Graph Coloring
Register Allocation

Register Allocation

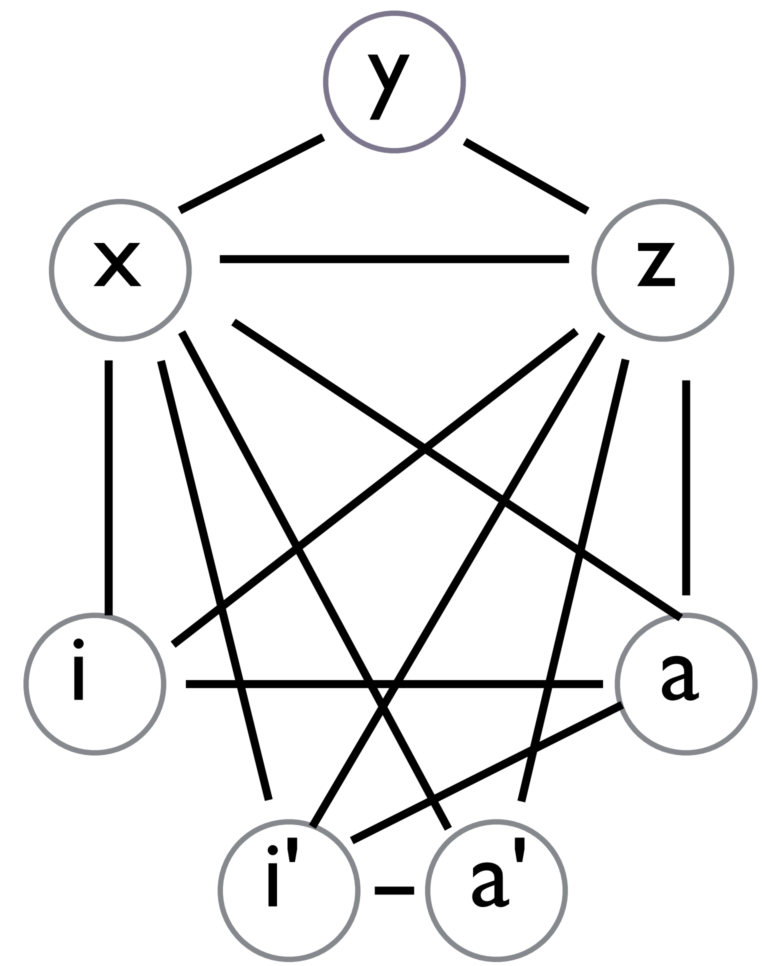
3(.5) Steps

1. **Liveness analysis**: identify when each variable's value is needed in the program
2. **Conflict analysis**: identify which variables interfere with each other
3. **Graph Coloring**: assign variables to registers so that interfering registers are assigned different registers.
    1. Spilling: if necessary, assign some variables to stack slots

Example

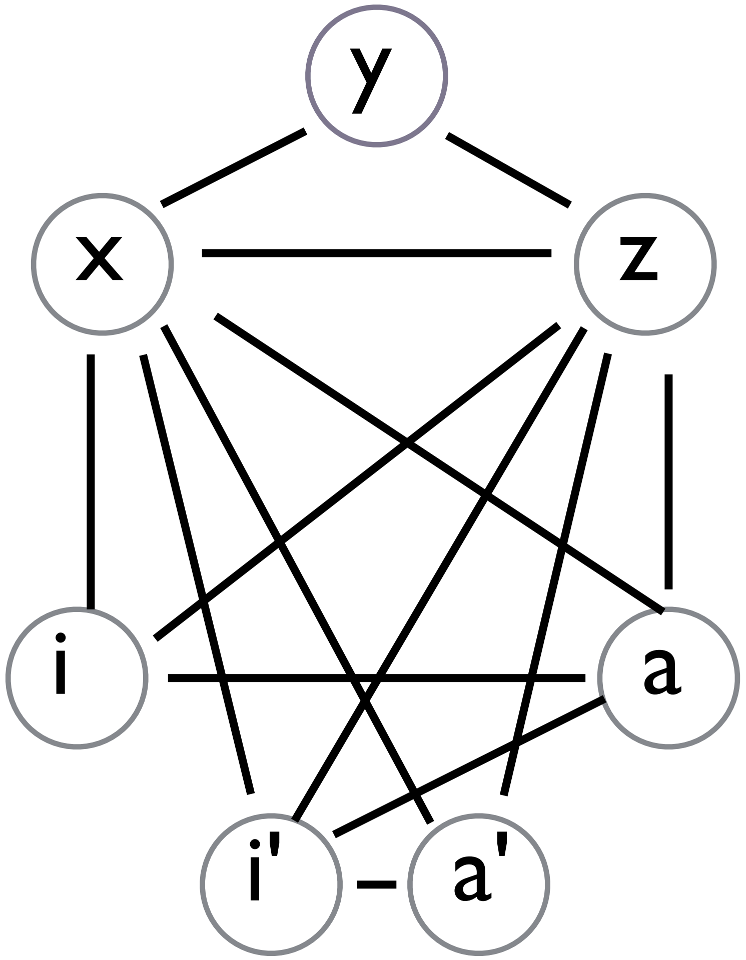
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

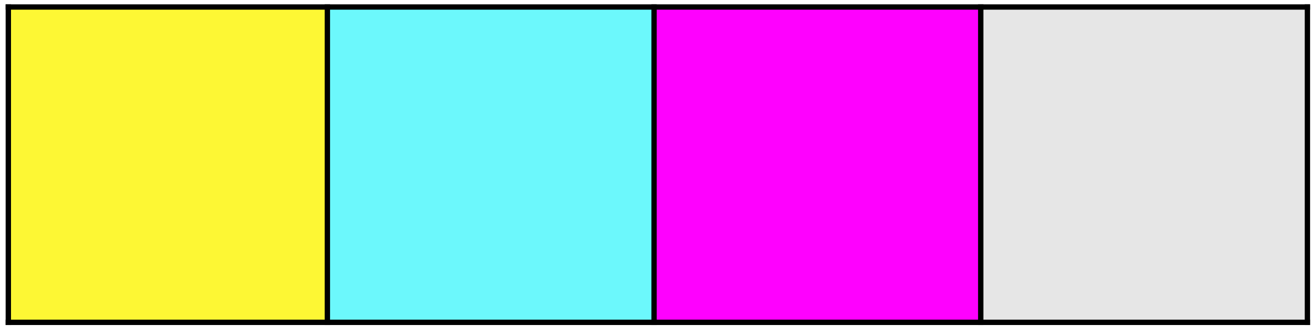
Interference Graph



Example

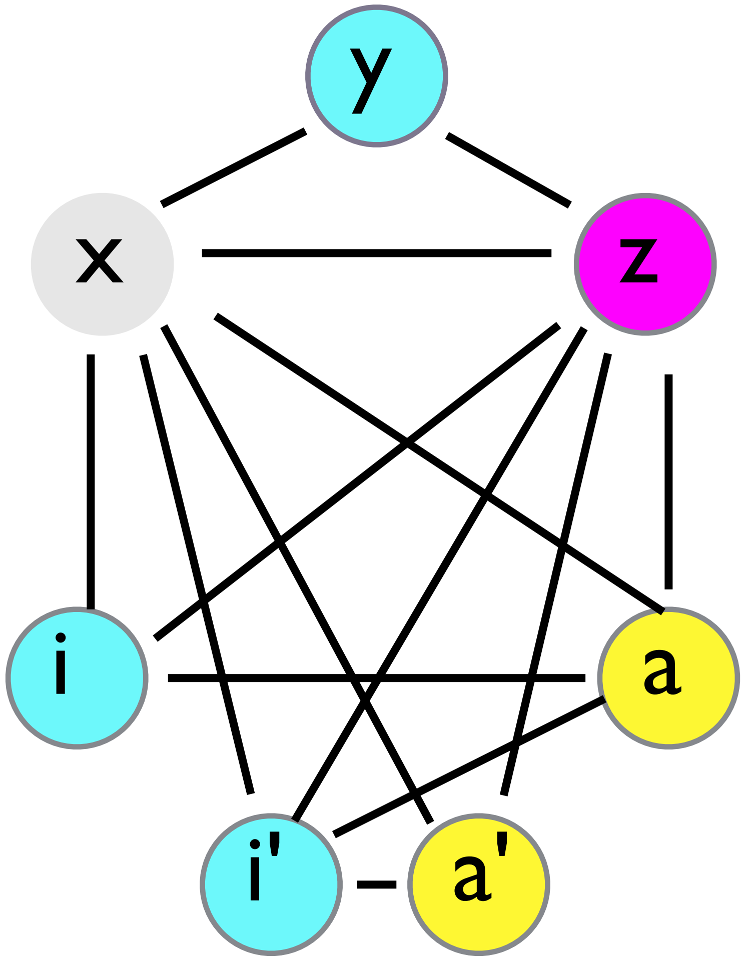
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

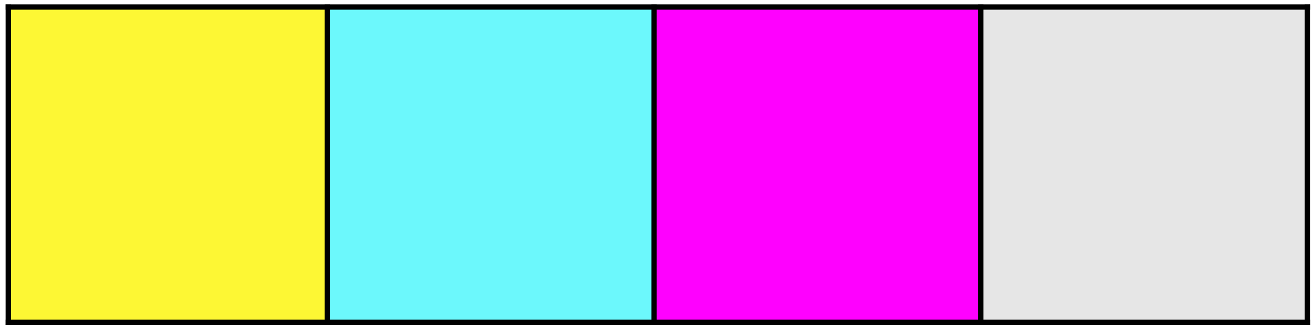




Example

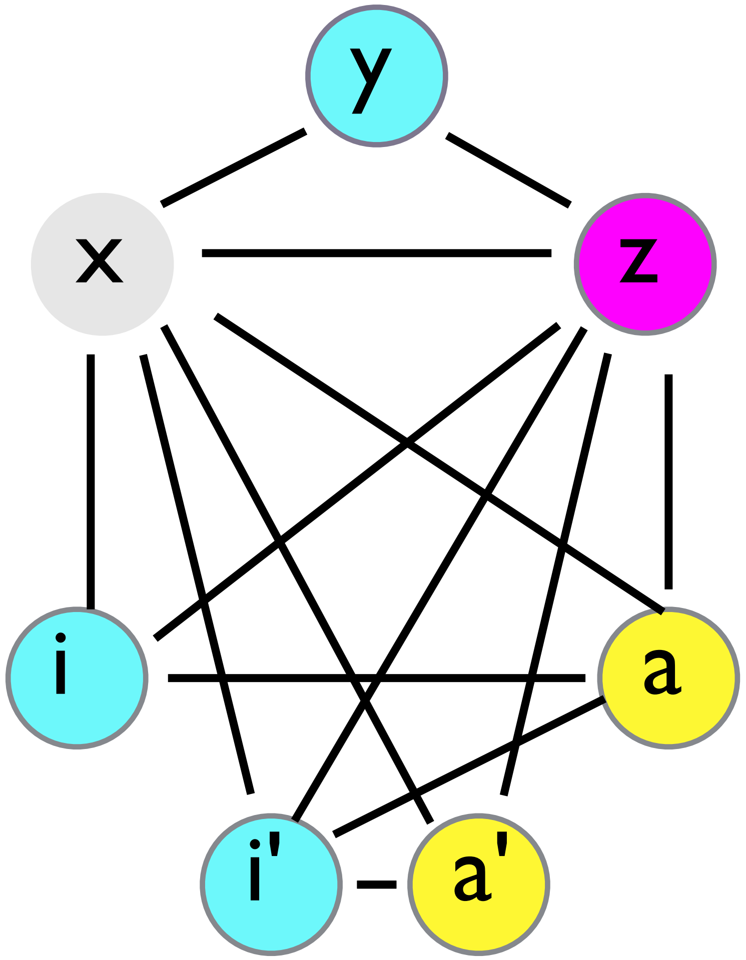
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```





Example

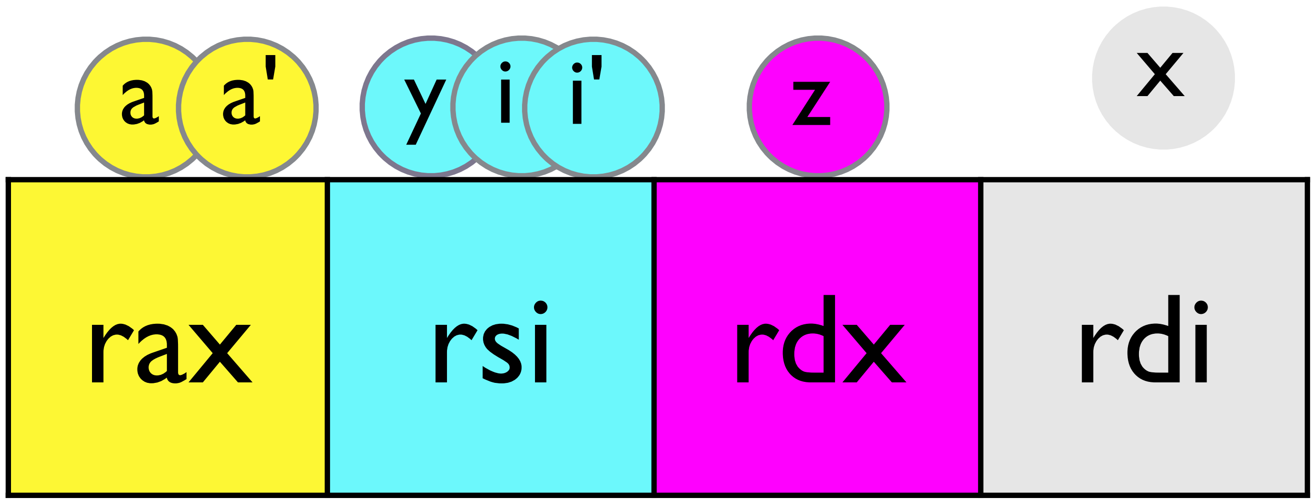
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



rax	rsi	rdx	rdi

Example

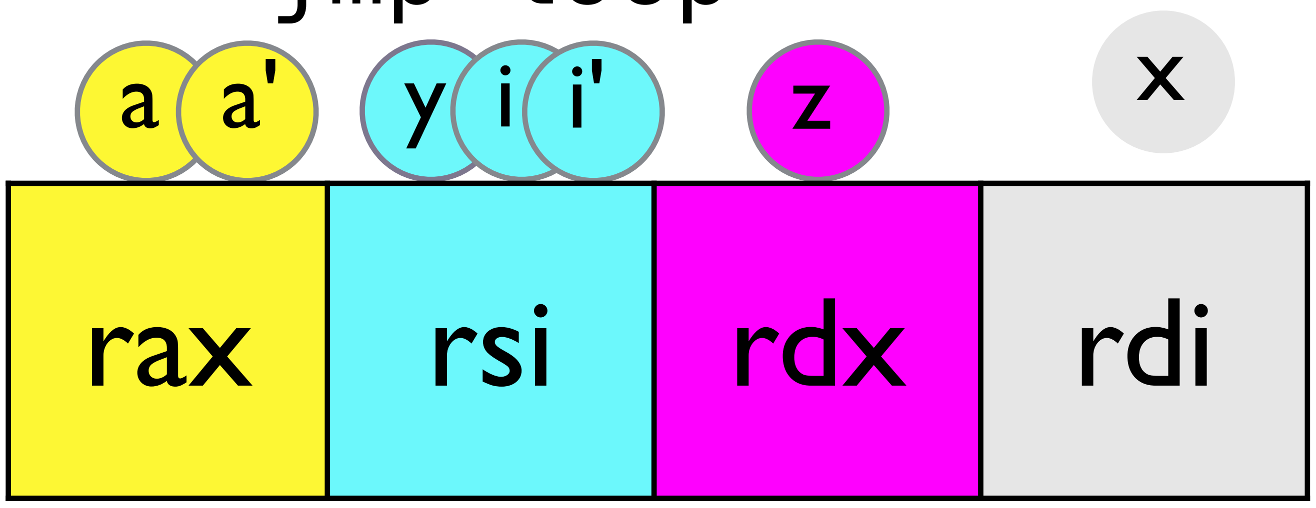
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



