Course 142A Compilers & Interpreters

Code Analysis

Lecture Week 5

Prof. Dr. Luc Bläser

Last Week - Quiz

```
a = 1;
if (…) {
  a = a + 1;
  b = a;
} else {
  b = 2;
}
c = b + 1;
writeInt(c);
```

How could this be optimized?

Constant Propagation

```
a = 1;
if (…) {
  a = a + 1;
  b = a;
} else {
  b = 2;
}
c = b + 1;
writeInt(c);
```

```
a = 1;
if (…) {
  a = 2;
  b = 2;
} else {
  b = 2;
}
c = 3;
writeInt(c);
```

Copy Propagation

```
…
c = 3;
writeInt(c);
```

```
…
c = 3;
writeInt(3);
```

Dead Code Elimination

```
a = 1;
if (…) {
  a = 2;
  b = 2;
} else {
  b = 2;
}
c = 3;
writeInt(3);
```

```
writeInt(3);
```

How to Determine Optimizations?

- Static code analysis = at compile time
  - Statement about all possible runtime cases
- Facilitates optimizations
  - Constant propagation
  - Dead code elimination
  - …
- But also error detection
  - Reading uninitialized variables
  - Certainly failing runtime checks
  - …

Today's Topic

- Code Analysis
- Control Flow Graphs
- Dataflow Analysis
- Use Cases

Learning Goals

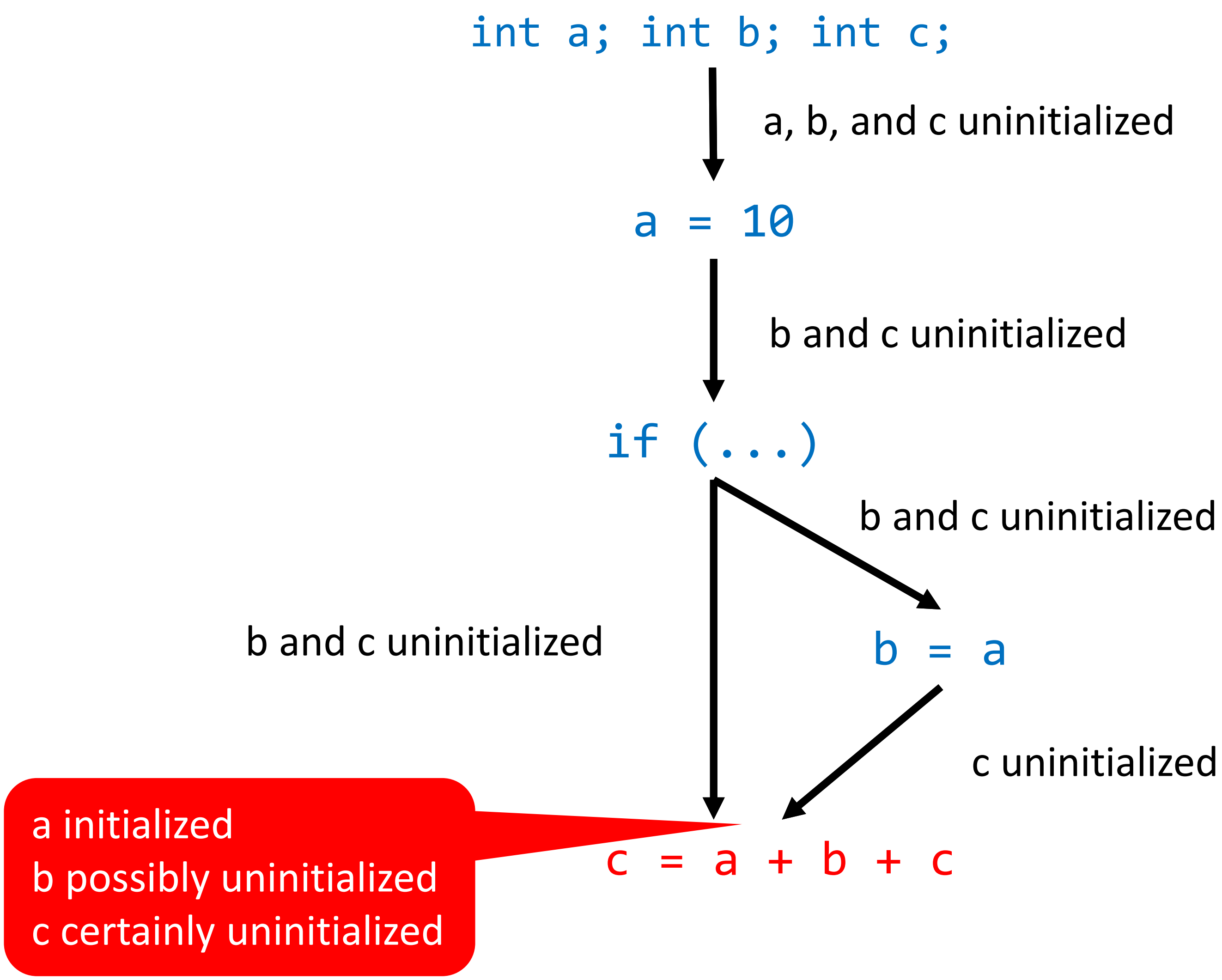
- Understand dataflow analysis as a generic code analysis method
- Know how to apply it for error detection and optimizations in a compiler

First Analysis Example

```
int a;
int b;
int c;
a = 10;
if (...) {
  b = a;
}
c = a + b + c;
```

Where do we read uninitialized variables?
How to detect this?

Analyze All Paths

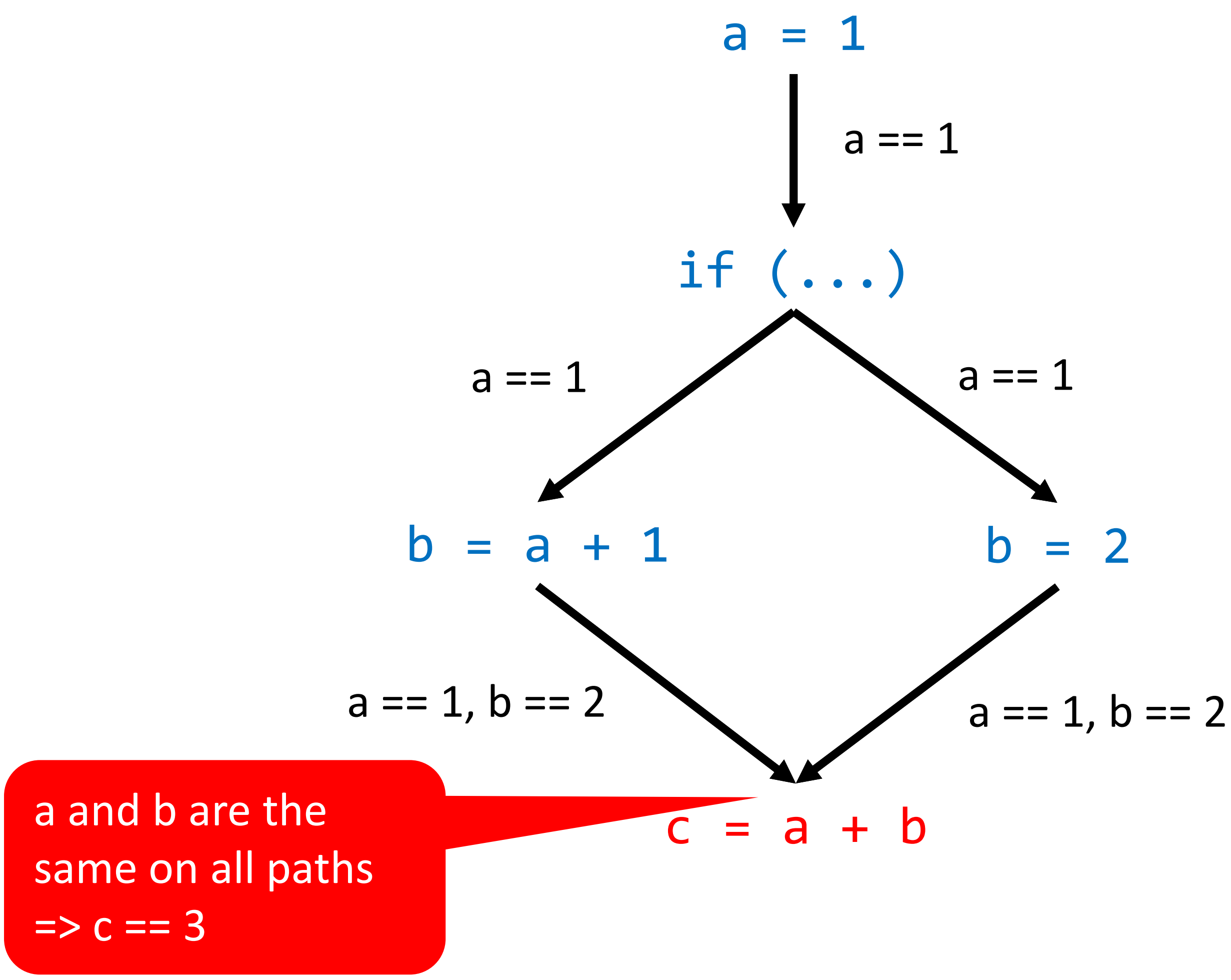


Second Analysis Example

```
a = 1;
if (...) {
  b = a + 1;
} else {
  b = 2;
}
c = a + b;
```

Is the value of c constant?
If yes, what is its value?

