CS410P/510 Programming Language Compilation

Winter 2024

Lecture on Optimization

CODE OPTIMIZATION

• Really “improvement” rather than “optimization;” results are seldom optimal.

• Remove inefficiencies in user code and (at least as importantly) in compiler-generated code.

• Can be applied at several phases in pipeline, chiefly on intermediate or assembly code.

• Goal is usually to decrease execution time; sometimes it is important to decrease code size.

• Can operate at several levels of granularity:

- **“Local”** : within basic blocks

- **“Global”** : entire functions

- **“Interprocedural”** : entire programs (maybe even multiple source files)

• **Most** of a serious modern compiler is devoted to optimization.

Some Important Classic Optimizations

- Constant folding (partial evaluation)
- Constant propagation
- Dead code elimination
- Useless code elimination
- Common subexpression elimination (redundancy elimination)
- Invariant hoisting from loops
- Strength reduction (replacing an expensive operation with an equivalent cheaper one)
- Function inlining

We hope for modest constant factor improvements in running time and code space.

Asympototic improvements generally require a different algorithm, which the compiler is very unlikely to discover!

KEY CONSIDERATIONS FOR ANY OPTIMIZATION

- Safety: Transformation **must** maintain observable behavior of program (on all inputs).
- Profitability: Transformation **should** speed up execution (or shrink code size, or both).
- Opportunity: We need an efficient way to find out where we can apply transformation safely and profitably.

THEORETICAL UNDERPINNINGS

- Mostly about finding static **approximations** to dynamic behavior.
- But a very *ad hoc* subject, with relatively few unifying principles!

Some useful tools:

- Control-flow graphs
- Data-flow analysis
- Pointer analysis
- Dominators
- Static single assignment
- Polyhedral analysis

OPPORTUNITIES FOR OPTIMIZATION

• Compensating for abstractions in source language code, e.g.

- Array access requires non-trivial address calculations

- Object-oriented languages call many small methods, using expensive dynamic dispatch

• Utilizing resources in target code, e.g.

- Certain processors have specialized instructions for common patterns

- Co-processors (GPUs, etc.) – although today few compilers use these automatically

(Note: modern CPUs do lots of dynamic optimization in hardware, which may lessen the importance/impact of compiler optimization.)

• Optimization opportunities are **cumulative**: doing one transformation often enables others.

Where are Optimizations Done?

• On assembly code

- particularly useful when optimization depends on details of target machine architecture

• At source level, by rewriting the program.

- e.g., like our $\mathcal{L}_{\text{Int}}$ PartialEvaluation pass

- can be limited by expressiveness of source language

• On an intermediate language, typically with explicit control flow structure and unlimited registers and memory

- e.g. our $\mathcal{C}_{\text{If}}$

- portable across different target (and source!) languages

- often the “sweet spot” for general-purpose optimizations

LOCAL OPTIMIZATIONS

These work within a single basic block, so control flow is trivial.

Some simple examples in $\mathcal{C}_{\text{lf}}$:

- Constant folding; static conditional evaluation

```
L1: a = 2 + 3                    L1: a = 5
    if a > 4 goto L2        ⇒        goto L2
    else goto L3
```

- Together with constant propagation

```
L1: a = -9                  ⇒    L1: a = -9
    b = 10 + a                       b = 1
    c = c + b                        c = c + 1
```

(Subsequent transformations might get rid of the assignment to `b`.)

"PEEPHOLE" OPTIMIZATIONS

Local optimizations performed on machine code using minimal context information, e.g.

