

# Data-Flow Analysis

**Jeff Smits & Eelco Visser**



**CS4200 | Compiler Construction | December 16 & 23, 2021**

# Reading Material

The following papers add background, conceptual exposition, and examples to the material from the slides. Some notation and technical details have been changed; check the documentation.

This paper introduces FlowSpec, the declarative data-flow analysis specification language in Spooftax. Although the design of the language described in this paper is still current, the syntax used is already dated, i.e. the current FlowSpec syntax in Spooftax is slightly different.

SLE 2017

<https://doi.org/10.1145/3136014.3136029>

# FlowSPEC: Declarative Dataflow Analysis Specification

Jeff Smits  
TU Delft  
The Netherlands  
j.smits-1@tudelft.nl

Eelco Visser  
TU Delft  
The Netherlands  
e.visser@tudelft.nl

## Abstract

We present FlowSPEC, a declarative specification language for the domain of dataflow analysis. FlowSPEC has declarative support for the specification of control flow graphs of programming languages, and dataflow analyses on these control flow graphs. We define the formal semantics of FlowSPEC, which is rooted in Monotone Frameworks. We also discuss implementation techniques for the language, partly used in the prototype implementation built in the SPOOFAX Language Workbench. Finally, we evaluate the expressiveness and conciseness of the language with two case studies. These case studies are analyses for GREEN-MARL, an industrial, domain-specific language for graph processing. The first case study is a classical dataflow analysis, scaled to this full language. The second case study is a domain-specific analysis of GREEN-MARL.

**CCS Concepts** • Software and its engineering → Domain specific languages;

**Keywords** control flow graph, dataflow analysis

## ACM Reference Format:

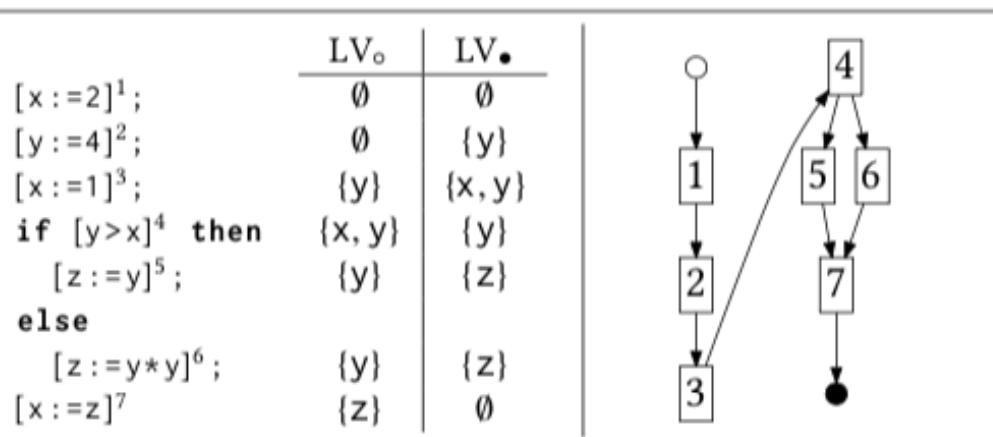
Jeff Smits and Eelco Visser. 2017. FlowSPEC: Declarative Dataflow Analysis Specification. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17)*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3136014.3136029>

## 1 Introduction

Dataflow analysis is a static analysis that answers questions on what *may* or *must* happen before or after a certain point in a program's execution. With dataflow analysis we can answer whether a value written to a variable *here* may be

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. *SLE'17, October 23–24, 2017, Vancouver, Canada*

© 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.  
ACM ISBN 978-1-4503-5525-4/17/10...\$15.00  
<https://doi.org/10.1145/3136014.3136029>



**Figure 1.** Classical dataflow analysis Live Variables (LV). On the left is an example program in the WHILE language, with added brackets to number program fragments. On the right is the control flow graph (CFG) of the program. In the centre is the analysis result. The LV<sub>o</sub> and LV<sub>•</sub> are before and after the CFG node's variables accesses respectively.

read *later*. Such dataflow analyses can be used to inform optimisations.

For example, consider Live Variables analysis, illustrated in Figure 1. This type of dataflow analysis can identify dead code, which can be removed as an optimisation. In the example this would be statement 1 since it writes x which is overwritten by statement 3 without being read in between. The Live Variables analysis provides a set of variables which are read before being written after each statement in LV<sub>•</sub>. The figure shows this in the LV<sub>•</sub> set of statement 1, which does not contain x.

Dataflow may also be part of a language's static semantics. For example, in Java a final field in a class must be initialised by the end of construction of an object of that class. Since constructor code can have conditional control flow, a dataflow analysis is necessary to check that all possible execution paths through constructors actually assign a value to the final field [Gosling et al. 2005, sect. 16.9].

Dataflow analyses are often operationally encoded, whether in a general purpose language, an attribute grammar system or a logic programming language. This encoding is both an overhead for the engineer implementing it, as well as an overhead in decoding for anyone who wishes to understand the analysis.

In formal, mathematical descriptions of a dataflow analysis, the common patterns are often factored out. This shows commonalities between different analyses, allows the study of those commonalities and differences, as well as general

# Journal version of the SLE paper.

This paper introduces FlowSpec, the declarative data-flow analysis specification language in Spoofax.

Journal of Computer Languages 2020

<https://doi.org/10.1016/j.col.2019.100924>

Journal of Computer Languages 57 (2020) 100924

Contents lists available at ScienceDirect



Journal of Computer Languages

journal homepage: [www.editorialmanager.com/cola/default.aspx](http://www.editorialmanager.com/cola/default.aspx)



## FLOWSPEC: A declarative specification language for intra-procedural flow-Sensitive data-flow analysis

Jeff Smits<sup>a,\*</sup>, Guido Wachsmuth<sup>b</sup>, Eelco Visser<sup>a</sup>

<sup>a</sup> Programming Languages Research Group, Delft University of Technology, Van Mourik Broekmanweg 6, XE Delft 2628, the Netherlands

<sup>b</sup> Oracle Labs, Prime Tower, Floor 17, Hardstrasse 201, Zürich 8005, Switzerland

### HIGHLIGHTS

- Data-flow analysis is the static analysis of programs to estimate their approximate run-time behavior or approximate intermediate run-time values. It is an integral part of modern language specifications and compilers. In the specification of static semantics of programming languages, the concept of data-flow allows the description of well-formedness such as definite assignment of a local variable before its first use. In the implementation of compiler back-ends, data-flow analyses inform optimizations.
- Data-flow analysis has an established theoretical foundation. What lags behind is implementations of data-flow analysis in compilers, which are usually ad-hoc. This makes such implementations difficult to extend and maintain. In previous work researchers have proposed higher-level formalisms suitable for whole-program analysis in a separate tool, incremental analysis within editors, or bound to a specific intermediate representation.
- In this paper, we present FlowSpec, an executable formalism for specification of data-flow analysis. FlowSpec is a domain-specific language that enables direct and concise specification of data-flow analysis for programming languages, designed to express flow-sensitive, intra-procedural analyses.
- We define the formal semantics of FlowSpec in terms of monotone frameworks. We describe the design of FlowSpec using examples of standard analyses. We also include a description of our implementation of FlowSpec.
- In a case study we evaluate FlowSpec with the static analyses for GreenMarl, a domain-specific programming language for graph analytics.

### ARTICLE INFO

MSC:  
68N15

### ABSTRACT

Data-flow analysis is the static analysis of programs to estimate their approximate run-time behavior or approximate intermediate run-time values. It is an integral part of modern language specifications and compilers. In the specification of static semantics of programming languages, the concept of data-flow allows the description of well-formedness such as definite assignment of a local variable before its first use. In the implementation of compiler back-ends, data-flow analyses inform optimizations.

Data-flow analysis has an established theoretical foundation. What lags behind is implementations of data-flow analysis in compilers, which are usually ad-hoc. This makes such implementations difficult to extend and maintain. In previous work researchers have proposed higher-level formalisms suitable for whole-program analysis in a separate tool, incremental analysis within editors, or bound to a specific intermediate representation.

In this paper, we present FLOWSPEC, an executable formalism for specification of data-flow analysis. FLOWSPEC is a domain-specific language that enables direct and concise specification of data-flow analysis for programming languages, designed to express flow-sensitive, intra-procedural analyses. We define the formal semantics of FLOWSPEC in terms of monotone frameworks. We describe the design of FLOWSPEC using examples of standard analyses. We also include a description of our implementation of FLOWSPEC.

In a case study we evaluate FLOWSPEC with the static analyses for GREEN-MARL, a domain-specific programming language for graph analytics.

\* Corresponding author.

E-mail addresses: [j.smits-1@tudelft.nl](mailto:j.smits-1@tudelft.nl) (J. Smits), [guido.wachsmuth@oracle.com](mailto:guido.wachsmuth@oracle.com) (G. Wachsmuth), [e.visser@tudelft.nl](mailto:e.visser@tudelft.nl) (E. Visser).

<https://doi.org/10.1016/j.col.2019.100924>

Received 11 August 2019; Accepted 20 September 2019

Available online 23 November 2019

2590-1184/ © 2019 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

# Documentation for FlowSpec at the metaborg.org website.

<http://www.metaborg.org/en/latest/source/langdev/meta/lang/flowspec/index.html>

The screenshot shows the official documentation for Spoofax. The sidebar includes links for 'The Spoofax Language Workbench', 'Examples', 'Publications', 'TUTORIALS' (which is currently selected), 'REFERENCE MANUAL', and 'Data-Flow Analysis with FlowSpec'. The main content area displays a table of contents for 'Data-Flow Analysis with FlowSpec' with sections like '1. Introduction', '2. Language Reference', '3. Stratego API', etc. The footer contains links for 'Read the Docs' and 'v: latest'.

Docs » Data Flow Analysis Definition with FlowSpec

[Edit on GitHub](#)

## Data Flow Analysis Definition with FlowSpec

Programs that are syntactically well-formed are not necessarily valid programs. Programming languages typically impose additional *context-sensitive* requirements on programs that cannot be captured in a syntax definition. Languages use data and control flow to check certain extra properties that fall outside of names and type systems. The FlowSpec ‘Flow Analysis Specification Language’ supports the specification of rules to define the static control flow of a language, and data flow analysis over that control flow. FlowSpec supports flow-sensitive intra-procedural data flow analysis.

### Table of Contents

- [1. Introduction](#)
  - [1.1. Control Flow Graphs](#)
  - [1.2. Data Flow Analysis over Control Flow Graphs](#)
- [2. Language Reference](#)
  - [2.1. Lexical matters](#)
  - [2.2. Terms and patterns](#)
  - [2.3. Modules](#)
  - [2.4. Control Flow](#)
  - [2.5. Data Flow](#)
  - [2.6. Lattices](#)
  - [2.7. Types](#)
  - [2.8. Expressions](#)
  - [2.9. Functions](#)
- [3. Stratego API](#)
  - [3.1. Setup](#)
  - [3.2. Running the analysis](#)
  - [3.3. Querying analysis](#)
  - [3.4. Hover text](#)
  - [3.5. Profiling information](#)
- [4. Configuration](#)
  - [4.1. Prepare your project](#)
  - [4.2. Inspecting analysis results](#)

# Data-Flow Analysis

# What is Data-Flow Analysis?

## Static approximation of runtime behaviour

- What has or will be computed
- What extra invariants do some data adhere to
- Data dependence between data/variables where the data lives

# Available Expressions

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

- $a + b$  is already computed when you get to the condition
- There is no need to compute it again

# Live Variables

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

The first value of x is never observed,  
because it isn't read after the assignment

# What is Data-Flow Analysis?

## Static approximation of runtime behaviour

- What has or will be computed
- What extra invariants do some data adhere to
- Data dependence between data/variables where the data lives

# Flow-Sensitive Types

```
void hello(String? name) {  
    if (is String name) {  
        // name is of type String here  
        print("Hello, ``name``!");  
    }  
    else {  
        print("Hello, world!");  
    }  
}
```

- Ceylon (<https://ceylon-lang.org/>)
- Union and intersection types
- **String?** = String | Null
- **is** like Java's **instanceof**
- General name: path-sensitive data-flow analysis

# What is Data-Flow Analysis?

## Static approximation of runtime behaviour

- What has or will be computed
- What extra invariants do some data adhere to
- Data dependence between data/variables where the data lives

# Reaching Definitions

```
let
  var x : int := 5
  var y : int := 1
in
  while x > 1 do
    (
      y := x * 2;
      x := y - 1
    )
end
```

- The inverse relation of live variables
- RD gives us the possible origins of the current value of a variable

# Reaching Definitions

```
1 let
2   var x : int := 5 _____ x ↦ 2
3   var y : int := 1 _____ x ↦ 2; y ↦ 3
4 in
5   while x > 1 do
6     (
7       y := x * 2; _____ x ↦ 2, 8; y ↦ 3, 7
8       x := y - 1 _____ x ↦ 2, 8; y ↦ 7
9     )
10 end
```

- Analysis result is a multi-map (shown here after each statement)
- Propagate information along the control-flow graph

# Reaching Definitions

```
1 let
2   var x : int := 5 _____ x ↦ 2   {(x,2)}
3   var y : int := 1 _____ x ↦ 2; y ↦ 3   {(x,2) (y,3)}
4 in
5   while x > 1 do
6     ( _____ x ↦ 2,8; y ↦ 3,7   {(x,2) (x,8) (y,3) (y,7)}
7     y := x * 2; _____ x ↦ 2,8; y ↦ 7   {(x,2) (x,8) (y,7)}
8     x := y - 1 _____ x ↦ 8; y ↦ 7   {(x,8) (y,7)}
9   )
10 end
```

- Analysis result is a set of pairs (shown here after each statement)
- Propagate information along the control-flow graph

# Control-Flow

## What is Control-Flow?

- “Order of evaluation”

## Discuss a series of example programs

- What is the control flow?
- What constructs in the program determine that?

