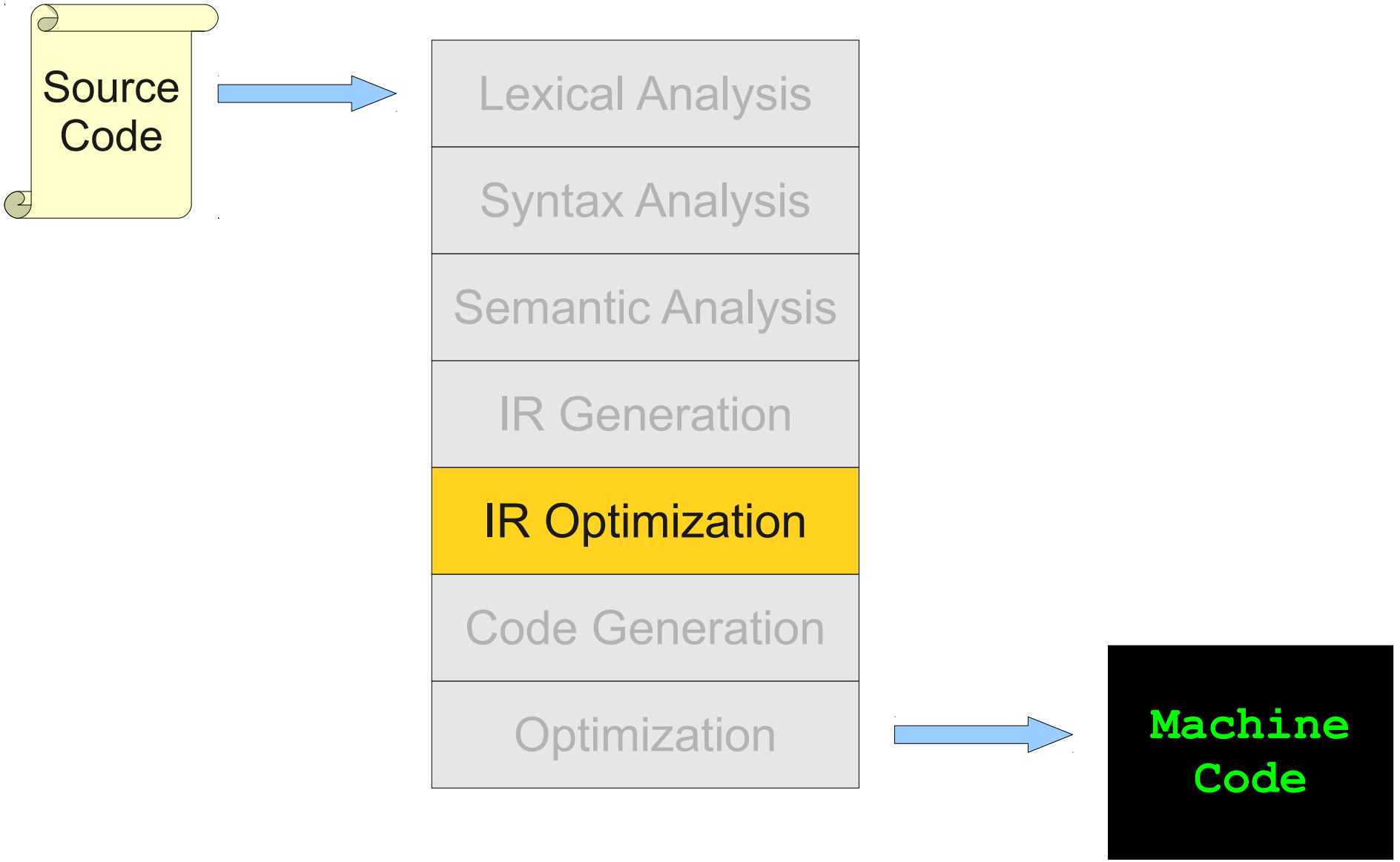
Register Allocation

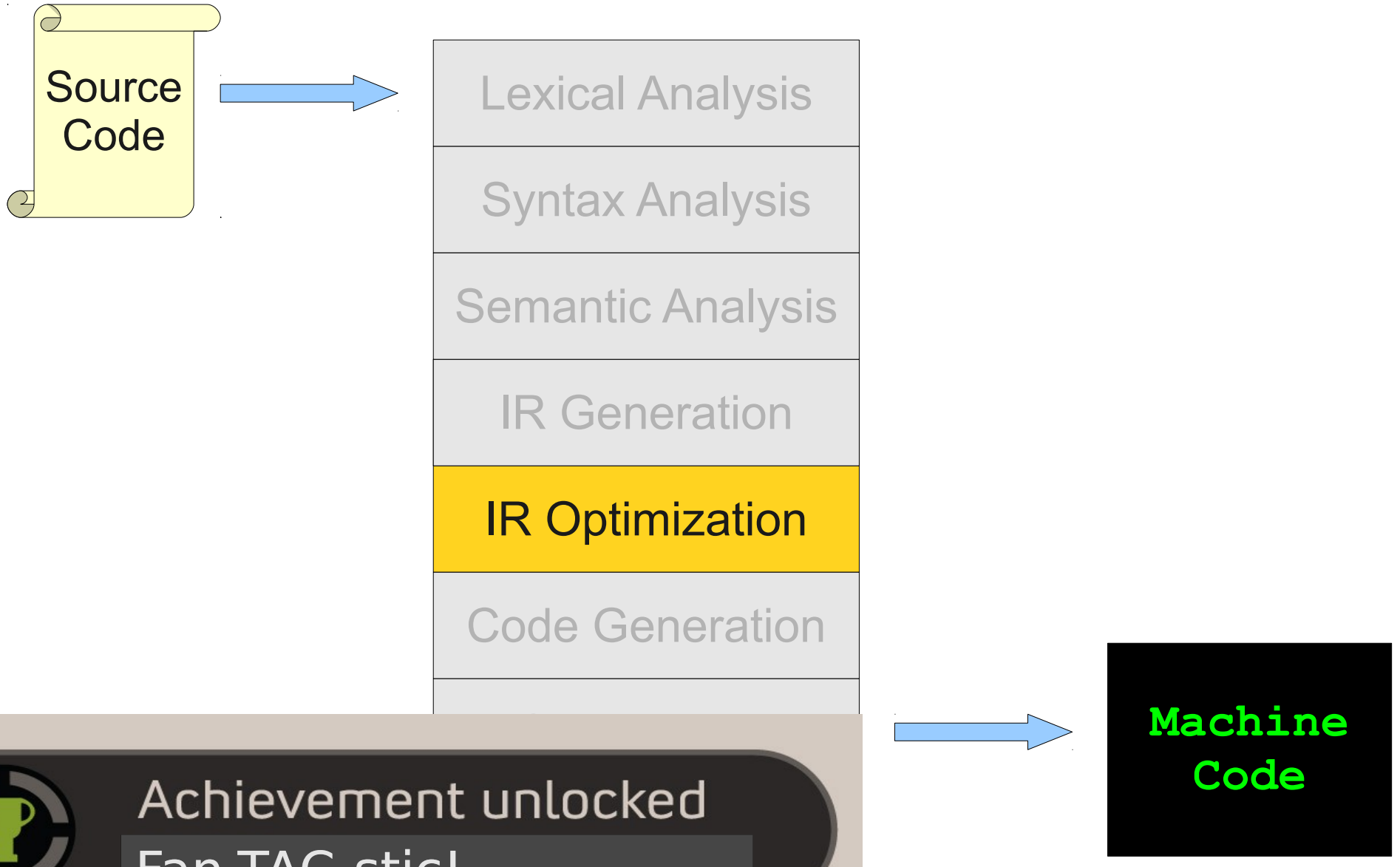
Announcements

- Programming Project 4 due **Wednesday, August 10** at 11:59PM.
  - OH all this week and Sunday.
  - Ask questions via email!
  - Ask questions via Piazza!
- Online course evaluation available on Axess.
  - Please give feedback!

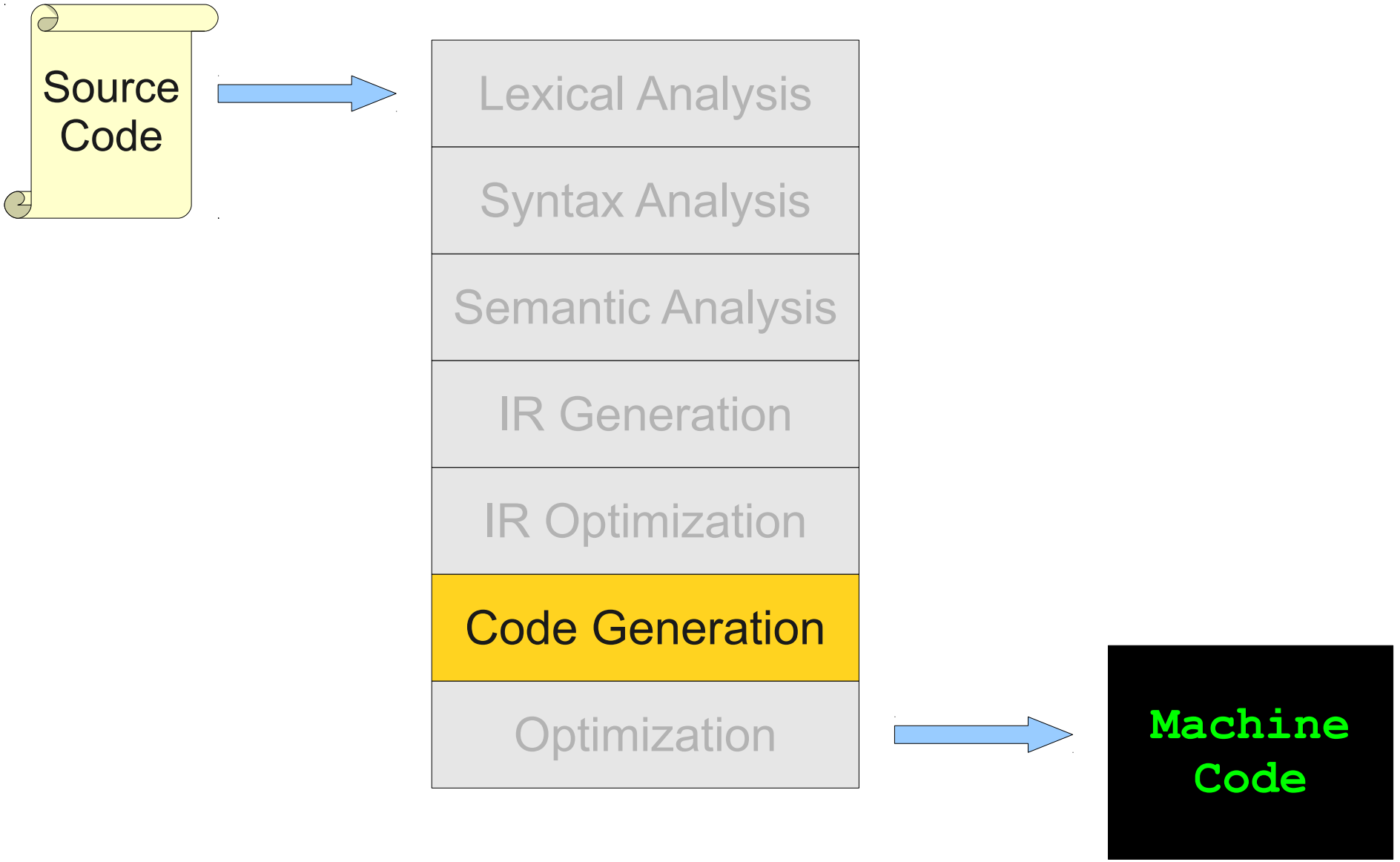
Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Machine Code
Achievement unlocked
Fan-TAC-stic!

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Code Generation at a Glance

- At this point, we have optimized IR code that needs to be converted into the target language (e.g. assembly, machine code).
- Goal of this stage:
  - Choose the appropriate machine instructions for each IR instruction.
  - Divvy up finite machine resources (registers, caches, etc.)
  - Implement low-level details of the runtime environment.
- Machine-specific optimizations are often done here, though some are treated as part of a final optimization phase.

Overview

- **Register Allocation** (Today)
  - How to assign variables to finitely many registers?
  - What to do when it can't be done?
  - How to do so efficienty?
- **Garbage Collection** (Monday)
  - How to detect reclaimable memory?
  - How to reclaim memory efficiently?

Memory Tradeoffs

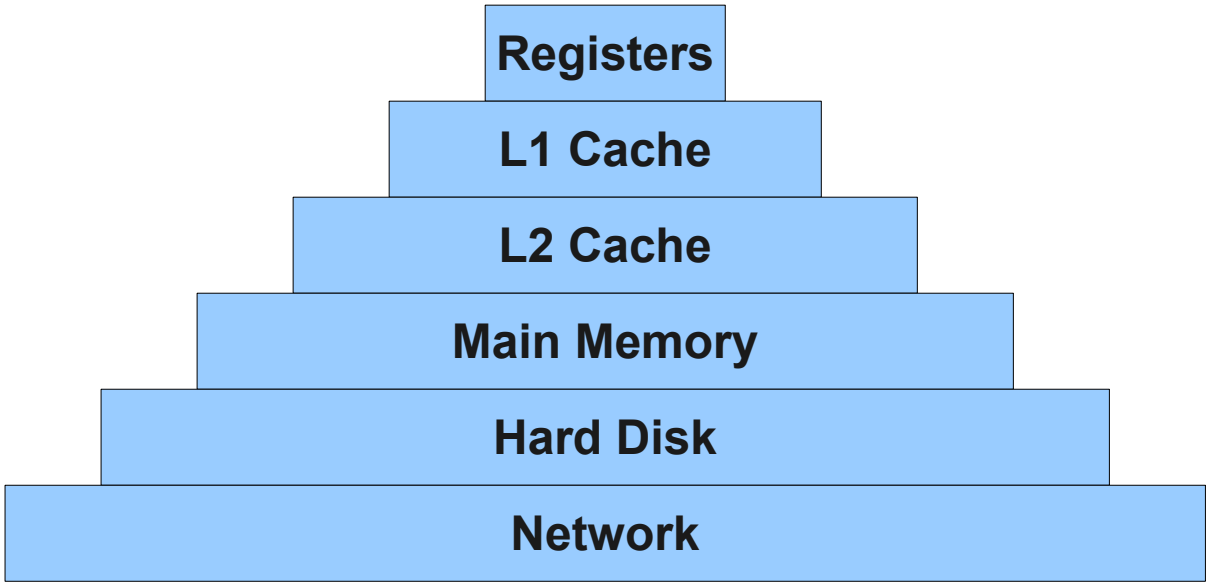
- There is an enormous tradeoff between **speed** and **size** in memory.
- SRAM is fast but very expensive:
  - Can keep up with processor speeds in the GHz.
  - As of 2007, cost is $10/MB
  - Good luck buying 1TB of the stuff!
- Hard disks are cheap but very slow:
  - As of 2011, you can buy a 2TB hard drive for $70
  - As of 2011, good disk seek times are measured in ms (about two to four million times slower than a processor cycle!)

The Memory Hierarchy

- **Idea**: Try to get the best of all worlds by using multiple types of memory.

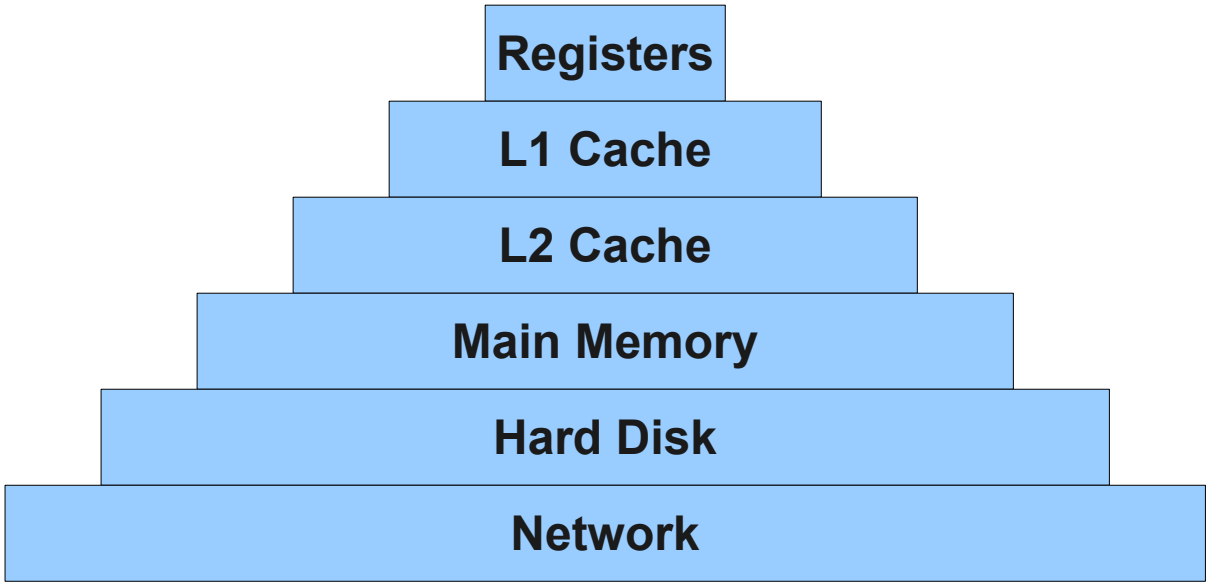
The Memory Hierarchy

- **Idea**: Try to get the best of all worlds by using multiple types of memory.



The Memory Hierarchy

- **Idea**: Try to get the best of all worlds by using multiple types of memory.



256B - 8KB

16KB – 64KB

1MB - 4MB

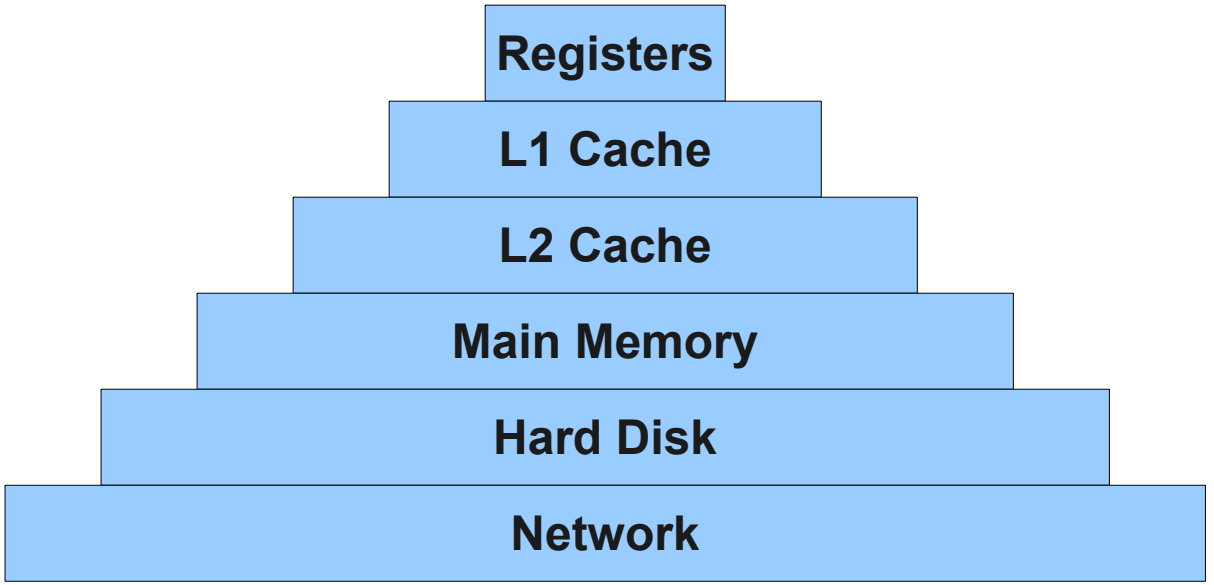
4GB – 32GB

100GB+

HUGE

The Memory Hierarchy

- **Idea**: Try to get the best of all worlds by using multiple types of memory.



Registers	256B - 8KB	0.25 – 1ns
L1 Cache	16KB – 64KB	1ns – 5 ns
L2 Cache	1MB - 4MB	5ns – 25ns
Main Memory	4GB – 32GB	25ns – 100ns
Hard Disk	100GB+	3 – 10ms
Network	**HUGE**	10 – 2000ms

The Challenges of Code Generation

- Almost all programming languages expose a coarse view of the memory hierarchy:
  - All variables live in “memory.”
  - Disk and network explicitly handled separately.
- (Interesting exception: Stanford's **Sequoia** programming language)
- Challenges in code generation:
  - Position objects in a way that takes maximum advantage of the memory hierarchy.
  - Do so without hints from the programmer.

Registers

- Most machines have a set of **registers**, dedicated memory locations that
  - can be accessed quickly,
  - can have computations performed on them, and
  - exist in small quantity.
- Using registers intelligently is a critical step in any compiler.
  - A good register allocator can generate code orders of magnitude better than a bad register allocator.

Register Allocation

- In TAC, there are an unlimited number of variables.
- On a physical machine there are a small number of registers:
  - x86 has four general-purpose registers and a number of specialized registers.
  - MIPS has twenty-four general-purpose registers and eight special-purpose registers.
- **Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers.

Challenges in Register Allocation

- **Registers are scarce**.
  - Often substantially more IR variables than registers.
  - Need to find a way to reuse registers whenever possible.
- **Registers are complicated**.
  - x86: Each register made of several smaller registers; can't use a register and its constituent registers at the same time.
  - x86: Certain instructions must store their results in specific registers; can't store values there if you want to use those instructions.
  - MIPS: Some registers reserved for the assembler or operating system.
  - Most architectures: Some registers must be preserved across function calls.

Goals for Today

- Introduce register allocation for a MIPS-style machine:
  - Some number of indivisible, general-purpose registers.
- Explore three algorithms for register allocation:
  - Naive (“no”) register allocation.
  - Linear scan register allocation.
  - Graph-coloring register allocation.

An Initial Register Allocator

- **Idea**: Store every value in main memory, loading values only when they're needed.
- To generate a code that performs a computation:
  - Generate **load** instructions to pull the values from main memory into registers.
  - Generate code to perform the computation on the registers.
  - Generate **store** instructions to store the result back into main memory.

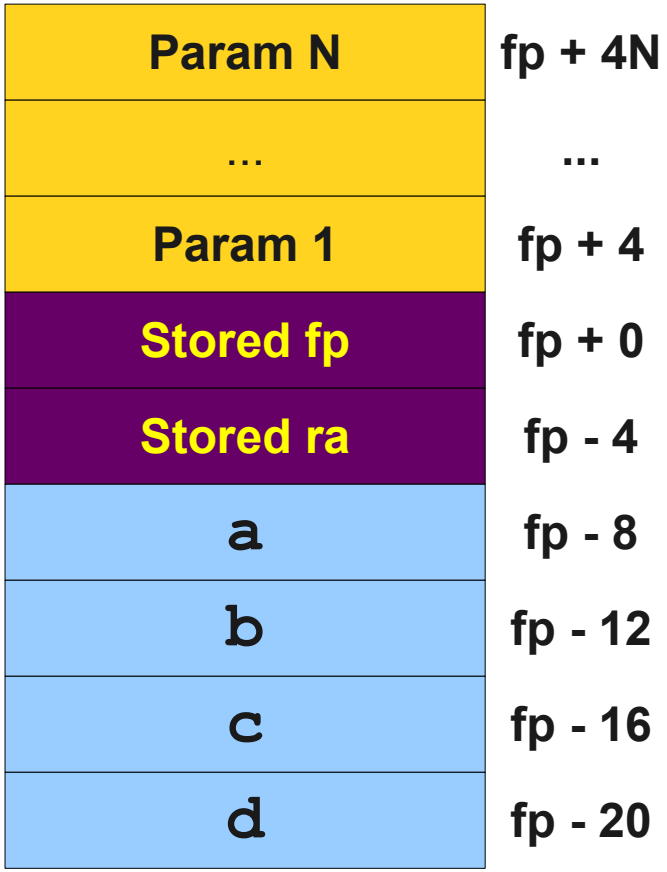
Our Register Allocator In Action

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

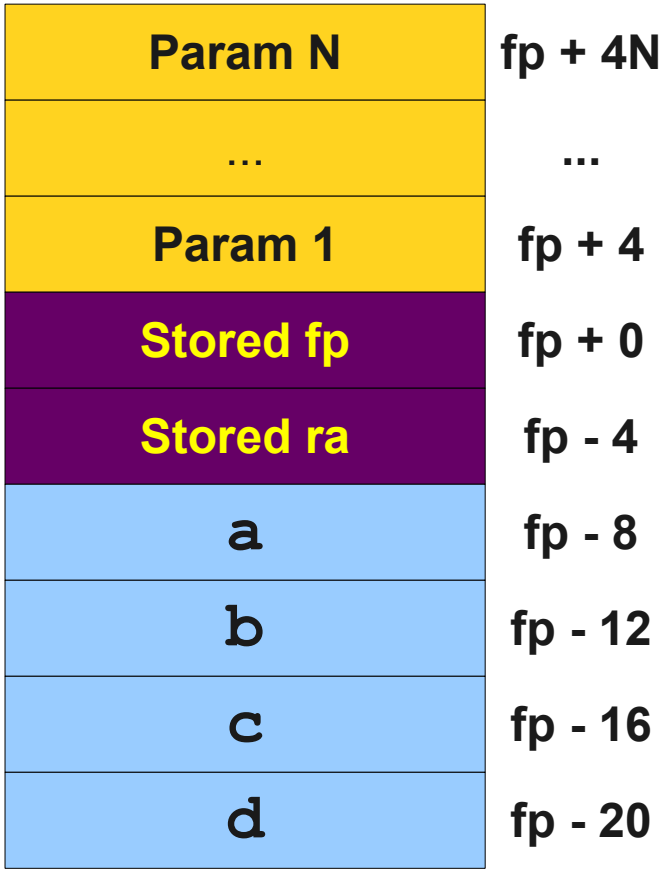
Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```



Our Register Allocator In Action

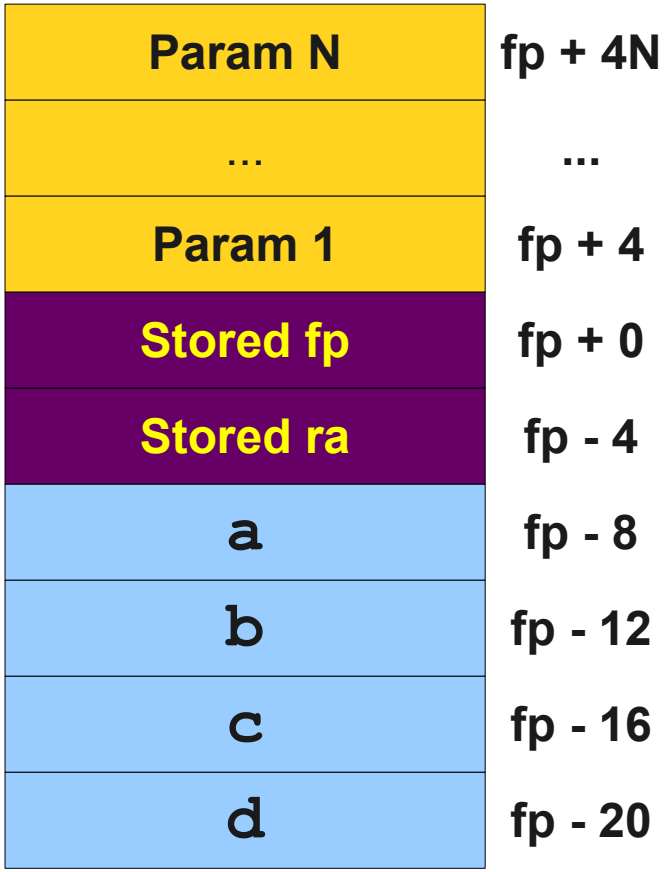
```
a = b + c;
d = a;
c = a + d;
```



Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

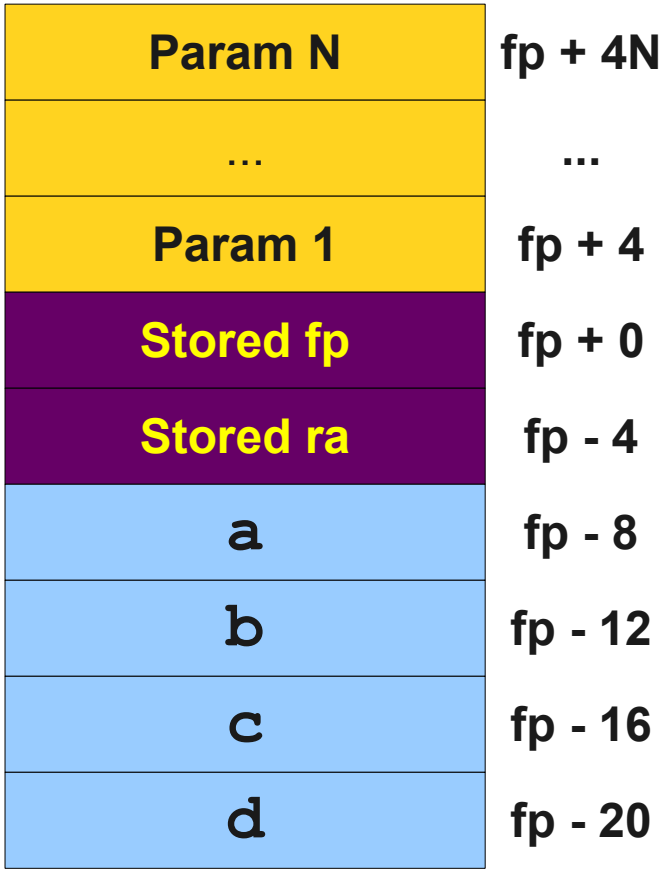
```
lw   $t0, -12(fp)
```



Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

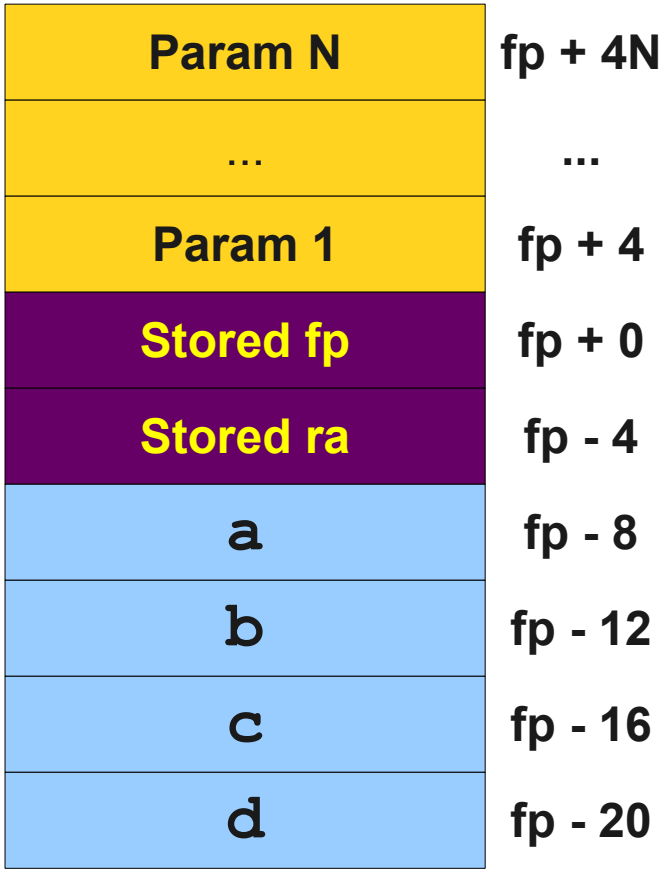
```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
```



Our Register Allocator In Action

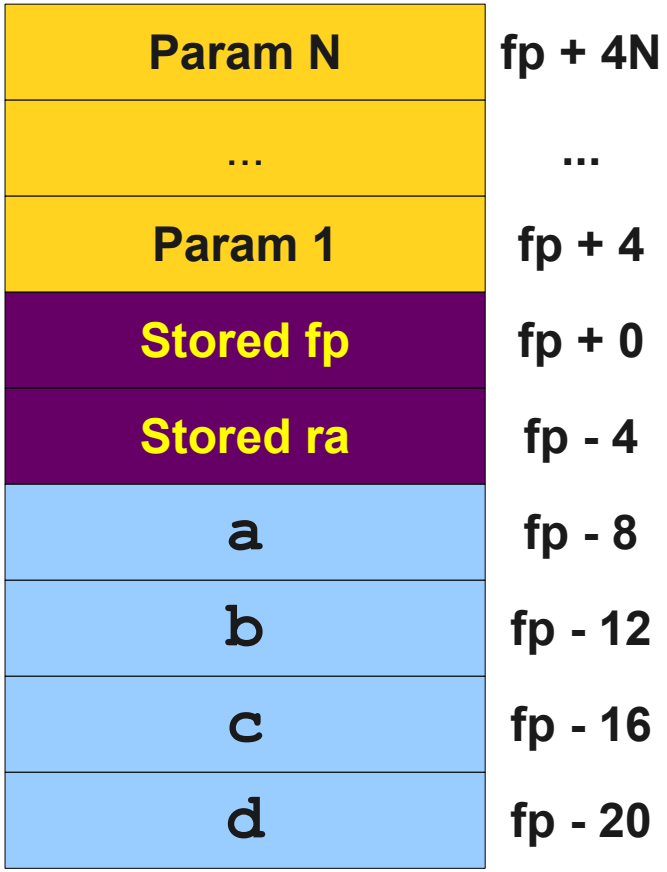
```
a = b + c;
d = a;
c = a + d;
```

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add $t2, $t0, $t1
```



Our Register Allocator In Action

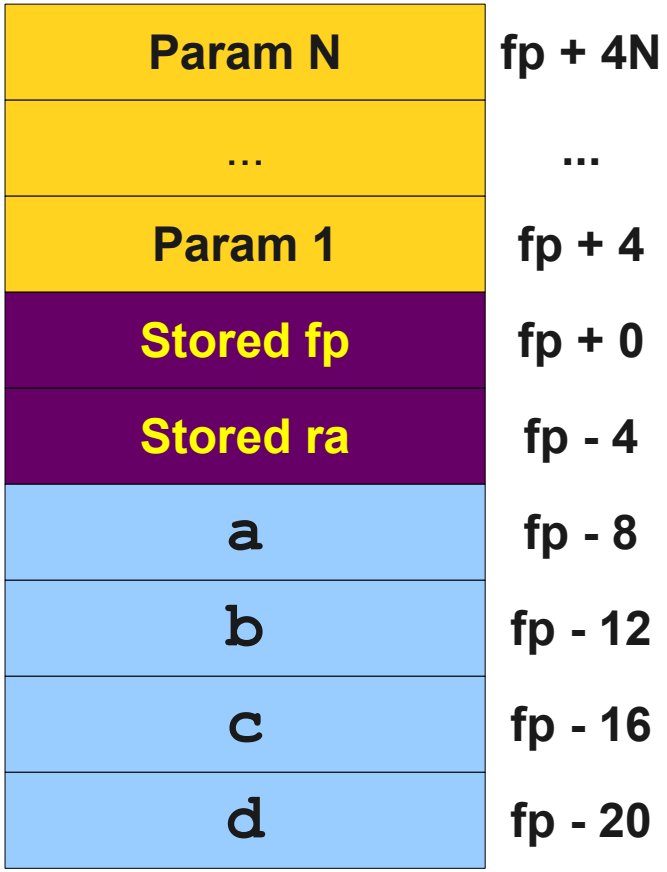
```
a = b + c;
d = a;
c = a + d;
```




```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)
```

Our Register Allocator In Action

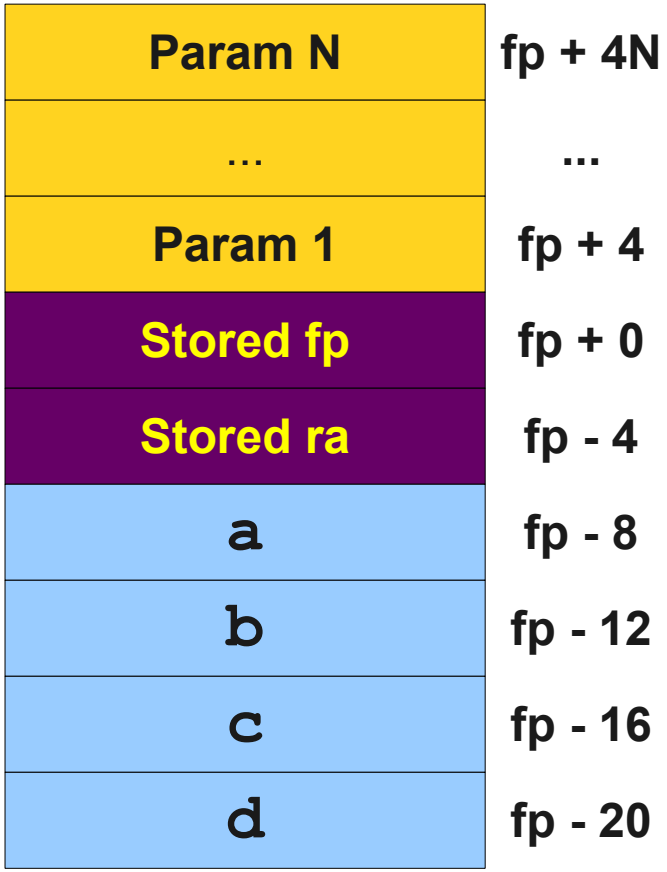
```
a = b + c;
d = a;
c = a + d;
```




```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

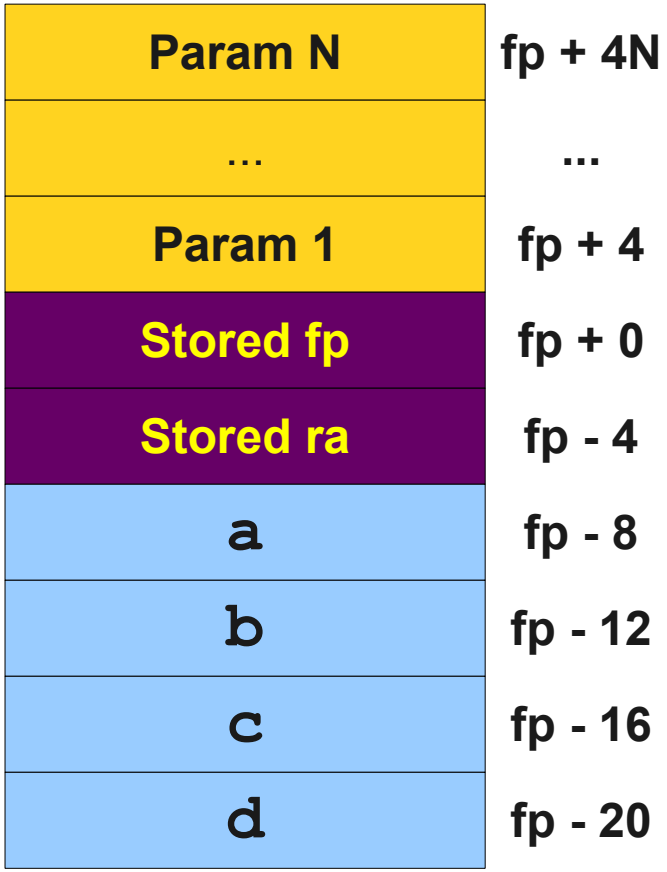



```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)

lw   $t0,  -8(fp)
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

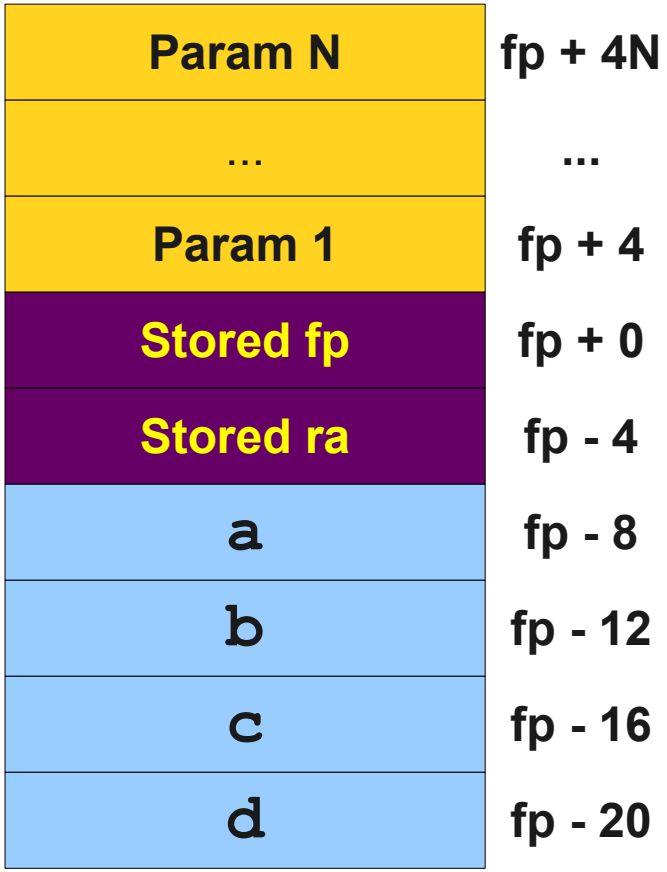



```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)

lw   $t0,  -8(fp)
sw   $t0, -20(fp)
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```

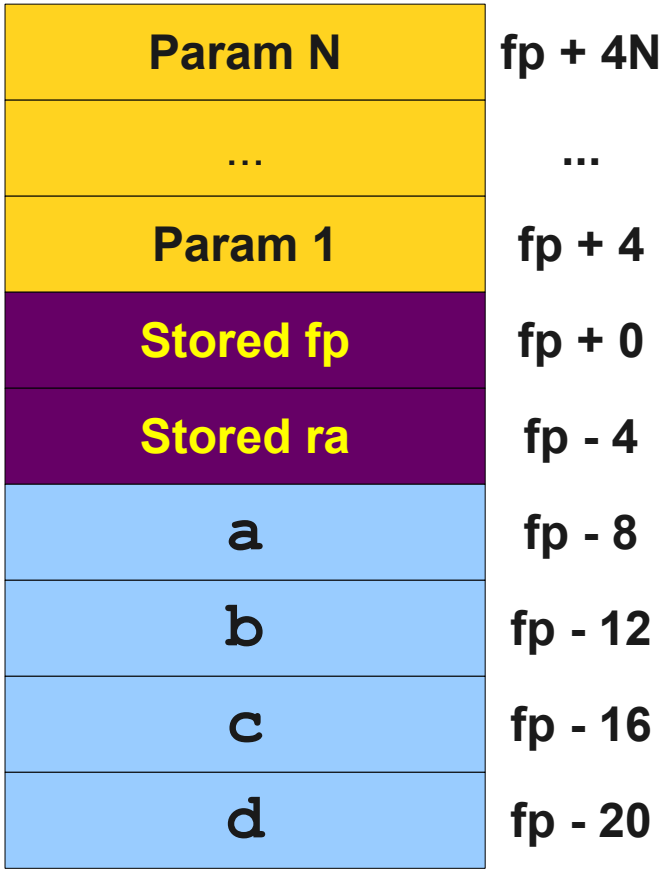



```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)

lw   $t0,  -8(fp)
sw   $t0, -20(fp)
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```



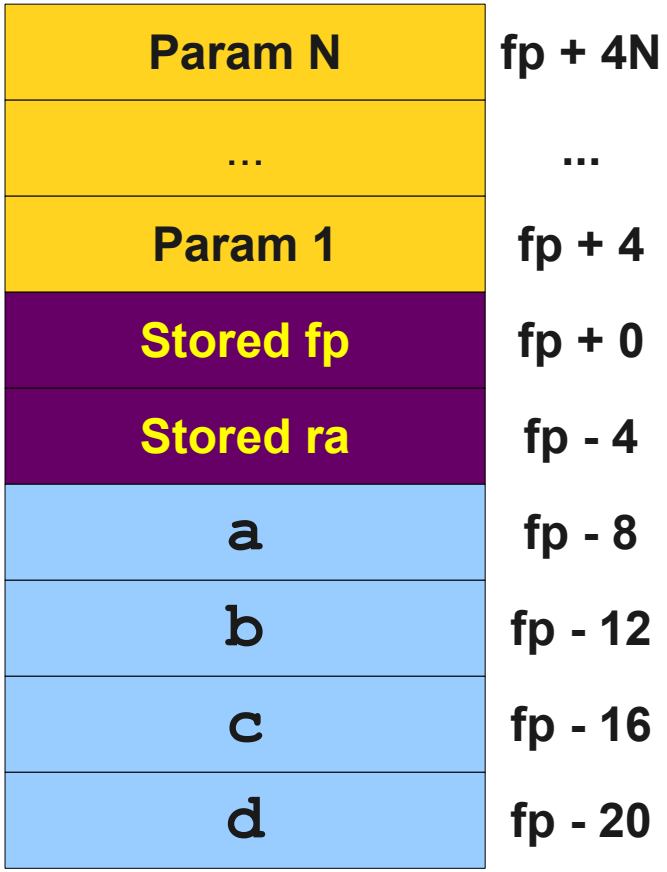

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)

lw   $t0,  -8(fp)
sw   $t0, -20(fp)

lw   $t0,  -8(fp)
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```



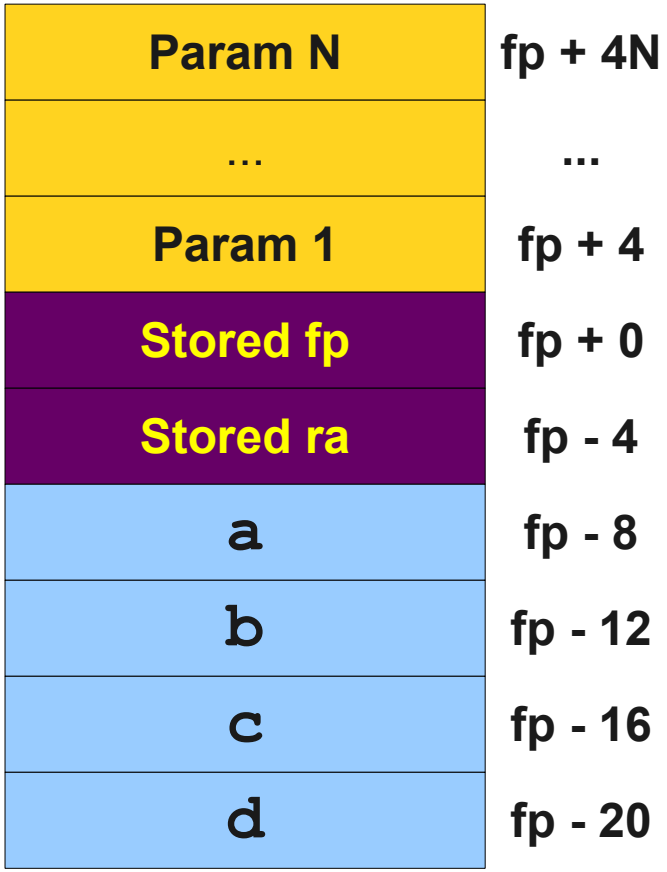

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)

lw   $t0,  -8(fp)
sw   $t0, -20(fp)

lw   $t0,  -8(fp)
lw   $t1, -20(fp)
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```



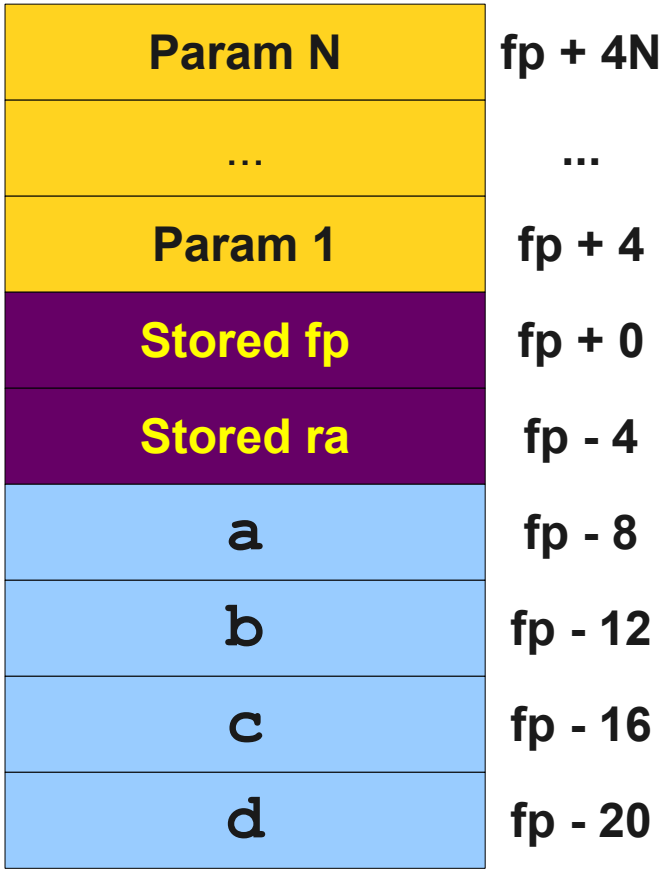

```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)

lw   $t0,  -8(fp)
sw   $t0, -20(fp)

lw   $t0,  -8(fp)
lw   $t1, -20(fp)
add  $t2, $t0, $t1
```

Our Register Allocator In Action

```
a = b + c;
d = a;
c = a + d;
```




```
lw   $t0, -12(fp)
lw   $t1, -16(fp)
add  $t2, $t0, $t1
sw   $t2,  -8(fp)

lw   $t0,  -8(fp)
sw   $t0, -20(fp)

