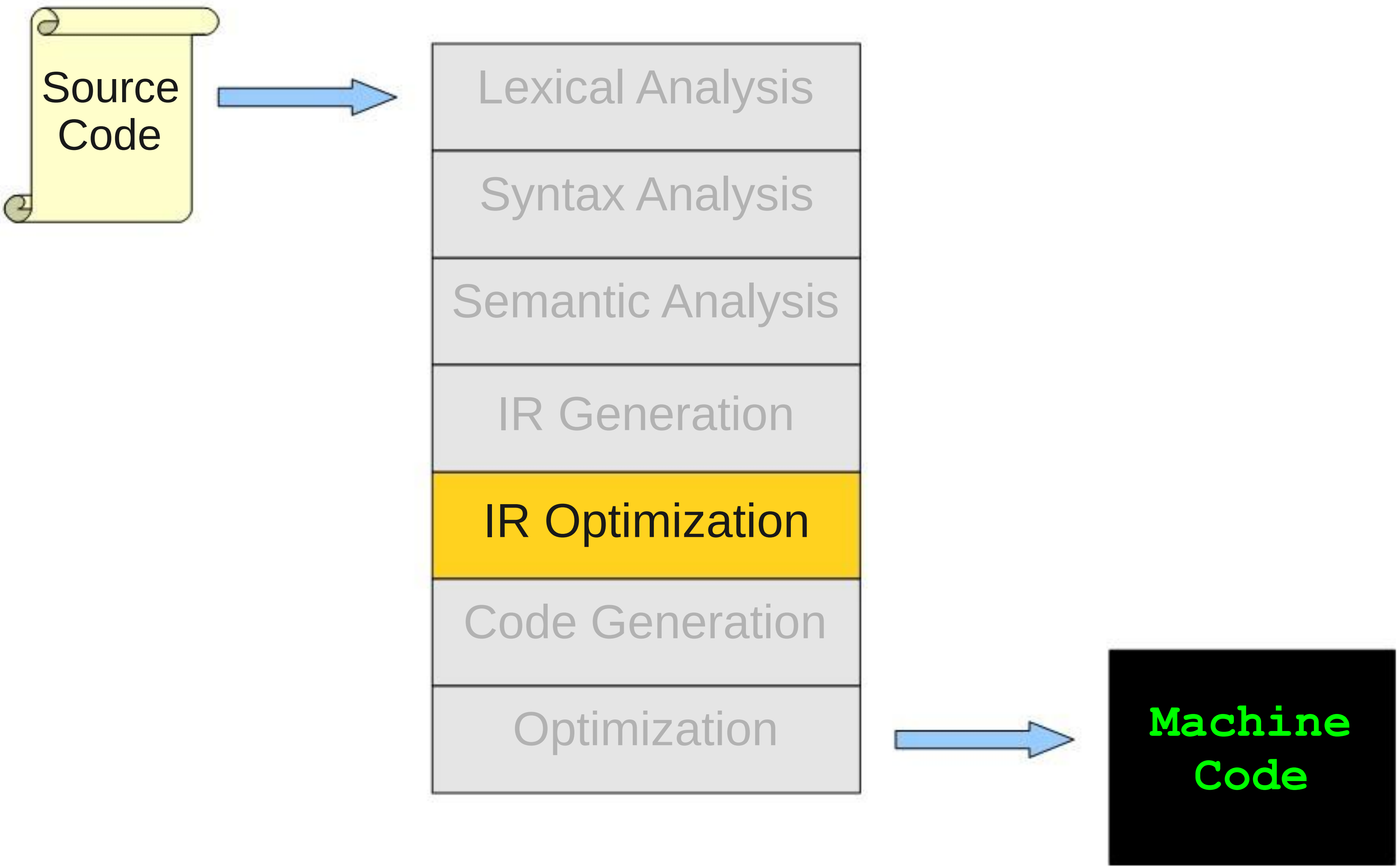
Where We Are


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

Copy Propagation

- If we have a variable assignment

  $\mathbf{v_1} = \mathbf{v_2}$

  then as long as $\mathbf{v_1}$ and $\mathbf{v_2}$ are not reassigned, we can rewrite expressions of the form

  $a = \ldots \mathbf{v_1} \ldots$

  as

  $a = \ldots \mathbf{v_2} \ldots$

  provided that such a rewrite is legal.
- This will help immensely later on, as you'll see.

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *(_tmp1) ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp6) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp0 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp0 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = 4 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = 4 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = 4 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Copy Propagation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

Dead codes

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = 4 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Dead Code Elimination

- An assignment to a variable **v** is called **dead** if the value of that assignment is never read anywhere.
- Dead code elimination removes dead assignments from IR.

Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

Dead codes

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

For Comparison

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
_tmp4 = _tmp0 + b ;
_tmp7 = *(_tmp2) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
```

Applying Local Optimizations

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

- To get maximum effect, we may have to apply these optimizations (Common subexpression elimination, Copy propagation, Dead code elimination) numerous times.

Applying Local Optimizations

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

Common Subexpression Elimination

Applying Local Optimizations

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

Common Subexpression Elimination

Applying Local Optimizations

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

Copy Propagation

Applying Local Optimizations

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

Copy Propagation

Applying Local Optimizations

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

Common Subexpression Elimination (Again)

Applying Local Optimizations

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

Common Subexpression Elimination (Again)

Applying Local Optimizations

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

Done

Other Types of Local Optimization

- **Arithmetic Simplification**
  - Replace “hard” operations with easier ones.
  - e.g. rewrite `x = a * 4;` as `x = a << 2;`

- **Constant Folding**
  - Evaluate expressions at compile-time if they have a constant value.
  - e.g. rewrite `x = 4*5;` as `x = 20;`.

Implementing Local Optimization

Available Expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the **available expressions** in a program.
- An expression is called **available** if some variable in the program holds the value of that expression.
- Whenever we execute a statement **a** = **b** + **c**:
  - Any previous expression holding **a** is invalidated.
  - The expression **a** = **b** + **c** becomes available.

Available Expressions

`{ }` Initially, no available expression.