- Algebraic Simplification

```
addq $0, %rsp                    ⇒        (nothing)
```

- Redundant load or store removal

```
movq -20(%rbp), %r10             ⇒        movq -20(%rbp), %r10
movq %r10, -20(%rbp)
```

- Strength Reduction

```
imulq $8, %r10                   ⇒        salq $3, %r10
```

- Use of machine idioms

```
imulq $8, %r10                   ⇒        leaq 20(%r11,%r10,8), %r10
addq %r11, %r10
addq  $20,%r10
```

WHOLE-PROCEDURE OPTIMIZATIONS

- Consider entire control-flow graph (CFG) of procedure instead of just one basic block at a time.
- Typically requires deeper analysis of code (e.g. data-flow analysis).
- Example: Reorder blocks; remove jumps to jumps; remove unreachable code

```
L1: cmpx x,2          ⇒      L1: cmpx x,2
    jl L2                        jl L4
    jmp L3                 L3: addx $1,x
L2: jmp L4                       jmp L1
    jmp L1                 L4: ...
L3: addx $1,x
    jmp L1
L4: ...
```

MORE WHOLE-PROCEDURE OPTIMIZATIONS

- Example: Remove useless code (e.g. stores to non-live variables)

```
L1: x = x + 1          ⇒      L1: y = z + w
    y = z + w                     x = y + z
    x = y + z                     if x < 10 goto L1;
    if x < 10  goto L1;           else goto L2
    else goto L2
```

LOOP OPTIMIZATIONS

Loop optimizations are most important whole-procedure transformations.

- **Code motion**: “hoist” expensive calculations above the loop.
- Use **induction variables** and reduction in strength. Change only one index variable on each loop iteration, and choose one that’s cheap to change.
- Partially **unrolling** the loop can reduce per-iteration overheads and improve instruction scheduling.

Loop Optimization Example

Illustrating hoisting and strength reduction.

```
  x = 0
  while x < 1000:
    a[x] = a[y]
    x = x + 1
```

```
L0: x = 0
L1: if x < 1000 goto L2
    else goto L3
L2: j = y * 8
    u = a + j
    v = *u
    i = x * 8
    t = a + i
    *t = v
    x  = x + 1
    goto L1
L3: ...
```

$\Rightarrow$

```
L0: t = a
    j = y * 8
    u = a + j
    w = a + 8000
L1: if t < w goto L2
    else goto L3
L2: v = *u
    *t = v
    t = t + 8
    goto L1
L3: ...
```

INTERPROCEDURAL OPTIMIZATION

Procedure inlining is most important.

- Replace a procedure call with a copy of the procedure body (including initial assignments to parameters).
- Applicable when body is not too big, or is called only once.

Benefits:

- Saves overhead of procedure entry/exit, argument passing, etc.
- Permits other optimizations to work over procedure boundaries.
- Particularly useful for languages that encourage use of small procedures (e.g. OO state get/set methods).

Cost:

- Risk of “code explosion.”
- Doesn’t work when callee is not statically known (e.g. OO dynamic dispatch or FP first-class calls).

COMPILER CORRECTNESS

Optimizing compilers are complex artifacts, and they have bugs!

Some promising approaches to enhancing compiler correctness:

- Randomized testing (can find dark corners that human-written tests may miss)
- Formal verification of correctness using machine-assisted theorem proving

REDUNDANCY ELIMINATION

One important optimization opportunity is removing repeated calculations of the same value.

For remainder of lecture, we consider how this can be done at different levels of granularity.

Consider this $\mathcal{C}_{\text{lf}}$ sequence:

```
g = x + y
h = u - v
w = g + h
u = x + y   # redundant calculation
x = u - v   # not redundant (why not?)
```

Value numbering is an approach to finding and eliminating **common subexpressions**

- Process each instruction in order.
- Maintain a mapping from identifiers (e.g. `x`) and arithmetic expressions (e.g. `(#1 + #2)`) to **value numbers**.
- If an entry in the mapping already exists, rewrite the instruction to use it.

