

Lecture 10: Data-Flow Analysis

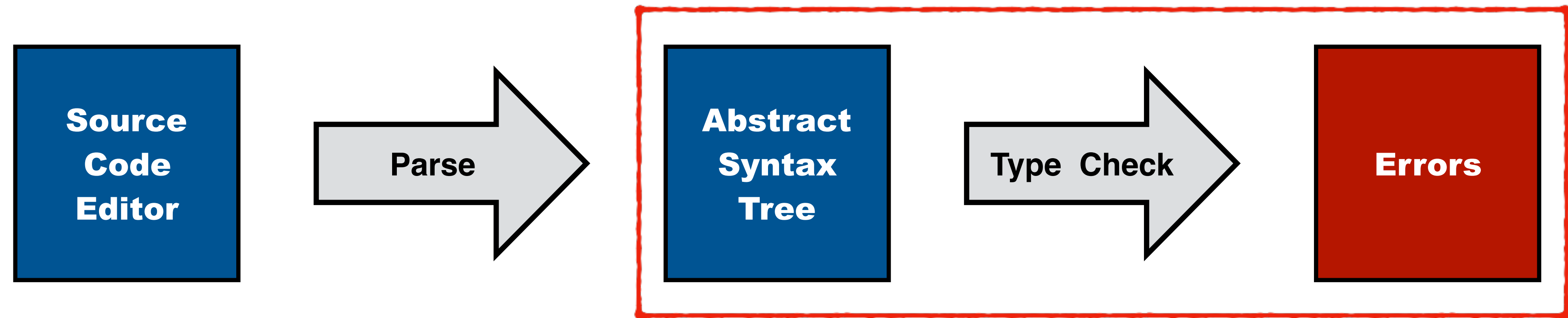
Jeff Smits

CS4200 Compiler Construction

TU Delft

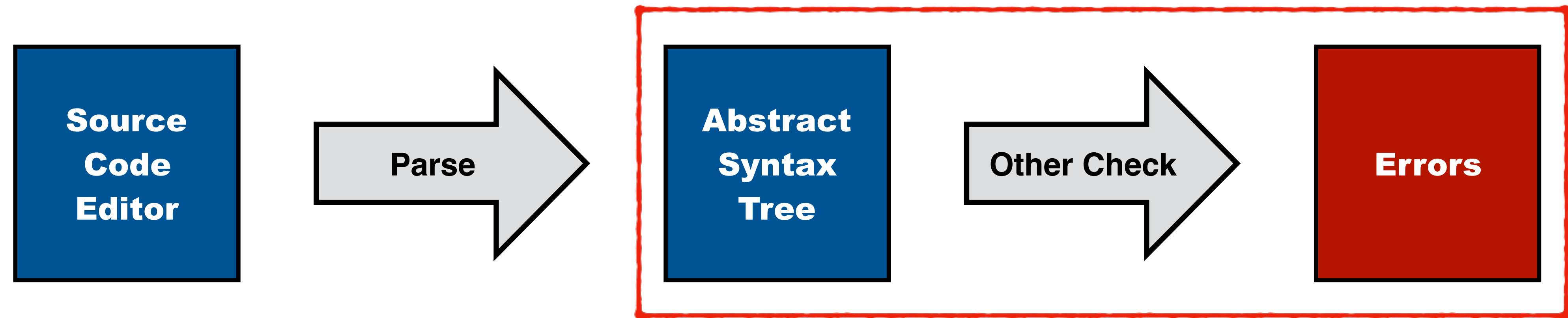
October 2018

Earlier Lecture



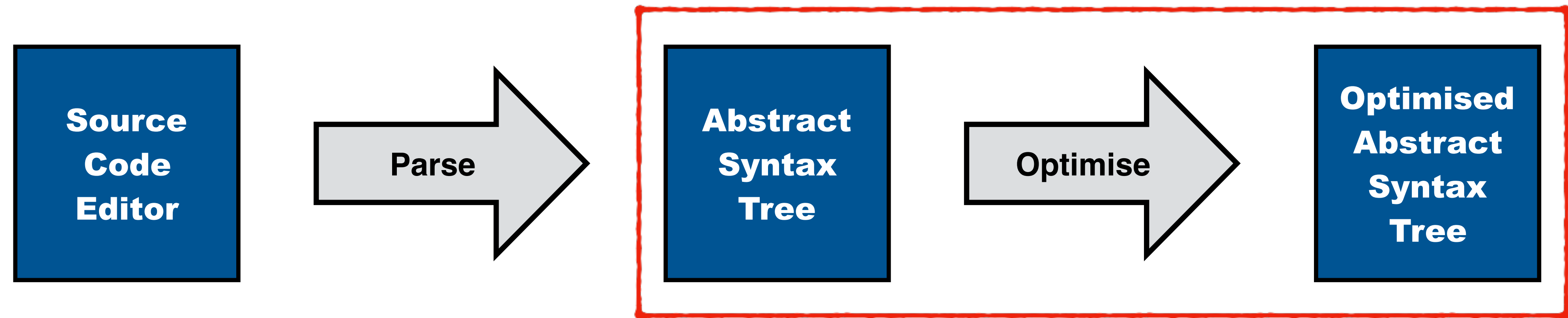
Check that names are used correctly and that expressions are well-typed

