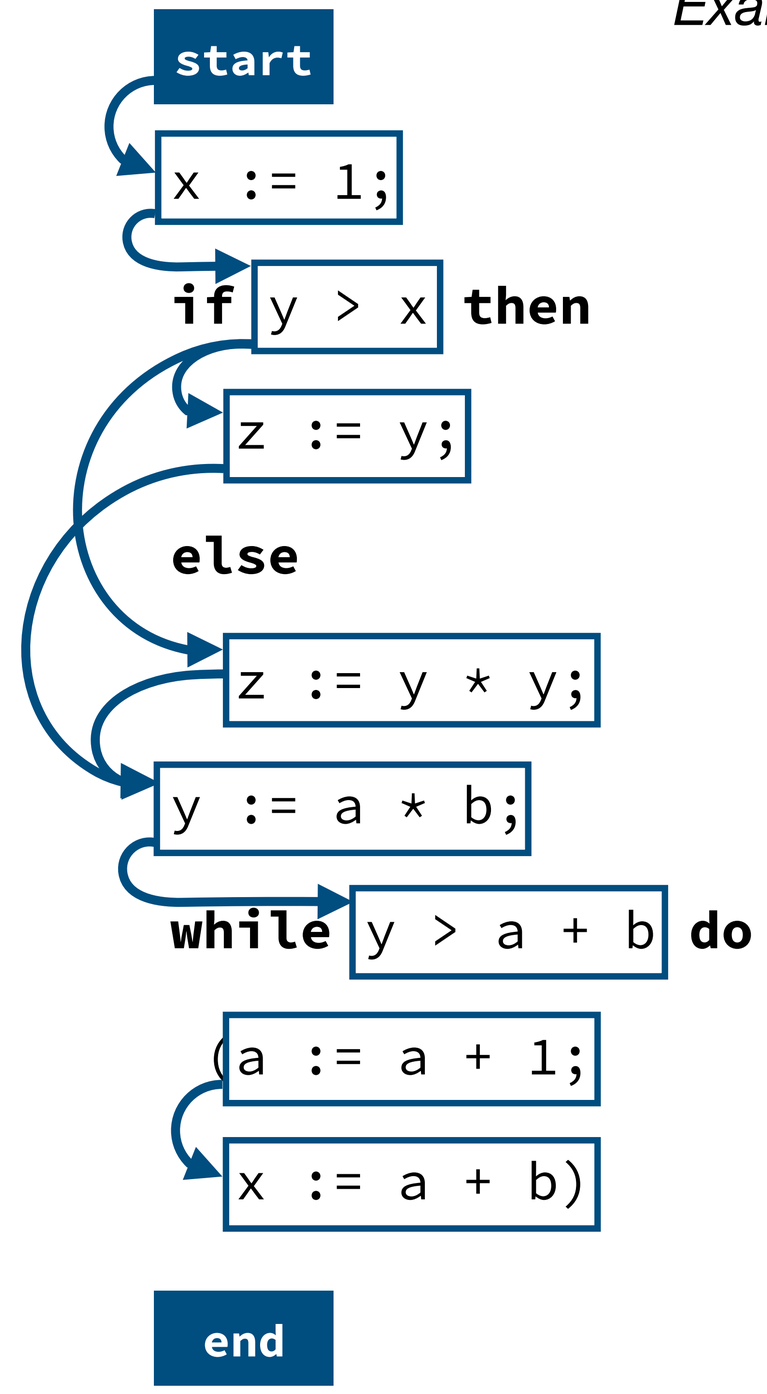


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
           c -> cfg e -> exit

cfg While(c, b) =
  entry -> c -> cfg b -> c -> exit
```

Example program

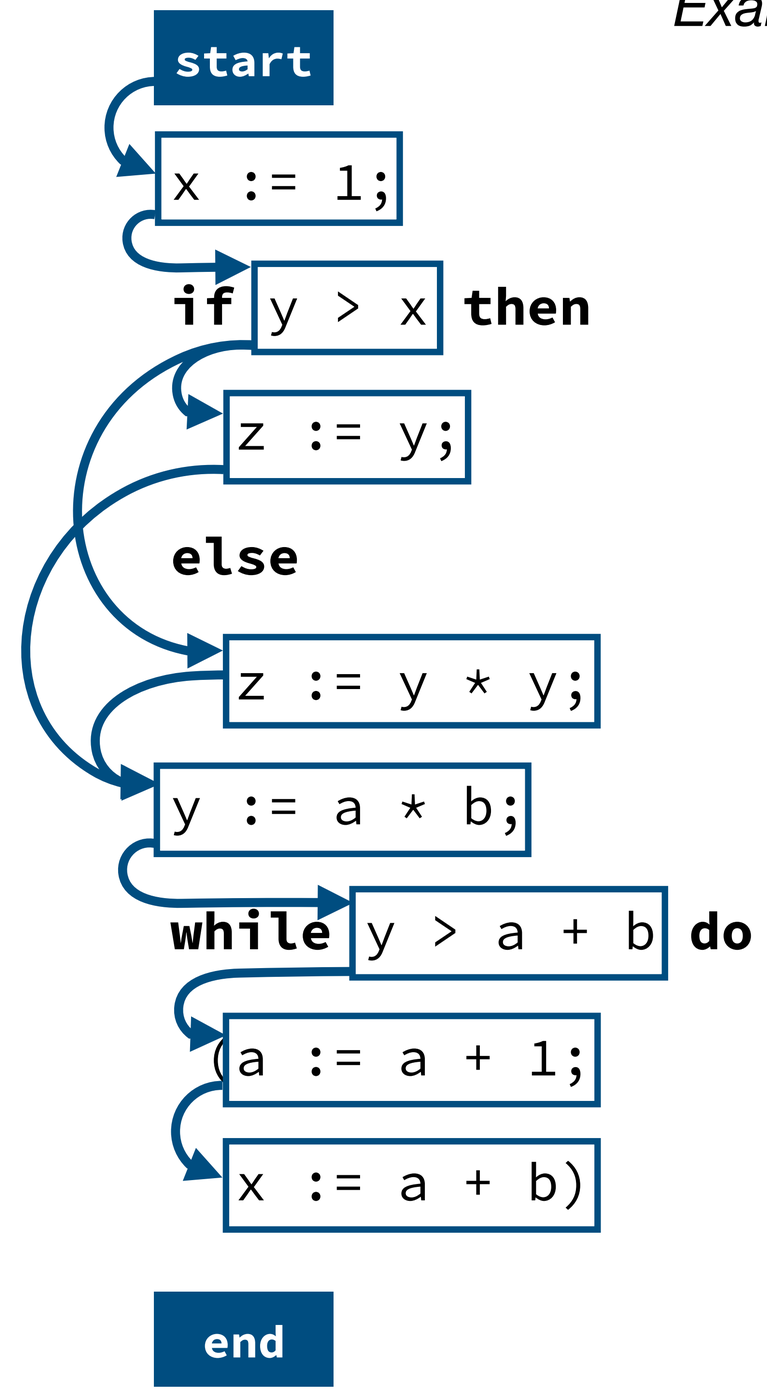


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
           c -> cfg e -> exit

cfg While(c, b) =
  entry -> c -> cfg b -> c -> exit
```