lw   $t0,  -8(fp)
lw   $t1, -20(fp)
add  $t2, $t0, $t1
sw   $t2, -16(fp)
```

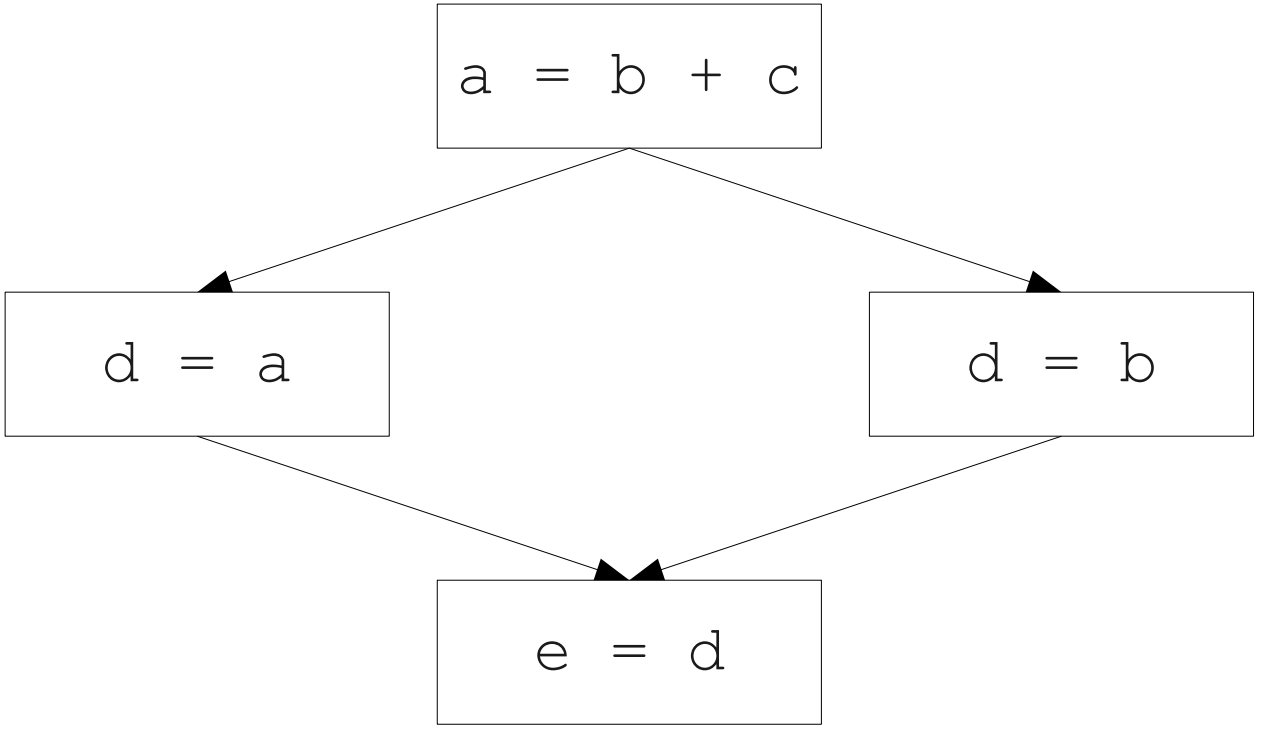
Analysis of our Allocator

- Disadvantage: **Gross inefficiency**.
  - Issues unnecessary loads and stores by the dozen.
  - Wastes space on values that could be stored purely in registers.
  - Easily an order of magnitude or two slower than necessary.
  - Unacceptable in any production compiler.
- Advantage: **Simplicity**.
  - Can translate each piece of IR directly to assembly as we go.
  - Never need to worry about running out of registers.
  - Never need to worry about function calls or special-purpose registers.
  - Good if you just needed to get a prototype compiler up and running.

Building a Better Allocator

- **Goal**: Try to hold as many variables in registers as possible.
  - Reduces memory reads/writes.
  - Reduces total memory usage.
- We will need to address these questions:
  - Which registers do we put variables in?
  - What do we do when we run out of registers?

Register Consistency



Register Consistency

- At each program point, each variable must be in the same location.
  - Does **not** mean that each variable is always stored in the same location!
- At each program point, each register holds at most one live variable.
  - Can assign several variables the same register if no two of them ever will be read together.

Live Ranges and Live Intervals

- Recall: A variable is **live** at a particular program point if its value may be read later before it is written.
  - Can find this using global liveness analysis.
- The **live range** for a variable is the set of program points at which that variable is live.
- The **live interval** for a variable is the smallest subrange of the IR code containing all a variable's live ranges.
  - A property of the IR code, **not** the CFG.
  - Less precise than live ranges, but simpler to work with.

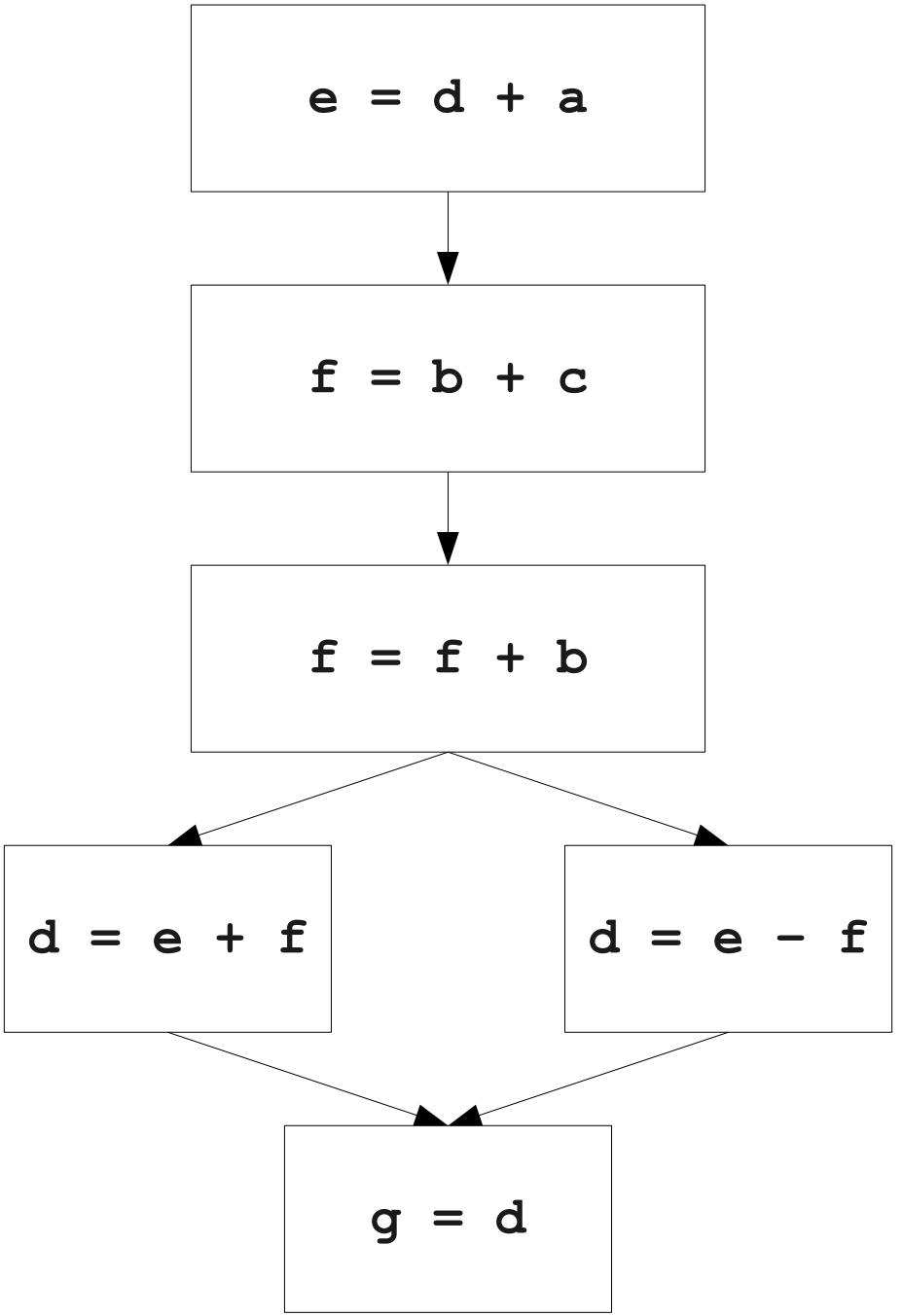
Live Ranges and Live Intervals

Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

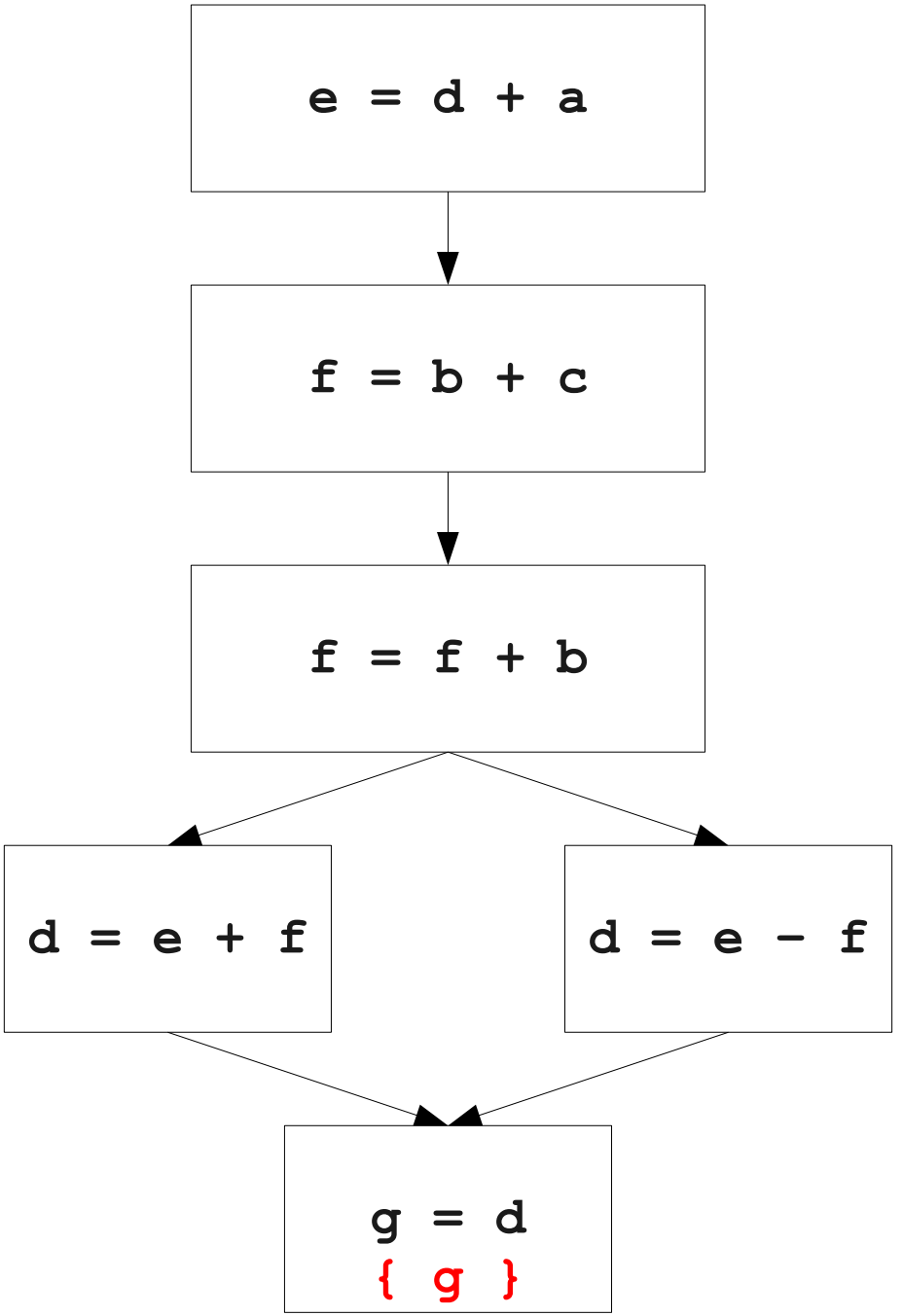
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



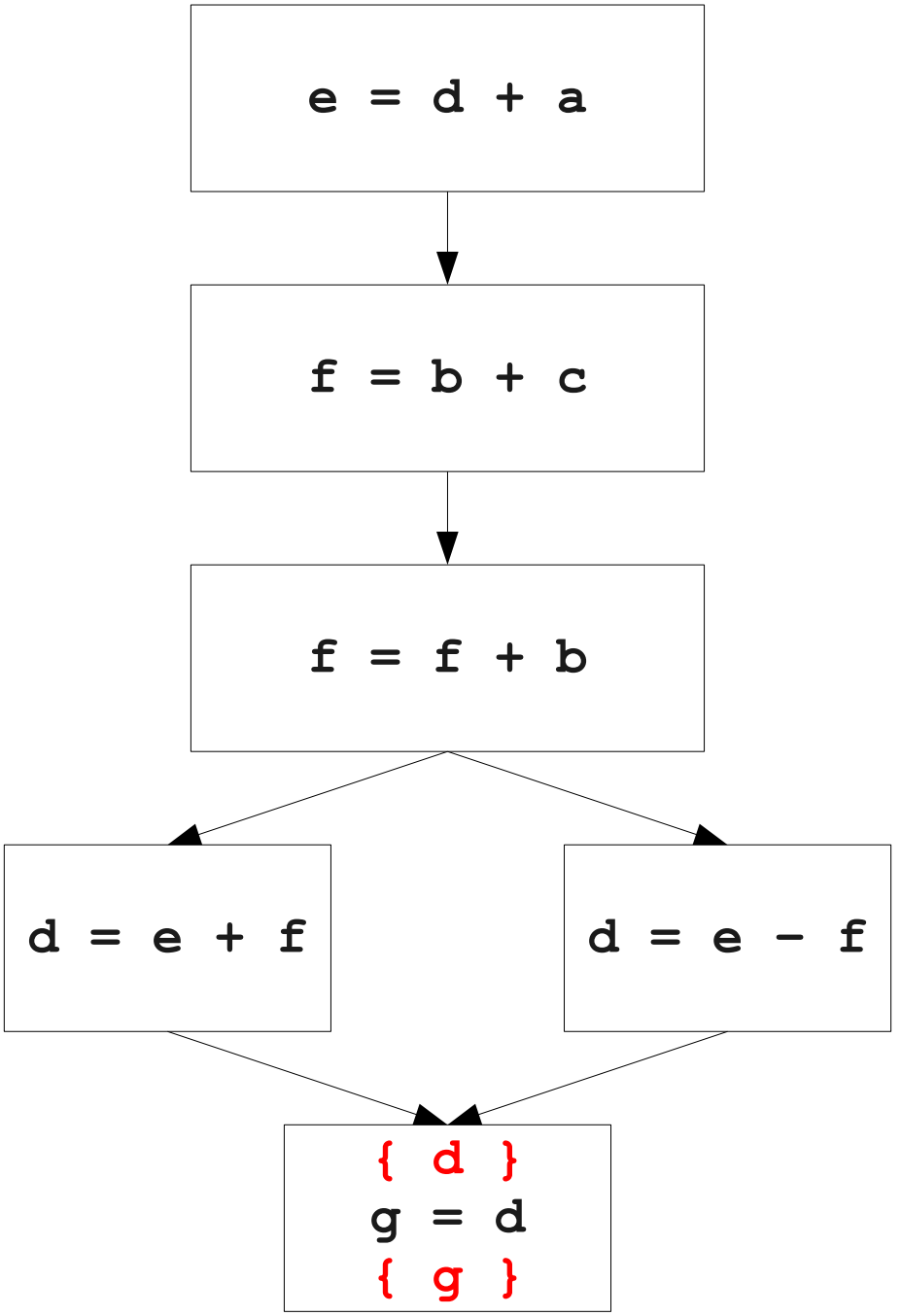
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



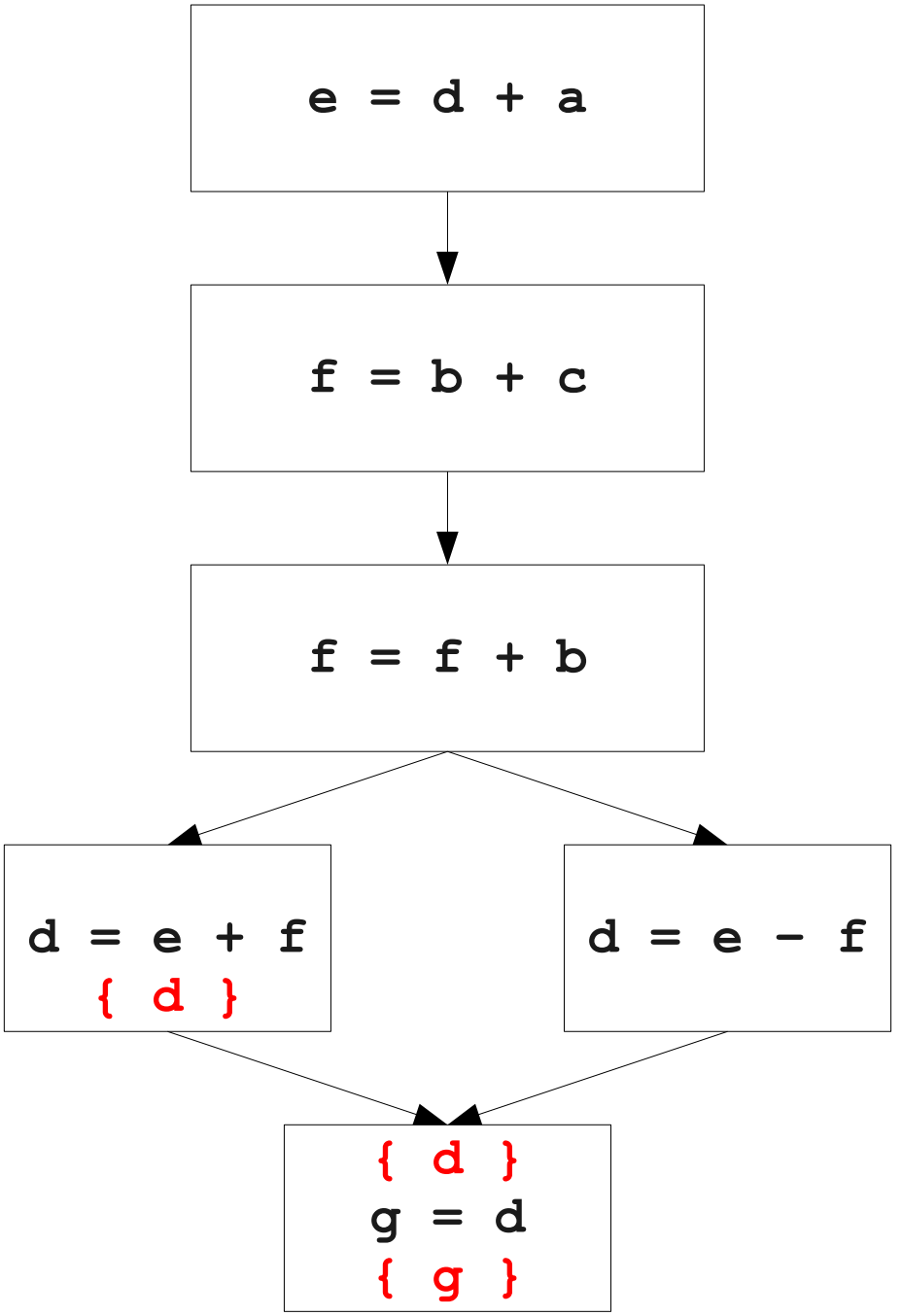
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



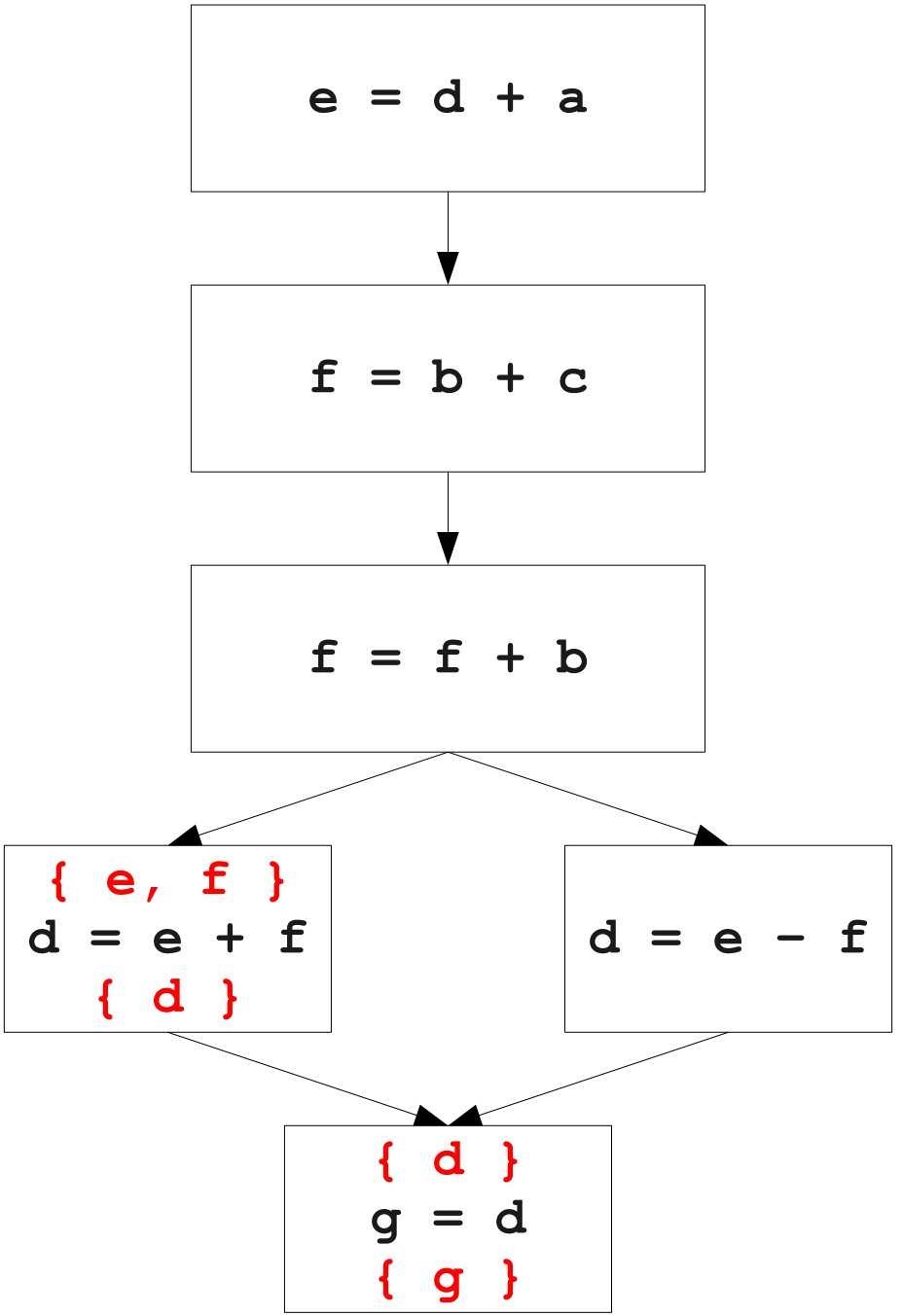
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



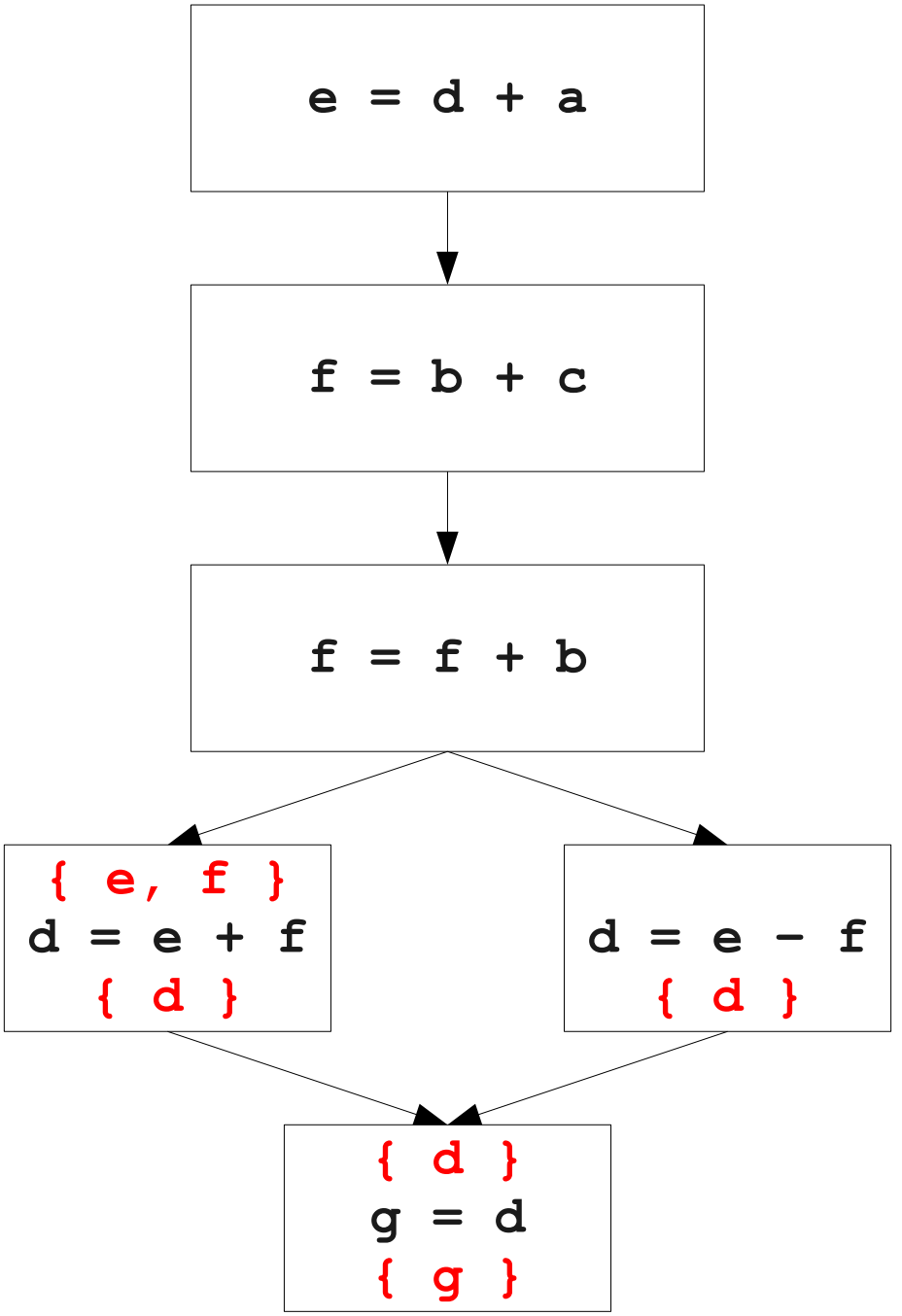
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



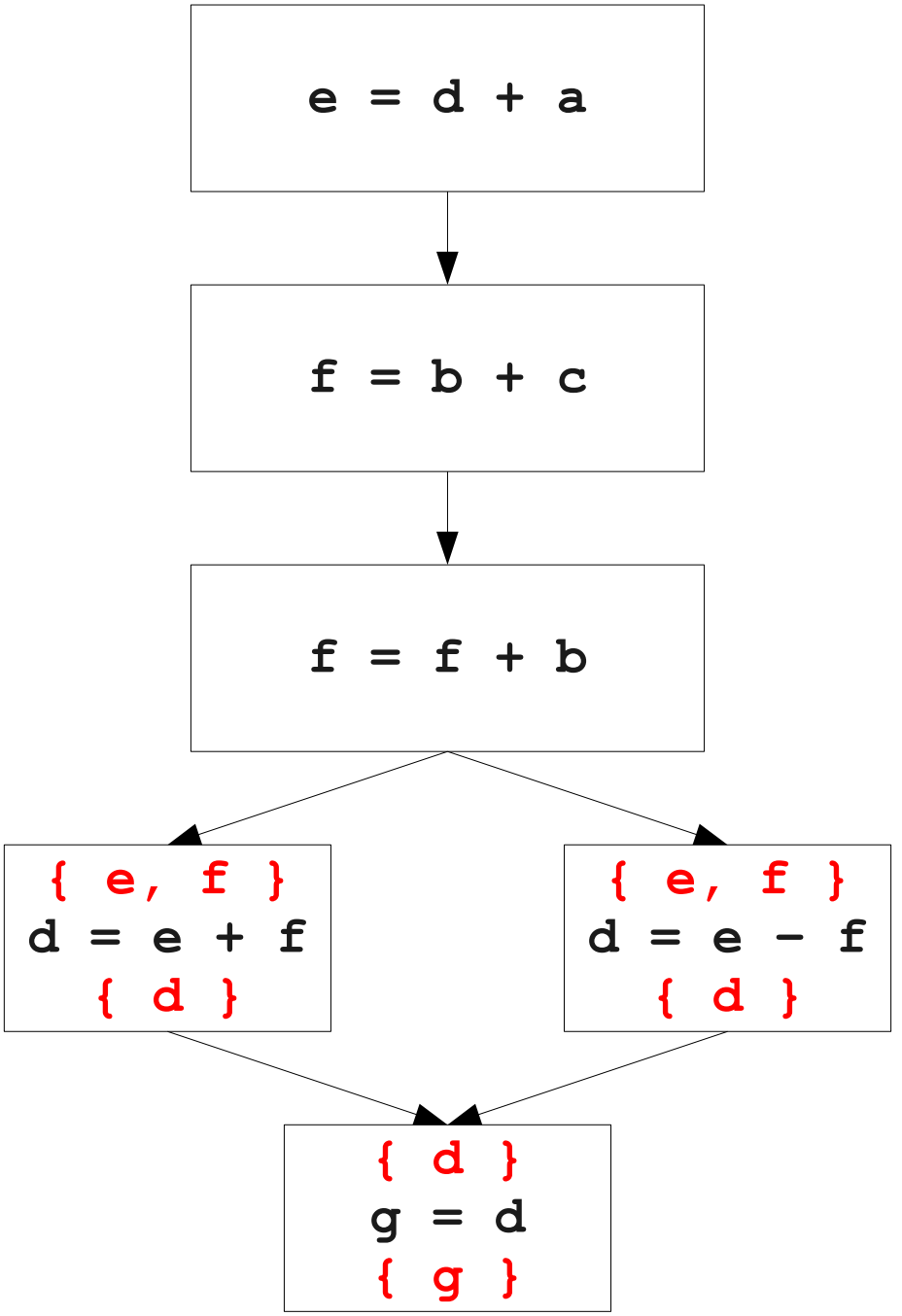
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



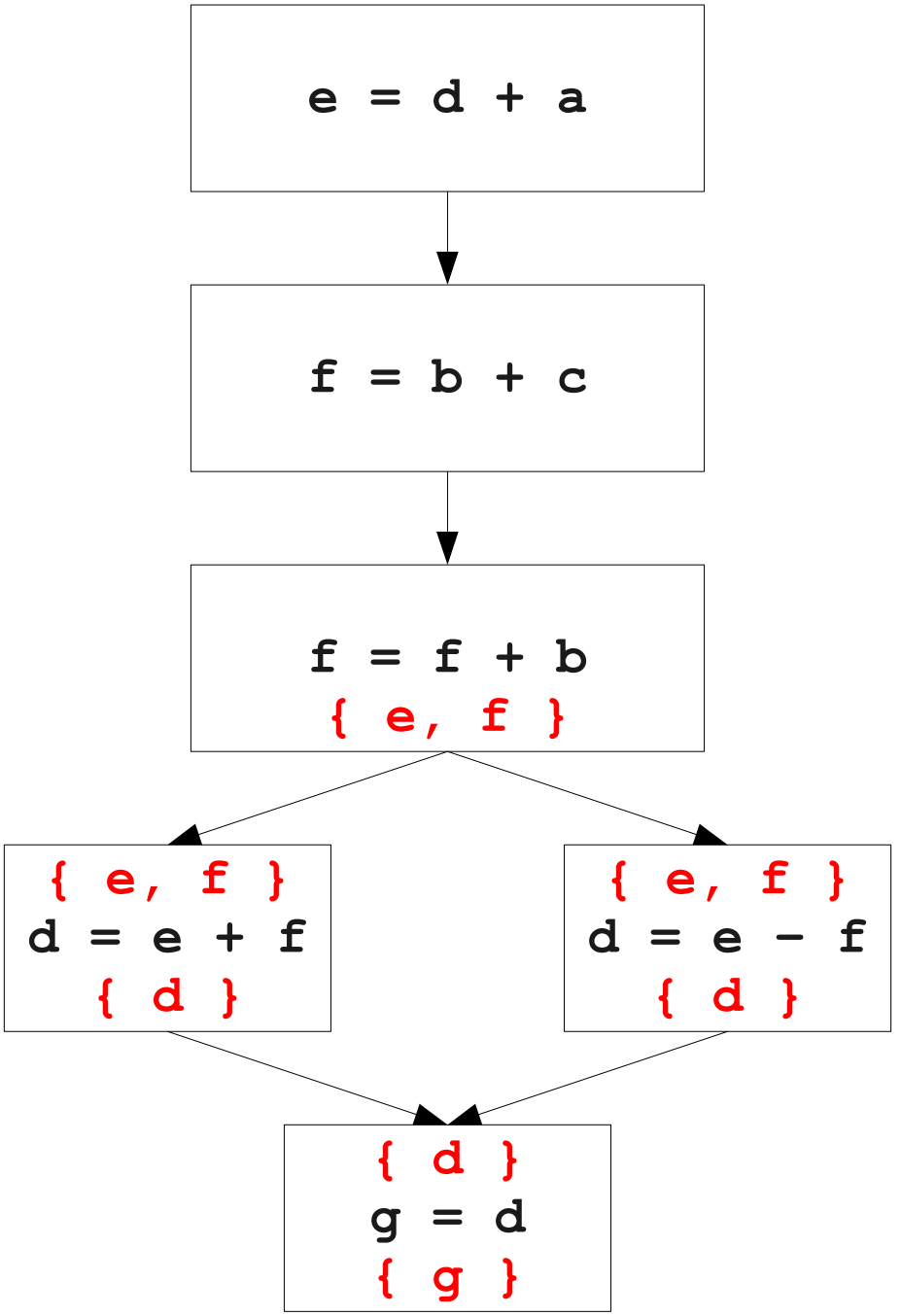
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



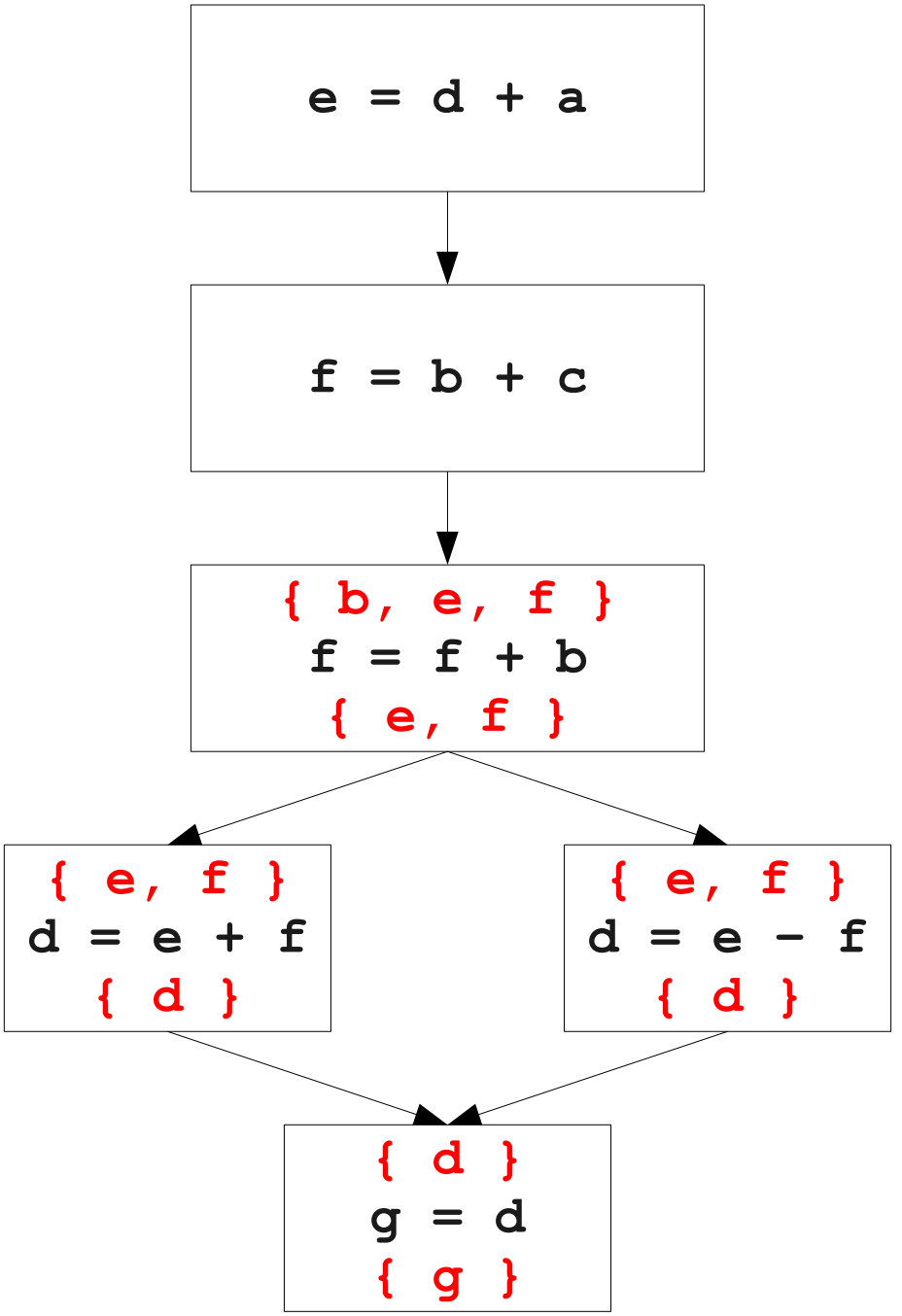
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



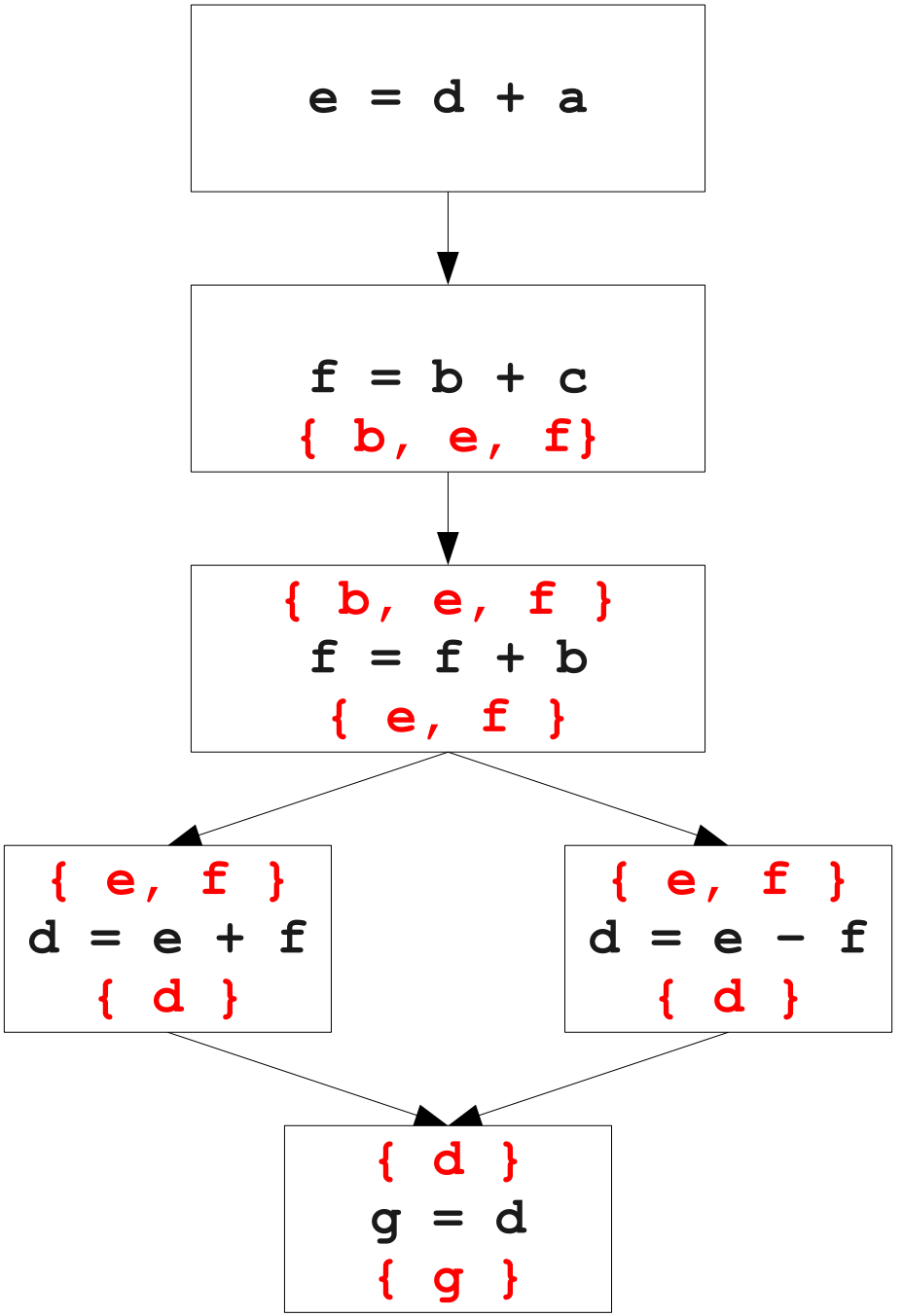
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



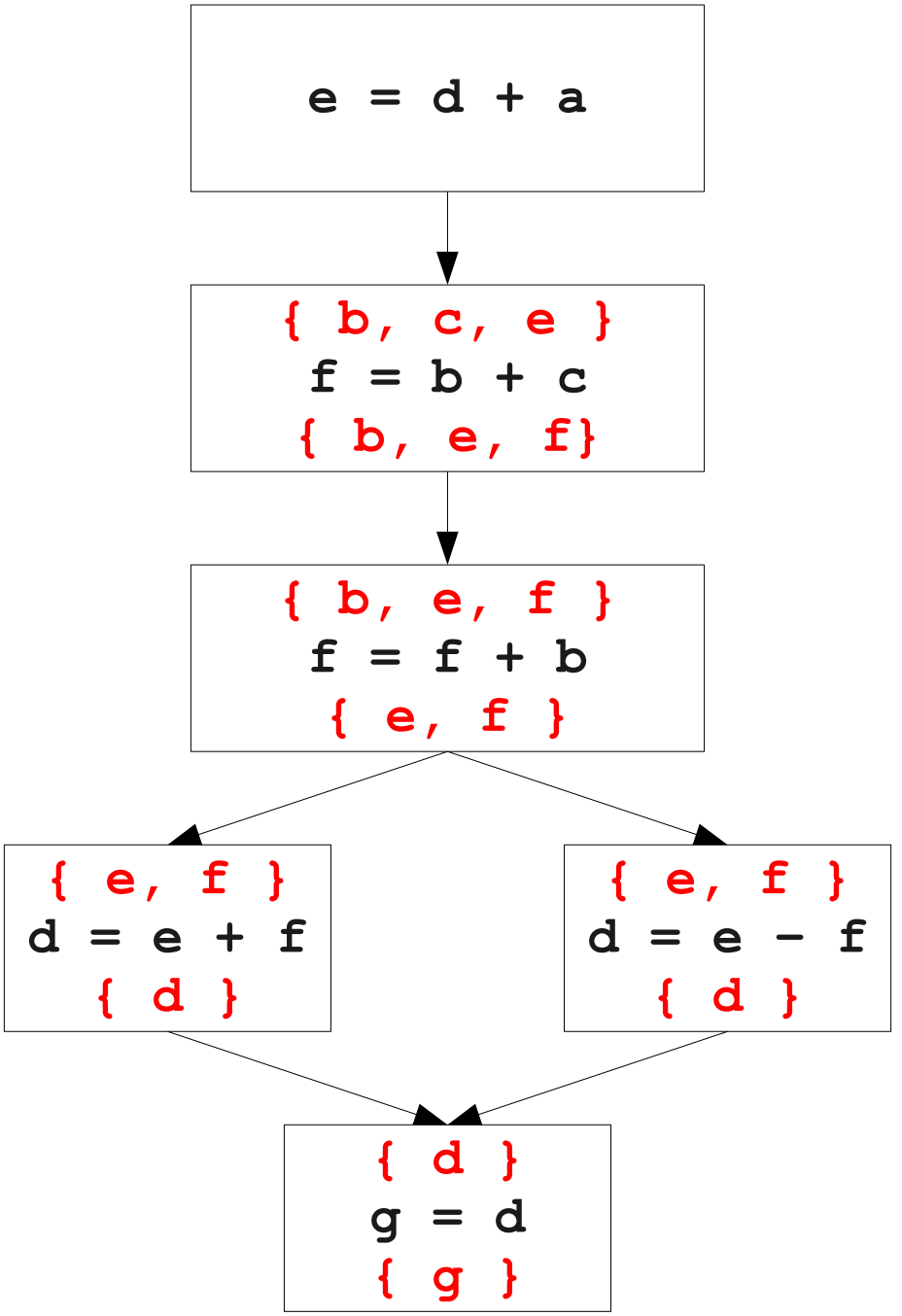
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



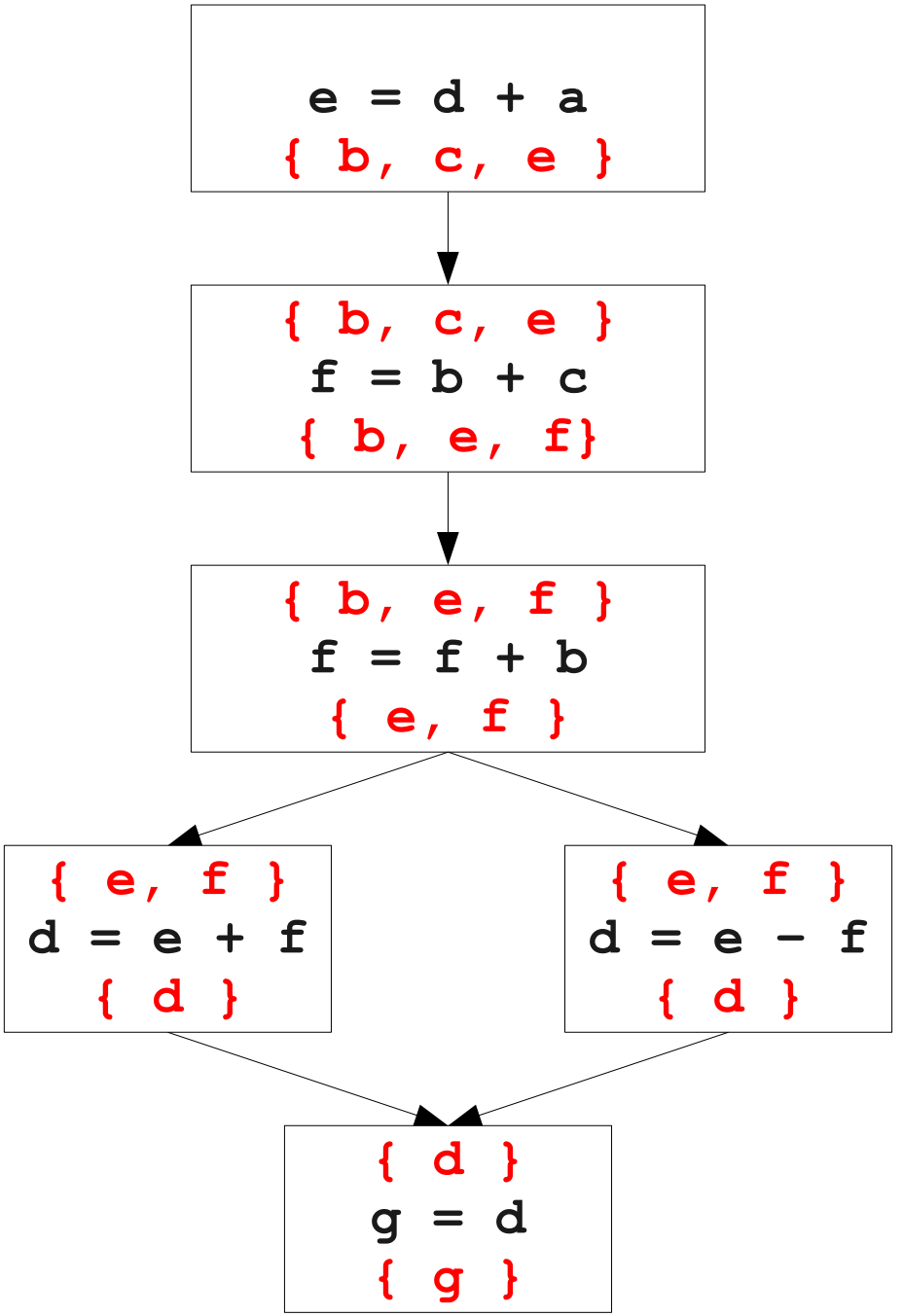
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



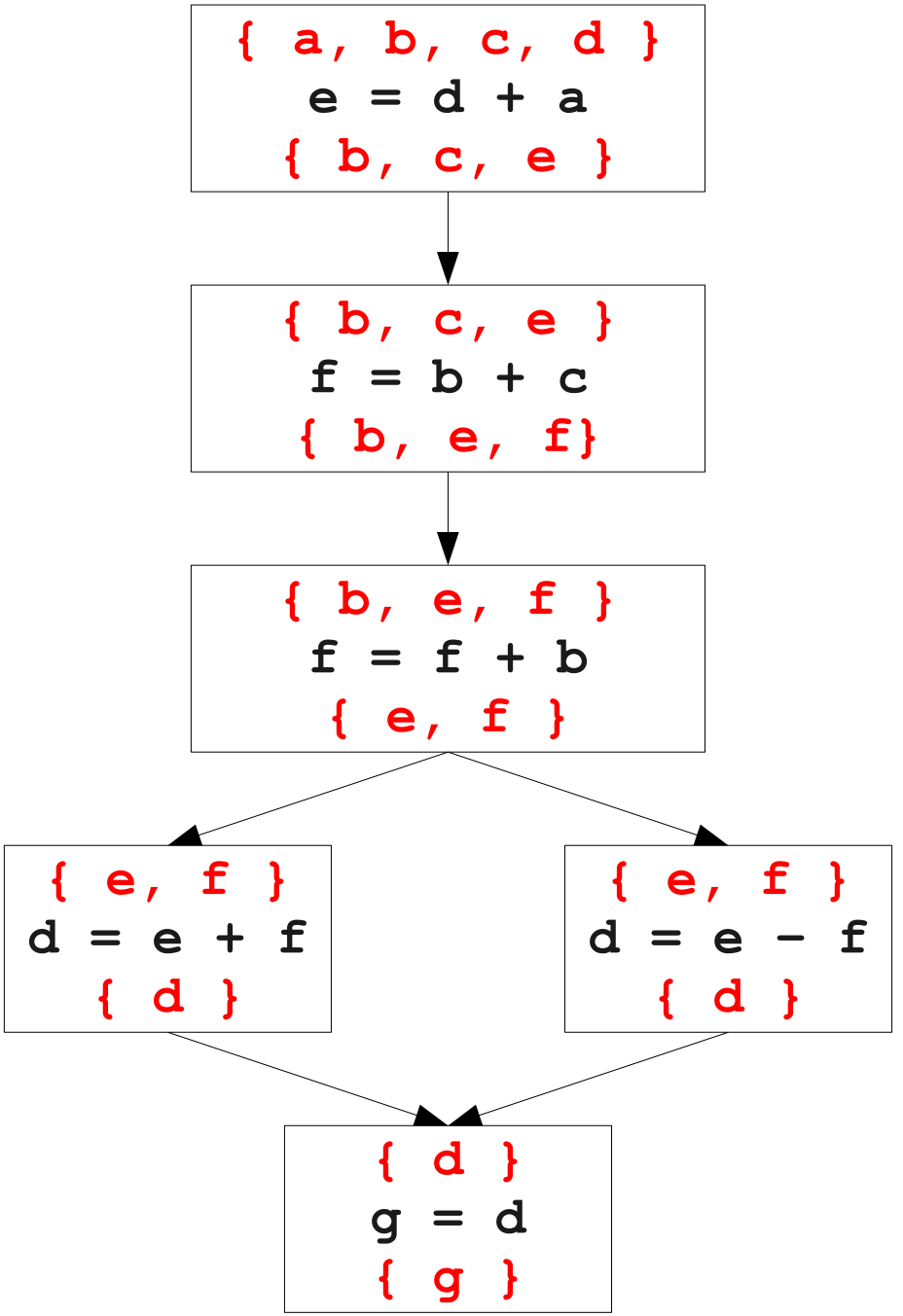
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



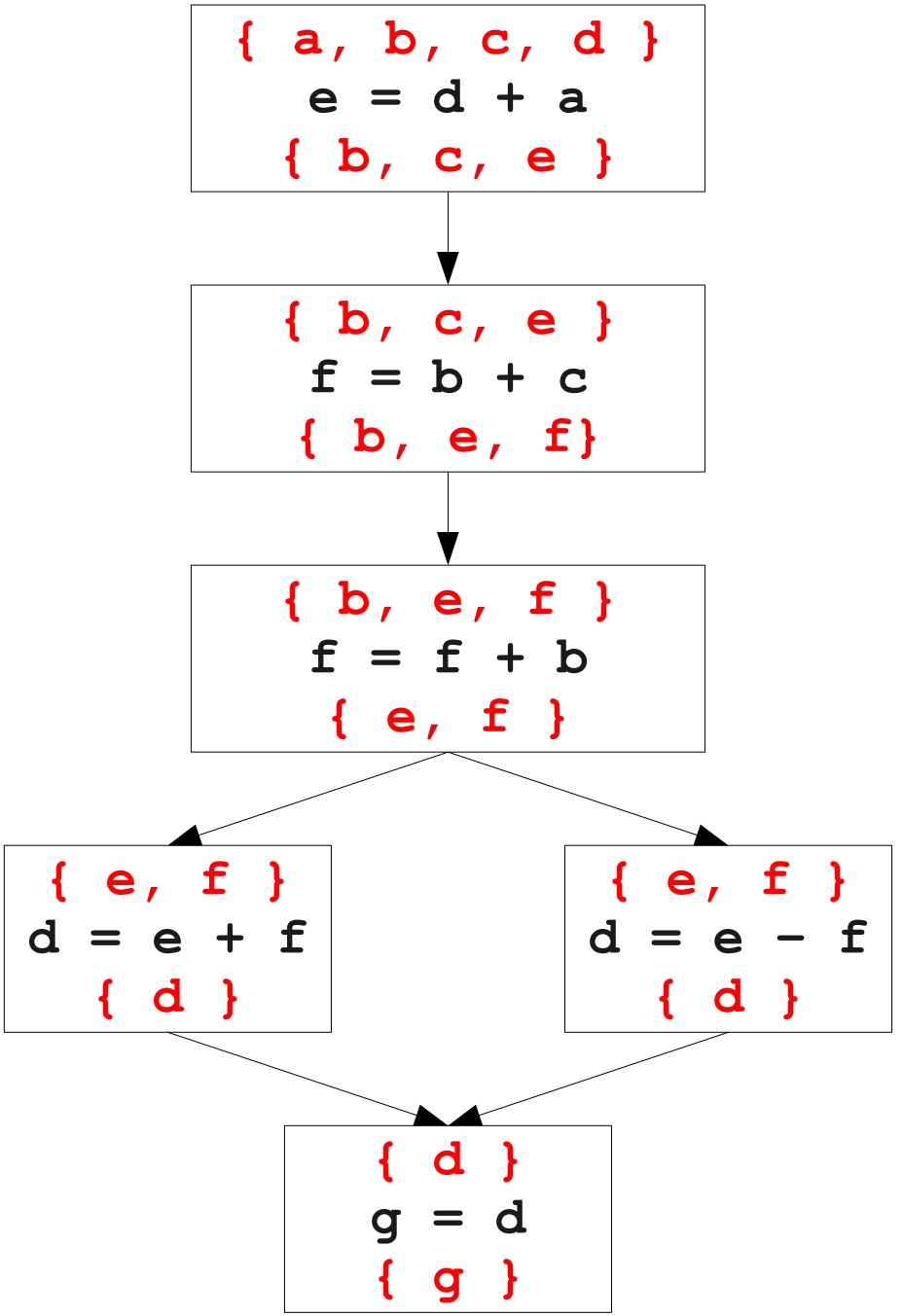
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

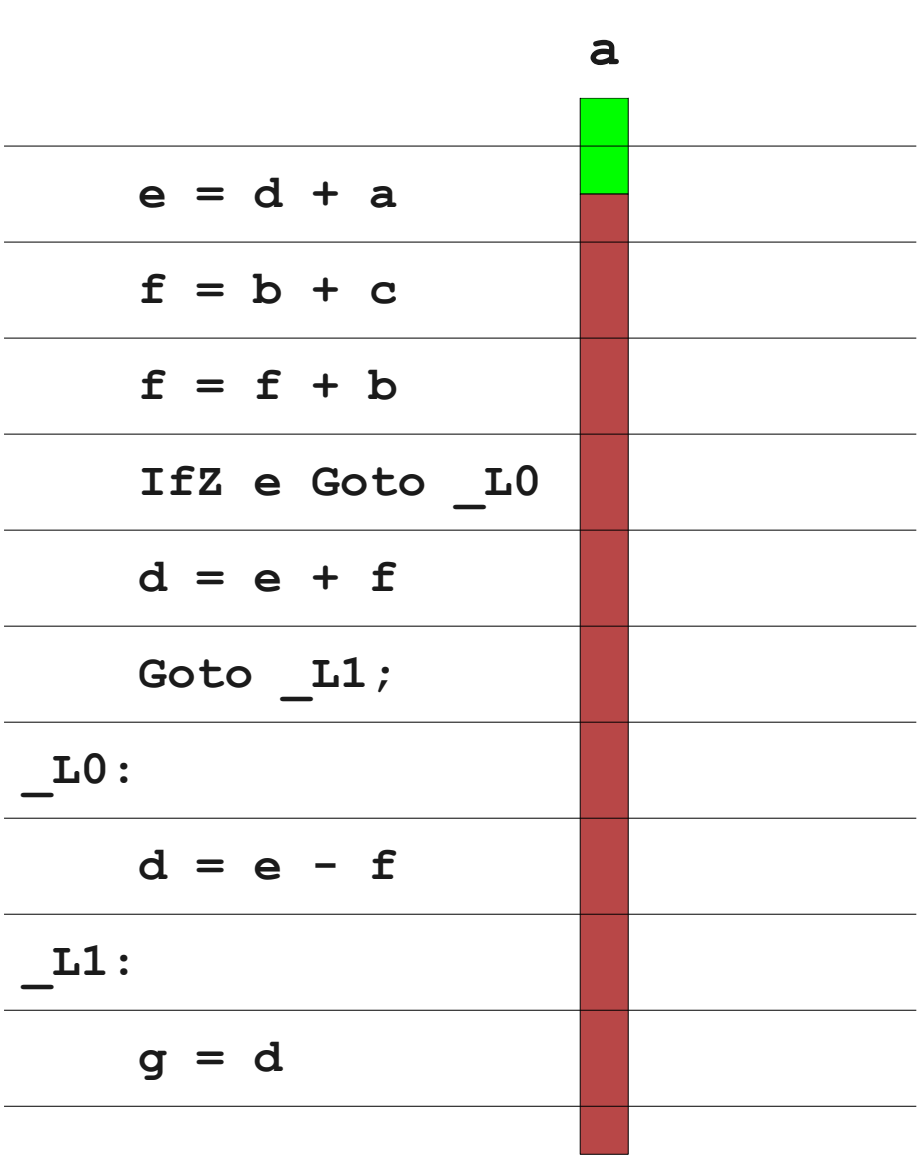


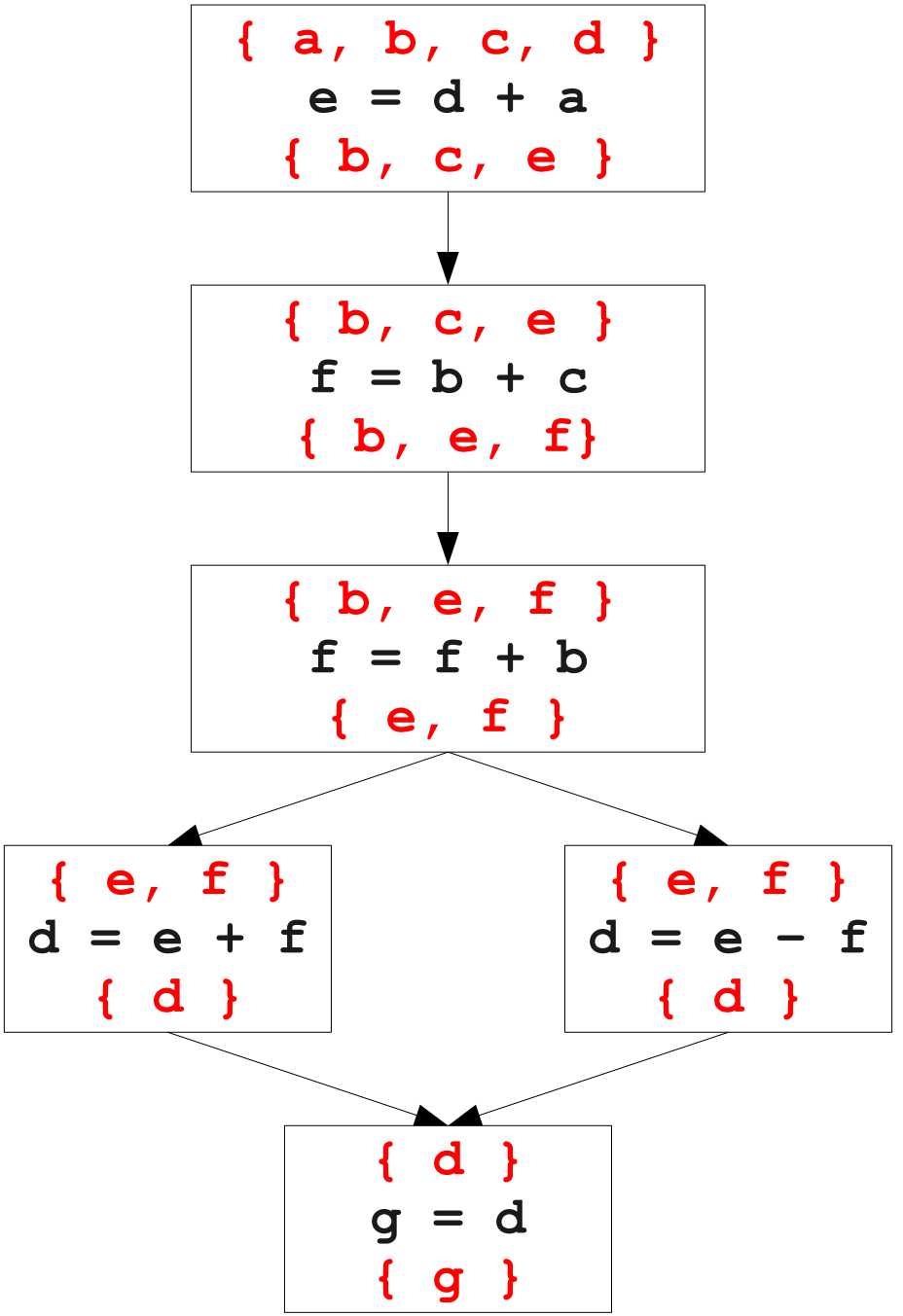
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

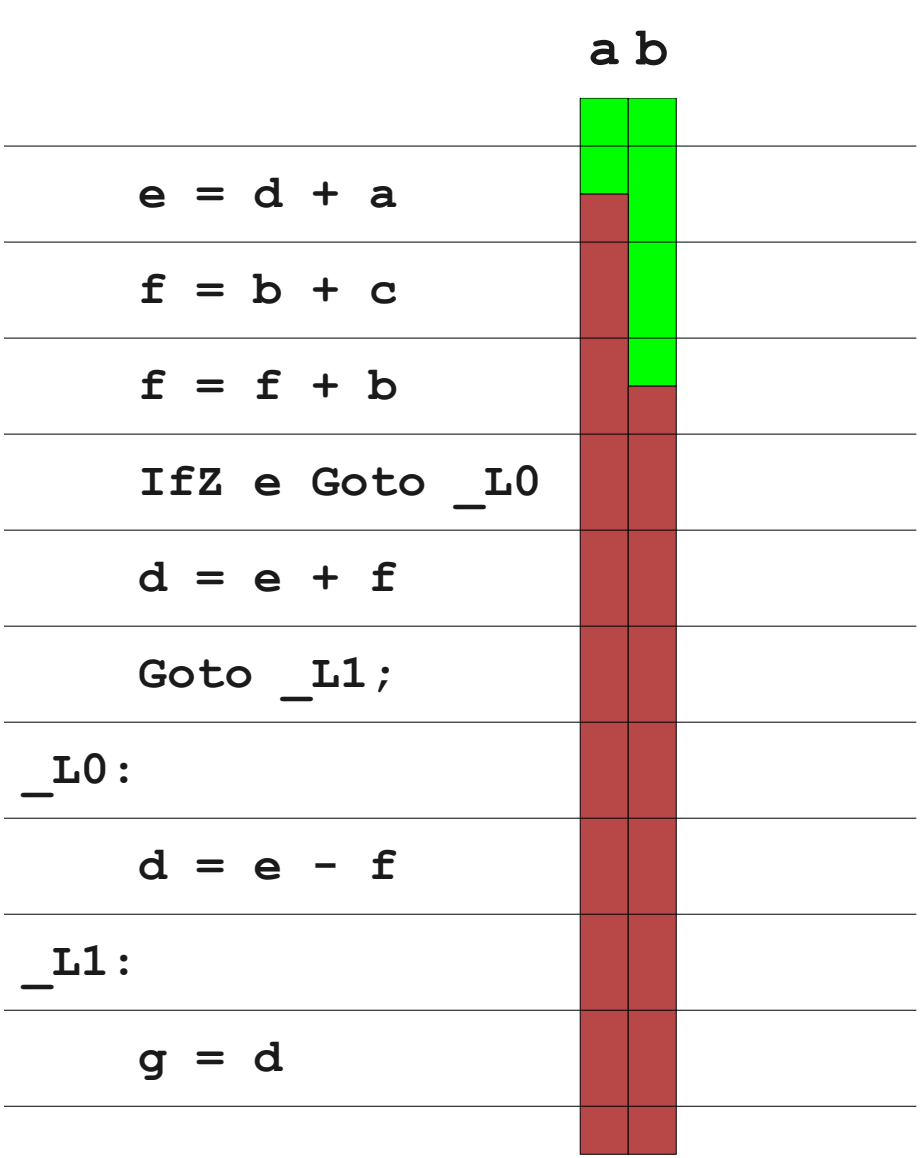


