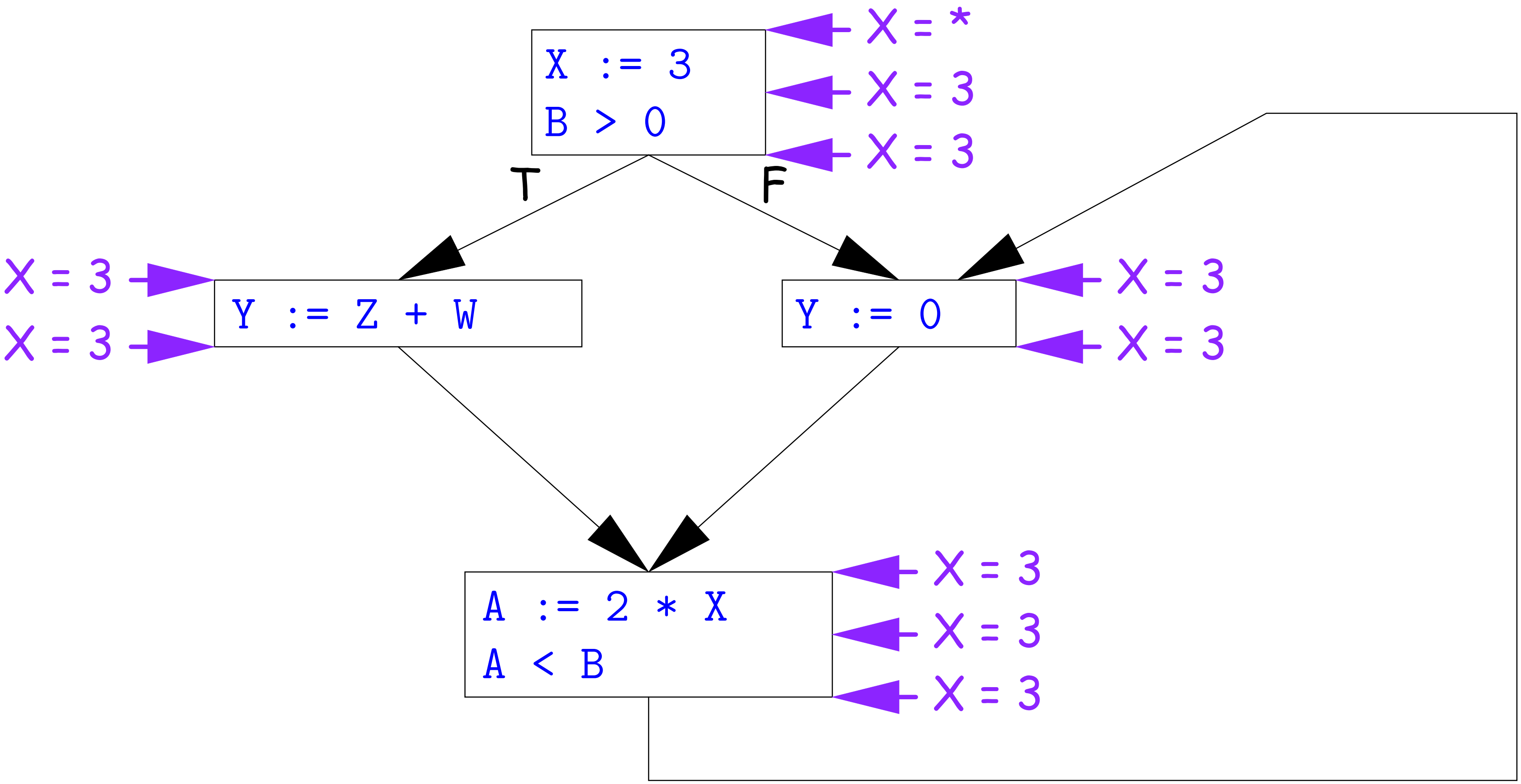
Lecture 25: Code Optimization, Part 2

[Adapted in part from notes by R. Bodik and G. Necula]

Liveness Analysis

Once constants have been globally propagated, we would like to eliminate dead code



After constant propagation, `X := 3` is dead code (assuming this is the entire CFG)

Terminology: Live and Dead

- In the program

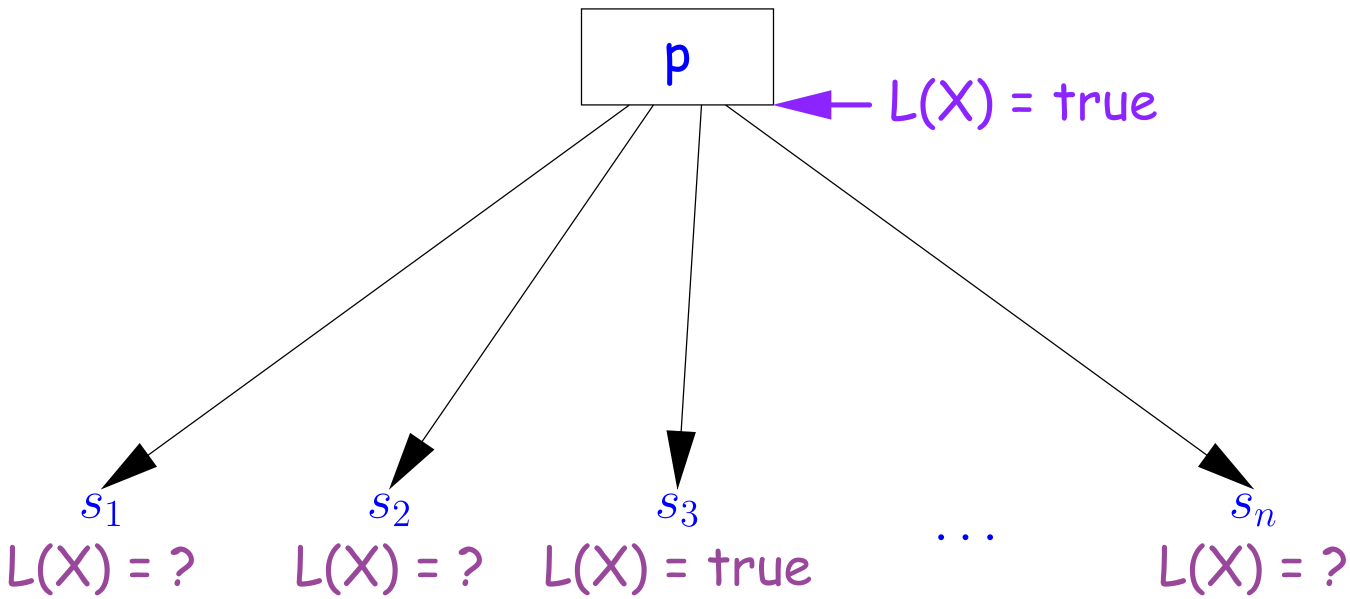
```
X := 3;  /*(1)*/  X = 4;  /*(2)*/  Y := X  /*(3)*/
```

- the variable `X` is *dead* (never used) at point (1), *live* at point (2), and may or may not be live at point (3), depending on the rest of the program.
- More generally, a variable `x` is live at statement `s` if
  - There exists a statement `s'` that uses `x`;
  - There is a path from `s` to `s'`; and
  - That path has no intervening assignment to `x`
- A statement `x := ...` is dead code (and may be deleted) if `x` is dead after the assignment.

