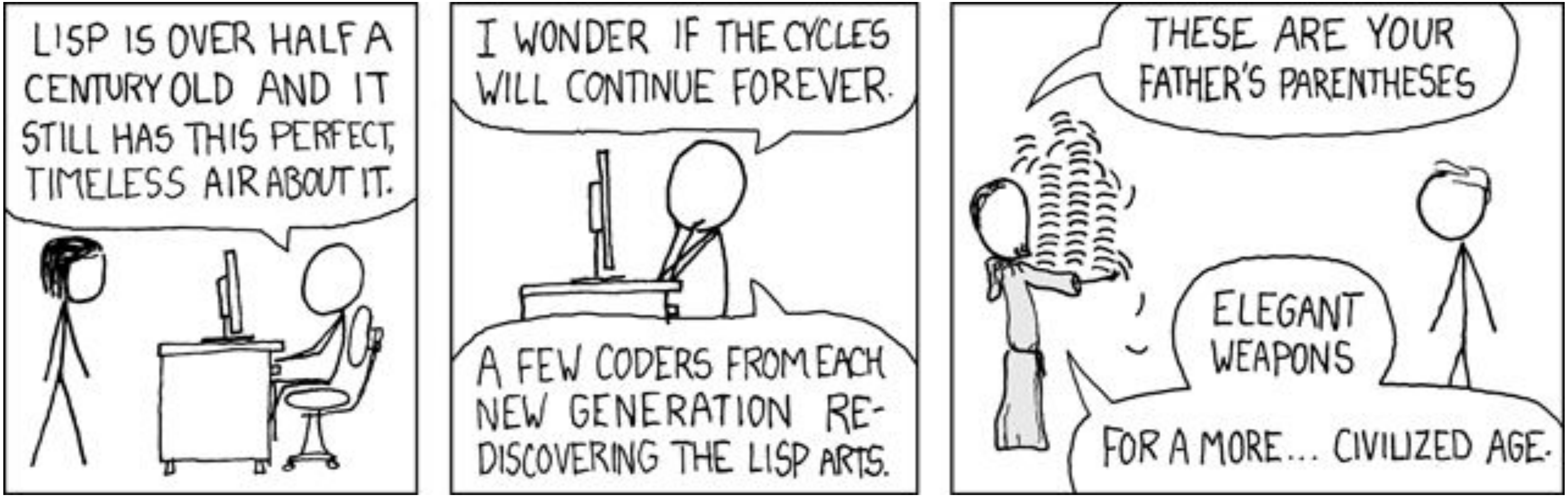
6.110 Computer Language Engineering

Recitation 5: Introduction to SSA

Feb 28, 2025

Weekly updates ←

Introduction to SSA



[https://xkcd.com/297/]

Coming up soon...

Mon 3/3	Tue 3/4	Wed 3/5	Thu 3/6	Fri 3/7
No class! 😋 (plz work on ur phase 2)			**R6** x86	**R7** Phase 3

Phase 2 DUE

Weekly updates

Project phase 2 is due **Friday, Mar 7**

We are grading your phase 1 reports and repo

Expect individualized feedbacks on your code

- Code style suggestions
- Comment on design choices
- **Correctness bugs** that will bite you in the long run

Attention **ANTLR users**

Especially if ChatGPT wrote your grammar …

and/or it looks like this:

```
expr: ... |
   expr bin_op expr |
   ... ;

bin_op: arith_op | rel_op | eq_op | cond_op;
```

Your grammar **does NOT handle precedence**

Lexer Parser Sample ?

```
parser grammar DecafParser;
options { tokenVocab=DecafLexer; }



expr: location |
      method_call |
      literal |
      INT LEFT_PAREN expr RIGHT_PAREN |
      LONG LEFT_PAREN expr RIGHT_PAREN |
      LEN LEFT_PAREN ID RIGHT_PAREN |
      expr bin_op expr |
      MINUS expr |
      EXCLAMATION expr |
      LEFT_PAREN expr RIGHT_PAREN;

extern_arg: expr | STRING_LIT;

bin_op: arith_op | rel_op | eq_op | cond_op;
arith_op: MULTIPLY | DIVIDE | MODULO | PLUS | MINUS;
rel_op: LESS_THAN | GREATER_THAN | LESS_THAN_EQUAL | GREATER_TH
eq_op: COMP_EQ | NOT_EQ;
cond_op: LOGICAL_AND | LOGICAL_OR;
```

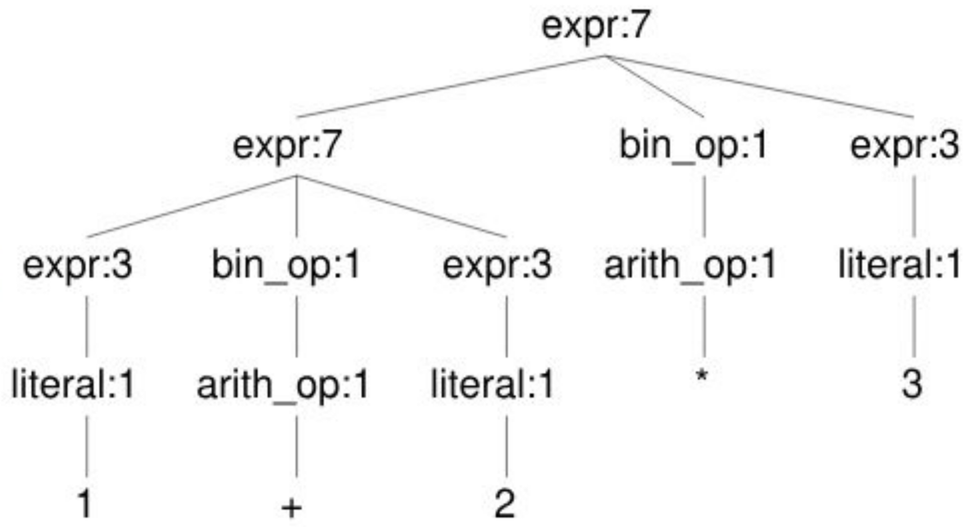
Input sample.expr ?

```
1 + 2 * 3
```

Start rule ?

expr Run Show profiler

Tree Hierarchy



Moral of the story

Please check LLM output

Weekly updates

Introduction to SSA ←

Note: This is **completely optional!**

You are not required to implement SSA in your compiler, nor is implementing it worth any extra credit.

Today's content focuses on theory (unlike previous recitations), and is based on chapters 1-3 of the SSA book*.

* [*SSA-based Compiler Design*, edited by Rastello and Tichadou, draft available at https://pfalcon.github.io/ssabook/latest/book-full.pdf]

What is SSA?

Static Single-Assignment

Is a property of the program code
(i.e. static property)

↓

Static Single-Assignment

Is a property of the program code
(i.e. static property)

Static Single-Assignment

Every variable is assigned to exactly once

What is SSA?

- A form of **low-level IR** in which every variable is defined exactly once
- Ways to think about this:
  - Variables are immutable
  - Every appearance of the same variable has the same value
  - “SSA is Functional Programming” [Appel 1998]

SSA in basic blocks

Basic block

```
a ← 1
b ← a + 1
a ← a + b
c ← a + 1
a ← b + c
```

SSA in basic blocks

Basic block

```
a ← 1
b ← a + 1
a ← a + b
c ← a + 1
a ← b + c
```

Many definitions and uses of **a**

SSA in basic blocks

Basic block

a ← 1
b ← **a** + 1
a ← **a** + b
c ← **a** + 1
a ← b + c

Many definitions and uses of **a**

These two expressions have different values!

SSA in basic blocks

Basic block

```
a ← 1
b ← a + 1
a ← a + b
c ← a + 1
a ← b + c
```

Let's color-code the definitions and uses of **a**

SSA in basic blocks

Basic block

$$a_1 \leftarrow 1$$
$$b_1 \leftarrow a_1 + 1$$
$$a_2 \leftarrow a_1 + b_1$$
$$c_1 \leftarrow a_2 + 1$$
$$a_3 \leftarrow b_1 + c_1$$

Let's color-code the definitions and uses of **a**

… and rename them to distinct names

SSA in basic blocks

Basic block

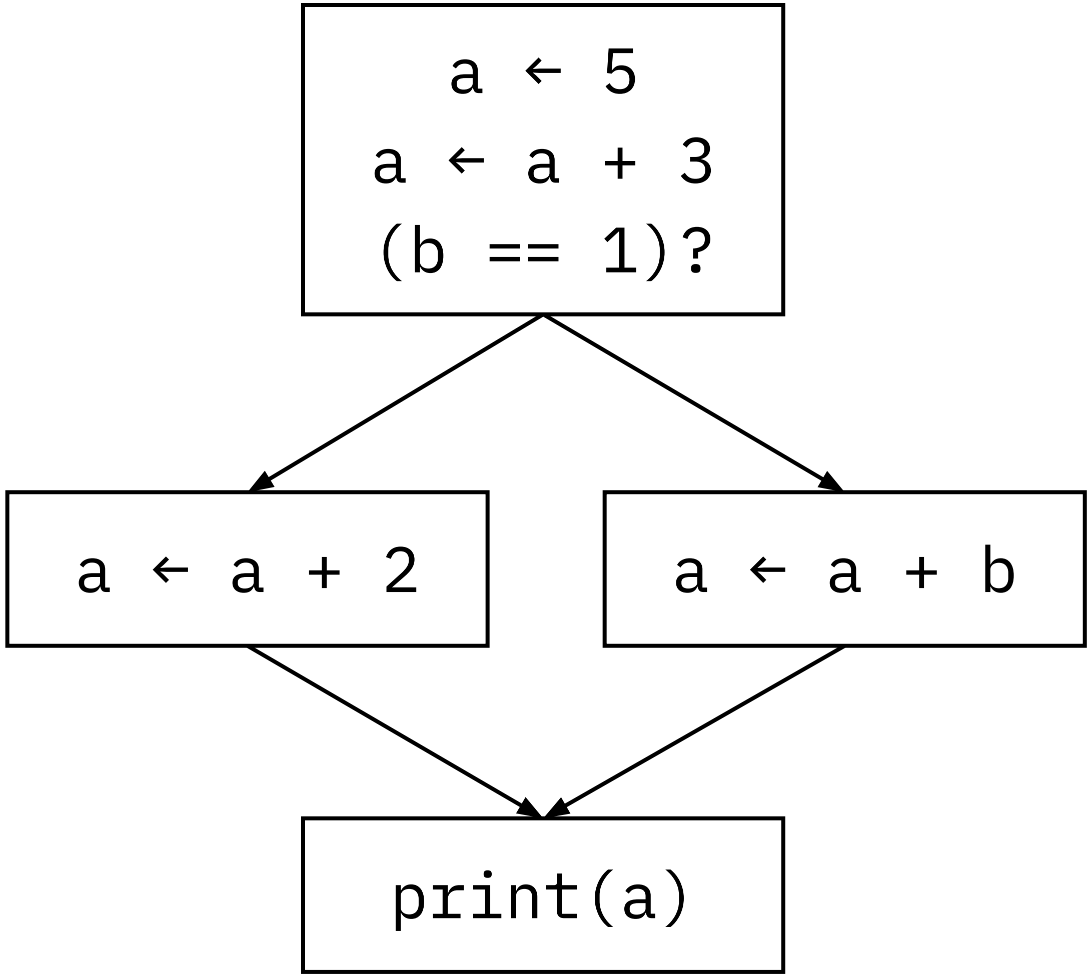
$a_1 \leftarrow 1$
$b_1 \leftarrow a_1 + 1$
$a_2 \leftarrow a_1 + b_1$
$c_1 \leftarrow a_2 + 1$
$a_3 \leftarrow b_1 + c_1$