Example

```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

```
f:
  mov rax, 0
loop:
  cmp rsi, 0
  jne els
  imul rax, rdx
  ret
els:
  sub rsi, 1
  add rcx, rdi
  jmp loop
```



Graph Coloring Register Allocation

Given our register conflict graph, want to assign a register to each variable so that no interfering variables are assigned the same register.

Graph Coloring Register Allocation

Given our register conflict graph, want to assign a register to each variable so that no interfering variables are assigned the same register.

Equivalent to graph coloring of the interference graph

- think of each register as a "color" and we want to paint each node so that no adjacent nodes are the same color.

Graph Coloring Register Allocation

Given our register conflict graph, want to assign a register to each variable so that no interfering variables are assigned the same register.

Equivalent to graph coloring of the interference graph

- think of each register as a "color" and we want to paint each node so that no adjacent nodes are the same color.

Efficient algorithm for graph coloring -> efficient algorithm for graph coloring!

Graph Coloring is Hard

Determining a whether a graph is k-colorable is NP-complete for k > 2.

- So no polytime algorithm is known

Graph Coloring is Hard

Determining a whether a graph is k-colorable is NP-complete for k > 2.

- So no polytime algorithm is known

Does that mean register allocation is NP-hard?

Is Register Allocation Hard?

Is Register Allocation Hard?

Chaitin et al, "Register allocation via coloring", *Computer Languages* 1981

- Showed that the register allocation problem for a language with assignments and arbitrary control flow (goto) is equivalent to graph coloring

Is Register Allocation Hard?

Chaitin et al, "Register allocation via coloring", *Computer Languages* 1981

- Showed that the register allocation problem for a language with assignments and arbitrary control flow (goto) is equivalent to graph coloring
- every graph arises as the interference graph of some program

So register allocation of their language is NP complete.

Is Register Allocation Hard?

Chaitin et al, "Register allocation via coloring", *Computer Languages* 1981

- Showed that the register allocation problem for a language with assignments and arbitrary control flow (goto) is equivalent to graph coloring
- every graph arises as the interference graph of some program

So register allocation of their language is NP complete.

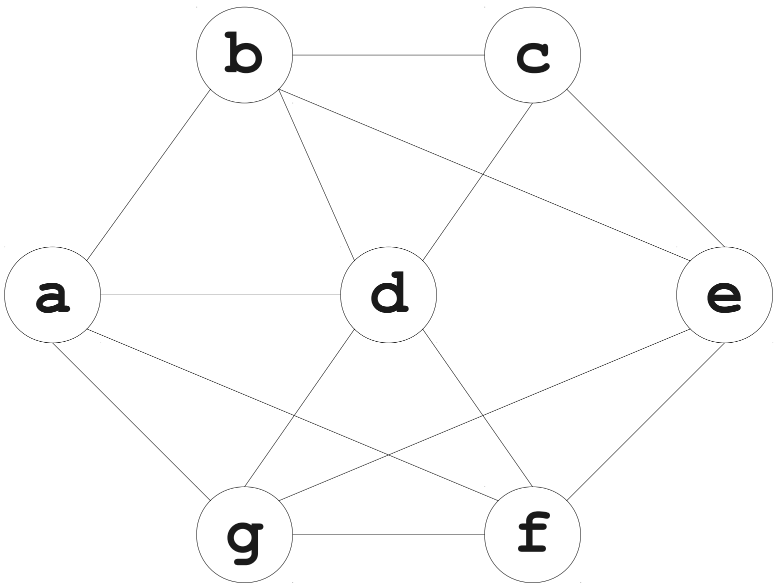
- But our programs are more restrictive: Functional/SSA form...we'll come back to this

Chaitin's Algorithm

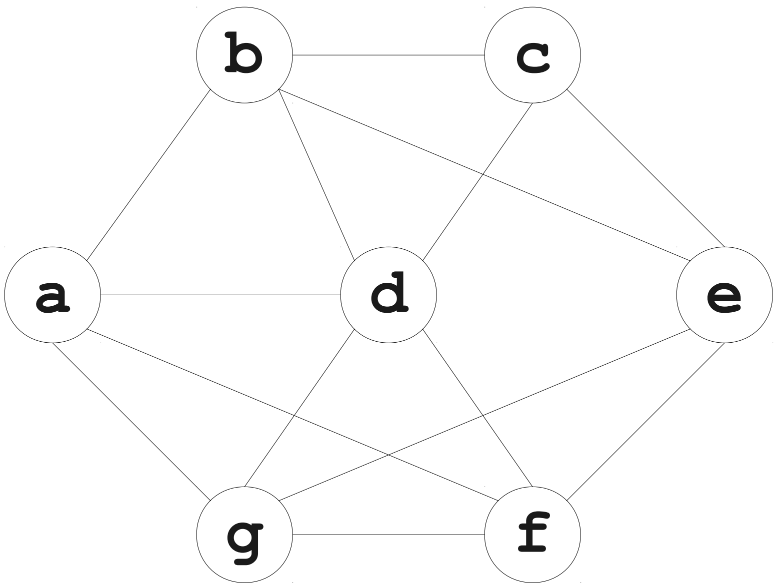
- Intuition:
  - Suppose we are trying to k-color a graph and find a node with fewer than k edges.
  - If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in.
  - Reason: With fewer than k neighbors, some color must be left over.
- Algorithm: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



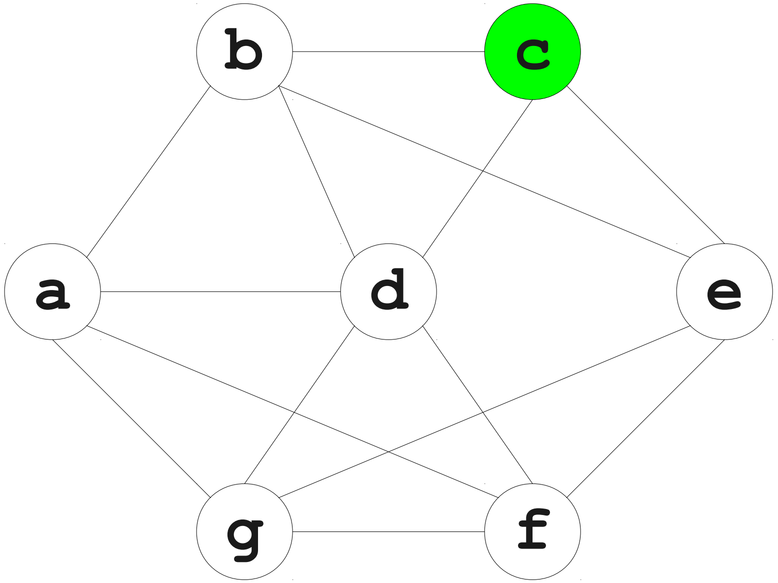
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

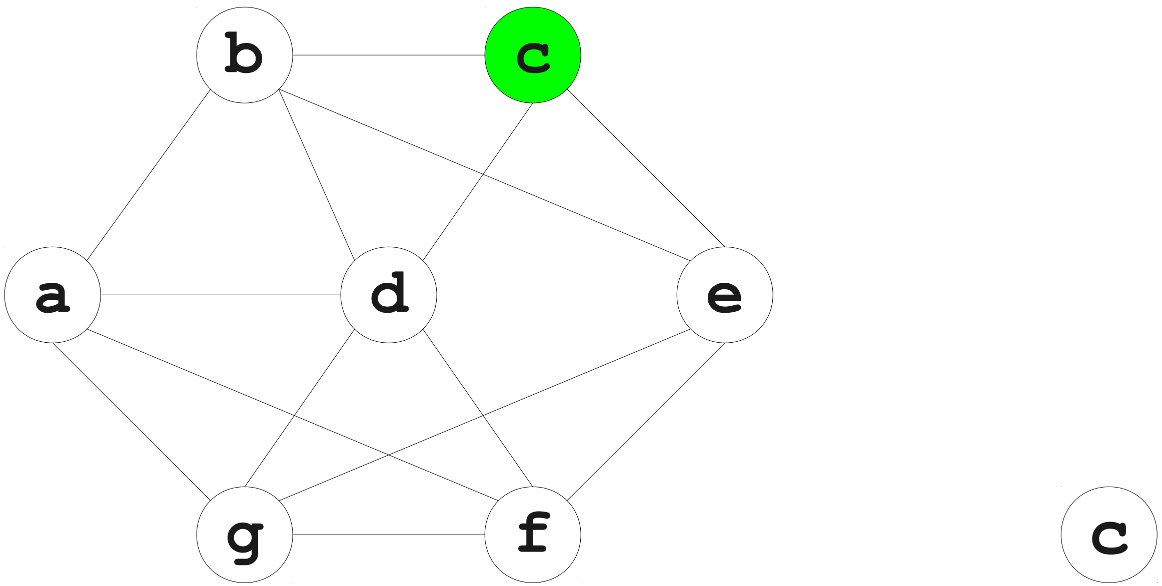
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

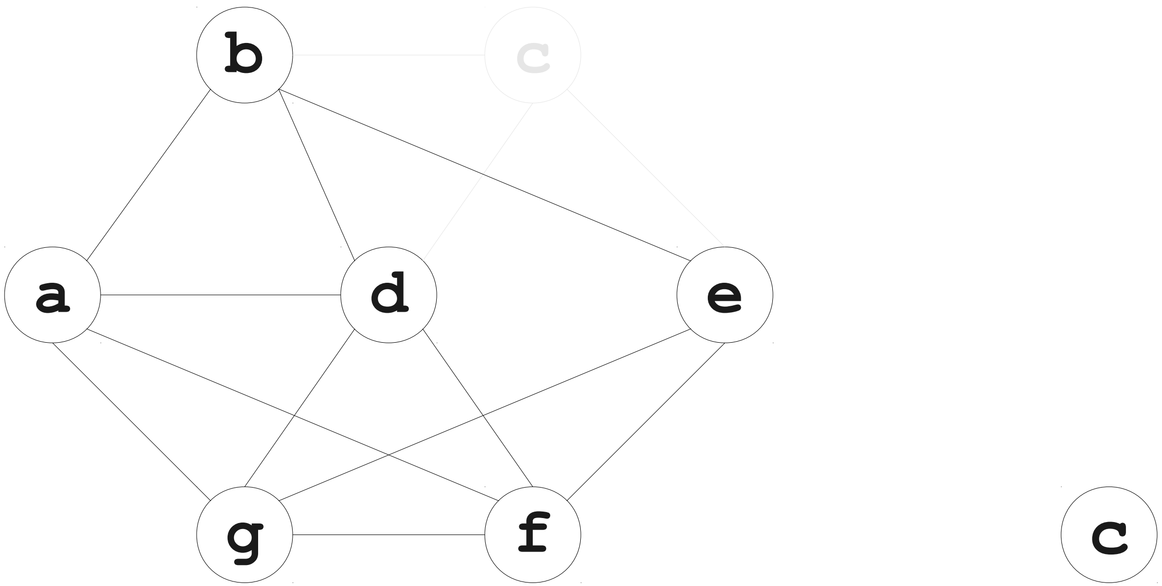
Chaitin's Algorithm



Registers

R$_0$	R$_1$	R$_2$	R$_3$

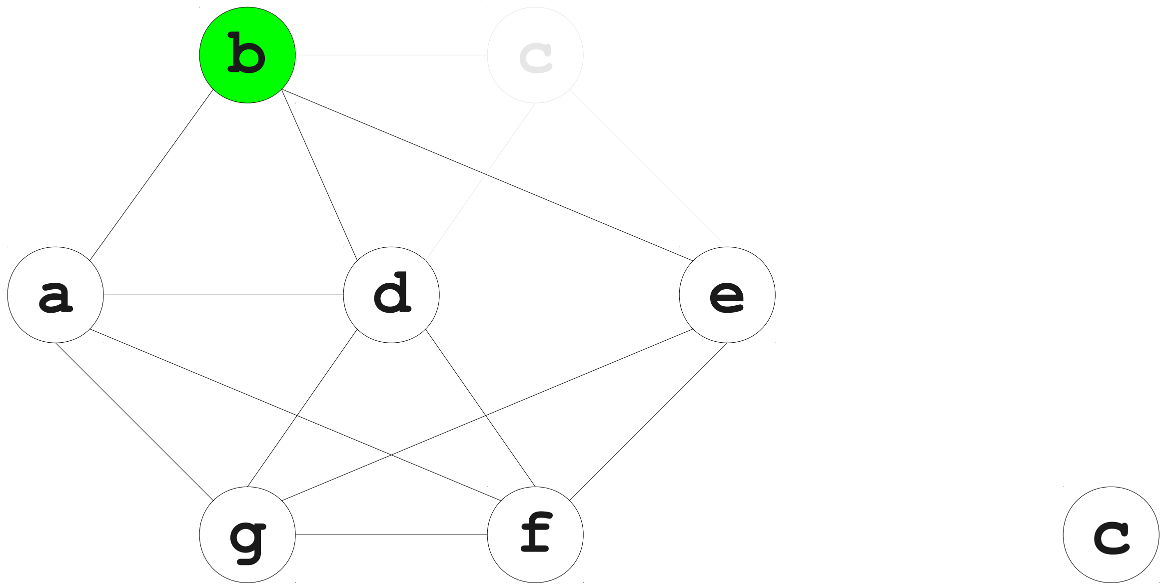
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

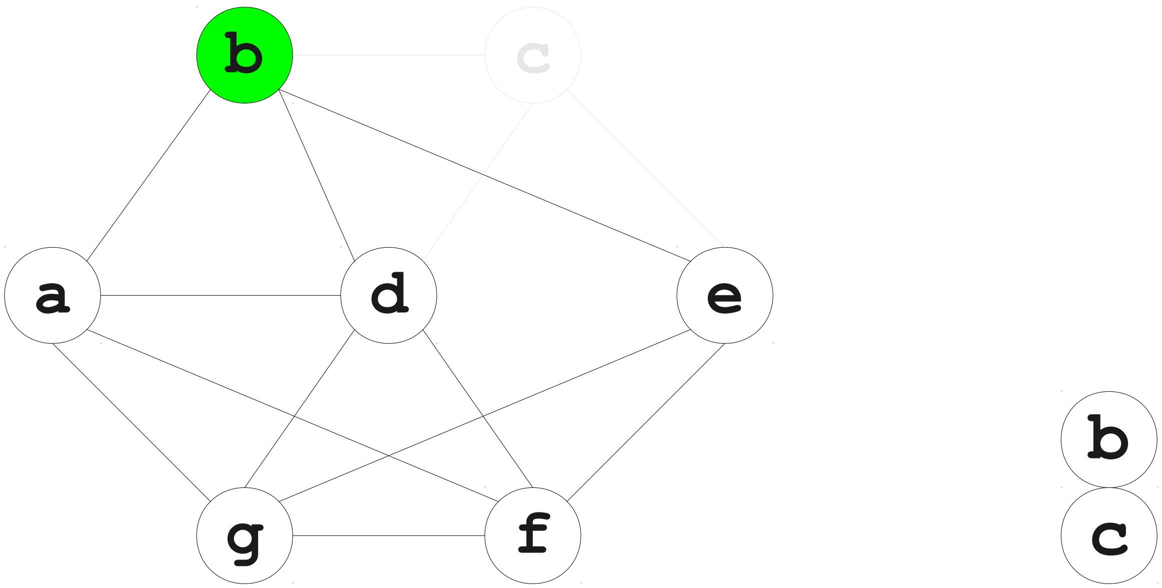
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

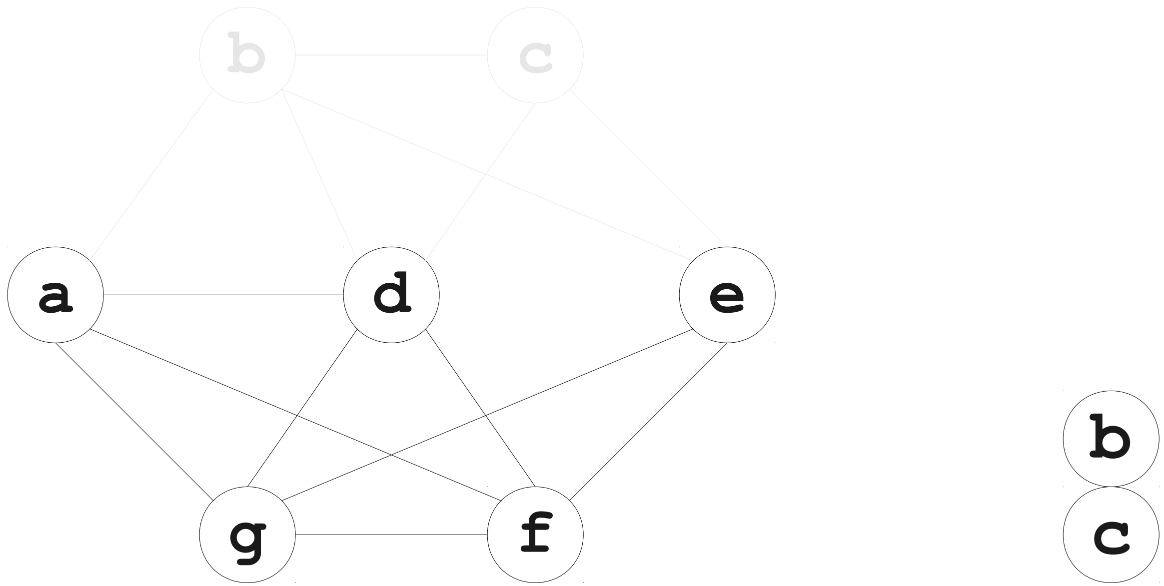
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

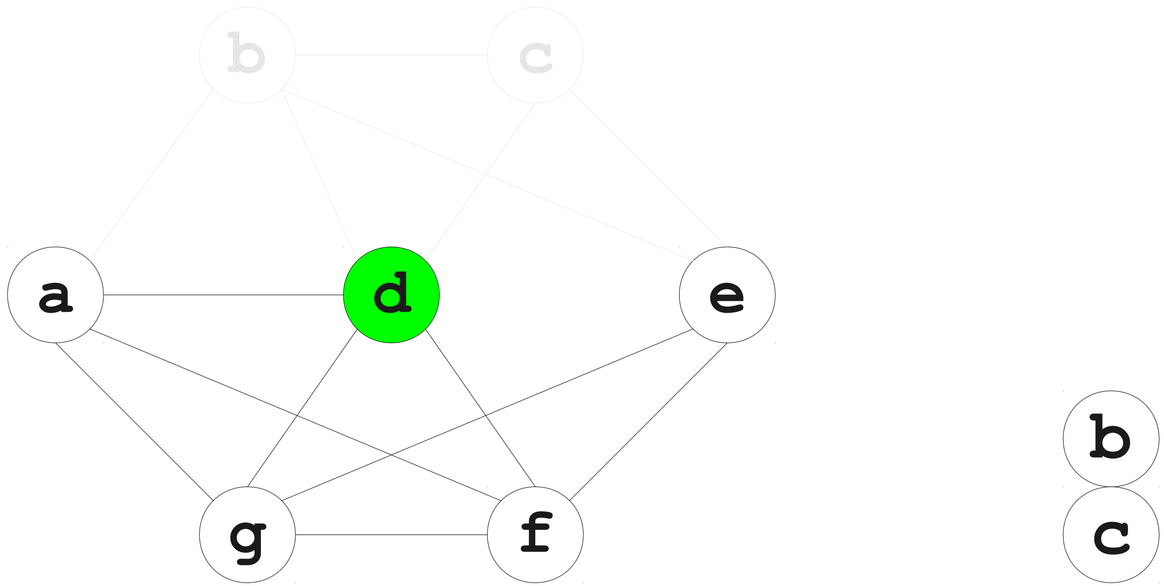
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

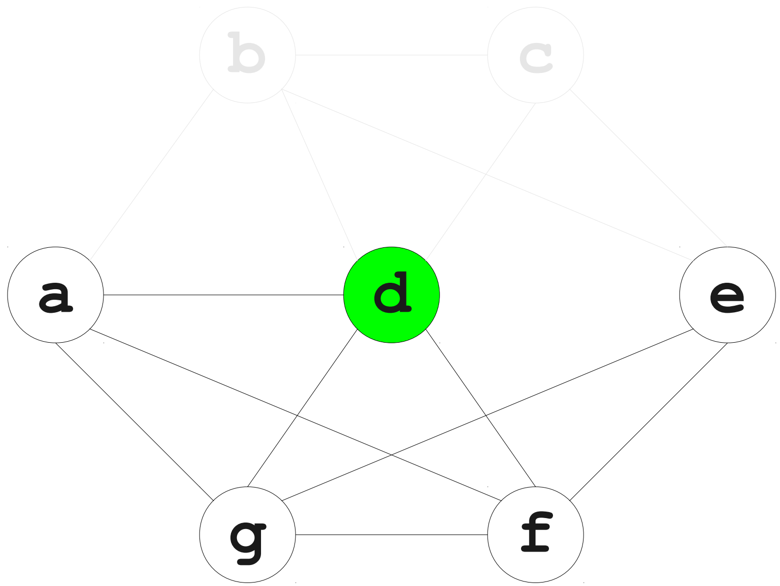
Chaitin's Algorithm

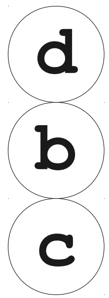


Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm

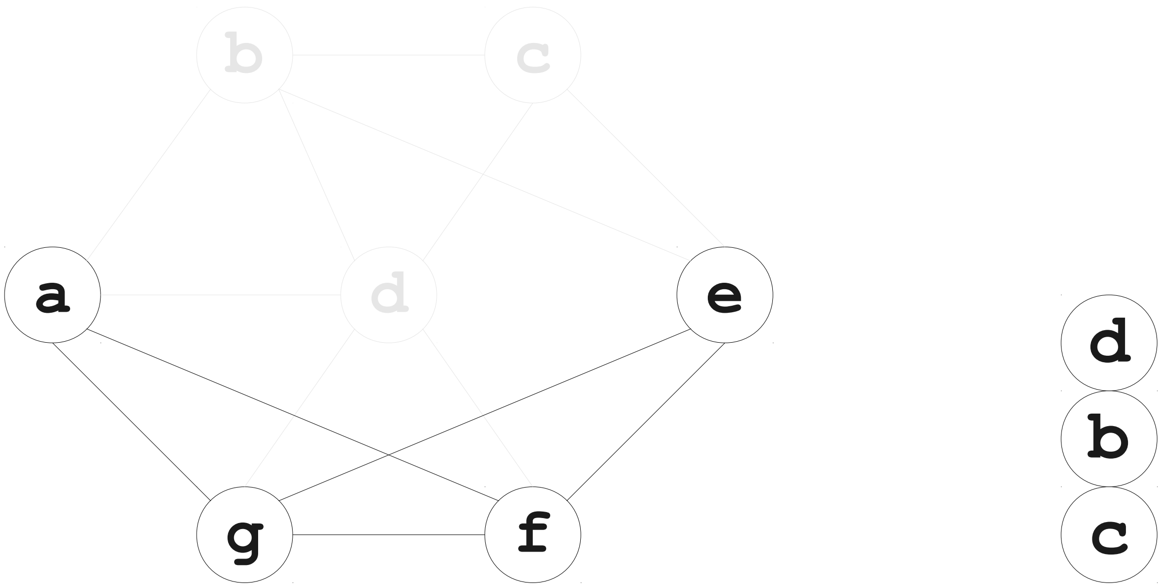




Registers

R_0	R_1	R_2	R_3

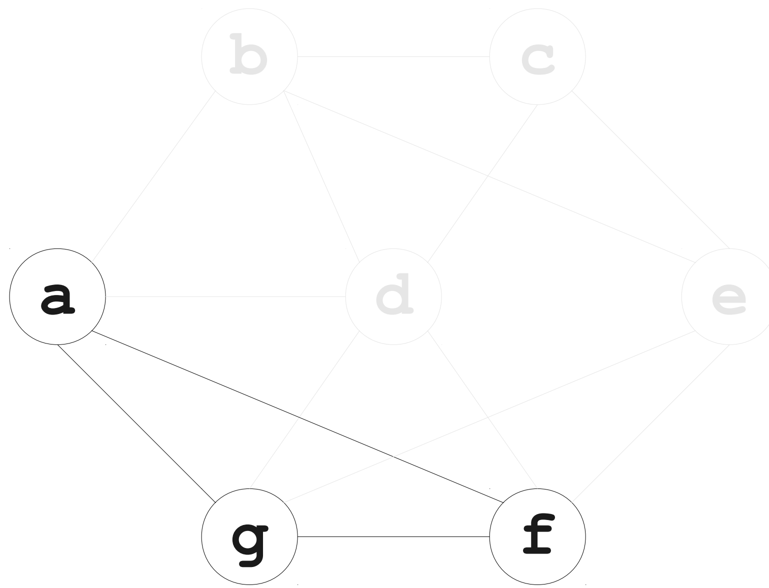
Chaitin's Algorithm

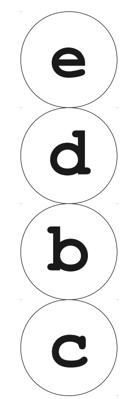


Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm

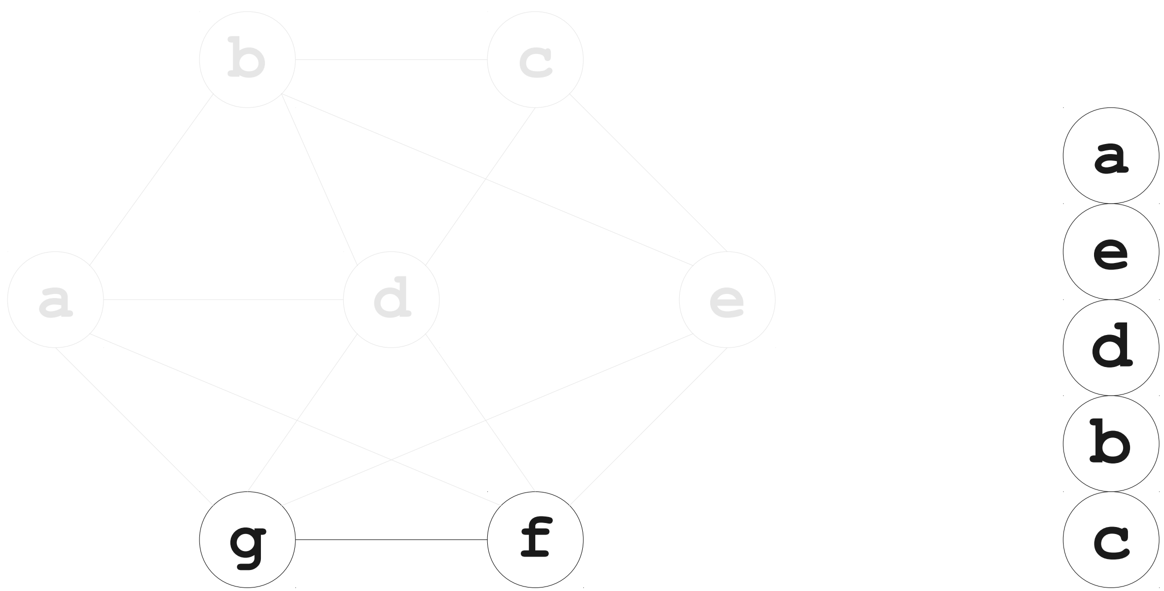




Registers

R_0	R_1	R_2	R_3

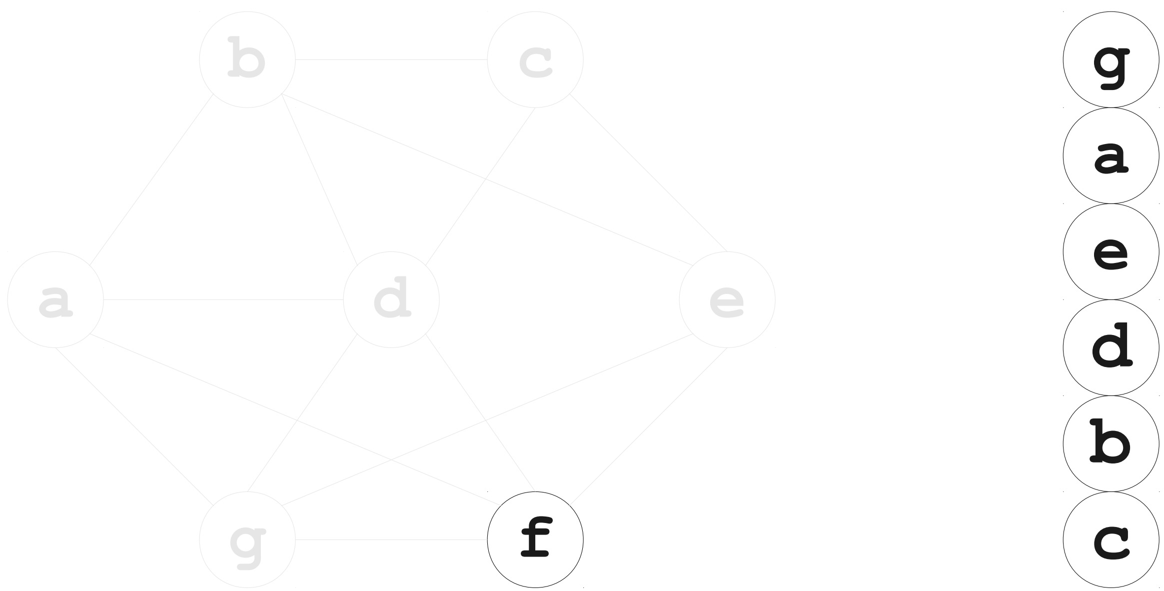
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



Registers

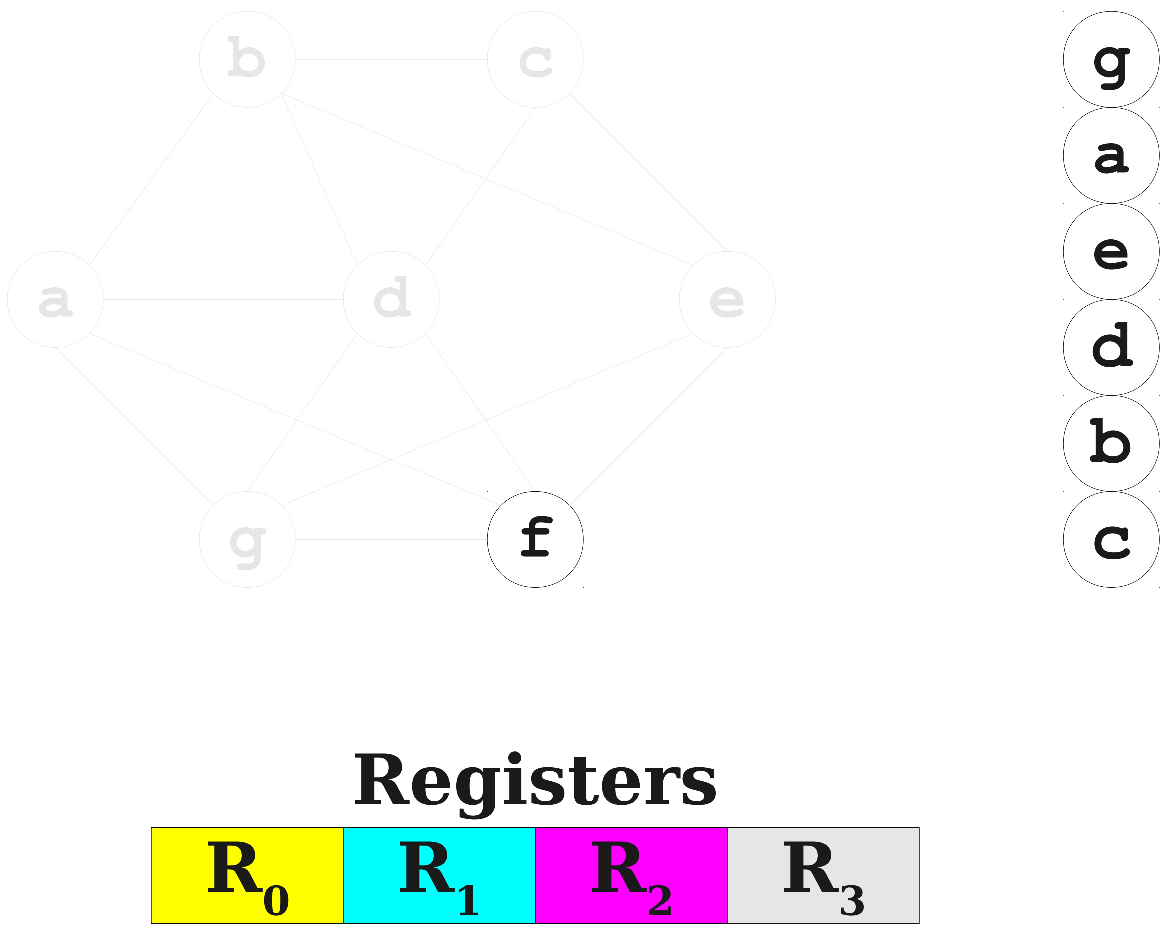
R_0	R_1	R_2	R_3

Chaitin's Algorithm

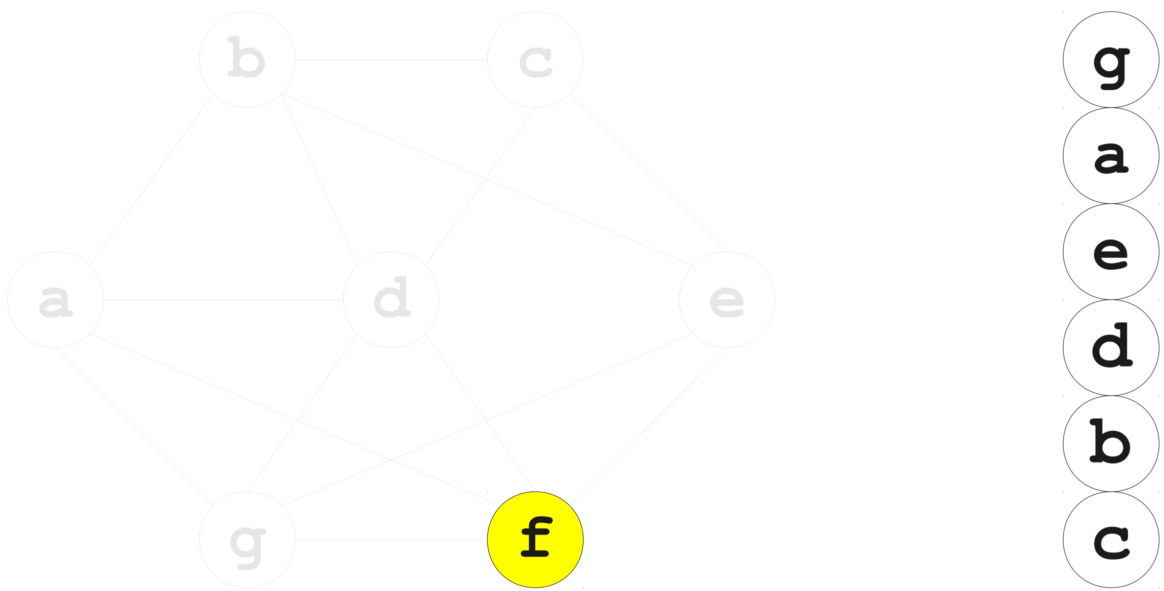
Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



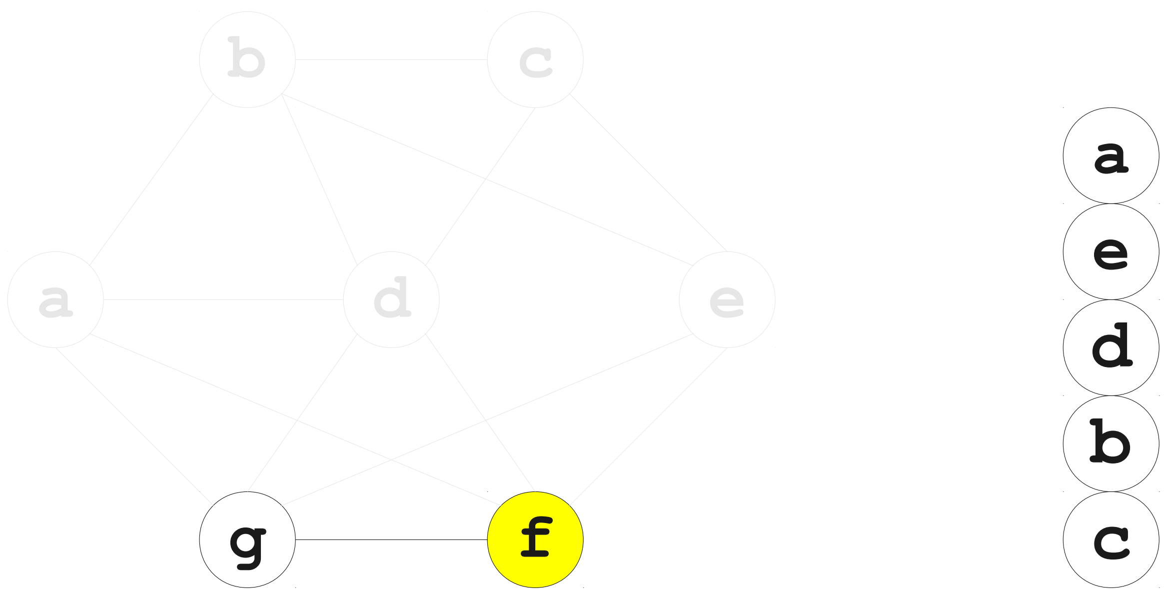
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

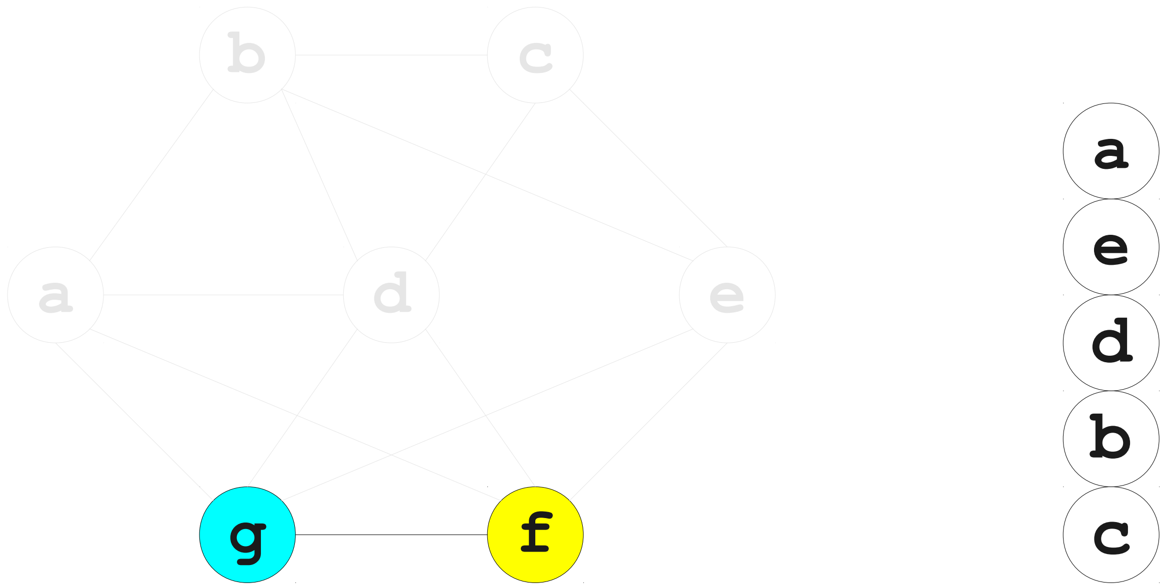
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

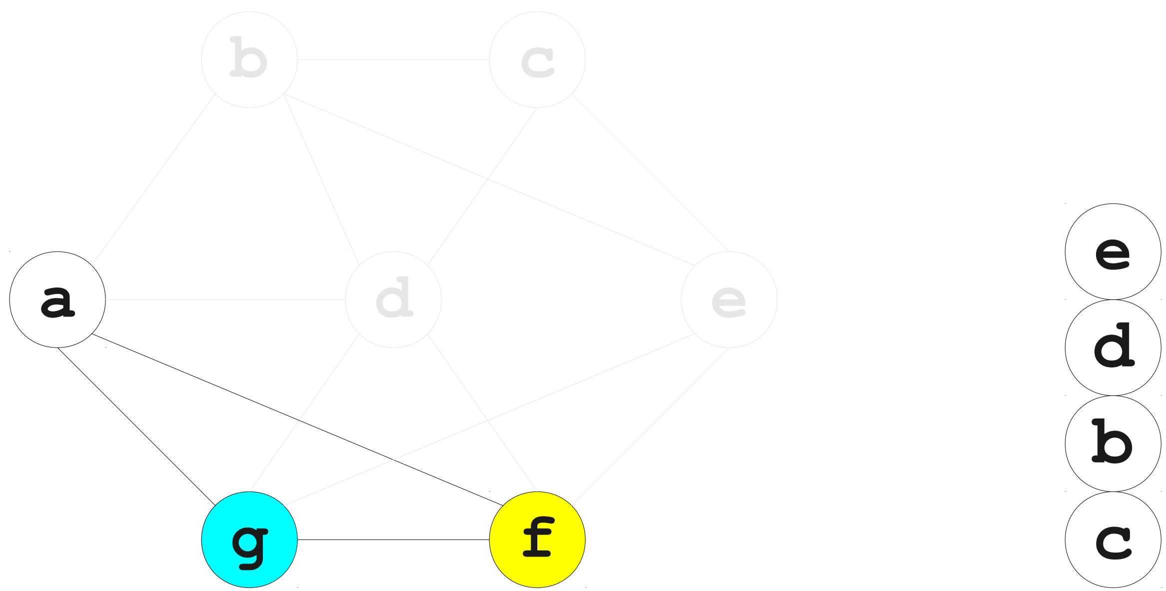
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

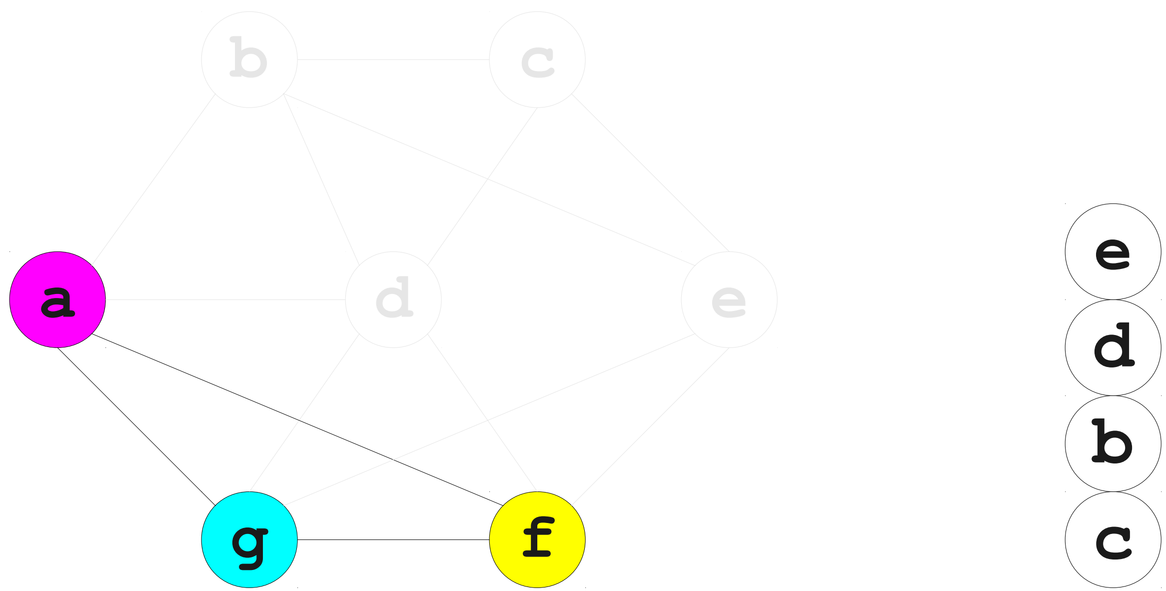
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

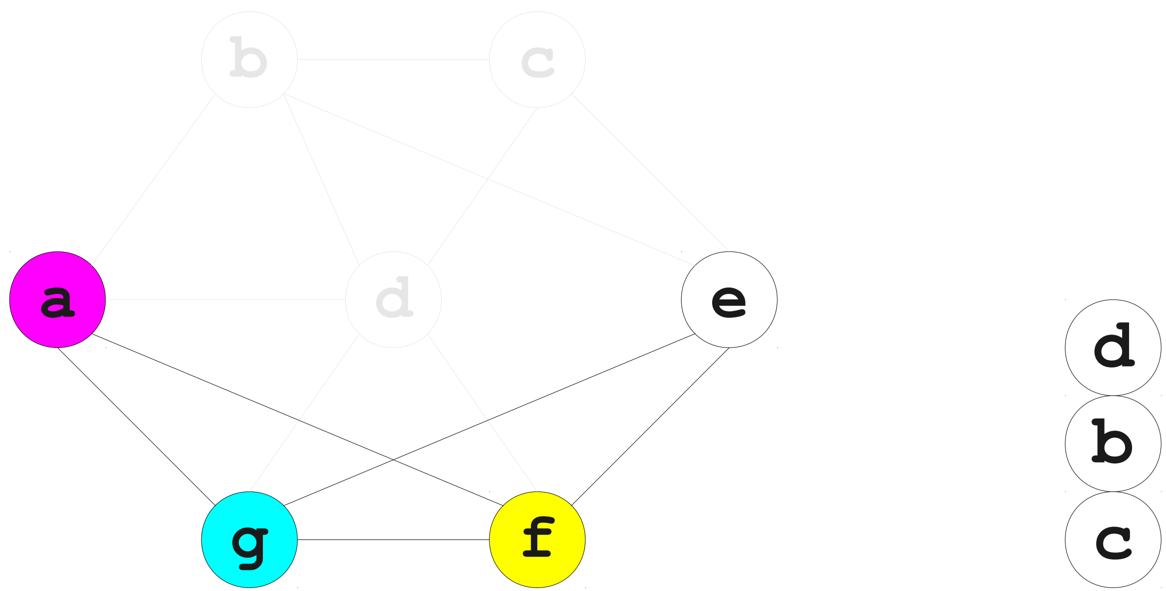
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

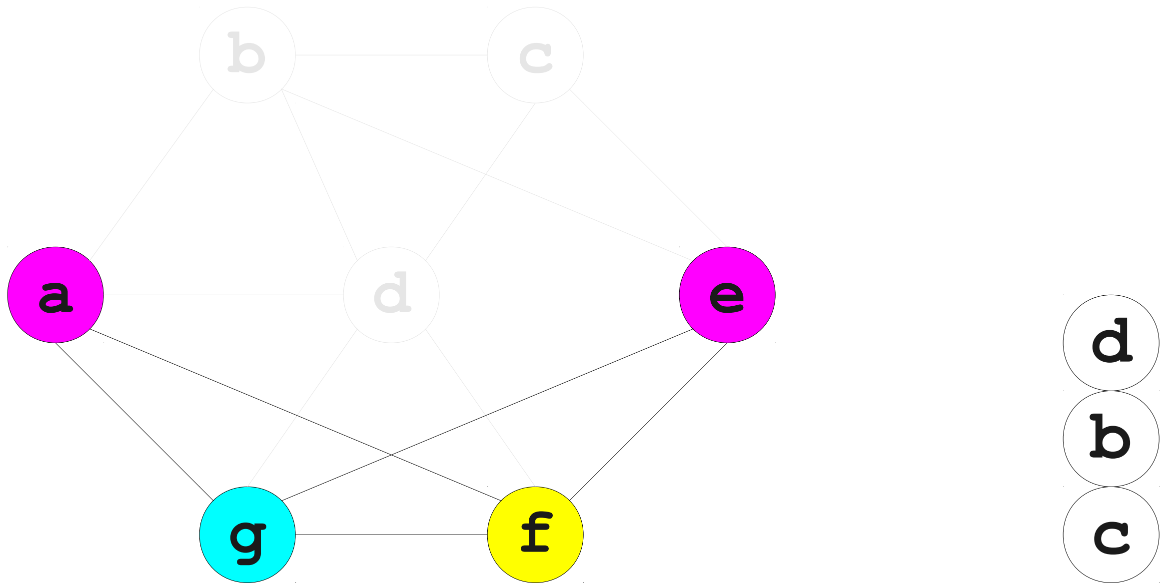
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

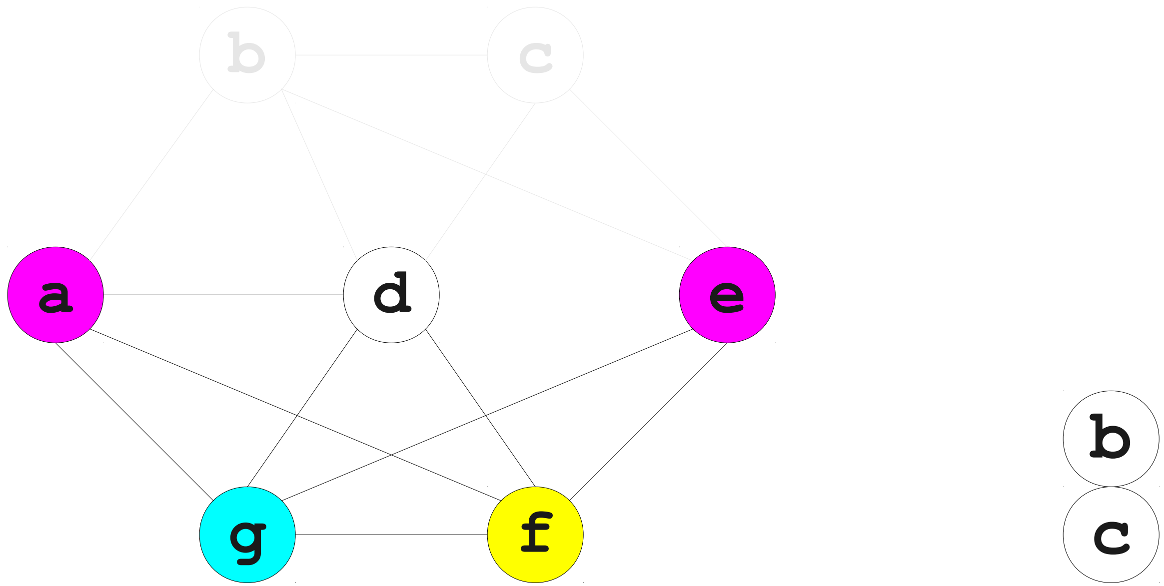
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

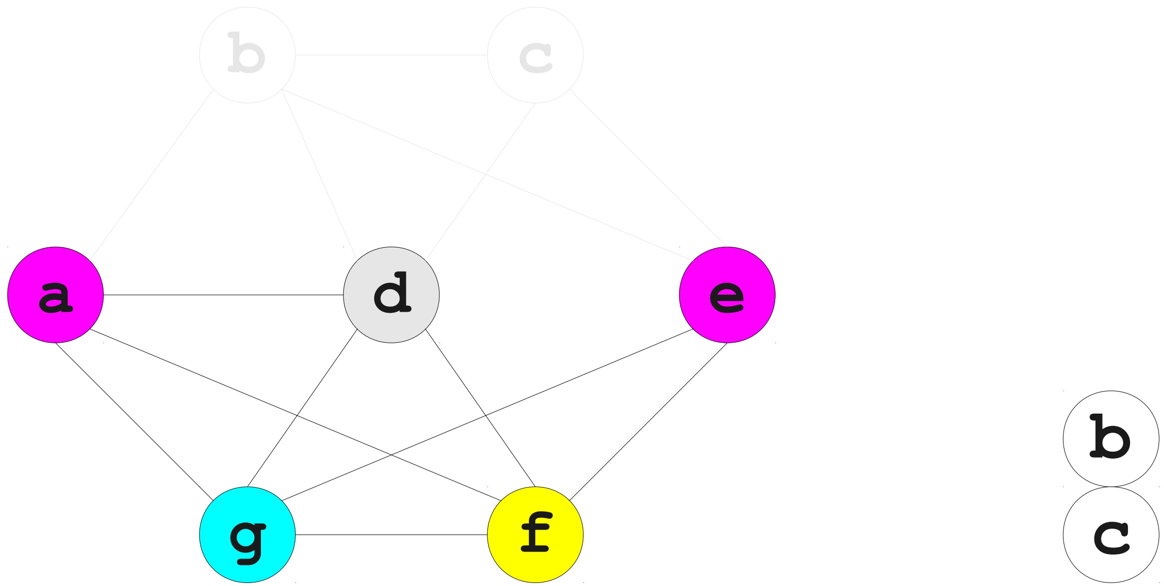
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

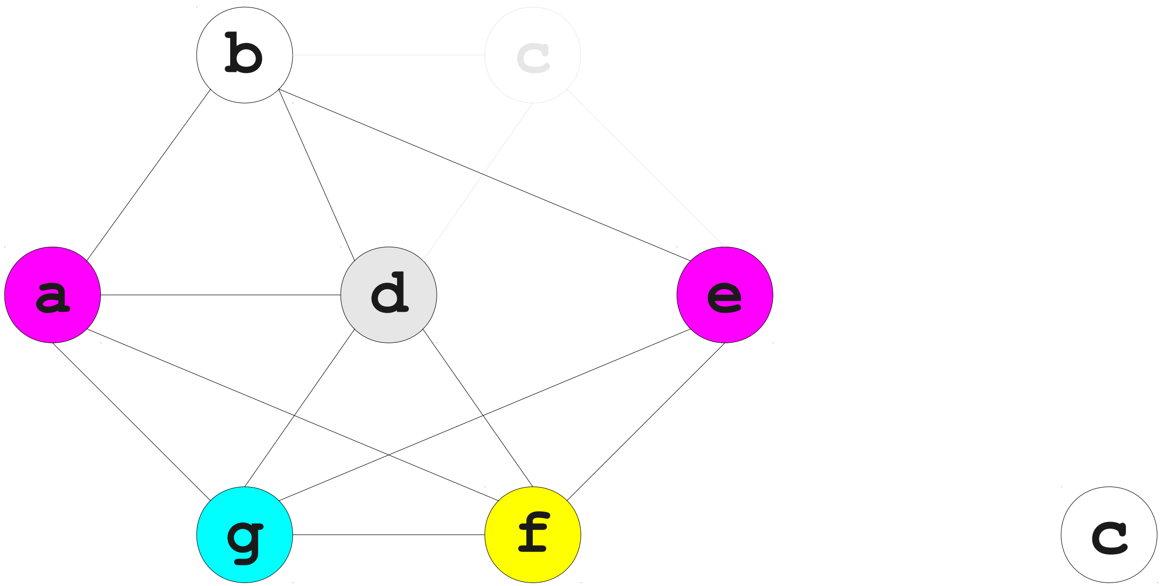
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

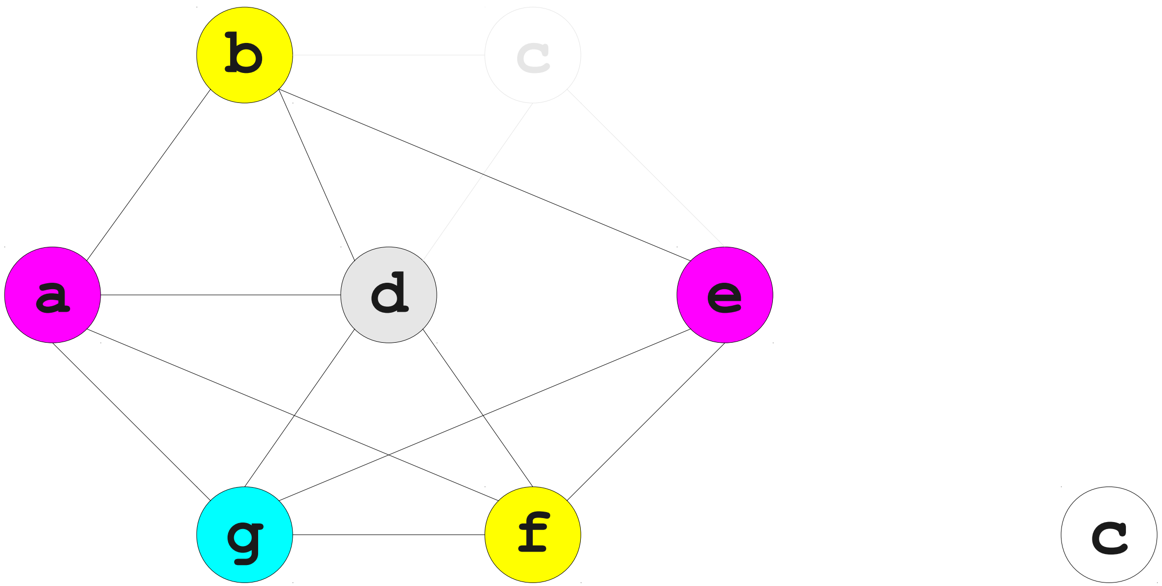
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

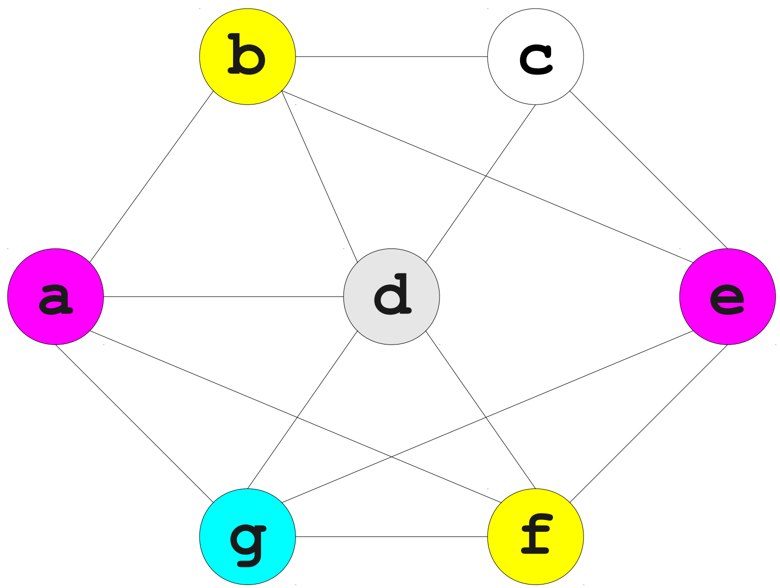
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