Analyze All Paths

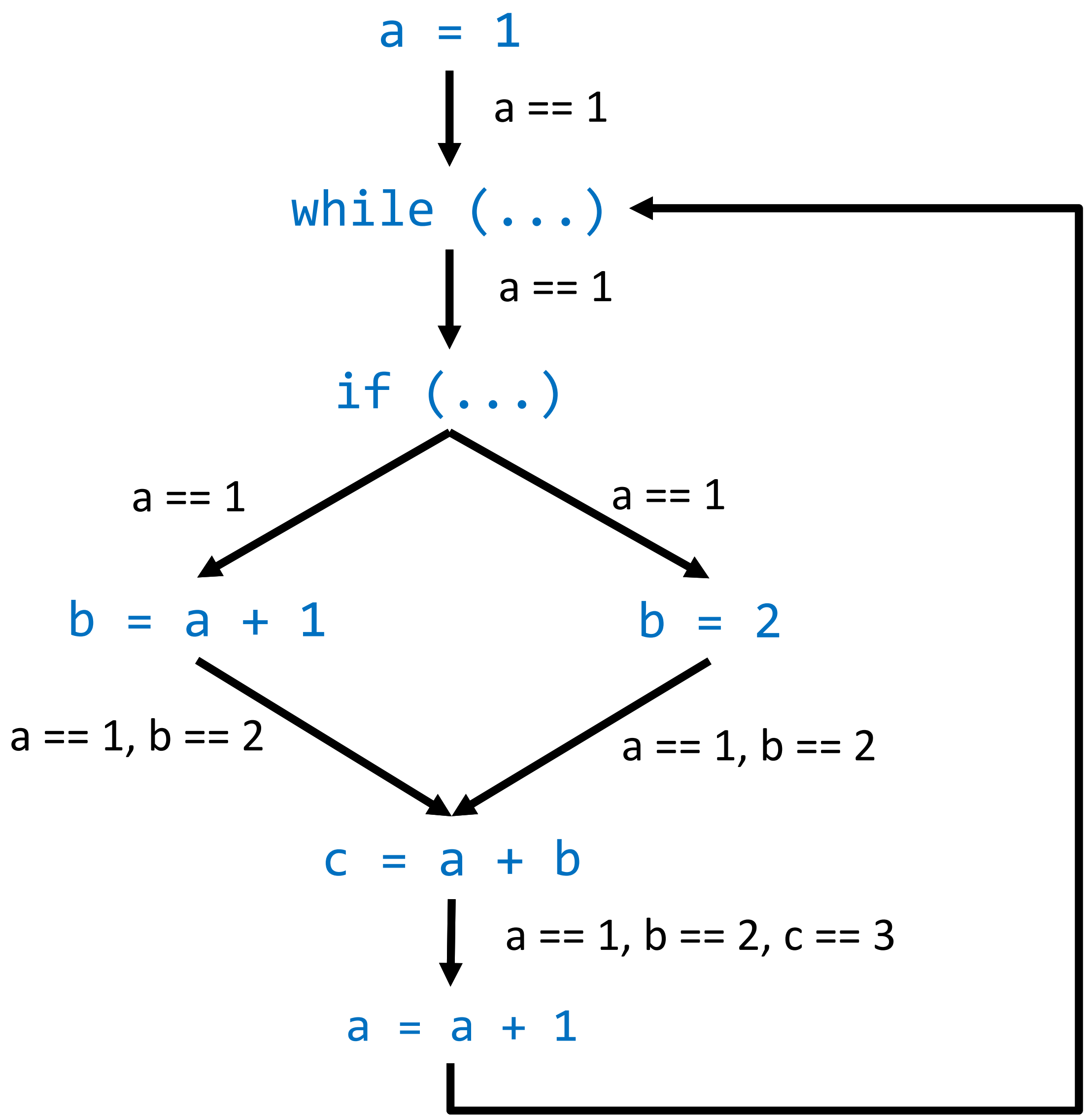


Small Modification

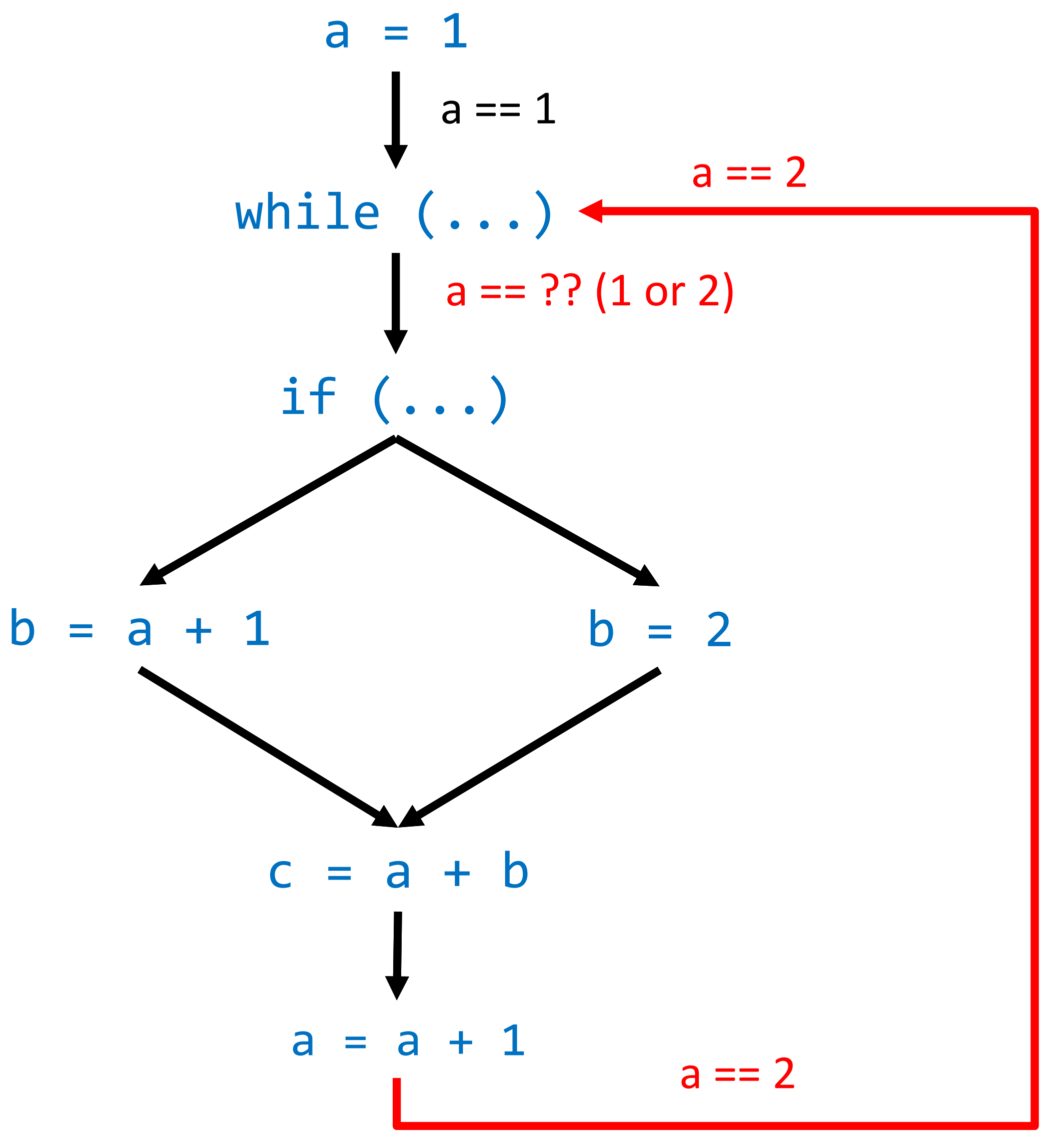
```
a = 1;
while (...) {
  if (...) {
    b = a + 1;
  } else {
    b = 2;
  }
  c = a + b;
  a = a + 1;
}
```

Is c still constant?