Let's color-code the definitions and uses of **a**

... and rename them to distinct names

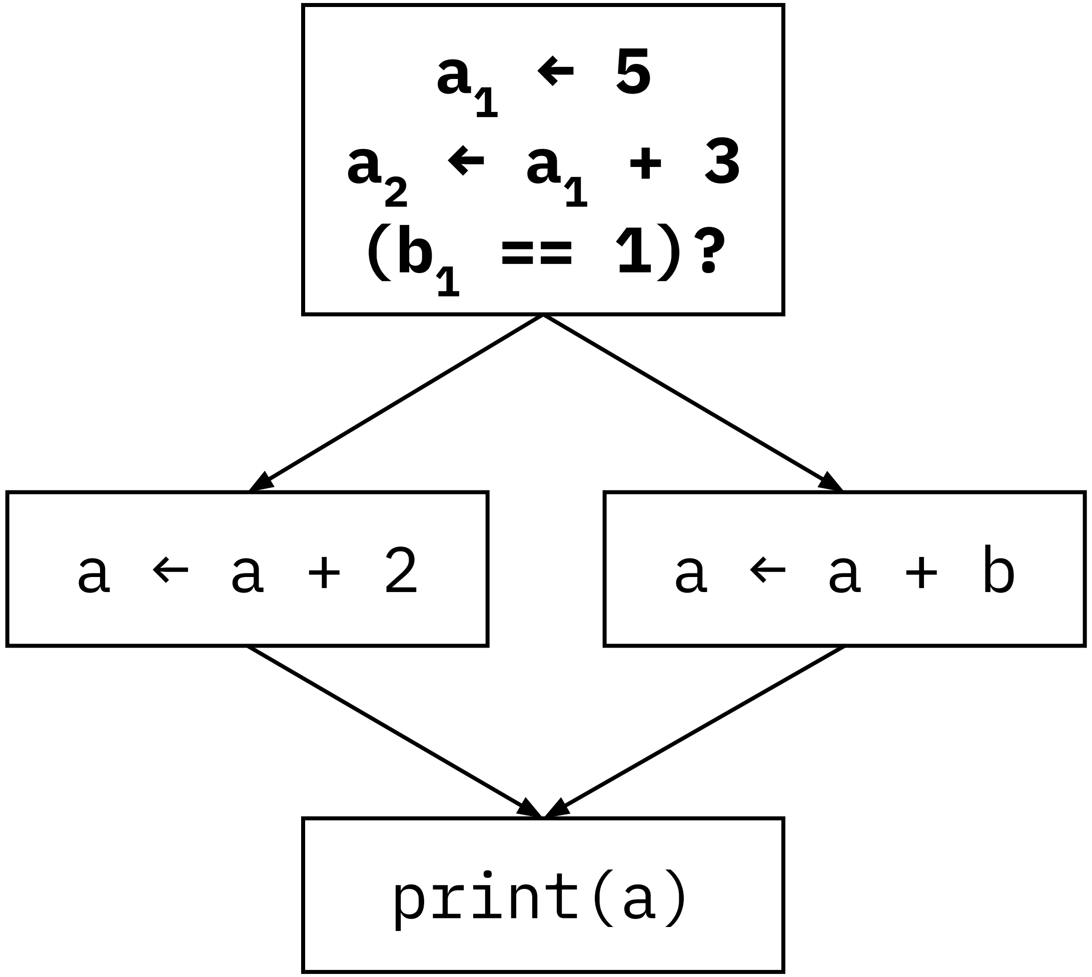
This is now in SSA form!
So far, so good

SSA in CFGs



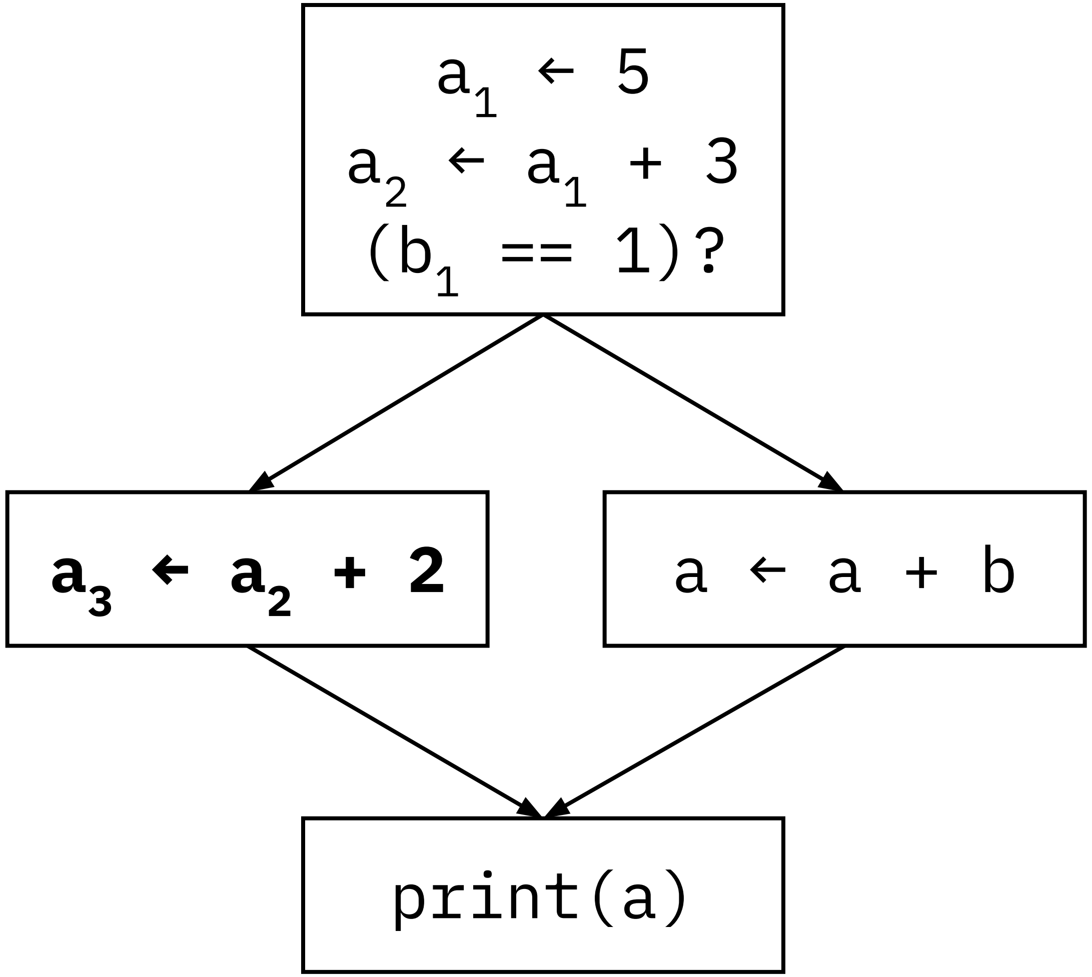
Let’s write each basic block in SSA form

SSA in CFGs



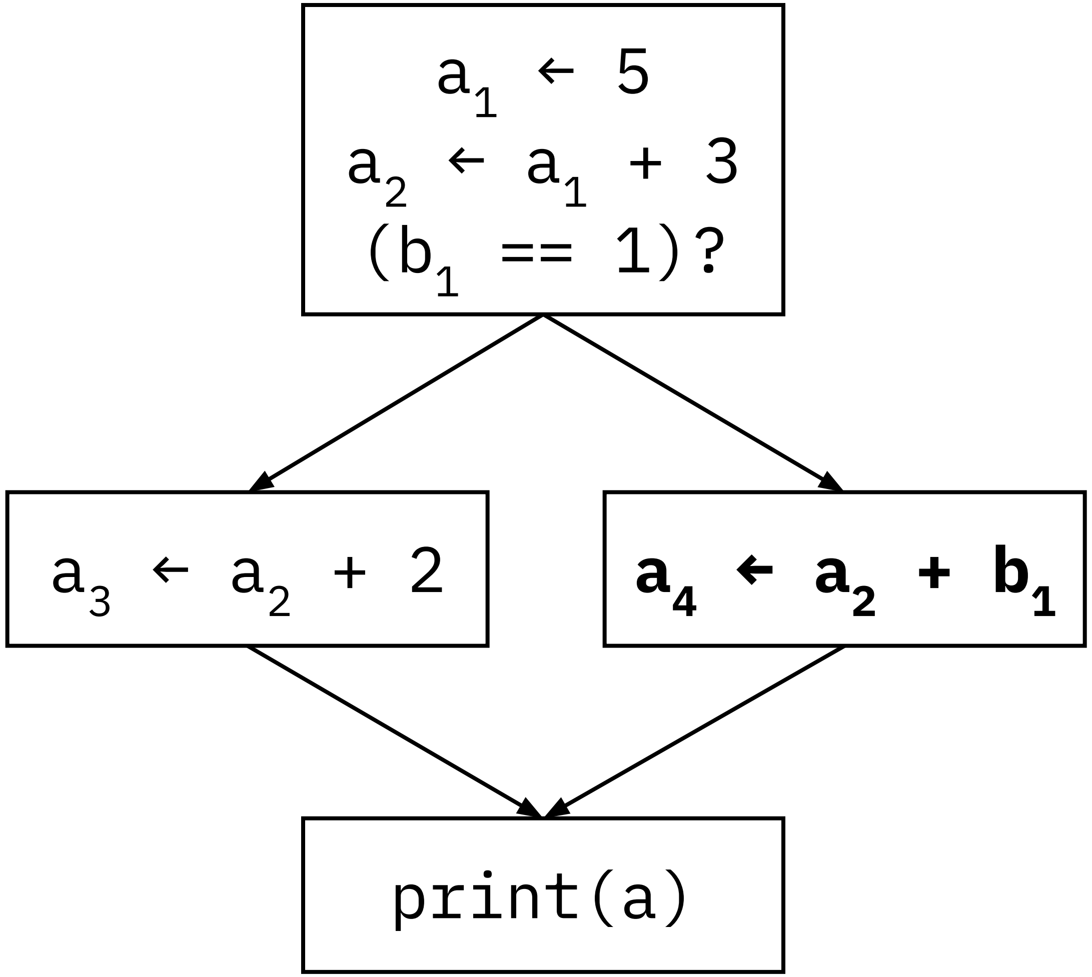
Let's write each basic block in SSA form

SSA in CFGs



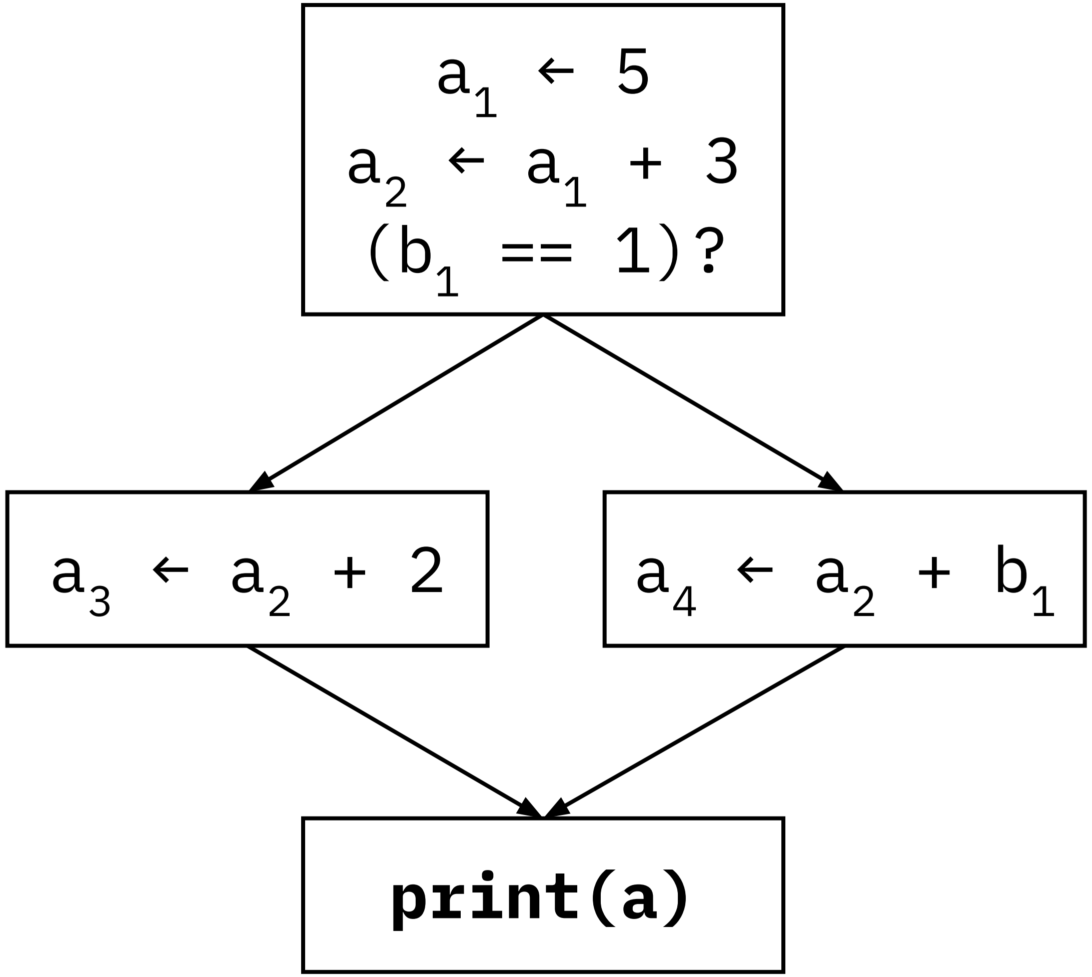
Let’s write each basic block in SSA form