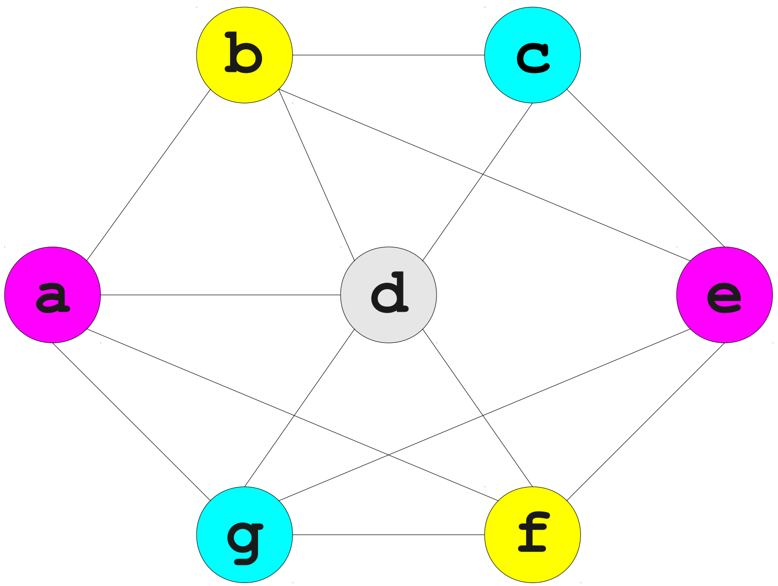
Chaitin's Algorithm



Registers

R0	R1	R2	R3

Chaitin's Algorithm



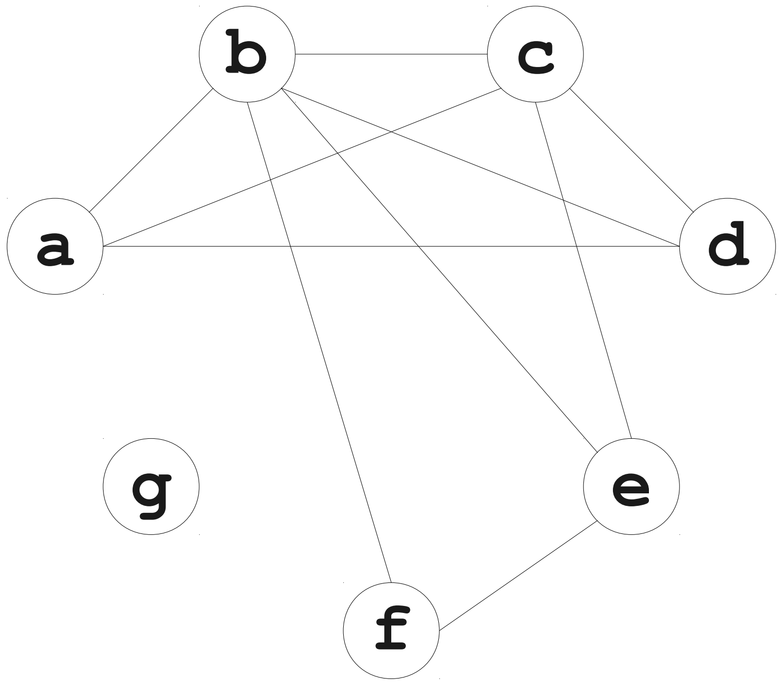
Registers

R_0	R_1	R_2	R_3

One Problem

- What if we can't find a node with fewer than *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.

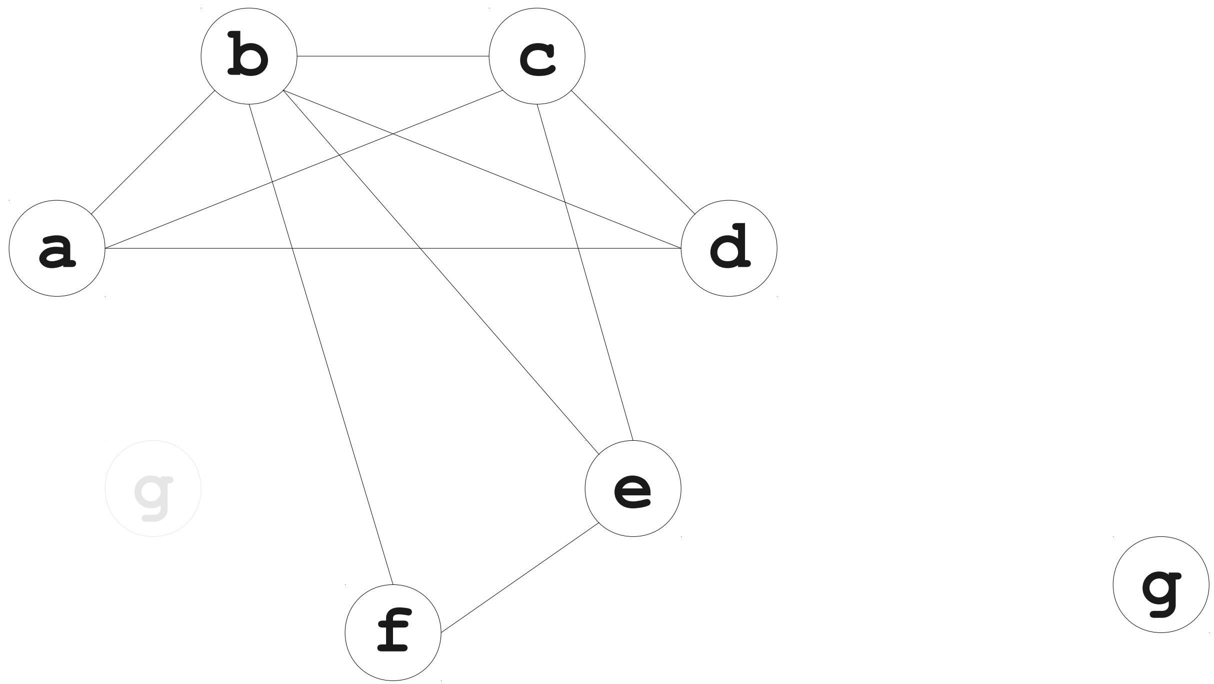
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

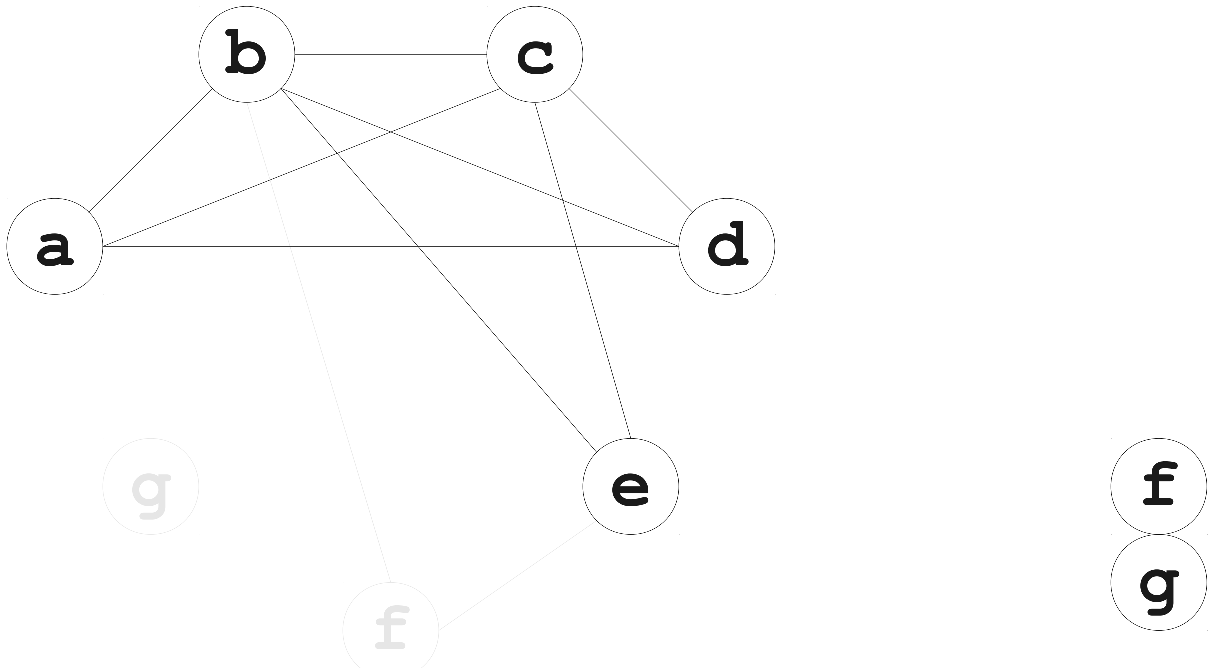
Chaitin's Algorithm Reloaded



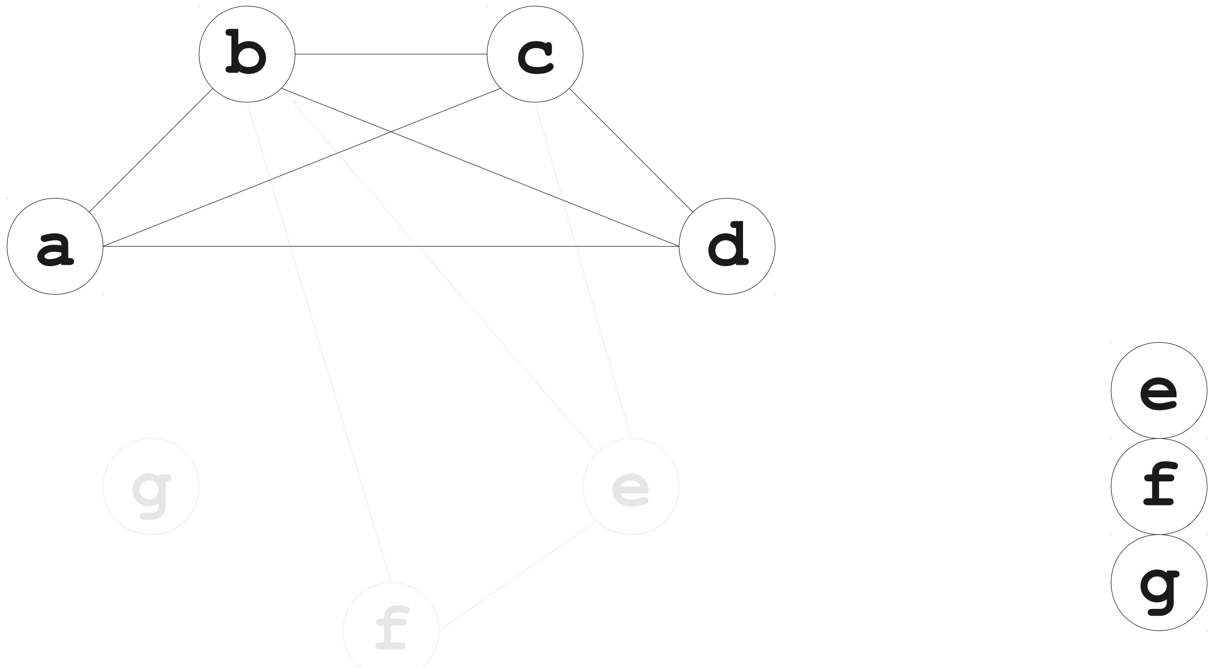
Registers

R_0	R_1	R_2

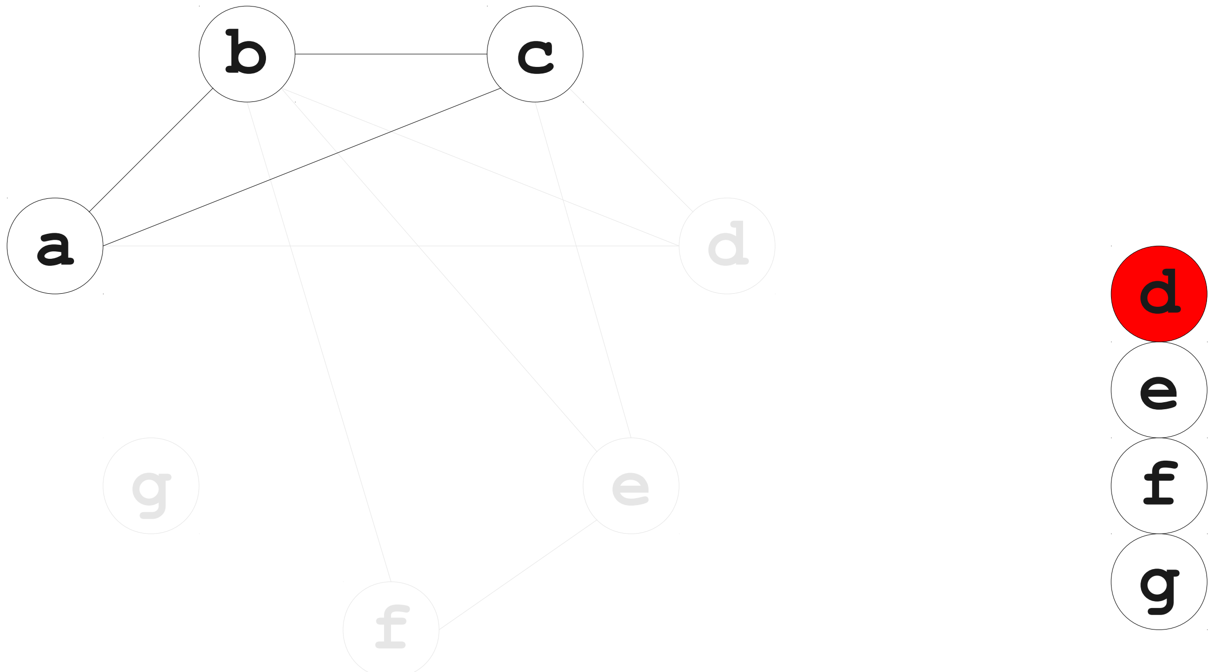
Chaitin's Algorithm Reloaded



Chaitin's Algorithm Reloaded



Chaitin's Algorithm Reloaded



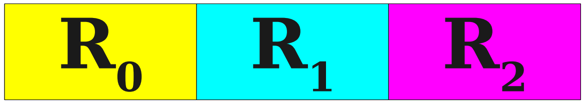
Chaitin's Algorithm Reloaded

Registers

R_0	R_1	R_2

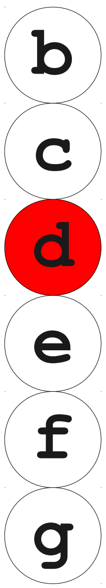
Chaitin's Algorithm Reloaded

Registers



Chaitin's Algorithm Reloaded

Chaitin's Algorithm Reloaded



Registers

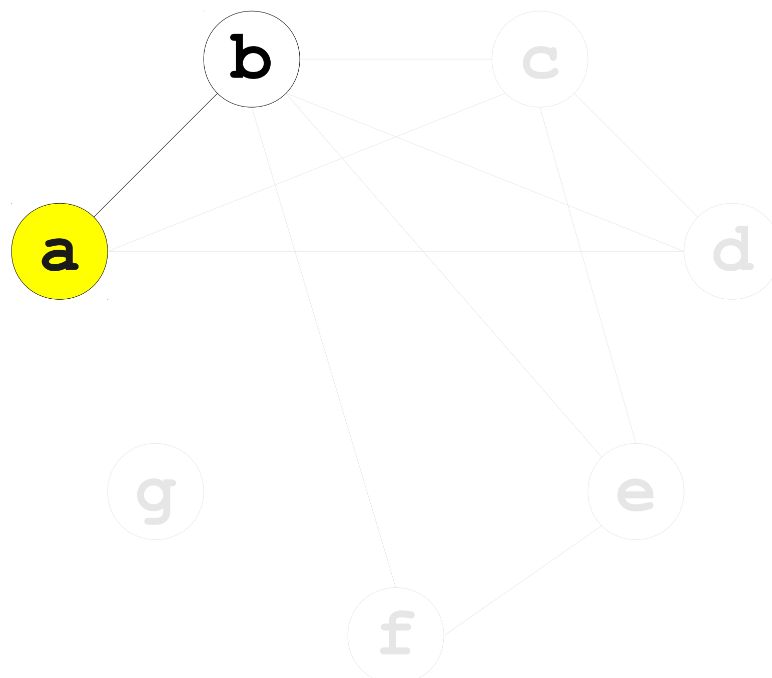
R_0	R_1	R_2

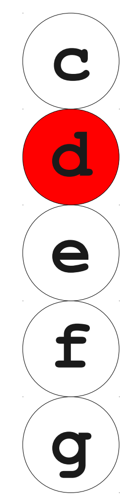
Chaitin's Algorithm Reloaded

Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded

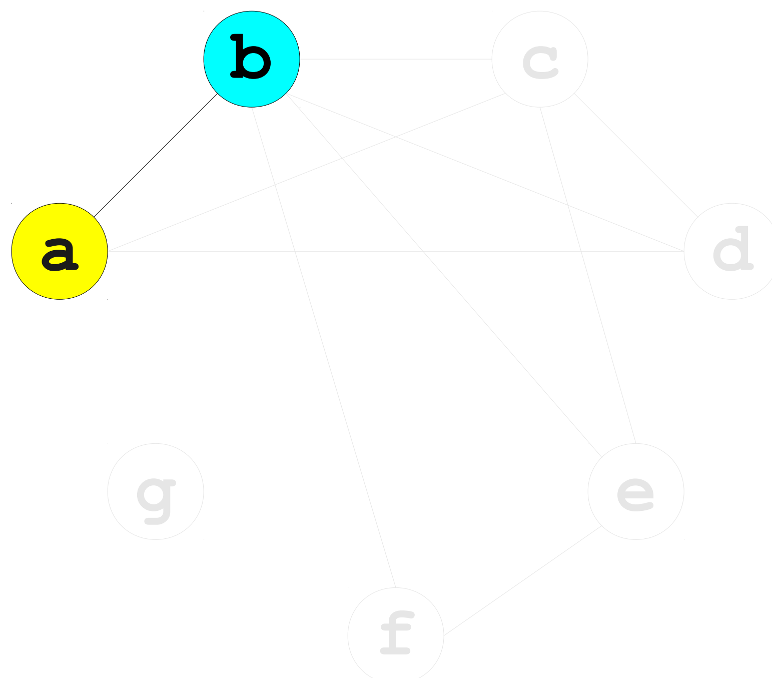


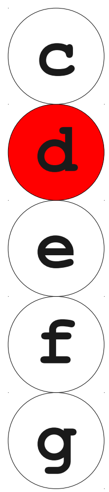


Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded

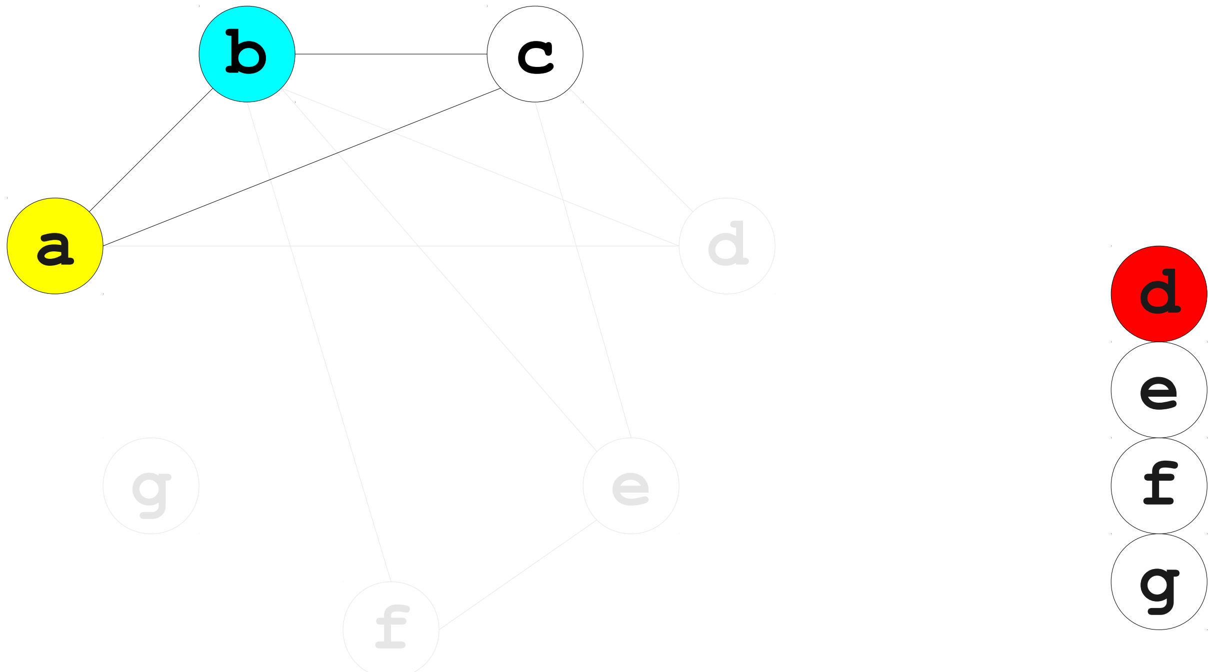




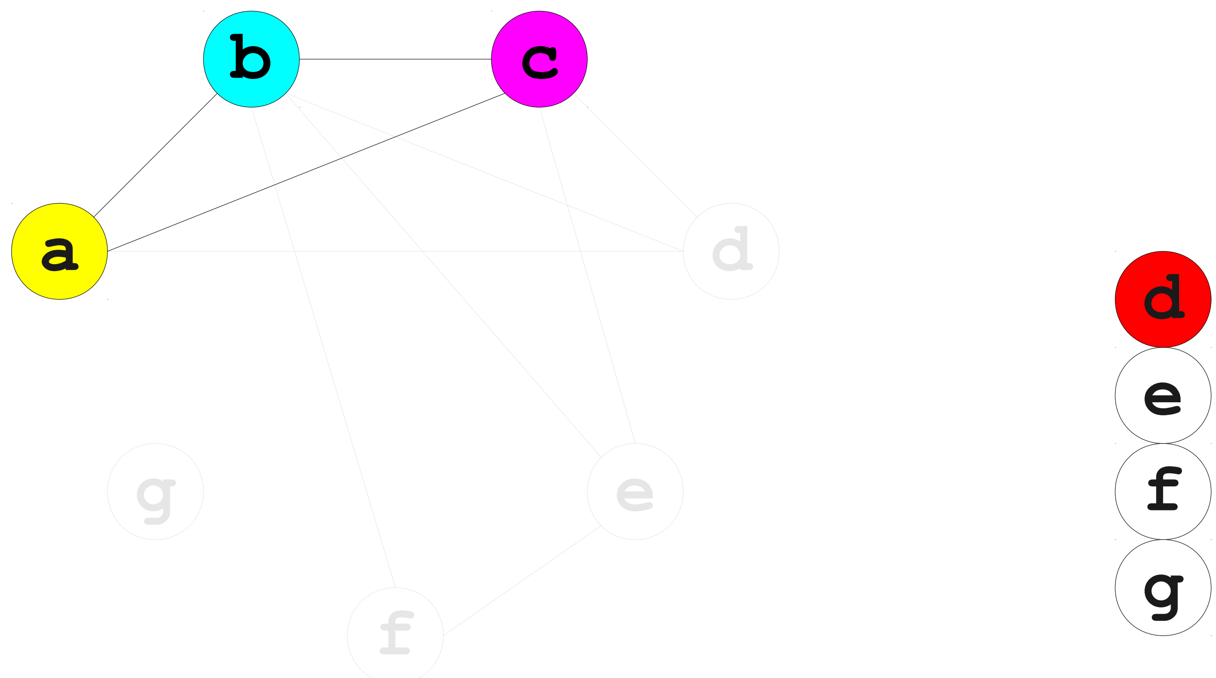
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



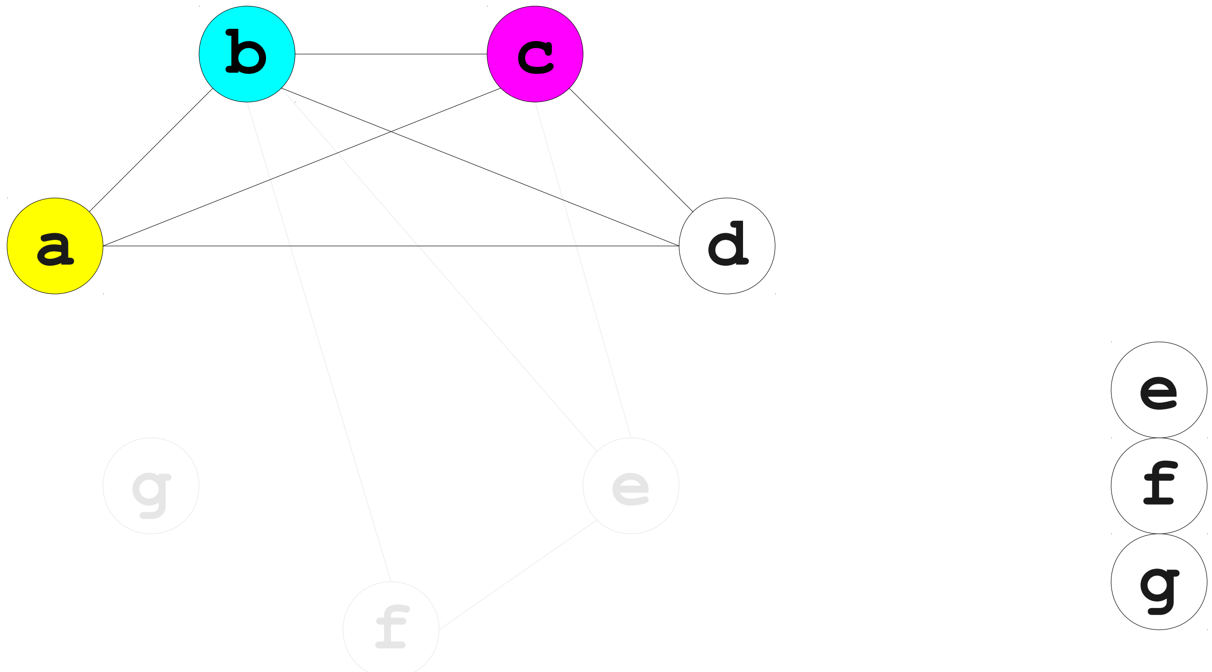
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

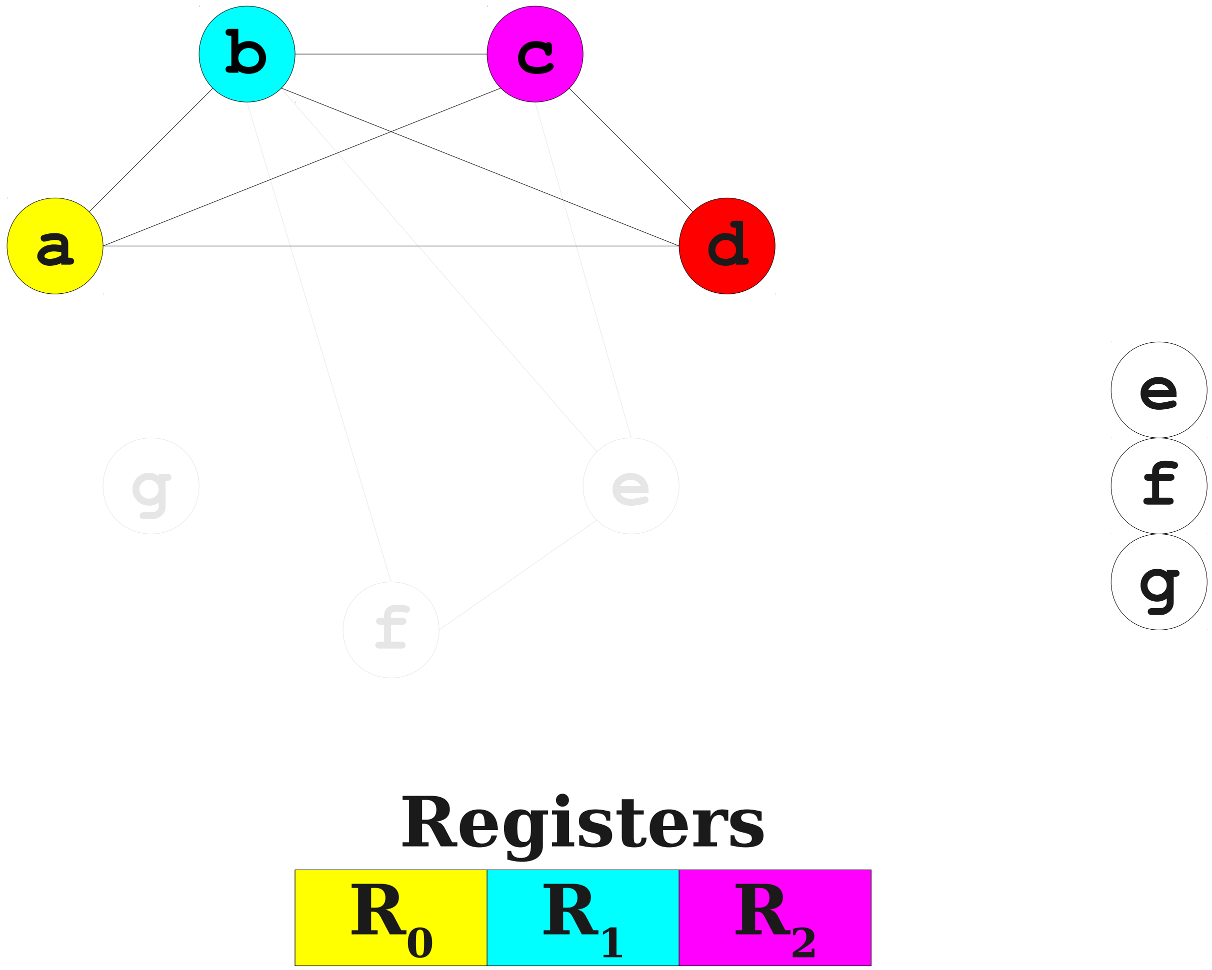
Chaitin's Algorithm Reloaded



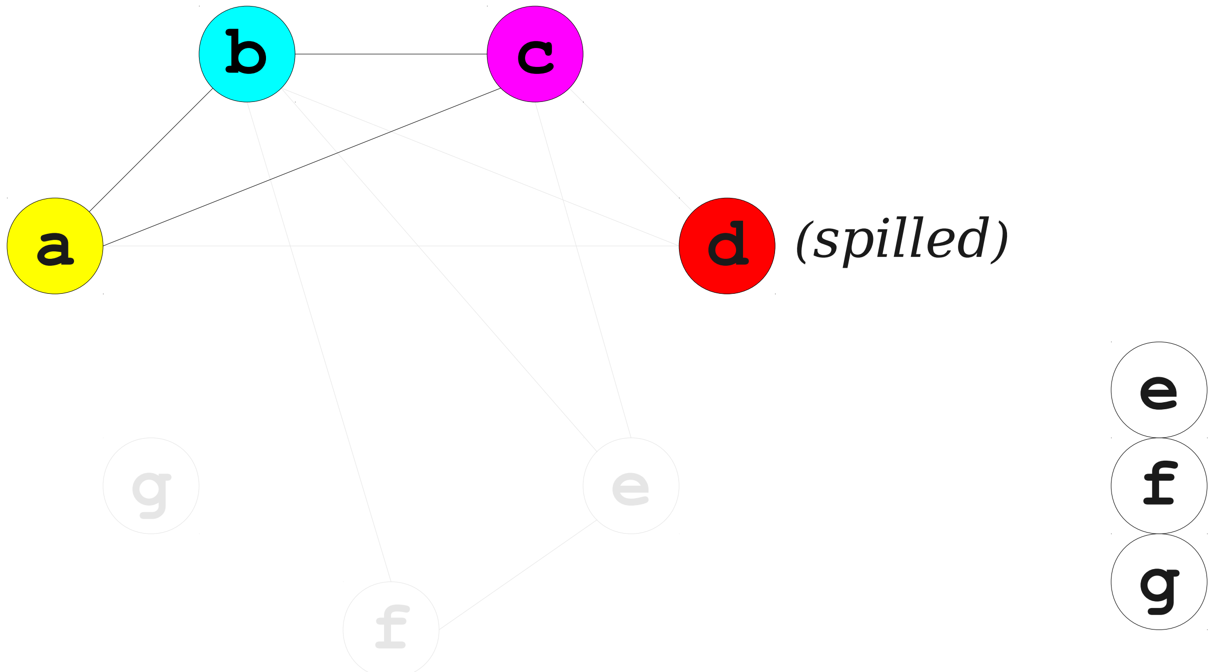
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



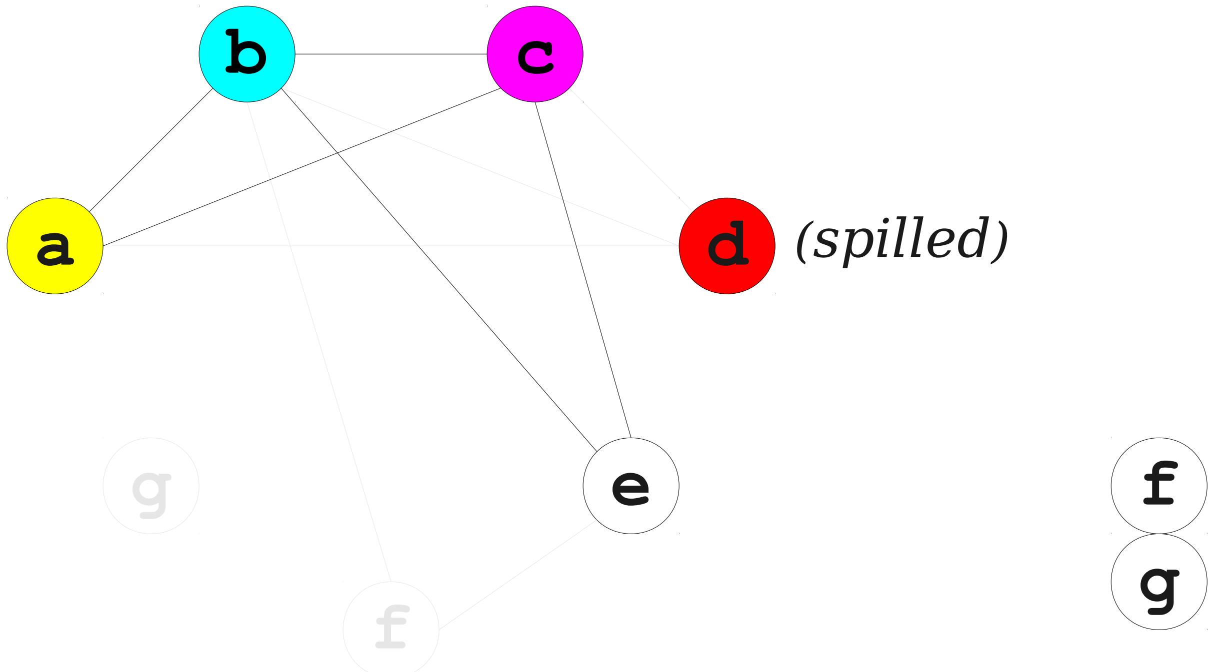
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

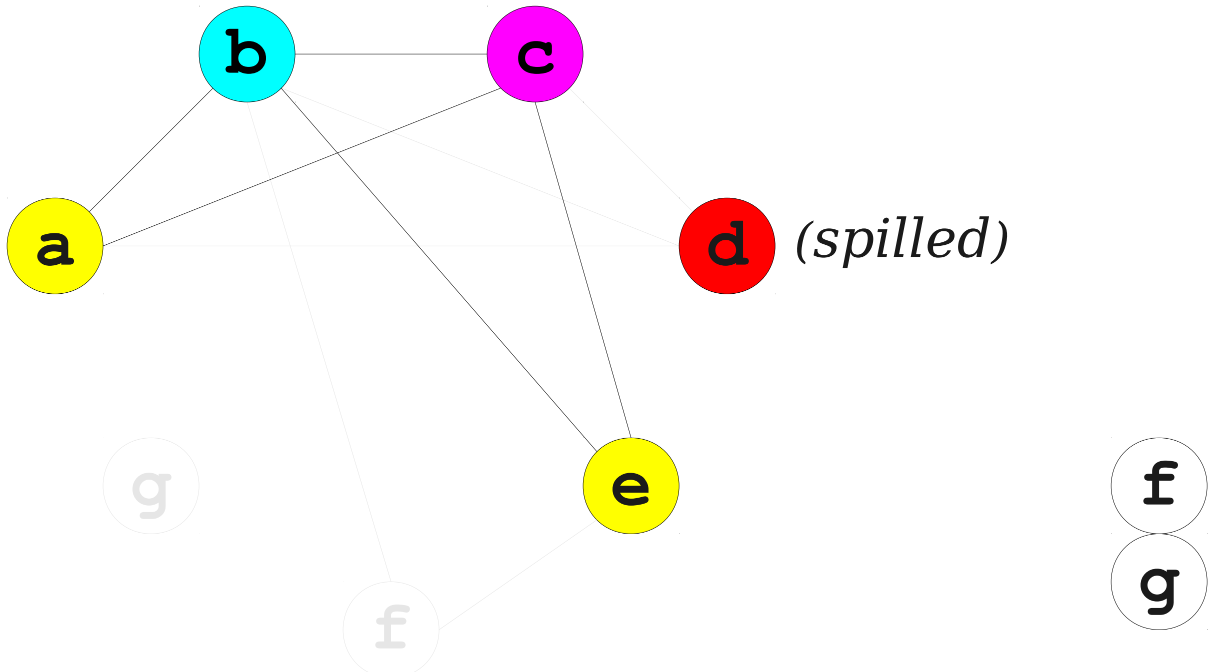
Chaitin's Algorithm Reloaded



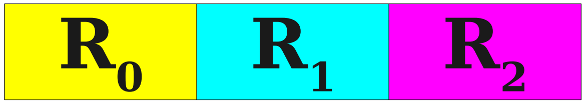
Registers

R_0	R_1	R_2

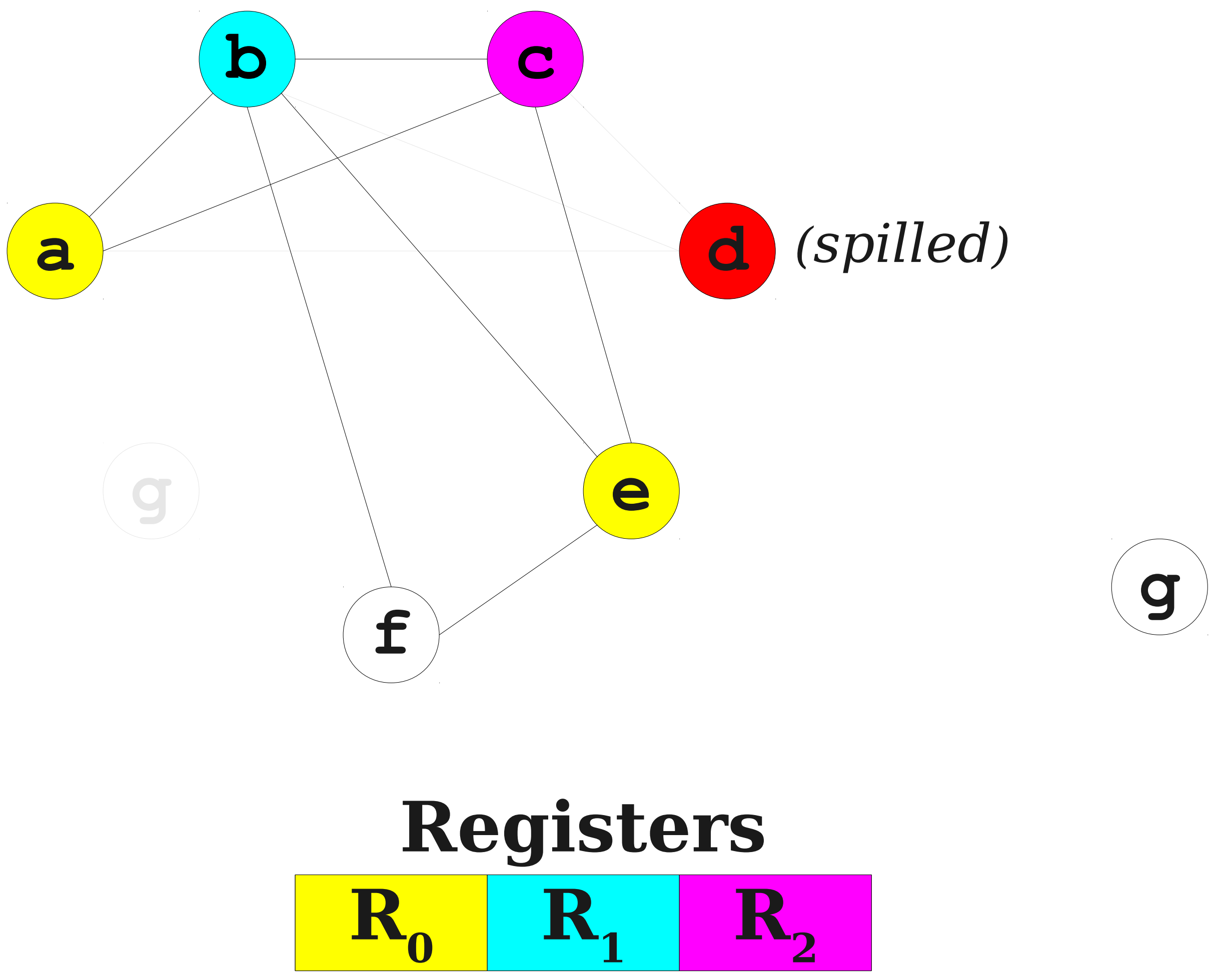
Chaitin's Algorithm Reloaded



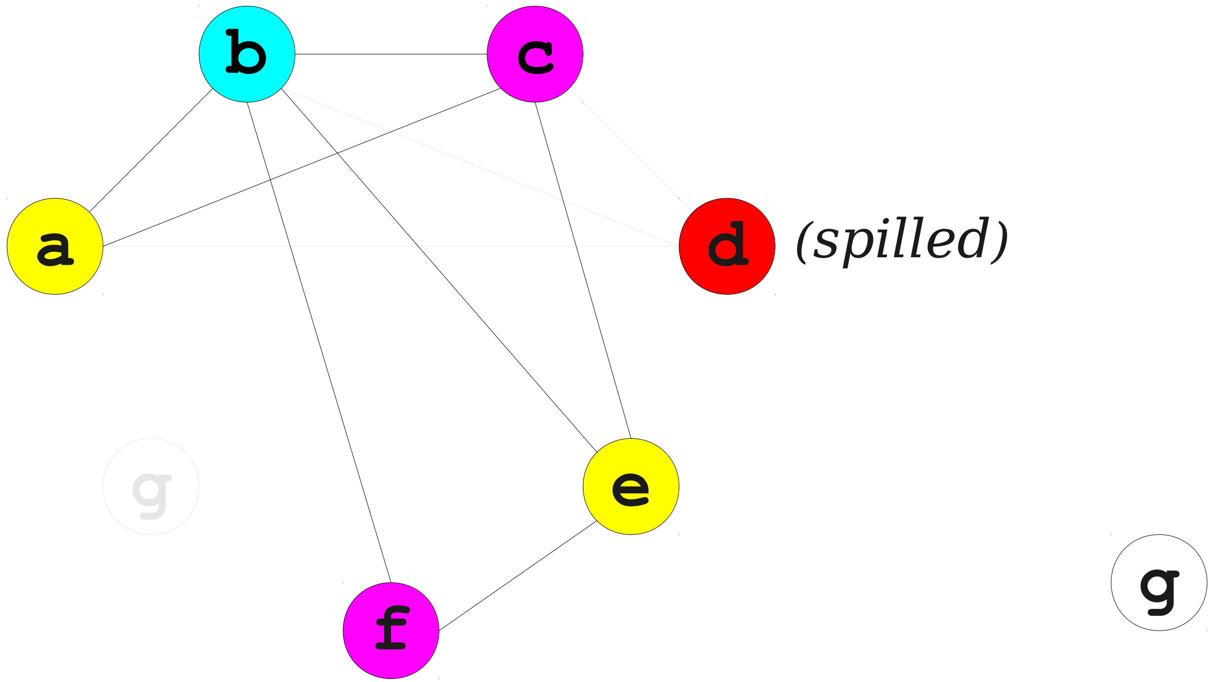
Registers



Chaitin's Algorithm Reloaded



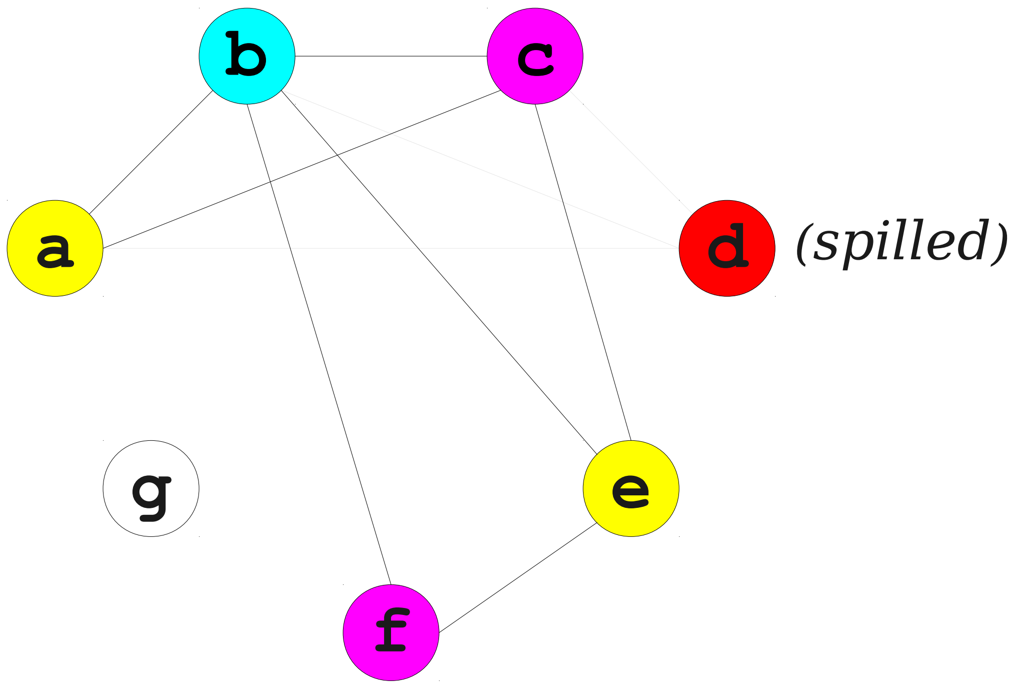
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

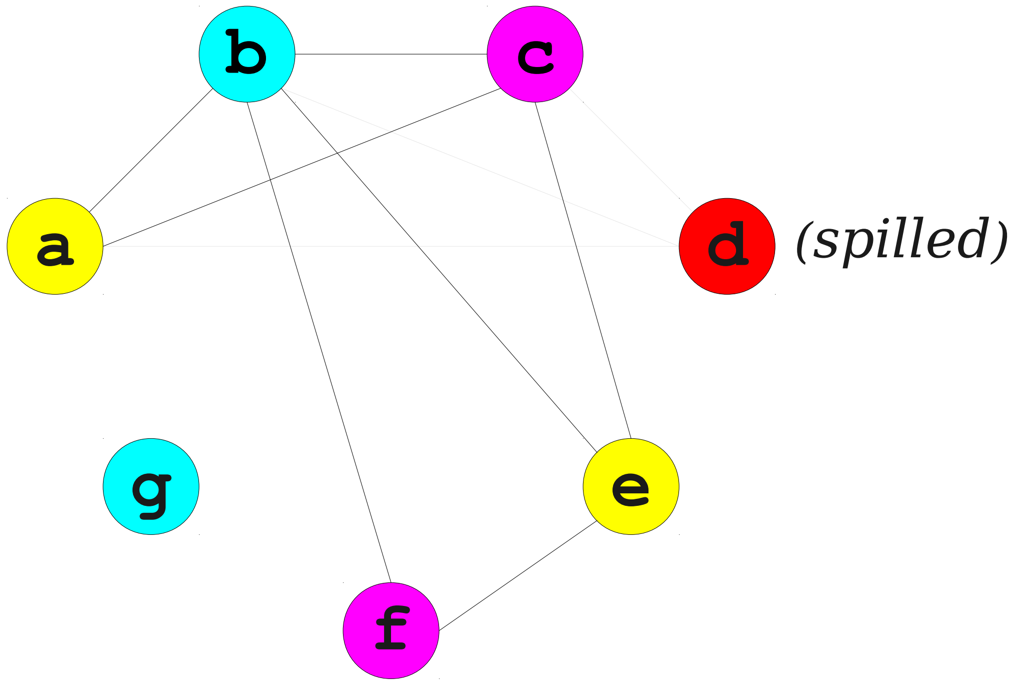
Chaitin's Algorithm Reloaded



Registers

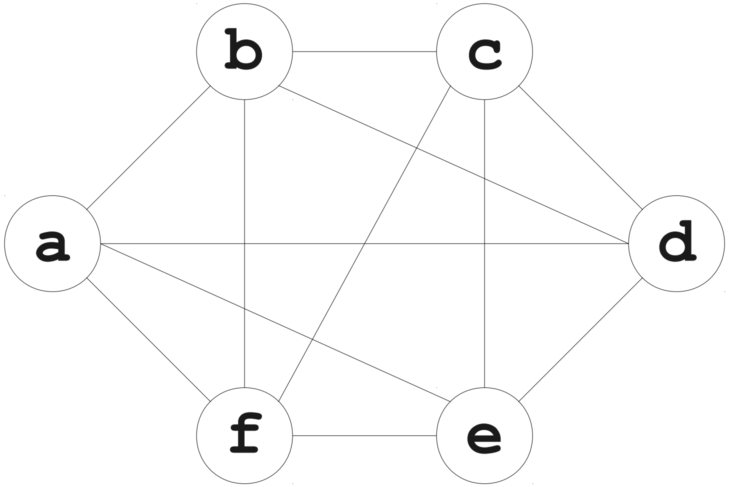
R_0	R_1	R_2

Chaitin's Algorithm Reloaded

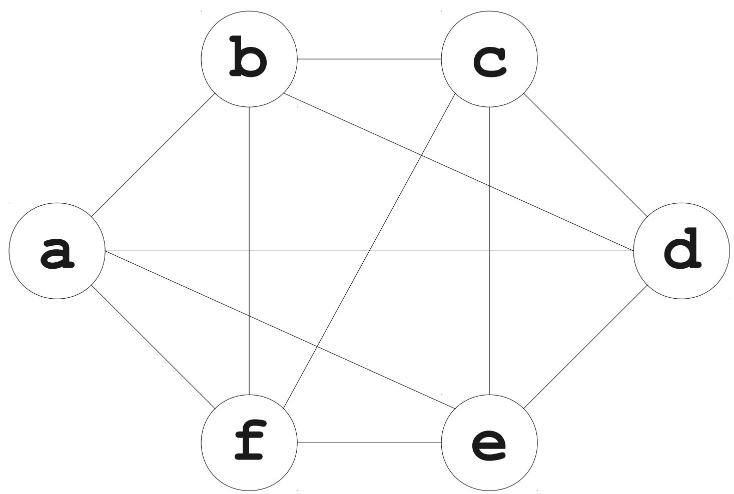


Another Example

Another Example



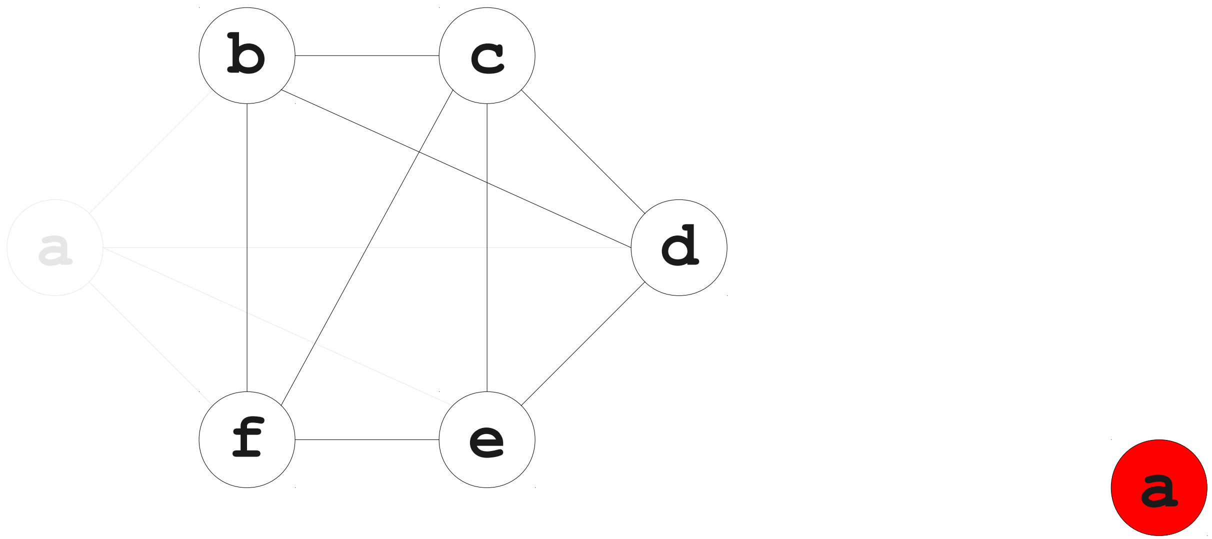
Another Example



Registers

R_0	R_1	R_2

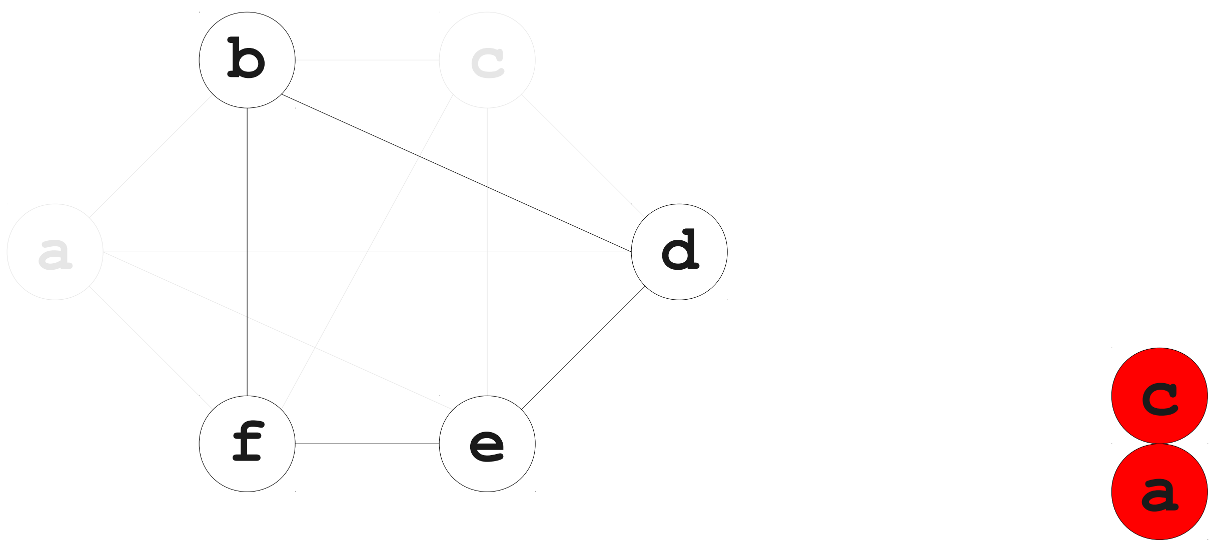
Another Example



Registers

R_0	R_1	R_2

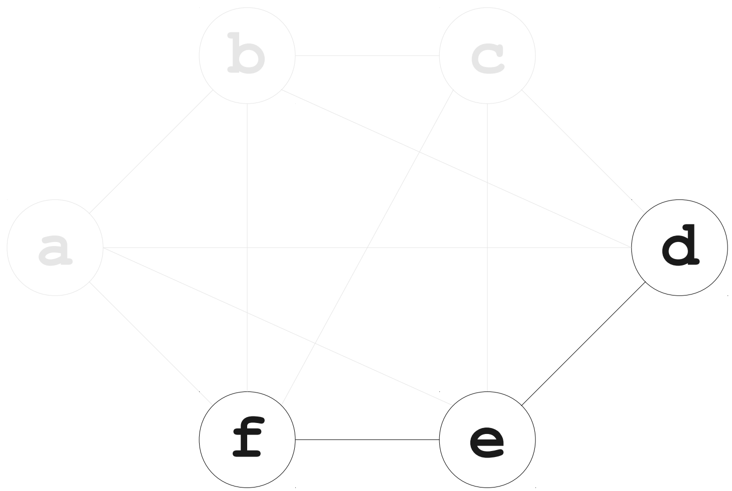
Another Example

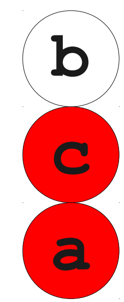


Registers

R_0	R_1	R_2

Another Example

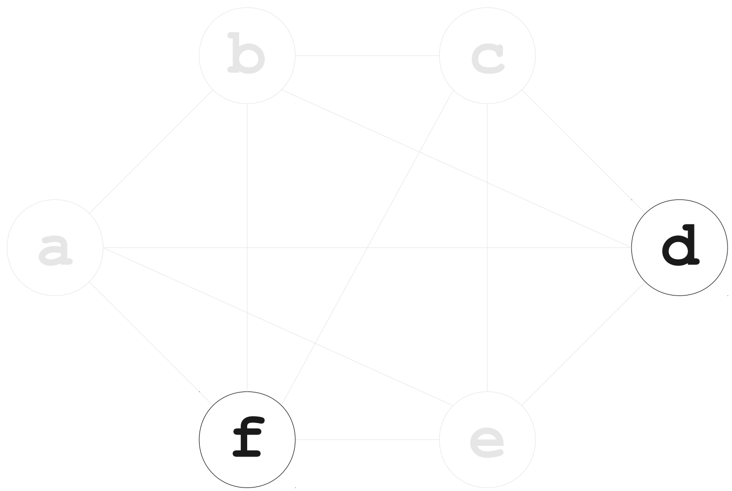