This Lecture



Check that variables are initialised, statements are reached, etc.

This Lecture



Eliminate common subexpressions, reduce loop strength, etc.

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 Spoofax. 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 Spoofax 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>

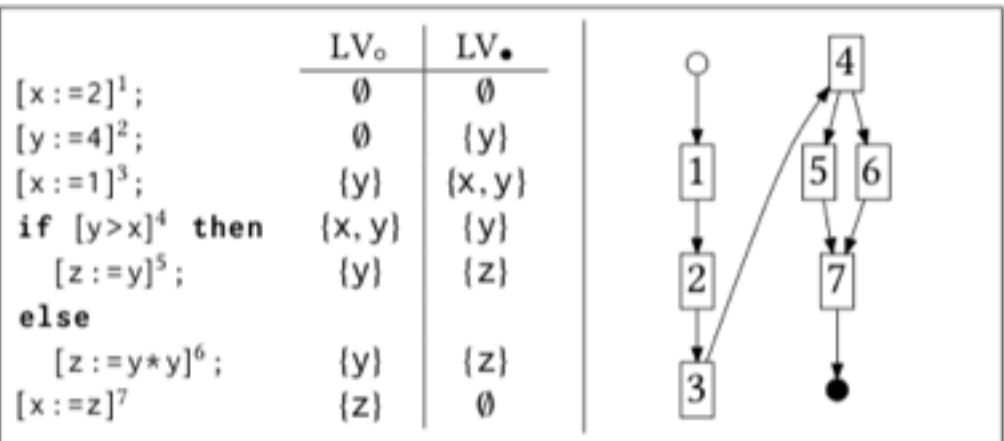


Figure 1. Classical dataflow analysis Live Variables (LV). On the left is an example program in the WHILE language, with added brackets to number program fragments. On the right is the control flow graph (CFG) of the program. In the centre is the analysis result. The LV_0 and LV_1 are before and after 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_1 . The figure shows this in the LV_1 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

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 Spoofax website with a blue header. The header contains a home icon, the text "Spoofax", and "latest". Below the header is a white search bar with the text "Search docs". The main content area has a white background. It starts with "The Spoofax Language Workbench", followed by "Examples" and "Publications". A blue bar with the text "TUTORIALS" in white separates the sections. Below this, there is a list of tutorials: "Installing Spoofax", "Creating a Language Project", "Using the API", and "Getting Support". Another blue bar with the text "REFERENCE MANUAL" in white follows. Below this, there is a list of reference manual topics: "Language Definition with Spoofax", "Abstract Syntax with ATerms", "Syntax Definition with SDF3", "Static Semantics with NaBL2", "Data-Flow Analysis with FlowSpec", "Transformation with Stratego", "Dynamic Semantics with DynSem", "Editor Services with ESV", and "Language Testing with SPT". At the bottom, there is a white box with the text "spec/index.html" and a green button with the text "Read the Docs".

[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

Control-Flow

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?

```
float distance = 12.0;  
float velocity = time / distance;
```

Sequence

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

What is Control-Flow?

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

Writes and reads

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

What is Control-Flow?

```
counter.next() + counter.next()
```

Expressions & side-effects

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

What kind 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 the 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
- Code completion
- Refactoring
- Optimisation
- ...

Control-Flow Graphs

What is a Control-Flow Graph?