LOCAL VALUE NUMBERING

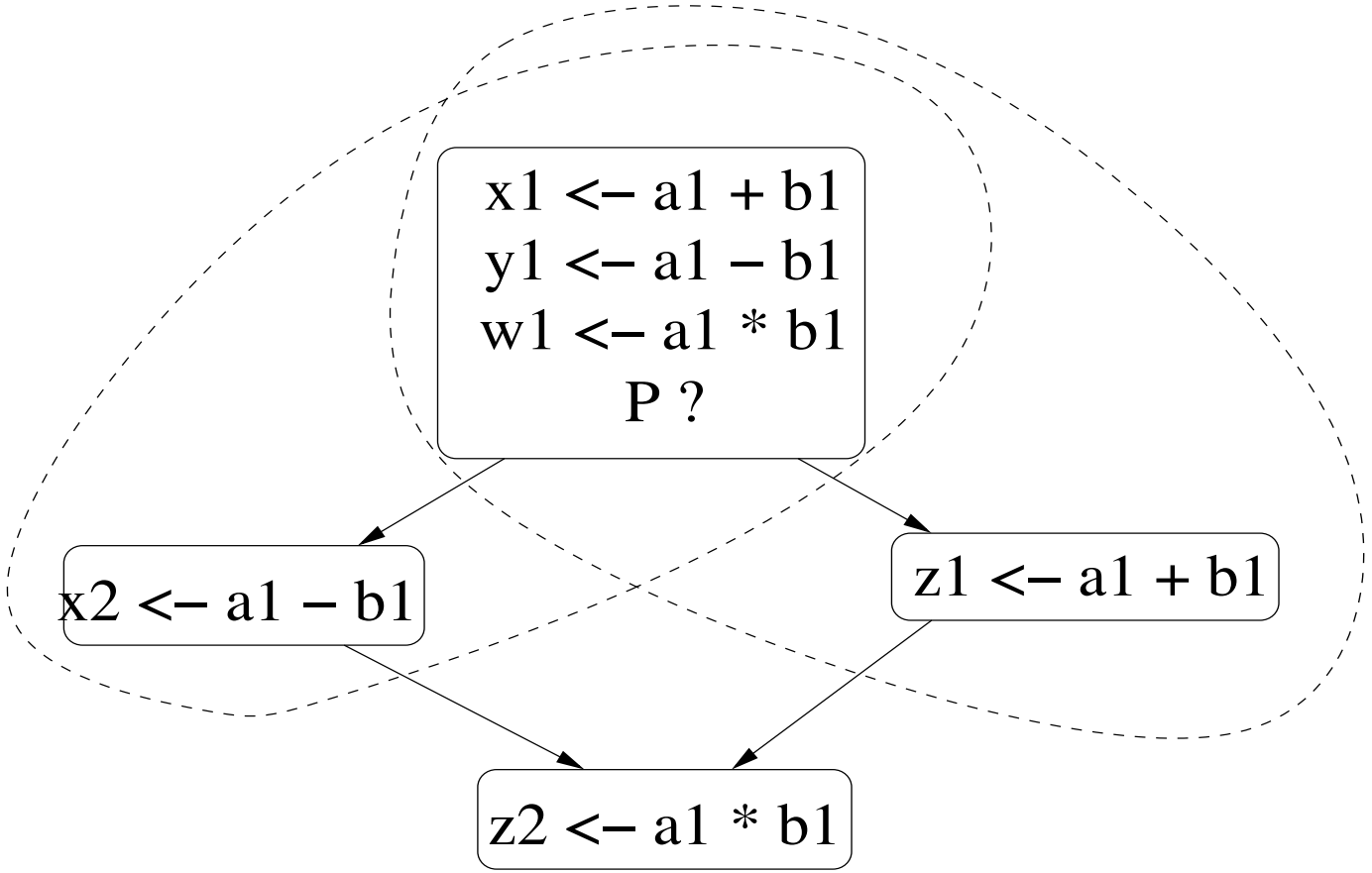
Initial code	Final code	Mapping entries	
`g = x + y`	`g = x + y`	`x -> #1`	`#1:x`
		`y -> #2`	`#2:y`
		`(#1 + #2) -> #3`	
		`g -> #3`	`#3:g`
`h = u - v`	`h = u - v`	`u -> #4`	`#4:u`
		`v -> #5`	`#5:v`
		`(#4 - #5) -> #6`	
		`h -> #6`	`#6:h`
`w = g + h`	`w = g + h`	`(#3 + #6) -> #7`	
		`w -> #7`	`#7:w`
`u = x + y`	`u = g`	`u -> #3`	
`x = u - v`	`x = u - v`	`(#3 - #5) -> #8`	
		`x -> #8`	`#8:x`

- Now can potentially replace uses of `u` by `g` and eliminate the assignment `u = g`.
- This scheme works better when all names are assigned just once.