Example program

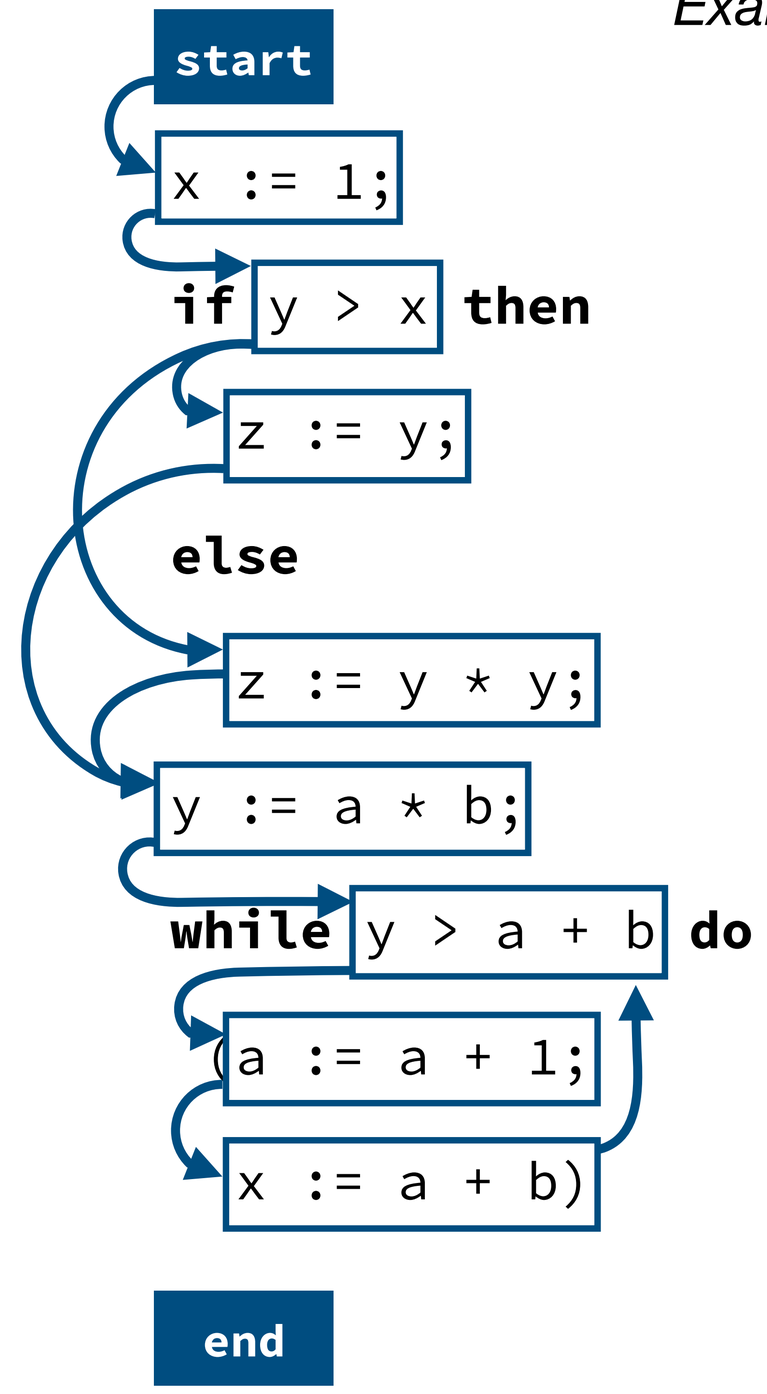


Control-flow graphs in FlowSpec

FlowSpec

```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
           c -> cfg e -> exit

cfg While(c, b) =
  entry -> c -> cfg b -> c -> exit
```