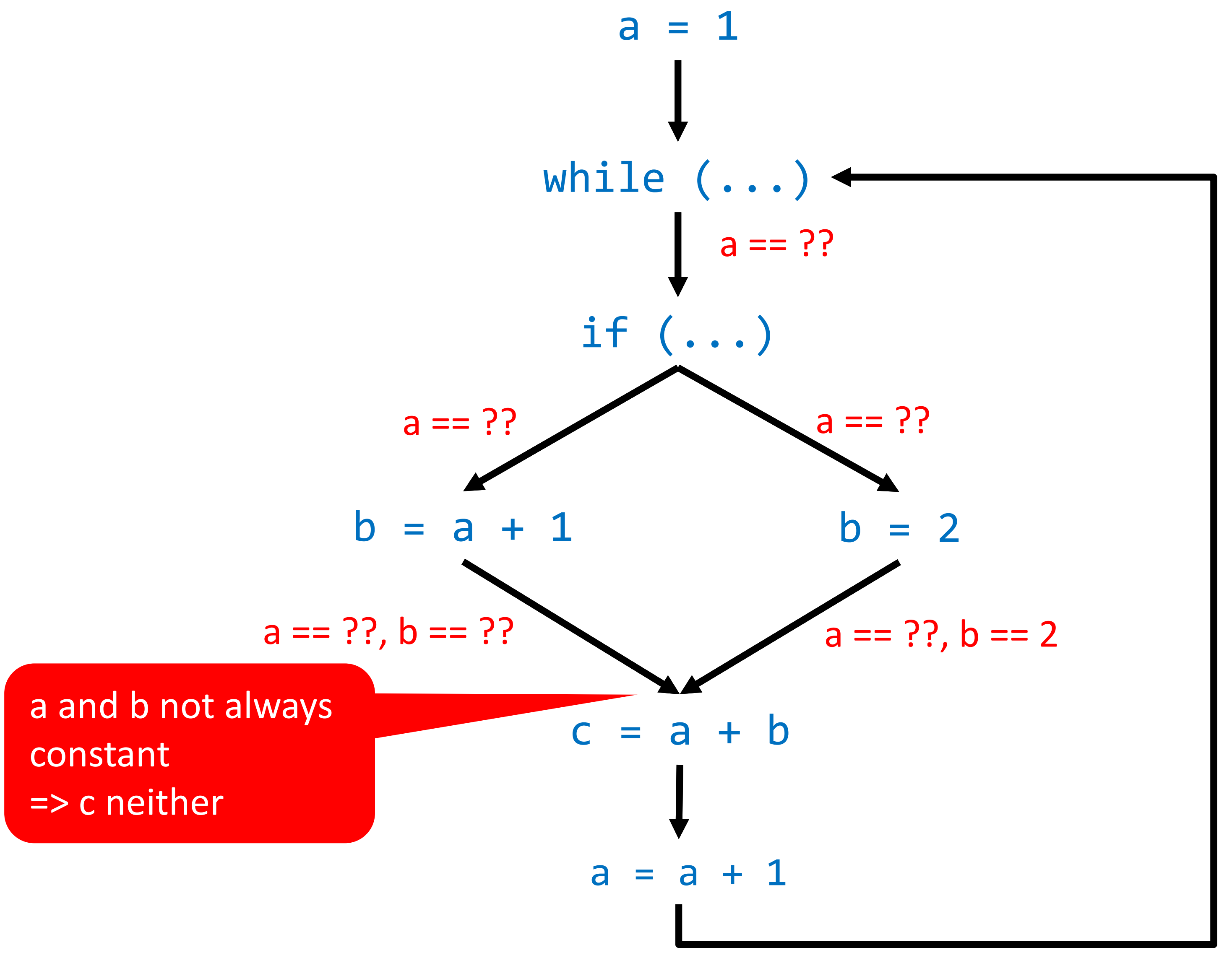
Analyze All Paths



Analyze All Paths



Analyze All Paths



Our Approach

- Construct a Control Flow Graph
  - Graph showing program paths like on previous slides
- Dataflow Analysis
  - Propagate information through graph, until it is stable

Control Flow Graph

- Representation of all possible program paths
  - Typically inside a method (intra-method)
- Node = Basic Block
  - Straight code section
  - Entry only at the beginning: No label in the middle
  - Exit only at the end: No branch in the middle
- Edge
  - Conditional or unconditional branch

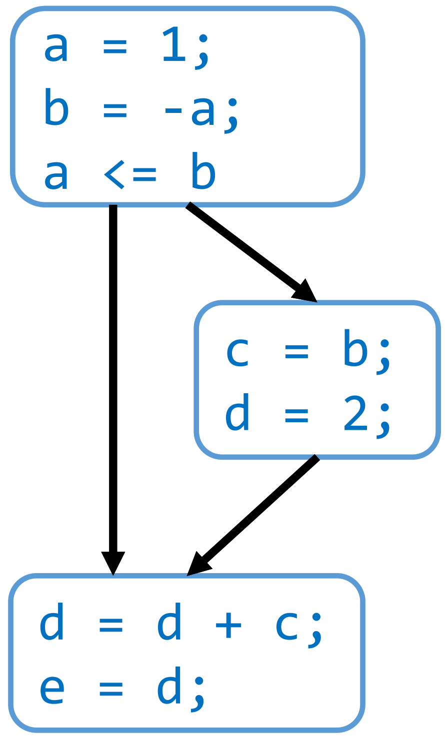
Basic Blocks

- Delimited by branch entry and exit

```
a = 1;
b = -a;
if (a <= b) {
   c = b;
   d = 2;
}
d = d + c;
e = d;
```

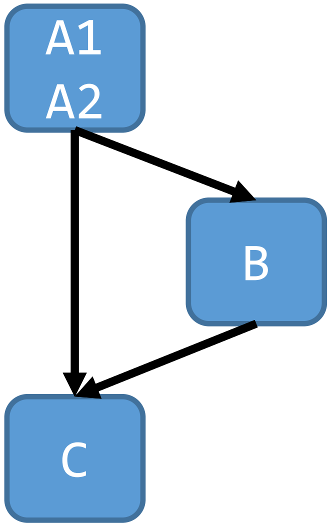
Control Flow Graph (CFG)

- Connected basic blocks according to possible branches

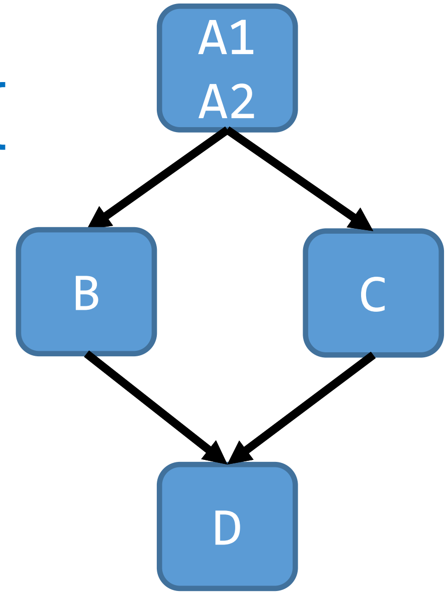


CFG for If-Statement

```
A1;
if (A2) {
  B;
}
C;
```

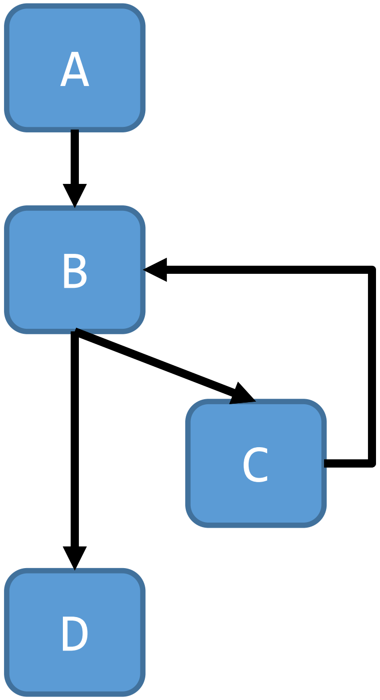



```
A1;
if (A2) {
  B;
} else {
  C;
}
D;
```



CFG for While-Statement

```
A;
while (B) {
  C;
}
D;
```

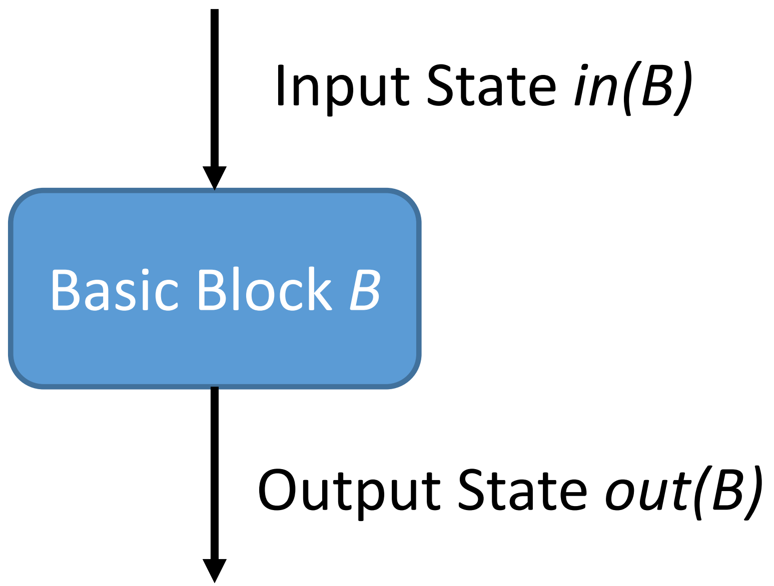


Dataflow Analysis

- Fixpoint iteration over Control Flow Graph
  - Propagate analysis information (state) through blocks
  - Until state is stable for each block (fixpoint)
- Dataflow Analysis is generic
  - Applicable for various use cases
  - State will be defined per use case

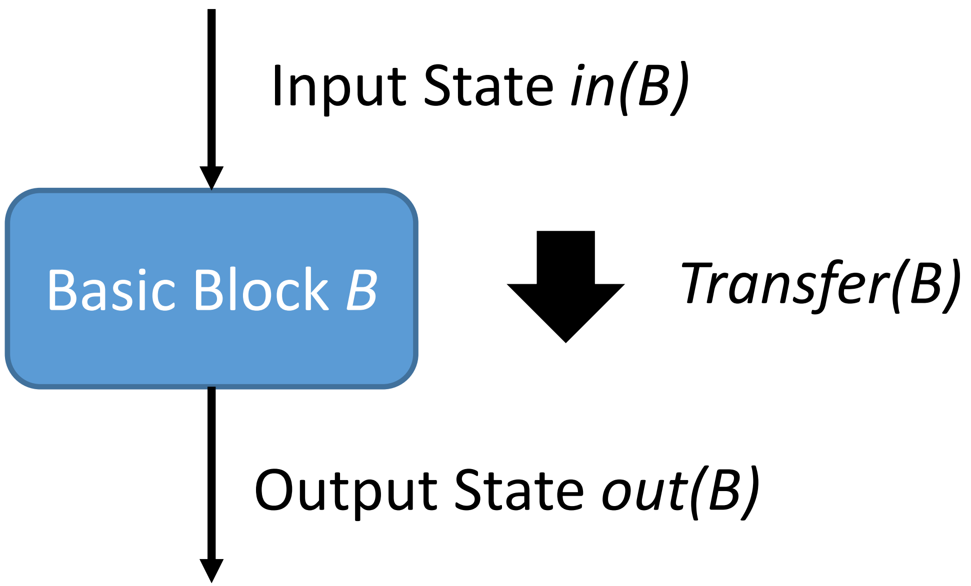
State

- Input state und output state per basic block
- Analysis information before and after the block