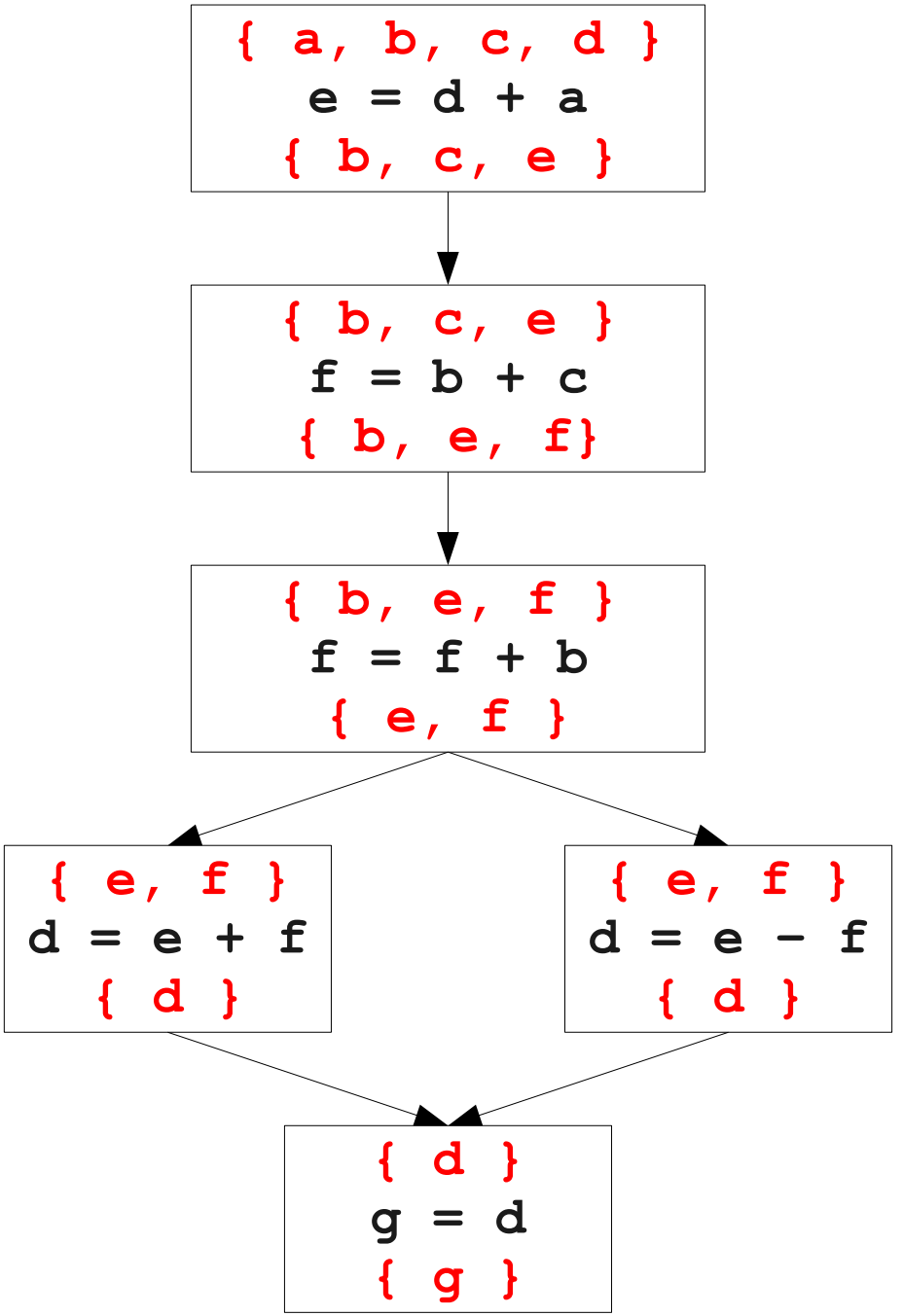
Live Ranges and Live Intervals



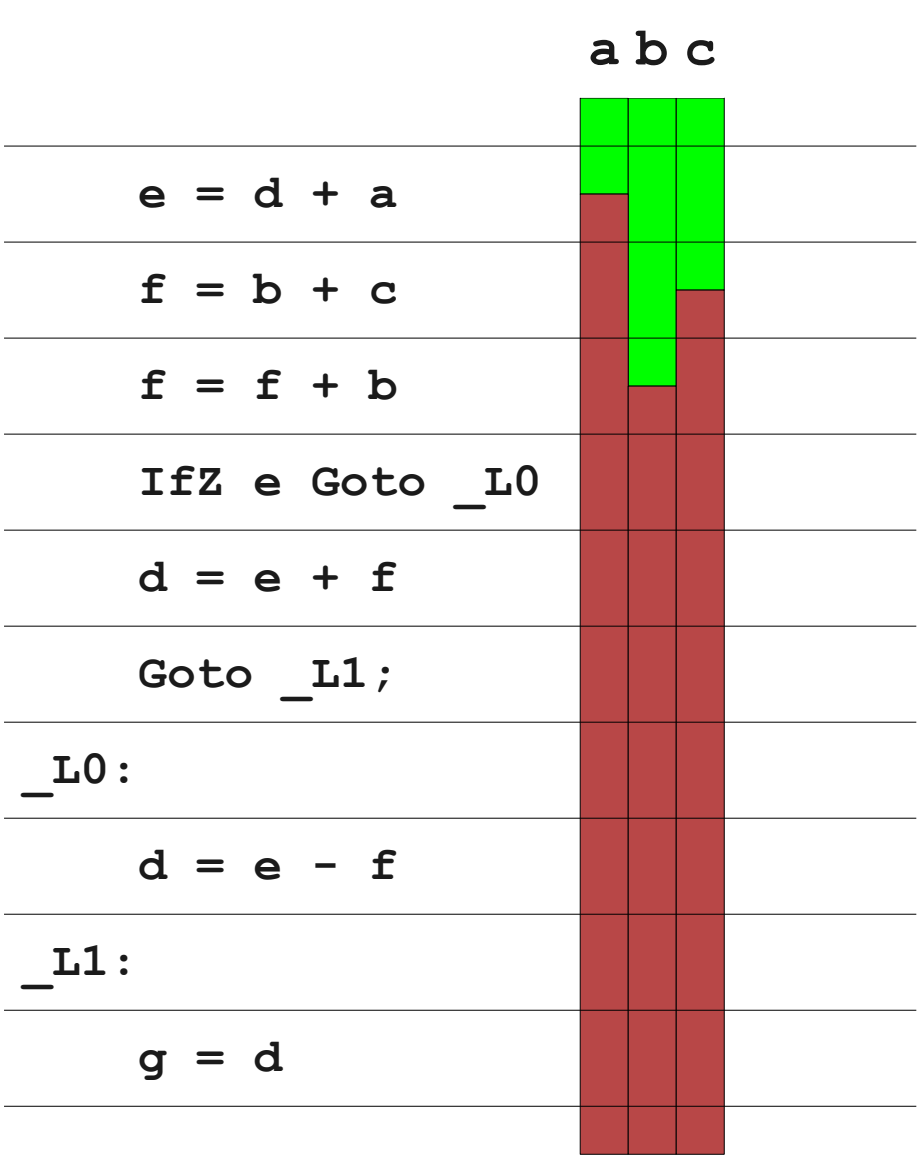


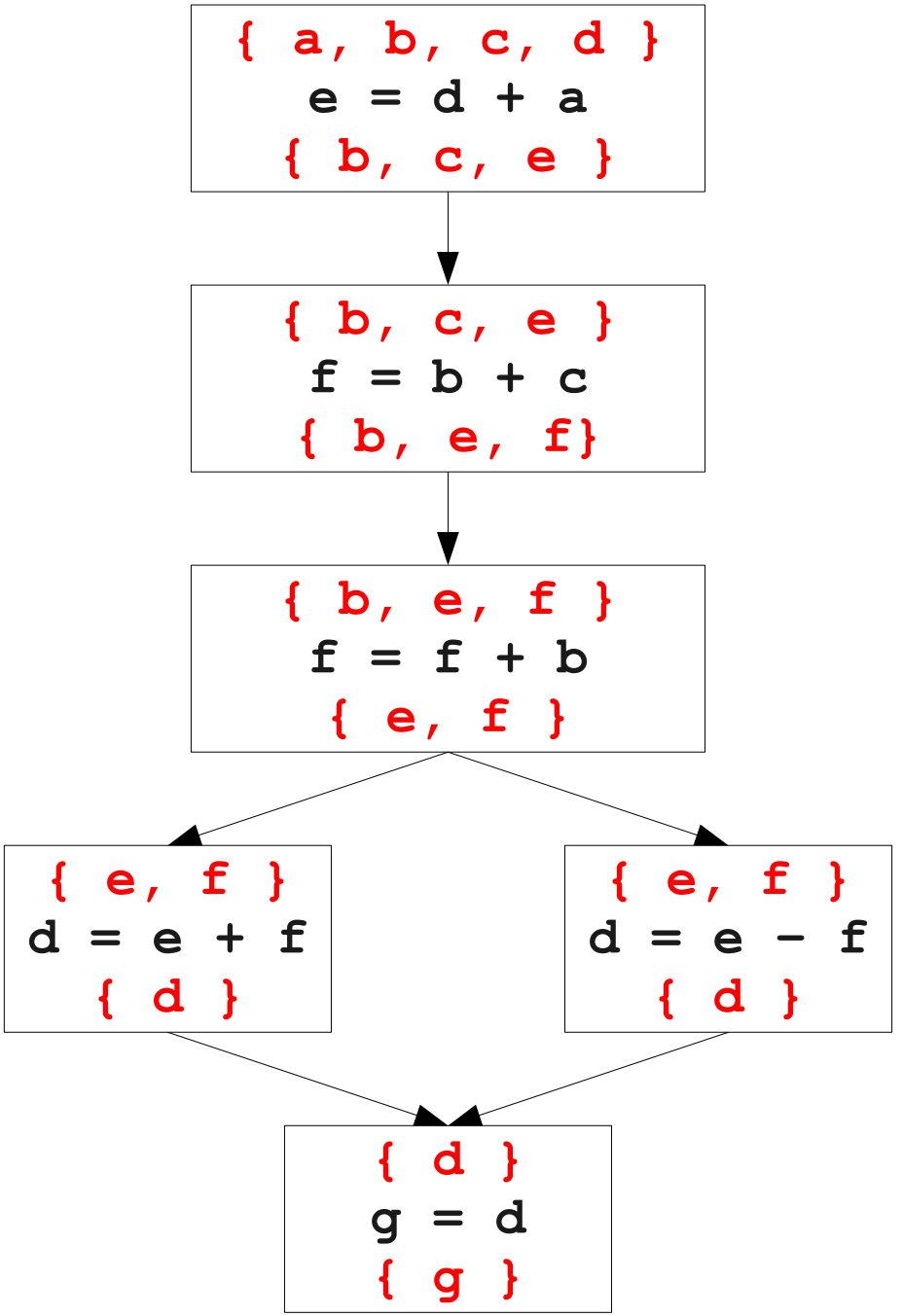
Live Ranges and Live Intervals



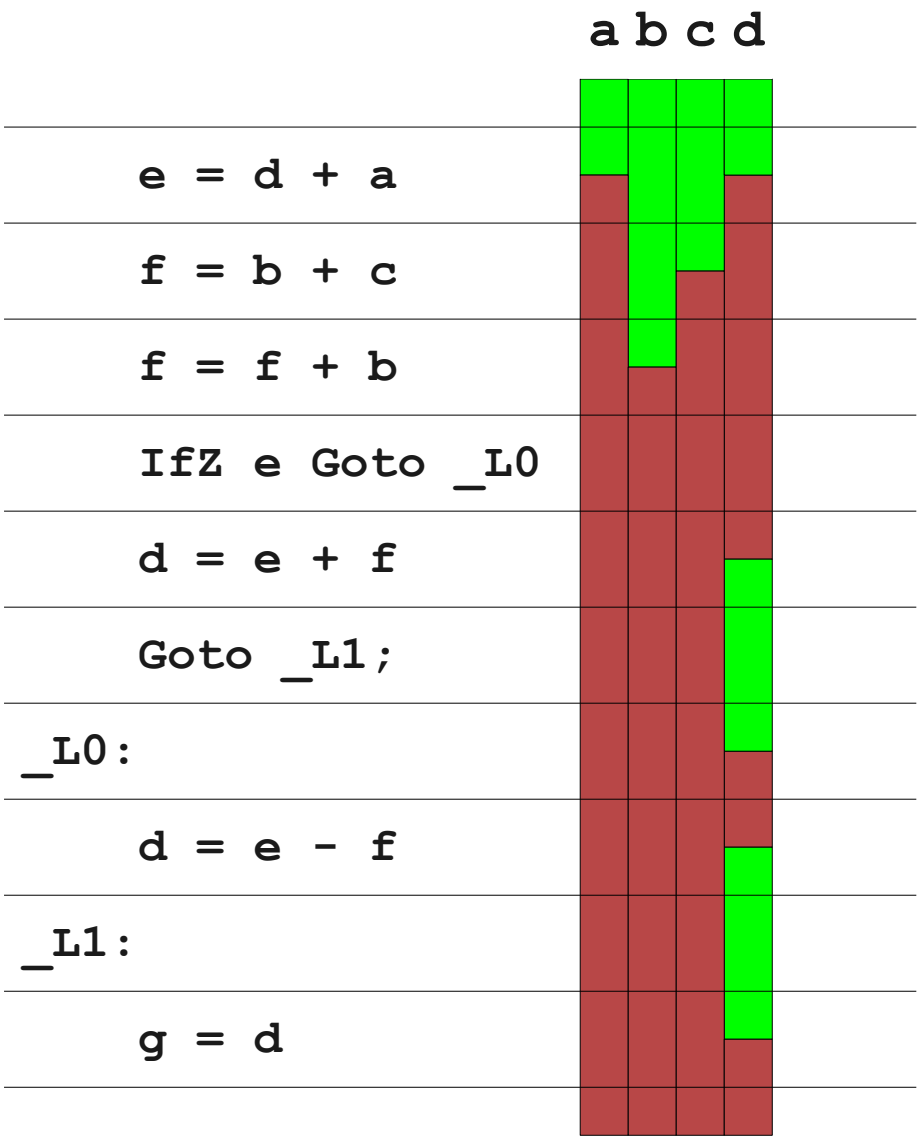


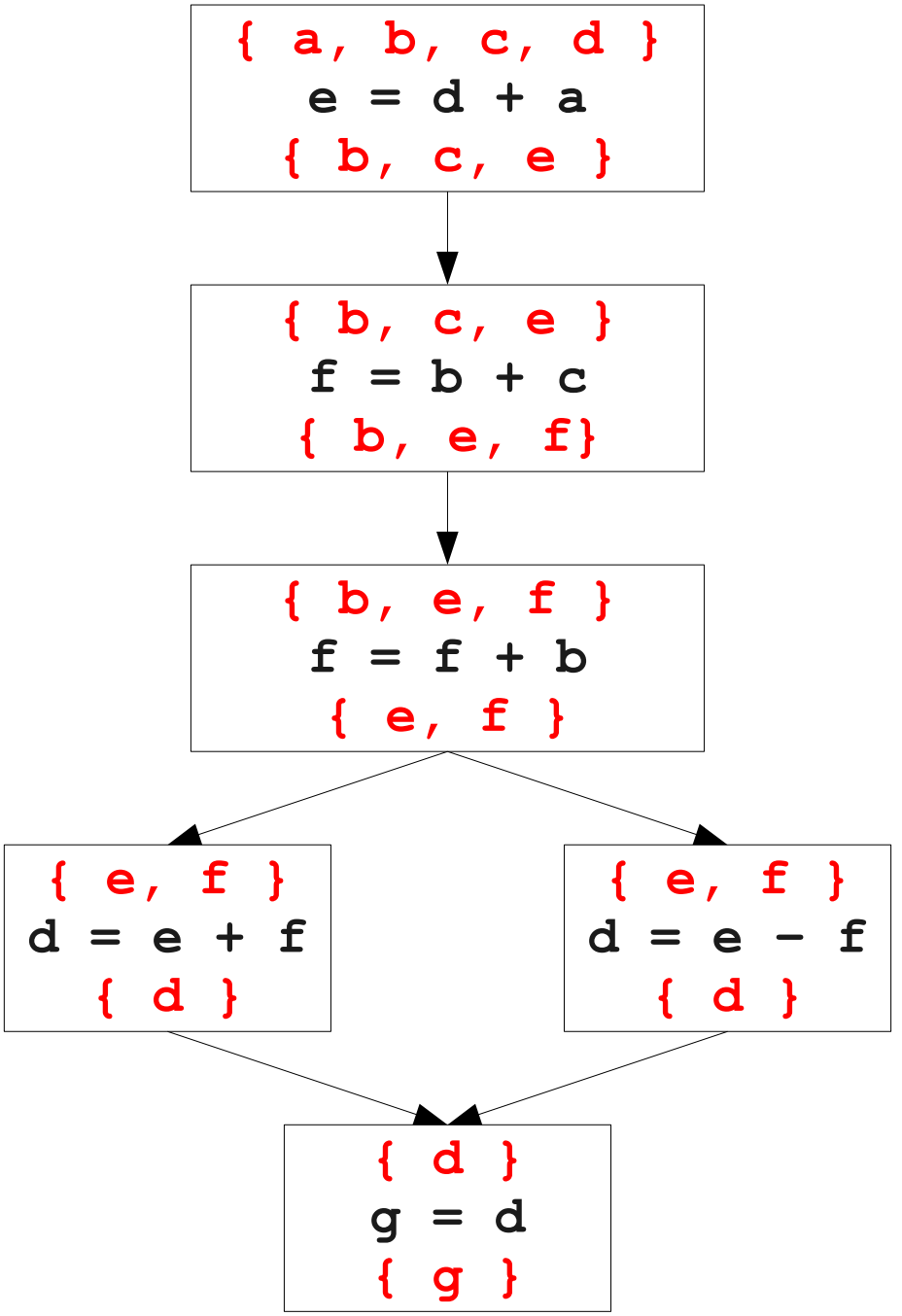
Live Ranges and Live Intervals



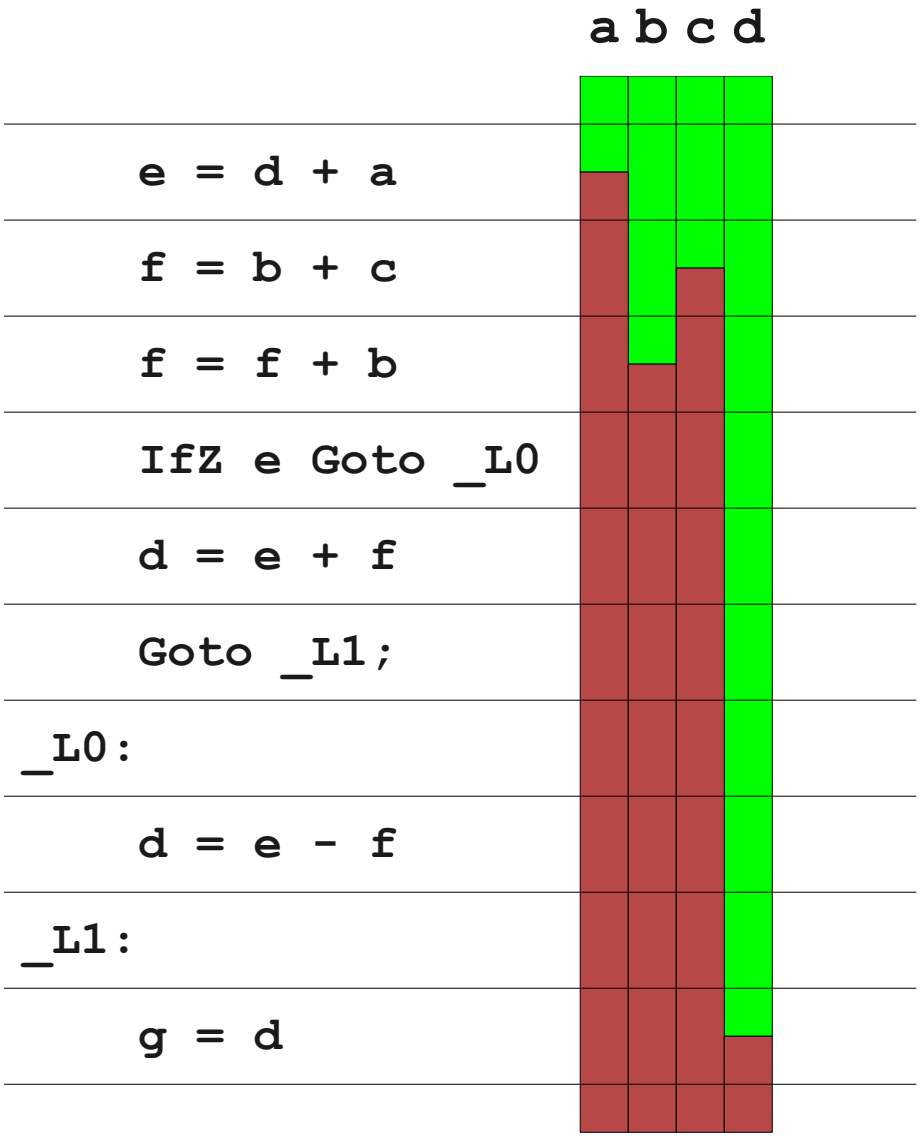


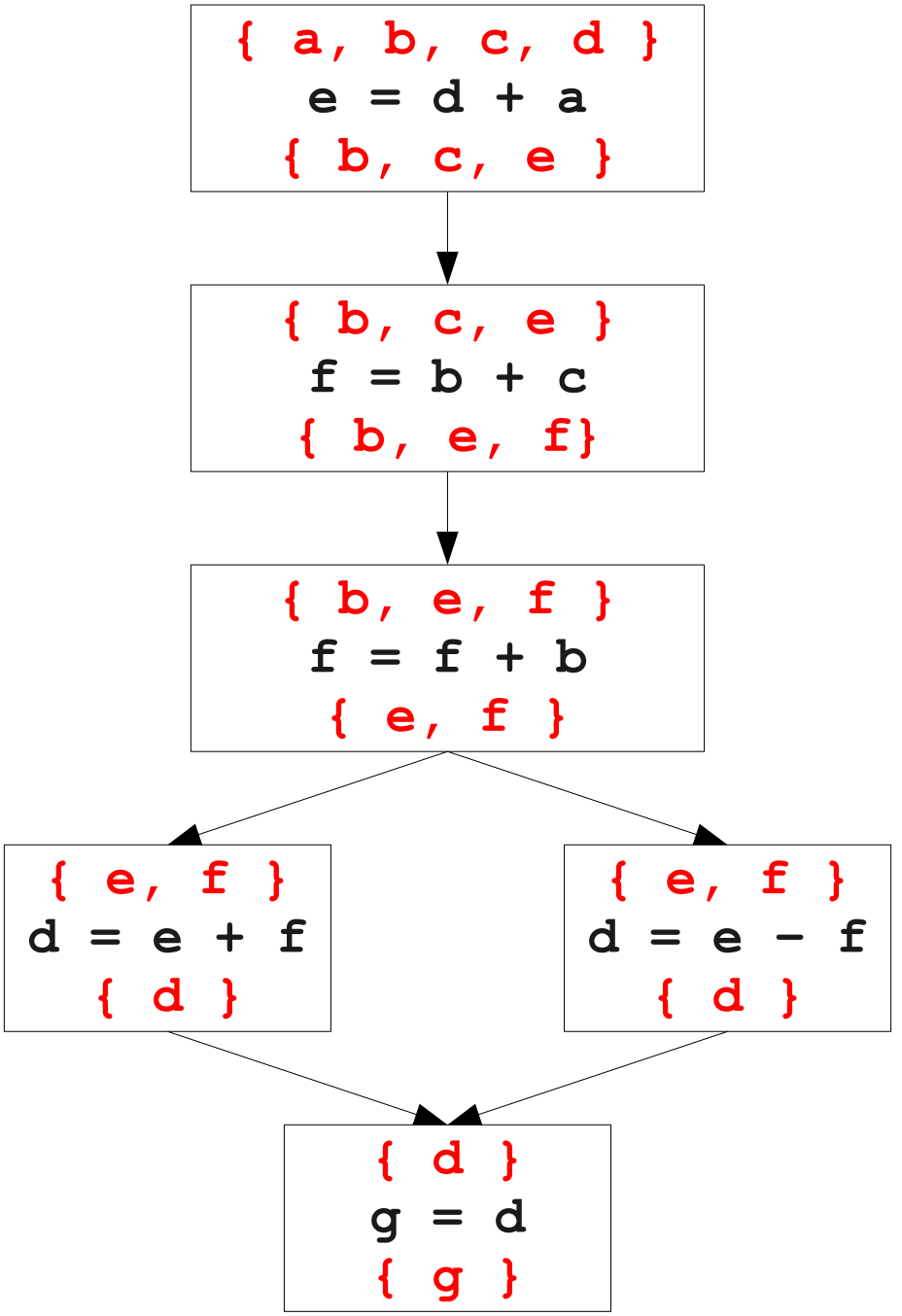
Live Ranges and Live Intervals



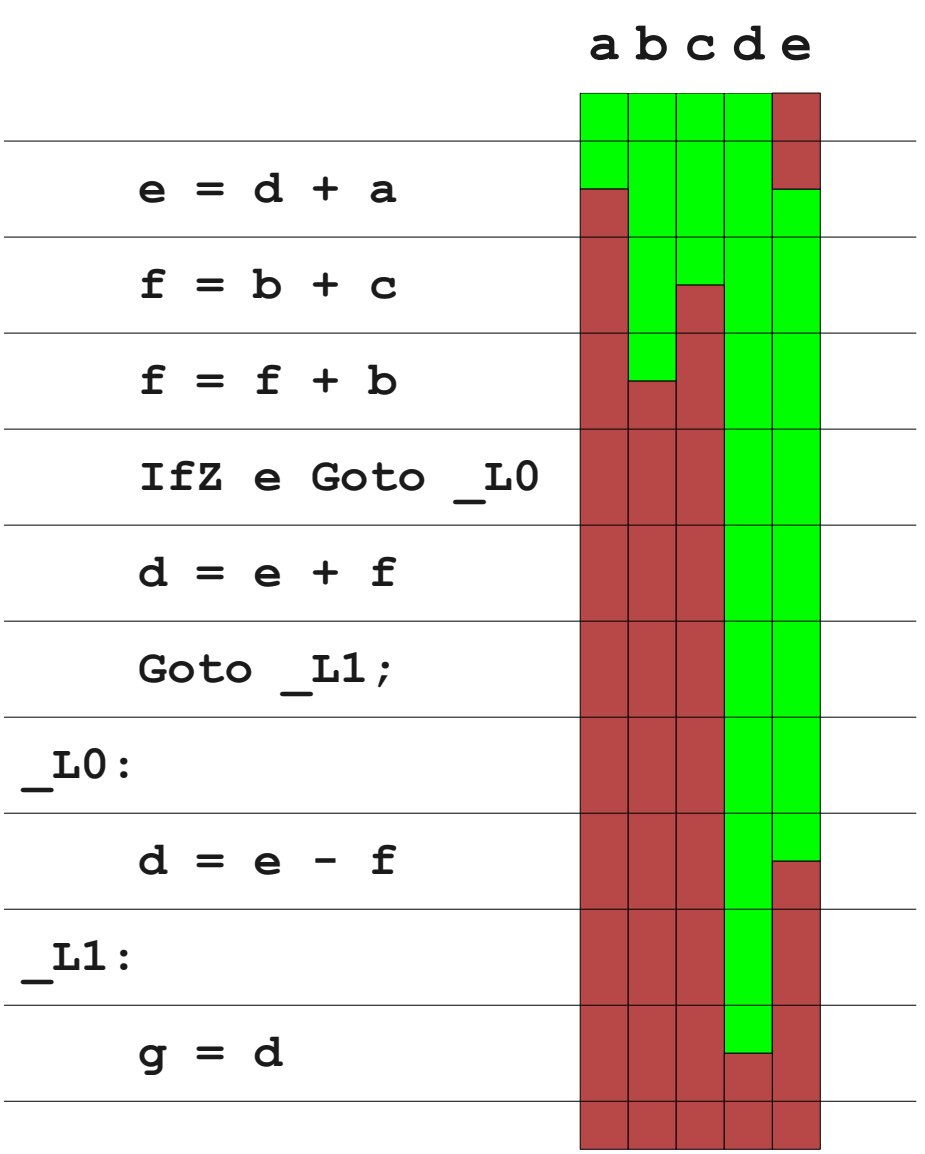


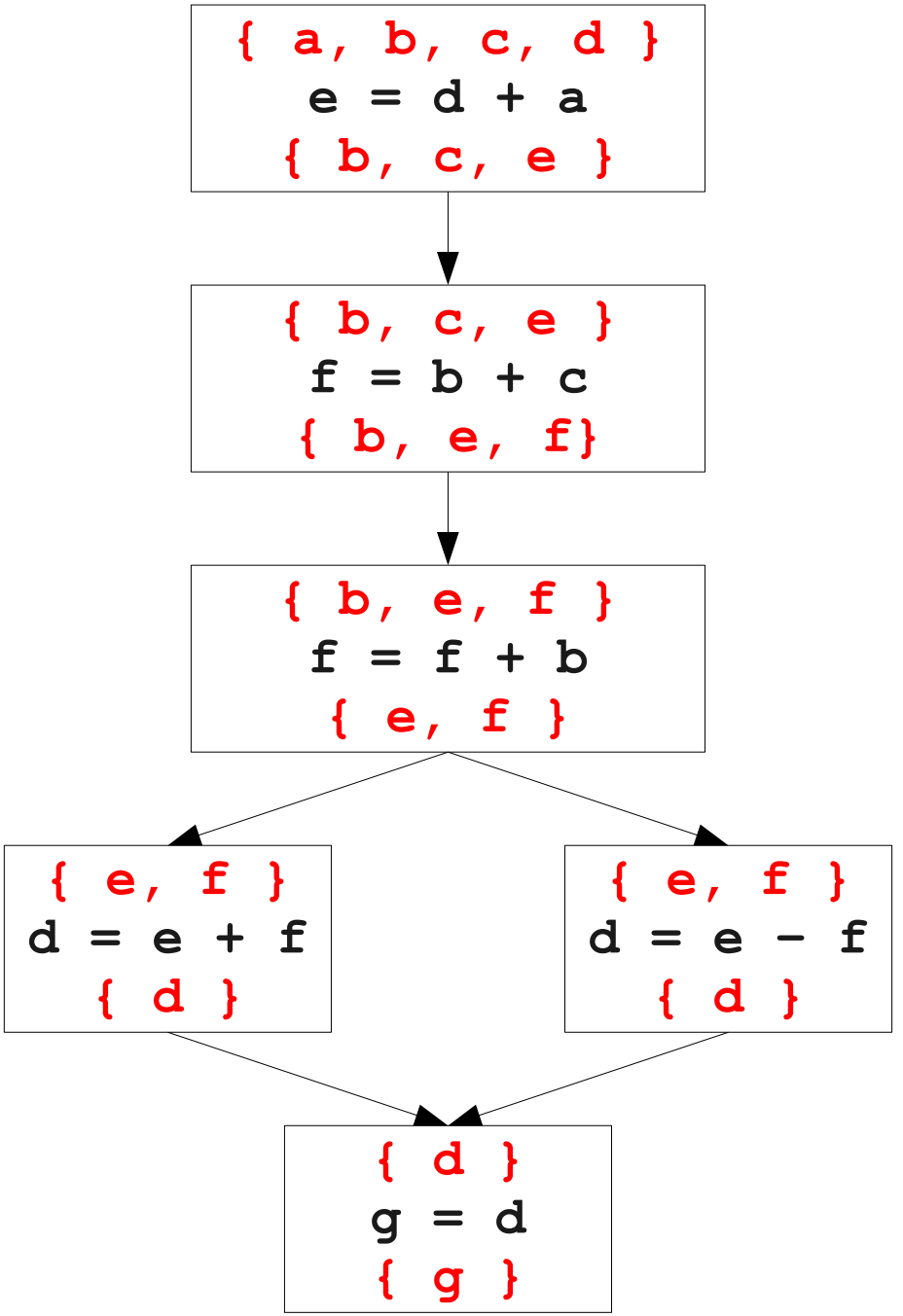
Live Ranges and Live Intervals



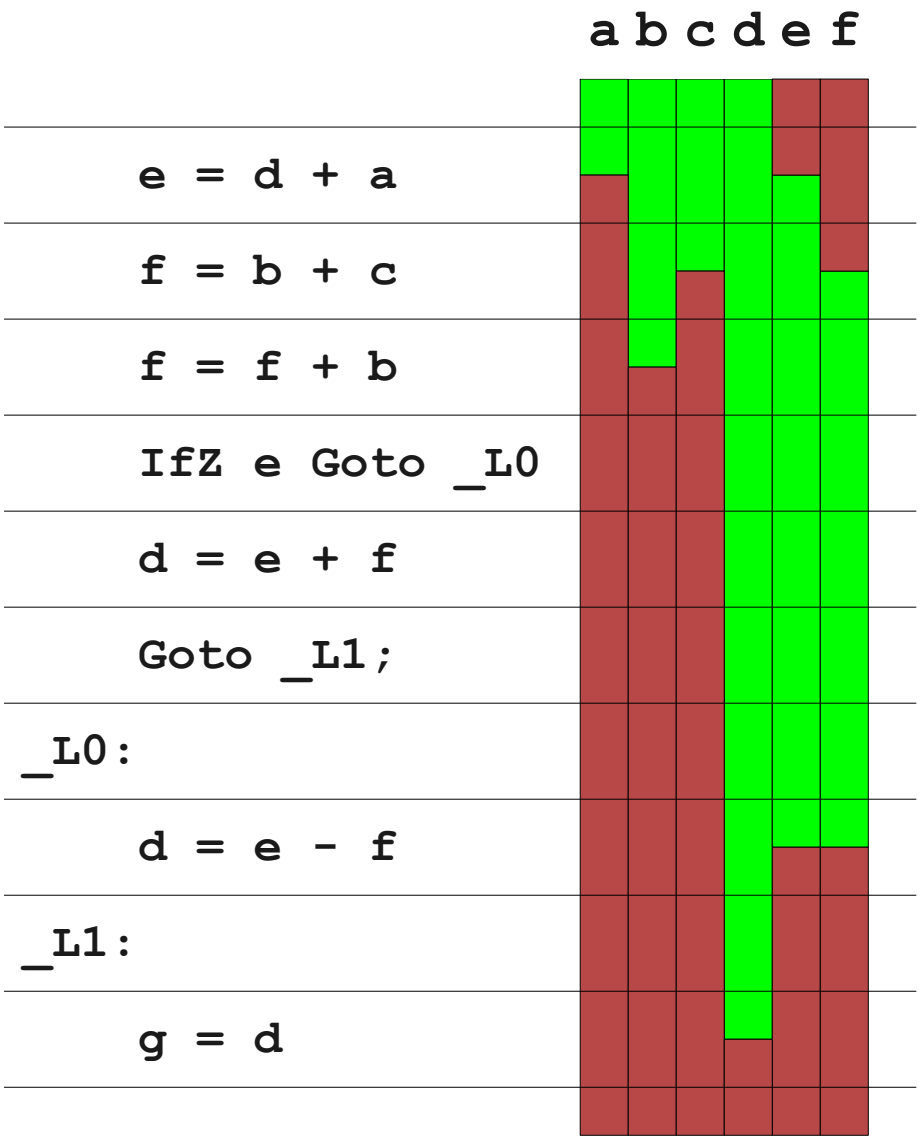


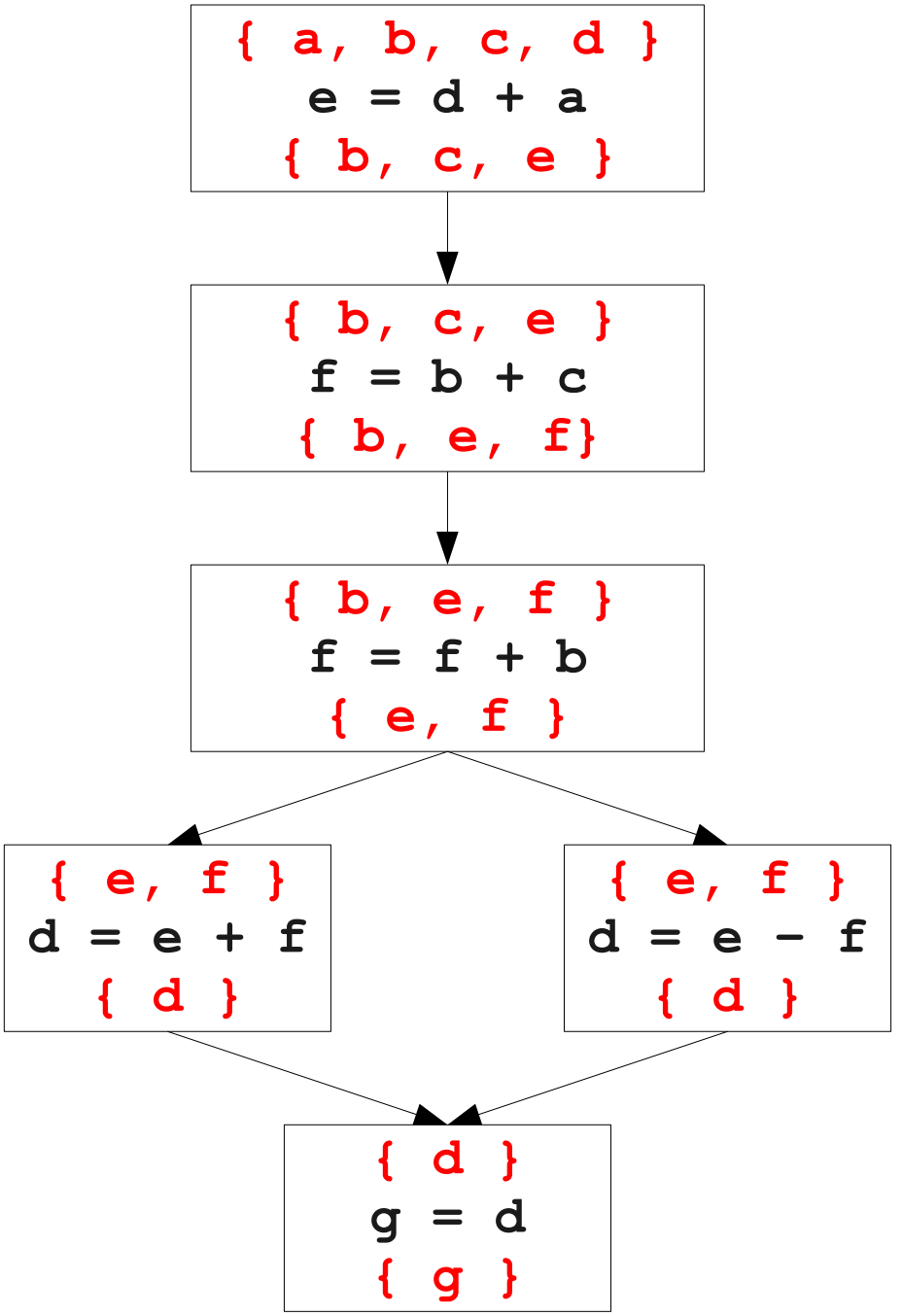
Live Ranges and Live Intervals



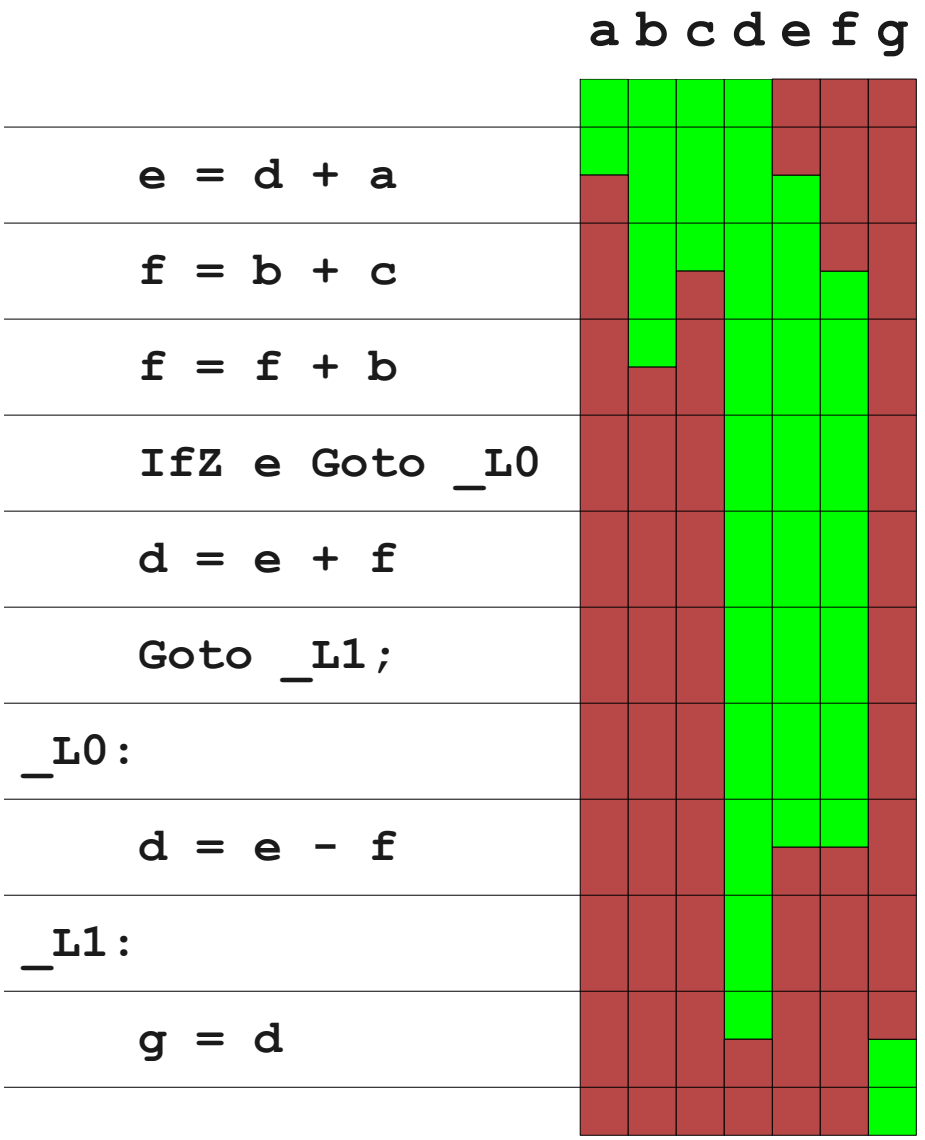


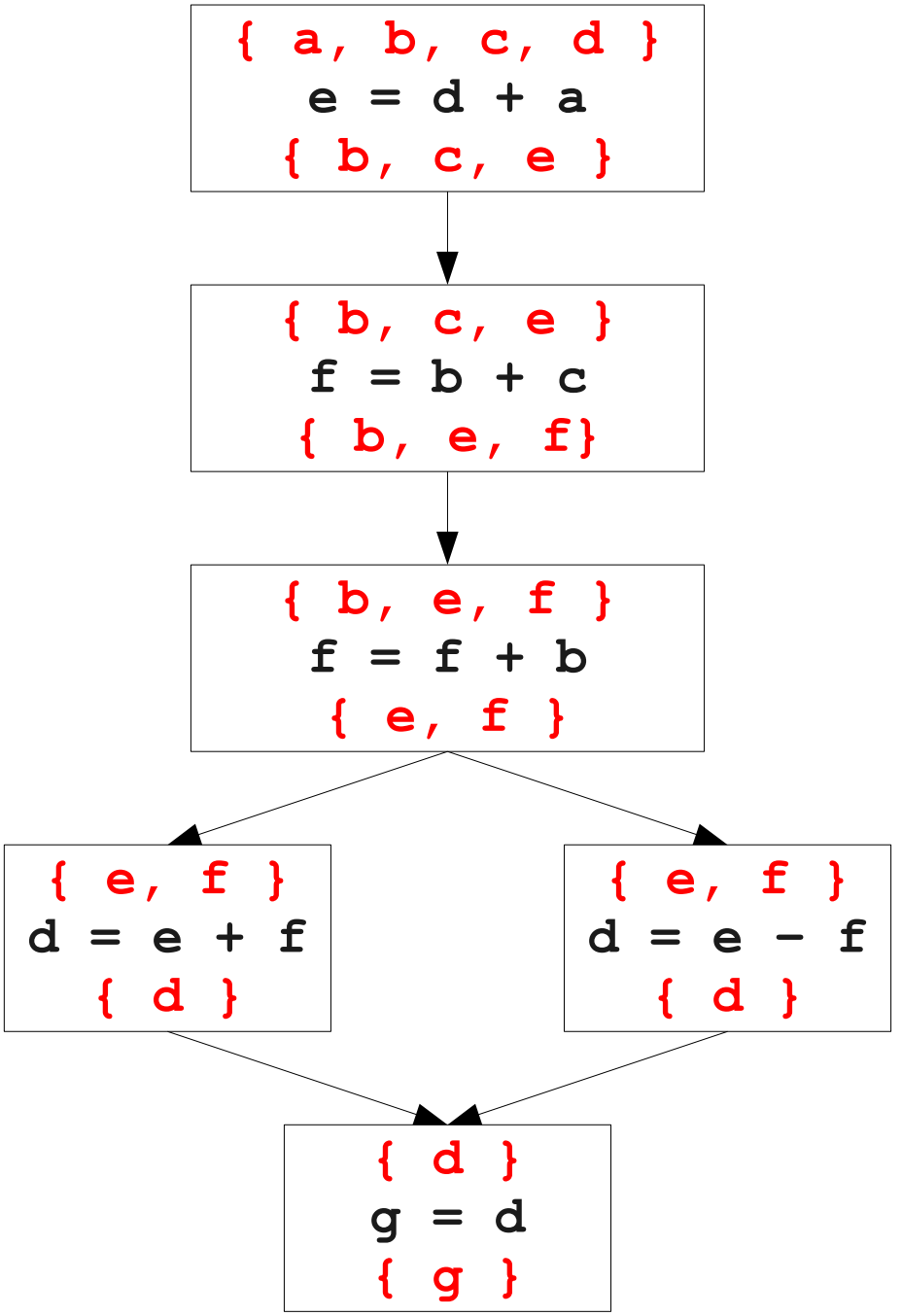
Live Ranges and Live Intervals





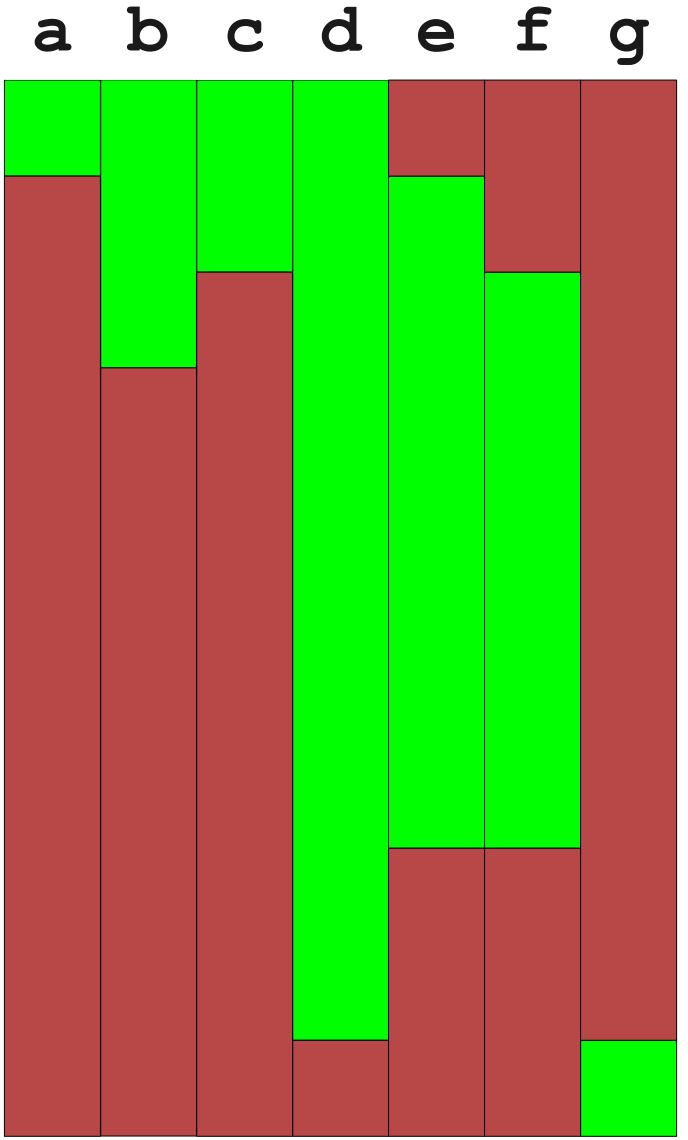
Live Ranges and Live Intervals



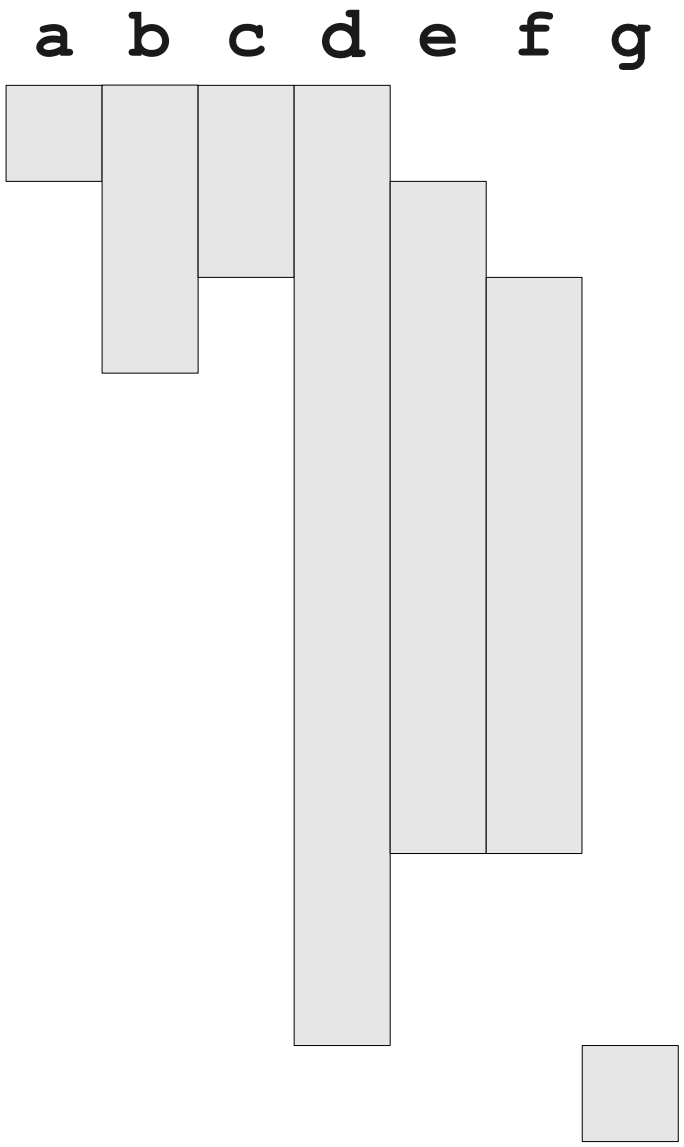


Register Allocation with Live Intervals

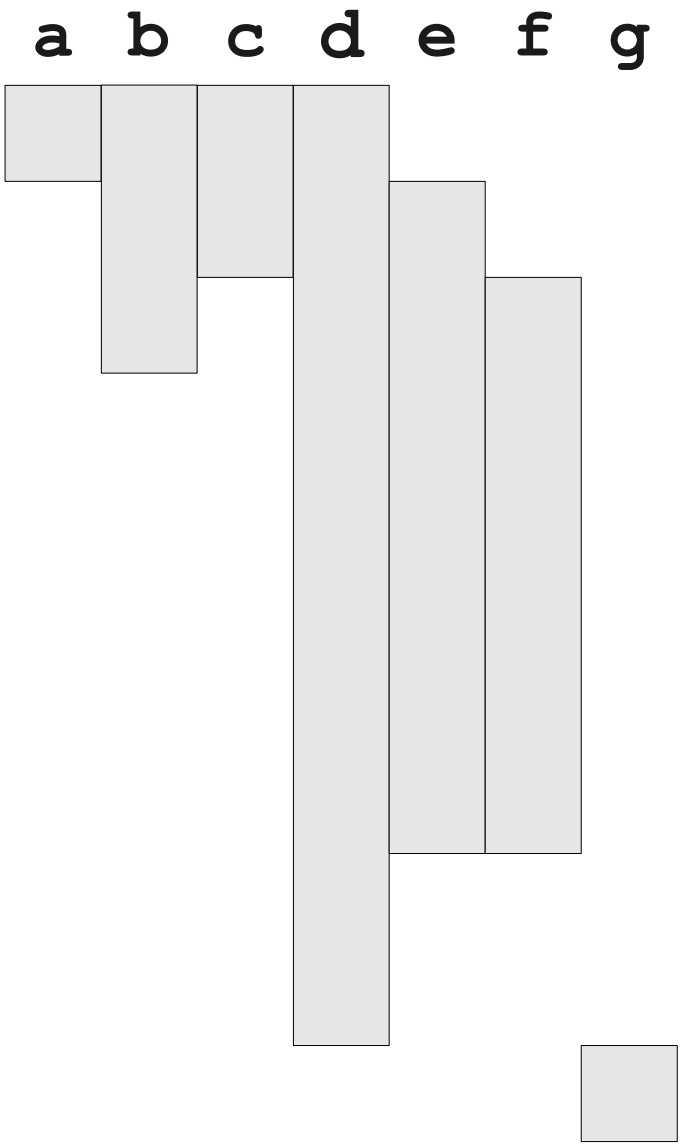
- Given the live intervals for all the variables in the program, we can allocate registers using a simple greedy algorithm.
- Idea: Track which registers are free at each point.
- When a live interval begins, give that variable a free register.
- When a live interval ends, the register is once again free.
- We can't always fit everything into a register; we'll see what do to in a minute.



Register Allocation with Live Intervals

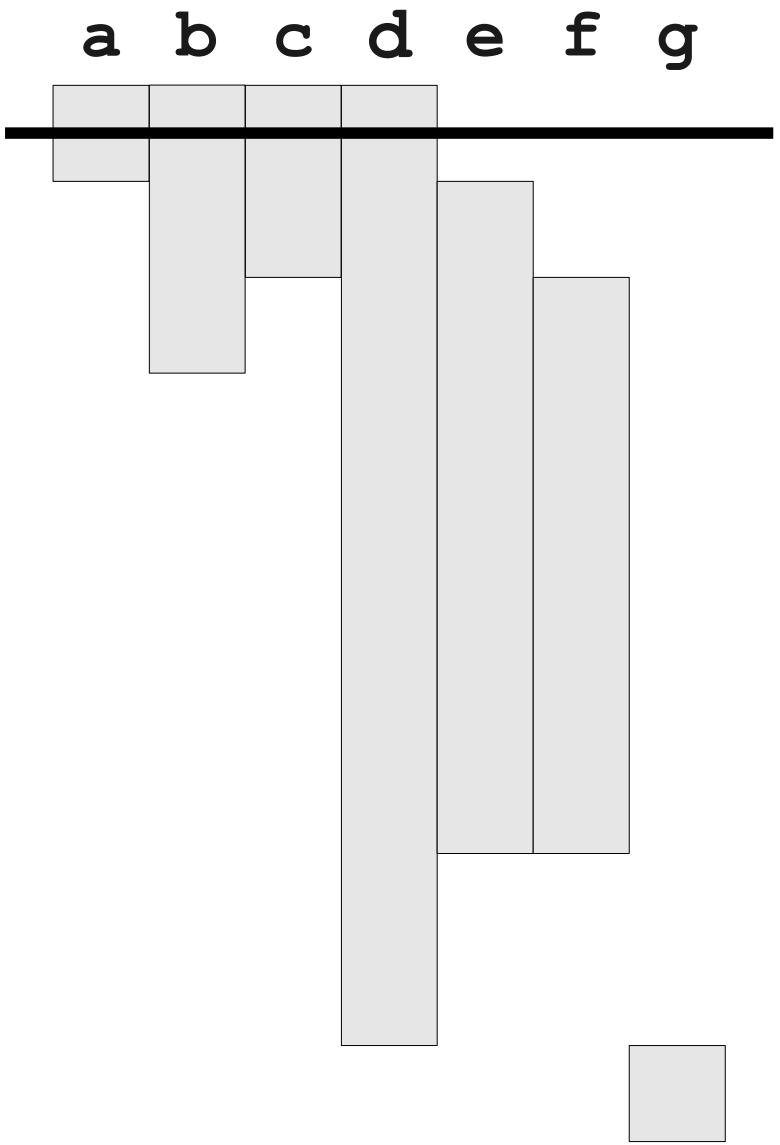


Register Allocation with Live Intervals



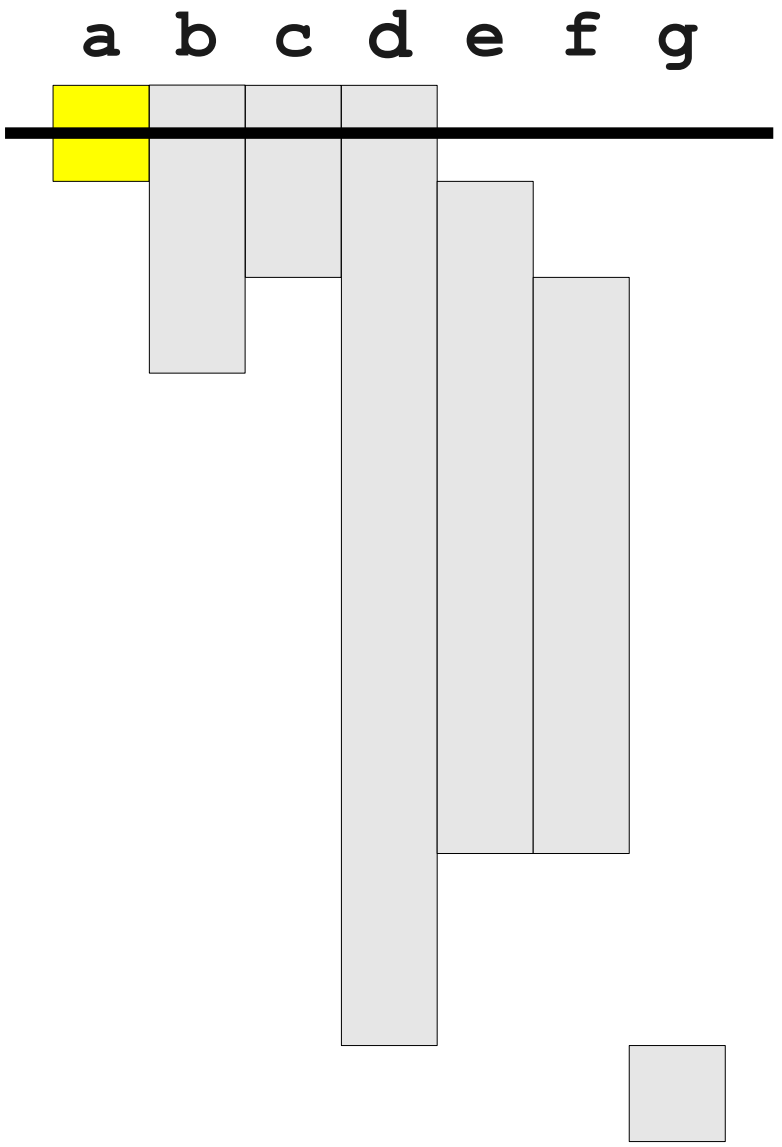
Free Registers

Register Allocation with Live Intervals



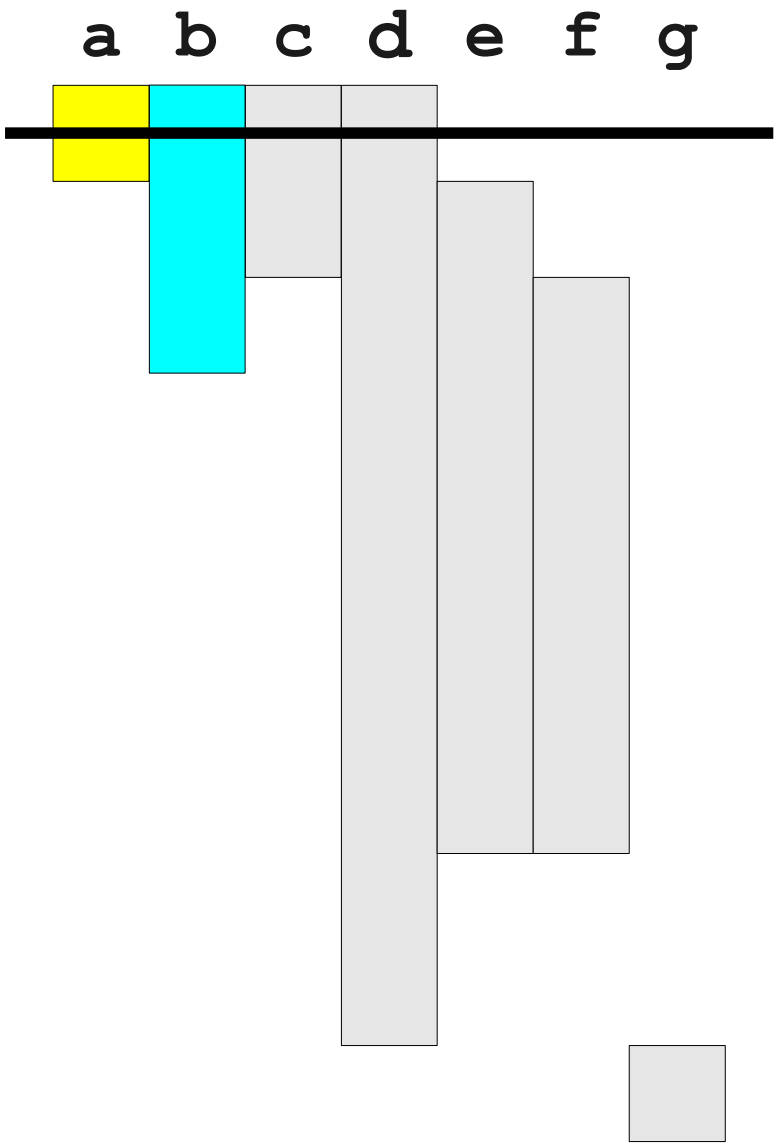
Free Registers

Register Allocation with Live Intervals



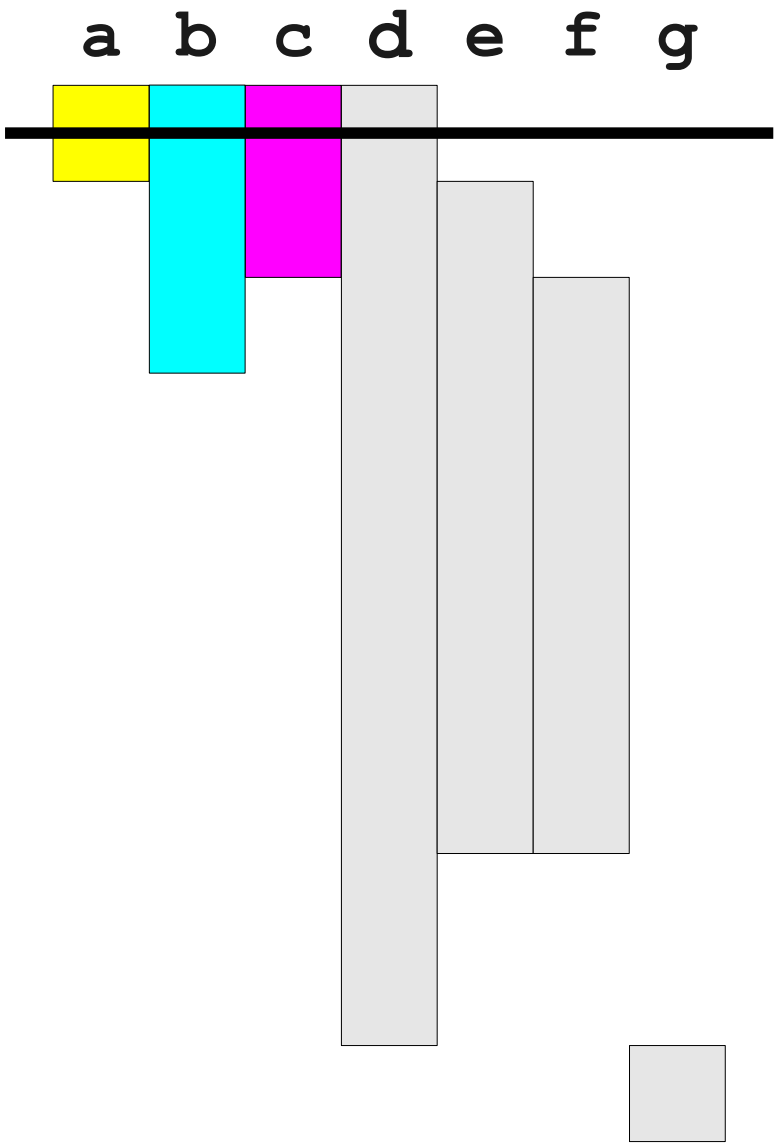
Free Registers

Register Allocation with Live Intervals



Free Registers

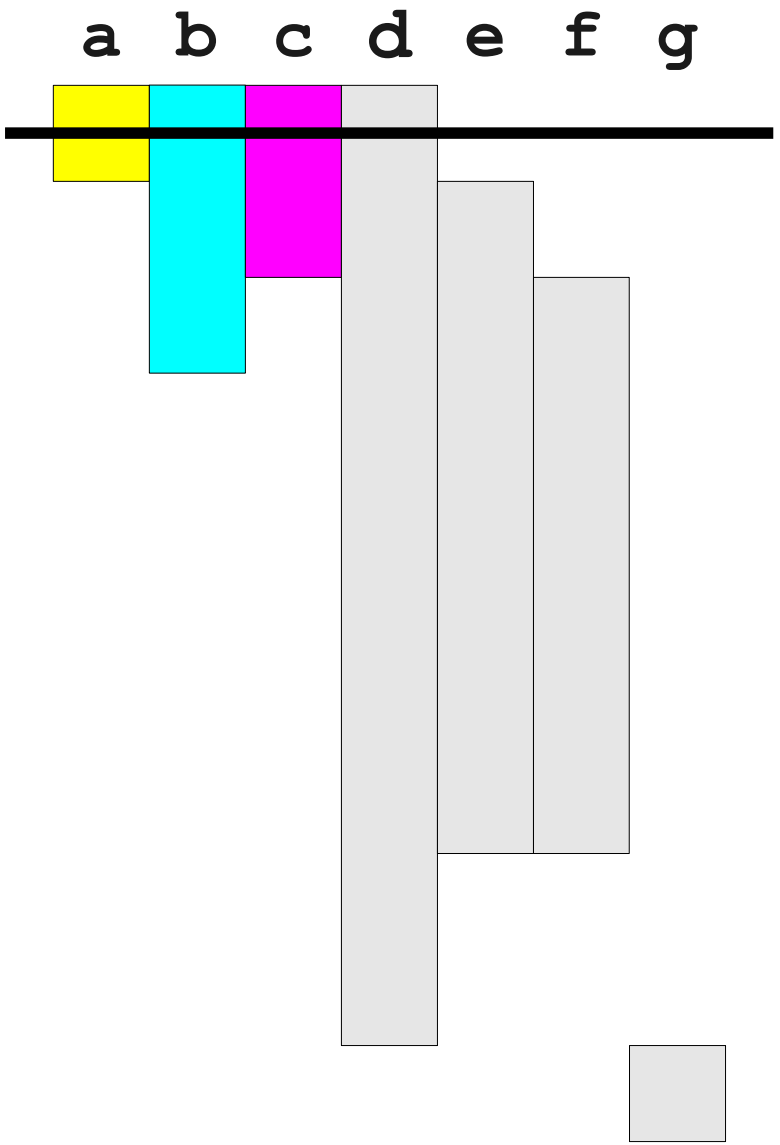
Register Allocation with Live Intervals



Free Registers

R_0	R_1	R_2

Register Allocation with Live Intervals



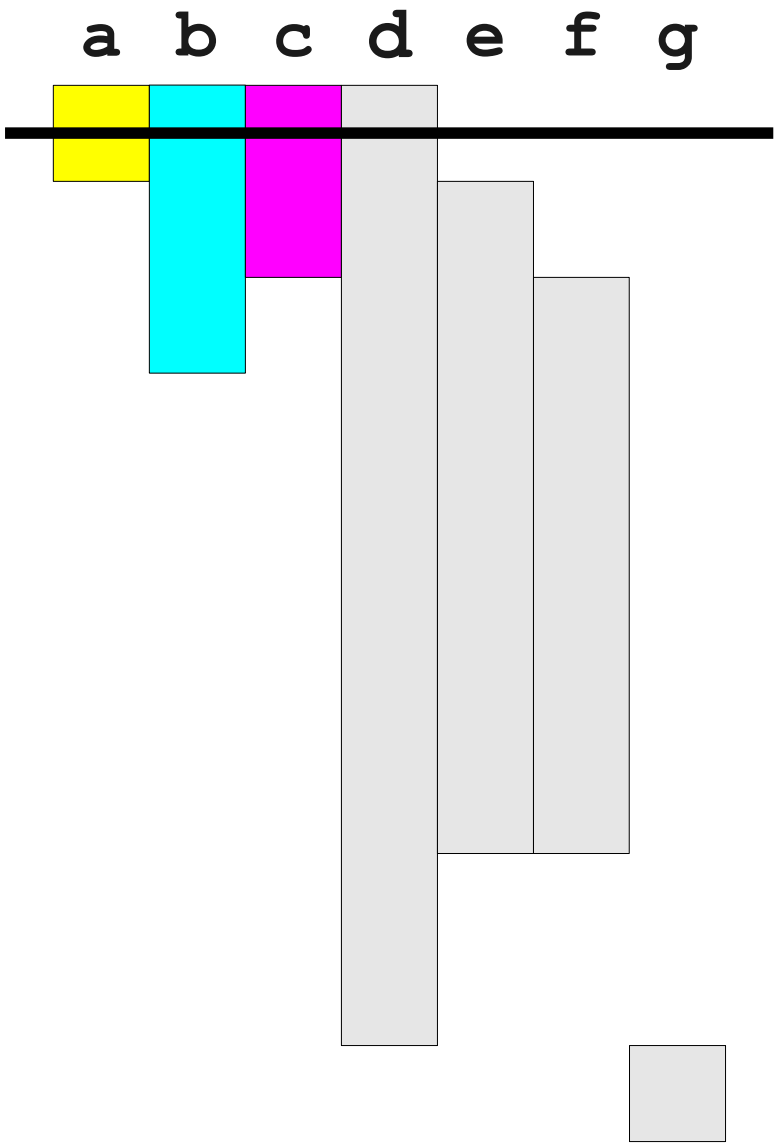
Free Registers

What do we do now?

Register Spilling

- If a register cannot be found for a variable v, we may need to **spill** a variable.
- When a variable is spilled, it is stored in memory rather than a register.
- When we need a register for the spilled variable, temporarily evict a register to memory.
- When done with that register, write its value to the storage spot for v (if necessary) and load the old value back.
- Note: Some register allocation algorithms can handle spilling much more intelligently than this.
- Spilling is slow, but sometimes necessary.

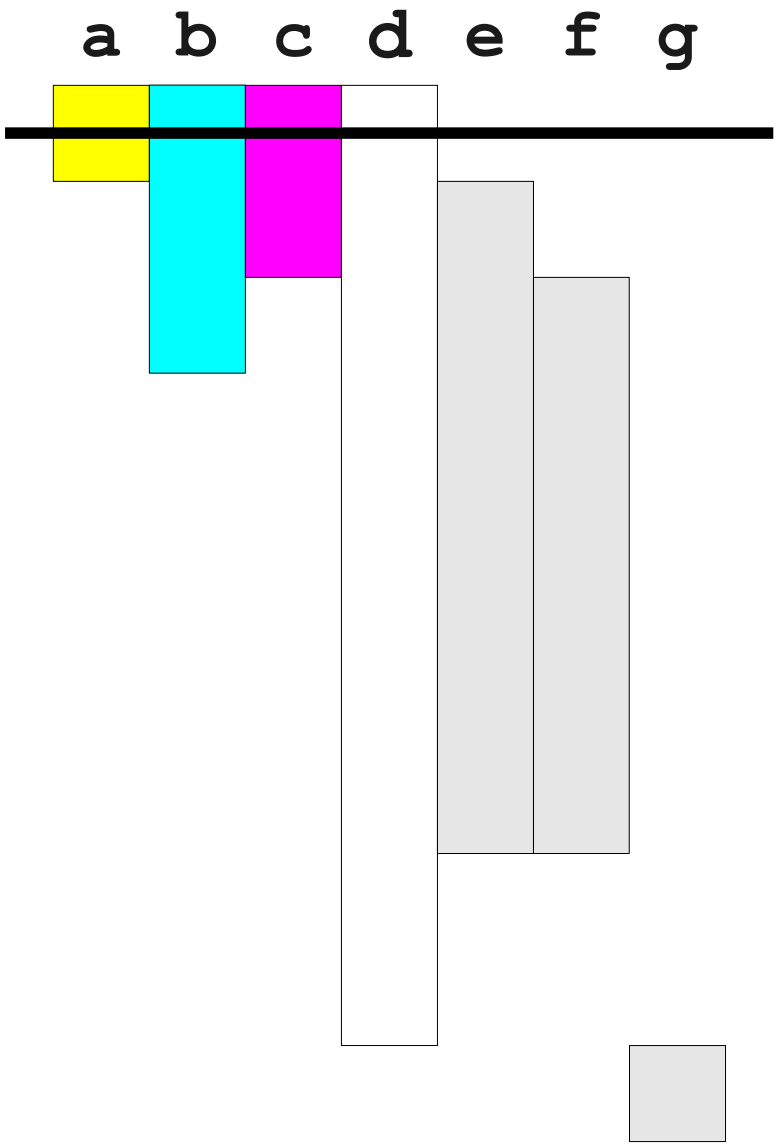
Register Allocation with Live Intervals



Free Registers

R_0	R_1	R_2

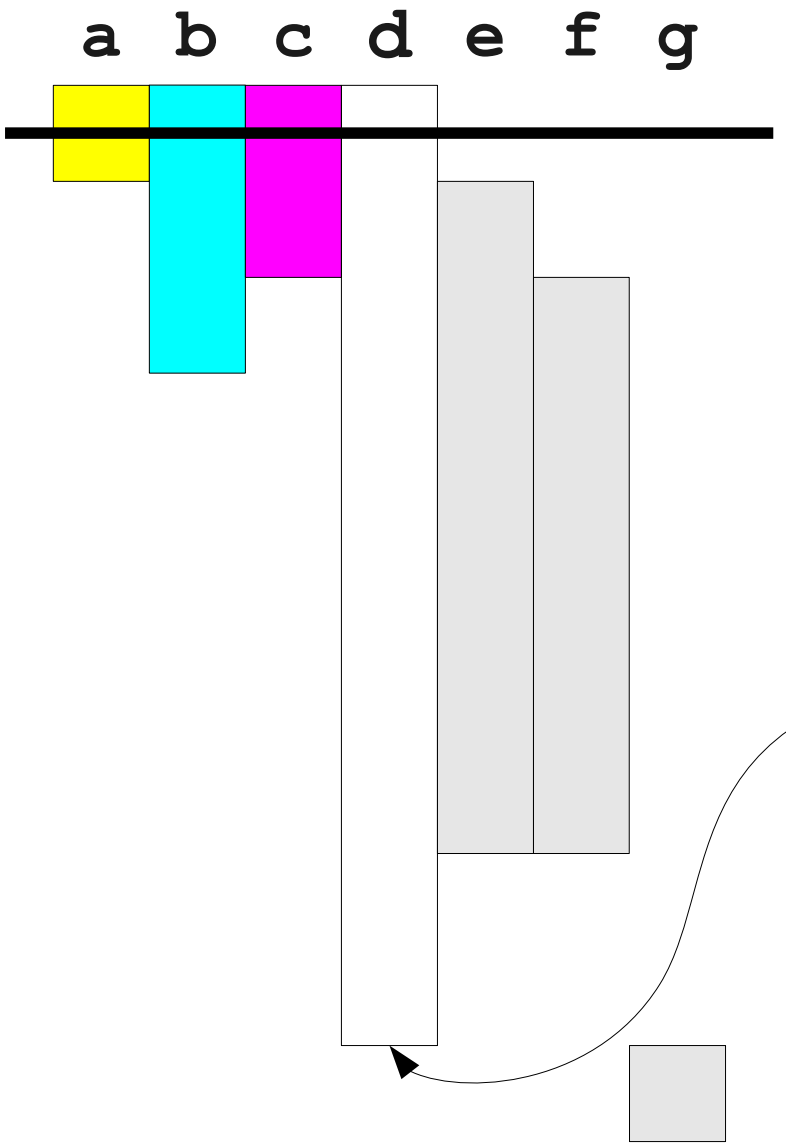
Register Allocation with Live Intervals



Free Registers

R_0	R_1	R_2

Register Allocation with Live Intervals

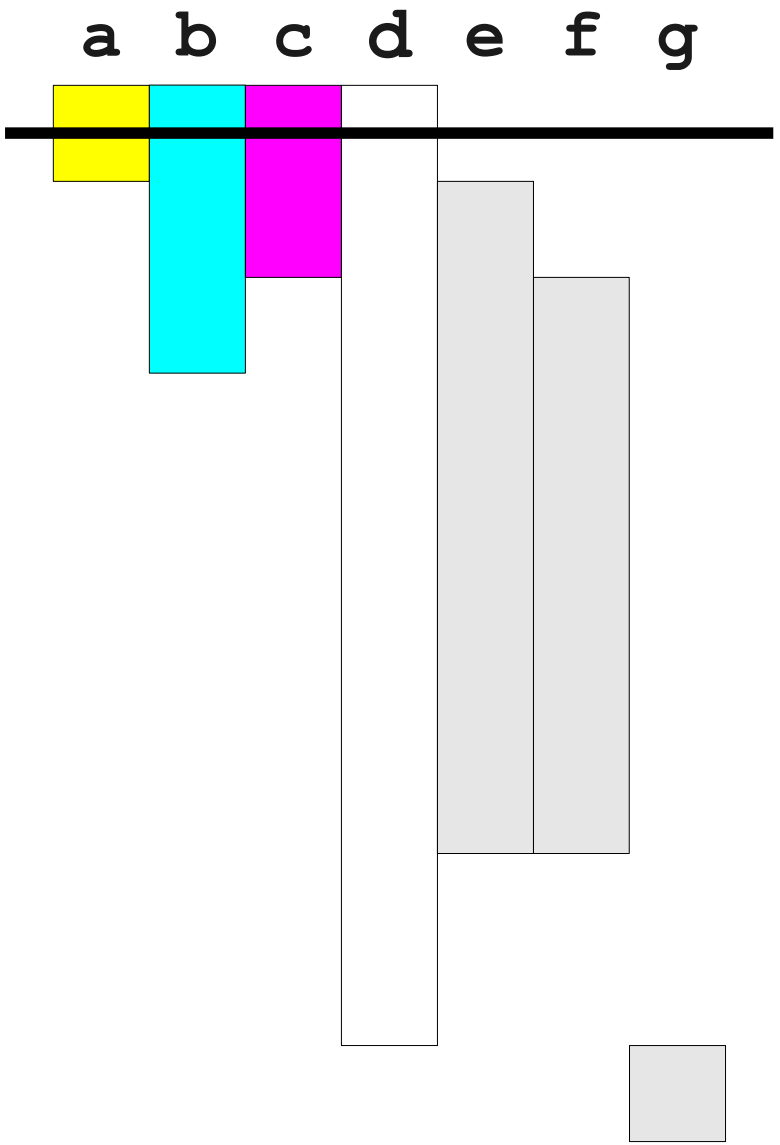


Free Registers

R_0	R_1	R_2

Choose to spill **d** here because it ends the latest in the future.

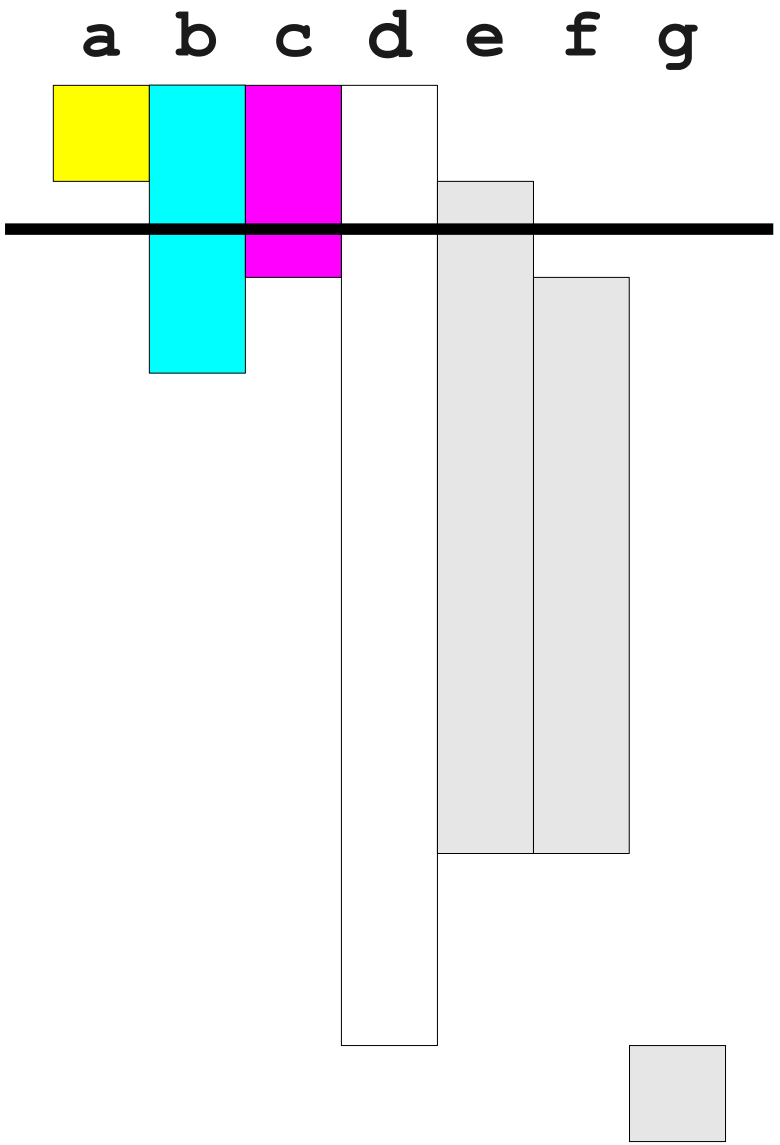
Register Allocation with Live Intervals



Free Registers

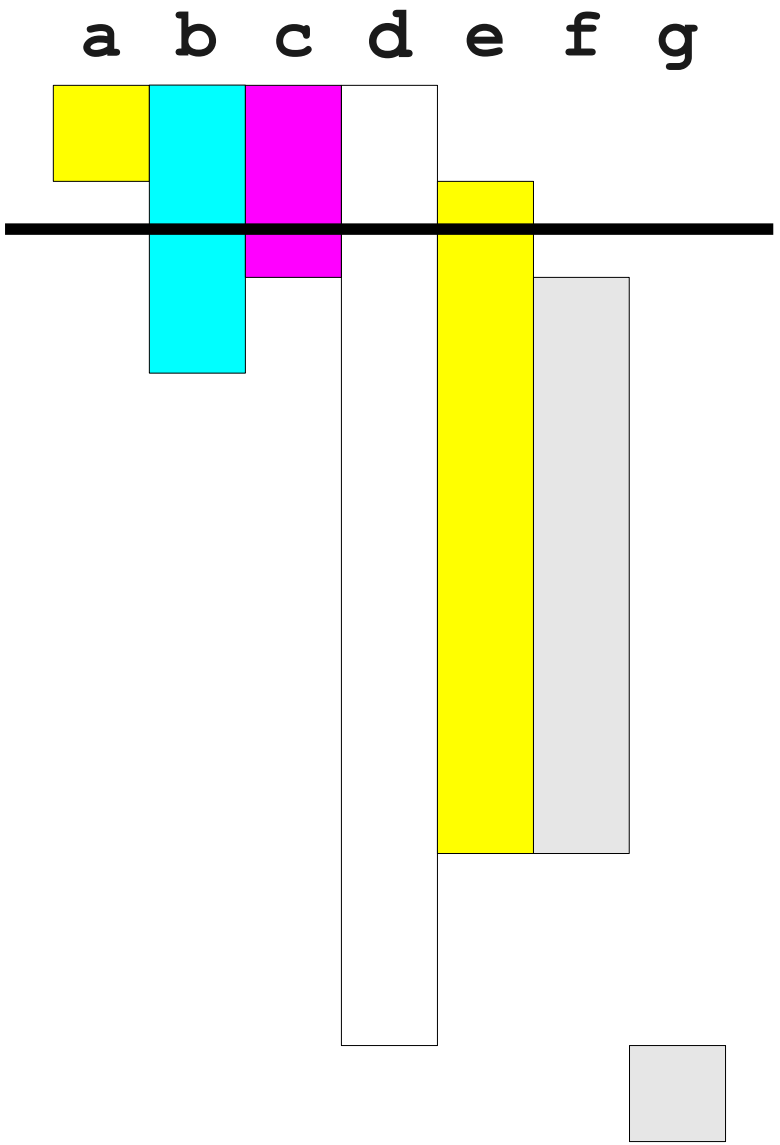
R_0	R_1	R_2

Register Allocation with Live Intervals



Free Registers

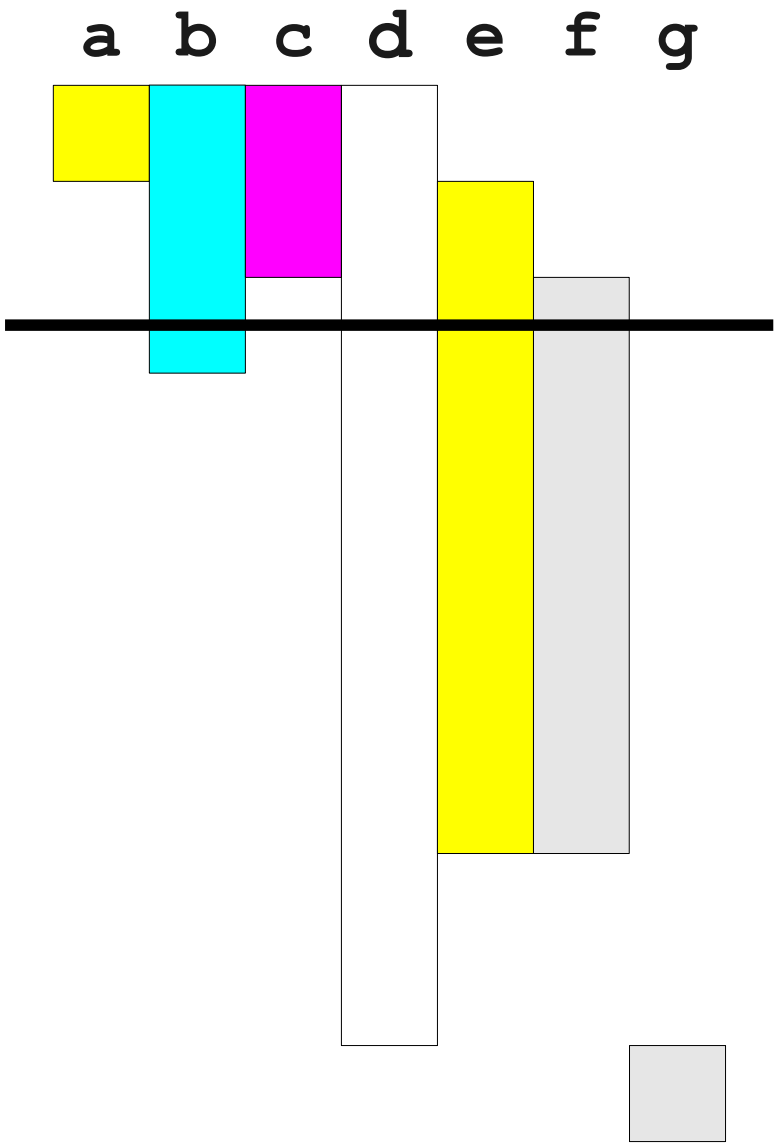
Register Allocation with Live Intervals



Free Registers

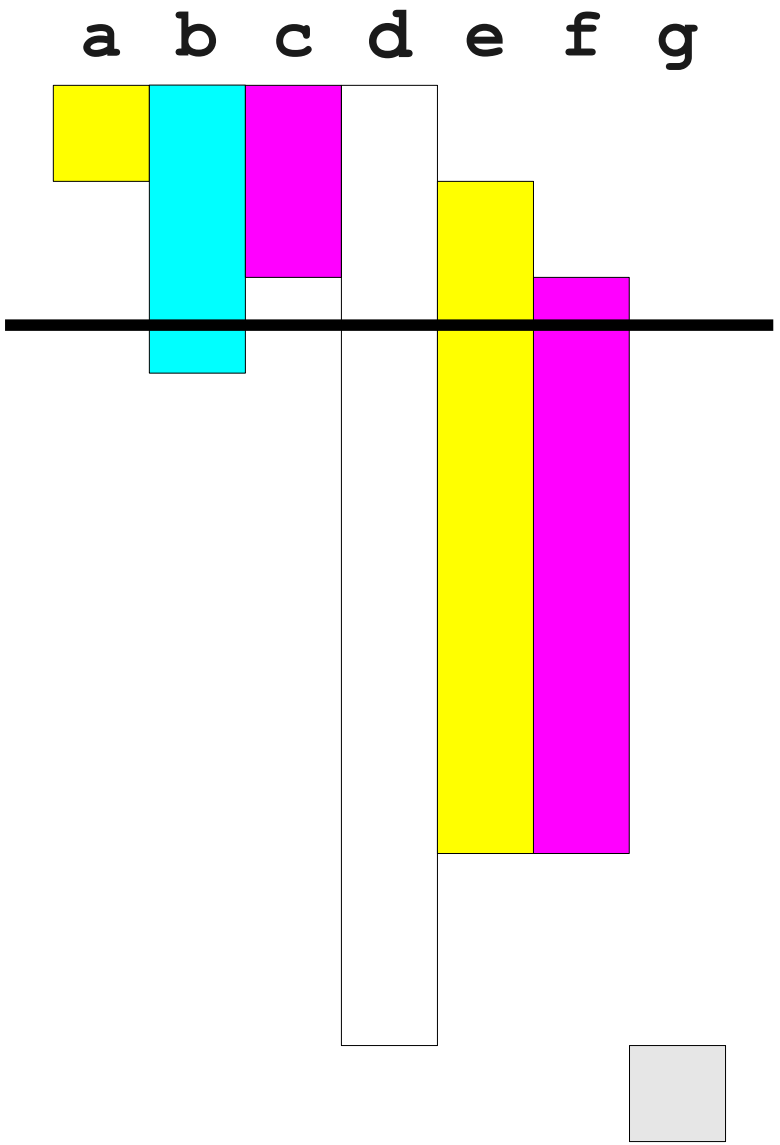
R_0	R_1	R_2

Register Allocation with Live Intervals



Free Registers

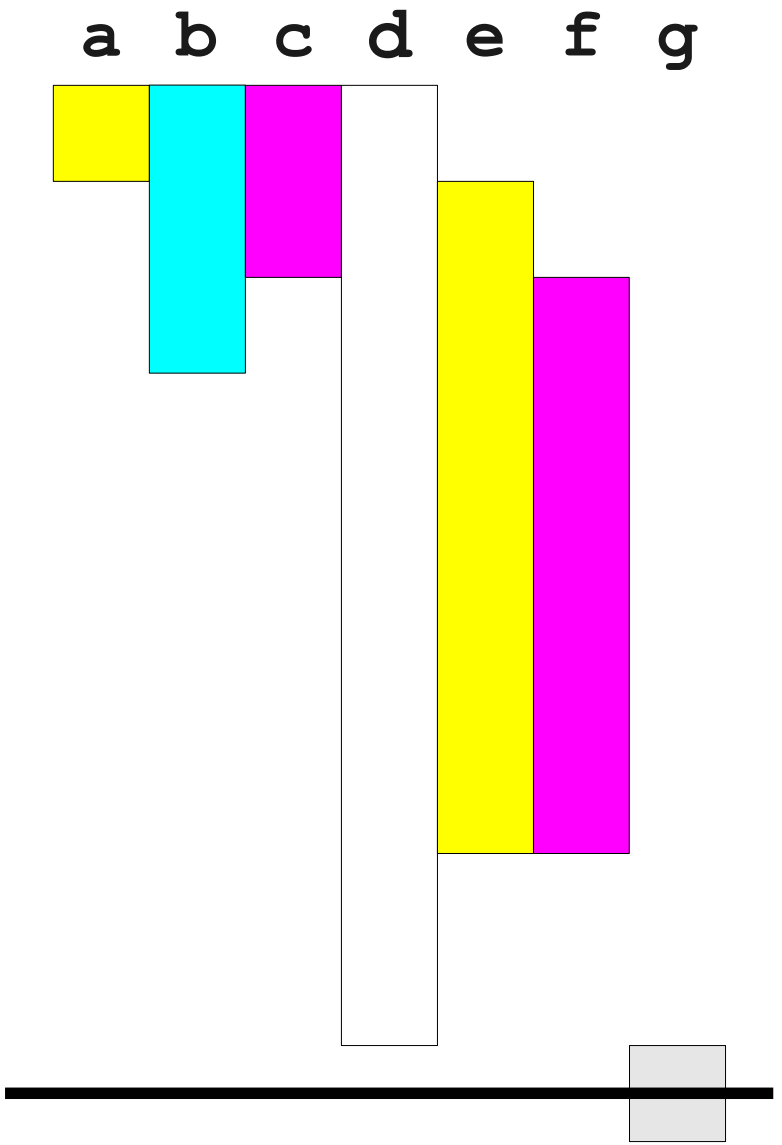