Computing Liveness

- We can express liveness as a function of information transferred between adjacent statements, just as in copy propagation
- Liveness is simpler than constant propagation, since it is a boolean property (true or false).
- That is, the lattice has two values, with `false<true`.
- It also differs in that liveness depends on what comes *after* a statement, not before—we propagate information *backwards* through the flow graph, from Lout (liveness information at the end of a statment) to Lin.

Liveness Rule 1

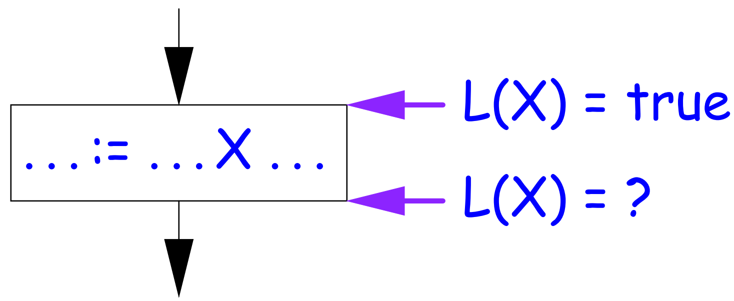


- So

  Lout(x, p) = lub { Lin(x, s) such that p is a predecessor of s }.

- Here, least upper bound (lub) is the same as "or".

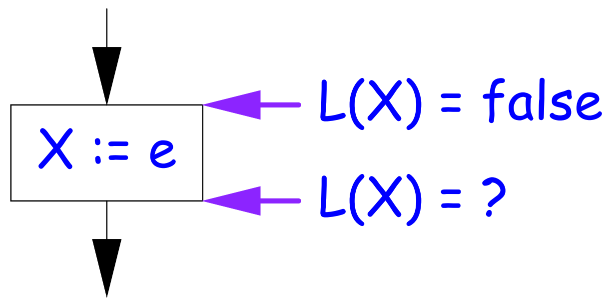
Liveness Rule 2



Lin(X, s) = true if s uses the previous value of X.

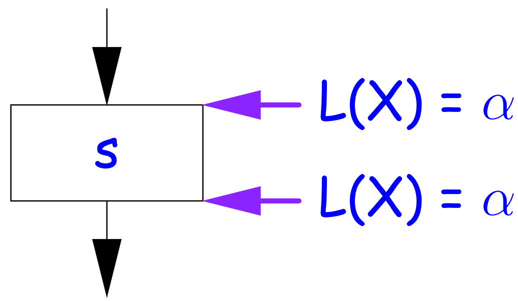
- The same rule applies to any other statement that uses the value of X, such as tests (e.g., `X < 0`).

Liveness Rule 3



Lin(X, X := e) = false if `e` does not use the previous value of `X`.

Liveness Rule 4

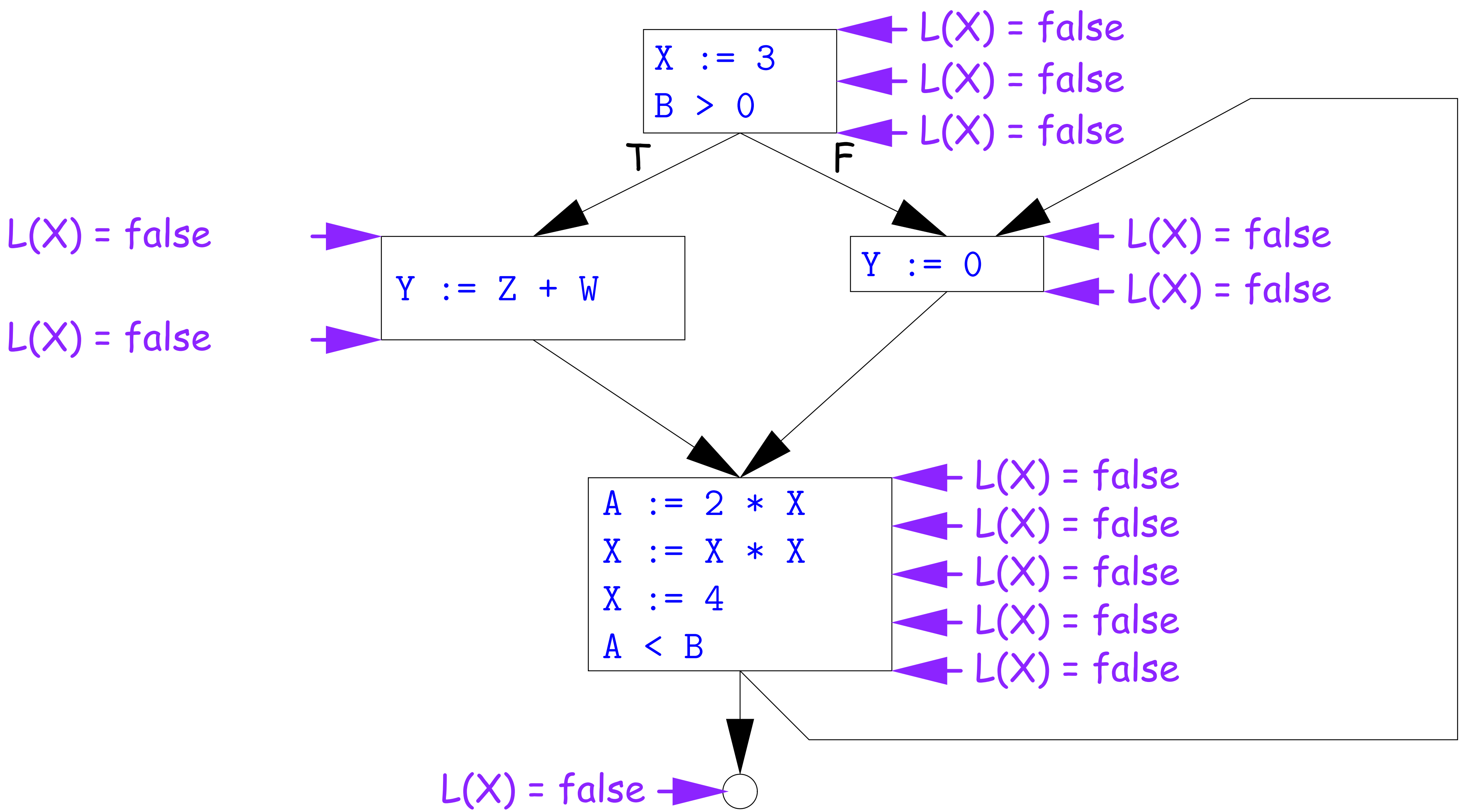


Lout(X, s) = Lin(X, s) if `s` does not mention `X`.