```
  a = b;

  c = b;

d = a + b;

e = a + b;

  d = b;

f = a + b;
```

Available Expressions

```
{}
a = b;
{ a = b }
c = b;

d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;

e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;

d = b;

f = a + b;
```

Available Expressions

```
{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;

f = a + b;
```

Available Expressions

```
{}
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, ~~d = a + b~~, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
```

Available Expressions

```
{}
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{}
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{}
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{}
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{}
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{}
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{}
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{}
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Common Subexpression Elimination

```
{}
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = e;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

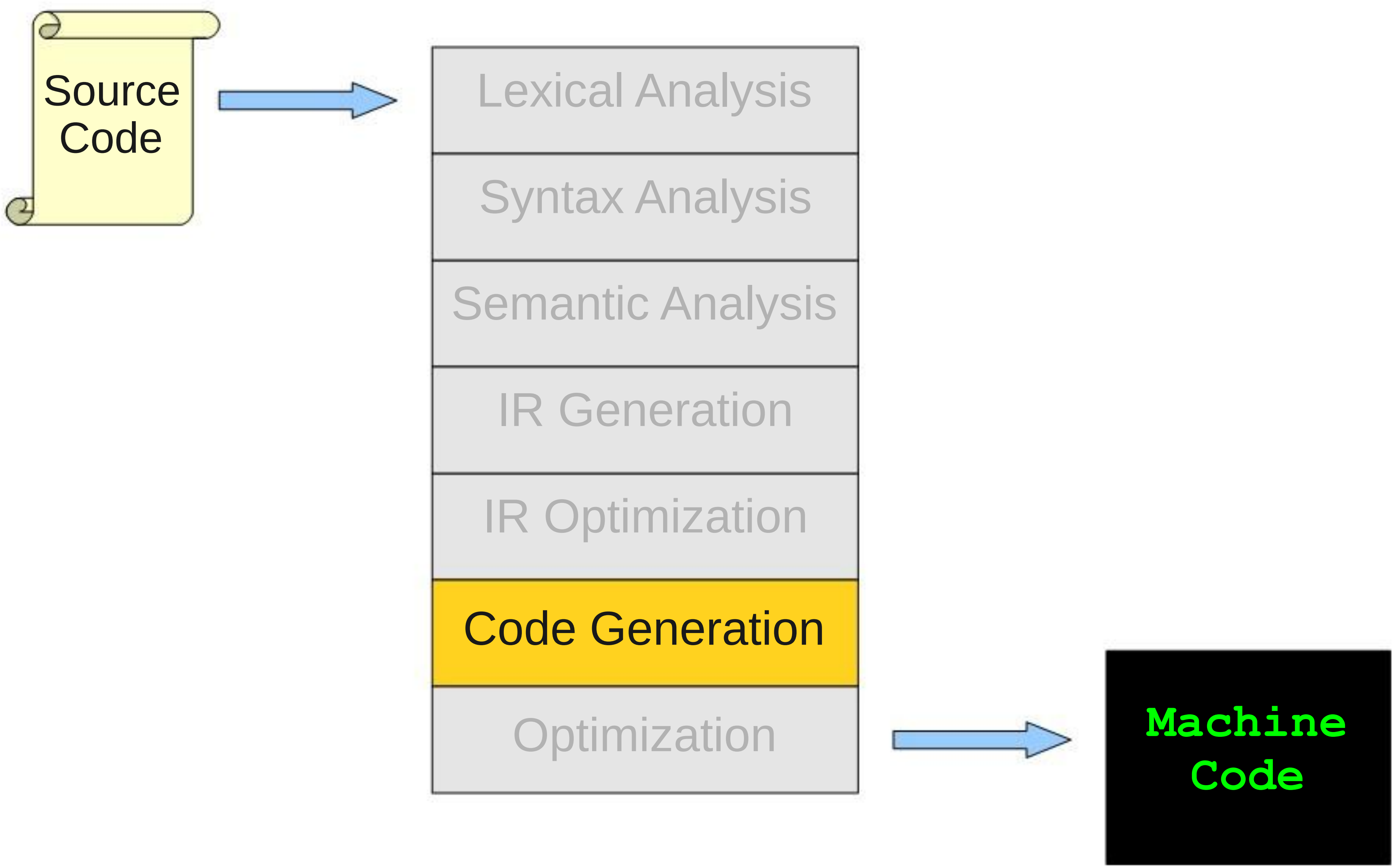
Common Subexpression Elimination

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

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

Register Allocation

Where We Are



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

The Memory Hierarchy

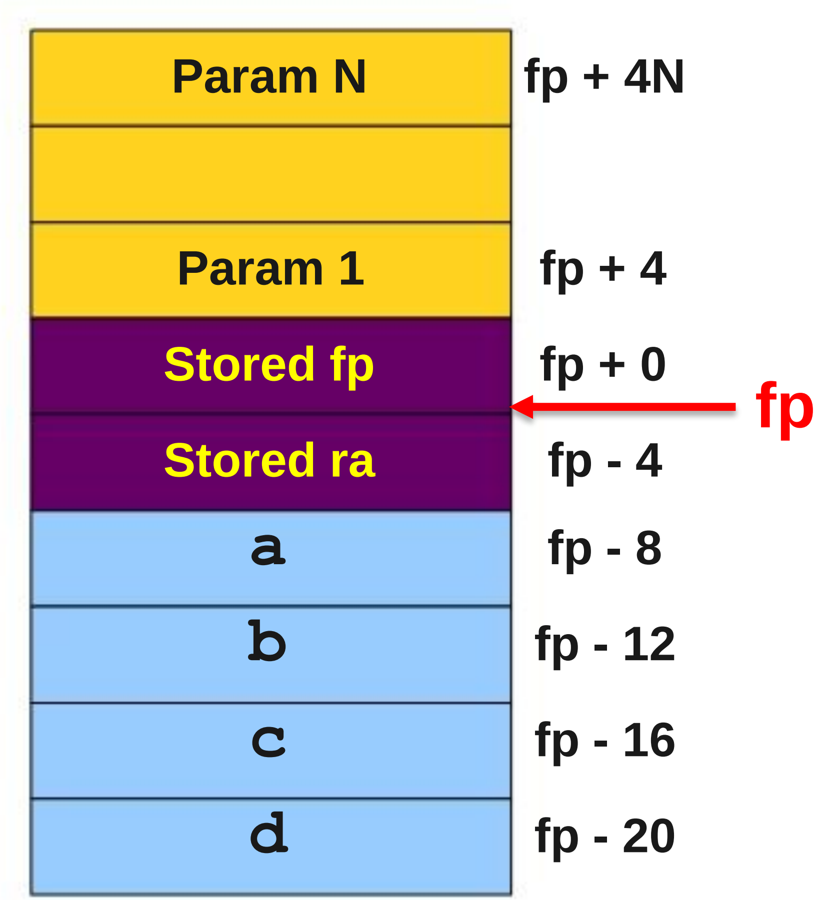
	Size	Time
Registers	256B - 8KB	0.25 - 1ns
L1 Cache	16KB - 64KB	1ns - 5 ns
L2 Cache	1MB - 4MB	5ns - 25ns
Main Memory	4GB - 32GB	25ns - 100ns
Hard Disk	100GB+	3 - 10ms
Network	**HUGE**	10 - 2000ms

Register Allocation

- **Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers.
  - Need to find a way to reuse registers whenever possible.
- 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.

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.
  - Slower.
  - 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.