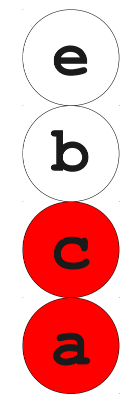


Registers

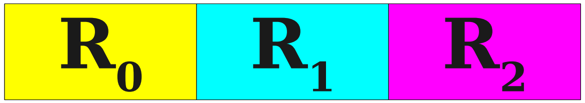
R_0	R_1	R_2

Another Example

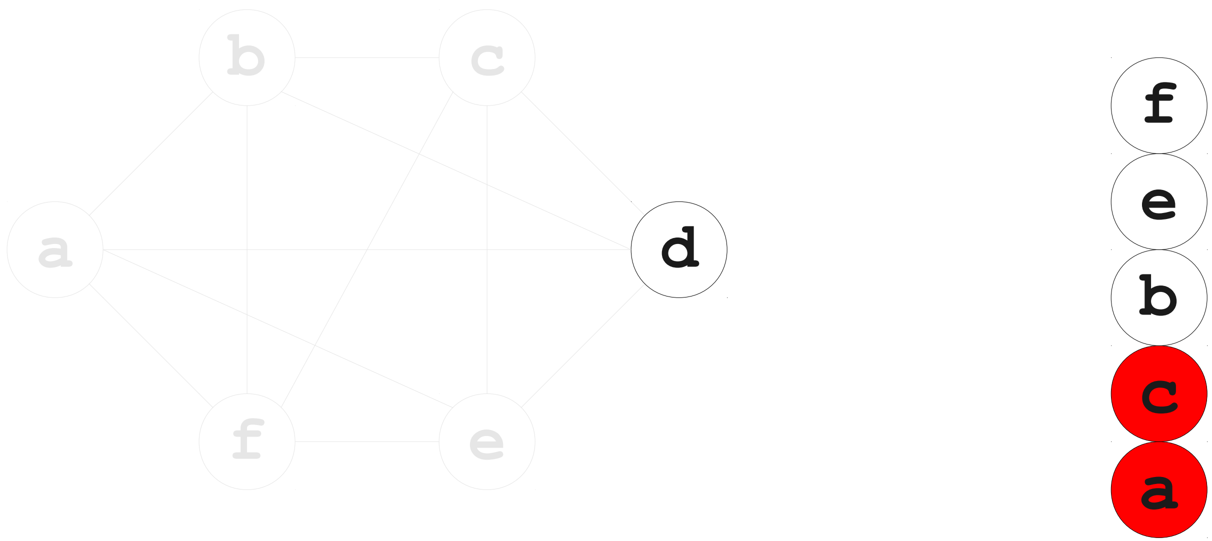




Registers



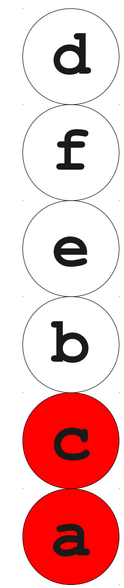
Another Example



Registers

R_0	R_1	R_2

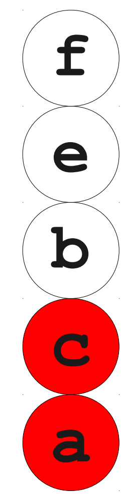
Another Example



Registers

R_0	R_1	R_2

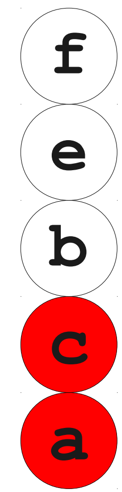
Another Example



Registers

R_0	R_1	R_2

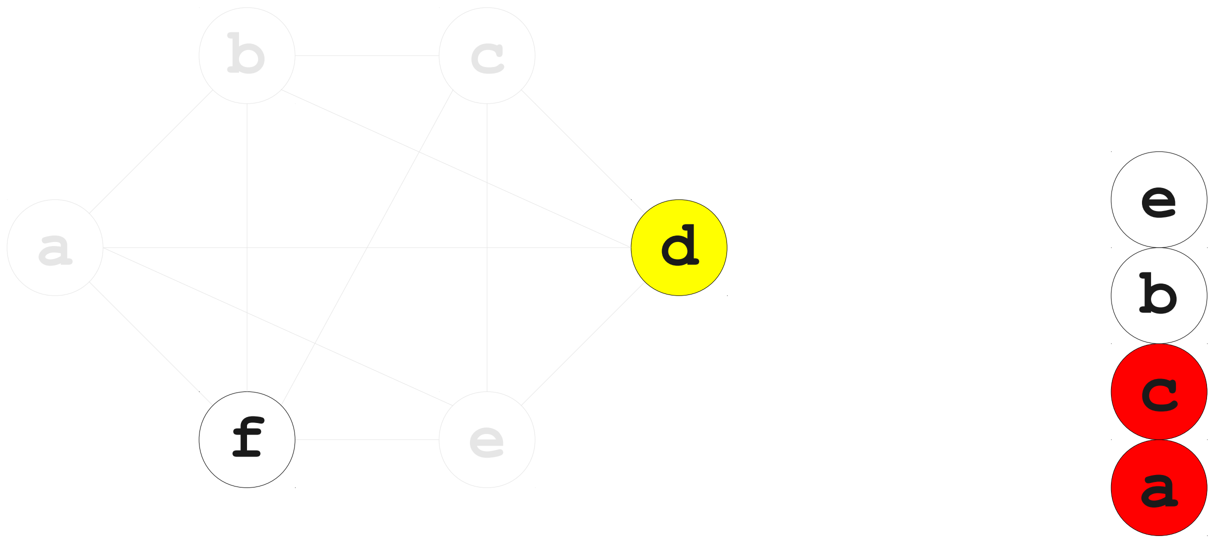
Another Example



Registers

R_0	R_1	R_2

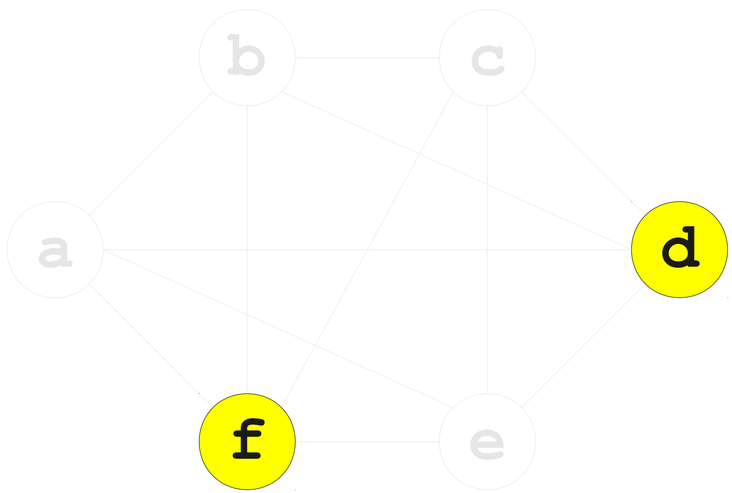
Another Example

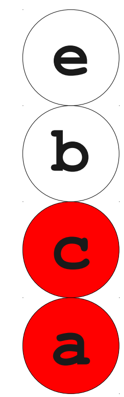


Registers

R_0	R_1	R_2

Another Example

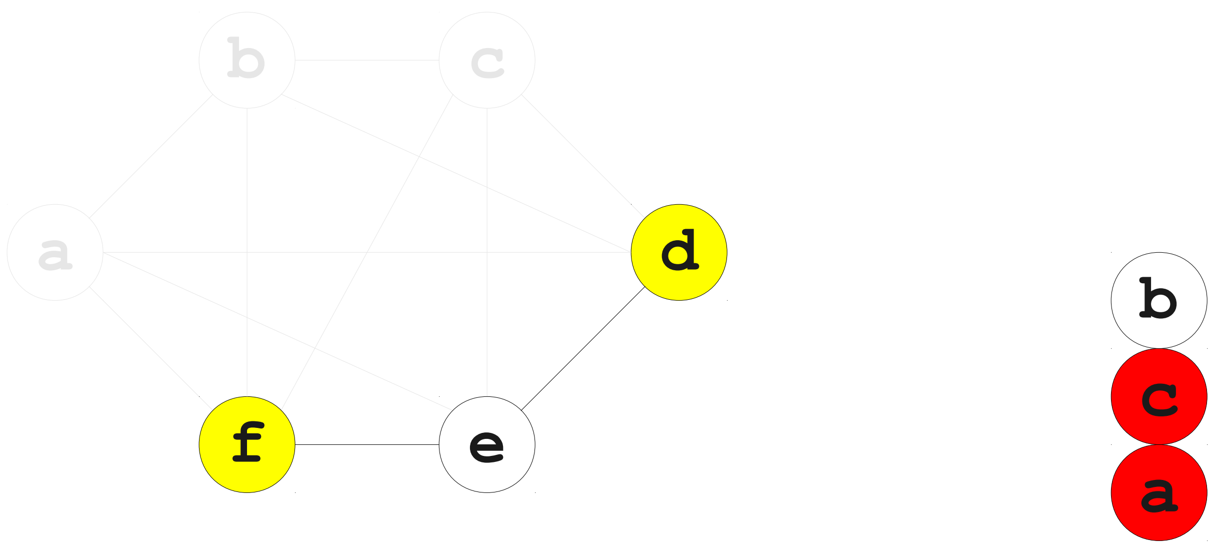




Registers

R_0	R_1	R_2

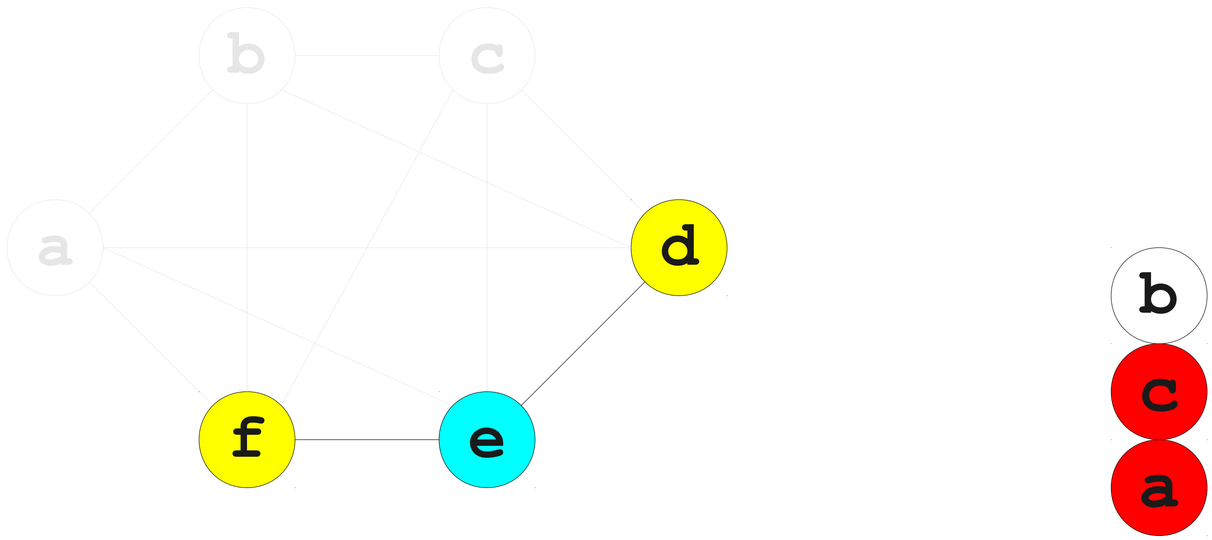
Another Example



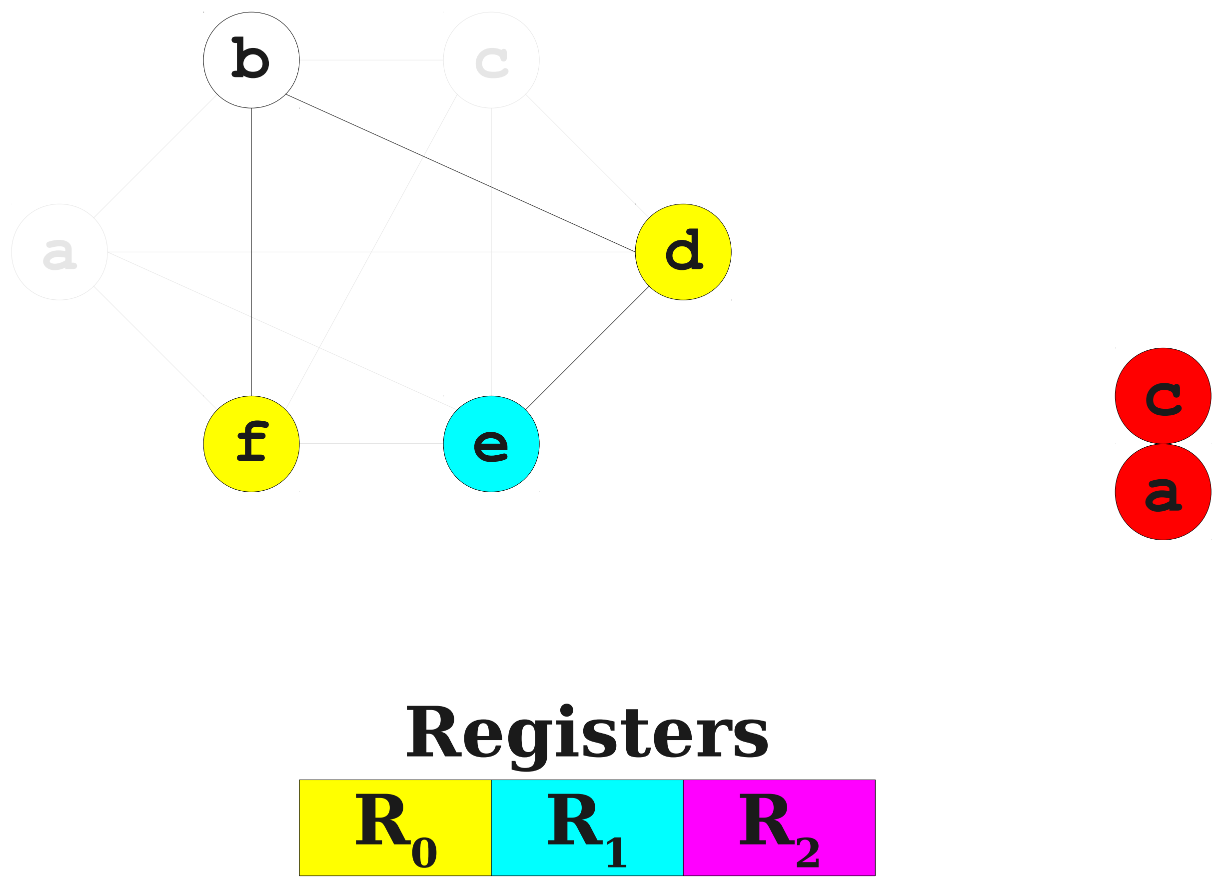
Registers

R_0	R_1	R_2

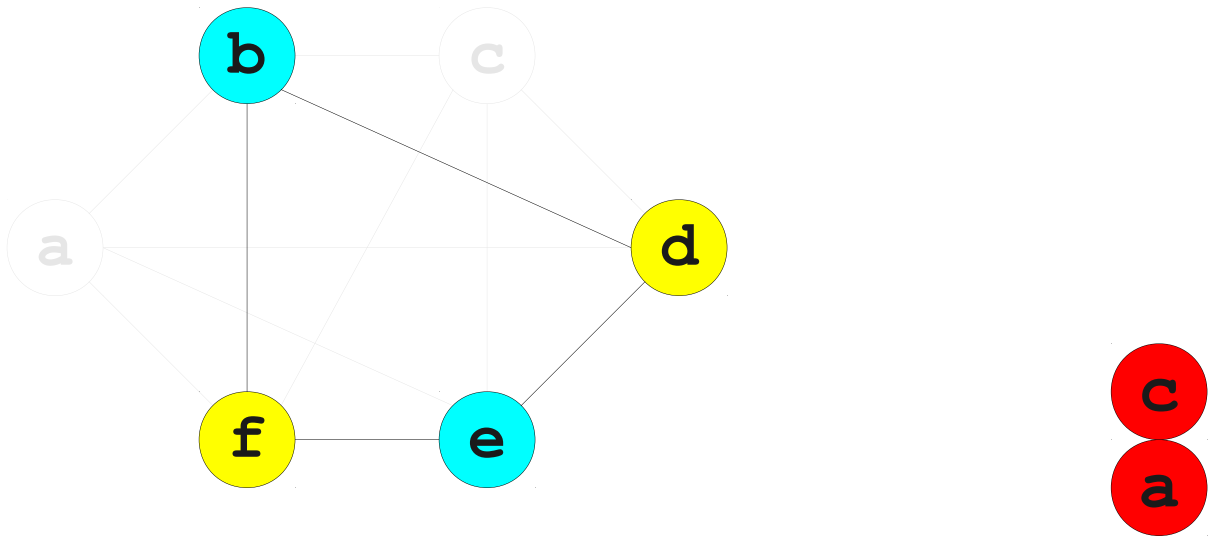
Another Example



Another Example



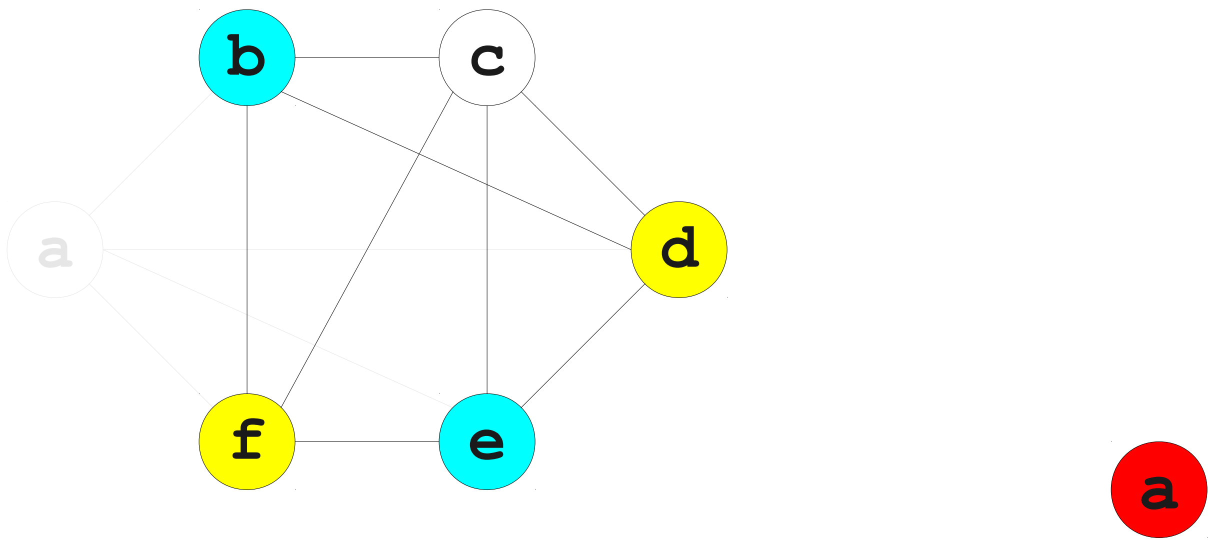
Another Example



Registers

R_0	R_1	R_2

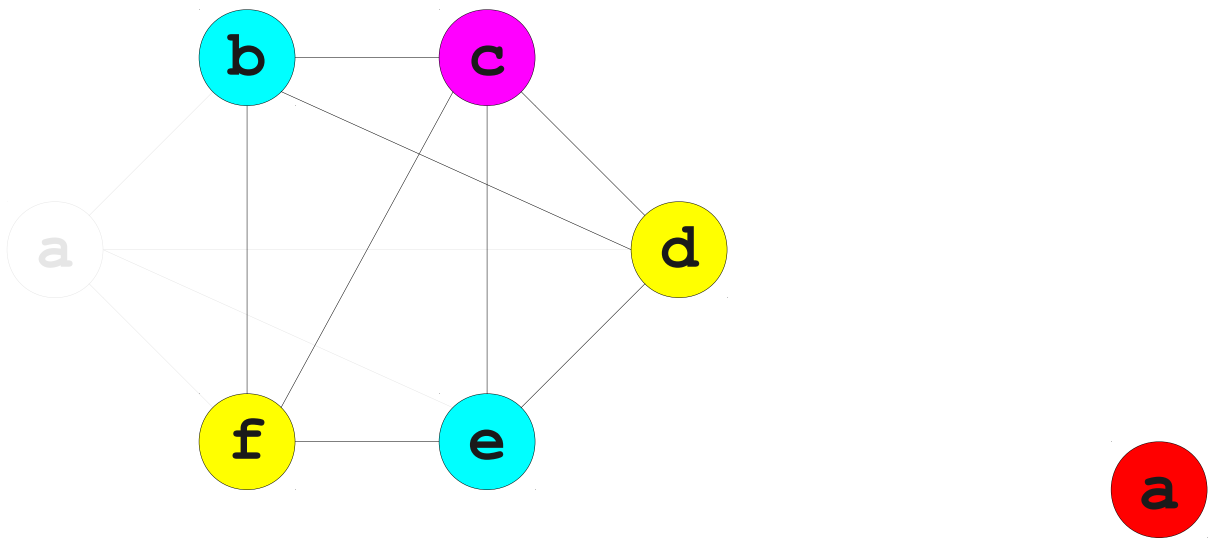
Another Example



Registers

R_0	R_1	R_2

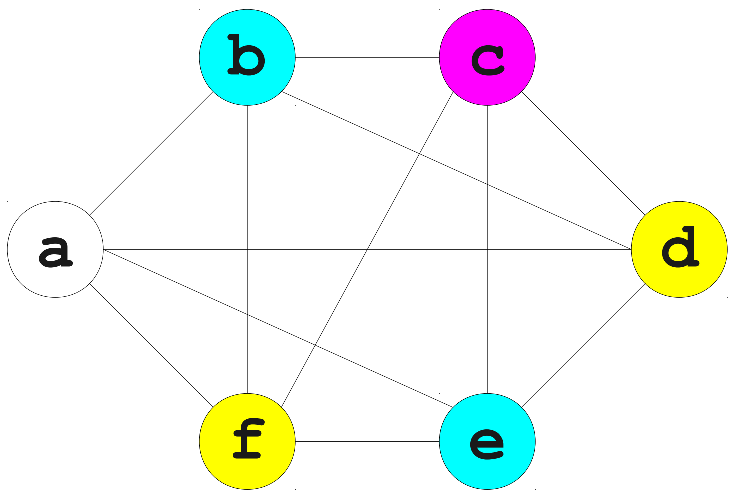
Another Example



Registers

R_0	R_1	R_2

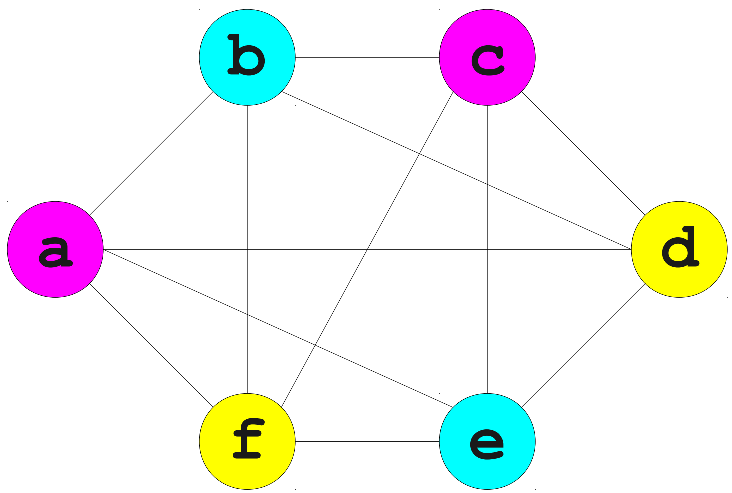
Another Example



Registers

R_0	R_1	R_2

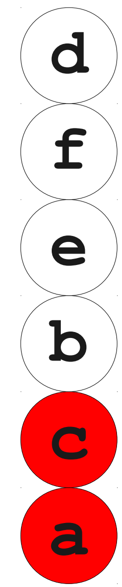
Another Example



Registers

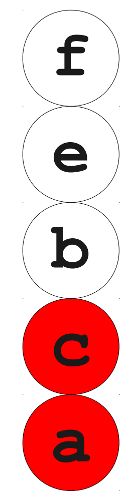
R_0	R_1	R_2

Another Example



Registers

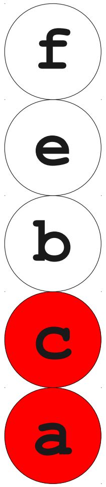
Another Example



Registers

R_0	R_1	R_2

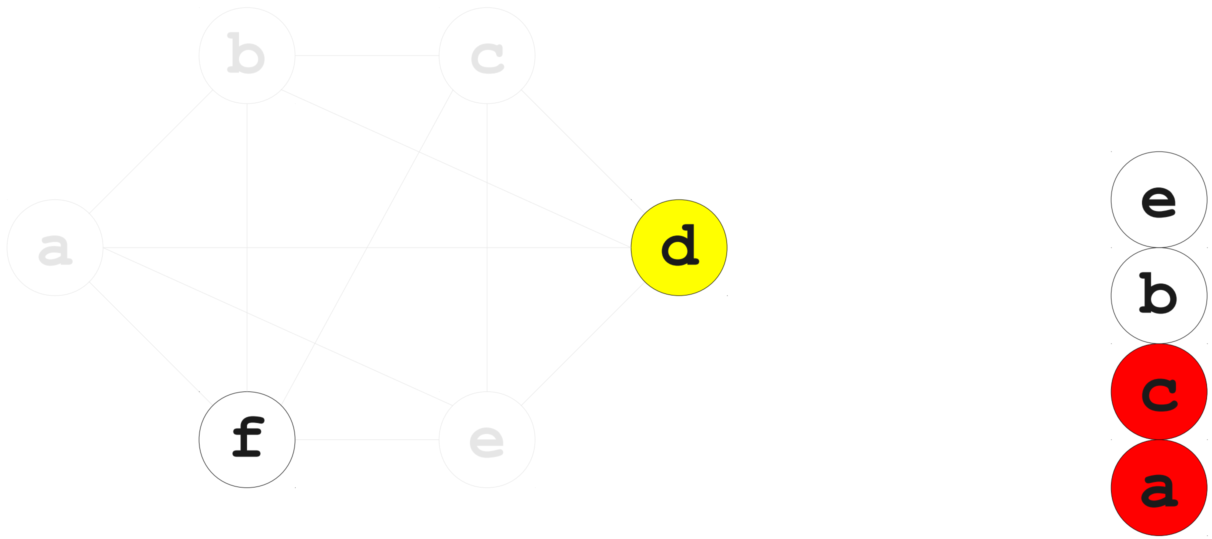
Another Example



Registers

R_0	R_1	R_2

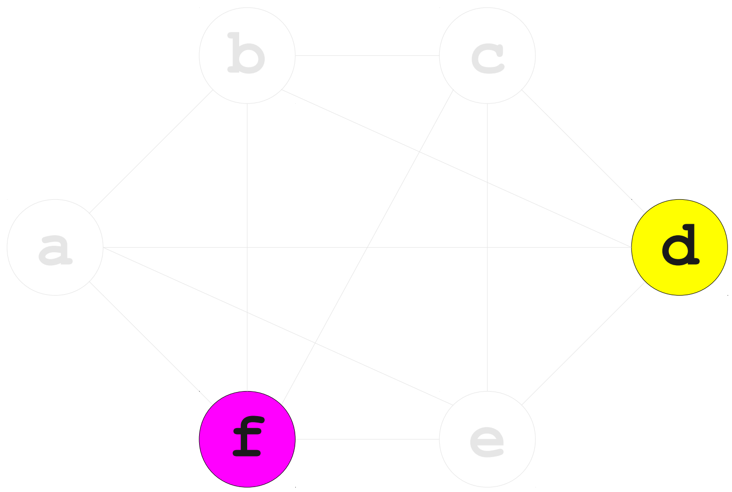
Another Example

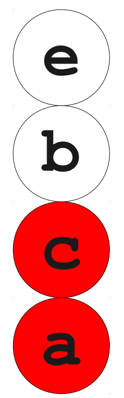


Registers

R_0	R_1	R_2

Another Example

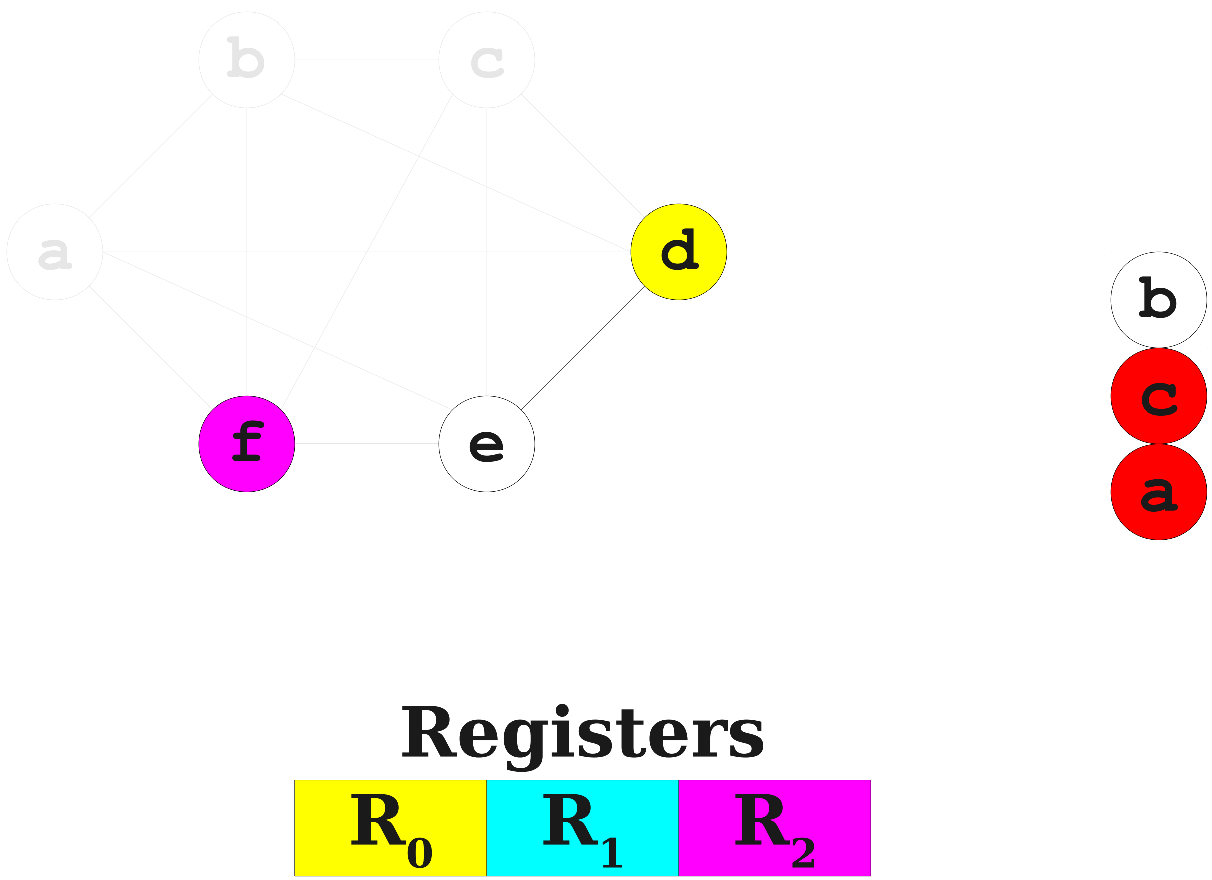




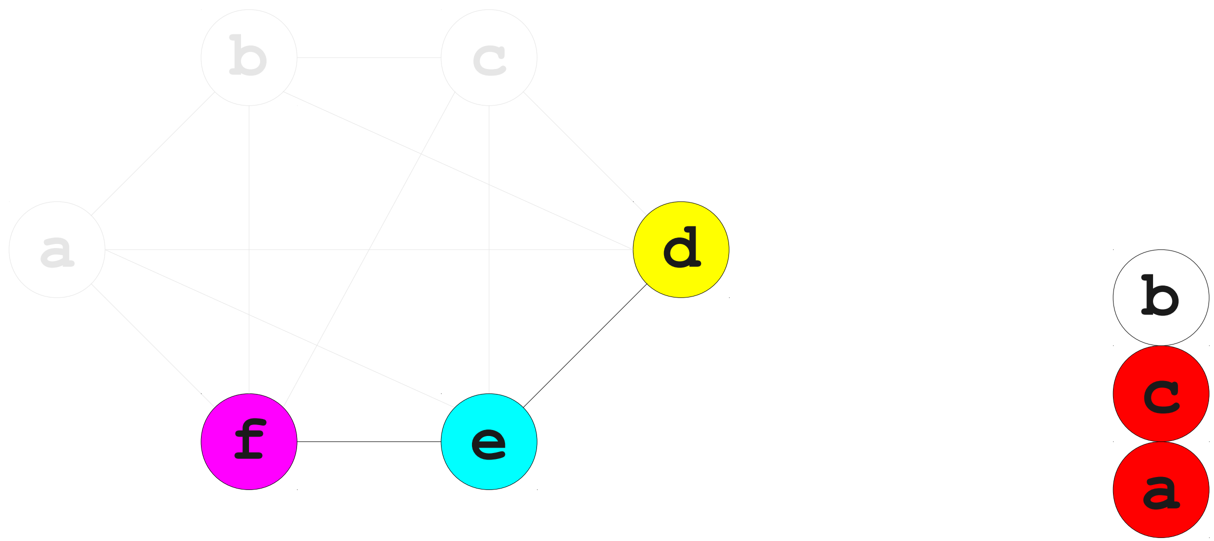
Registers

R_0	R_1	R_2

Another Example



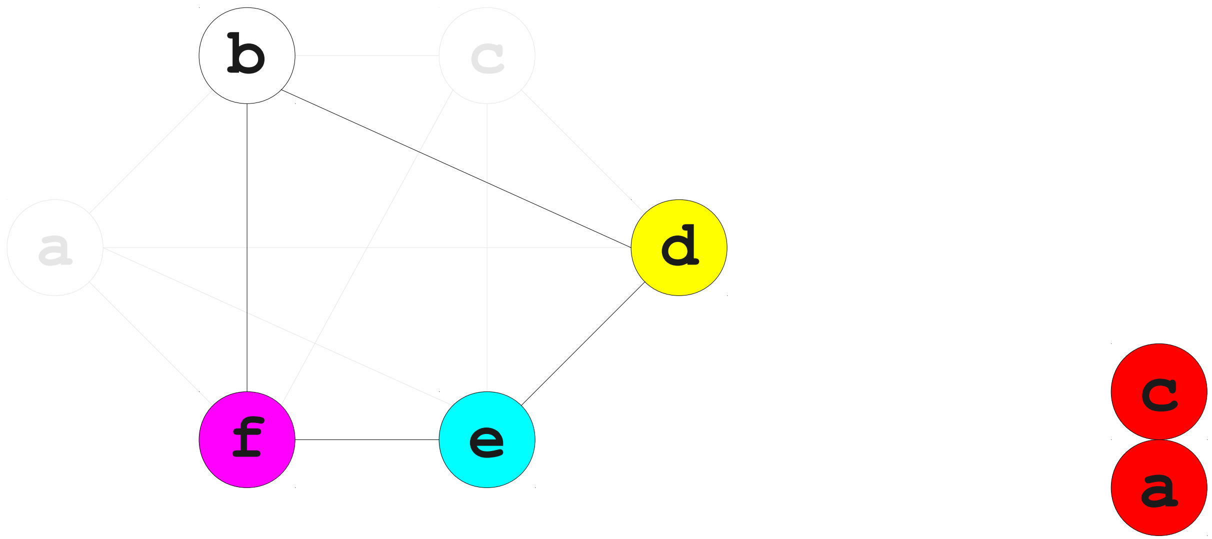
Another Example



Registers

R_0	R_1	R_2

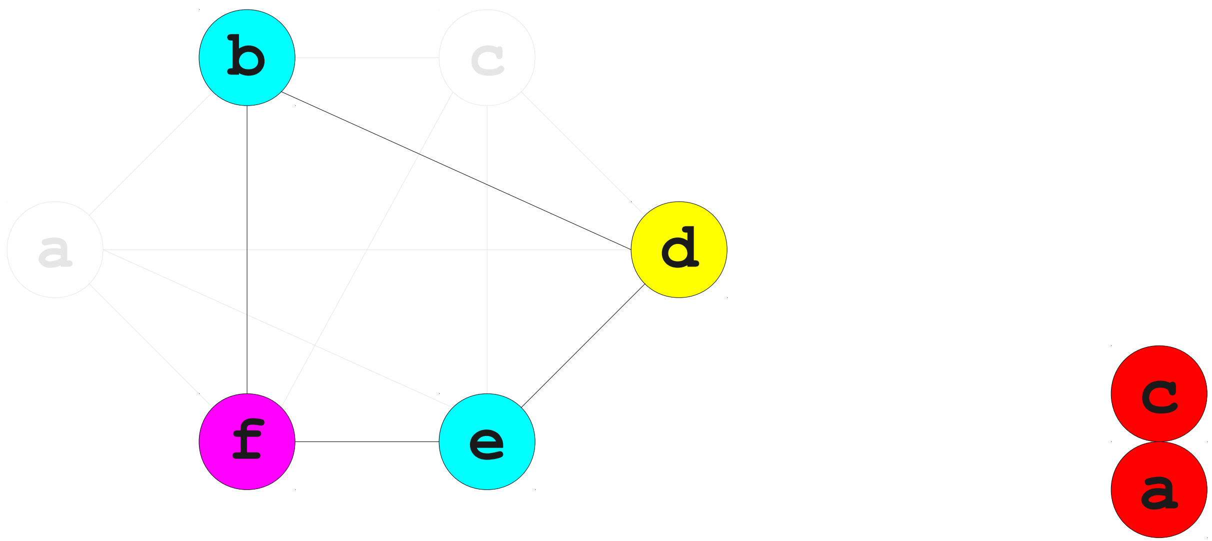
Another Example



Registers

R_0	R_1	R_2

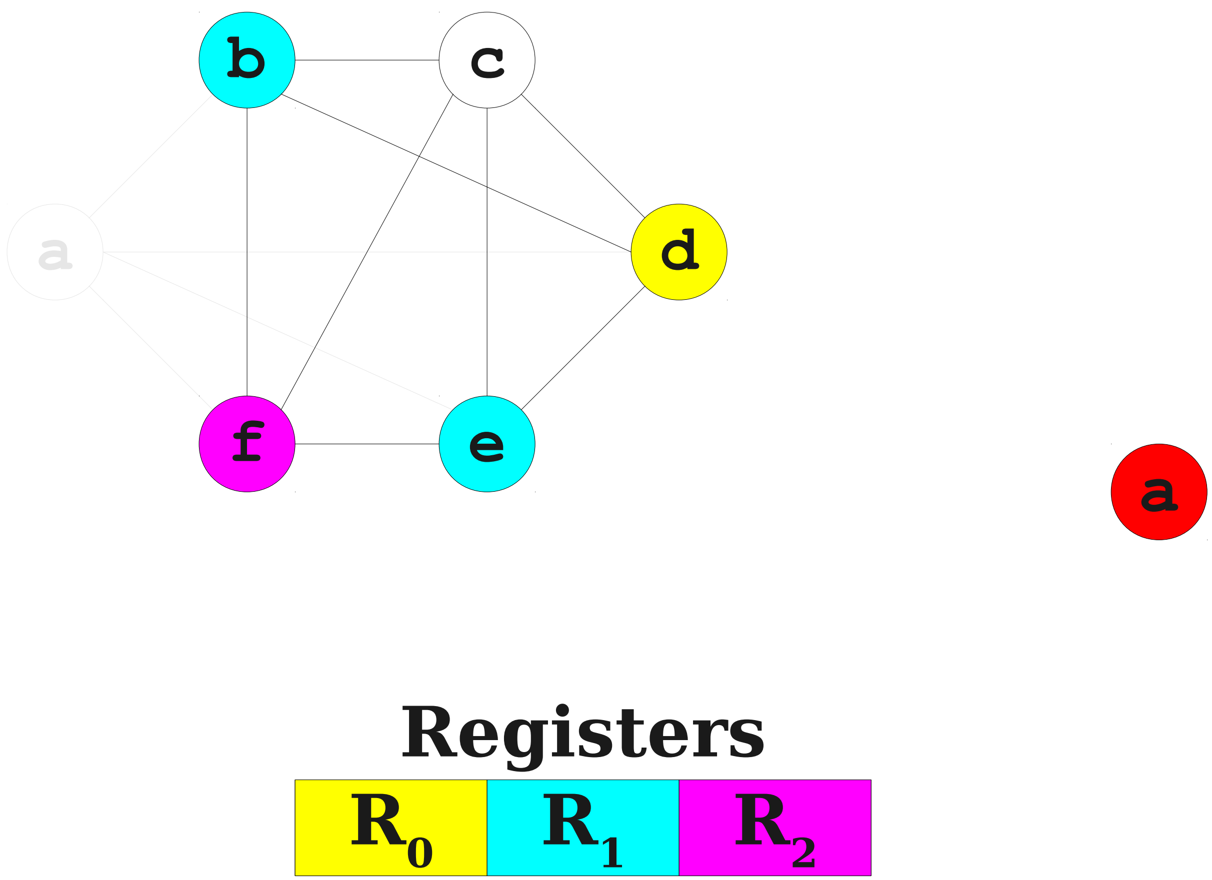
Another Example



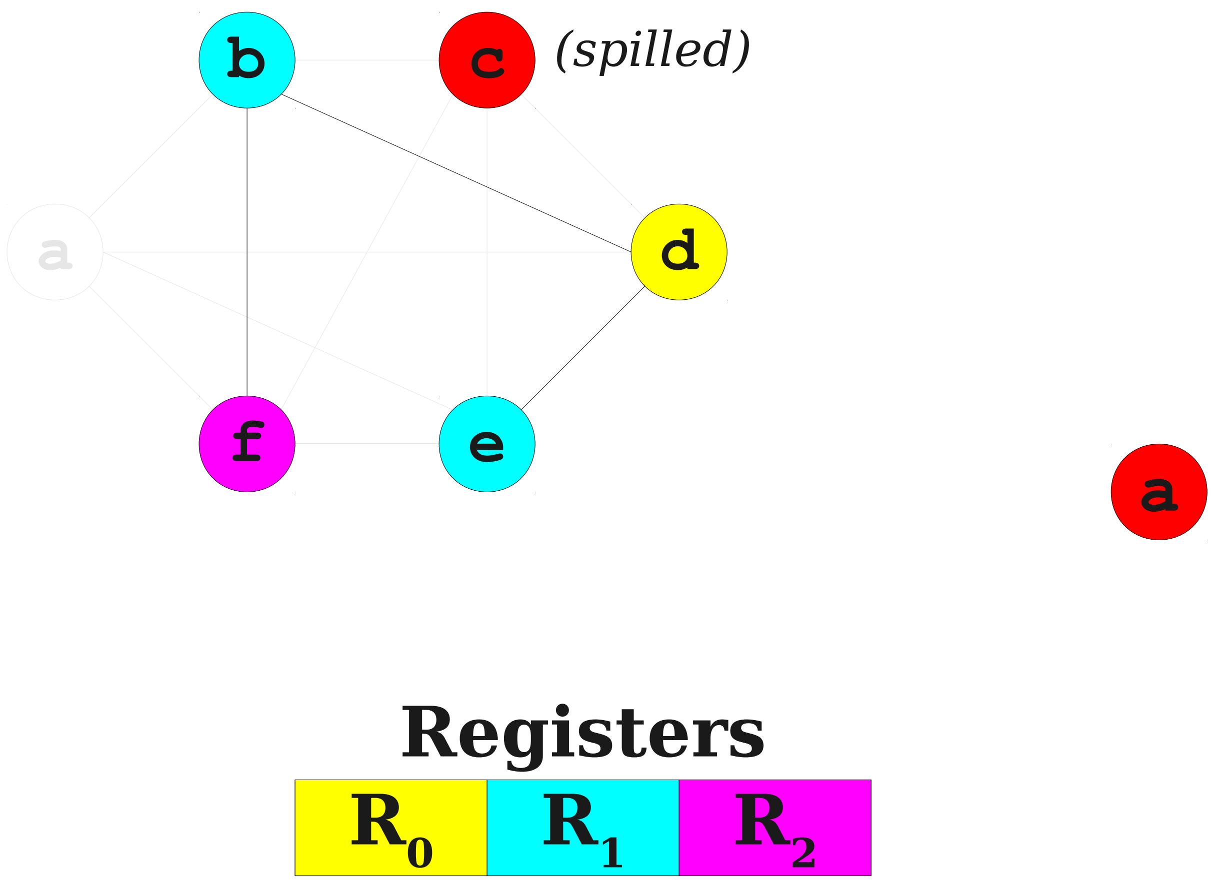
Registers

R_0	R_1	R_2

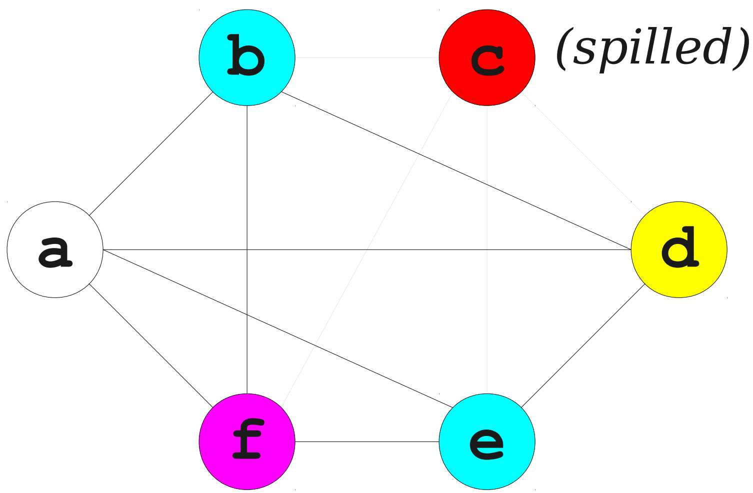
Another Example



Another Example



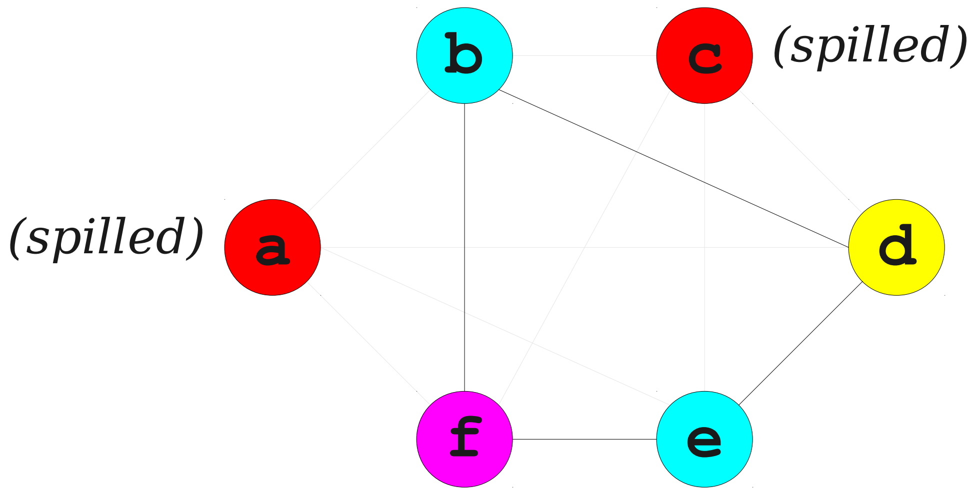
Another Example



Registers

R_0	R_1	R_2

Another Example



Registers

R_0	R_1	R_2

Chaitin's Algorithm

Chaitin's algorithm is efficient ($O(|V| + |E|)$), simple to implement

Chaitin's Algorithm

Chaitin's algorithm is efficient (O(|V| + |E|), simple to implement

- How good the coloring is depends on the order we color the nodes to the graph
  - called the **elimination ordering**

Chaitin's Algorithm

Chaitin's algorithm is efficient ($O(|V| + |E|)$), simple to implement