SSA in CFGs



Let's write each basic block in SSA form

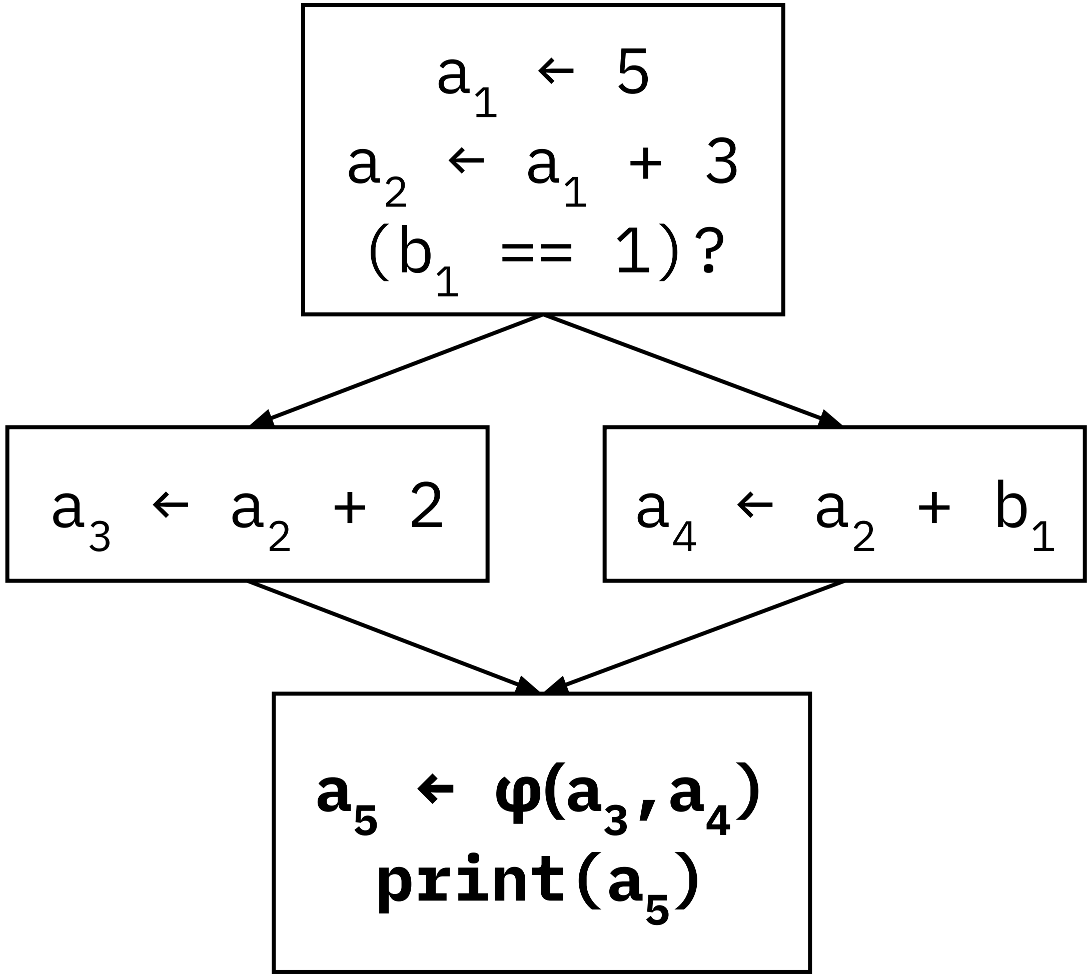
SSA in CFGs



Let’s write each basic block in SSA form

Oops, what do we do here?

SSA in CFGs



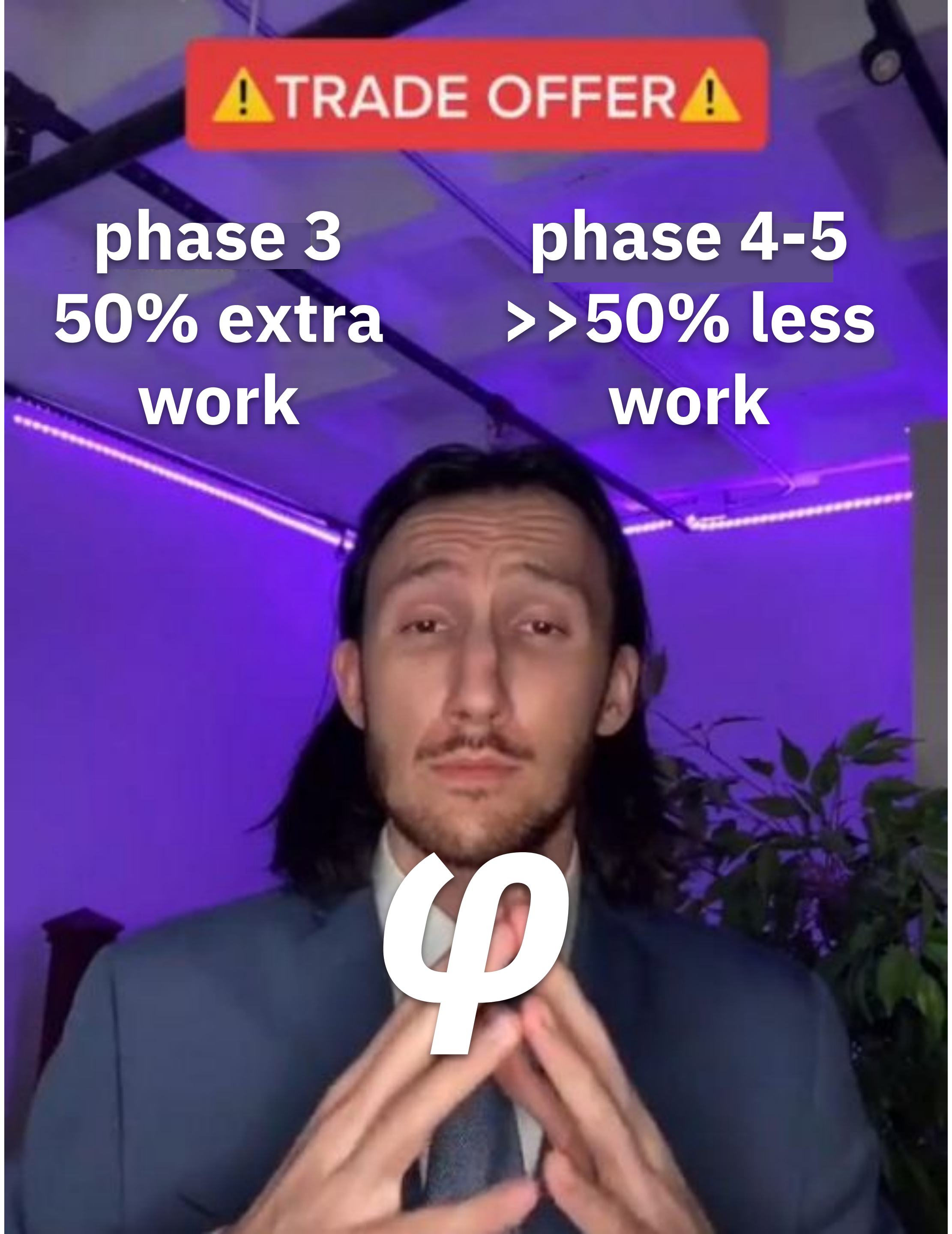
Merge values using **phi-function**

$\boldsymbol{\varphi(a_3, a_4)}$ means select either $\mathbf{a_3}$ or $\mathbf{a_4}$ based on the control flow path taken

Summary: what is SSA?

- A form of **low-level IR** in which every variable is defined exactly once
- Control-flow graph with every assignment gets a unique name
- Use **phi-function** to deal with merge points

Why is SSA useful?


⚠TRADE OFFER⚠
phase 3
50% extra
work
phase 4-5
>>50% less
work
φ

SSA makes program analysis simpler and faster

FAQ for optimizers

Upon seeing a variable assignment (**definition**)

- *Where might this definition be used?*
- Given a def, find **reachable uses**

Upon seeing a variable **use**

- *Where might the values come from?*
- Given a use, find **reaching defs**

Def → use

Where might this definition be used?

- If there’s no use, don’t need the assignment! (**Liveness Analysis** / **Dead Code Elimination**)
- If use is far away, maybe defer the assignment (**Code Motion**)
- Put immediate / frequently used vars in registers (**Register Allocation**)

Use → def

Where might the values come from?

- If only one def & is constant, replace use with const. (**Constant Propagation**)
- If only one def & is copy (x=y), replace use x with y (**Copy Propagation**)
- If value is loop variable (x=2*i), simplify (i++; *x+=2*) (Induction Variable / **Scalar Evolution**)

Reaching definitions

How do we know that the result of an assignment (**definition**) may be **use**d at [*use site*]?

Without SSA, need to do analysis

With SSA, just check if the definition and the use are for the same variable

Available expressions

Recall: in general, an expression **x+y** is available at a point **p** if

1. every path from the initial node to **p** must evaluate **x+y** before reaching **p**,
2. and there are no assignments to **x** or **y** after the evaluation but before **p**.

With SSA, no need to worry about **2.**

Liveness

Recall: in general,

- A variable **v** is live at point **p** if
  - **v** is used along some path starting at **p**, and
  - no definition of **v** along the path before the use

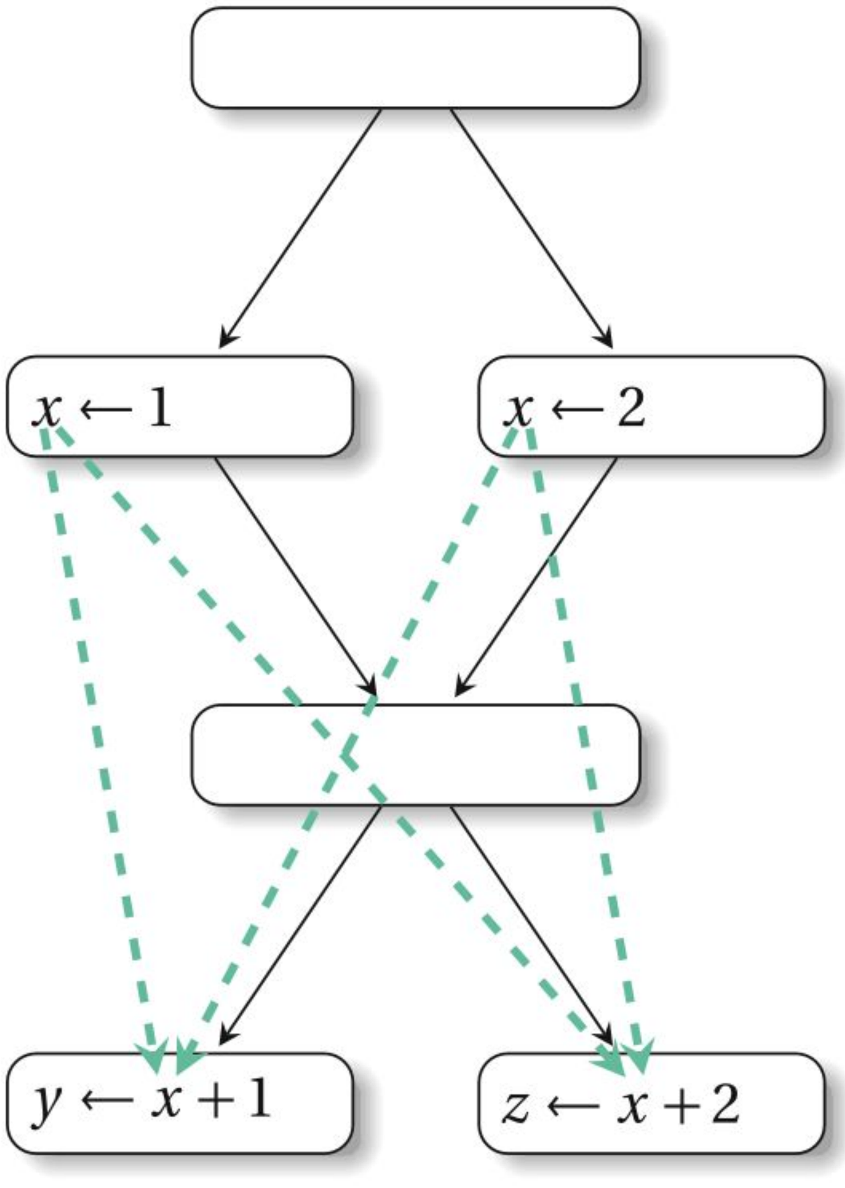
With SSA,

A variable **v** is live at its definition point if it has no uses

- In some sense, the work is done during the conversion to SSA instead…
  - but this work is done once and helps for many different program analyses
- SSA factors out one key aspect of program analysis: **def-use chains**