Register Allocation with Live Intervals



Free Registers

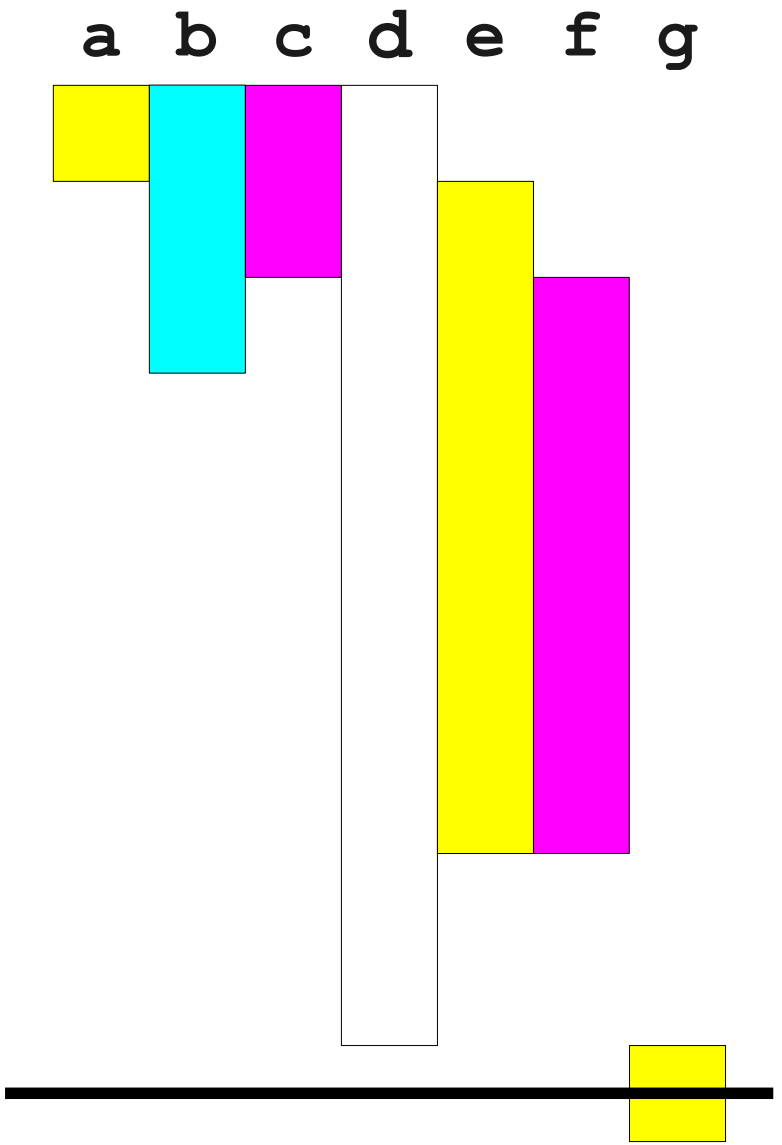
R_0	R_1	R_2

Register Allocation with Live Intervals



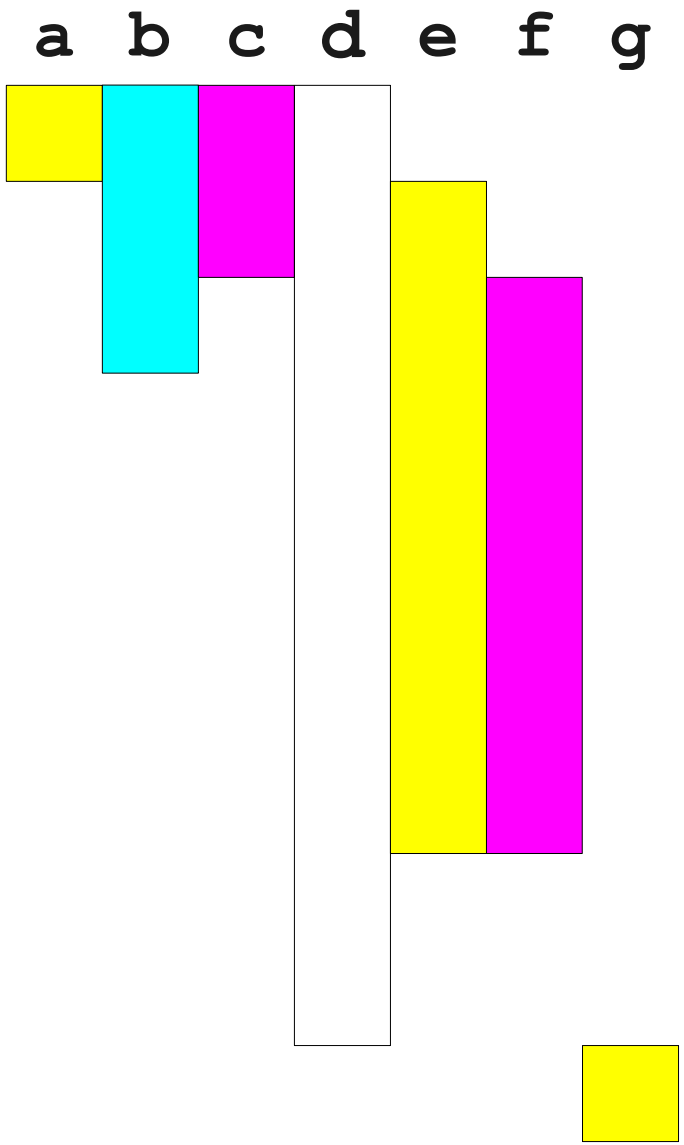
Free Registers

Register Allocation with Live Intervals



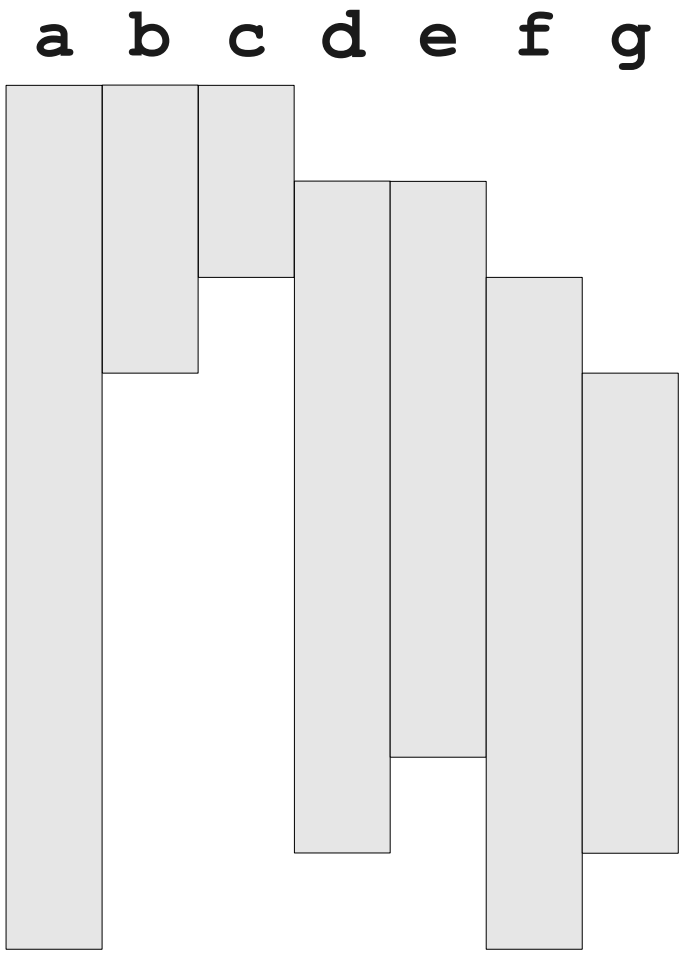
Free Registers

Register Allocation with Live Intervals

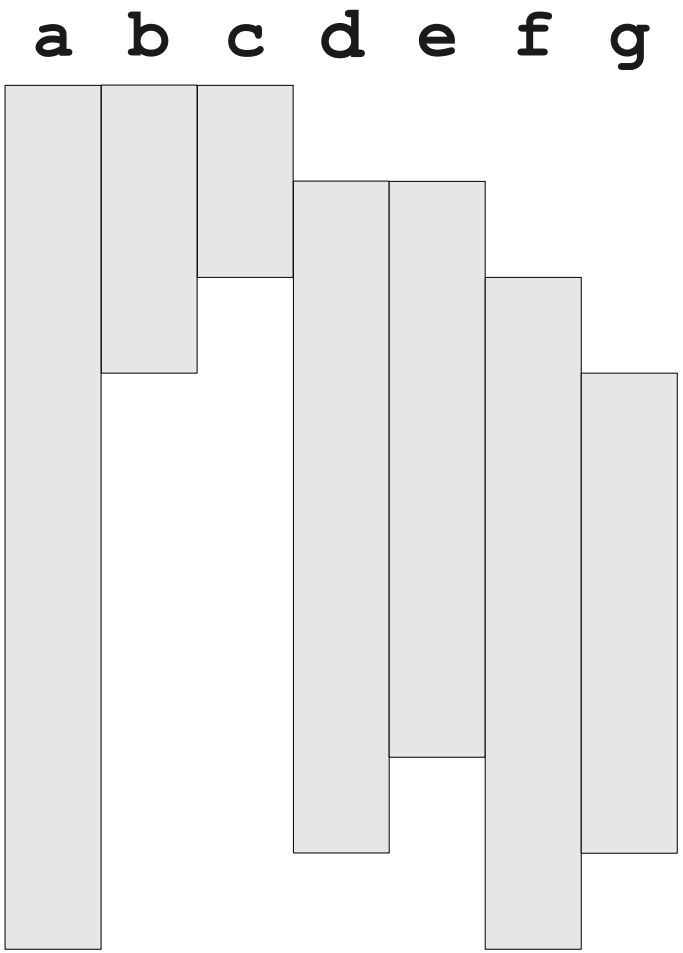


Another Example

Another Example

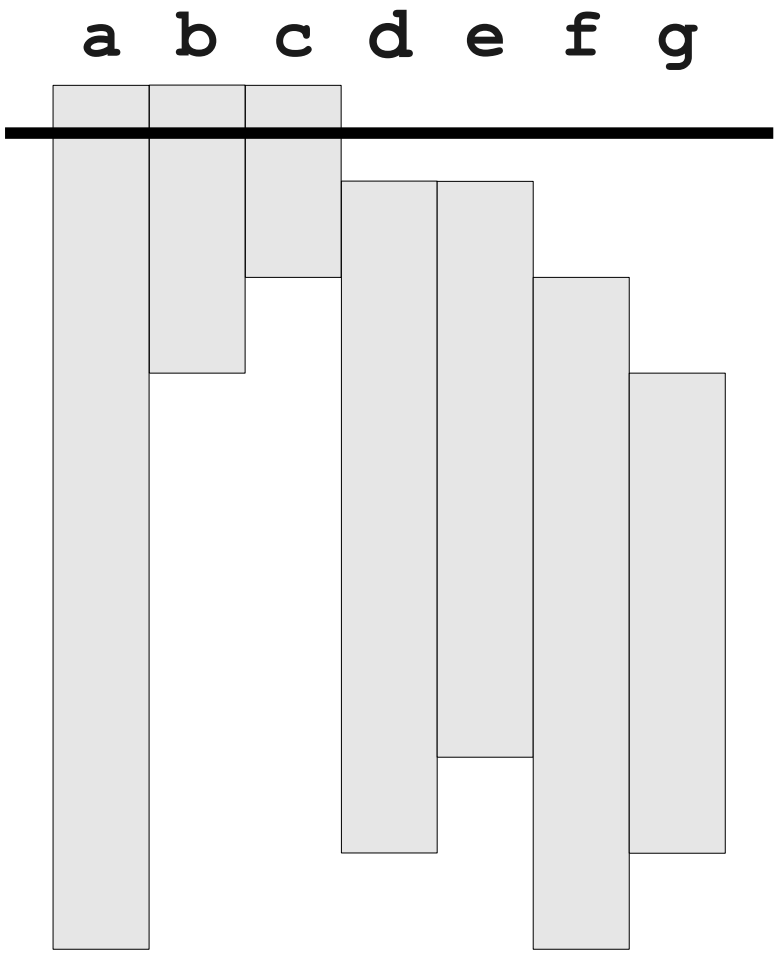


Another Example



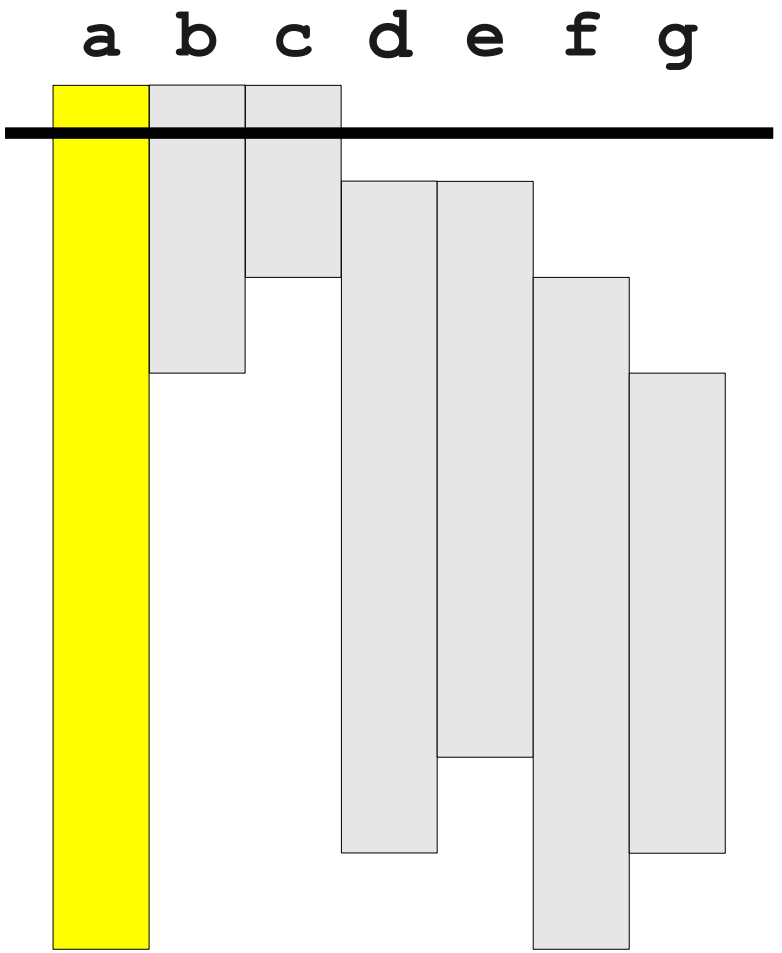
Free Registers

Another Example



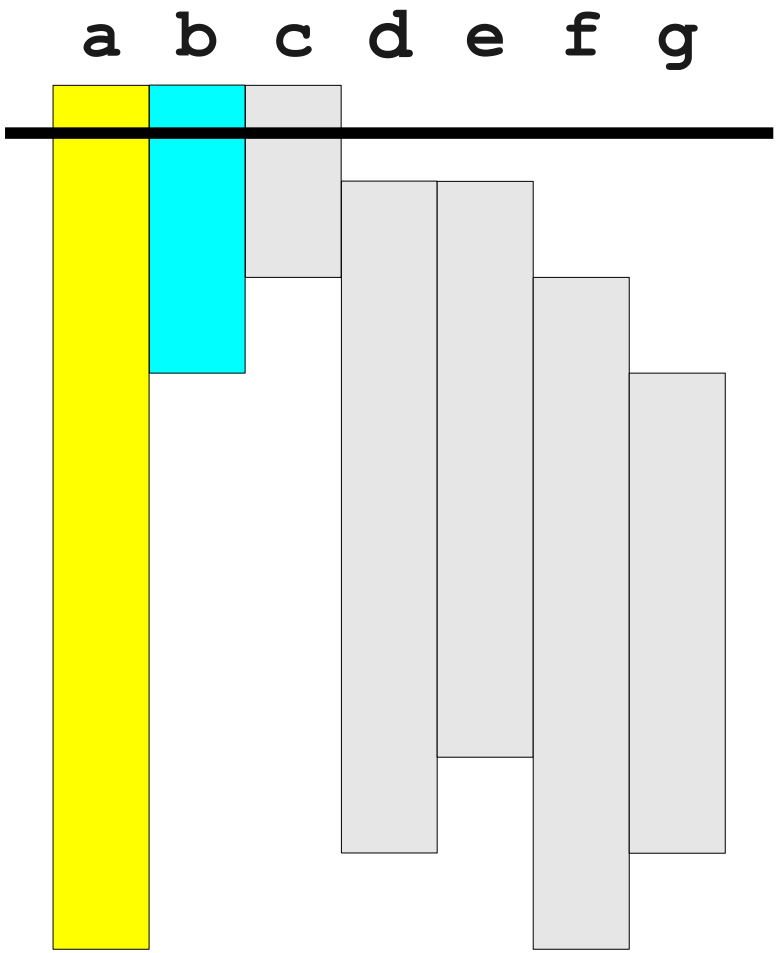
Free Registers

Another Example



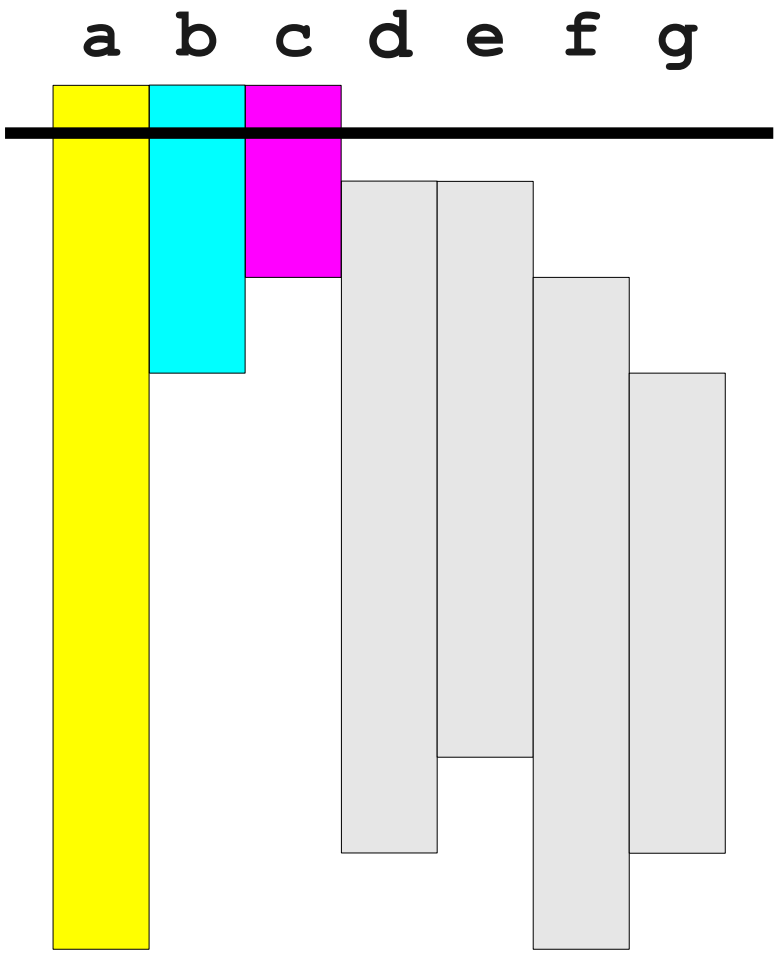
Free Registers

Another Example



Free Registers

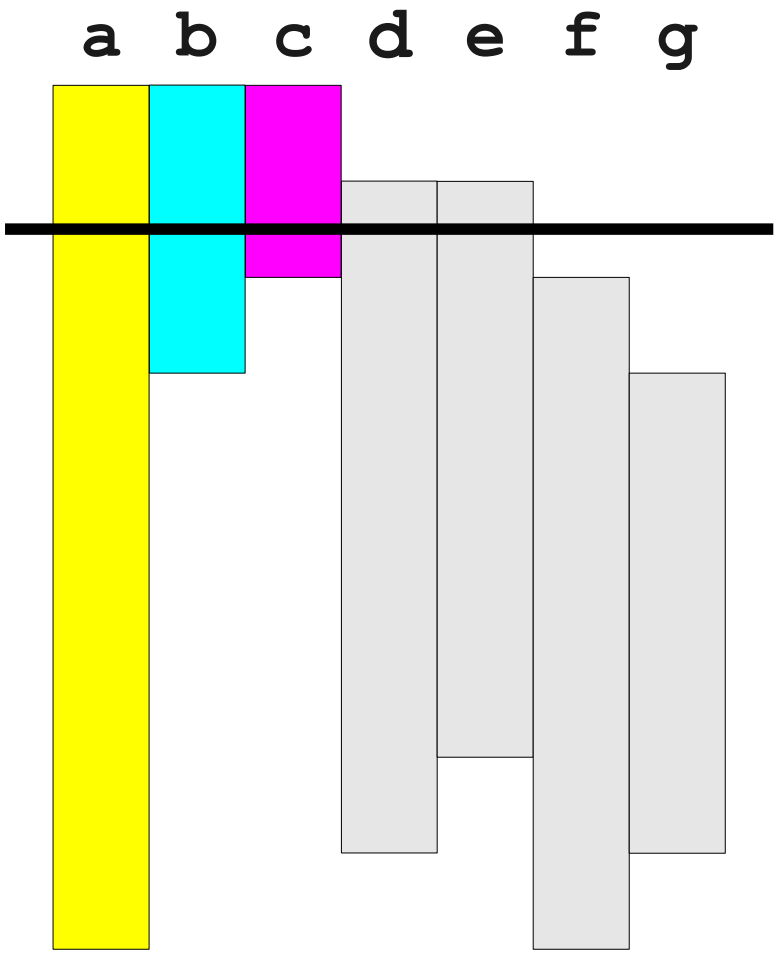
Another Example



Free Registers

R_0	R_1	R_2

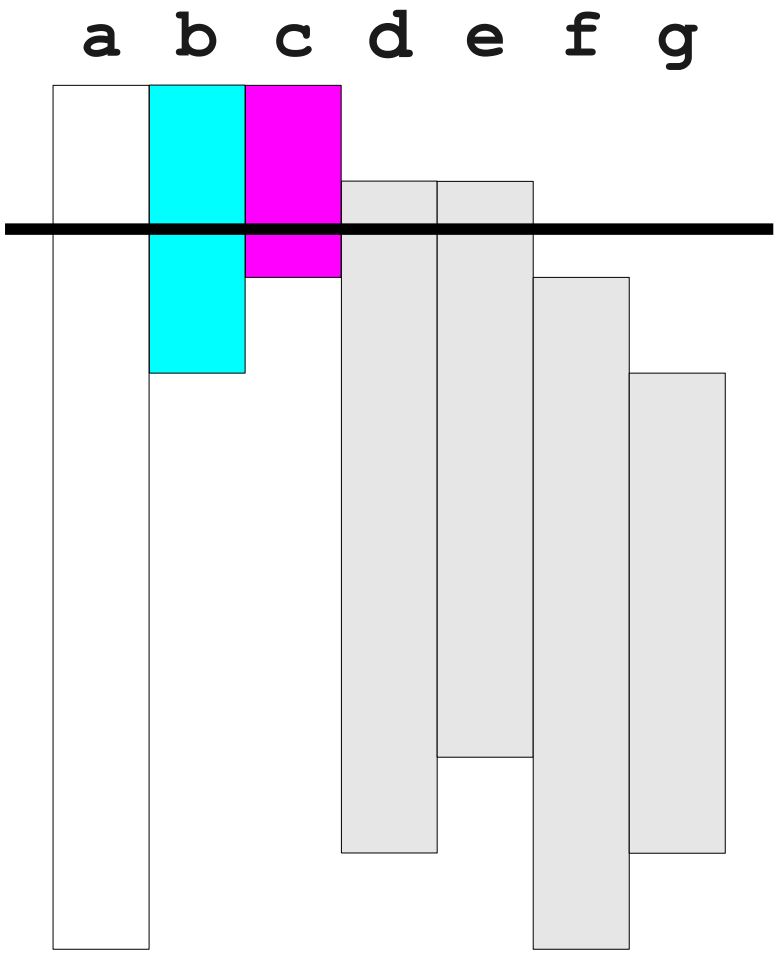
Another Example



Free Registers

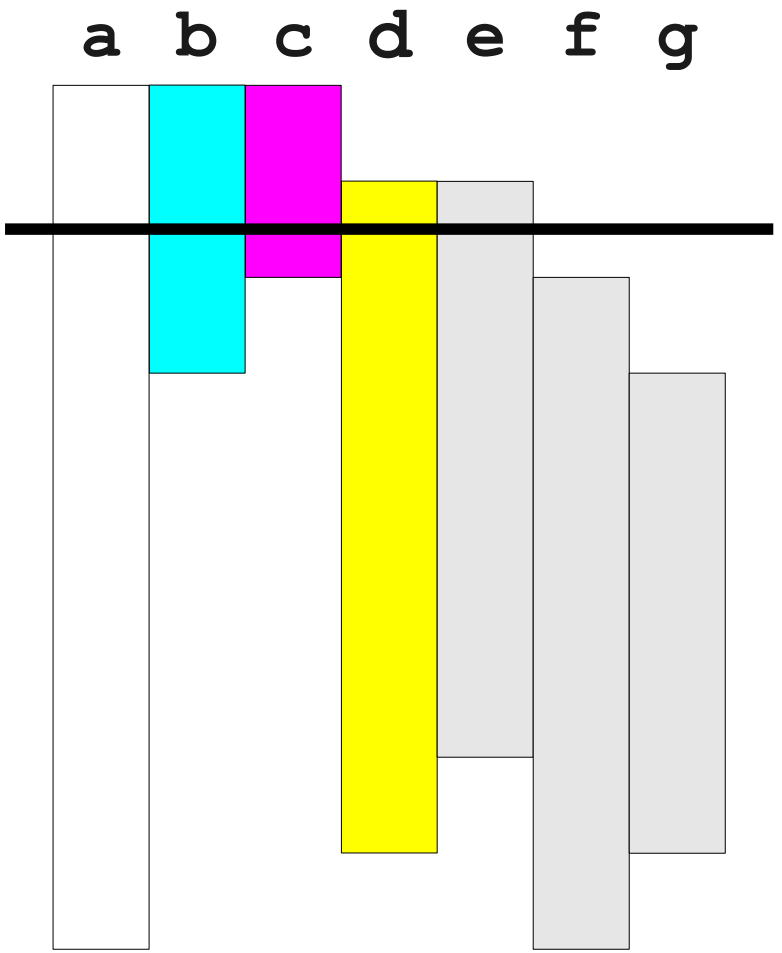
R_0	R_1	R_2

Another Example



Free Registers

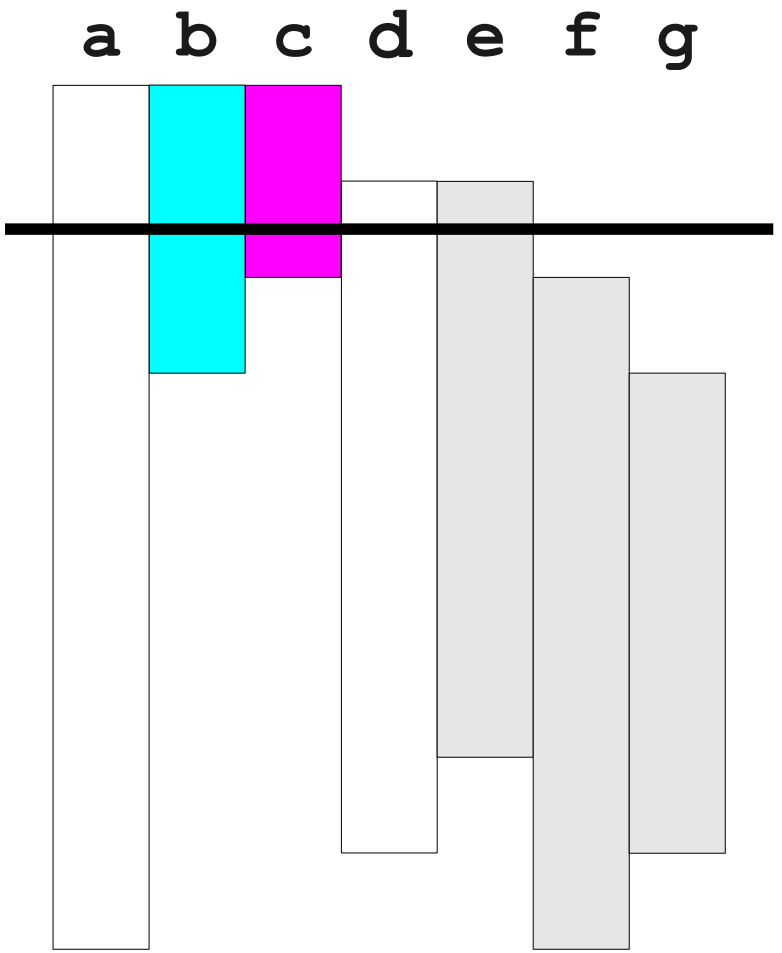
Another Example



Free Registers

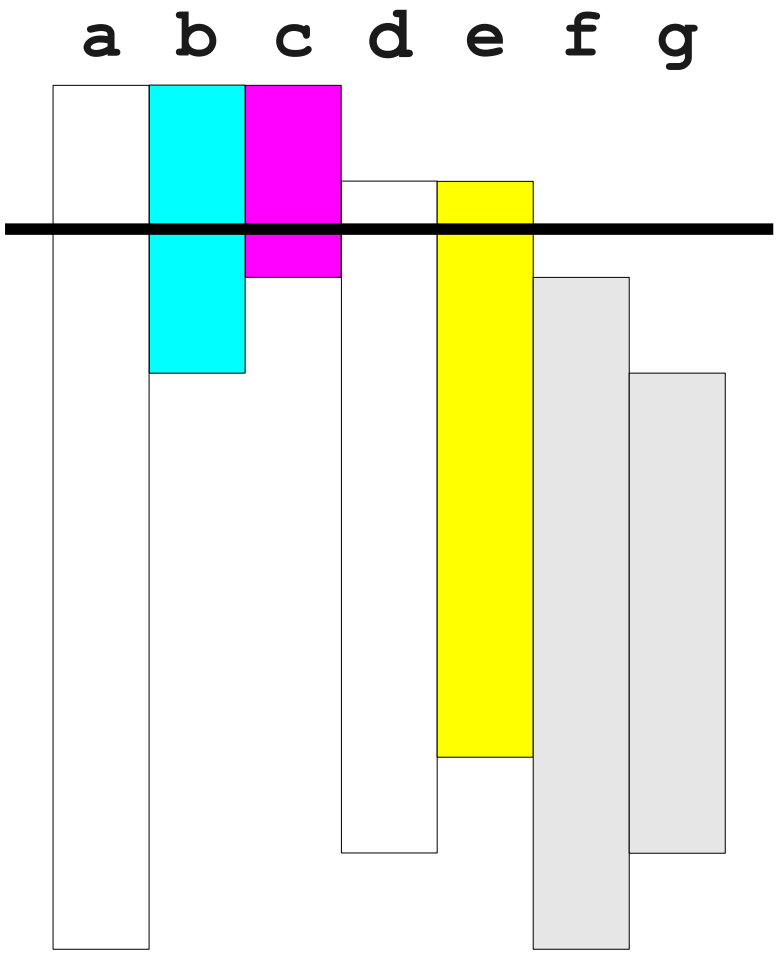
R_0	R_1	R_2

Another Example



Free Registers

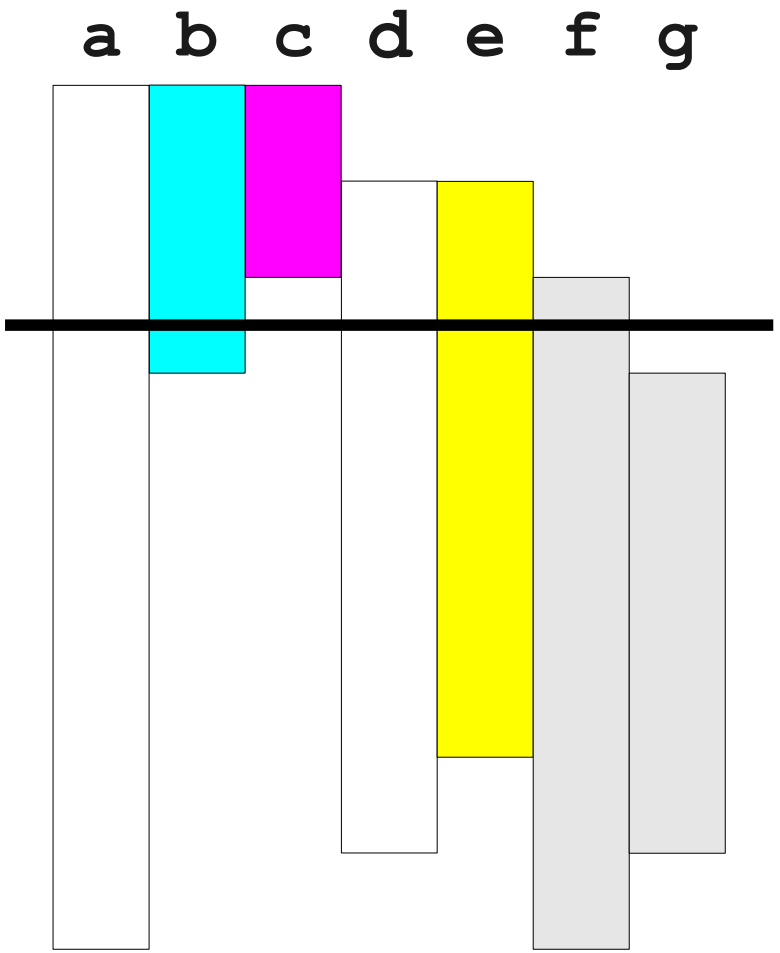
Another Example



Free Registers

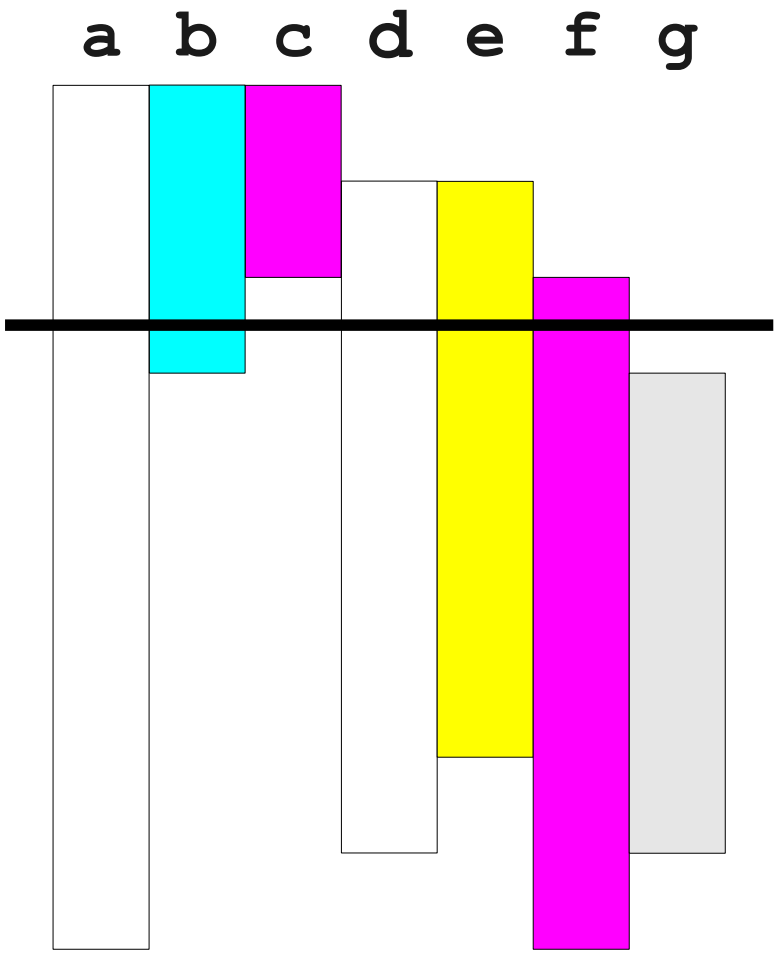
R_0	R_1	R_2

Another Example



Free Registers

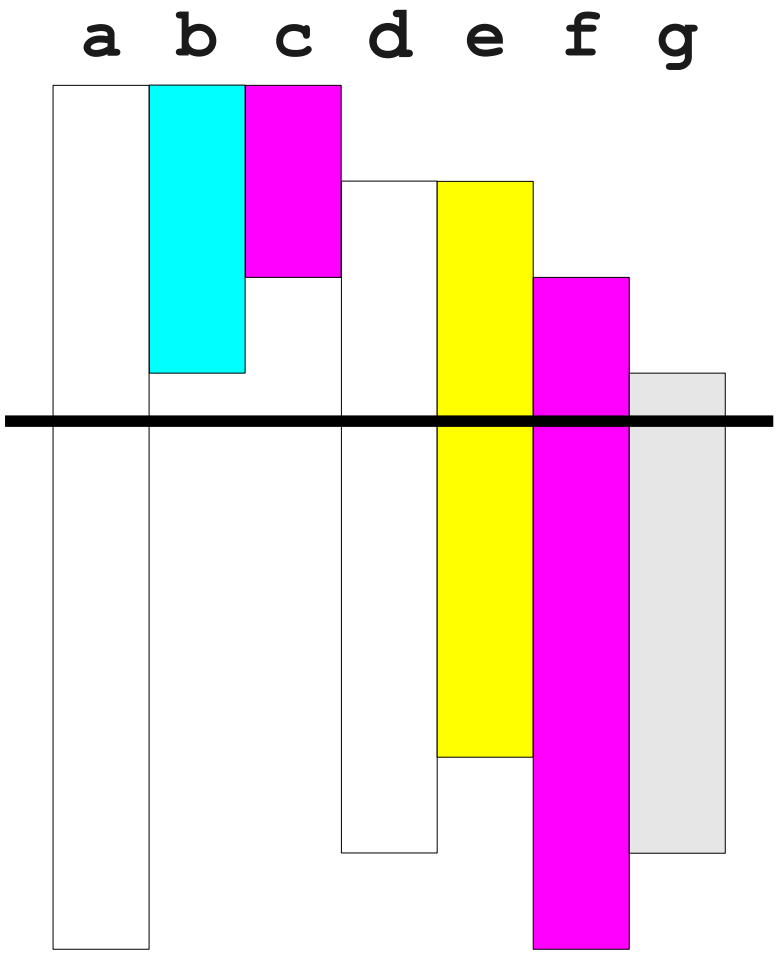
Another Example



Free Registers

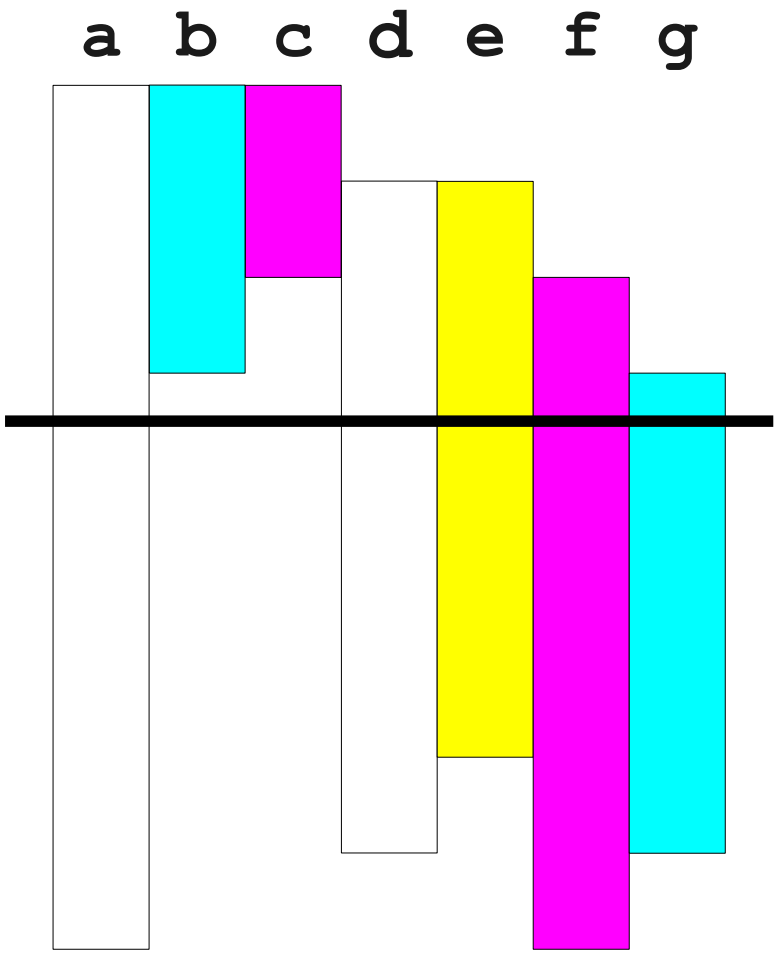
R_0	R_1	R_2

Another Example



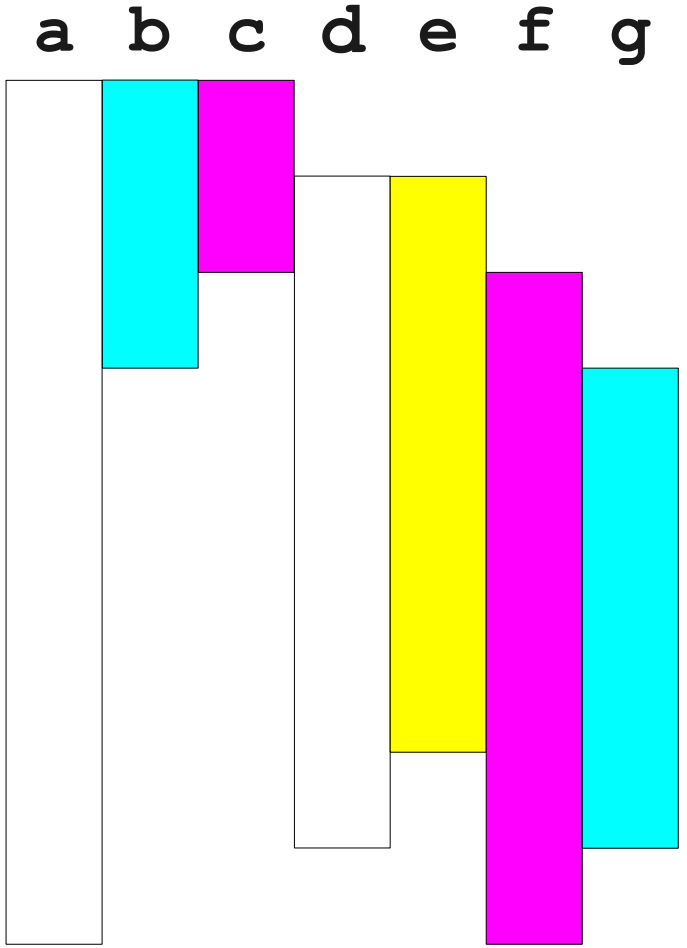
Free Registers

Another Example



Free Registers

Another Example



Linear Scan Register Allocation

- This algorithm is called **linear scan register allocation** and is a comparatively new algorithm.
- Advantages:
  - Very efficient (after computing live intervals, runs in linear time)
  - Produces good code in many instances.
  - Allocation step works in one pass; can generate code during iteration.
  - Often used in JIT compilers like Java HotSpot.
- Disadvantages:
  - Imprecise due to use of live **intervals** rather than live **ranges**.
  - Other techniques known to be superior in many cases.

Correctness Proof Sketch

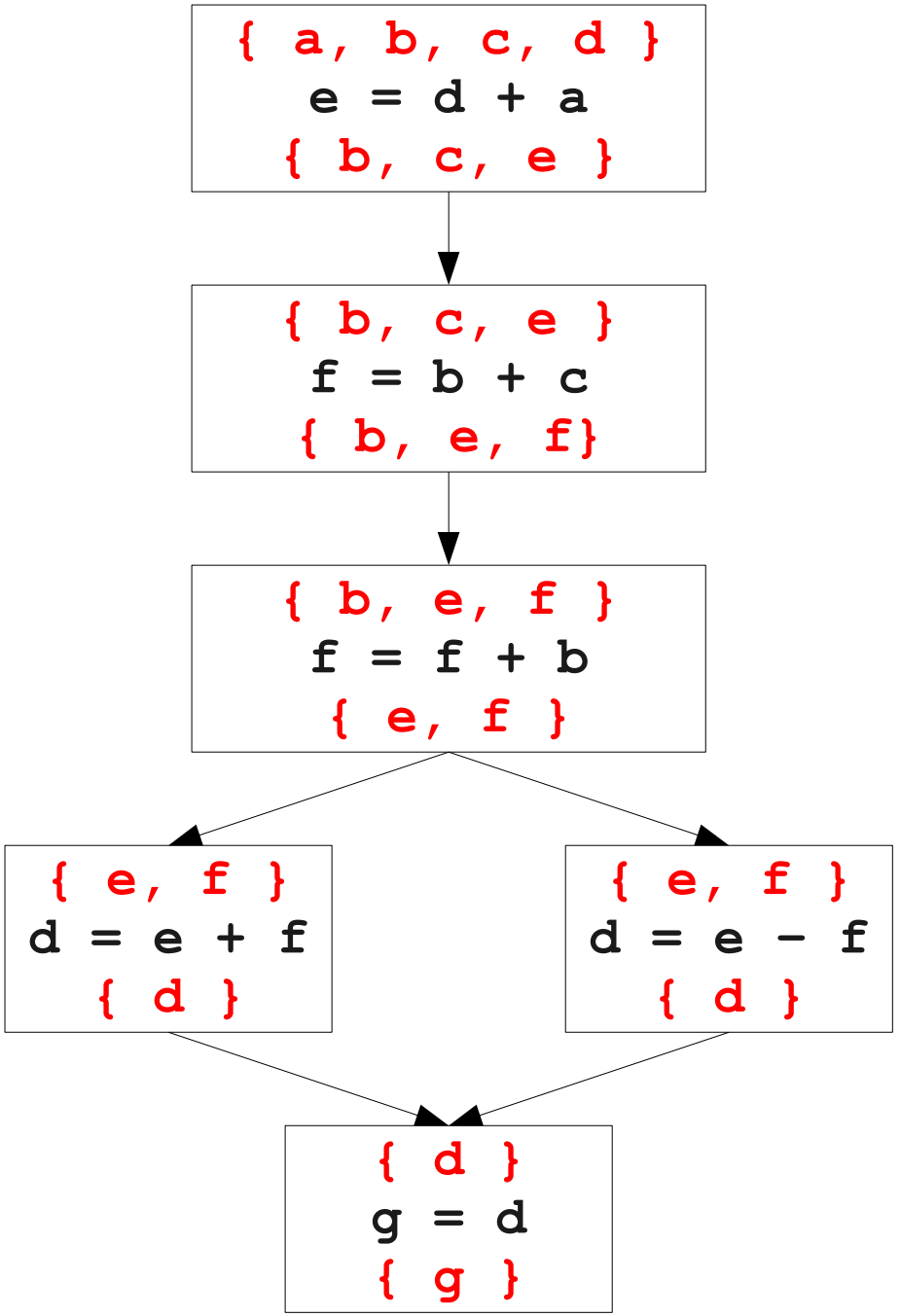
- No register holds two live variables at once:
  - Live intervals are conservative approximations of live ranges.
  - No two variables with overlapping live ranges placed in the same register.
- At each program point, every variable is in the same location:
  - All variables assigned a unique location.

Second-Chance Bin Packing

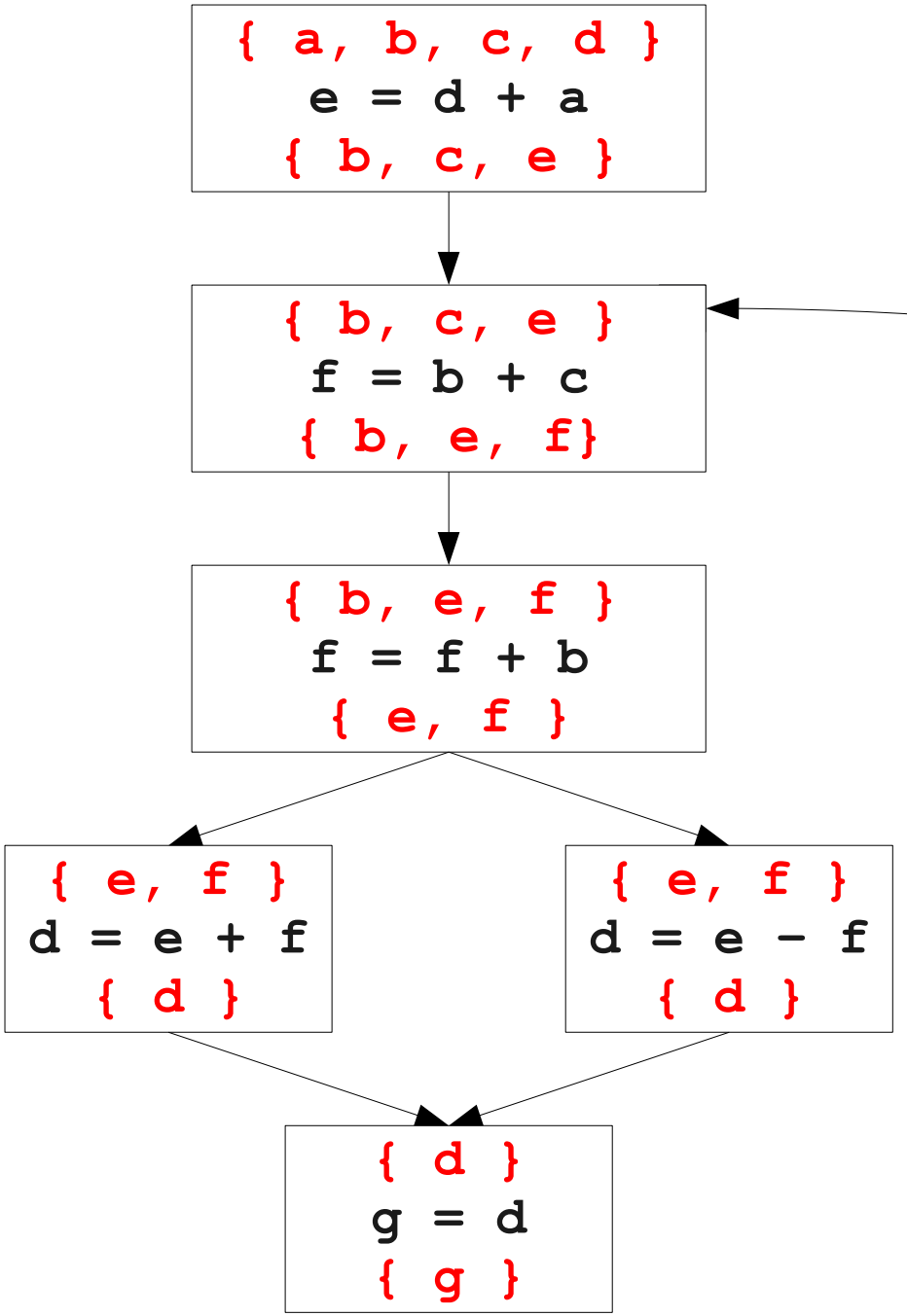
- A more aggressive version of linear-scan.
- Uses live **ranges** instead of live **intervals**.
- If a variable must be spilled, don't spill all uses of it.
  - A later live range might still fit into a register.
- Requires a final data-flow analysis to confirm variables are assigned consistent locations.
- See “Quality and Speed in Linear-scan Register Allocation” by Traub, Holloway, and Smith.

An Entirely Different Approach

An Entirely Different Approach

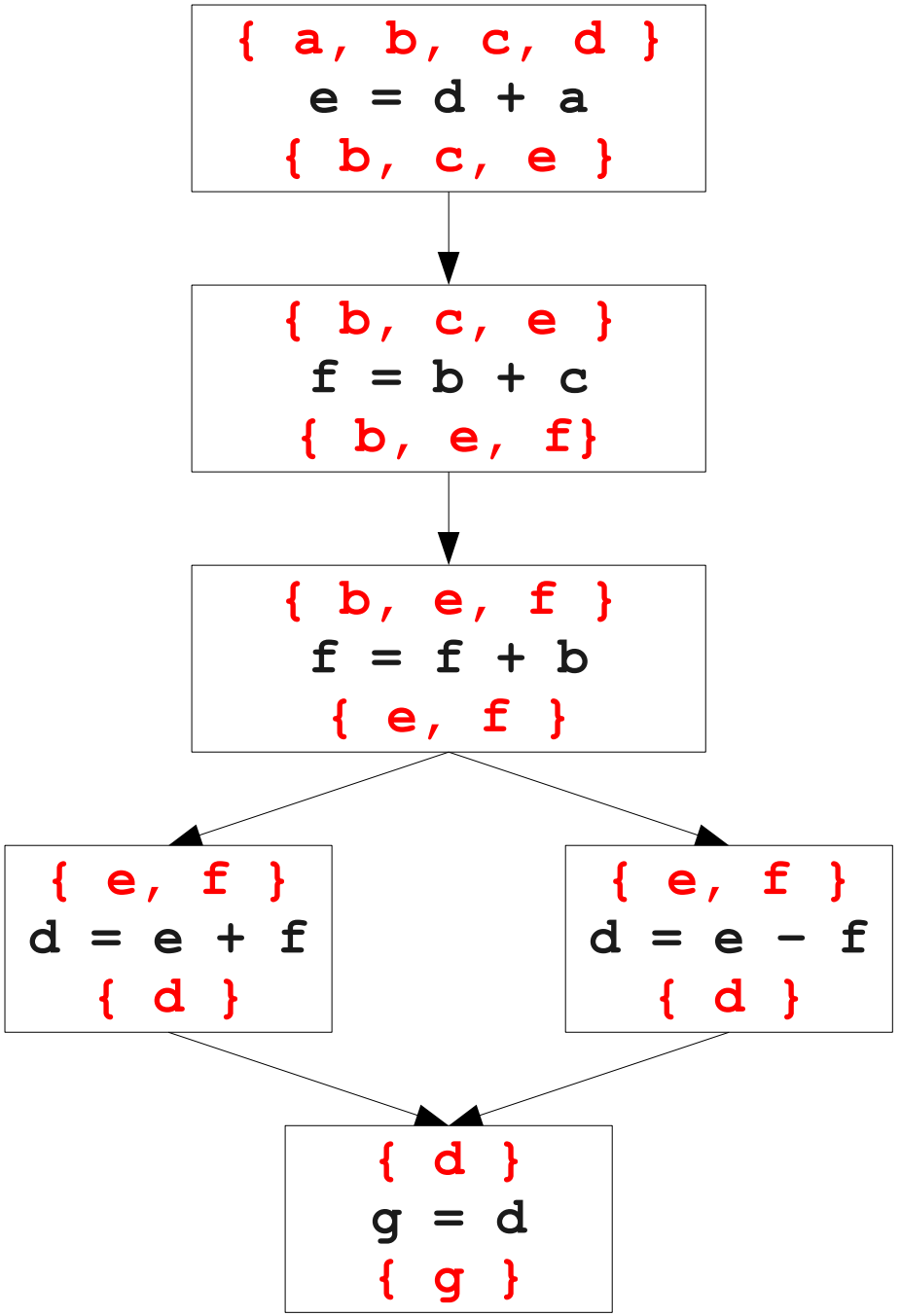


An Entirely Different Approach

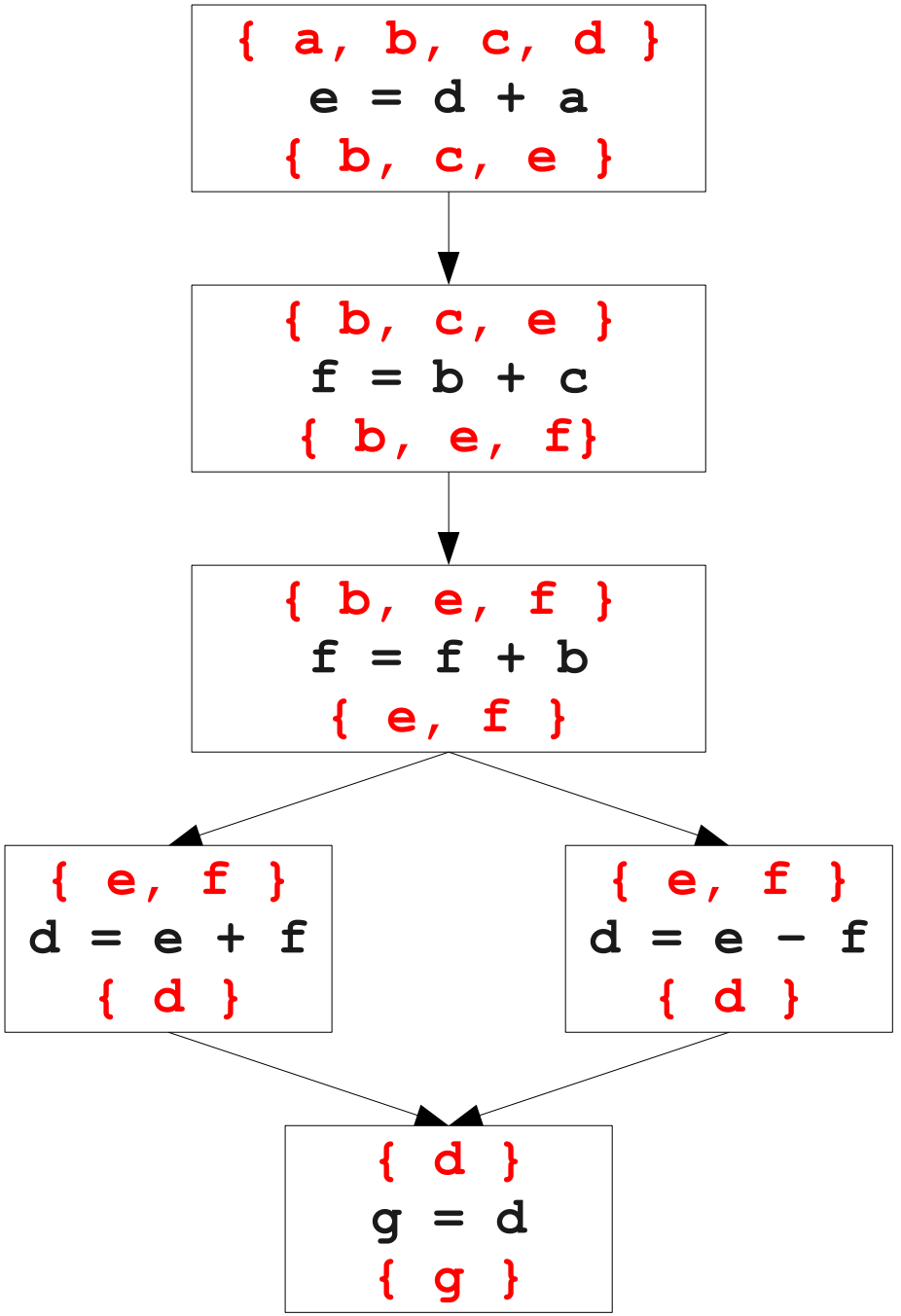


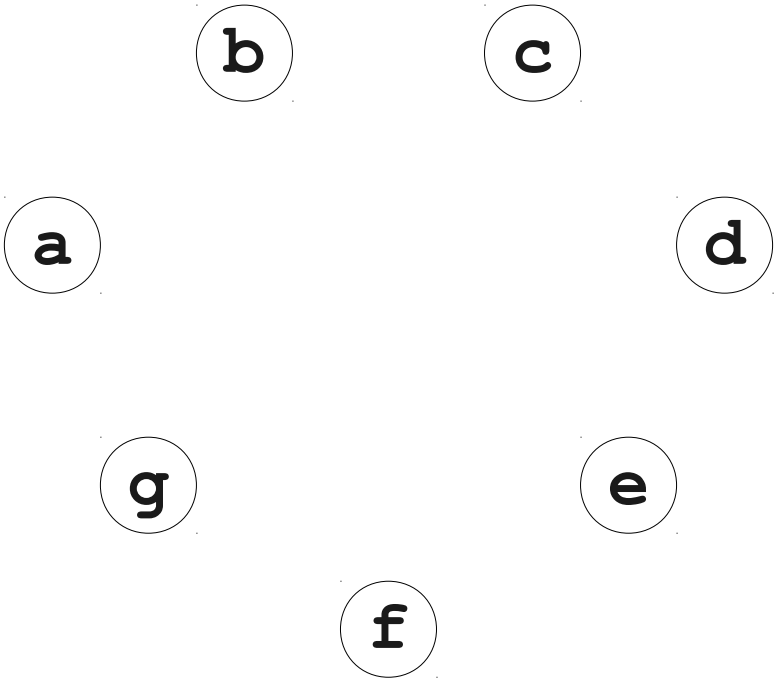
What can we infer from all these variables being live at this point?

An Entirely Different Approach

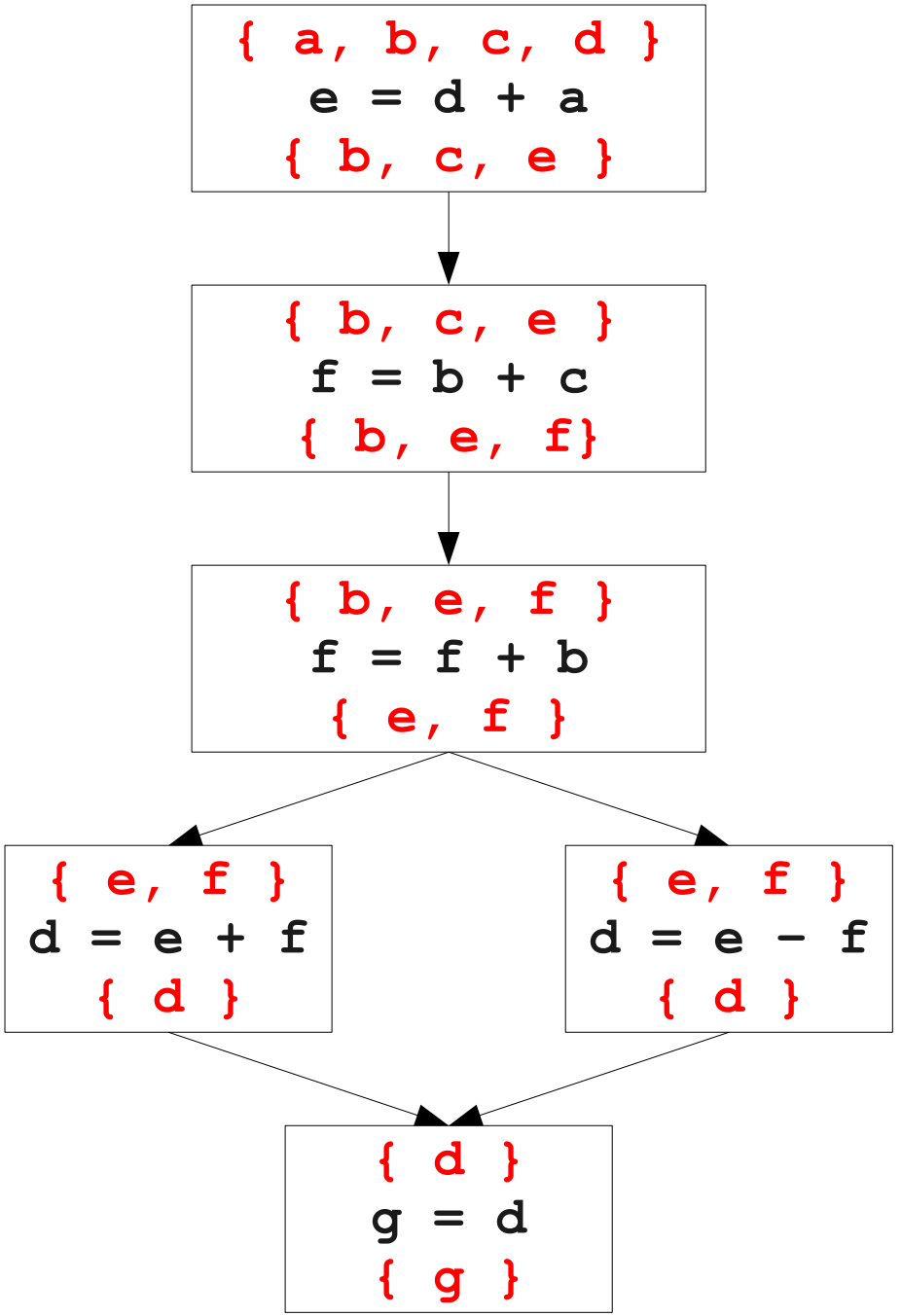


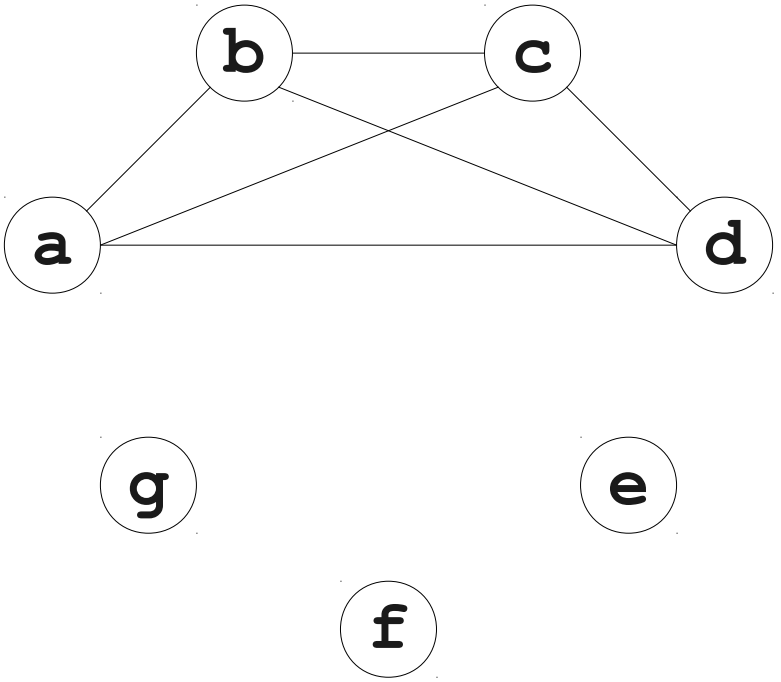
An Entirely Different Approach



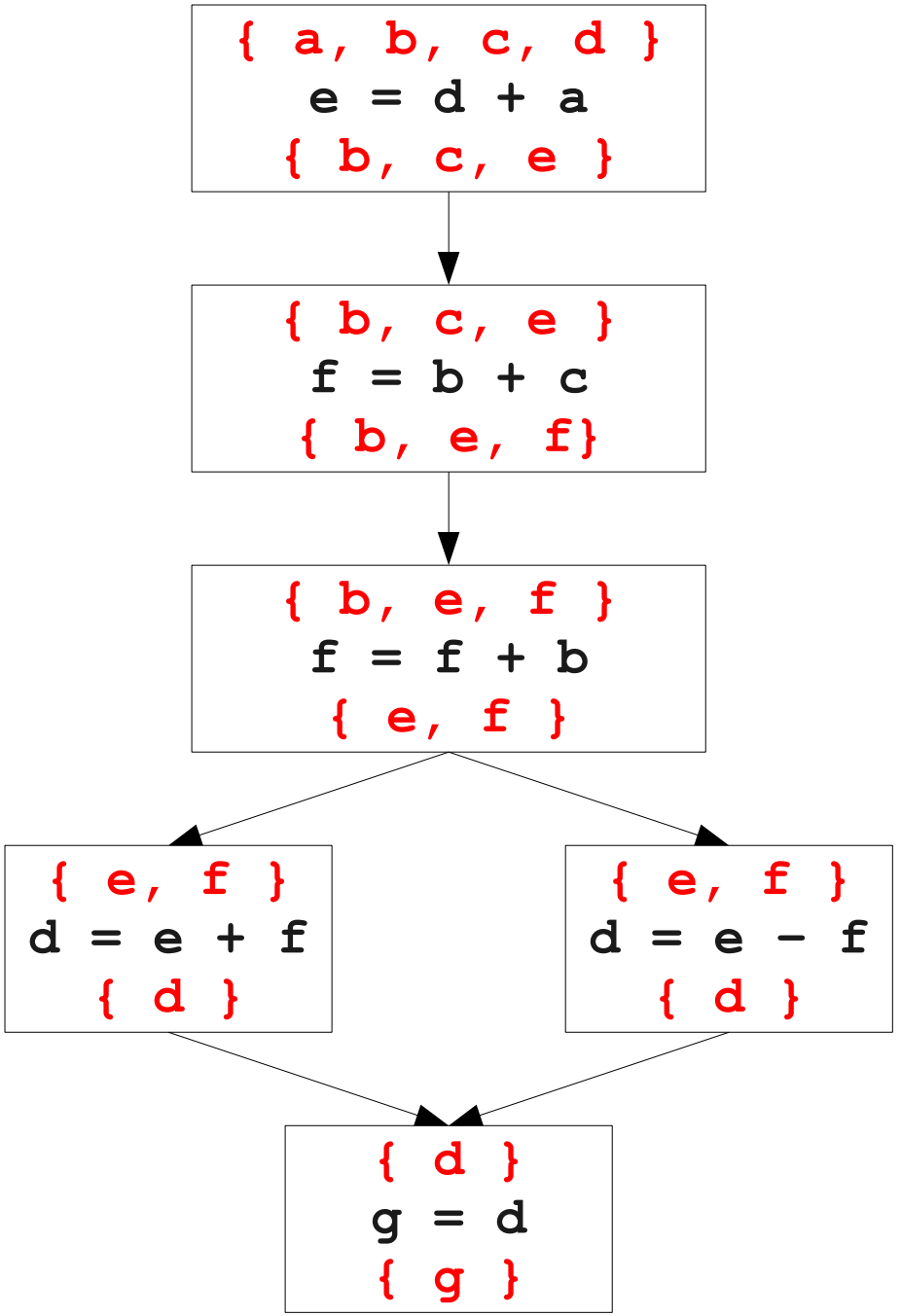