Example program

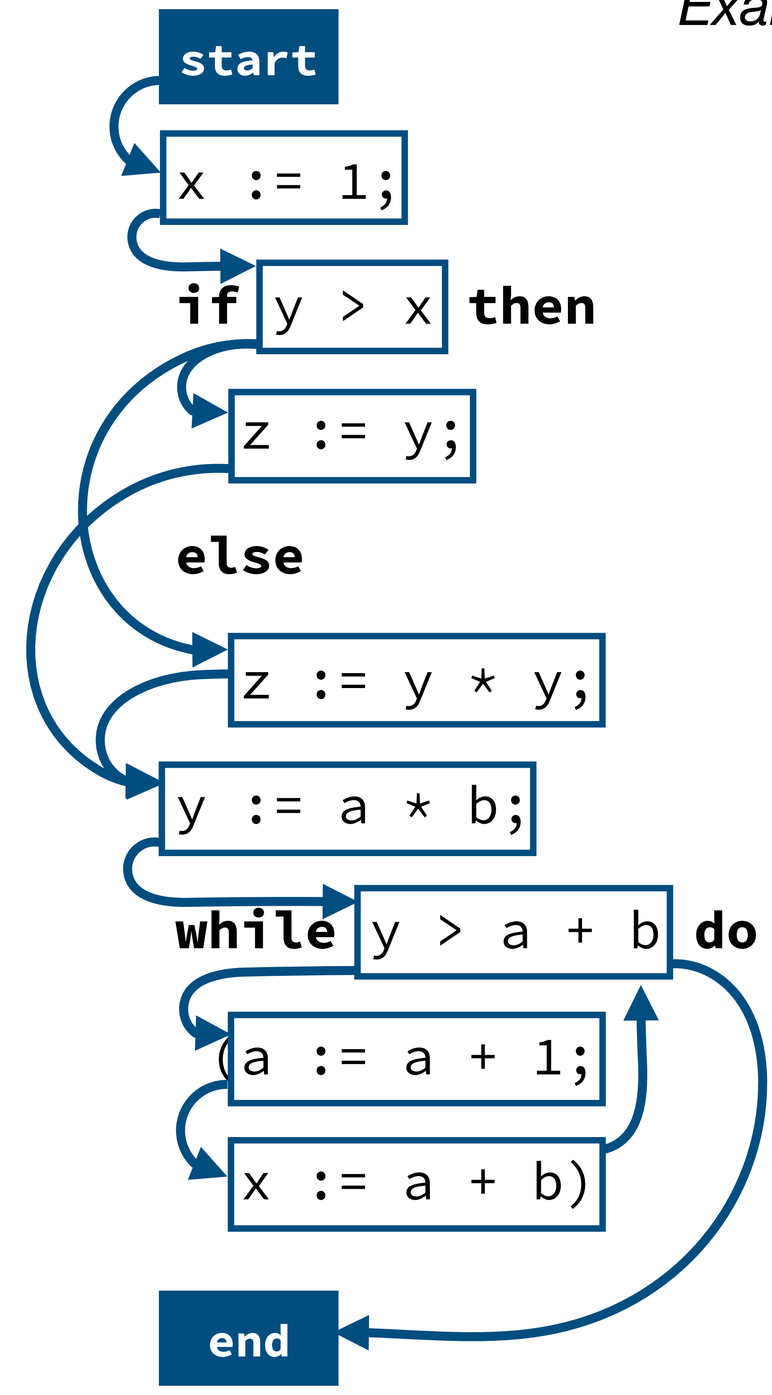


Control-flow graphs in FlowSpec

FlowSpec

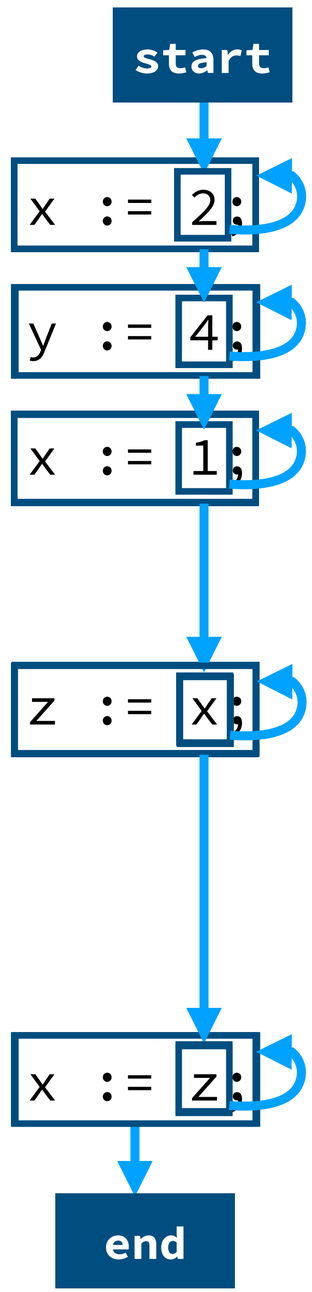
```
cfg root Mod(s) =
  start -> cfg s -> end

cfg a@Assign(_, _) =
  entry -> a -> exit

cfg Seq(s1, s2) =
  entry -> cfg s1 -> cfg s2 -> exit

cfg IfThenElse(c, t, e) =
  entry -> c -> cfg t -> exit,
           c -> cfg e -> exit

cfg While(c, b) =
  entry -> c -> cfg b -> c -> exit
```

Example program



Live Variables in FlowSpec

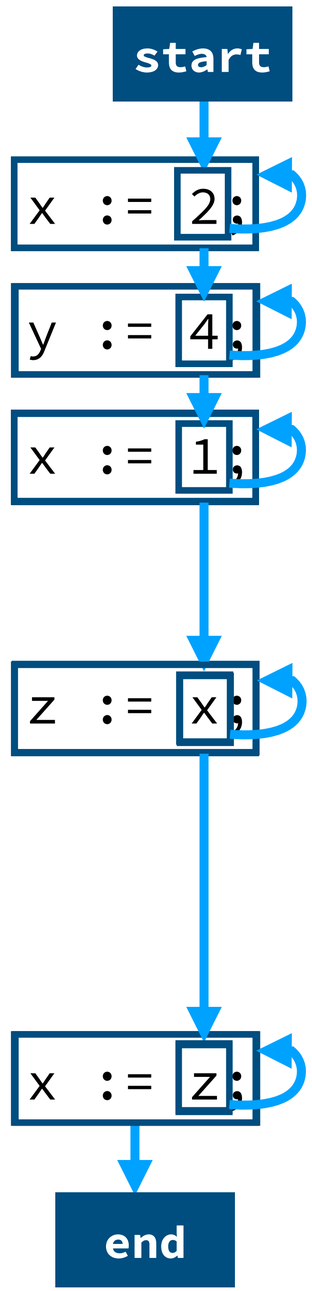
A variable is *live* if the current value of the variable *may* be read further along in the program



Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:     Set(name)
```

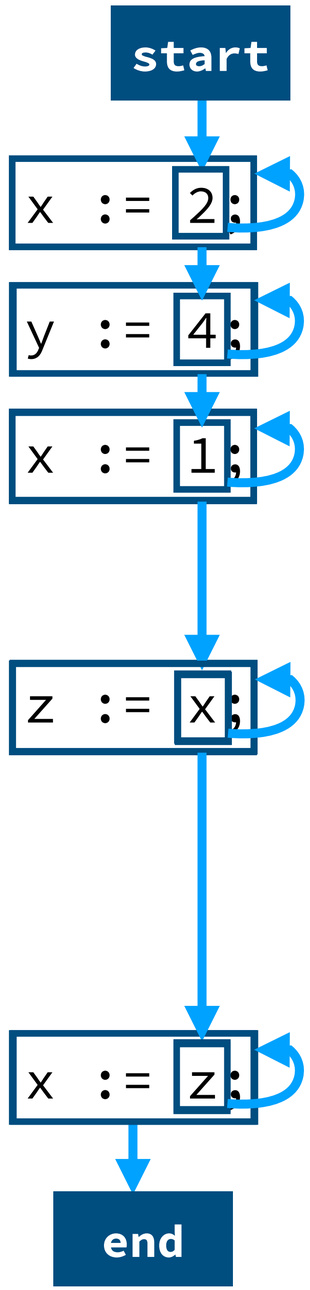


Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:     Set(name)
```

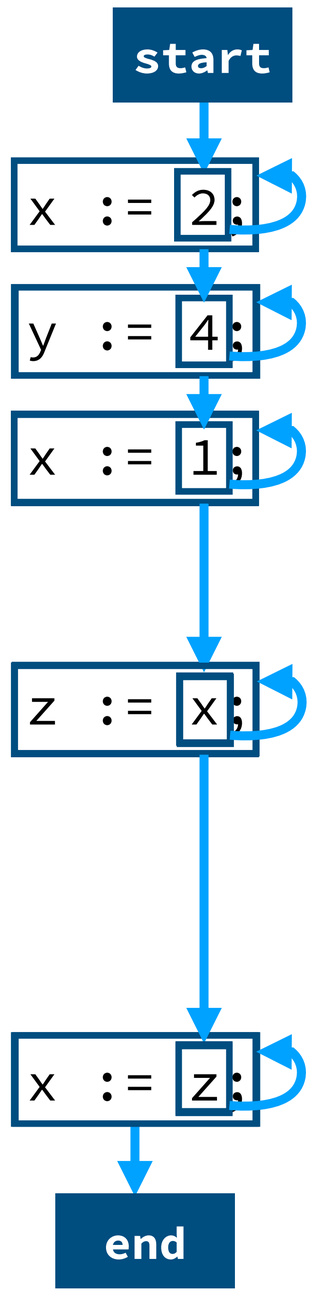
```
live(end) =
  {}

live(_ -> next) =
  live(next)
```



Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:     Set(name)

live(end) =
  {}

live(_ -> next) =
  live(next)
```



{}

{}

{}

{}

{}

{}

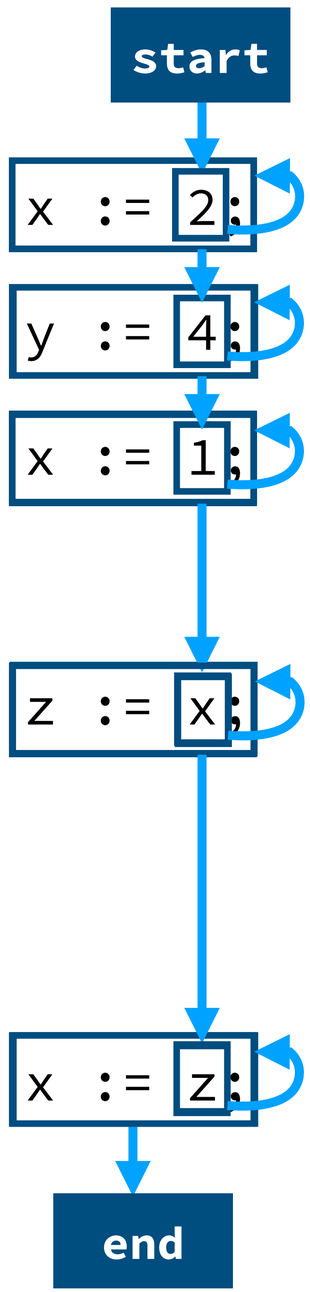
{}

{}

Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:     Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(end) =
  {}

live(_ -> next) =
  live(next)
```



