ægraphs: Acyclic E-graphs

for Efficient Optimization in a Production Compiler

Chris Fallin ***(Fastly)***
EGRAPHS 2023
chris@cfallin.org

In the beginning, there was a compiler backend…

Cranelift

Cranelift

- Open-source general-purpose optimizing compiler backend
  - Written in Rust + a pattern-matching DSL (~200KLoC, ~130KLoC tests)
  - SSA input, four ISAs (x86-64, aarch64, riscv64, s390x)

Cranelift

- Open-source general-purpose optimizing compiler backend
  - Written in Rust + a pattern-matching DSL (~200KLoC, ~130KLoC tests)
  - SSA input, four ISAs (x86-64, aarch64, riscv64, s390x)
- Used in production as part of Wasmtime
- O(3-5) active developers at any time

Cranelift

- **Speed:** JIT focus
- **Simplicity:** “not LLVM”
- **Verifiability:** explicitly design with fuzzing + formal techniques + … in mind
- **Research-friendliness:** we need new ideas to compete with larger peers

1. **Why** *we want a rewrite-based optimizer*
2. **How** *to turn a CFG into an egraph and back again*
3. **Cycles** *why they occur, and what to do about them*
4. **Results** *how well does it work?*
5. **Lessons** *in translating research to production*

1. Why	*we want a rewrite-based optimizer* ←
2. **How**	*to turn a CFG into an egraph and back again*
3. Cycles	*why they occur, and what to do about them*
4. Results	*how well does it work?*
5. Lessons	*in translating research to production*

Cranelift, circa mid-2022

- Focus on codegen quality & mid-end optimizations
- We had: GVN, constant folding, LICM, some simple rewrites

Cranelift, circa mid-2022

- Focus on codegen quality & mid-end optimizations
- We had: GVN, constant folding, LICM, some simple rewrites

```
v1 = …
v2 = iadd_imm v1, 16
…
v10 = iadd_imm v1, 16
```

→

```
v1 = …
v2 = iadd_imm v1, 16
…
v10 -> v2
```

Cranelift, circa mid-2022

- Focus on codegen quality & mid-end optimizations
- We had: GVN, constant folding, LICM, some simple rewrites
- We added: alias analysis => redundant load elim + store-to-load forwarding

Cranelift, circa mid-2022

- Focus on codegen quality & mid-end optimizations
- We had: GVN, constant folding, LICM, some simple rewrites
- We added: alias analysis => redundant load elim + store-to-load forwarding

```
v1 = …
v2 = load.i64 v1+8
…
v10 = load.i64 v1+8
```

```
v1 = …
v2 = load.i64 v1+8
…
v10 -> v2
```

The Pass-Order Problem

- What do we do with this program?

```
v1 = …
v2 = iadd_imm v1, 16
v3 = load.i64 v2
…
v10 = iadd_imm v1, 16
v11 = load.i64 v10
```

The Pass-Order Problem

- What do we do with this program?

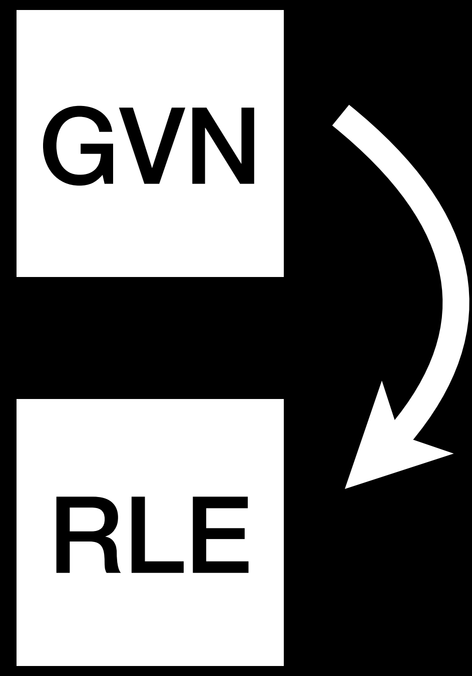
```
v1 = …
v2 = iadd_imm v1, 16
v3 = load.i64 v2
…
v10 -> v2
v11 = load.i64 v10
```

GVN

The Pass-Order Problem

- What do we do with this program?

```
v1 = …

v2 = iadd_imm v1, 16

v3 = load.i64 v2

…

v10 -> v2

v11 -> v3
```



The Pass-Order Problem

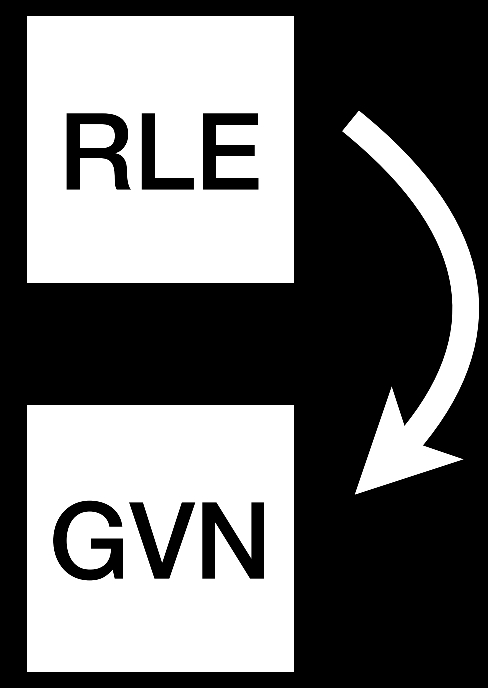
- What do we do with this program?

```
v1 = …
v2 = load.i64 v1+8
v3 = iadd v2, v1
…
v10 = load.i64 v1+8
v11 = iadd v10, v1
```

The Pass-Order Problem

- What do we do with this program?

```
v1 = …

v2 = load.i64 v1+8

v3 = iadd v2, v1

…

v10 -> v2

v11 -> v3
```



The Pass-Order Problem

- Proposed optimization pipeline:

```
fn optimize(&mut self) {
  self.gvn();
  self.rle();
  self.gvn();
}
```

The Pass-Order Problem

- Proposed optimization pipeline:

```
fn optimize(&mut self) {
  self.gvn();
  self.rle();
  self.gvn();
  self.rle(); // XXX just in case
}
```

The Pass-Order Problem

- Surely other production compilers have solved this problem?

The Pass-Order Problem

- Surely other production compilers have solved this problem?

GCC:

The Pass-Order Problem

- Surely other production compilers have solved this problem?

GCC:

```
NEXT_PASS (pass_cse);
```

The Pass-Order Problem

- Surely other production compilers have solved this problem?

GCC:

```
NEXT_PASS (pass_cselim);
…
NEXT_PASS (pass_cse_sincos);
NEXT_PASS (pass_cse_reciprocals);
…
NEXT_PASS (pass_cse);
…
NEXT_PASS (pass_cse_after_global_opts);
…
NEXT_PASS (pass_cse2);
…
NEXT_PASS (pass_postreload_cse);
```

The Pass-Order Problem

- Surely other production compilers have solved this problem?

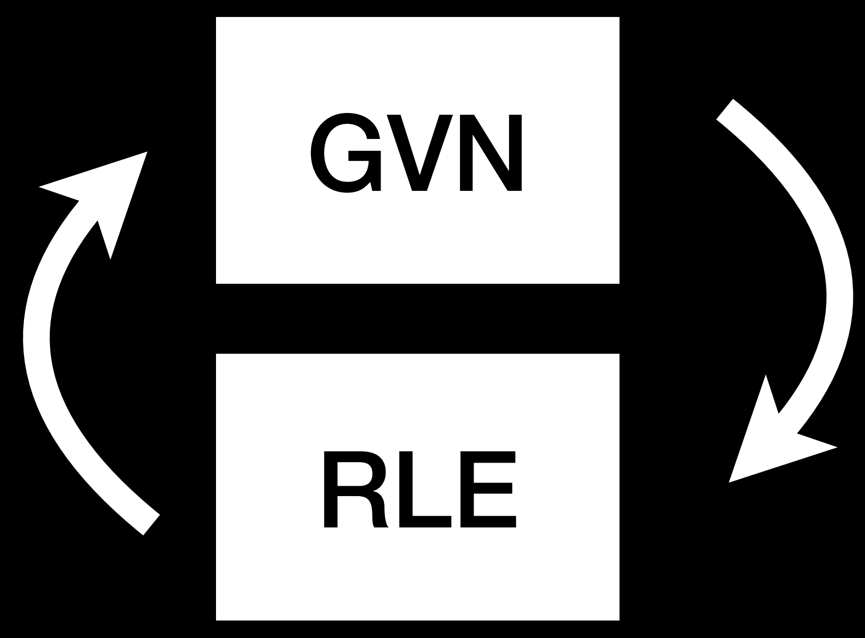
GCC:

540 lines (passes.def)

The Pass-Order Problem

(or: “Fix-point All The Things?”)

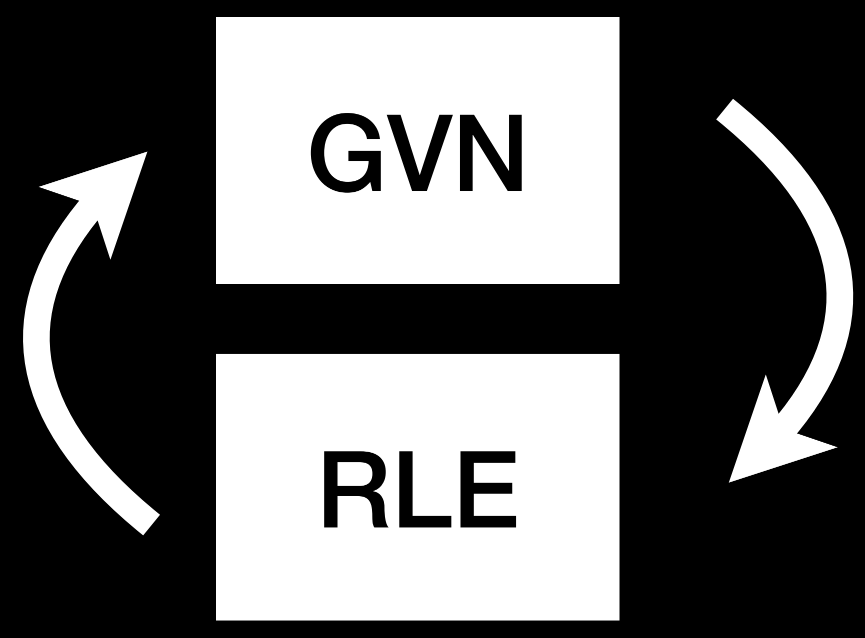
- Goal: find a way to put all (ish) of our optimizations in a single fixpoint loop



… cprop, algebraic rewrites, strength reduction, …

The Pass-Order Problem
(or: “Fix-point All The Things?”)

- Goal: find a way to put all (ish) of our optimizations in a single fixpoint loop
  - Remember: compile-cost focus! (We can’t afford to run a pass N times)
  - We’d prefer not to maintain a brittle heuristic pass order



… cprop, algebraic rewrites,
strength reduction, …

Adding some “simple” rewrites

```
match opcode {
    Opcode::IaddImm
    | Opcode::ImulImm
    | Opcode::BorImm
    | Opcode::BandImm
    | Opcode::BxorImm => {
        // Fold binary_op(C2, binary_op(C1, x)) into binary_op(binary_op(C1, C2), x)
        if let ValueDef::Result(arg_inst, _) = pos.func.dfg.value_def(arg) {
```

Adding some “simple” rewrites (!)

```
match opcode {
    Opcode::IaddImm
    | Opcode::ImulImm
    | Opcode::BorImm
    | Opcode::BandImm
    | Opcode::BxorImm => {
        // Fold binary_op(C2, binary_op(C1, x)) into binary_op(binary_op(C1, C2), x)
        if let ValueDef::Result(arg_inst, _) = pos.func.dfg.value_def(arg) {
            if let InstructionData::BinaryImm64 {
                opcode: prev_opcode,
                arg: prev_arg,
                imm: prev_imm,
            } = &pos.func.dfg.insts[arg_inst]
            {
                if opcode == *prev_opcode
                    && ty == pos.func.dfg.ctrl_typevar(arg_inst)
                {
                    let lhs: i64 = imm.into();
                    let rhs: i64 = (*prev_imm).into();
```

Adding some “simple” rewrites (!!!)

```
Opcode::IaddImm
| Opcode::ImulImm
| Opcode::BorImm
| Opcode::BandImm
| Opcode::BxorImm => {
    // Fold binary_op(C2, binary_op(C1, x)) into binary_op(binary_op(C1, C2), x)
    if let ValueDef::Result(arg_inst, _) = pos.func.dfg.value_def(arg) {
        if let InstructionData::BinaryImm64 {
            opcode: prev_opcode,
            arg: prev_arg,
            imm: prev_imm,
        } = &pos.func.dfg.insts[arg_inst]
        {
            if opcode == *prev_opcode
                && ty == pos.func.dfg.ctrl_typevar(arg_inst)
            {
                let lhs: i64 = imm.into();
                let rhs: i64 = (*prev_imm).into();
                let new_imm = match opcode {
                    Opcode::BorImm => lhs | rhs,
                    Opcode::BandImm => lhs & rhs,
                    Opcode::BxorImm => lhs ^ rhs,
                    Opcode::IaddImm => lhs.wrapping_add(rhs),
                    Opcode::ImulImm => lhs.wrapping_mul(rhs),
                    _ => panic!("can't happen"),
                };
                let new_imm = immediates::Imm64::from(new_imm);
                let new_arg = *prev_arg;
                pos.func
                    .dfg
                    .replace(inst)
                    .BinaryImm64(opcode, ty, new_imm, new_arg);
                imm = new_imm;
                arg = new_arg;
            }
        }
    }
```

Adding some simple rewrites

```
(rule (simplify
       (iadd (fits_in_64 ty)
             (iconst ty (u64_from_imm64 k1))
             (iconst ty (u64_from_imm64 k2))))
      (subsume (iconst ty (imm64_masked ty (u64_add k1 k2)))))
```

(Cranelift's ISLE term-rewriting DSL)

Adding some simple rewrites

```
;; ineg(ineg(x)) == x.
(rule (simplify (ineg ty (ineg ty x))) (subsume x))
```

👍

Rewrite Systems for Optimization

- Many kinds of optimizations can be expressed as value rewrites
  - Constant prop (1 + 2 => 3), algebraic (x + 0 => x), strength reduction, …

Rewrite Systems for Optimization

- Many kinds of optimizations can be expressed as value rewrites
    - Constant prop (1 + 2 => 3), algebraic (x + 0 => x), strength reduction, …
- Those that can’t are often “rewrite-adjacent”
    - Normalization of input terms to rewriter => GVN
    - Placement of rewritten terms => LICM, code motion in general

Rewrite Systems for Optimization

- Many kinds of optimizations can be expressed as value rewrites
    - Constant prop (1 + 2 => 3), algebraic (x + 0 => x), strength reduction, …
- Those that can’t are often “rewrite-adjacent”
    - Normalization of input terms to rewriter => GVN
    - Placement of rewritten terms => LICM, code motion in general
- Rewriting is a well-defined framework that works well for verification!
    - “This value is equal to that value”

1. Why	*we want a rewrite-based optimizer*
2. **How**	*to turn a CFG into an egraph and back again*
3. Cycles	*why they occur, and what to do about them*
4. Results	*how well does it work?*
5. Lessons	*in translating research to production*

1. **Why** *we want a rewrite-based optimizer*

2. **How** *to turn a CFG into an egraph and back again*

3. **Cycles** *why they occur, and what to do about them*

4. **Results** *how well does it work?*

5. **Lessons** *in translating research to production*

Optimization pipeline

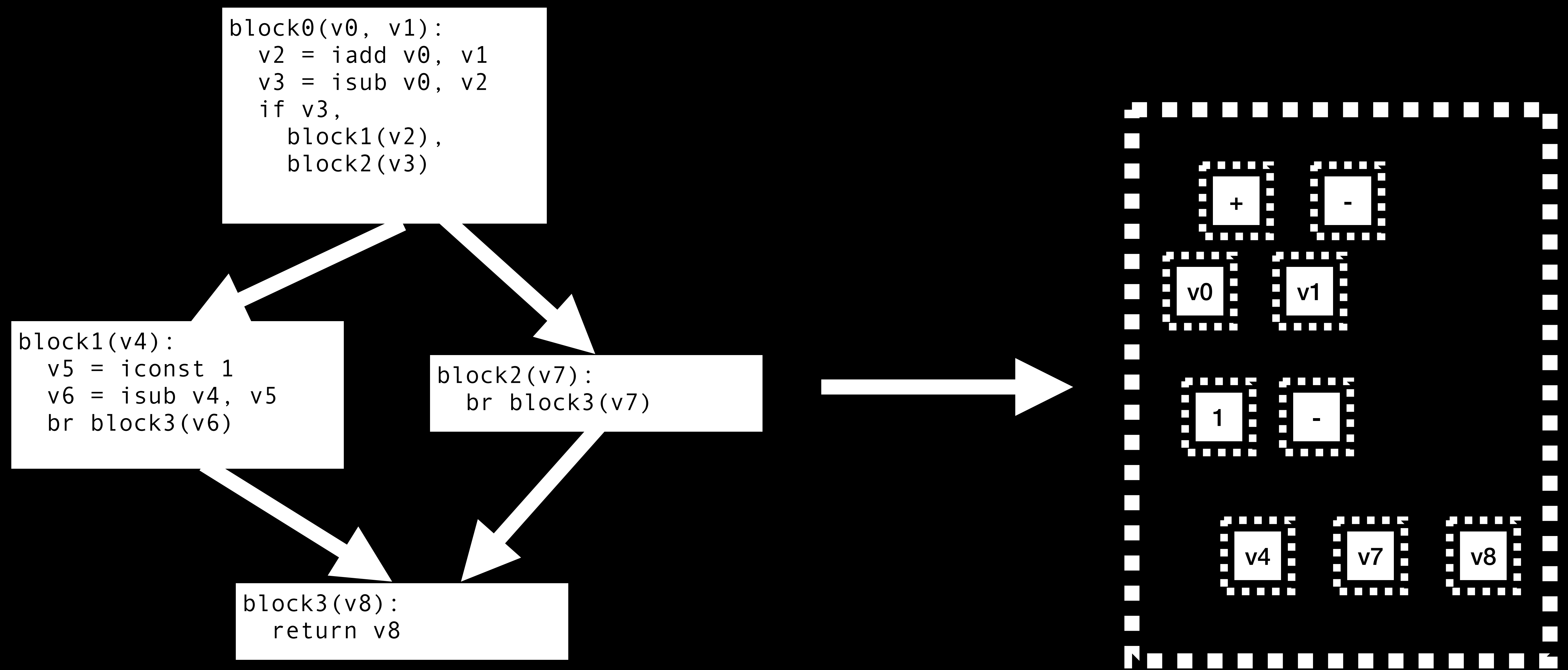
```
block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)
```

```
block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)
```

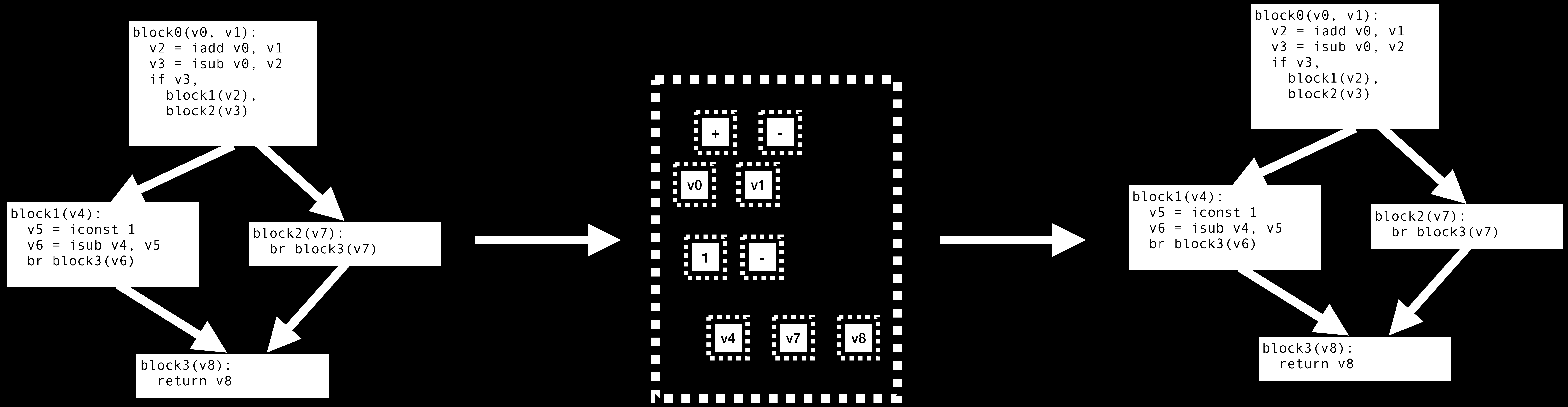
```
block2(v7):
  br block3(v7)
```

```
block3(v8):
  return v8
```

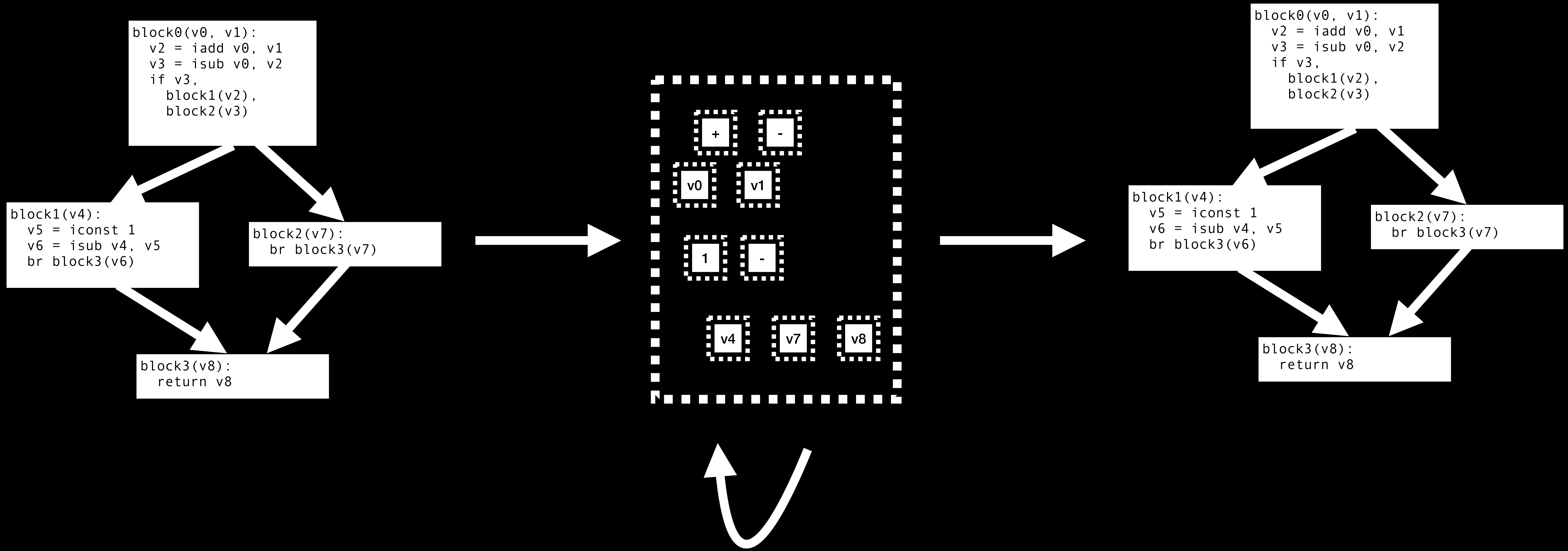
Optimization pipeline



Optimization pipeline



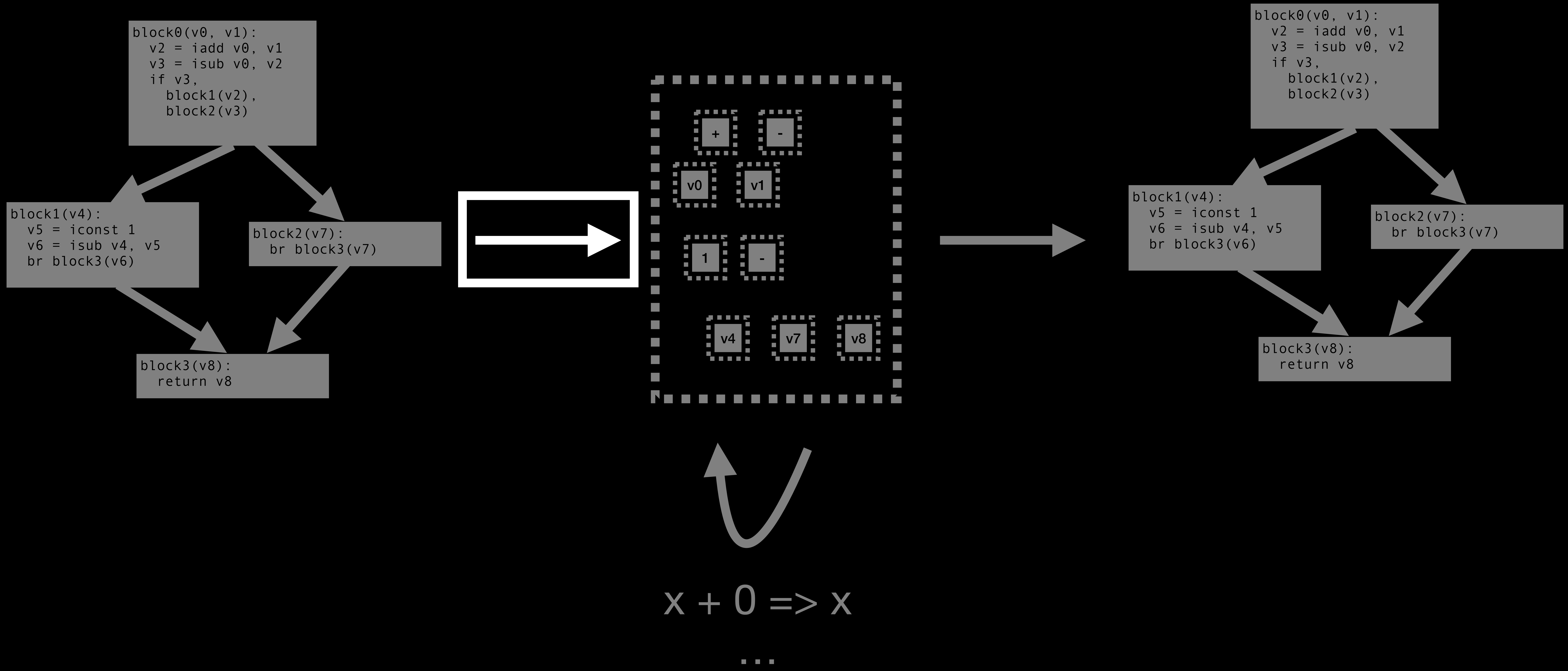
Optimization pipeline



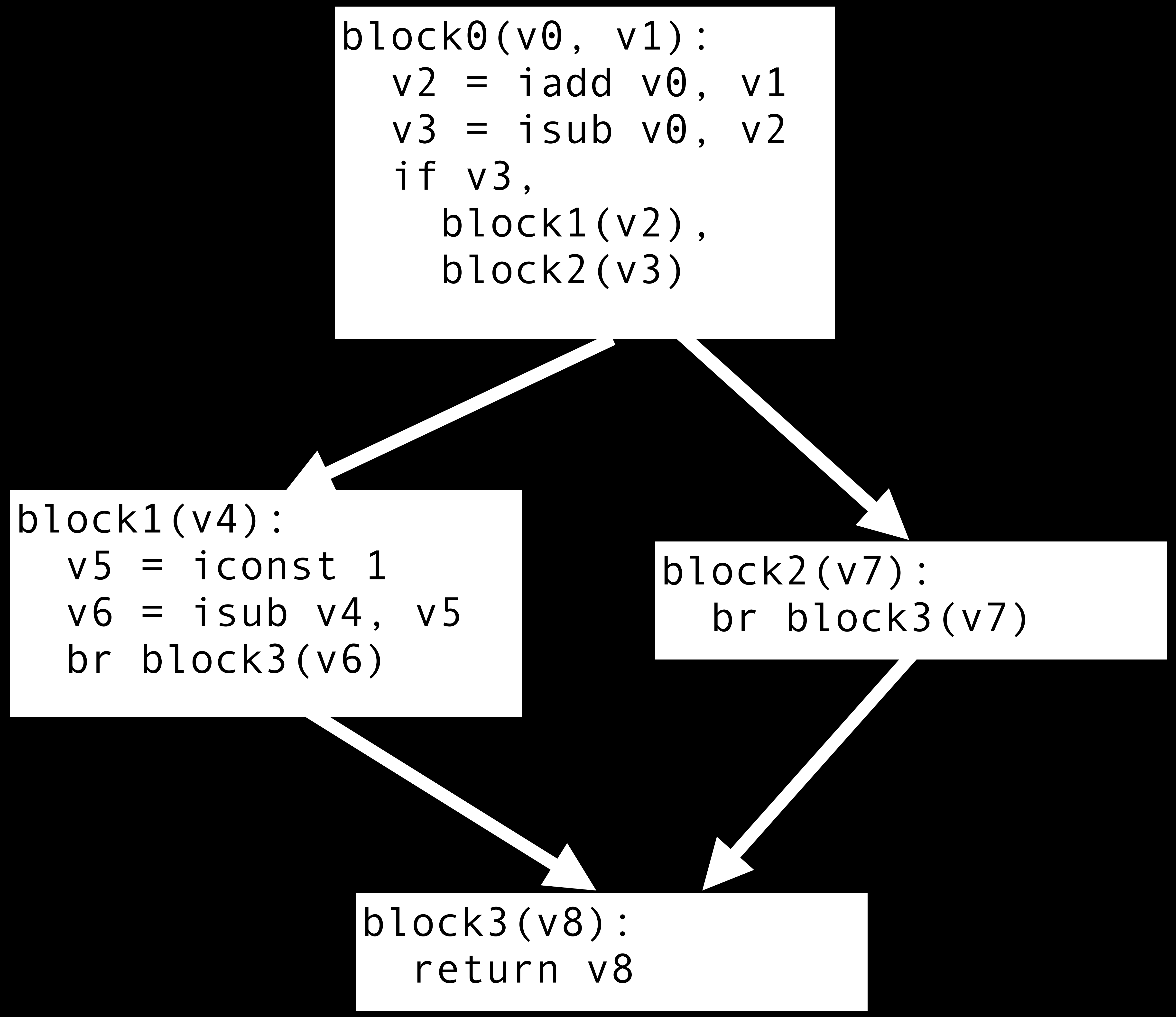
x + 0 => x

…

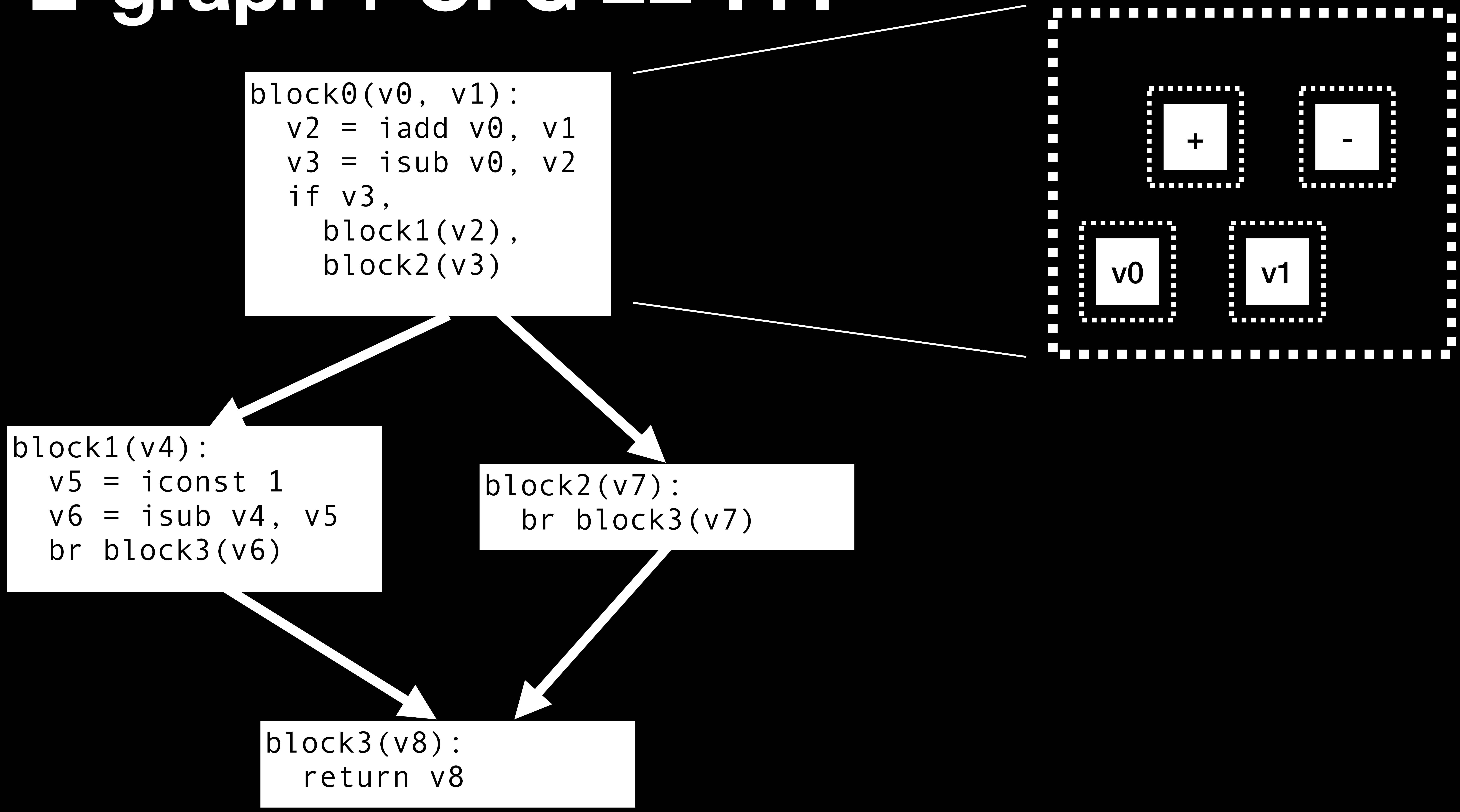
Optimization pipeline



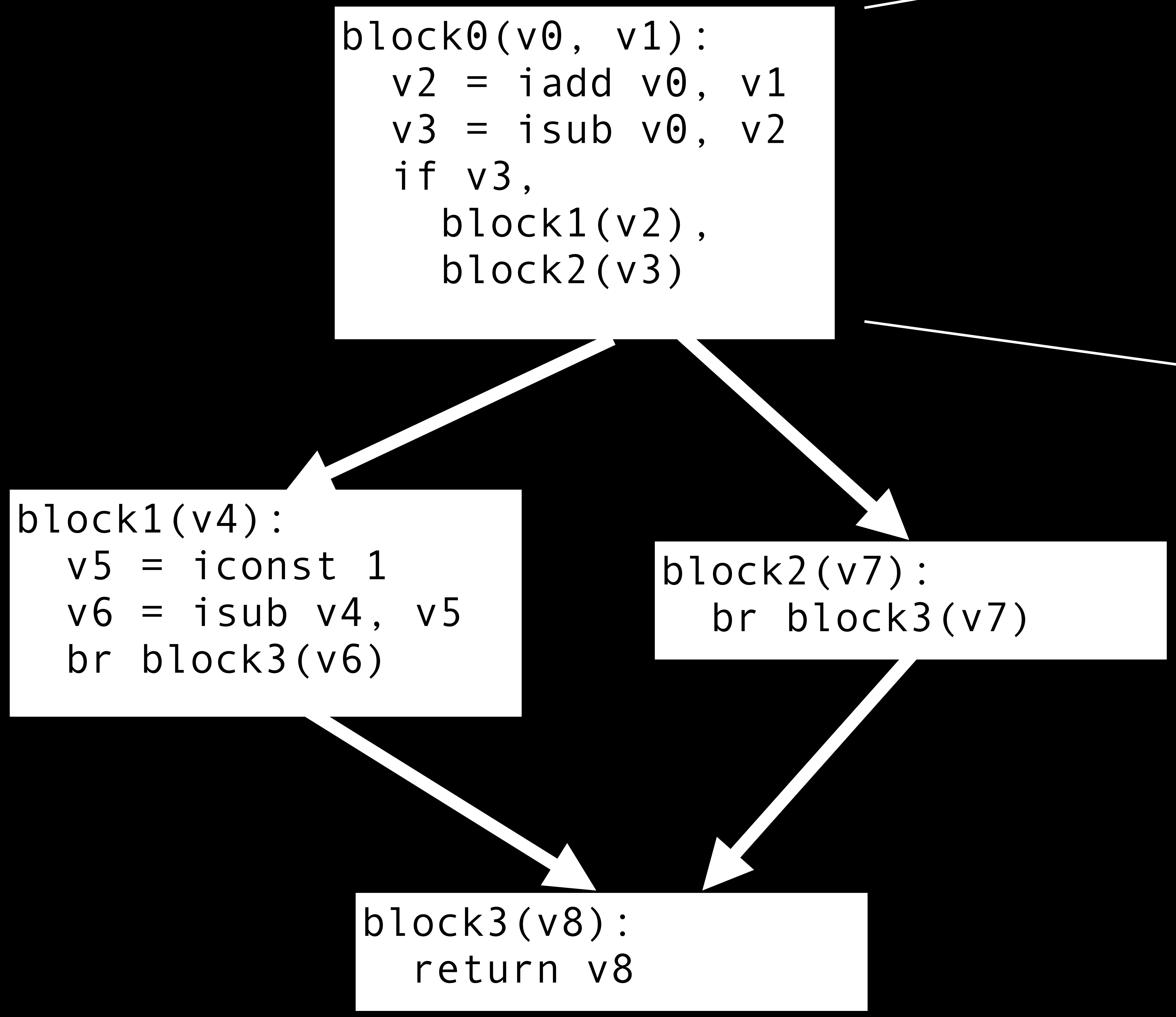
E-graph + CFG == ???

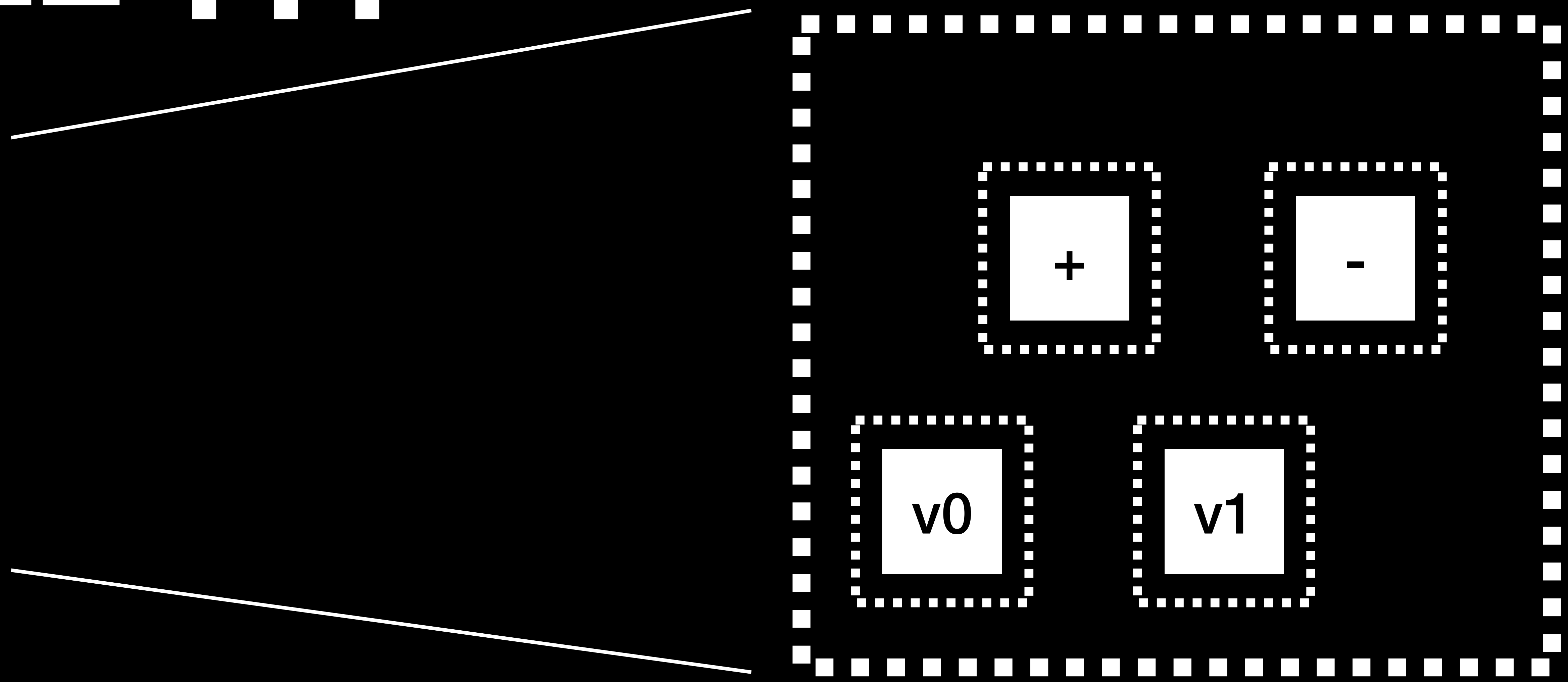

block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)
block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8

E-graph + CFG == ???


block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)
block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8
+
-
v0
v1

E-graph + CFG == ???

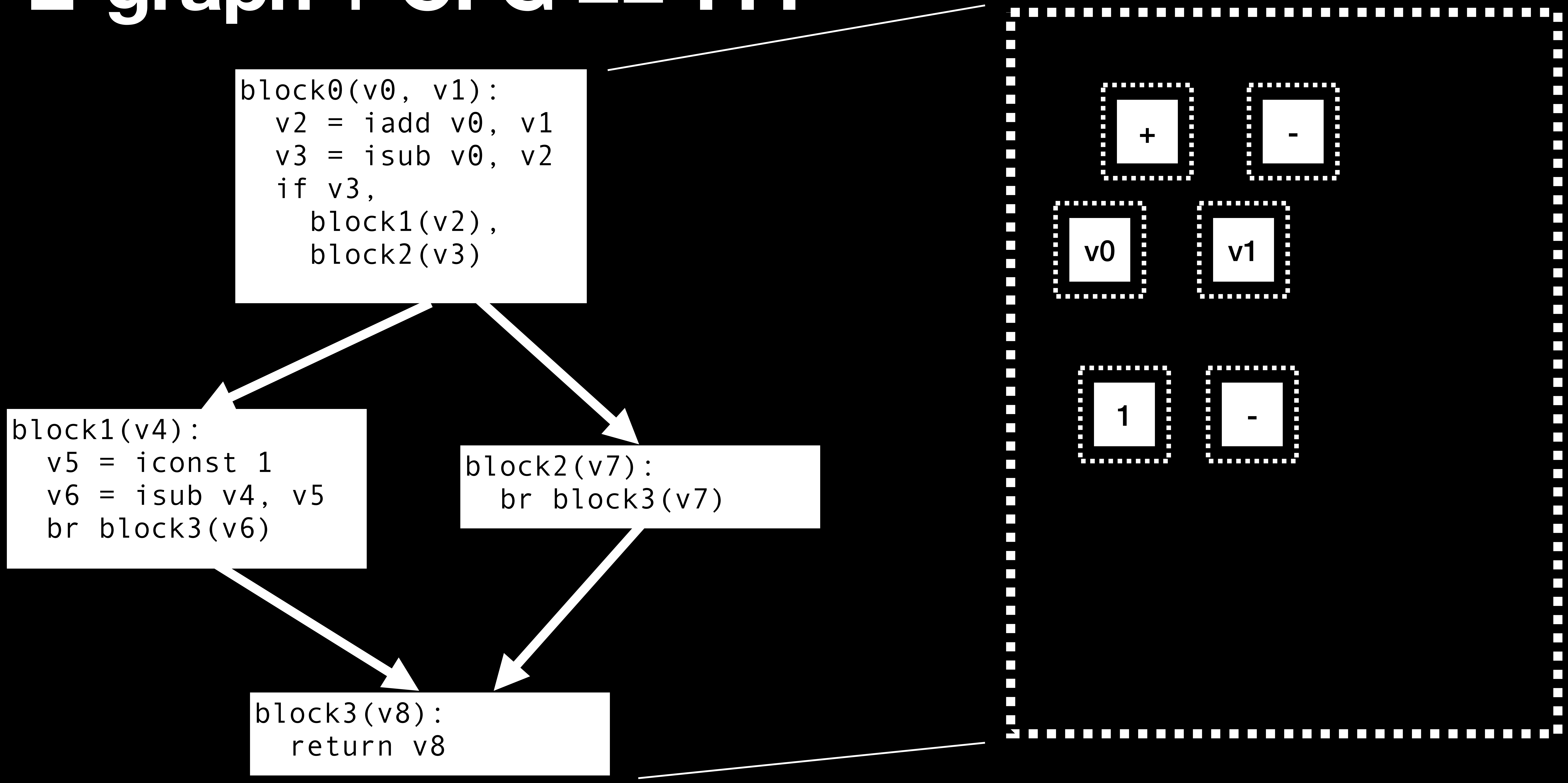




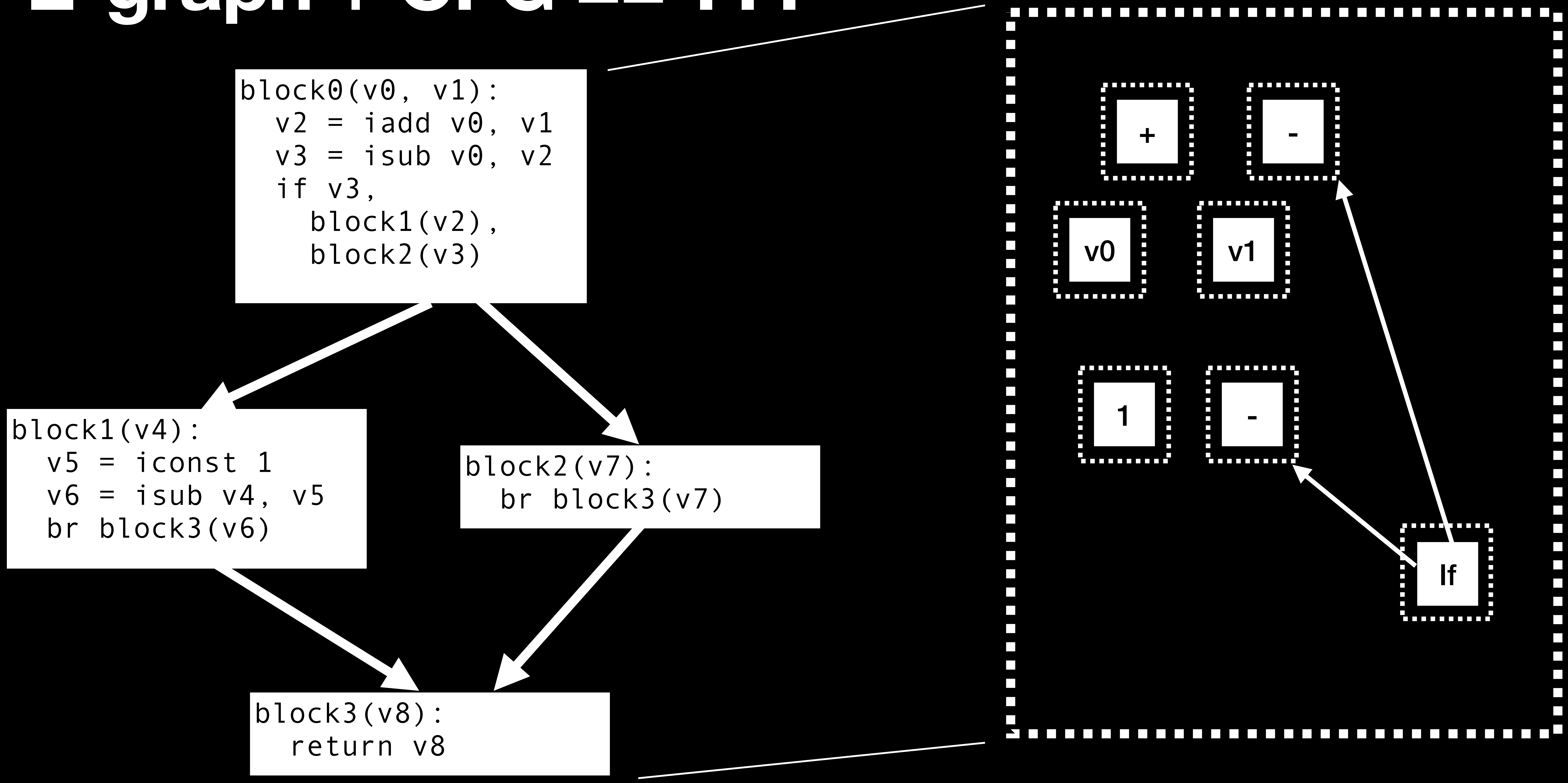
egraph per basic block:

\+ simple
- limited rewrite scope
- limited sharing/amortization
- rules out control optimizations

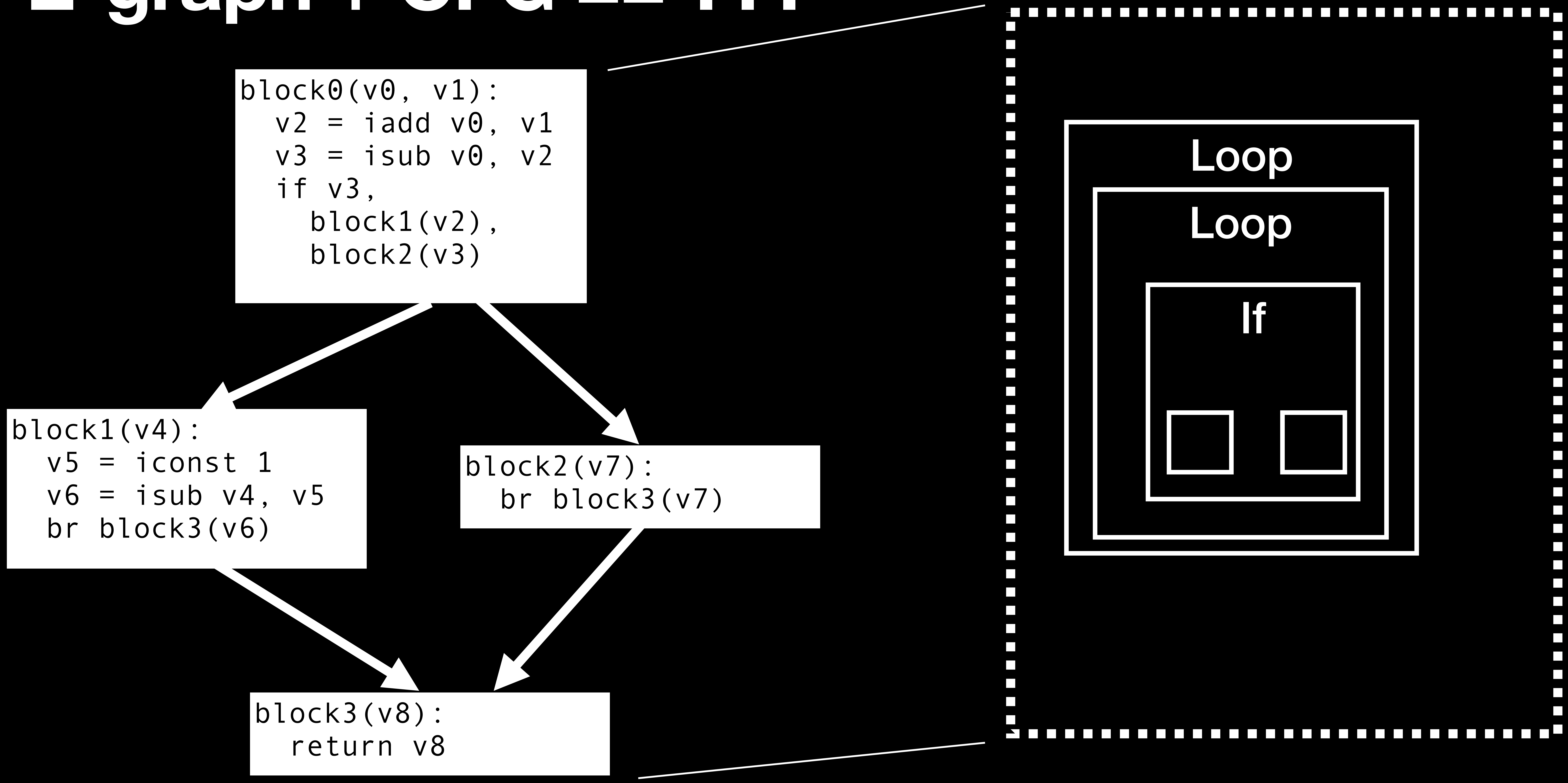
E-graph + CFG == ???


block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)
block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8
+
-
v0
v1
1
-

E-graph + CFG == ???


block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)
block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)
block2(v7):
  br block3(v7)
block3(v8):
  return v8
+
-
v0
v1
1
-
If

E-graph + CFG == ???

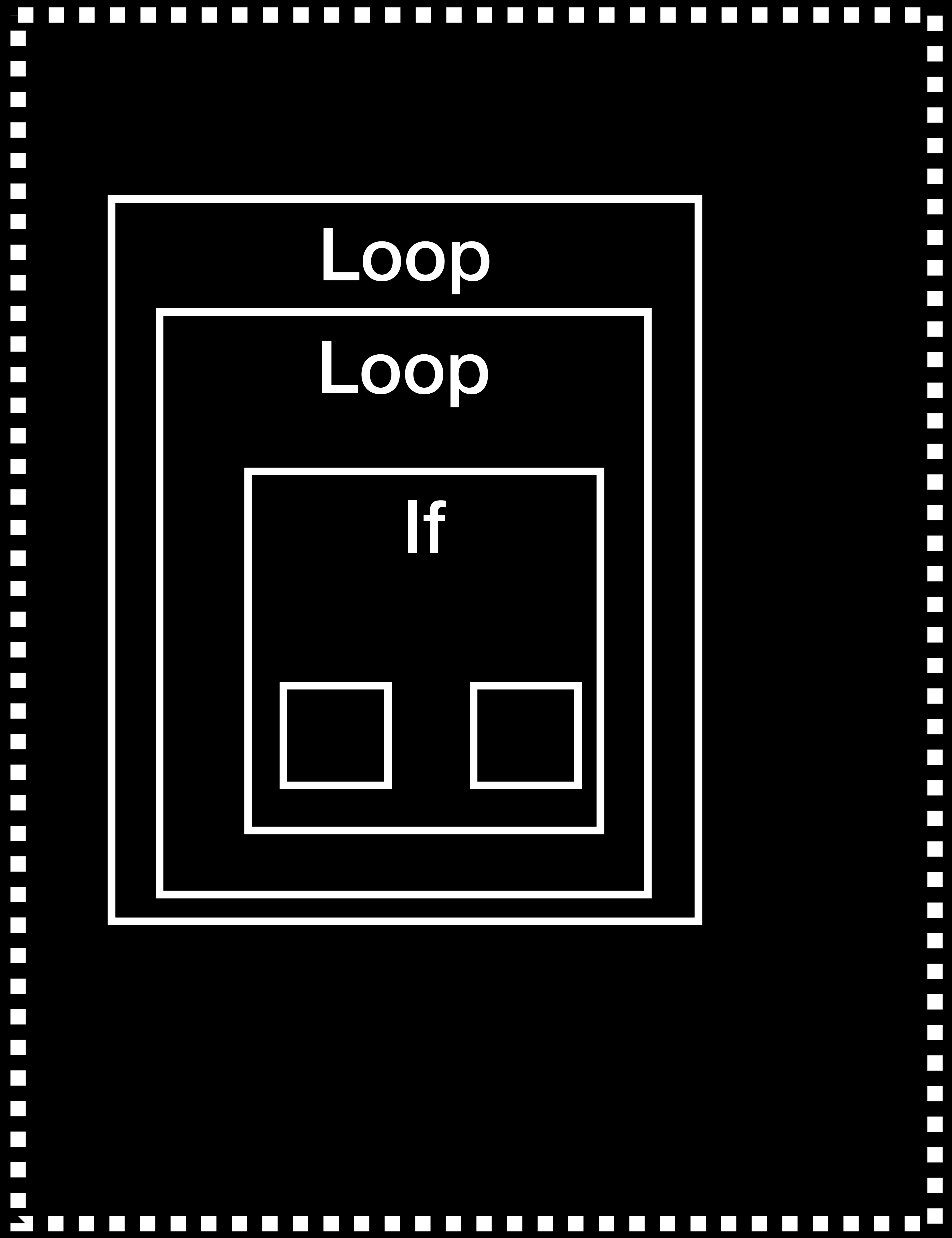

block0(v0, v1):
v2 = iadd v0, v1
v3 = isub v0, v2
if v3,
block1(v2),
block2(v3)
block1(v4):
v5 = iconst 1
v6 = isub v4, v5
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8
Loop
Loop
If

E-graph + CFG == ???

Region-nodes in egraph:

+ powerful optimizations!
+ strongly normalizing
+ more compact IR
+ cheaper analysis?

- very different from CFG
  (conversion overheads)
- side-effects are tricky
- issues with irreducible
  control flow

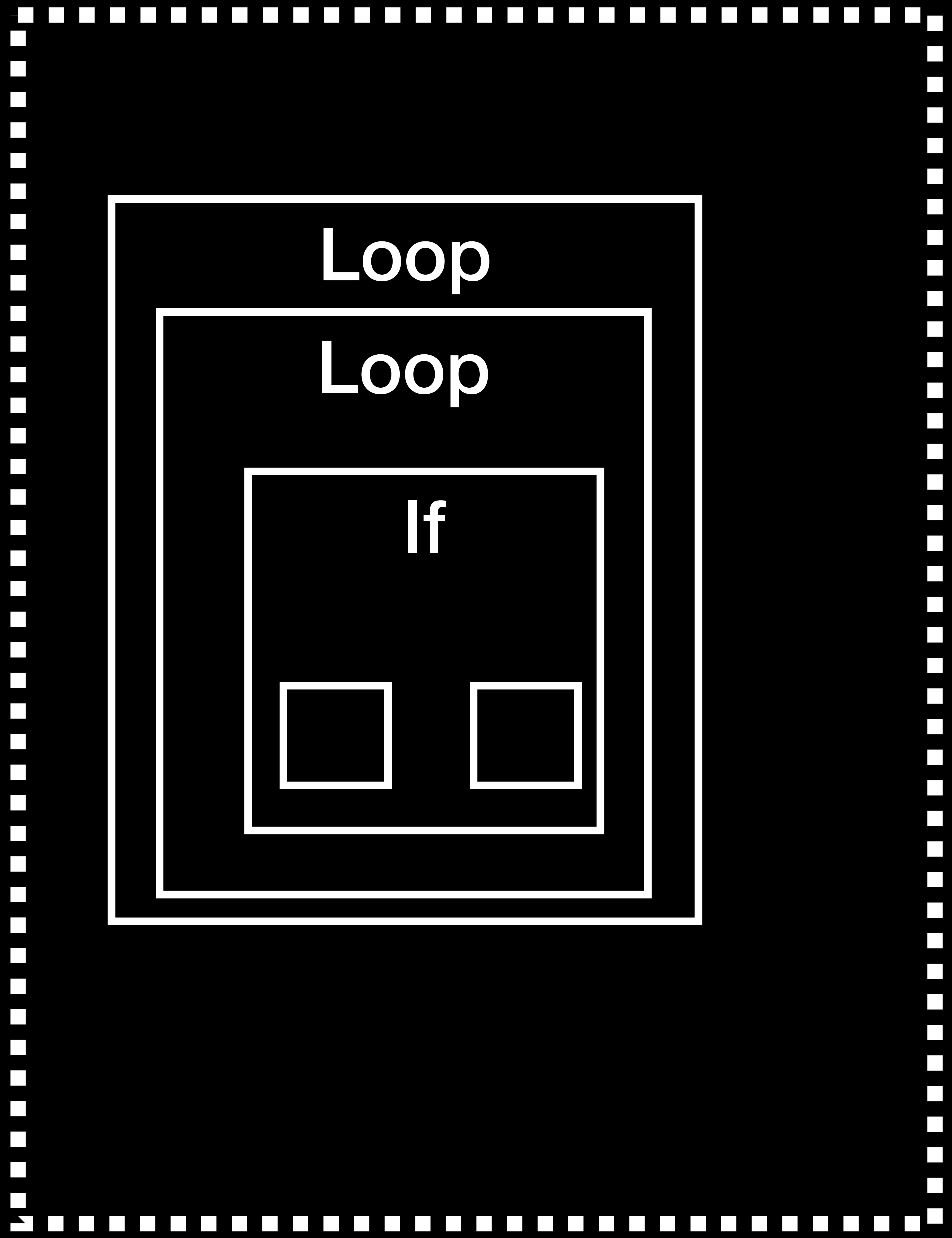


E-graph + CFG == ???

Region-nodes in egraph:

+ powerful optimizations!
+ strongly normalizing
+ more compact IR
+ cheaper analysis?

- very different from CFG (conversion overheads)
- side-effects are tricky
- issues with irreducible control flow



Jamey Sharp's prototype: https://github.com/jameysharp/optir

E-graph + CFG == ???