Def-use chains

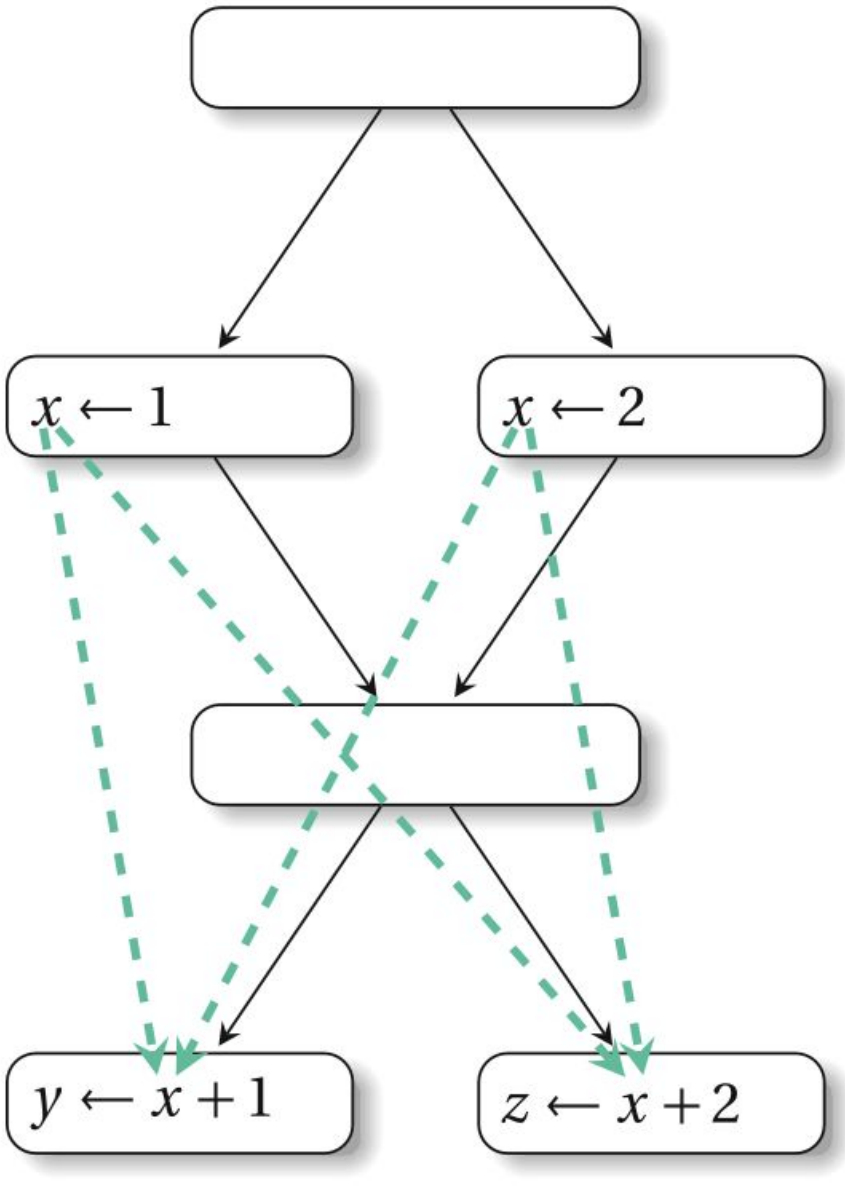
- It’s slow to propagate dataflow information through every node
- Optimization: compute **def-use chains**, which link each definition to its uses. This speeds up propagation of information!



[Figure 2.1a in SSA book]

Def-use chains

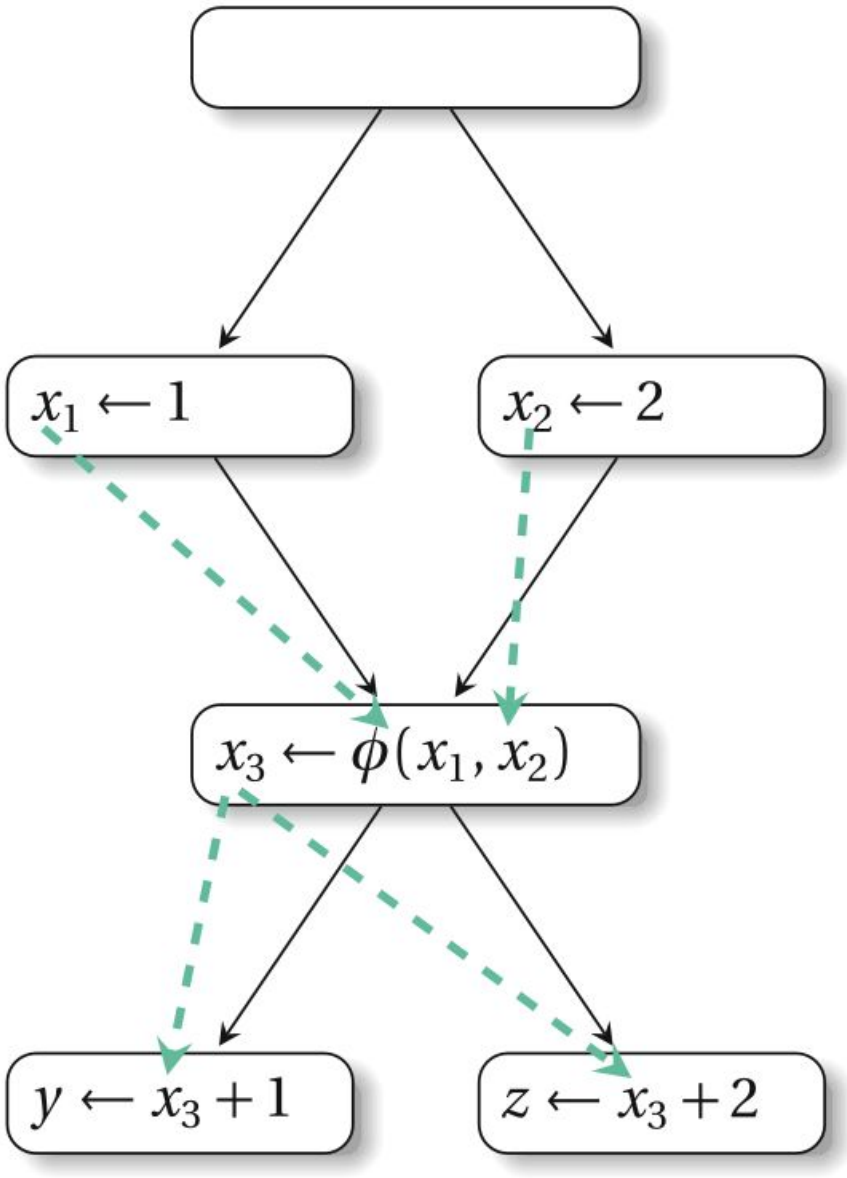
- **Problem:** number of def-use chains can be quadratic
- N defs, N uses, each use can be from any def → N^2 def-use chains!



[Figure 2.1a in SSA book]

Def-use chains

- **Problem:** number of def-use chains can be quadratic
- N defs, N uses, each use can be from any def → N^2 def-use chains!
- **With SSA,** each use can only be from one def → **O(N) def-use chains!**



[Figure 2.1b in SSA book]

How to implement SSA?

Implementing SSA

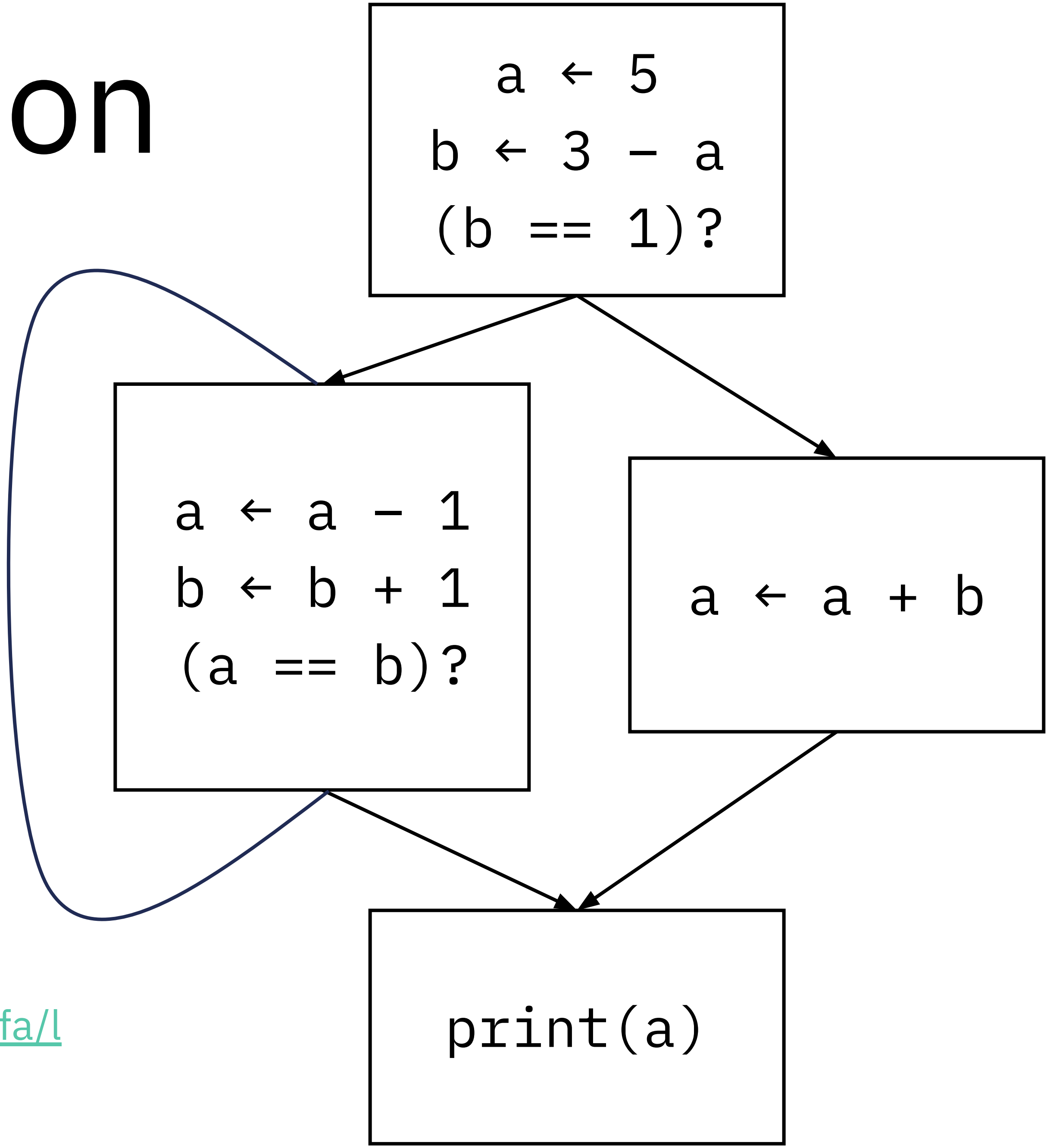
Two main tasks:

- Converting into SSA form (construction)
- Converting out of SSA form (destruction)

SSA construction

Naive method:

1. Add φ-nodes at the beginning of every basic block

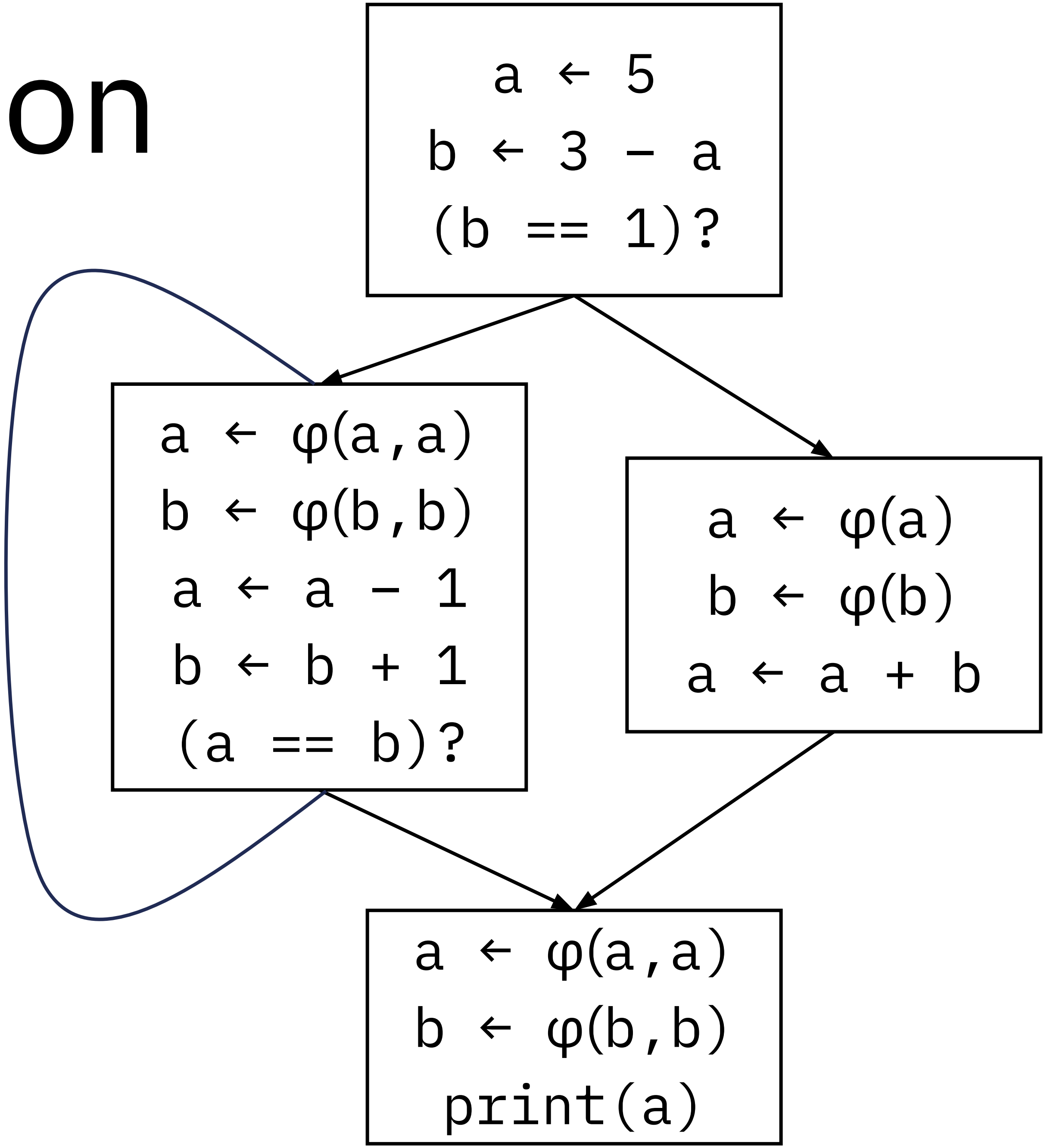


[This section is based on Harvard CS153 slides: https://groups.seas.harvard.edu/courses/cs153/2018fa/lectures/Lec23-SSA.pdf]