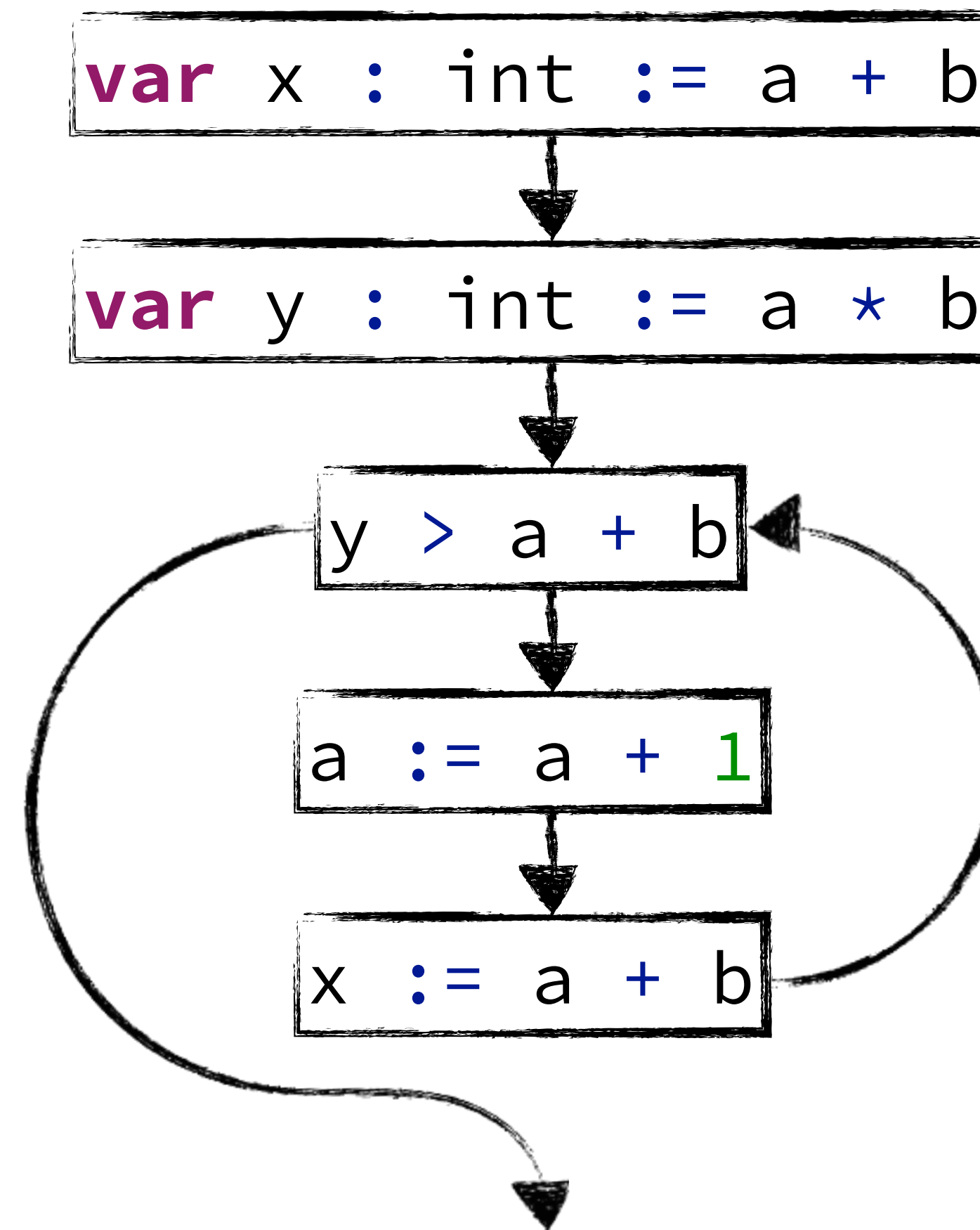
A **control flow graph (CFG)** in **computer science** is a **representation**, using **graph** notation, of all paths that might be traversed through a **program** during its **execution**.