  - Good if you just needed to get a prototype compiler up and running.

Building a Better Allocator

- **Goal**:
  - Reduces memory reads/writes.

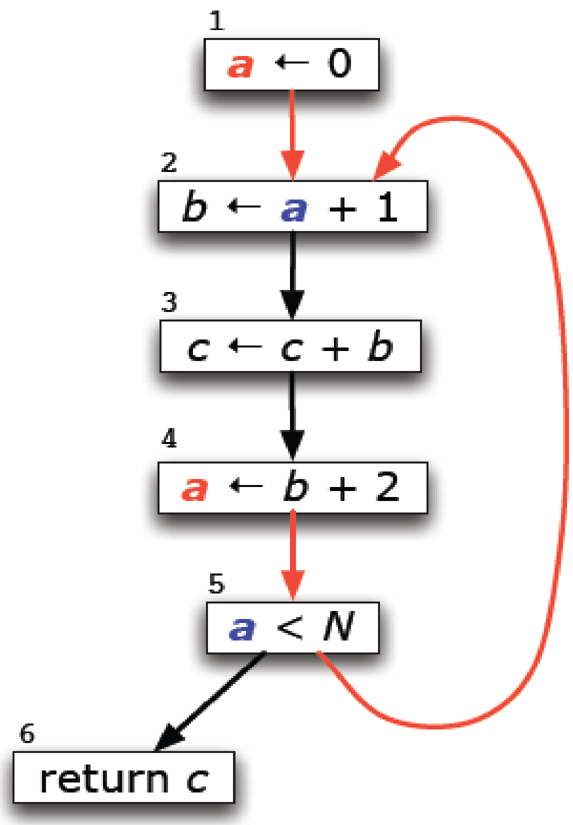
- We will need to address these questions:
  - Which registers do we put variables in?
  - What do we do when we run out of registers?

Live Ranges and Live Intervals

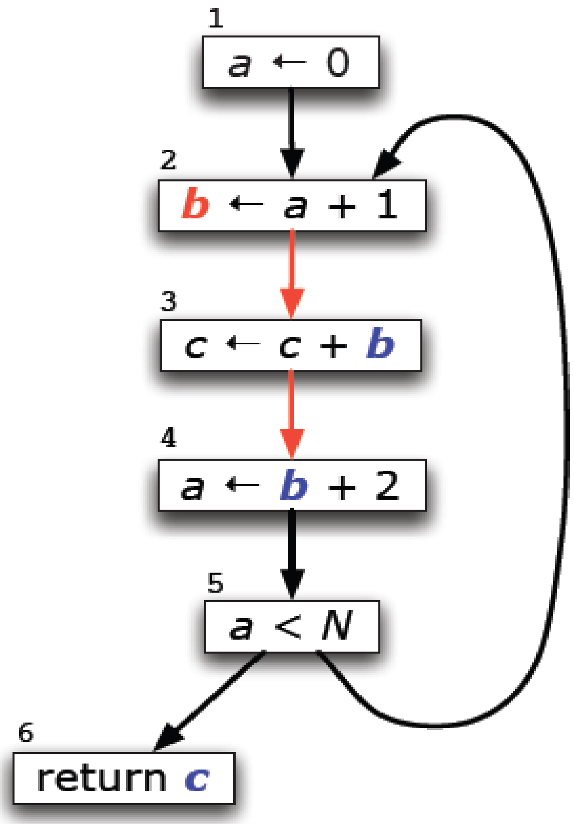
- The **live range** for a variable is the set of program points at which that variable is live.

- The **live interval** for a variable is the smallest subrange of the IR code containing all a variable's live ranges.

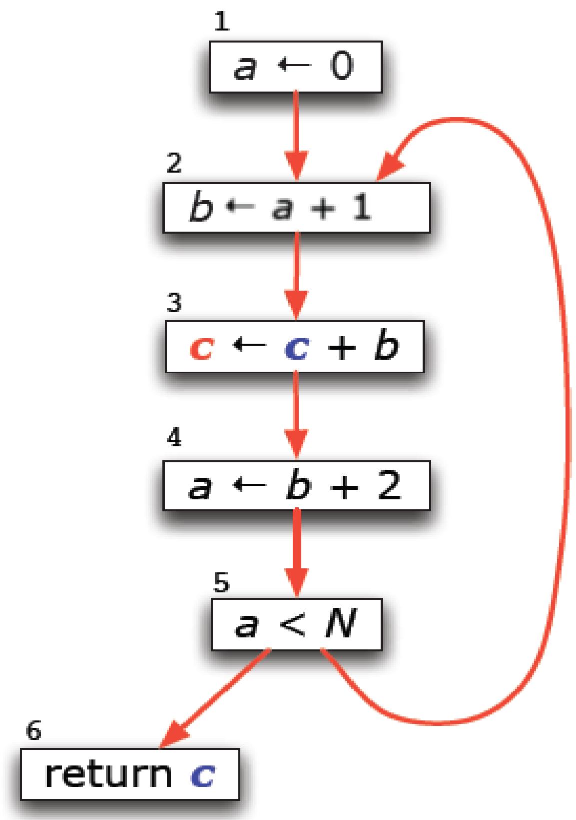
Live Ranges



Live ranges of a



Live ranges of b



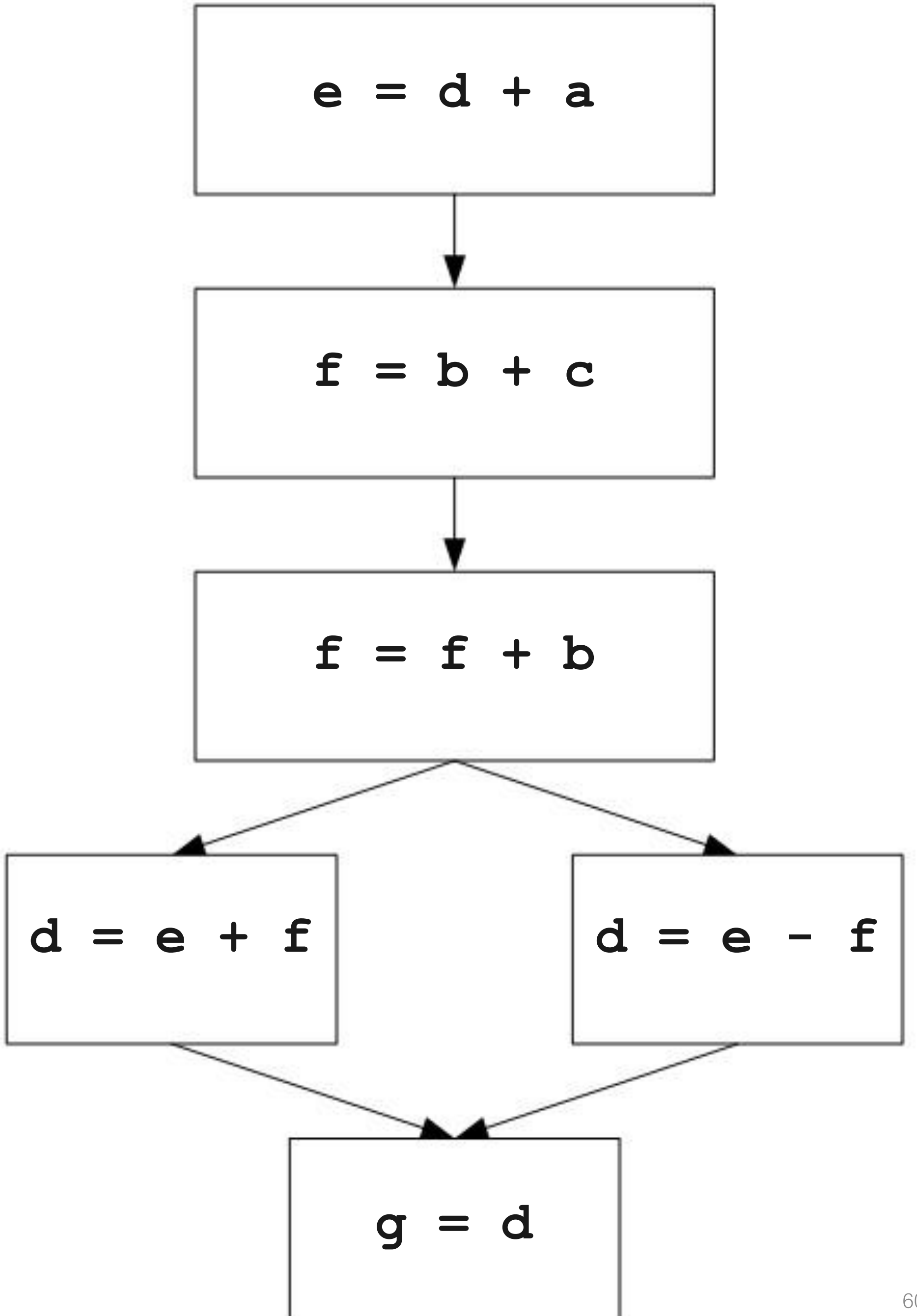
Live ranges of c

	Var a	Var b	Var c
Live Ranges	Lines 1-2 Lines 4-5-2	Lines 2-3-4	Lines 1-2-3 , 3-4-5-2-3 Lines 1-2-3, 3-4-5-6

Live Ranges and Live Intervals

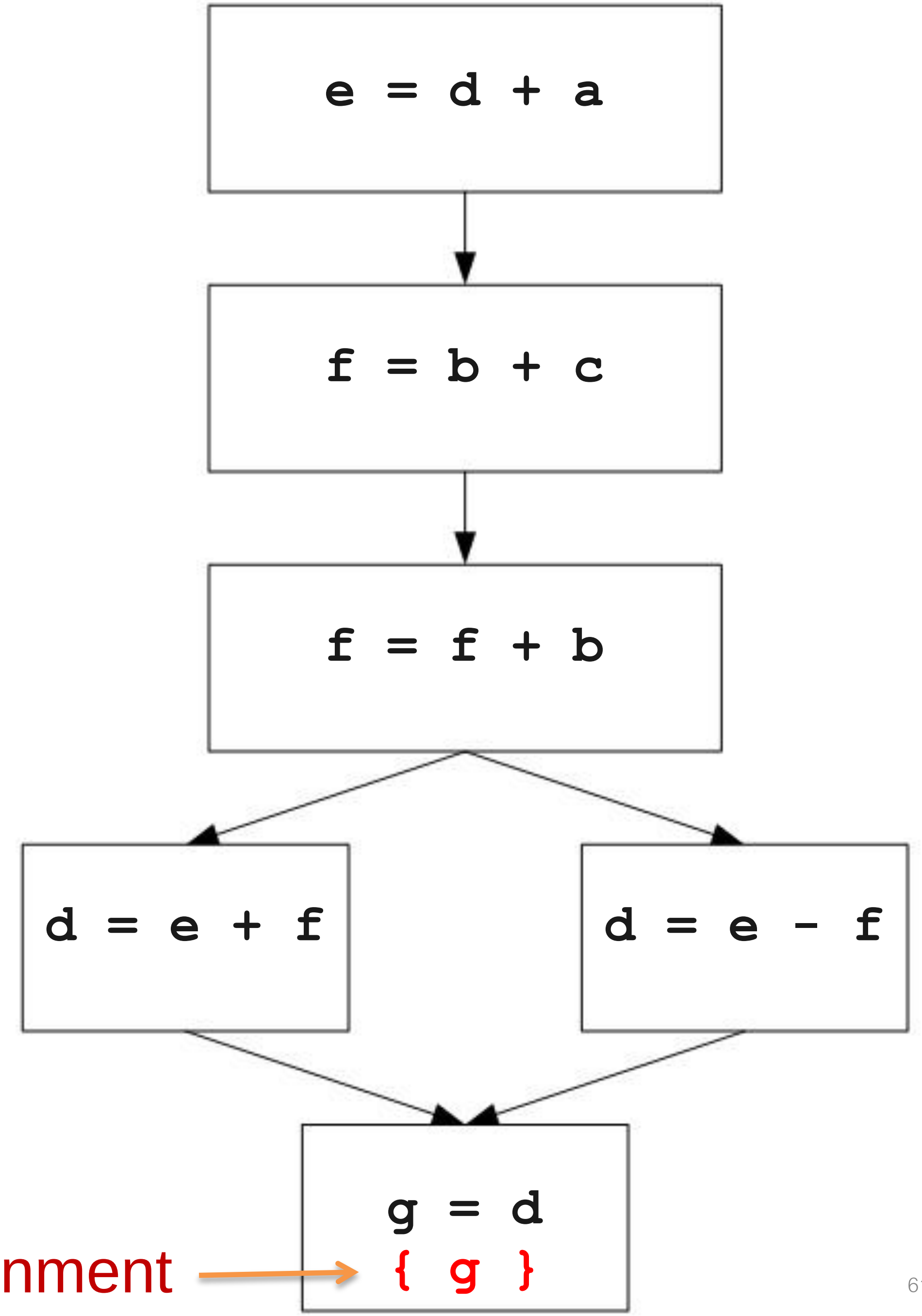
Example