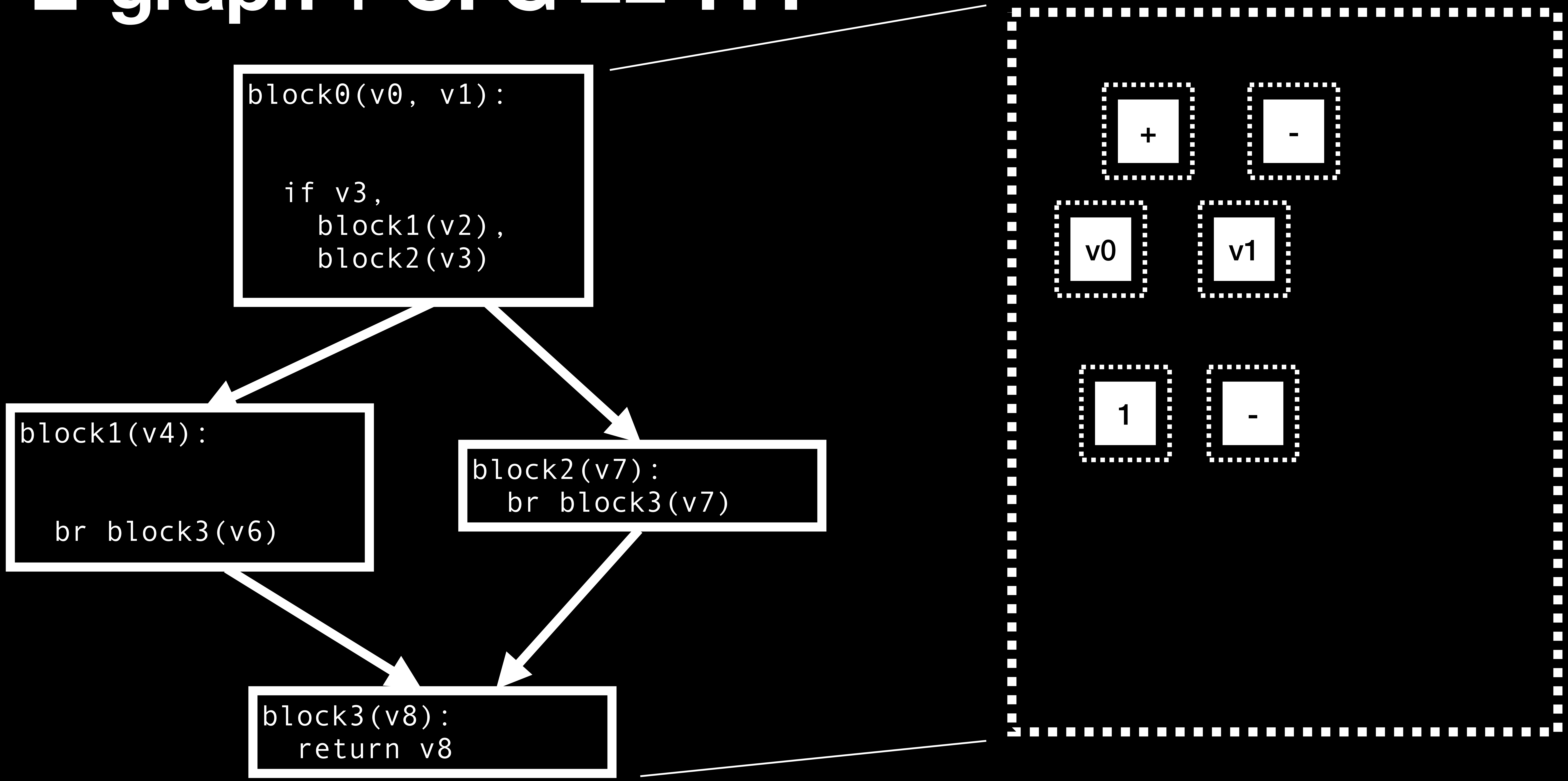
```
block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)
```

```
block1(v4):
  v5 = iconst 1
  v6 = isub v4, v5
  br block3(v6)
```

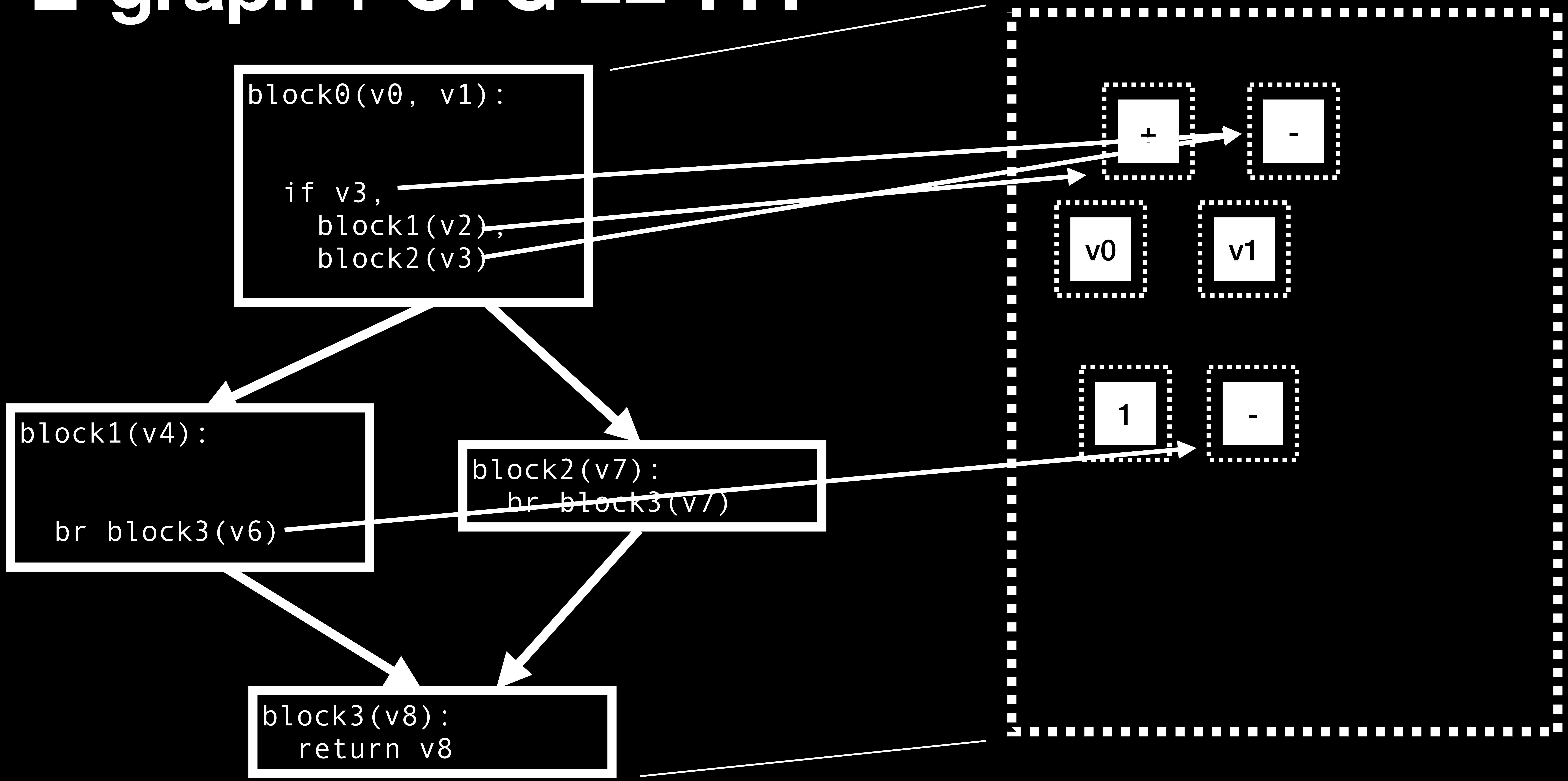
```
block2(v7):
  br block3(v7)
```

```
block3(v8):
  return v8
```

E-graph + CFG == ???

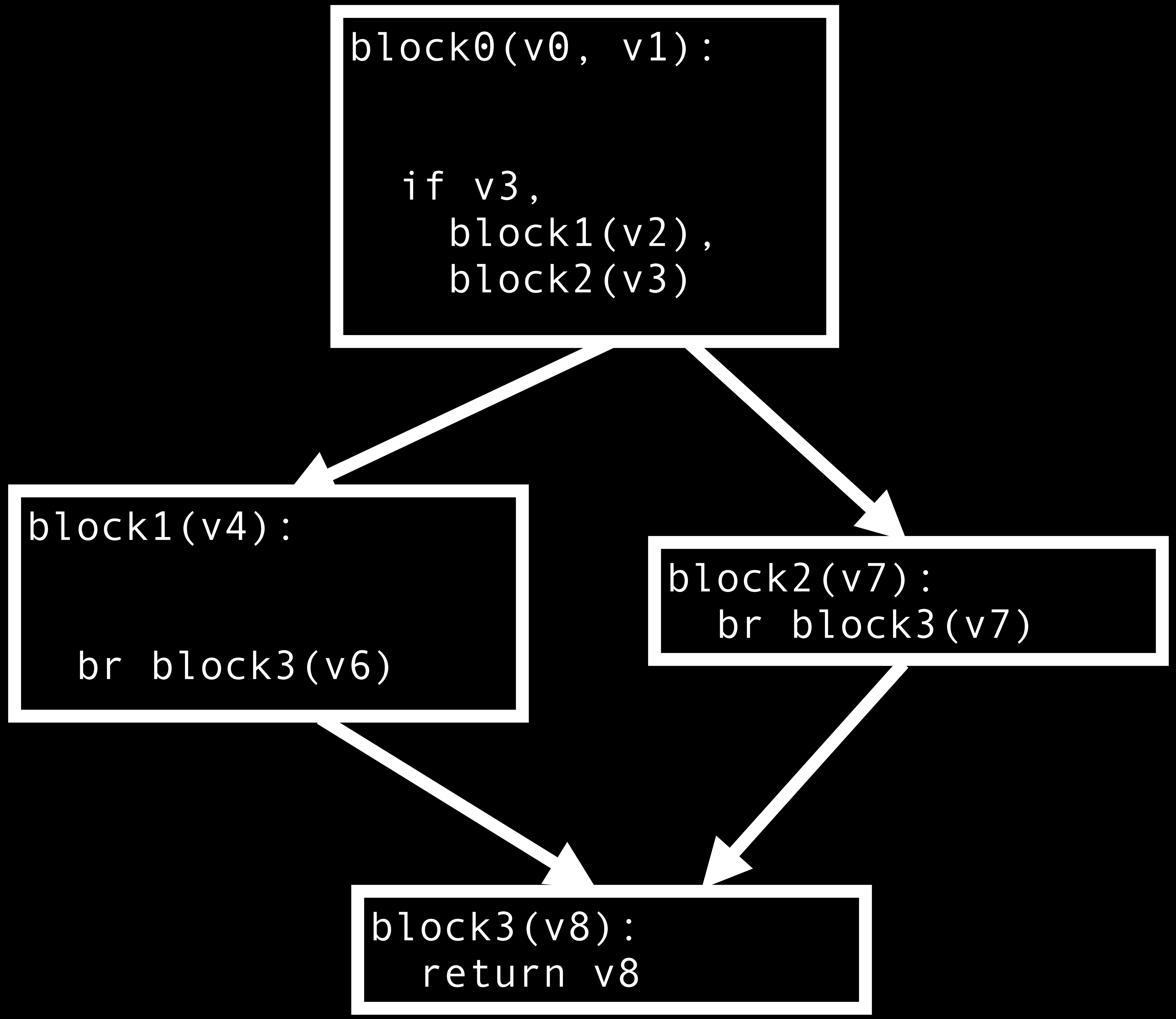

block0(v0, v1):
if v3,
block1(v2),
block2(v3)
block1(v4):
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8
+
-
v0
v1
1
-

E-graph + CFG == ???



block0(v0, v1):
if v3,
block1(v2),
block2(v3)
block1(v4):
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8
+
-
v0
v1
1
-

E-graph + CFG == ???



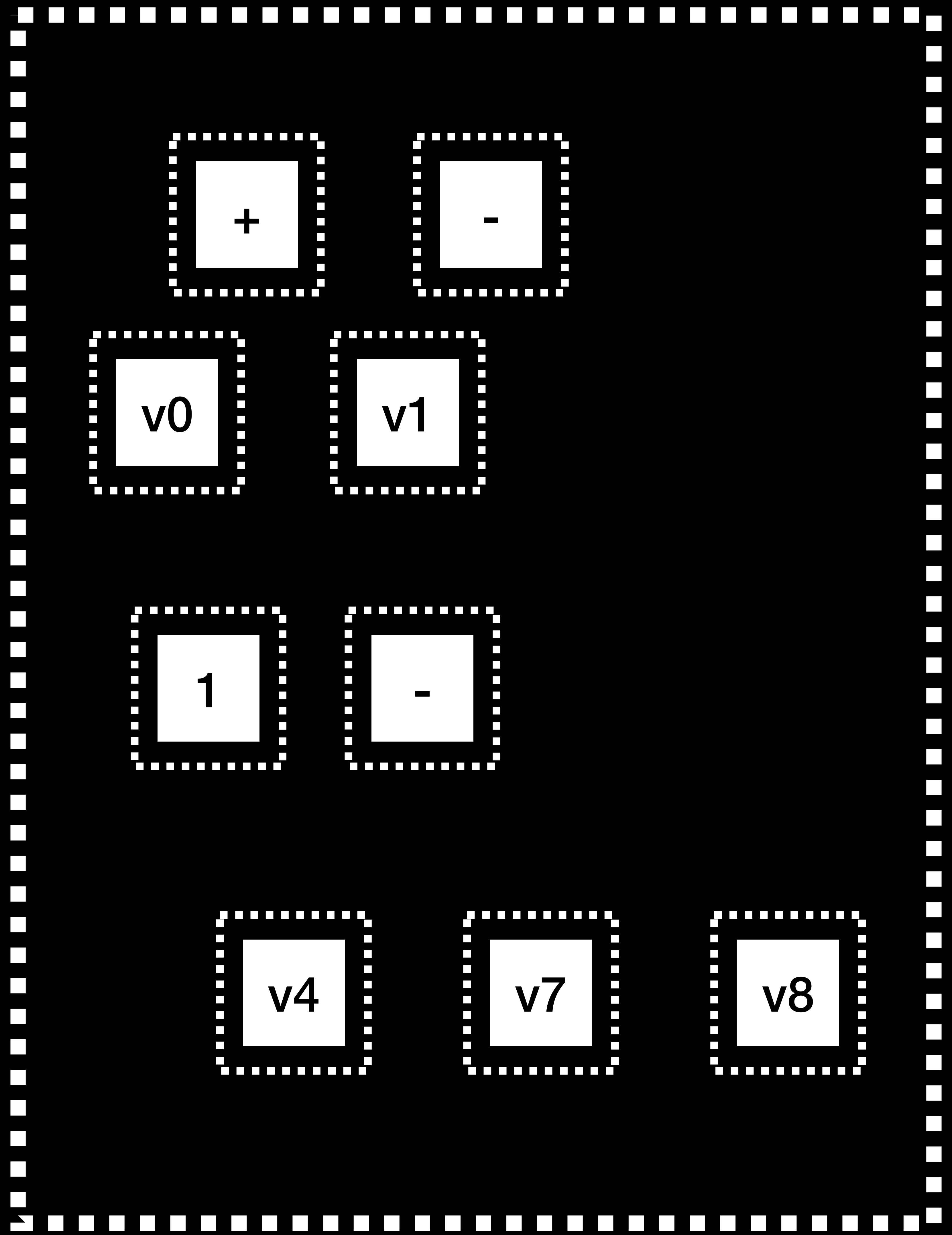
CFG skeleton contains:

- all blocks, with blockparams
- side-effecting operators
- block terminators (branches)

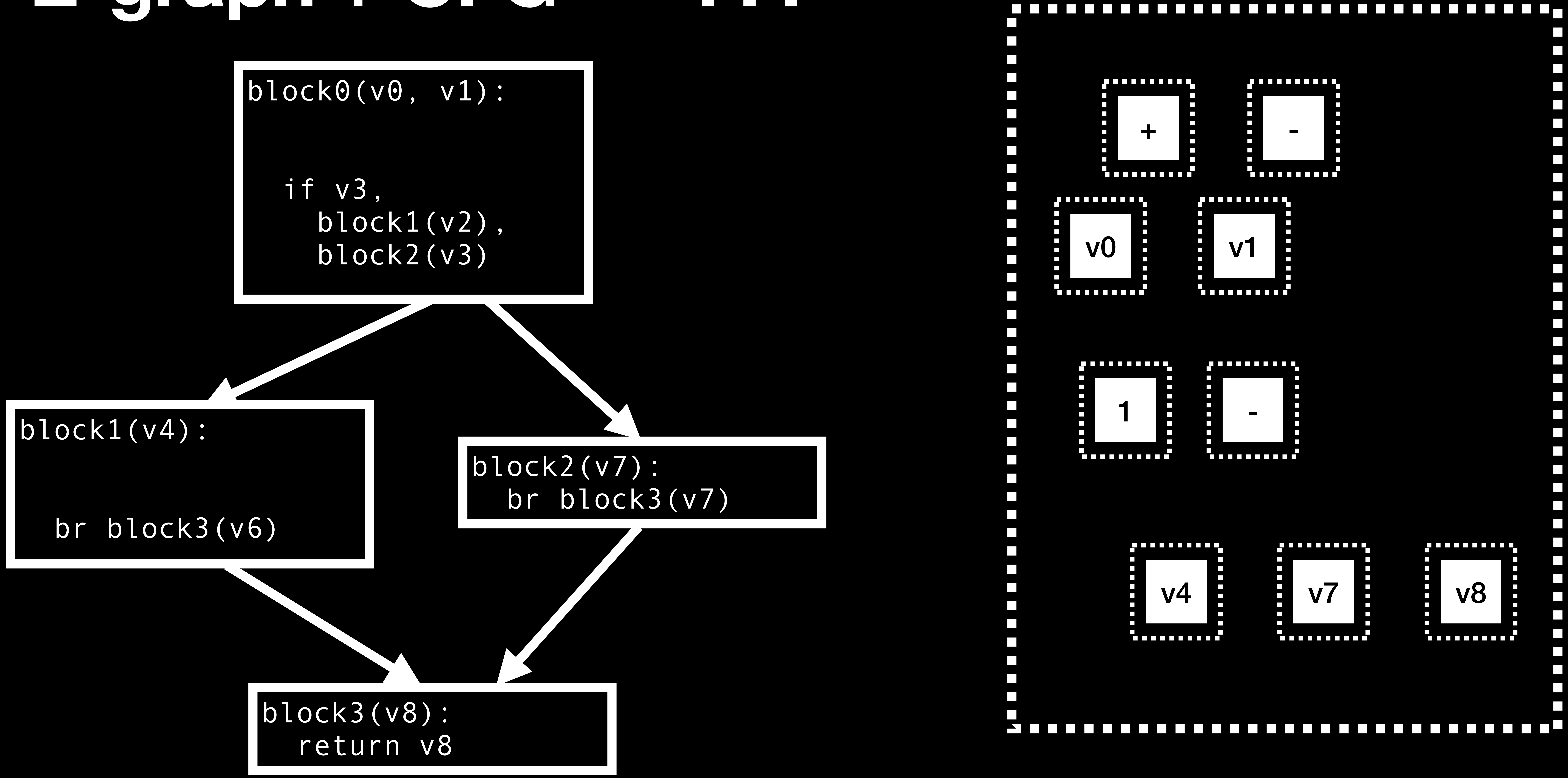
E-graph + CFG == ???

egraph contains:

- blockparam values, as terminals
- all pure operators, *without* associated location



E-graph + CFG == ???


block0(v0, v1):
if v3,
block1(v2),
block2(v3)
block1(v4):
br block3(v6)
block2(v7):
br block3(v7)
block3(v8):
return v8
+
-
v0
v1
1
-
v4
v7
v8

E-graph + CFG == ???

egraph with CFG skeleton:

+ cheap to convert to/from CFG
    + algorithmically *and* in implementation
+ optimizations across function scope (mostly)

- harder to express rewrites that alter side-effects
- need special support for “seeing through” blockparams

E-graph + CFG == ???

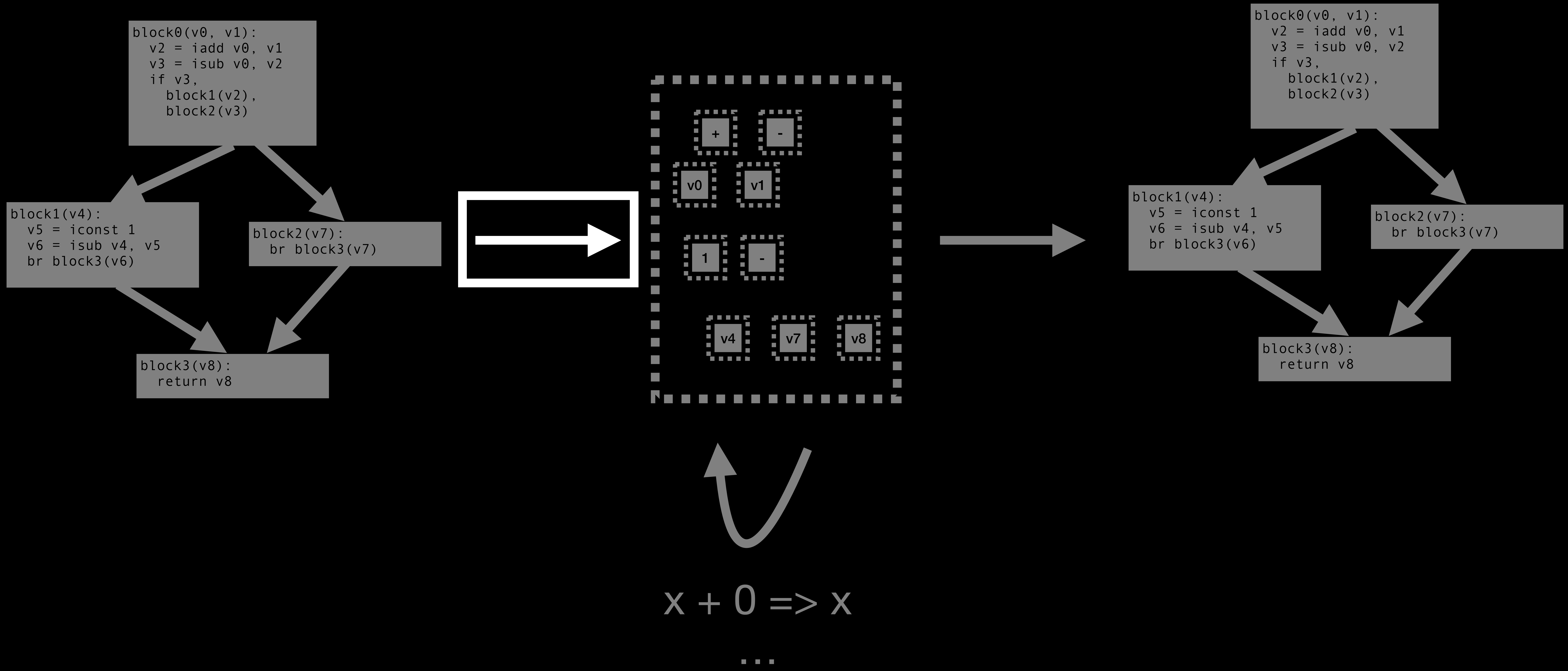
egraph with CFG skeleton:

+ cheap to convert to/from CFG
  + algorithmically *and* in implementation
+ optimizations across function scope (mostly)

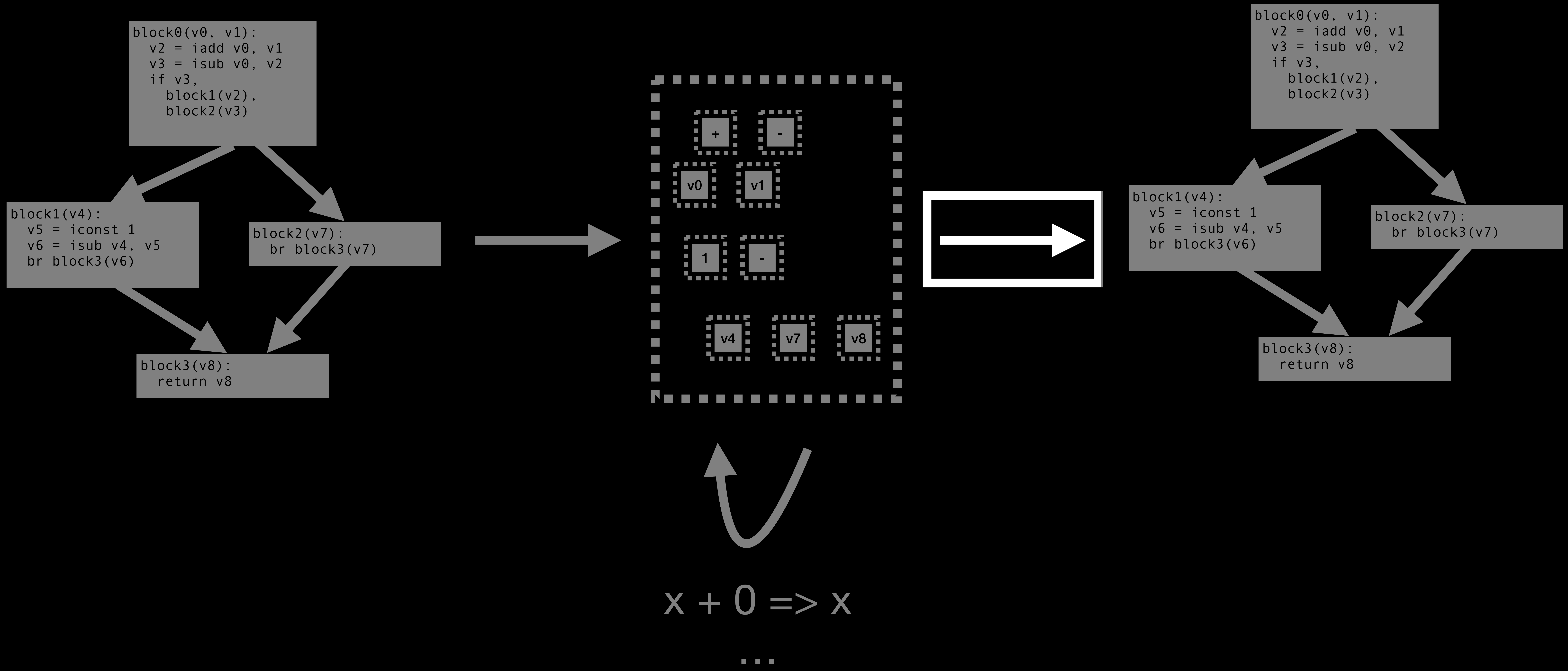
- harder to express rewrites that alter side-effects
- need special support for “seeing through” blockparams

→ good enough for now! (incremental approach)

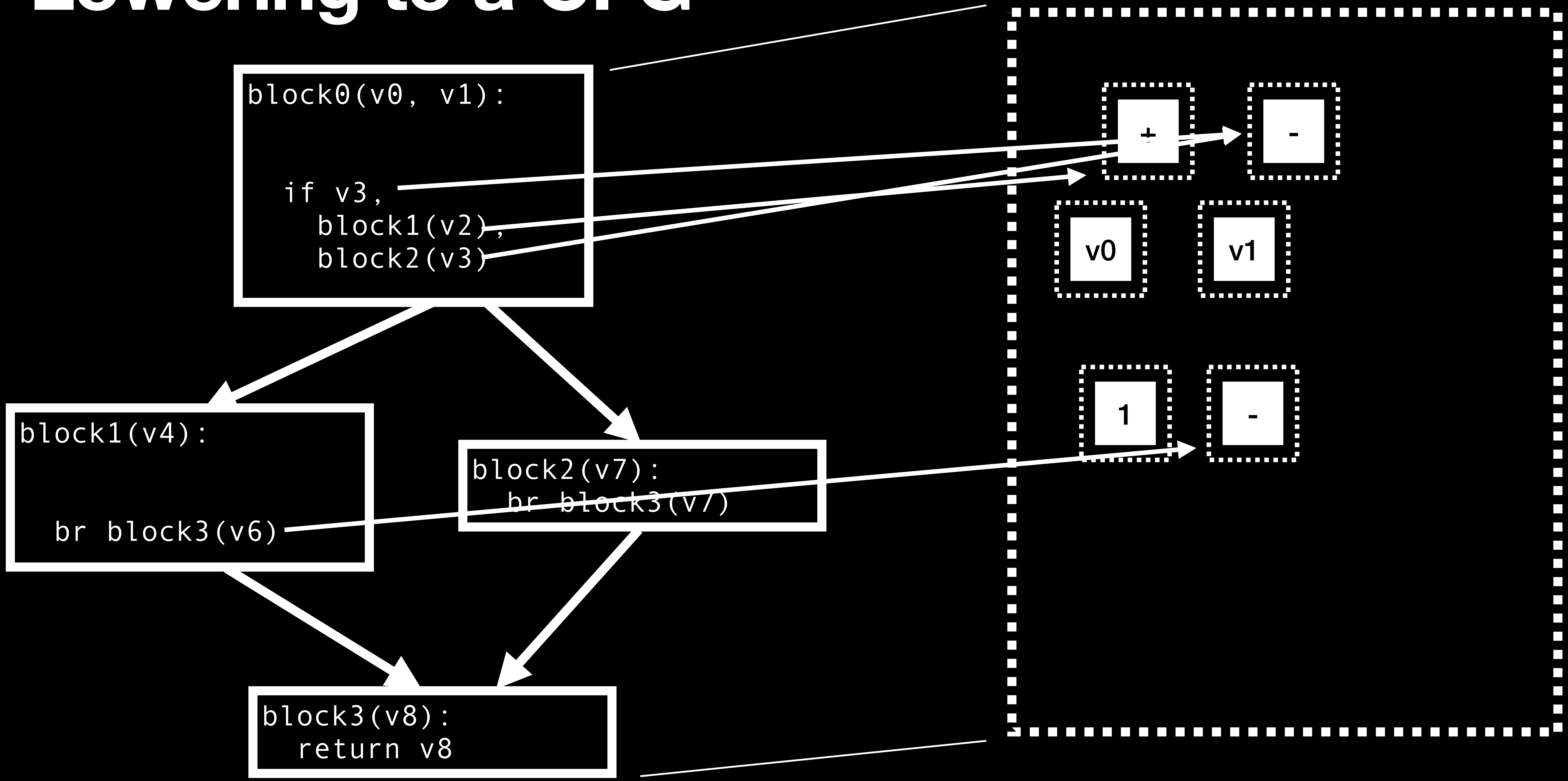
Optimization pipeline



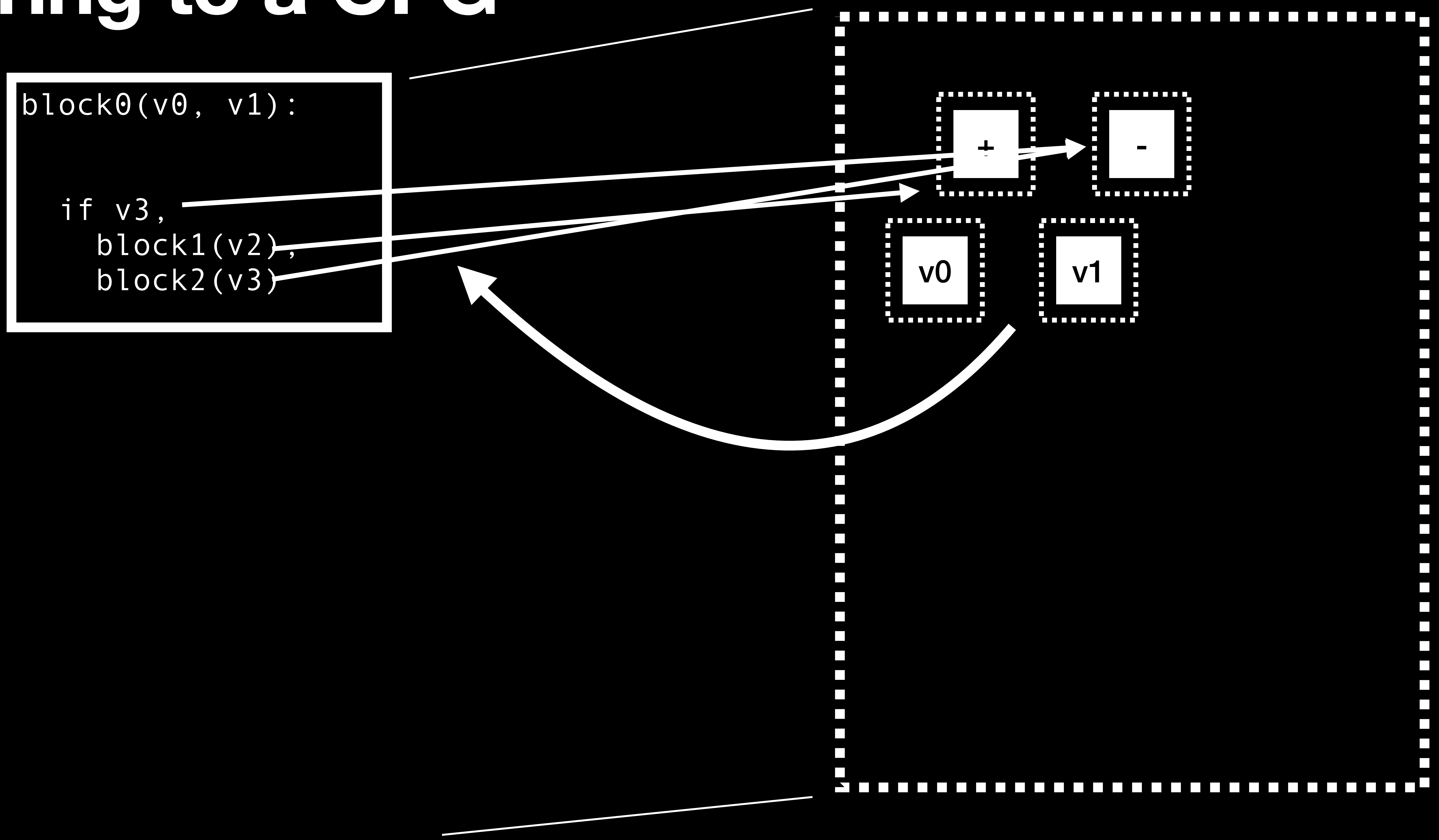
Optimization pipeline



Lowering to a CFG

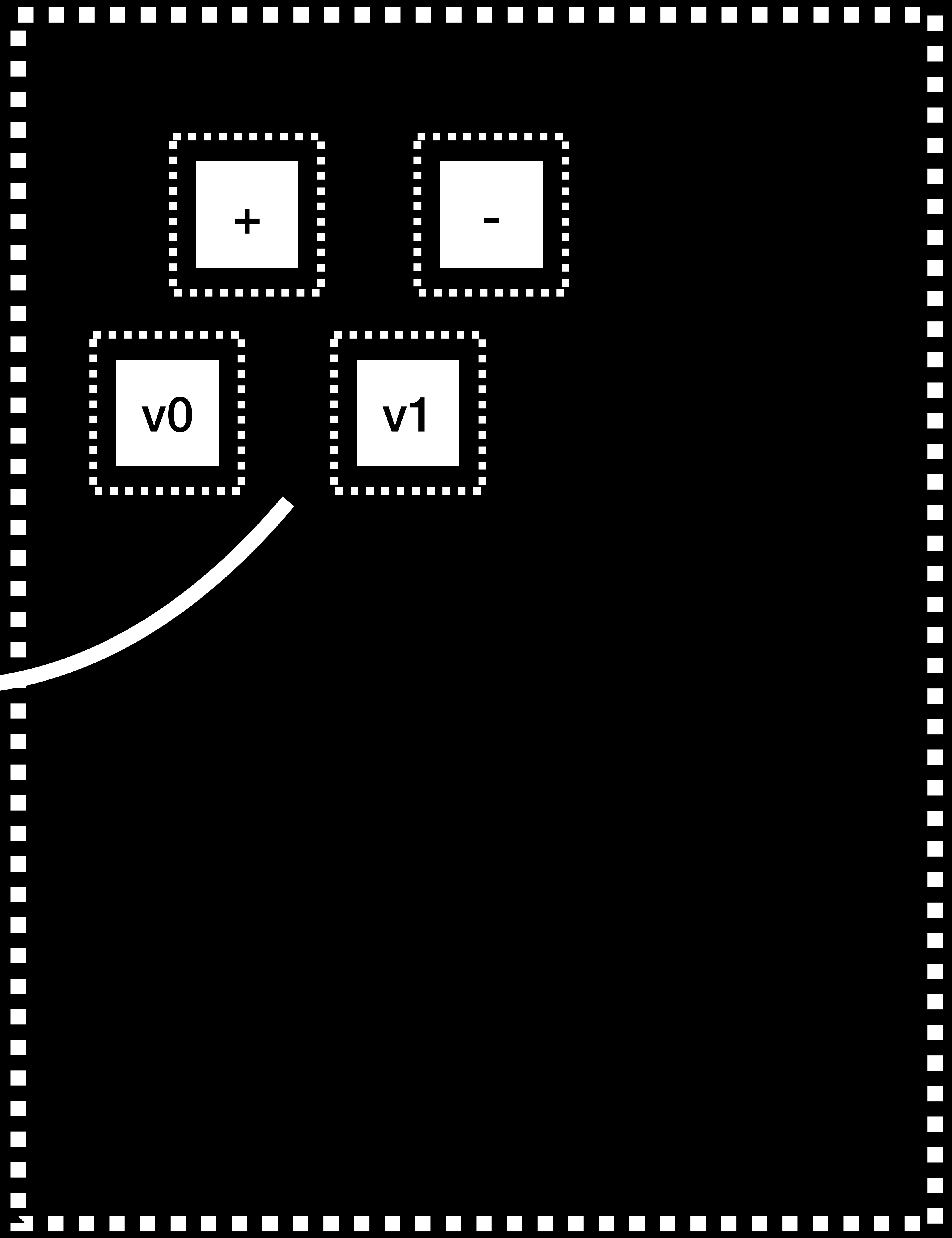


Lowering to a CFG



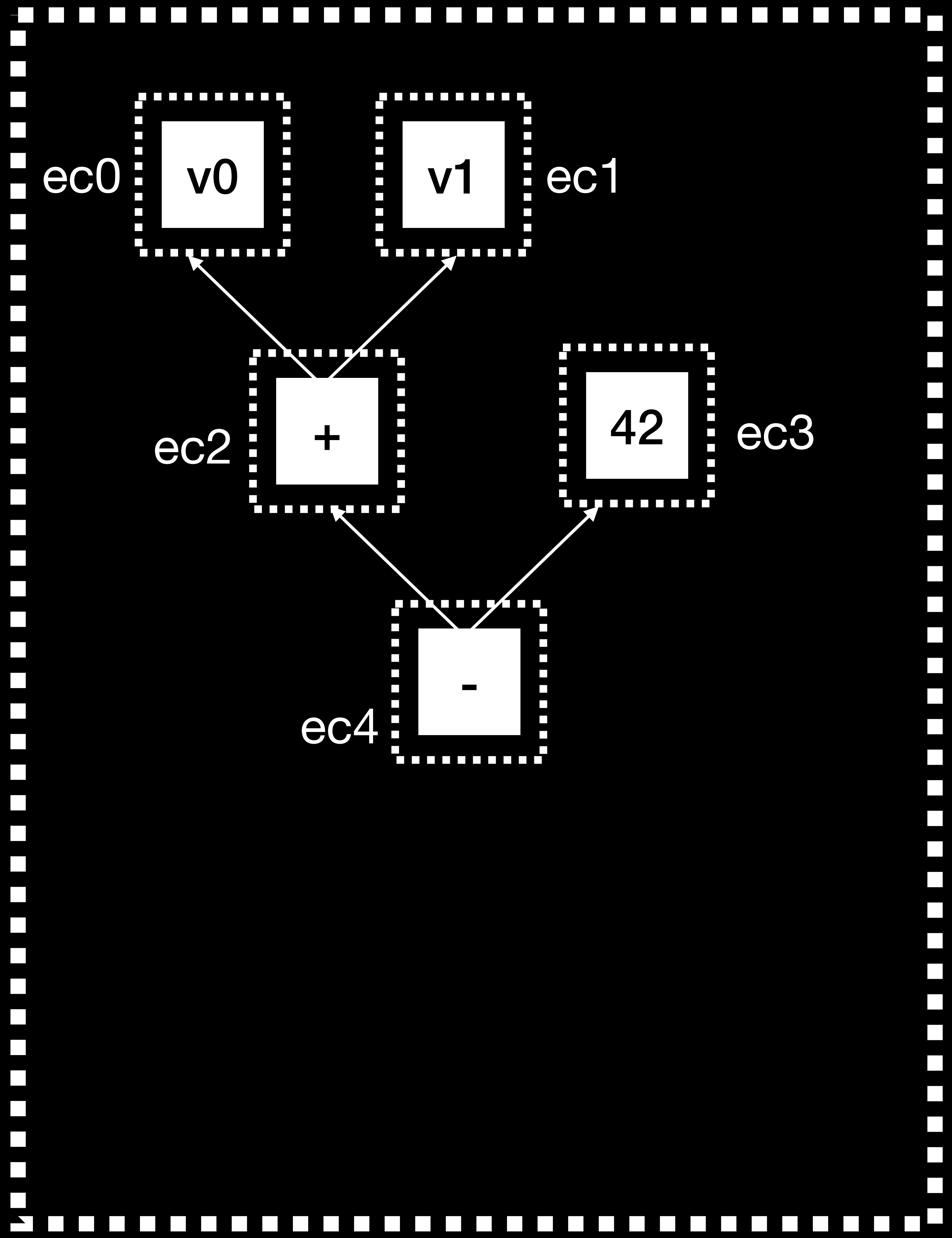
Lowering to a CFG

```
block0(v0, v1):
  v2 = iadd v0, v1
  v3 = isub v0, v2
  if v3,
    block1(v2),
    block2(v3)
```



Elaboration

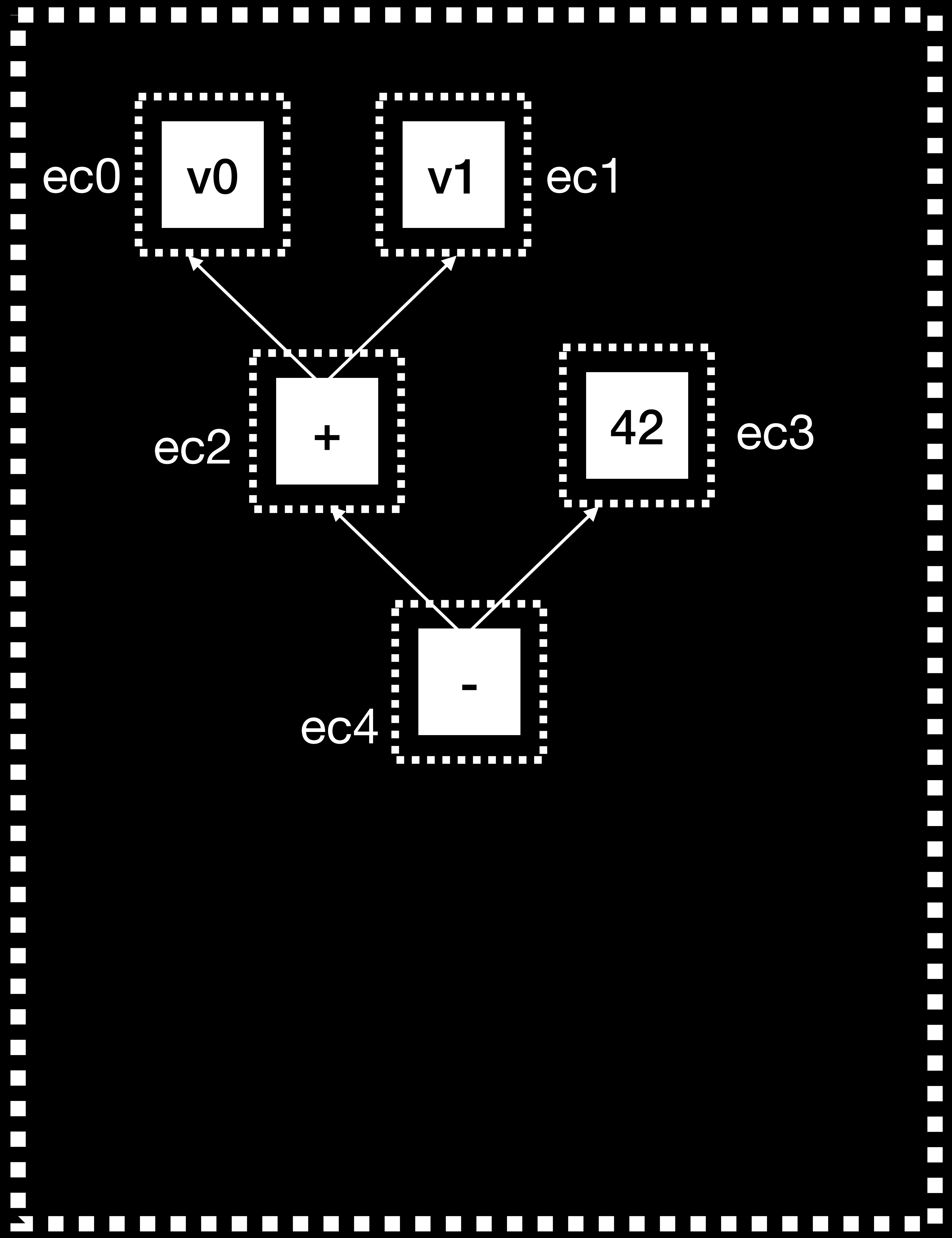
```
block0(v0, v1):
  store ec0, ec4
  return ec2
```



Elaboration

```
block0(v0, v1):
  store ec0, ec4
  return ec2
```

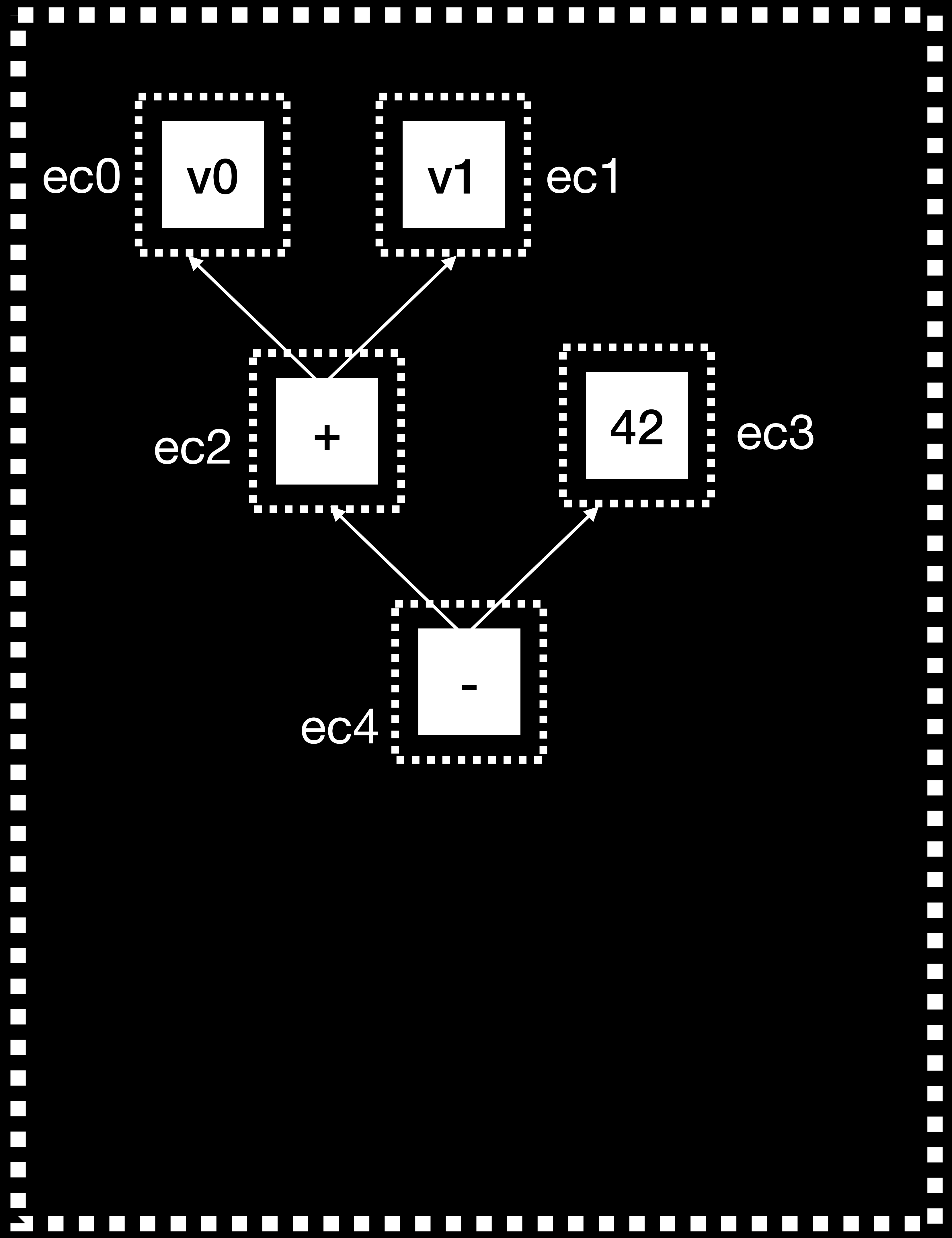
eclass	elaborated
ec0	v0
ec1	v1



Elaboration

```
block0(v0, v1):
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```

eclass	elaborated
ec0	v0
ec1	v1
ec4	v2

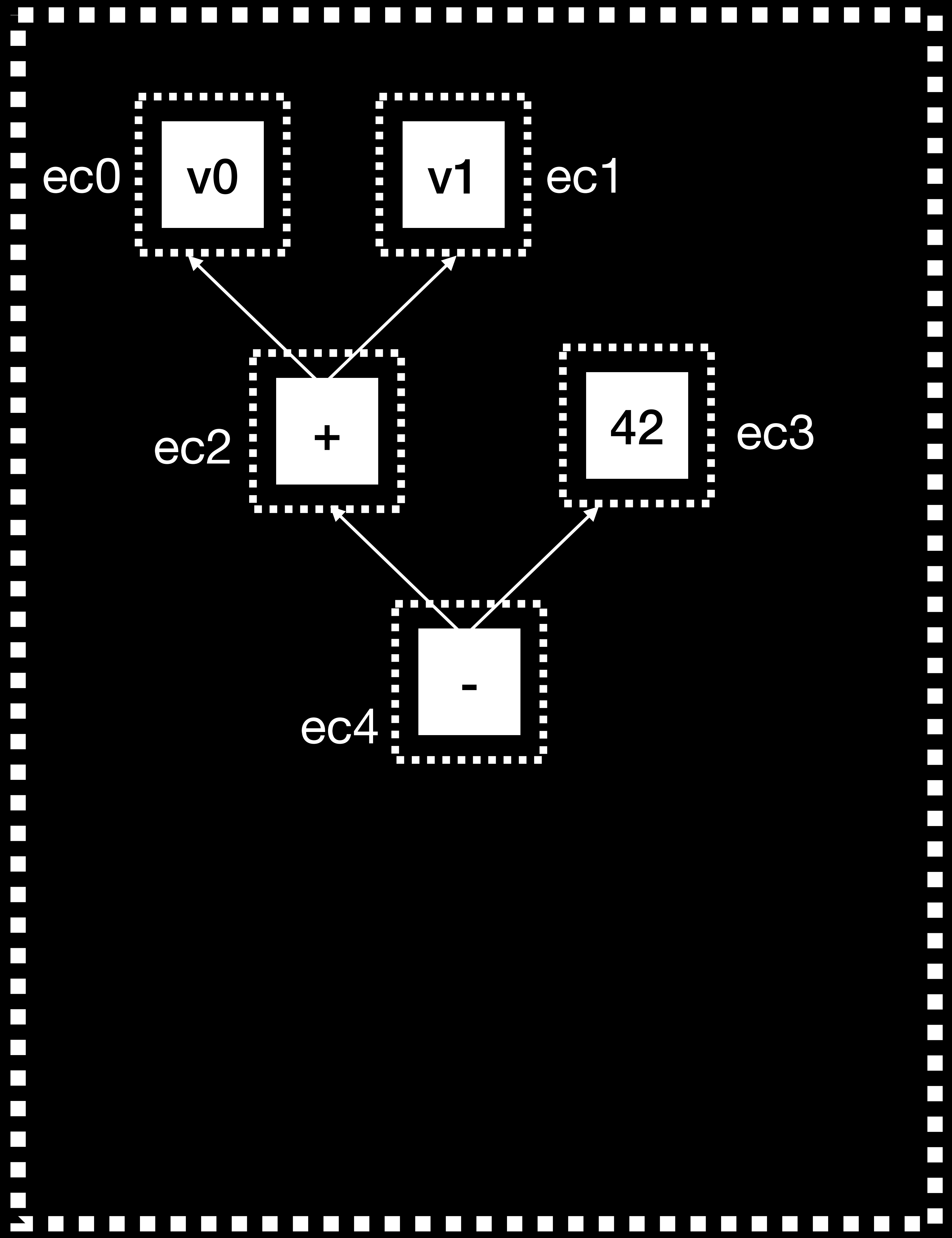


** Note: assume extraction (node selection) is done already!*

Elaboration

```
block0(v0, v1):
  v3 = iadd ec0, ec1
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```

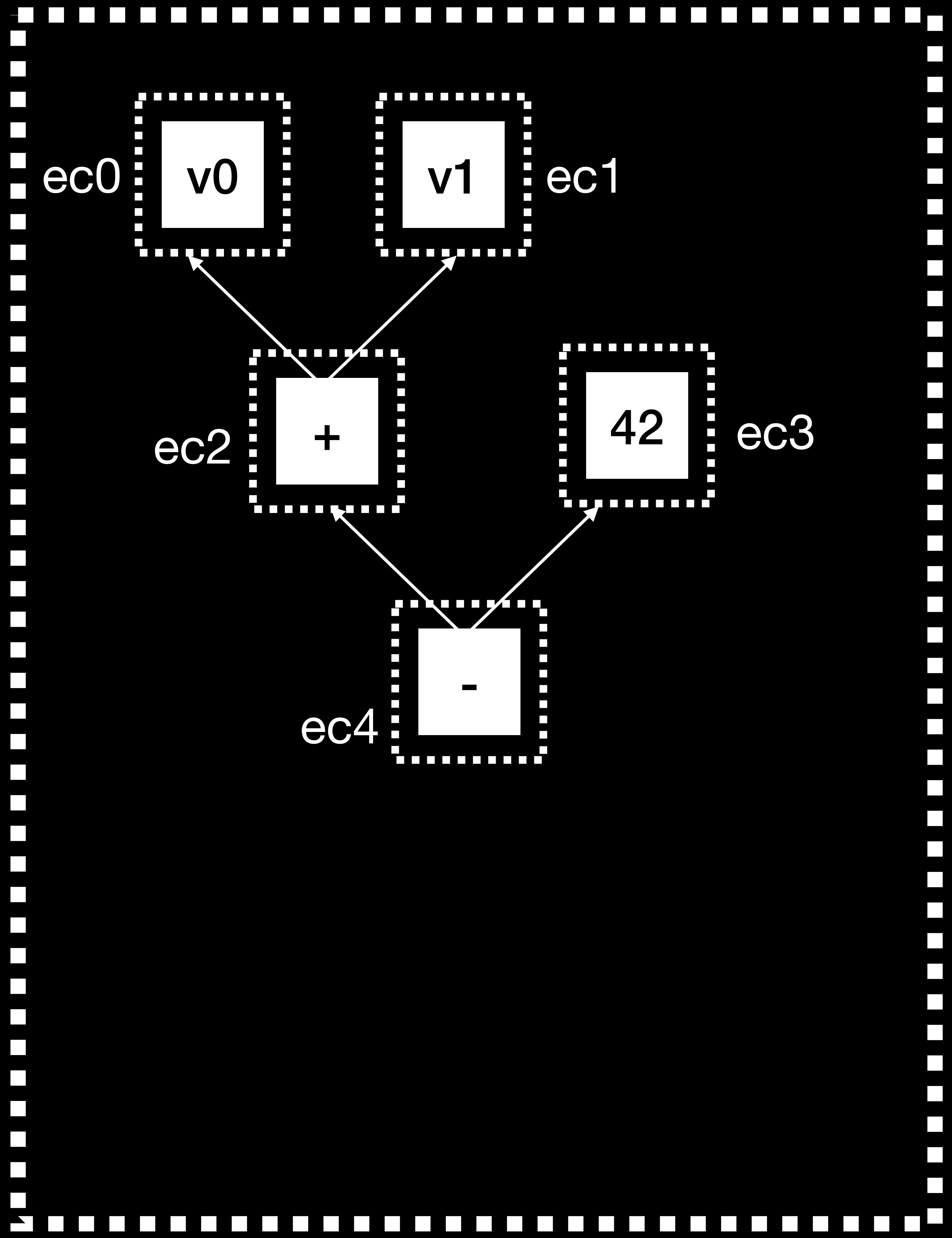
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec4	v2



Elaboration

```
block0(v0, v1):
  v3 = iadd v0, v1
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```

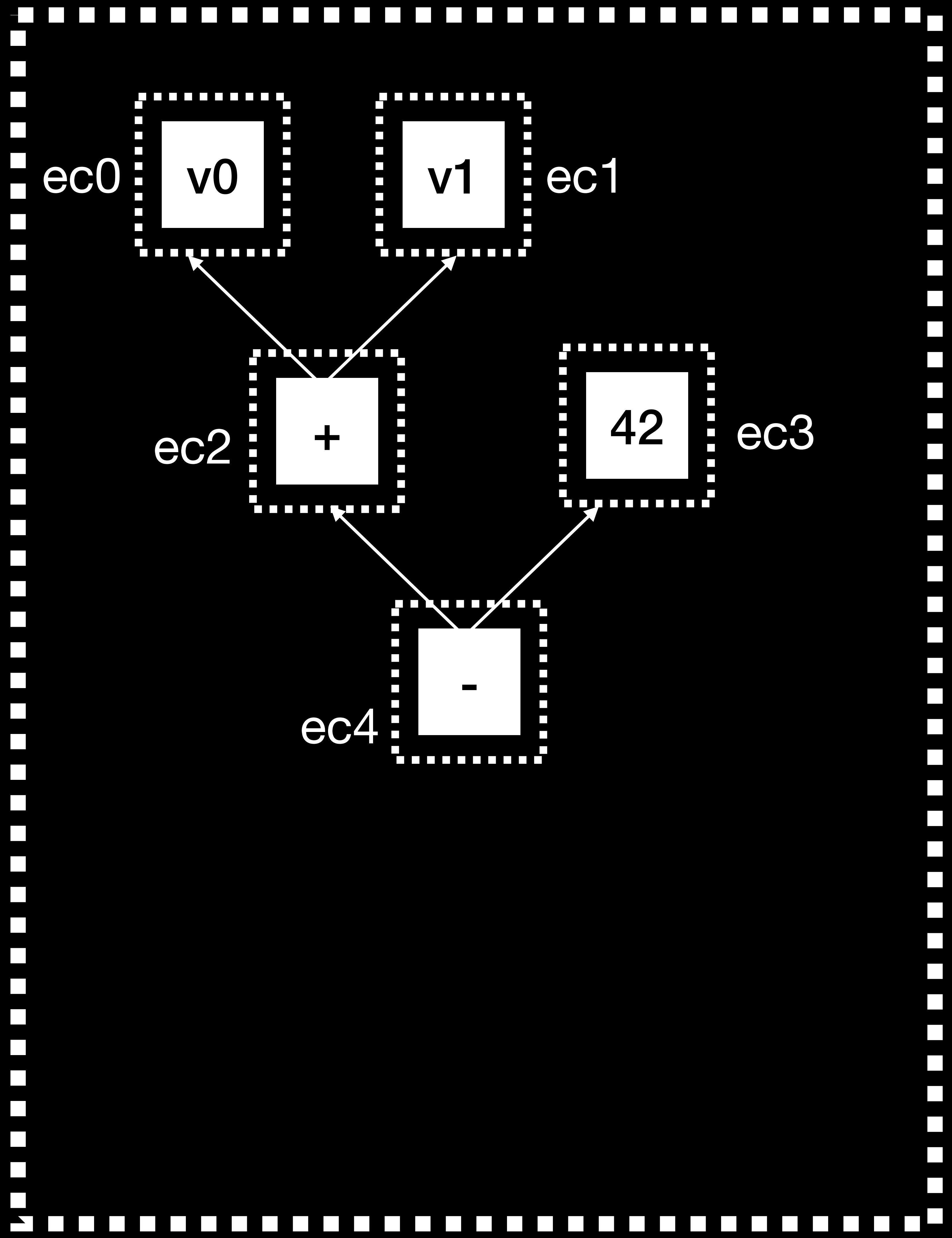
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec4	v2



Elaboration

```
block0(v0, v1):
  v3 = iadd v0, v1
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```

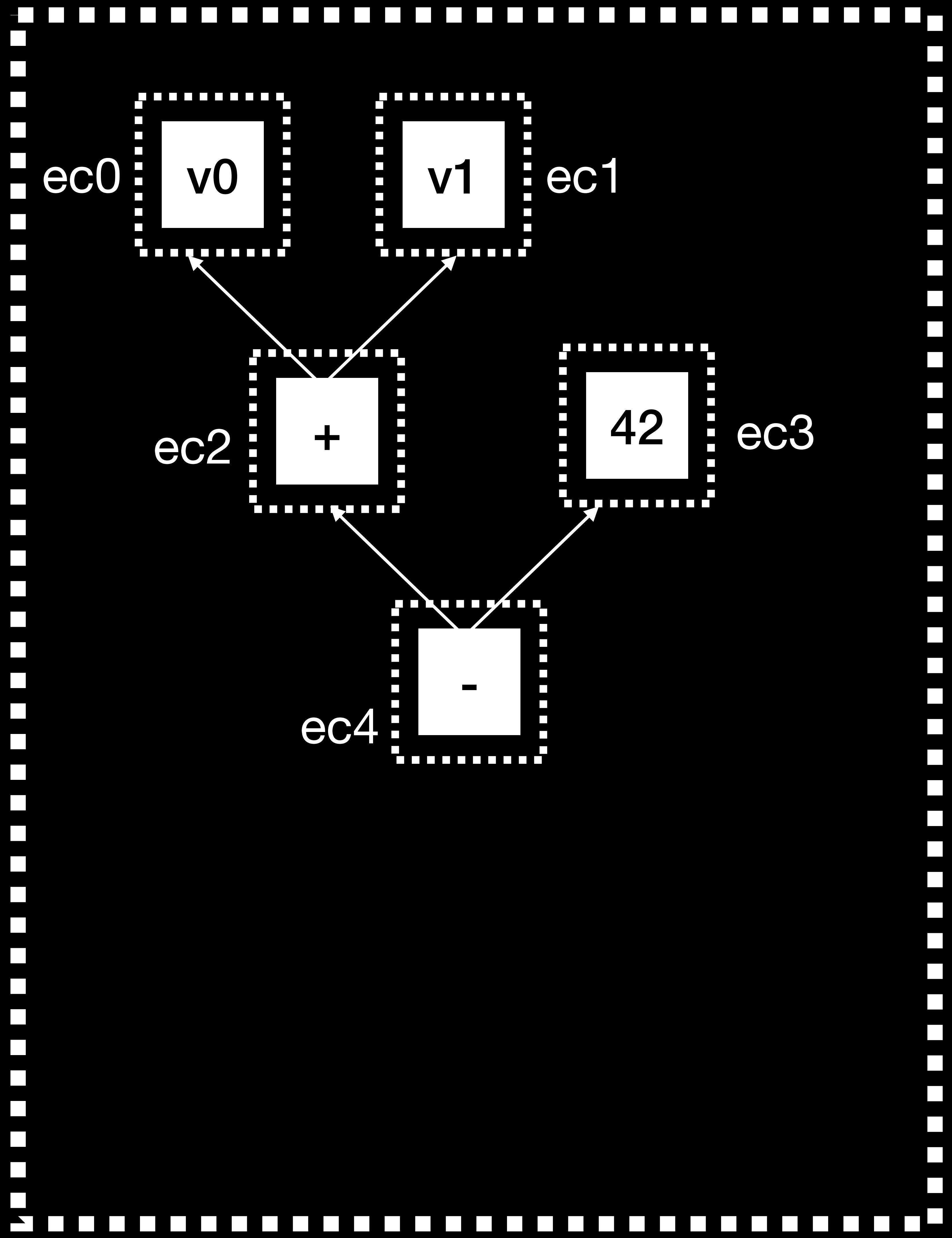
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec4	v2



Elaboration

```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub ec2, ec3
  store ec0, ec4
  return ec2
```

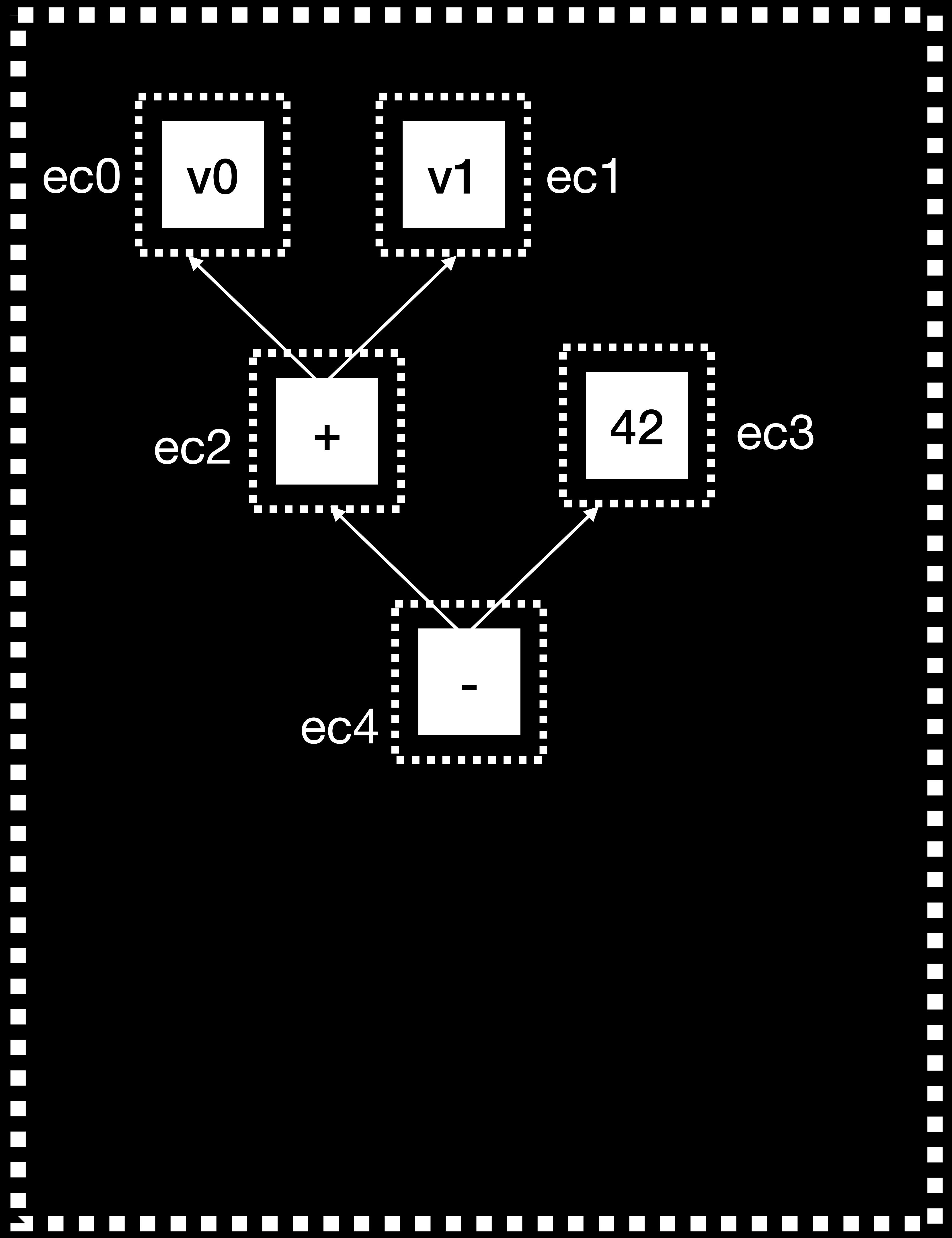
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2