SSA construction

Naive method:

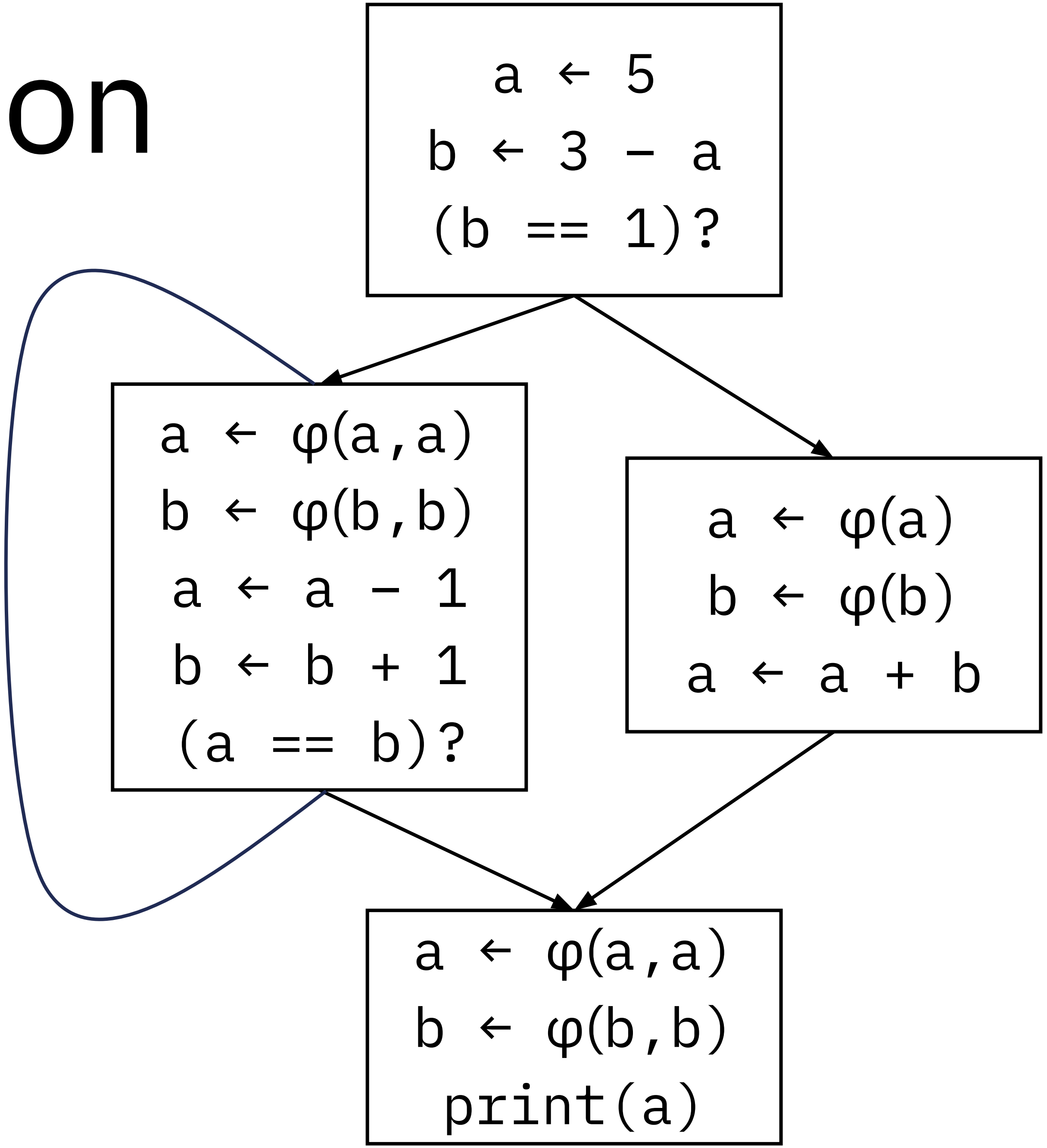
1. Add φ-nodes at the beginning of every basic block



SSA construction

Naive method:

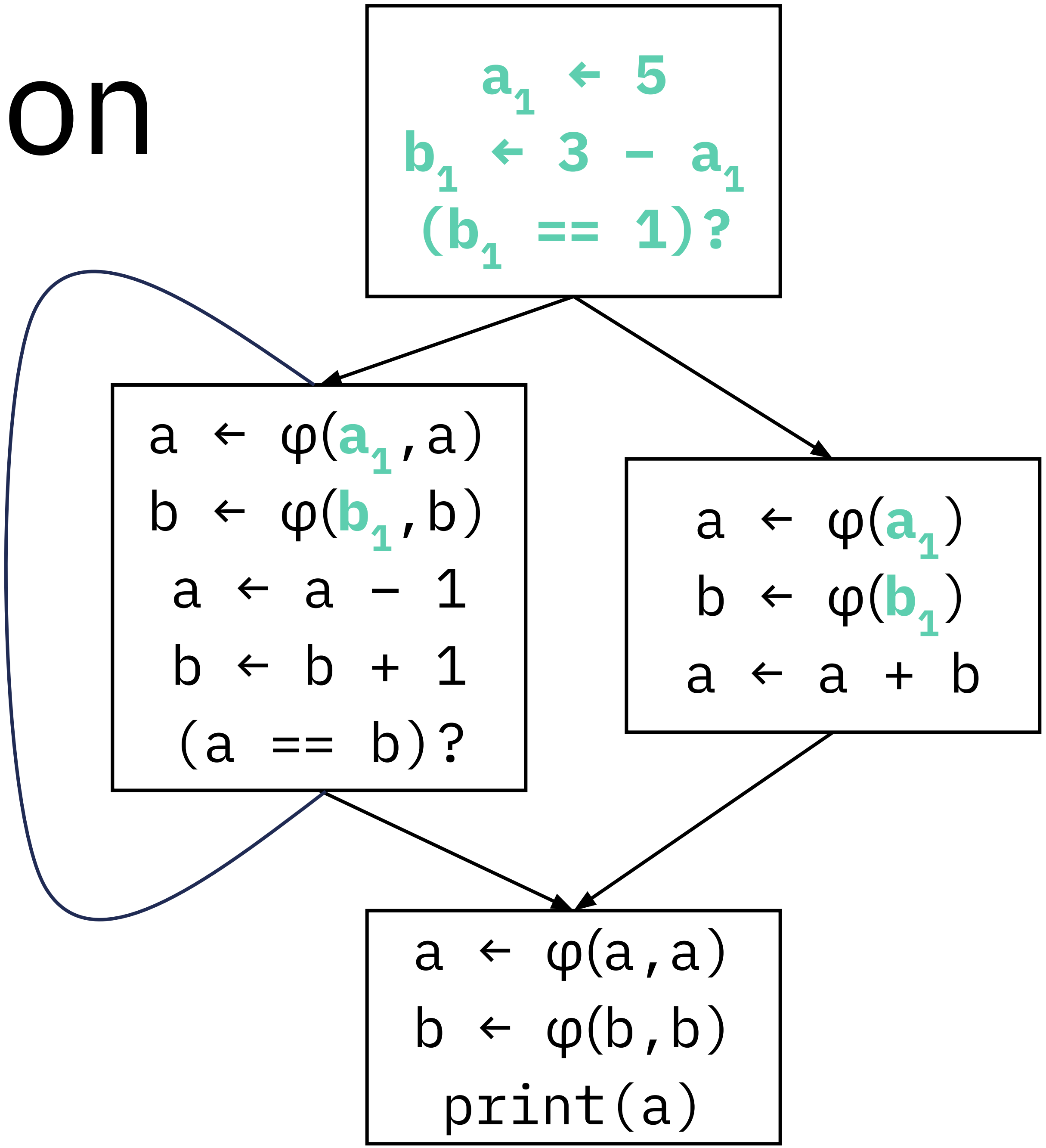
1. Add φ-nodes at the beginning of every basic block
2. Convert each basic block to SSA, and propagate the last definition to φ-nodes of successor blocks



SSA construction

Naive method:

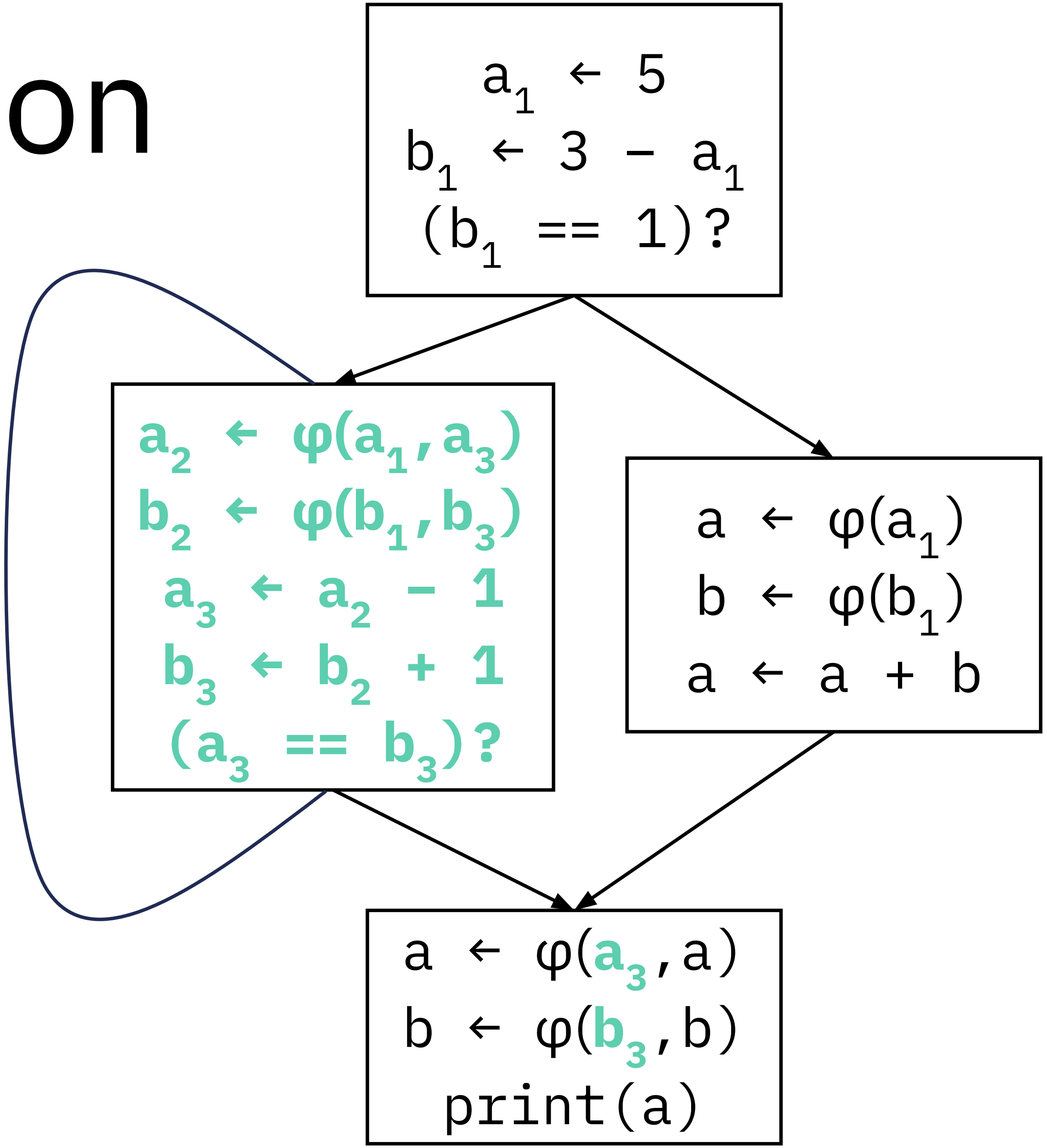
1. Add φ-nodes at the beginning of every basic block
2. Convert each basic block to SSA, and propagate the last definition to φ-nodes of successor blocks



SSA construction

Naive method:

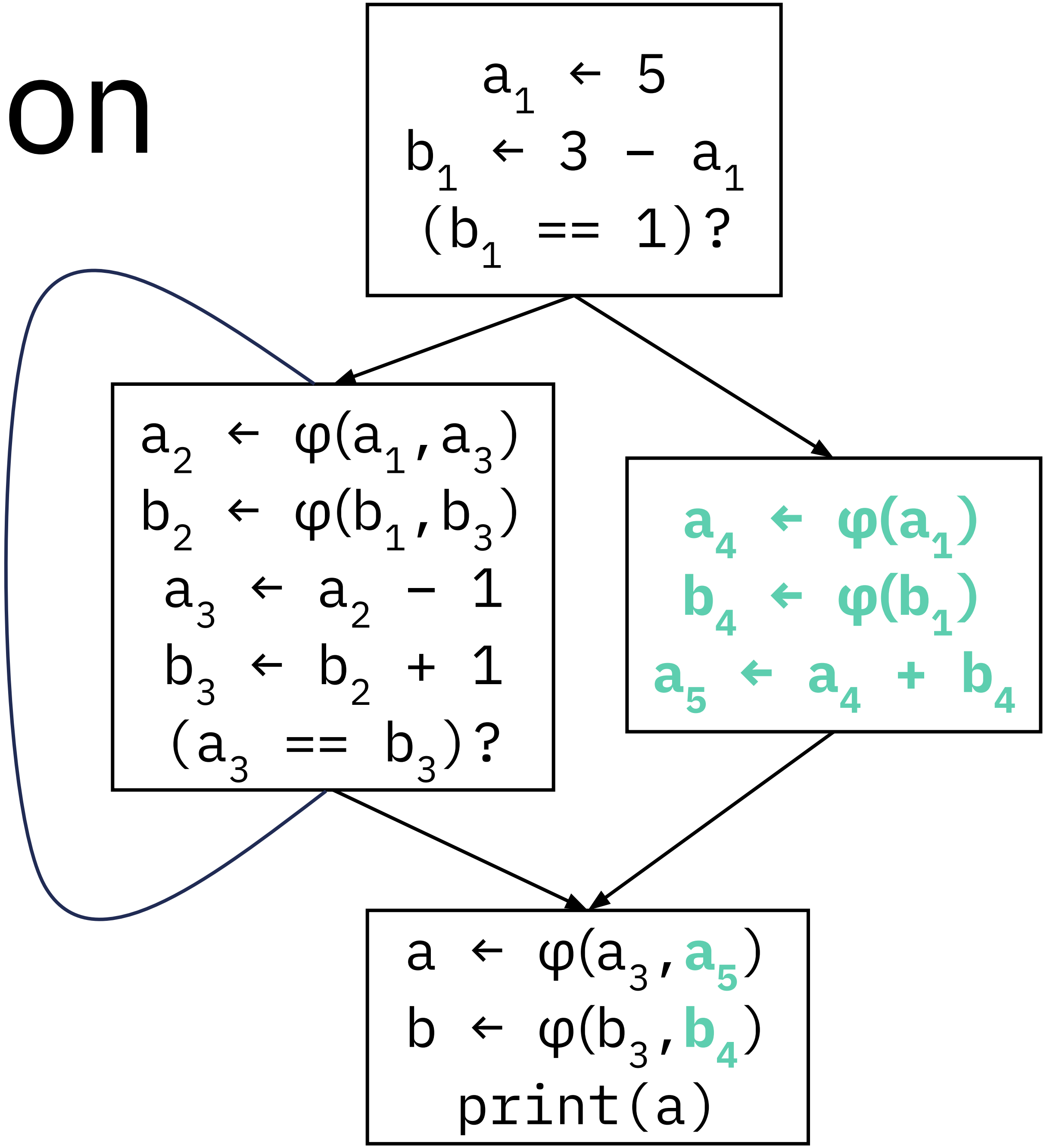
1. Add φ-nodes at the beginning of every basic block
2. Convert each basic block to SSA, and propagate the last definition to φ-nodes of successor blocks



SSA construction

Naive method:

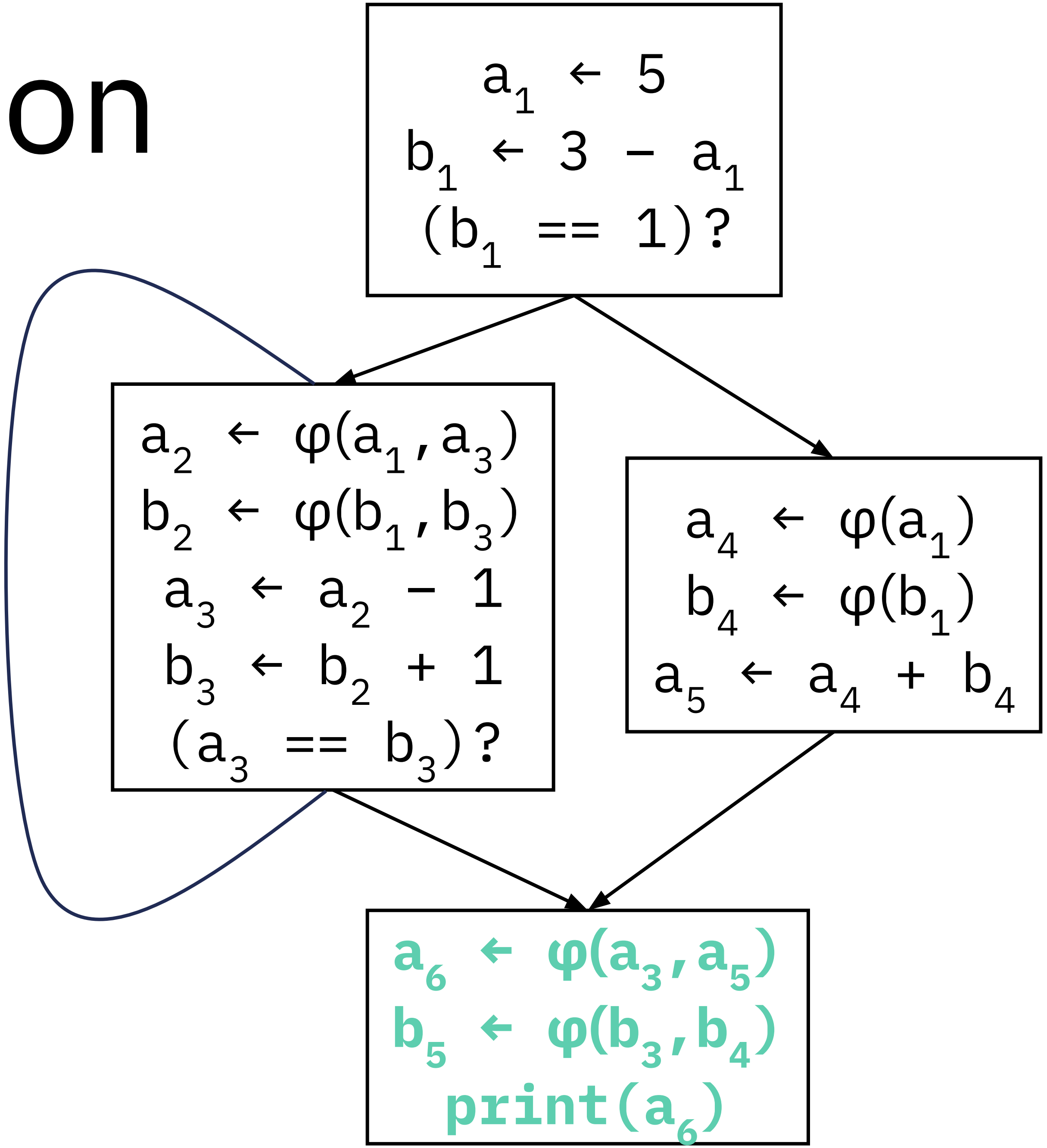
1. Add φ-nodes at the beginning of every basic block
2. Convert each basic block to SSA, and propagate the last definition to φ-nodes of successor blocks



SSA construction

Naive method:

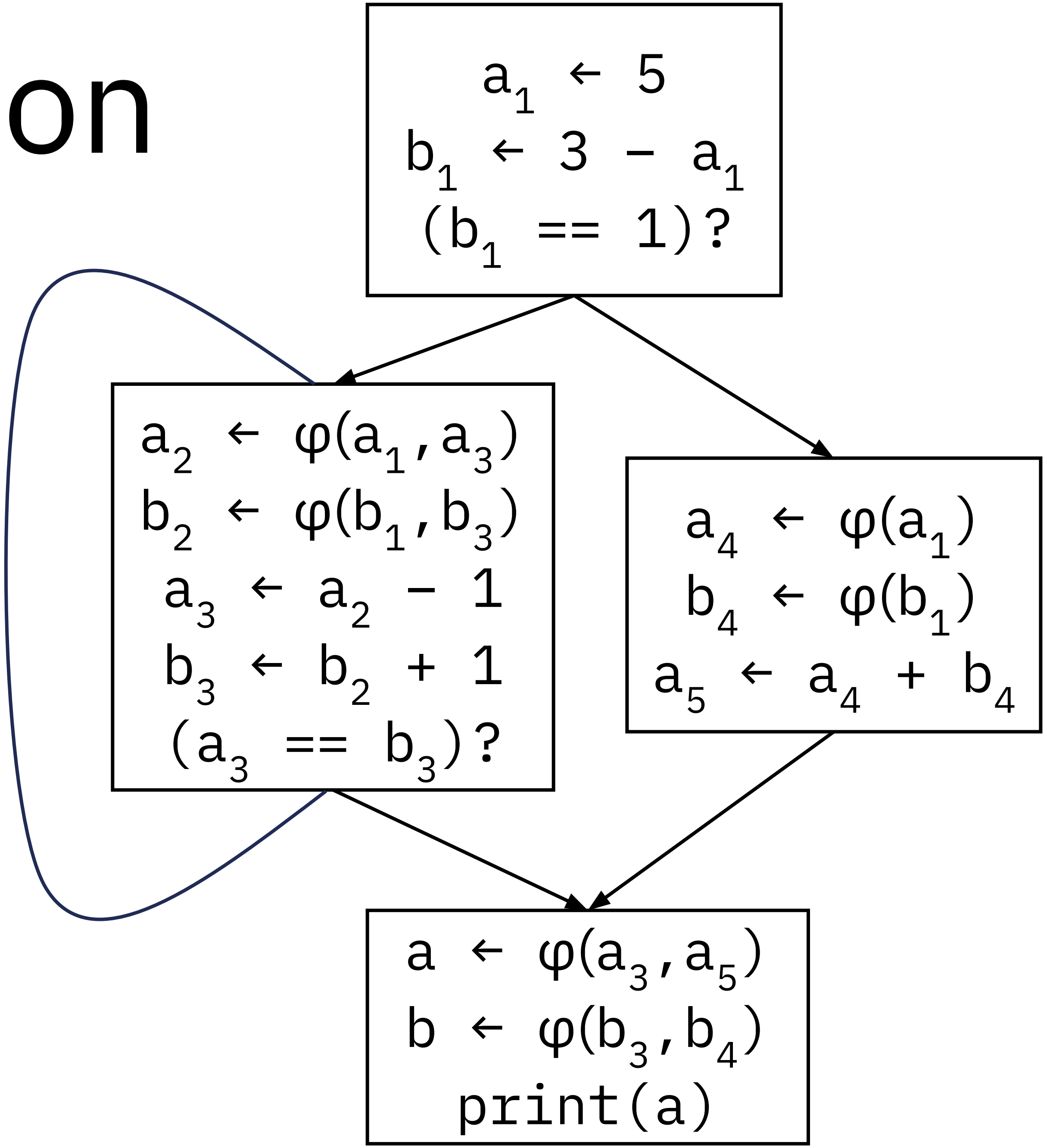
1. Add φ-nodes at the beginning of every basic block
2. Convert each basic block to SSA, and propagate the last definition to φ-nodes of successor blocks



SSA construction

Issue:
too many φ-nodes

To reduce φ-nodes, can run copy propagation and dead code elimination afterwards



SSA construction, but better

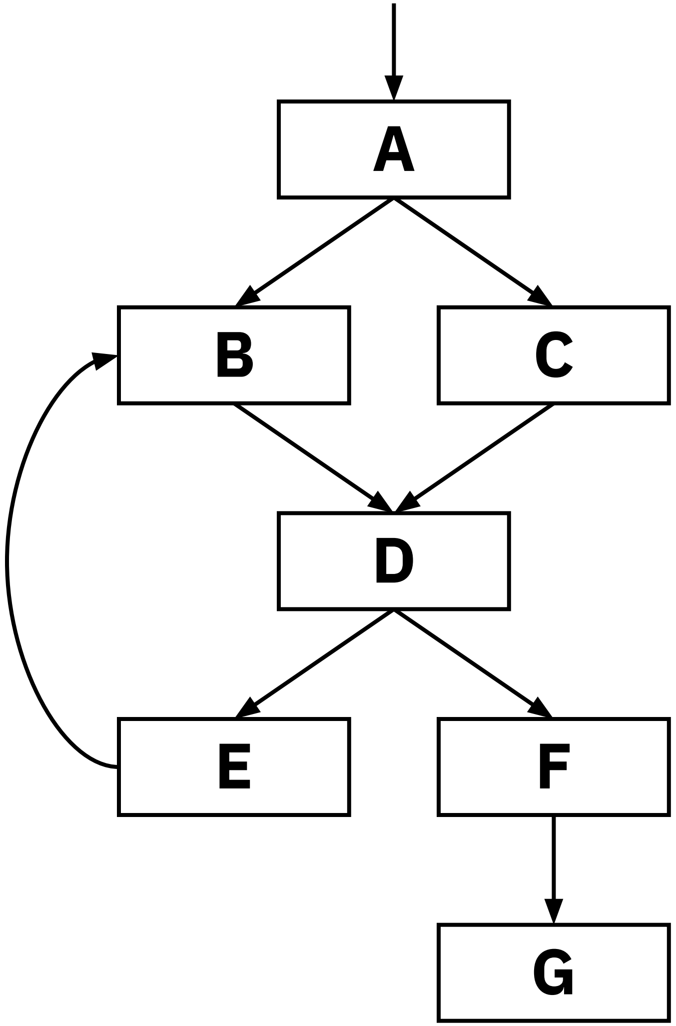
Standard method:

1. Compute the **dominator tree**
2. For each assignment of **x** (in basic block **B**), compute the **iterated dominance frontier $DF^+(B)$**
   and put φ-nodes for **x** at every block in **$DF^+(B)$**.
3. Rename variables in each basic block, where blocks are traversed in DFS order in dominator tree

Domination

In a control-flow graph:

- A node **n** **dominates** a node **m** if every path from the entry block to **m** goes through **n.**

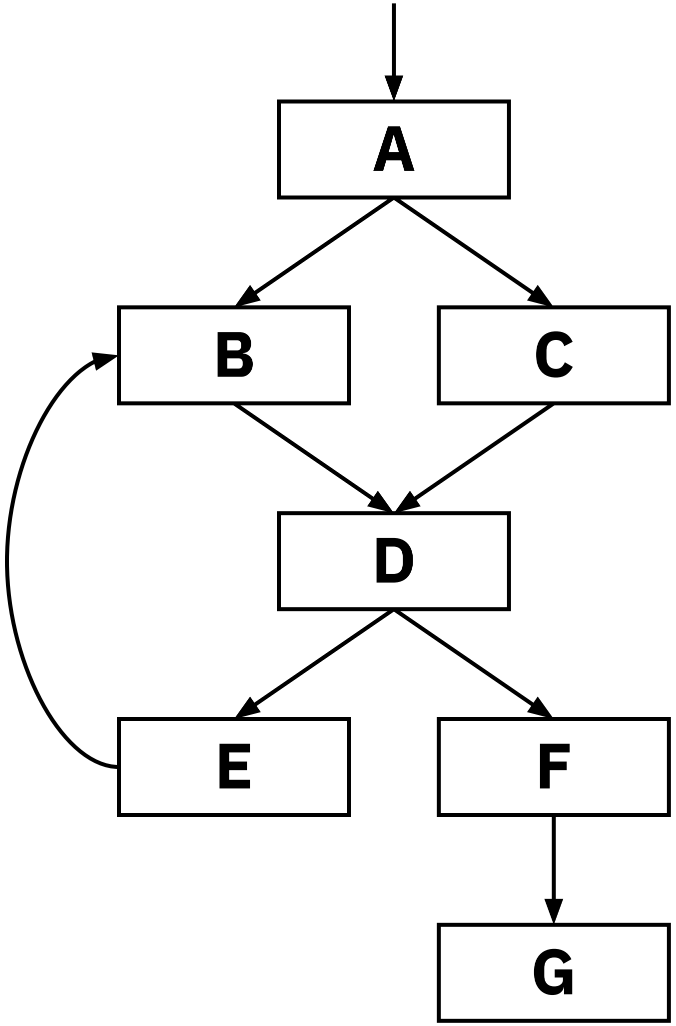


D dominates **D, E, F, G**

Domination

In a control-flow graph:

- A node **n** **dominates** a node **m** if every path from the entry block to **m** goes through **n.**
  - If **m ≠ n**, then **n** ***strictly* dominates m.**

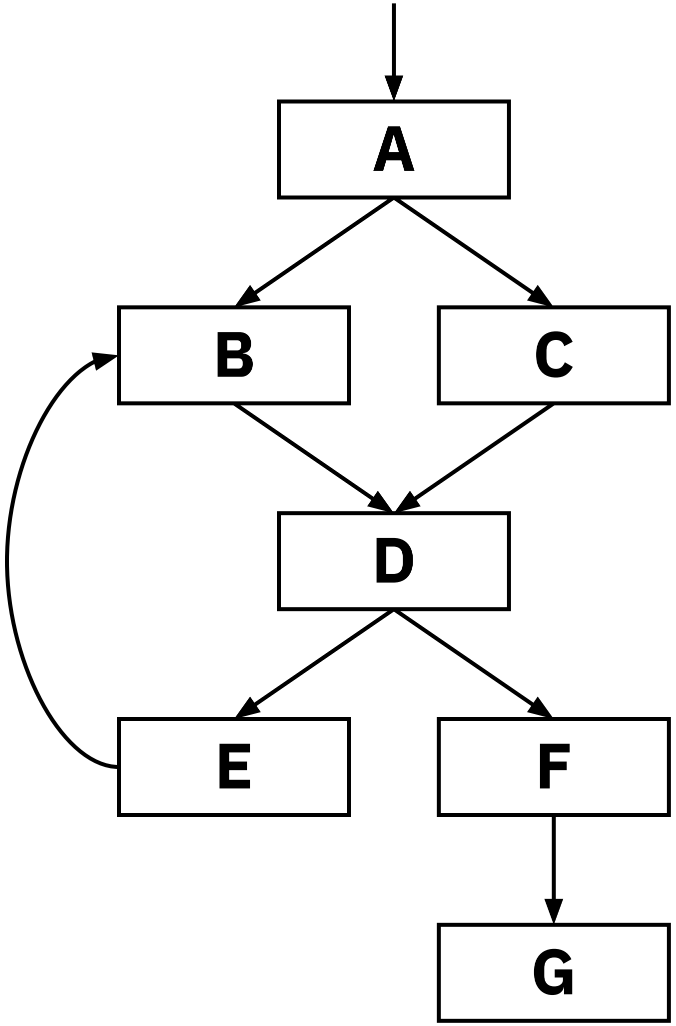


D strictly dominates **E, F, G**

Domination

In a control-flow graph:

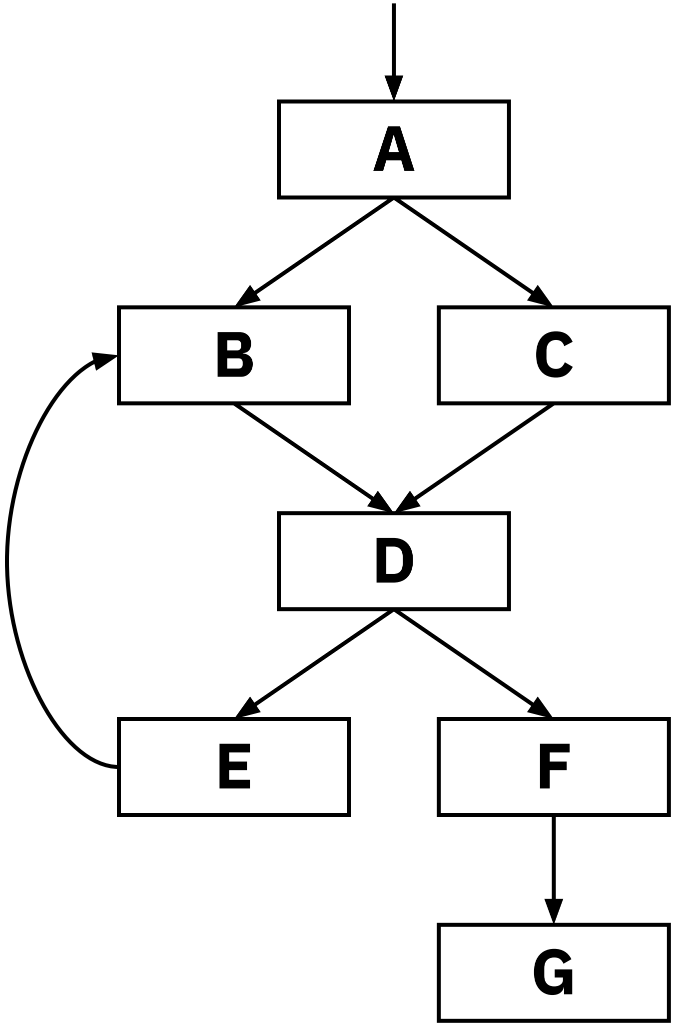
- A node **n** **dominates** a node **m** if every path from the entry block to **m** goes through **n.**
  - If **m ≠ n**, then **n** ***strictly* dominates m.**
  - If there are no nodes **x** such that **n** strictly dominates **x** and **x** strictly dominates **m**, then **n** ***immediately* dominates m.**



D immediately dominates **E, F**

Dominator tree

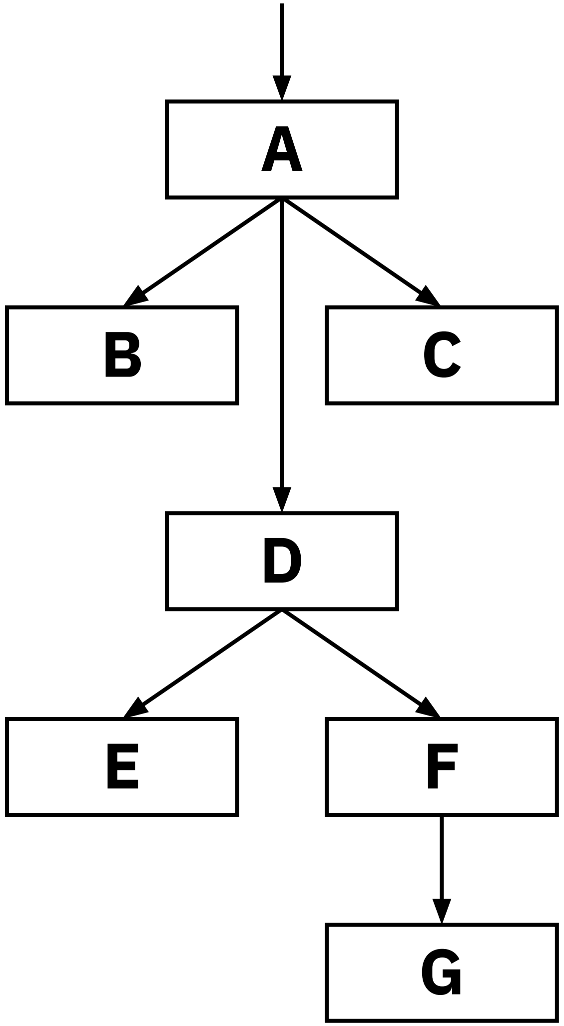
- Each node (except the entry node) has a unique **immediate dominator**



The immediate dominator of **D** is **A**

Dominator tree

- Each node (except the entry node) has a unique **immediate dominator**
- The **dominator tree** is the tree where there is an edge **n** to **m** if **n** immediately dominates **m**

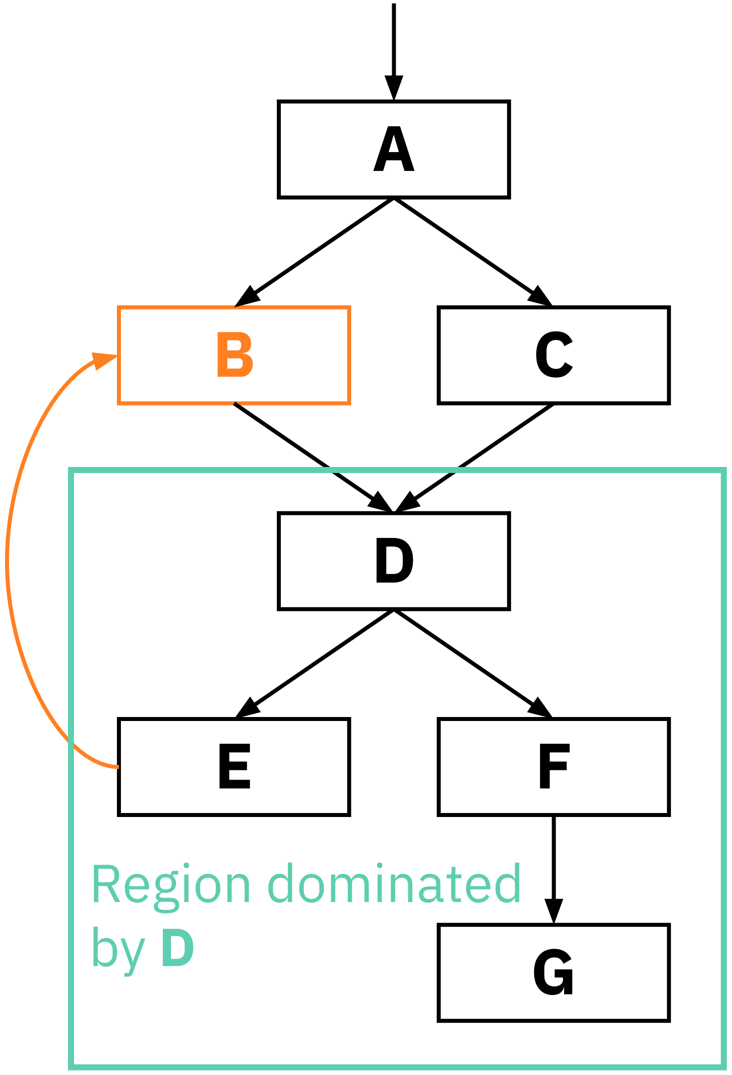


Dominator tree

Dominance frontier

The **dominance frontier** **DF(n)** of a node **n** is the border of the CFG region dominated by **n**.