```
    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

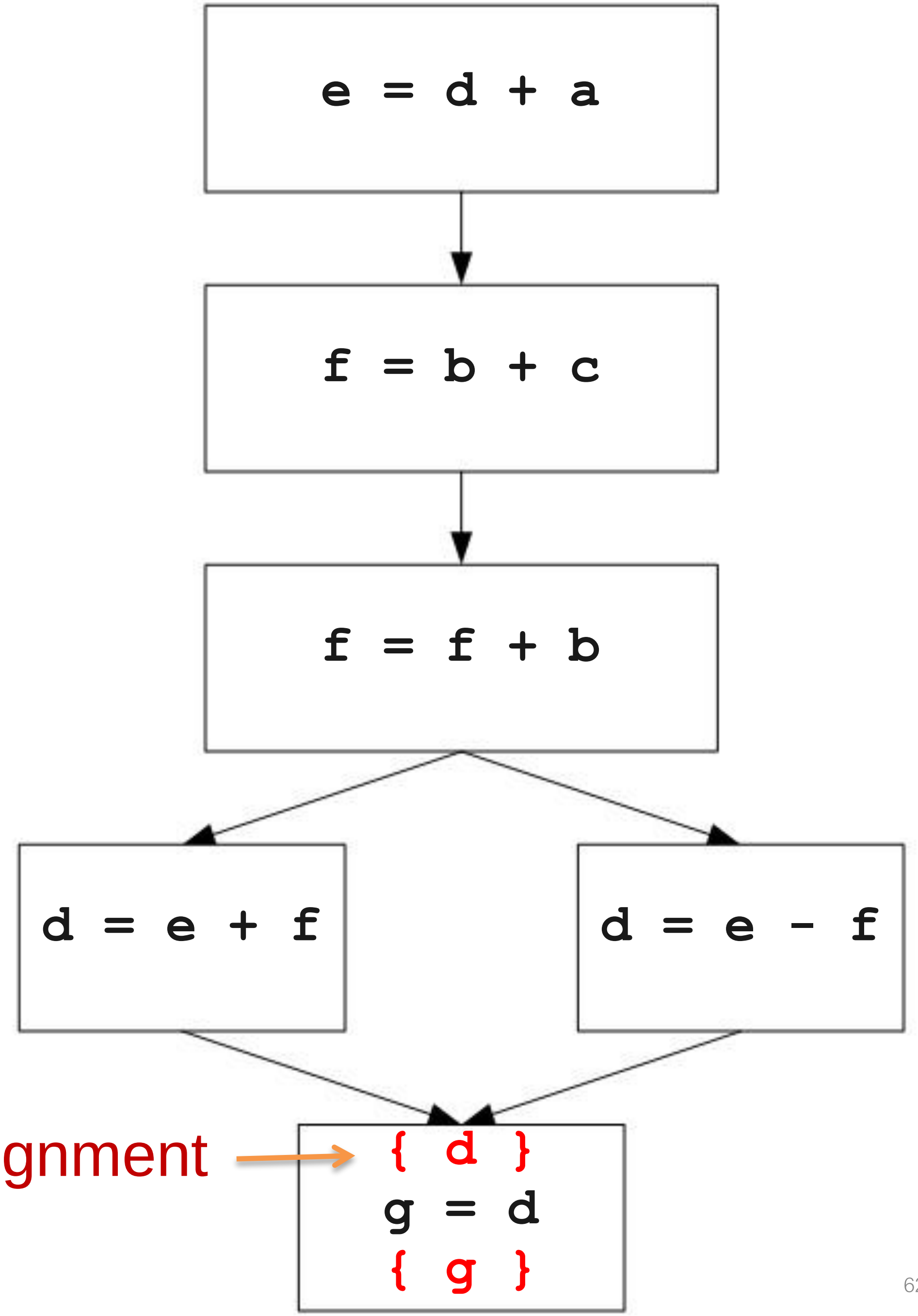
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



g is live after assignment

Live Ranges and Live Intervals

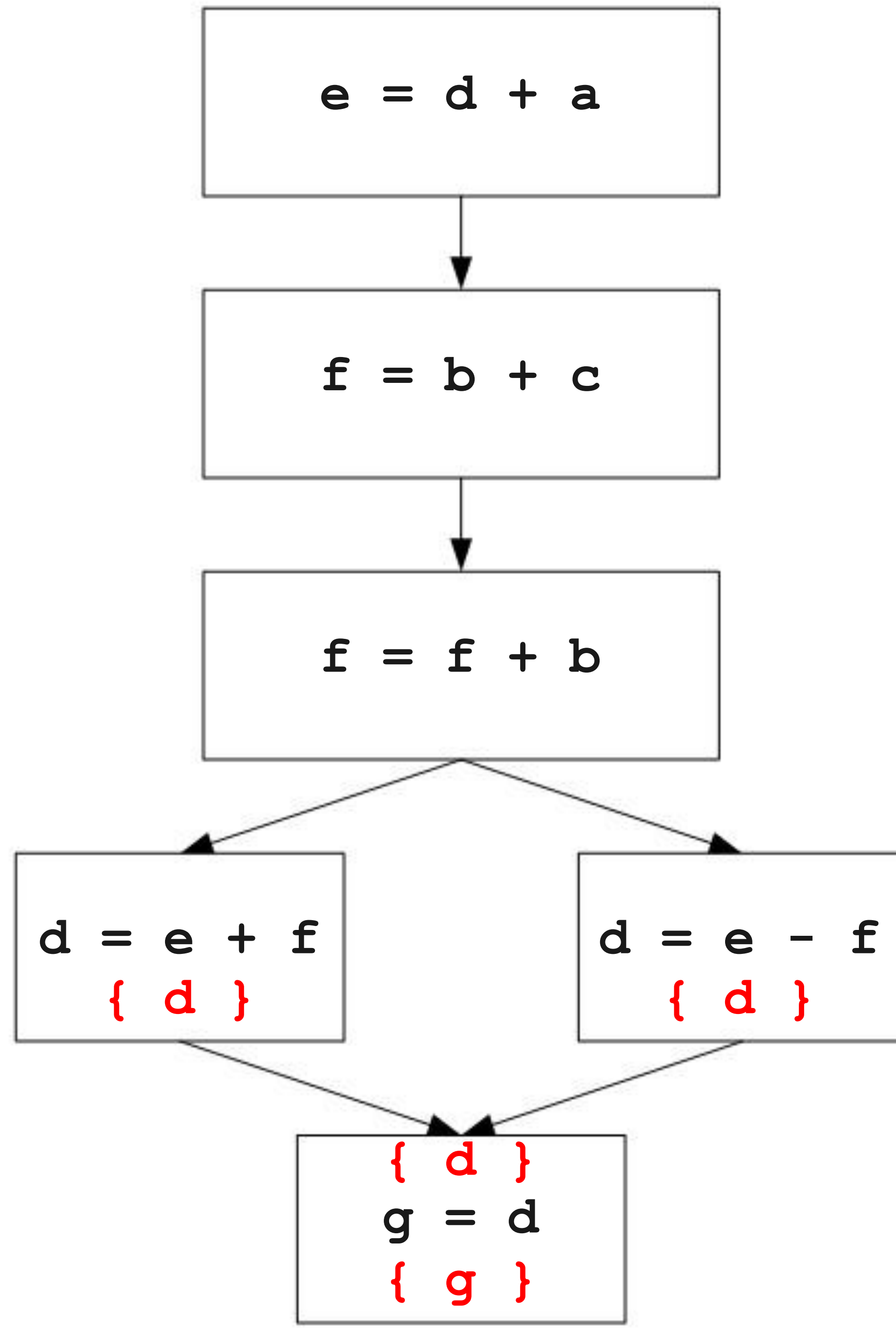
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