Elaboration

```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store ec0, ec4
  return ec2
```

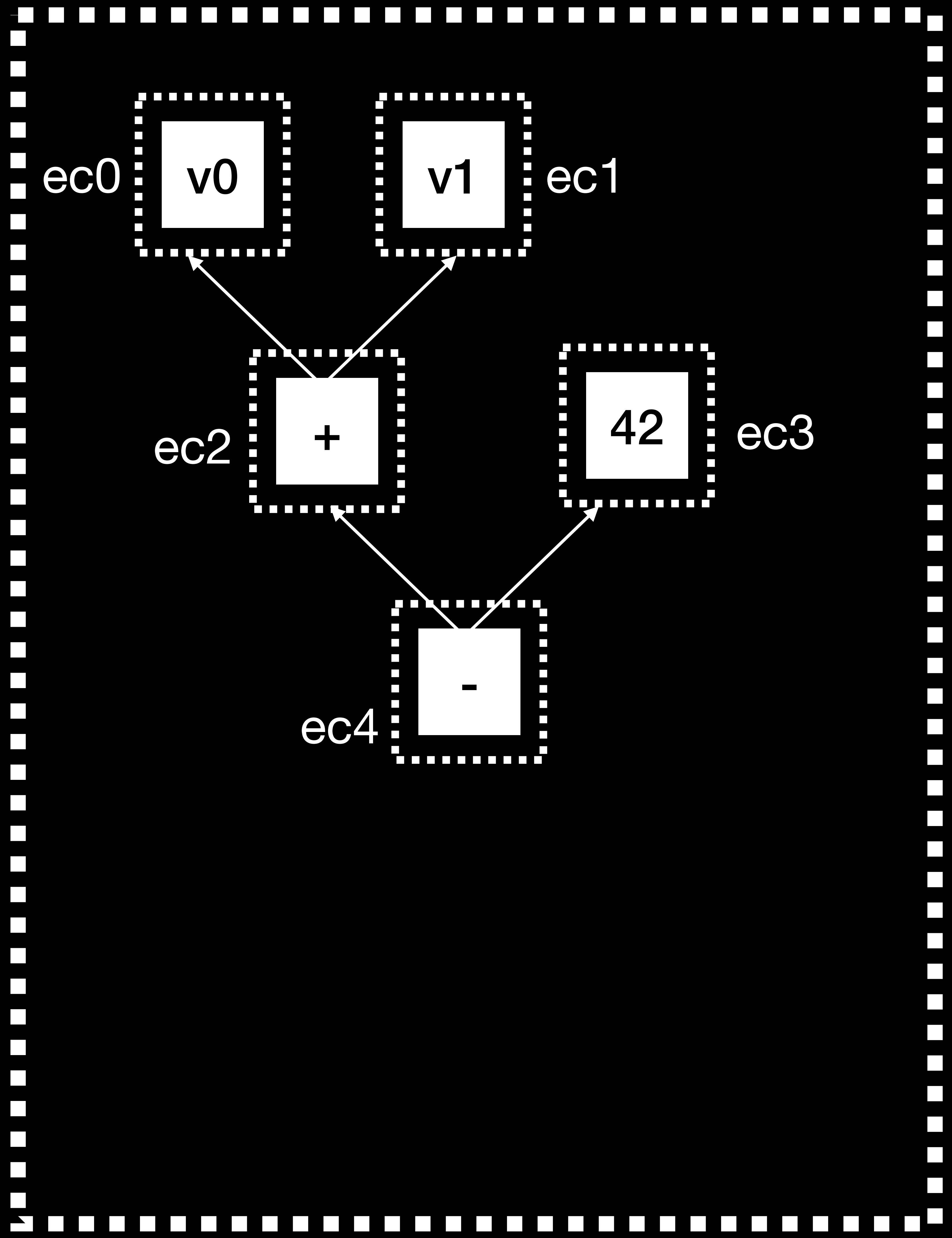
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2



Elaboration

```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store v0, v2
  return ec2
```

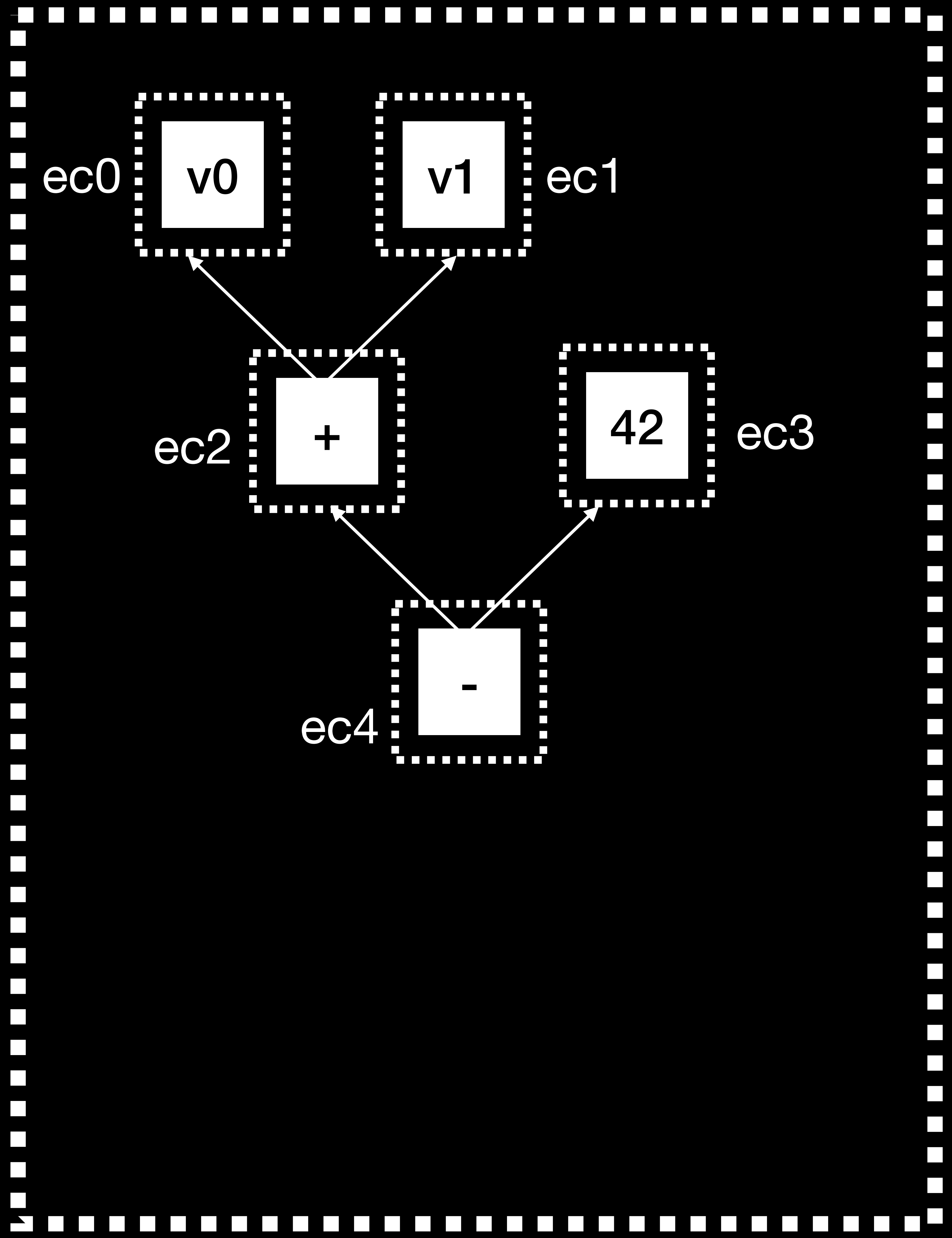
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2



Elaboration

```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store v0, v2
  return ec2
```

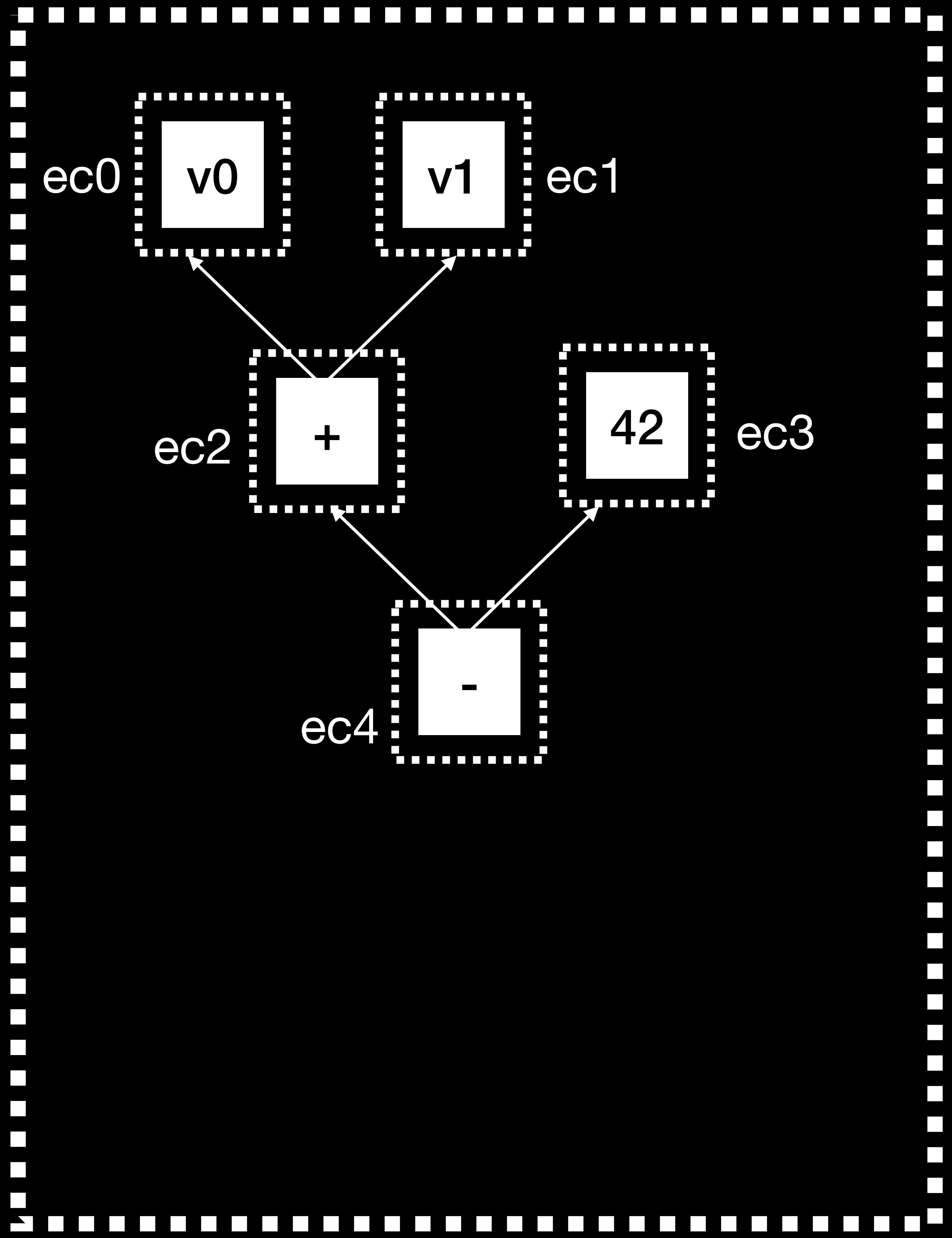
eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2



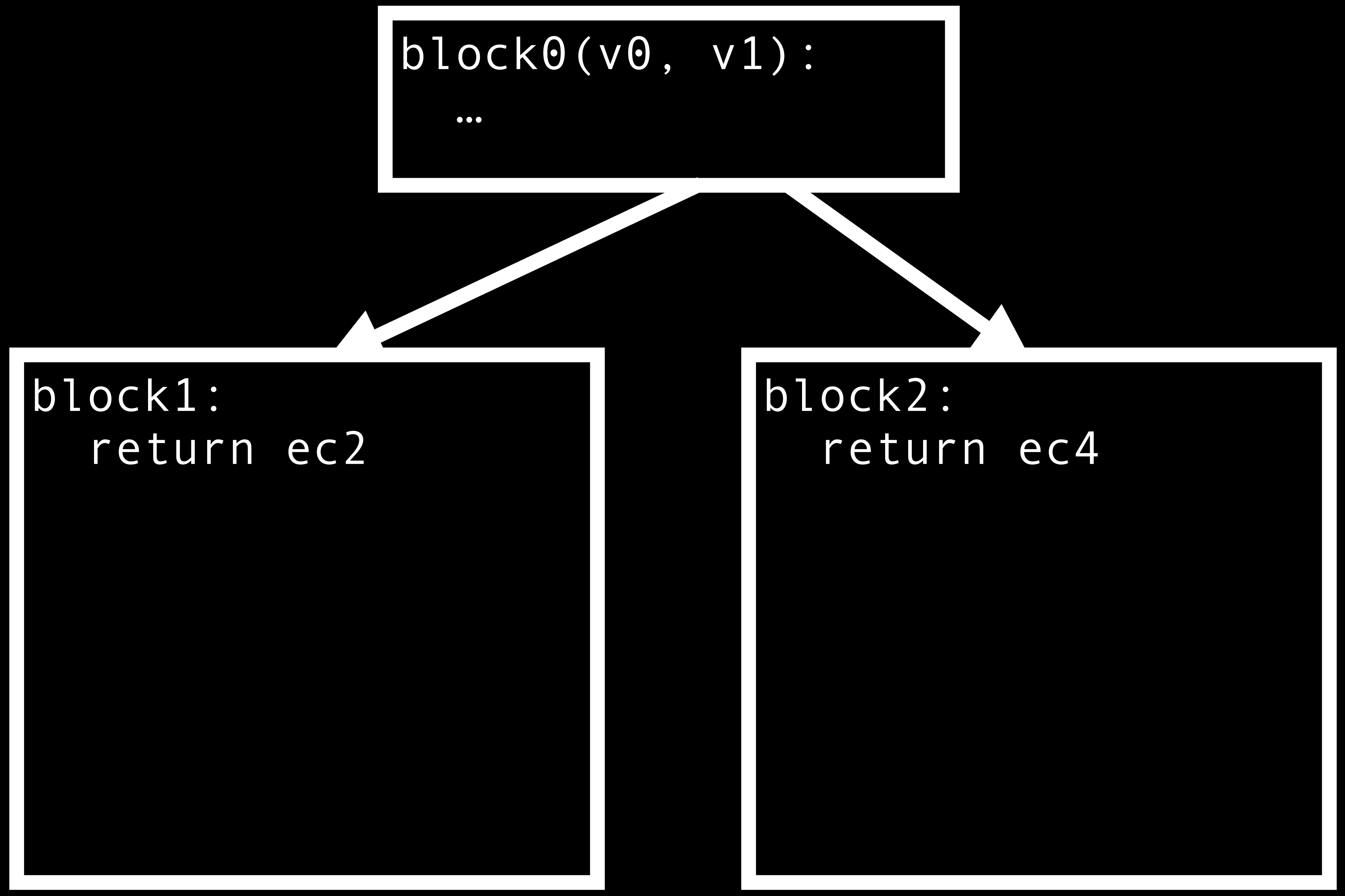
Elaboration

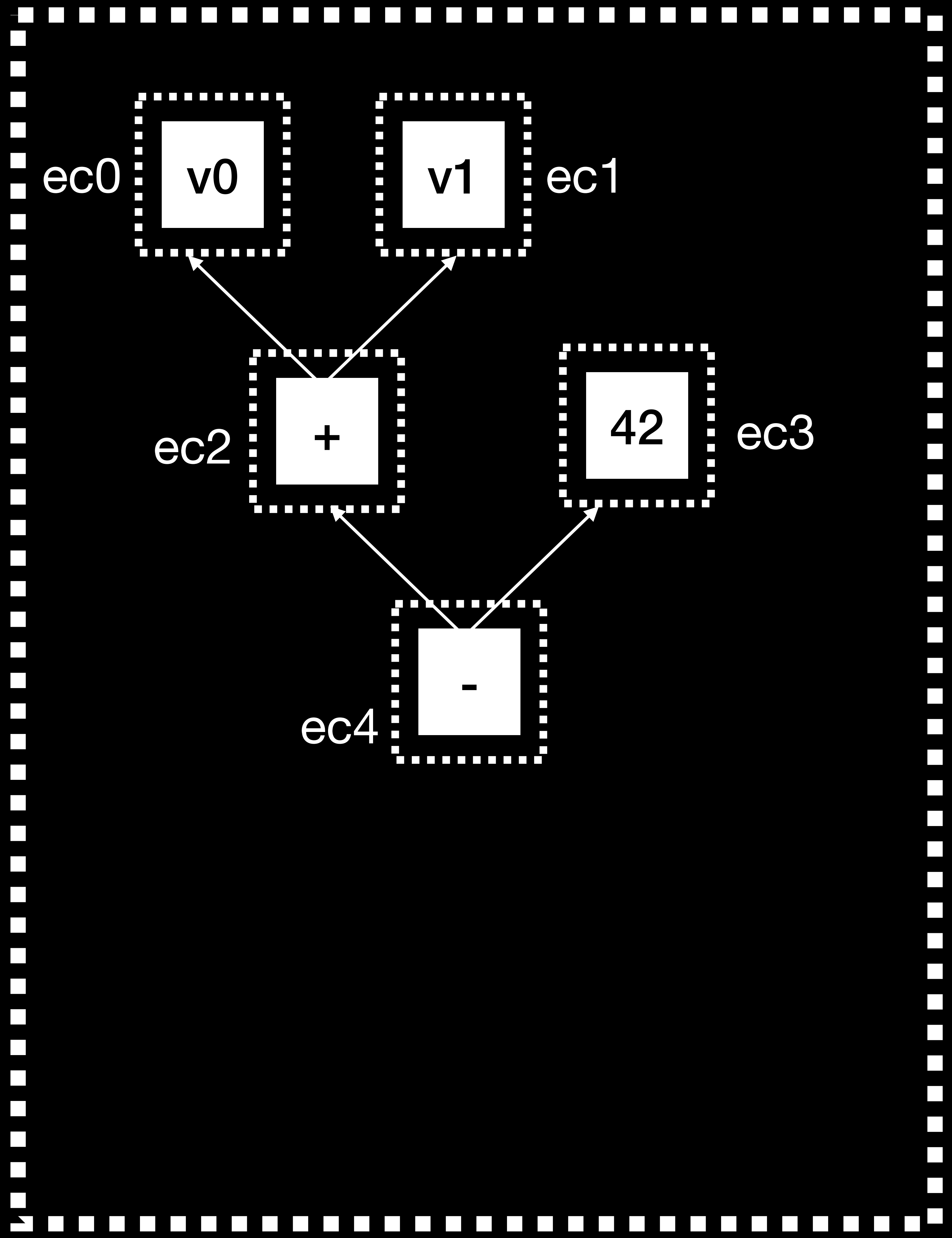
```
block0(v0, v1):
  v3 = iadd v0, v1
  v4 = iconst 42
  v2 = isub v3, v4
  store v0, v2
  return v3
```

eclass	elaborated
ec0	v0
ec1	v1
ec2	v3
ec3	v4
ec4	v2

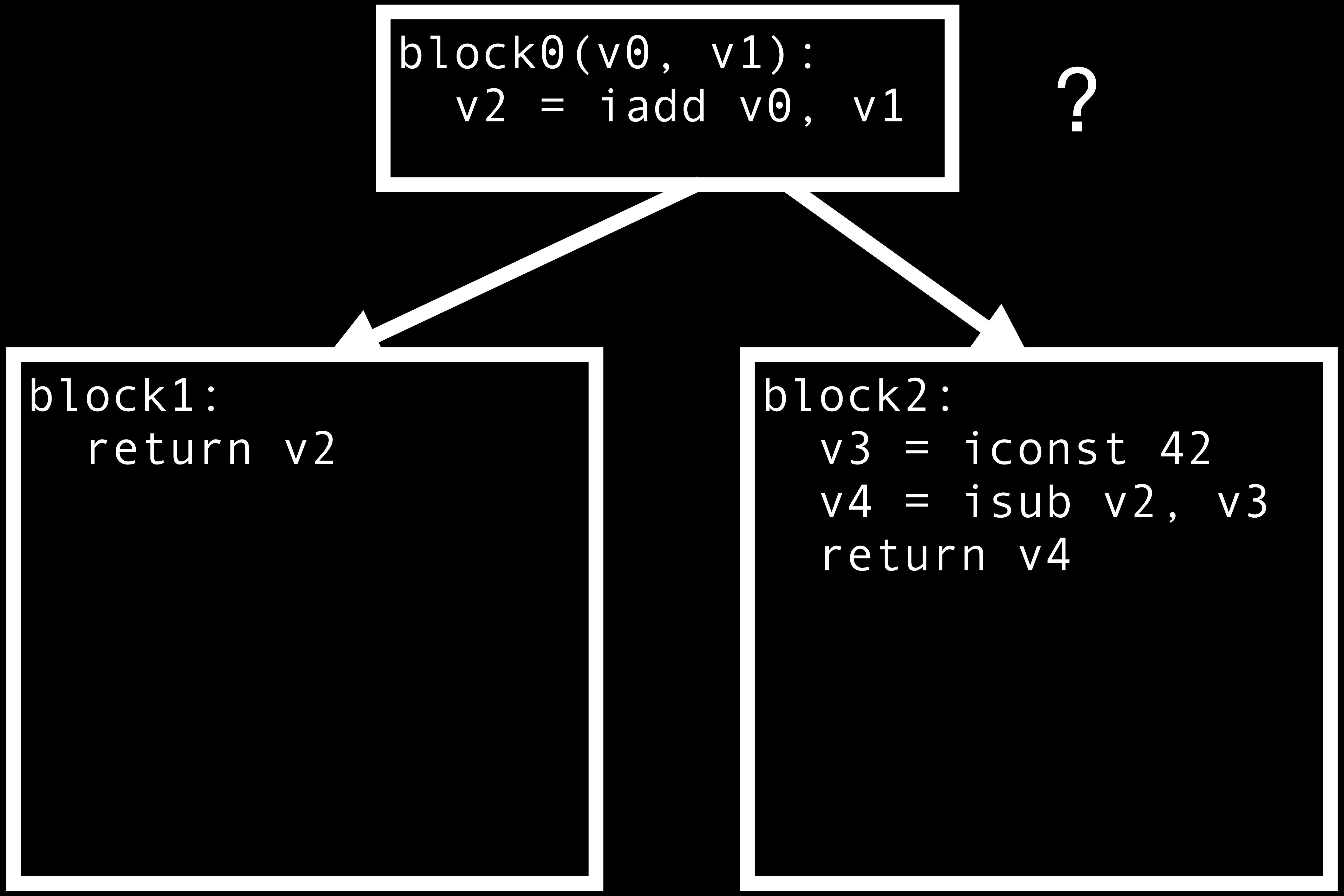


Elaboration… twice?

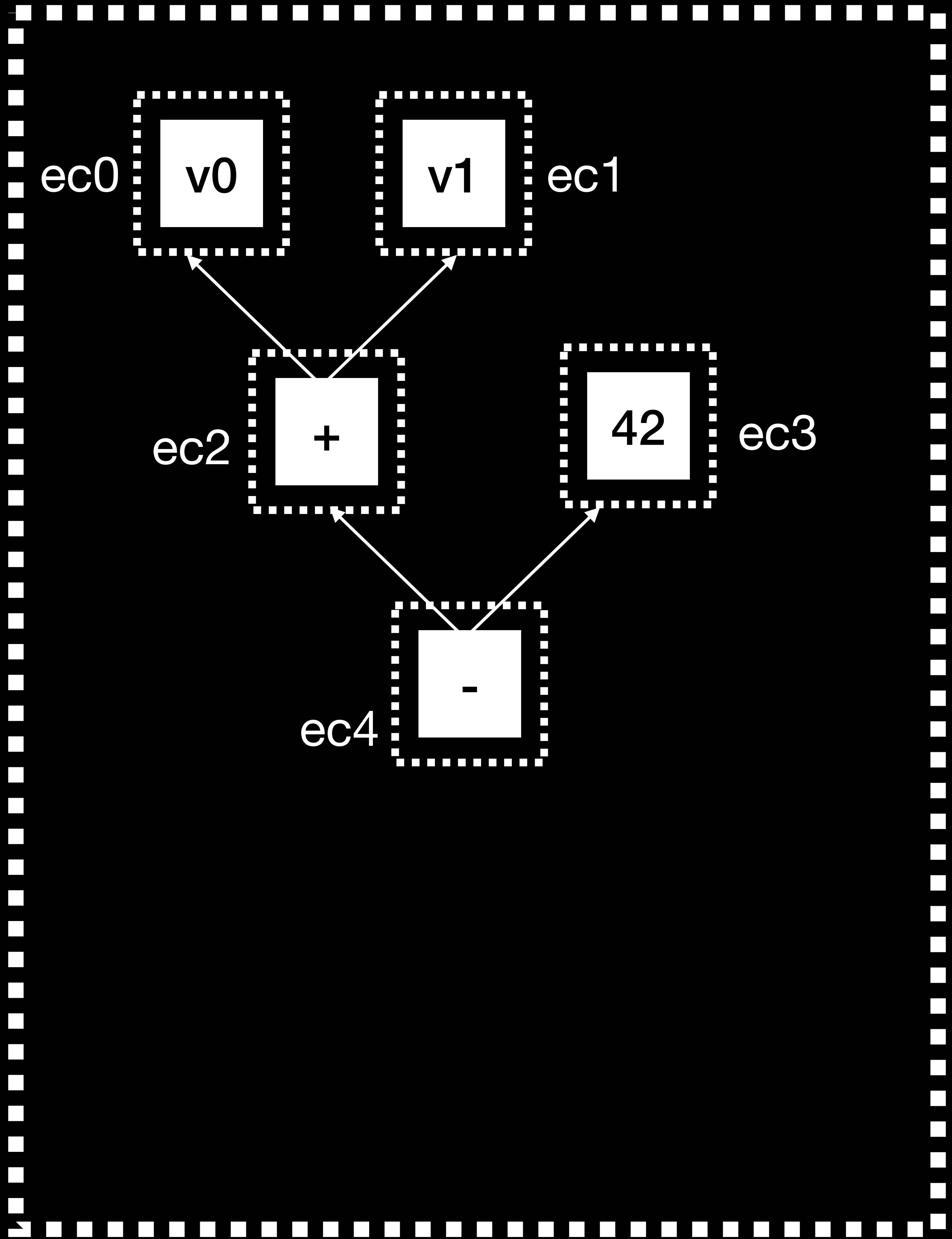




Elaboration… twice?



?



Elaboration… twice?

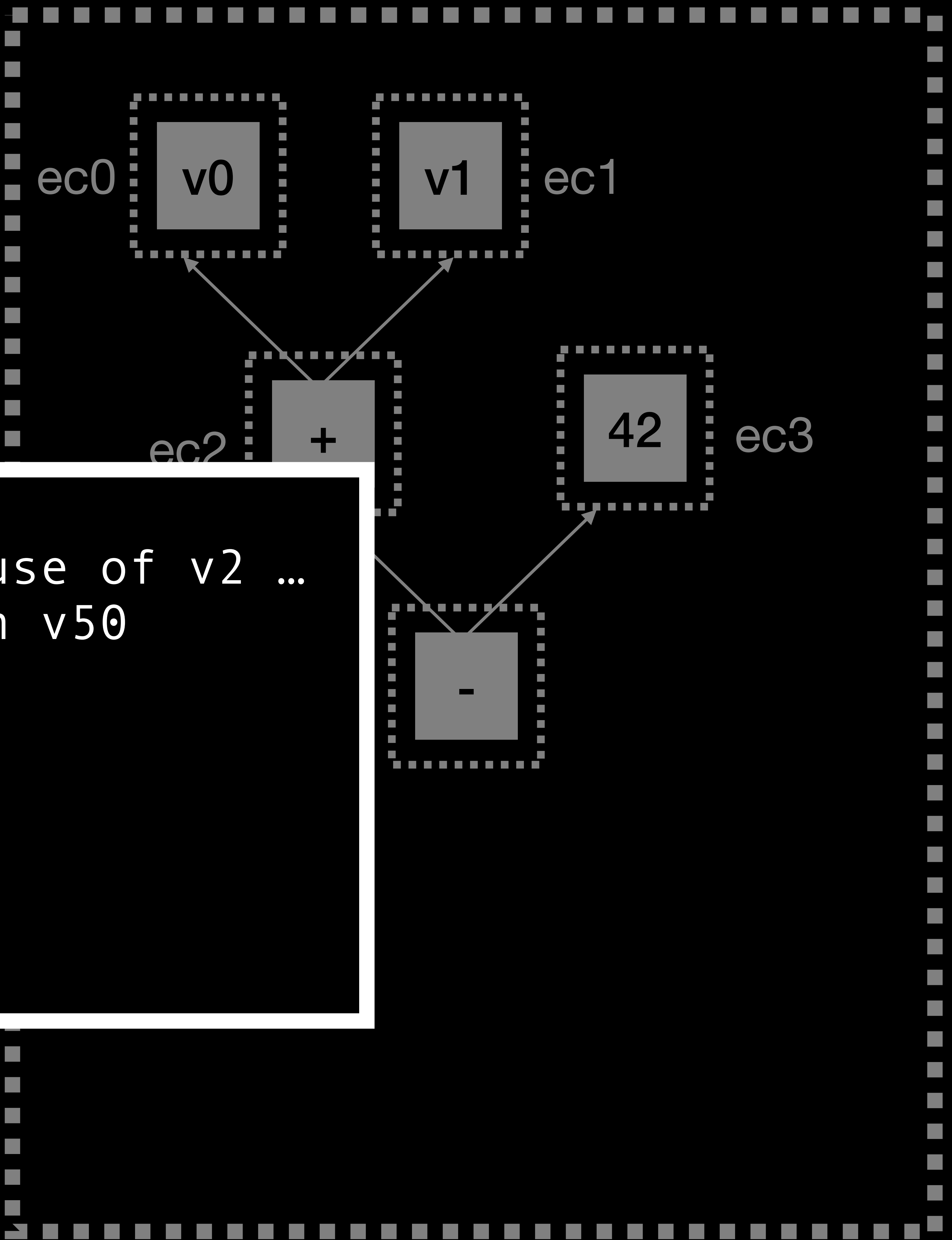
```
block0(v0, v1):
  v2 = iadd v0, v1
```

partial redundancy!

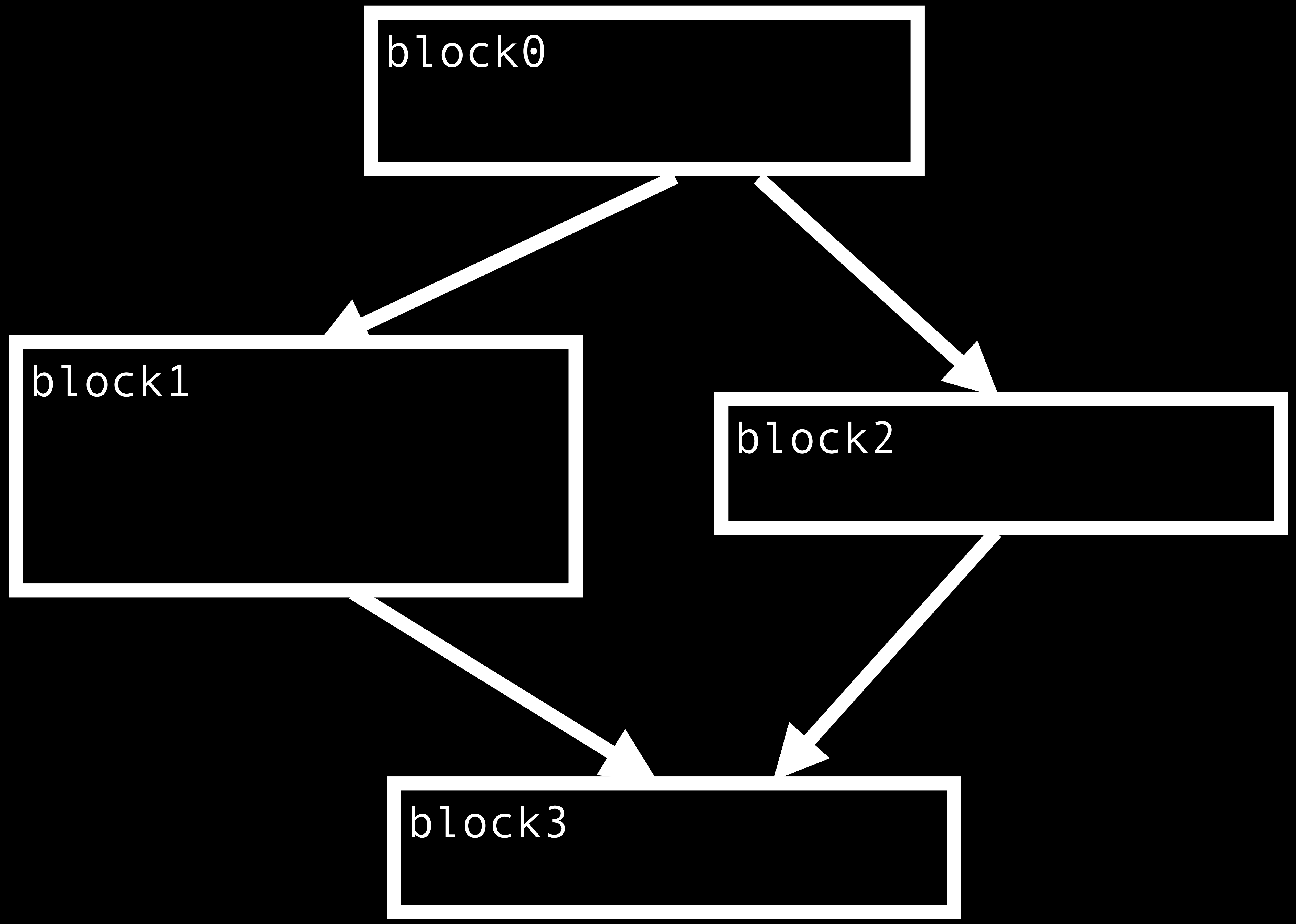
```
block1:
  return v2
```

```
block2:
  v3 = iconst 42
  v4 = isub v2, v3
  return v4
```

```
block2:
  … no use of v2 …
  return v50
```

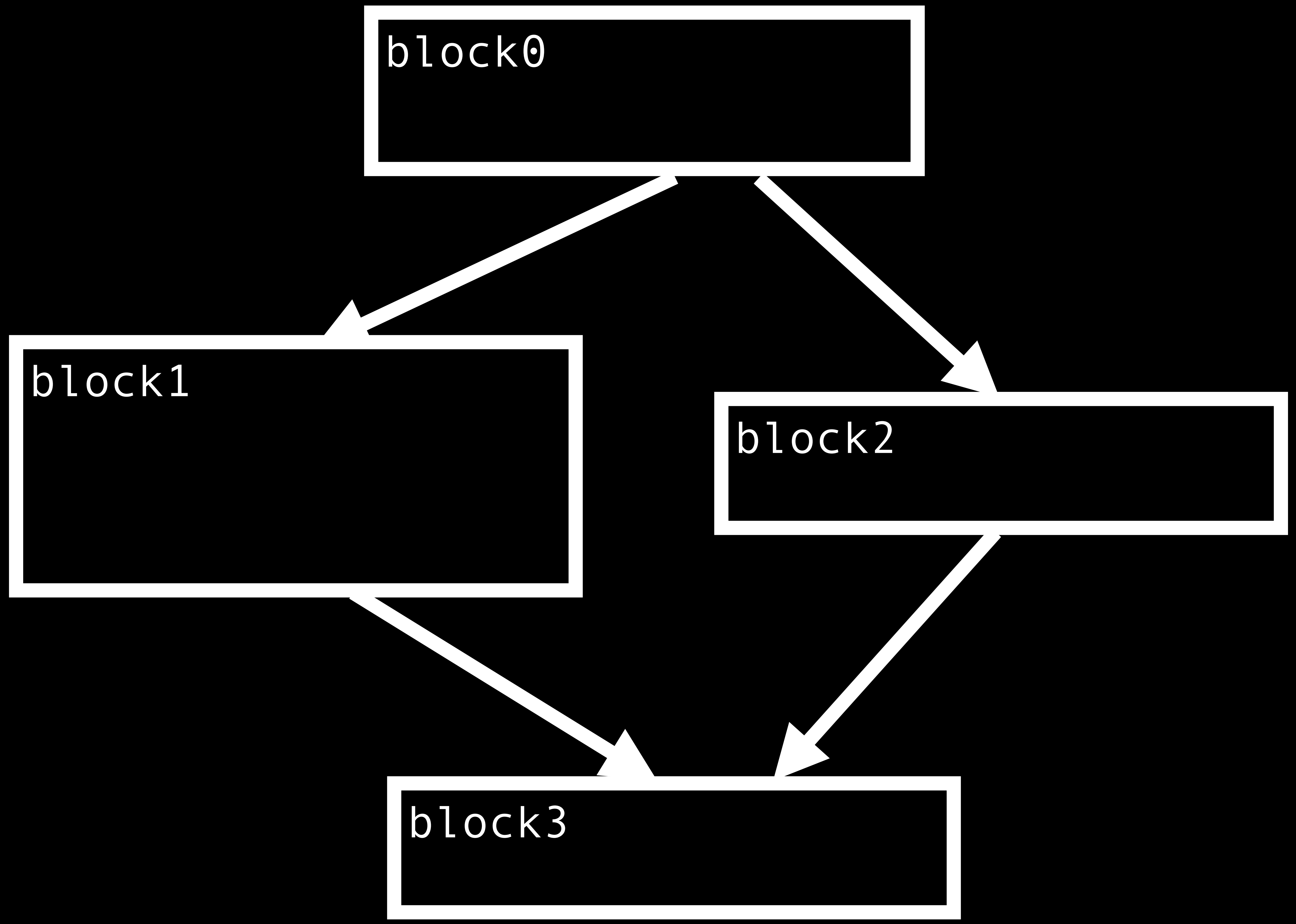


SSA

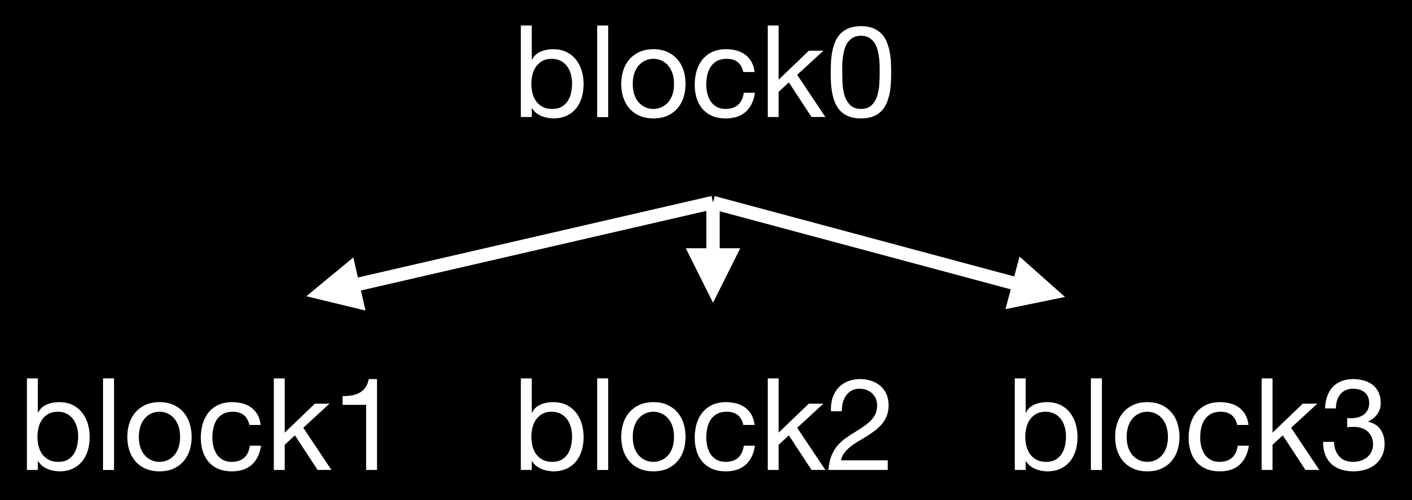


CFG

SSA

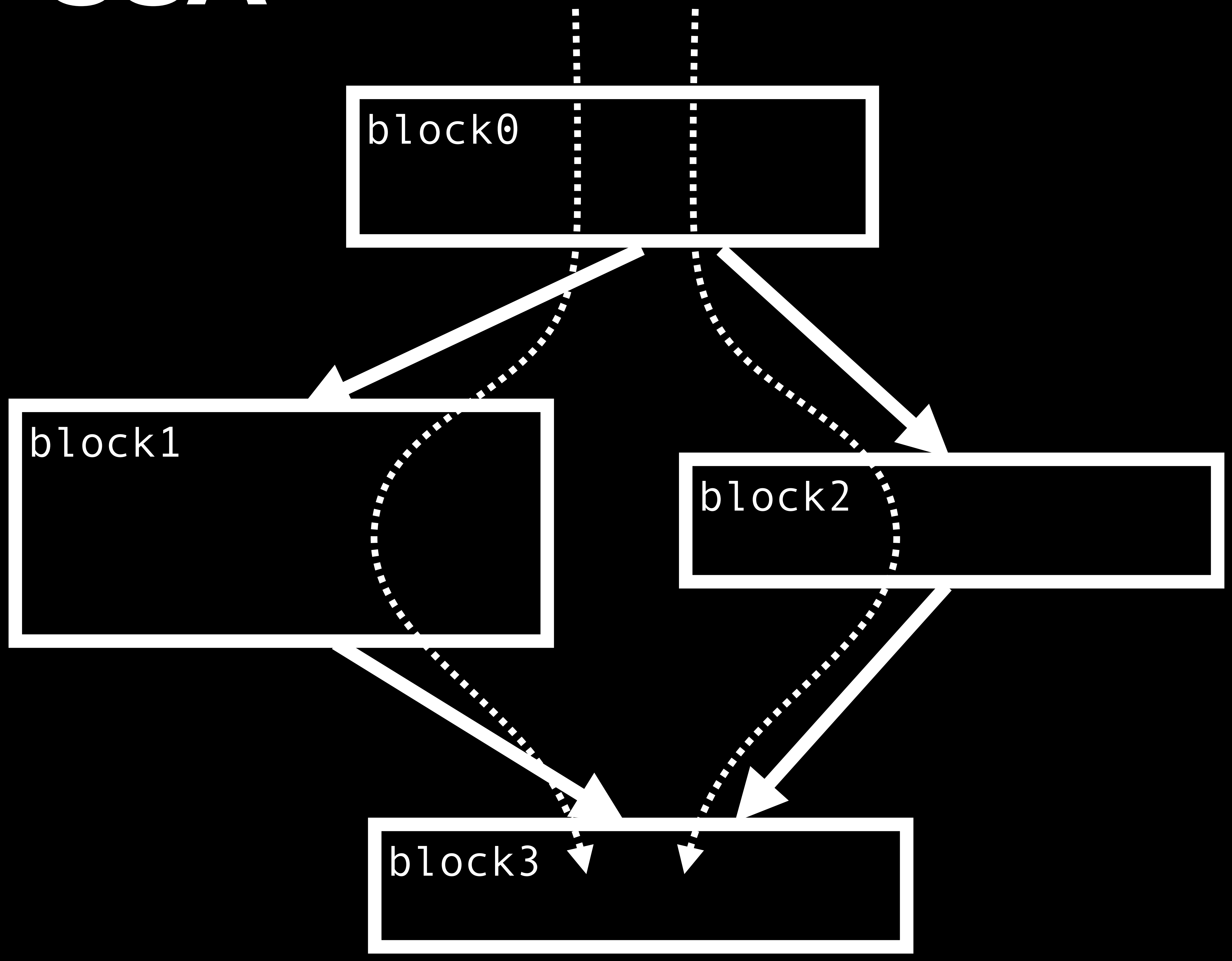


CFG

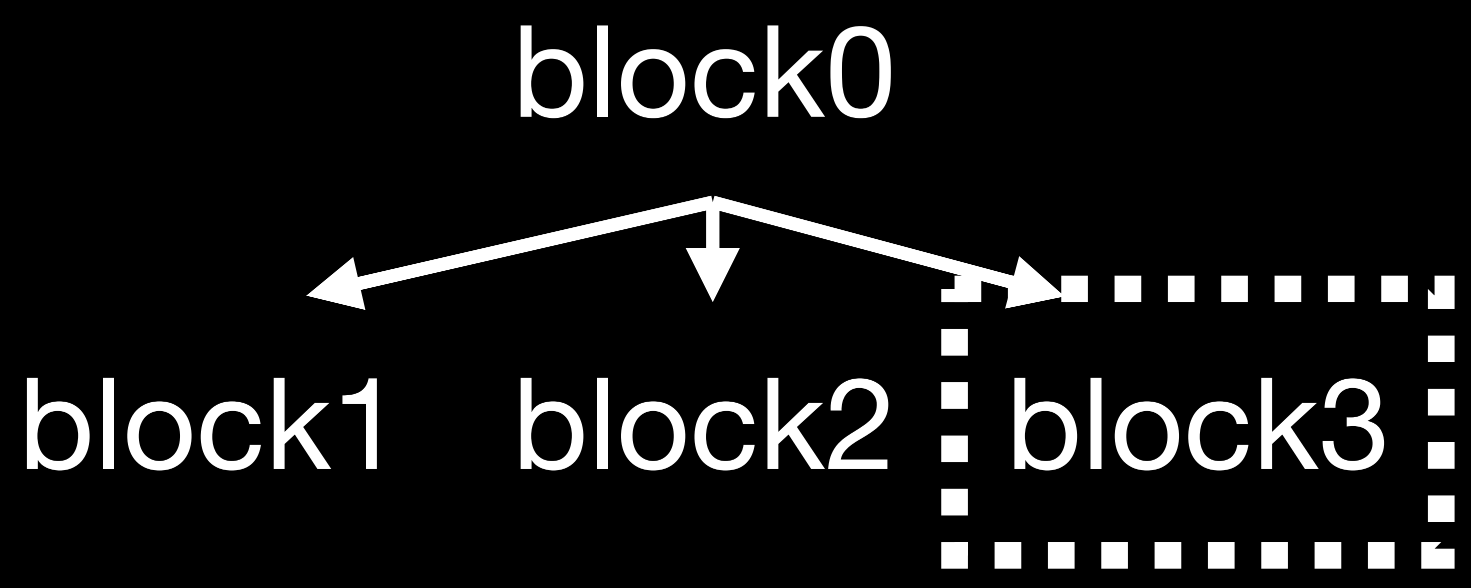


Dominator Tree

SSA



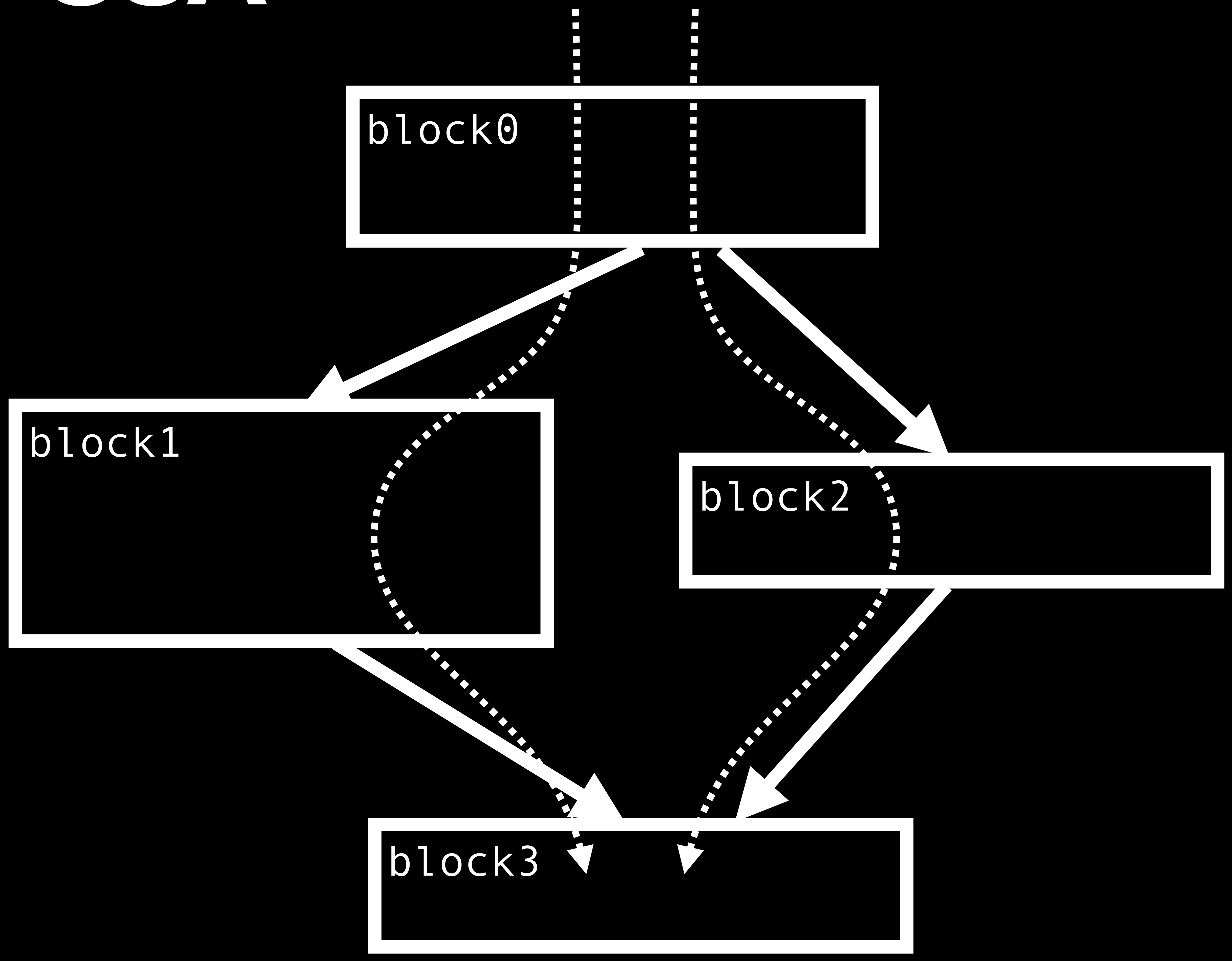
CFG



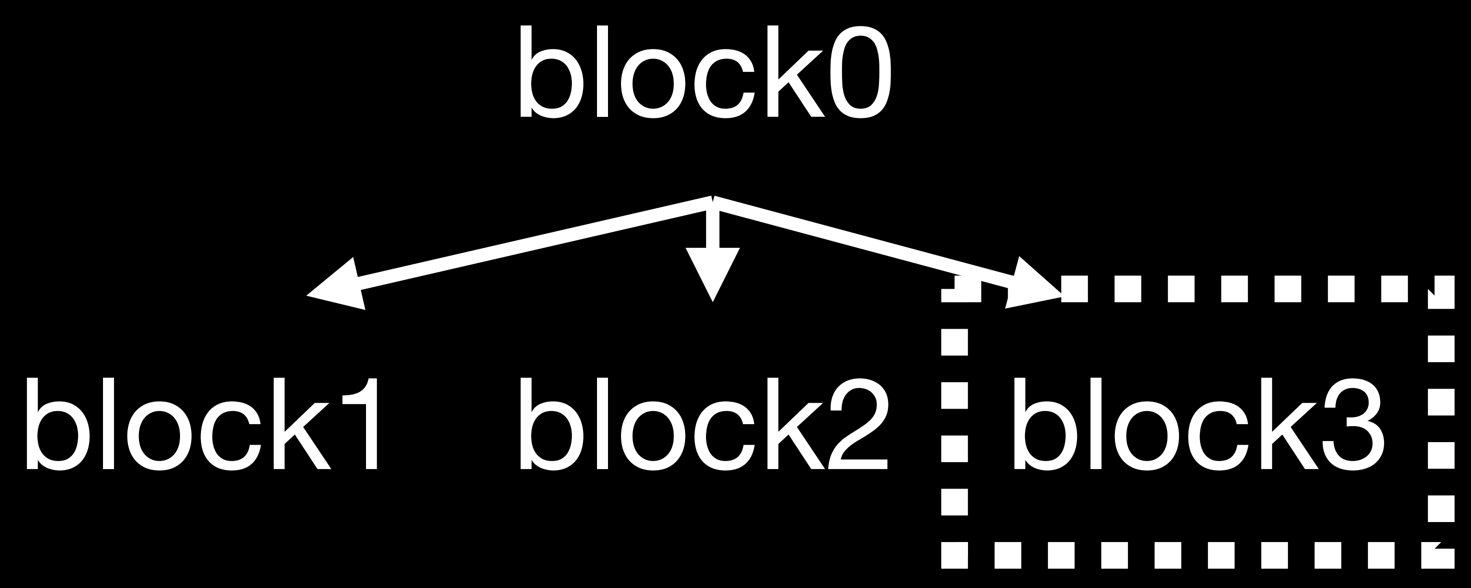
A dominates B if
all paths to B first pass through A.

Dominator Tree

SSA



CFG

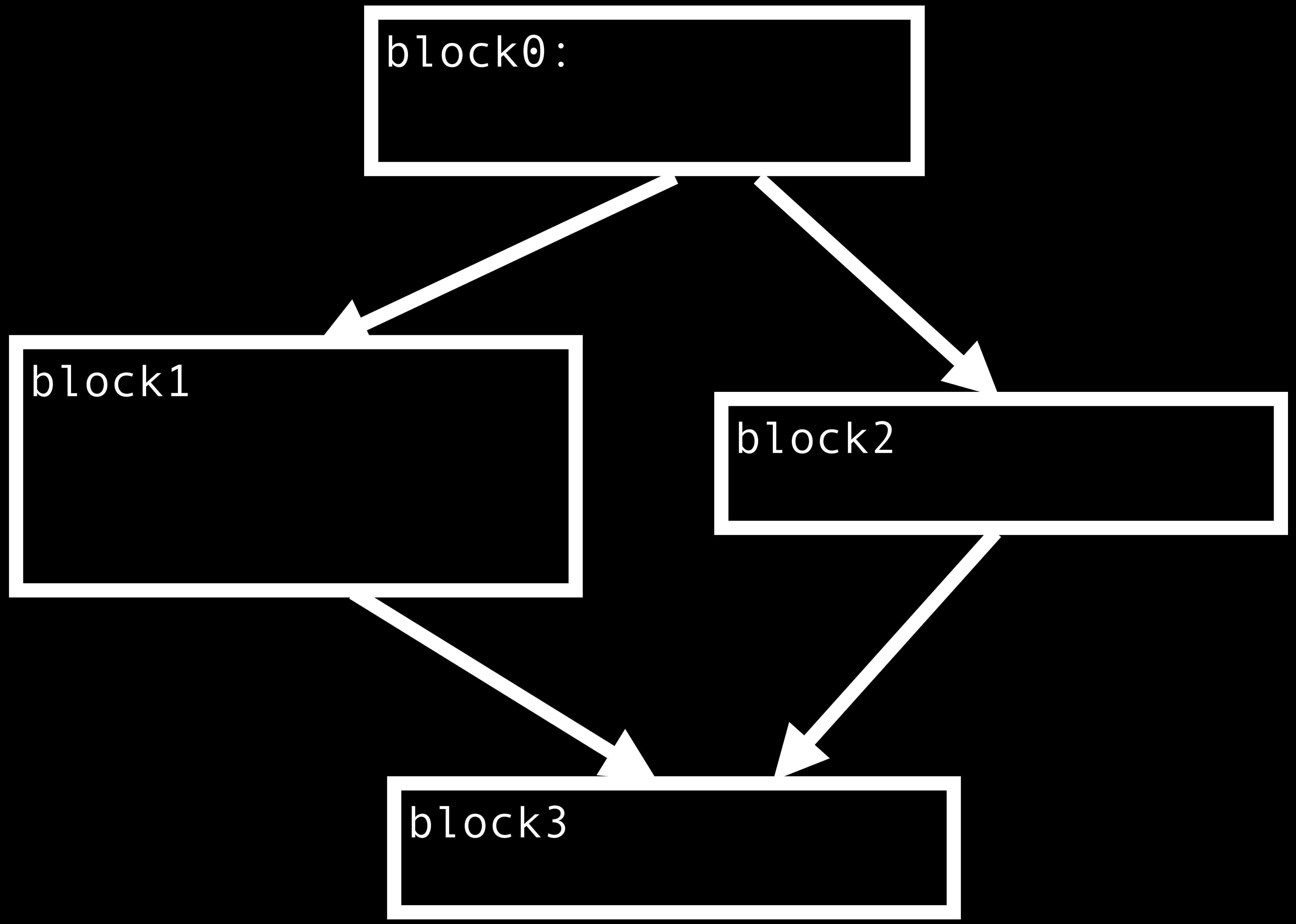


A dominates B if
all paths to B first pass through A.

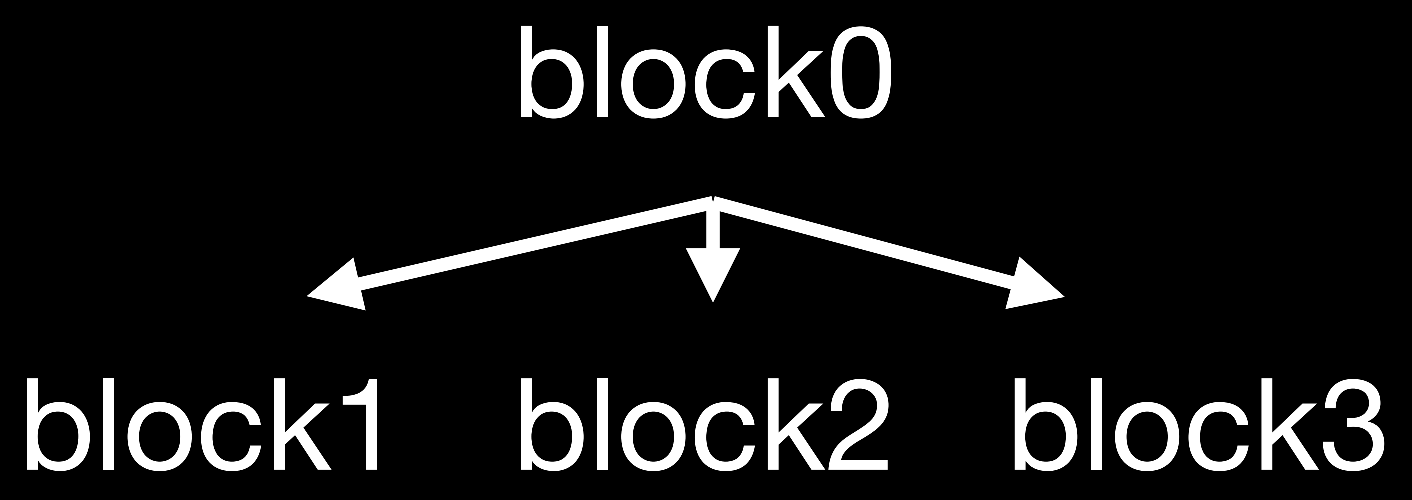
Dominance forms a *tree*.
Many compiler algorithms work
by traversing the domtree.