# What is Control-Flow?

```
function id(x) { return x; }  
id(4); id(true);
```

## Function calls

- Calling a function passes control to that function
- A **return** passes control back to the caller

# What is Control-Flow?

```
if (c) { a = 5 } else { a = "four" }
```

Branching

- Control is passed to one of the two branches
- This is dependent on the value of the condition

# What is Control-Flow?

```
while (c) { a = 5 }
```

Looping

- Control is passed to the loop body depending on the condition
- After the body we start over

# What is Control-Flow?

```
function1(a);  
function2(b);
```

Sequence

- No conditions or anything complicated
- But still order of execution

# What is Control-Flow?

```
distance = distance + 1;
```

Reads and Writes

- The expression needs to be evaluated, before we can save its result

# What is Control-Flow?

```
int i = 2;  
int j = (i=3) * i;
```

## Expressions & side-effects

- Order in sub-expressions is usually undefined
- Side-effects make sub-expression order relevant

# Kinds of Control-Flow

- Sequential statements
- Conditional if / switch / case
- Looping while / do while / for / foreach / loop
- Exceptions throw / try / catch / finally
- Continuations call/cc
- Async-await threading
- Coroutines / Generators yield
- Dispatch function calls / method calls
- Loop jumps break / continue
- ... many more ...

# Why Control-Flow?

## Shorter code

- No need to repeat the same statement 10 times

## Parametric code

- Extract reusable patterns
- Let user decide repetition amount

## Expressive power

- Playing with Turing Machines

## Reason about program execution

- What happens when?
- In what order?

# Control-Flow and Language Design

## Imperative programming

- Explicit control-flow constructs

## Declarative programming

- What, not how
- Less explicit control-flow
- More options for compilers to choose order
- Great if your compiler is often smarter than the programmer

# Separation of Concerns in Data-Flow Analysis

## Representation

- Represent control-flow of a program
- Conduct and represent results of data-flow analysis

## Declarative Rules

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

## Language-Independent Tooling

- Data-Flow Analysis
- Errors/Warnings
- Code completion
- Refactoring
- Optimisation
- ...

# 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.

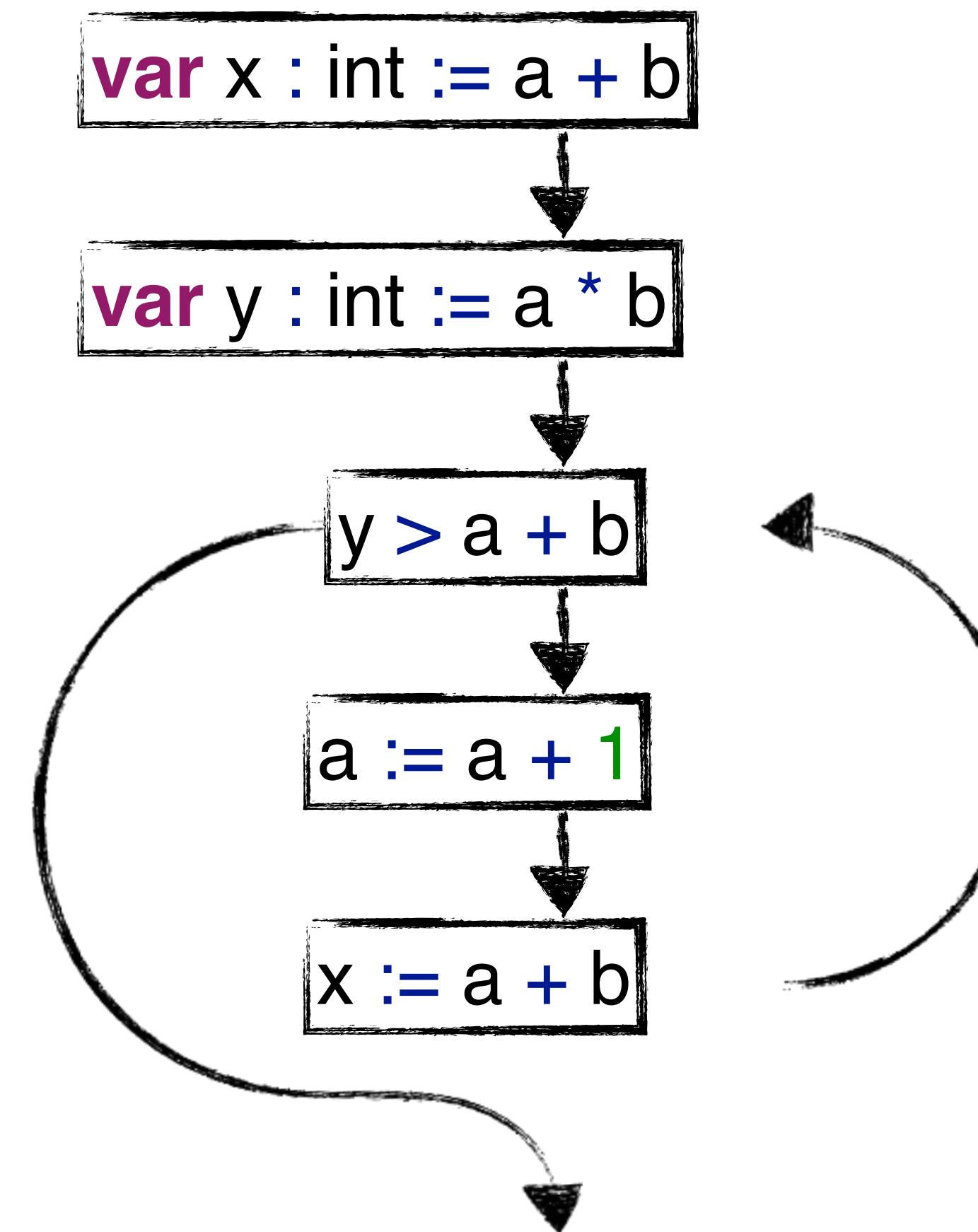
[https://en.wikipedia.org/wiki/Control\\_flow\\_graph](https://en.wikipedia.org/wiki/Control_flow_graph)