SUPERLOCAL VALUE NUMBERING

Can do better by analyzing over paths in **extended** basic blocks.

(An EBB has one entry, but can have multiple exits. It forms a subtree of the CFG; all the blocks in the EBB except perhaps the root have a unique predecessor inside the EBB).



DOMINATORS

We still aren't taking full advantage of facts of the form "this instruction is certain to be executed before this other instruction." Capture this idea using **dominators**.

To define dominators, assume that CFG has a distinguished start node S, and has no disconnected subgraphs (nodes unreachable from S).

Then we say node d **dominates** node n if **all** paths from S to n include d.

(In particular, every node dominates itself.)

Fact: d dominates n iff $d = n$ or d dominates all predecessors of n.

So can define the set $D(n)$ of nodes that dominate n as follows:

- $D(S) = \{S\}$
- $D(n) = \{n\} \cup (\bigcap_{p \in pred(n)} D(p))$

where $pred(n) =$ set of predecessors of n in CFG.

DOMINATOR TREE

The **immediate dominator** of n, $idom(n)$, is defined thus:

- $idom(n)$ dominates n
- $idom(n)$ is not n
- $idom(n)$ does not dominate any other dominator of n (except n itself)

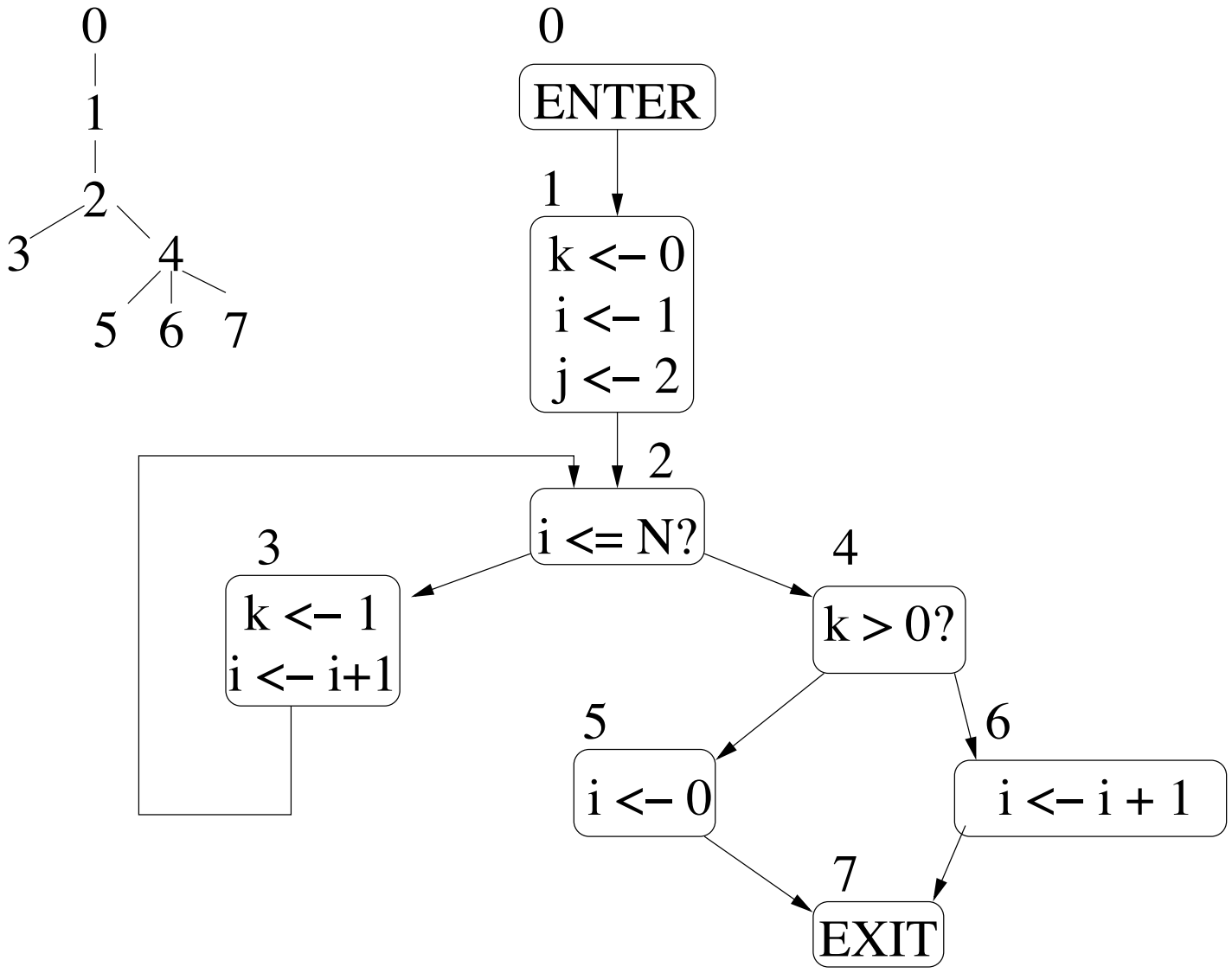
Fact: every node (except S) has a unique immediate dominator