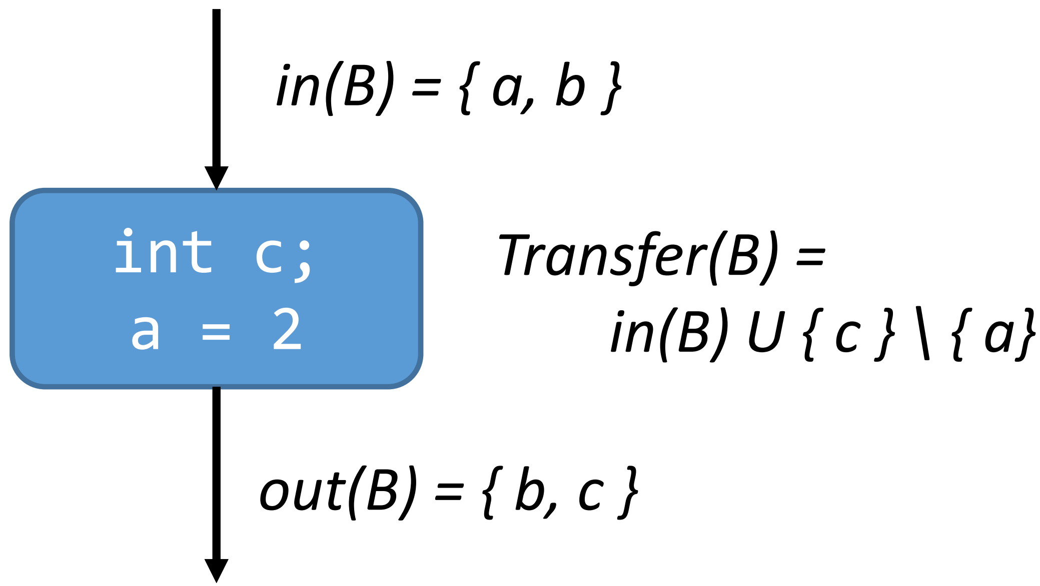
Transfer

- Mapping per block: Input State -> Output State
- Defines block's effect on analysis information



Analysis Example

- State = Set of uninitialized variables
- Transfer = Add variable declarations, remove assigned variables



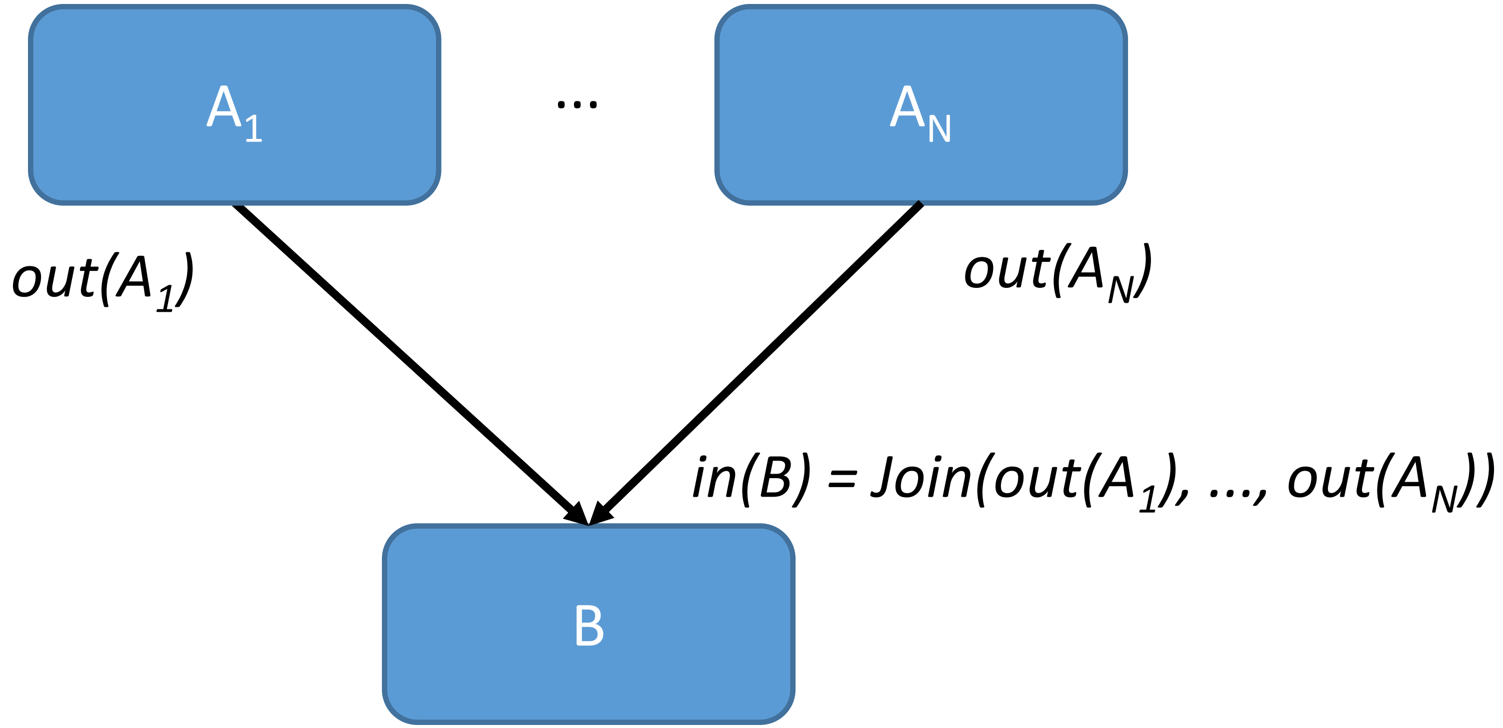
Gen and Kill Set

- Transfer can be described by two sets
- Gen Set: Elements to add
  - Example: Variable declarations
- Kill Set: Elements to remove
  - Example: Assigned variables