- How good the coloring is depends on the order we color the nodes to the graph
  - called the **elimination ordering**
- For every graph, there is a elimination ordering such that Chaitin's algorithm produces an optimal coloring
  - therefore finding this optimal elimination ordering for a general graph is NP-complete

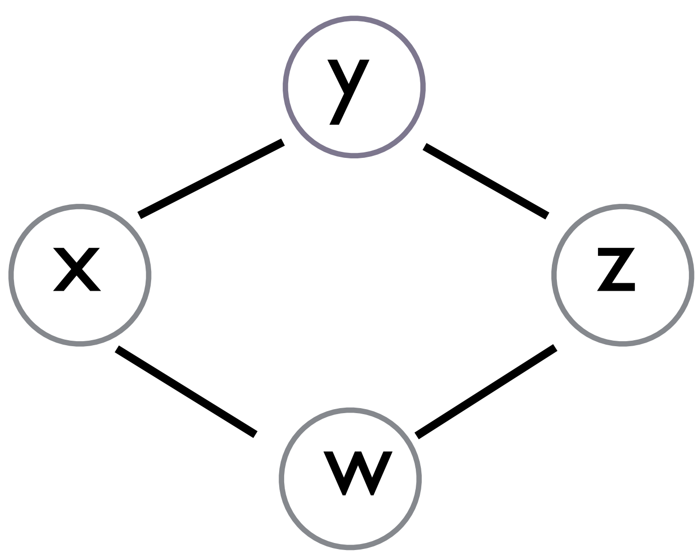
Graph Coloring SSA Programs

Hack et al, "Register Allocation for Programs in SSA-Form", *Compiler Construction* 2006

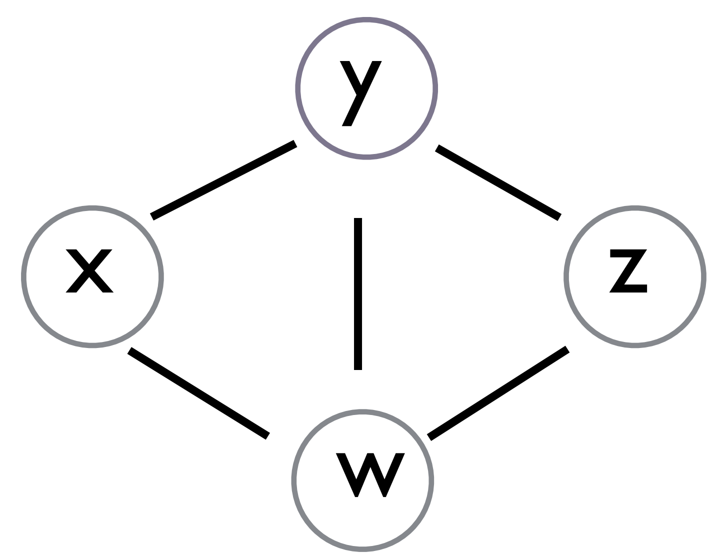
Graph Coloring SSA Programs

Hack et al, "Register Allocation for Programs in SSA-Form", *Compiler Construction* 2006

- The interference graphs of an SSA program are all **chordal**
  - Every cycle >= 4 nodes has a **chord**



Not chordal



chordal

Coloring Chordal Graphs

Theorem: Every chordal graph has a **perfect elimination ordering**

Coloring Chordal Graphs

Theorem: Every chordal graph has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors earlier in the order

Coloring Chordal Graphs

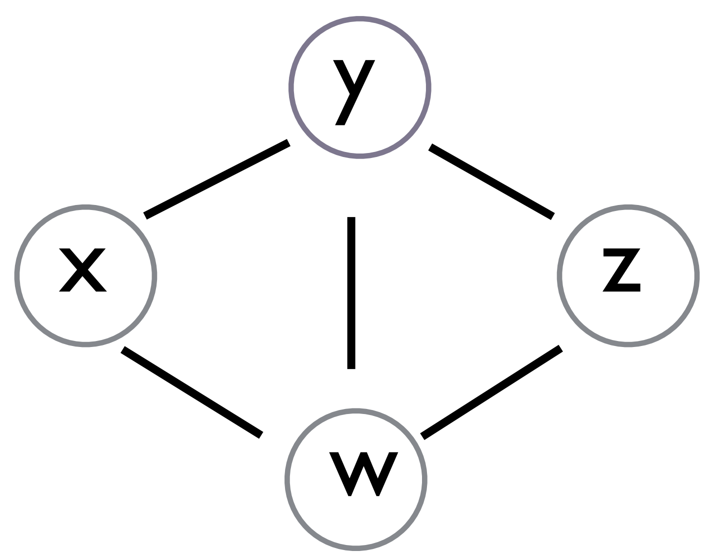
Theorem: Every chordal graph has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors earlier in the order
- Chaitin's algo produces an optimal coloring if we use a PEO

Coloring Chordal Graphs

Theorem: Every chordal graph has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors earlier in the order
- Chaitin's algo produces an optimal coloring if we use a PEO



x,y,z,w
not perfect: N(w) non-clique

w,x,y,z
perfect

Every SSA Interference Graph is Chordal

Theorem: A graph is chordal iff it has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors later in the order

SSA programs have a simple PEO:

"in-scope" or "dominance" relation

Every SSA Interference Graph is Chordal

Theorem: A graph is chordal iff it has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors later in the order

SSA programs have a simple PEO:

"in-scope" or "dominance" relation

- a variable x dominates y if is in scope when y is defined (includes simultaneous defs)

Every SSA Interference Graph is Chordal

Theorem: A graph is chordal iff it has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors later in the order

SSA programs have a simple PEO:

"in-scope" or "dominance" relation

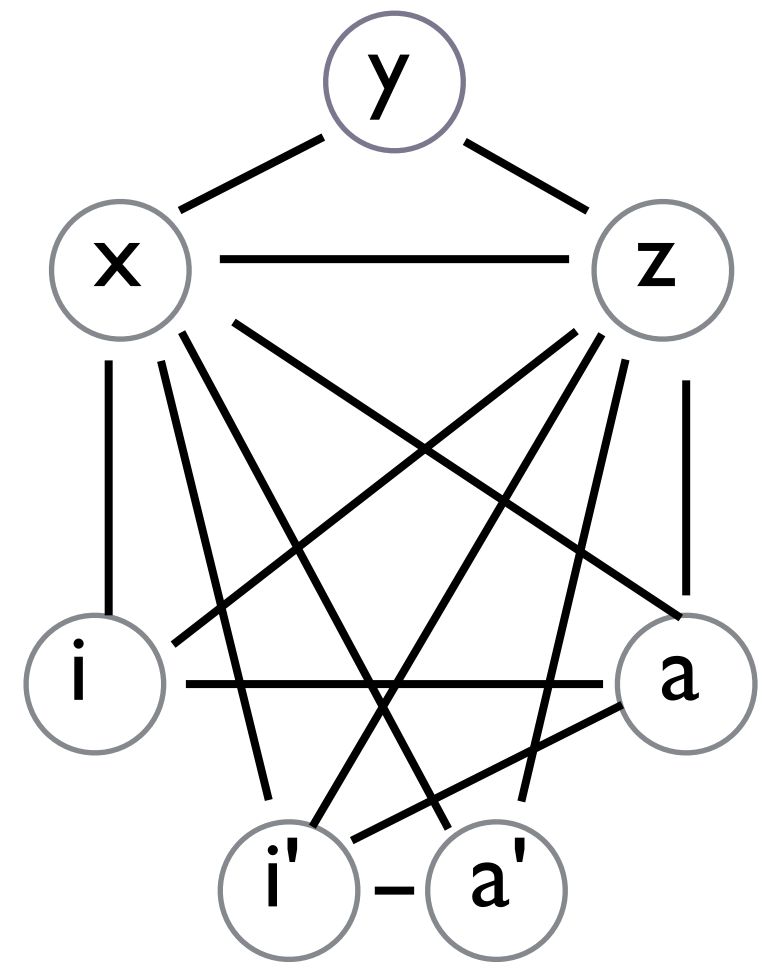
  - a variable x dominates y if is in scope when y is defined (includes simultaneous defs)

- x's definition is "closer to the root" of the AST than y
- easy to compute: pre-order traversal of the nodes

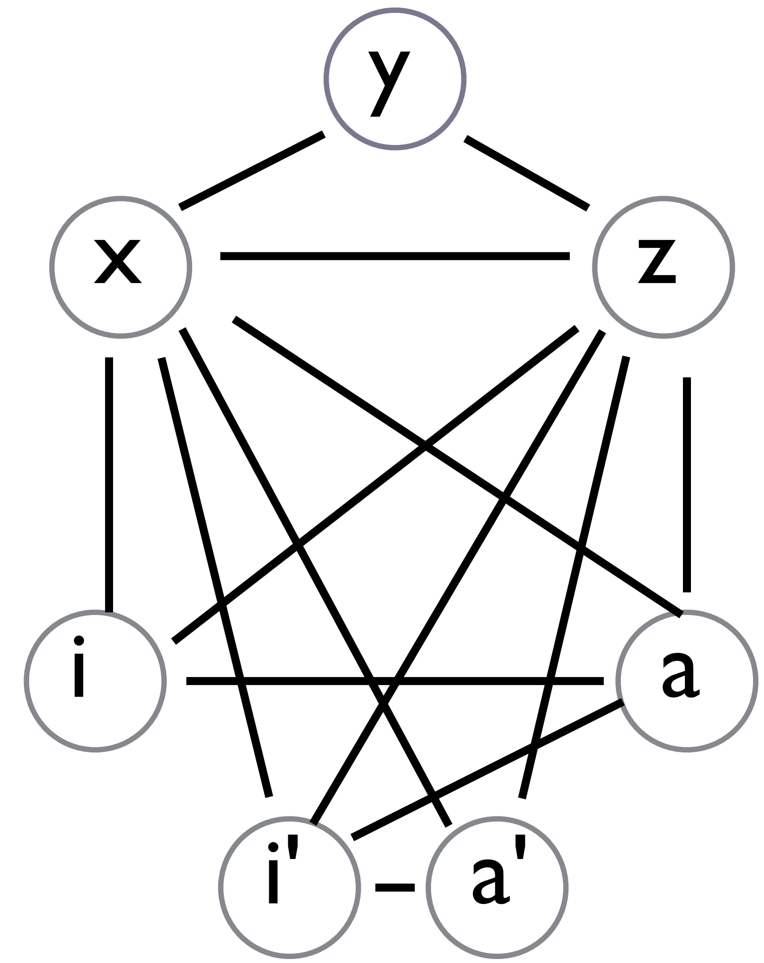
Coloring a Chordal Graph

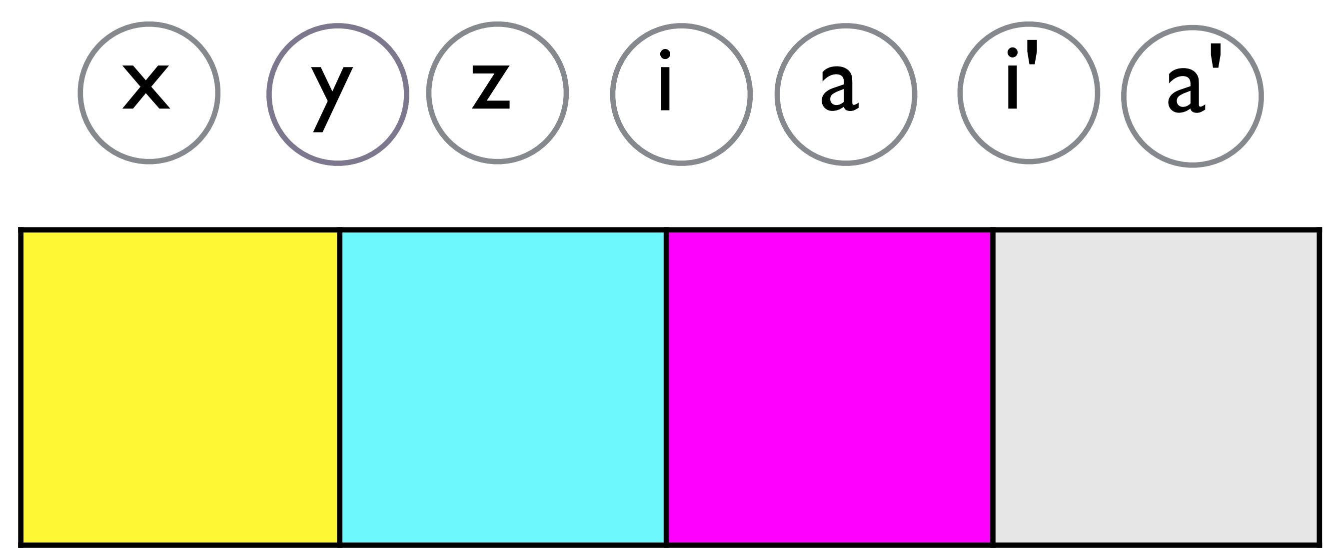
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

Interference Graph

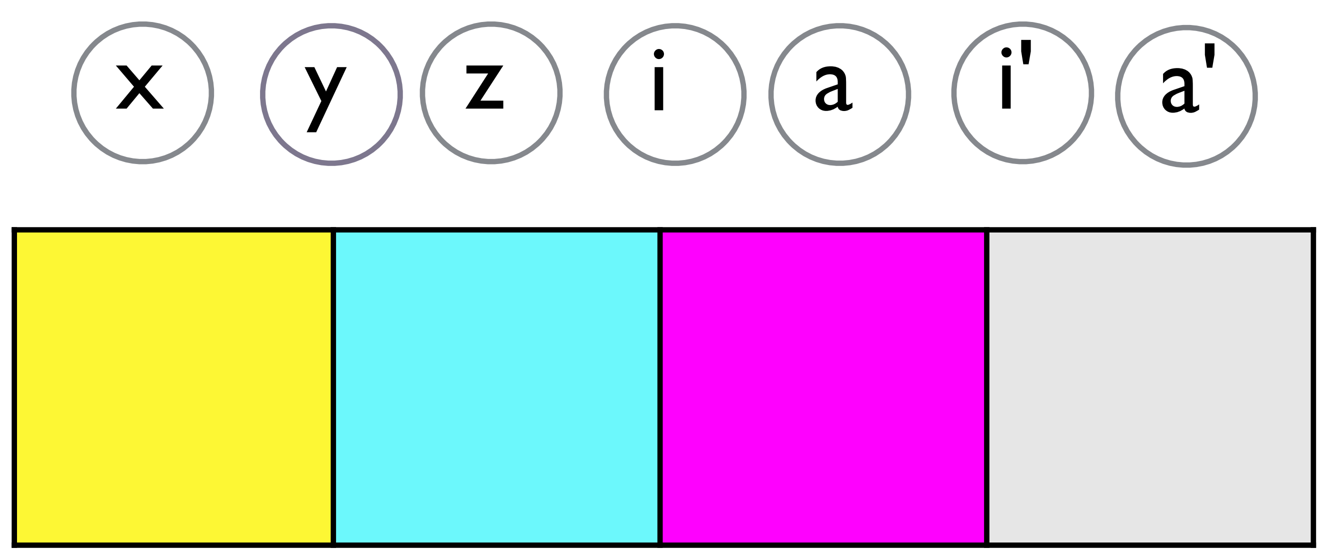