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

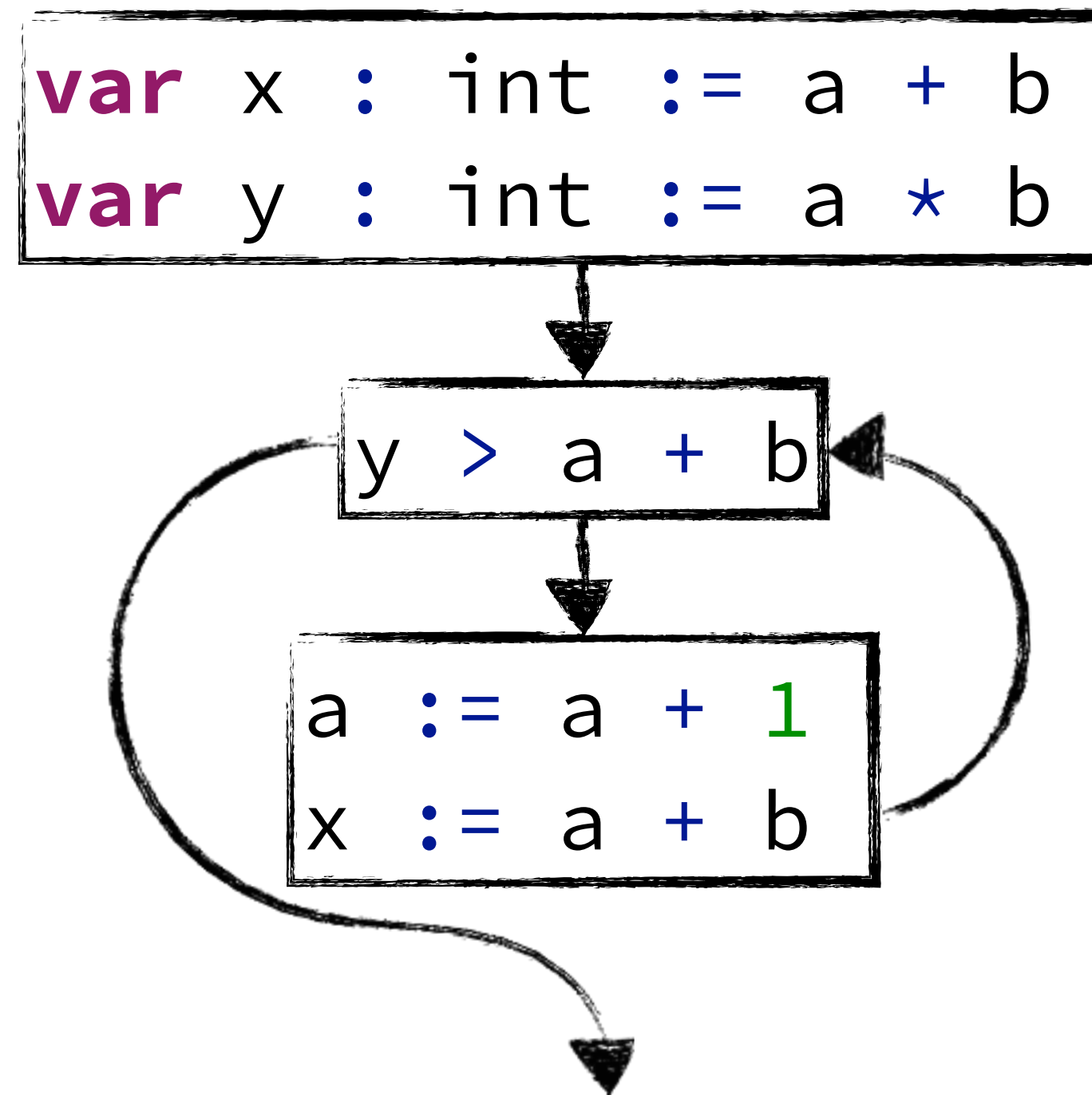
Control-Flow Graph example

```
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



Basic blocks



Control Flow Graphs

Nodes

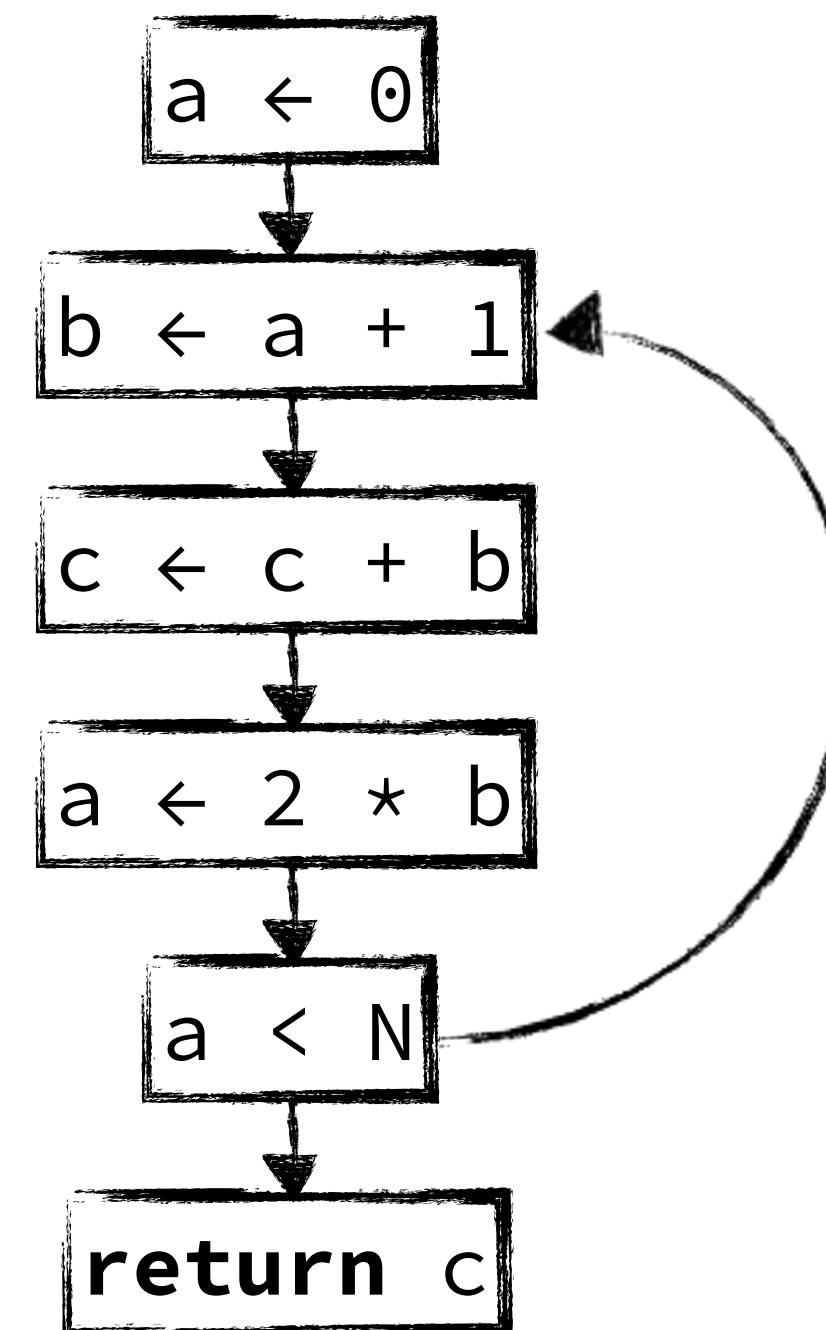
- Usually innermost statements and expressions
- Or blocks for consecutive statements (basic blocks)

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
- Code completion
- Refactoring
- Optimisation
- ...

Separation of Concerns in Data-Flow Analysis