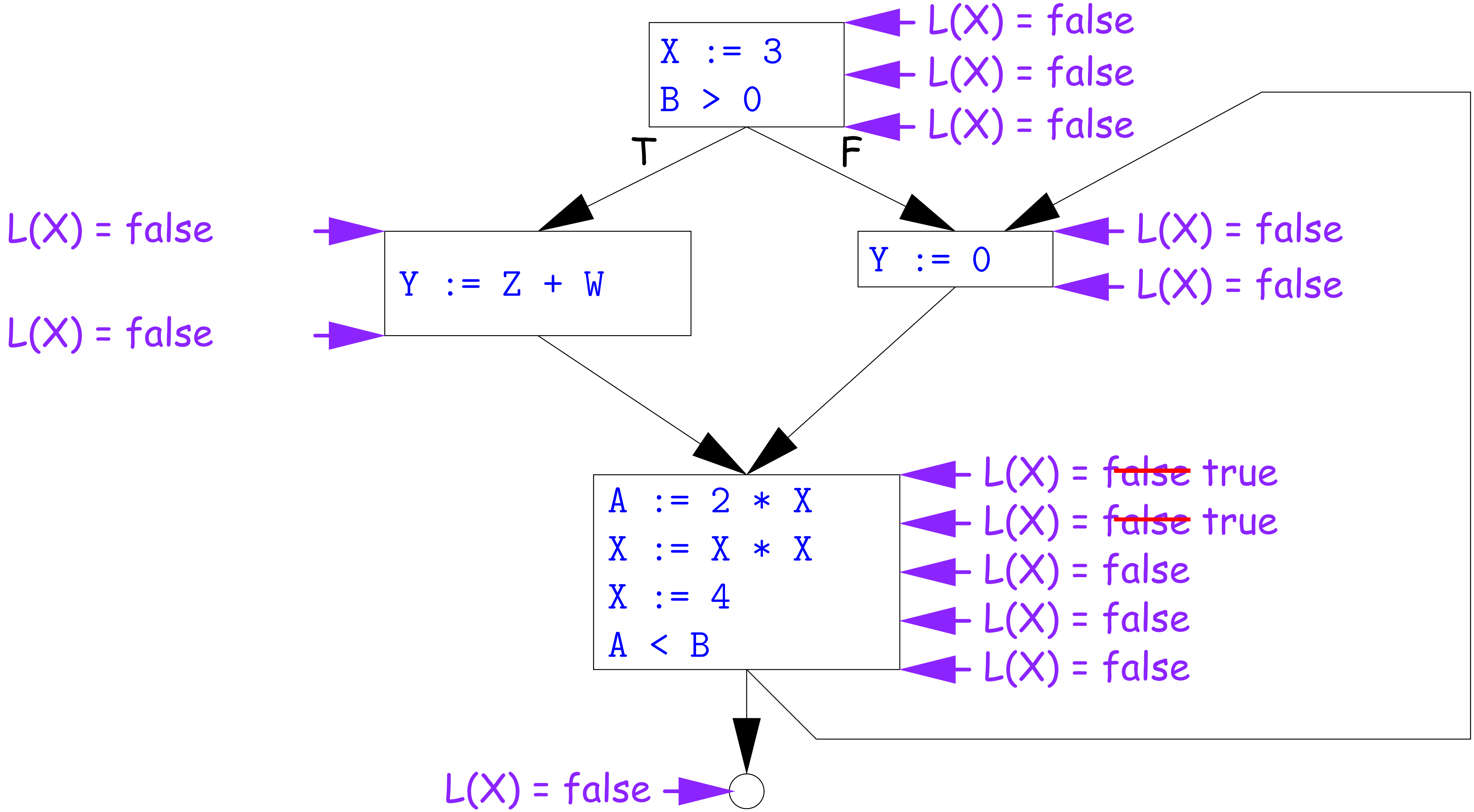
Propagation Algorithm for Liveness

- Initially, let all Lin and Lout values be false.
- Set Lout value at the program exit to true iff x is going to be used elsewhere (e.g., if it is global and we are analyzing only one procedure).
- As before, repeatedly pick s where one of 1–4 does not hold and update using the appropriate rule, until there are no more violations.
- When we're done, we can eliminate assignments to X if X is dead at the point after the assignment.