# Control-Flow Graph Example

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

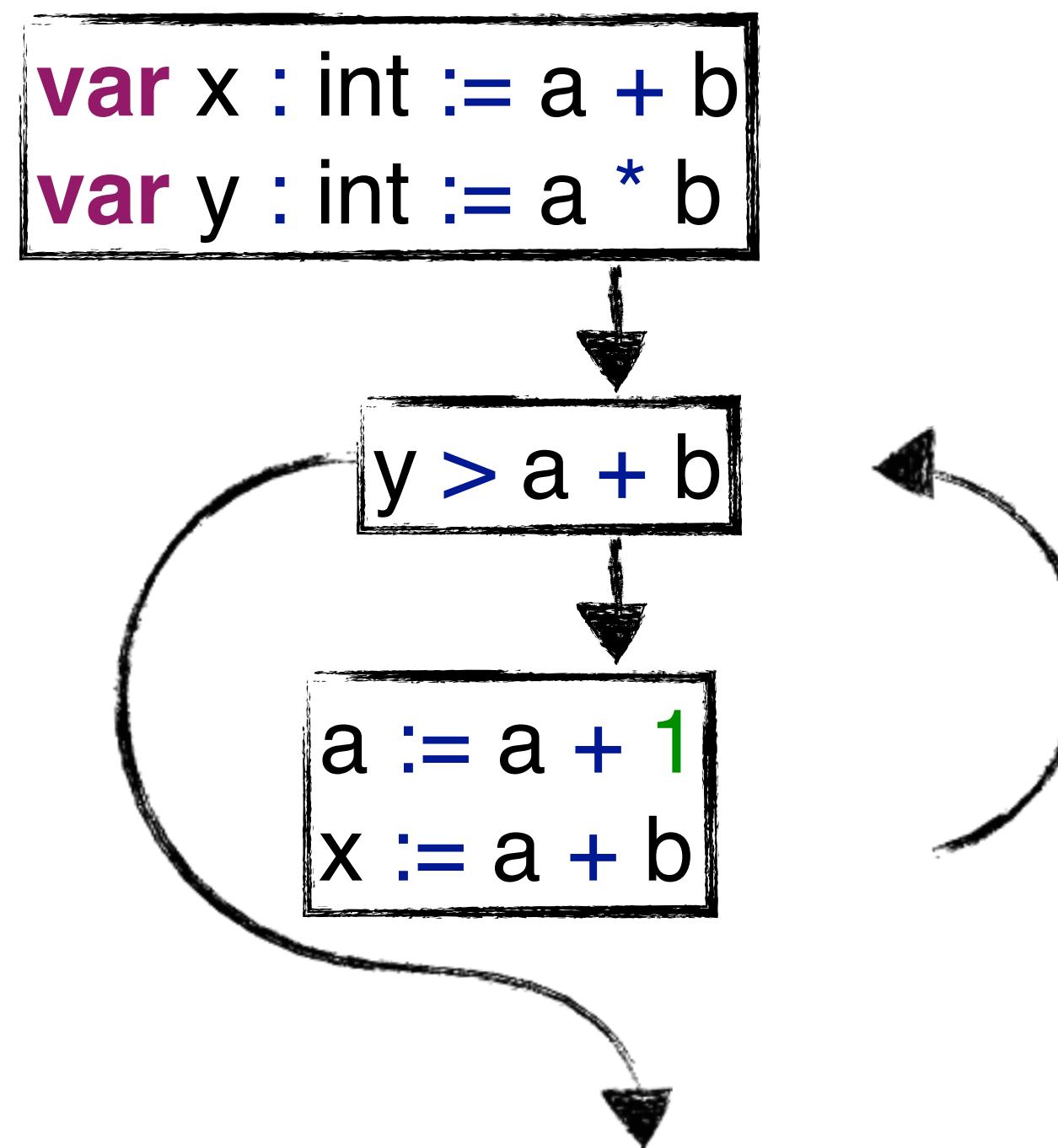
# Control-Flow Graph Example

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



# Basic Blocks

```
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

## Nodes

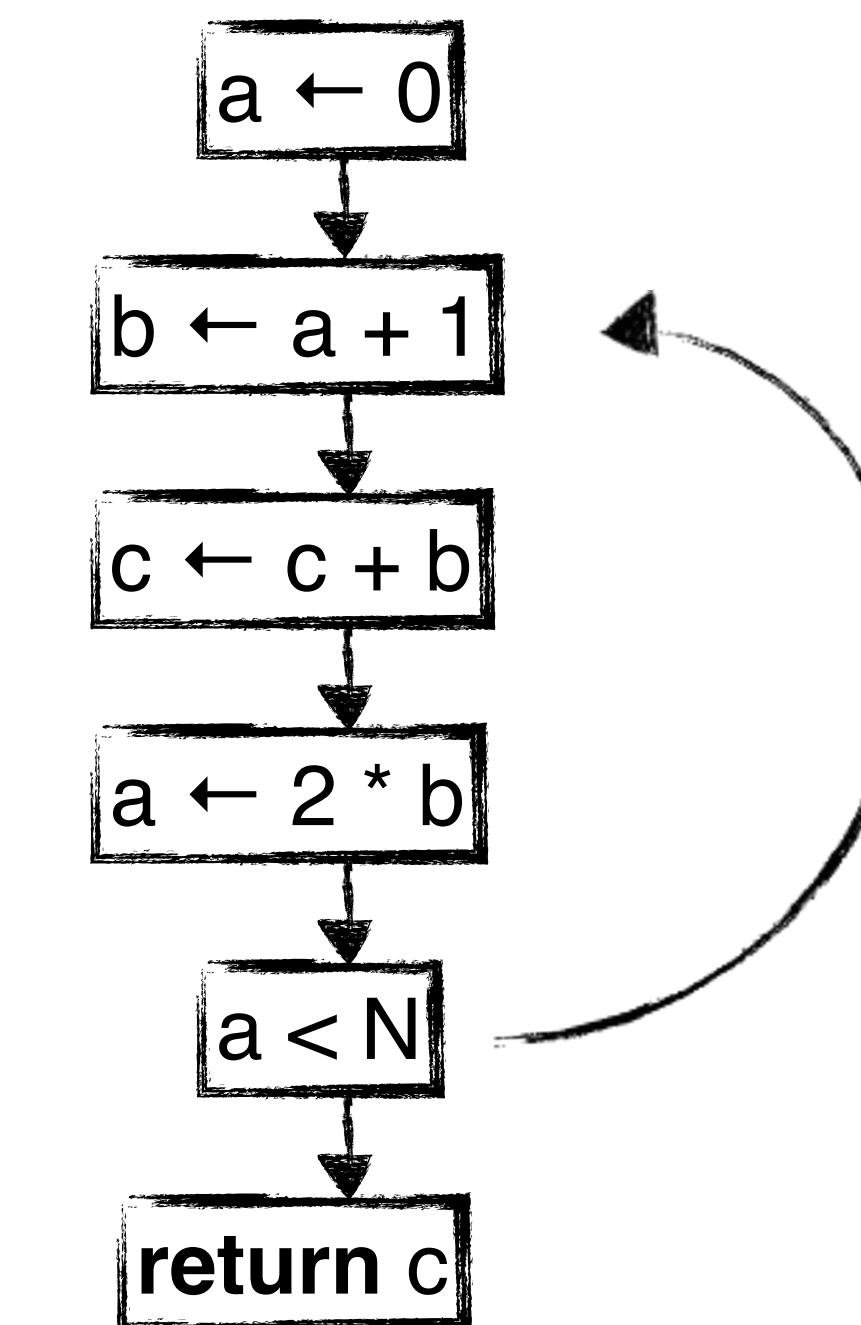
- Usually innermost statements and expressions
- Or blocks for consecutive statements (basic blocks)

## Edges

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

# Equivalent to Unstructured Control-Flow

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



# Separation of Concerns in Data-Flow Analysis

## Representation

- Represent control-flow of a program
- Conduct and represent results of data-flow analysis

## Declarative Rules

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

## Language-Independent Tooling

- Data-Flow Analysis
- Errors/Warnings
- Code completion
- Refactoring
- Optimisation
- ...

# Separation of Concerns in Data-Flow Analysis

## Representation

- Control Flow Graphs (CFGs)
- Conduct and represent results of data-flow analysis

## Declarative Rules

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

## Language-Independent Tooling

- Data-Flow Analysis
- Errors/Warnings
- Code completion
- Refactoring
- Optimisation
- ...

# Data-Flow

## What is Data-Flow?

- Possible values (data) that flow through the program
- Relations between those data (data dependence)

## Discuss a series of example programs

- What is wrong or can be optimised?
- What is the flow we can use for this?
- What would the data-flow information look like?

# What is wrong here?

```
public int ComputeFac(int num) {  
    return num;  
    int num_aux;  
    if (num < 1)  
        num_aux = 1;  
    else  
        num_aux = num * this.ComputeFac(num-1);  
    return num_aux;  
}
```

Dead code (control-flow)

- Most of the code is never reached because of the early return
- This is usually considered an error by compilers

# What is “wrong” here?

```
x := 2;  
y := 4;  
x := 1;  
// x and y used later
```

Dead code (data-flow)

Live variable analysis

- The first value of x is never observed
- This is sometimes warned about by compilers

# What is suboptimal here?

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

Common subexpression elimination

Available expression analysis

- $a + b$  is already computed when you get to the condition
- There is no need to compute it again

# What is suboptimal here?

```
for i := 1 to 100 do
(
    x[i] := y[i];
    if w > 0 then
        y[i] := 0
)
```

Loop unswitching

Data-dependence analysis

- The if condition is not dependent on i, x or y
- Still it is checked in the loop, which is slowing the loop

# Separation of Concerns in Data-Flow Analysis

## Representation

- Control Flow Graphs (CFGs)
- Conduct and represent results of data-flow analysis

## Declarative Rules

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

## Language-Independent Tooling

- Data-Flow Analysis
- Errors/Warnings
- Code completion
- Refactoring
- Optimisation
- ...

# Separation of Concerns in Data-Flow Analysis

## Representation

- Control Flow Graphs (CFGs)
- Data-flow information on CFG nodes

## Declarative Rules

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

## Language-Independent Tooling

- Data-Flow Analysis
- Errors/Warnings
- Code completion
- Refactoring
- Optimisation
- ...

# Separation of Concerns in Data-Flow Analysis

## Representation

- Control Flow Graphs (CFGs)
- Data-flow information on CFG nodes

## Declarative Rules

- A domain-specific meta-language for Spoofax: FlowSpec

## Language-Independent Tooling

- Data-Flow Analysis
- Errors/Warnings
- Code completion
- Refactoring
- Optimisation
- ...

# Tiger in FlowSpec

# Control-Flow Rules

## Map abstract syntax to control-flow (sub)graphs

- Match an AST pattern
- List all CFG edges of that AST
- Mark subtrees as CFG nodes
- Or splice in their control-flow subgraph
- Use special “context” nodes to connect subgraph to outside graph

# Control-Flow Graphs in FlowSpec

FlowSpec

```
root Mod(s) =  
  start → s → end
```

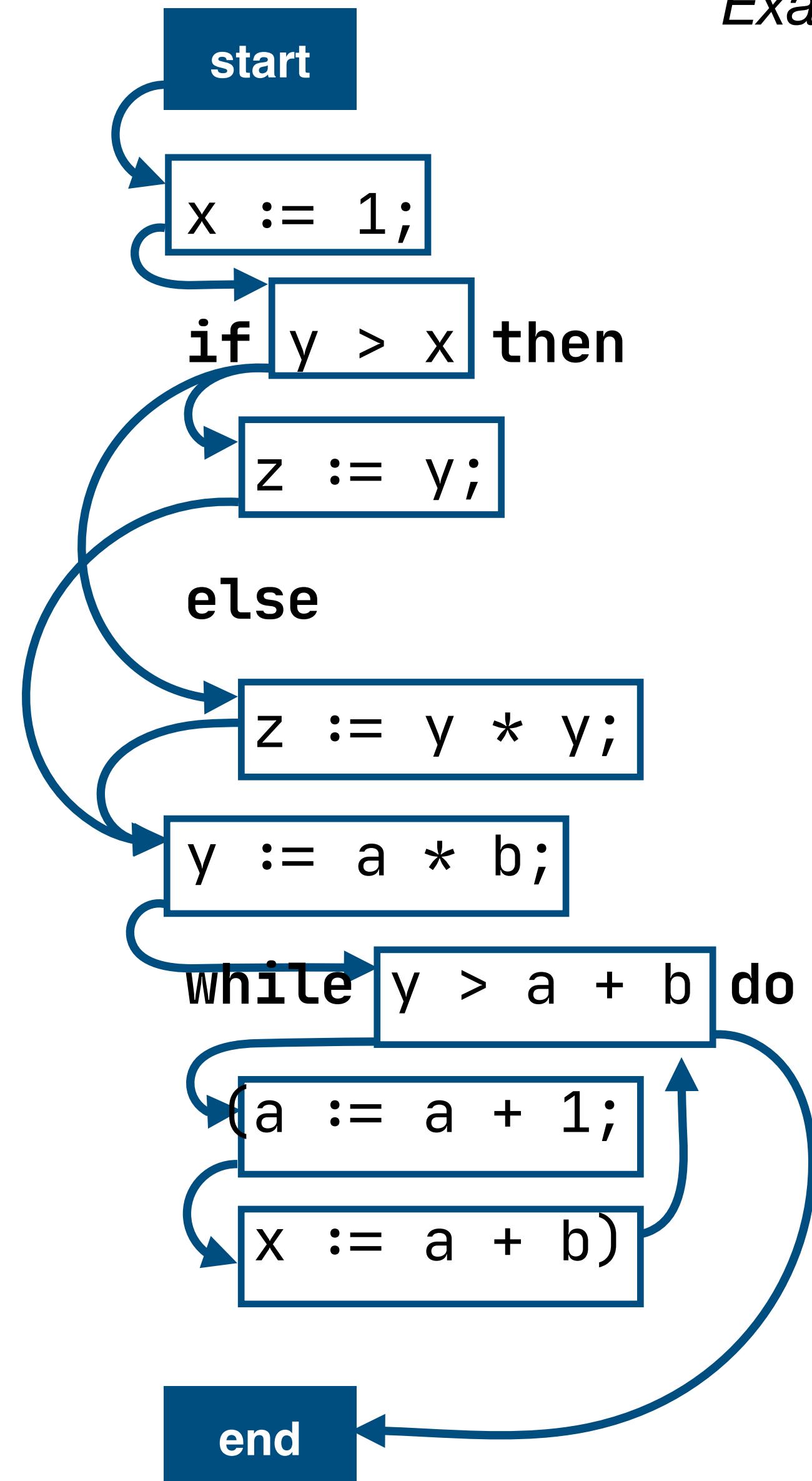
```
node Assign(_, _)
```

```
Seq(s1, s2) =  
  entry → s1 → s2 → exit
```

```
IfThenElse(c, t, e) =  
  entry → node c → t → exit,  
        node c → e → exit
```