Dominator Tree

SSA

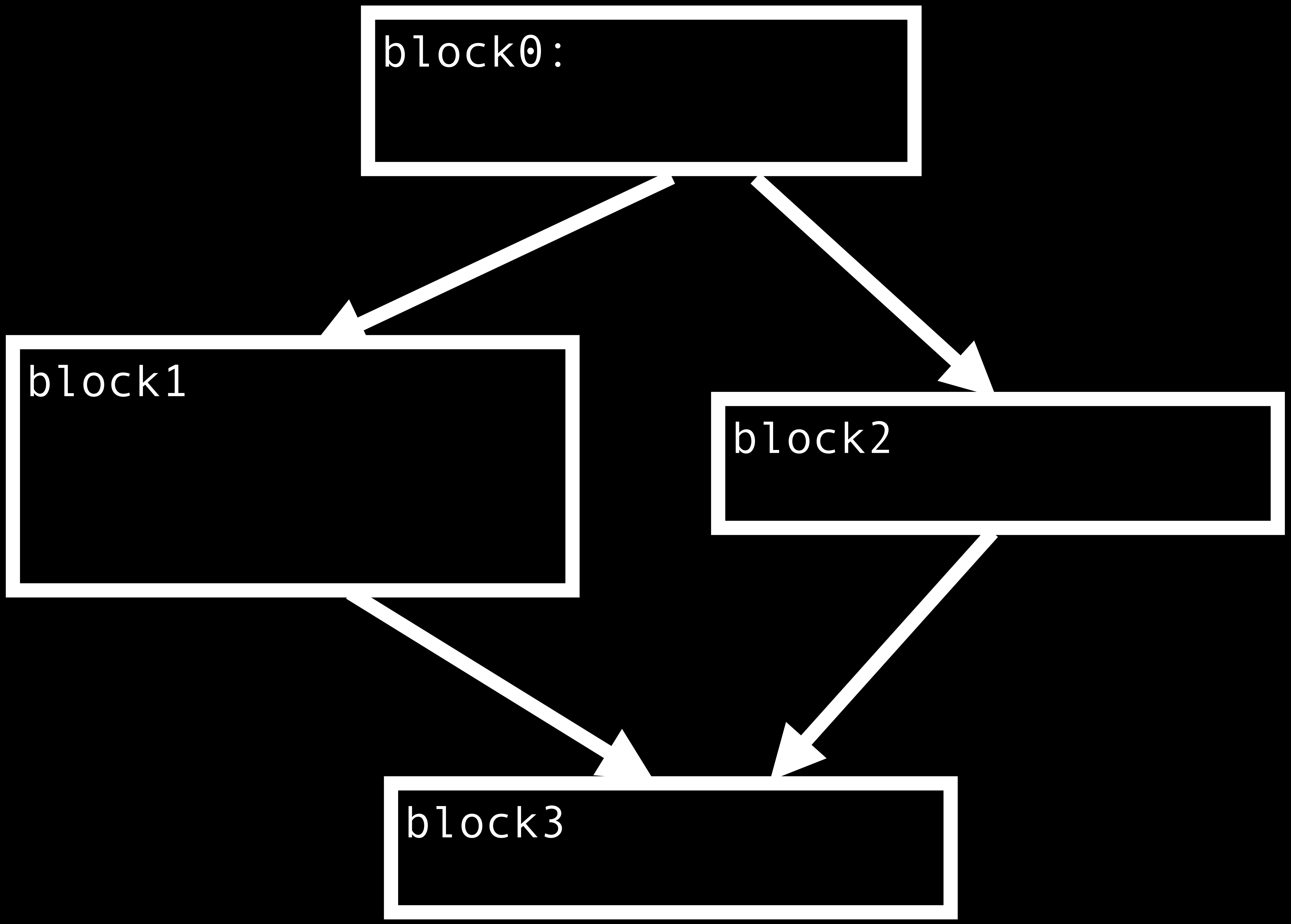


CFG

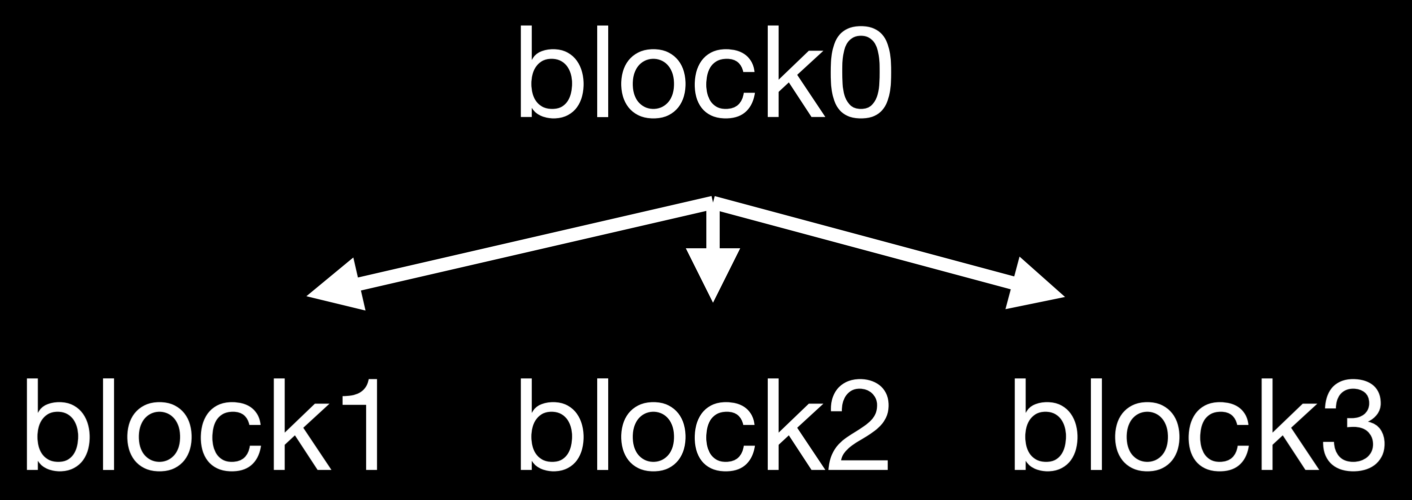


Dominator Tree

SSA



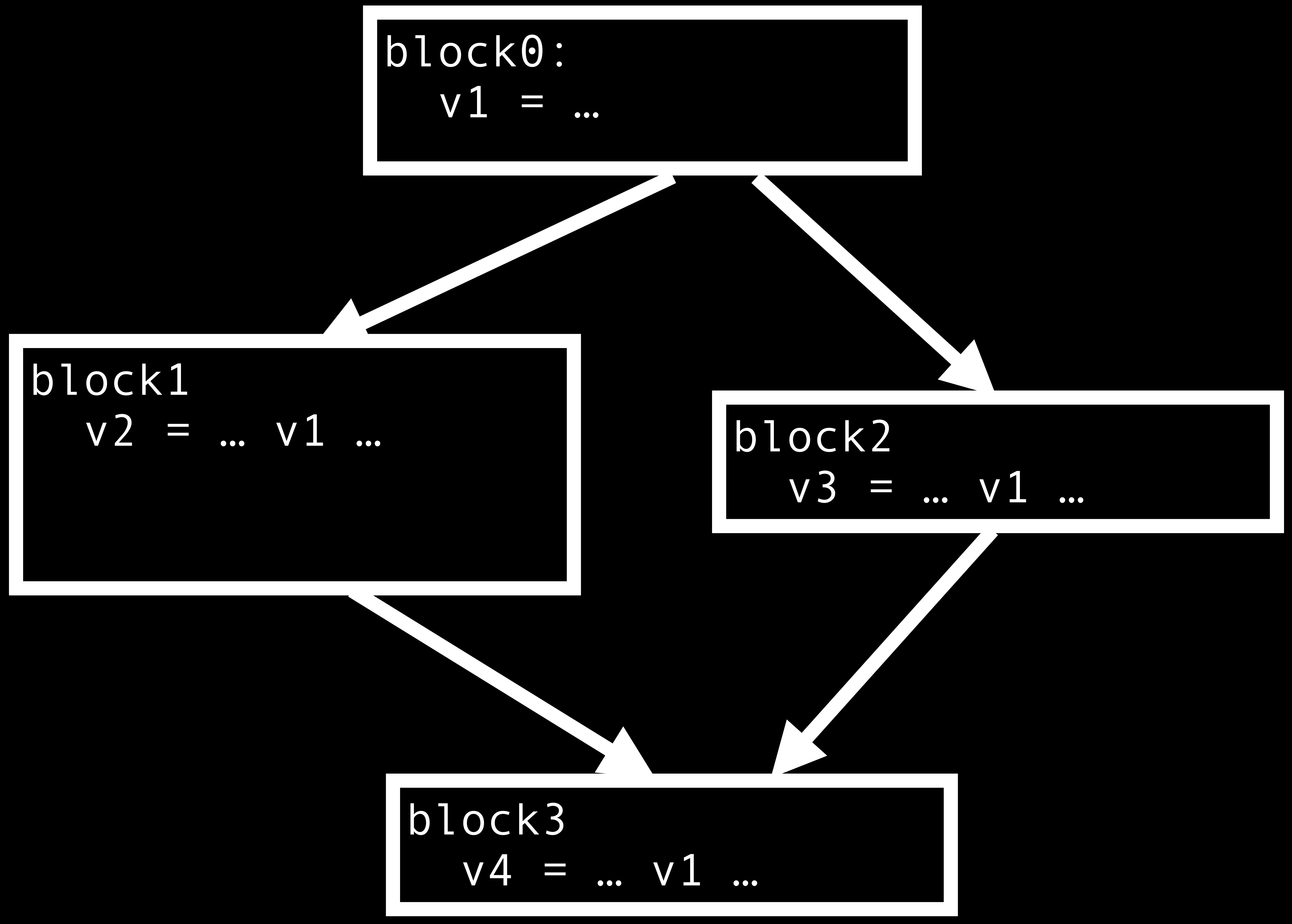
CFG



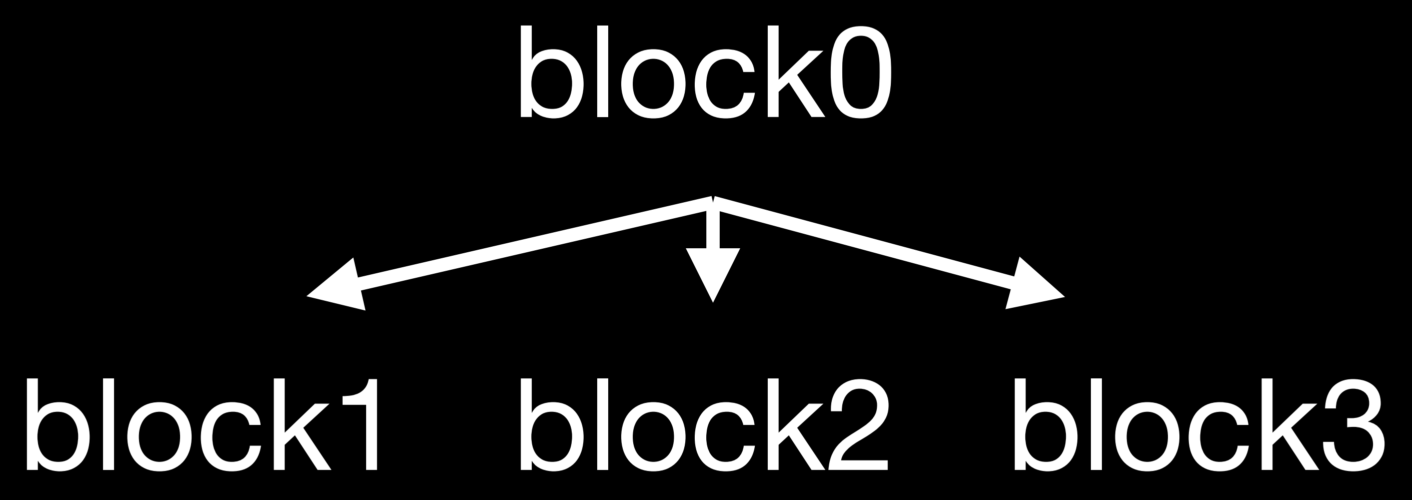
SSA: A value's definition dominates its uses.

Dominator Tree

SSA



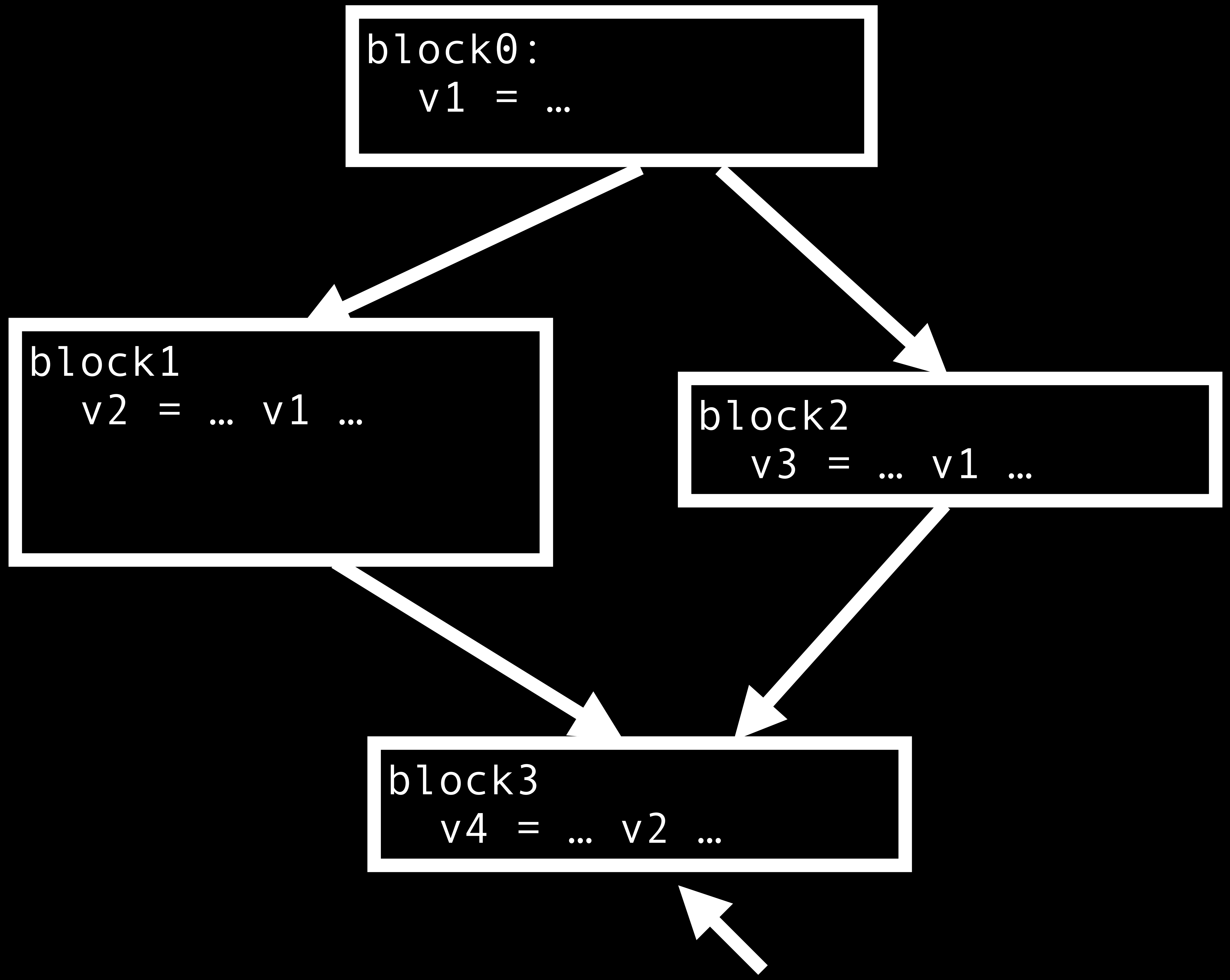
CFG



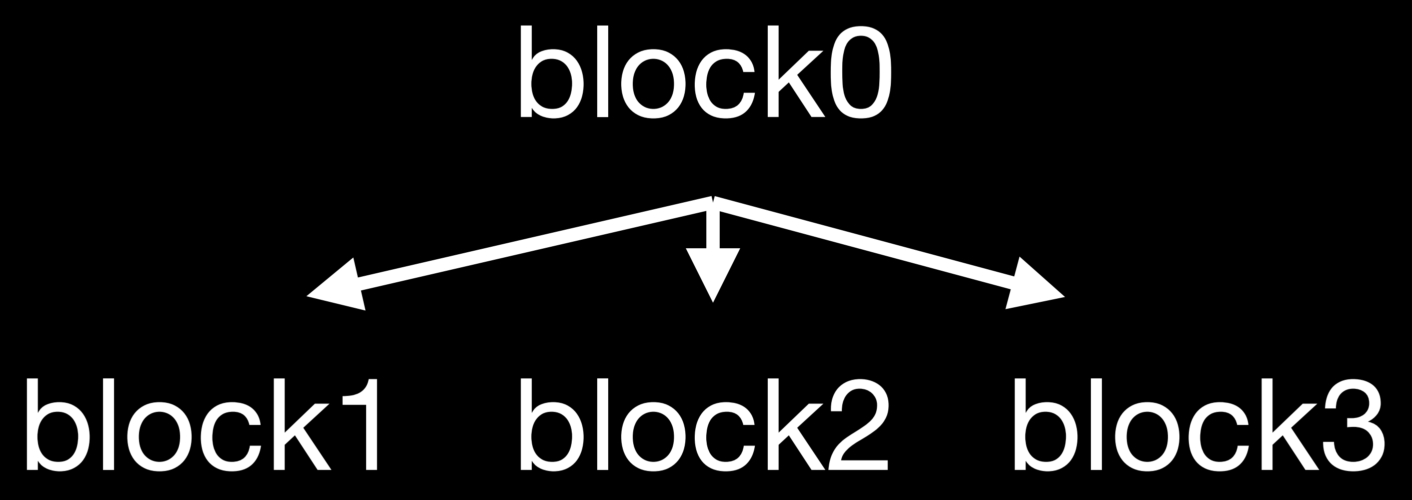
SSA: A value's definition dominates its uses.

Dominator Tree

SSA



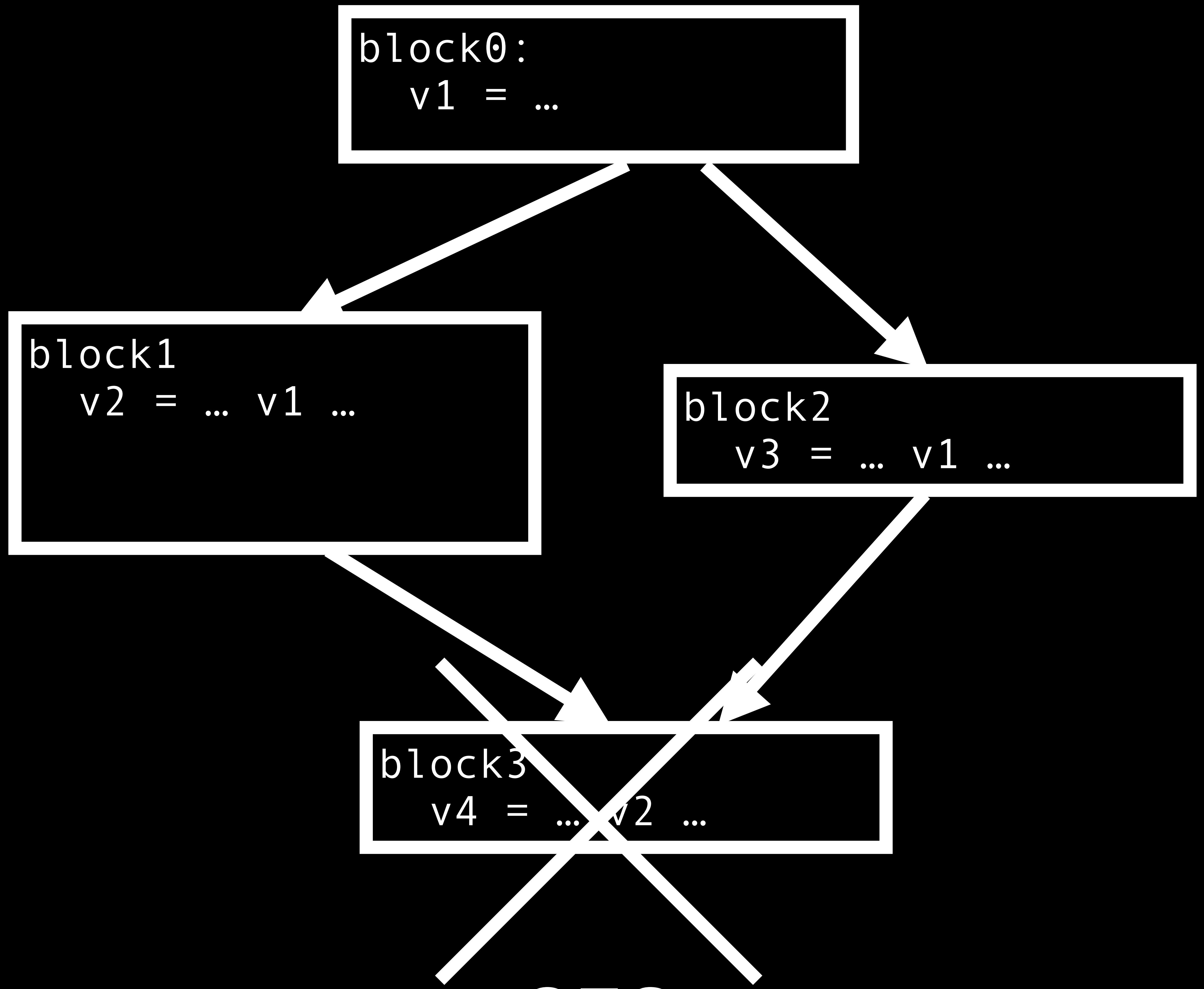
CFG



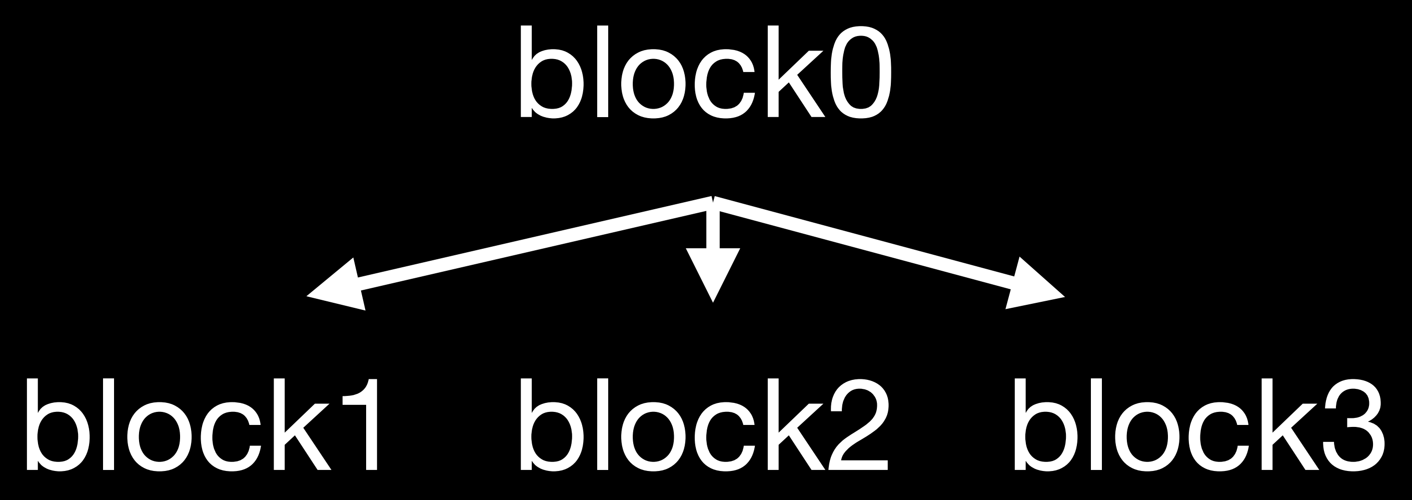
SSA: A value’s definition dominates its uses.

Dominator Tree

SSA



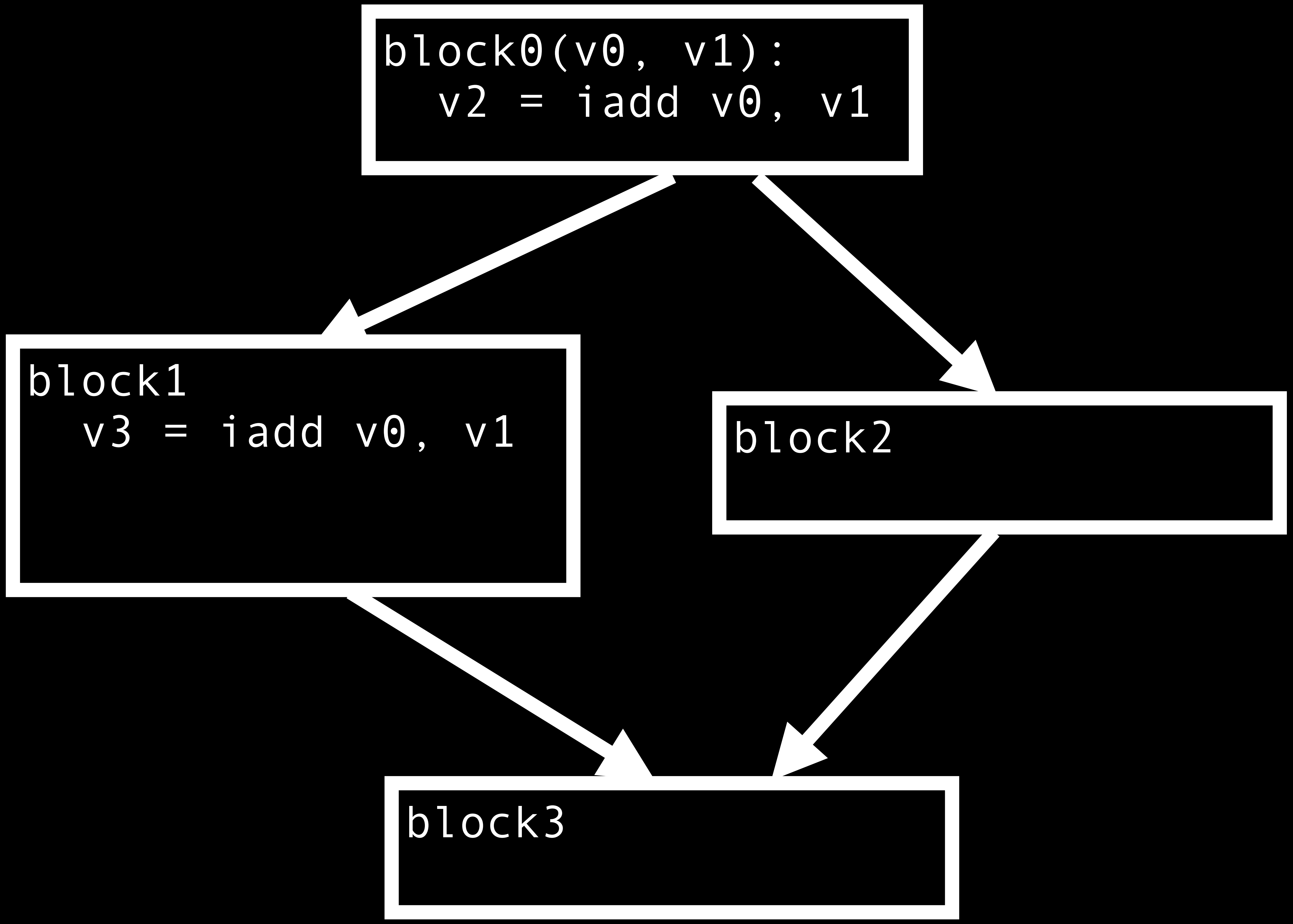
CFG



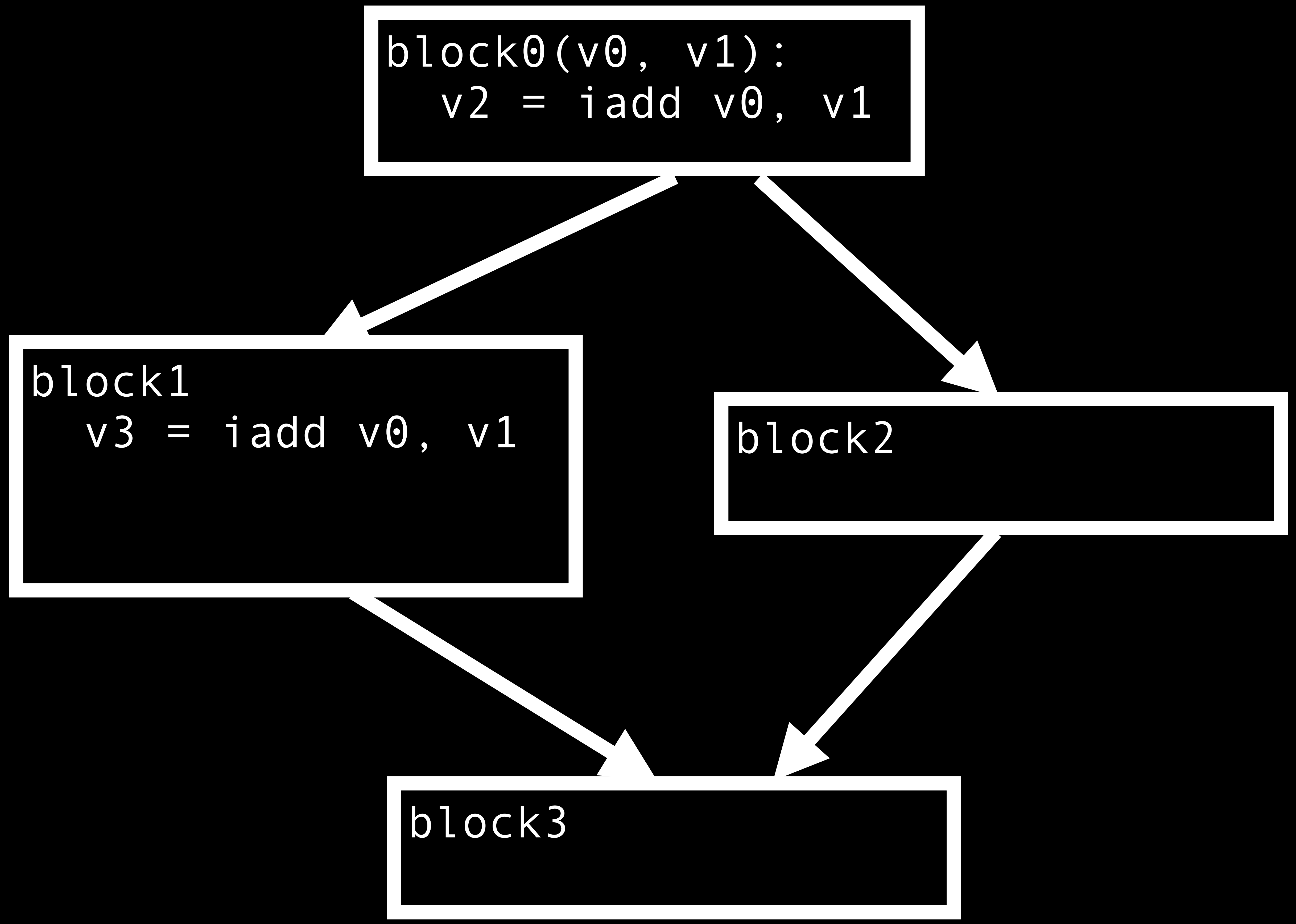
SSA: A value's definition dominates its uses.

Dominator Tree

GVN

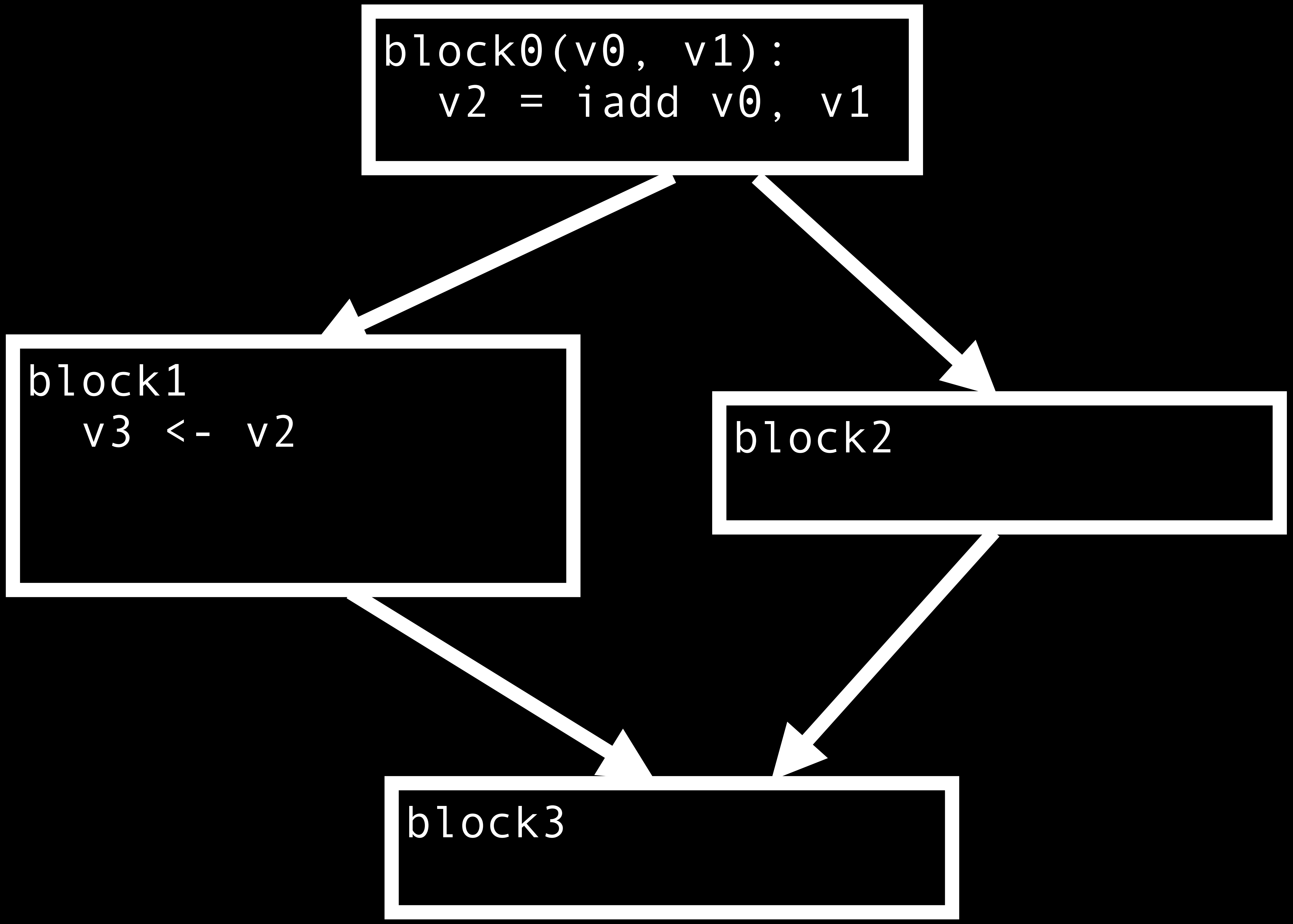


GVN



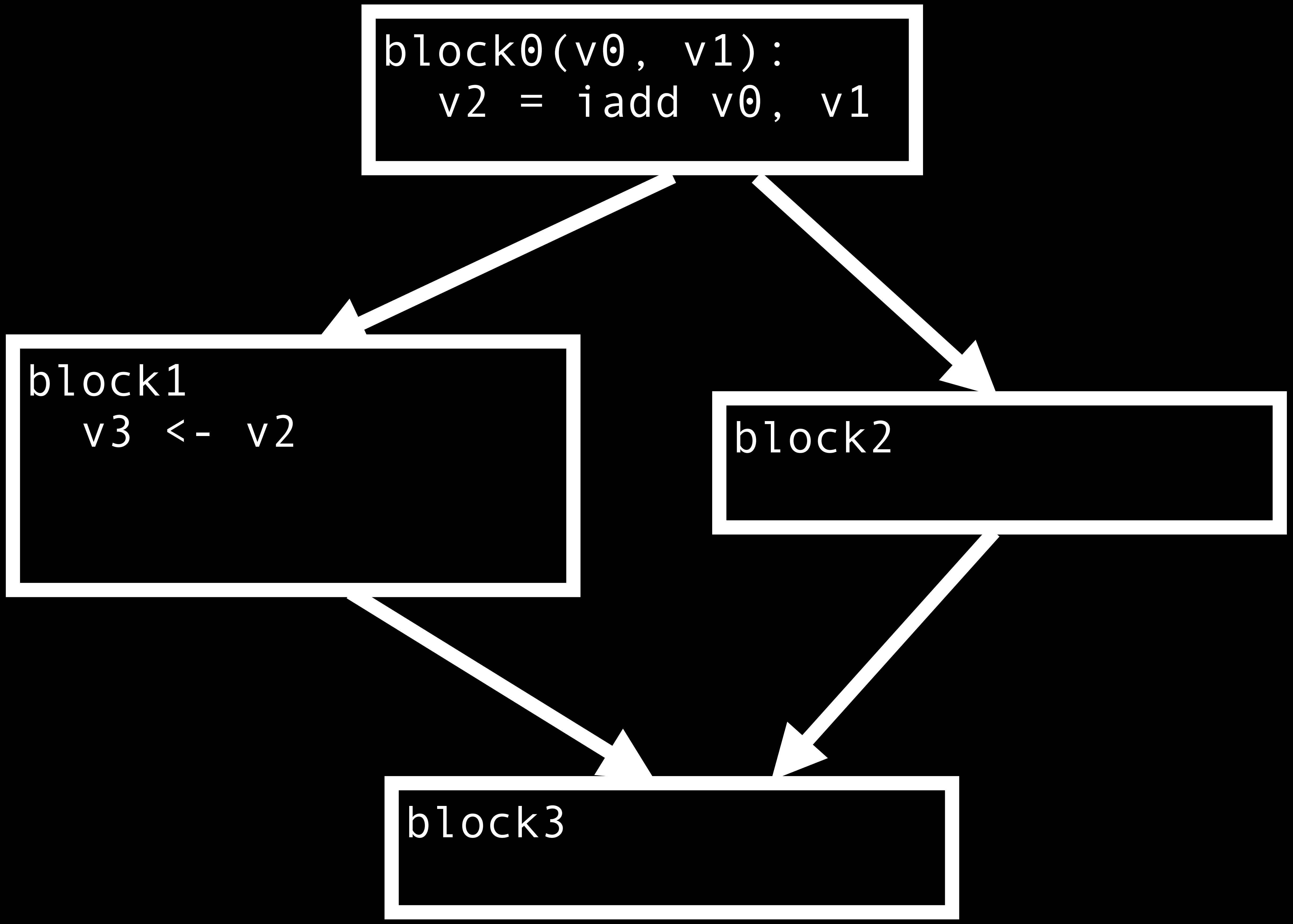
GVN (Global Value Numbering):
If an operator dominates a duplicate copy of itself, reuse the original.

GVN



GVN (Global Value Numbering):
If an operator dominates a duplicate copy of itself, reuse the original.

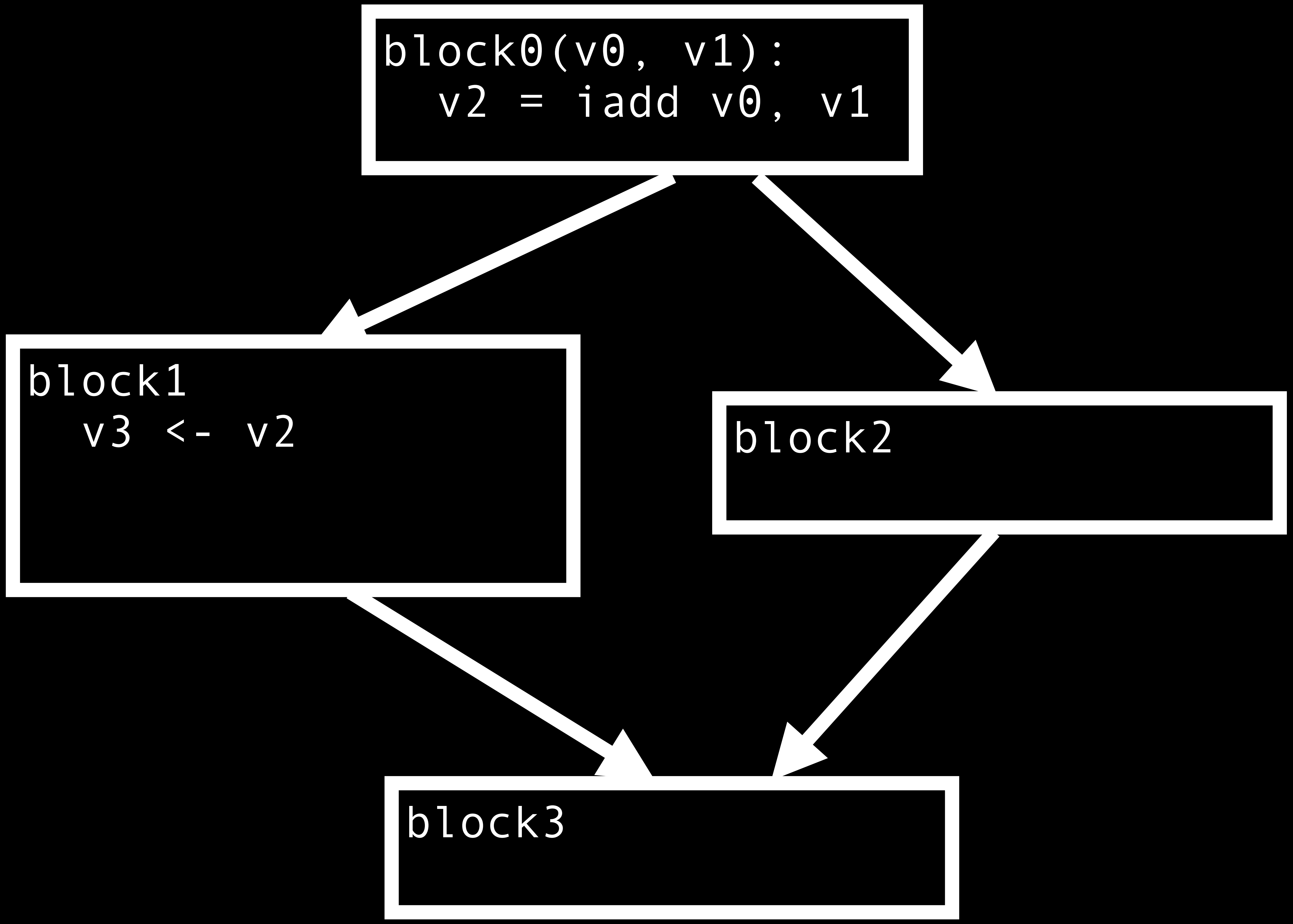
GVN



GVN (Global Value Numbering):
If an operator dominates a duplicate copy of itself, reuse the original.

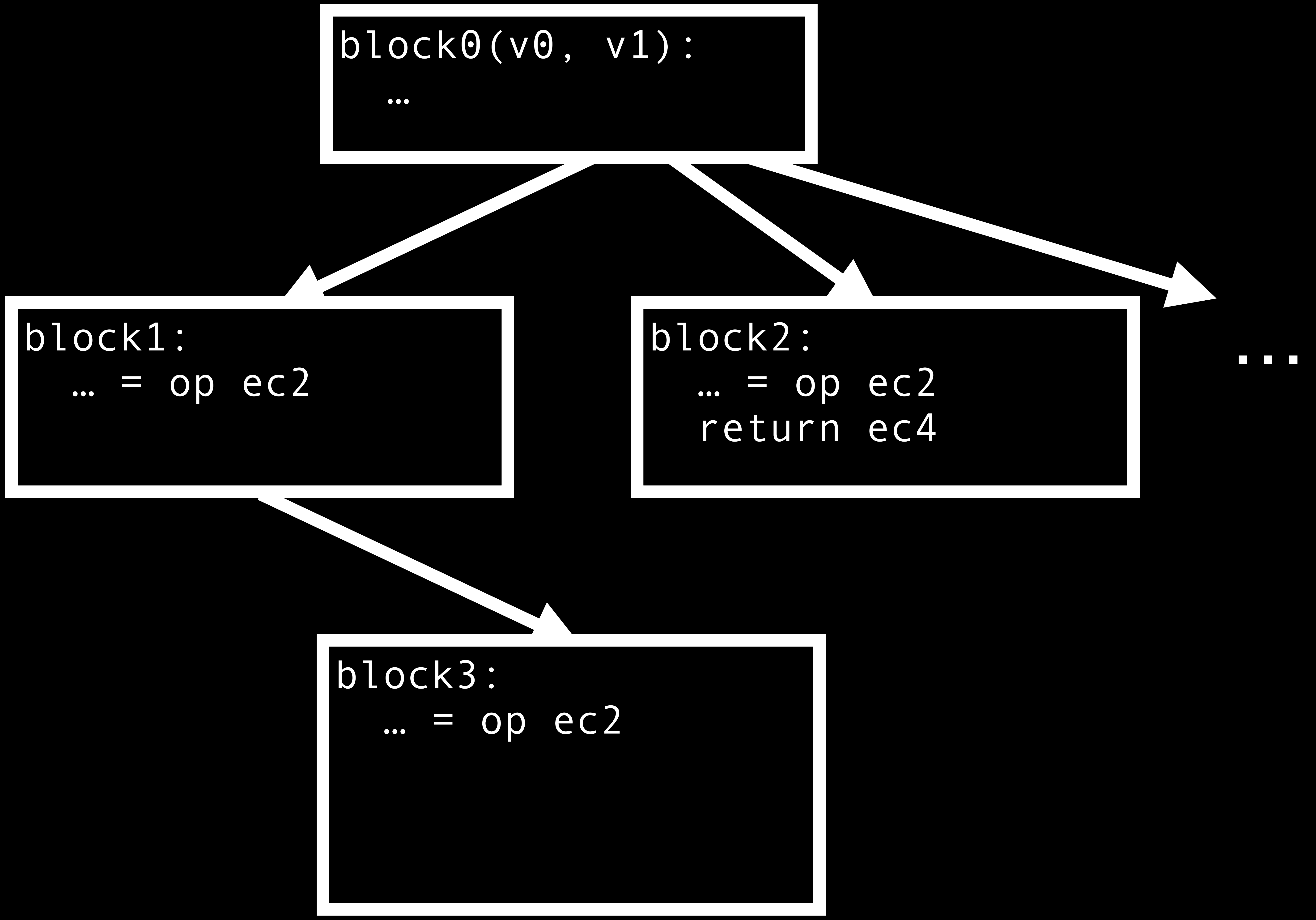
Implement with *domtree preorder traversal* and a *scoped map*.

GVN

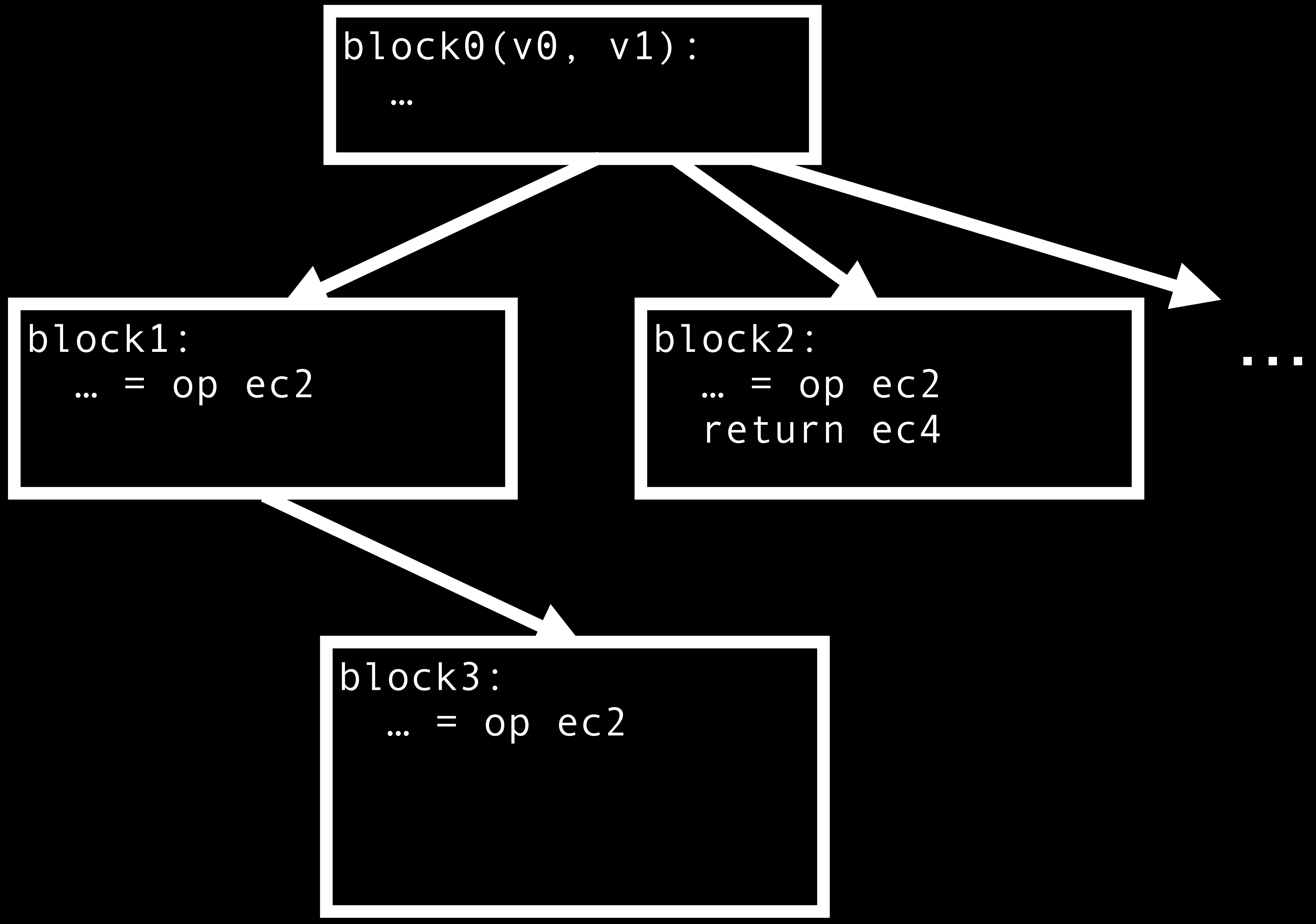


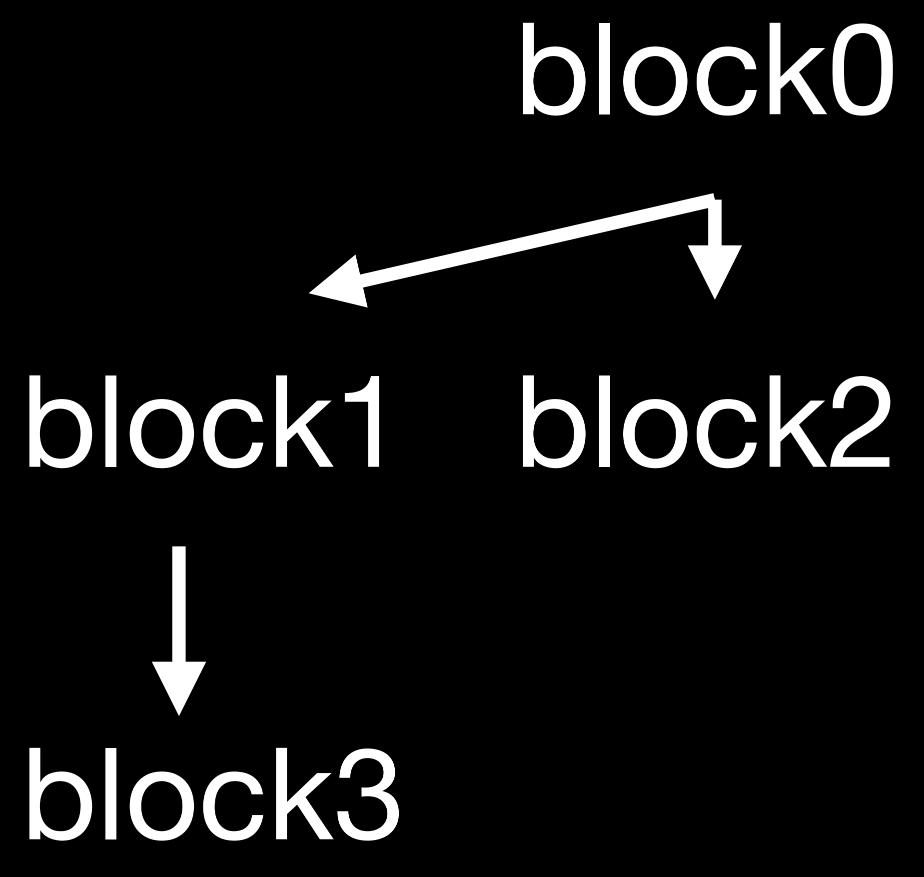
Implement with
domtree preorder traversal
and a *scoped map*.