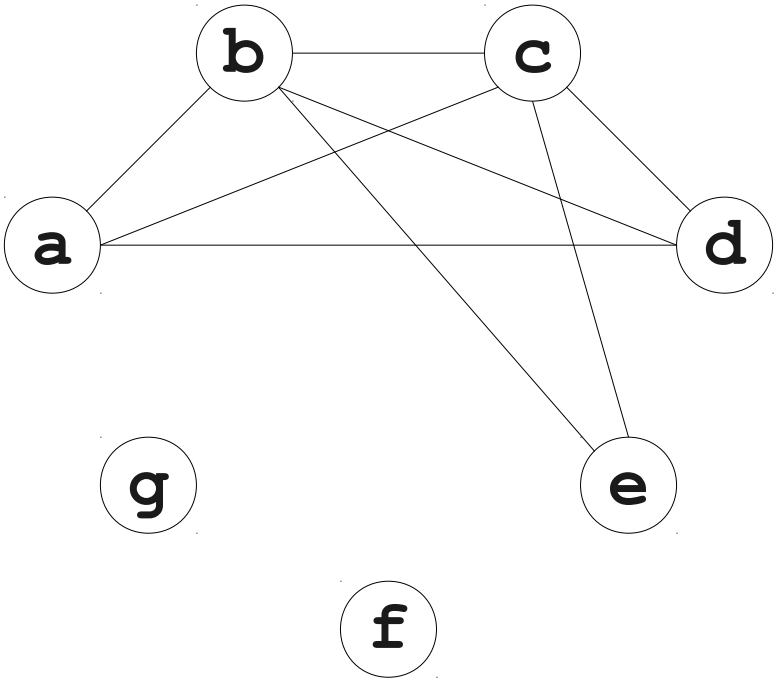
An Entirely Different Approach



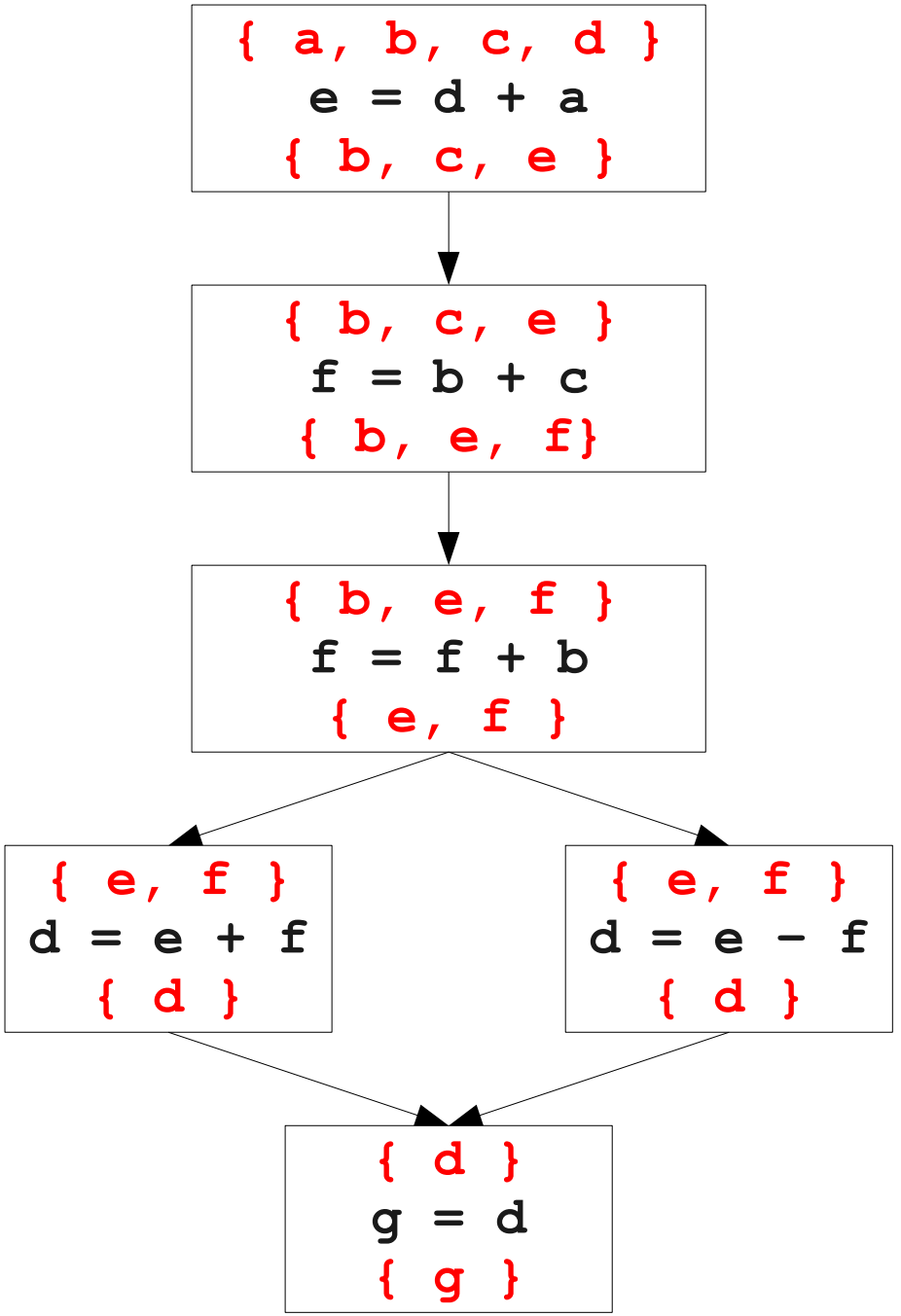


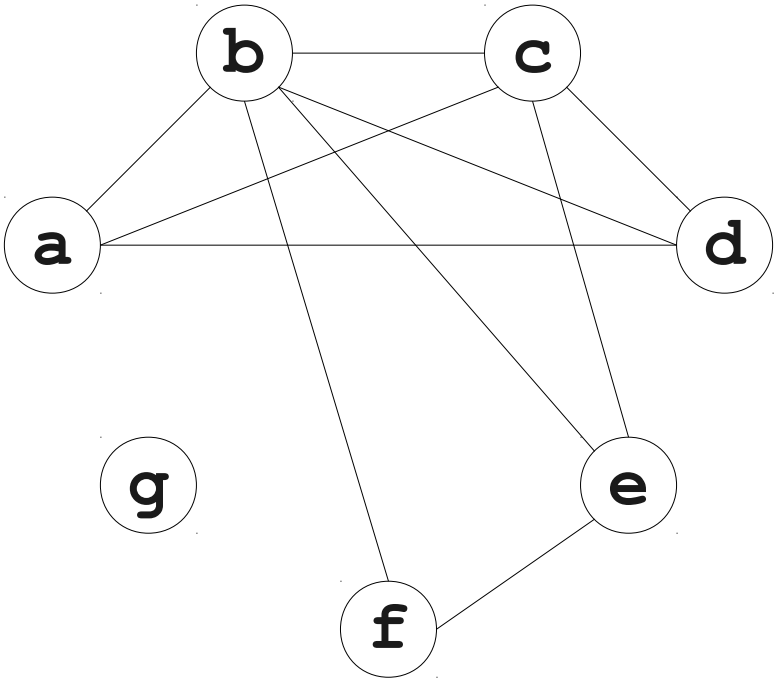
An Entirely Different Approach



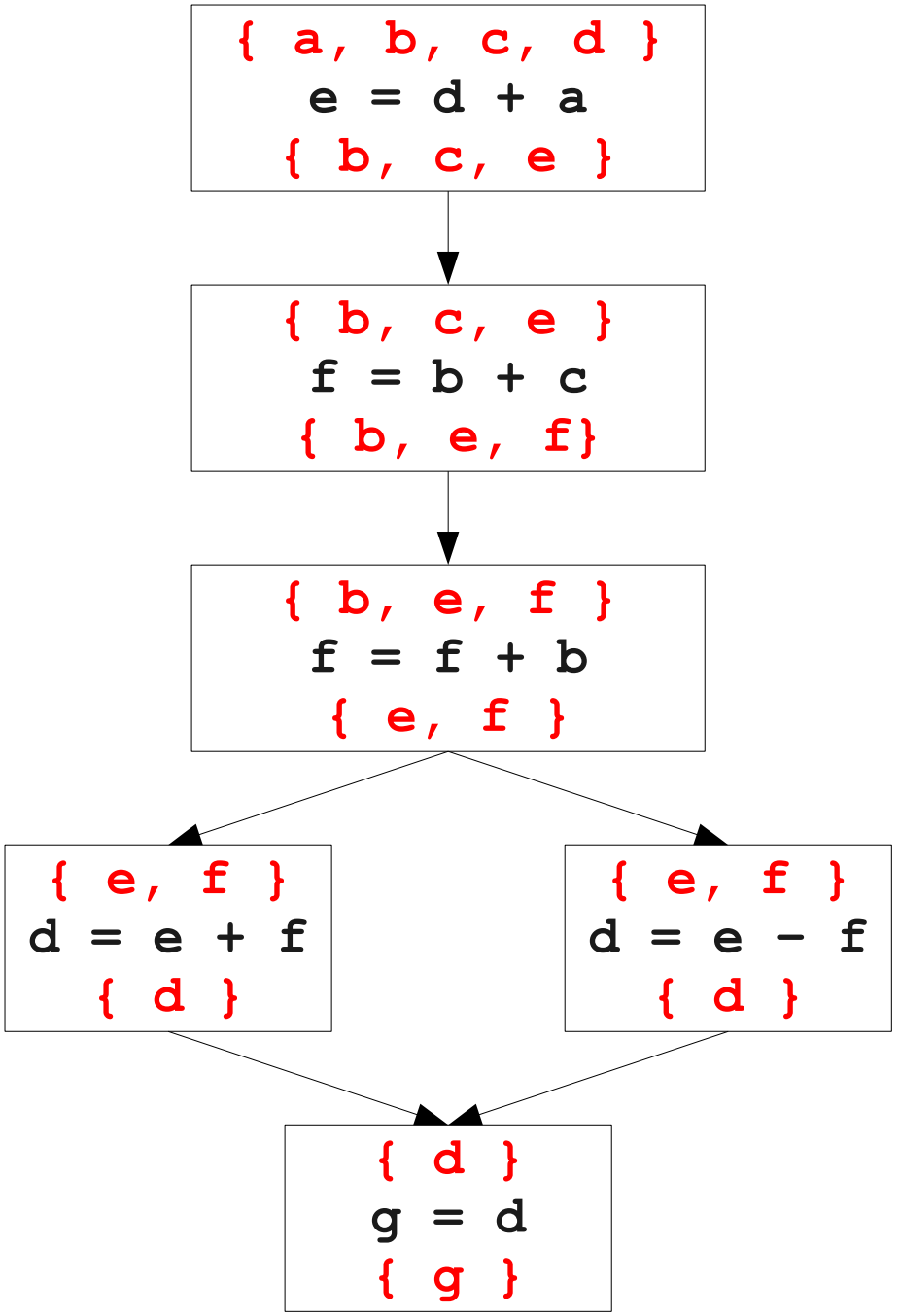


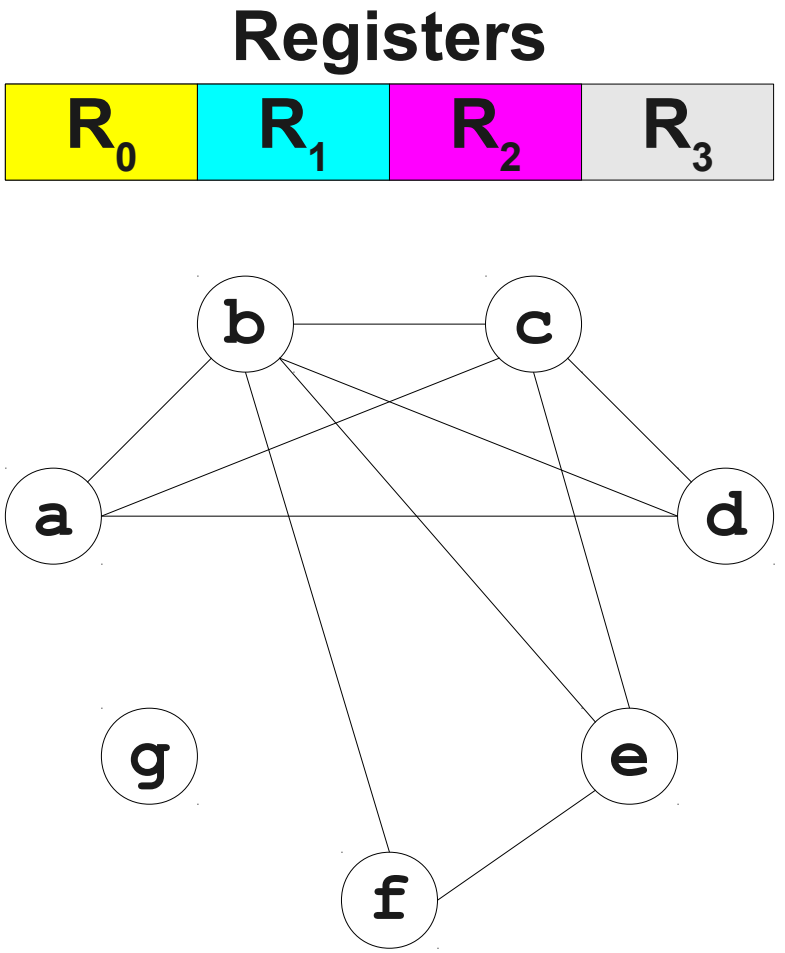
An Entirely Different Approach



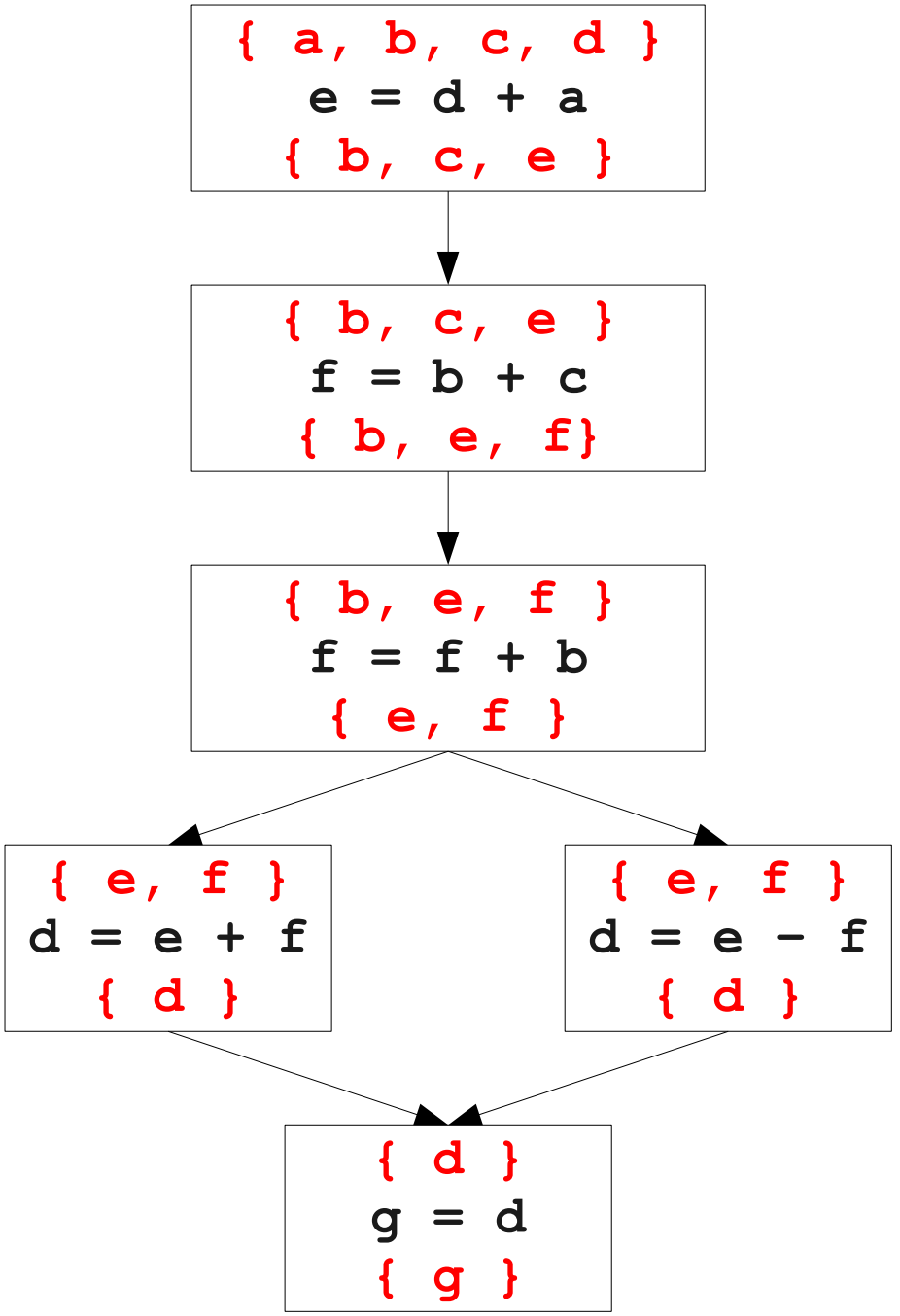


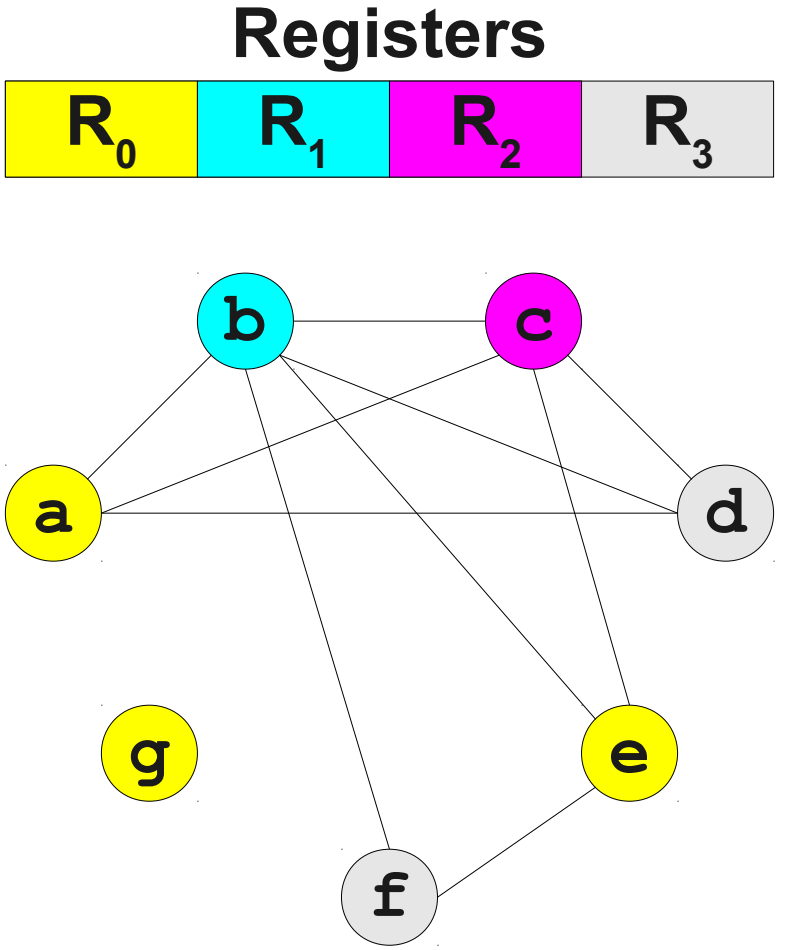
An Entirely Different Approach





An Entirely Different Approach





The Register Interference Graph

- The **register interference graph** (RIG) of a control-flow graph is an undirected graph where
  - Each node is a variable.
  - There is an edge between two variables that are live at the same program point.
- Perform register allocation by assigning each variable a different register from all of its neighbors.
- There's just one catch...

The One Catch

- This problem is equivalent to **graph-coloring**, which is **NP-hard** if there are at least three registers.
- No good polynomial-time algorithms (or even good approximations!) are known for this problem.
- We have to be content with a heuristic that is good enough for RIGs that arise in practice.

The One Catch to The One Catch

The One Catch to The One Catch

If you can figure out a way to assign registers to arbitrary RIGs, you've just proven P = NP and will get a **$1,000,000 check** from the Clay Mathematics Institute.

The One Catch to The One Catch

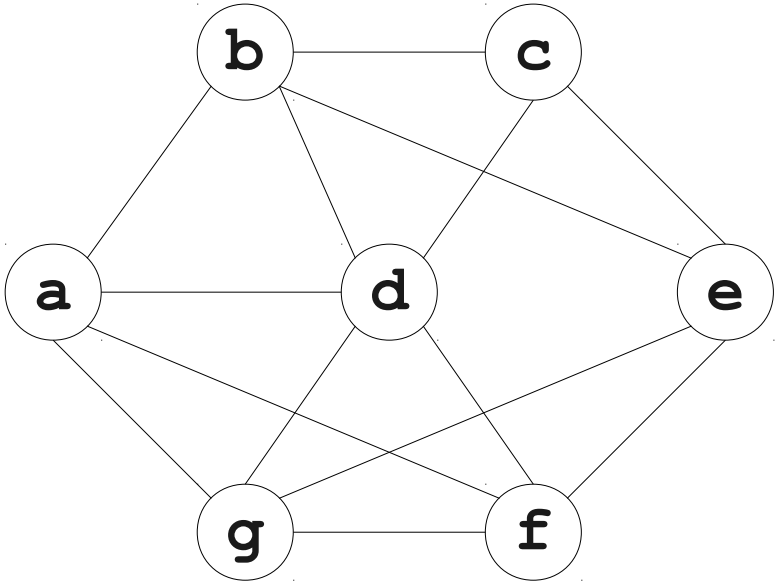
If you can figure out a way to assign registers to arbitrary RIGs, you've just proven P = NP and will get a **$1,000,000 check** from the Clay Mathematics Institute.

Chaitin's Algorithm

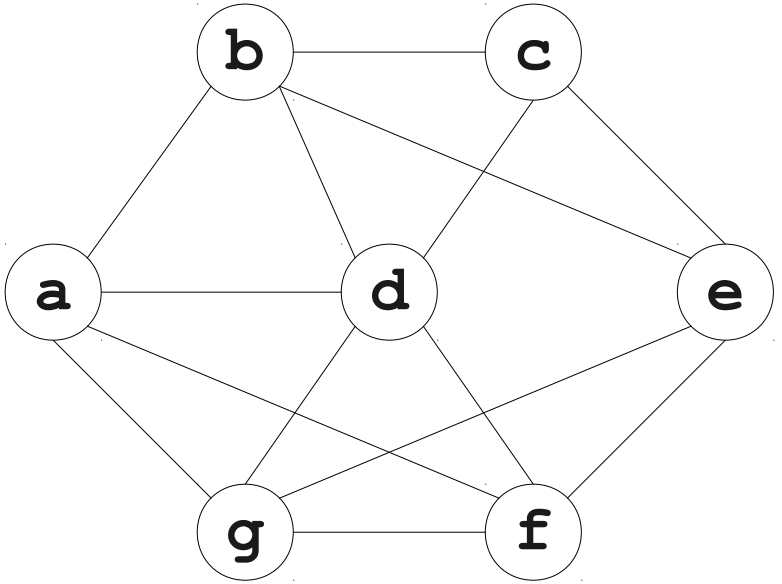
- Intuition:
  - Suppose we are trying to k-color a graph and find a node with fewer than k edges.
  - If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in.
  - Reason: With fewer than k neighbors, some color must be left over.
- Algorithm:
  - Find a node with fewer than k outgoing edges.
  - Remove it from the graph.
  - Recursively color the rest of the graph.
  - Add the node back in.
  - Assign it a valid color.

Chaitin's Algorithm

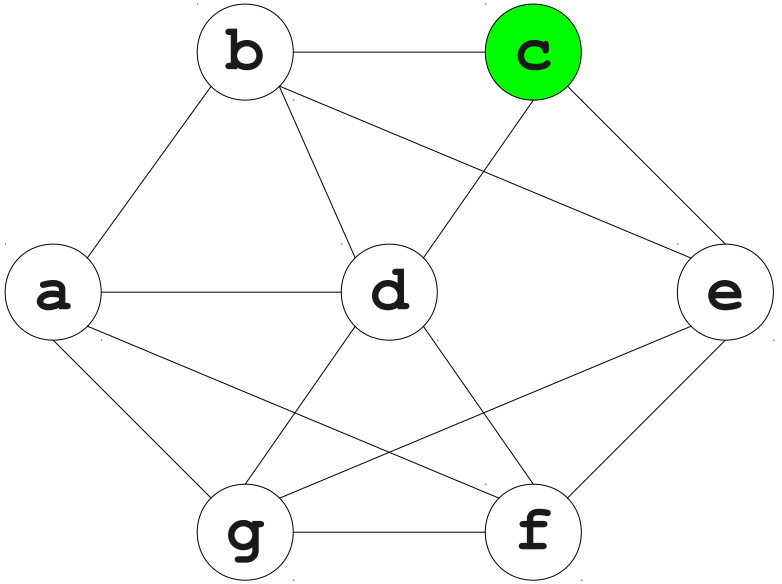
Chaitin's Algorithm



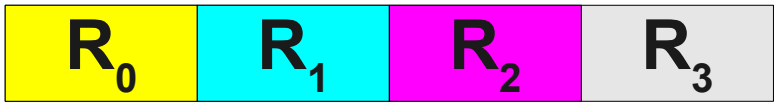
Chaitin's Algorithm



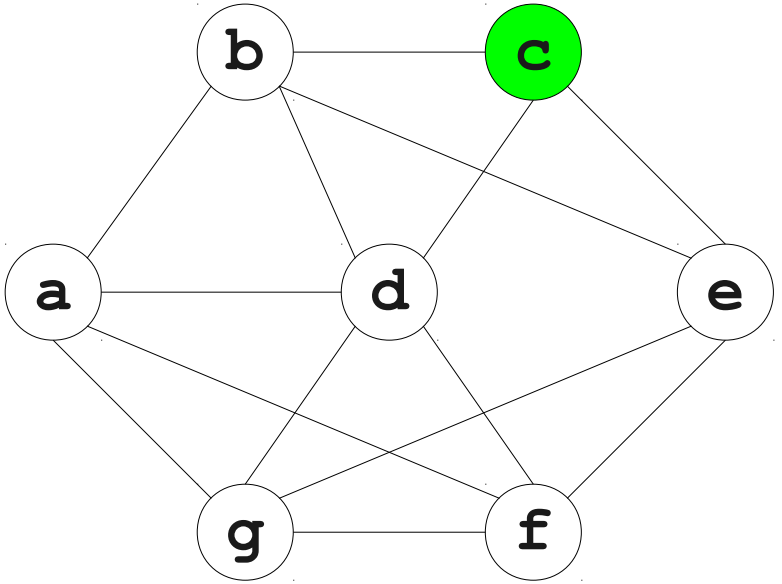
Chaitin's Algorithm

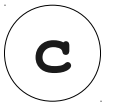


Registers

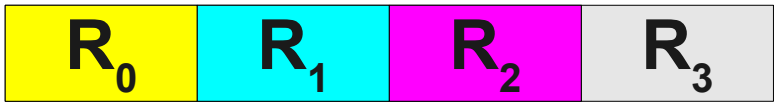


Chaitin's Algorithm

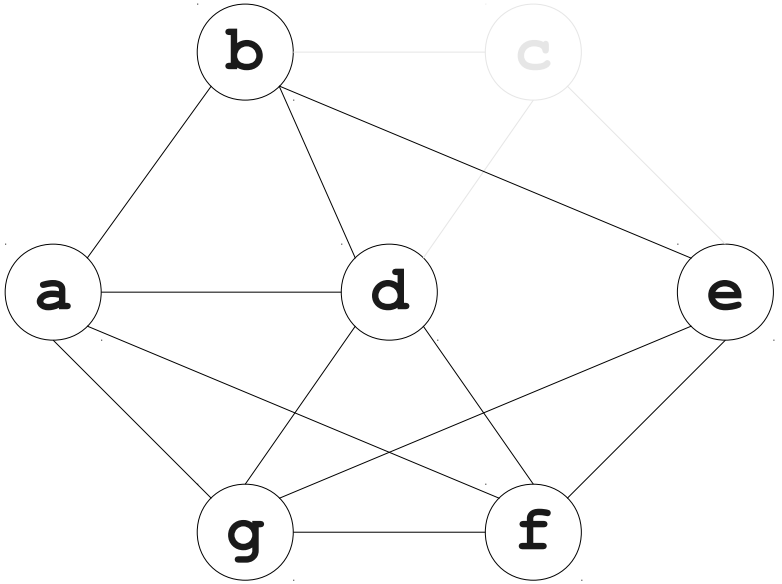




Registers



Chaitin's Algorithm

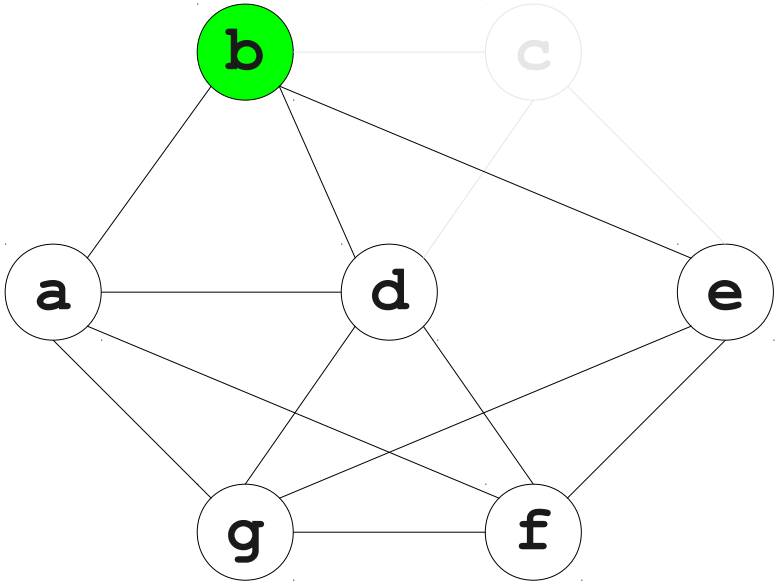


c

Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm

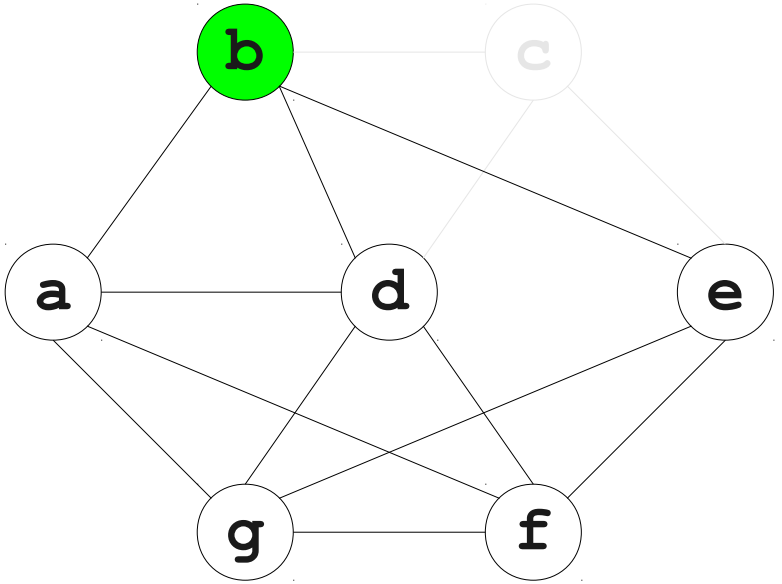


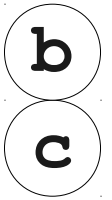
c

Registers

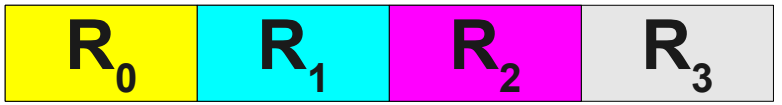
R_0	R_1	R_2	R_3

Chaitin's Algorithm

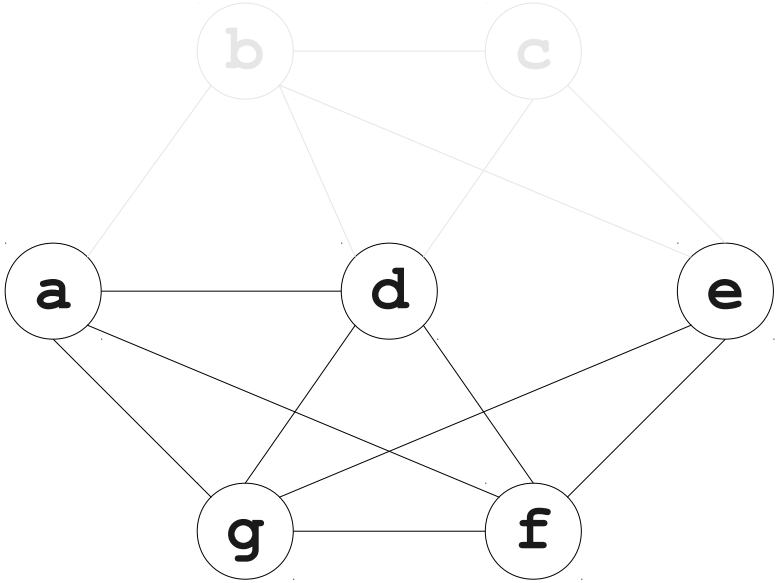


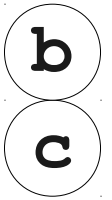


Registers

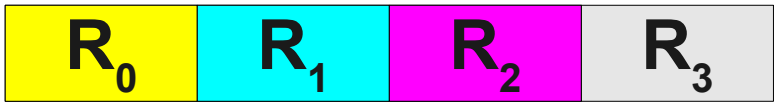


Chaitin's Algorithm

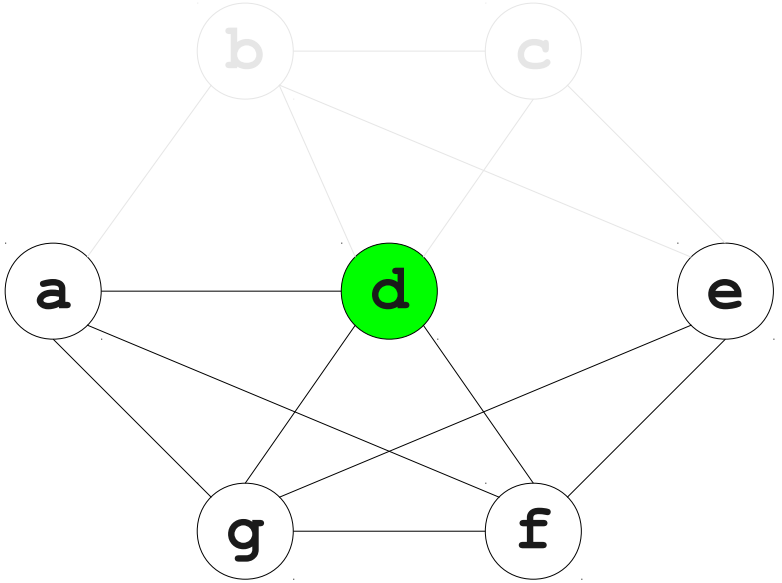


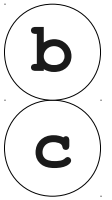


Registers

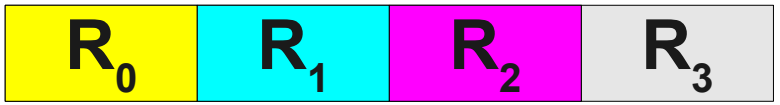


Chaitin's Algorithm

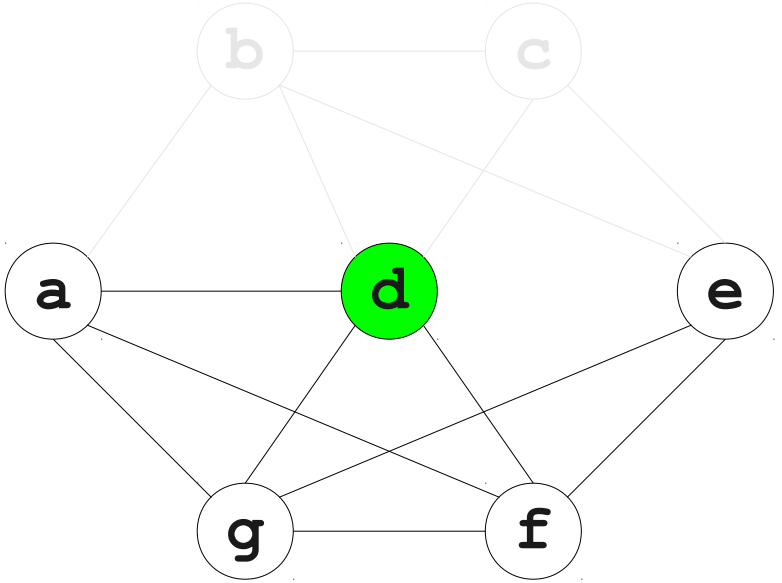


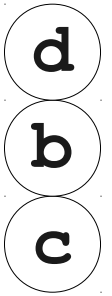


Registers

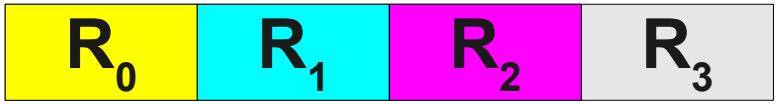


Chaitin's Algorithm

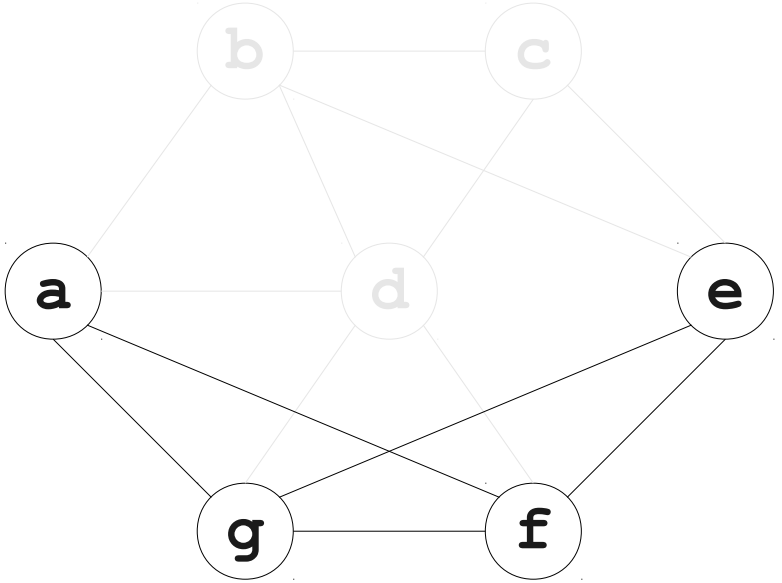


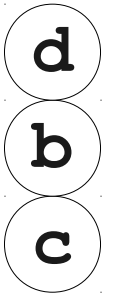


Registers

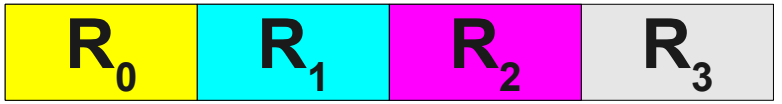


Chaitin's Algorithm

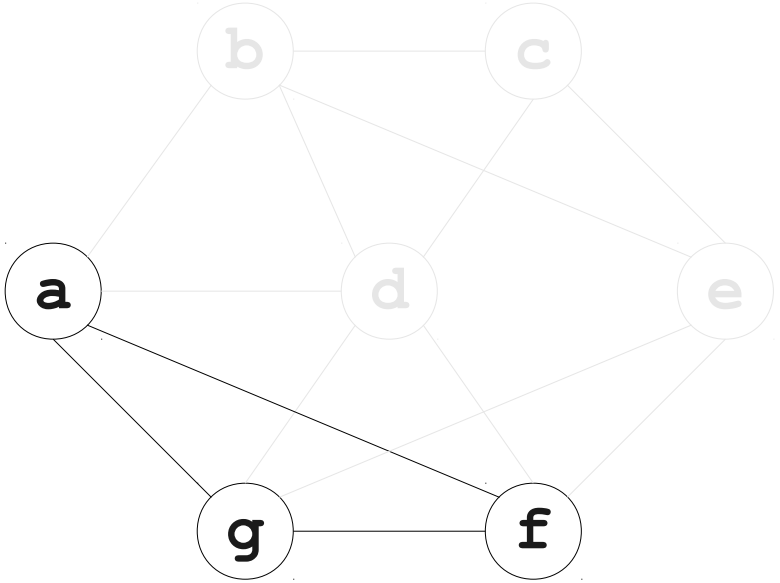