```
While(c, b) =  
  entry → node c → b → node c,  
        node c → exit
```

Example program



# Data-Flow Rules

## Define effect of control-flow graph nodes

- Match an AST pattern on one side of a CFG edge
- Propagate the information from the other side of the edge
- Adapt that information as the effect of the matched CFG node

# Live Variables in FlowSpec

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

## properties

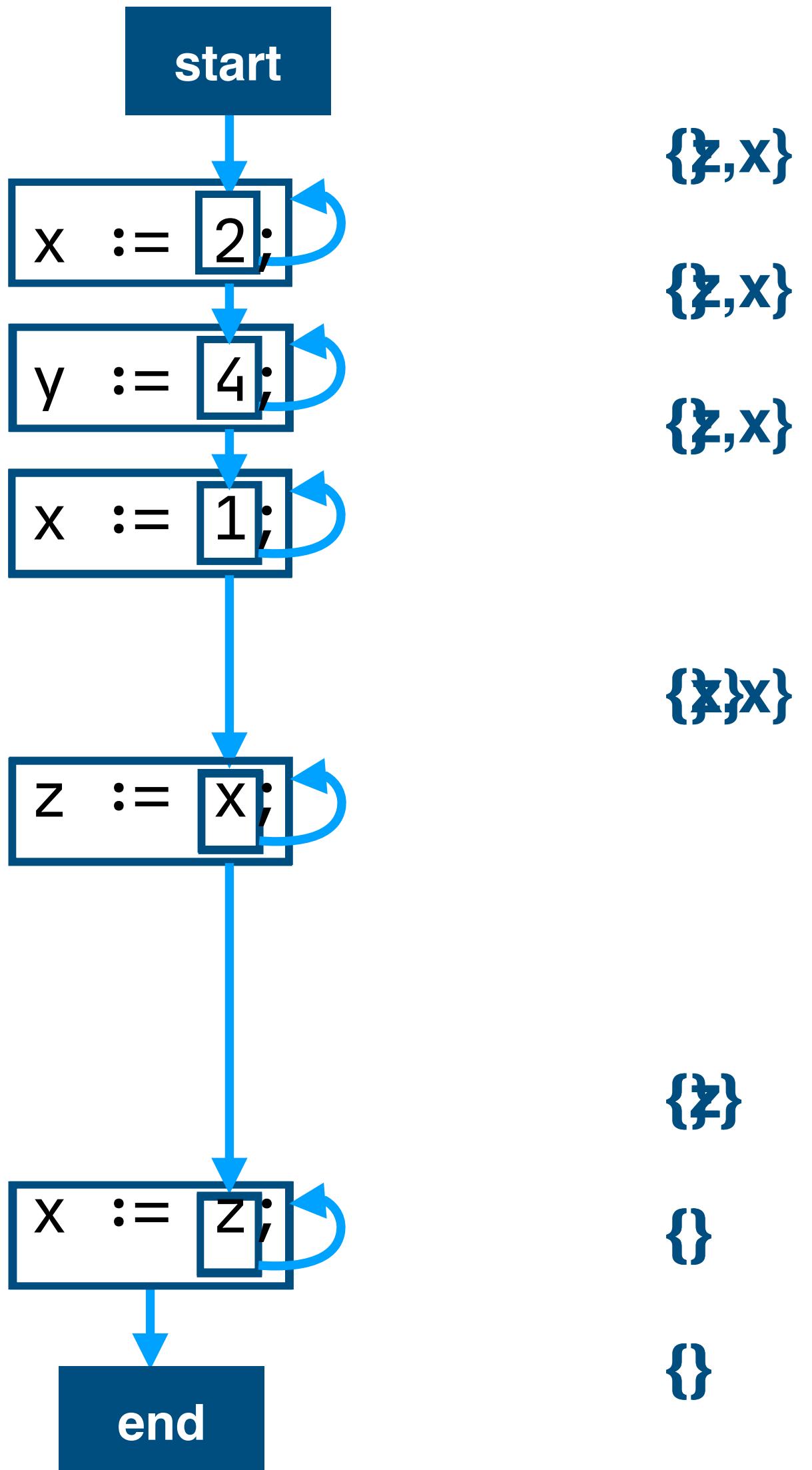
```
live: Set(name)
```

## property rules

```
live(Ref(n) → next) =  
live(next) ∨ { Var{n} }
```

```
live(Assign(n, _) → next) =  
{ m | m ← live(next), Var{n} ≠ m }
```

```
live(_.end) =  
{}
```



# Live Variables in FlowSpec

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

## properties

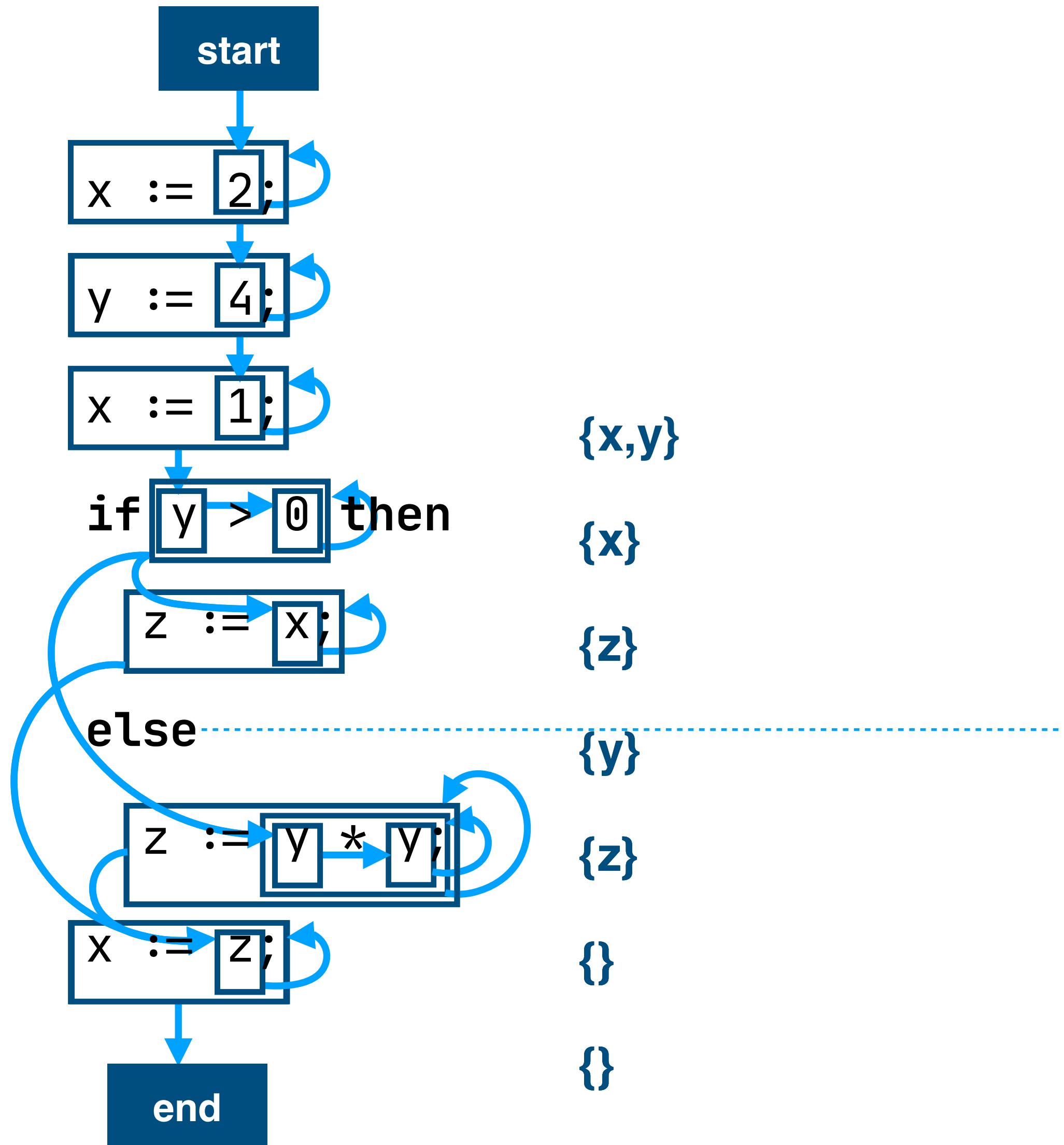
```
live: Set(name)
```

## property rules

```
live(Ref(n) → next) =  
live(next) ∨ {n}
```

```
live(Assign(n, _) → next) =  
{ m | m ← live(next), n ≠ m }
```

```
live(_.end) =  
{}
```



# Available Expressions in FlowSpec

An expression is *available* if it *must* have been evaluated previously and its variables not reassigned

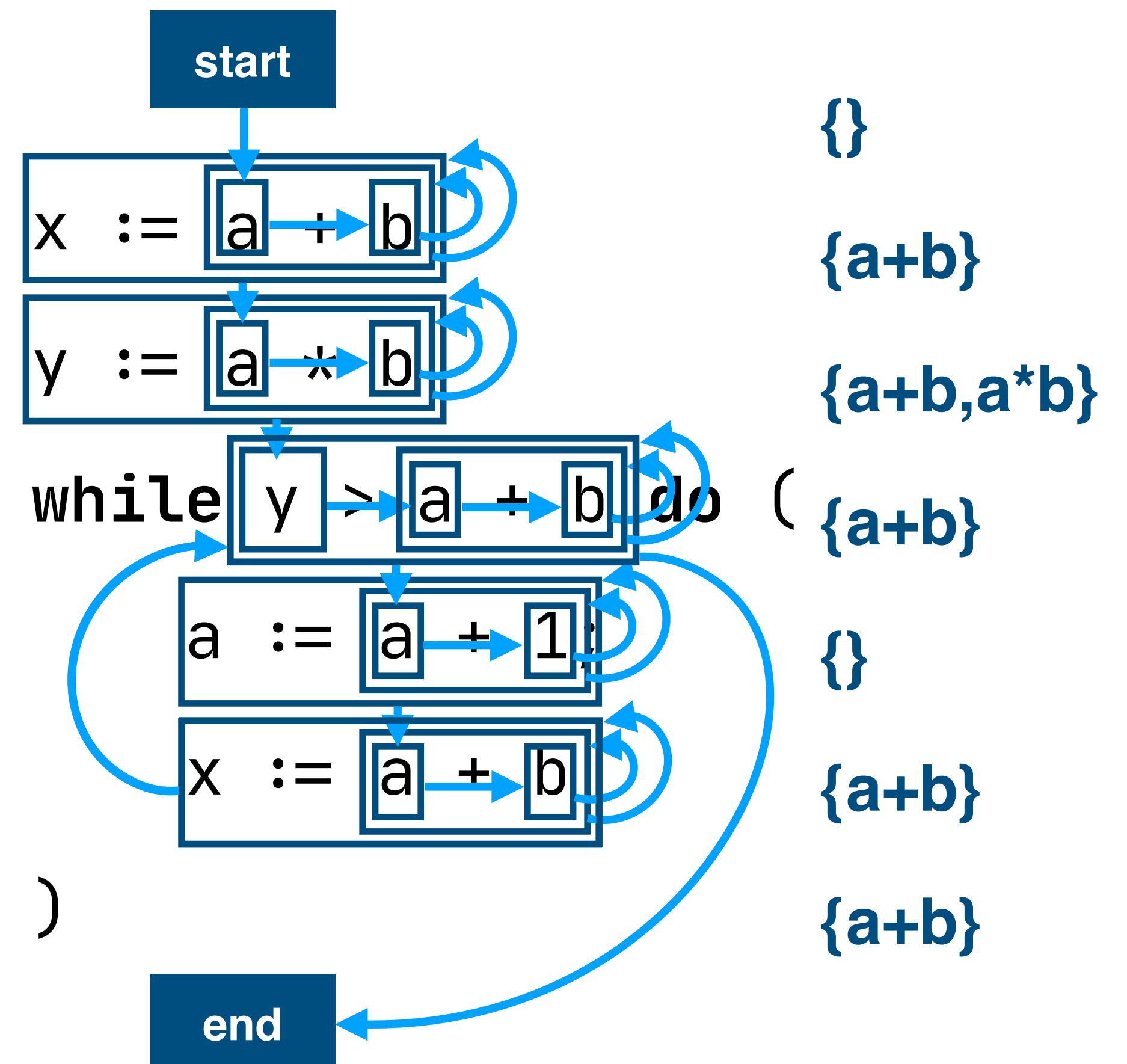
## properties

```
available: MustSet(term)
```

## property rules

```
available(prev → Assign(n, e)) =  
{ expr |  
expr ← available(prev) \v {e},  
!(n in reads(expr)) }
```

```
available(_.start) =  
{}
```



# Summary

# Summary: Data-Flow Analysis Specification

## Control-Flow

- Order of execution
- Reasoning about what is reachable

## Data-Flow

- Flow of data through a program
- Reasoning about data, and dependencies between data

## FlowSpec

- Control-Flow rules to construct the graph
- Annotate with information from analysis by Data-Flow rules

Next

## From Specification to Implementation

# Traditional Kill/Gen Sets

# 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) }
```