Scoped Elaboration

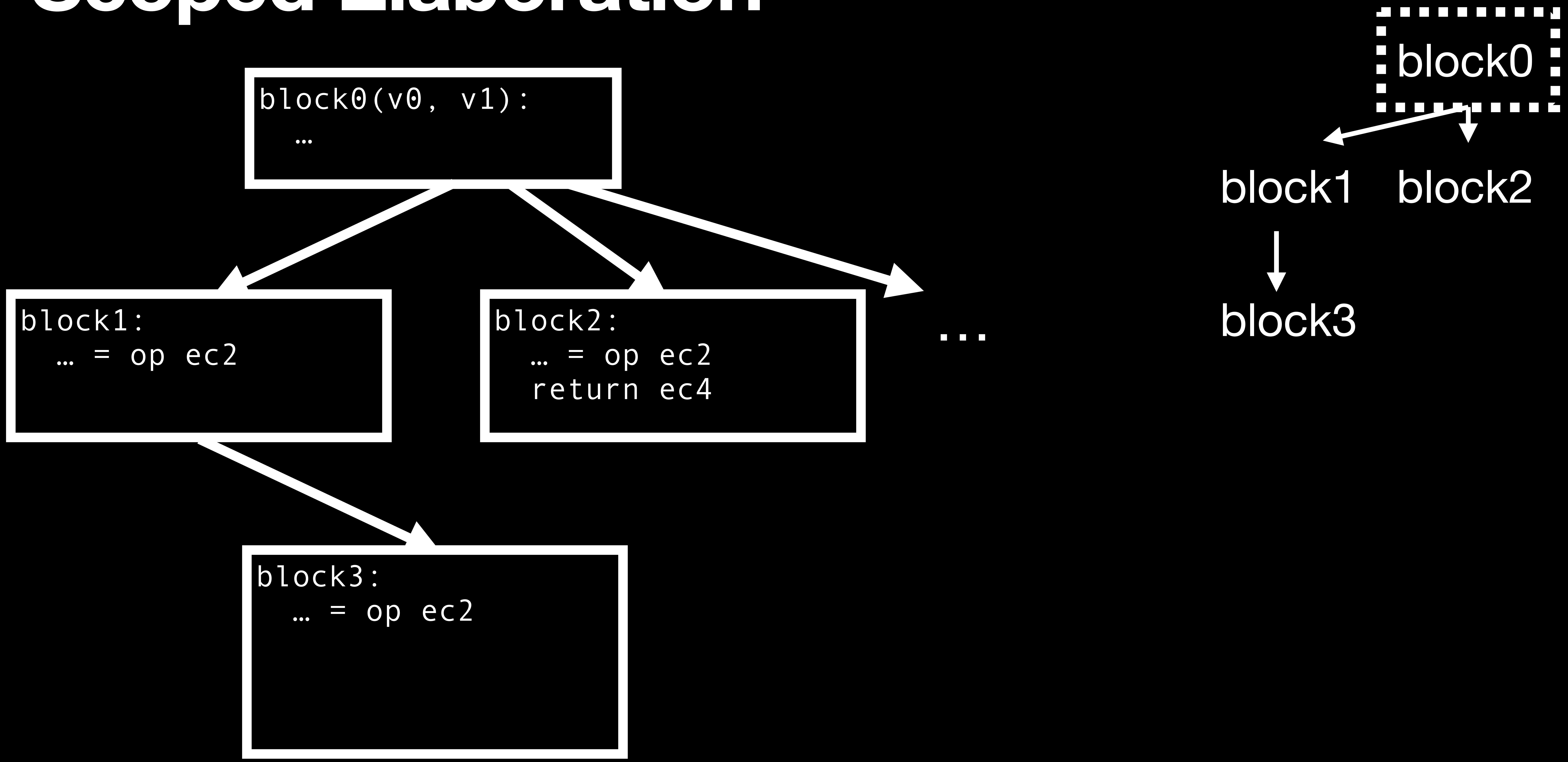


Scoped Elaboration

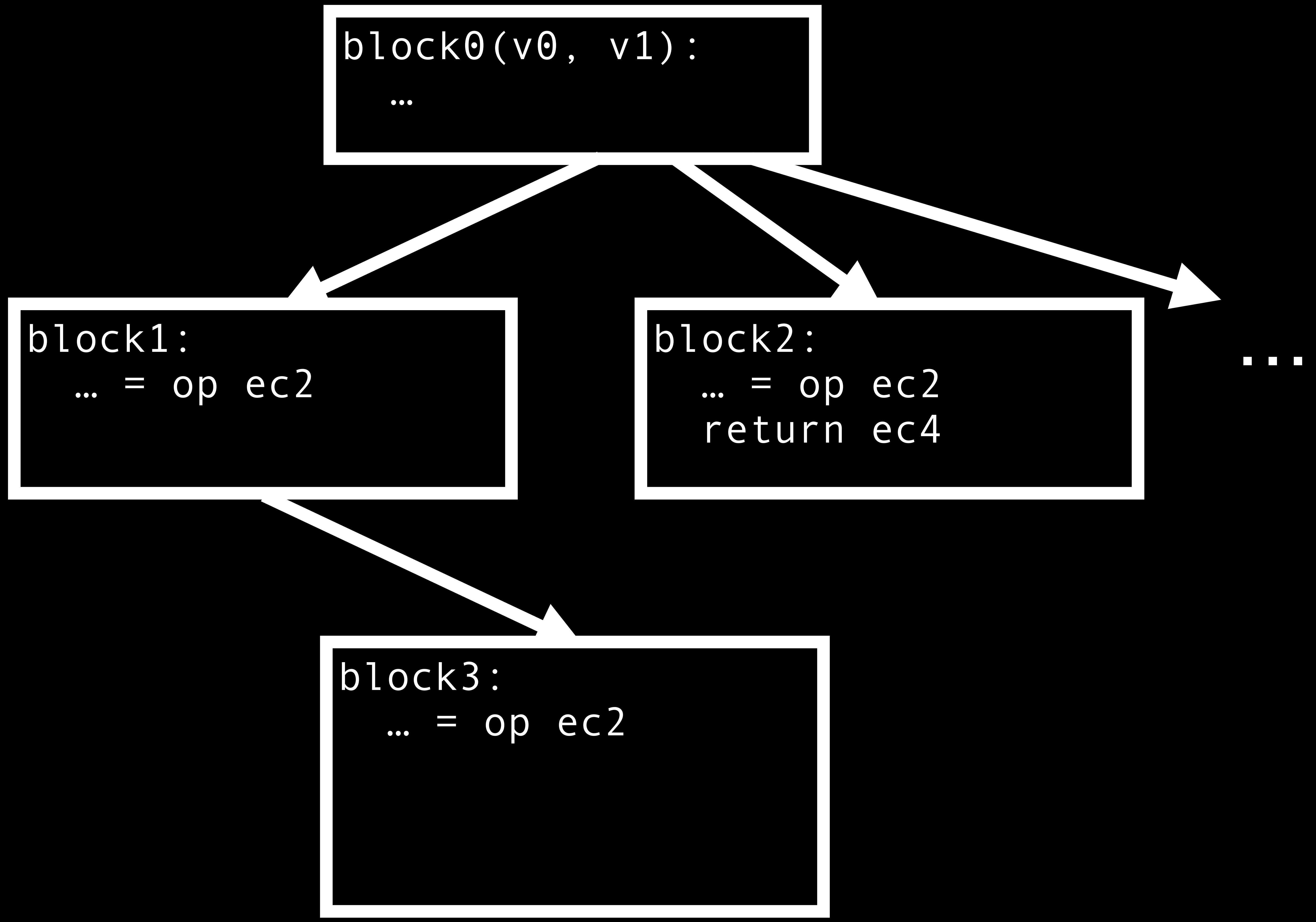


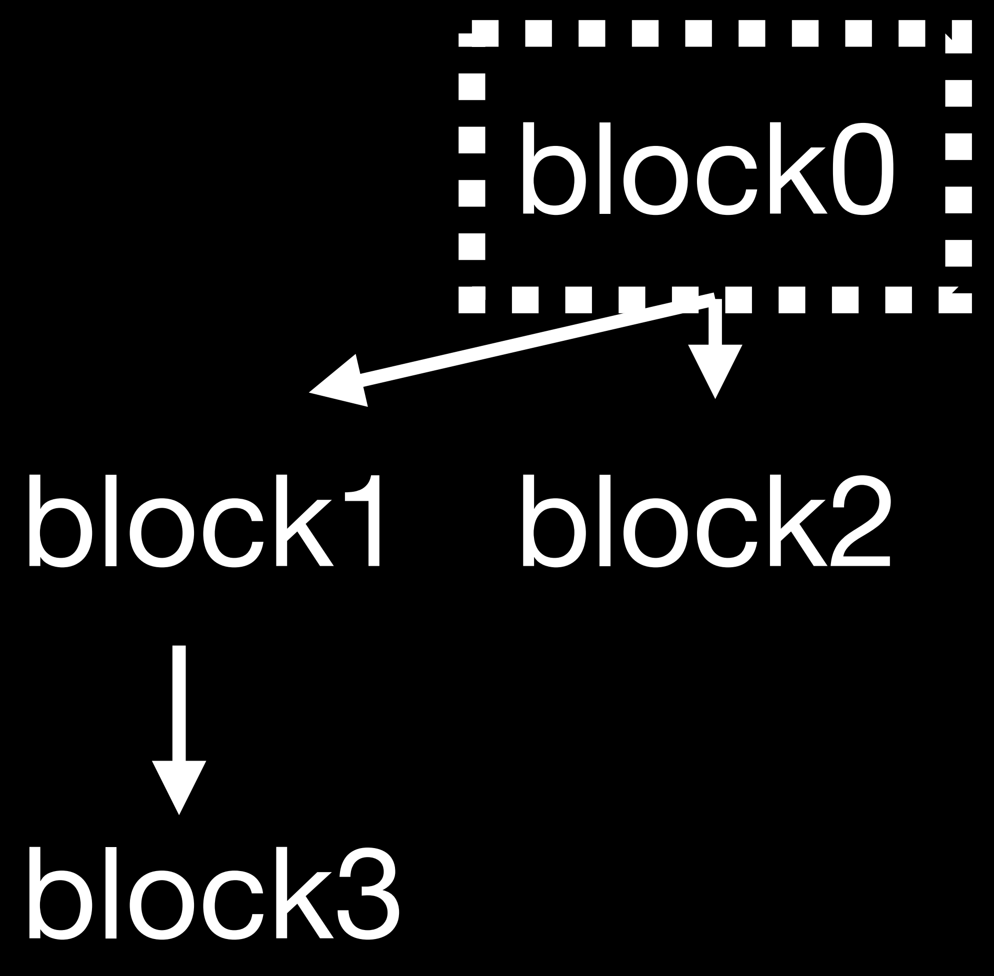


Scoped Elaboration



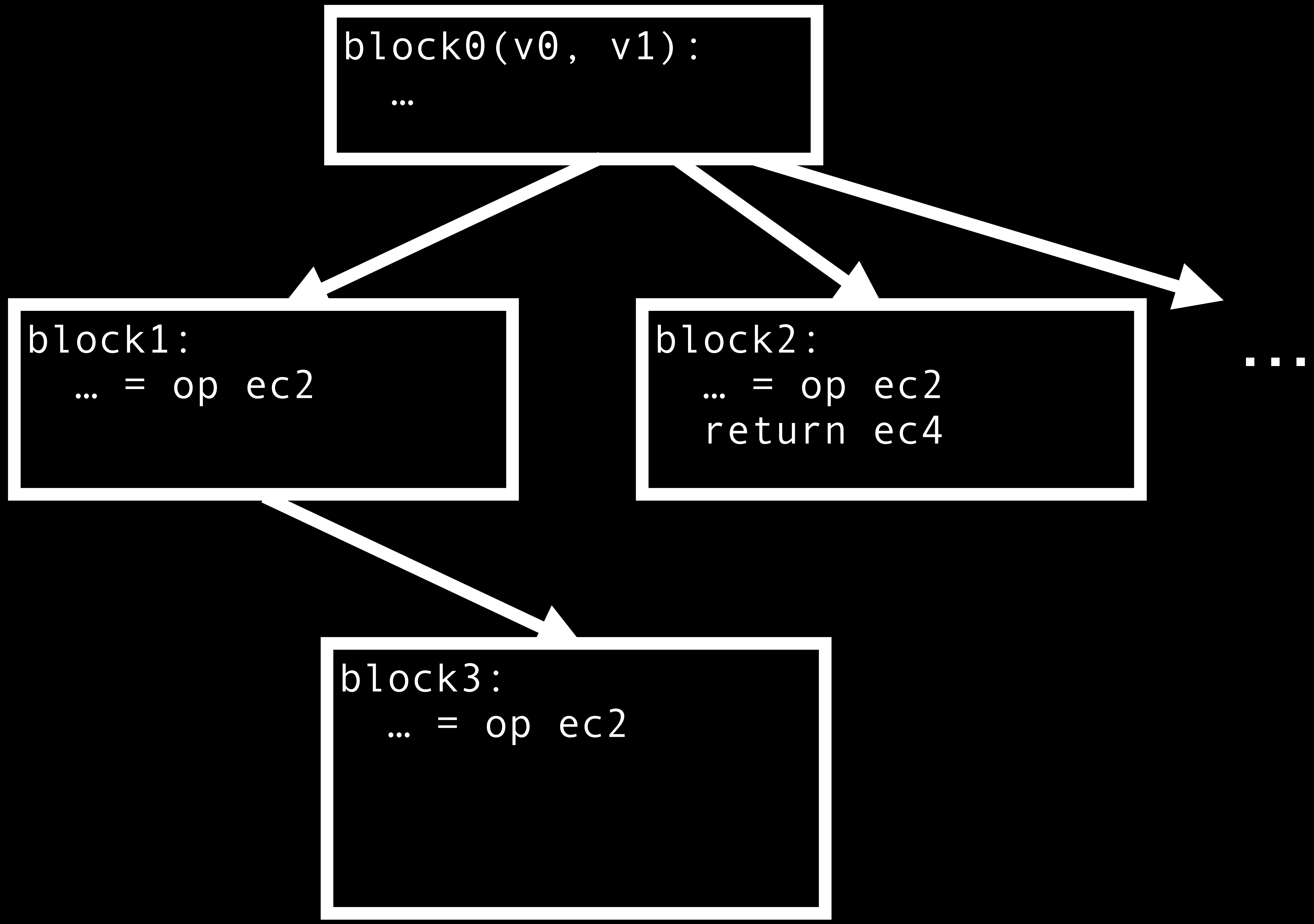
Scoped Elaboration

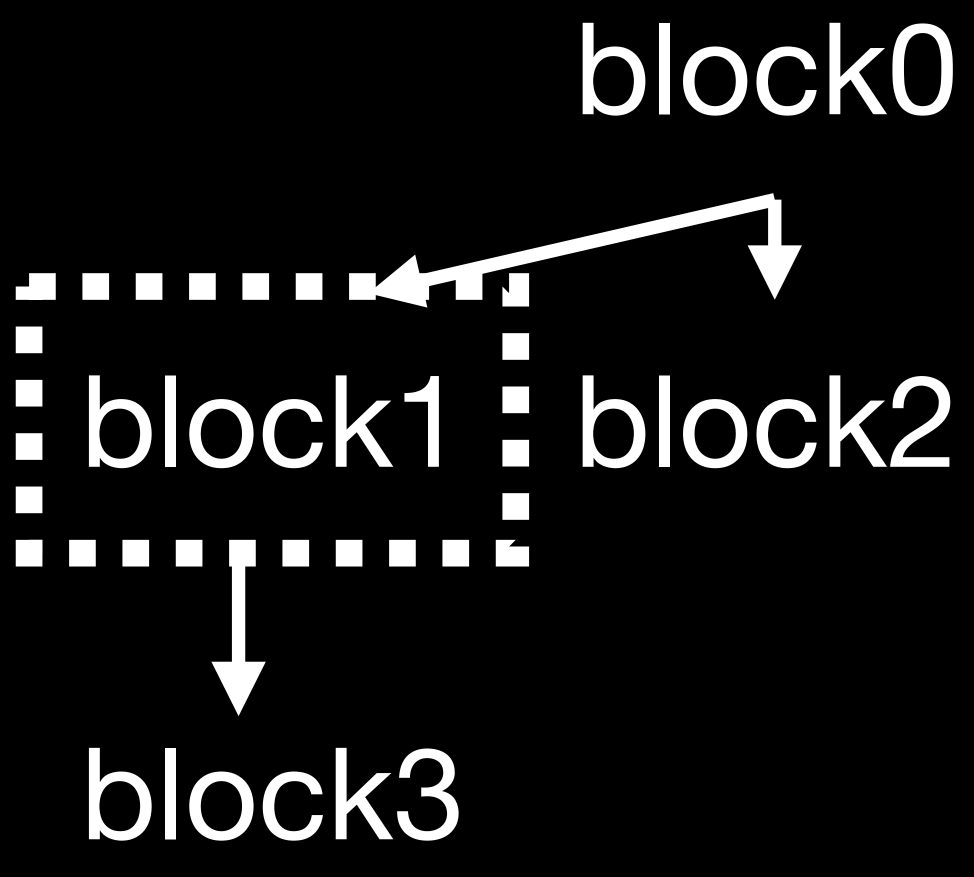




eclass	elaborated
ec0	v0
ec1	v1

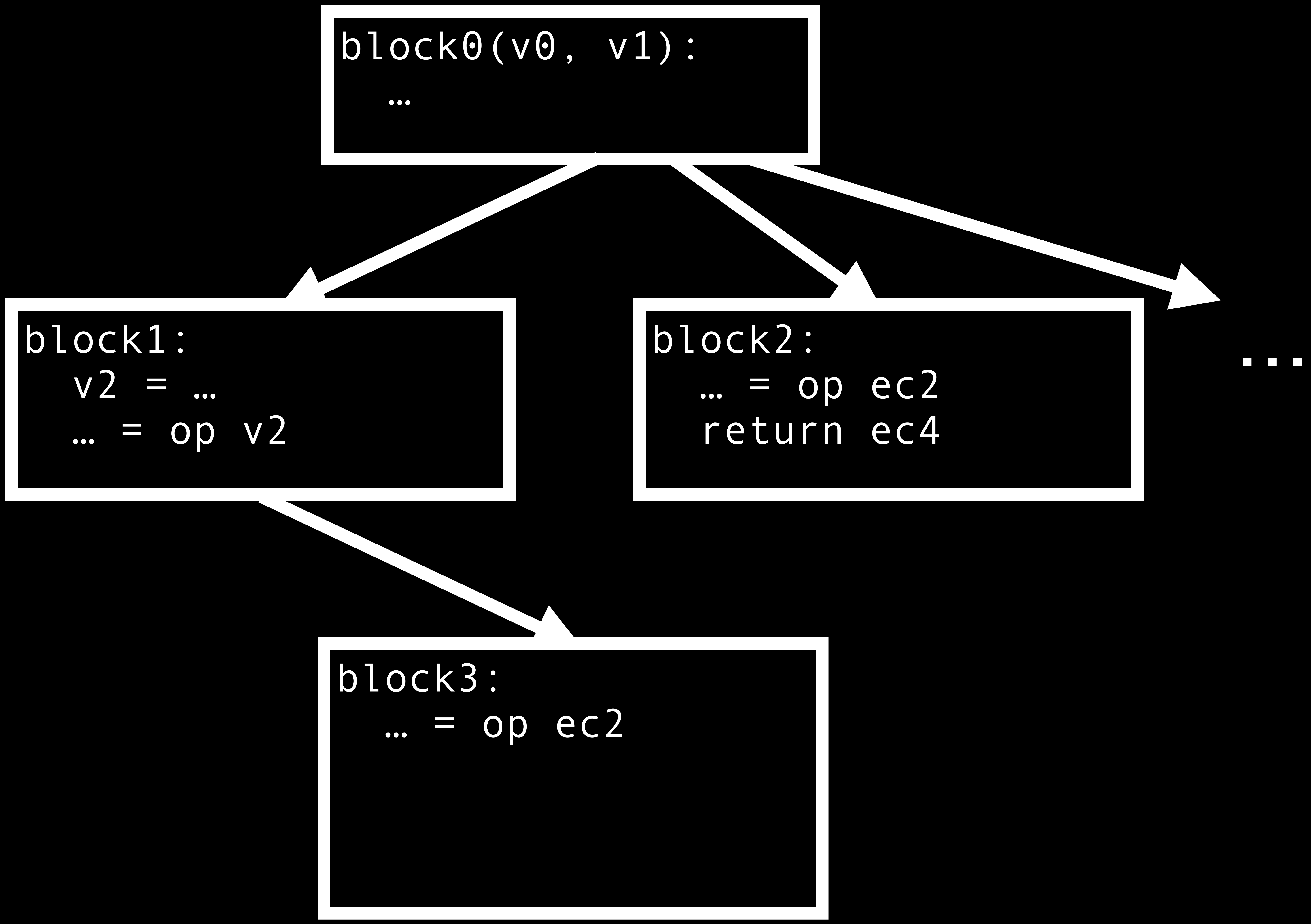
Scoped Elaboration

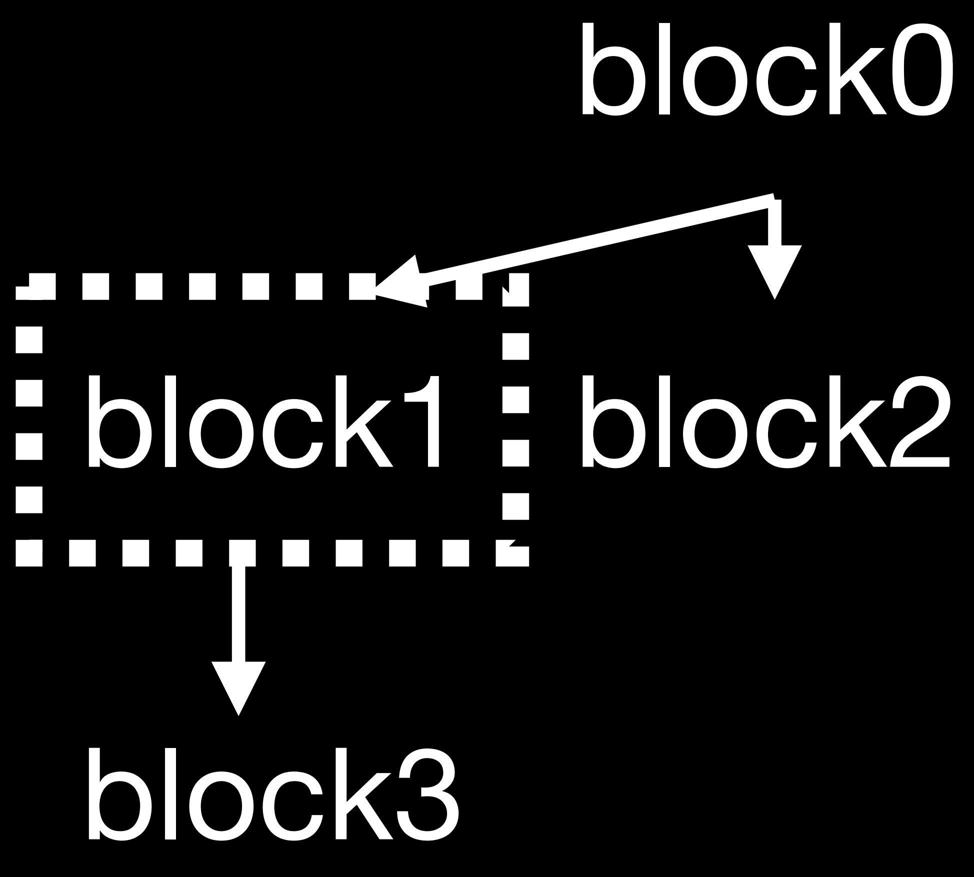




eclass	elaborated
ec0	v0
ec1	v1

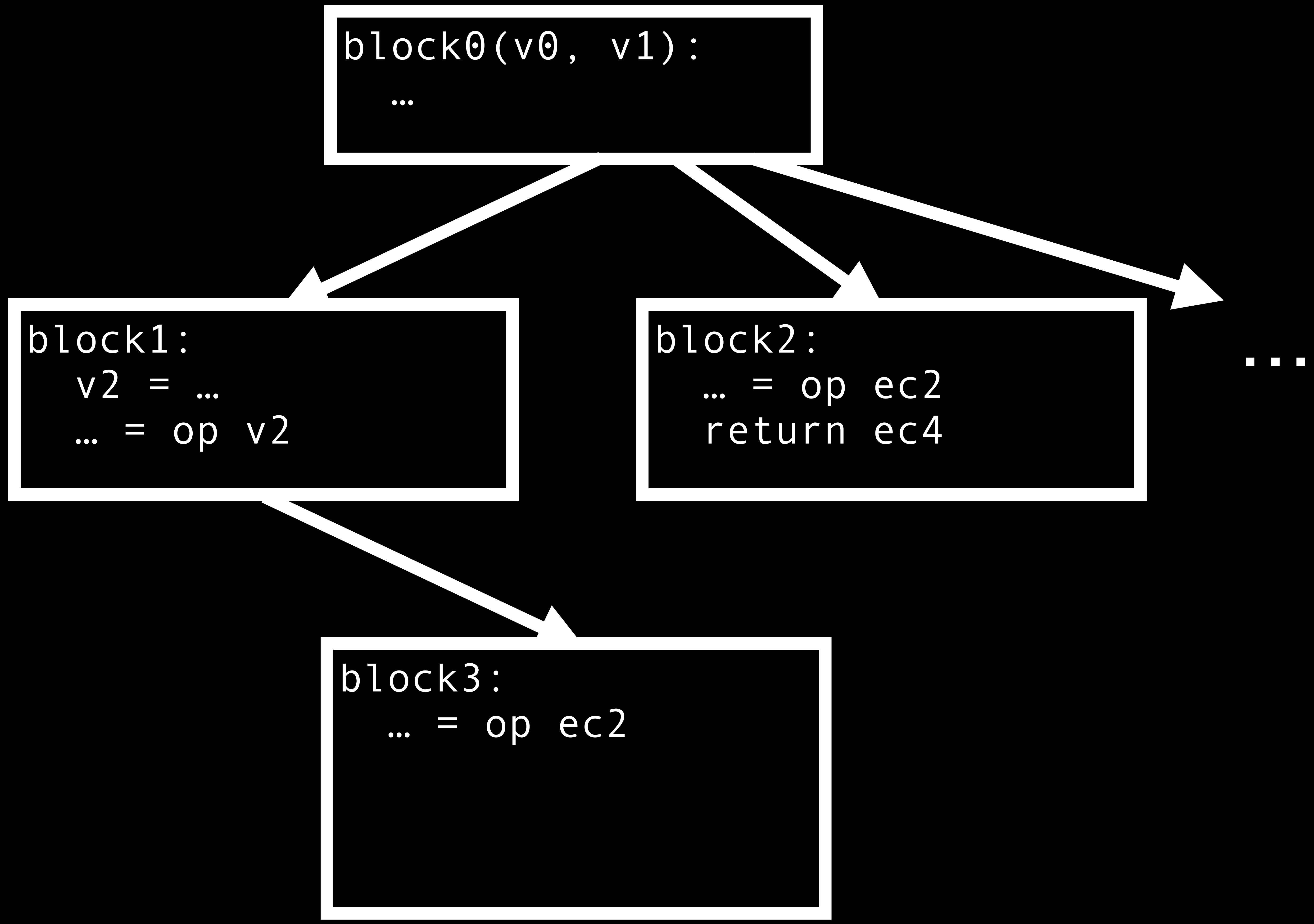
Scoped Elaboration

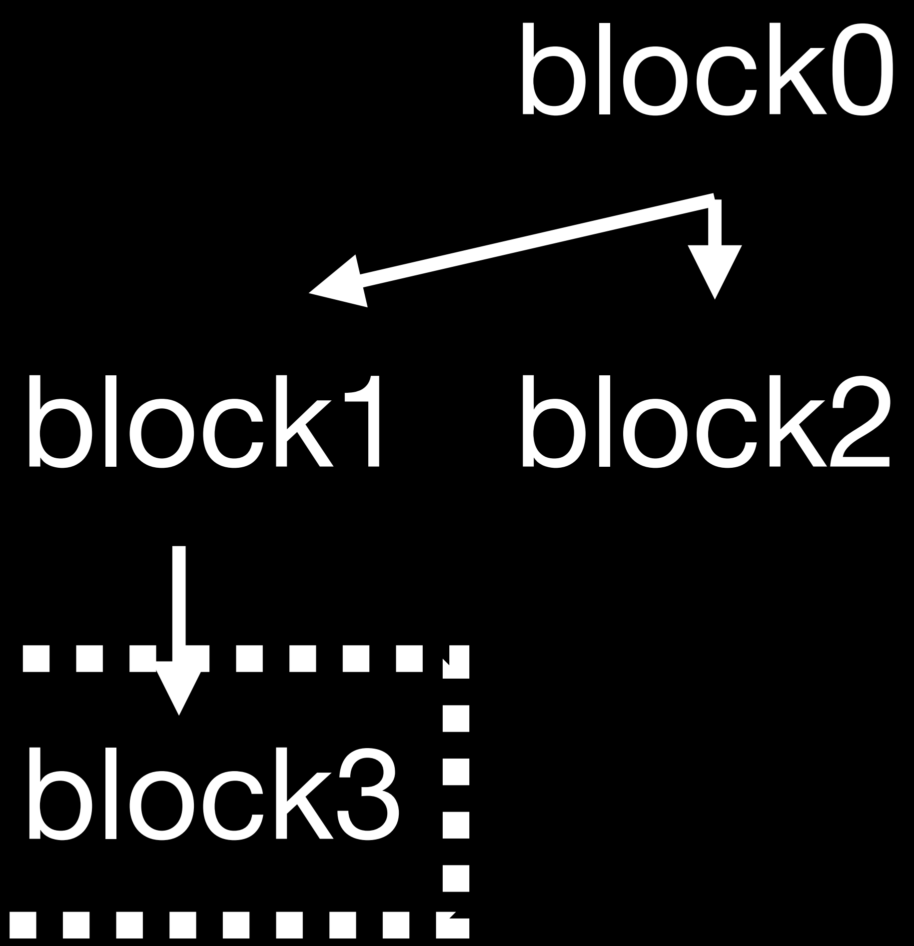




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

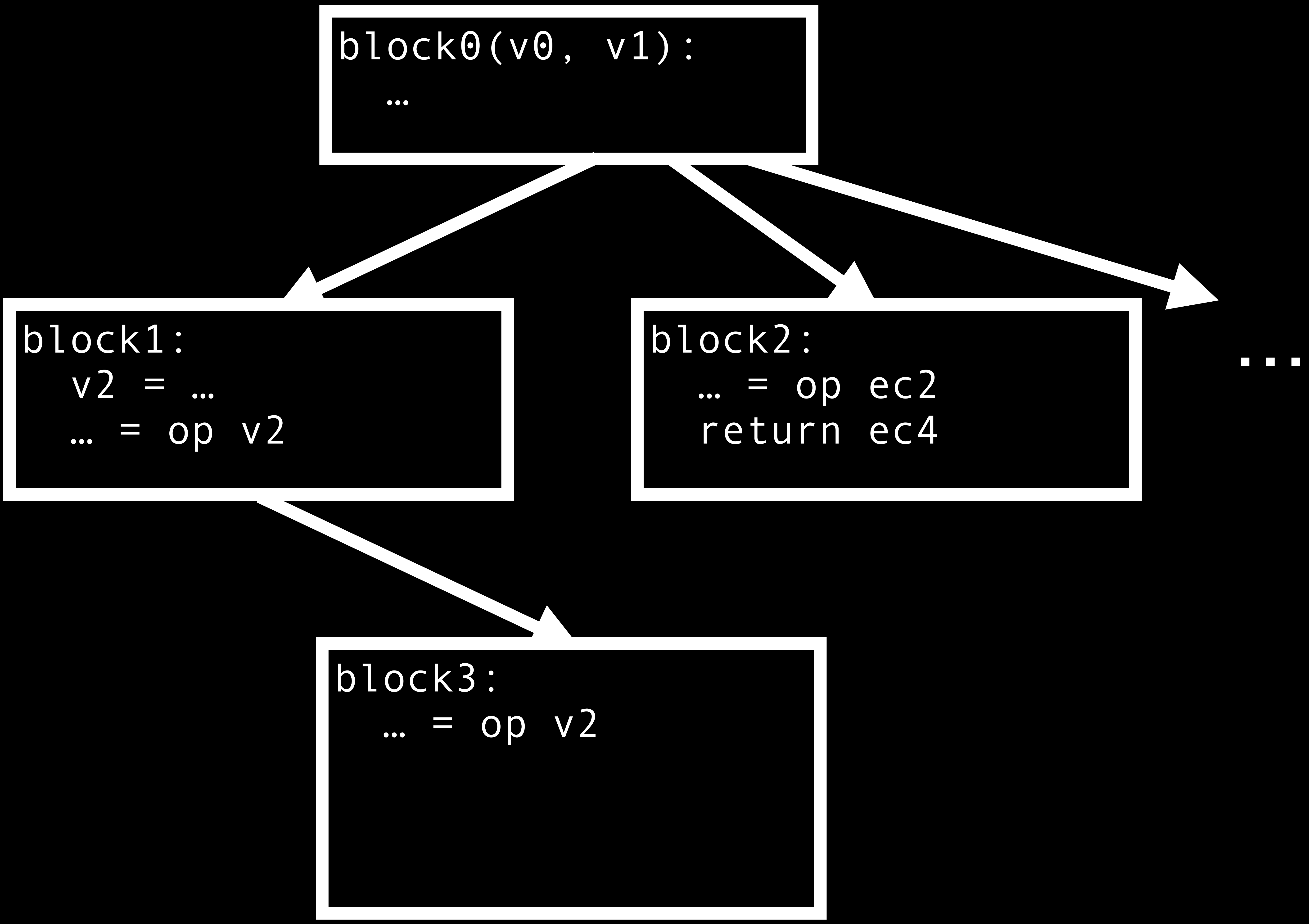
Scoped Elaboration

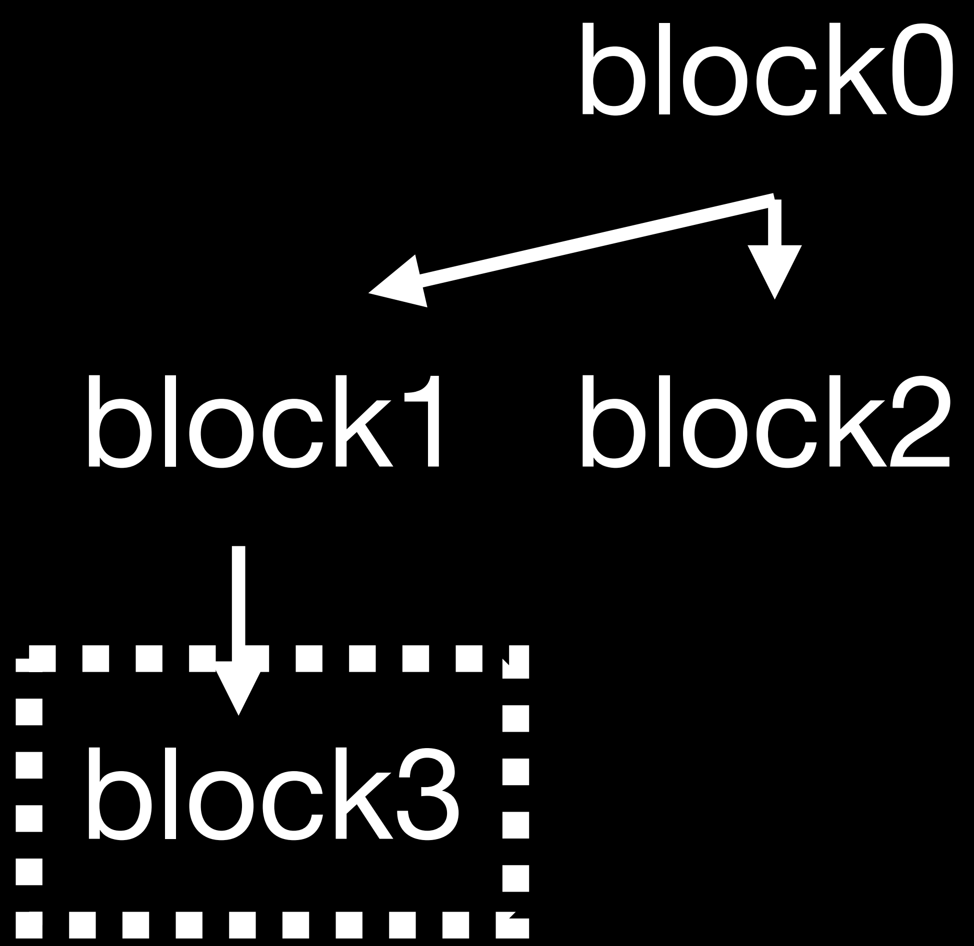




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

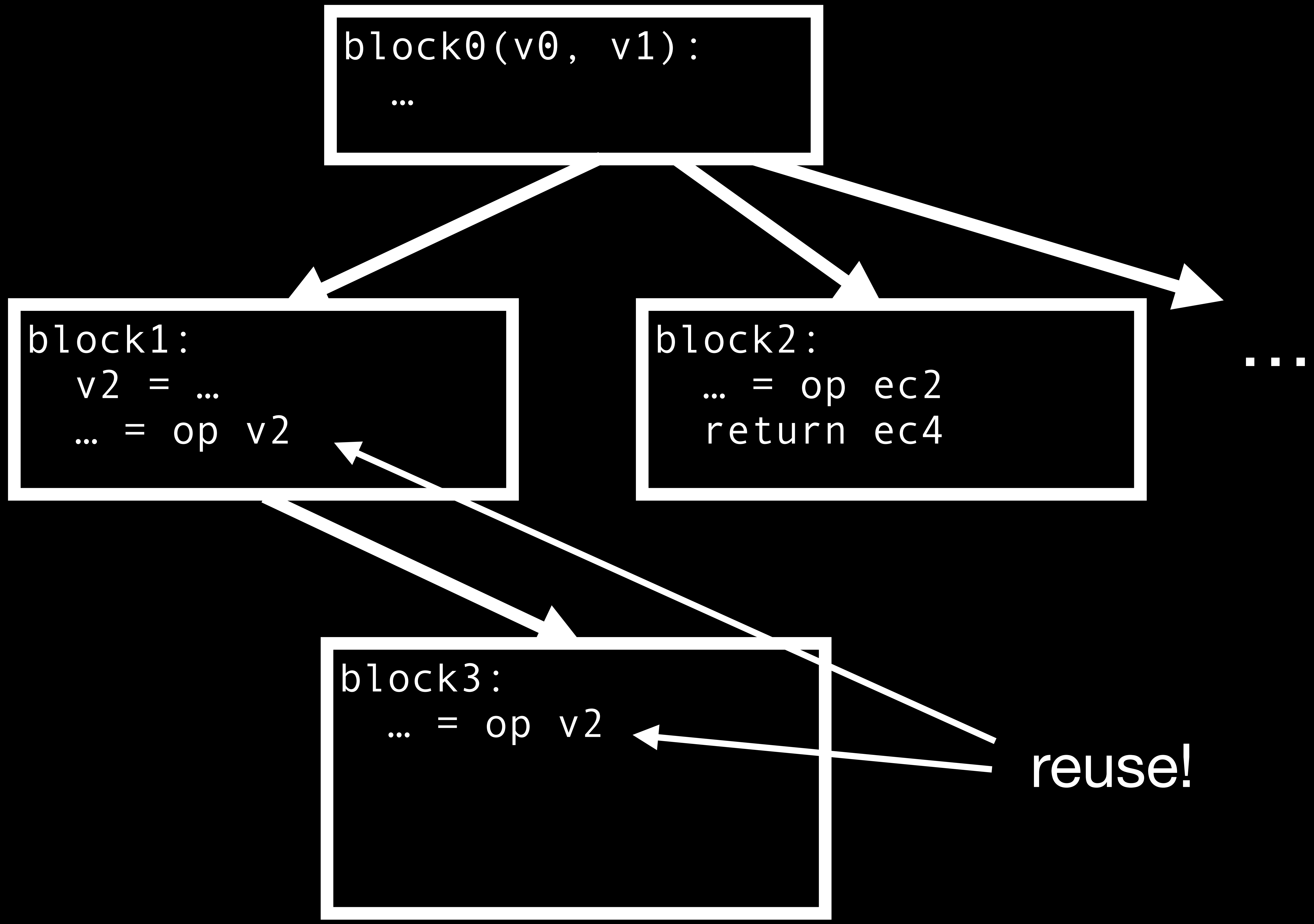
Scoped Elaboration

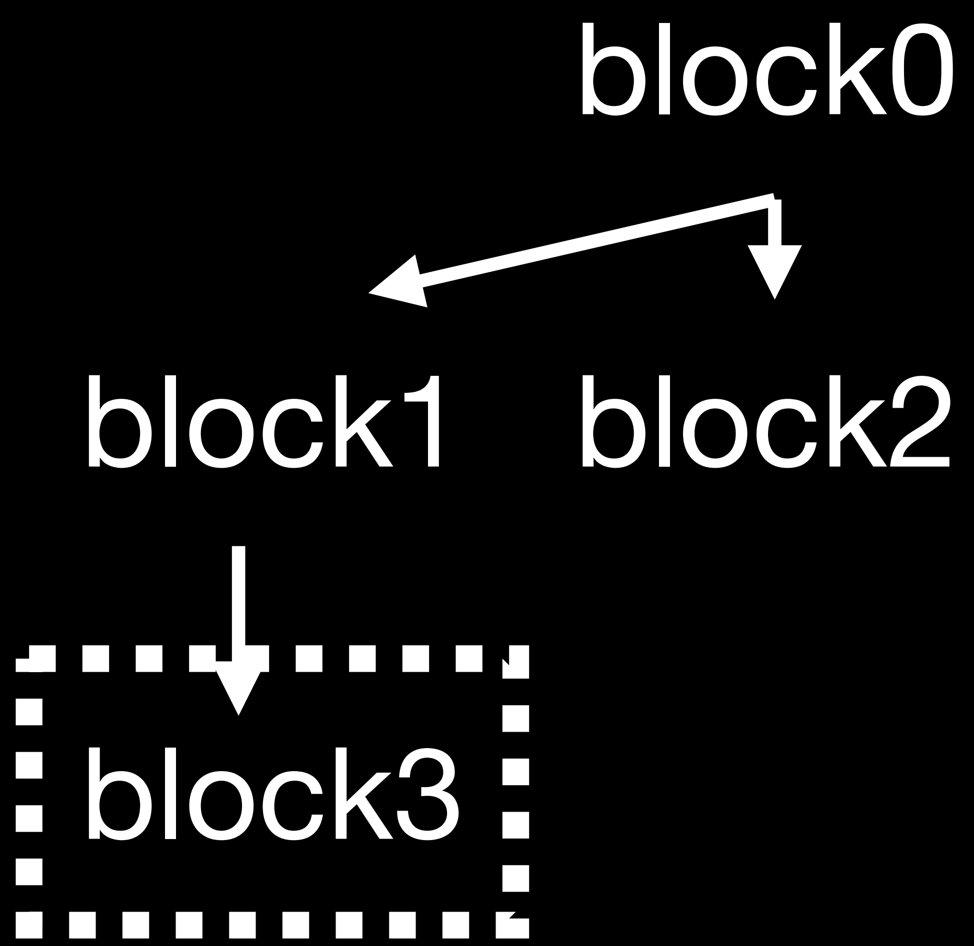




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

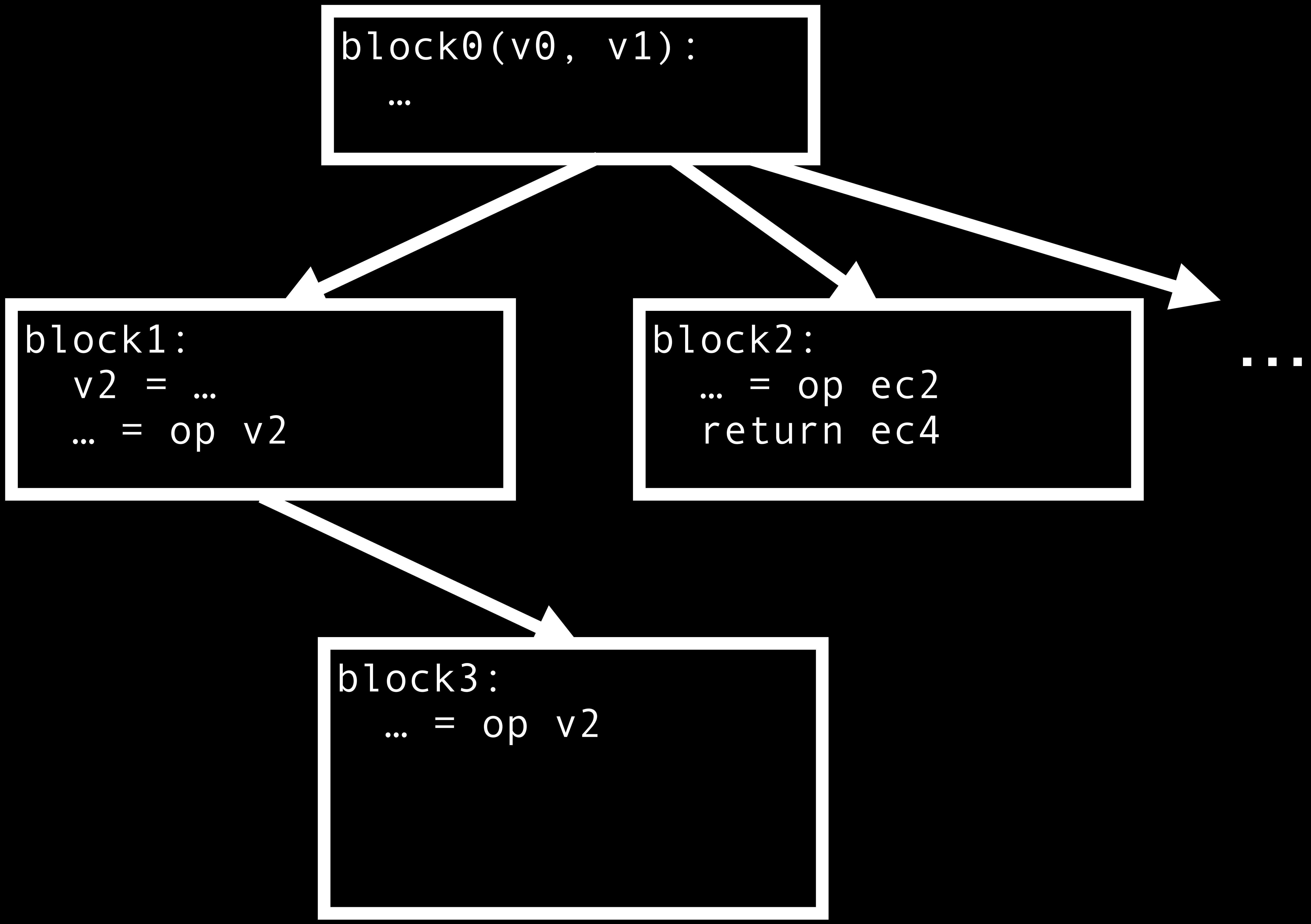
Scoped Elaboration

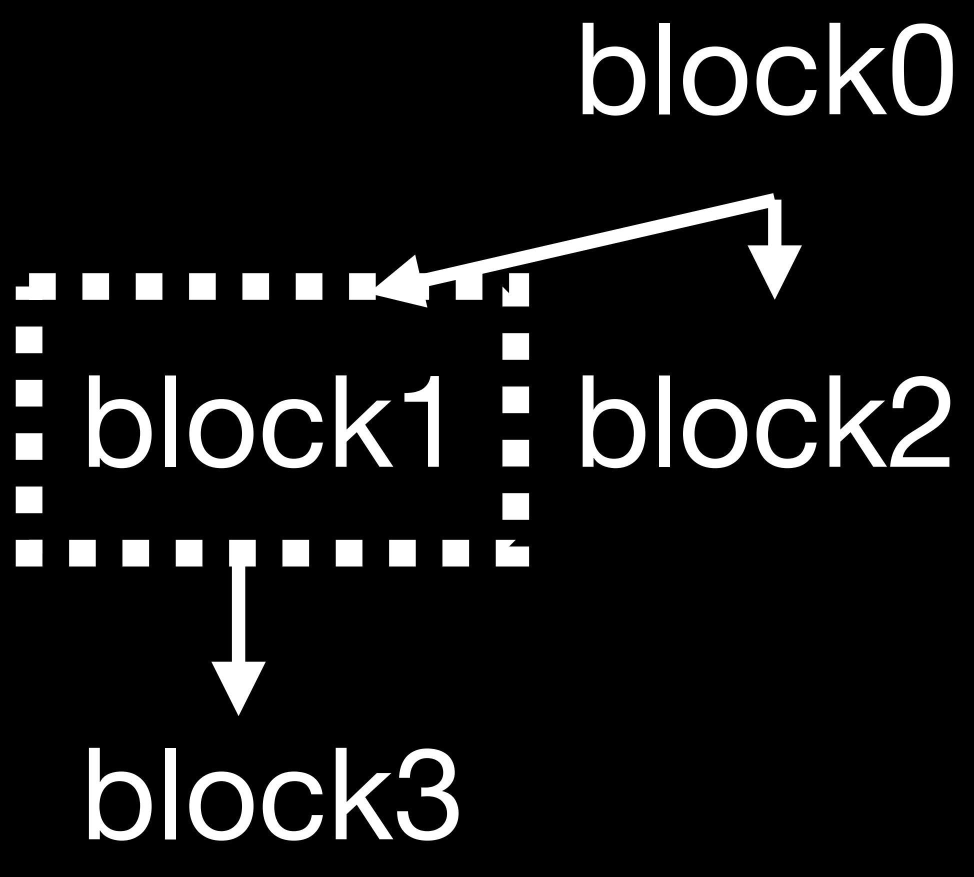




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

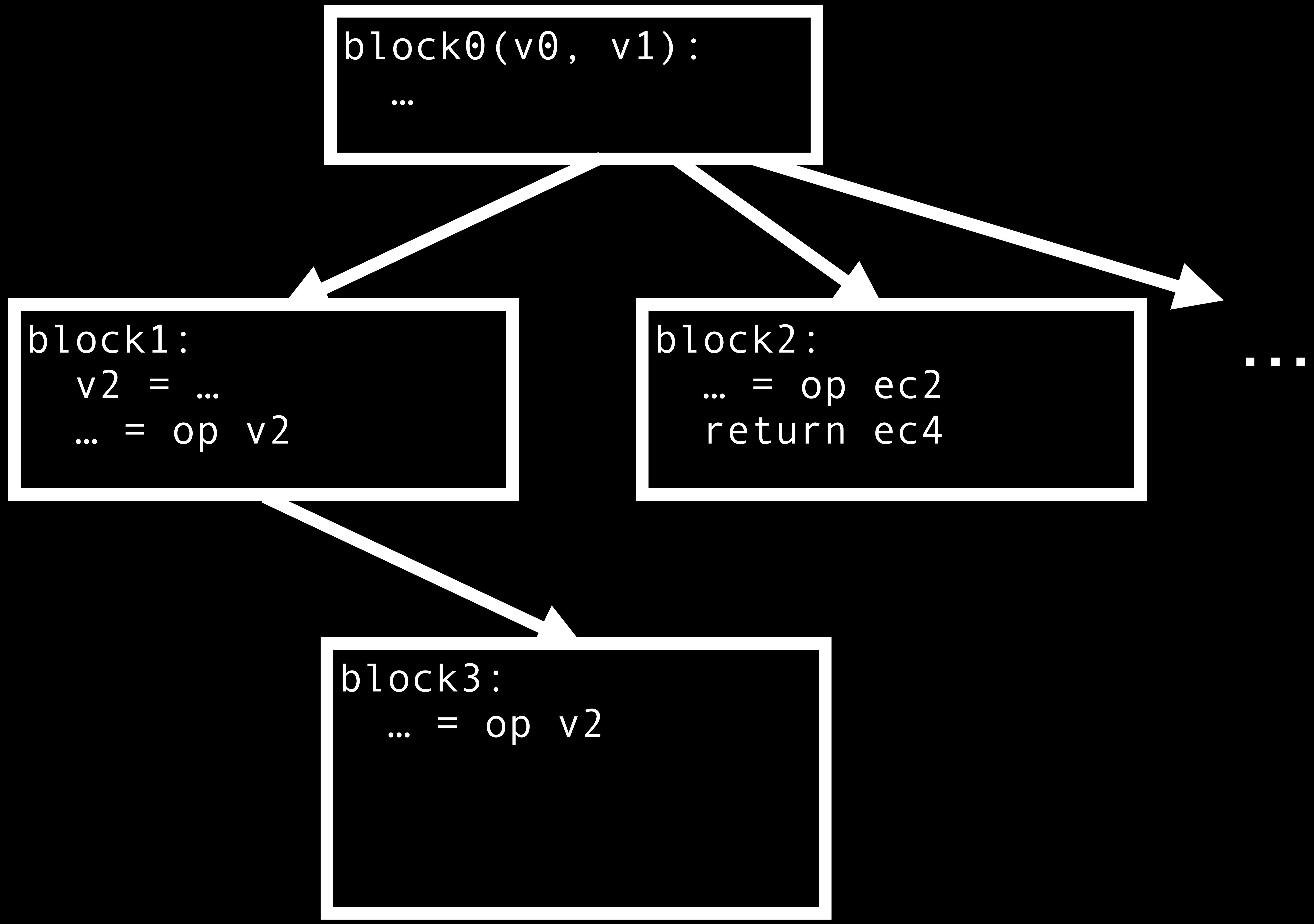
Scoped Elaboration

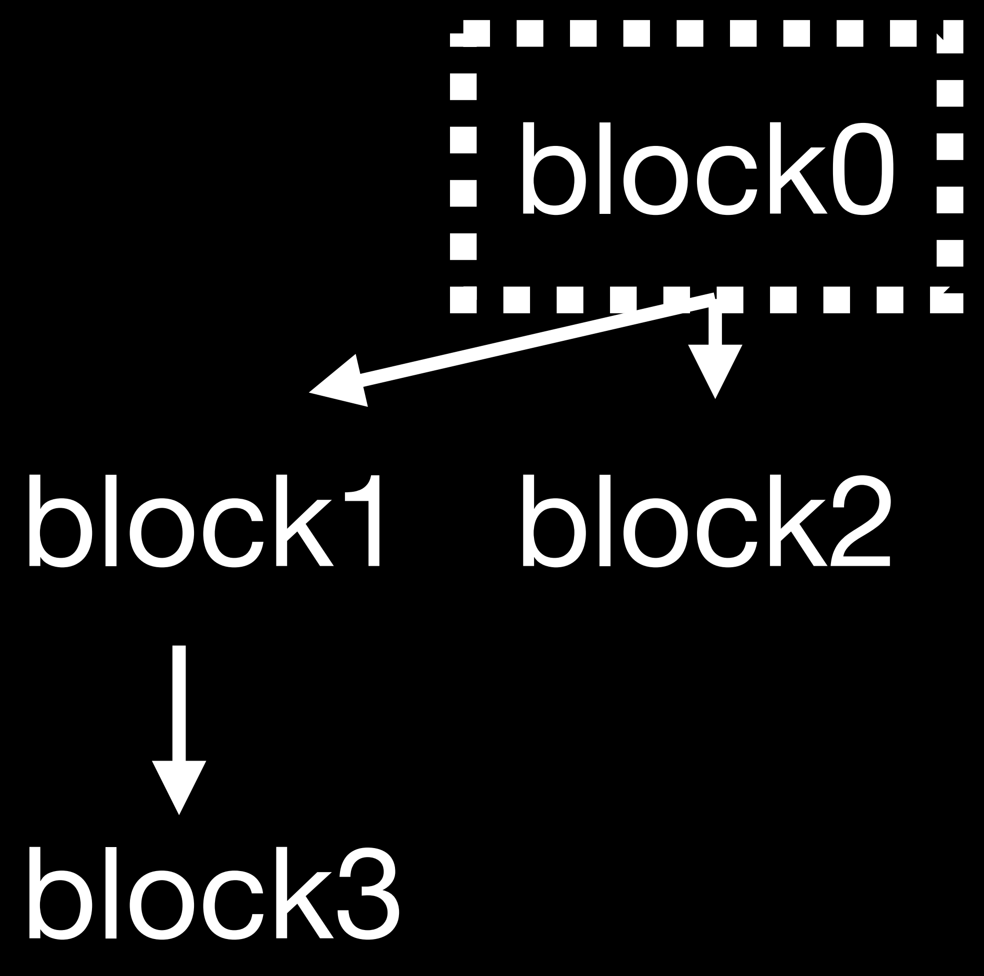




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

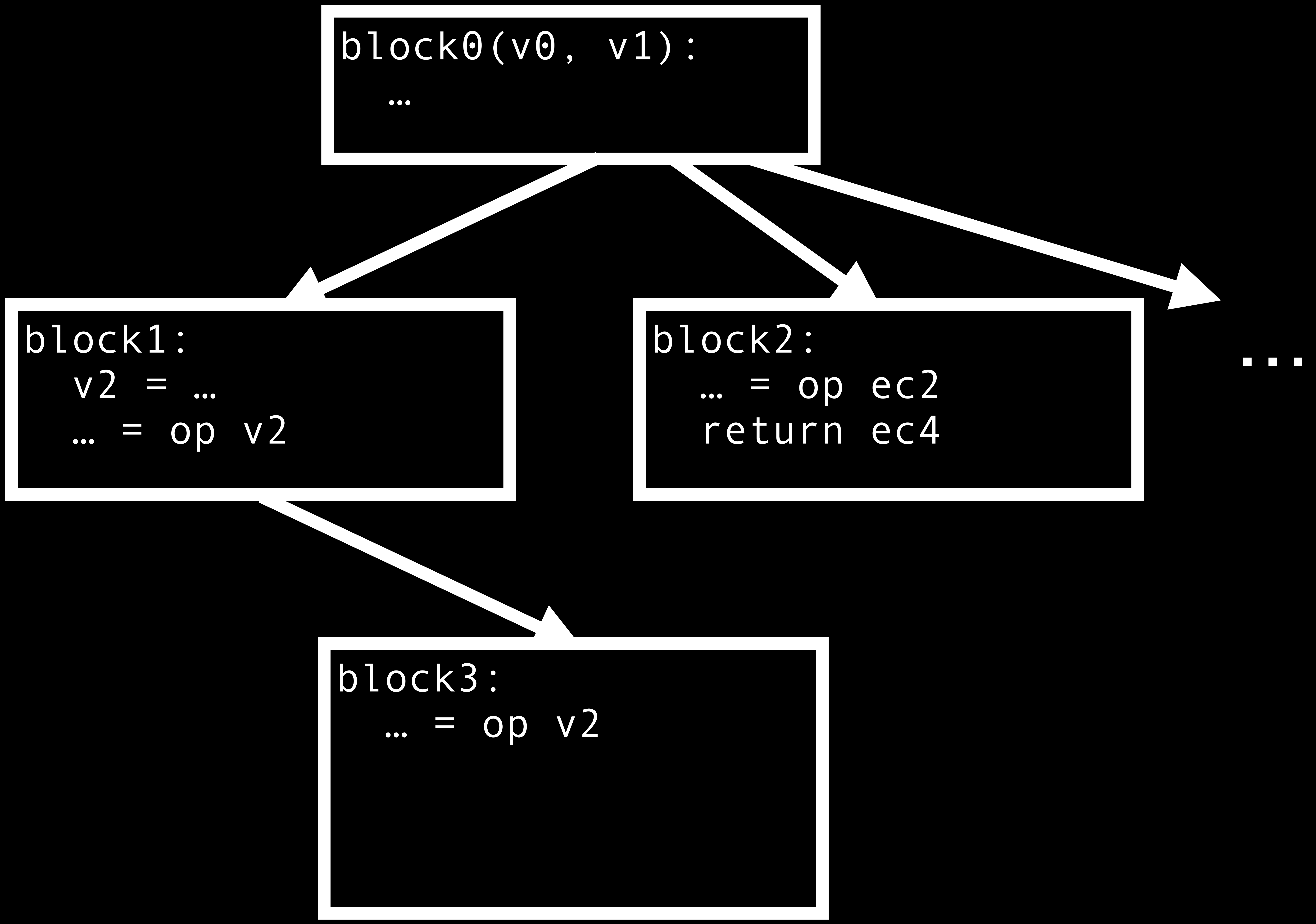
Scoped Elaboration

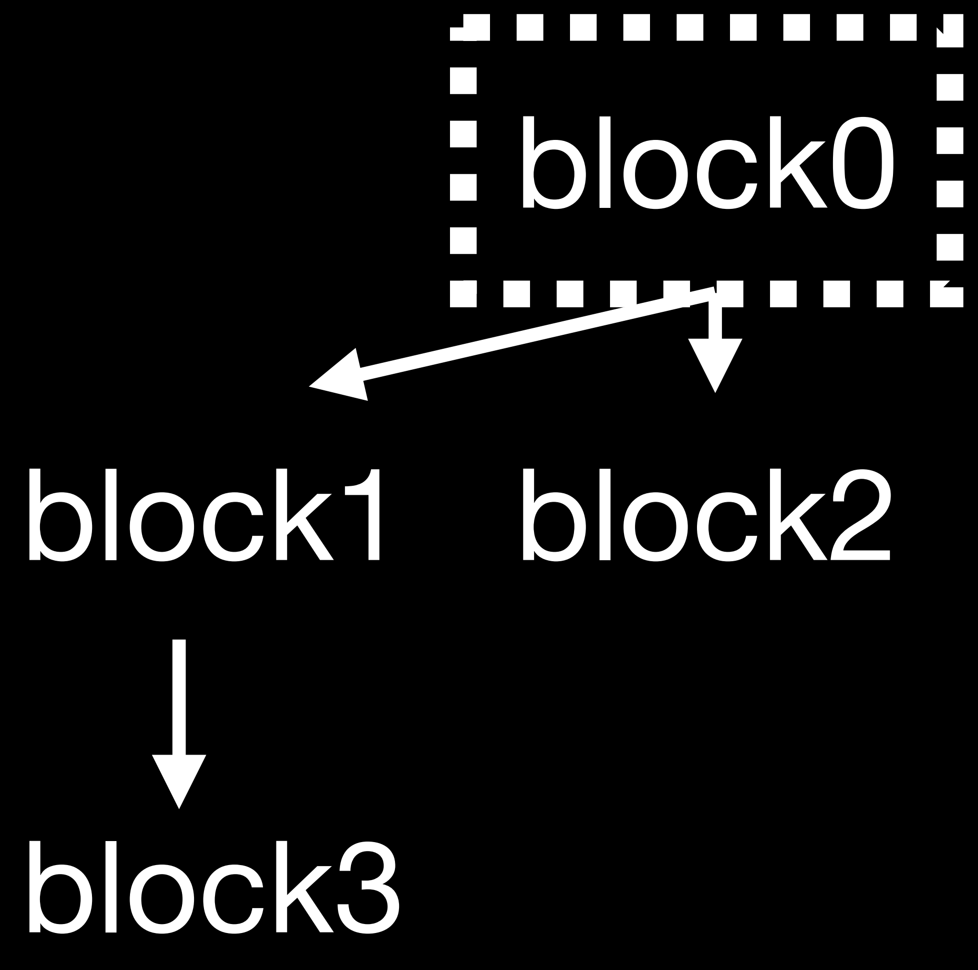




eclass	elaborated
ec0	v0
ec1	v1
ec2	v2

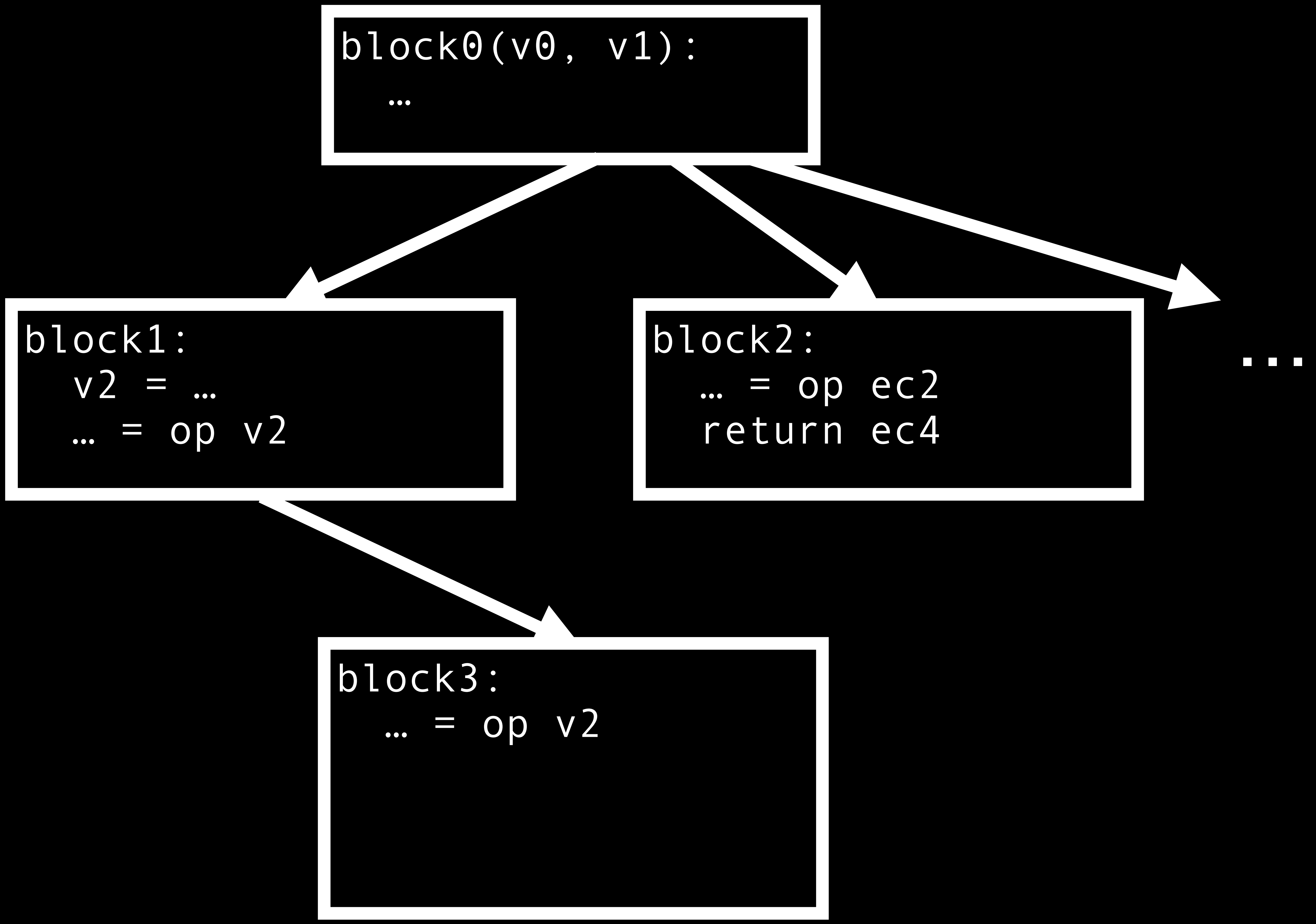
Scoped Elaboration

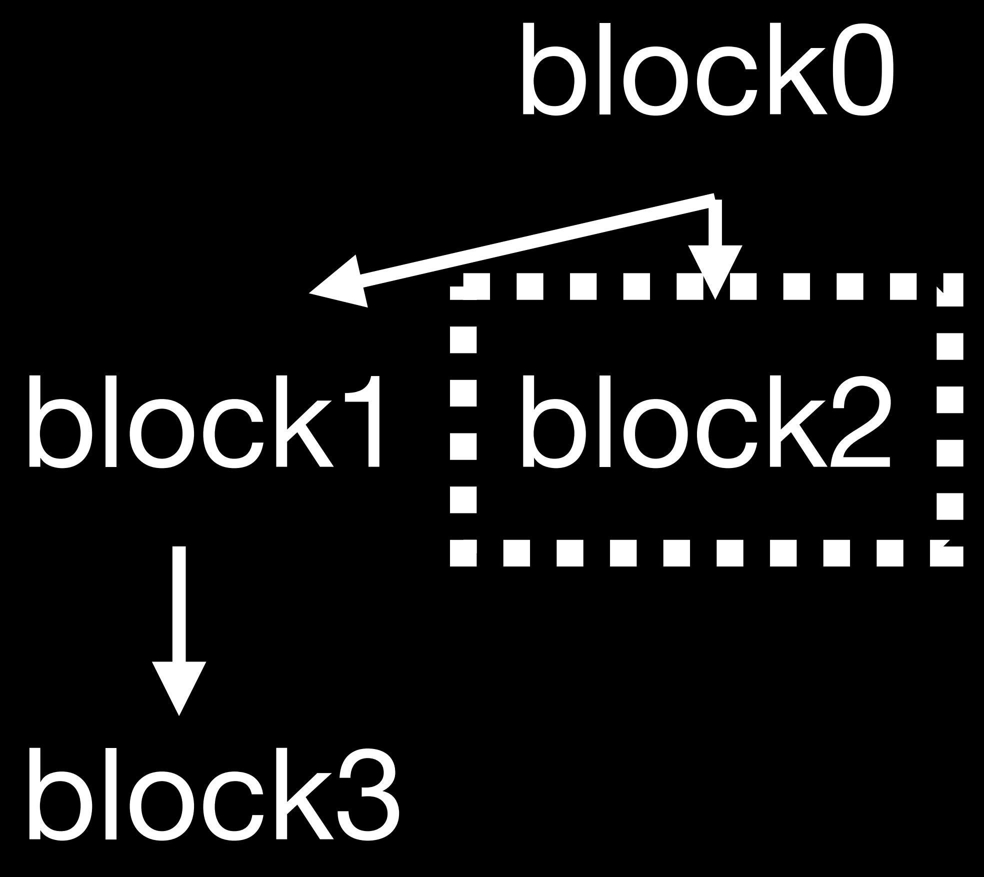




eclass	elaborated
ec0	v0
ec1	v1

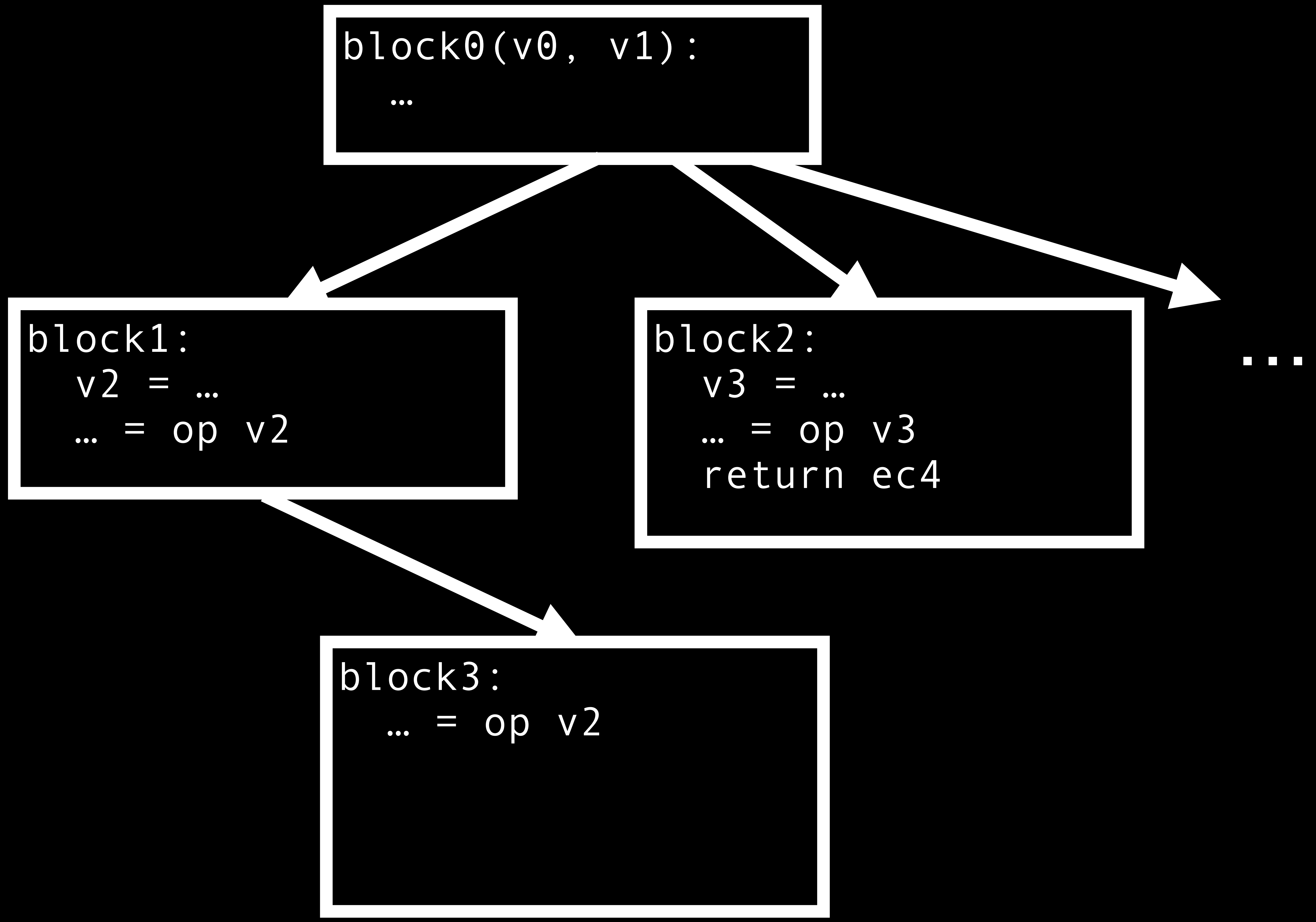
Scoped Elaboration

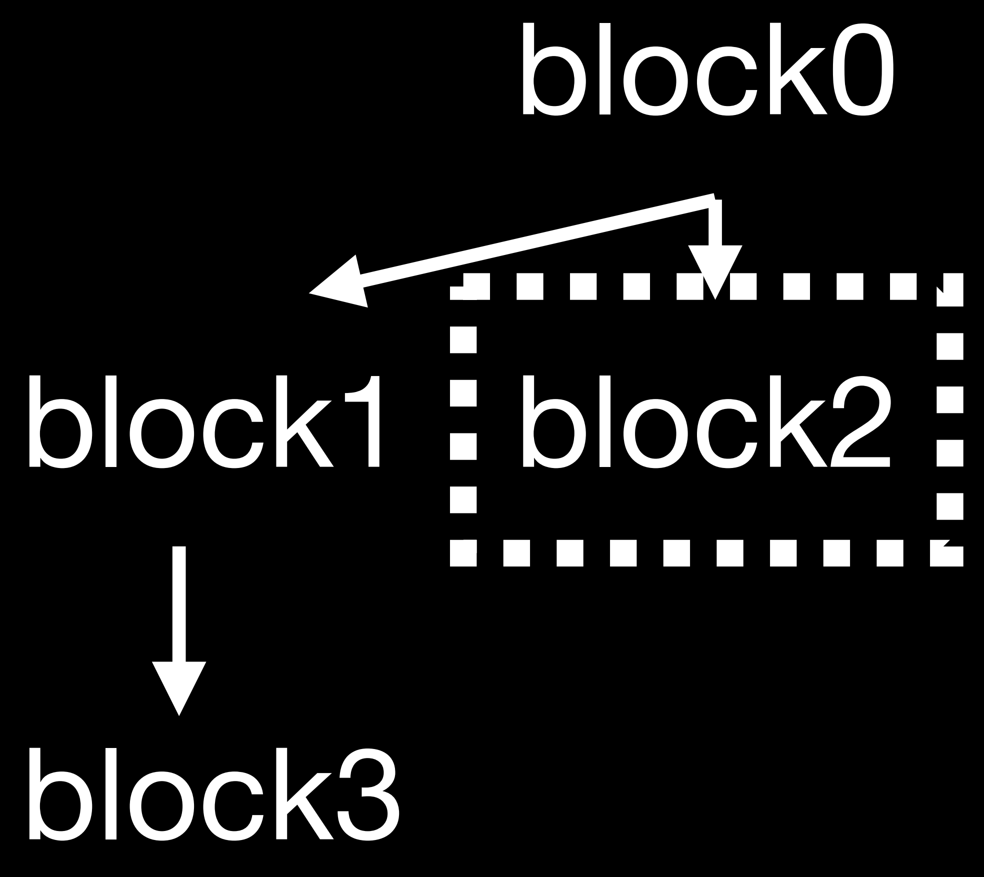




eclass	elaborated
ec0	v0
ec1	v1

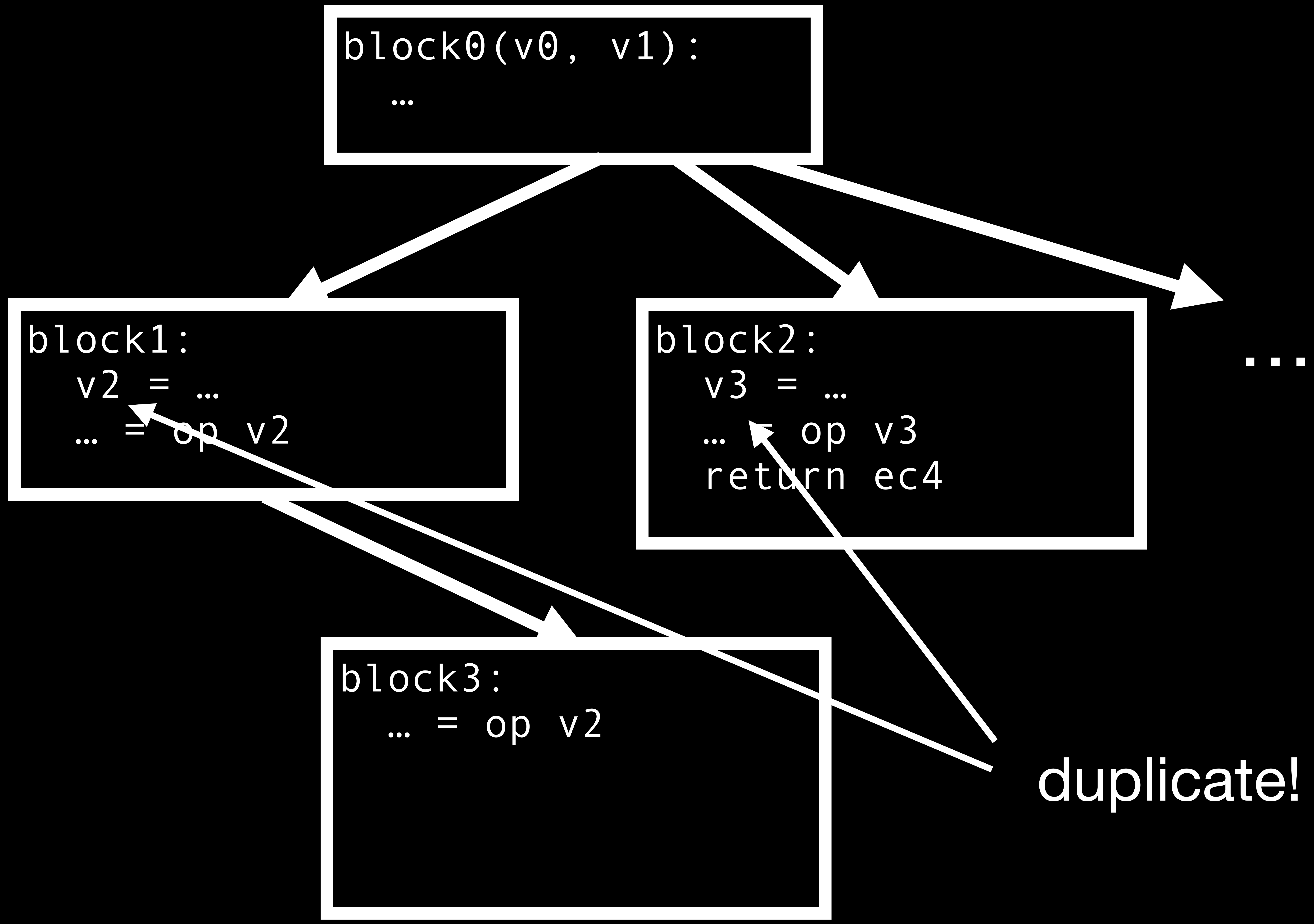
Scoped Elaboration

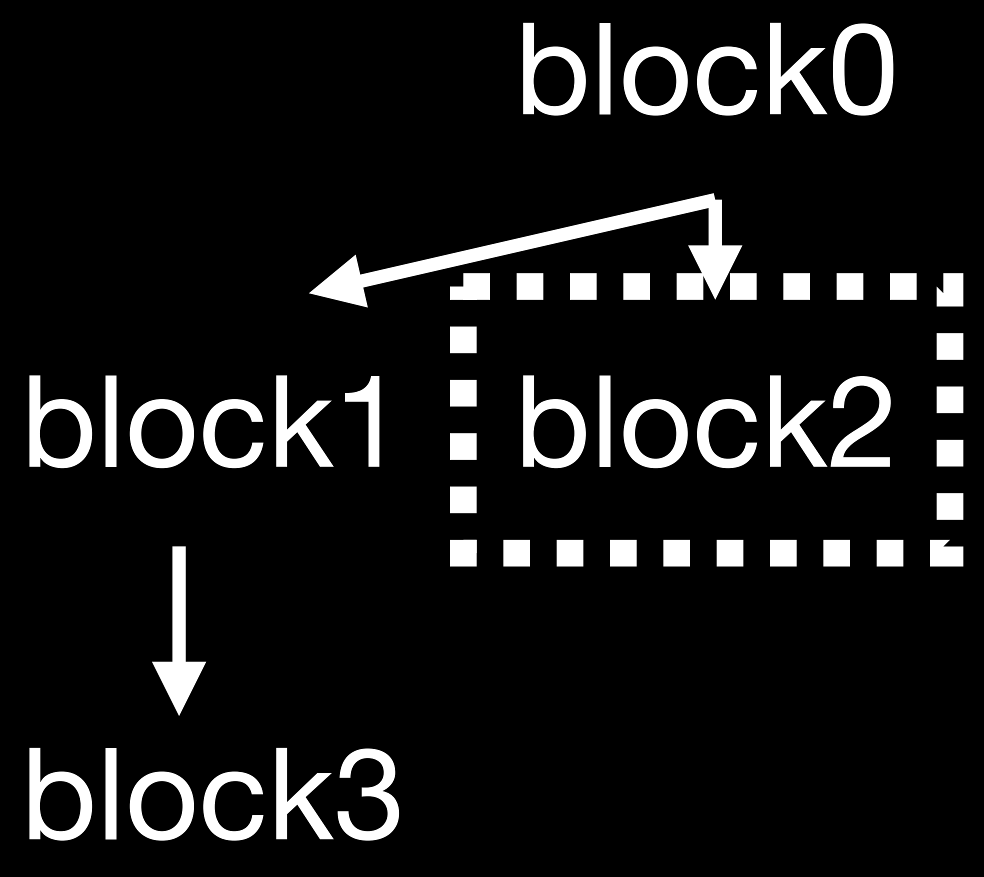




eclass	elaborated
ec0	v0
ec1	v1
ec2	v3

Scoped Elaboration

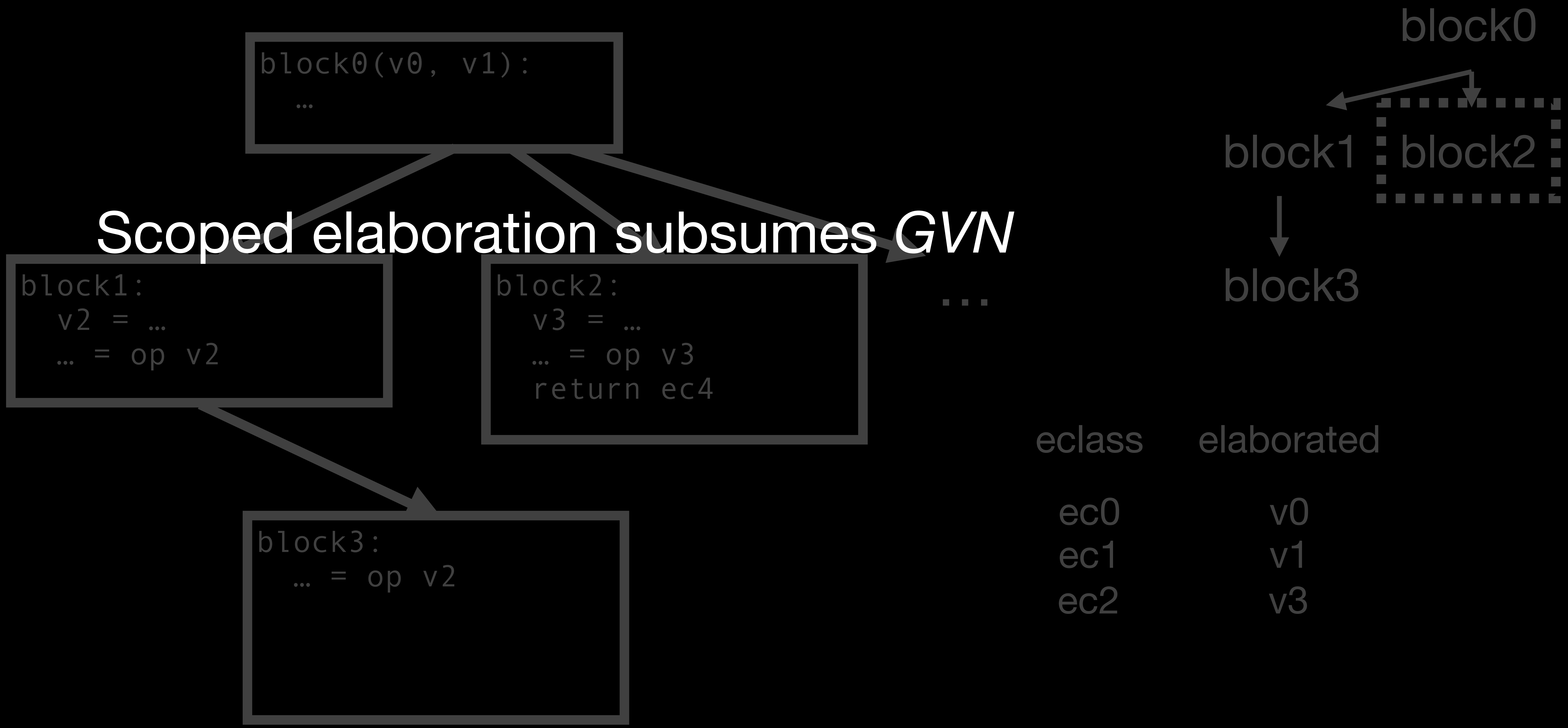




eclass	elaborated
ec0	v0
ec1	v1
ec2	v3

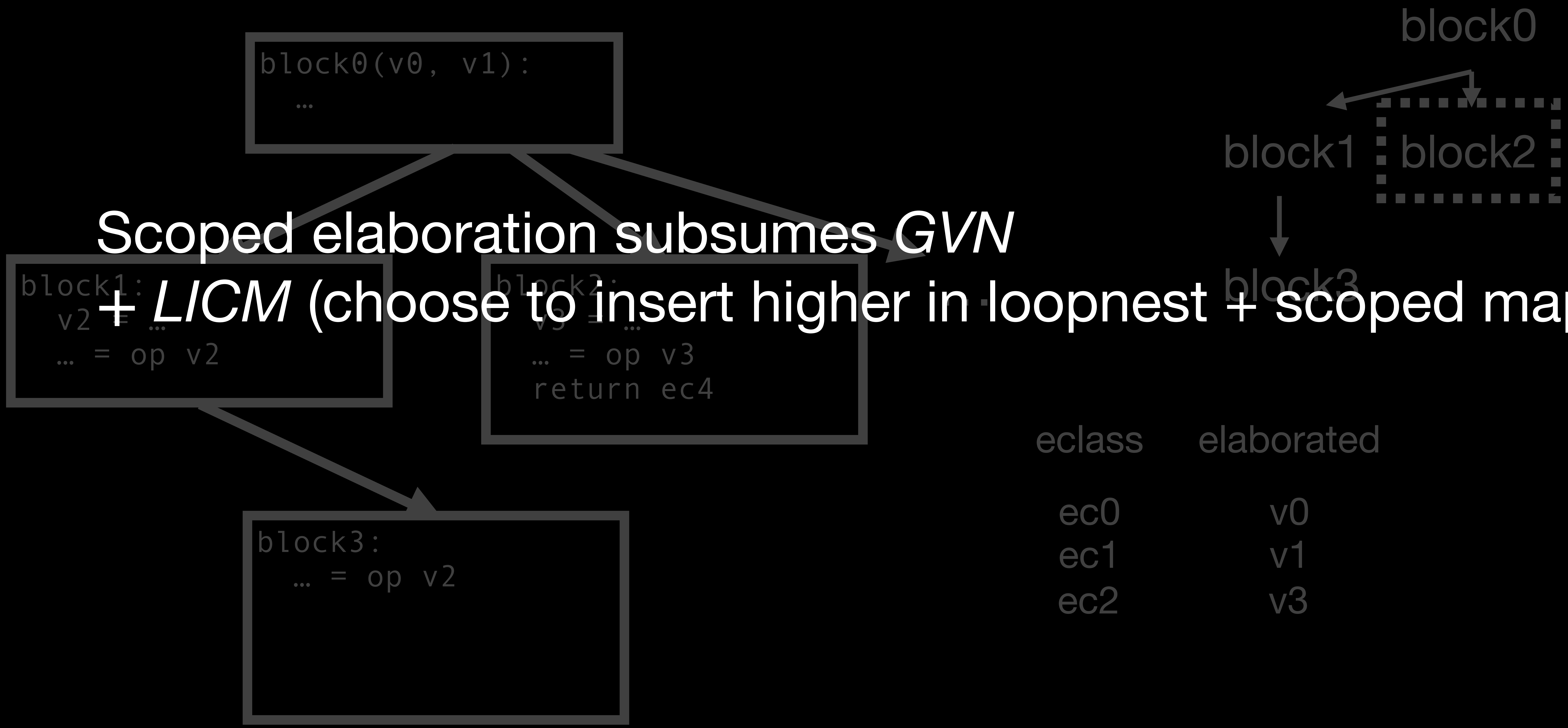
Scoped Elaboration

Scoped elaboration subsumes *GVN*

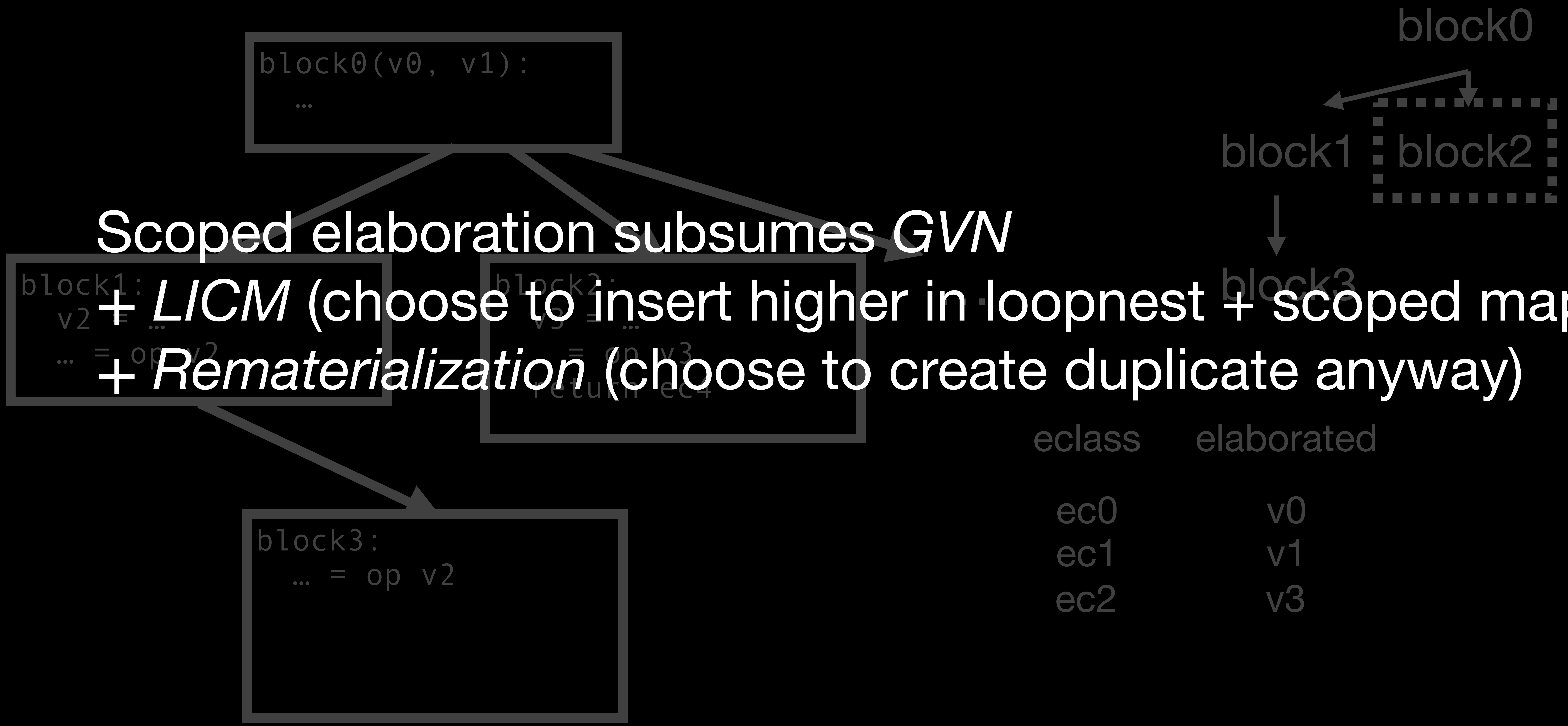


Scoped Elaboration

Scoped elaboration subsumes *GVN*
+ *LICM* (choose to insert higher in loopnest + scoped map)



Scoped Elaboration



Scoped elaboration subsumes *GVN*
+ *LICM* (choose to insert higher in loopnest + scoped map)
+ *Rematerialization* (choose to create duplicate anyway)

1. **Why** *we want a rewrite-based optimizer*

2. **How** *to turn a CFG into an egraph and back again*

3. **Cycles** *why they occur, and what to do about them*

4. **Results** *how well does it work?*