Hence the immediate dominator relation defined a tree, called the **dominator tree**, whose nodes are the nodes of the CFG, where the parent of a node is its immediate dominator.

Have $D(n) = \{n\} \cup (\text{ancestors of } n \text{ in dominator tree})$

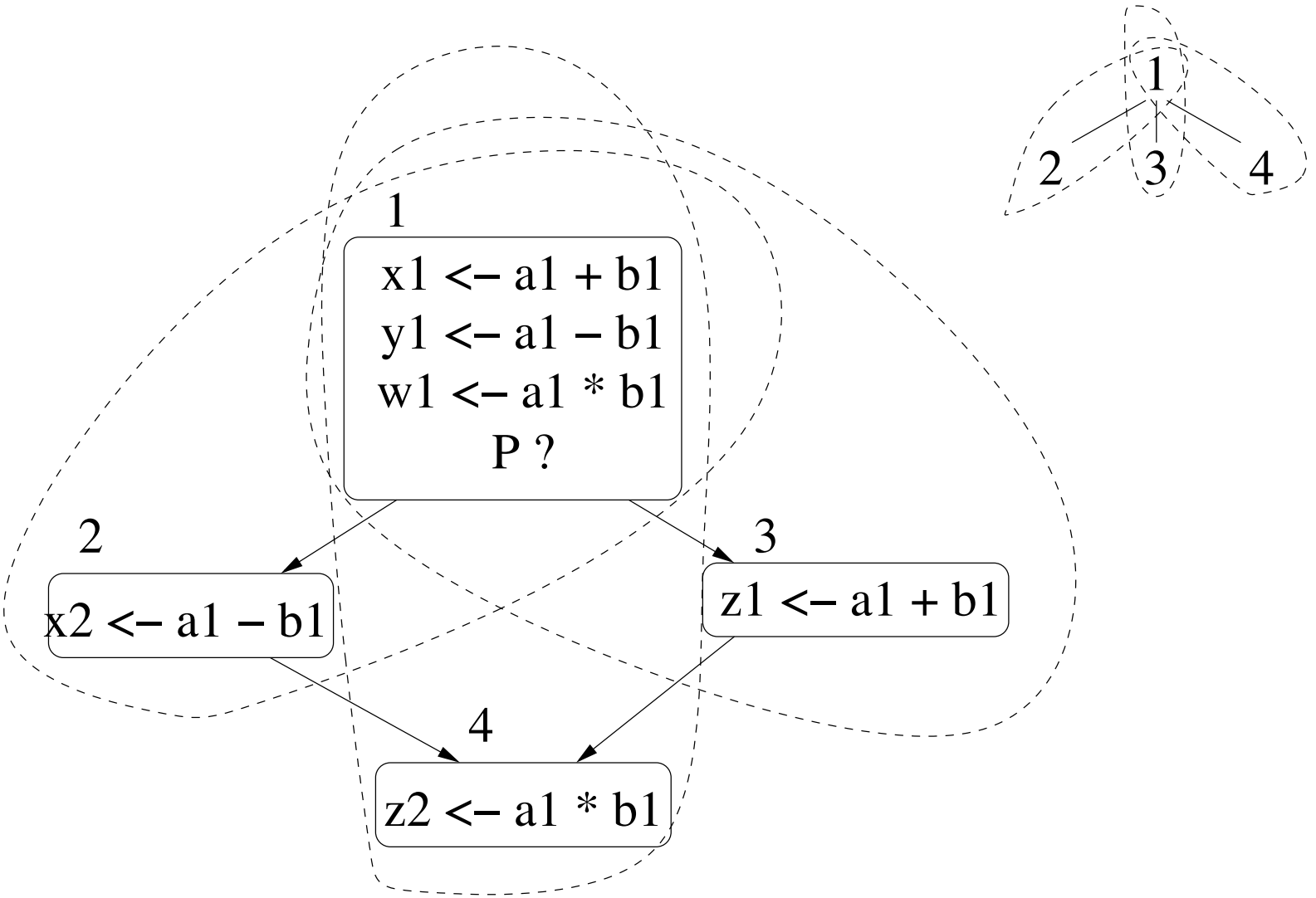
(Nontrivial) Fact: The dominator tree of a CFG can be computed in almost-linear time.