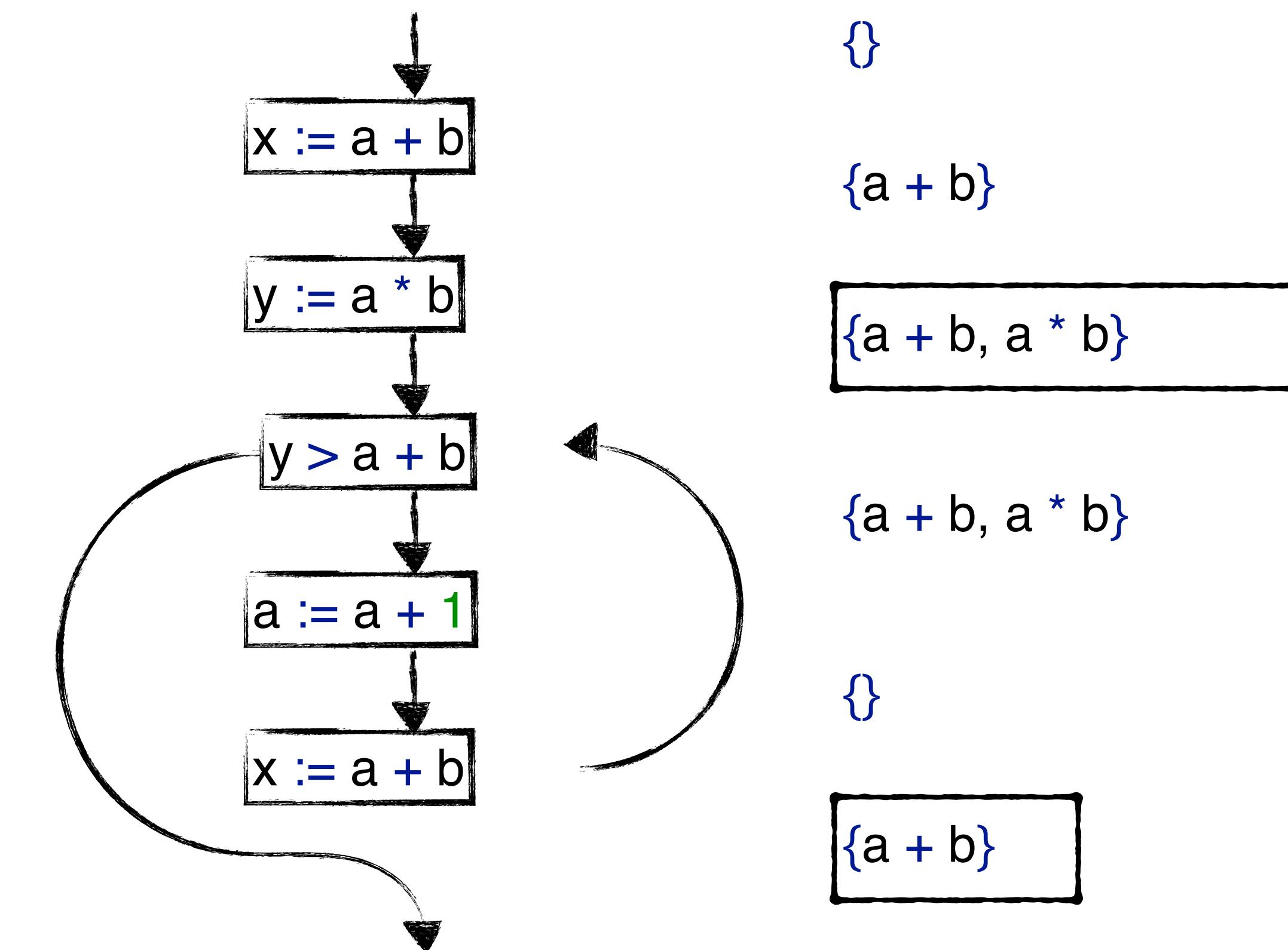
previousSet \ **kill**(currentNode)  $\cup$  **gen**(currentNode)

# 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) }
```



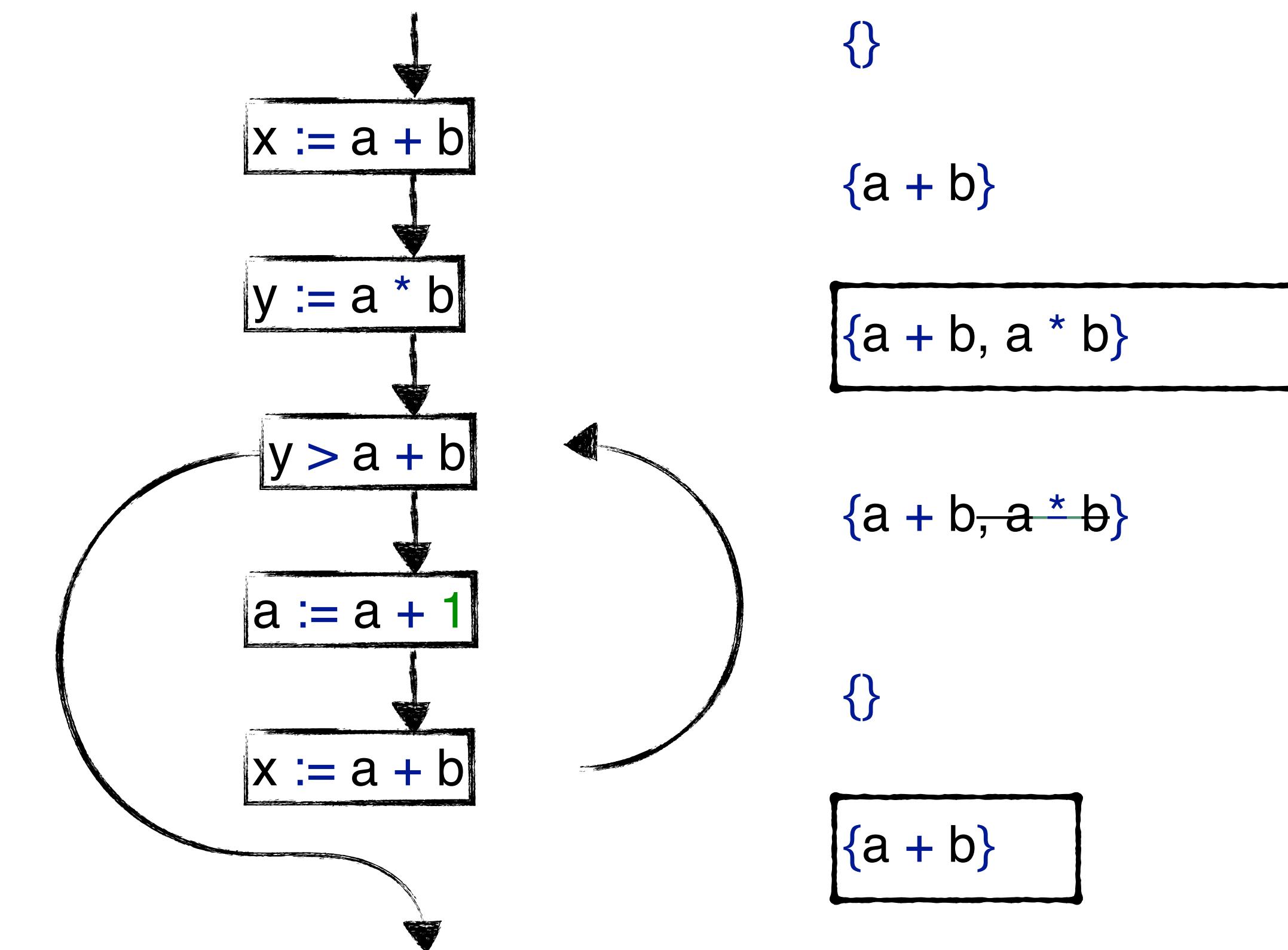
previousSet \ **kill**(currentNode)  $\cup$  **gen**(currentNode)

# 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) }
```



previousSet \ **kill**(currentNode)  $\cup$  **gen**(currentNode)

# Live Variables

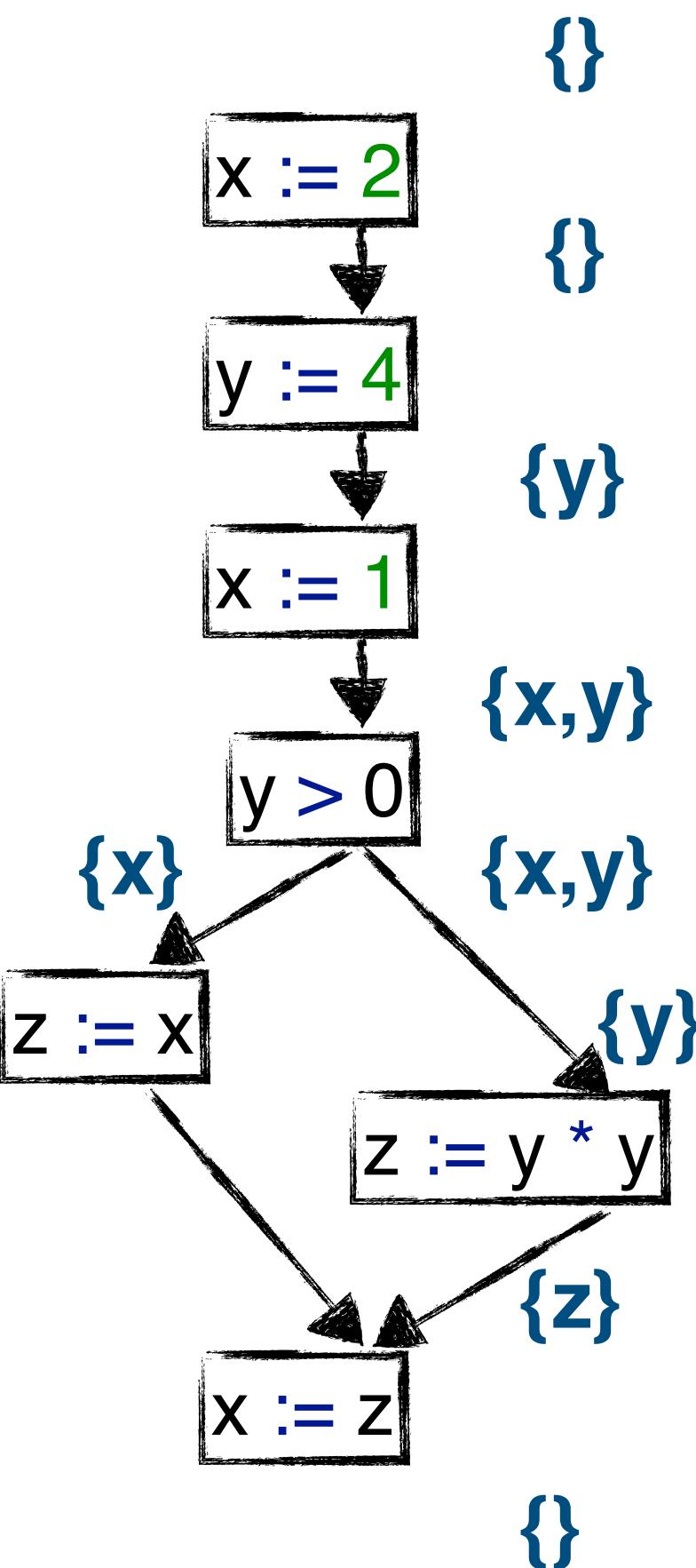
“A **variable** is **live** if there exists a path from there to a use of the variable, with no re-definition of the variable on that path.”