$$Transfer(B) = in(B) \cup gen(B) \setminus kill(B)$$

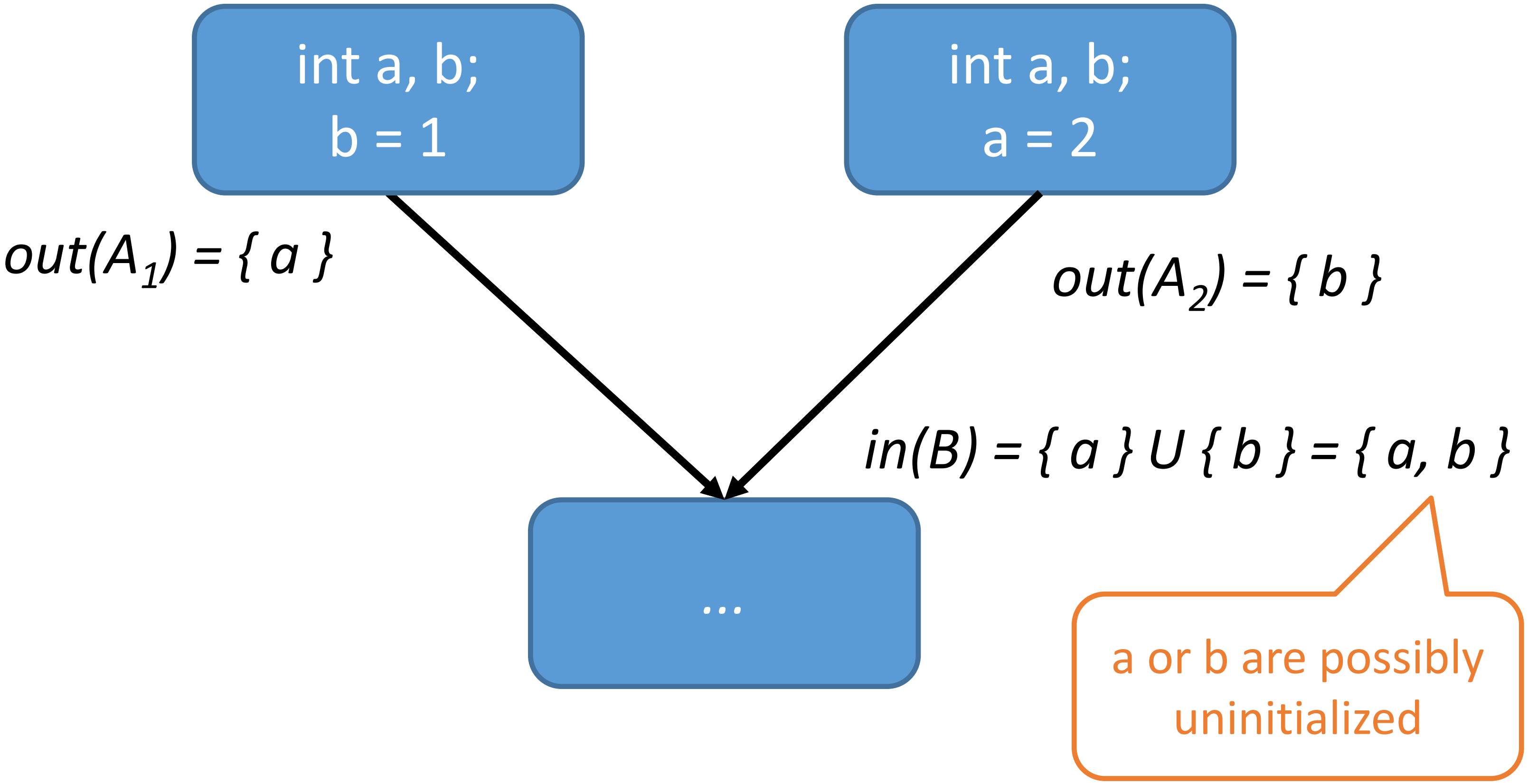
Join

- Combine output state of predecessors to input state of the successor



Analysis Example

- Join = Union set of predecessors



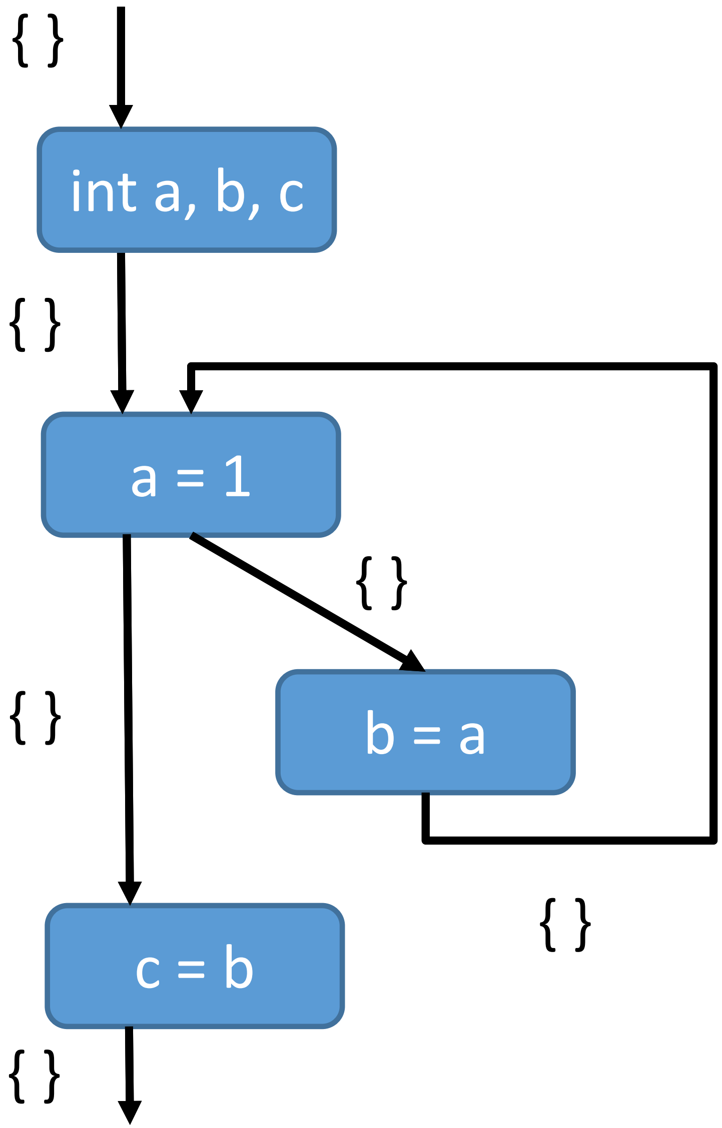
Dataflow Analysis

```
boolean stable;
do {
  stable = true;
  for (var block : graph.allBlocks())  {
      in[block] = join(block.predecessors().outStates());
      var oldOut = out[block];
      out[block] = transfer(in[block]);
      if (!out[block].equals(oldOut)) {
         stable = false;
      }
  }
} while (!stable);
```