Representation

- **Control Flow Graphs**
- Conduct and represent results of data-flow analysis

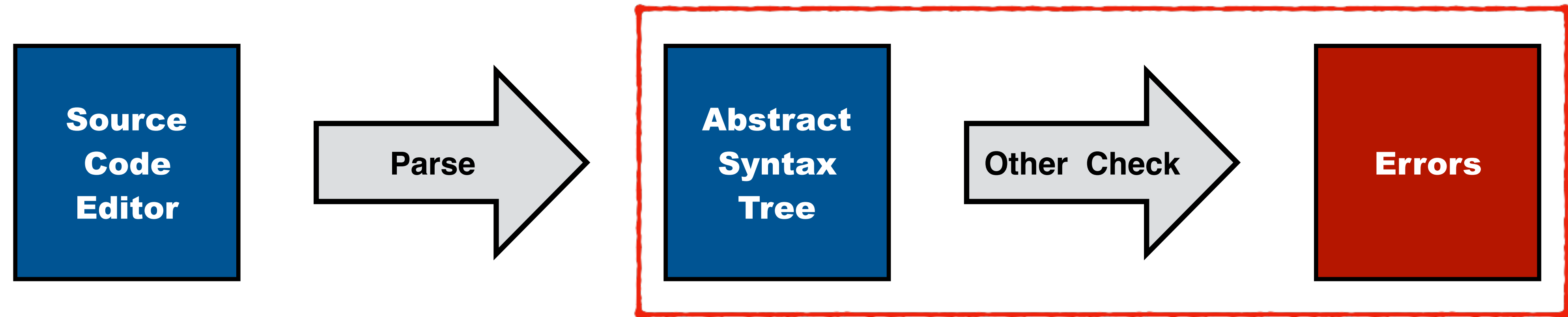
Declarative Rules

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

Language-Independent Tooling

- Data-Flow Analysis
- Code completion
- Refactoring
- Optimisation
- ...

Data-Flow



Check that code is reachable or observable

Data-Flow

What is Data-Flow?

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

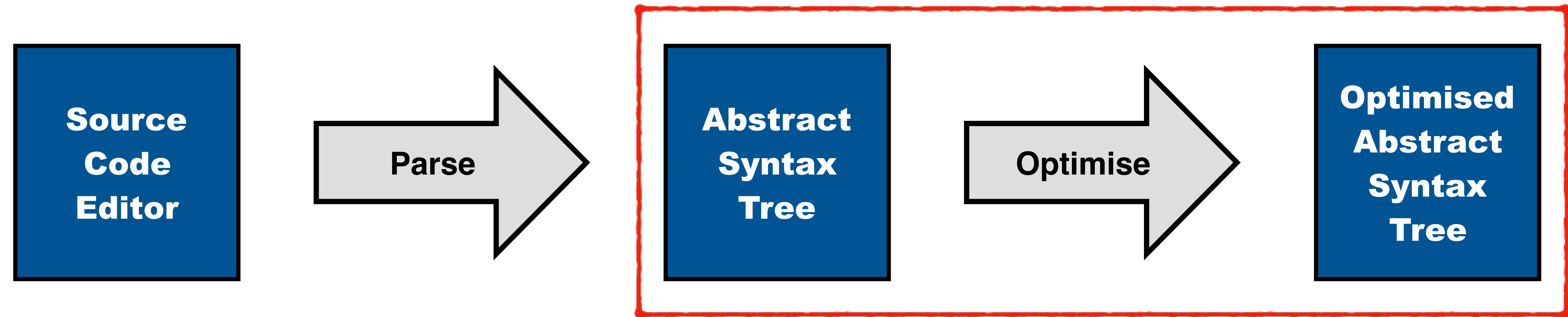
What is “wrong” here?