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

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

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

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



previousSet \ **kill**(currentNode) **U gen**(currentNode)

# Traditional set based analysis

## Sets as analysis information

### Kill and gen sets per control node type

- $\text{previousSet} \setminus \text{kill}(\text{currentNode}) \cup \text{gen}(\text{currentNode})$

### 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
```

single step

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



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

## Kill/gen doesn't work here

- We need the previous information to compute the current

## Can we use a set for this map?

- Keys map to single values, so no

## But what if we keep multiple values?

- Analysing loops may not terminate

# Example: Non-termination

```

let
  var a : int := 0
  var b : int := a + 1
in
  while y > a + b do
    a := a + 1
end

```

a  $\mapsto$  0  
           a  $\mapsto$  0; b  $\mapsto$  1  
           a  $\mapsto$  0; b  $\mapsto$  1  
           a  $\mapsto$  0, 1; b  $\mapsto$  1  
           a  $\mapsto$  0, 1, 2; b  $\mapsto$  1

# 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

**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 fixpoint
- How can we check this?

**A fixpoint of function  $f$  is  $x$  if  $f(x) = x$**

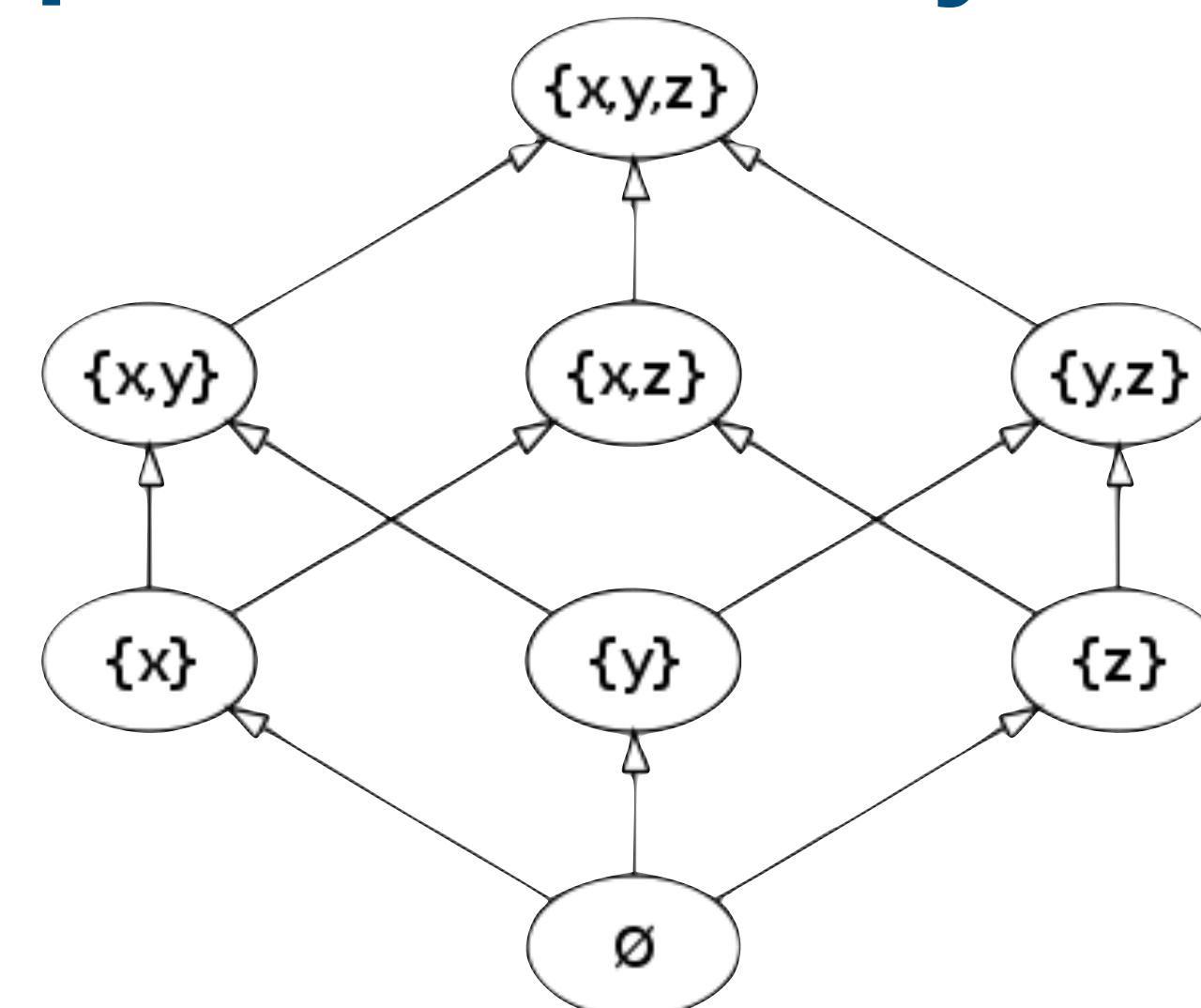
# Lattice Theory

A set  $X$  is totally ordered under  $\leq$  if for  $a, b, c \in X$

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

A partial ordering drops the totality constraint

- e.g. subset inclusion:



# Lattice Theory

A Lattice is a partially ordered set where

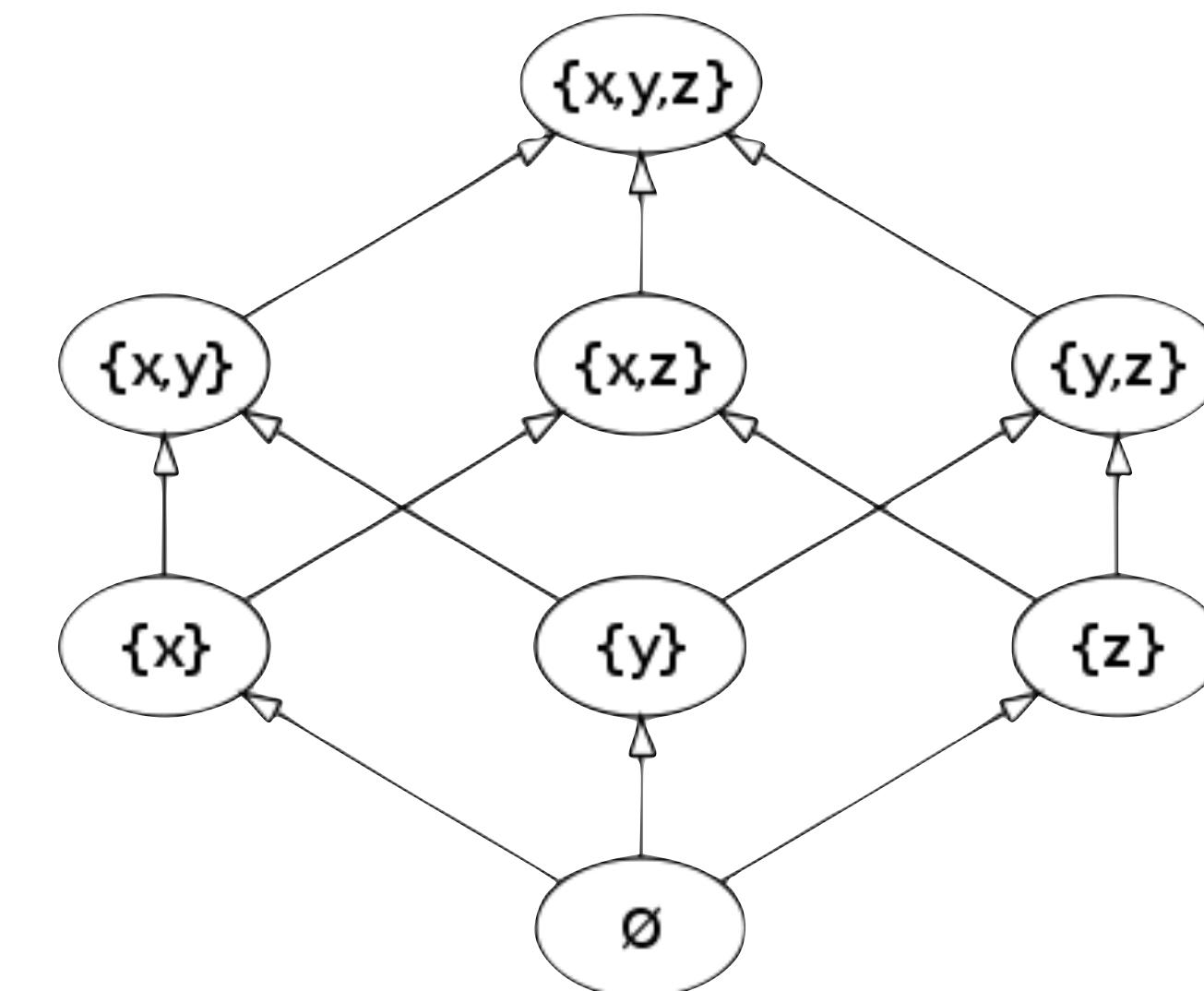
- every two elements have a unique least upper bound (or supremum or join)
- every two elements have a unique greatest lower bound (or infimum or meet)

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  $\perp$  respectively

# Lattices for Data-Flow Analysis

**Consider  $\top$  as the coarsest approximation**

- It is a safe approximation,
- because it says we are 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

## 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  $\top$
- *This only works if the lattice is of finite height*

## General interval analysis has an infinite lattice

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

## 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

## 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

# Executing Monotone Frameworks

# Executing Monotone Frameworks

**Great formal model for reasoning**

- Fairly simple
- Makes intuitive sense
- Has nice mathematical properties

**But how to execute?**

## Control-flow graph

- graph
- start node
- reverse beforehand if backward analysis

## Lattice instance for data-flow information:

- Lattice  $L$
- Least Upper Bound  $\sqcup$
- Bottom value  $\perp \in L$

## Initial data-flow information for start node

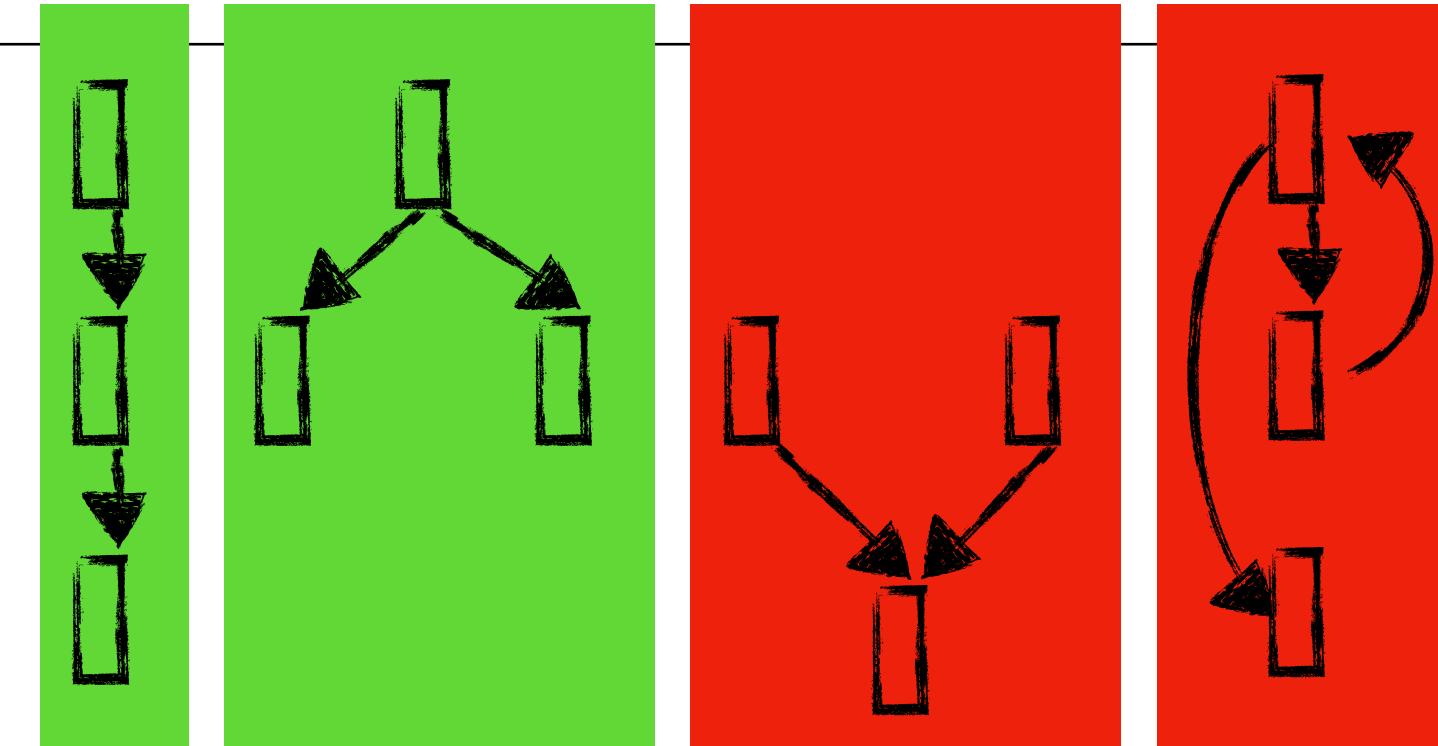
## Transfer function $f : (L \rightarrow L)$ per control-flow graph node

- Denotes the data-flow effect of the CFG node

# Naive Approach

```
start_node.value = initial_value
walk(start_node)