5. **Lessons** *in translating research to production*

1. Why *we want a rewrite-based optimizer*

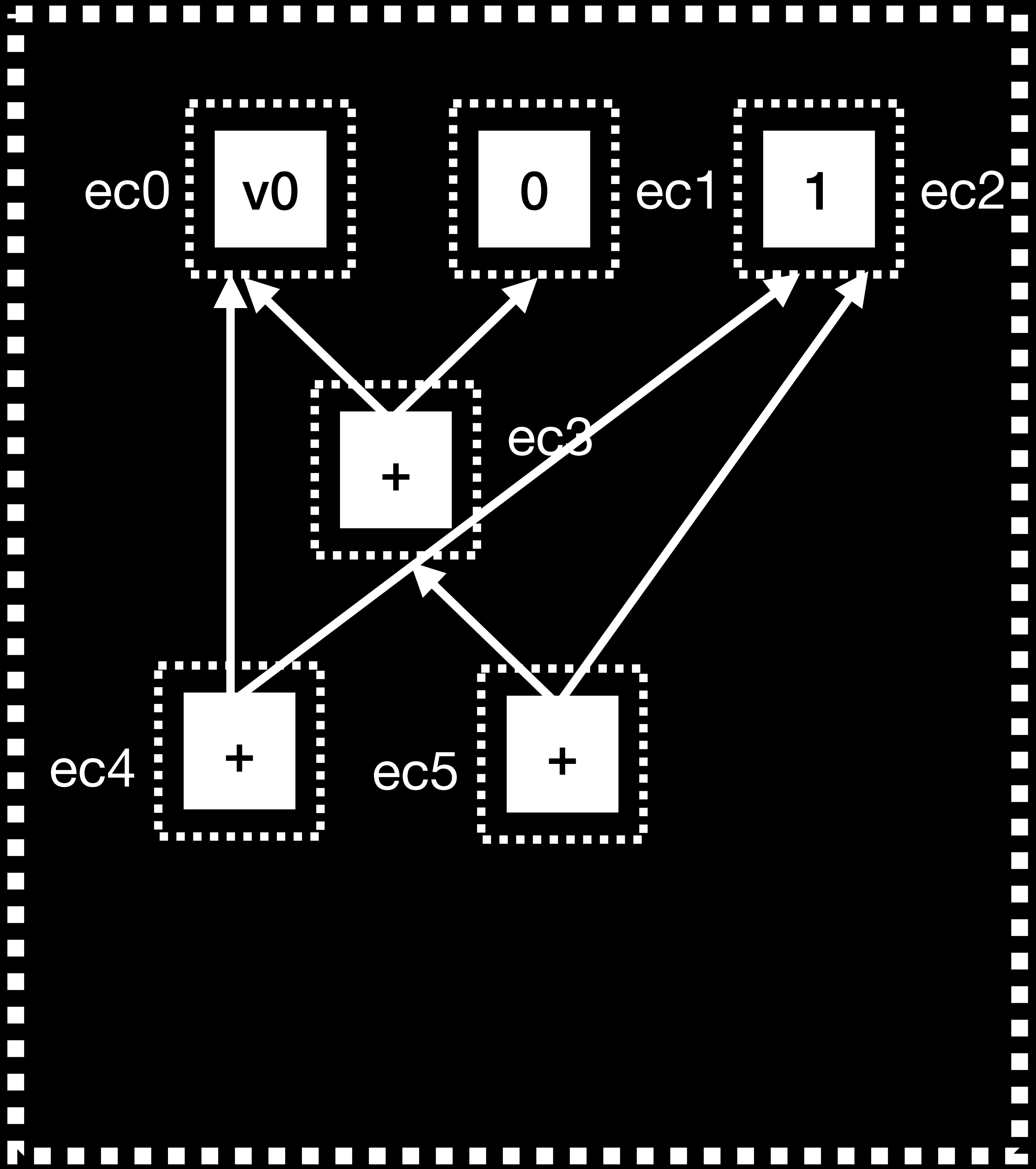
2. **How** *to turn a CFG into an egraph and back again*

3. Cycles *why they occur, and what to do about them*

4. Results *how well does it work?*

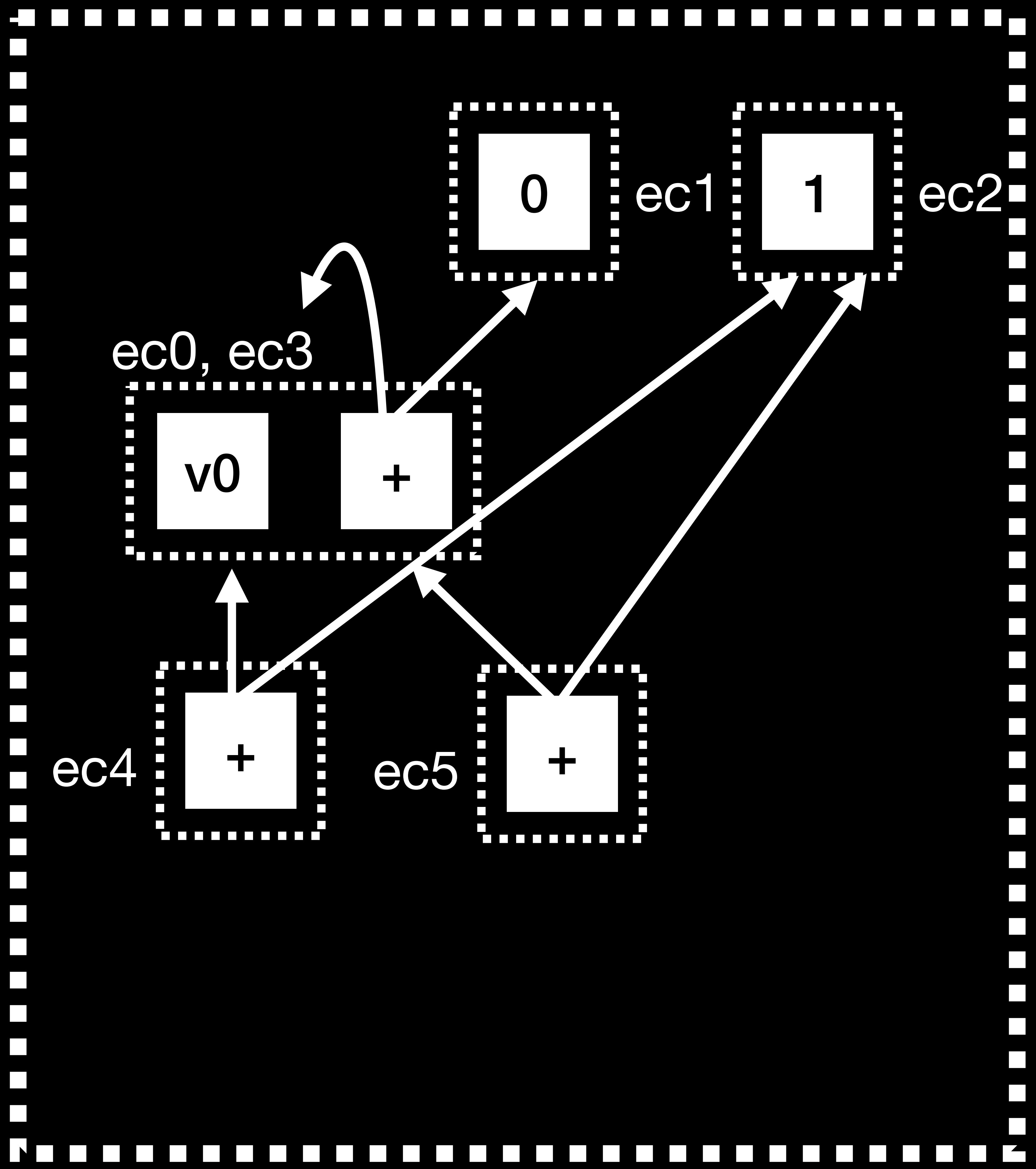
5. Lessons *in translating research to production*

Rewrites and Repair



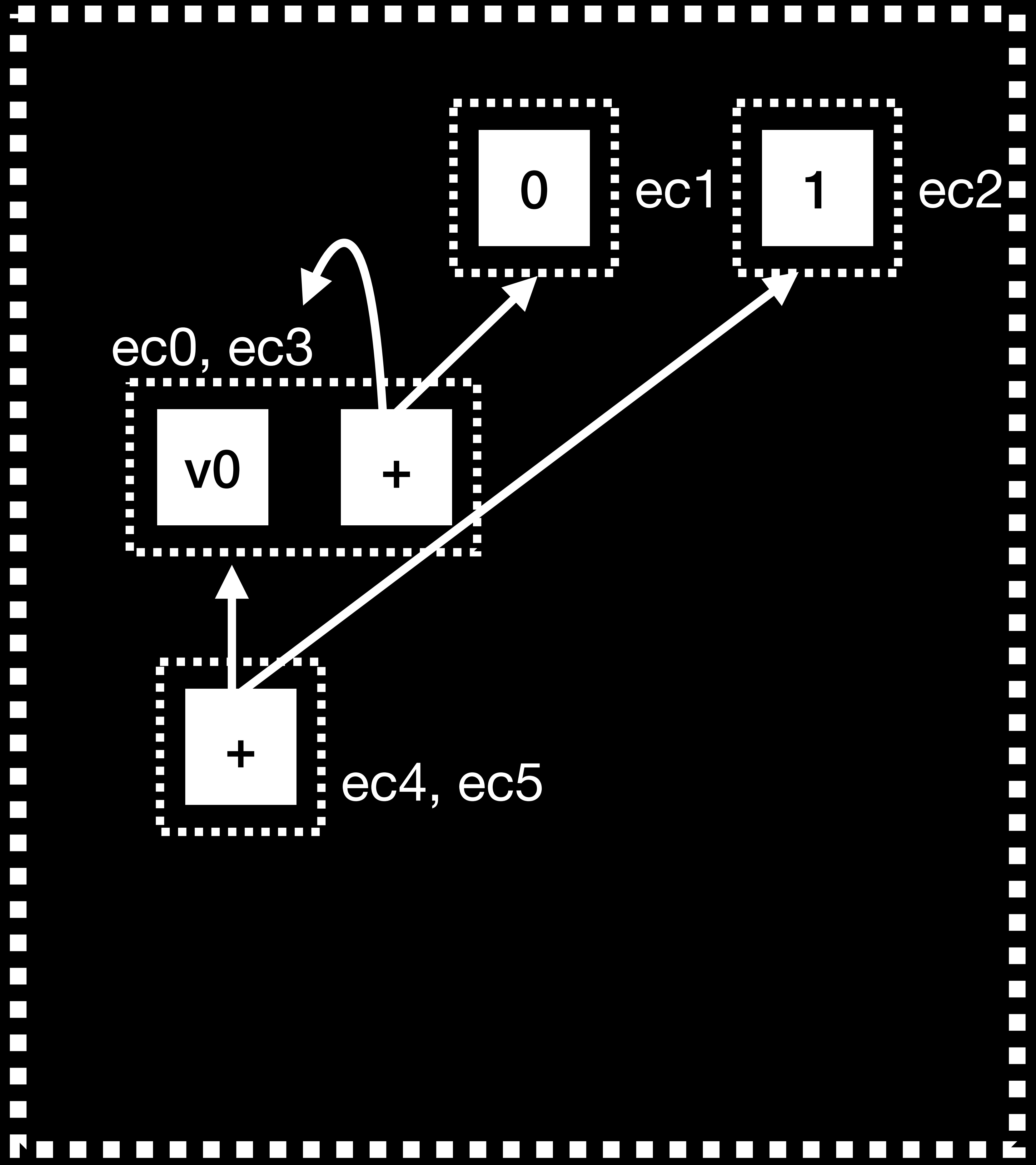
Rewrite: x + 0 => x

Rewrites and Repair



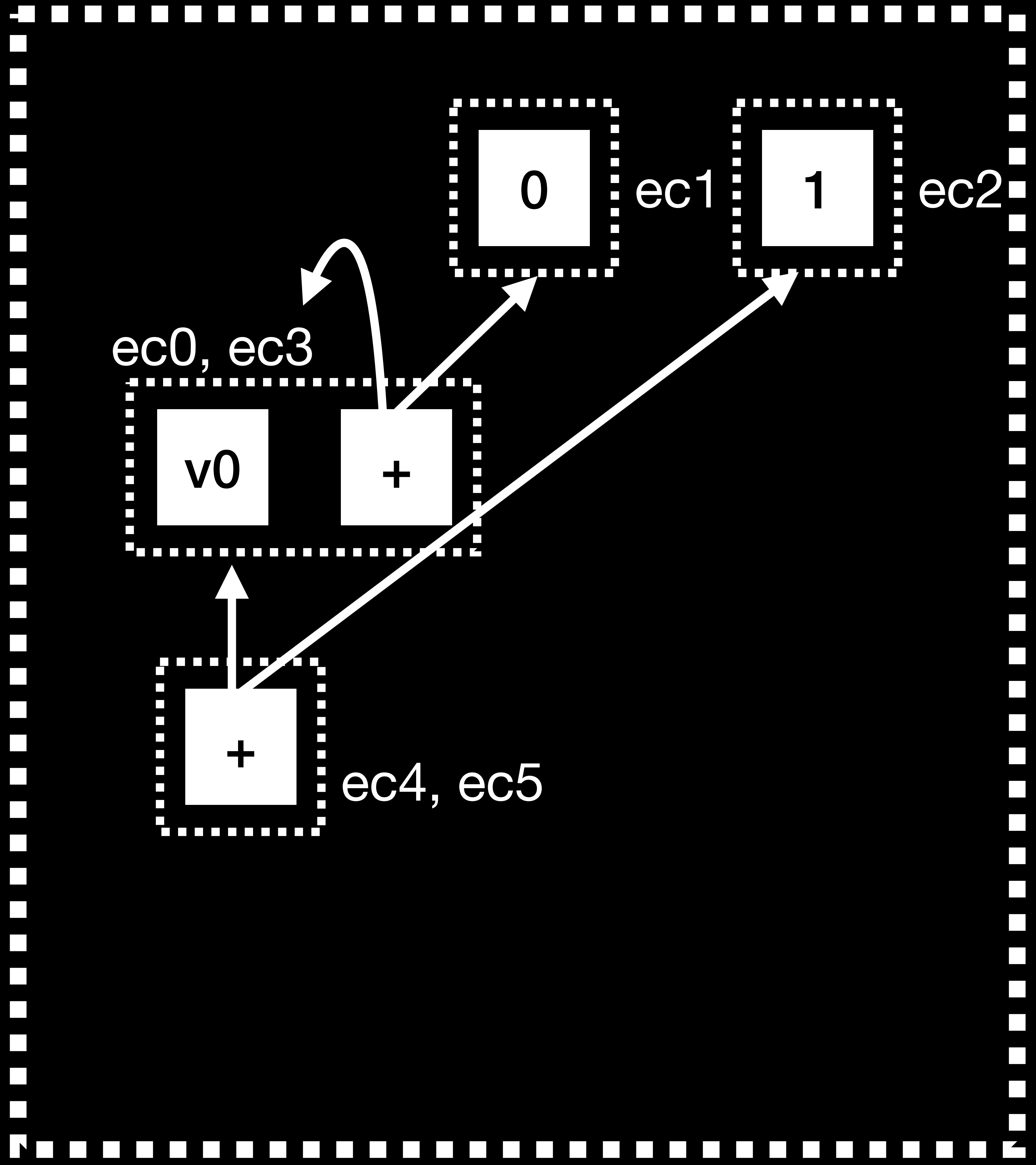
Rewrite: x + 0 => x

Rewrites and Repair



Rewrite: x + 0 => x

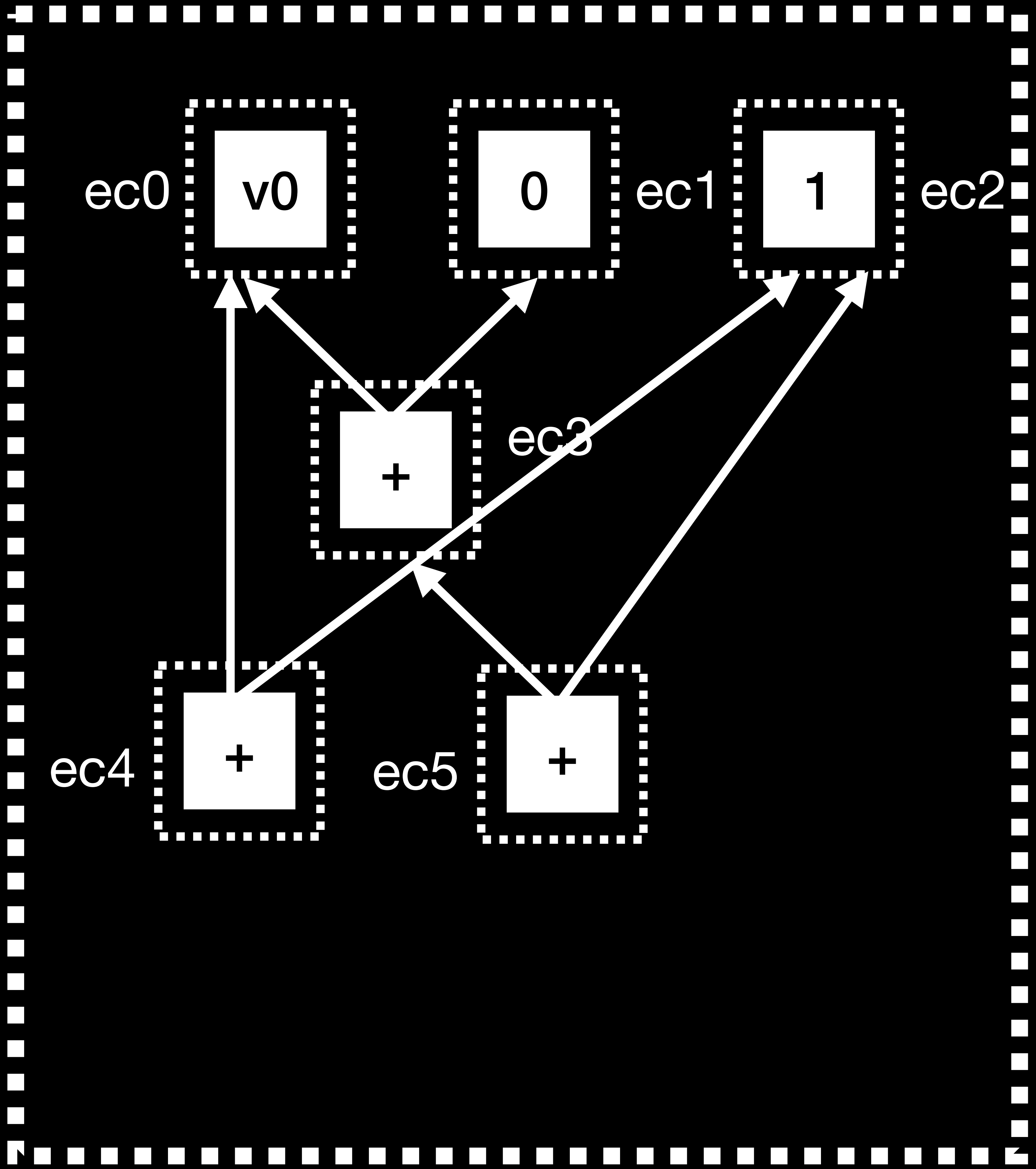
Rewrites and Repair



Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

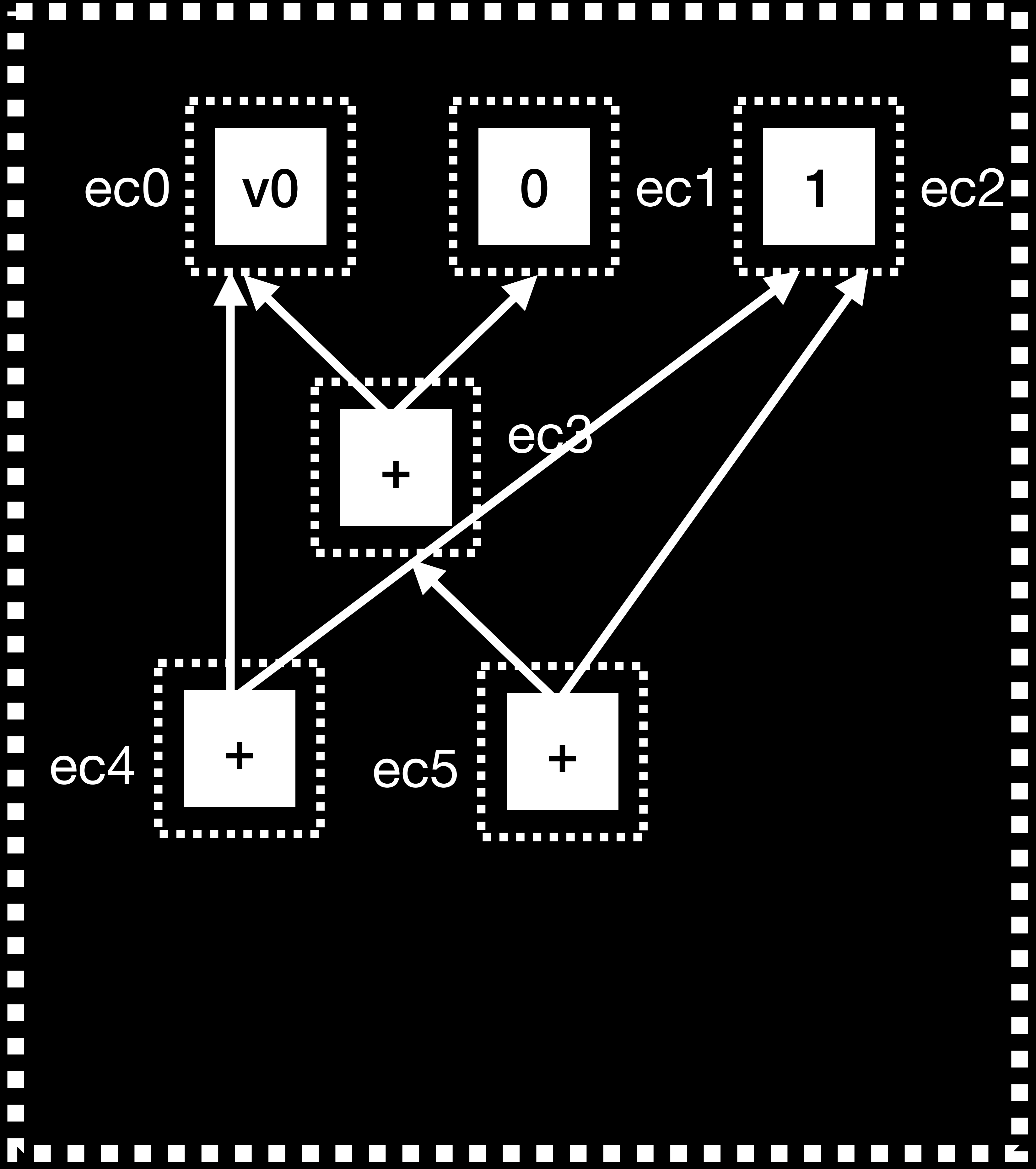
Rewrites and Repair



Rewrite: x + 0 => x

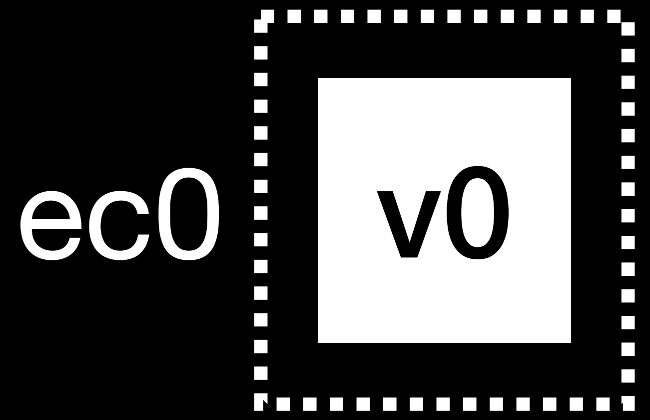
Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

Rewrites and Repair

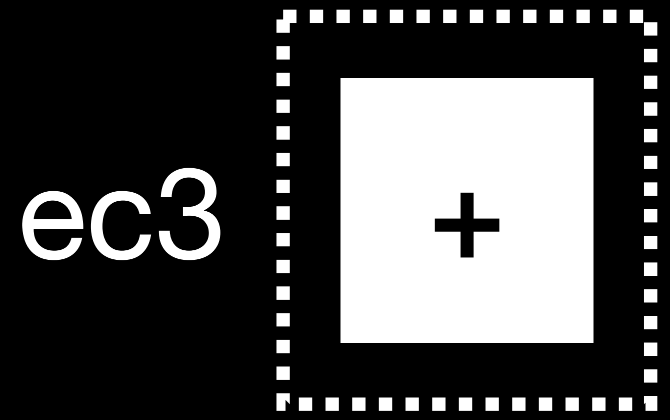


Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

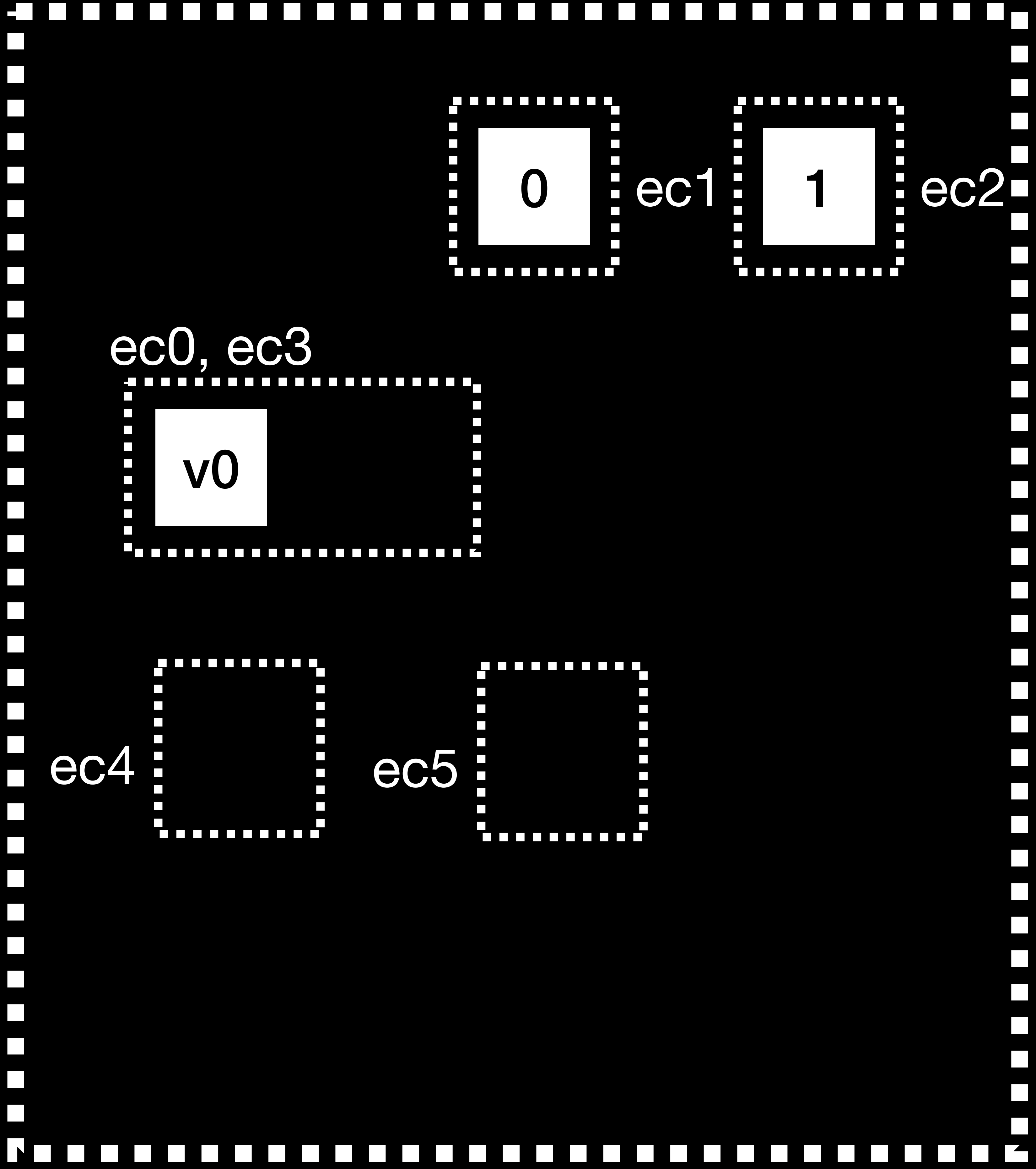


Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2)}



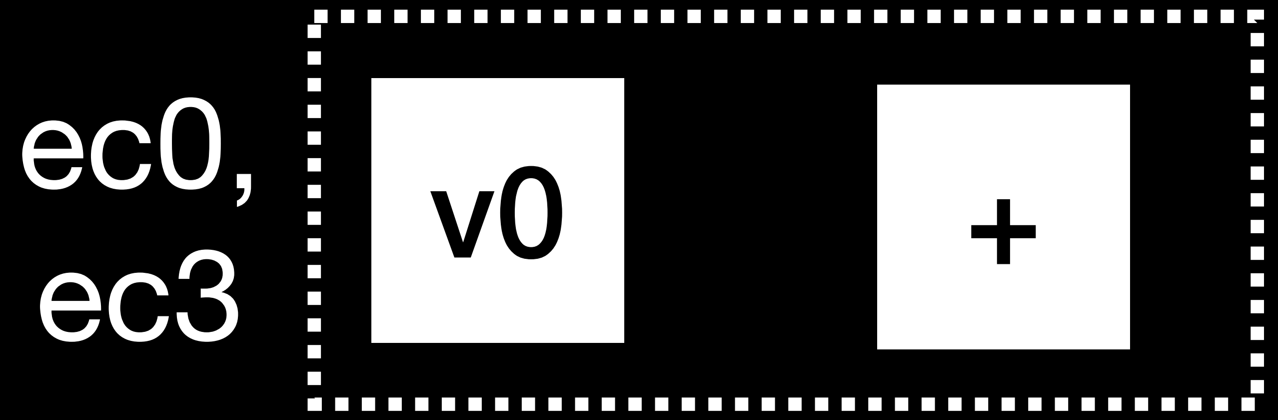
Parents:
{ec5 := (+ ec3 ec2)}

Rewrites and Repair



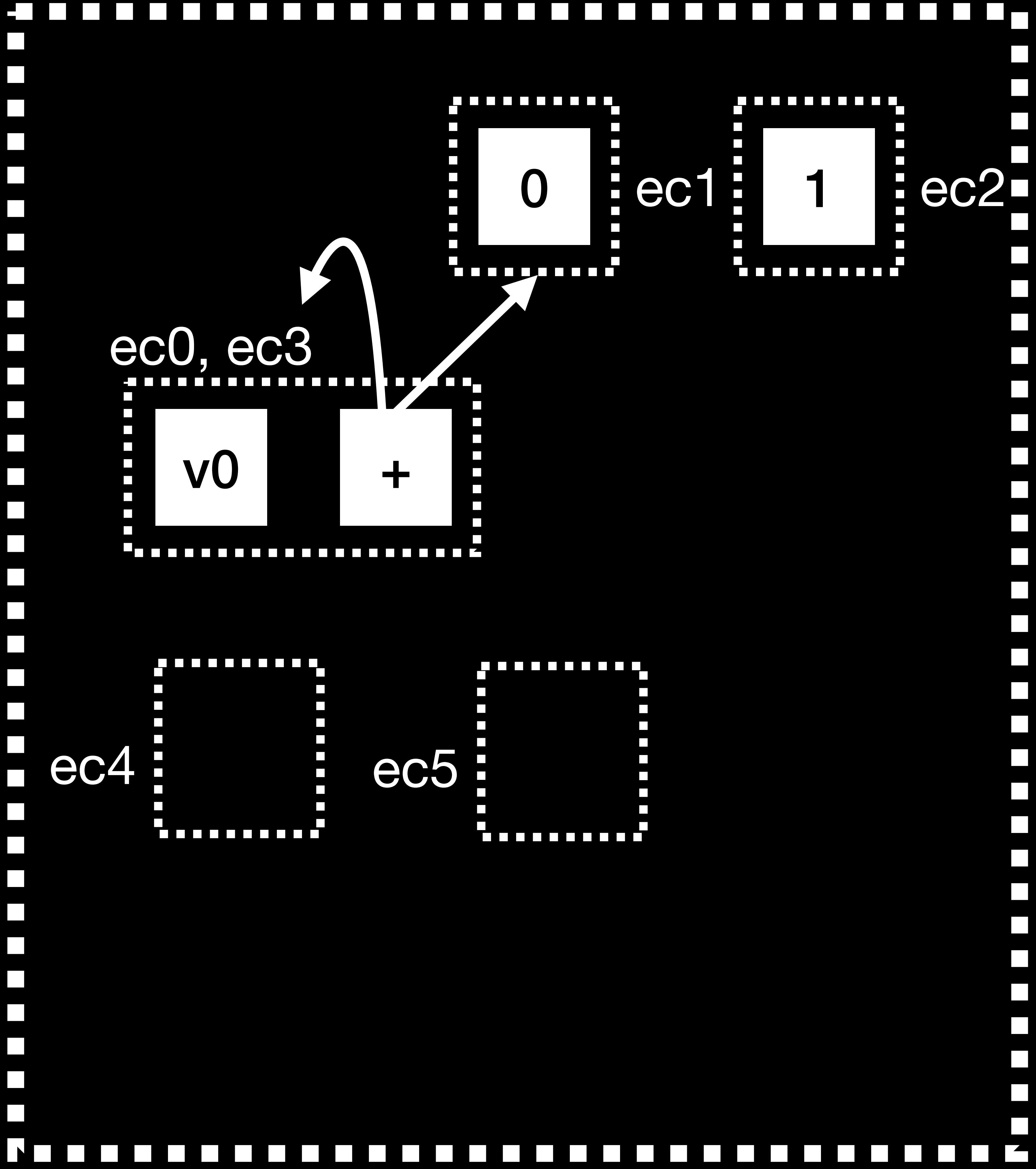
Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*



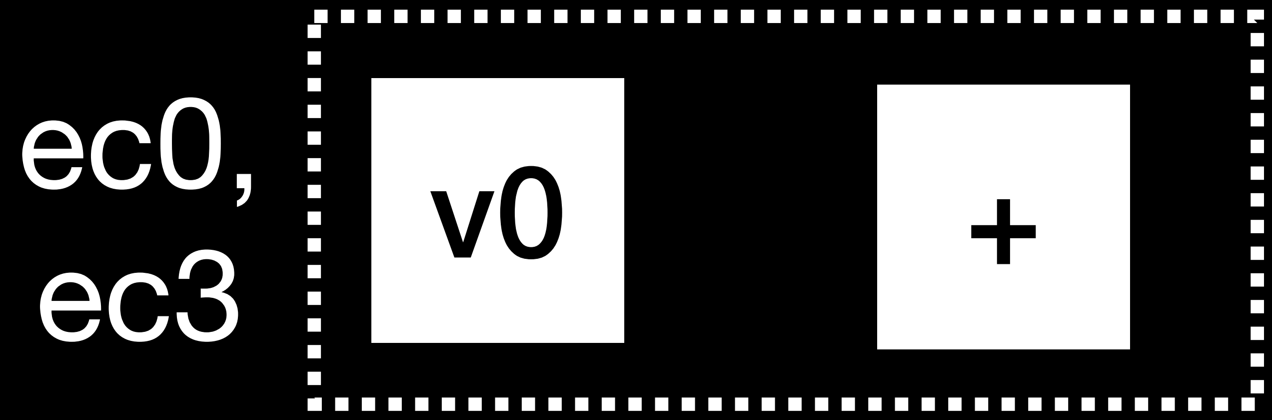
Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2),
ec5 := (+ ec3 ec2)}

Rewrites and Repair



Rewrite: x + 0 => x

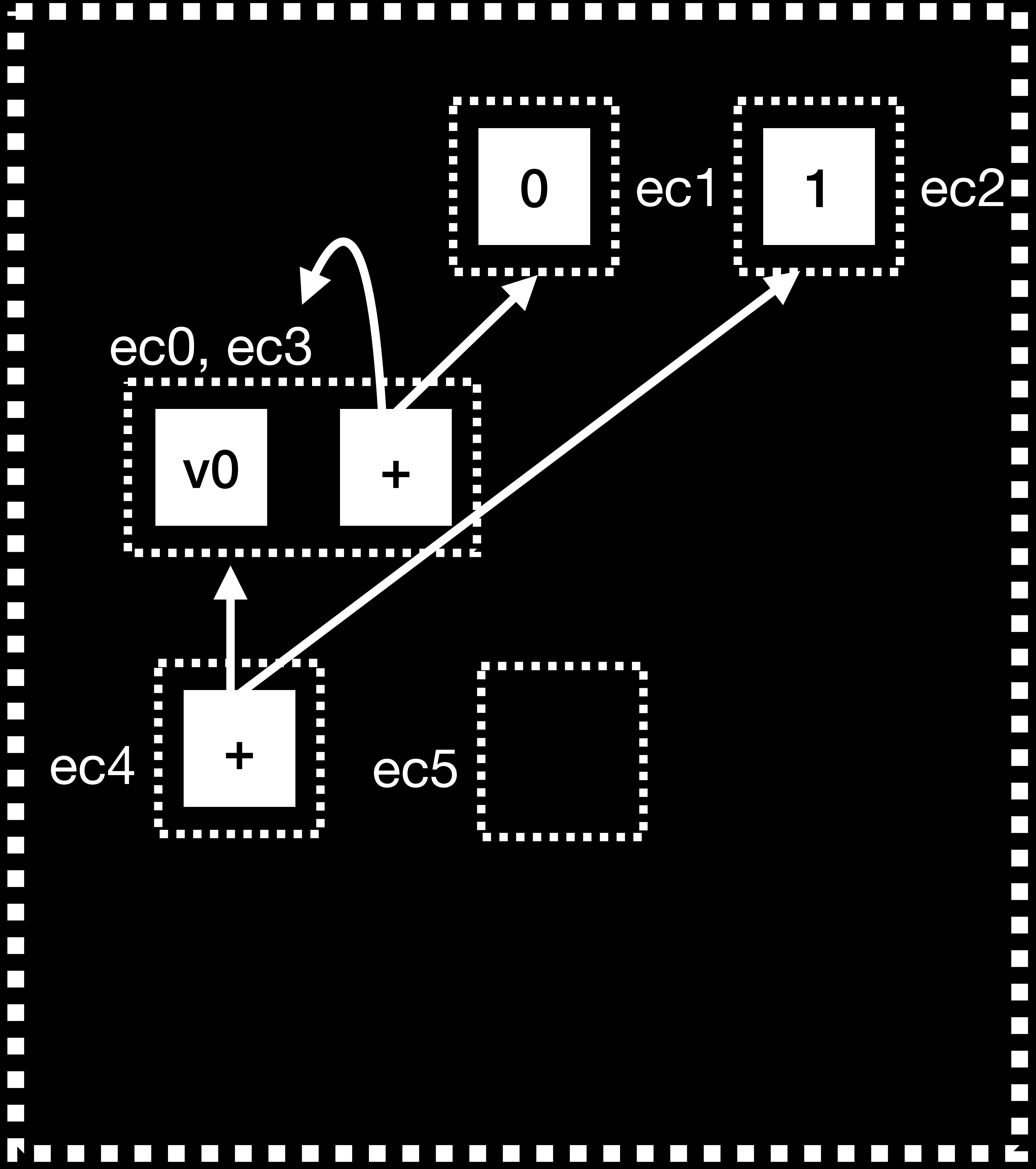
Fixup requires backlinks (parent pointers) and re-interning, which are *costly*



Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2),
ec5 := (+ ec3 ec2)}

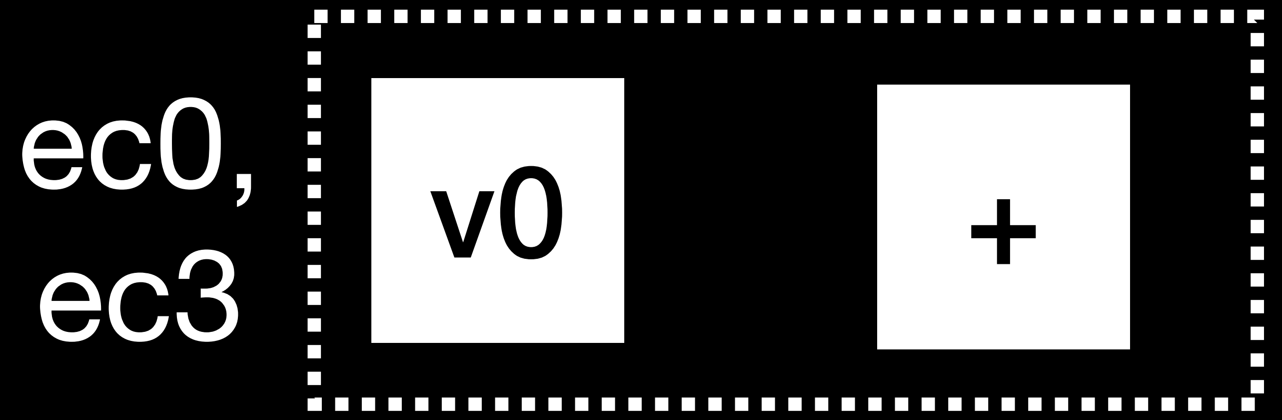
Re-intern
ec3 := (+ ec0 ec1)

Rewrites and Repair



Rewrite: x + 0 => x

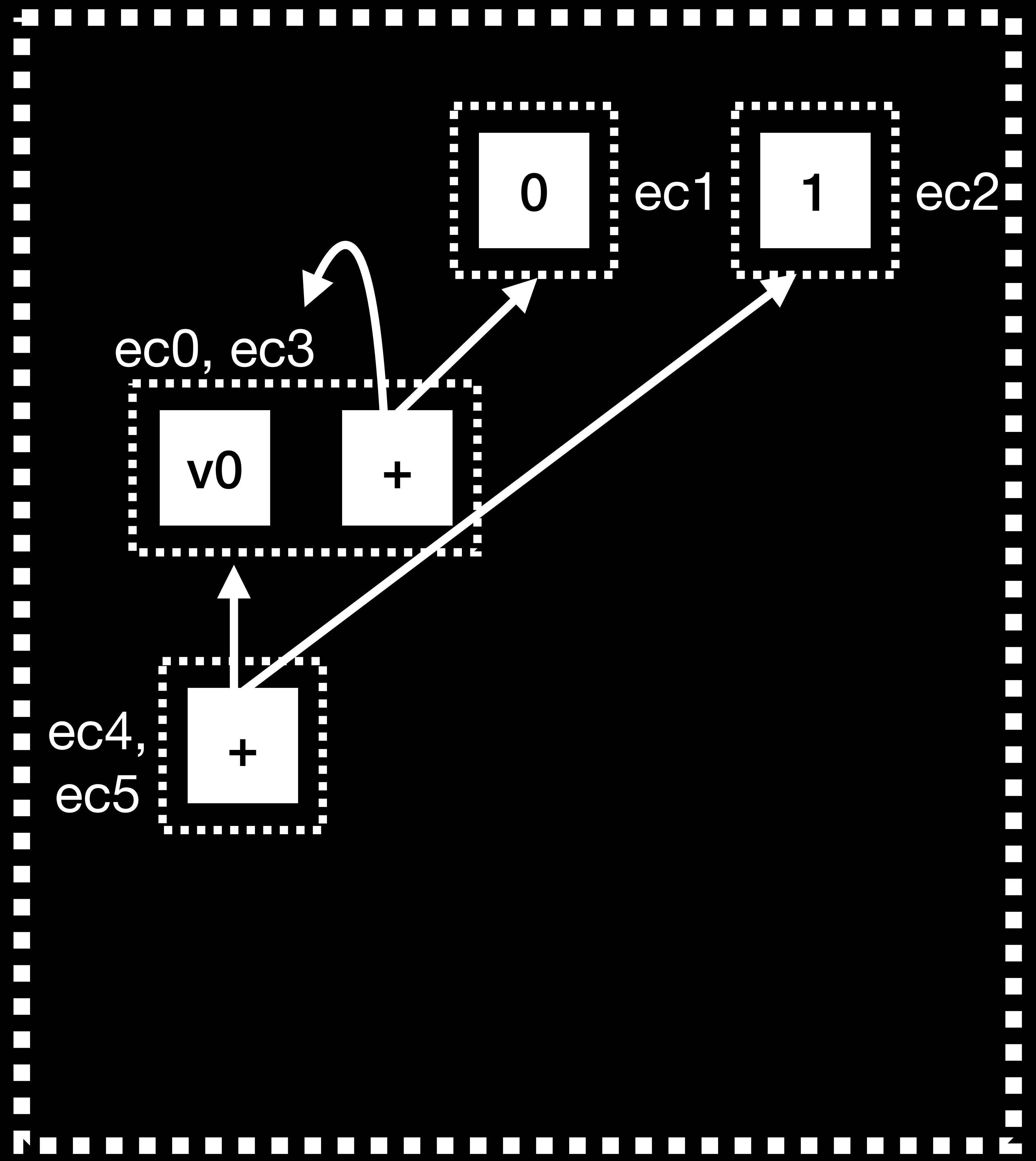
Fixup requires backlinks (parent pointers) and re-interning, which are *costly*



Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2),
ec5 := (+ ec3 ec2)}

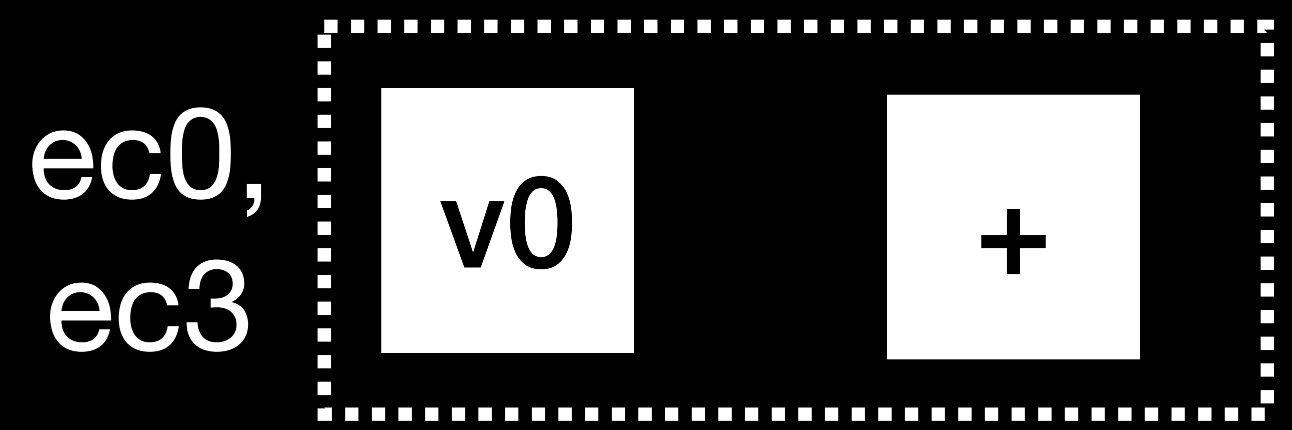
Re-intern
ec4 := (+ ec0 ec2)

Rewrites and Repair



Rewrite: x + 0 => x

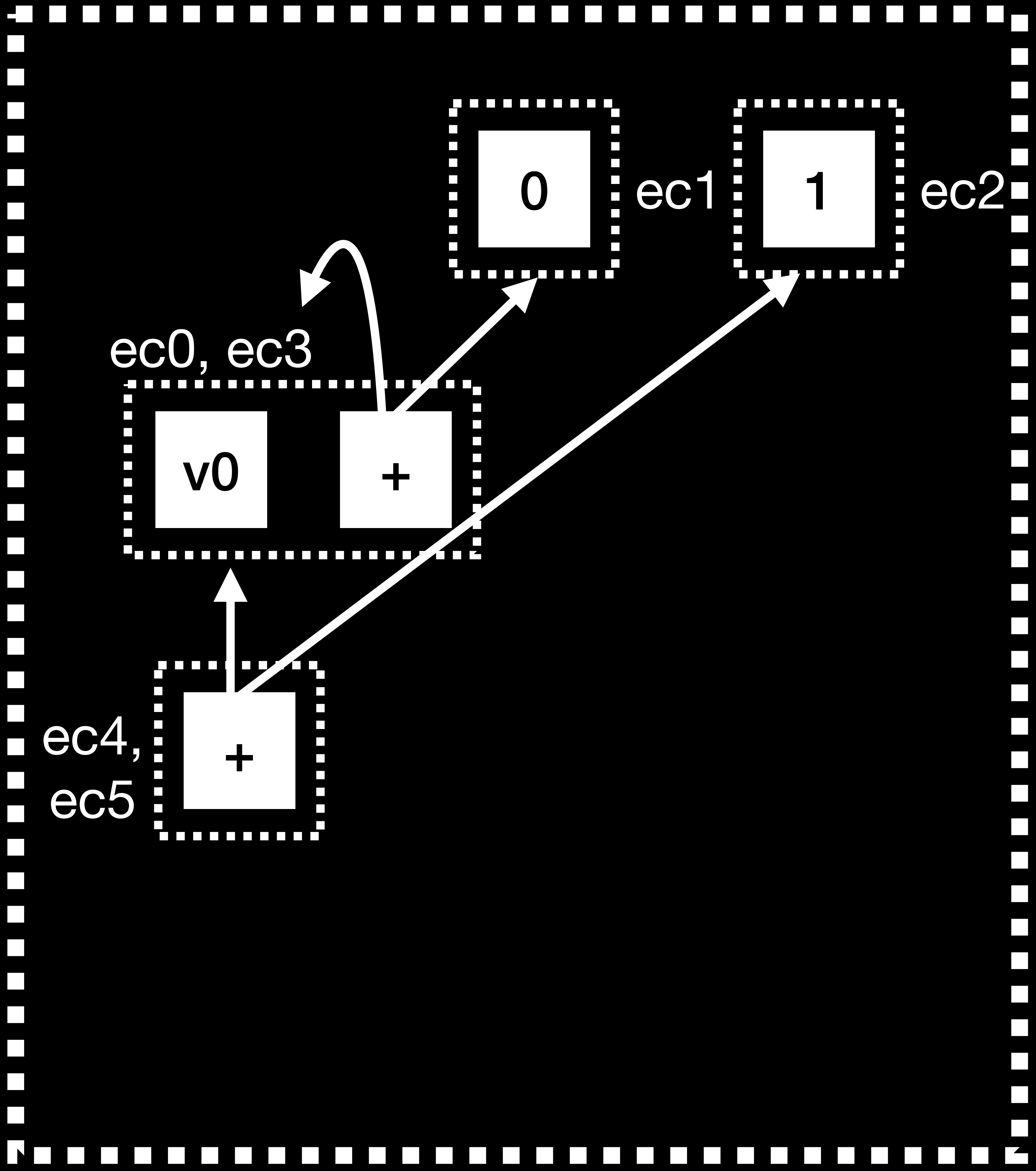
Fixup requires backlinks (parent pointers) and re-interning, which are *costly*



Parents:
{ec3 := (+ ec0, ec1),
ec4 := (+ ec0, ec2),
ec5 := (+ ec3 ec2)}

Re-intern
ec5 := (+ ec3 ec2)
= (+ ec0 ec2)
= ec4

Rewrites and Repair

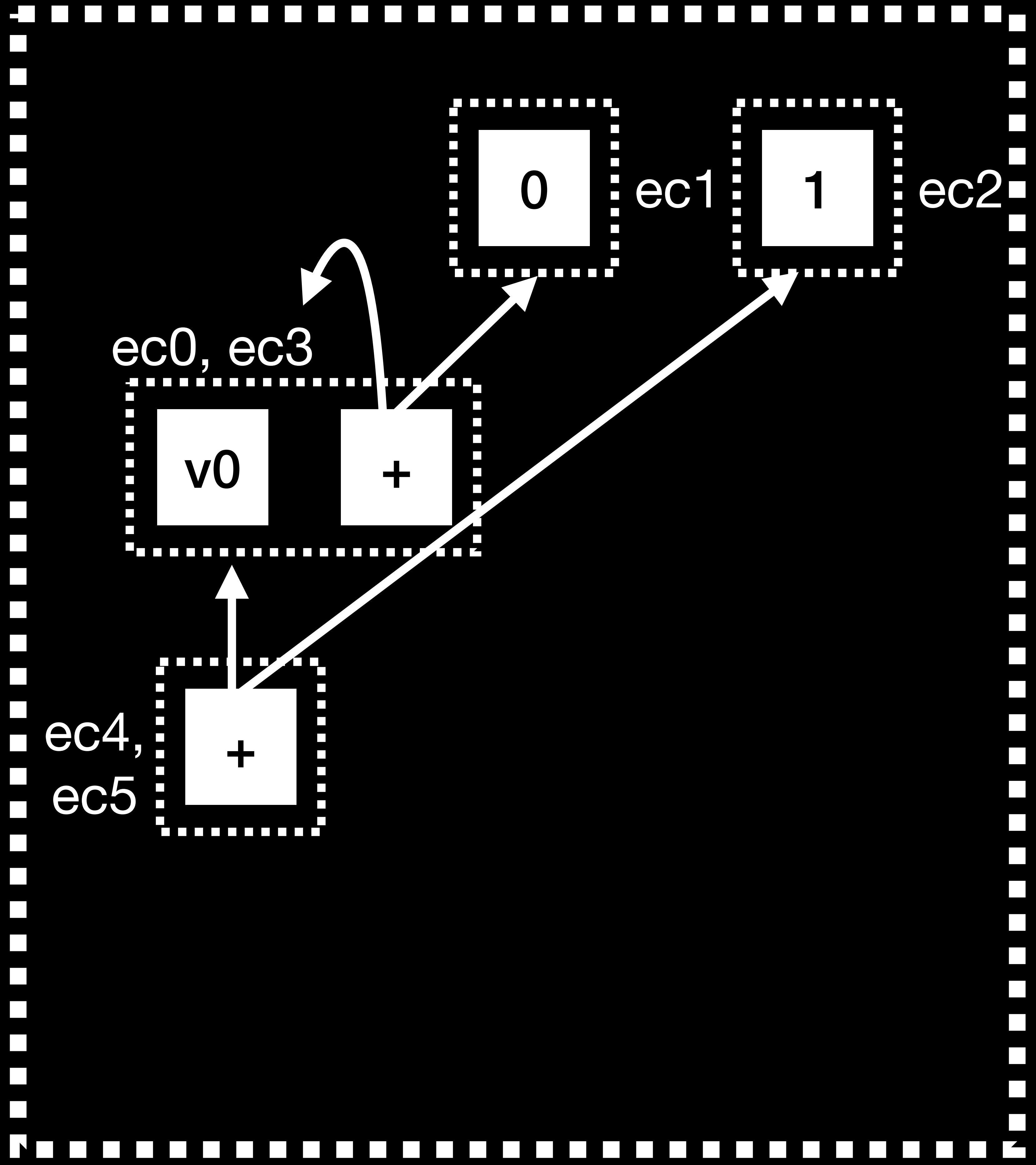


Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

Eliminate:
- Parent lists?
- Duplicated storage of nodes?
- Merging of parent lists, with dedup’ing?