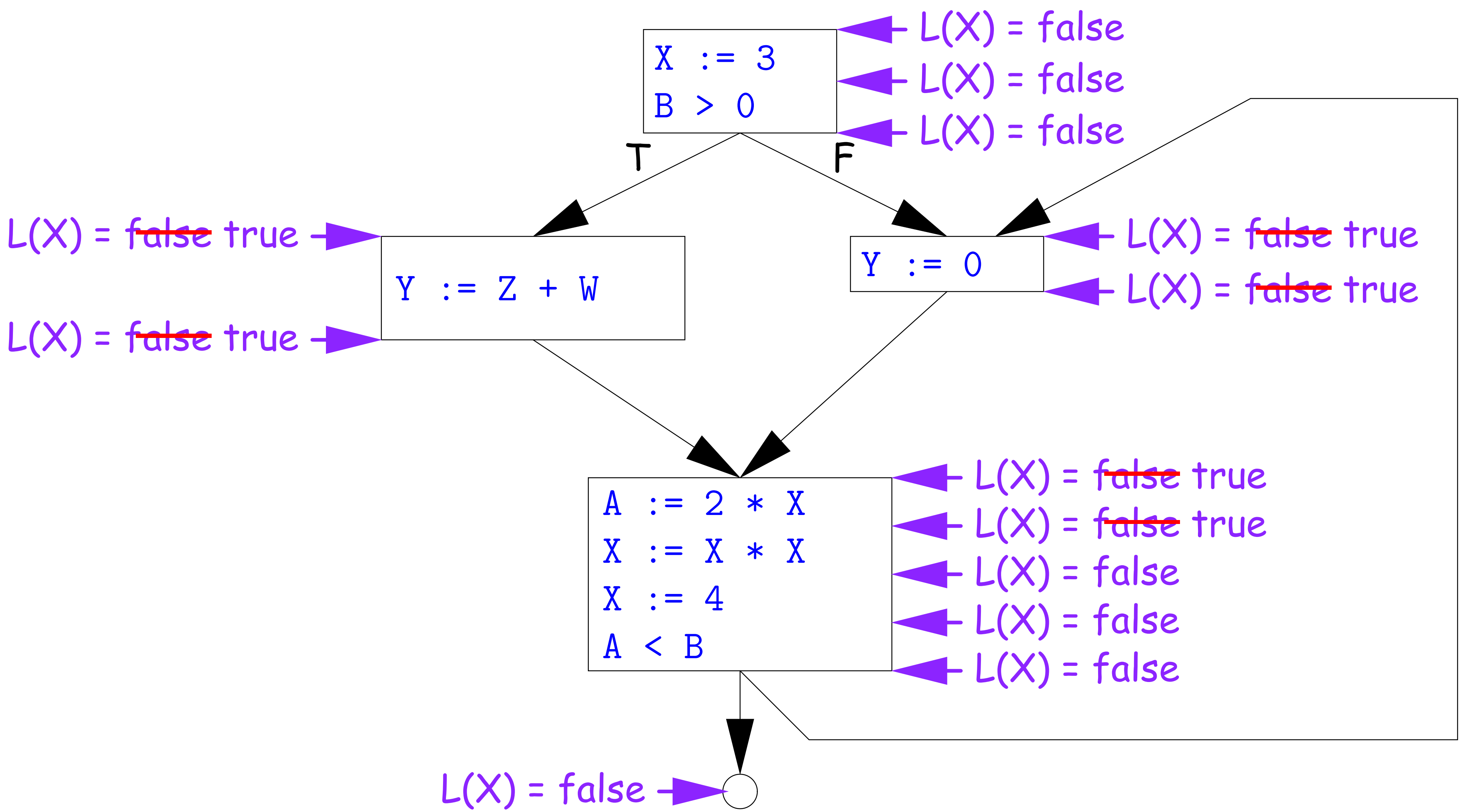
Example of Liveness Computation



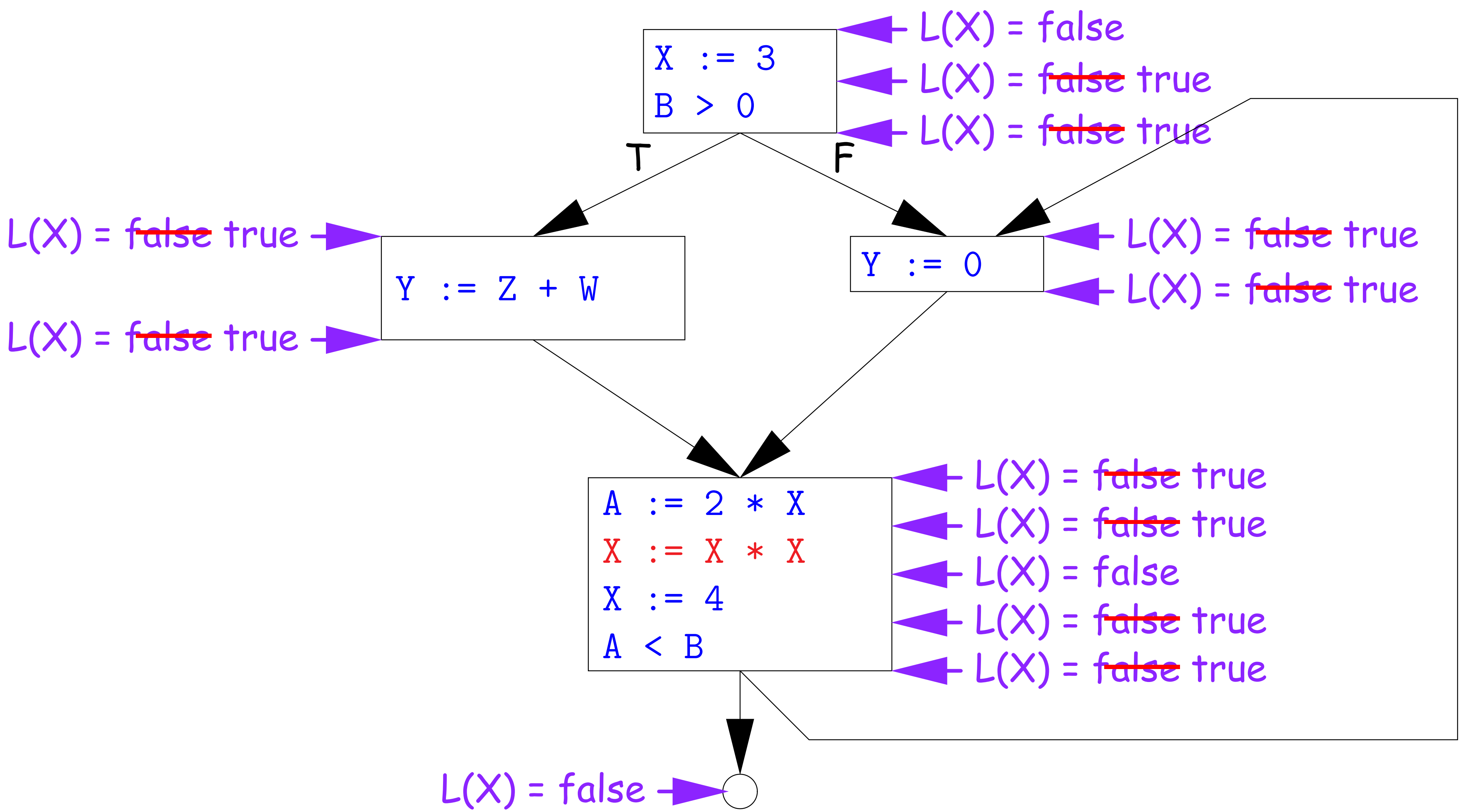
Example of Liveness Computation



Example of Liveness Computation



Example of Liveness Computation



Termination

- As before, a value can only change a bounded number of times: the bound being 1 in this case.
- Termination is guaranteed
- Once the analysis is computed, it is simple to eliminate dead code, but having done so, we must recompute the liveness information.

SSA and Global Analysis

- For local optimizations, the single static assignment (SSA) form was useful.
- But applying it to a full CFG is requires a trick.
- E.g., how do we avoid two assignments to the temporary holding `x` after this conditional?

```
if a > b:
    x = a
else:
    x = b
# where is x at this point?
```

- Answer: a small kludge known as ϕ "functions"
- Turn the previous example into this:

```
if a > b:
    x1 = a
else:
    x2 = b
x3 = φ(x1, x2)
```

ϕ Functions

- An artificial device to allow SSA notation in CFGs.
- In a basic block, each variable is associated with one definition,
- ϕ functions in effect associate each variable with a *set of possible definitions.*
- In general, one tries to introduce them in strategic places so as to minimize the total number of ϕs.
- Although this device increases number of assignments in IL, register allocation can remove many by assigning related IL registers to the same real register.
- Their use enables us to extend such optimizations as CSE elimination in basic blocks to *Global CSE Elimination.*
- With SSA form, easy to tell (conservatively) if two IL assignments compute the same value: just see if they have the same right-hand side. The same variables indicate the same values.

Loops

- In a CFG, a loop is simply a set of basic blocks, L, containing an entry block, e, such that
  - All paths from the entry node of the entire CFG to a block in L include e;
  - All predecessors of a node in L are also in L (except for e, which must have a predecessor outside L).
  - Every node in L has a path in L back to e.
- Here, for example,

```
j = i+1;
while (j < N)
    A[j] = A[j] / A[i]
```

The entry node contains the test `j < n` and the rest of the loop is the node containing the assigment to `A[j]`, which then loops back to the entry.