```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

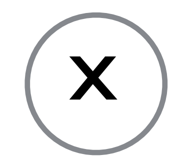


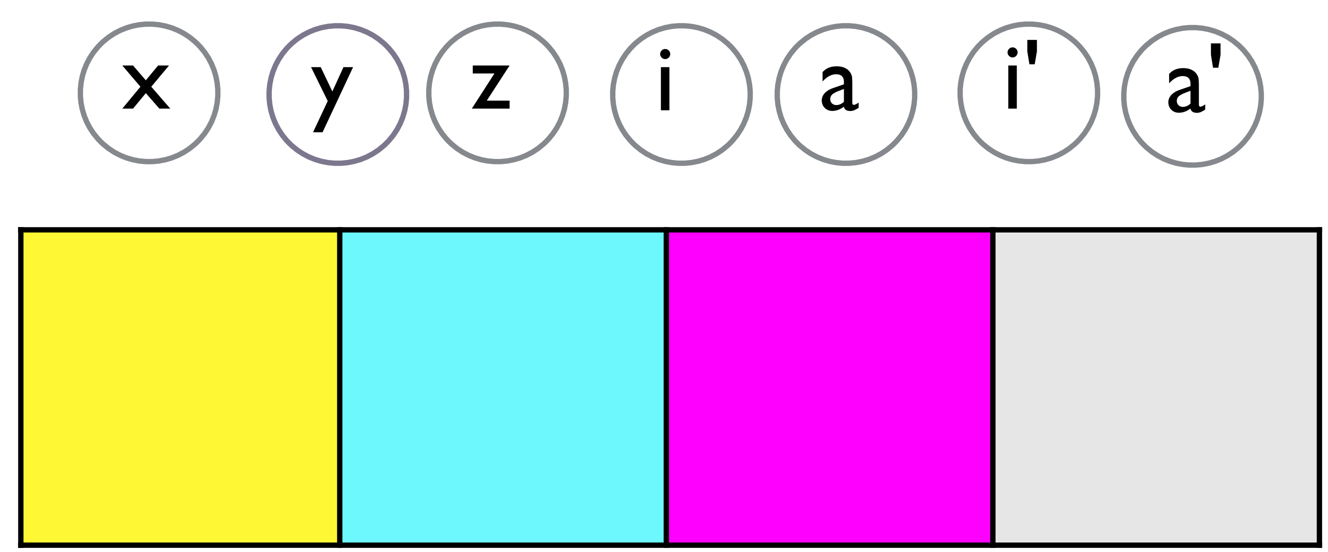


```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

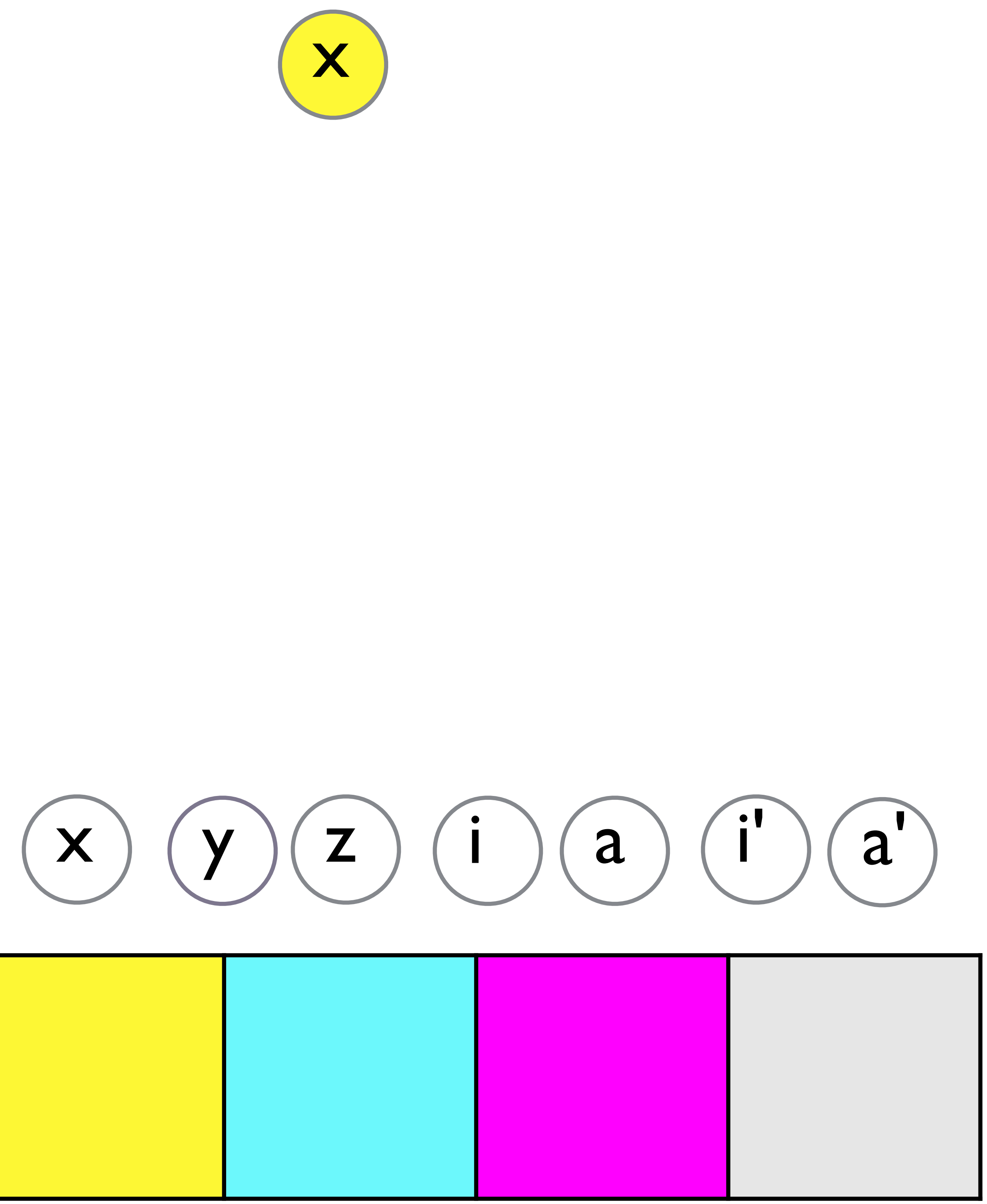


```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

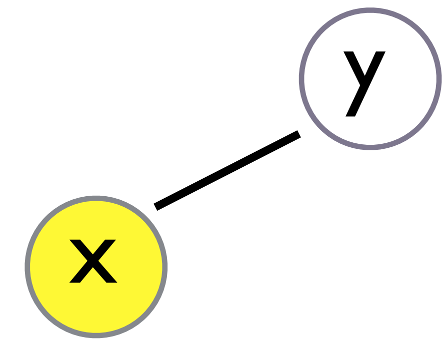


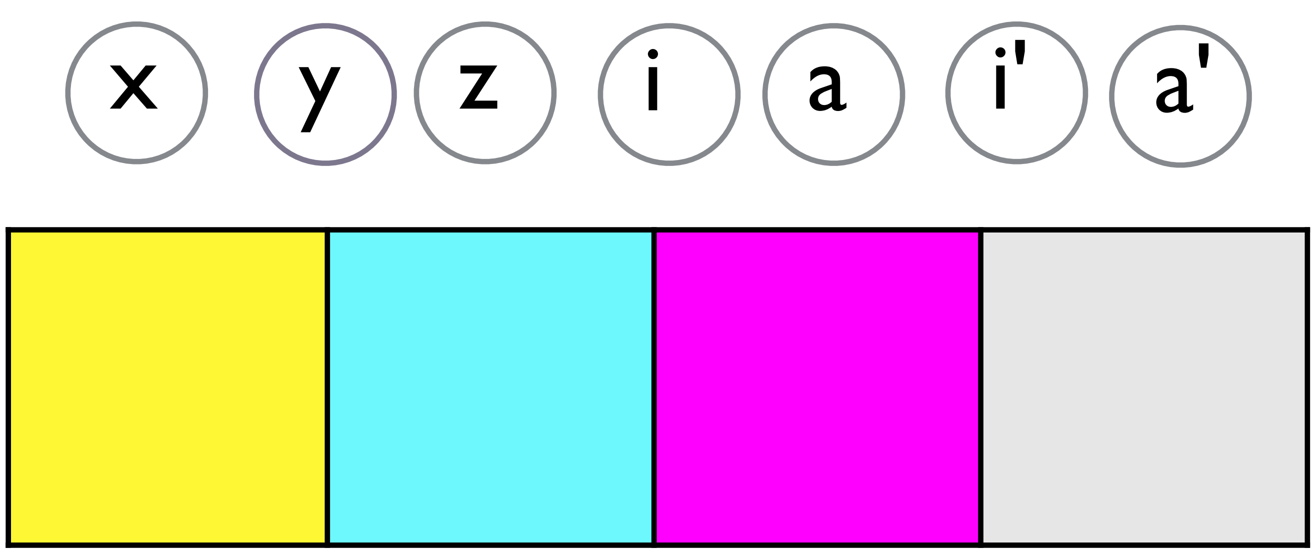


```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

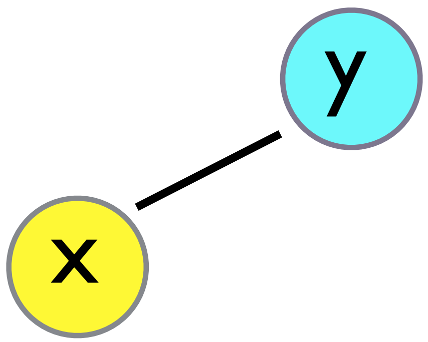


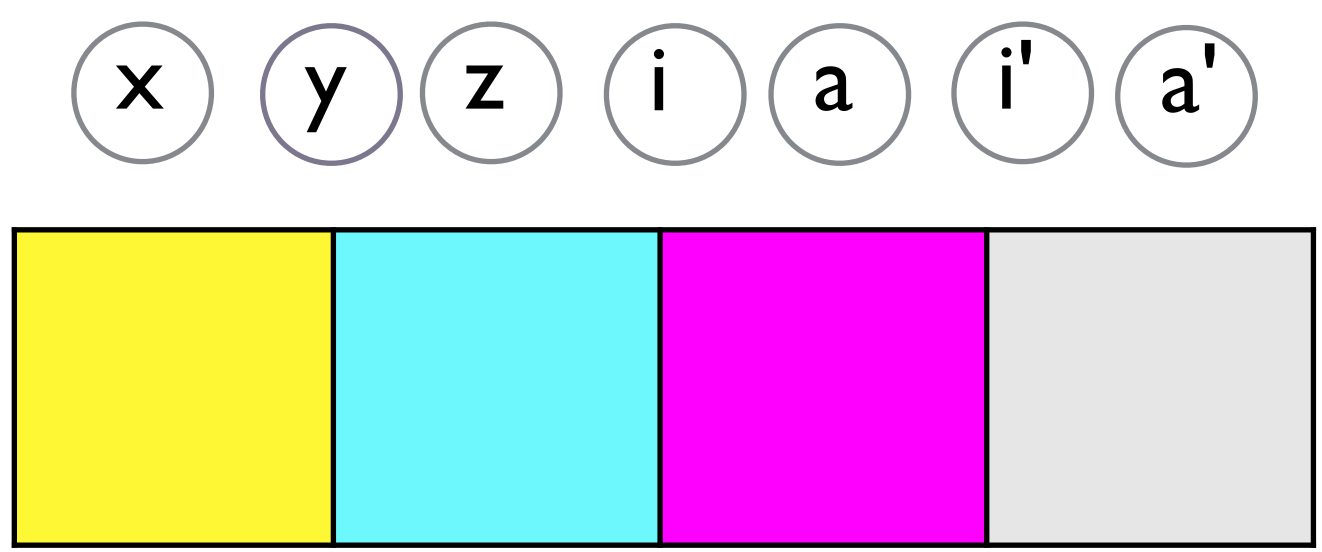
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



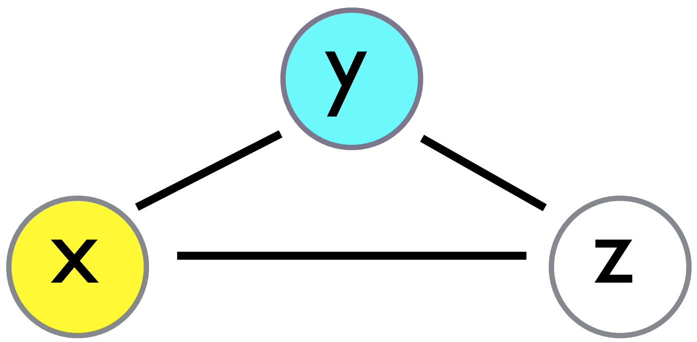


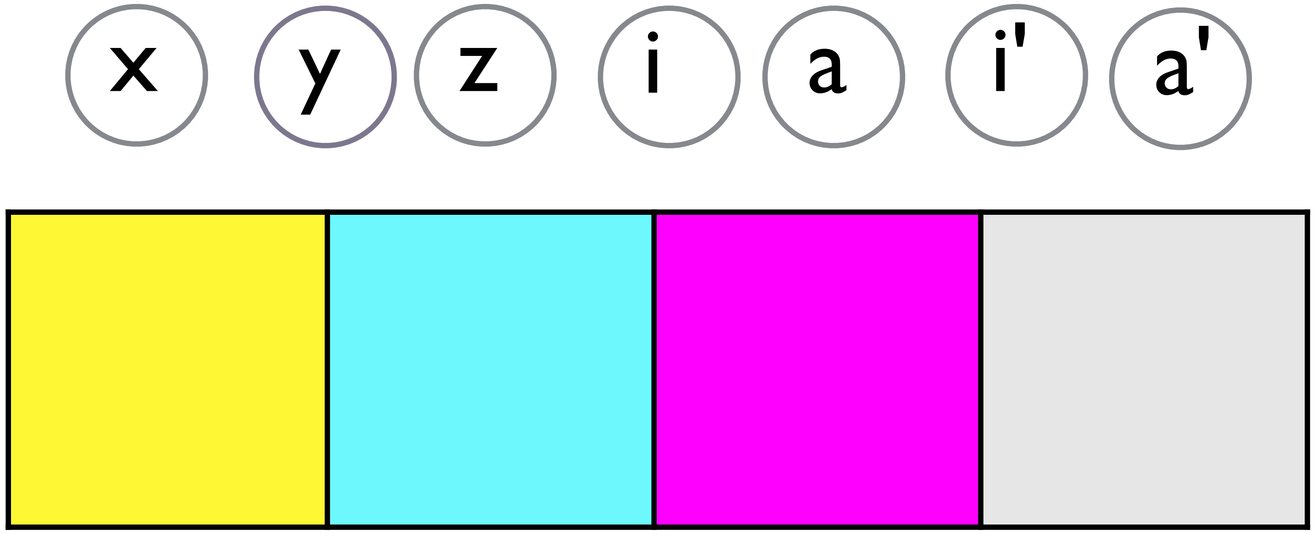
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



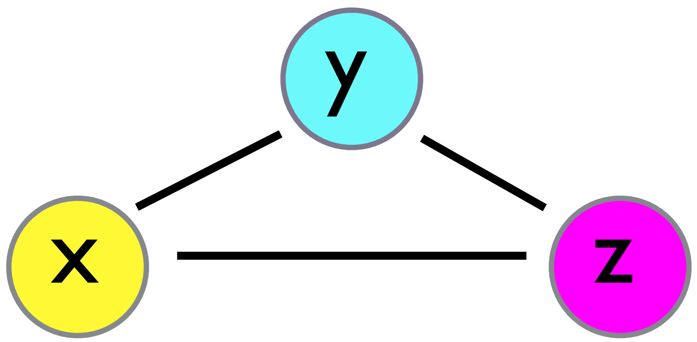


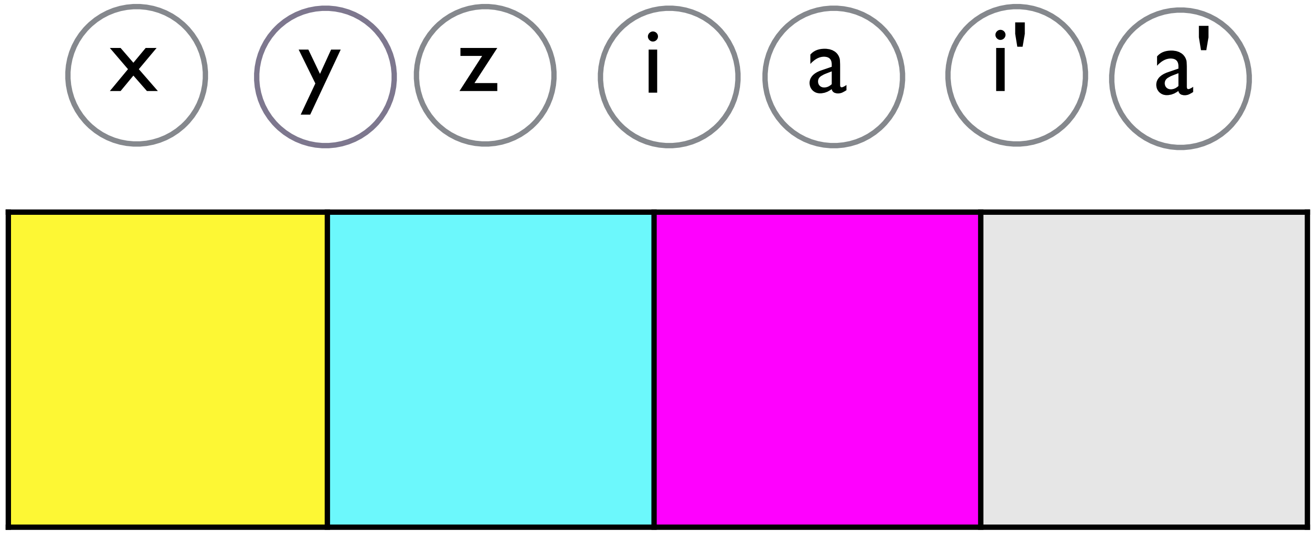
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



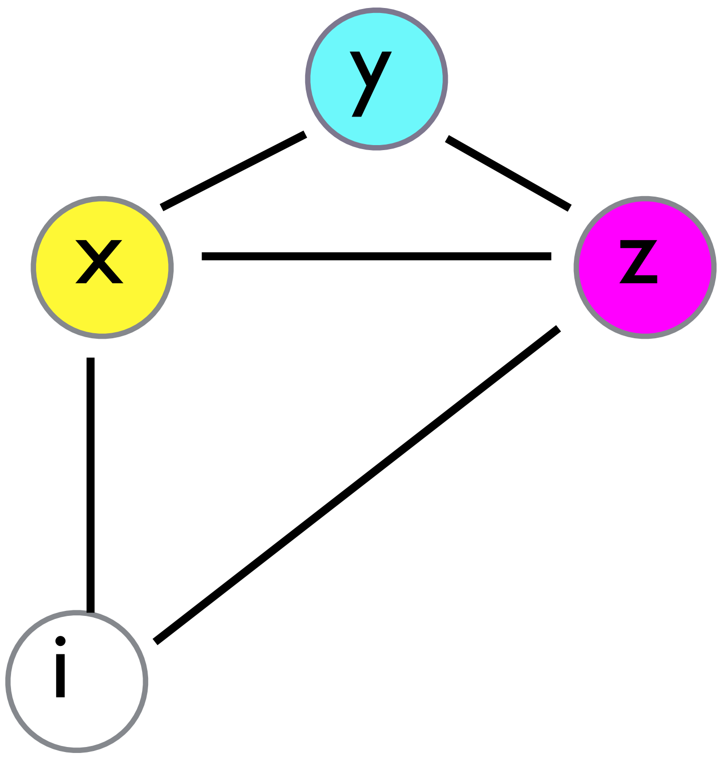


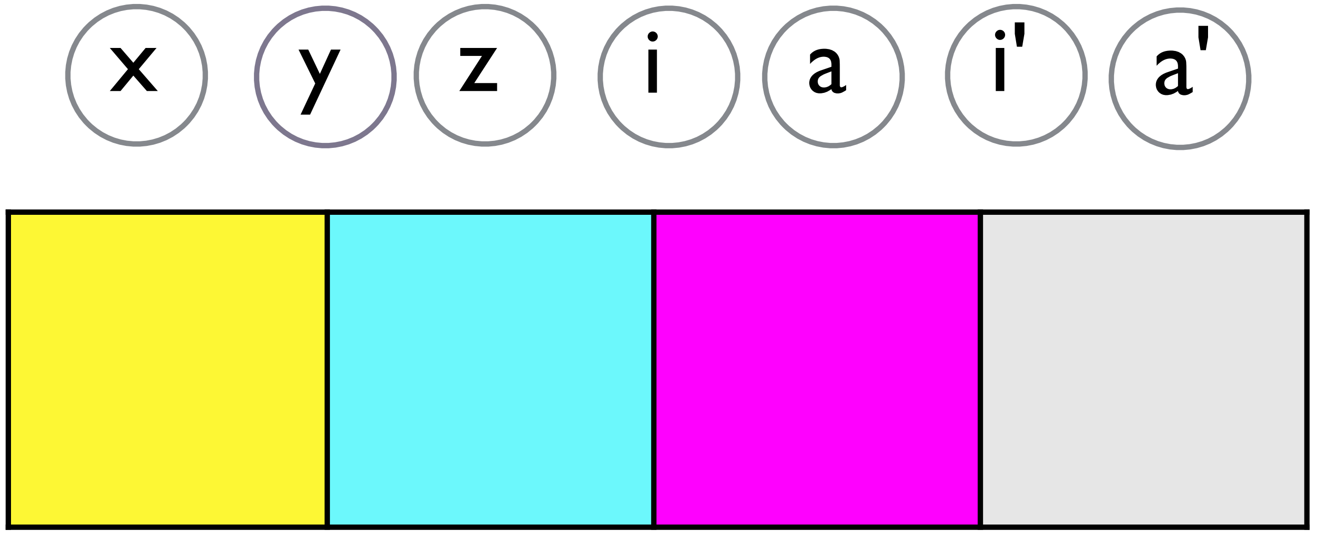
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



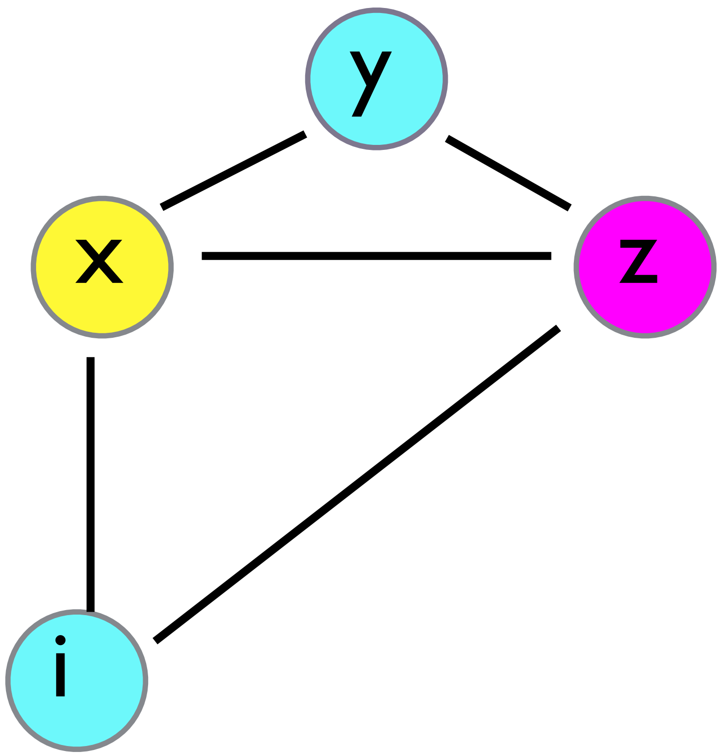


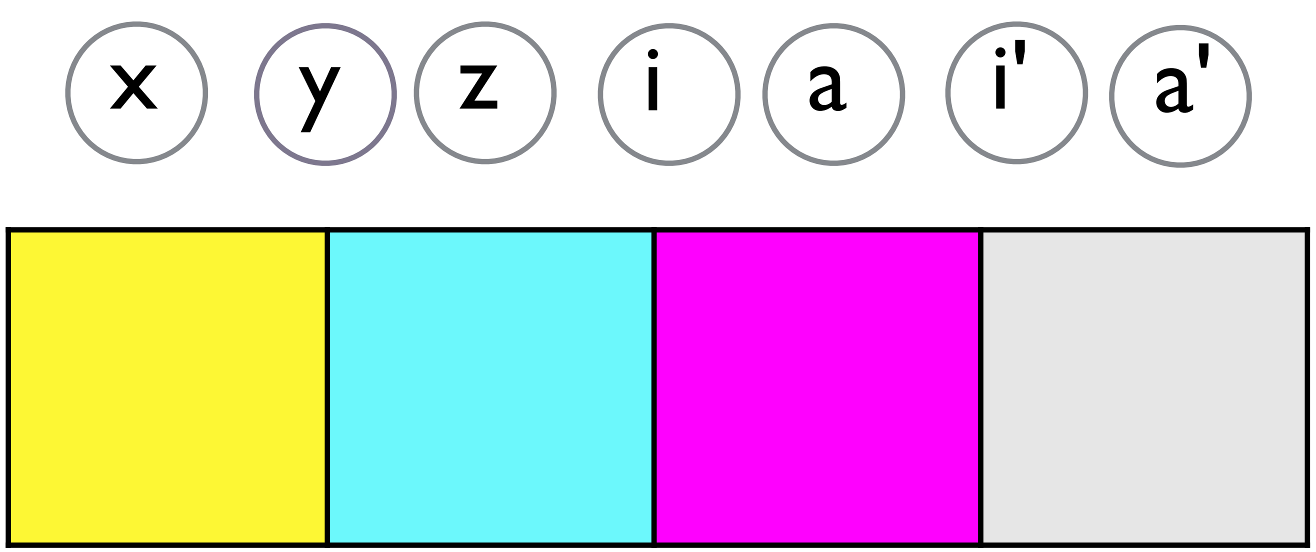
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



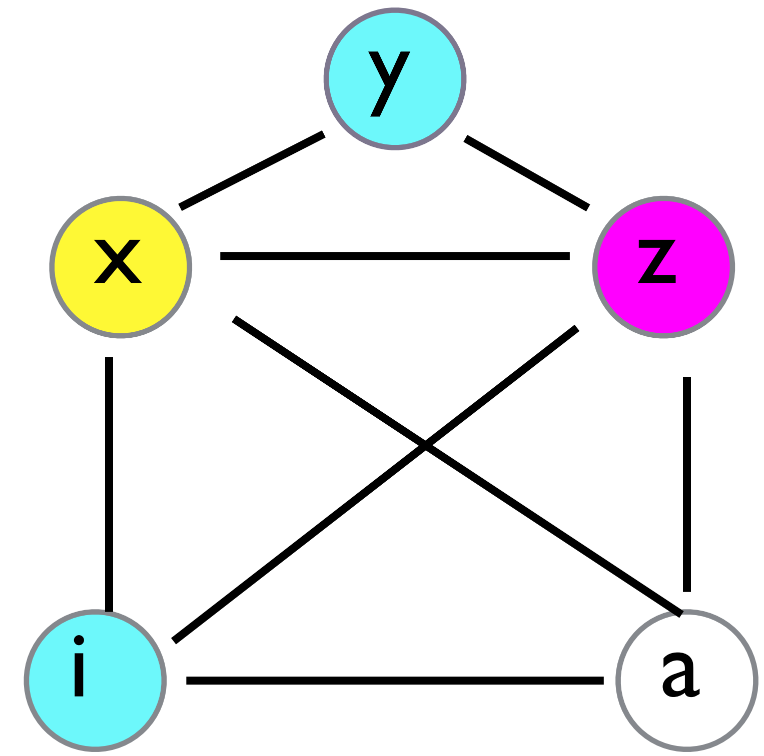


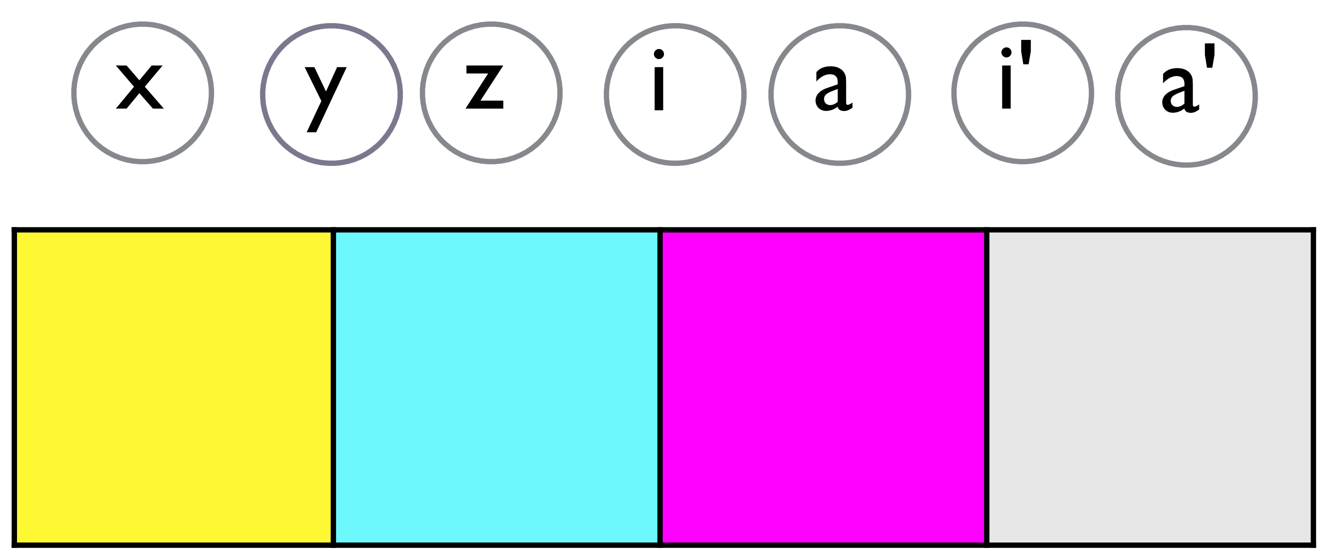
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



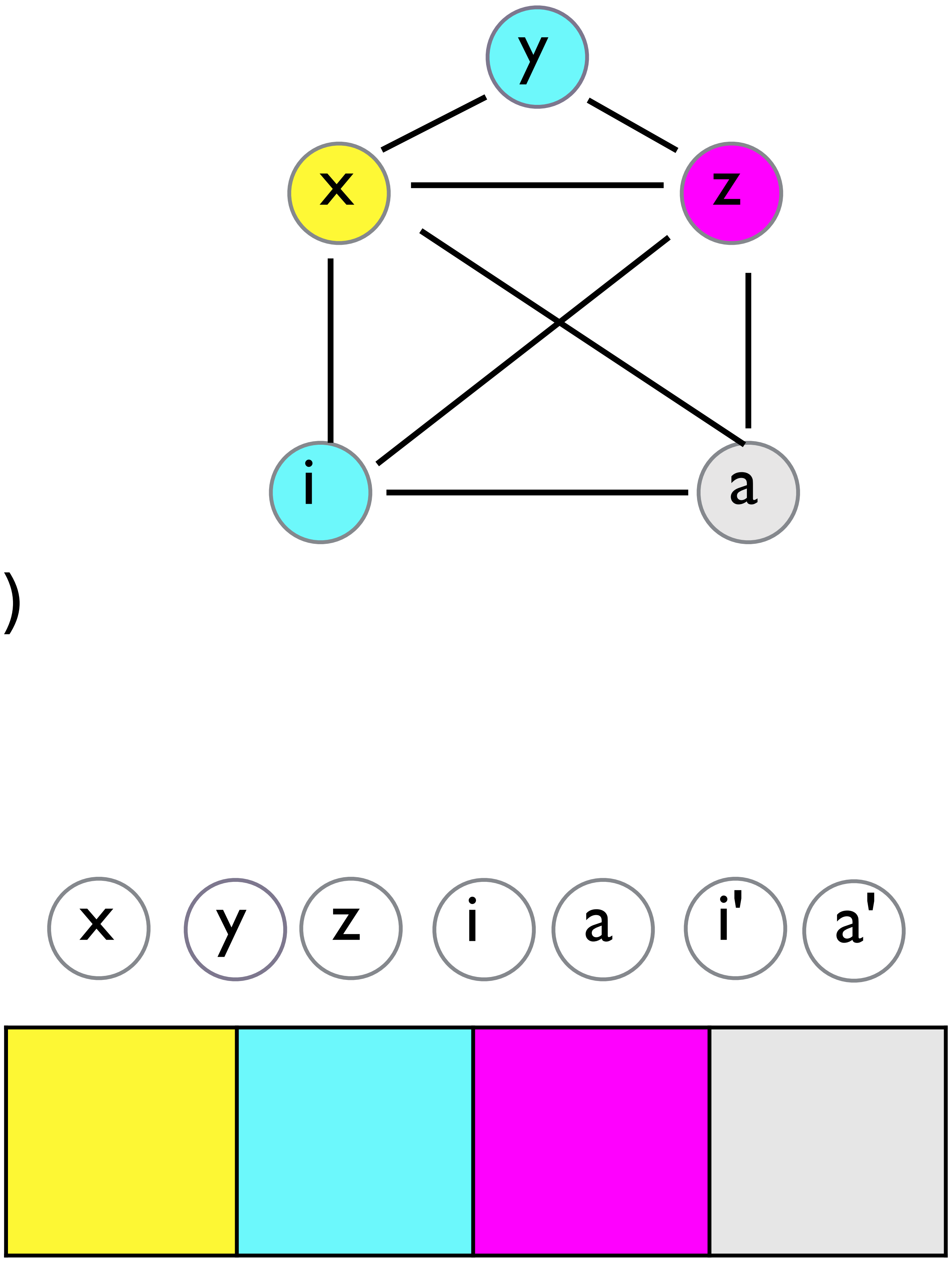