DOMINATOR TREE EXAMPLE



DOMINATOR-BASED VALUE NUMBERING

Do analysis over paths in dominator tree.



AVAILABLE EXPRESSIONS

Even with dominator-based VN, we cannot find redundant expressions computed on **different** paths.

An alternative approach is to compute **available expressions**.

An expression e is **available** at node n if on **every** path from S to n, e is evaluated and none of its constituent variables is redefined between that evaluation and n.

If an expression is available at a node where it is being recomputed, it is possible to replace the recomputation by a variable representing the result of the previous computation.

This is a classic **data flow analysis** problem, specified thus:

$gen(\texttt{t <- b bop c}) = \{\texttt{b bop c}\}$ $\quad$ $kill(\texttt{t <- \_}) = \bigcup_{\forall u,bop}\{\texttt{t bop u}, \texttt{u bop t}\}$

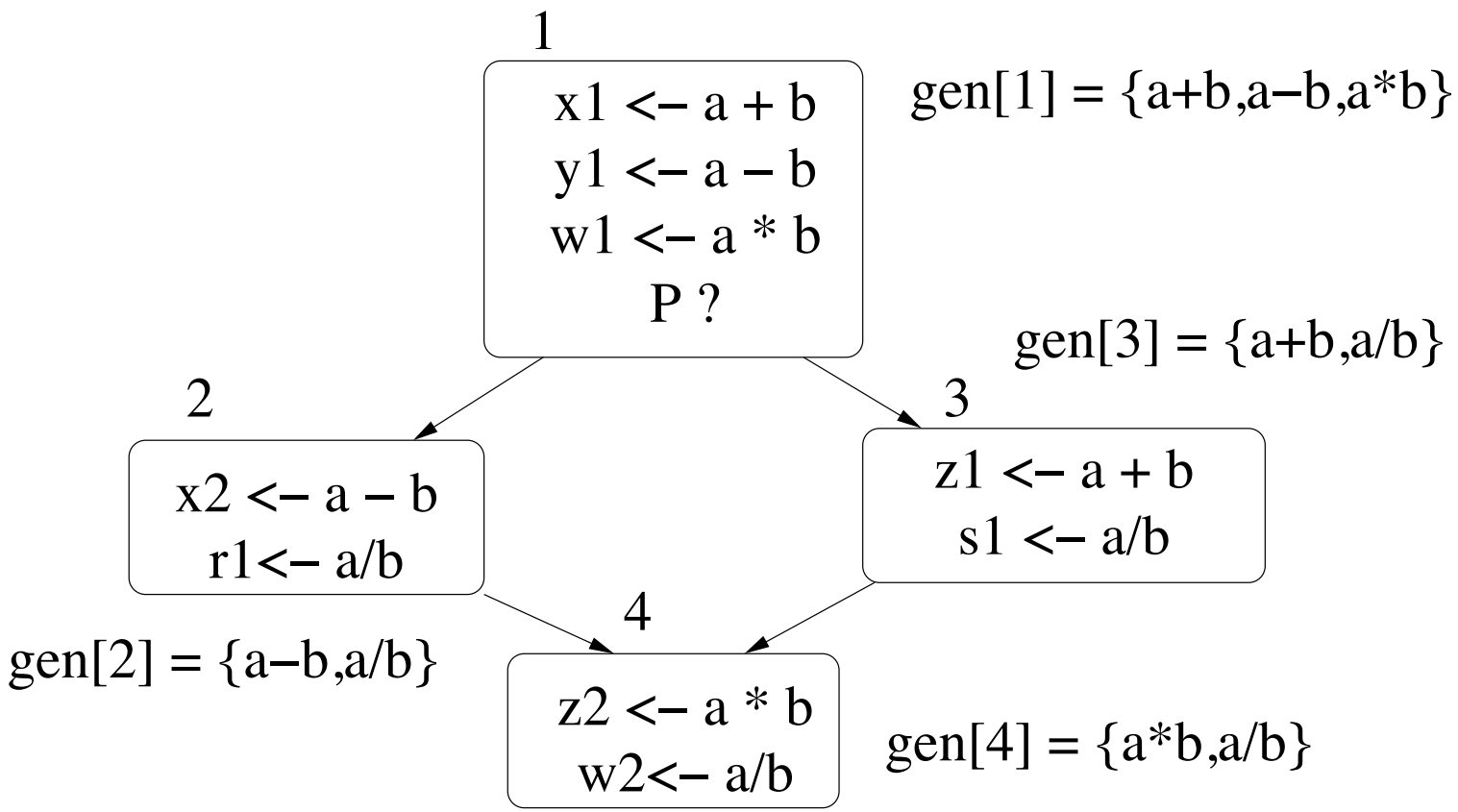
$gen(\textit{other instruction}) = \emptyset$ $\quad$ $kill(\textit{other instruction}) = \emptyset$

$in(n) = \bigcap_{p \in pred(n)} out(p)$

$out(n) = (in(n) \cup gen(n)) - kill(n)$

Here we want $in(n)$, the set of expressions available on entry to n.

AVAILABLE EXPRESSIONS EXAMPLE



SOLUTIONS