Invariant Code Motion

- Consider the loop

```
while (i < N)
    A[i] = A[i] + j * x;
```

- Since `j * x` does not change in the loop, we can rewrite this as

```
tmp = j * x;
while (i < N)
    A[i] = A[i] + tmp;
```

- This is an example of *invariant code motion out of a loop*.
- What tells us that `j*x` does not change?
- We see that all assignments to `j` and `x` that apply at the point where the product is computed are outside the loop.
- And this we can get by observing where the assignments to the SSA-form for those variables are.

Code Motion Caveat

- Code motion is not always appropriate.
- If the code to be moved, has side effects, or might cause an exception, could change the results.
- If the code is expensive, you will increase the time required for the program when the loop is not executed.
- Hence, you will see compilers rewrite loops like this:

```
if (i < N) {
    /* Preheader */
    while (i < N)
        A[i] = A[i] + j * x;
}
```

where `Preheader` marks a spot where the compiler can insert a new block to hold code moved out of the loop.

Summary

- We've seen two kinds of analysis:
  - Constant propagation is a *forward analysis:* information is pushed from inputs to outputs.
  - Liveness is a *backwards analysis*: information is pushed from outputs back towards inputs.
- But both make use of essentially the same algorithm.
- Numerous other analyses fall into these categories, and allow us to use a similar formulation:
  - An abstract domain (abstract relative to actual values);
  - Local rules relating information between consecutive program points around a single statement; and
  - Lattice operations like least upper bound (or *join*) or greatest lower bound (or *meet*) to relate inputs and outputs of adjoining statements.

Register Allocation

- Memory Hierarchy Management
- Register Allocation:
  - Register interference graph
  - Graph coloring heuristics
  - Spilling
- Cache Management

The Memory Hierarchy

Computers employ a variety of memory devices, trading off capacity, persistence, and speed (some years ago):

Device	Access time (latency)	Capacity
Registers	1 cycle	256–2000 bytes
Cache	2–5 cycles	256KB–16MB
Main memory	100 cycles	32MB — $>$16GB
Disk	20K–10M cycles	10GB — >1TB

Managing the Memory Hierarchy

- Programs are written as if there are only two kinds of memory: main memory and disk (variables and files).
- Programmer is responsible for moving data from disk to memory.
- Hardware is responsible for moving data between memory and caches
- Compiler is responsible for moving data between memory and registers (which the programmer usually doesn't see).
- Cache and register sizes are growing slowly: important to manage them well.
- The cost of a cache miss is growing, and the widening gap is bridged with more caches.

The Register Allocation Problem

- Our three-address code style uses temporaries profligately, simplifying code generation and optimization, but complicating final translation to assembly
- Hence, the register allocation problem:

  Rewrite the intermediate code to use fewer temporaries than there are machine registers

- So we must assign more temporaries to a register, without changing the program behavior

An Example

Consider the program

```
a := c + d
e := a + b
f := e - 1
```

assuming that assumption that `a` and `e` die after use. Then,

- Can reuse `a` after `a + b`
- Same with temporary `e` after `e - 1`
- Can allocate `a`, `e`, and `f` all to one register (r1):

```
r1 := c + d
r1 := r1 + b
r1 := r1 - 1
```

Basic Register Allocation Idea

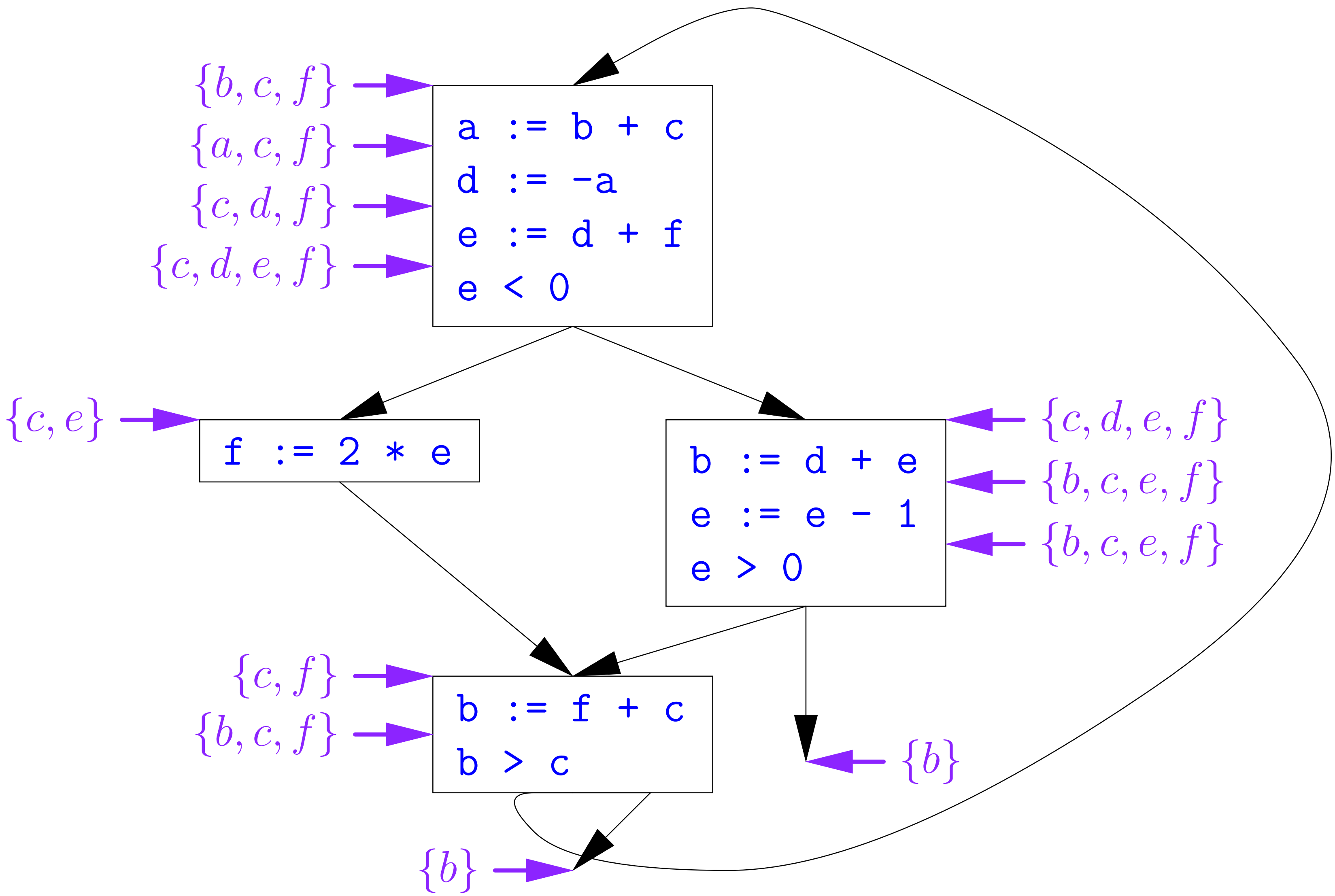
- So in general, since the value in a dead temporary is not needed for the rest of the computation,

  Any set of temporaries can share a single physical register if at most one is alive at any program point.