Fixpoint iteration

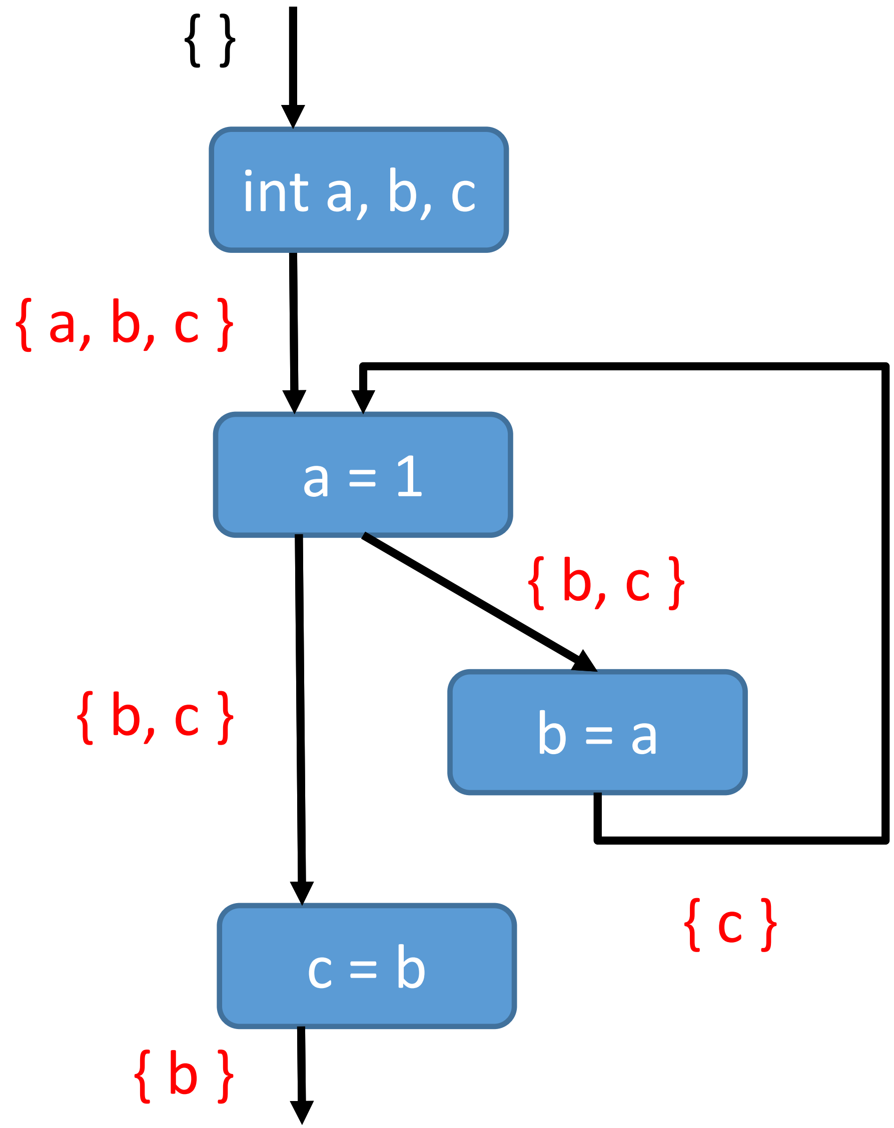
Uninitialized Variable Analysis

Initialize
dataflow



Uninitialized Variable Analysis

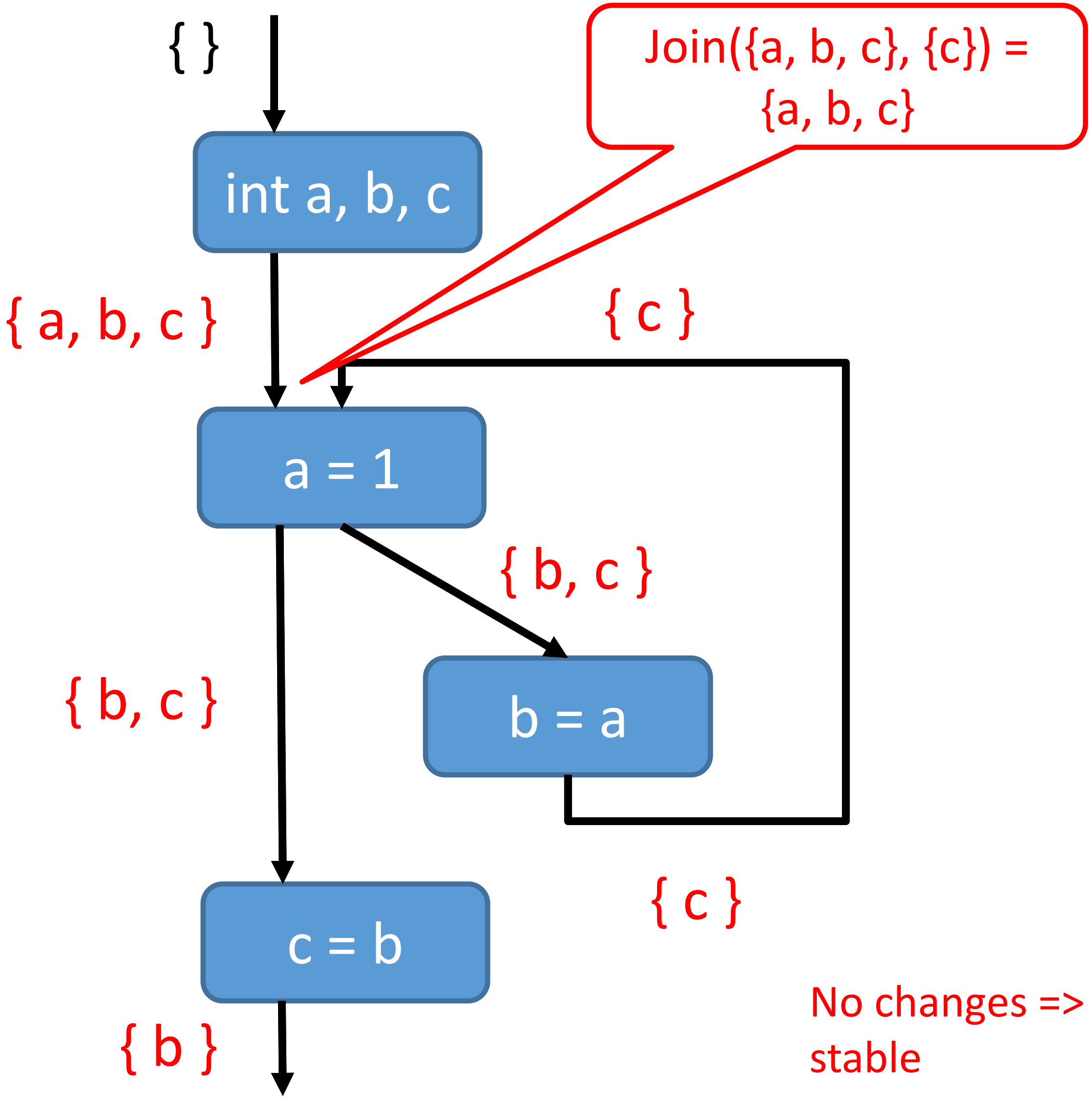
First round



Changes =>
Not yet stable

Uninitialized Variable Analysis

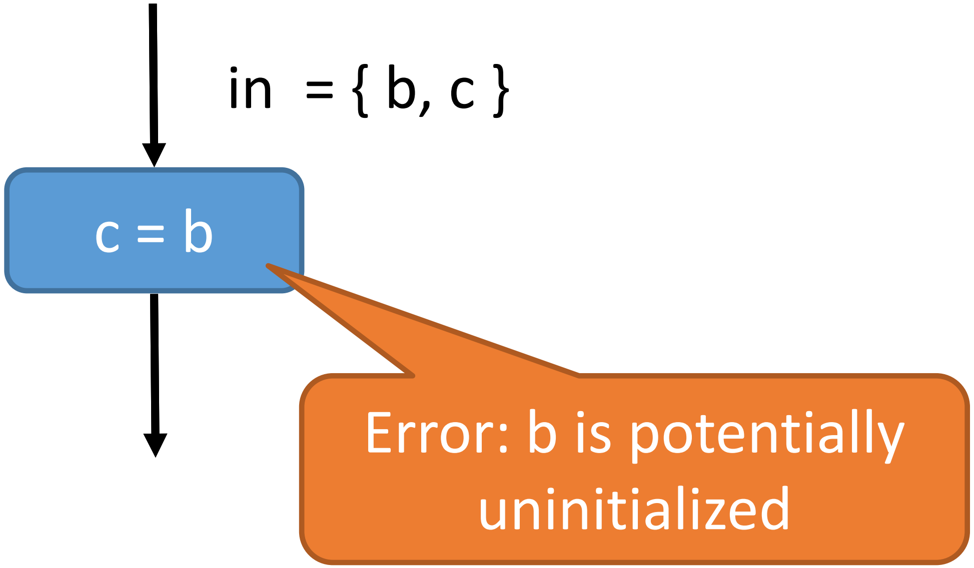
Second round



No changes =>
stable

Results After Dataflow Analysis

- Use stable input or output states for block properties
- E.g. compiler error for uninitialized reads



Discussion

- Conservative analysis
  - Considers all possible syntactic paths
- Context-free analysis
  - All paths are selected, regardless of their condition
- Error reporting is also conservative
  - It exists at least one path where error could occur
- Fixpoint iteration needs to terminate
  - E.g. if set grows monotonically by joins

Error <=> potentially uninitialized

No error <=> certainly initialized

Revisiting Constant Propagation

- Example of last week

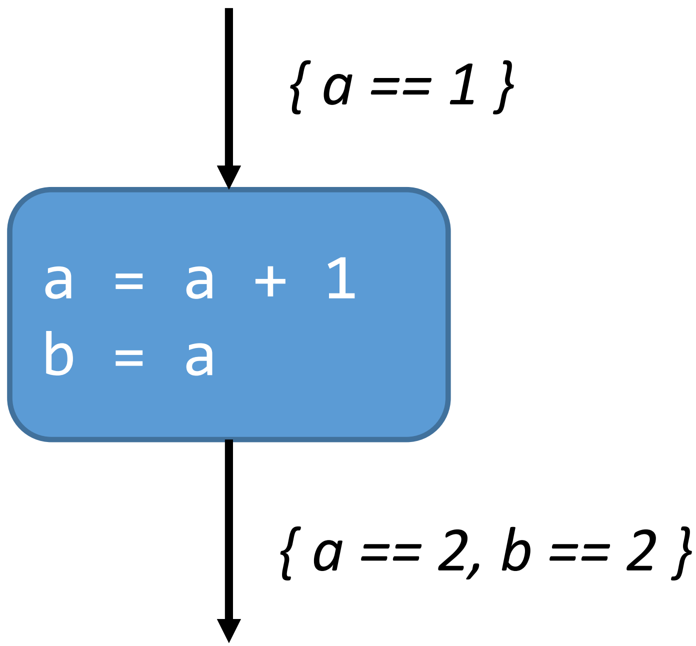
```
a = 1;
if (...) {
    a = a + 1;
    b = a;
} else {
    b = 2;
}
c = b + 1;
```

c is certainly 3

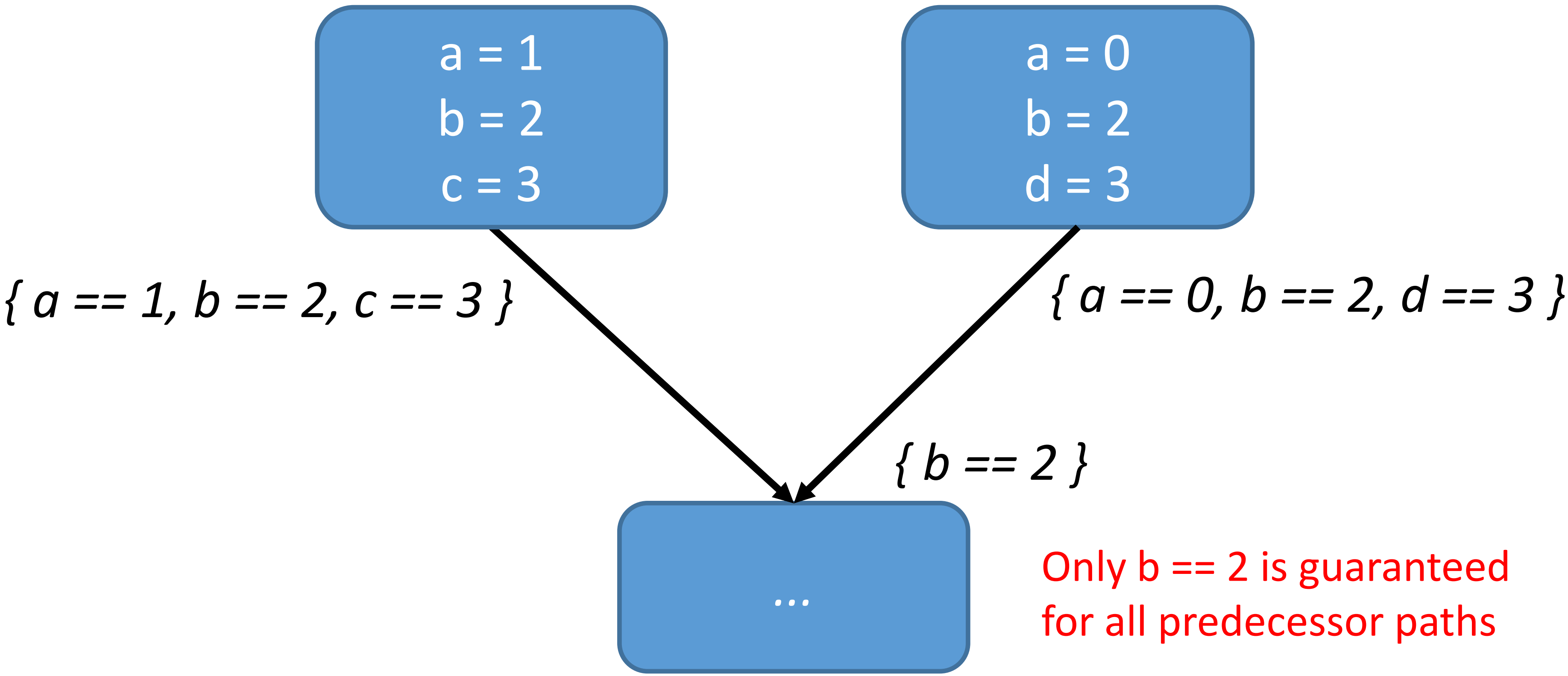
Configuring the Analysis

- State: Variables with their associated constant value
  - { a == const1, b == const2, ... }
- Transfer for a = E
  - Kill: Remove a == ... (if existing)
  - Gen: If E is constant, add a == E
- Join(X, Y) = $X \cap Y$
  - Intersection