{}

{}

{}

{}

{}

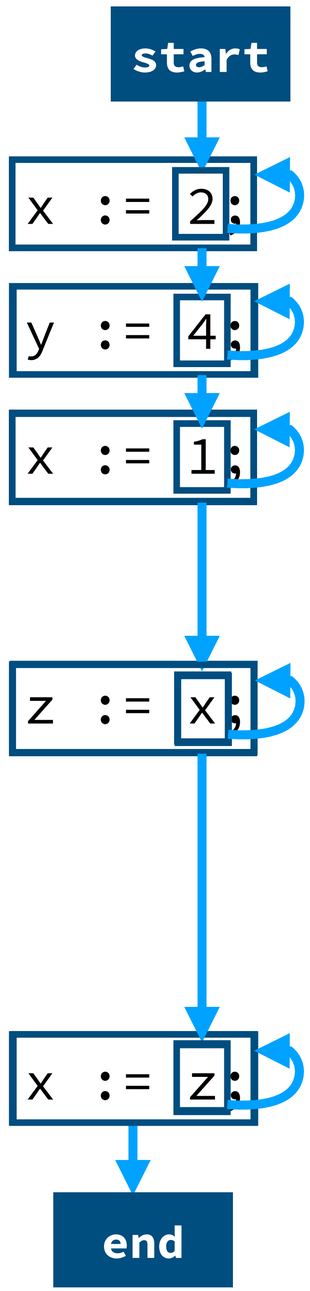
{}

{}

Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

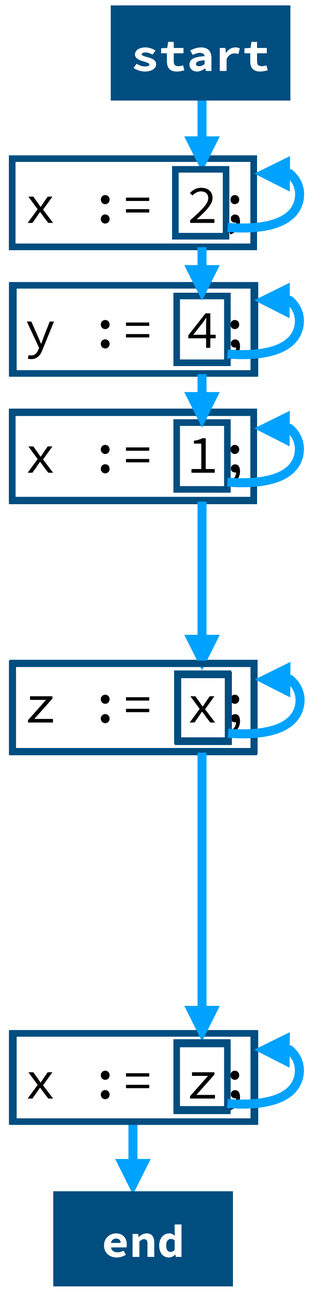
```
prop live:     Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(end) =
  {}

live(_ -> next) =
  live(next)
```



Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:     Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(end) =
  {}

live(_ -> next) =
  live(next)
```

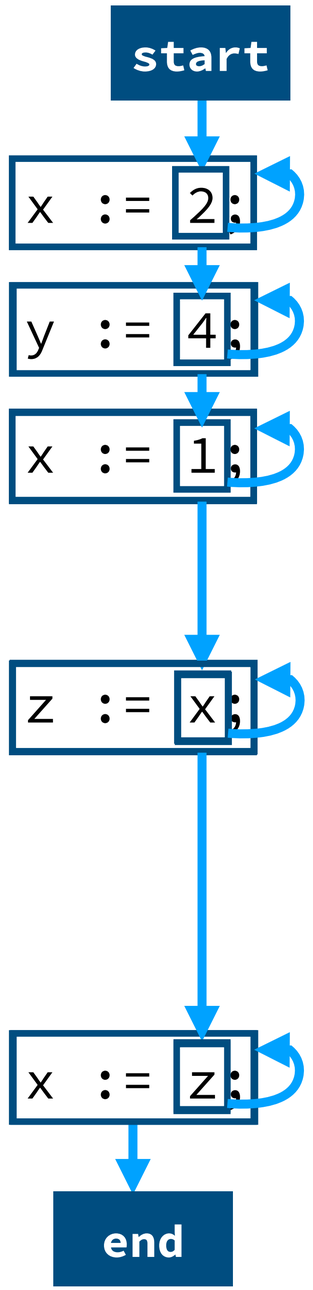



```
{z,x}
{z,x}
{z,x}
{z,x}
{z,x}
{z}
{}
{}
```

Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:     Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```



{z,x}

{z,x}

{z,x}

{z,x}

{z,x}

{z}

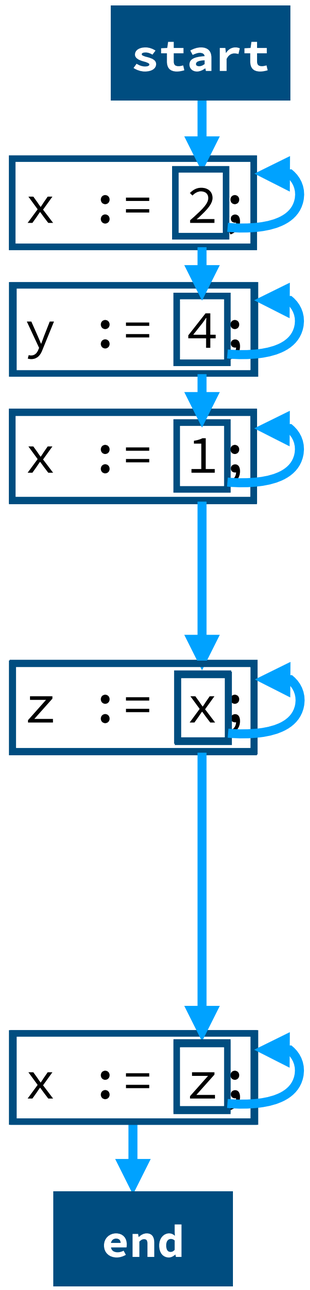
{}

{}

Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

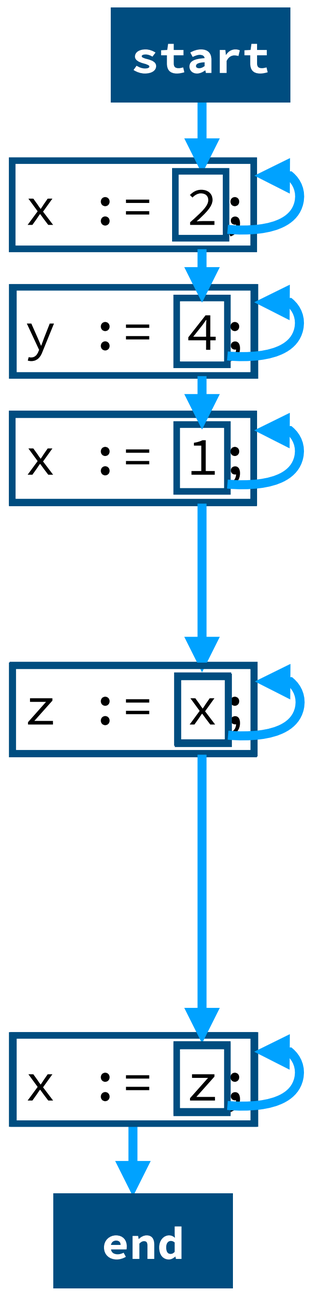
```
prop live:     Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```



Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:     Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```



{}

{}

{}

{x}

{z}

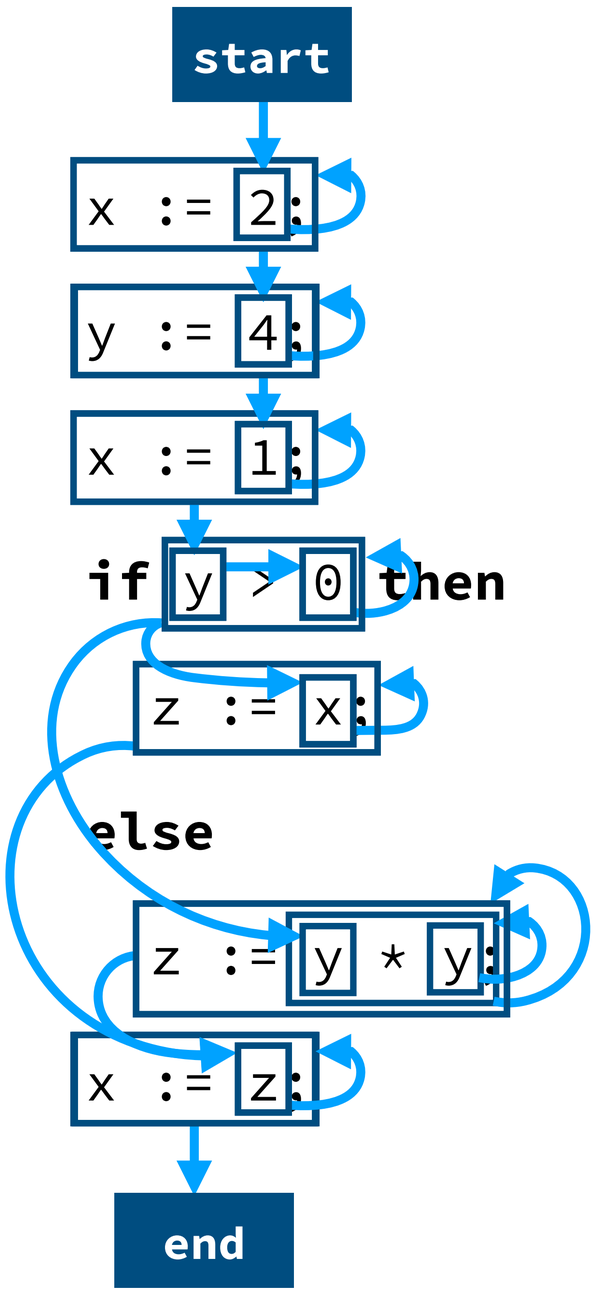
{}

{}

Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:     Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```



Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:    Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```

```
x := 2;

y := 4;

x := 1;

if y > 0 then

  z := x;

else

  z := y * y;

x := z;
```

Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

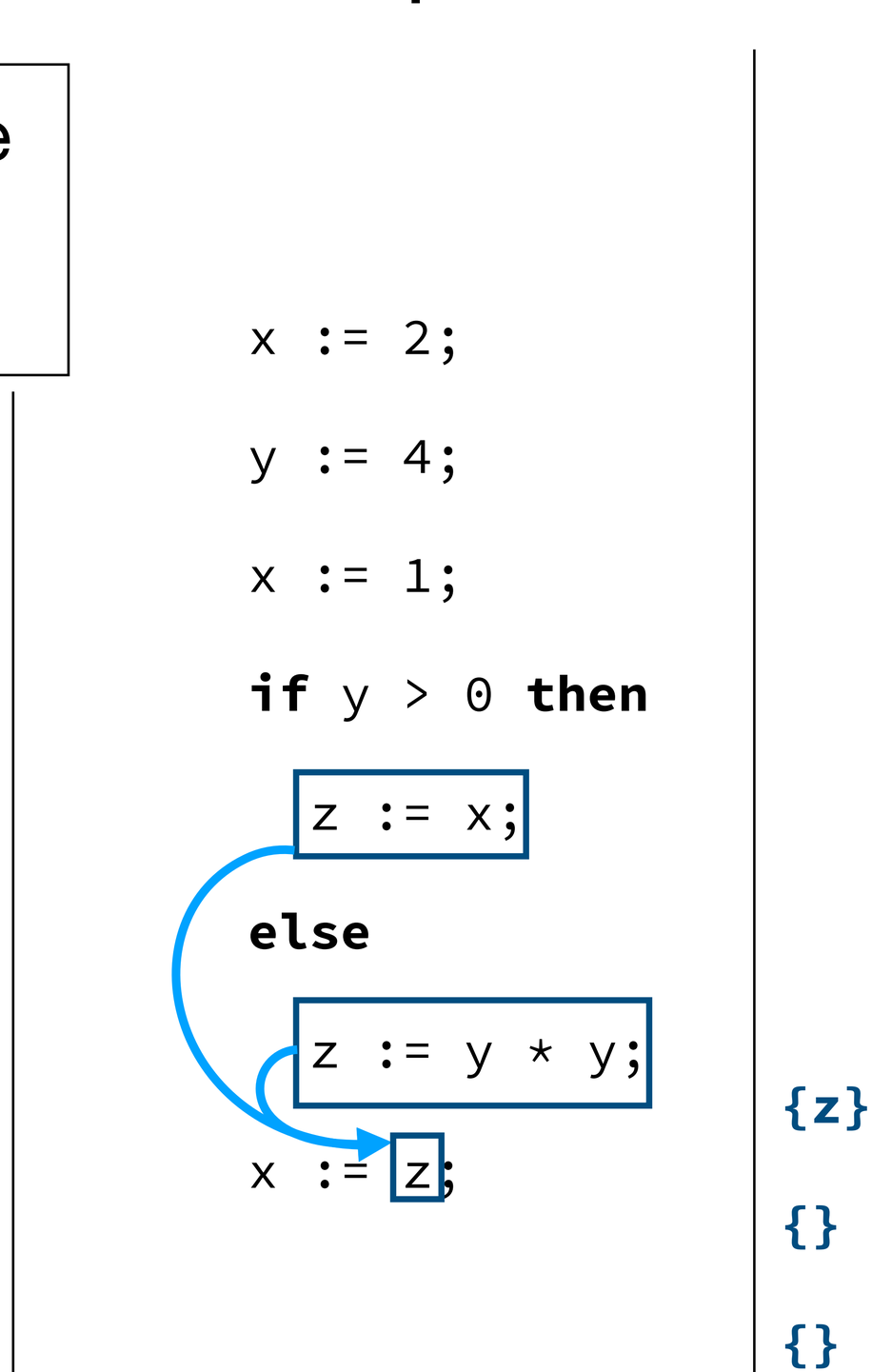
```
prop live:    Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```