This is a forwards data flow problem, with initial approximation

```
in[1] = ∅
in[2] = in[3] = in[4] = {a+b,a-b,a*b,a/b}
```

Here's the (unique) solution to the data flow equations.

```
in[1] = {}                      out[1] = {a+b,a-b,a*b}
in[2] = {a+b,a-b,a*b}           out[2] = {a+b,a-b,a*b,a/b}
in[3] = {a+b,a-b,a*b}           out[3] = {a+b,a-b,a*b,a/b}
in[4] = {a+b,a-b,a*b,a/b}       out[4] = {a+b,a-b,a*b,a/b}
```

So nothing needs to be recomputed in nodes 2, 3, or 4.

FOR FURTHER INFORMATION

• Keith Cooper and Linda Torczon, *Engineering a Compiler*, 2nd ed., Morgan Kaufmann, 2012, has thorough and practical coverage of many standard optimizations. (The slides on redundancy analysis are inspired by their treatment.)

• Steve Muchnick, *Advanced Compiler Design & Implmentation*, Morgan Kaufmann, 1997, is the most encyclopedic treatment of the optimization ecosystem.

• Anders Møller and Michael Schwartzbach, *Static Program Analysis*, on-line at `https://cs.au.dk/~amoeller/spa/spa.pdf`, 2020, treats the theoretical underpinnnings of the analyses that drive optimization.

• Xavier Leroy, "Formal verification of a realistic compiler,"*Commun. ACM*, 52(7), pp. 107–115, 2009, describes the CompCert verified C compiler.