d must be live before assignment

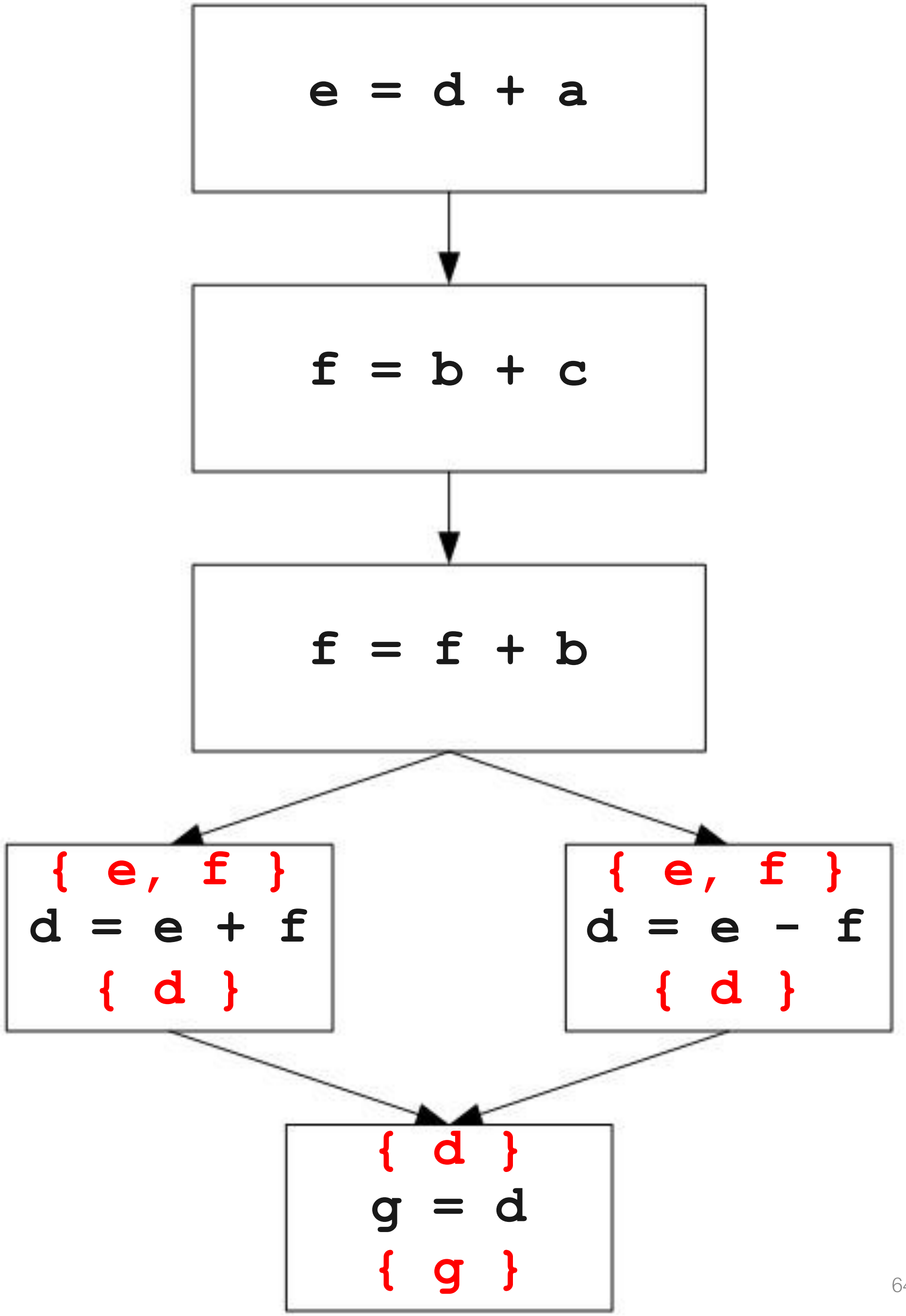
Live Ranges and Live Intervals

```
    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

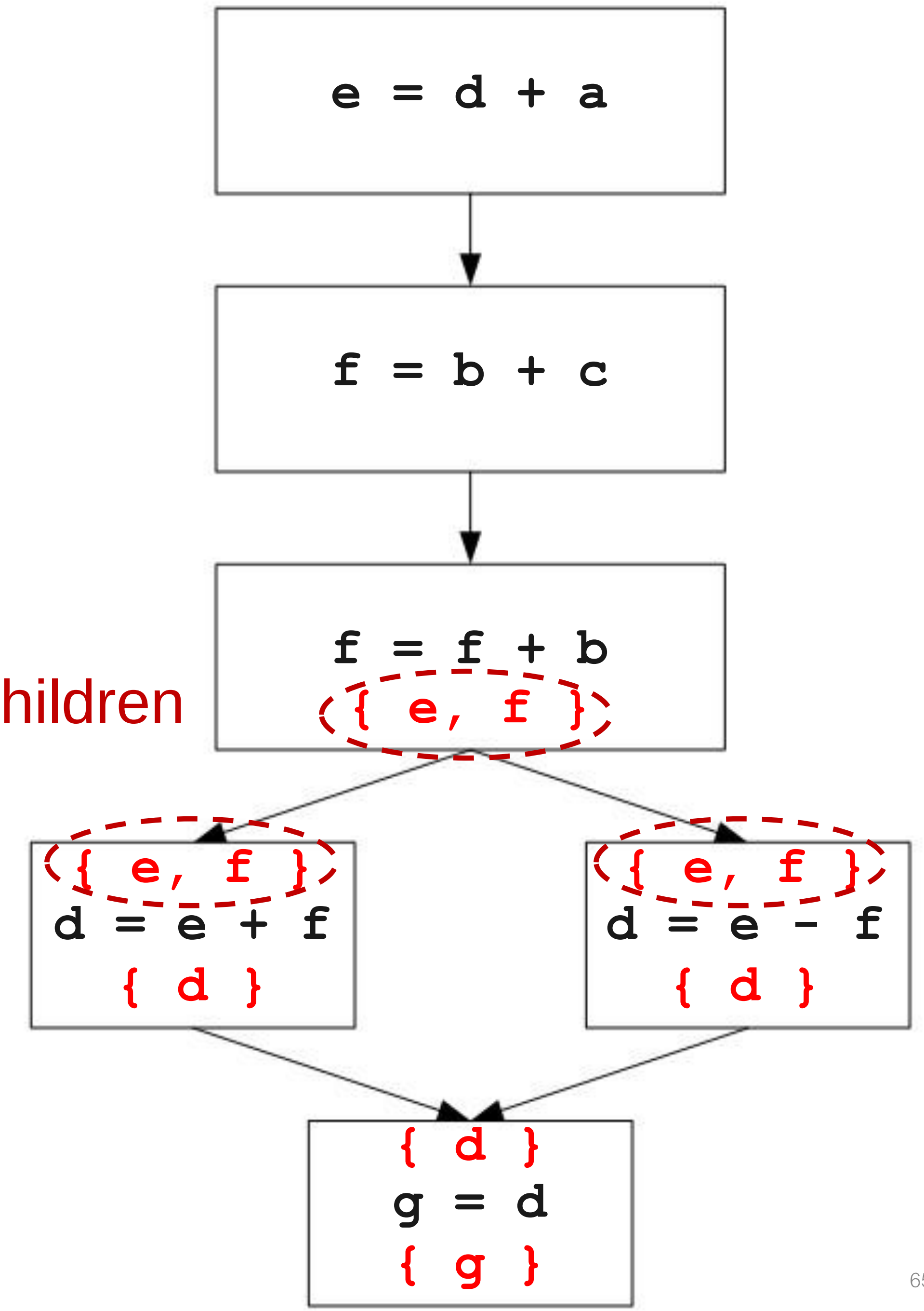
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