```
x := 2;

y := 4;

x := 1;

if y > 0 then

  z := x;

else

  z := y * y;

x := z;
```

Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live:    Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```



Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

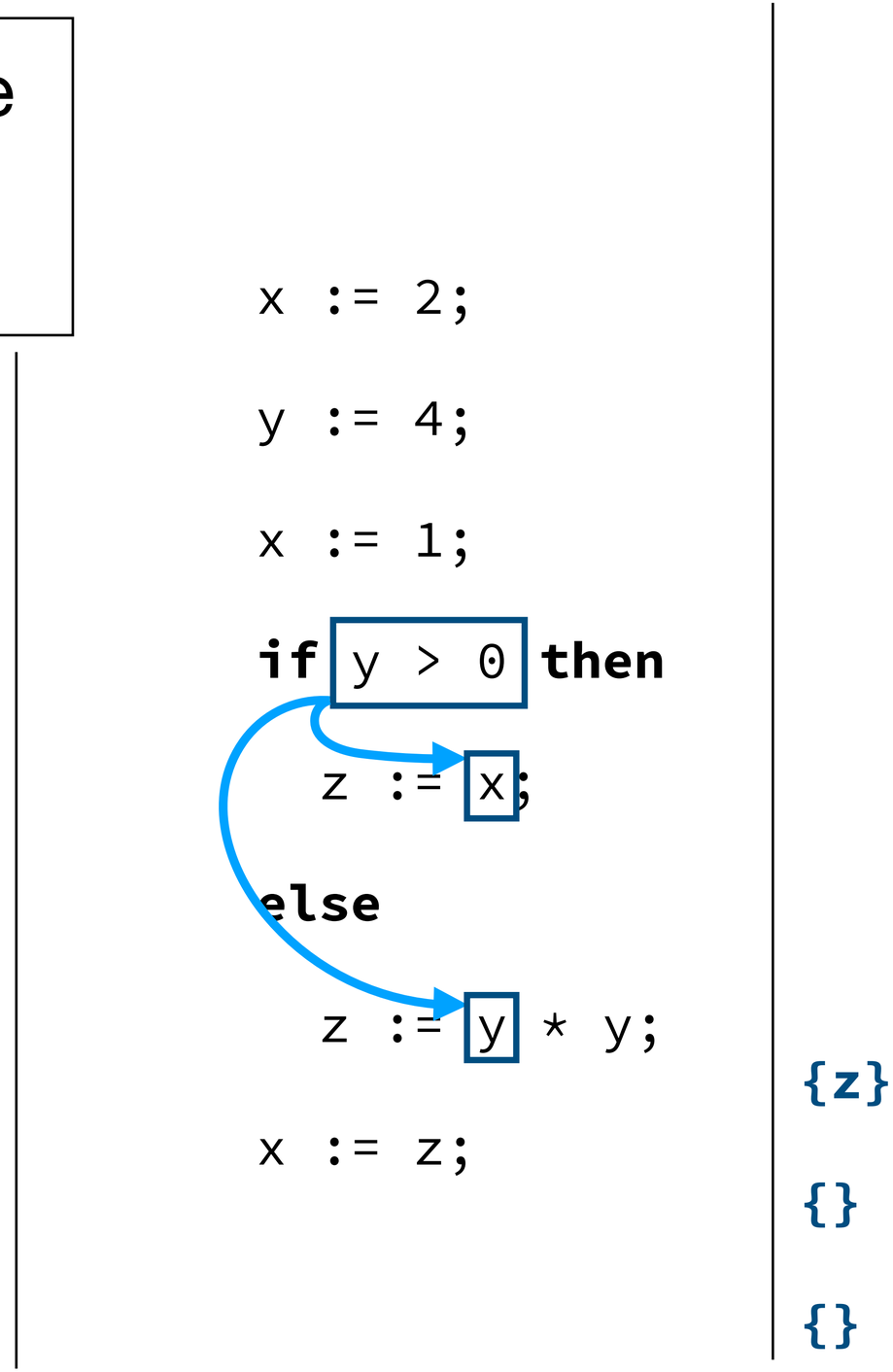
```
prop live:    Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```

```
x := 2;                  {}

y := 4;                  {}

x := 1;                  {}

if y > 0 then            {}

  z := x;                {}

else                     {}

  z := y * y;            {z}

x := z;                  {}
```

Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

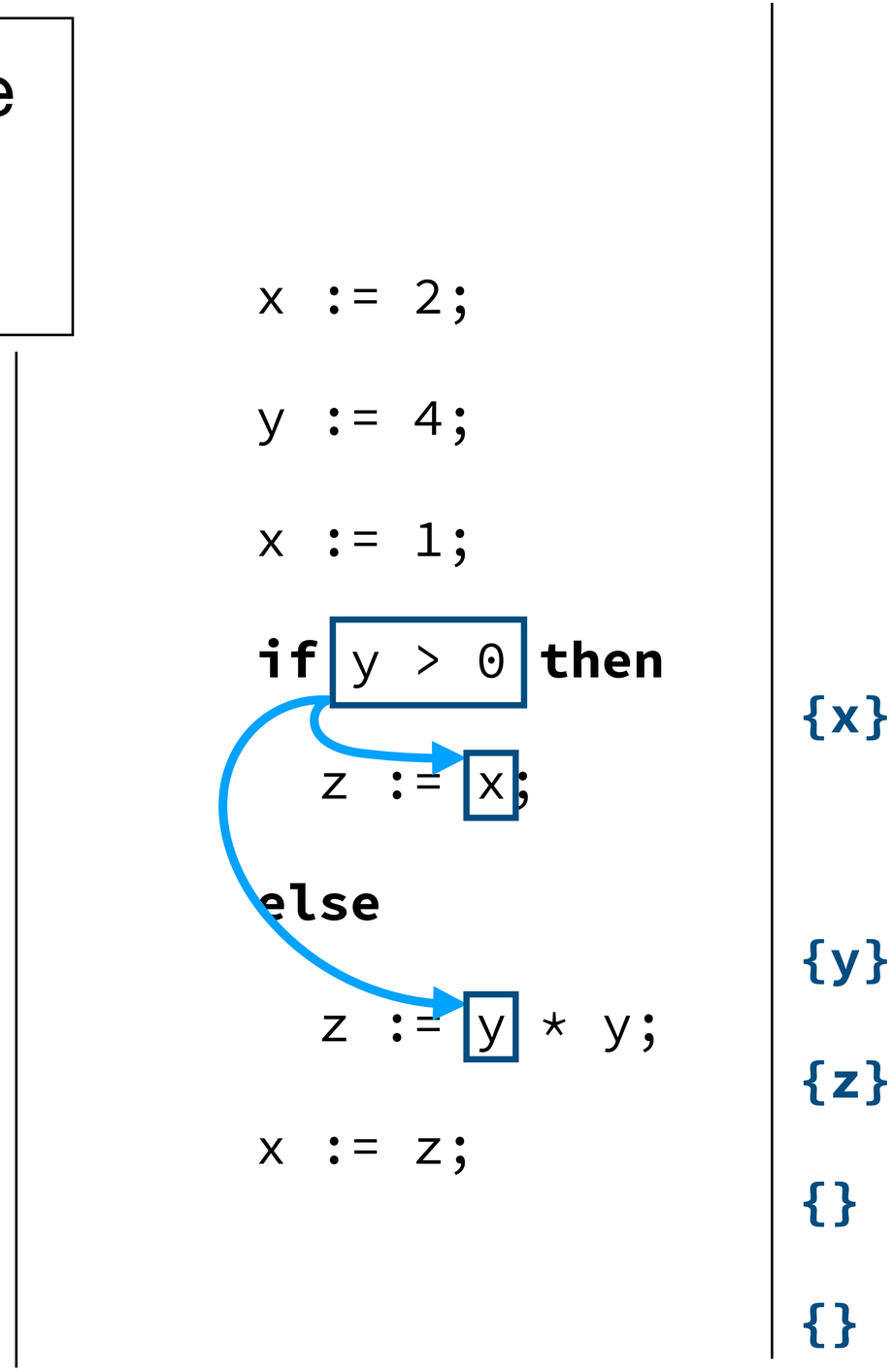
```
prop live:    Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```



Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

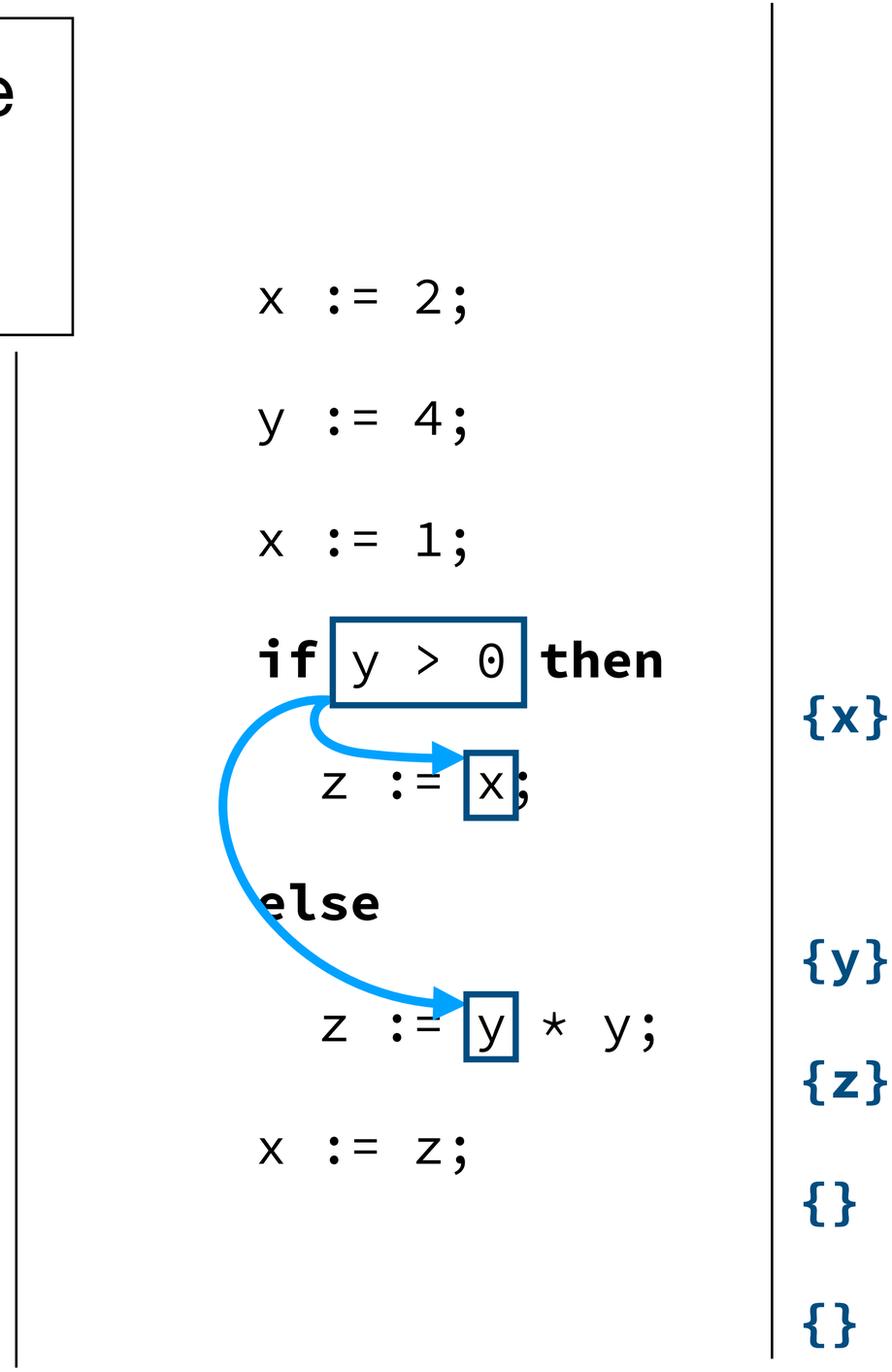
```
prop live:    Set(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```



Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

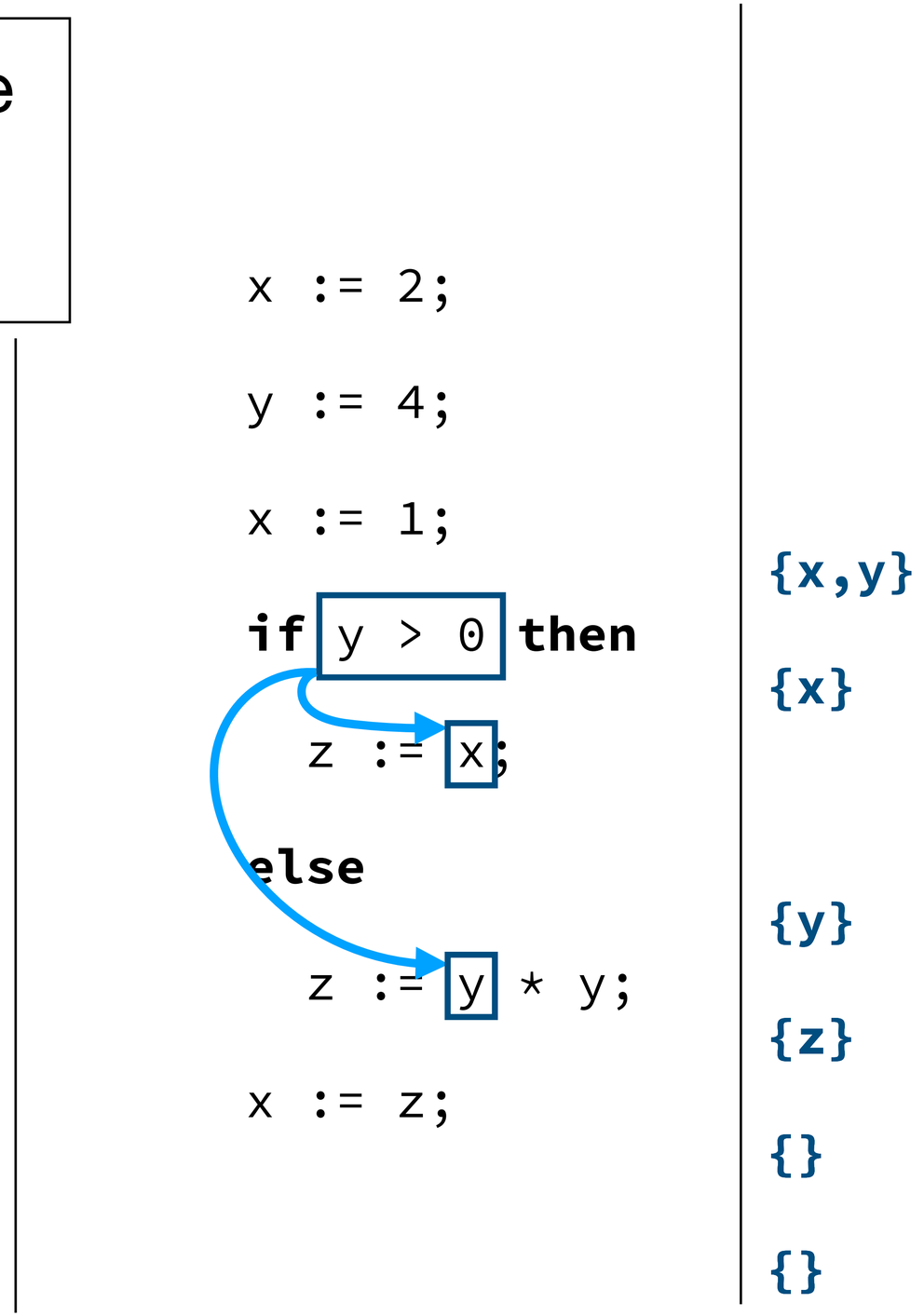
```
prop live: MaySet(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```



Live Variables in FlowSpec

A variable is *live* if the current value of the variable *may* be read further along in the program