- This rule is easy to apply to basic blocks. General CFGs are considerably trickier.

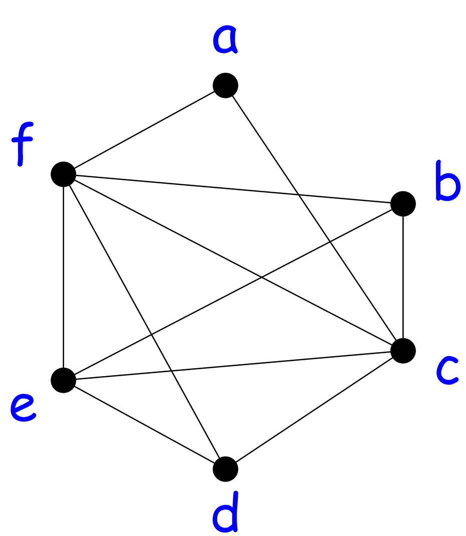
Going Global: Allocation in CFGs (I)

First step is to compute live variables before each statement. In this example, assume that variable b is live at exit.



Allocation in CFGs (II): Register Interference Graphs

- The sets in the previous slide indicate sets of virtual registers that are simultaneously alive at all points in the program, and therefore cannot share a physical register.
- Can summarize all these sets by constructing an undirected graph with a node for each virtual register, and an edge between any two virtual registers that appear together in the same set somewhere in the program.
- Call this the *register interference graph (RIG)*.



- The RIG extracts exactly the information needed to characterize legal register assignments
- Gives global (over the entire CFG) picture of the register requirements

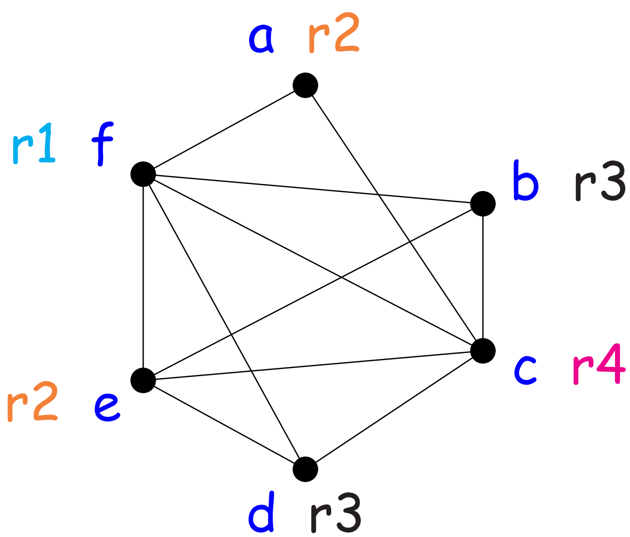
Allocation in CFGs (III): Graph Coloring

- A *coloring* of a graph is an assignment of colors to nodes, such that nodes connected by an edge have different colors.
- A graph is *k-colorable* if it has a coloring with k colors.
- In our problem, *colors = registers.* That is,

  If we have k available machine registers and our register interference graph is k-colorable, then the coloring gives us a register assignment.

Graph Coloring: Example

Consider the sample RIG:



- There is *no* coloring with fewer than 4 colors
- There *are* 4-colorings of this graph

Before .	After
`A: a := b + c`	`A: r2 := r3 + r4`
`d := -a`	`r3 := -r2`
`e := d + f`	`r2 := r3 + r1`
`if e >= 0 jump C`	`if r1 >= 0 jump C`
`B: f := 2 * e`	`B: r1 := 2 * r2`
`jump D`	`jump D`
`C: b := d + e`	`C: r3 := r3 + r2`
`e := e - 1`	`r2 := r2 - 1`
`if e <= 0 jump E`	`if r2 <= 0 jump E`
`D: b := f + c`	`D: r3 := r1 + r4`
`if b <= c jump A`	`if r3 <= r4 jump A`
`E:`	`E:`

Allocation in CFGs (III): Computing Graph Colorings

- The remaining problem is to compute a coloring for the interference graph.
- Unfortunately, this problem is hard (NP-hard). No guaranteed fast algorithms are known,
- And besides, a coloring might not exist for a given number of registers.
- For (1), we'll use heuristics.

Graph Coloring Heuristic: Motivation

- Observation:
  - Pick a node t with $< k$ neighbors in RIG.
  - Eliminate t and its edges from RIG.
  - If the resulting graph has a k-coloring then so does the original graph.
- Reason: whatever $n \leq k-1$ colors t's neighbors have, we know we'll always be able to color t (since there are k colors). Therefore, eliminating t cannot affect the colorability of the other nodes.

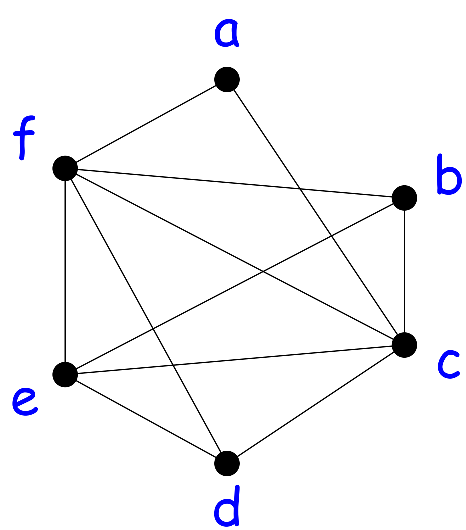
Graph Coloring Heuristic

The following works well in practice:

- Pick a node t with $< k$ neighbors.
- Push t on a stack and remove it from the RIG.
- Repeat until the graph has no nodes.
- Then start popping nodes from the stack and adding them back to the graph, assigning colors to each as we go (starting with the last node added).
- At each step, we know we can pick a color different from those assigned to already colored neighbors, by the observation on the last slide.

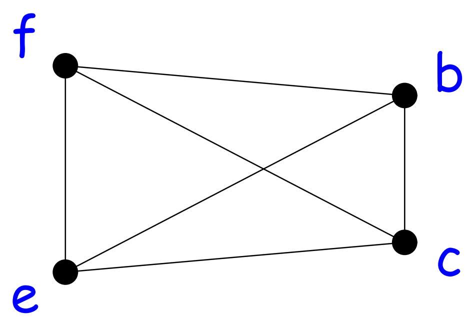
Example of Using the Heuristic (I)

Start with our sample RIG and with $k = 4$:



Stack: []

Now remove a and then d, giving

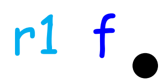


Stack: [d, a] (top on left)

Now all nodes have < 4 neighbors; remove. Stack is [f, e, b, c, d, a].

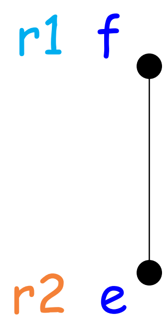
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.
- At each step, guaranteed there's a free register.