(To be precise, this is the set of nodes **m** such that **n** dominates an immediate predecessor of **m** but not **m**.)



The dominance frontier of **D** is {**B**}

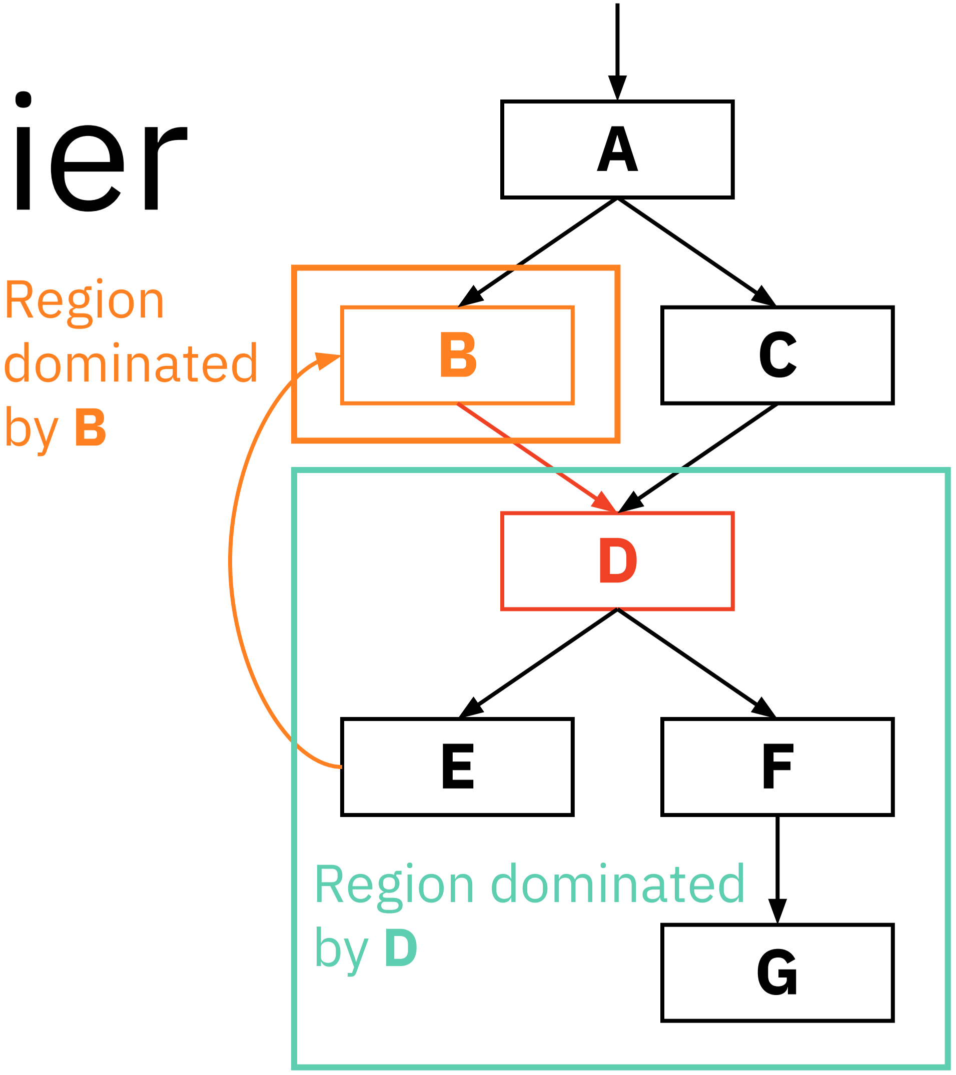
Dominance frontier

The **dominance frontier DF(n)** of a node **n** is the border of the CFG region dominated by **n**.

(To be precise, this is the set of nodes **m** such that **n** dominates an immediate predecessor of **m** but not **m**.)

The **iterated dominance frontier** $\mathbf{DF^+(n)}$ is the limit of the sequence

$$\mathbf{DF^0(n) = \{n\},}$$
$$\mathbf{DF^{i+1}(n) = DF(\{n\} \cup DF^i(n))}$$



$\mathbf{DF^+(D) = \{B, D\}}$

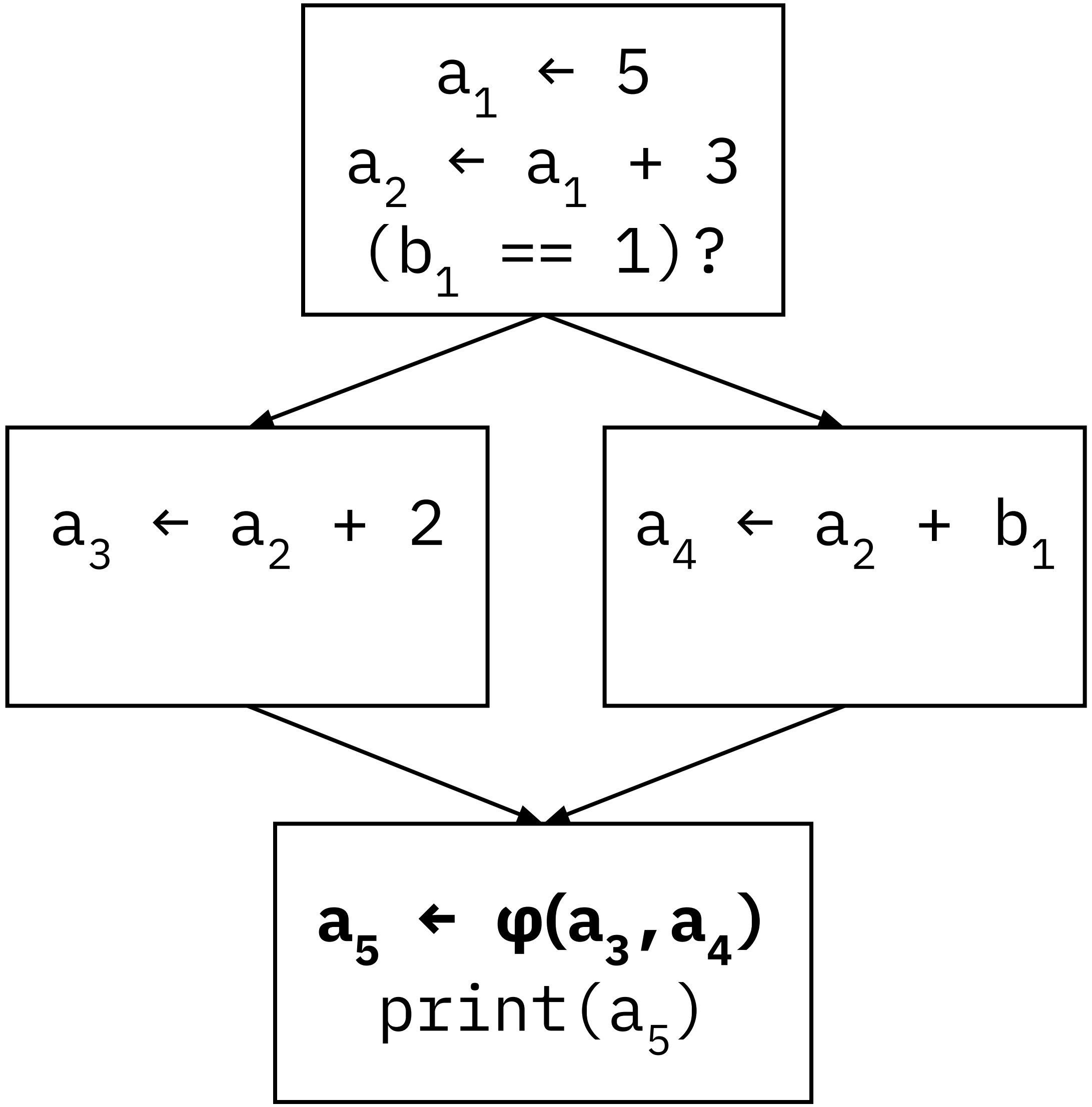
SSA construction, but better

Standard method:

1. Compute the **dominator tree**
2. For each assignment of **x** (in basic block **B**), compute the **iterated dominance frontier $DF^+(B)$** and put φ-nodes for **x** at every block in **$DF^+(B)$**.
3. Rename variables in each basic block, where blocks are traversed in DFS order in dominator tree

SSA destruction

Simplest method: add assignments to the end of predecessor blocks of φ-nodes

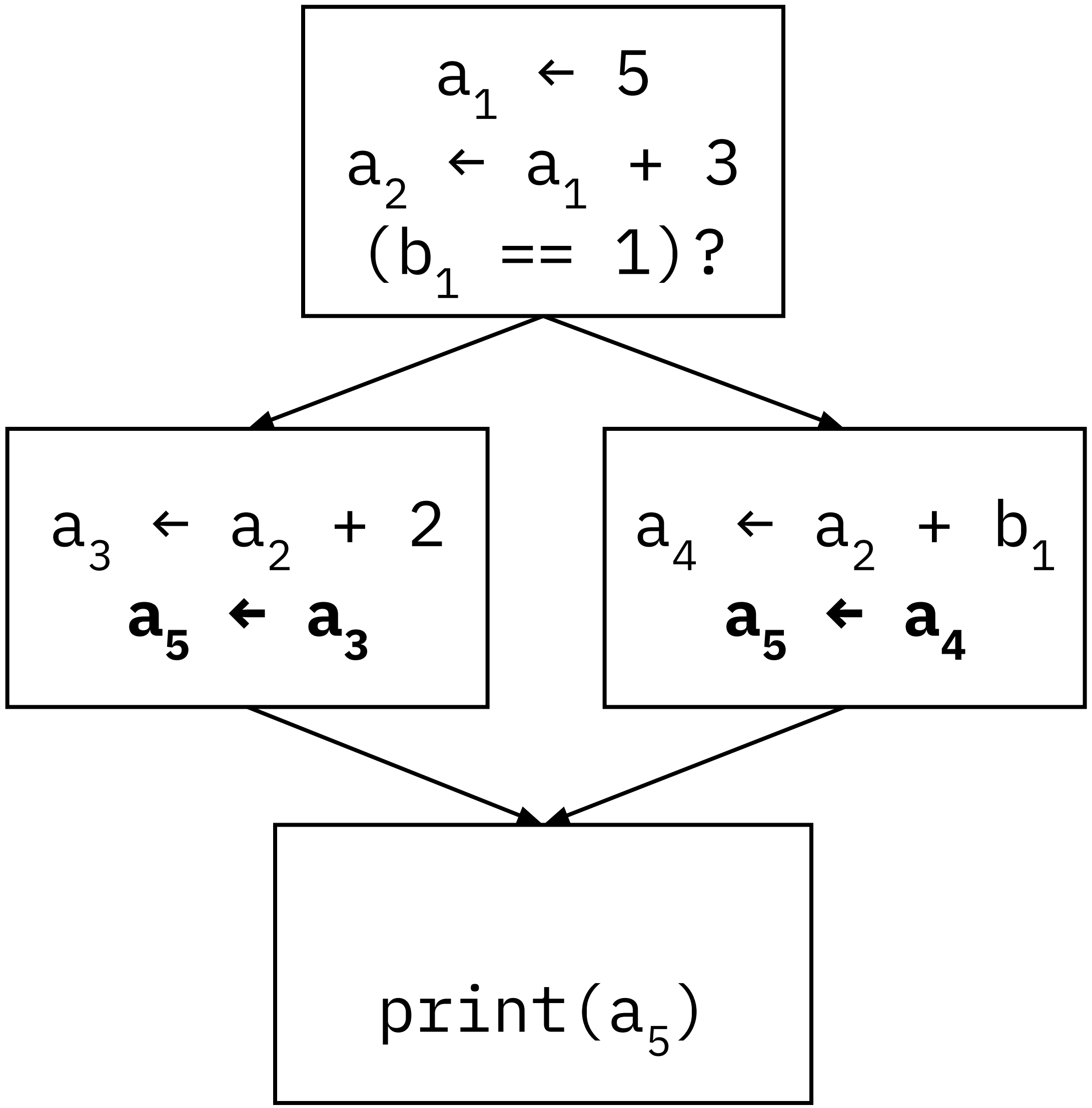


SSA destruction

Simplest method: add assignments to the end of predecessor blocks of φ-nodes

This creates extra copies, but a coalescing register allocator can deal with it

(Caveat: **cycles**)



That’s all for today!

If you want to learn more, consider reading the SSA book*!

* [*SSA-based Compiler Design*, edited by Rastello and Tichadou,
draft available at https://pfalcon.github.io/ssabook/latest/book-full.pdf]