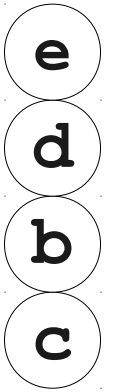


Registers

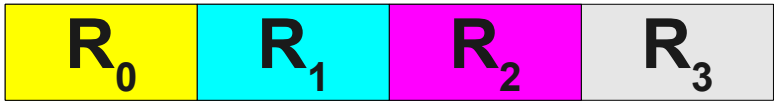


Chaitin's Algorithm

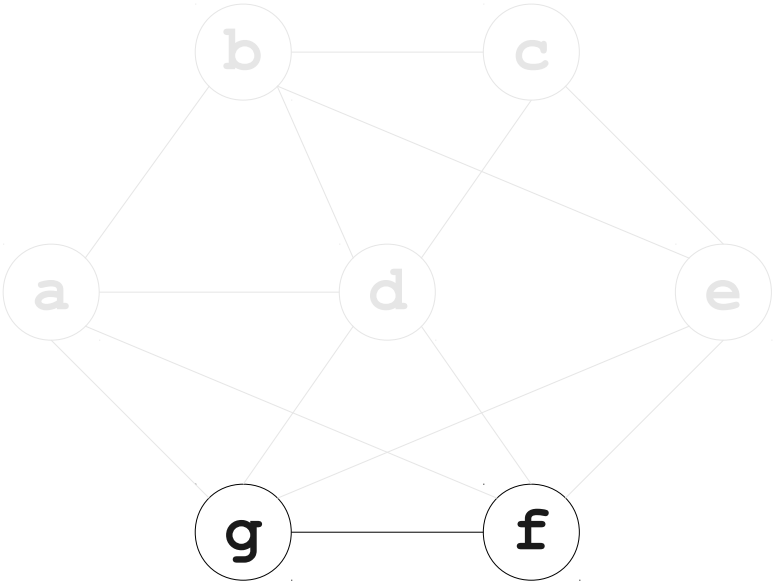


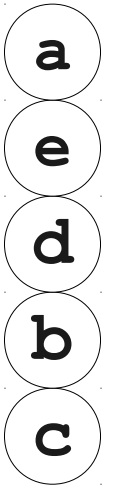


Registers

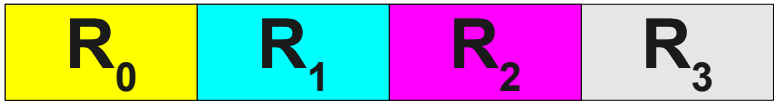


Chaitin's Algorithm

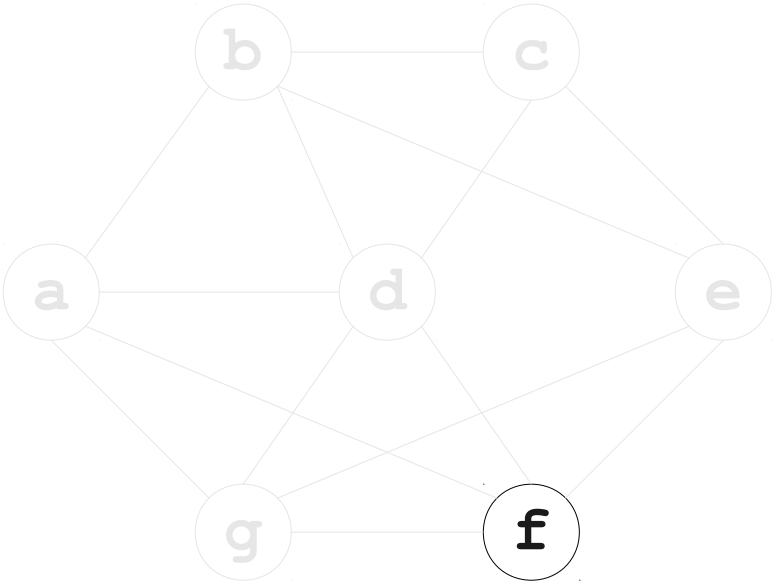


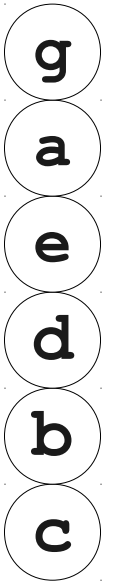


Registers

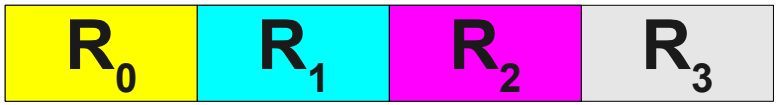


Chaitin's Algorithm

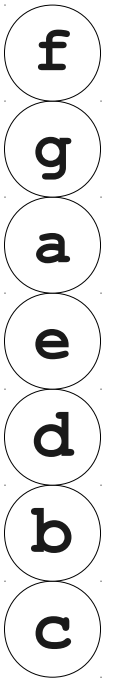




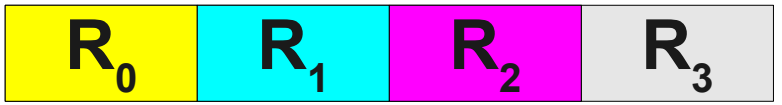
Registers



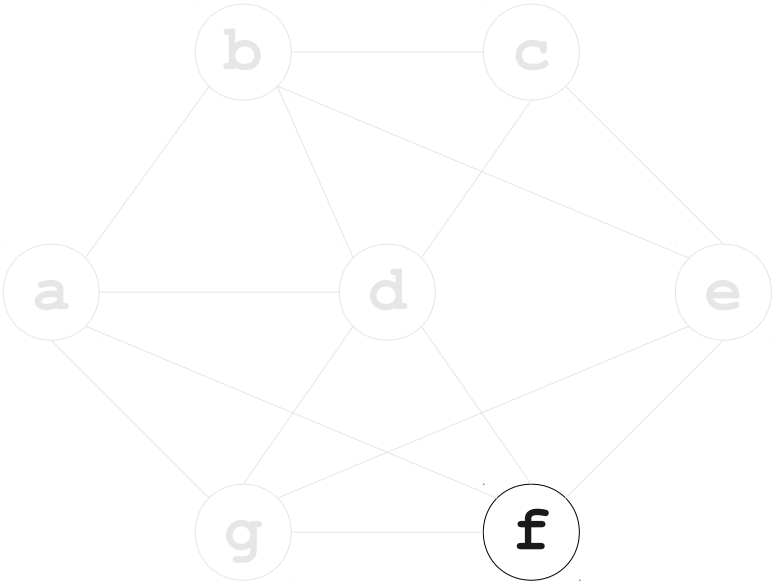
Chaitin's Algorithm

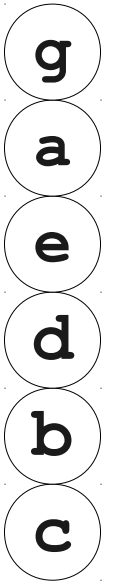


Registers

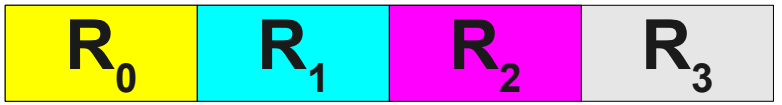


Chaitin's Algorithm

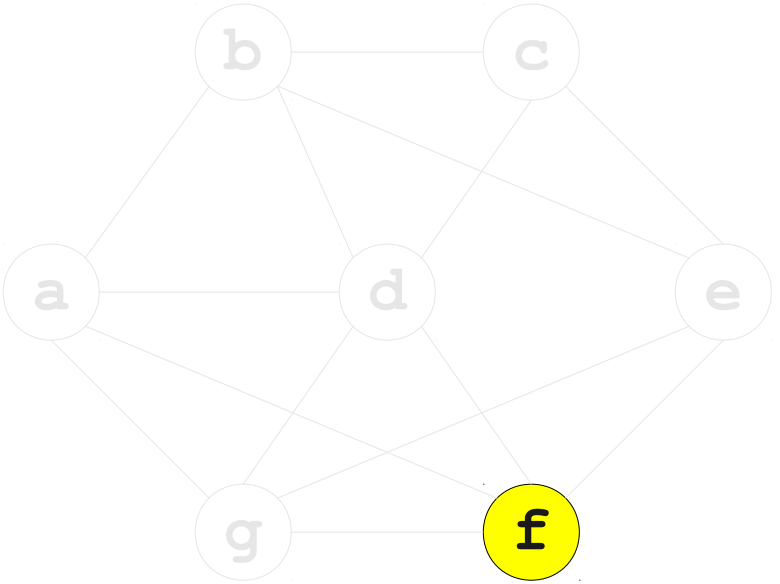


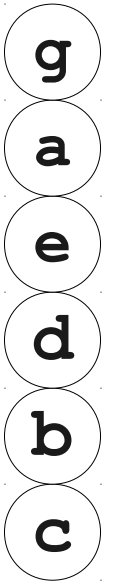


Registers

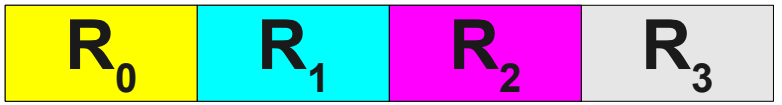


Chaitin's Algorithm

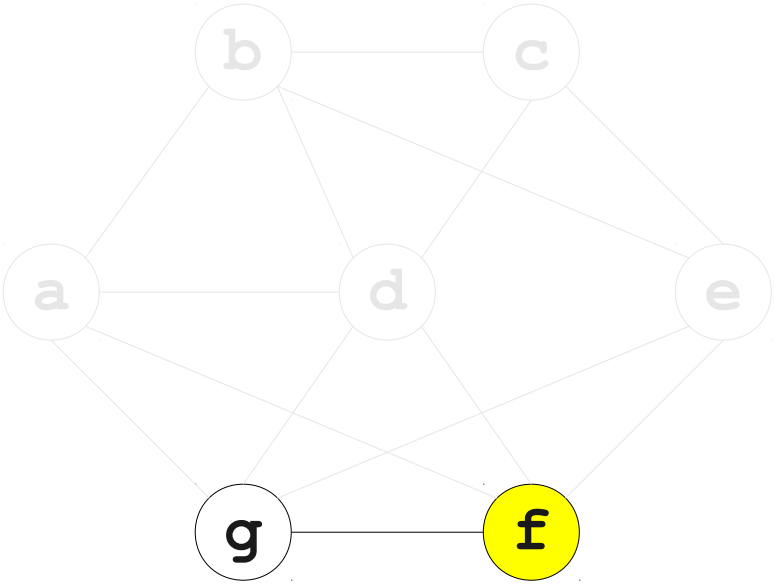


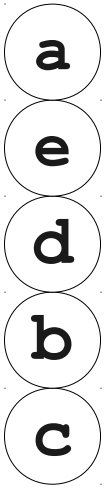


Registers

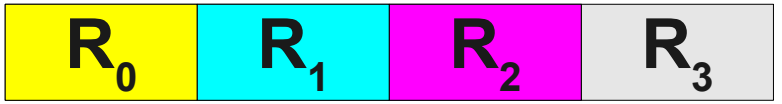


Chaitin's Algorithm

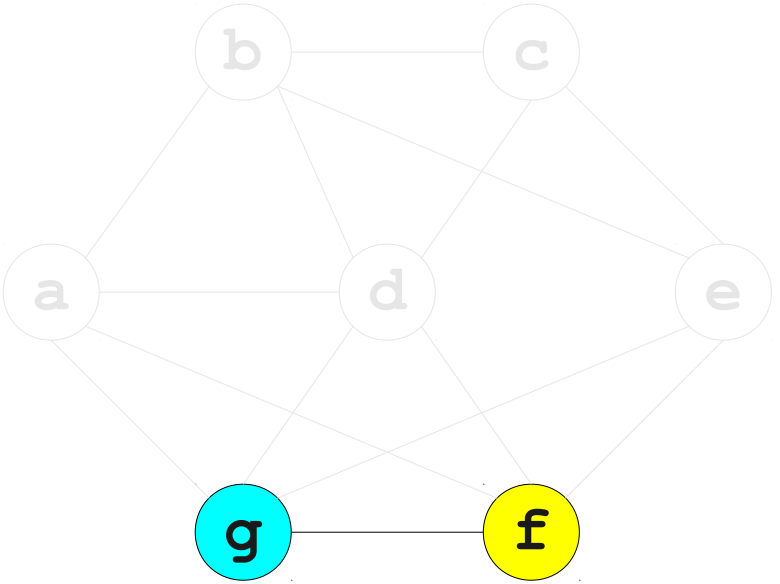


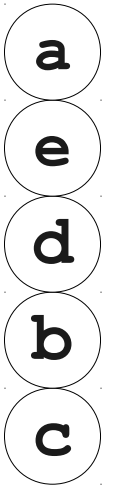


Registers

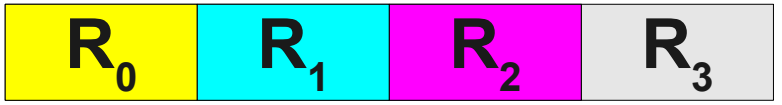


Chaitin's Algorithm

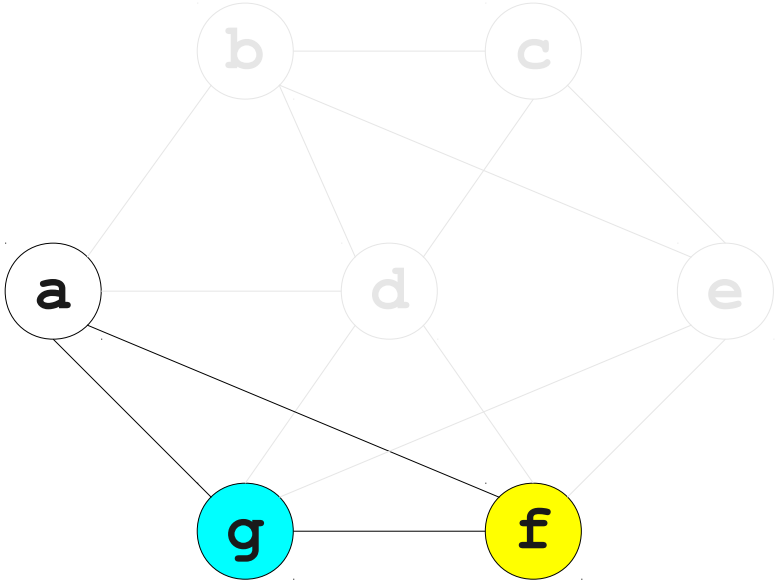


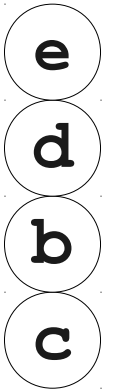


Registers

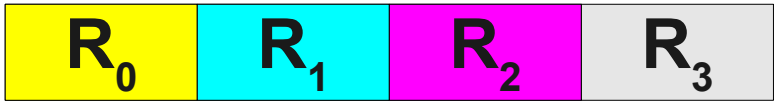


Chaitin's Algorithm

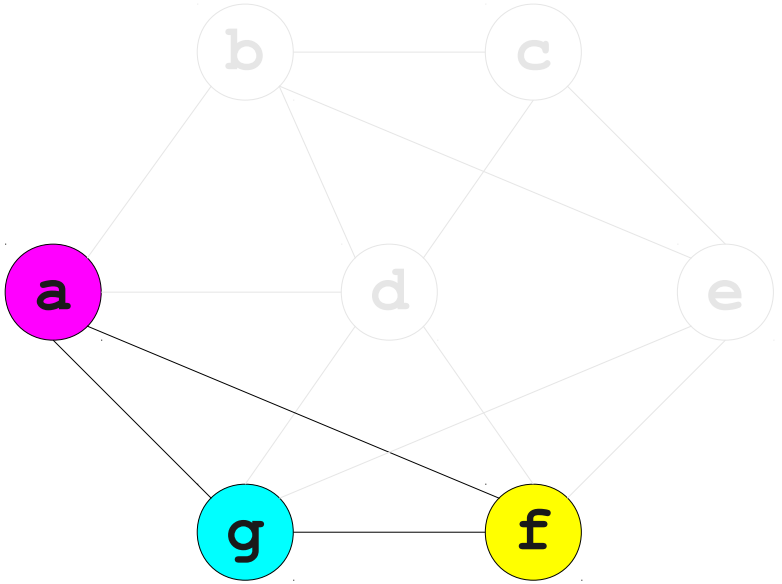


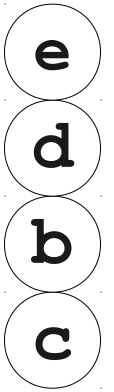


Registers

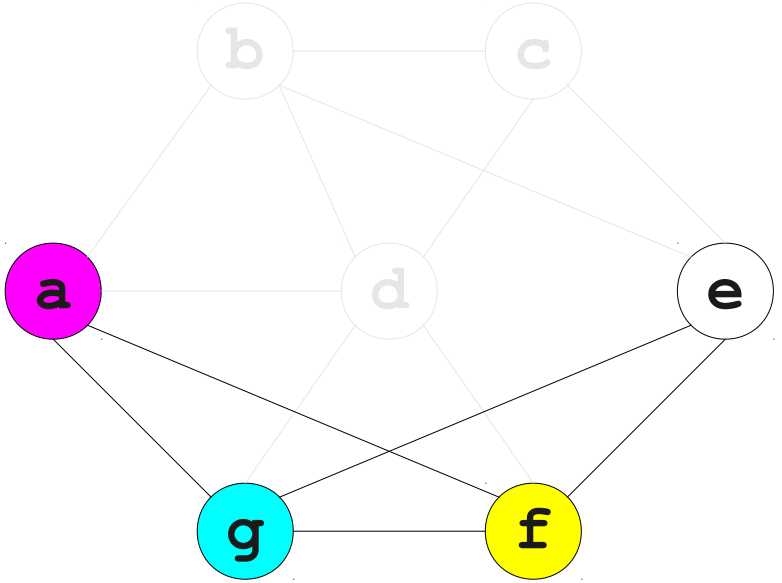


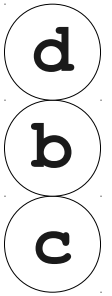
Chaitin's Algorithm



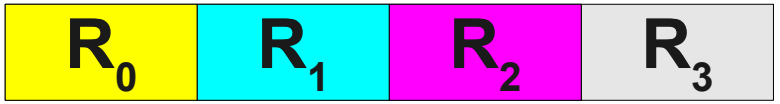


Chaitin's Algorithm

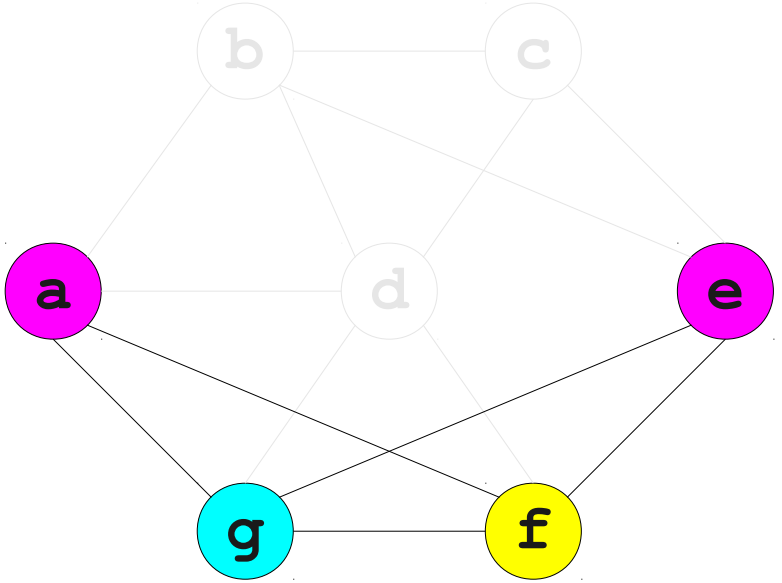


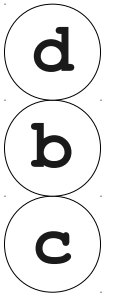


Registers

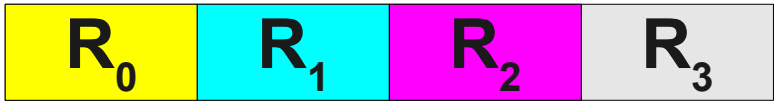


Chaitin's Algorithm

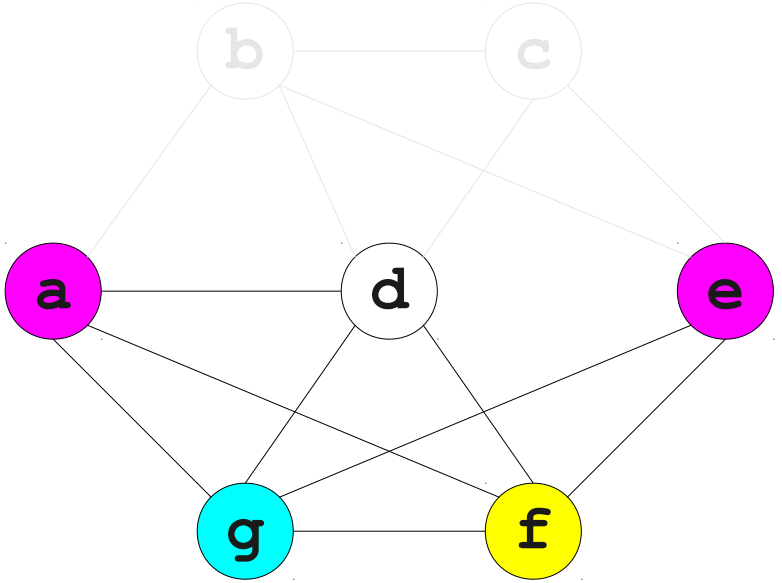


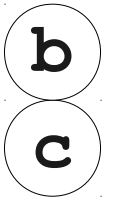


Registers

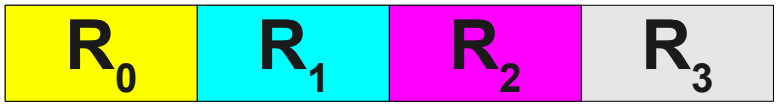


Chaitin's Algorithm

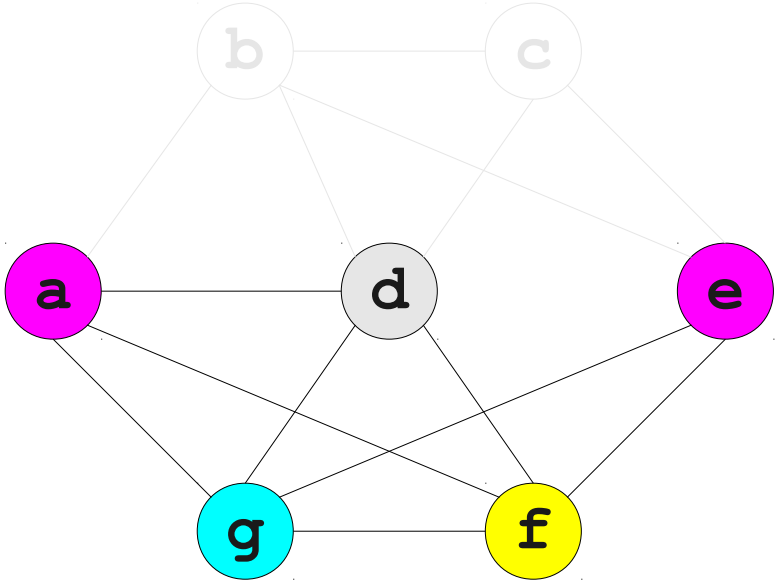


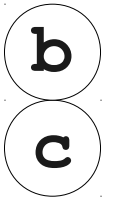


Registers

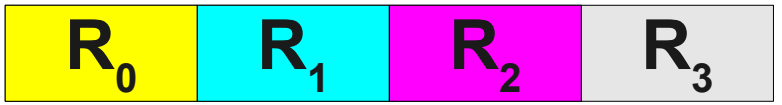


Chaitin's Algorithm

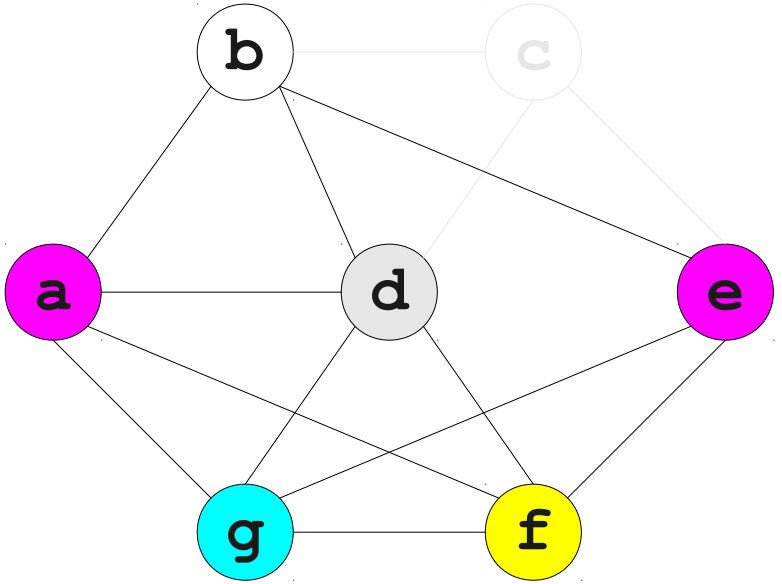




Registers



Chaitin's Algorithm

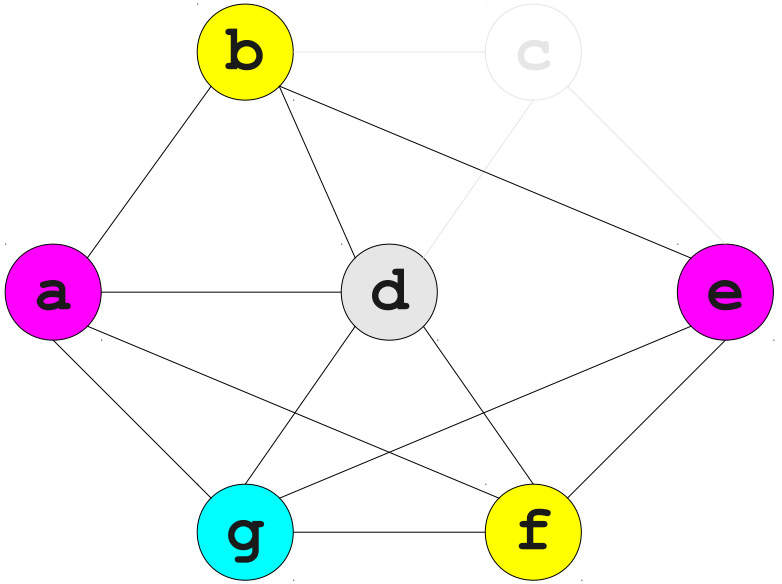


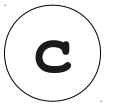
c

Registers

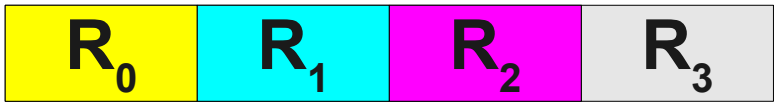
R_0	R_1	R_2	R_3

Chaitin's Algorithm

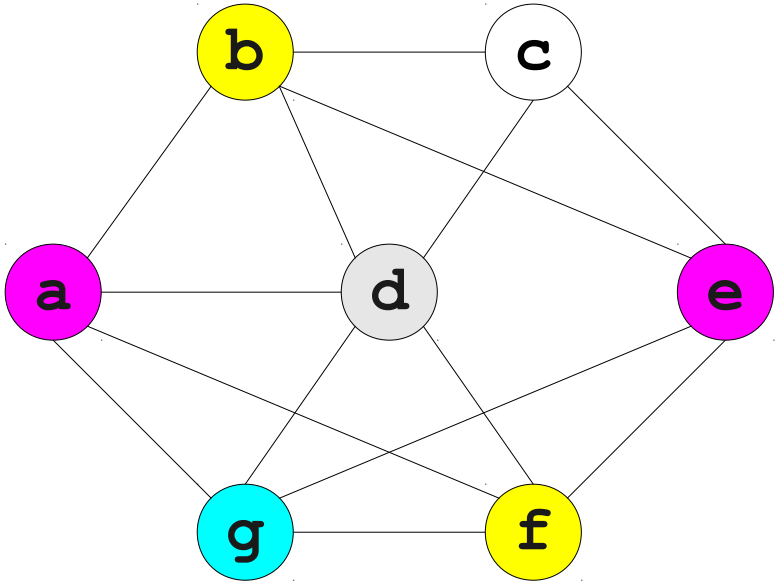




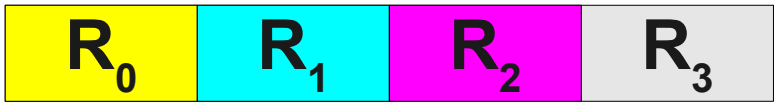
Registers



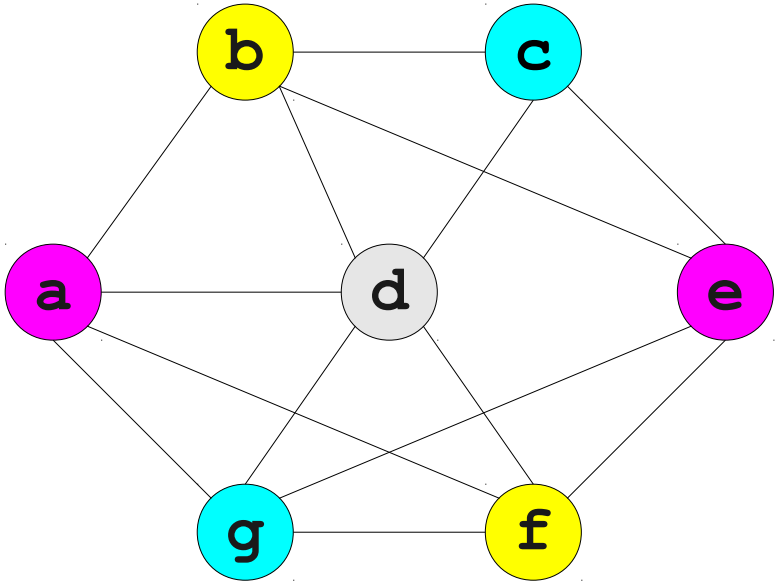
Chaitin's Algorithm



Registers



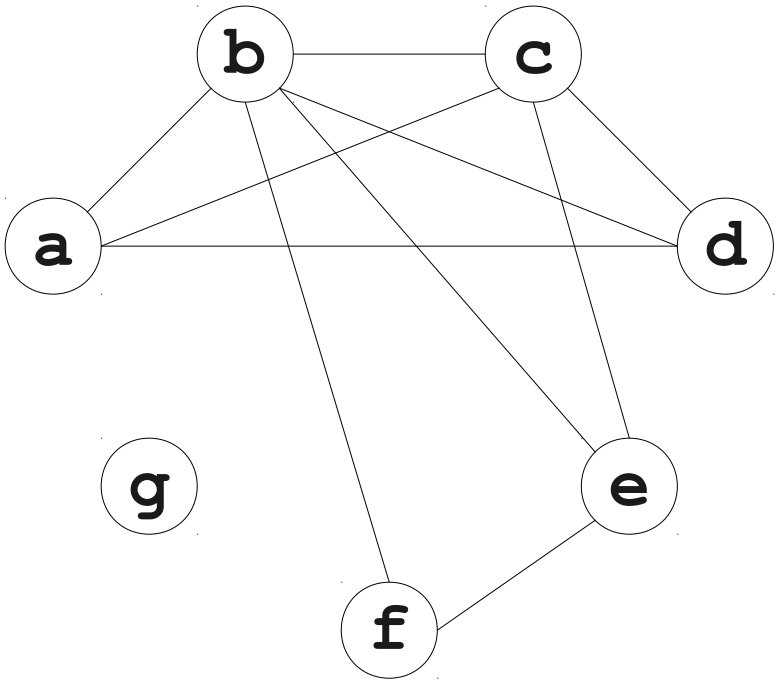
Chaitin's Algorithm



One Problem

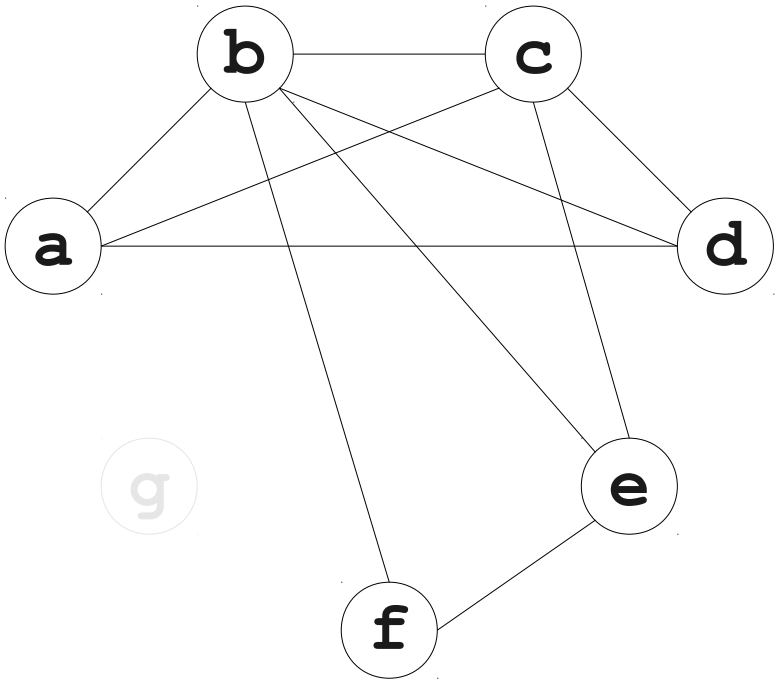
- What if we can't find a node with < k neighbors?
- Choose and remove an arbitrary node, marking it “troublesome.”
  - Use heuristics to choose which one.
- When adding node back in, it may be possible to find a valid color.
- Otherwise, we have to spill that node.

Chaitin's Algorithm Reloaded



Registers

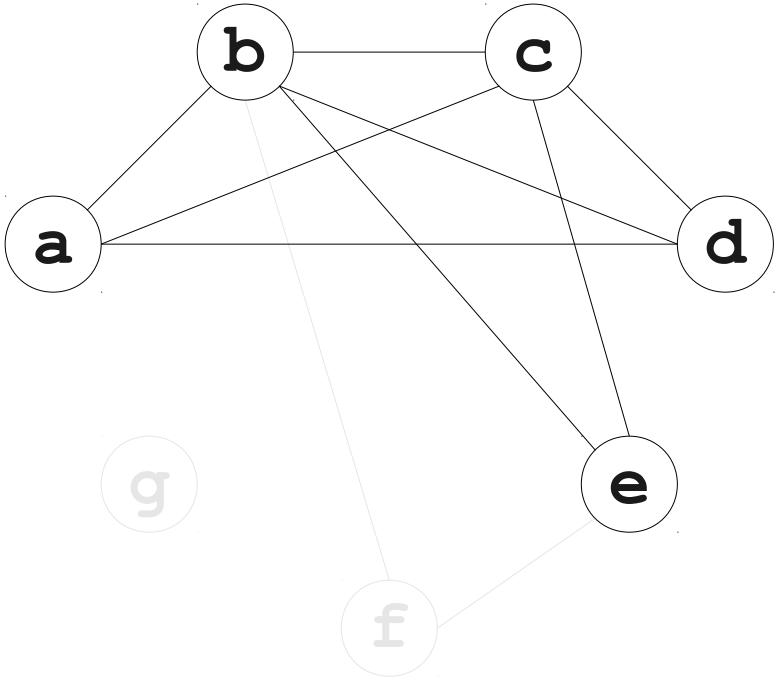
Chaitin's Algorithm Reloaded



g

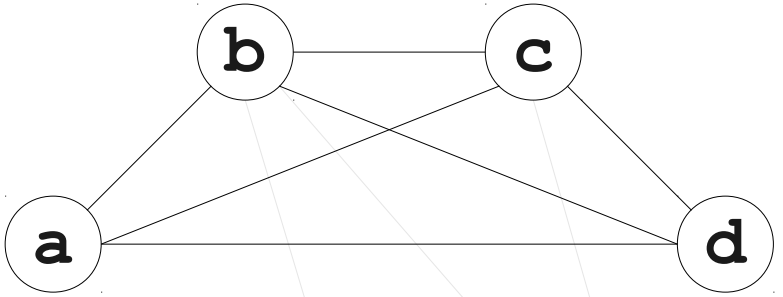
Registers

Chaitin's Algorithm Reloaded



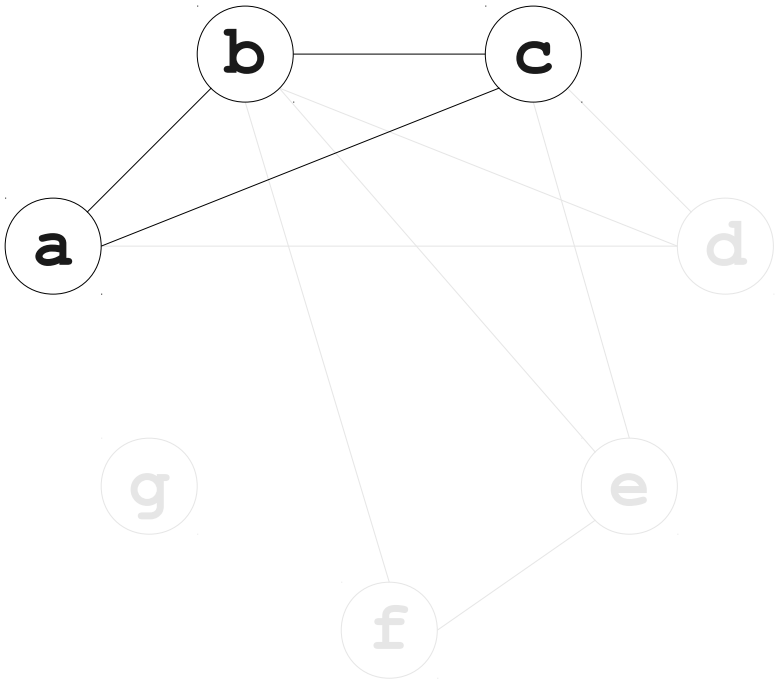
Registers

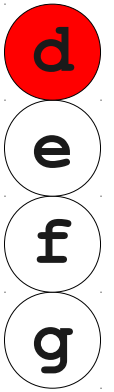
Chaitin's Algorithm Reloaded



Registers

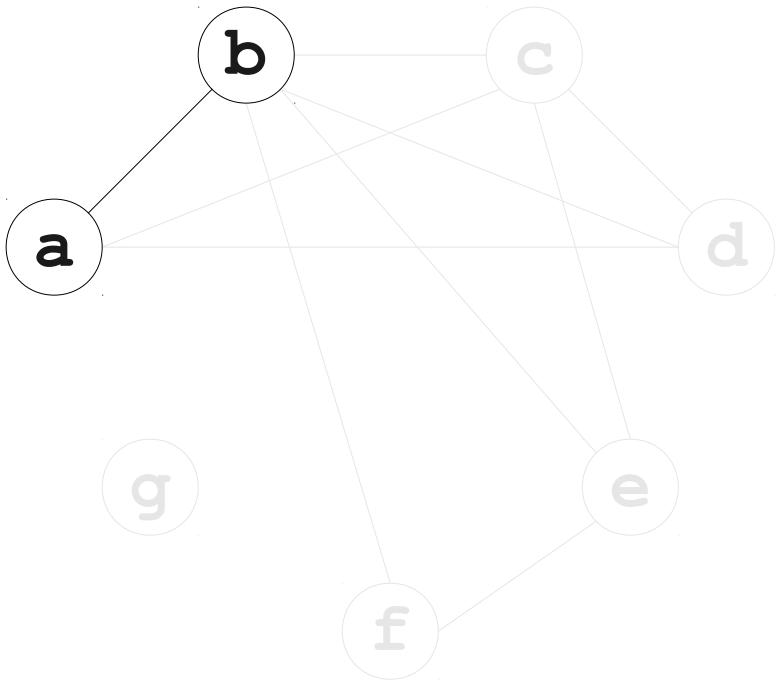
Chaitin's Algorithm Reloaded

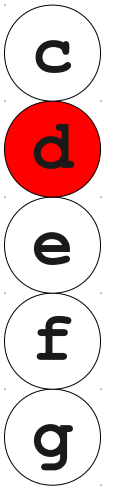




Registers

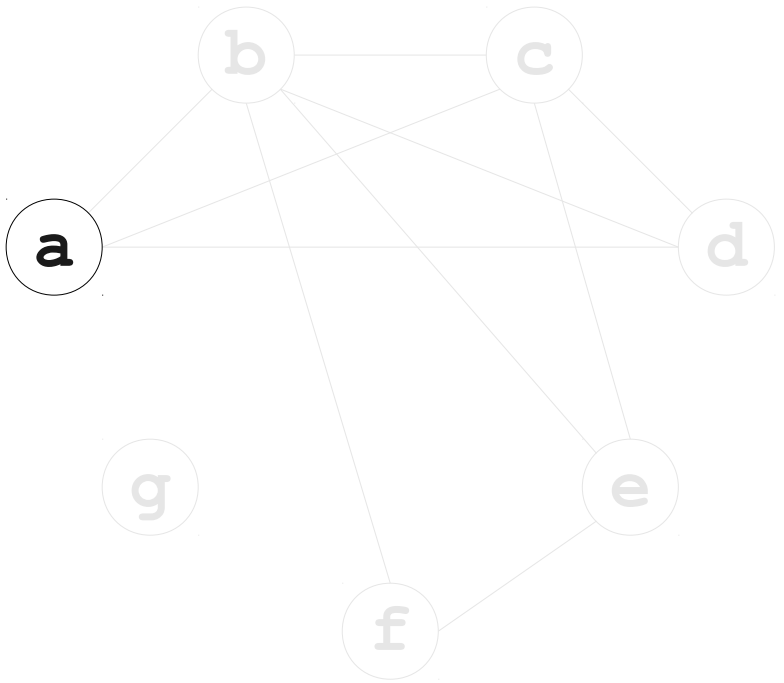
Chaitin's Algorithm Reloaded