```
prop live: MaySet(name)

live(Ref(n) -> next) =
  live(next) \/ {n}

live(Assign(n, _) -> next) =
  { m | m <- live(next), n != m }

live(end) =
  {}

live(_ -> next) =
  live(next)
```



Closing

But wait! There’s more!

But wait! There's more!

Context-sensitivity

But wait! There's more!

Context-sensitivity

- Follow function calls / where were we called from?

But wait! There's more!

Context-sensitivity

- Follow function calls / where were we called from?
- Dynamic dispatch: where did objects come from?

But wait! There's more!

Context-sensitivity

- Follow function calls / where were we called from?
- Dynamic dispatch: where did objects come from?
- Control-flow depends on data-flow depends on control-flow…

But wait! There's more!

Context-sensitivity

- Follow function calls / where were we called from?
- Dynamic dispatch: where did objects come from?
- Control-flow depends on data-flow depends on control-flow…

Incrementality

But wait! There's more!

Context-sensitivity

- Follow function calls / where were we called from?
- Dynamic dispatch: where did objects come from?
- Control-flow depends on data-flow depends on control-flow…

Incrementality

- Optimisation: transform the program into a more efficient one

But wait! There's more!

Context-sensitivity

- Follow function calls / where were we called from?
- Dynamic dispatch: where did objects come from?
- Control-flow depends on data-flow depends on control-flow…

Incrementality

- Optimisation: transform the program into a more efficient one
- User changes part of the program (typing, or refactoring)

But wait! There's more!

Context-sensitivity

- Follow function calls / where were we called from?
- Dynamic dispatch: where did objects come from?
- Control-flow depends on data-flow depends on control-flow…

Incrementality

- Optimisation: transform the program into a more efficient one
- User changes part of the program (typing, or refactoring)
- Analysis results are *partially* invalid now

But wait! There's more!

Context-sensitivity

- Follow function calls / where were we called from?
- Dynamic dispatch: where did objects come from?
- Control-flow depends on data-flow depends on control-flow…

Incrementality

- Optimisation: transform the program into a more efficient one
- User changes part of the program (typing, or refactoring)
- Analysis results are *partially* invalid now

External PhD candidate Tamás Szabó researches incremental static analysis under supervision of Sebastian Erdweg

But wait! There’s more!

But wait! There's more!

Correctness

But wait! There's more!

Correctness

- Are you sure your analysis is correct?

But wait! There's more!

Correctness

- Are you sure your analysis is correct?
- Are the results guaranteed to safely approximate the dynamic semantics?

But wait! There's more!

Correctness

- Are you sure your analysis is correct?
- Are the results guaranteed to safely approximate the dynamic semantics?
- Abstract Interpretation

But wait! There's more!

Correctness

- Are you sure your analysis is correct?
- Are the results guaranteed to safely approximate the dynamic semantics?
- Abstract Interpretation
  - Principled transformation of dynamic semantics into static approximations

But wait! There's more!

Correctness

- Are you sure your analysis is correct?
- Are the results guaranteed to safely approximate the dynamic semantics?
- Abstract Interpretation
  - Principled transformation of dynamic semantics into static approximations
  - Galois connections

But wait! There's more!

Correctness

- Are you sure your analysis is correct?
- Are the results guaranteed to safely approximate the dynamic semantics?
- Abstract Interpretation
  - Principled transformation of dynamic semantics into static approximations
  - Galois connections

PhD candidate Sven Keidel (HB08.260) researches abstract interpretation under supervision of Sebastian Erdweg

But wait! There’s more!

But wait! There's more!

Correctness (2)

- Are you sure your optimisations are correct?
- Are the results guaranteed to have the same semantics?
- Can you generate a proof to show this property?
  - Perhaps using parameterised program equivalence
  - You might leverage scope graphs in your model!

But wait! There's more!

Correctness (2)

- Are you sure your optimisations are correct?
- Are the results guaranteed to have the same semantics?
- Can you generate a proof to show this property?
  - Perhaps using parameterised program equivalence
  - You might leverage scope graphs in your model!

PhD candidate Arjen Rouvoet (HB08.260) researches optimisation correctness under supervision of Eelco Visser

Future Work

A running FlowSpec prototype

- So, no, you can't try out FlowSpec yet
- Almost done though, promise!

Evaluation of FlowSpec

- Is it powerful enough to capture everything we want?

Expand the control flow model

- Unordered control flow in expressions
- Parallel control flow for concurrent/parallel loops

Path-sensitive data-flow analysis

- Learning facts about conditions in conditional control flow

Supporting mechanised language meta-theory

- Relating static analysis and dynamic semantics

Summary: Data-Flow Analysis

Representation: Control Flow Graphs + Data Flow information

- Standardised representation for control flow
- Annotate with information from analysis

Formalism: Lattices and Transfer Functions

- Captures the type of information
- How to combine that information
- How to check that the analysis terminates

Limitations: Intra-procedural, context-insensitive

- So no function/method calls
- No tracking of objects or pointers for dynamic dispatch
- But at least we are *flow-sensitive*

Separation of Concerns

Representation

- **Control Flow Graphs**
- **Before/After data-flow information on CFG nodes**

Declarative Rules

- To define control-flow rules of a language
- To define data-flow rules of a language

Language-Independent Tooling

- Analysis
- Code completion
- Refactoring
- Optimisation
- …

Separation of Concerns

Representation

- Control Flow Graphs
- Before/After data-flow information on CFG nodes

Declarative Rules

- FlowSpec

Language-Independent Tooling

- Analysis
- Code completion
- Refactoring
- Optimisation
- …