```
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



Eliminate common subexpressions, reduce loop strength, etc.

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

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

Tiger in FlowSpec

Separation of Concerns in Data-Flow Analysis

Representation

- **Control Flow Graphs**
- 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
- Code completion
- Refactoring
- Optimisation
- ...

Separation of Concerns in Data-Flow Analysis

Representation

- Control Flow Graphs
- 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
- Code completion
- Refactoring
- Optimisation
- ...

Control-Flow rules

Map abstract syntax to control-flow (sub)graphs

- Match an AST pattern
- List all edges of that AST
- Use special “context” nodes to connect a subgraph to the outside graph

Control-flow graphs in FlowSpec

FlowSpec

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

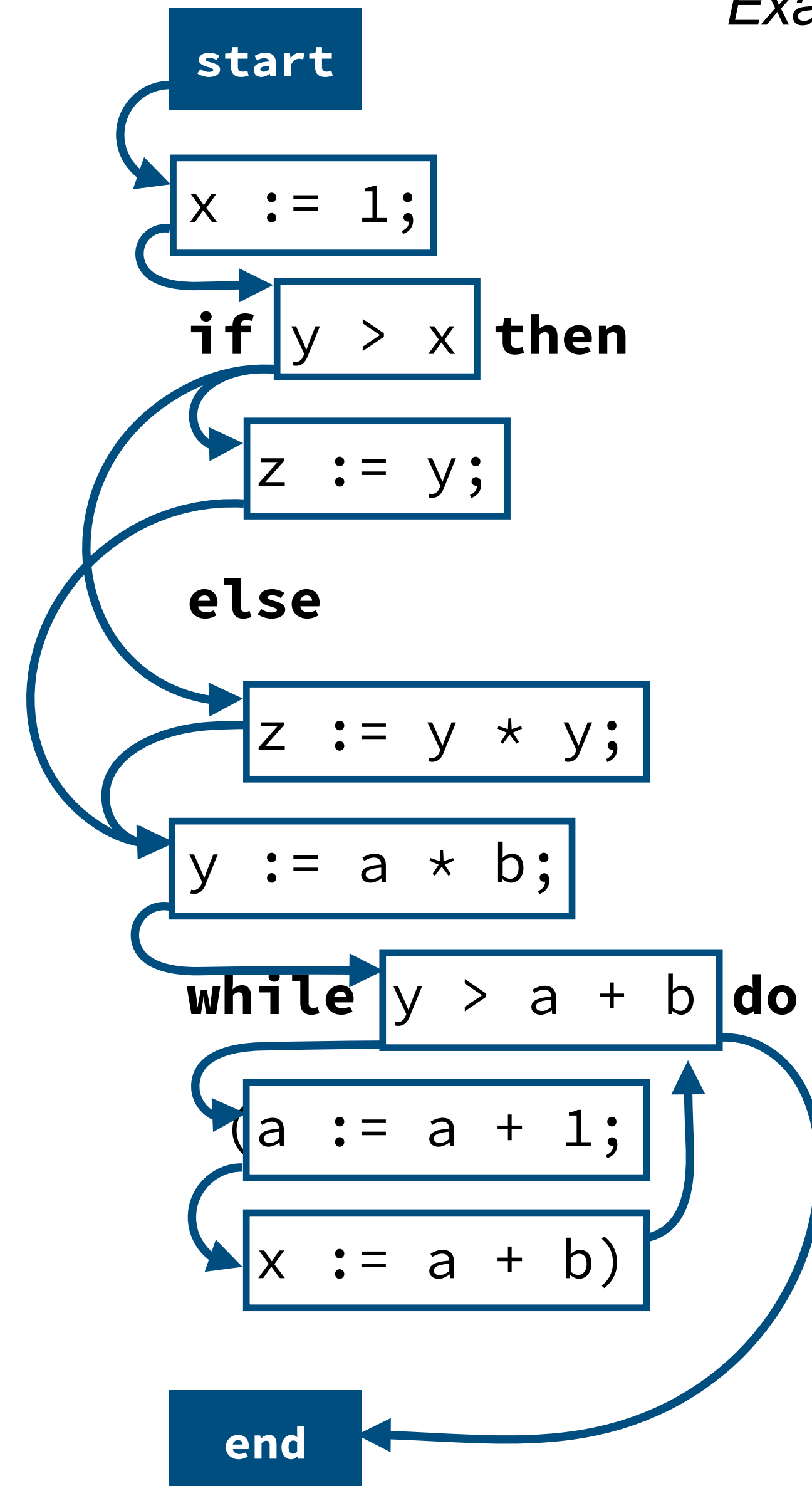
```
node Assign(_, _)
```

```
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 -> exit
```