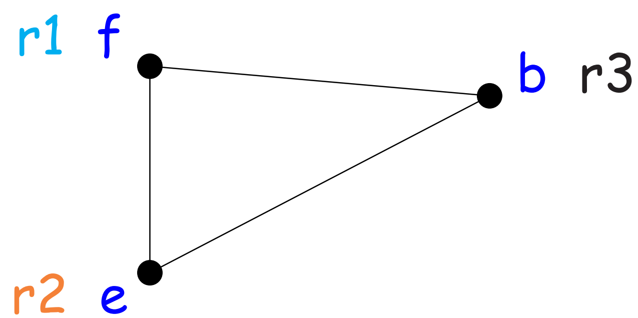
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.
- At each step, guaranteed there's a free register.



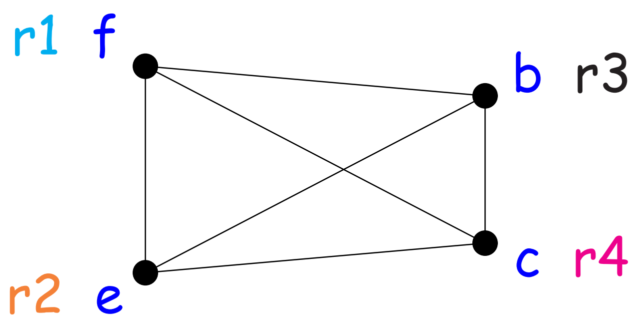
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.
- At each step, guaranteed there's a free register.



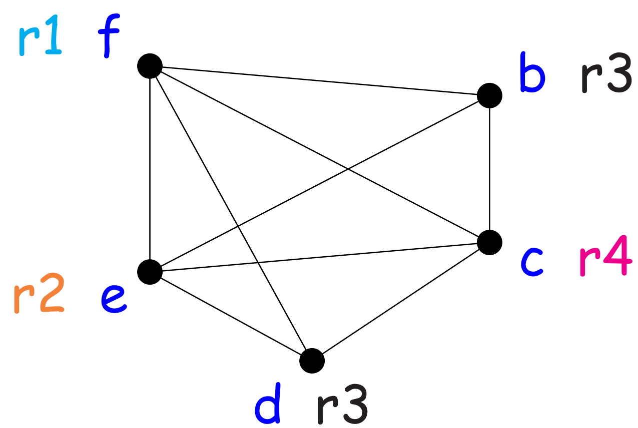
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.
- At each step, guaranteed there's a free register.



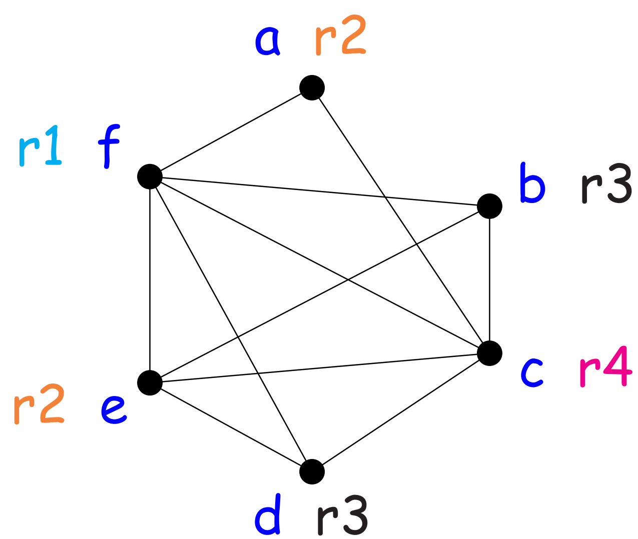
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.
- At each step, guaranteed there's a free register.



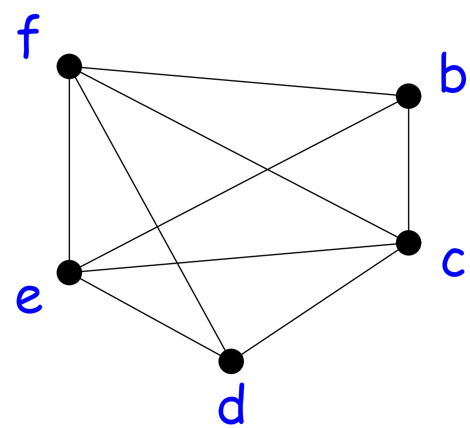
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.
- At each step, guaranteed there's a free register.



Spilling

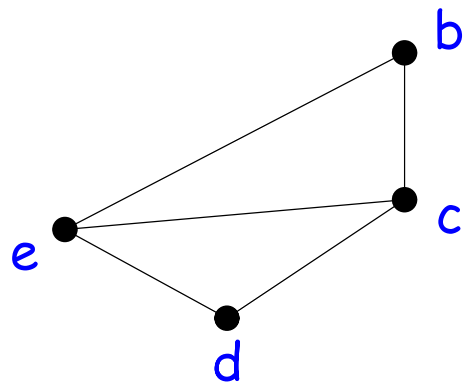
- What if during simplification we get to a state where all nodes have k or more neighbors?
- Example: try to find a 3-coloring of the RIG we've been using. After removing a, we get



- . . . and now we are stuck, since all nodes have ≥ 3 neighbors.
- So, pick a node as a candidate for *spilling,* that is, to reside in memory.

Example of Spilling

- Assume that *f* is picked as a candidate. When we remove it from the graph:

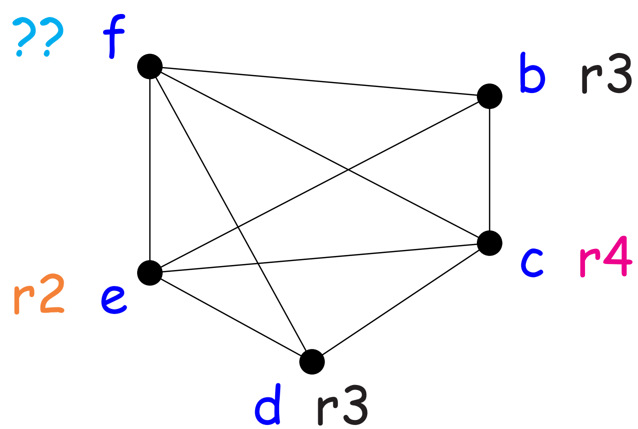


- Simplification now succeeds. We end up with the stack

  [e, c, b, d, f, a]

Example of Spilling (II)

- On the assignment phase we get to the point when we have to assign a color to f
- Sometimes, it just happens that among the 4 neighbors of f we use < 3 colors (*optimistic coloring*) ...



- ...but not this time.