Live Ranges and Live Intervals

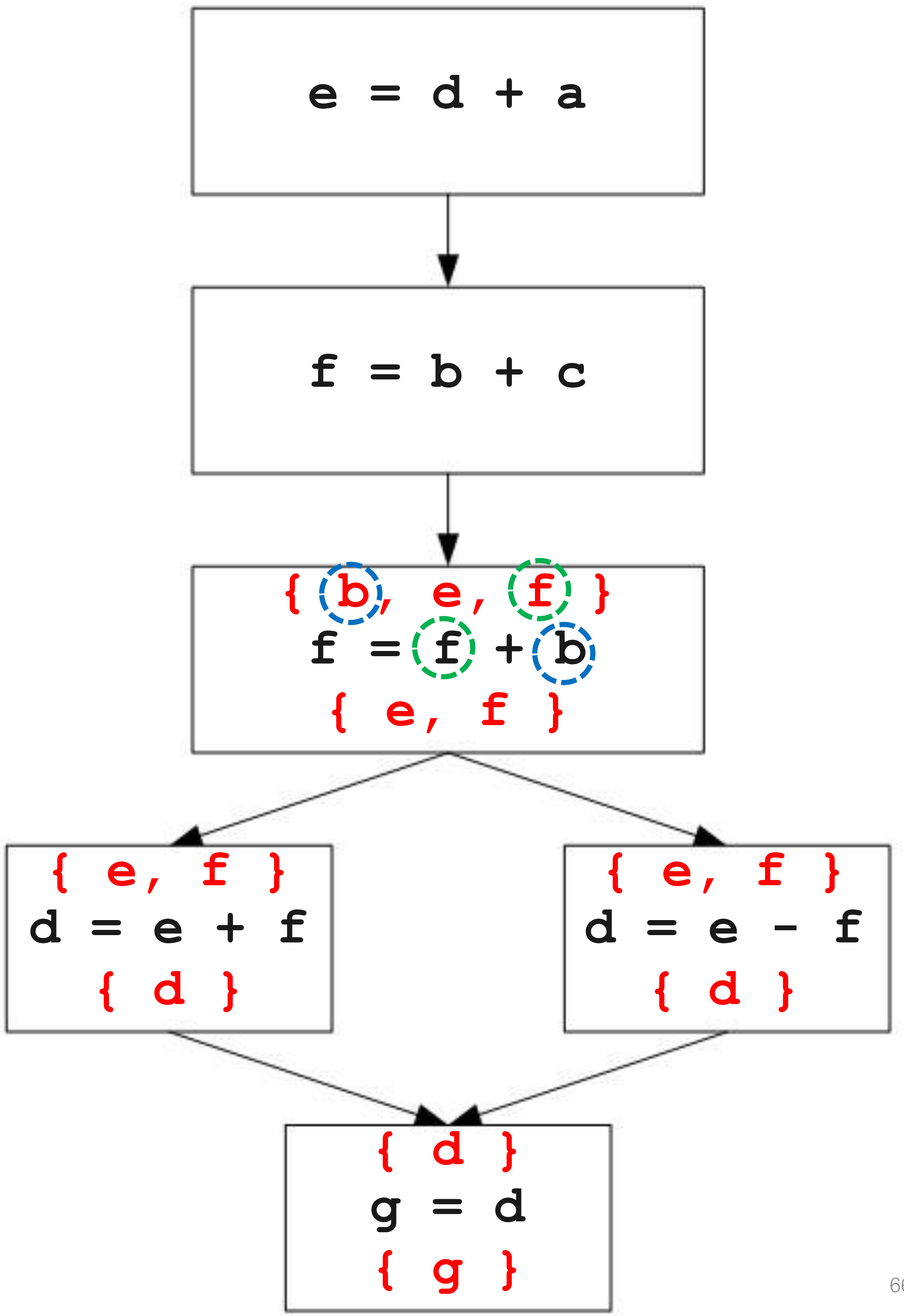
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

Union of both children



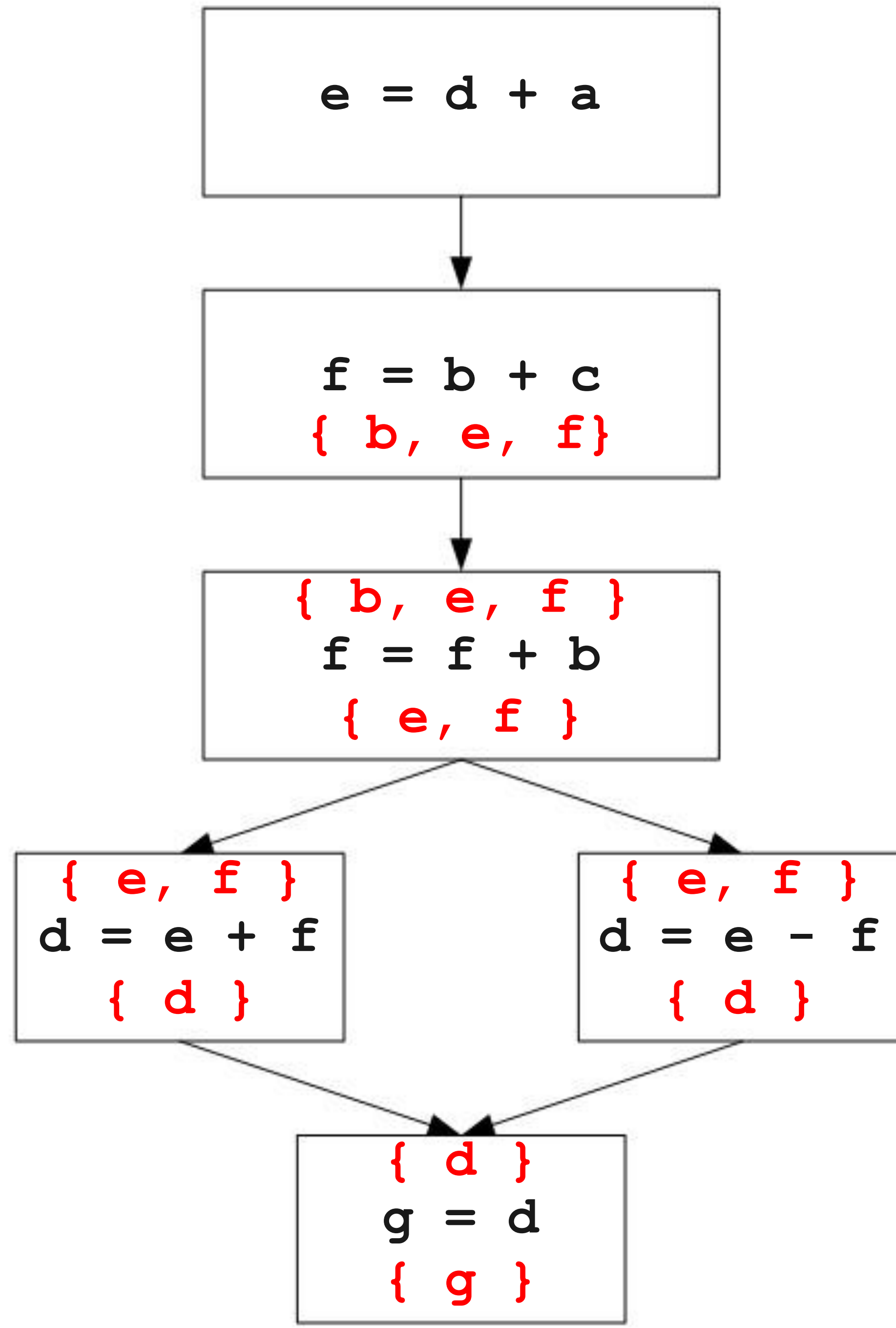
Live Ranges and Live Intervals