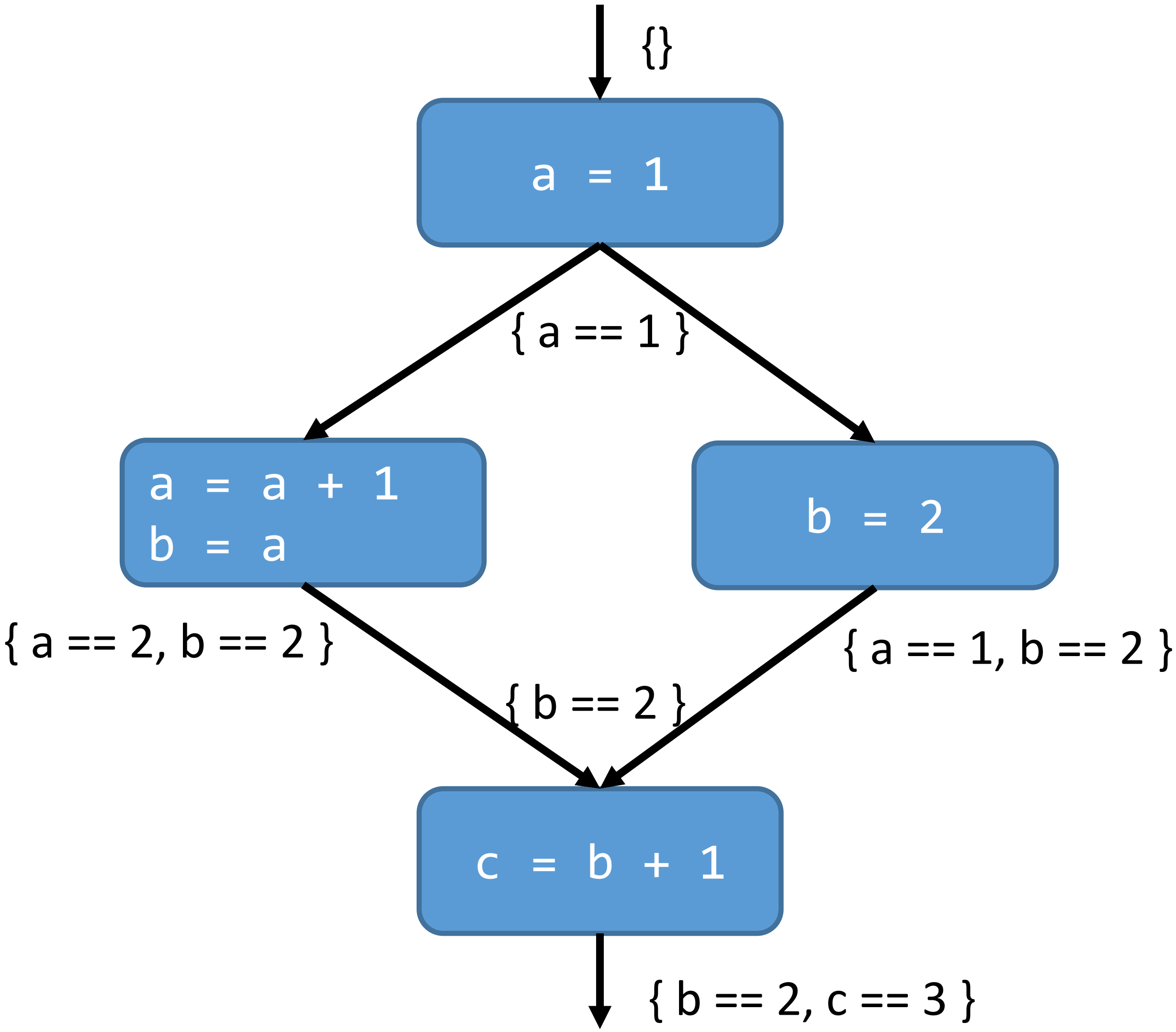
Constant Propagation: Transfer



Constant Propagation: Join



Constant Propagation Example



Review: Learning Goals

- ✓ Understand dataflow analysis as a generic code analysis method
- ✓ Know how to apply it for error detection and optimizations in a compiler

Continued in the next lecture

Further Reading

- Dragon Book, Code Optimization
  - Section 9.2-9.3: Dataflow analysis
  - Section 9.4: Constant propagation
- Optional, if interested
  - F. Nielson, H. R. Nielson, C. Hankin. Principles of Program Analysis, Springer, 2004.

Course 142A Compilers & Interpreters
Code Analysis Continued

Lecture Week 5, Wednesday

Prof. Dr. Luc Bläser

Quiz - Last Lecture

Constant Propagation

```
a = 1;
while (...) {
  if (...) {
     a = a + 1;
     b = a;
  } else {
     b = 2;
  }
  c = b + 1;
}
```

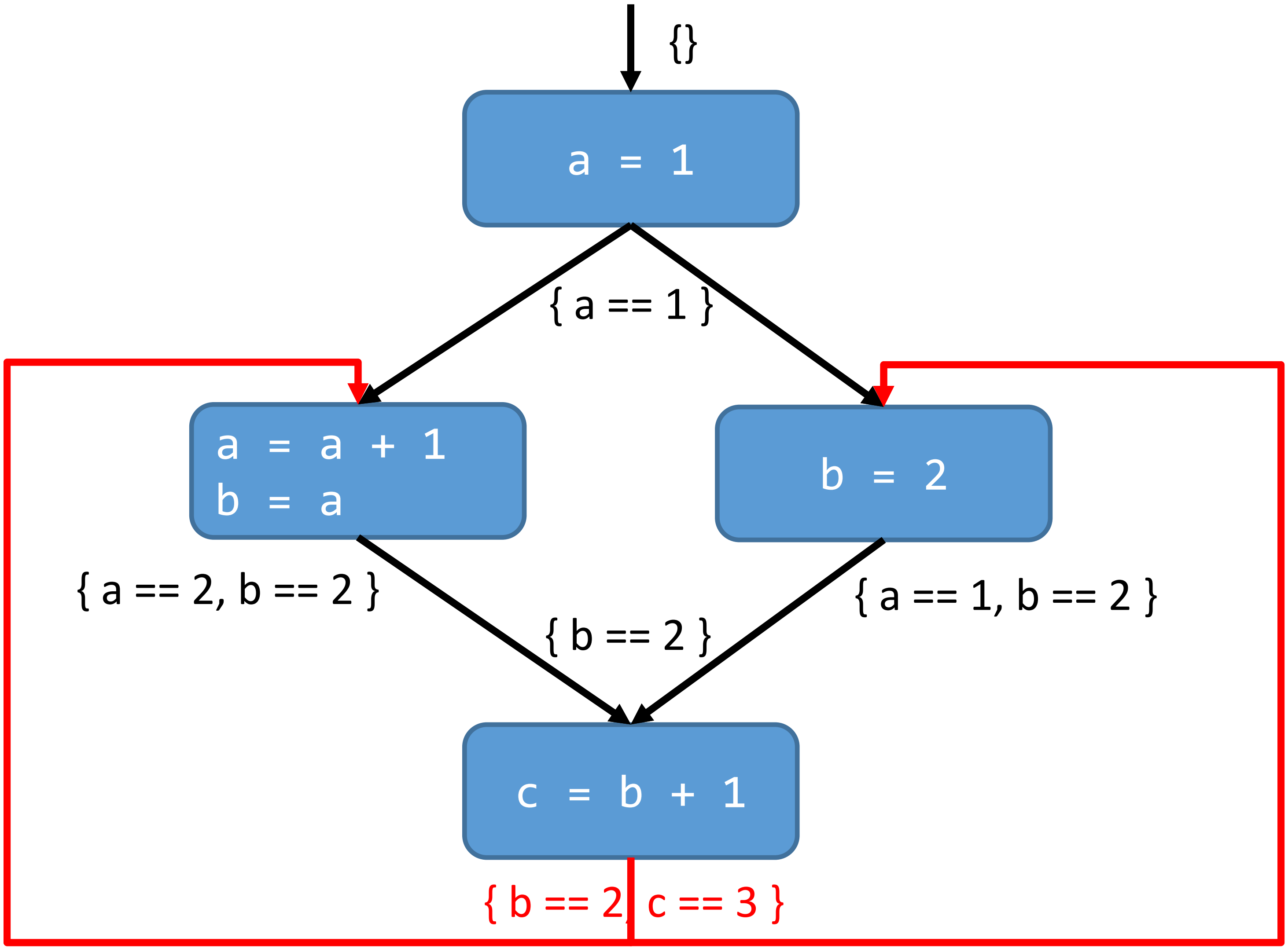
How do we configure the dataflow analysis?