Example program



Data-Flow rules

Define effect of control-flow 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

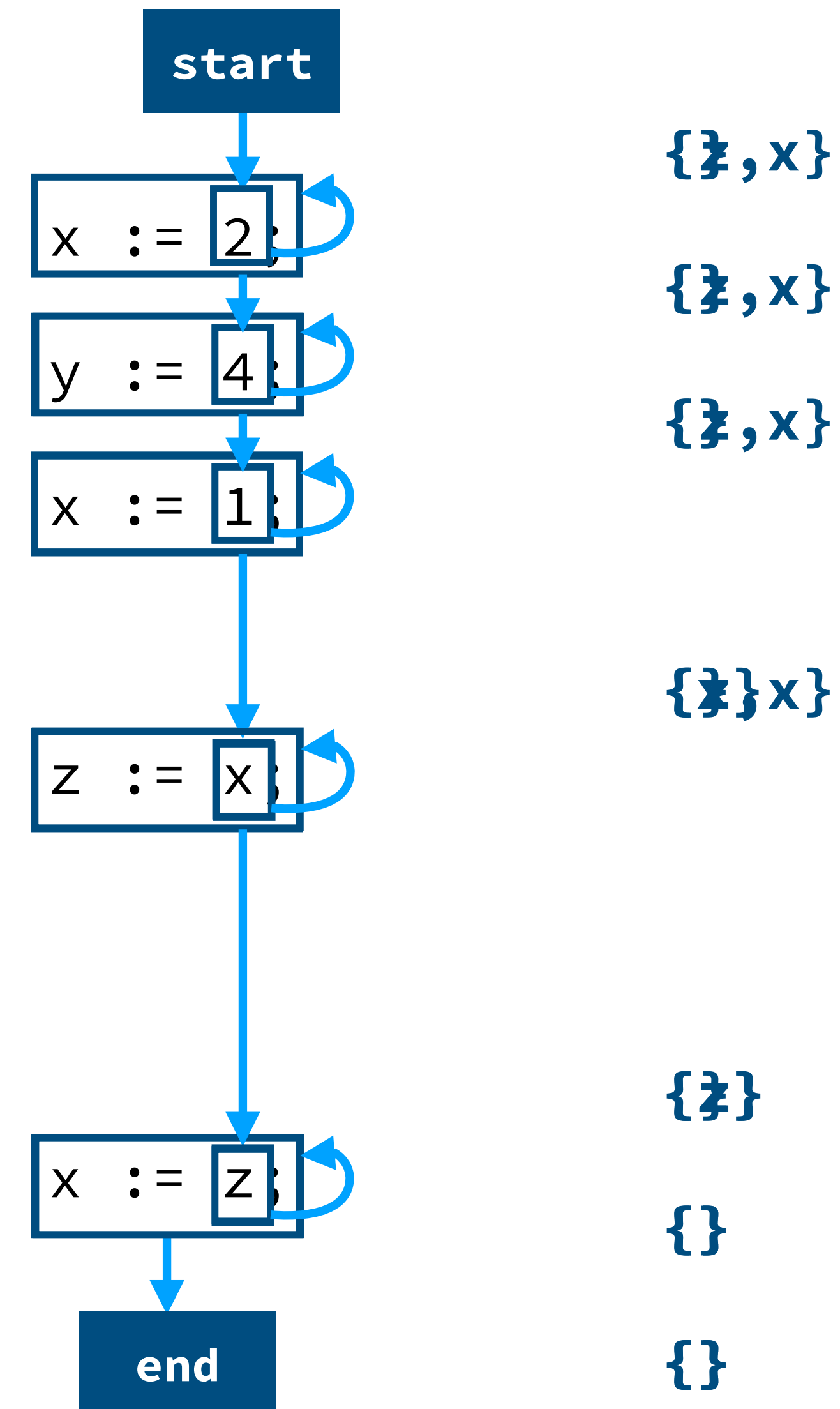
A variable is *live* if the current value of the variable *may* be read further along in the program