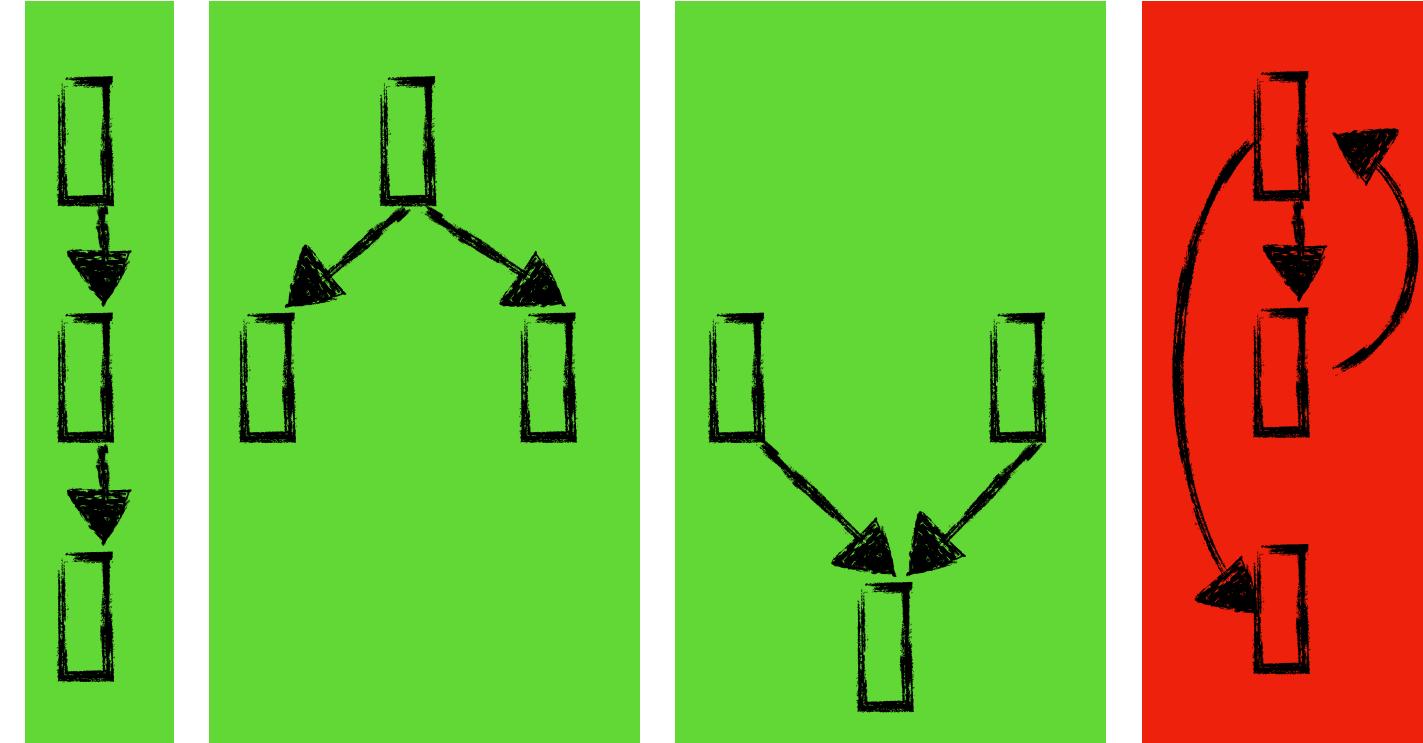
function walk(node) =
    for next in node.successors:
        next.value = node.transfer(node.value)
        walk(next)
```



- Fine for straight-line programs
- Distributes information along splits in control-flow
- Overrides values from one path with those of another
- Recursive: great for stack overflows on loops

# Using Lattices

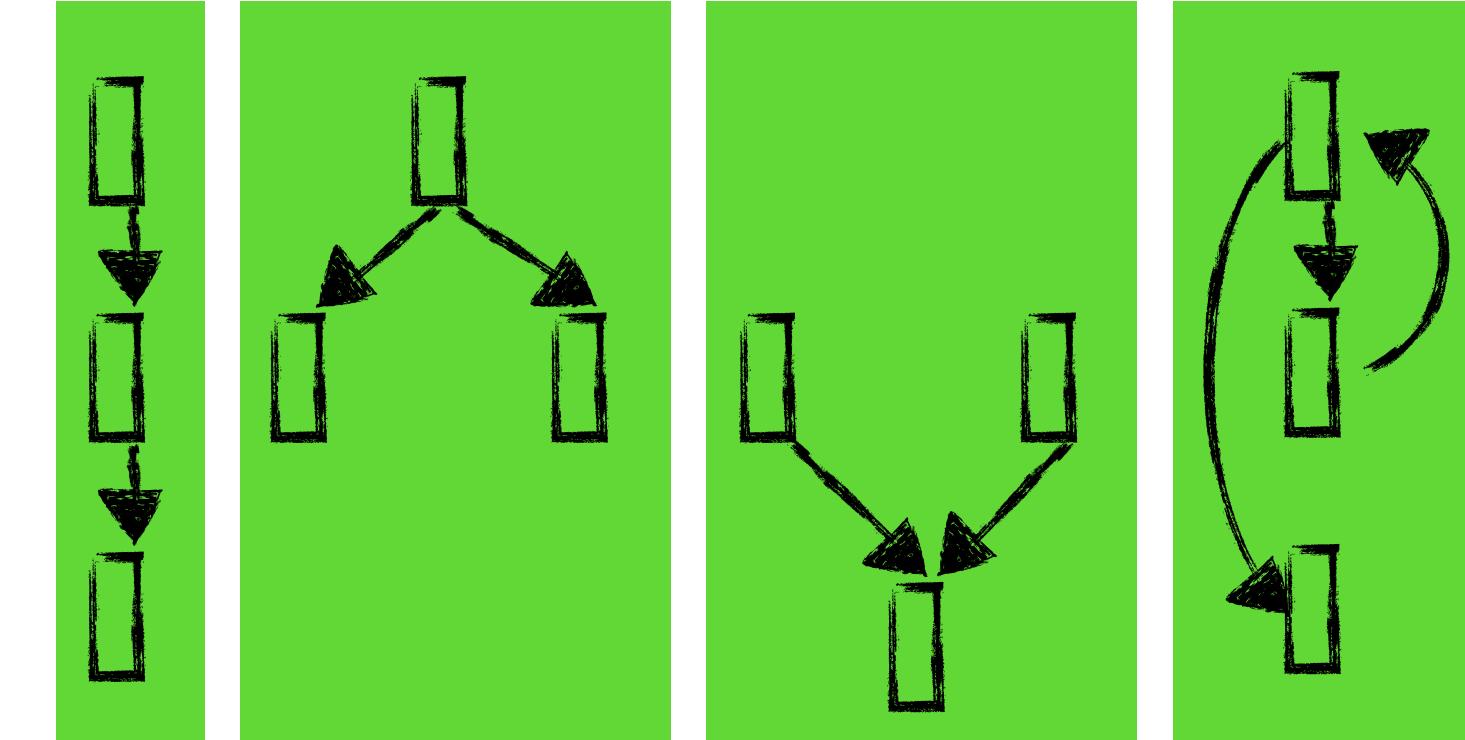
```
for node in nodes:  
    node.value = bottom  
start_node.value = initial_value  
walk(start_node)  
  
function walk(node) =  
    for next in node.successors:  
        next.value =  
            next.value  $\sqcup$  node.transfer(node.value)  
        walk(next)
```



- Fine for straight-line programs
- Distributes information along splits in control-flow
- Combines values from one path with those of another
- Recursive: great for stack overflows on loops

# Worklist: Iterative instead of Recursive

```
for node in nodes:  
    node.value = bottom  
start_node.value = initial_value  
worklist = nodes  
  