Rewrites and Repair



Rewrite: x + 0 => x

Fixup requires backlinks (parent pointers) and re-interning, which are *costly*

Eliminate:
- Parent lists?
- Duplicated storage of nodes?
- Merging of parent lists, with dedup'ing?

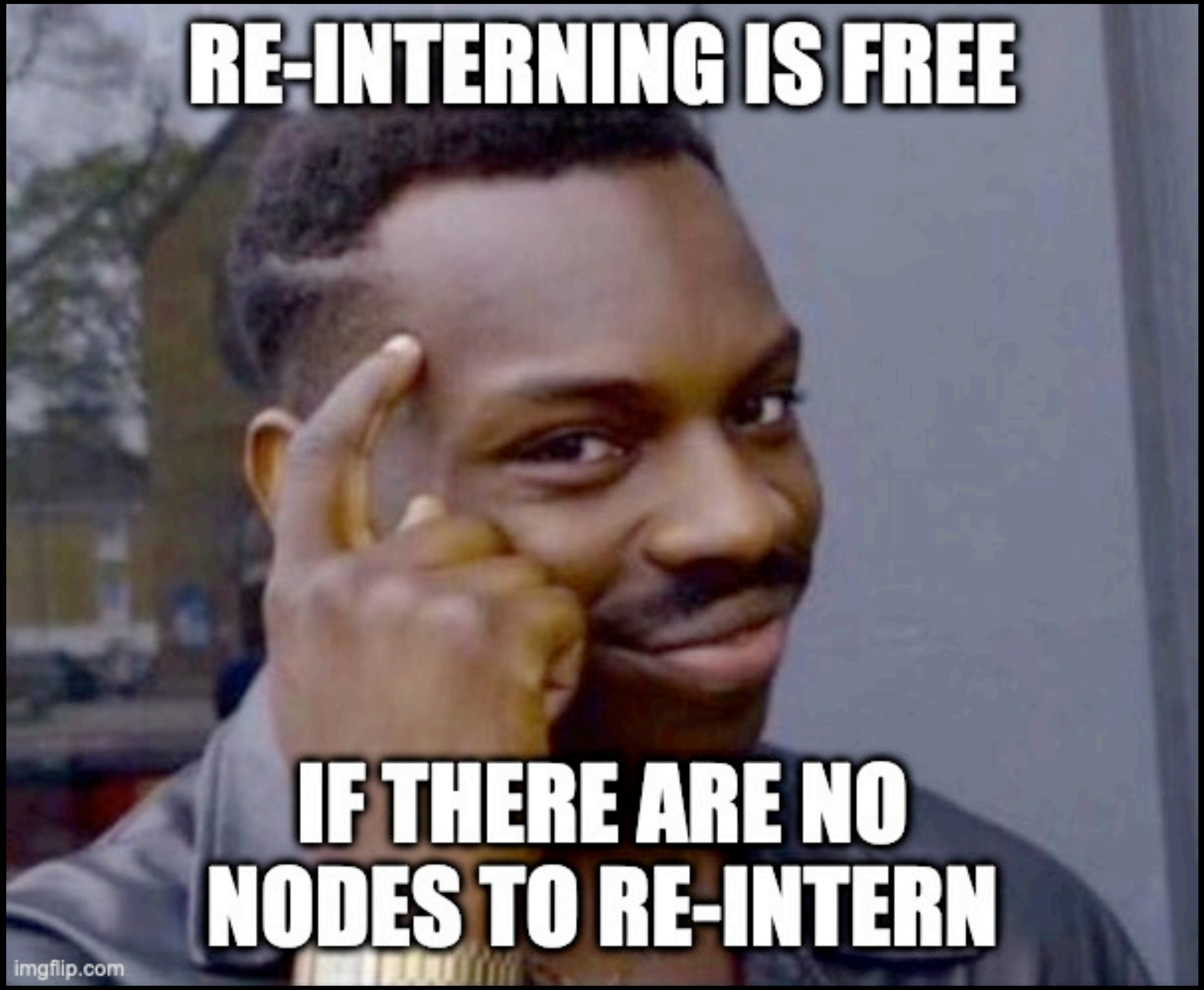
—> compile + memory overhead too high
vs. traditional compiler pipeline

Rewrites and Repair

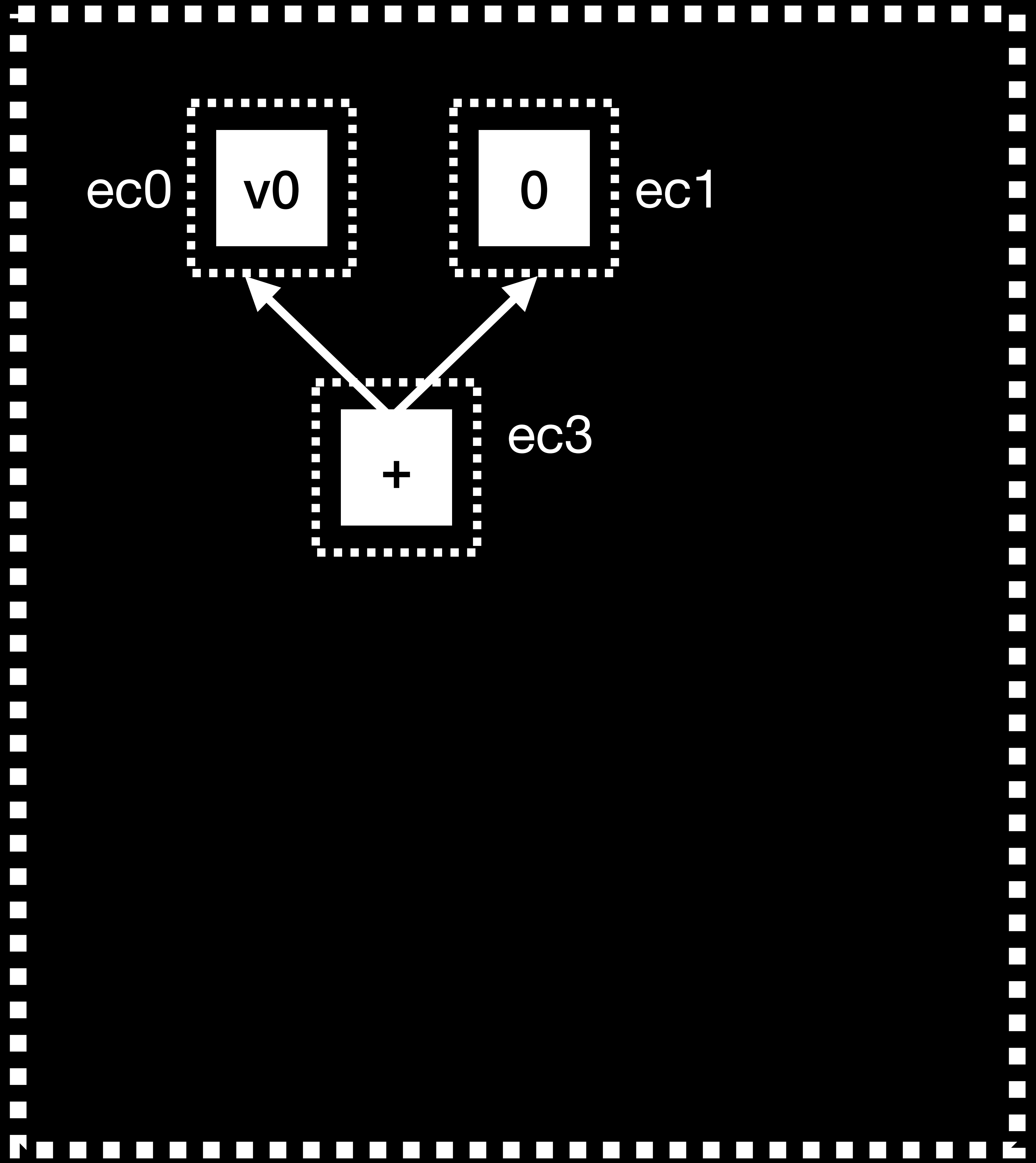
Idea: no need to repair uses if eclass is in *final form* before we use it!

Rewrites and Repair

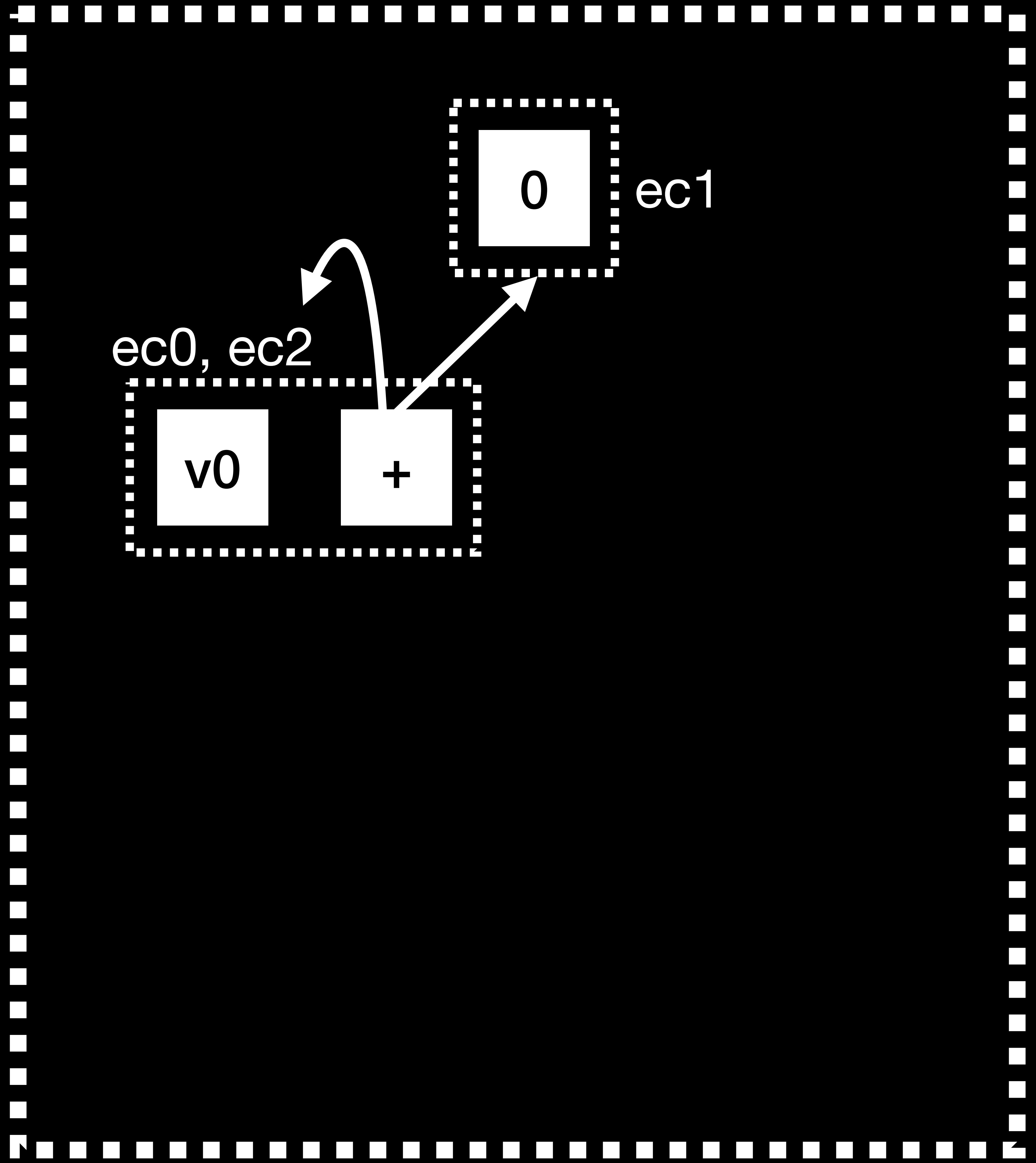
Idea: no need to repair uses if eclass is in *final form* before we use it!



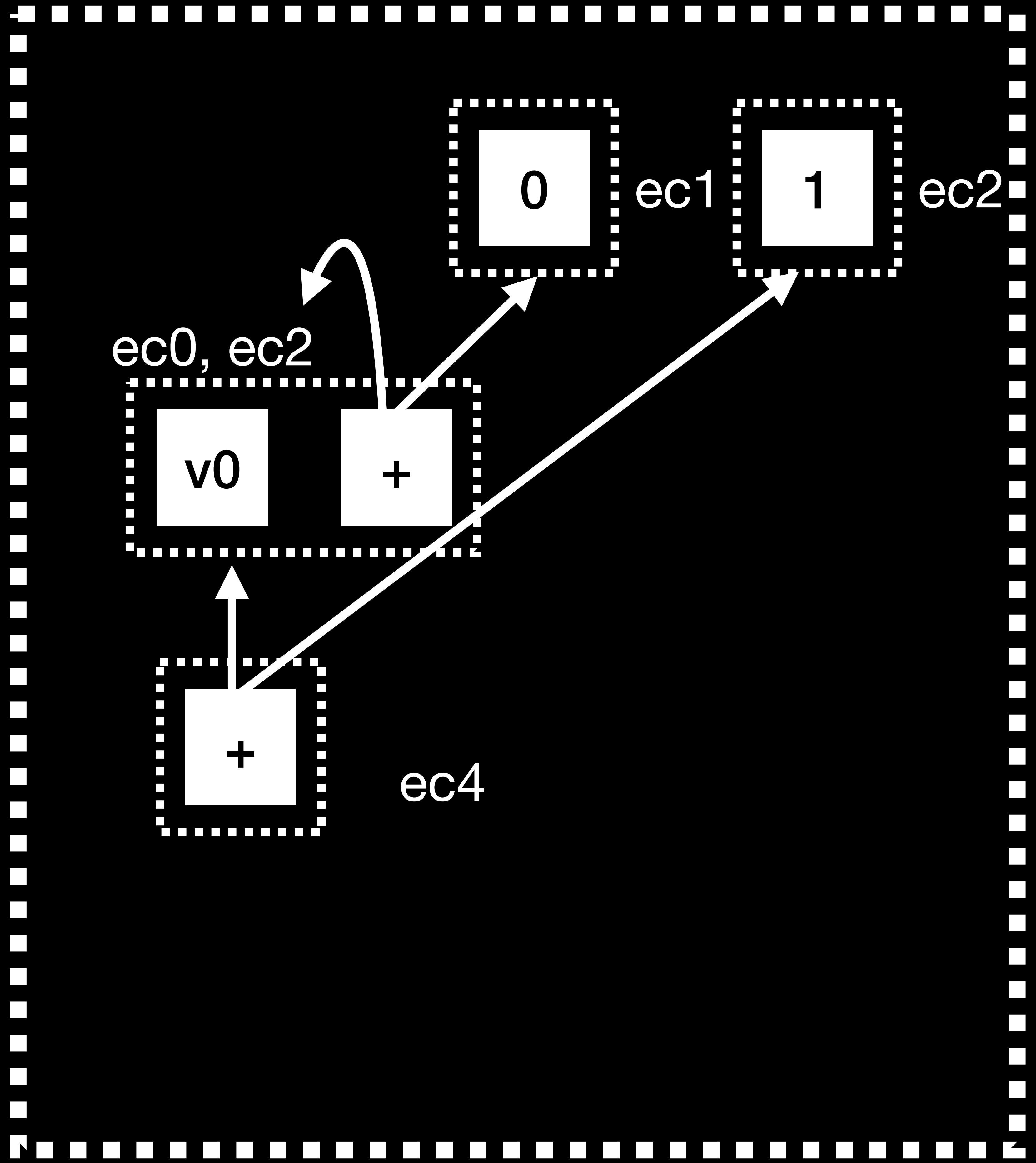
Rewrite eagerly?!



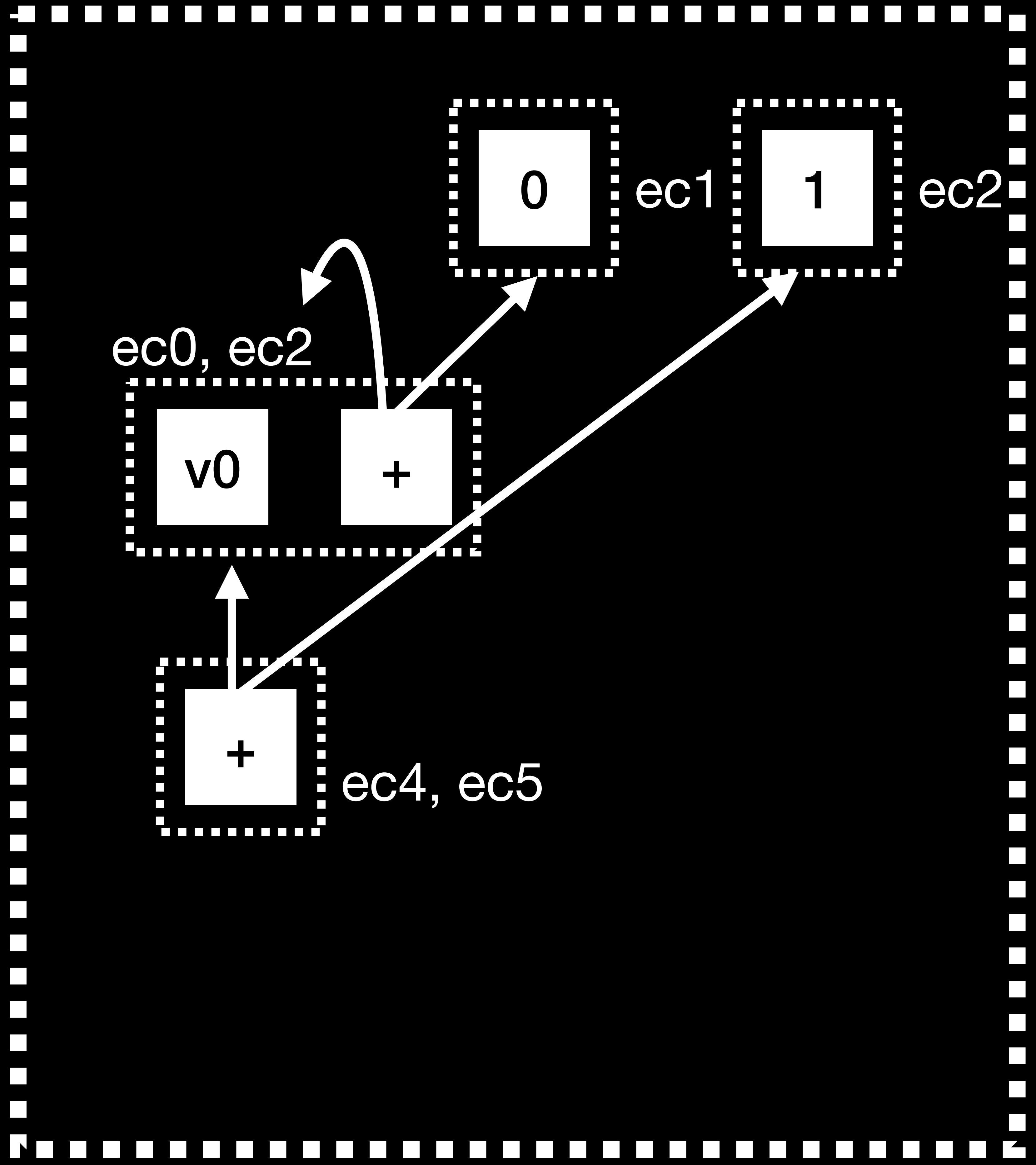
Rewrite eagerly?!



Rewrite eagerly?!

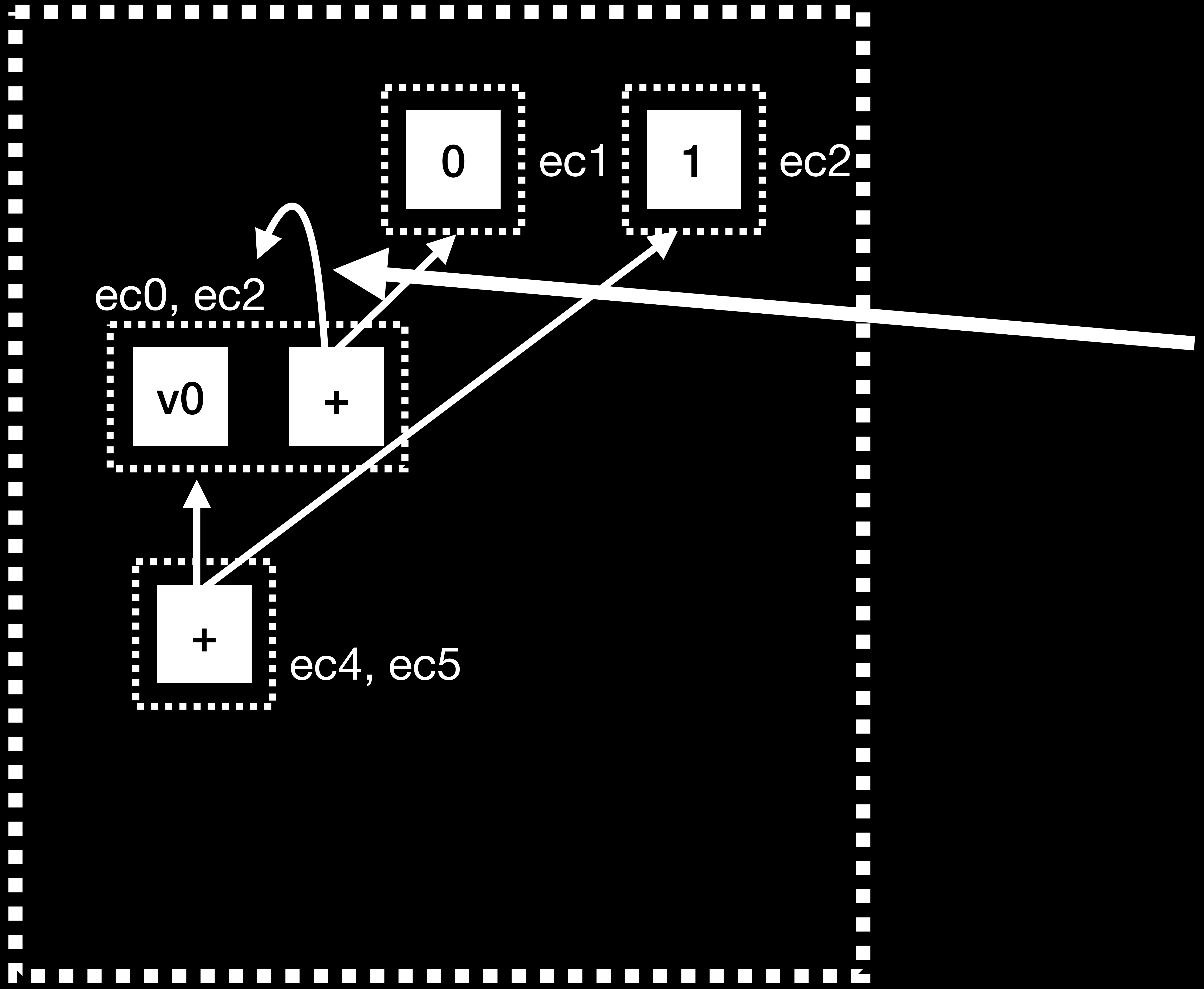


Rewrite eagerly?!



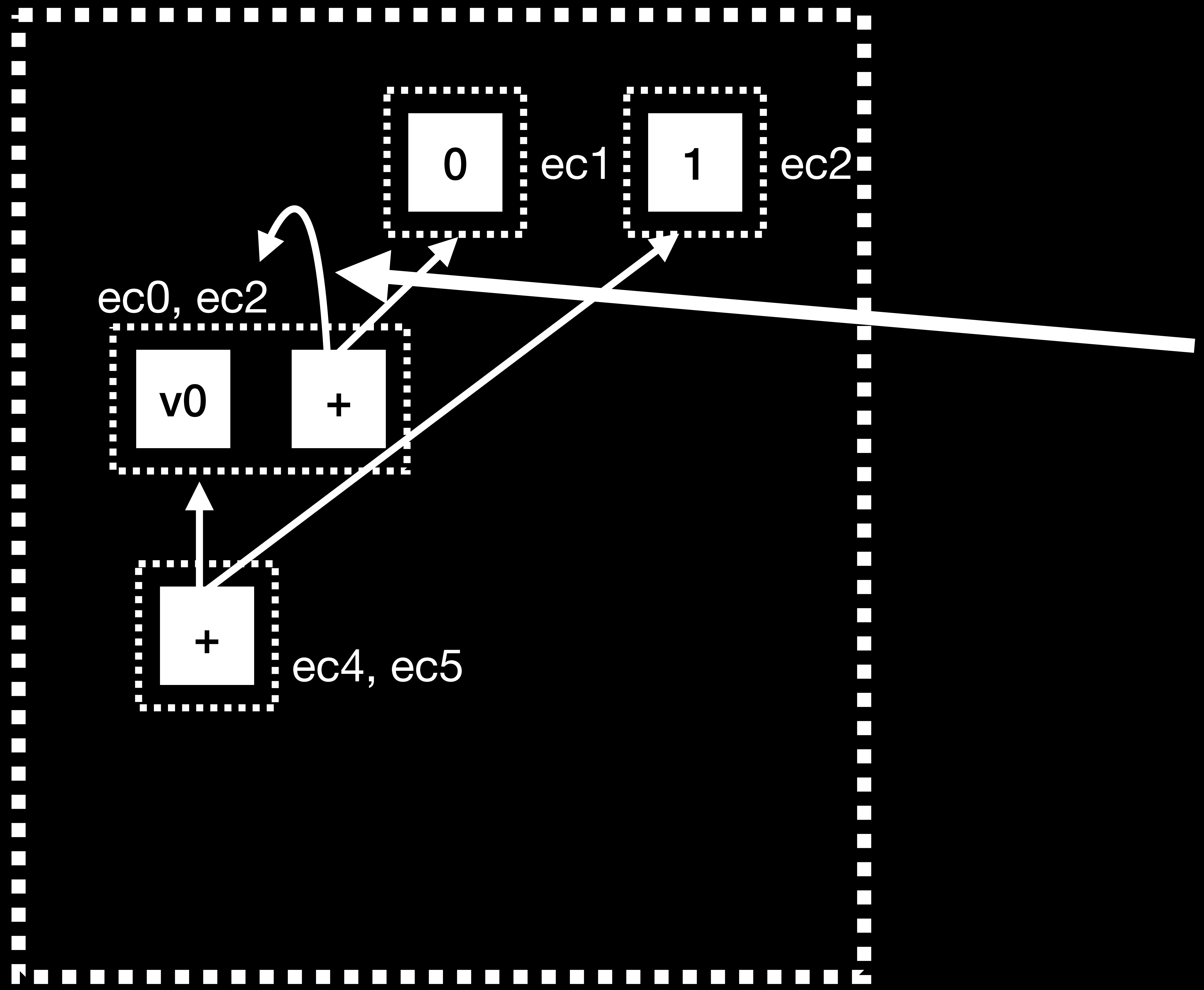
Rewrite already occurred
→ ec5 hash-conses to ec4

Rewrite eagerly?!



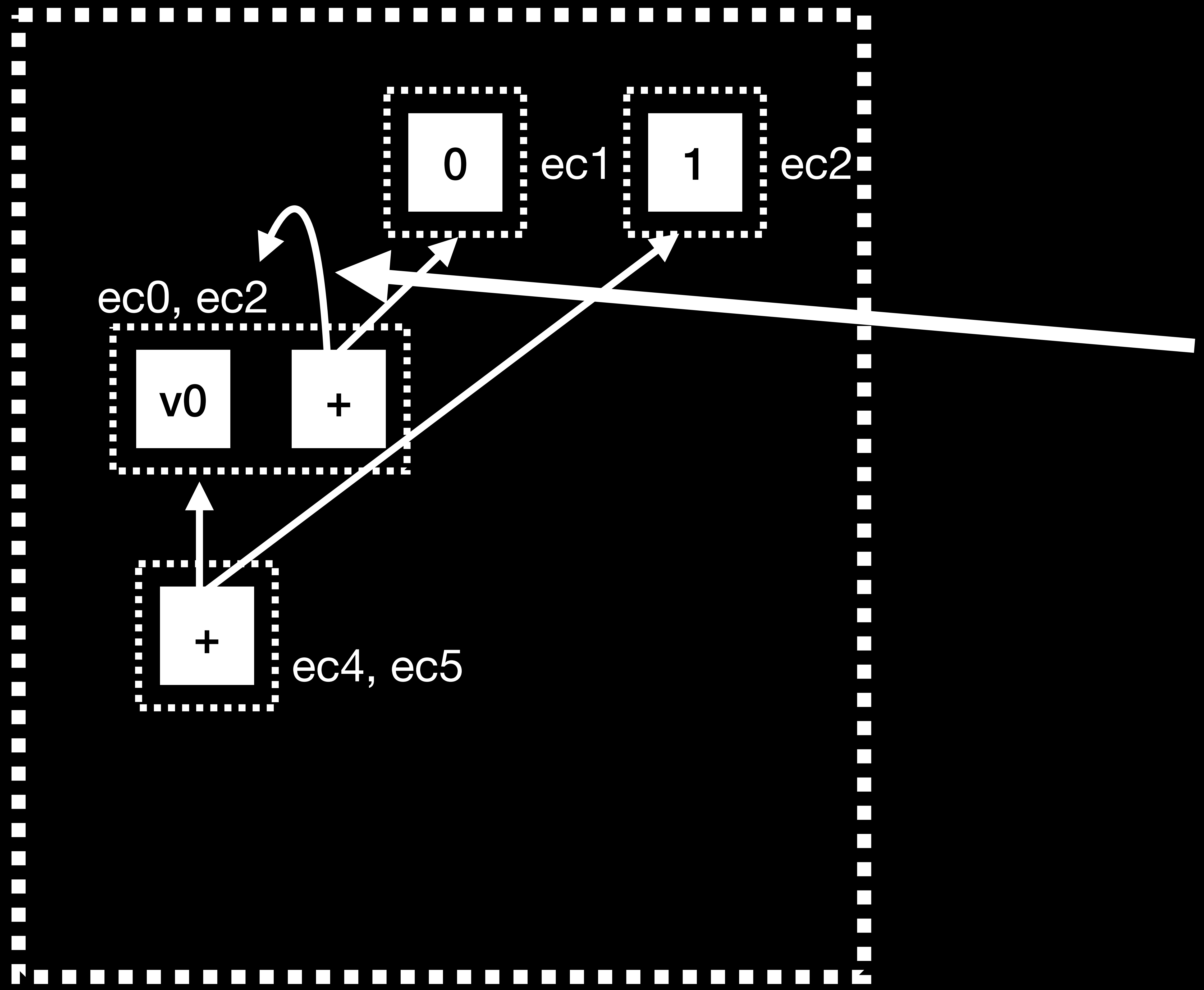
How do we handle this cycle?

Rewrite eagerly?!



How do we handle this cycle?
- Cycles preclude single pass (imply fixpoint algorithm)

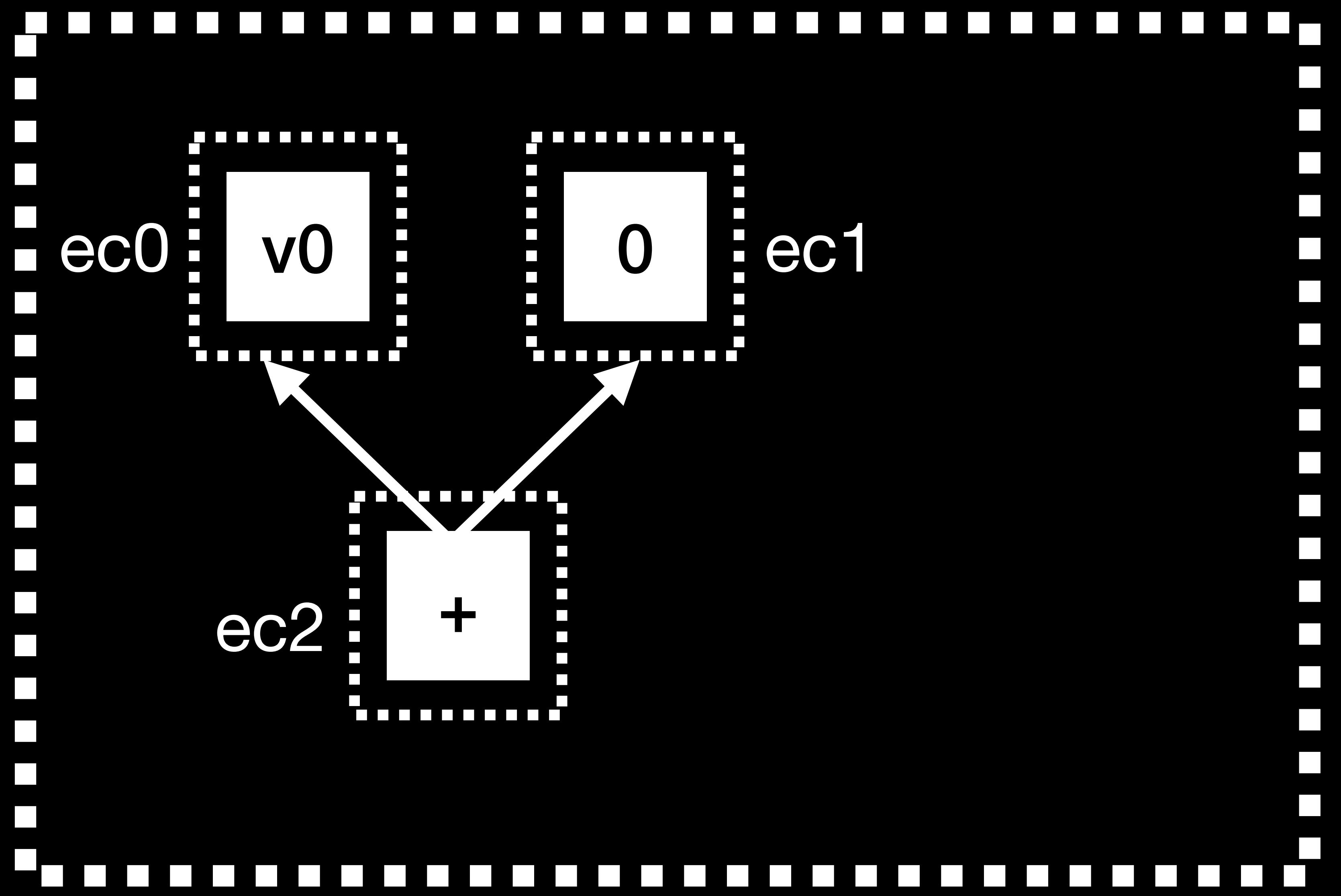
Rewrite eagerly?!



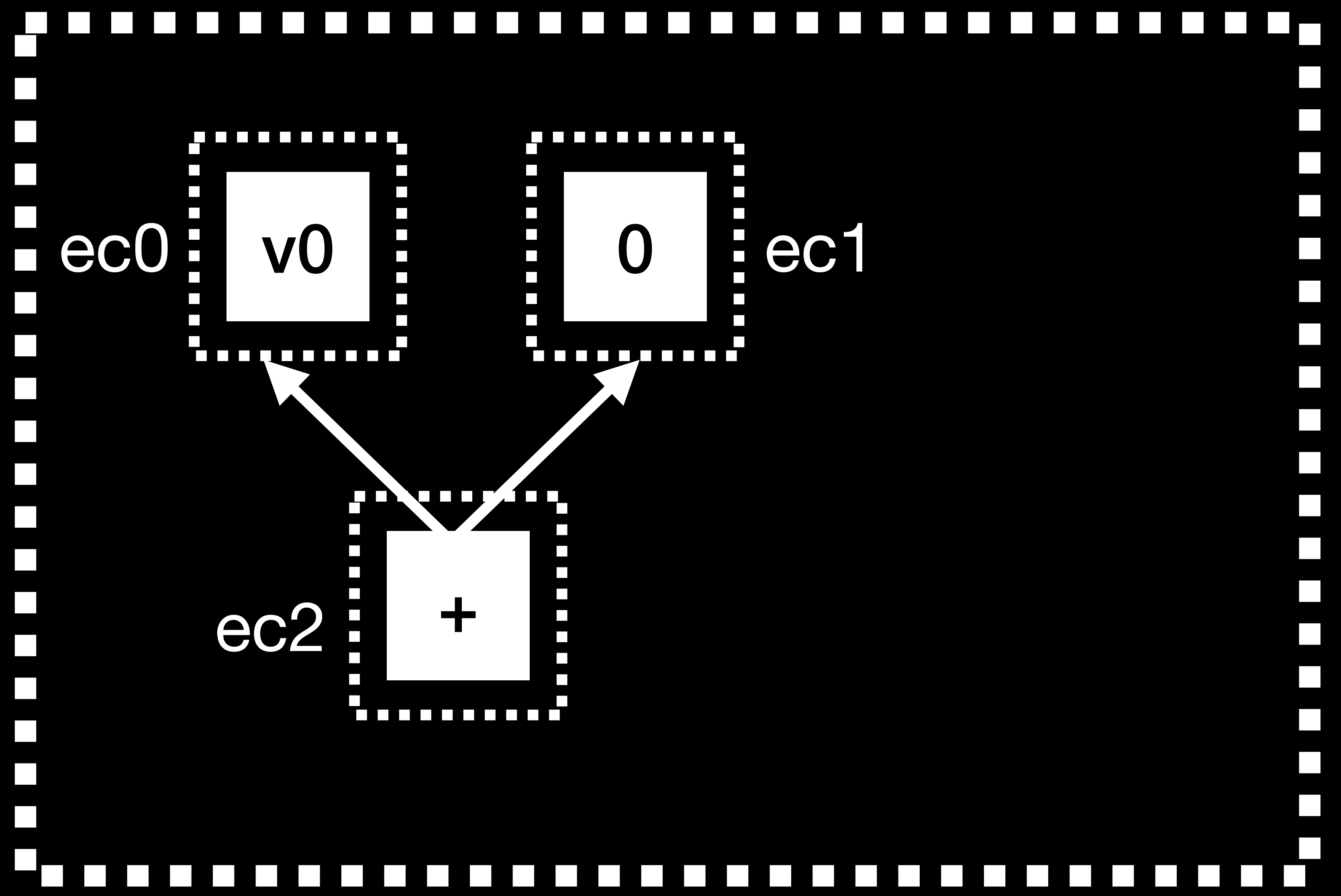
How do we handle this cycle?

- Cycles preclude single pass (imply fixpoint algorithm)
- We’re rewriting the arg after its use (no longer eager)
  —> need parent lists again

Cycles in E-graphs

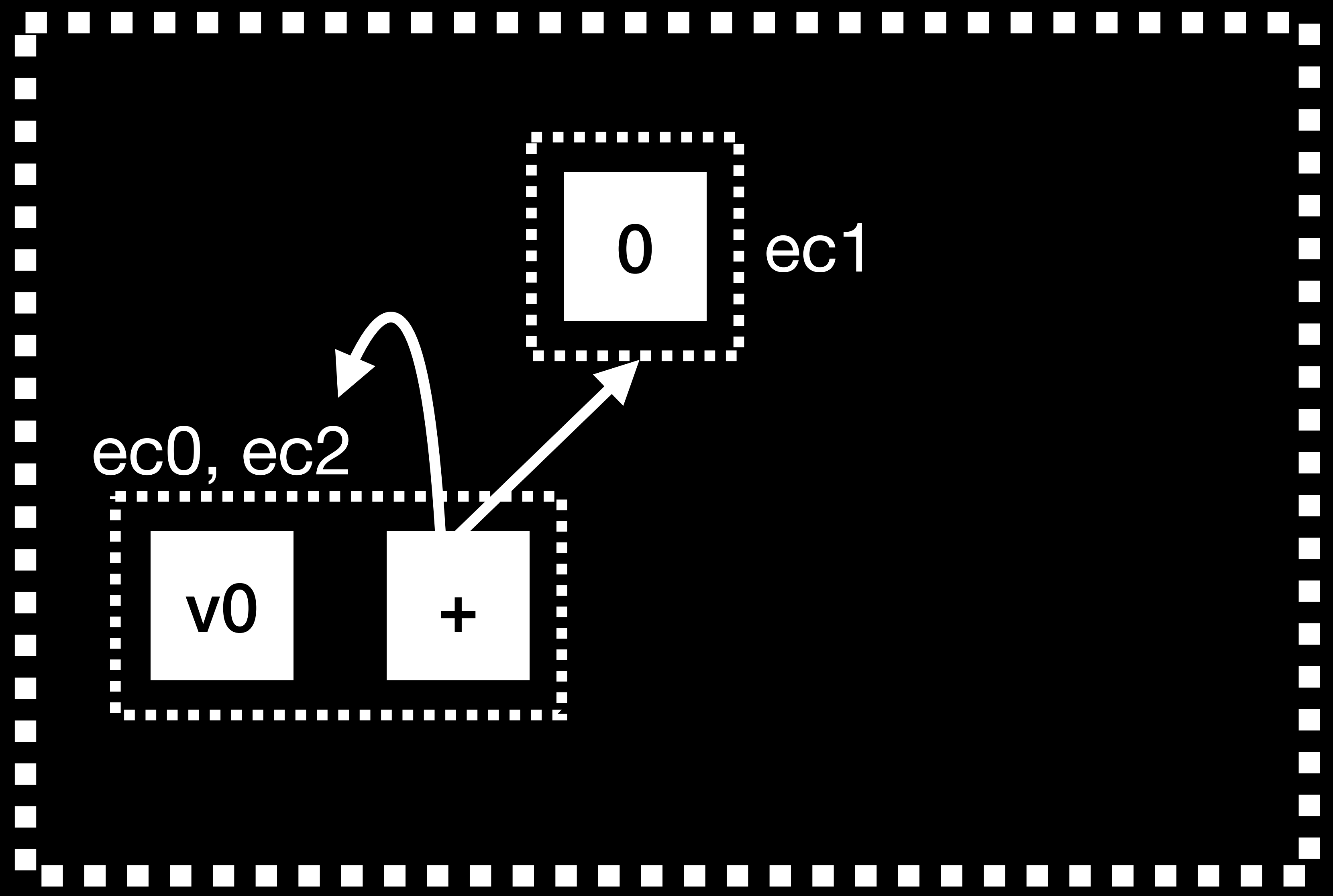


Cycles in E-graphs



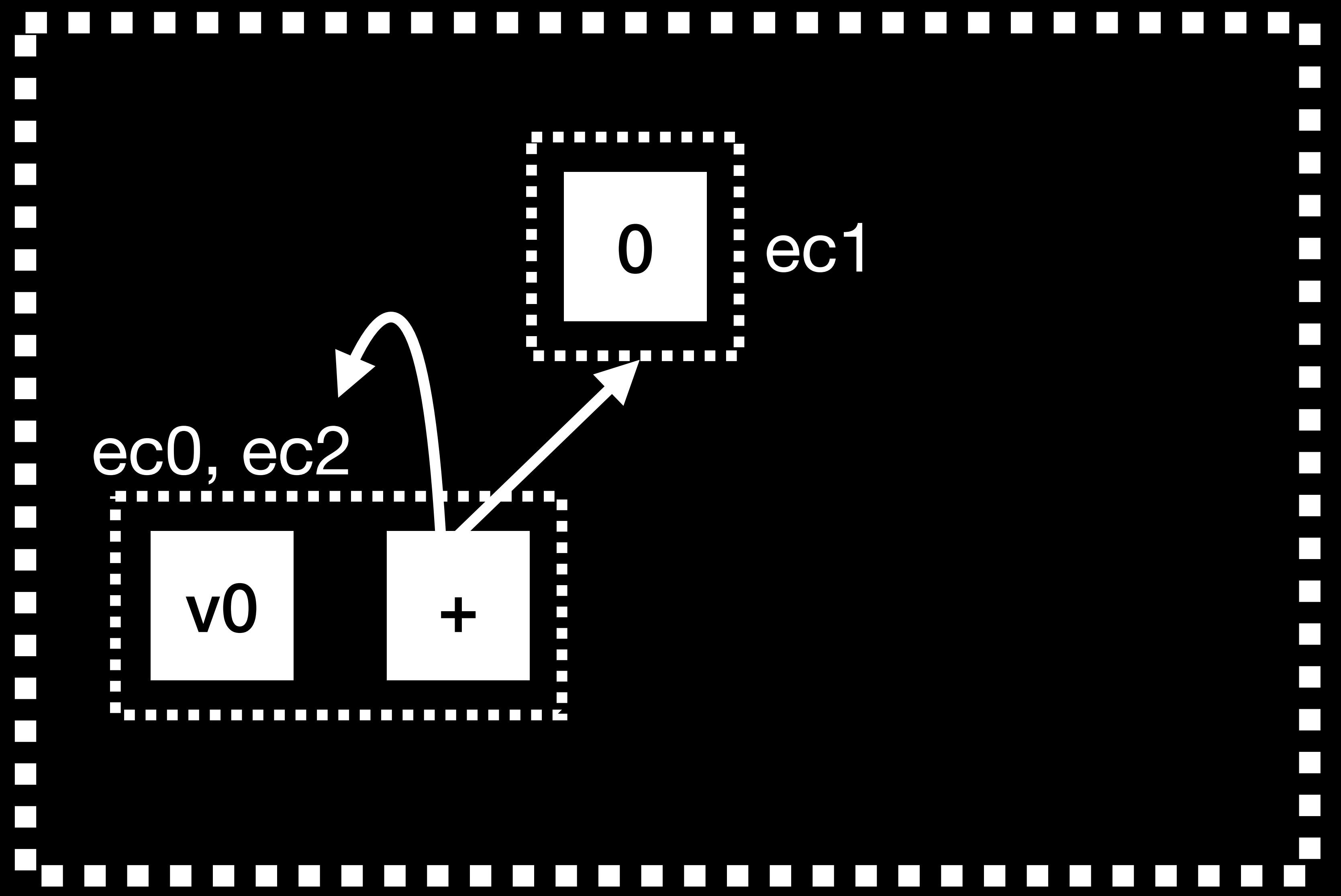
x + 0 => x

Cycles in E-graphs



x + 0 => x

Cycles in E-graphs

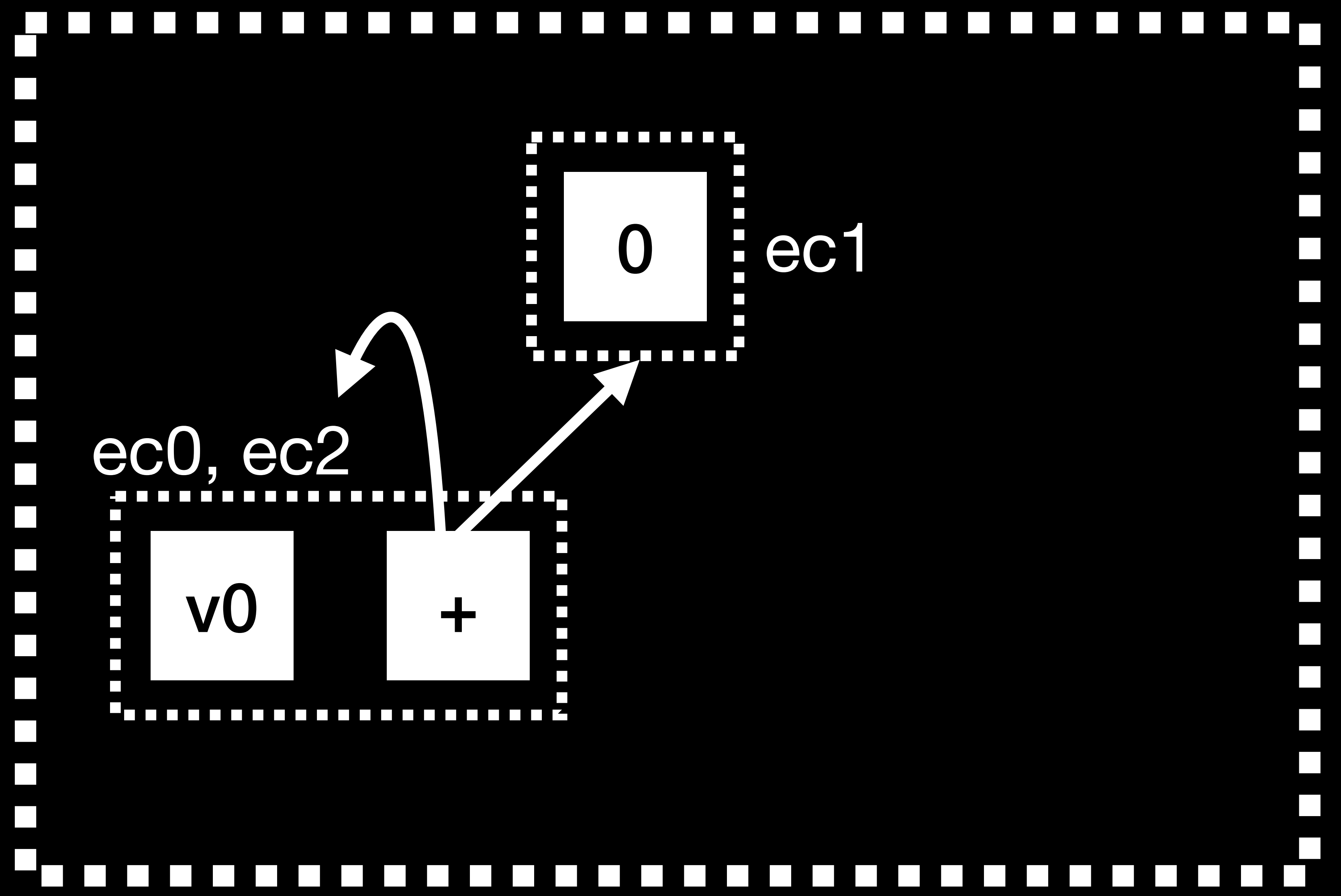


x + 0 => x

Observation:

- egraph does not record rewrite “direction”

Cycles in E-graphs

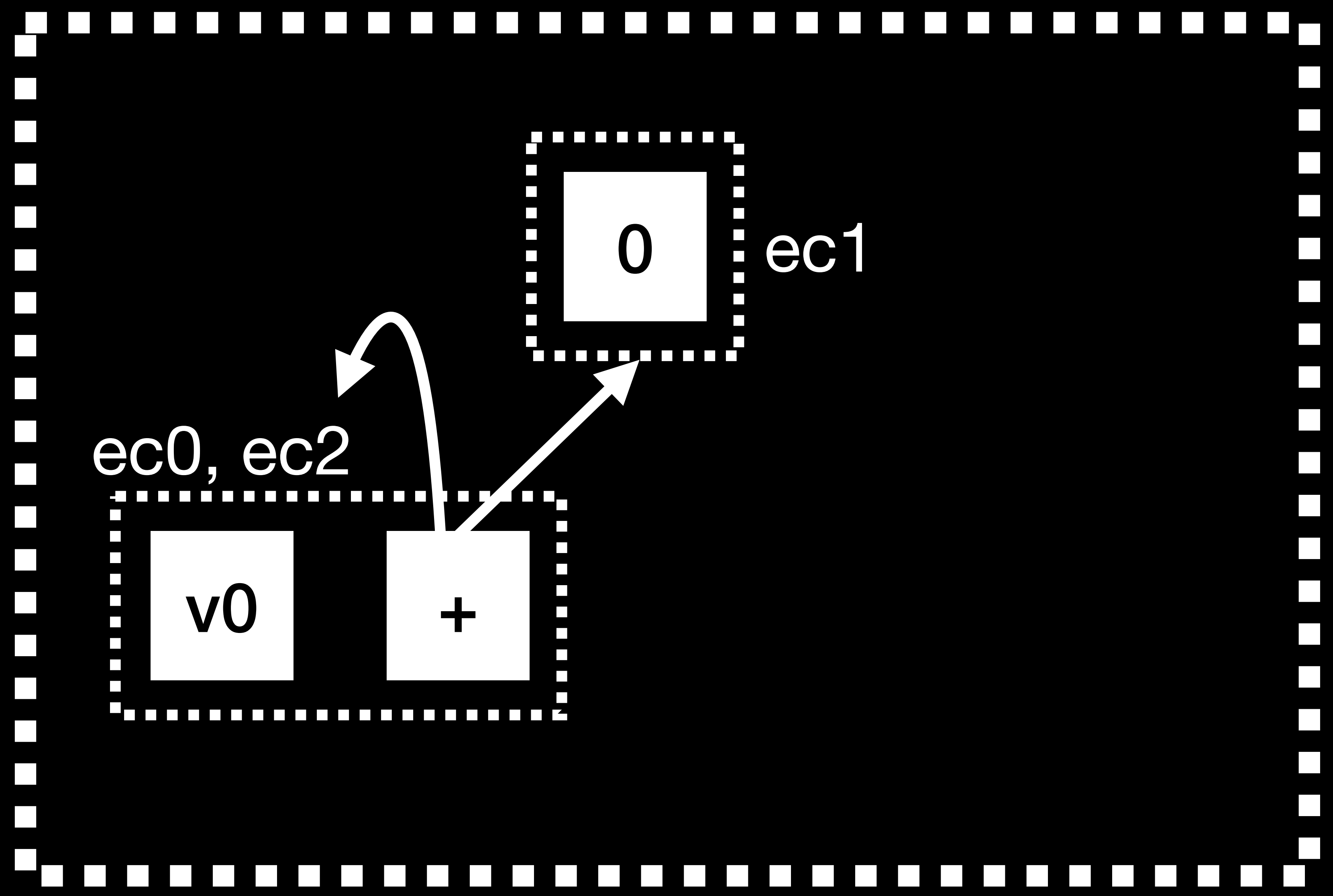


x + 0 => x

Observation:

- egraph does not record rewrite “direction”
- this egraph equivalent to
  - start with x
  - rewrite with x => x + 0

Cycles in E-graphs

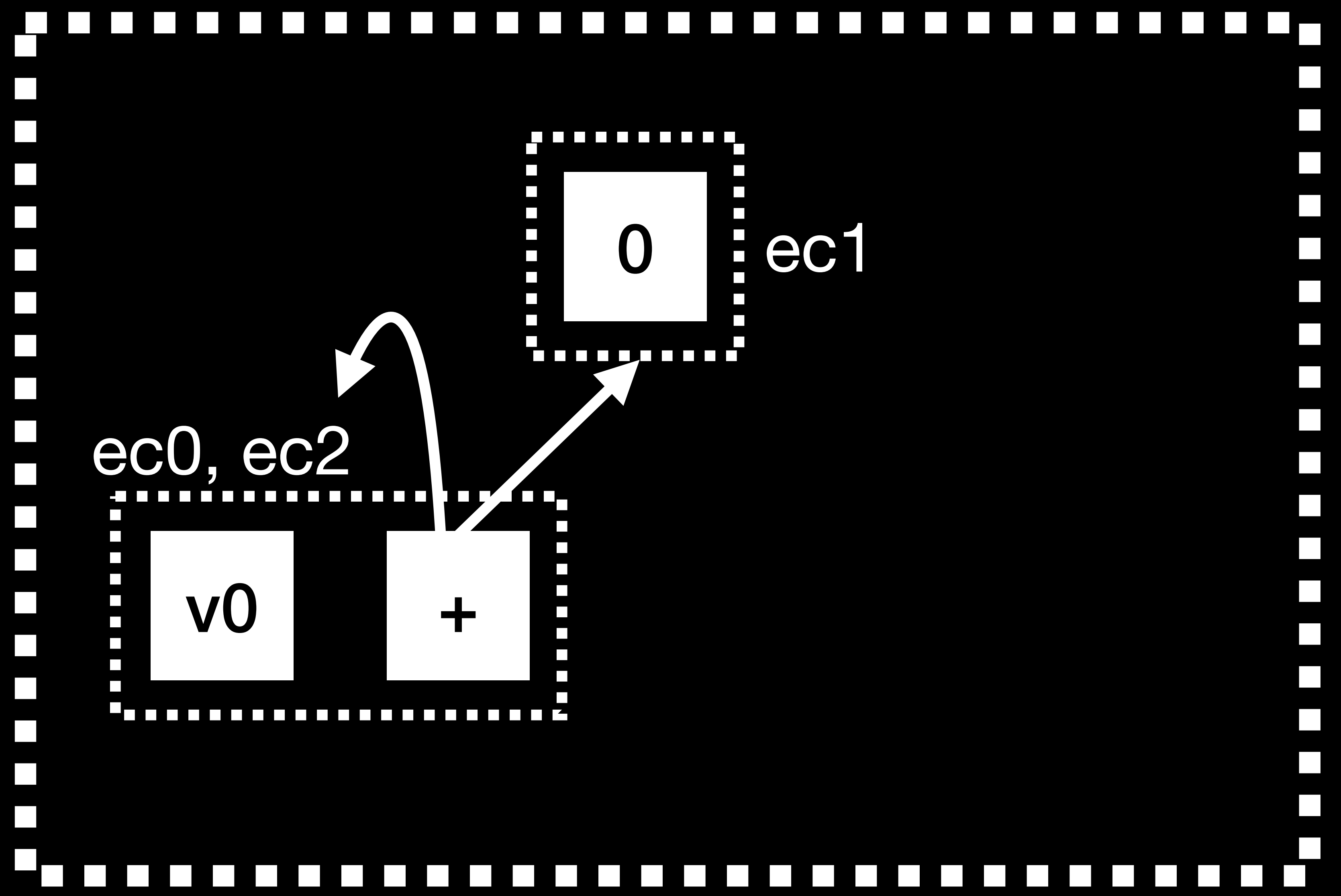


x + 0 => x

Observation:

- egraph does not record rewrite “direction”
- this egraph equivalent to
  - start with x
  - rewrite with x => x + 0
- rewrite rules that equate *part* to *whole* are (reverse)-generative

Cycles in E-graphs



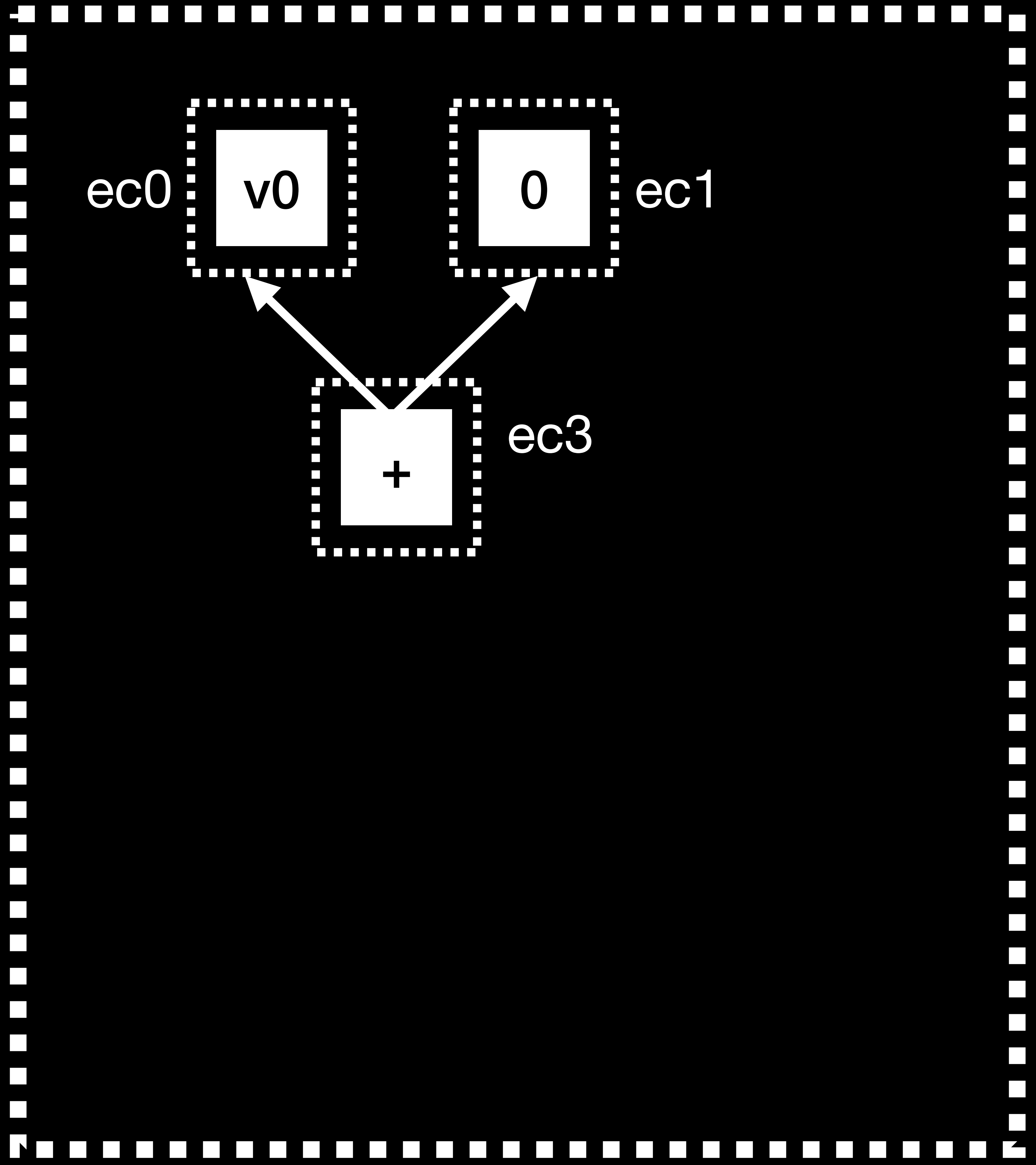
x + 0 => x

Observation:

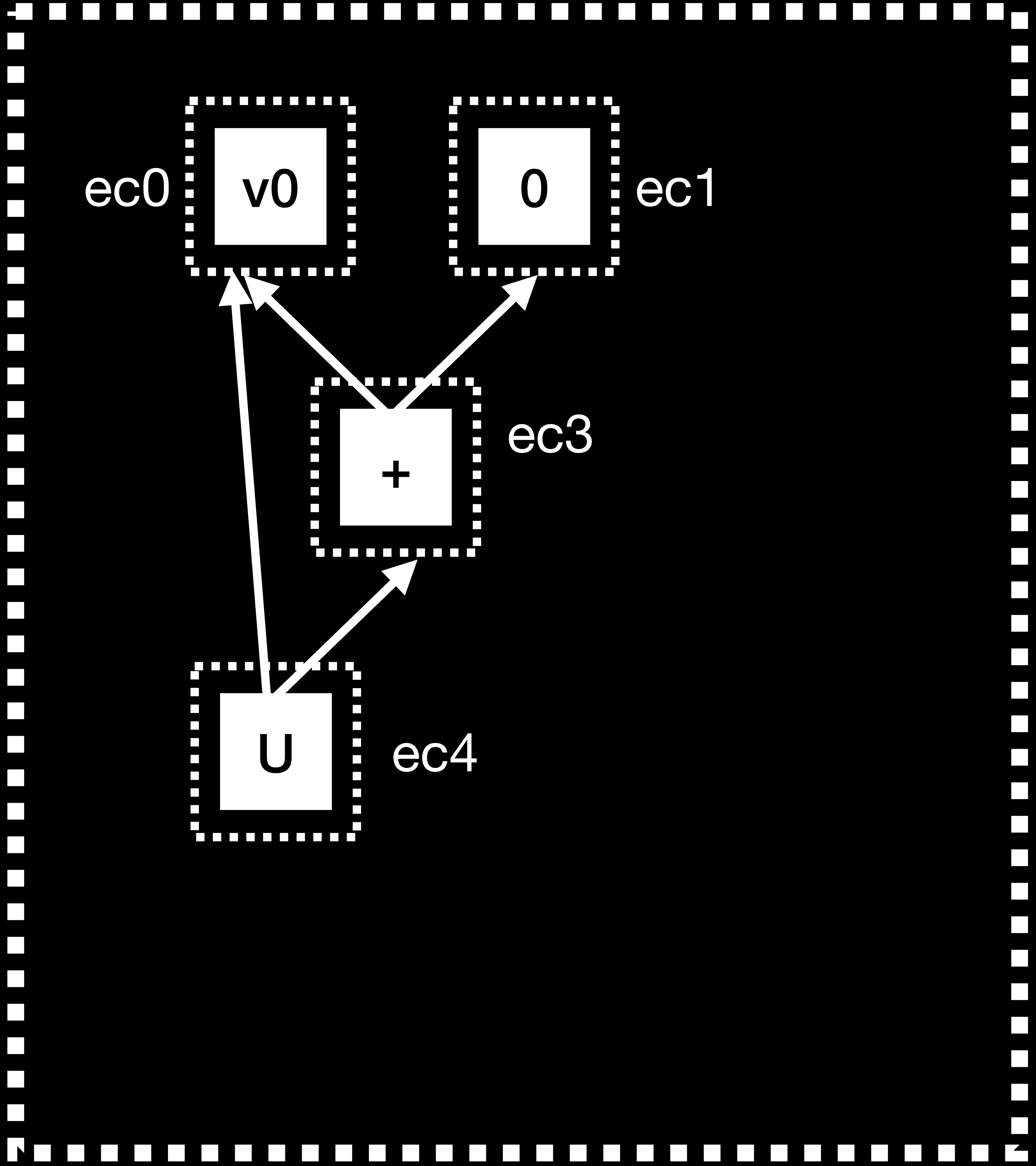
- egraph does not record rewrite “direction”
- this egraph equivalent to
  - start with x
  - rewrite with x => x + 0
- rewrite rules that equate *part* to *whole* are (reverse)-generative

Cycles occur even if original egraph is acyclic (e.g., from SSA)

Persistent immutable e-classes

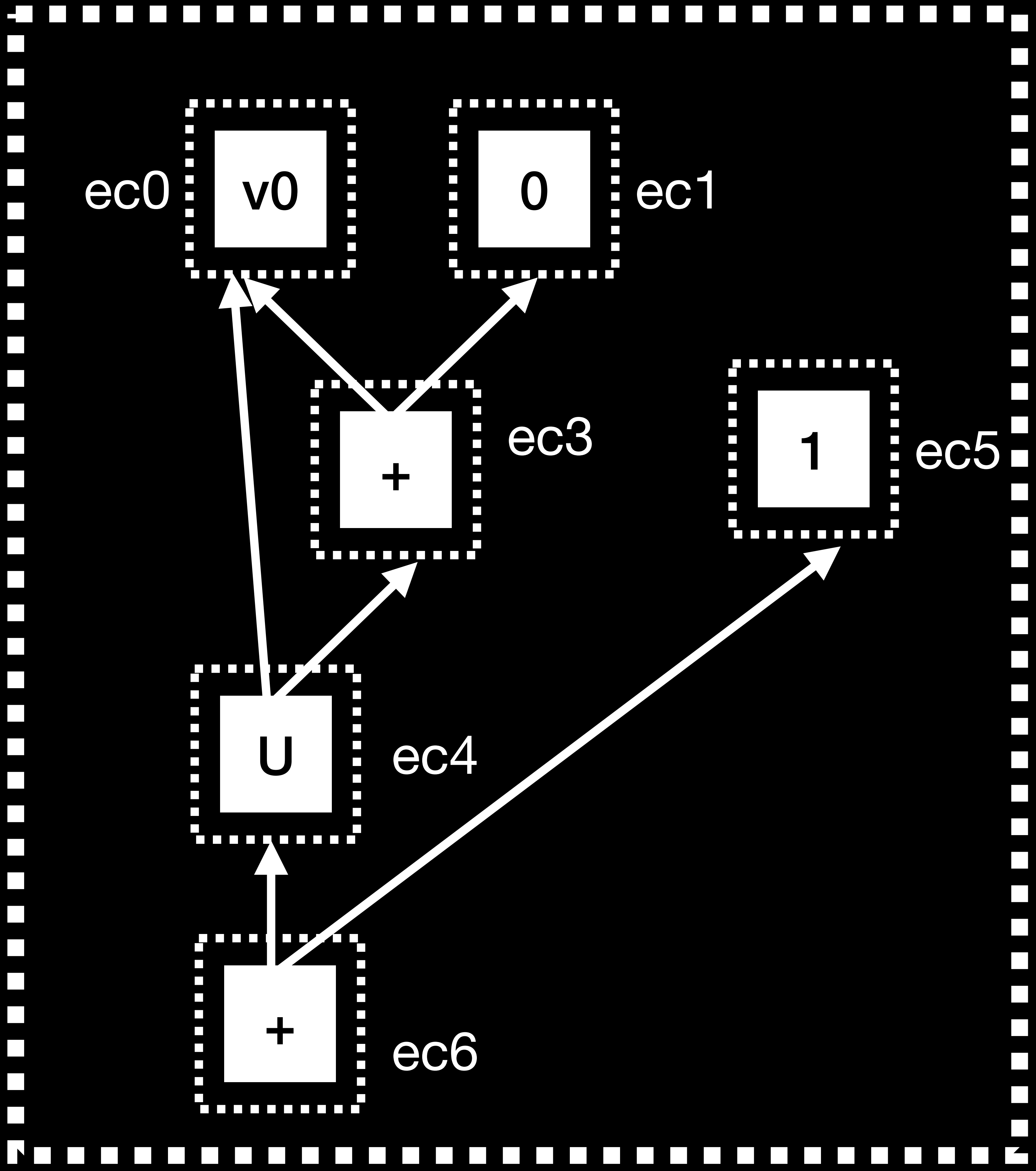


Persistent immutable e-classes



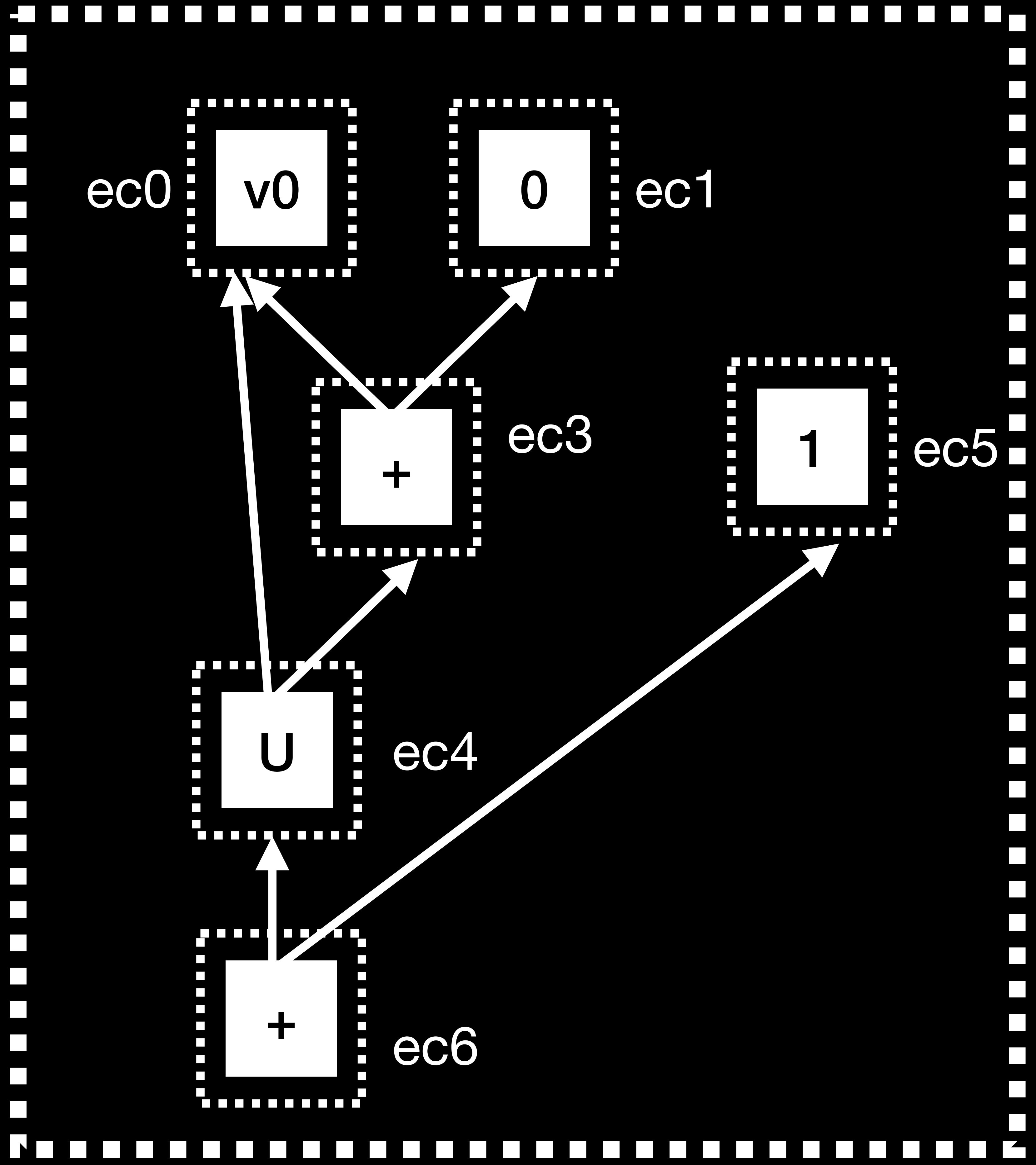
- Never rewrite a node
- Represent eclasses as *trees* of *union nodes*

Persistent immutable e-classes



- Never rewrite a node
- Represent eclasses as *trees* of *union nodes*

Persistent immutable e-classes



- Never rewrite a node
- Represent eclasses as *trees* of *union nodes*
- As we build the egraph, track *latest* id for a given value
- Invoke rewrite rules *when a node is created*
- enter into hashcons map *with final union'd ID*

Persistent immutable e-classes

Eager rewriting

Acyclicity

Persistent
immutable
data structure

Persistent immutable e-classes

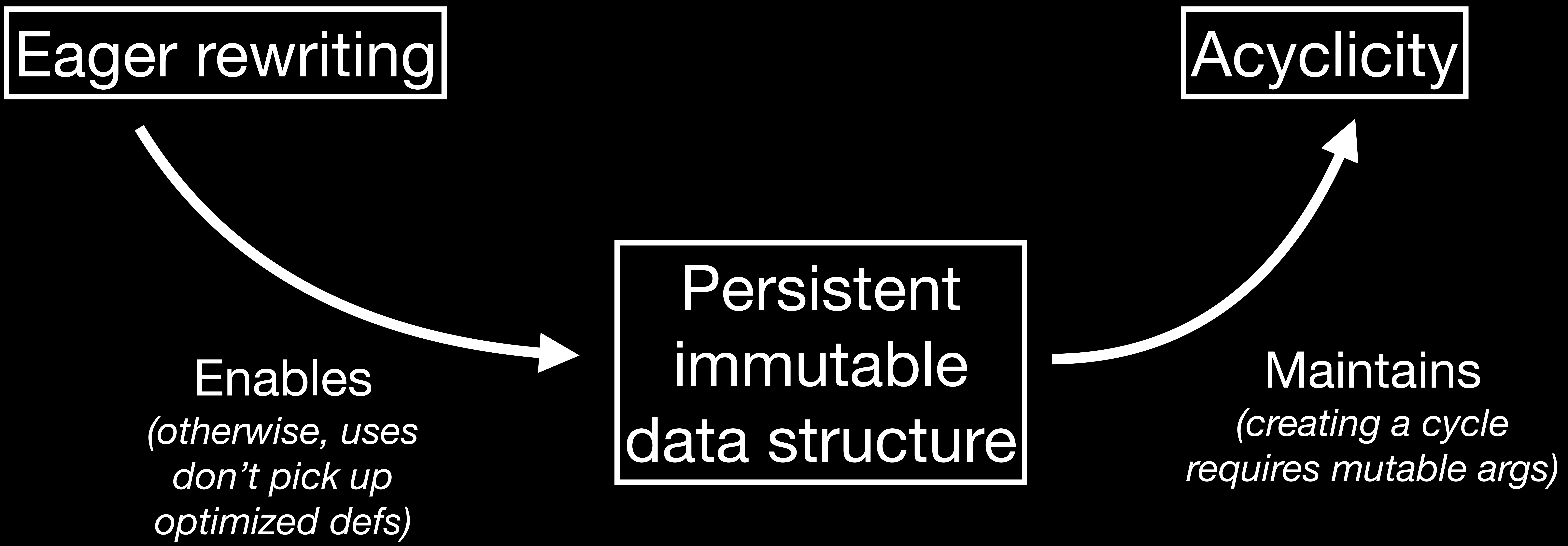
Eager rewriting

Acyclicity

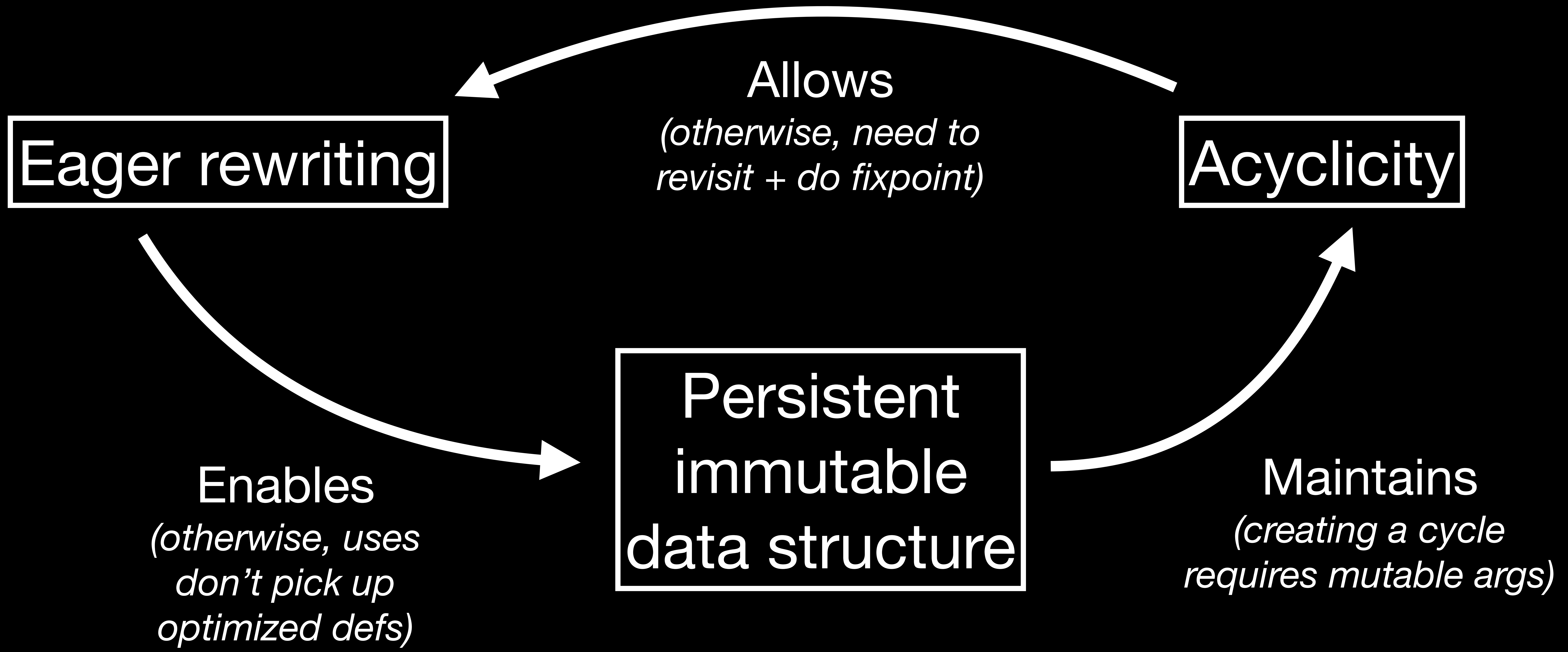
Enables
(otherwise, uses don't pick up optimized defs)

Persistent immutable data structure

Persistent immutable e-classes



Persistent immutable e-classes



E-graph vs. ægraph

egg-style egraph:
batched rewriting + repair

+ Strongly normalizing
+ Supports arbitrarily cyclic input

- Requires parent pointers and rehashing on fixup
- Repair step is a fixpoint

ægraph:
eager rewriting + immutable union nodes

- Can miss rewrites
  (depending on rule structure)
- Cannot support cyclic input
  (e.g., seeing through phi-nodes)

+ Single-pass rewrite
+ No parent pointers
  (minimal memory + maintenance overhead)

E-graph vs. ægraph

egg-style egraph:
batched rewriting + repair

+ Strongly normalizing
+ Supports arbitrarily cyclic input

- Requires parent pointers and rehashing on fixup
- Repair step is a fixpoint

ægraph:
eager rewriting + immutable union nodes

- Can miss rewrites
  (depending on rule structure)
- Cannot support cyclic input
  (e.g., seeing through phi-nodes)

+ Single-pass rewrite
+ No parent pointers
  (minimal memory + maintenance overhead)

→ We can avoid batched repair because *there is no repair*

How We Write Rules

- Two fundamental compromises: acyclicity and *more targeted rewrites*

How We Write Rules

- Two fundamental compromises: acyclicity and *more targeted rewrites*
  - No catchall associativity or commutativity rewrites!

```
(iadd a b) => (iadd b a)
```

```
  (iadd a (iadd b c))
=> (iadd (iadd a b) c)
```

How We Write Rules

- Two fundamental compromises: acyclicity and *more targeted rewrites*
  - No catchall associativity or commutativity rewrites!

~~(iadd a b) => (iadd b a)~~

~~(iadd a (iadd b c))~~
~~=> (iadd (iadd a b) c)~~

How We Write Rules

- Two fundamental compromises: acyclicity and *more targeted rewrites*
    - Only limited “non-directional” rewrites

How We Write Rules

- Two fundamental compromises: acyclicity and *more targeted rewrites*
  - Only limited “non-directional” rewrites

```
(bnot (band a b)) => (bor (bnot a) (bnot b))
```

How We Write Rules

- Two fundamental compromises: acyclicity and *more targeted rewrites*
  - Only limited “non-directional” rewrites

```
(bnot (band a b)) => (bor (bnot a) (bnot b))
```

→ OK (part of a “strategy”: push bnots downward)

→ But let’s not also have the other direction!

How We Write Rules

- Two fundamental compromises: acyclicity and *more targeted rewrites*
  - *No catchall associativity or commutativity rewrites!*
  - Only limited “non-directional” rewrites

How We Write Rules

- Two fundamental compromises: acyclicity and *more targeted rewrites*
  - *No catchall associativity or commutativity rewrites!*
  - Only limited “non-directional” rewrites
- Acyclicity precludes rules that operate over blockparams (phis)

How We Write Rules

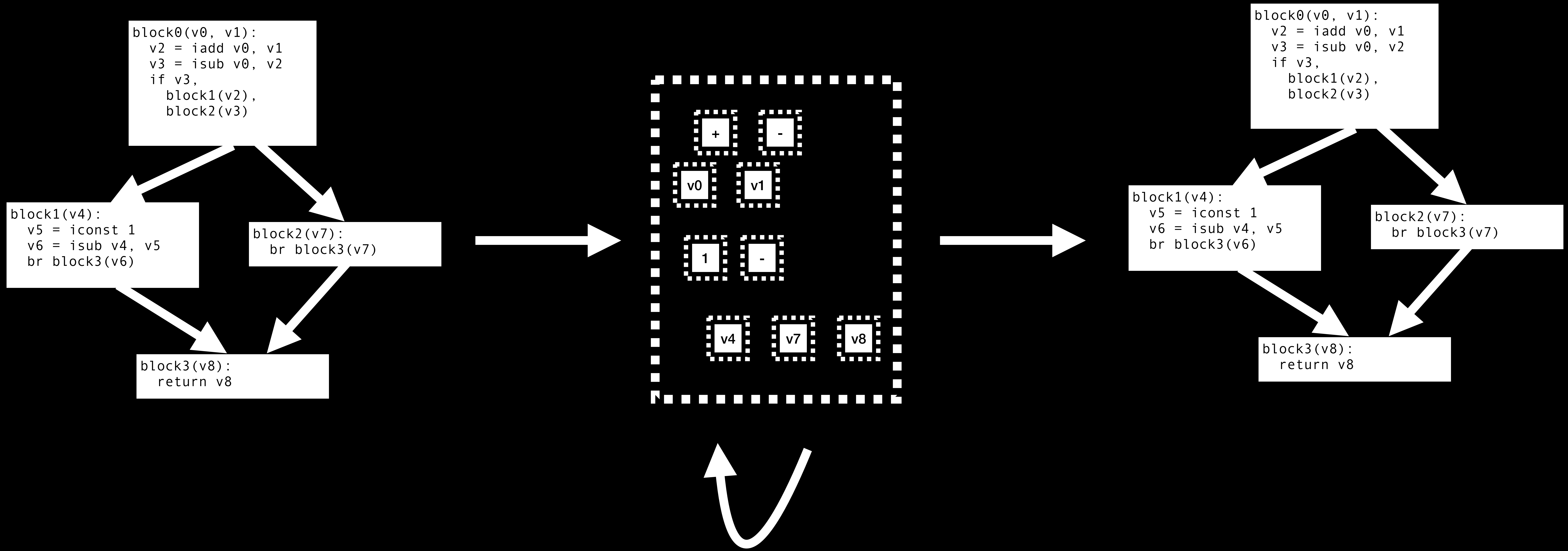
- Two fundamental compromises: acyclicity and *more targeted rewrites*
  - *No catchall associativity or commutativity rewrites!*
  - Only limited “non-directional” rewrites
- Acyclicity precludes rules that operate over blockparams (phis)

These limitations are OK!
At least as powerful as traditional rewrites;
and we’ve solved phase ordering;
and we can make use of “multi-version” + cost-based extraction.

What E-graphs Gave Us

- If not full EqSat + repair phase, what do ægraphs take from e-graphs?
  - **Rewriting:** a powerful unifying paradigm for optimizations
  - **Multiple value representations:** explores all rewrite paths; cost function makes final resolution in principled way
  - **Sea-of-nodes IR for pure values:** natural framework for code motion

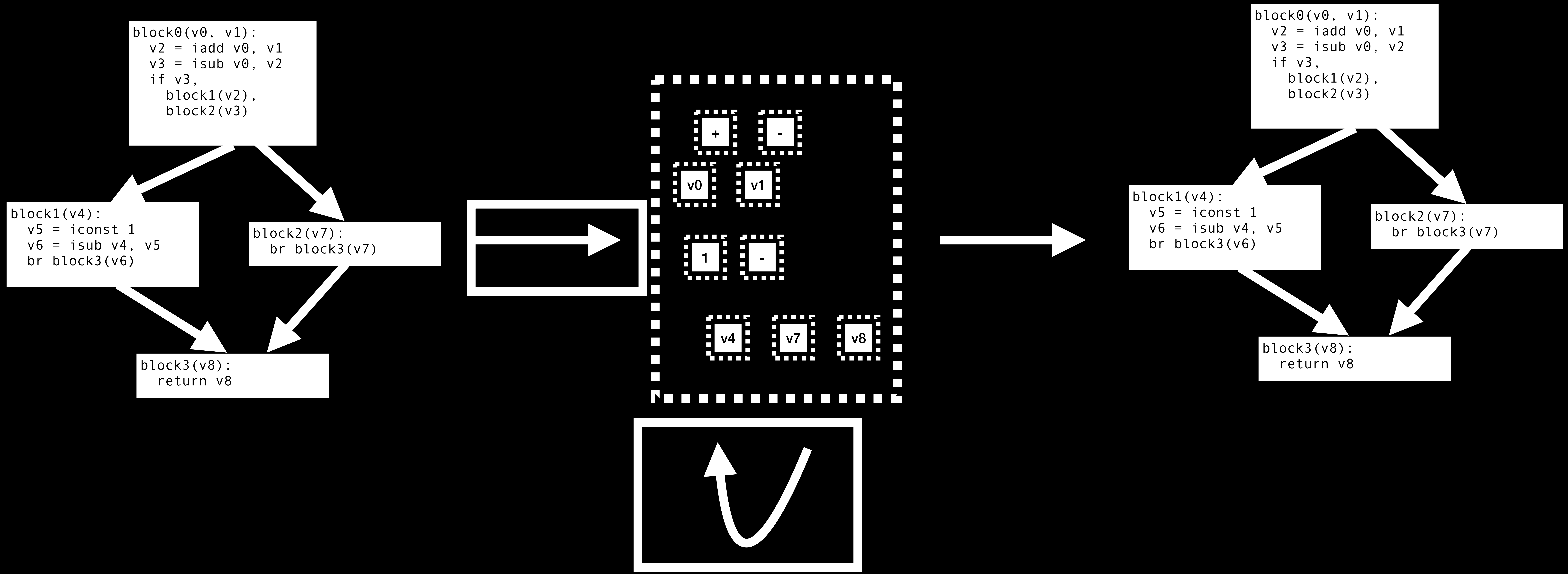
The ægraph Passes



x + 0 => x

…

The ægraph Passes

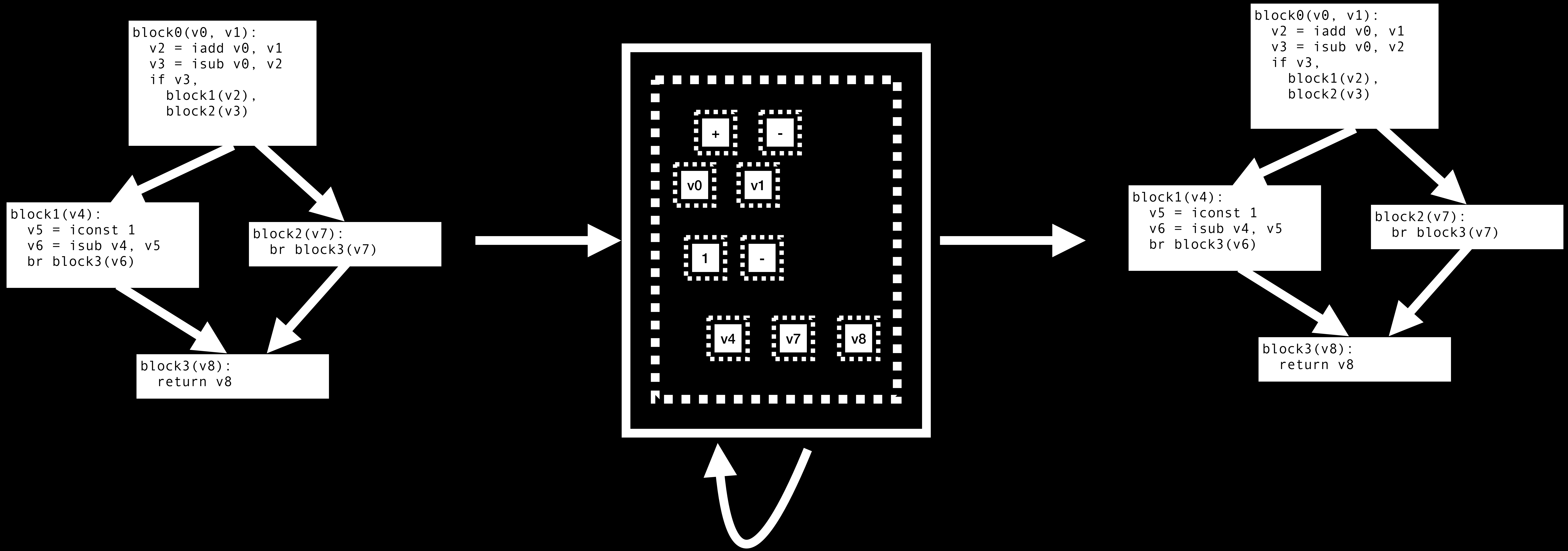


x + 0 => x

...

1. Build ægraph *and* eagerly rewrite

The ægraph Passes

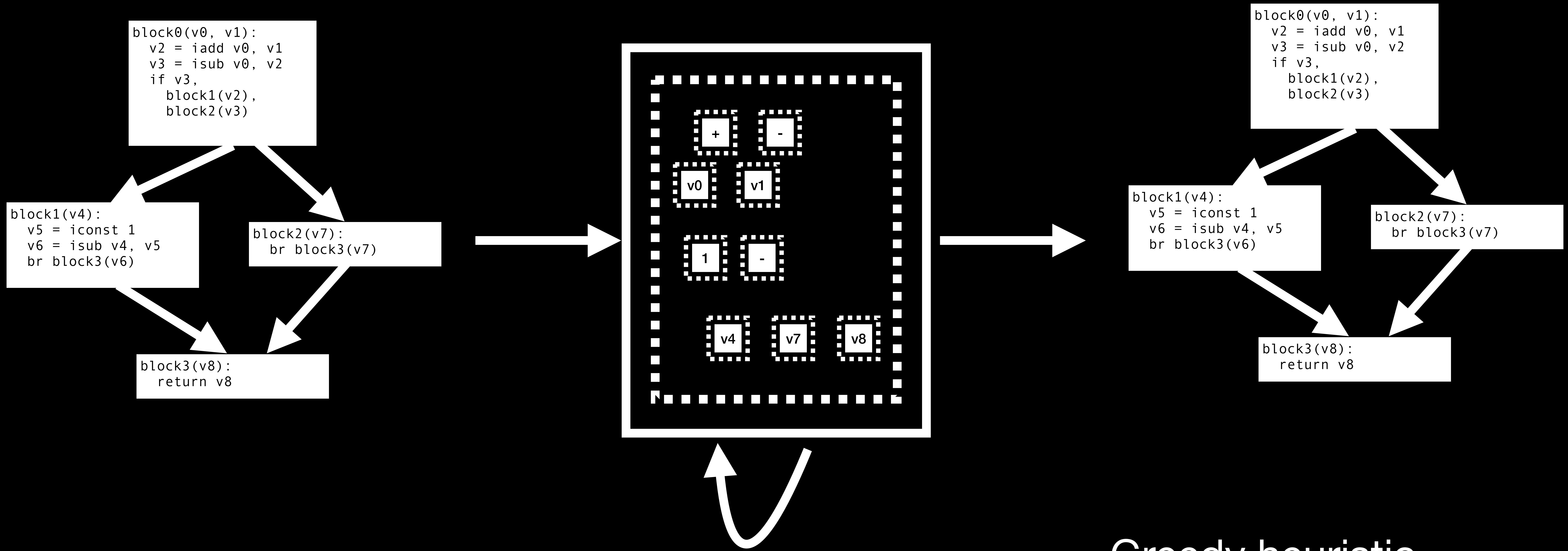


x + 0 => x

...

2. Perform extraction

The ægraph Passes



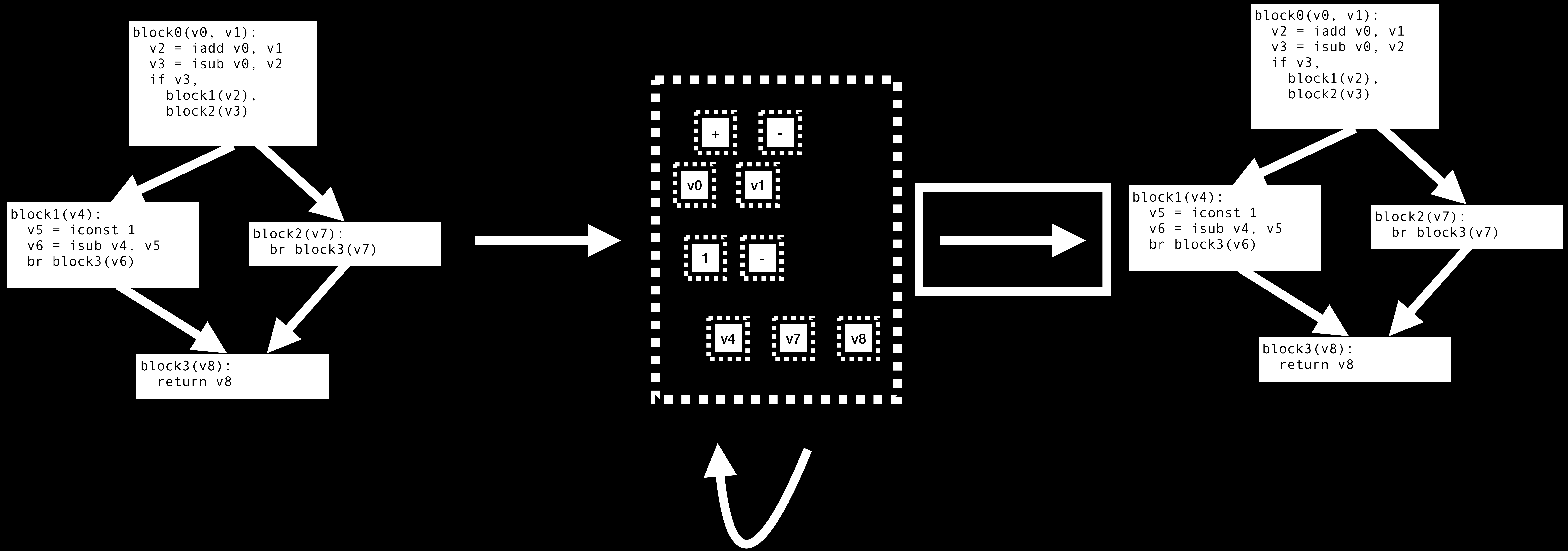
x + 0 => x

…

2. Perform extraction

- Greedy heuristic
- Dynamic programming (single pass)

The ægraph Passes



x + 0 => x

...

3. Scoped elaboration

The ægraph Passes

1. Build and rewrite
2. Extraction
3. Scoped elaboration

→ Three linear passes, no fix-point loops

1. Why *we want a rewrite-based optimizer*

2. How *to turn a CFG into an egraph and back again*

3. Cycles *why they occur, and what to do about them*

4. Results *how well does it work?*

5. Lessons *in translating research to production*

1. Why *we want a rewrite-based optimizer*

2. **How** *to turn a CFG into an egraph and back again*

3. Cycles *why they occur, and what to do about them*

4. Results *how well does it work?* ⟵

5. Lessons *in translating research to production*

Performance

Performance

SpiderMonkey.wasm

11% faster runtime

2% longer compile-time

Performance

SpiderMonkey.wasm

11% faster runtime

2% longer compile-time

bz2

3% higher runtime

2% faster compile-time

Performance

Speedups

gimli	22%
spidermonkey	11%
minicsv	9%
ratelimit	8%
switch	3%
fib2	3%
intgemm	1%

Performance

Speedups		*Slowdowns*	
gimli	22%	random	-31%
spidermonkey	11%	hex-simd	-16%
minicsv	9%	meshoptimizer	-14%
ratelimit	8%	ed25519	-13%
switch	3%	blake3-simd	-6%
fib2	3%	keccak	-4%
intgemm	1%	bz2	-3%

Performance

- Instruction scheduling: #6260
- Missing opt rules
  - Magic div constants: #6049

Speedups

gimli	22%
spidermonkey	11%
minicsv	9%
ratelimit	8%
switch	3%
fib2	3%
intgemm	1%

Slowdowns

random	-31%
hex-simd	-16%
meshoptimizer	-14%
ed25519	-13%
blake3-simd	-6%
keccak	-4%
bz2	-3%

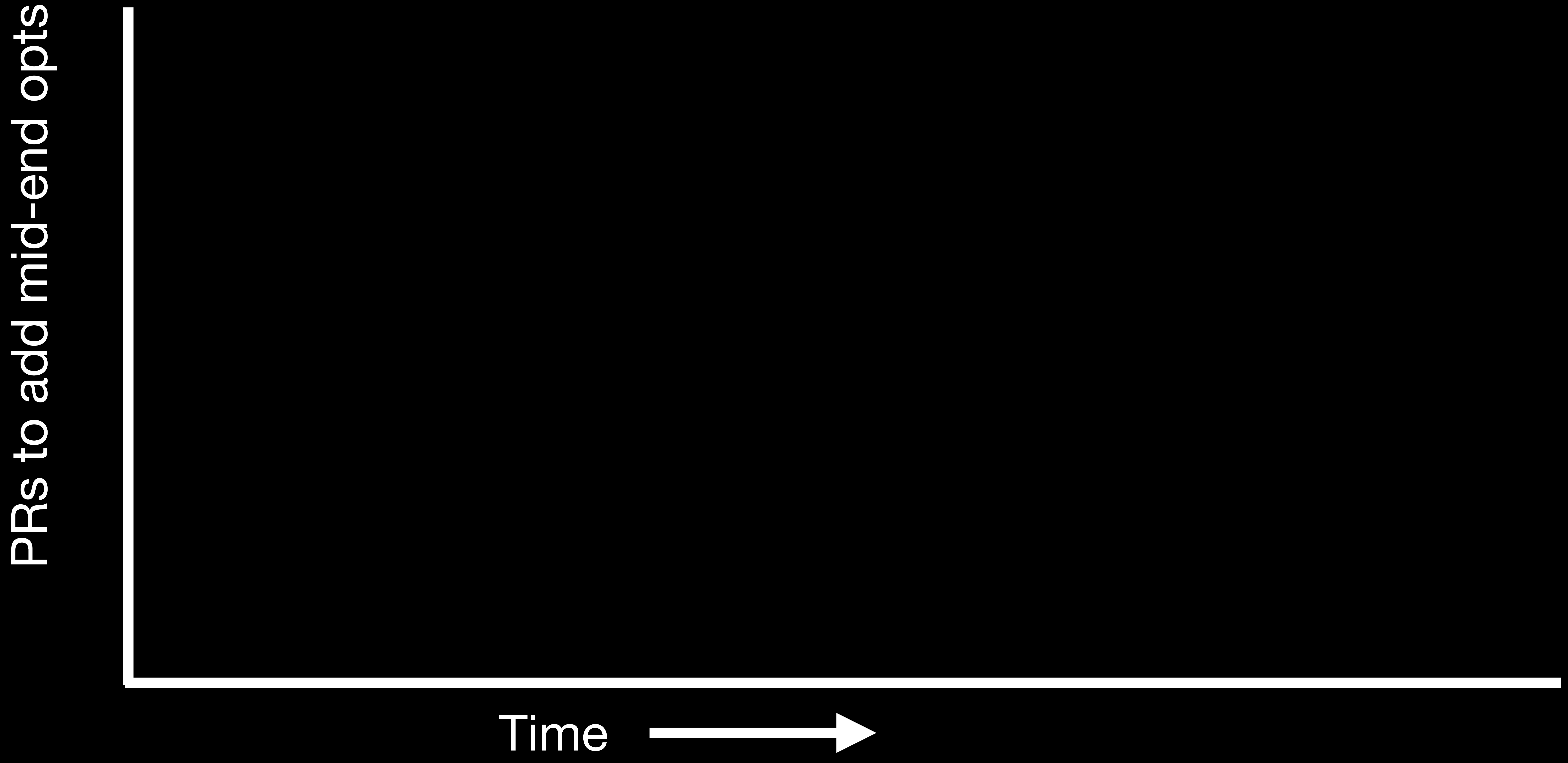
Performance

bjorn3 commented on Dec 14, 2022

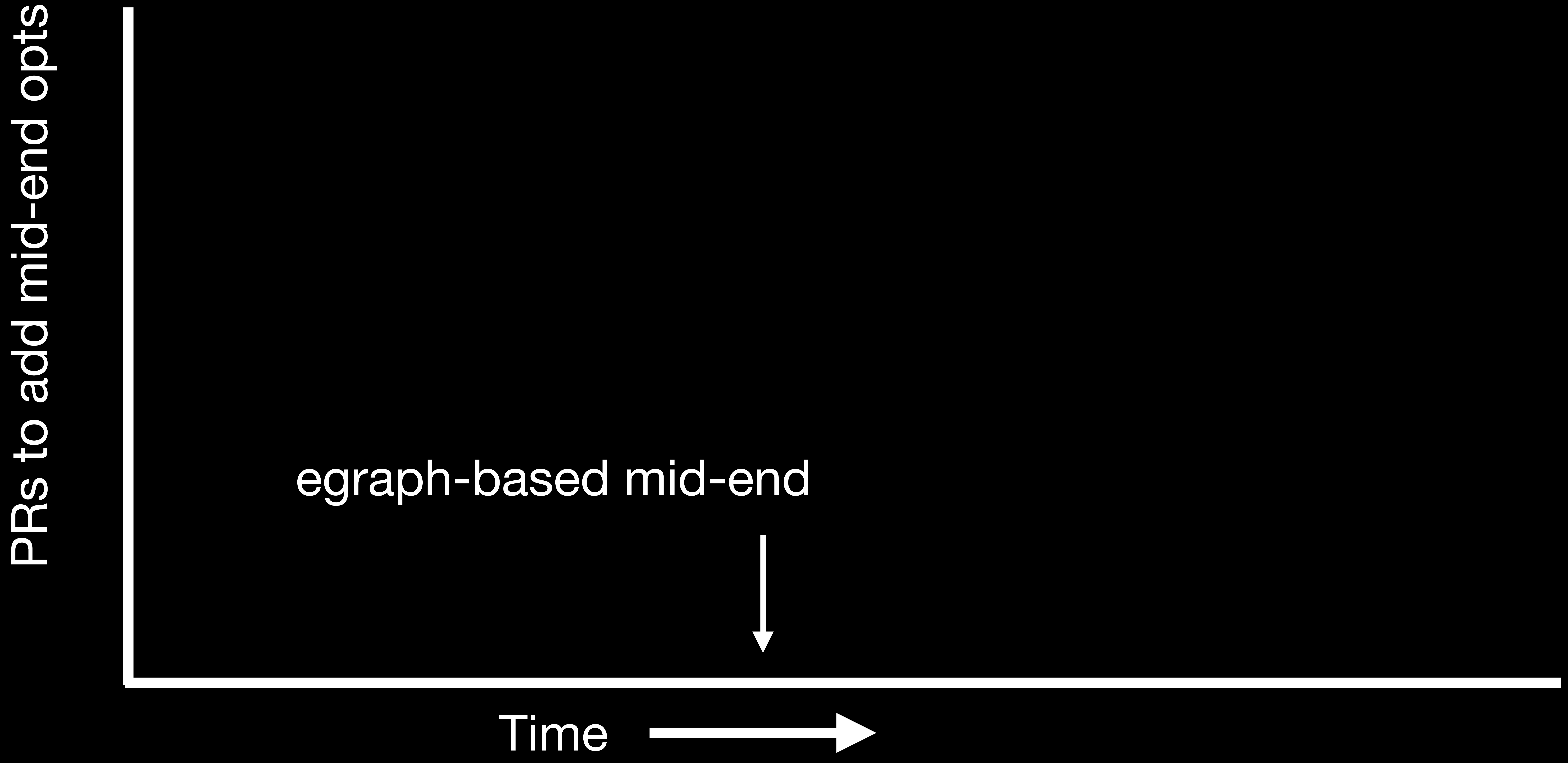
I just did some benchmarking of egraphs and the perf improvement is huge on the benchmark I tried:

```
Benchmark 1: ./raytracer_cg_clif
  Time (mean ± σ):      8.553 s ±  0.010 s    [User: 8.539 s, System: 0.014 s]
  Range (min … max):    8.543 s …  8.568 s    10 runs

Benchmark 2: ./raytracer_cg_clif_egraph
  Time (mean ± σ):      6.068 s ±  0.017 s    [User: 6.057 s, System: 0.011 s]
  Range (min … max):    6.047 s …  6.108 s    10 runs

Benchmark 3: ./raytracer_cg_clif_release
  Time (mean ± σ):      6.450 s ±  0.021 s    [User: 6.439 s, System: 0.012 s]
  Range (min … max):    6.410 s …  6.482 s    10 runs

Benchmark 4: ./raytracer_cg_clif_release_egraph
  Time (mean ± σ):      5.853 s ±  0.053 s    [User: 5.841 s, System: 0.012 s]
  Range (min … max):    5.779 s …  5.908 s    10 runs

Summary
  './raytracer_cg_clif_release_egraph' ran
    1.04 ± 0.01 times faster than './raytracer_cg_clif_egraph'
    1.10 ± 0.01 times faster than './raytracer_cg_clif_release'
    1.46 ± 0.01 times faster than './raytracer_cg_clif'
```

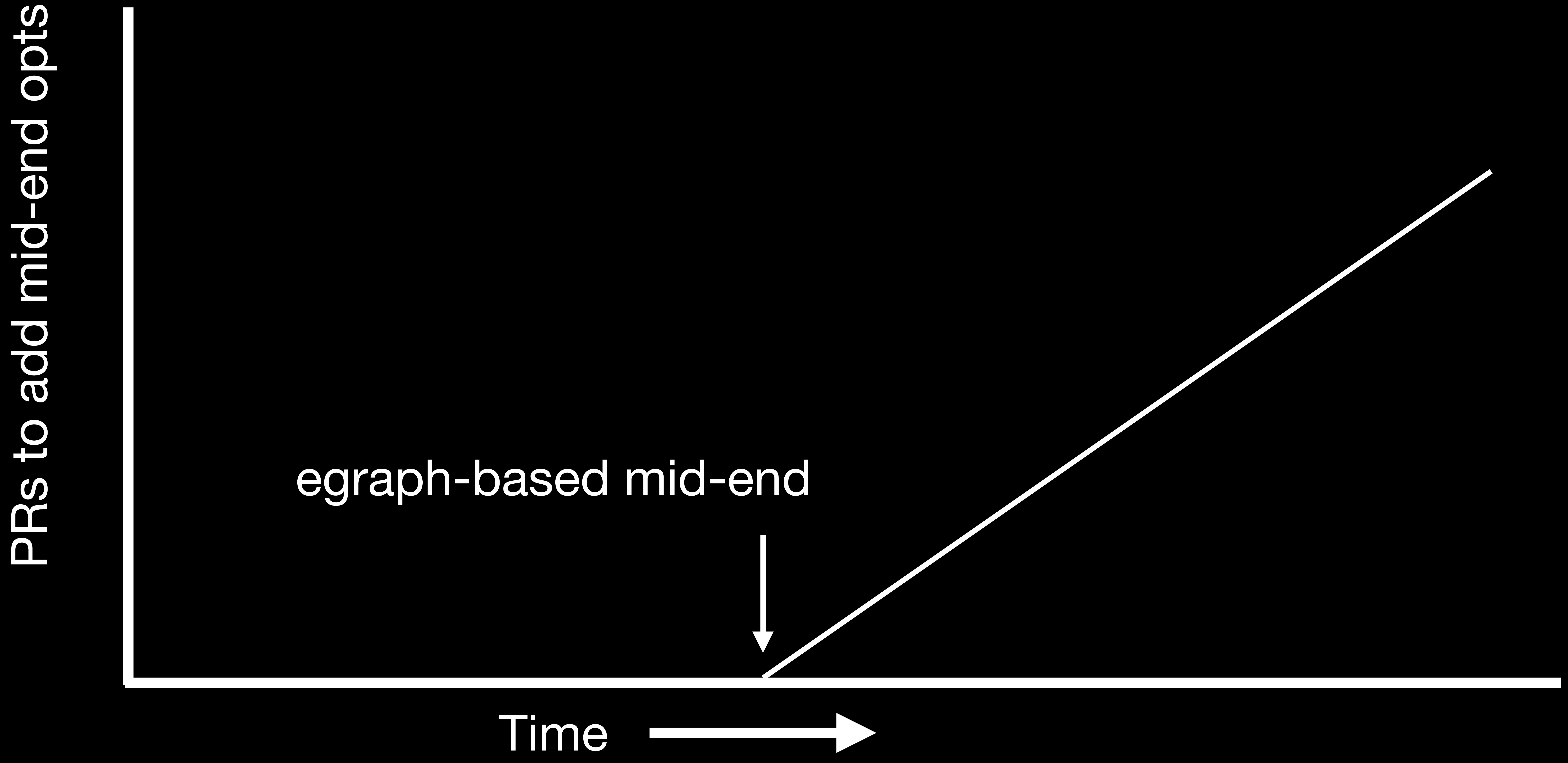
Project Health & Enablement



Project Health & Enablement



Project Health & Enablement



Project Health & Enablement

PRs to add mid-end opts

```
Optimize sign extension via shifts (#6220)
egraphs: Add `bmask` bit pattern optimization rule (#6196)
Add `multi_lane` precondition to `bitselect` => `{u,s}{min,max}` rewrite (#6201)
ISLE: simplify select/bitselect when both choices are the same (#6141)
Add egraph cprop optimizations for `splat` (#6148)
ISLE: rewrite loose inequalities to strict inequalities and strict inequalities to equalities (#6130)
ISLE: rewrite `and`/`or` of `icmp` (#6095)
ISLE: add synonyms for all variations of `icmp` (#6081)
cranelift: rewrite `iabs(ineg(x))` and `iabs(iabs(x))` (#6072)
cranelift: rewrite `x*-1` to `ineg(x)` (#6052)
craneleft: cancel `ineg` when args to `imul` (#6053)
cranelift: simplify `icmp` against UMAX/SMIN/SMAX (#6037)
cranelift: simplify `x-x` to `0` (#6032)
cranelift: simplify `fneg(fneg(x))` to `x` (#6034)
cranelift: simplify `ineg(ineg(x))` to `x` (#6033)
Add egraph optimization for fneg's cancelling out (#5910)
Cranelift: Generalize `(x << k) >> k` optimization (#5746)
cranelift: Optimize `select+icmp` into `{s,u}{min,max}` (#5546)
Cranelift: Collapse double extends into a single extend (#5772)
Generalize and/or/xor optimizations (#5744)
Algebraic opts: Reuse `iconst 0` from LHS (#5724)
Add some minor souper-harvested optimizations (#5735)
Cranelift: Only build iconst for ints <= 64 bits (#5723)
Legalize `b{and,or,xor}_not` into component instructions (#5709)
egraphs/cprop: Don't extend constants to `i128` (#5717)
Generalize u/sextend constant folding to all types (#5706)
Cranelift: Correctly wrap shifts in constant propagation (#5695)
Constant-fold icmp instructions (#5666)
Cranelift: Rewrite `or(and(x, y), not(y)) => or(x, not(y))` (#5676)
Cranelift: Rewrite `(x>>k)<<k` into masking off the bottom `k` bits (#5673)
Cranelift: constant propagate shifts (#5671)
Cranelift: Add egraph rule to rewrite `x * C ==> x << log2(C)` when `C` is a power of two (#5647)
egraph opt rules: do `(icmp cc x x) == {0,1}` only for integer types. (#5438)
```

Project Health & Enablement

PRs to add mid-end opts

```
Optimize sign extension via shifts (#6220)
egraphs: Add `bmask` bit pattern optimization rule (#6196)
Add `multi_lane` precondition to `bitselect` => `{u,s}{min,max}` rewrite (#6201)
ISLE: simplify select/bitselect when both choices are the same (#6141)
Add egraph cprop optimizations for `splat` (#6148)
ISLE: rewrite loose inequalities to strict inequalities and strict inequalities to equalities (#6130)
ISLE: rewrite `and`/`or` of `icmp` (#6095)
ISLE: add synonyms for all variations of `icmp` (#6081)
cranelift: rewrite `iabs(ineg(x))` and `iabs(iabs(x))` (#6072)
cranelift: rewrite `x*-1` to `ineg(x)` (#6052)
craneleft: cancel `ineg` when args to `imul` (#6053)
cranelift: simplify `icmp` against UMAX/SMIN/SMAX (#6037)
cranelift: simplify `x-x` to `0` (#6032)
cranelift: simplify `fneg(fneg(x))` to `x` (#6034)
cranelift: simplify `ineg(ineg(x))` to `x` (#6033)
Add egraph optimization for fneg's cancelling out (#5910)
Cranelift: Generalize `(x << k) >> k` optimization (#5746)
cranelift: Optimize `select+icmp` into `{s,u}{min,max}` (#5546)
Cranelift: Collapse double extends into a single extend (#5772)
Generalize and/or/xor optimizations (#5744)
Algebraic opts: Reuse `iconst 0` from LHS (#5724)
Add some minor souper-harvested optimizations (#5735)
Cranelift: Only build iconst for ints <= 64 bits (#5723)
Legalize `b{and,or,xor}_not` into component instructions (#5709)
egraphs/cprop: Don't extend constants to `i128` (#5717)
Generalize u/sextend constant folding to all types (#5706)
Cranelift: Correctly wrap shifts in constant propagation (#5695)
Constant-fold icmp instructions (#5666)
Cranelift: Rewrite `or(and(x, y), not(y)) => or(x, not(y))` (#5676)
Cranelift: Rewrite `(x>>k)<<k` into masking off the bottom `k` bits (#5673)
Cranelift: constant propagate shifts (#5671)
Cranelift: Add egraph rule to rewrite `x * C ==> x << log2(C)` when `C` is a power of two (#5647)
egraph opt rules: do `(icmp cc x x) == {0,1}` only for integer types. (#5438)
```

33 PRs in 5 months

… from 8 authors!

Project Health & Enablement

```
+
+ ;; A reduction-of-an-extend back to the same original type is the same as not
+ ;; actually doing the extend in the first place.
+ (rule (simplify (ireduce ty (sextend _ x @ (value_type ty)))) x)
+ (rule (simplify (ireduce ty (uextend _ x @ (value_type ty)))) x)
```

Nobody would take the time to write a manual pass to do that!

Performance: Qualitative Discussion

Q: How did we achieve near-parity?

Performance: Qualitative Discussion

Q: How did we achieve near-parity?

A: By doing nearly the same amount of work!

Performance: Qualitative Discussion

Q: How did we achieve near-parity?

A: By doing nearly the same amount of work!

- E-graph interning ≈ GVN
- E-nodes are stored as instructions (same data structure)
- Initially, rewrites in egraph are equivalent to old pipeline

Performance: Qualitative Discussion

Q: How did we achieve near-parity?

A: By doing nearly the same amount of work!

- Differences: code placement (reconstruct all vs. incremental)
multi-version (selection, rewrite multiple paths)

Performance: Qualitative Discussion

Q: How did we achieve near-parity?

A: By doing nearly the same amount of work!

“Pay as you go” is *crucial* for incremental adoption!

Possible Future Plans

- Instruction selector as extraction pass
  - We have left-hand-side patterns for what the ISA can do efficiently
  - Why not lower directly from eclasses?
  - Somewhat complex interactions with scoped elaboration + pass direction

Possible Future Plans

- Optimization through block parameters (phi-nodes)
  - Sparse conditional constant propagation! Unify branch-folding + const-prop
  - Challenge: deal with cycles
  - Are there limited forms that operate in a single pass? (skip if backedge?)

Possible Future Plans

- Non-greedy instruction selection
    - We do extraction before elaboration
    - Optimal extraction depends on elaboration:
        - multiple uses of a value can “share” its cost
        - if another inst *needs* a value that is expensive, it becomes sunk cost

Possible Future Plans

- Fused / unrolled rewrites
  - We have efficient rule dispatch (decision tree), but only one step at a time
  - Can we statically unroll a path of rewrites?
  - … and even elide insertion of intermediates if we know they're "bad" (more expensive, always subsumed)?

Possible Future Plans

- Instruction scheduling
  - The ægraph throws away location information
  - Scoped elaboration recomputes it
  - The “as late as possible” schedule that results is often quite bad
  - Heuristics from (i) register pressure, (ii) original code order, (iii) other?

1. **Why** *we want a rewrite-based optimizer*

2. **How** *to turn a CFG into an egraph and back again*

3. **Cycles** *why they occur, and what to do about them*

4. **Results** *how well does it work?* ←

5. **Lessons** *in translating research to production*

1. Why *we want a rewrite-based optimizer*

2. **How** *to turn a CFG into an egraph and back again*

3. Cycles *why they occur, and what to do about them*

4. Results *how well does it work?*

5. Lessons *in translating research to production* ←

Efficiency

- This is *really important* in production software
  - Every percentage point counts: 1% might cost an engineer-month to regain; and costs a lot operationally at scale

Efficiency

- This is *really important* in production software
  - Every percentage point counts: 1% might cost an engineer-month to regain; and costs a lot operationally at scale
- There is inherent tension w.r.t. moving fast + experimenting to “just get results”, but “algorithm is robustly fast” is its own kind of result too

Efficiency

- This is *really important* in production software
  - Every percentage point counts: 1% might cost an engineer-month to regain; and costs a lot operationally at scale
- There is inherent tension w.r.t. moving fast + experimenting to “just get results”, but “algorithm is robustly fast” is its own kind of result too
- Robustness/predictability is important (and distinct from “fast on average”)

Efficiency

- This is *really important* in production software
  - Every percentage point counts: 1% might cost an engineer-month to regain; and costs a lot operationally at scale
- There is inherent tension w.r.t. moving fast + experimenting to “just get results”, but “algorithm is robustly fast” is its own kind of result too
- Robustness/predictability is important (and distinct from “fast on average”)
- We (practicing software engineers) need to do a better job of documenting “all the usual tricks”!

Limits Induce Creativity

- This work in Cranelift started with “standard” e-graphs and `egg`
  - When it wasn’t fast enough, I could have stopped and moved on!
  - Requires the “correct” amount of unjustified optimism

Limits Induce Creativity

- “Bottom-up” vs. “top-down” thinking
  - “I want to do eqsat” —> optimize all the computation needed for this, vs…
    - I tried this first!
  - … “I have N linear passes” —> which ideas can I keep?

Tradeoffs… and Incrementalism

- It’s OK to not solve the entire problem!
  - The only real requirement is that we run the program correctly*
- Sometimes “this is the best point on the effort Pareto curve” and we’re done

Tradeoffs… and Incrementalism

- It’s OK to not solve the entire problem!
  - The only real requirement is that we run the program correctly*
- Sometimes “this is the best point on the effort Pareto curve” and we’re done
- Sometimes, we can come up with better ideas later
  - And this happens all the time in Cranelift
  - View the codebase as a living, evolving understanding of problem domain

Tradeoffs… and Incrementalism

- *Design for incrementalism* by:
  - Building frameworks (rewrite language/infra, …)
  - Building guardrails (good testing, typesafe abstractions, well-documented invariants)

Tradeoffs… and Incrementalism

- *Design for incrementalism* by:
  - Building frameworks (rewrite language/infra, …)
  - Building guardrails (good testing, typesafe abstractions, well-documented invariants)
- Accept limits and ship, then fulfill last 20% of needs while plane is flying

Community Leverage Multipliers

- Let’s talk about “design for ___” a bit more

Community Leverage Multipliers

- Let’s talk about “design for ___” a bit more
  - Design for *community*: find abstractions that allow modular, typesafe work and enable many uses (verification!)

Community Leverage Multipliers

- Let’s talk about “design for ___” a bit more
  - Design for *community*: find abstractions that allow modular, typesafe work and enable many uses (verification!)
  - We picked up the e-graph idea because
    - It’s a clean abstraction
    - It allows modular, easy contributions of mid-end optimizations
    - It bridges the gap with academia a bit and pulls in new ideas

E-graphs… in Industry?

- Isn't this bona-fide research? Am I not a software engineer in… industry?

E-graphs… in Industry?

- Isn't this bona-fide research? Am I not a software engineer in… industry?
  - Secret: software engineering is full of research problems
    - *Caveat: pick a domain like compilers*
  - Different *kinds* of problems with different considerations

E-graphs… in Industry?

- Isn’t this bona-fide research? Am I not a software engineer in… industry?
  - Secret: software engineering is full of research problems
    - *Caveat: pick a domain like compilers*
  - Different *kinds* of problems with different considerations
  - Different approach to risk; later in pipeline, less speculative
    - (thank you for exploring e-graphs first!)

E-graphs… in Industry?

- Research is *totally* relevant to industry if it addresses industry’s needs: robust, reliable, simple, reliable, fast, reliable

E-graphs… in Industry?

- Research is *totally* relevant to industry if it addresses industry's needs: robust, reliable, simple, reliable, fast, reliable
- Industry sometimes presents opportunities to rethink key infra (e.g. compiler)
  - It can be hard to convincingly make a case for this in a vacuum in academia
  - But good reasons exist (security, simplicity, agility, …)!

E-graphs… in Industry?

- Research is *totally* relevant to industry if it addresses industry's needs: robust, reliable, simple, reliable, fast, reliable
- Industry sometimes presents opportunities to rethink key infra (e.g. compiler)
    - It can be hard to convincingly make a case for this in a vacuum in academia
    - But good reasons exist (security, simplicity, agility, …)!
    - Academia is idea-rich and searches for problems/motivations;
      Industry is problem-rich and searches for ideas/solutions
    - Bridging the two is incredibly fruitful and rewarding!

Work with Cranelift!

- We love mentoring students and collaborating with researchers
  - Verification (VeriISLE, Veriwasm, …); chaos-mode randomized testing; exceptions; typed func-refs; e-graph-based fuzzing mutators; extensions of custom DSLs; …
  - There are many open problems and the need to solve them is immediate and directly motivated
  - It’s how we can work “smarter not harder” and keep in the game, as an underdog — we all win!

Thanks!

- Links
  - https://cranelift.dev/
  - https://bytecodealliance.zulipchat.com/
  - https://cfallin.org/