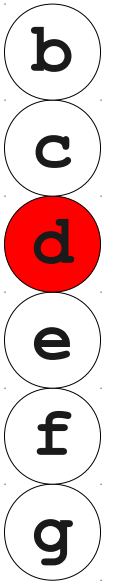




Registers

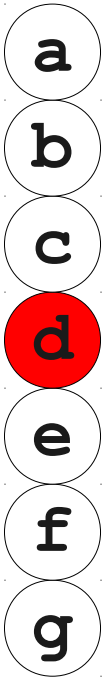
Chaitin's Algorithm Reloaded





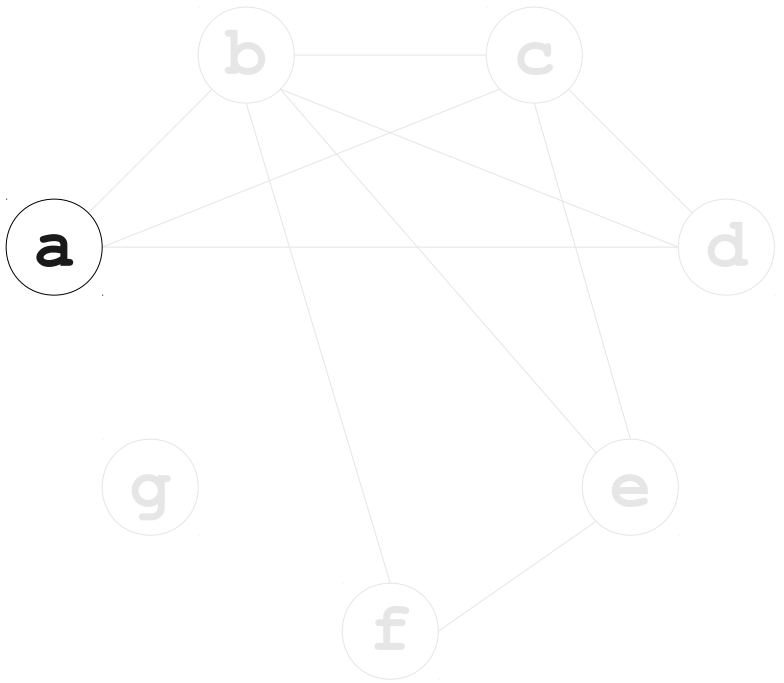
Registers

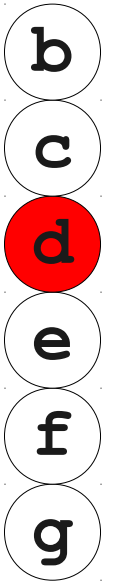
Chaitin's Algorithm Reloaded



Registers

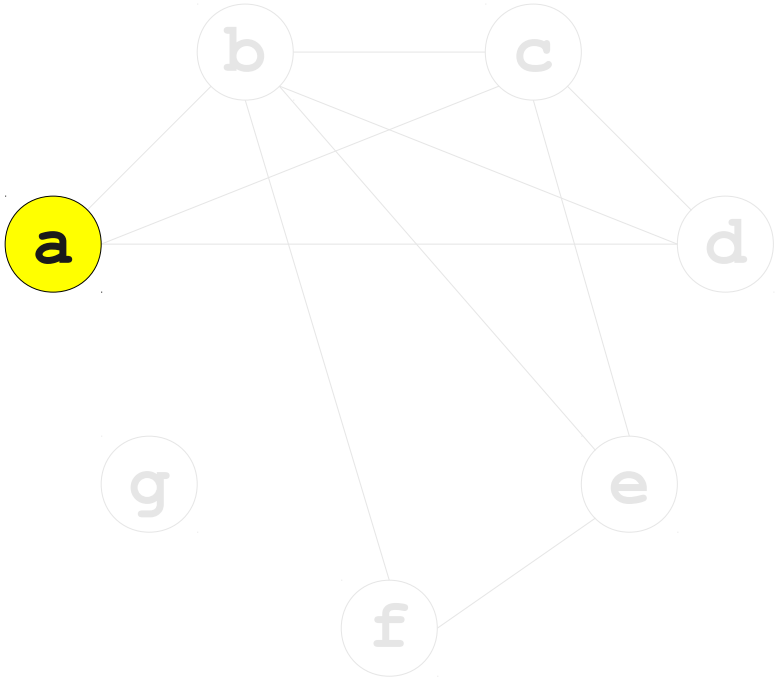
Chaitin's Algorithm Reloaded

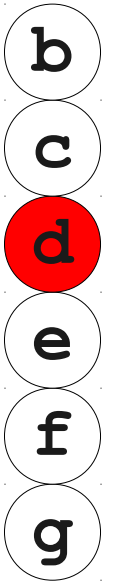




Registers

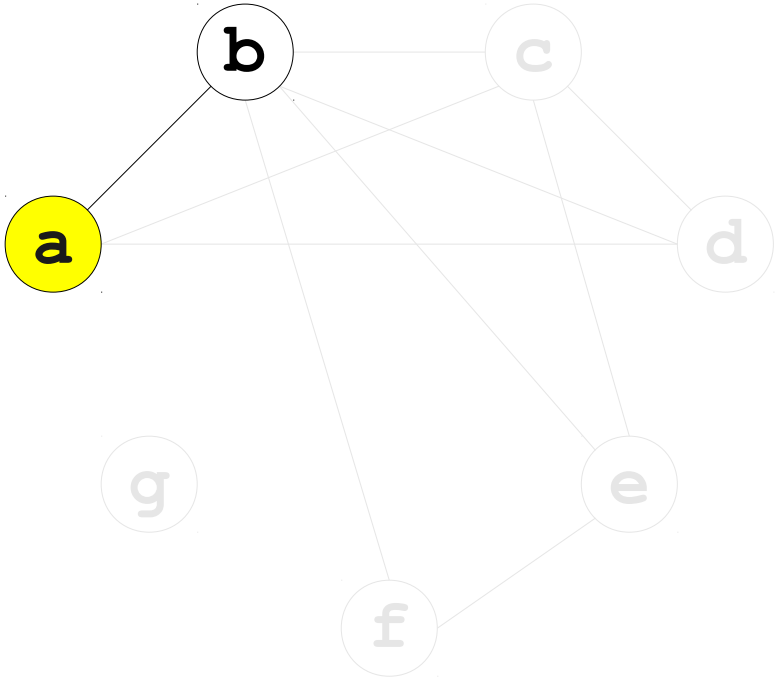
Chaitin's Algorithm Reloaded

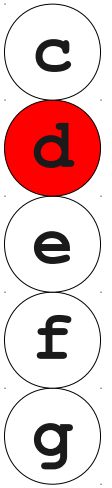




Registers

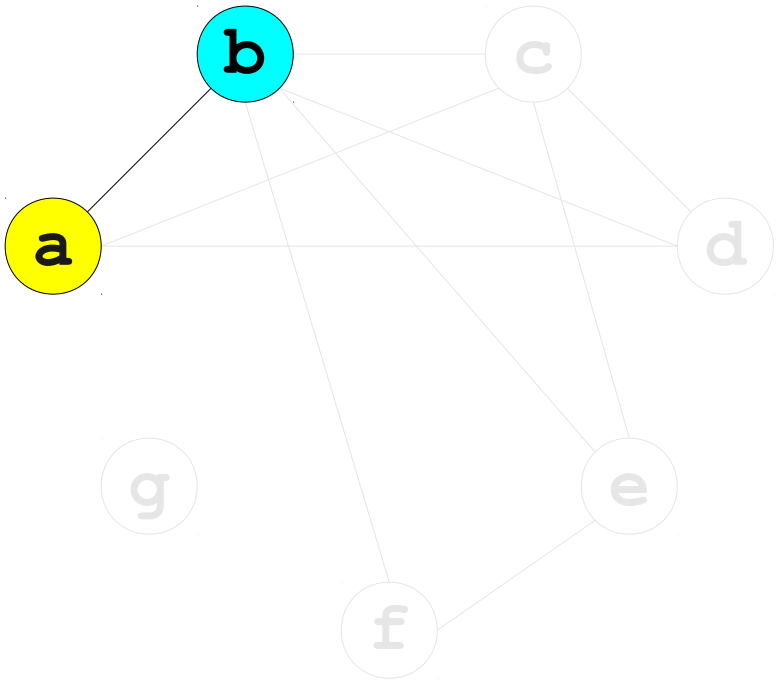
Chaitin's Algorithm Reloaded

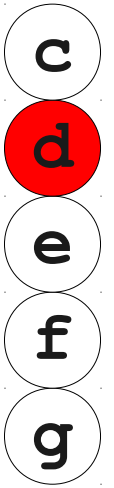




Registers

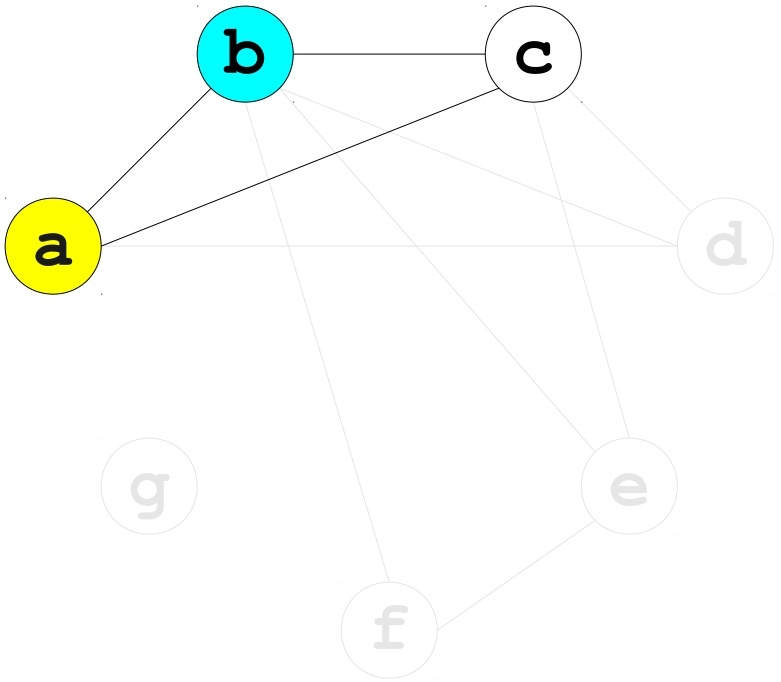
Chaitin's Algorithm Reloaded

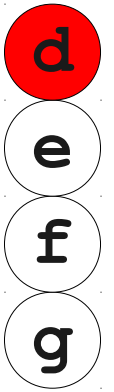




Registers

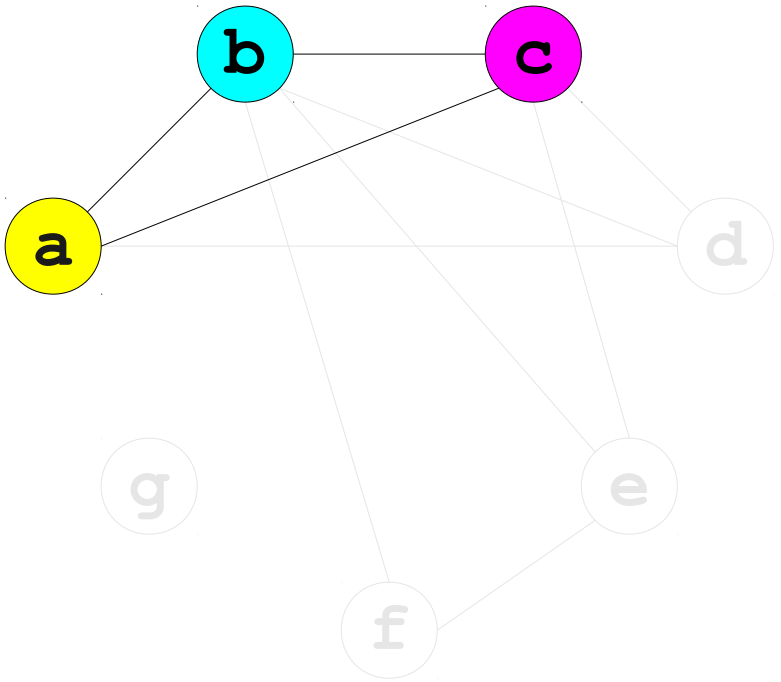
Chaitin's Algorithm Reloaded

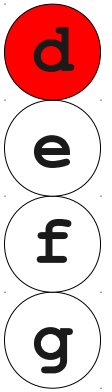




Registers

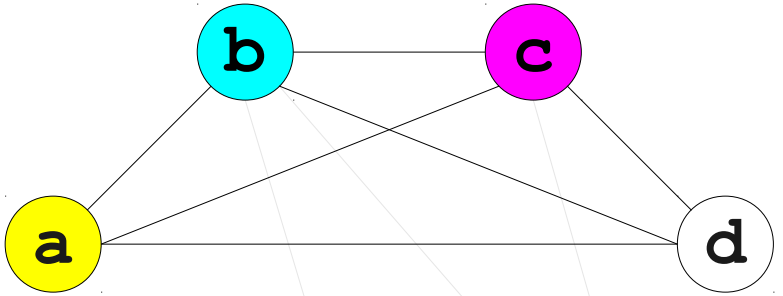
Chaitin's Algorithm Reloaded





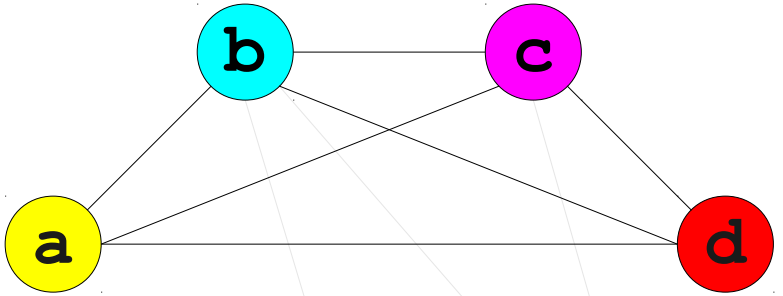
Registers

Chaitin's Algorithm Reloaded



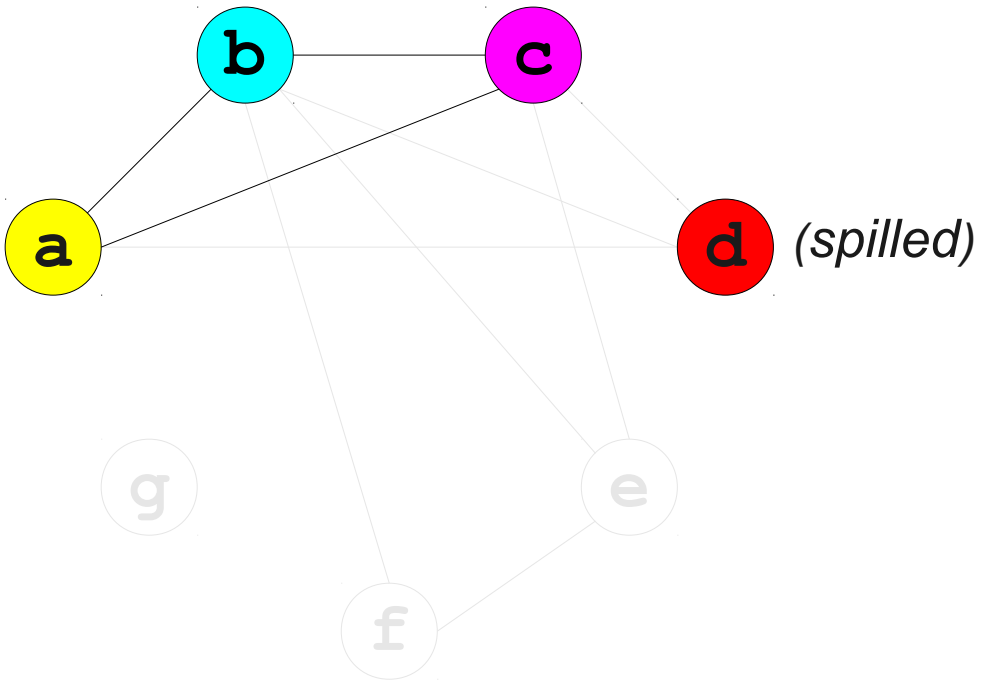
Registers

Chaitin's Algorithm Reloaded



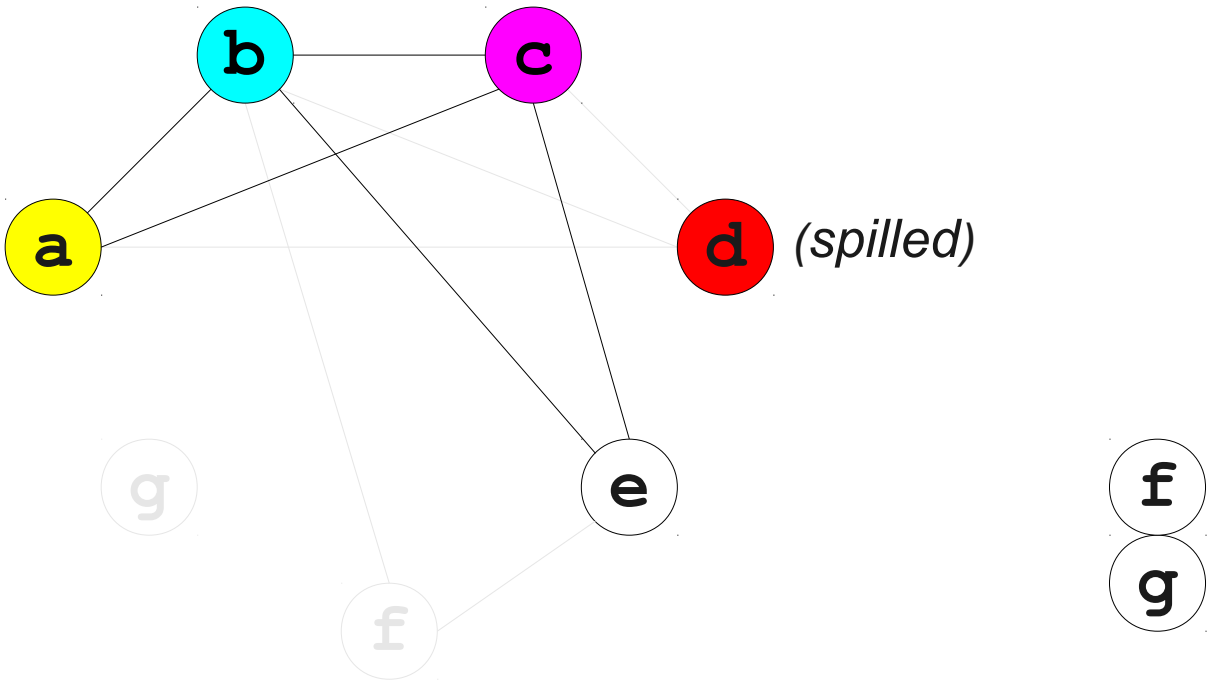
Registers

Chaitin's Algorithm Reloaded



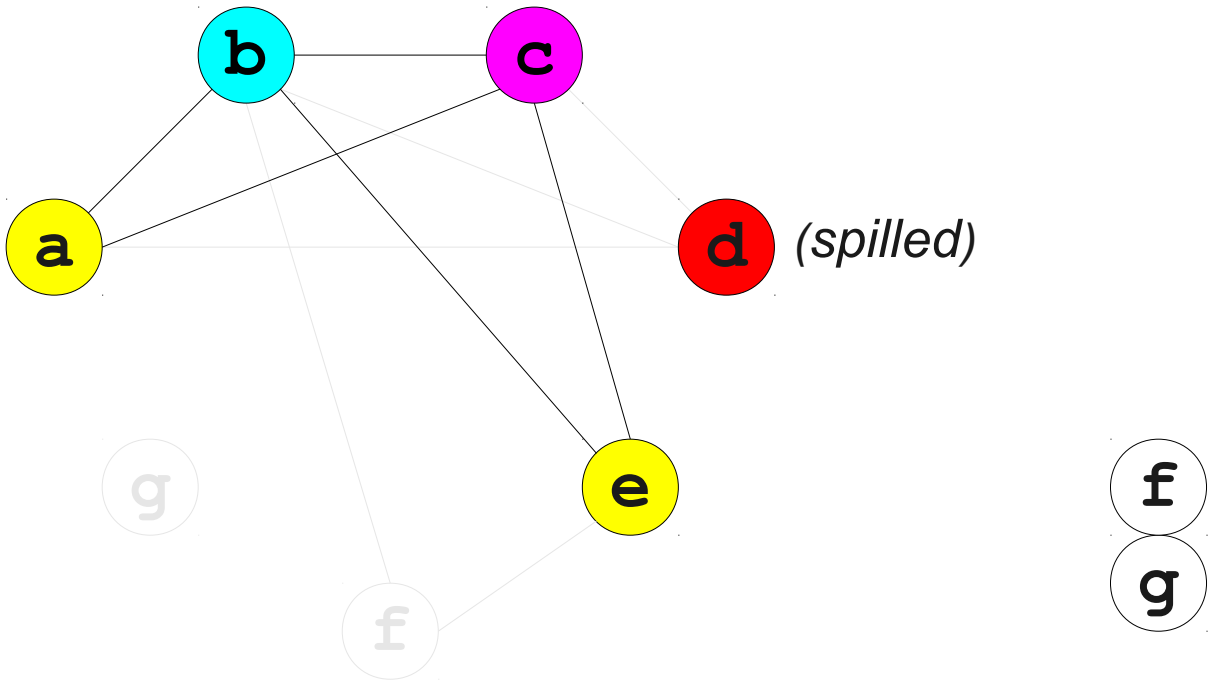
Registers

Chaitin's Algorithm Reloaded



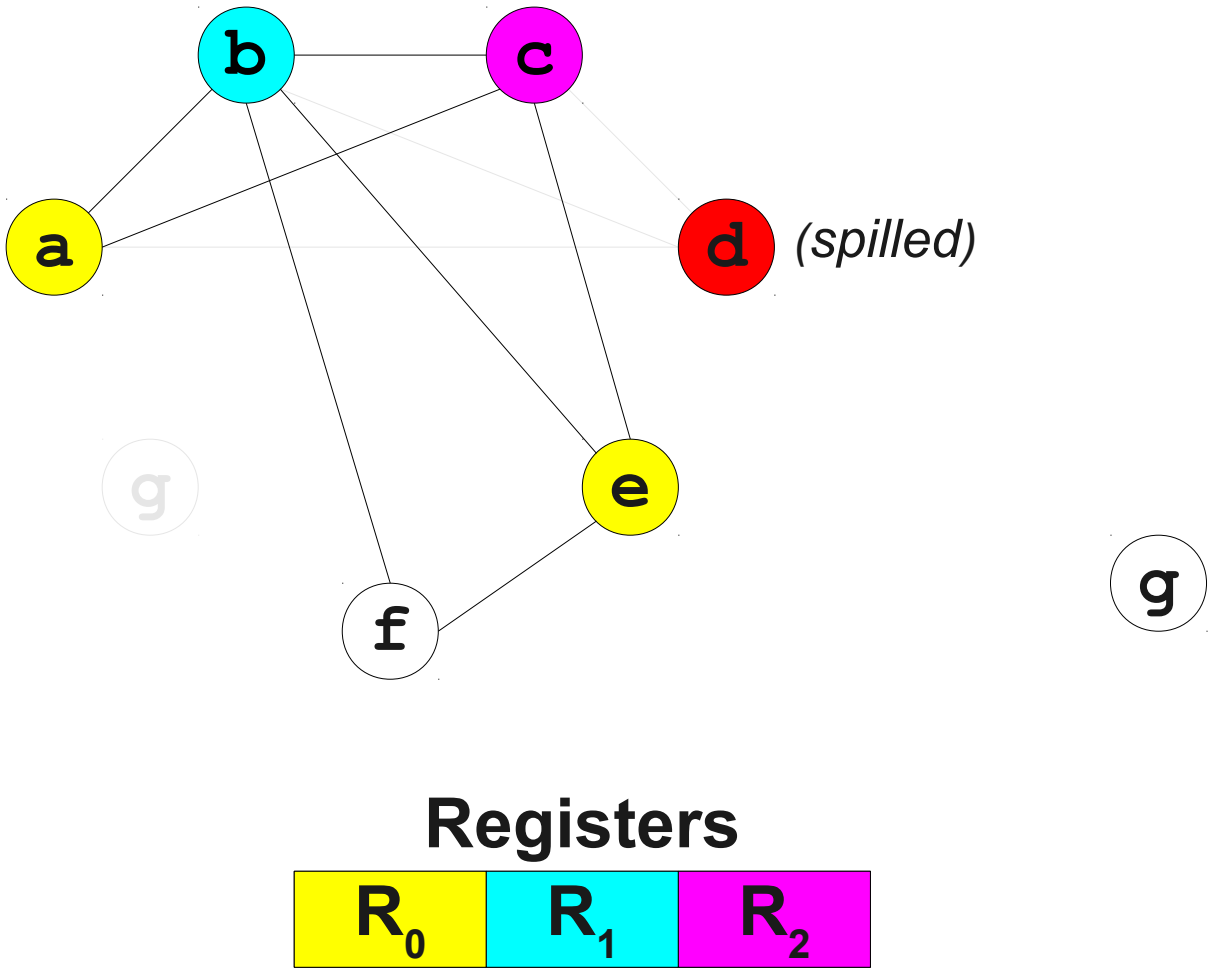
Registers

Chaitin's Algorithm Reloaded

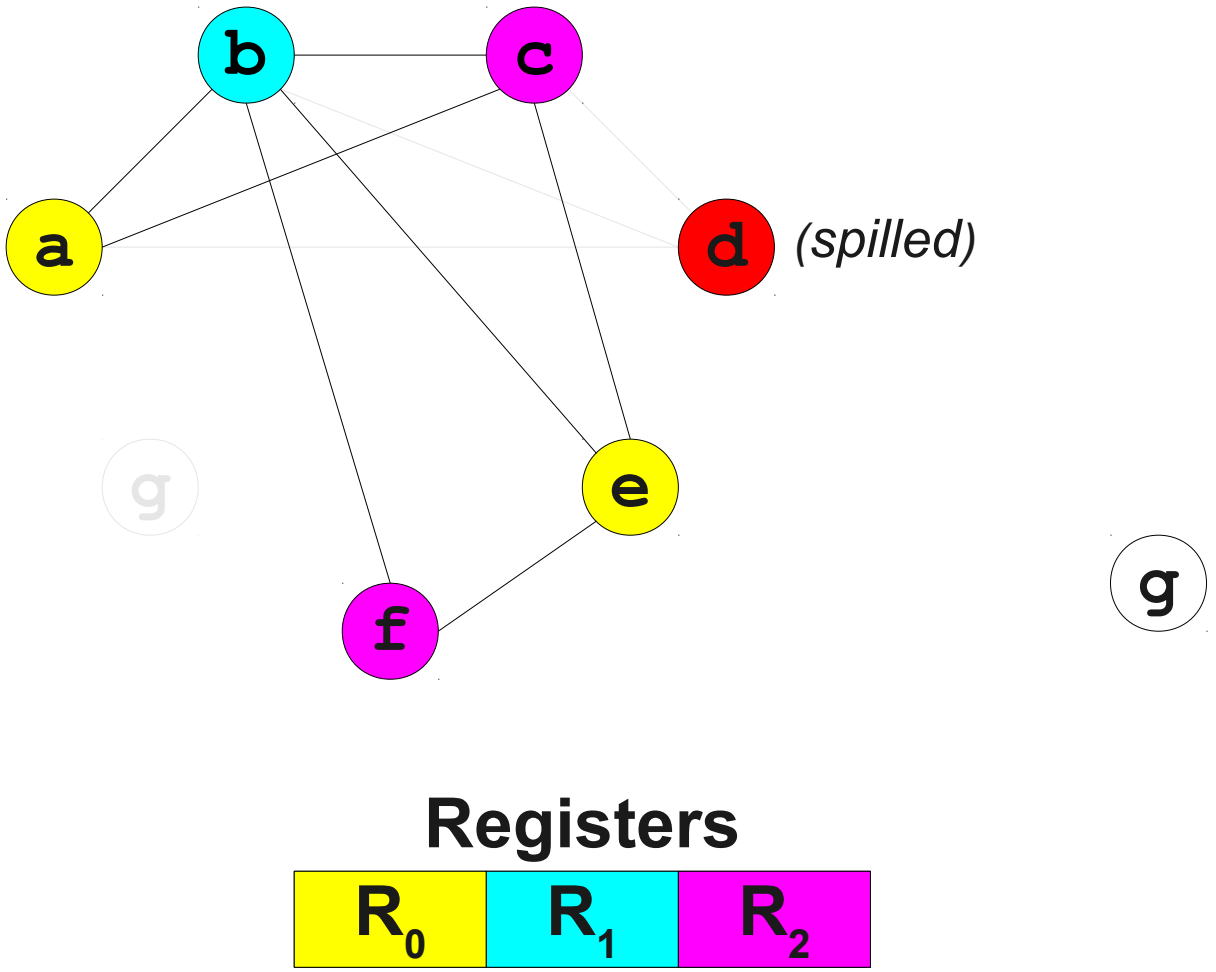


Registers

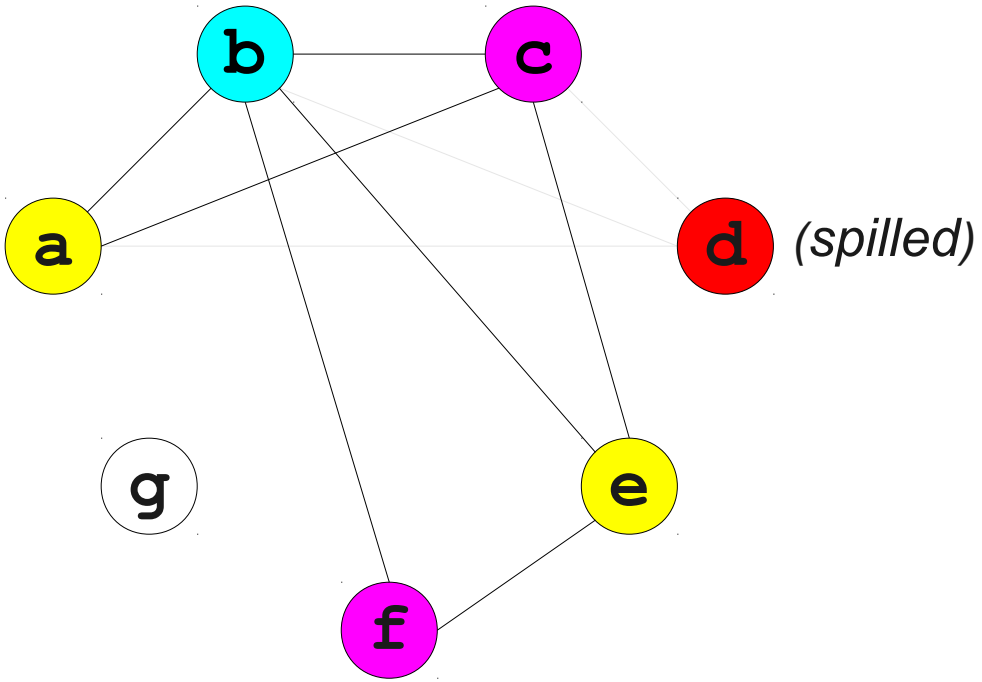
Chaitin's Algorithm Reloaded



Chaitin's Algorithm Reloaded

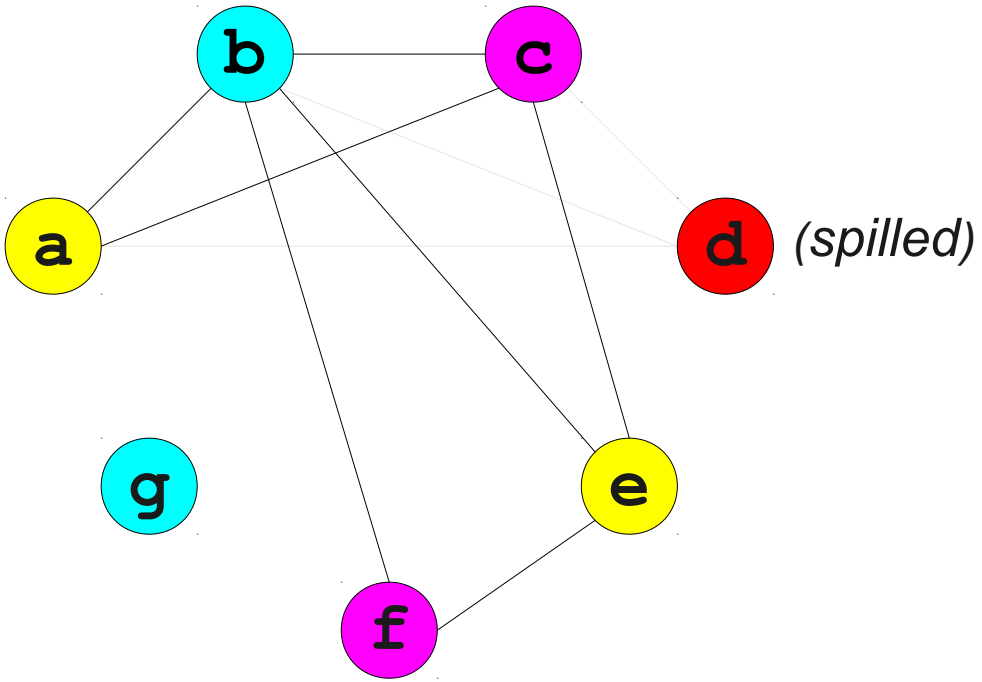


Chaitin's Algorithm Reloaded



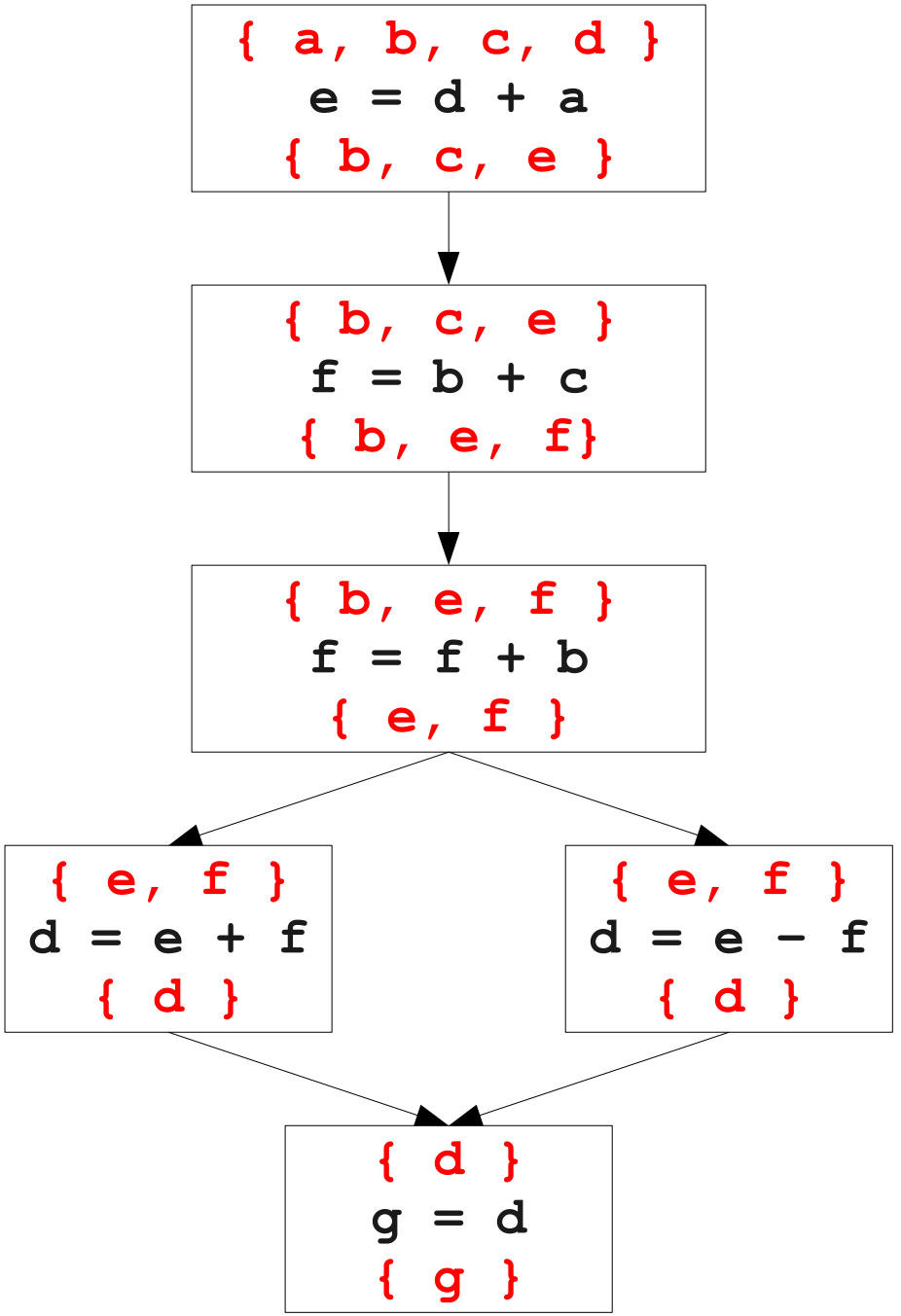
Registers

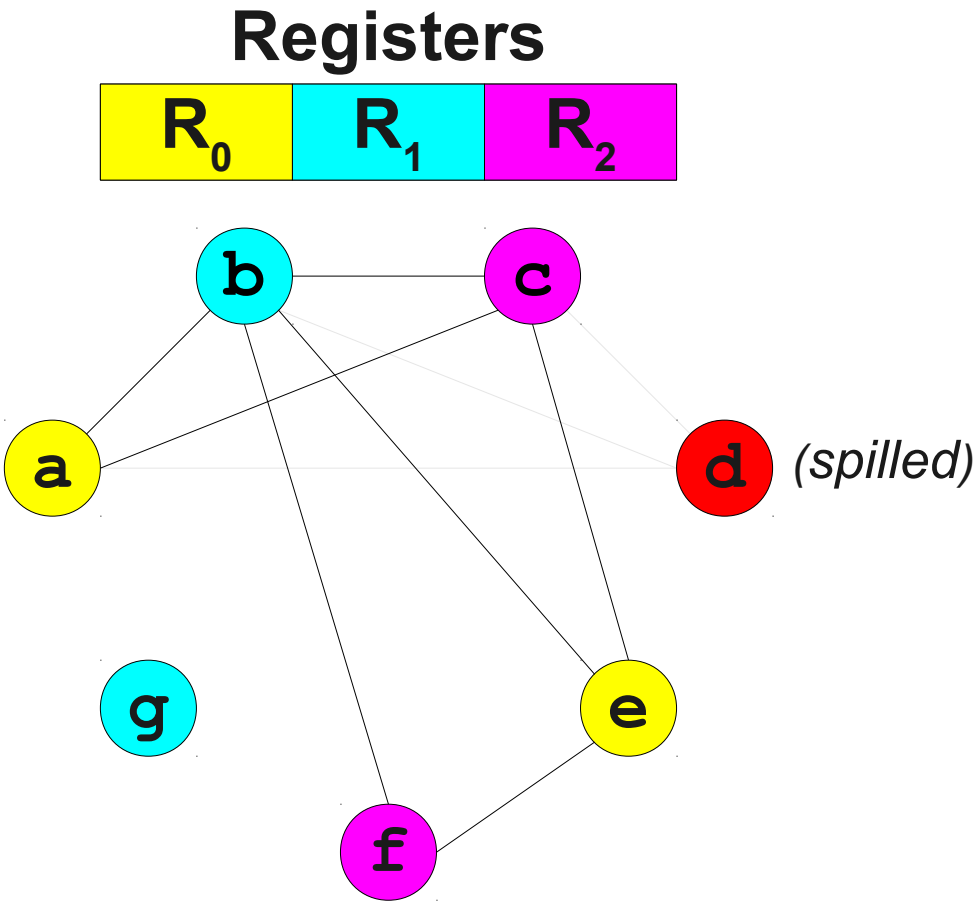
Chaitin's Algorithm Reloaded



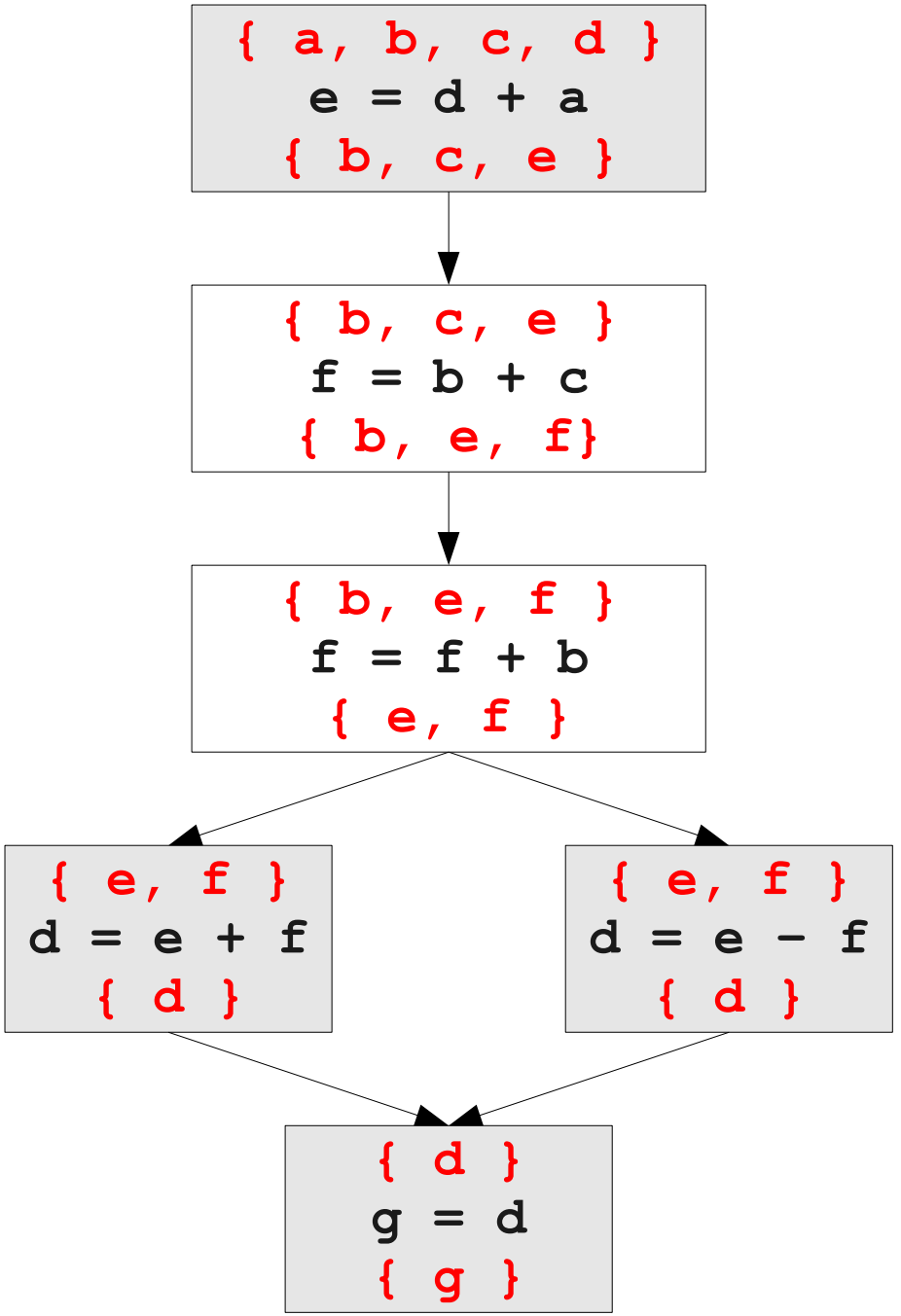
Registers

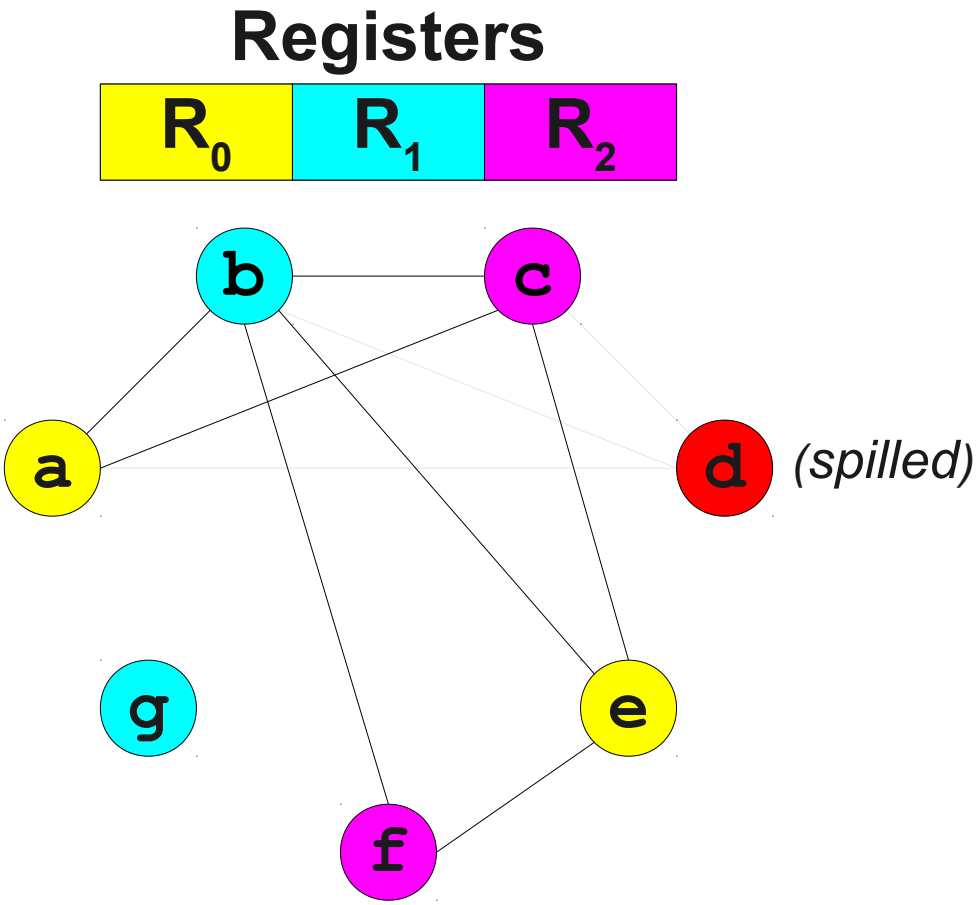
A Smarter Algorithm



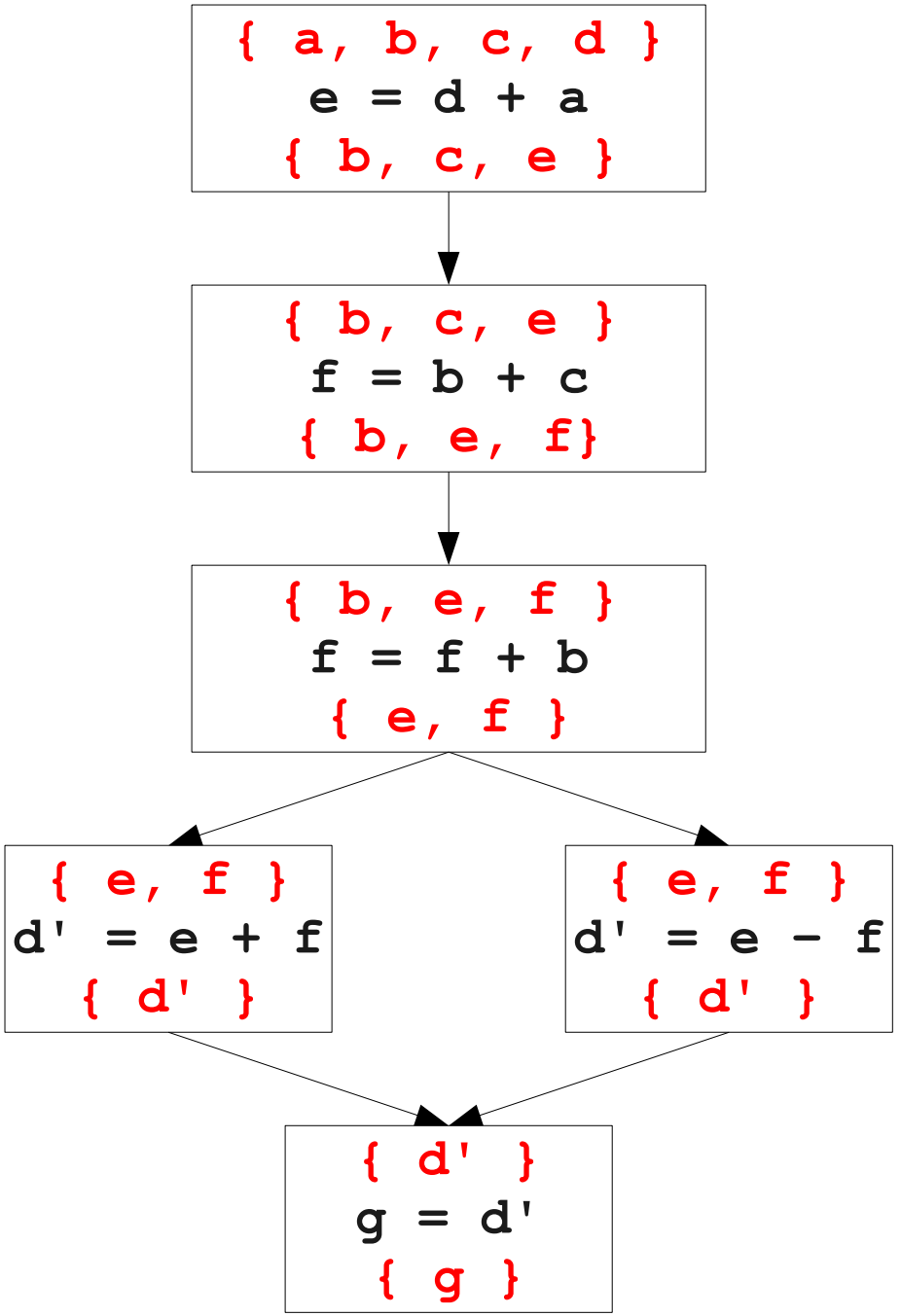


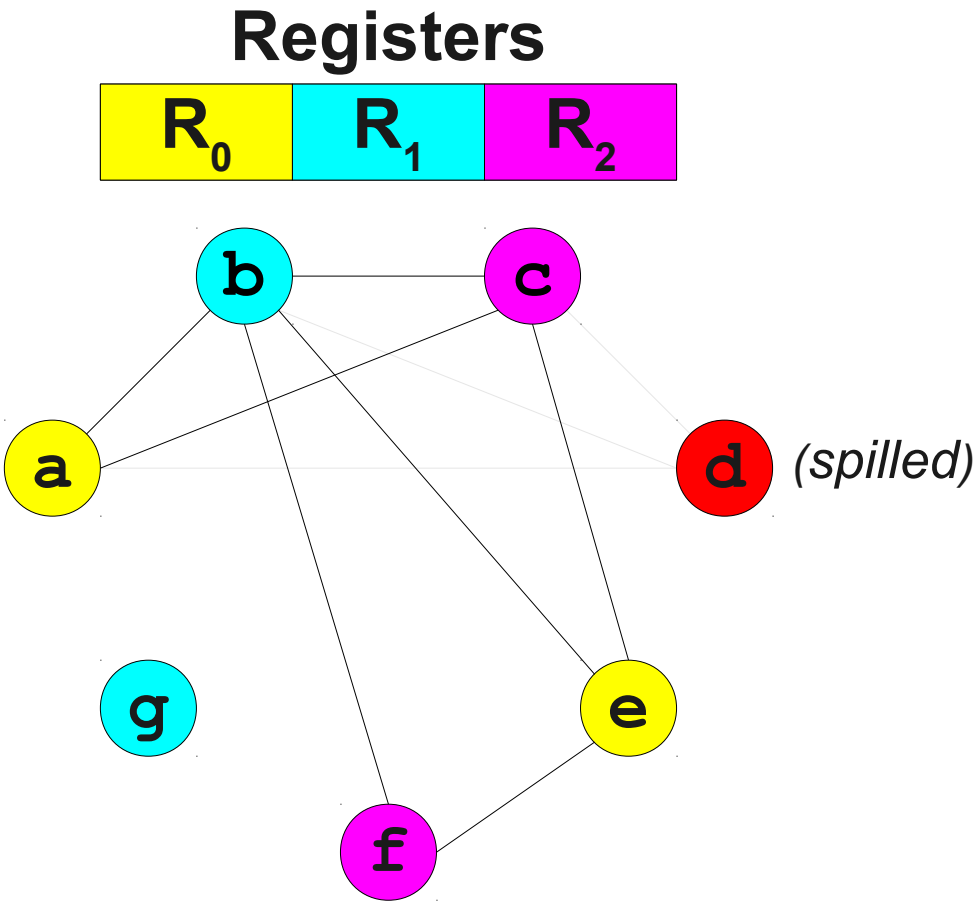
A Smarter Algorithm



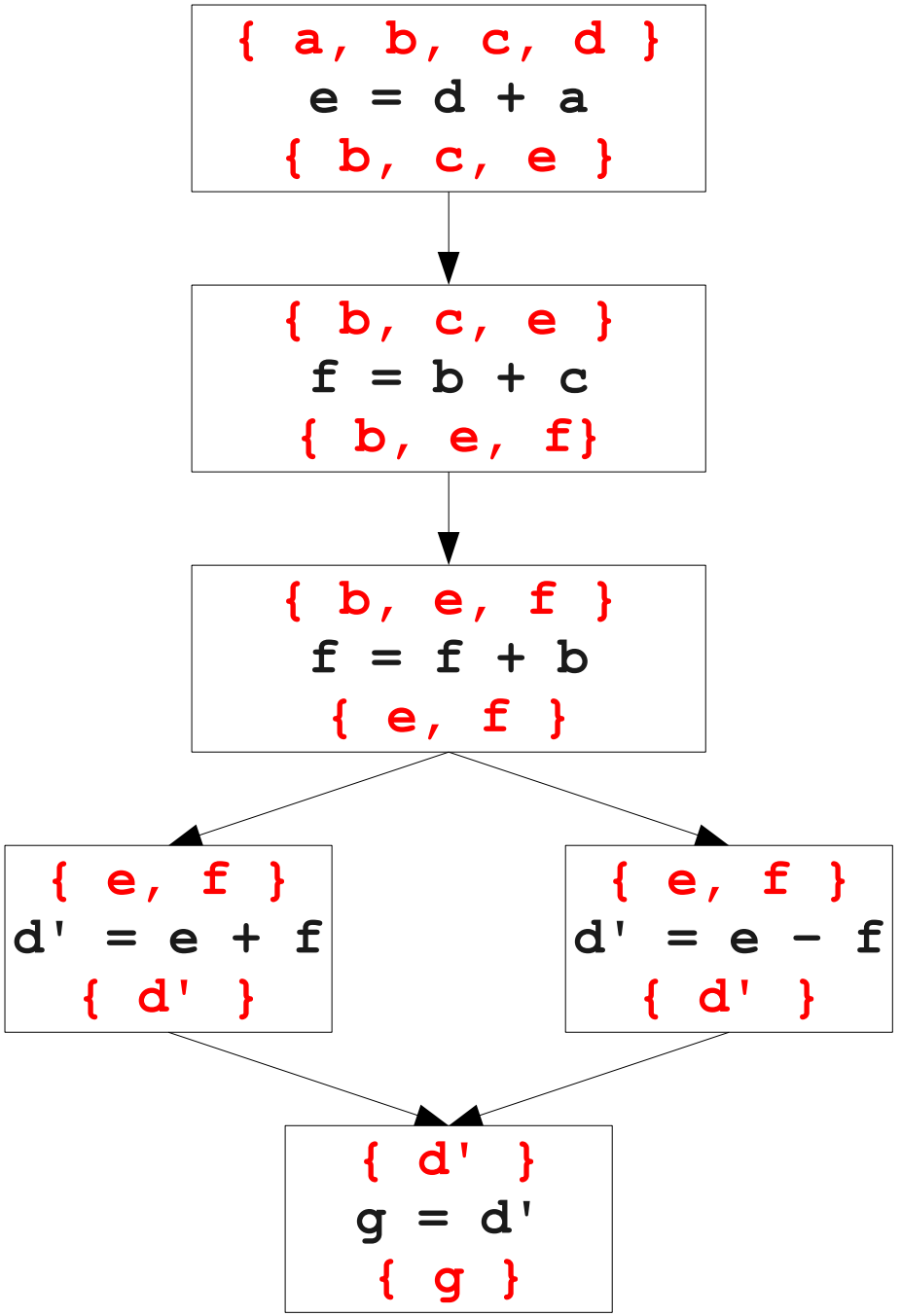


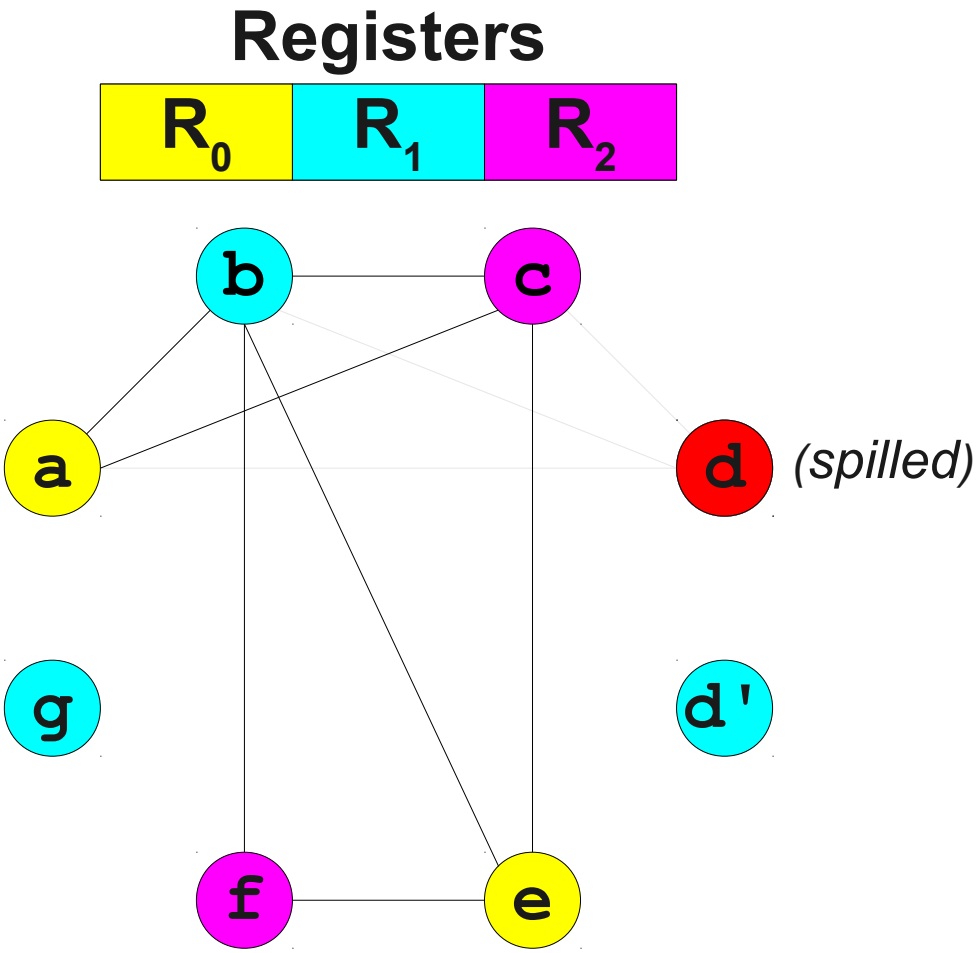
A Smarter Algorithm





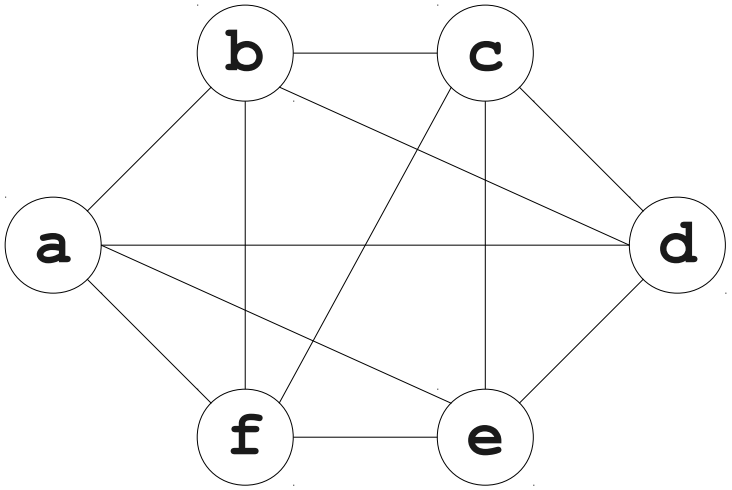
A Smarter Algorithm



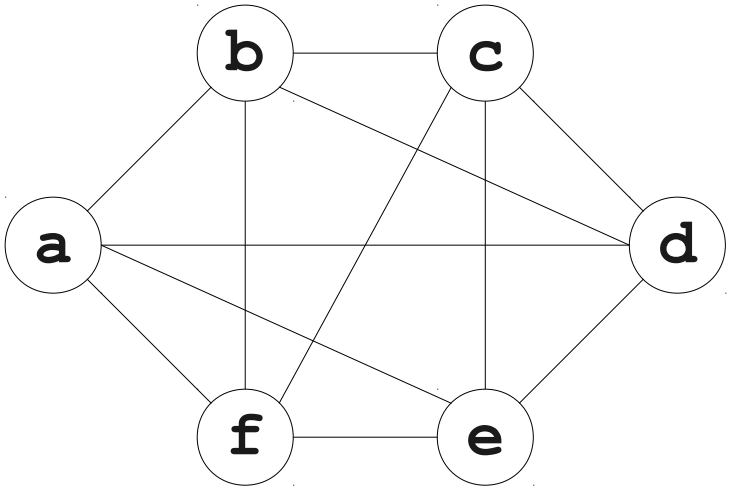


Another Example

Another Example

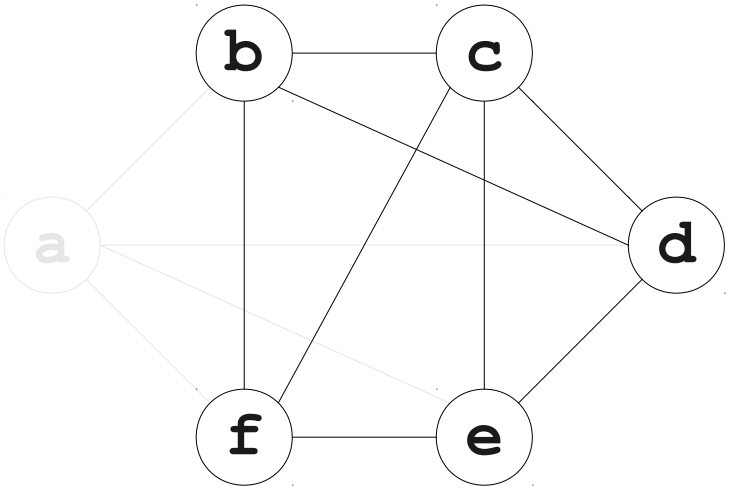


Another Example



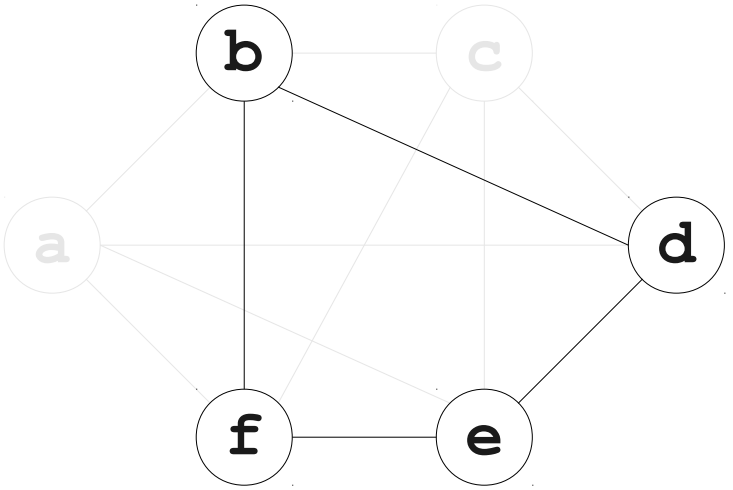
Registers

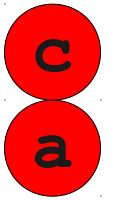
Another Example



Registers

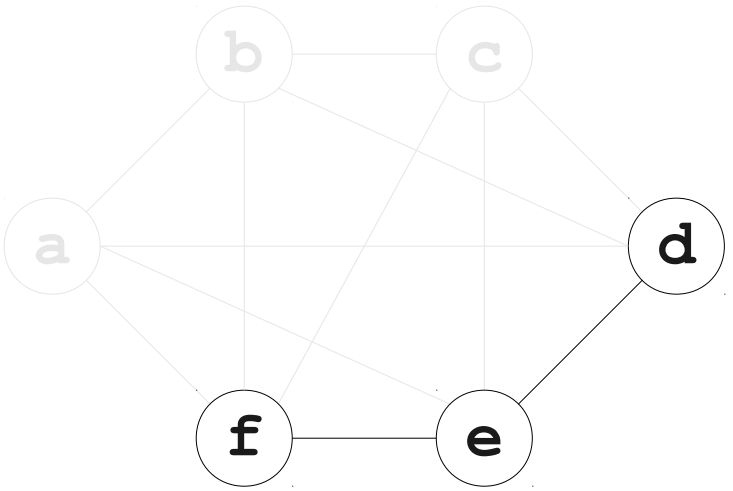
Another Example

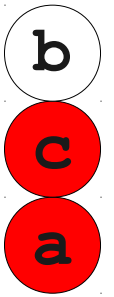




Registers

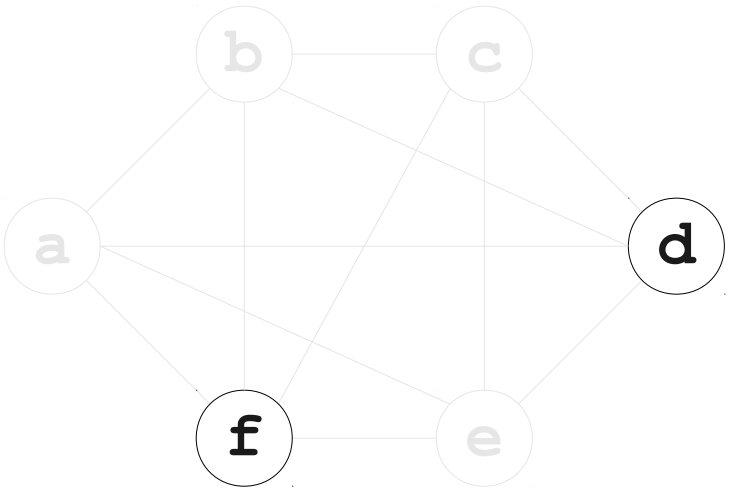
Another Example

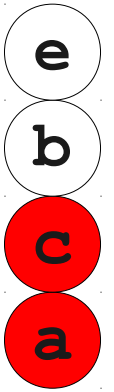




Registers

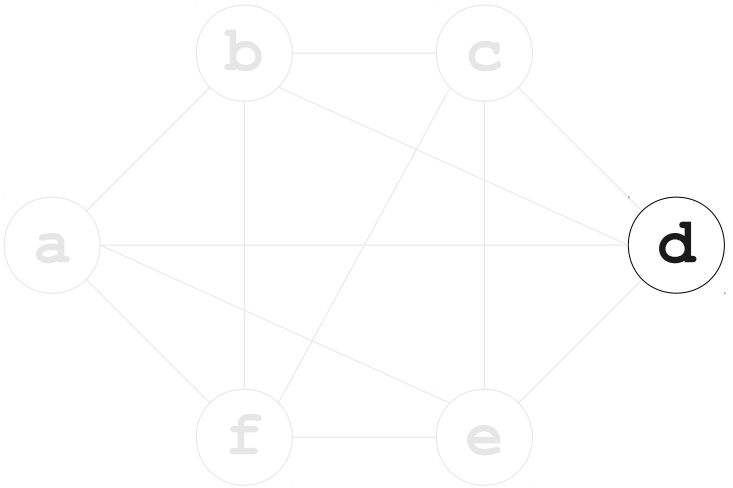
Another Example

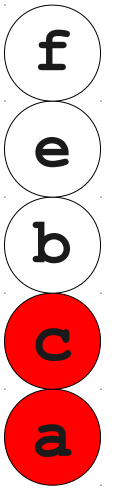




Registers

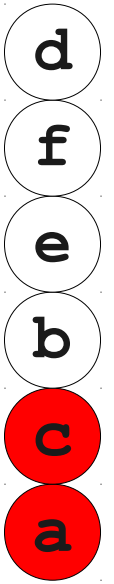
Another Example





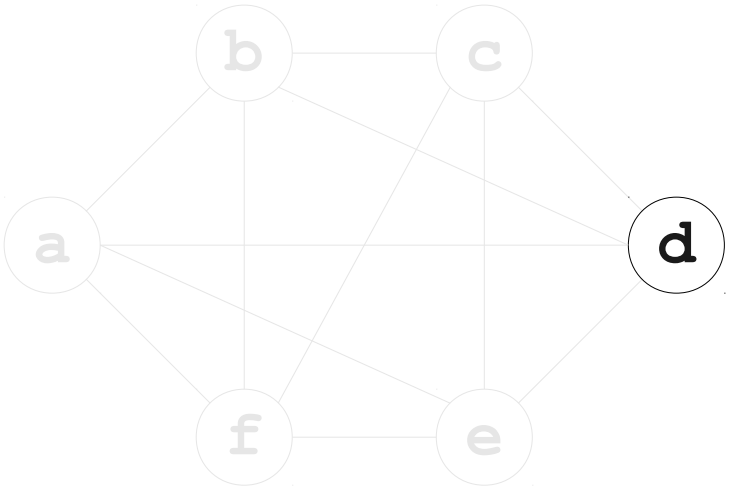
Registers

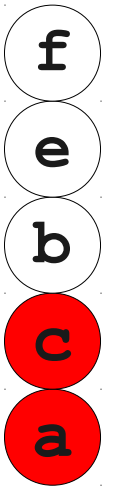
Another Example



Registers

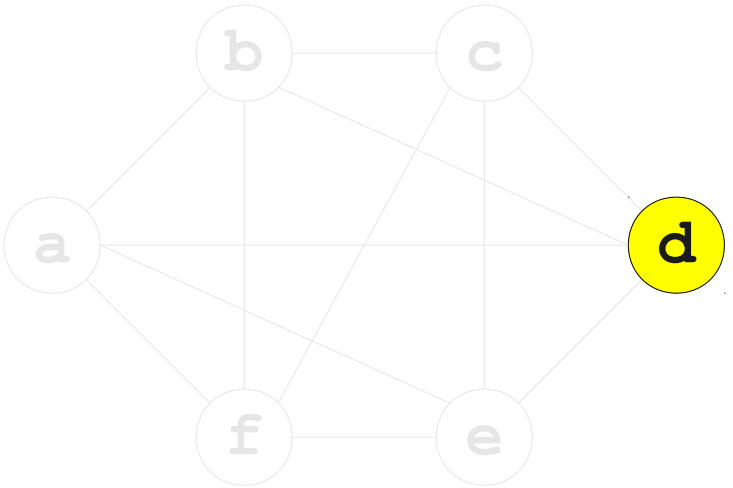
Another Example

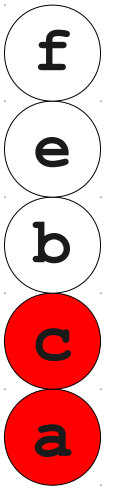




Registers

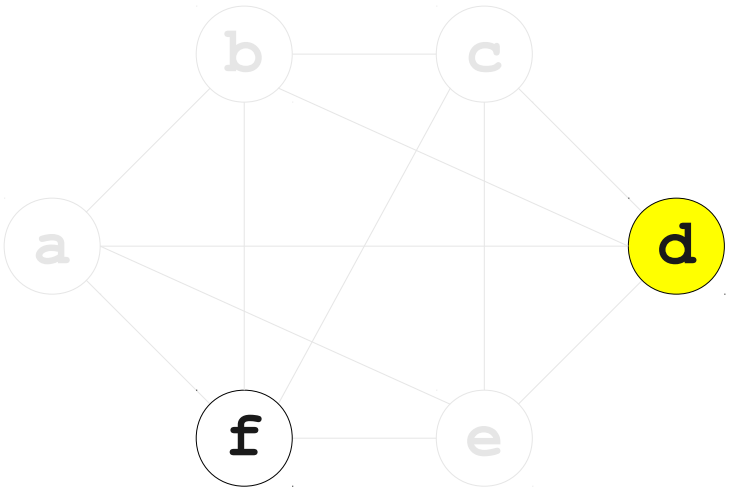
Another Example

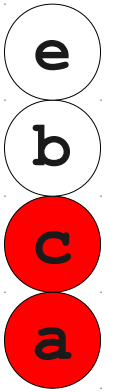




Registers

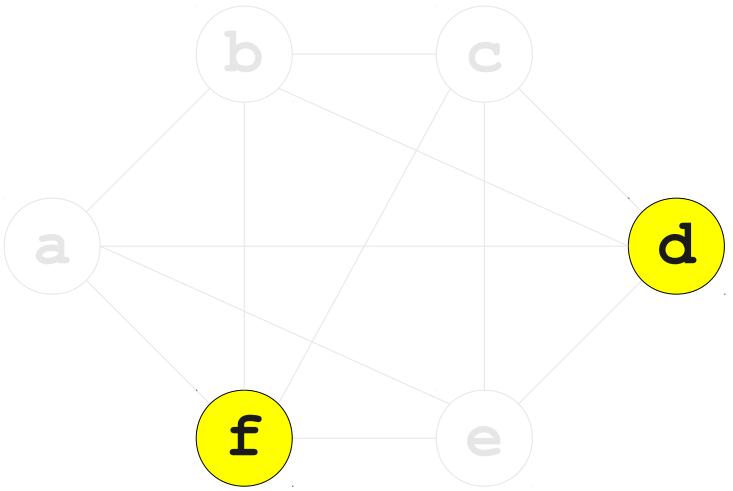
Another Example

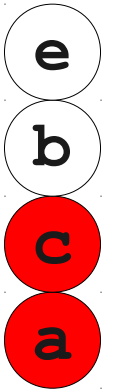




Registers

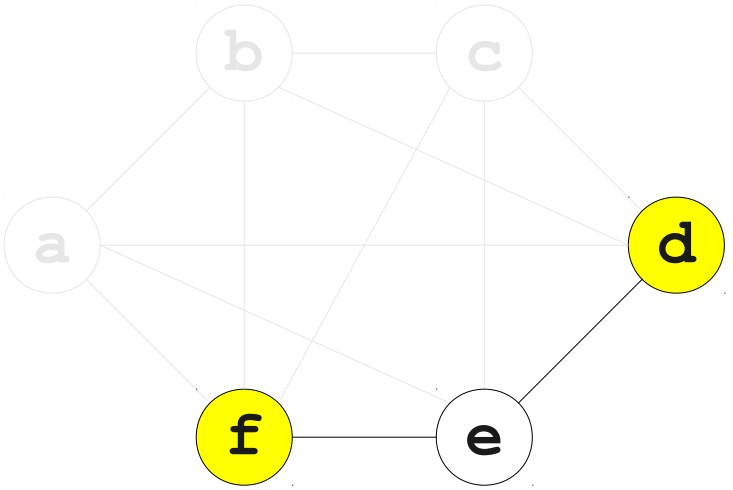
Another Example

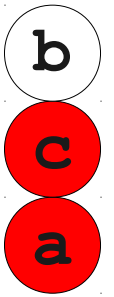




Registers

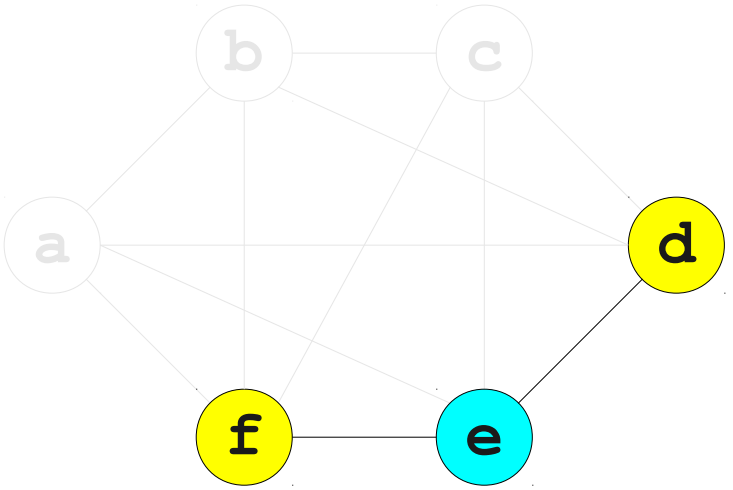
Another Example