```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

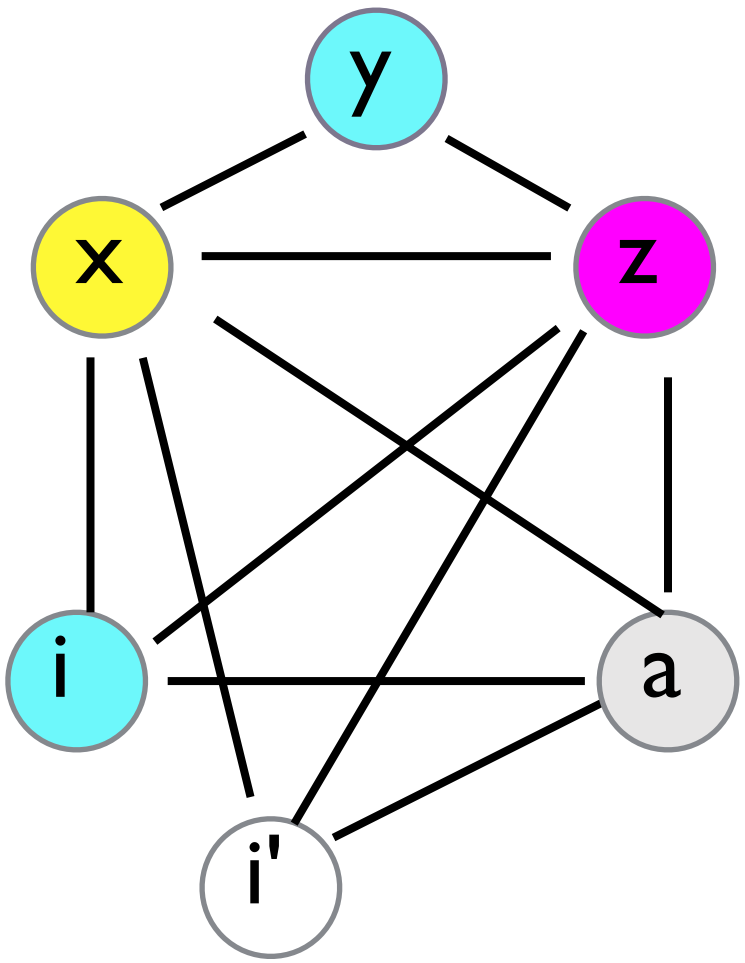


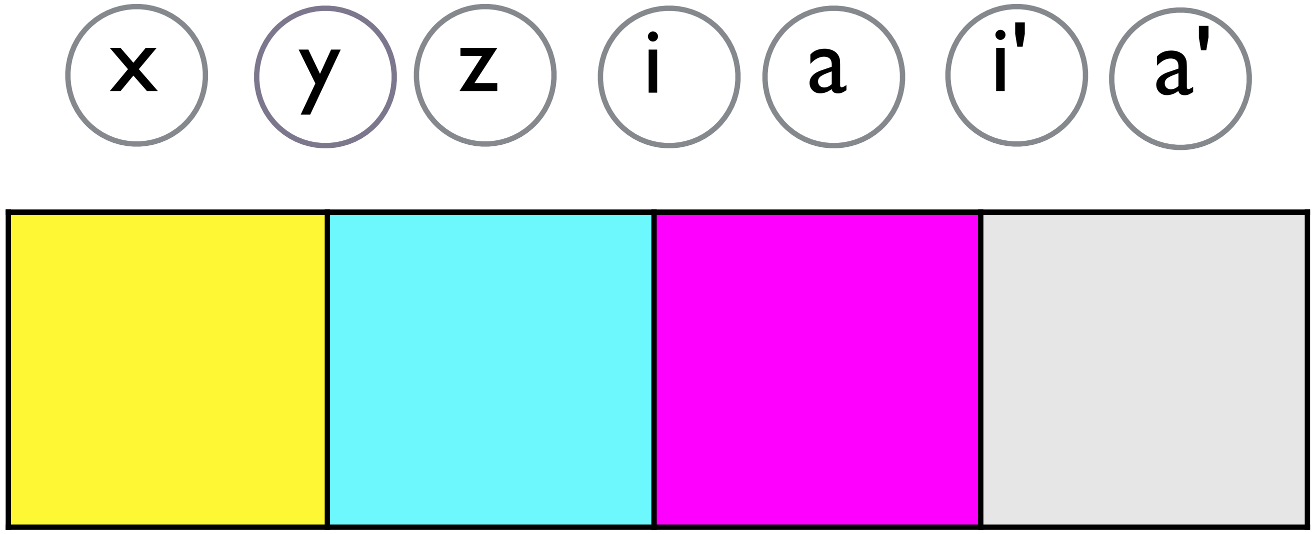


```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```

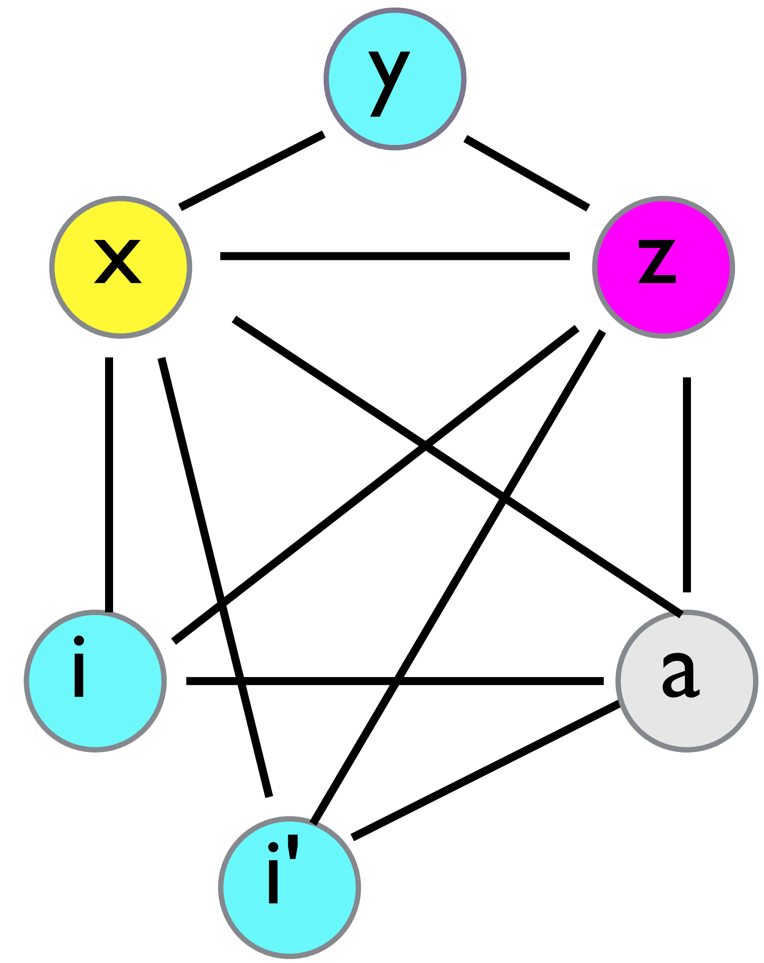


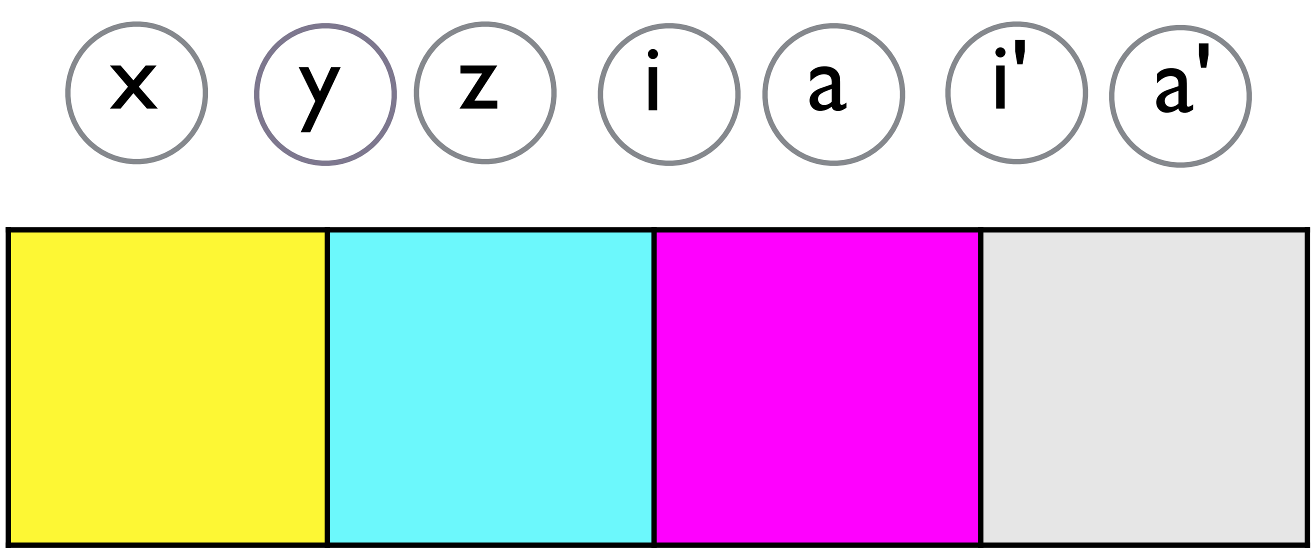
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



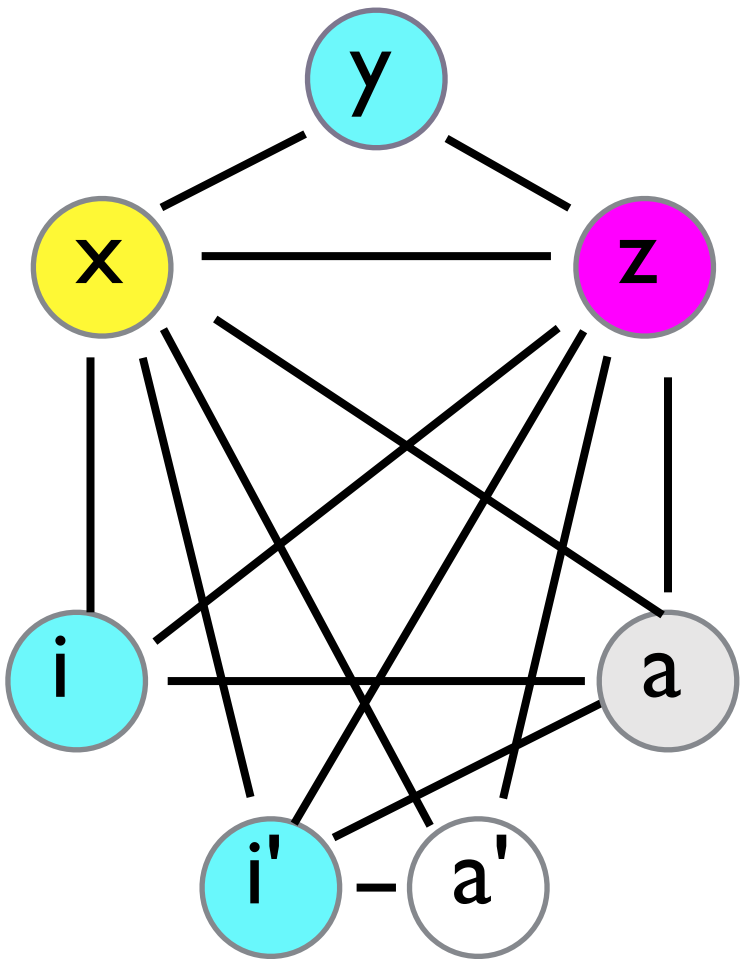


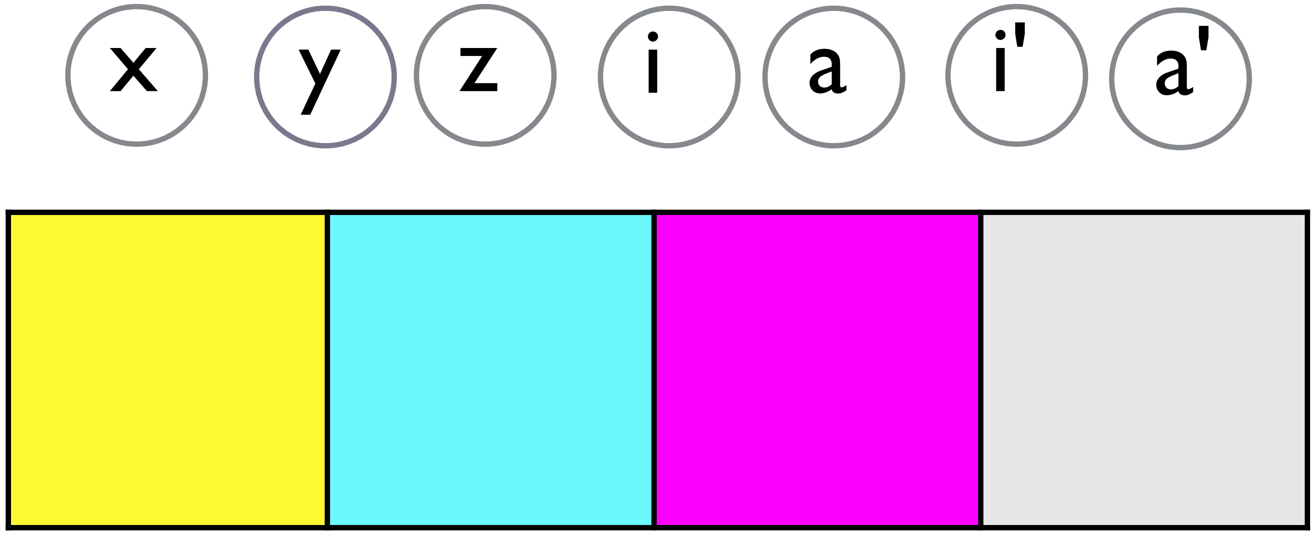
```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



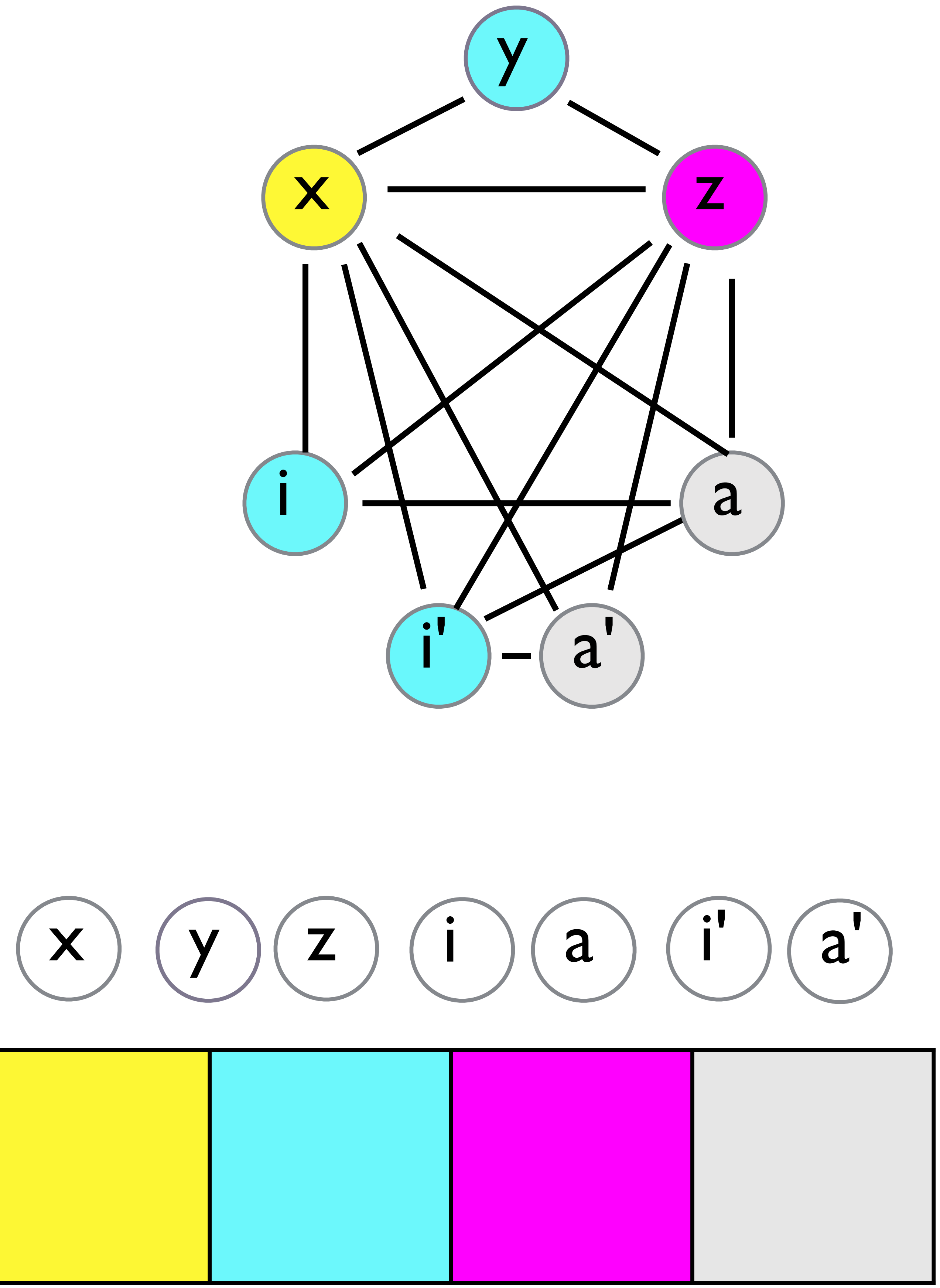


```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```





```
def f(x,y,z):
  def loop(i,a):
    if i == 0:
      a * z
    else:
      let i' = i - 1 in
      let a' = a + x in
      icall(loop; i', a')
  end
  icall(loop; y, 0)
```



Using Register Assignment

1. For each function definition, we'll run liveness, conflict analysis and register allocation, producing a mapping from variable names to registers/stack offsets.
2. How does your code generation change?

Effects on Codegen

- No longer always put result in RAX: put result in

let x = y * z

in ...

Want the result of y * z to go wherever x is stored, not RAX.

Implementing Local Tail Calls

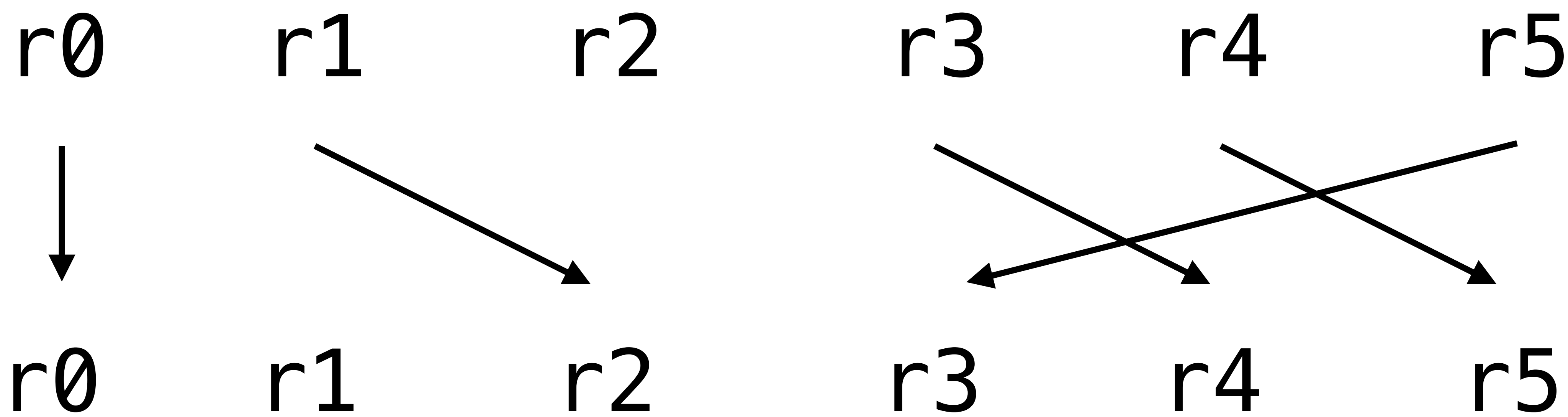
```
def f(a,b,c): e in
...
local_tail_call(f; [x,y,z])
```

Implementing Local Tail Calls

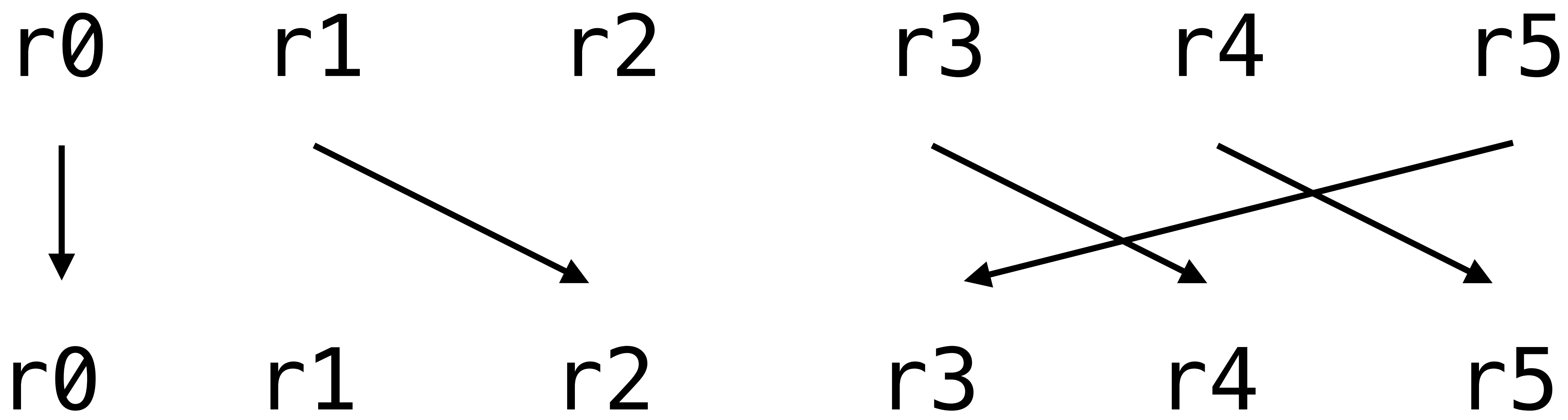
```
def f(a,b,c): e in
...
local_tail_call(f; [x,y,z])
```

```
mov r_x, r_a
mov r_y, r_b
mov r_z, r_z
jmp f
```

Implementing Local Tail Calls

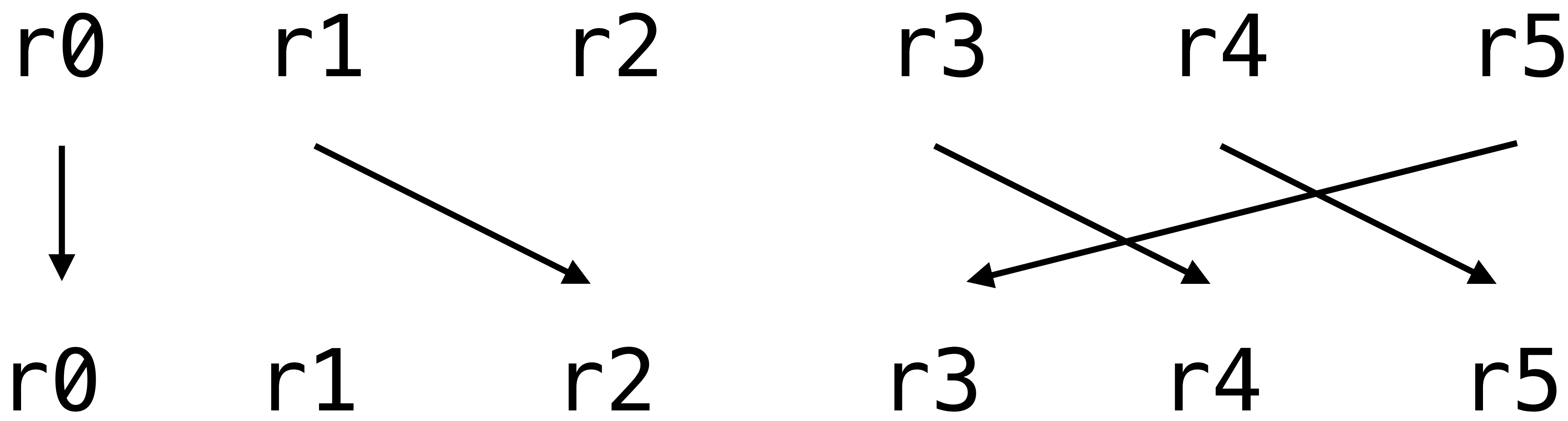


Implementing Local Tail Calls



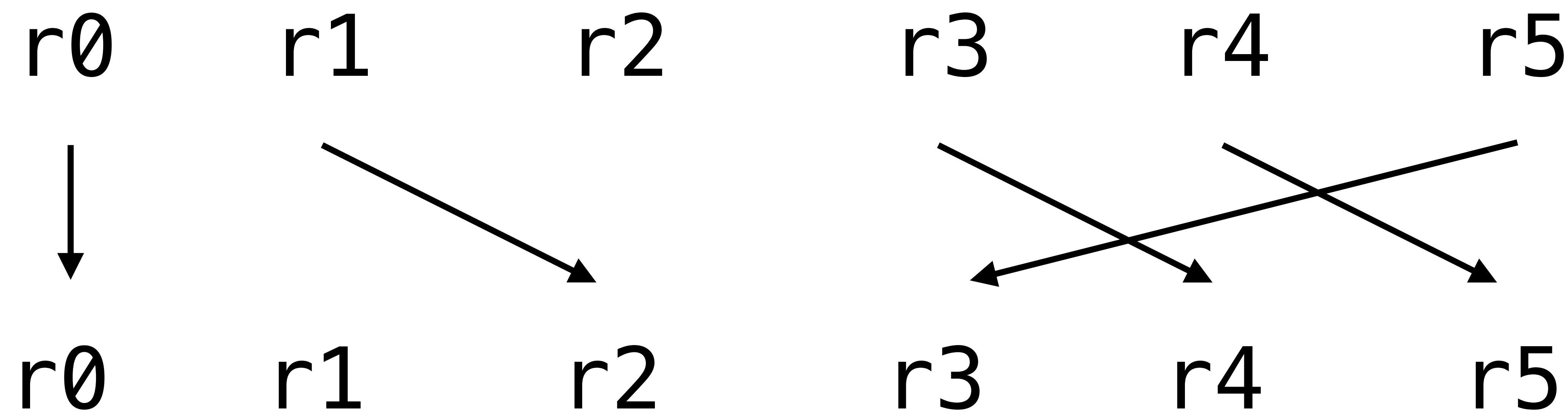
nop

Implementing Local Tail Calls




```
nop  mov r2, r1
```

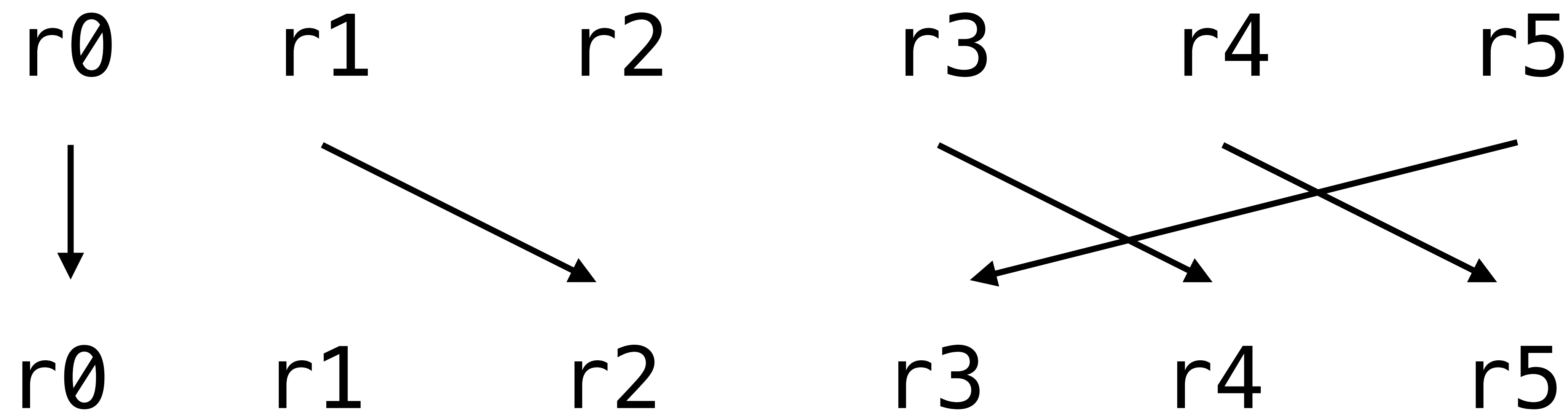
Implementing Local Tail Calls




```
nop  mov r2, r1
```

```
xchg rax, r3
xchg rax, r4
xchg rax, r5
xchg rax, r3
```

Implementing Local Tail Calls




```
nop  mov r2, r1
```

```
xchg rax, r3
xchg rax, r4
xchg rax, r5
xchg rax, r3
```

SSA reg allocation is polytime, but minimizing the resulting number of movs/xchg is NP hard

Register Allocation vs Calling Conventions

Now that we are using registers we need to take care to respect treatment of registers in the calling conventions we use.

Register Allocation vs Calling Conventions

Now that we are using registers we need to take care to respect treatment of registers in the calling conventions we use.

In System V AMD 64 Calling convention, registers are divided into two classes:

Register Allocation vs Calling Conventions

Now that we are using registers we need to take care to respect treatment of registers in the calling conventions we use.

In System V AMD 64 Calling convention, registers are divided into two classes:

- **volatile** aka **caller-save**: when you make a call, the value of these registers may change when the callee returns

Register Allocation vs Calling Conventions

Now that we are using registers we need to take care to respect treatment of registers in the calling conventions we use.

In System V AMD 64 Calling convention, registers are divided into two classes:

- **volatile** aka **caller-save**: when you make a call, the value of these registers may change when the callee returns
- **non-volatile** aka **callee-save**: when you make a call, the value of these registers will be the same when the callee returns

Volatile/Caller Save registers

volatile aka **caller-save**

let x = ... in

let y = f(z) in

x + y

if **x** is stored in a volatile register, its value may be overwritten by the call.

Volatile/Caller Save registers

volatile aka **caller-save**

let x = ... in

let y = f(z) in

x + y

if **x** is stored in a volatile register, its value may be overwritten by the call.

- Easy solution: save all live volatiles to the stack before a call, restore after the call

Volatile/Caller Save registers

volatile aka **caller-save**

let x = ... in

let y = f(z) in

x + y

if **x** is stored in a volatile register, its value may be overwritten by the call.

- Easy solution: save all live volatiles to the stack before a call, restore after the call
- Harder solution: add nodes to interference graph for volatile registers, add conflicts at every non-tail call

Non-volatile/Callee Save registers

```
def f(x):
  ...
  let y = ... in
  let z = x + y in z
```

if **y** is stored in a **non-volatile** register, its value must be **restored** when we return

Non-volatile/Callee Save registers

```
def f(x):
  ...
  let y = ... in
  let z = x + y in z
```

if **y** is stored in a **non-volatile** register, its value must be **restored** when we return

- Easy solution: save all non-volatiles to the stack at the beginning of every global function def, restore them before every return/external tail call

Non-volatile/Callee Save registers

```
def f(x):
  ...
  let y = ... in
  let z = x + y in z
```

if **y** is stored in a **non-volatile** register, its value must be **restored** when we return

- Easy solution: save all non-volatiles to the stack at the beginning of every global function def, restore them before every return/external tail call
- Harder solution: treat non-volatiles as "hidden args" of global fundefs, with ret/tail calls as uses.