while !worklist.empty():  
    node = worklist.pop()  
    for next in node.successors:  
        oldValue = next.value  
        newValue = node.transfer(node.value)  
        if !(newValue ⊑ oldValue):  
            next.value = oldValue ∪ newValue  
            worklist = worklist + [ next ]
```



If `initial_value == bottom` and a transfer function is identity:  
traversal will stop there, so don't just start from the `start_node`

- Fine for straight-line programs
- Distributes information along splits in control-flow
- Combines values from one path with those of another
- Worklist: works for loops too

# FlowSpec Design

# Framework Overview

## Control-flow graph

- graph
  - start node
  - reverse beforehand if backward analysis
- Control-flow rules  
Root rule(s)
- In edge direction of data-flow rules

## Lattice instance for data-flow information:

- Lattice  $L$
  - Least Upper Bound  $\sqcup$
  - Bottom value  $\perp \in L$
- In property definition  
In lattice definition

## Initial data-flow information for start node

In special data-flow rule

## Transfer function $f : (L \rightarrow L)$ per control-flow graph node

- Denotes the data-flow effect of the CFG node

In data-flow rule

# Variants of Data-Flow Analysis

## Many interacting features

- Intra-procedural or inter-procedural
  - ▶ Inter-procedural with dynamic dispatch means dynamic control flow analysis depending on the data-flow analysis
- Flow-insensitive, flow-sensitive or even path-sensitive
- Different kind of context-sensitivity for dynamic dispatch
  - ▶ Different contexts for the same program point are separately tracked
  - ▶ Call-sensitivity: limited “stacktrace”, call-path is tracked
  - ▶ Object-sensitivity: objects are tracked by the allocation point in the program

# Worklist Optimizations in FlowSpec

# Worklist Algorithm Optimizations

## Filter irrelevant CFG nodes

- With transfer function  $tf(x) = x$

## Order nodes

- Topological order would make sense
- But there are cycles in our graphs
- Every cycle should be computed to a fixpoint
  - Really we need each strongly connected component (SCC)
- Tarjan's SCCs algorithm gives SCCs in reverse topological order!
- Within each SCC the order should also not be random:
  - We use the reverse post-order of the spanning tree

# Tarjan's SCC algorithm

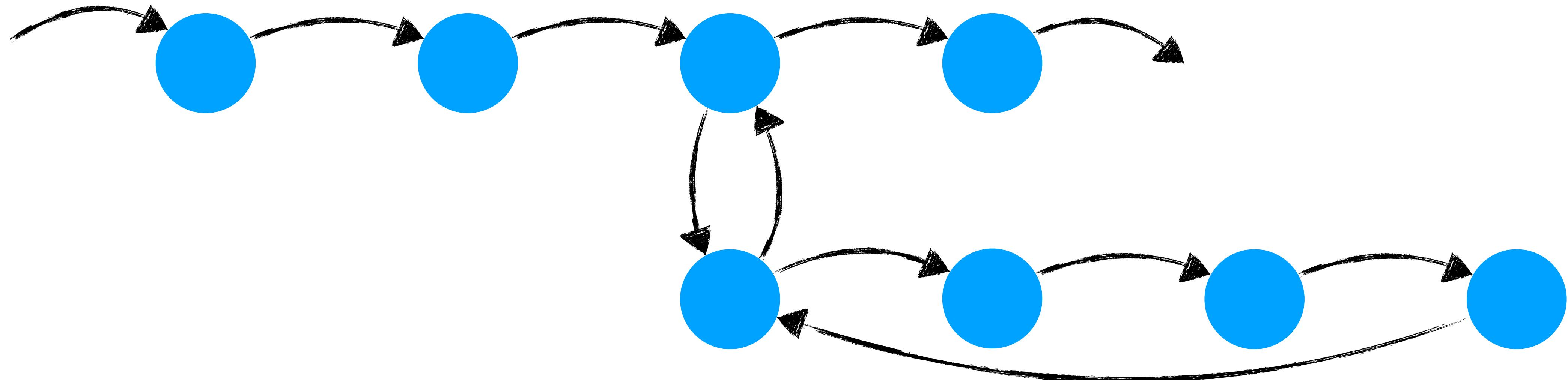
## Strongly Connected Component (SCC) identification

- Label nodes with increasing integers during a depth-first search
  - Multiple searches to make sure you reach all nodes in the graph
- The depth-first spanning forest (spanning trees from the searches) holds SCCs as subtrees
- Nodes that can reach the same lowest label are an SCC together

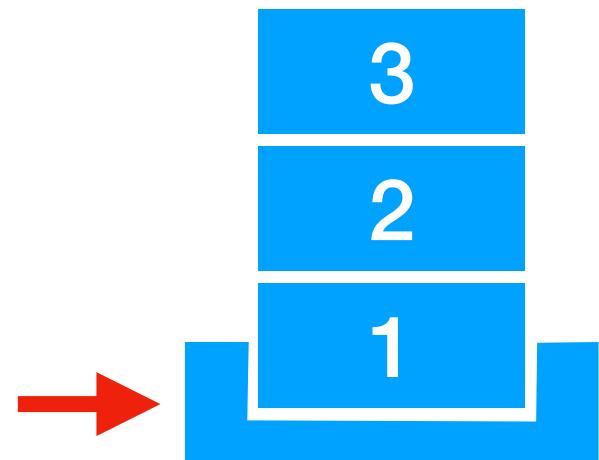
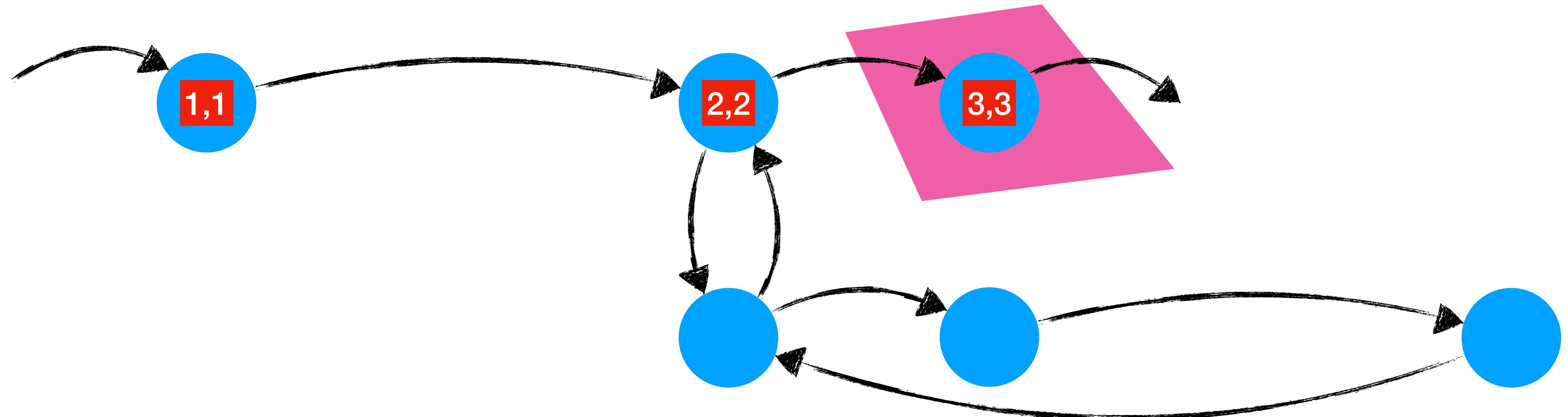
## The version in FlowSpec is slightly adapted

- To return the topological order instead of the reverse topo order
- To have reverse postorder inside SCCs

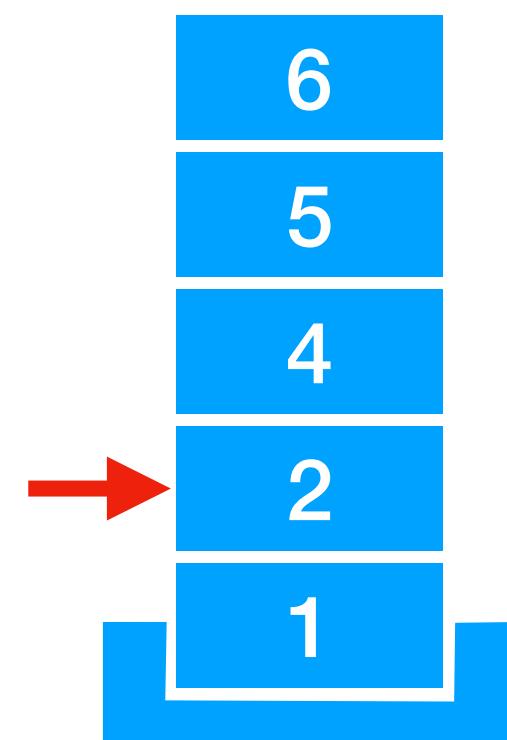
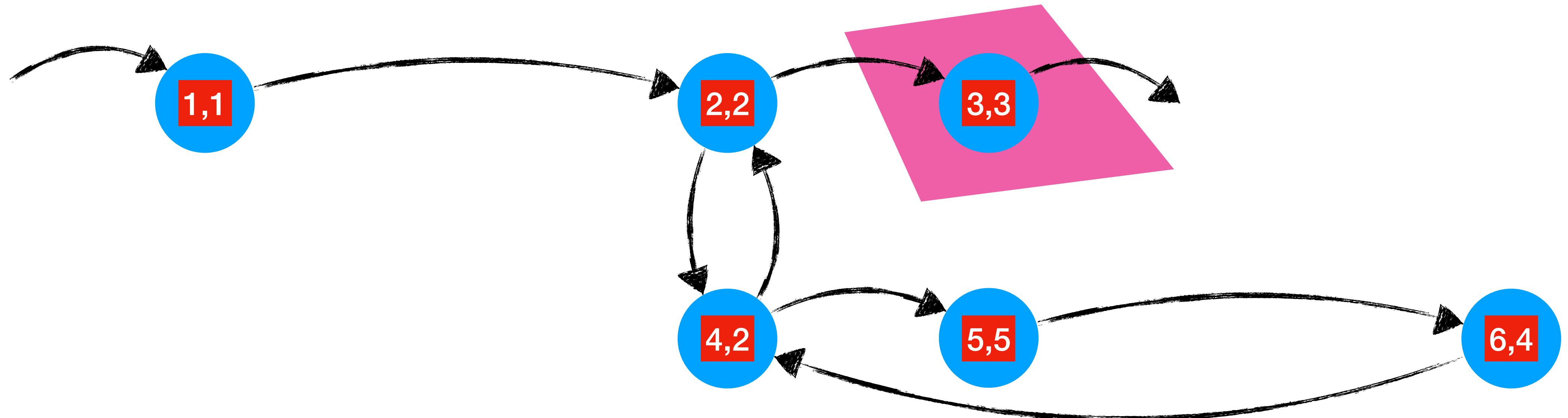
# CFG filtering



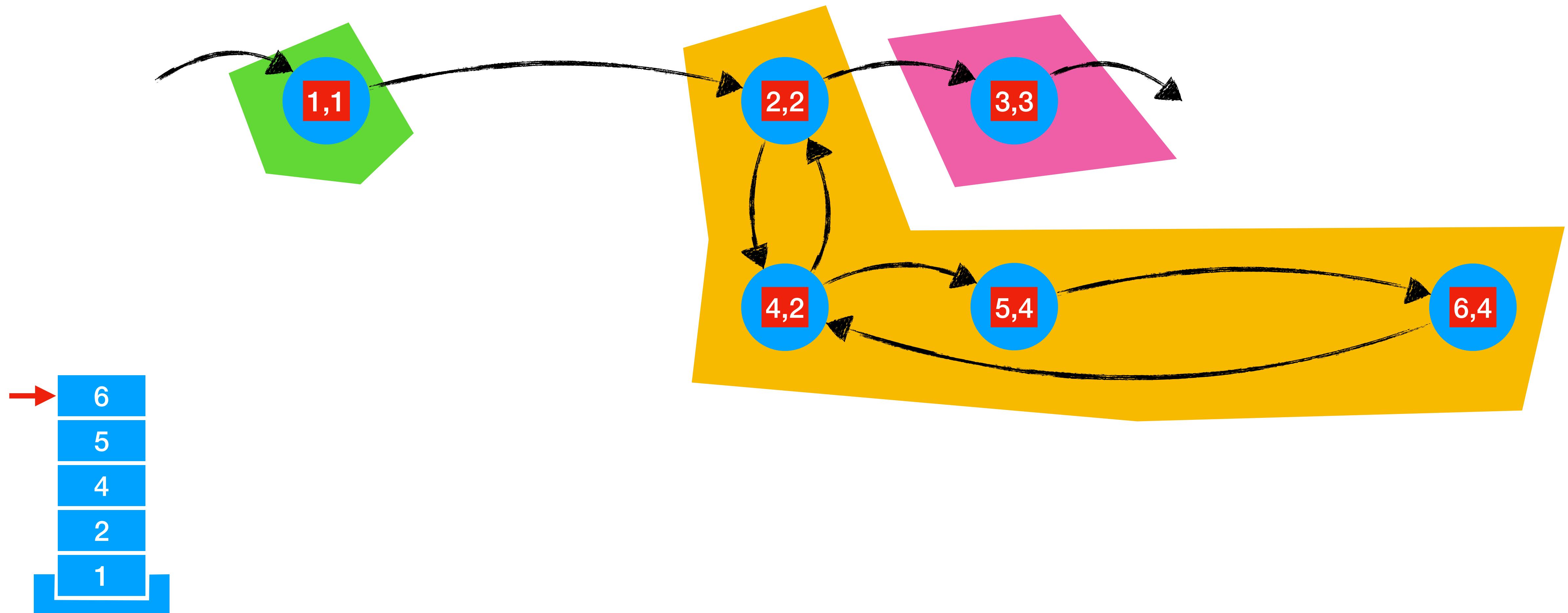
# Tarjan's SCC algorithm



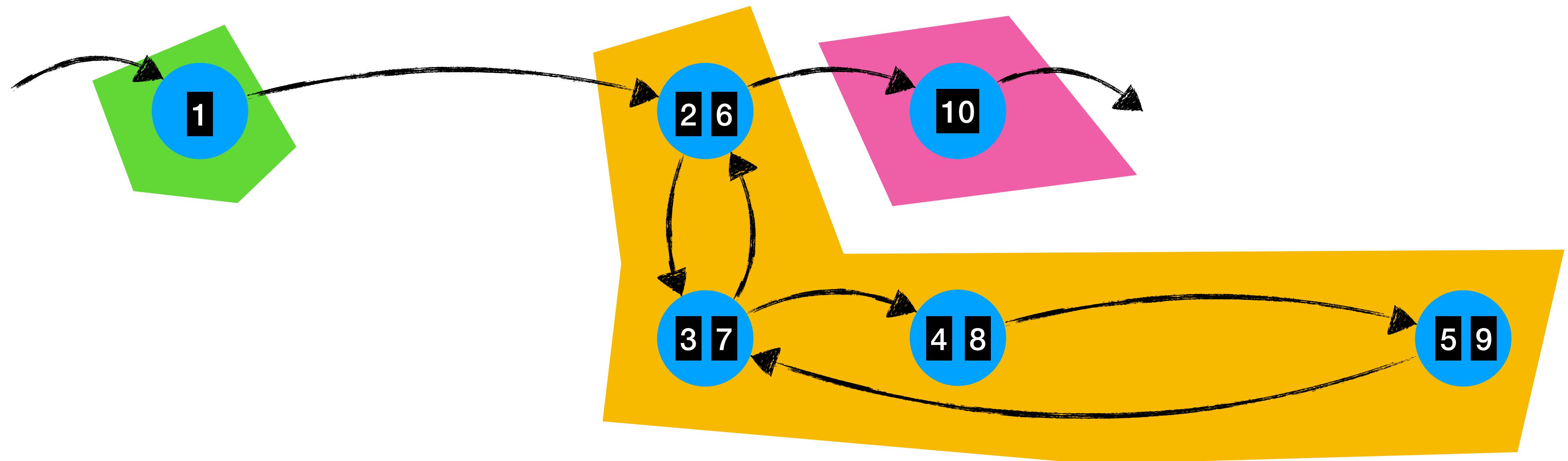
# Tarjan's SCC algorithm



# Tarjan's SCC algorithm



# Reverse postorder in SCC



# Conclusion

## Data-flow analysis and its uses

- Sets are not enough
- Lattices are the generalisation

## Monotone Frameworks

- Finite height lattices + monotone transfer functions = termination
- Execute by worklist algorithm

## FlowSpec design

- FlowSpec only does intra-procedural, flow-sensitive analysis
- Worklist algorithm with optimisations:
  - ▶ SCCs, reverse post-order within SCC, CFG filtering

Except where otherwise noted, this work is licensed under