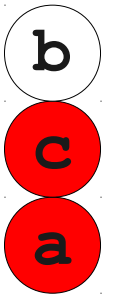




Registers

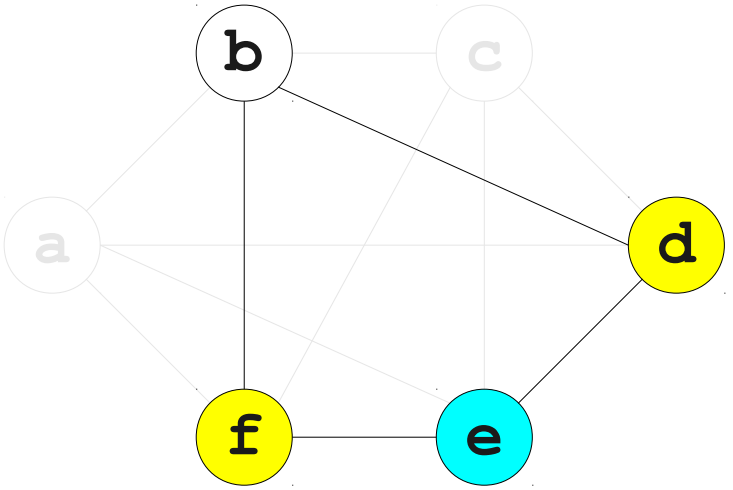
Another Example

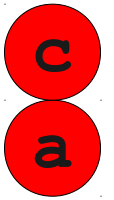




Registers

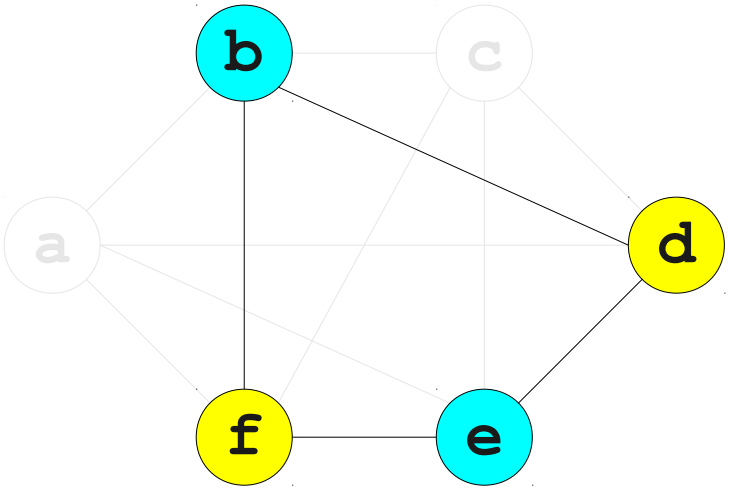
Another Example

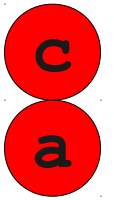




Registers

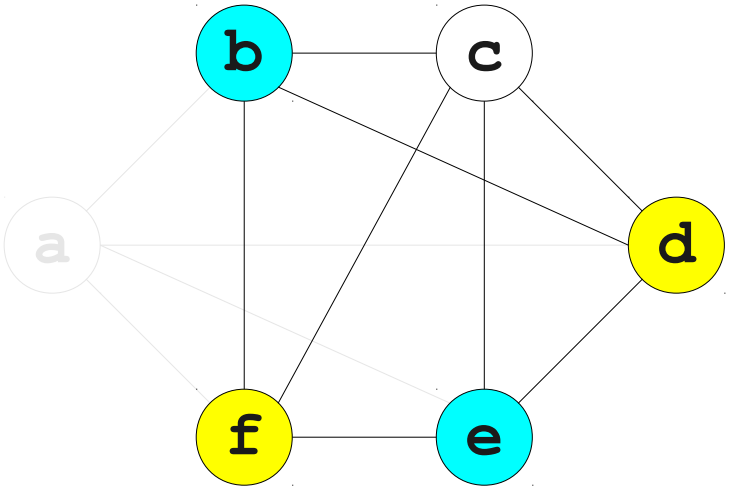
Another Example





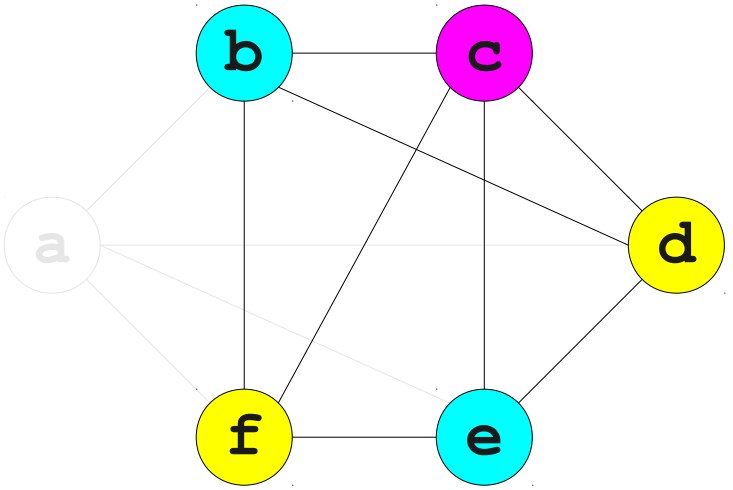
Registers

Another Example



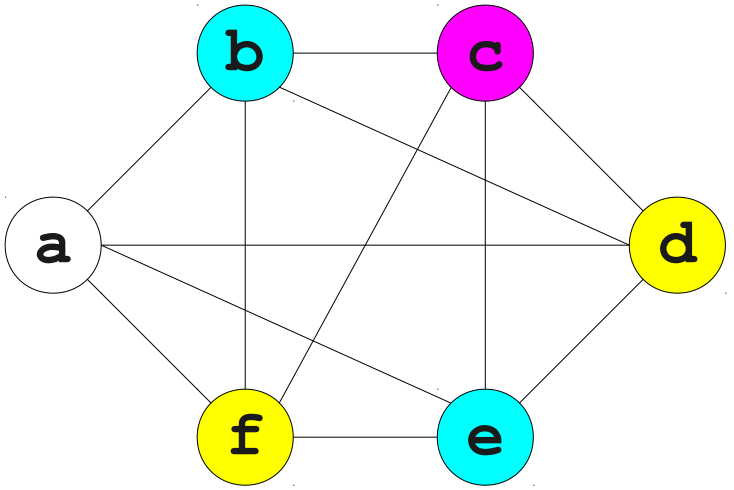
Registers

Another Example



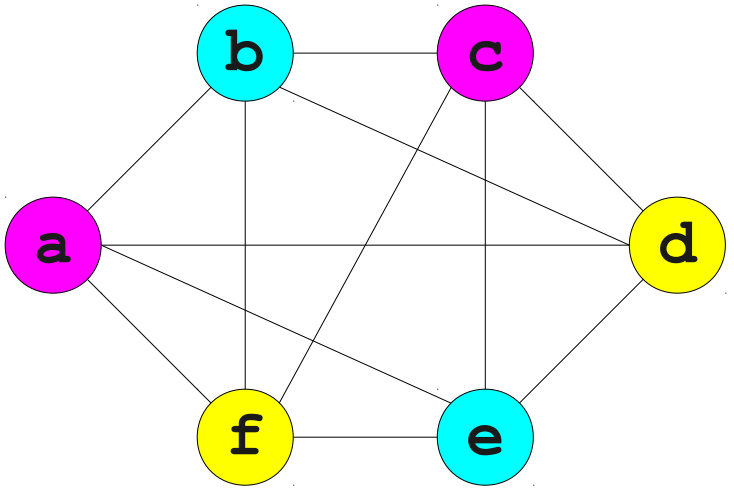
Registers

Another Example



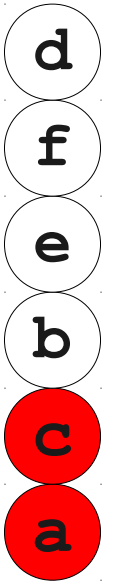
Registers

Another Example



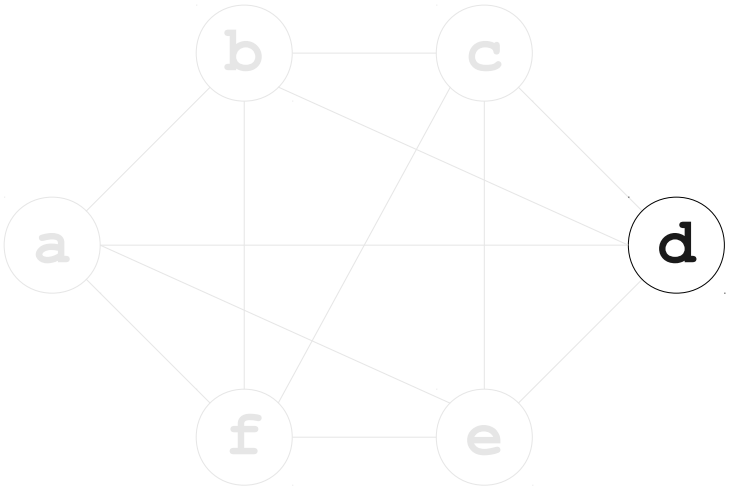
Registers

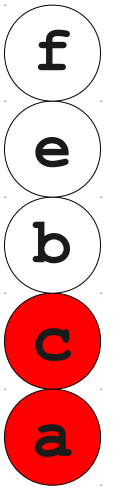
Another Example



Registers

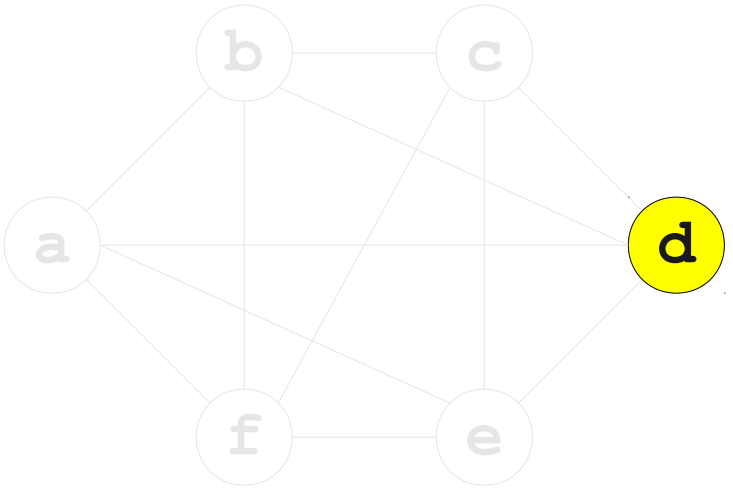
Another Example

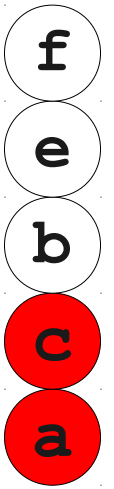




Registers

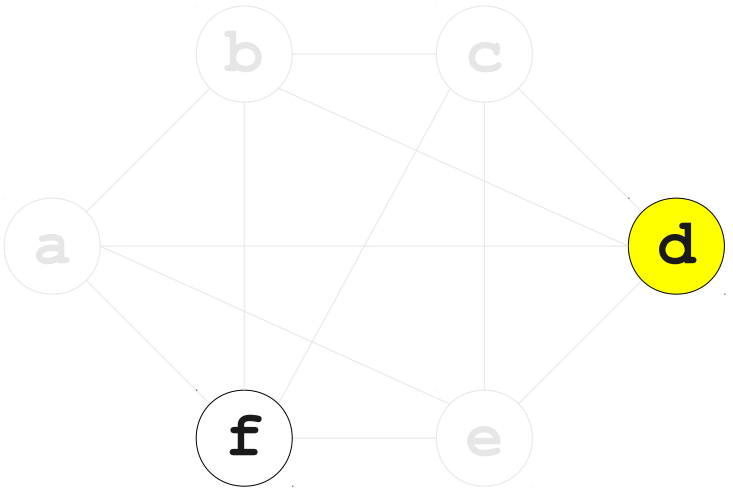
Another Example

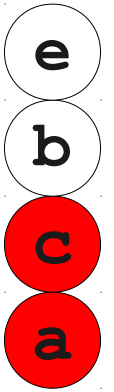




Registers

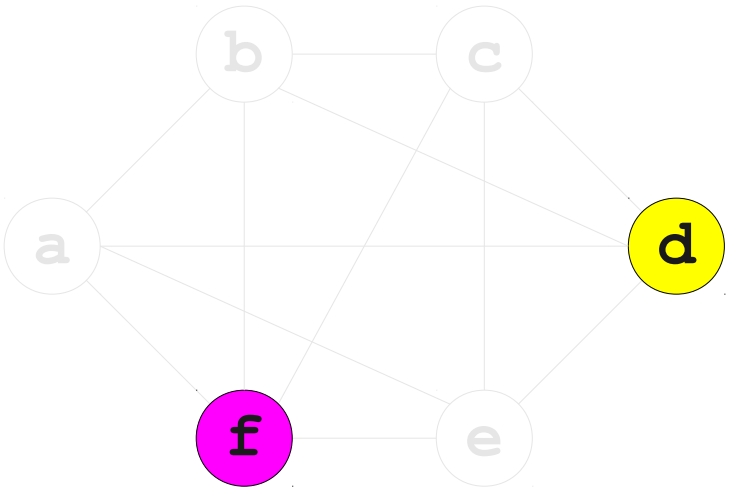
Another Example

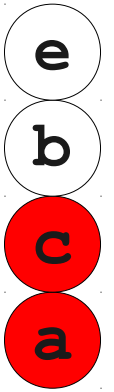




Registers

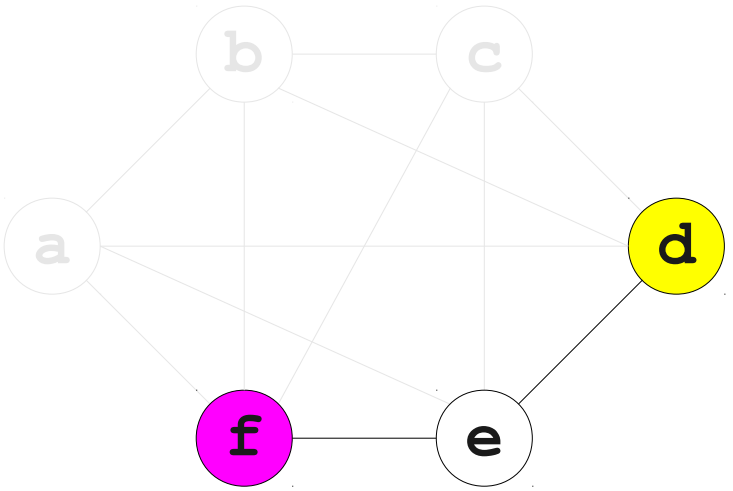
Another Example

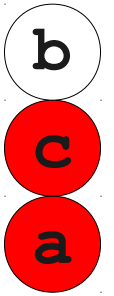




Registers

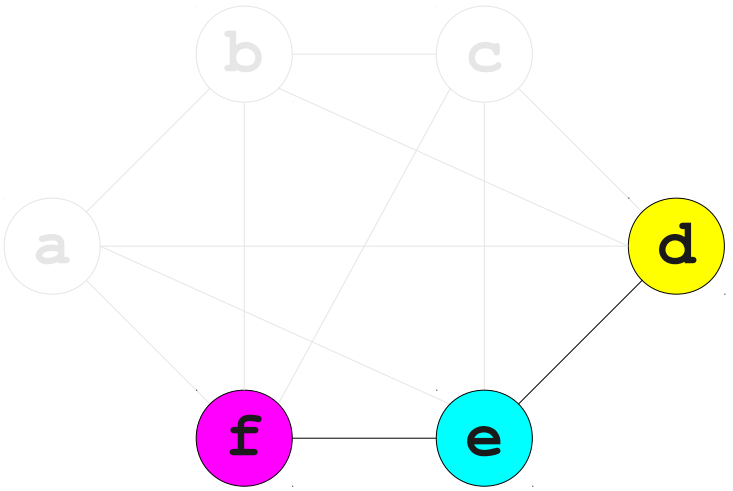
Another Example

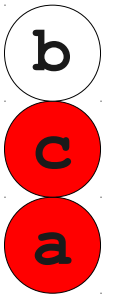




Registers

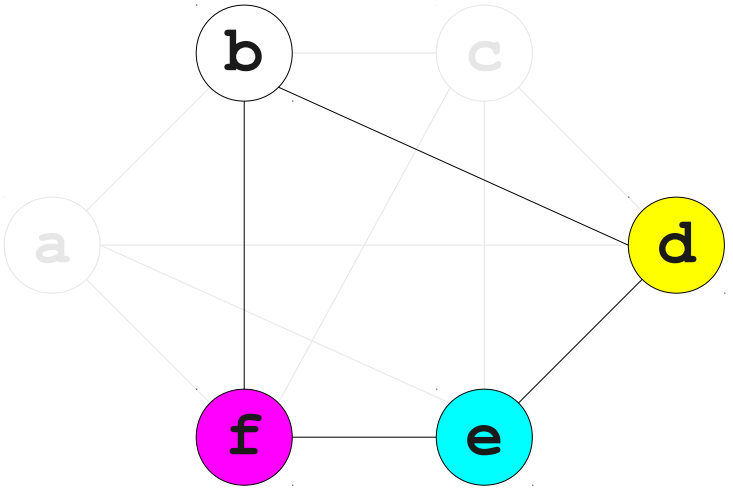
Another Example

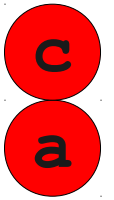




Registers

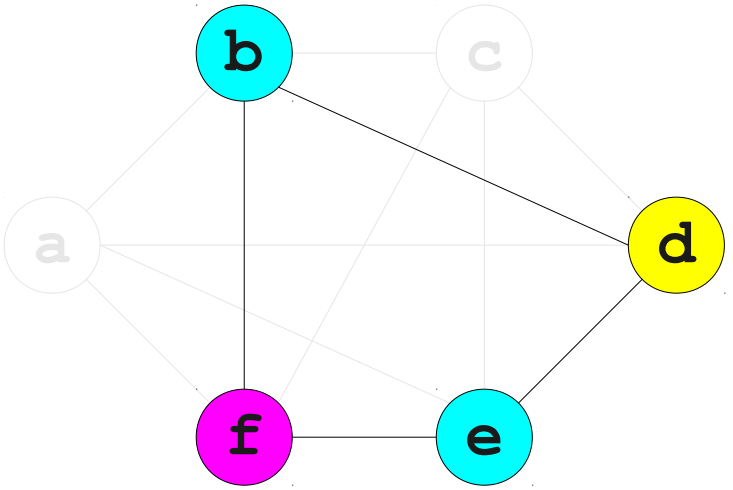
Another Example

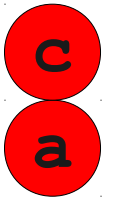




Registers

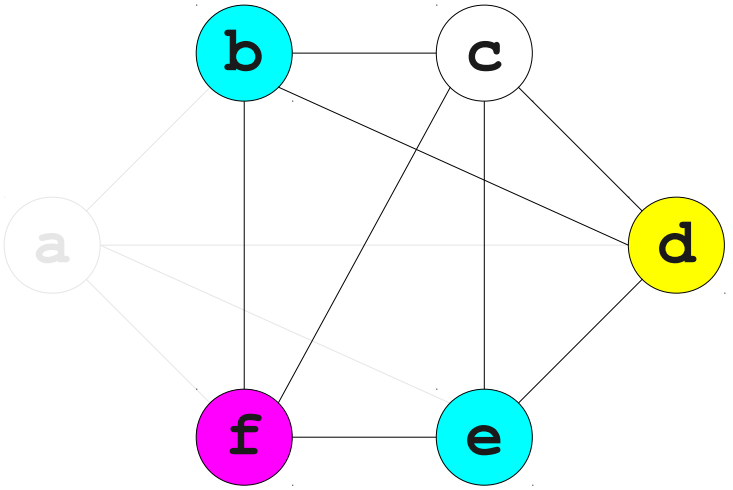
Another Example

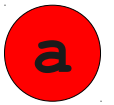




Registers

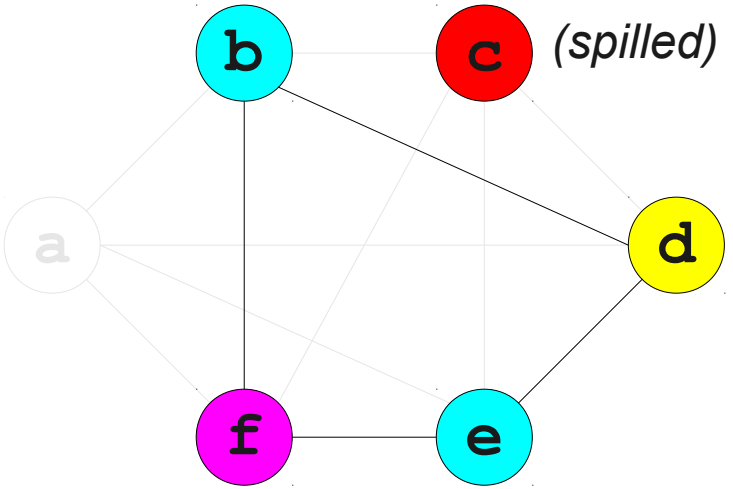
Another Example

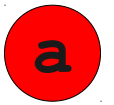




Registers

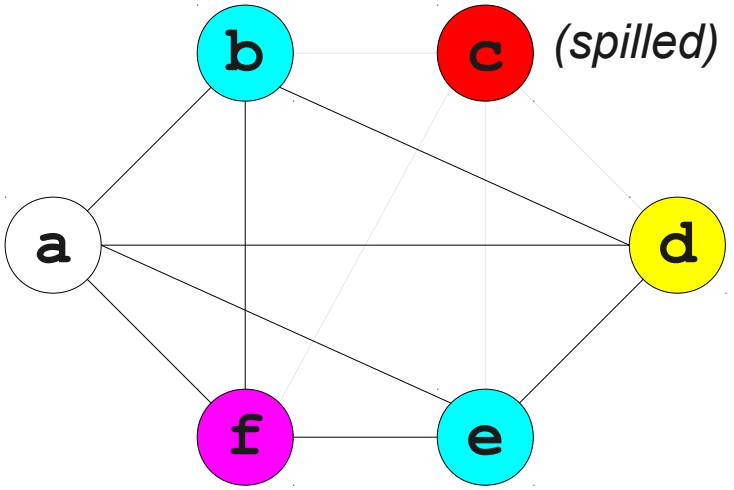
Another Example





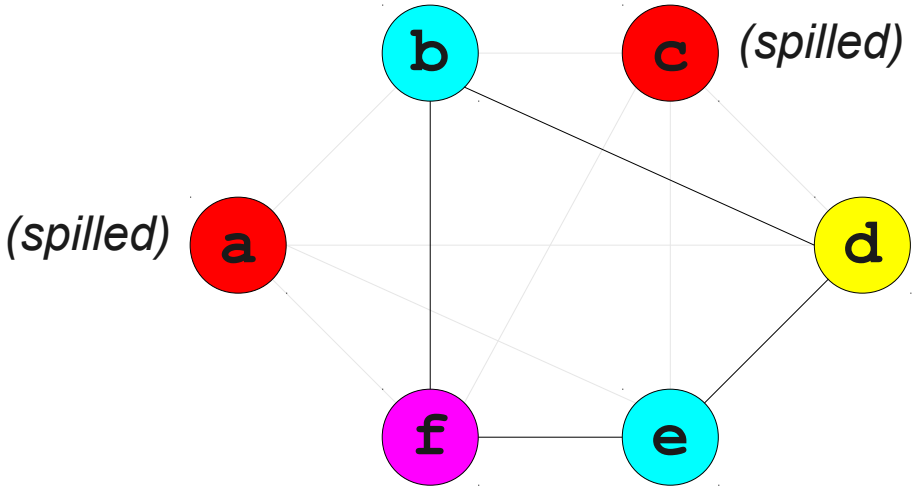
Registers

Another Example



Registers

Another Example



Registers

Chaitin's Algorithm

- Advantages:
  - For many control-flow graphs, finds an excellent assignment of variables to registers.
  - When distinguishing variables by use, produces a precise RIG.
  - Often used in production compilers like GCC.
- Disadvantages:
  - Core approach based on the NP-hard graph coloring problem.
  - Heuristic may produce pathologically worst-case assignments.

Correctness Proof Sketch

- No two variables live at some point are assigned the same register.
  - Forced by graph coloring.
- At any program point each variable is always in one location.
  - Automatic if we assign each variable one register.
  - Requires a few tricks if we separate by use case.

Improvements to the Algorithm

- Choose what to spill intelligently.
  - Use heuristics (least-commonly used, greatest improvement, etc.) to determine what to spill.
- Handle spilling intelligently.
  - When spilling a variable, recompute the RIG based on the spill and use a new coloring to find a register.

Summary of Register Allocation

- Critical step in all optimizing compilers.
- The **linear scan** algorithm uses **live intervals** to greedily assign variables to registers.
  - Often used in JIT compilers due to efficiency.
- **Chaitin's algorithm** uses the **register interference graph** (based on **live ranges**) and **graph coloring** to assign registers.
  - The basis for the technique used in GCC.

Next Time

- **Garbage Collection**
  - Reference Counting
  - Mark-and-Sweep
  - Stop-and-Copy
  - Incremental Collectors