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

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

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

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



Live Variables in FlowSpec

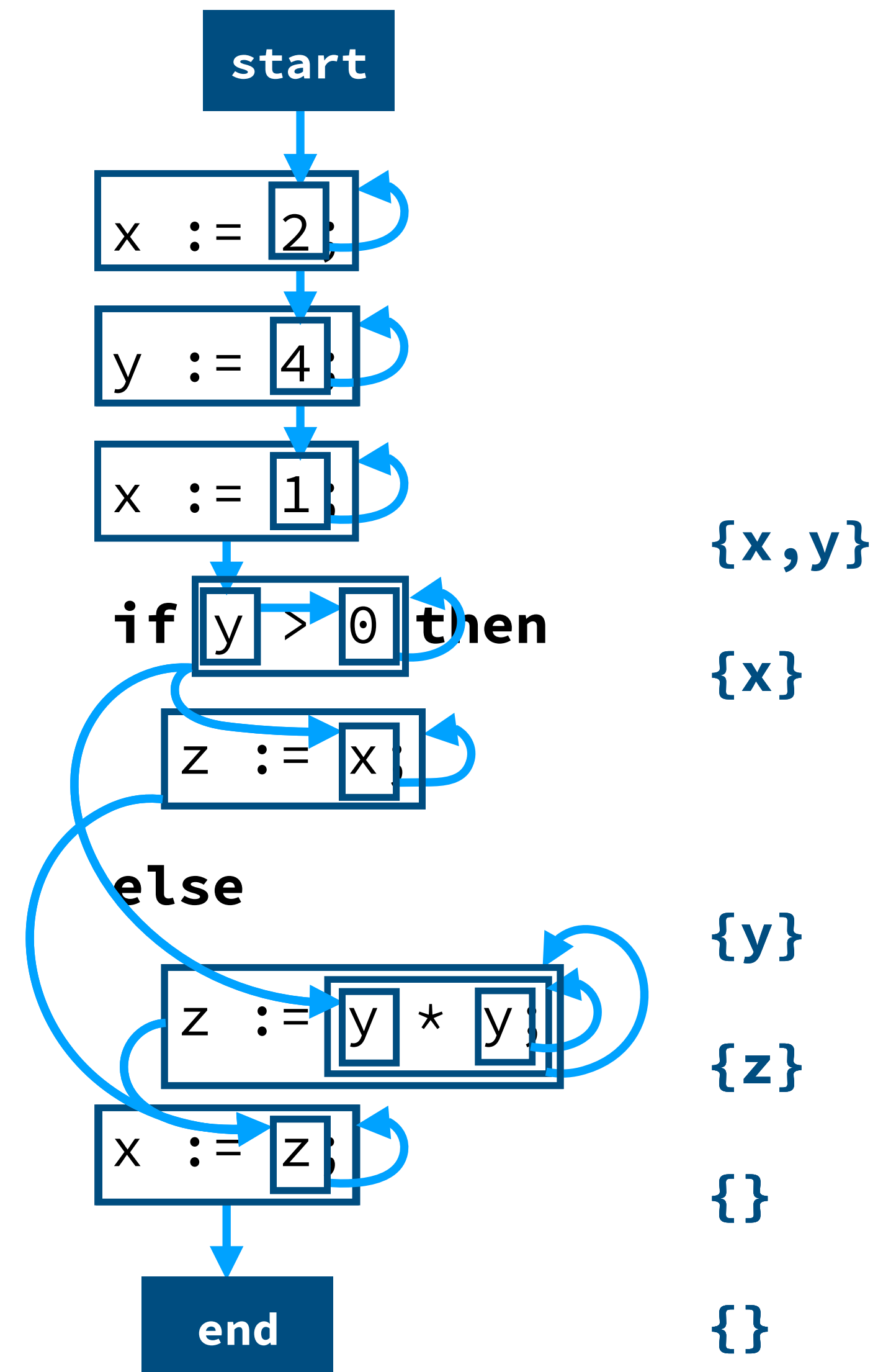
A variable is *live* if the current value of the variable *may* be read further along in the program

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

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

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

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



Conclusion

Summary

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

Except where otherwise noted, this work is licensed under