- State, Transfer, Join

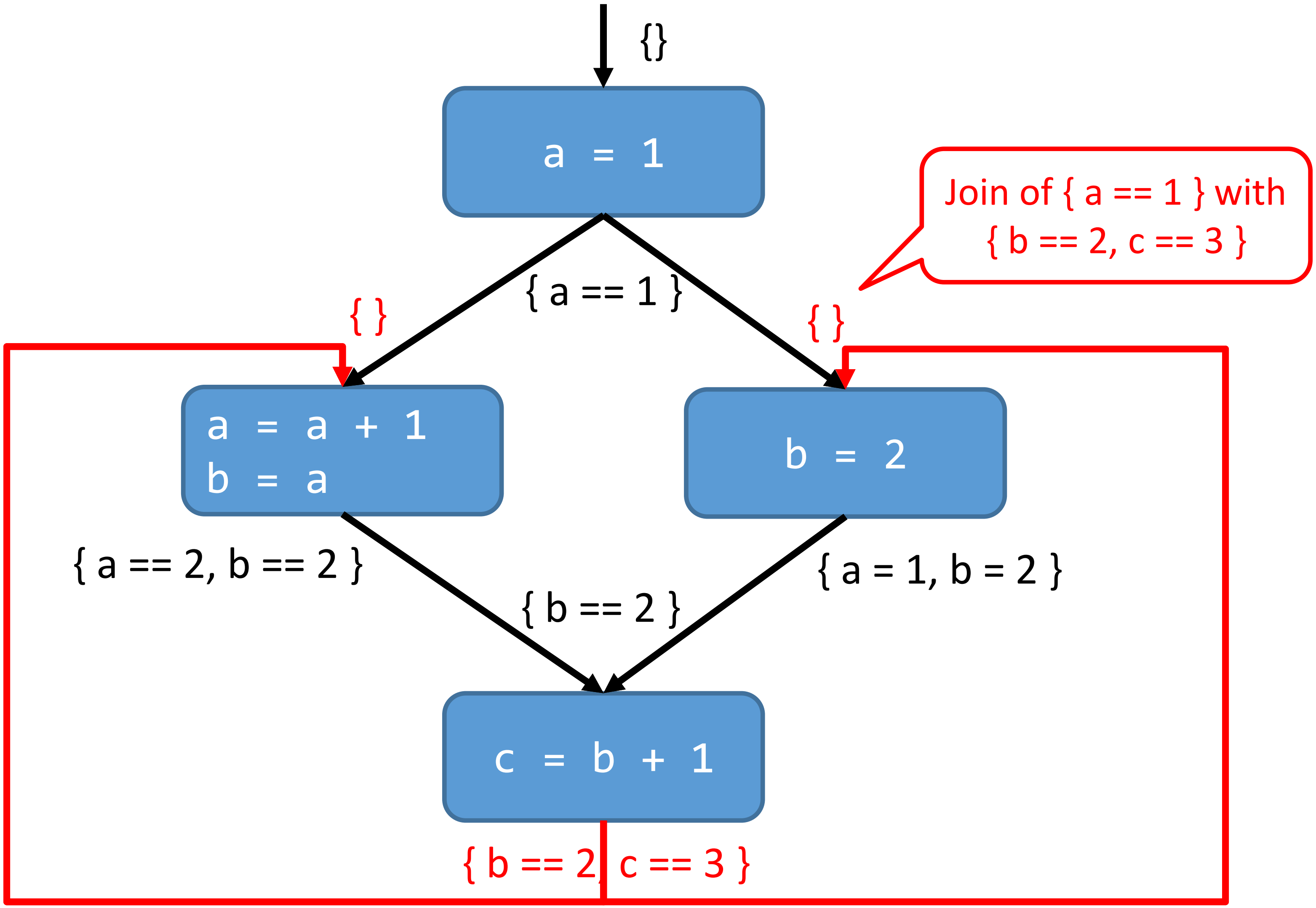
Today's Agenda

- Continue Dataflow Analysis
- Midterm Q&A

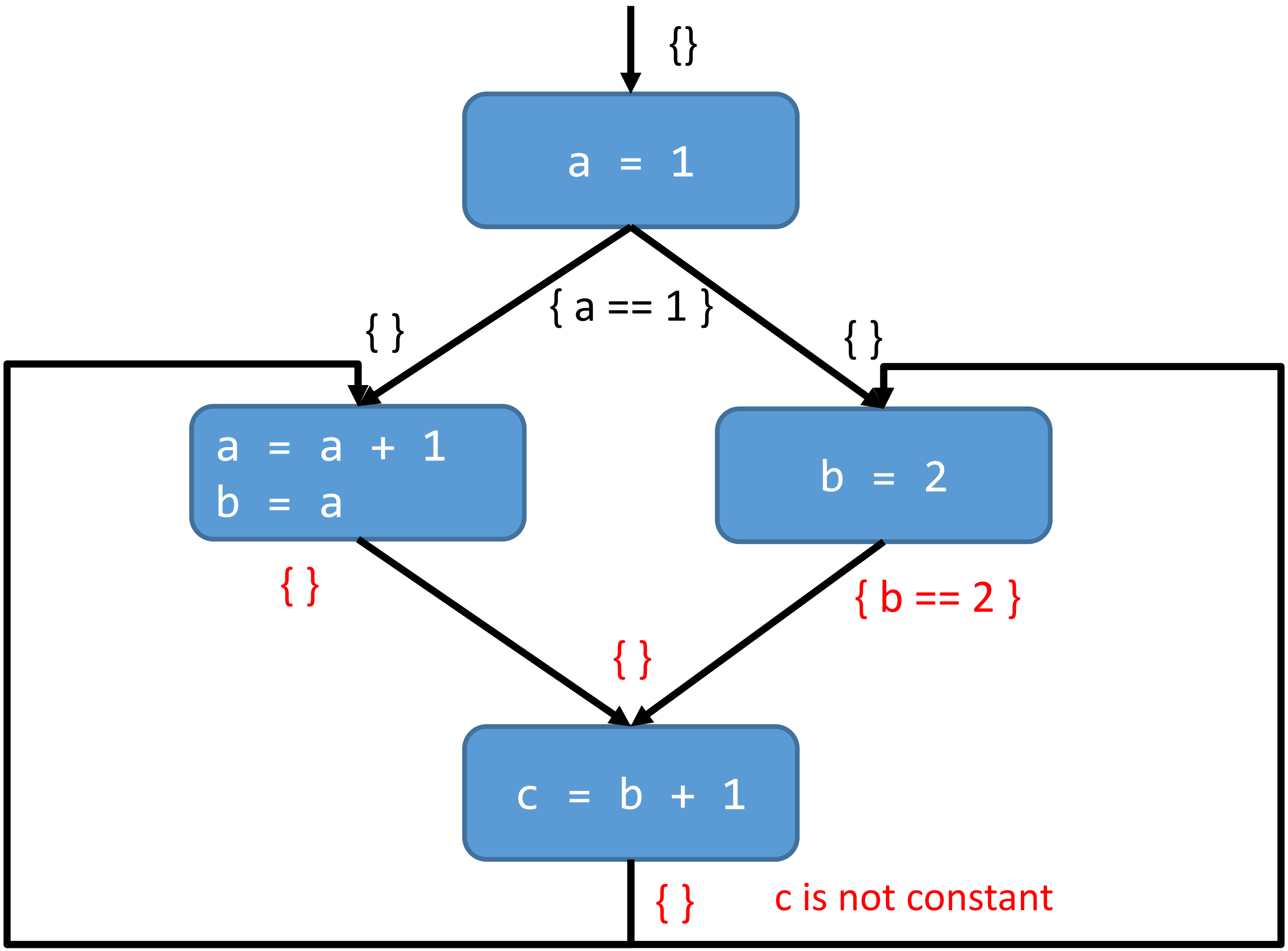
Constant Propagation Scenario



Dataflow Scenario



Dataflow Scenario



Backwards Analysis

- Dataflow analysis can also run backwards: From successors to predecessors
- Transfer: Out State -> In State
- Join: In states of successors -> out state of predecessor

Example: Live Variables

- Variables that may be used later
- Eliminate non-live variables (=dead)

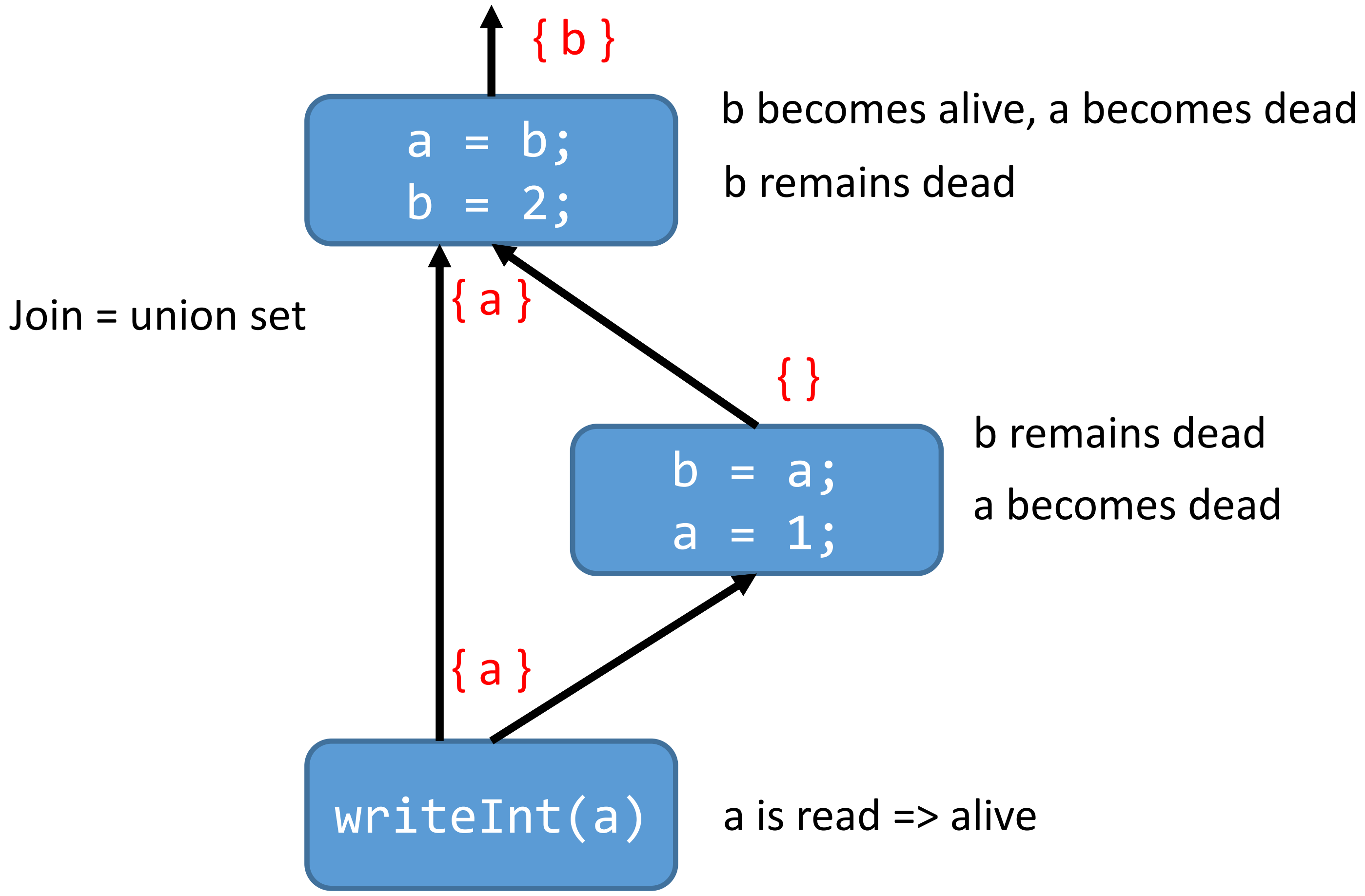
```
a = b;
b = 2;
if (...) {
  b = a;
  a = 1;
}
writeInt(a);
```

b is hereafter
no longer used

Live Variable Configuration

- Backwards direction
- State: Live variables (variables that may be read)
- Transfer for a = b ● c:
  - Gen: If a is alive in succeeding code, so are also b and c
  - Kill: a is no longer alive
- Join(X, Y) = X U Y

Backwards Analysis



Dead Code Elimination

- Remove assignments to non-live (dead) variables

```
a = b;
b = 2;
if (...) {
  b = a;
  a = 1;
}
writeInt(a)
```

```
a = b;
if (...) {
  a = 1;
}
writeInt(a)
```

Summary

- Dataflow is generic
- Allows different configurations

	Join = union	Join = intersection
Forward	Uninitialized Variables	Constant Propagation
Backward	Live Variables	(Busy Expressions)

Expressions, that are computed on all paths

Further Reading

- Dragon Book, Code Optimization
  - Section 9.2-9.3: Dataflow analysis
  - Section 9.4: Constant propagation
- Optional, if interested
  - F. Nielson, H. R. Nielson, C. Hankin. Principles of Program Analysis, Springer, 2004.