Example of Spilling (III)

- Since optimistic coloring failed we must spill register f: Allocate a memory location call it fa as the *home* of f (typically in the current stack frame).
- Before each operation that uses f, insert

```
f := *fa
```

- After each operation that defines (assigns to) f, insert

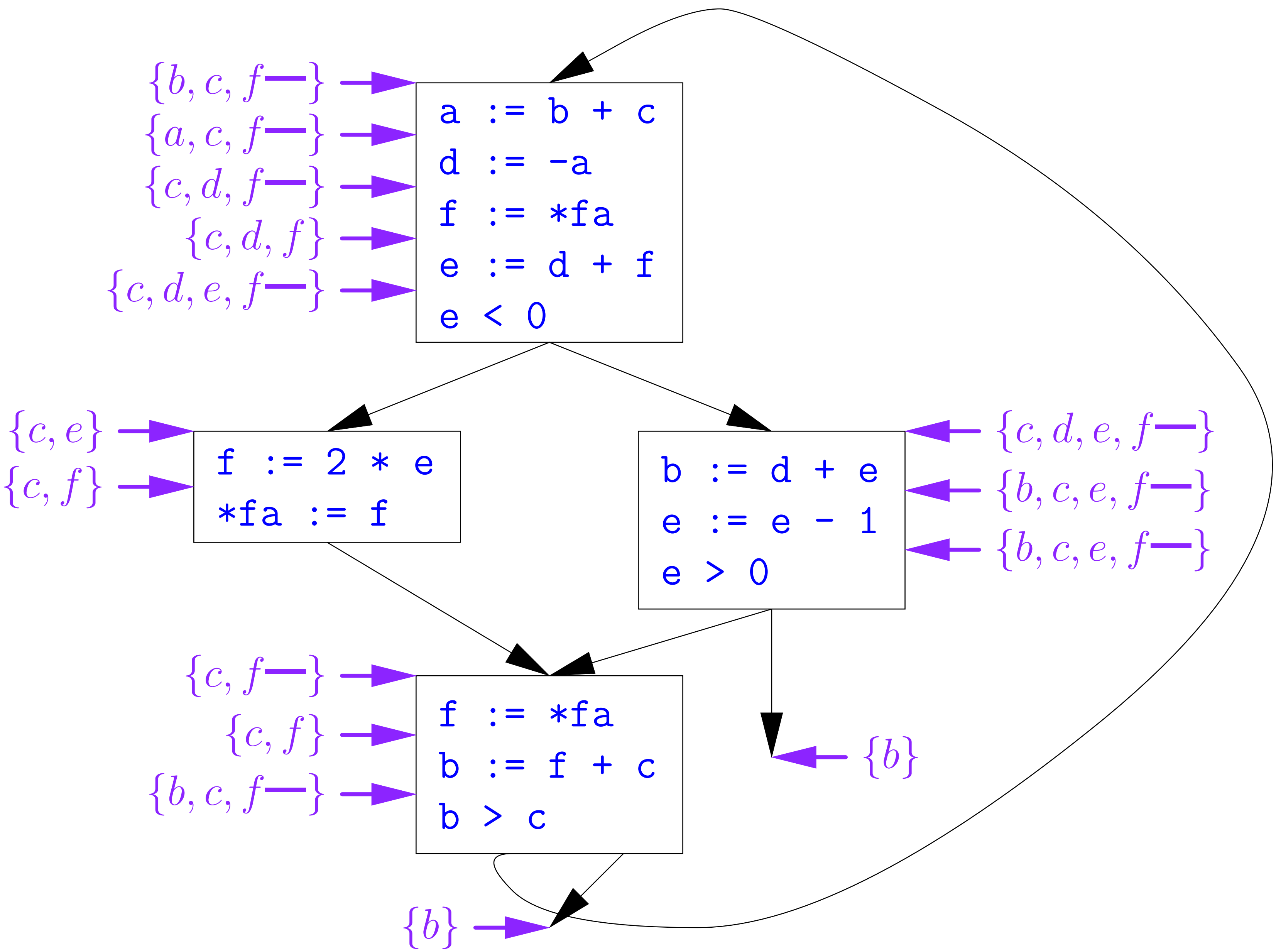
```
*fa := f
```

- This gives us:

```
A: a := b + c
d := -a
f := *fa
e := d + f
if e >= 0 jump C
B: f := 2 * e
*fa := f
jump D
```

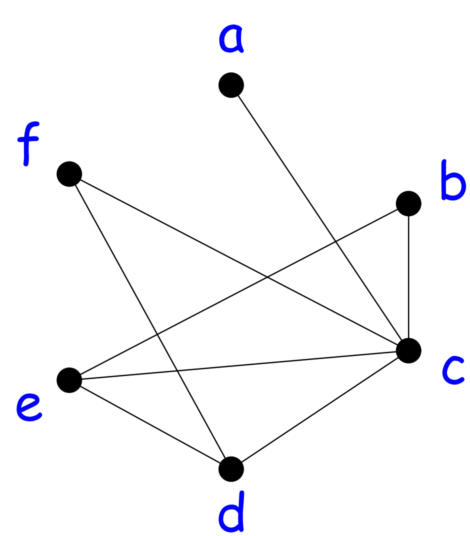
```
C: b := d + e
e := e - 1
if e <= 0 jump E
f := *fa
D: b := f + c
if b <= c jump A
E:
```

Recomputing Liveness Information



A New RIG

- The new liveness information is almost as before, except that that `f` is live only
  - Between an `f := *fa` and the next instruction, and
  - Between a `store f, fa` and the preceding instruction.
- That is, spilling reduces the live range of `f`, and thus the registers it interferes with, giving us this RIG:



- And this graph is 3-colorable (left to the reader).

What to Spill?

- In general, additional spills might be required to allow a coloring.
- The tricky part is deciding what to spill. Possible heuristics:
  - Spill temporaries with most conflicts
  - Spill temporaries with few definitions and uses
  - Avoid spilling in inner loops

Caches

- Compilers are very good at managing registers (much better than programmers: the C **register** declaration is really obsolete).
- Caches are another matter. The problem is still left to programmers, and it is still an open question whether compilers can do much in general to improve performance
- But they can (and a few do) perform some simple cache optimization

Cache Optimization

- Consider the loop

```
for(j = 1; j < 10; j += 1)
    for(i = 1; i < 1000000; i += 1)
       a[i] *= b[i]
```

- Why does this have terrible cache performance?
- On the other hand,

```
for(i = 1; i < 1000000; i += 1)
    for(j = 1; j < 10; j += 1)
        a[i] *= b[i]
```

  computes the same thing, but with much better (possibly 10x) performance [again why?].

- Compilers can do this: *loop interchange*.

Cache Optimization (II)

- Other kinds of memory layout decisions possible, such as *padding* rows of a matrix with extra bytes to avoid cache conflicts when traversing a column (or row in FORTRAN) of a matrix. [Why might that help?]
- *Prefetching* instructions on some hardware can inform cache of anticipated future memory fetches so that they can proceed in parallel. Again, it is possible for compilers to supply these to a limited extent.

Summary

- Both because it eases code generation, greatly improves performance, and because it is difficult for programmers to do it for themselves, register allocation is a "must have" optimization in production compilers for standard procedural languages.
- Graph coloring is a powerful register allocation scheme that compilers can apply automatically
- Good cache management could give even larger payoffs, but so far is difficult.