```
    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
```



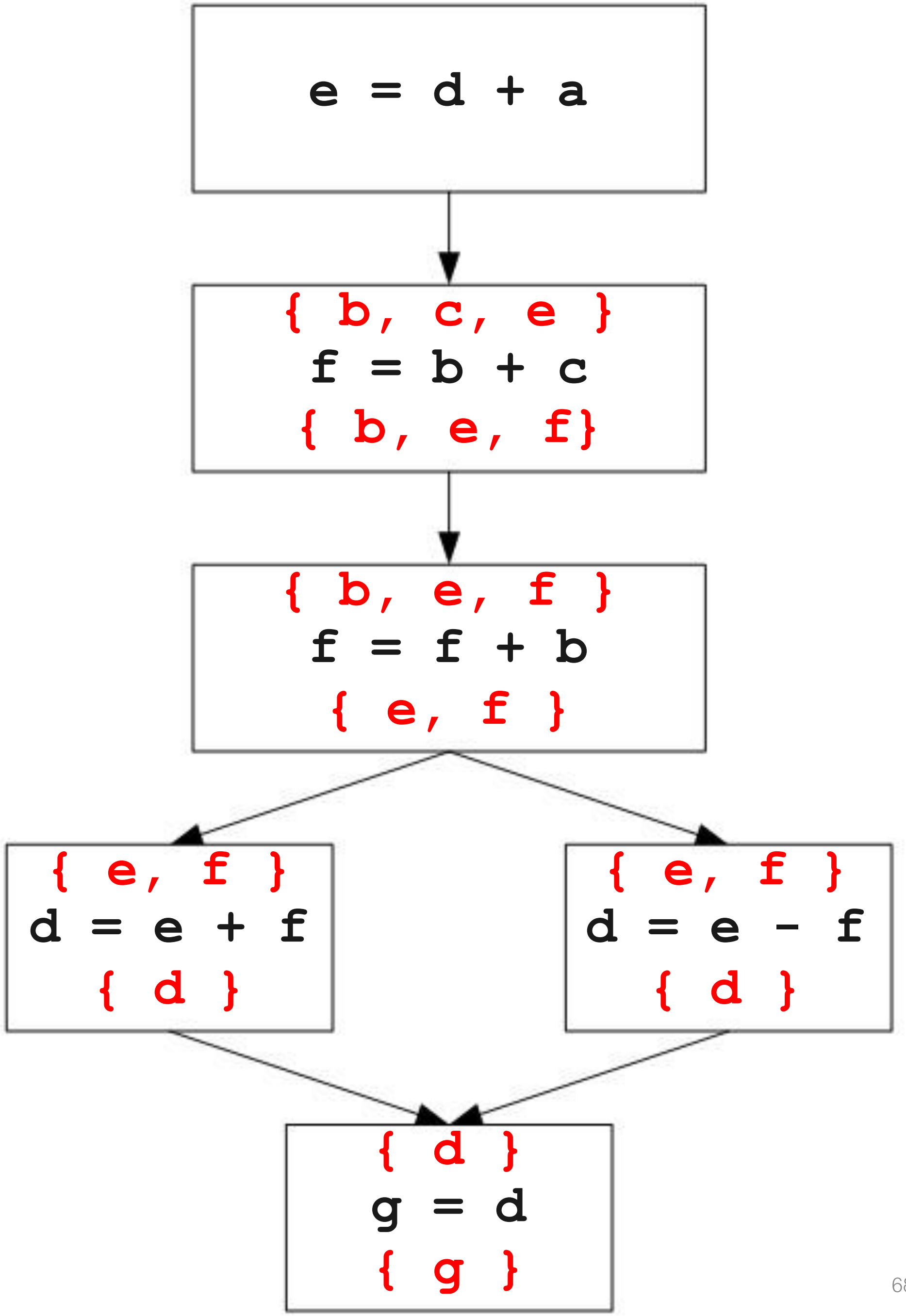
Live Ranges and Live Intervals

```
    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
```



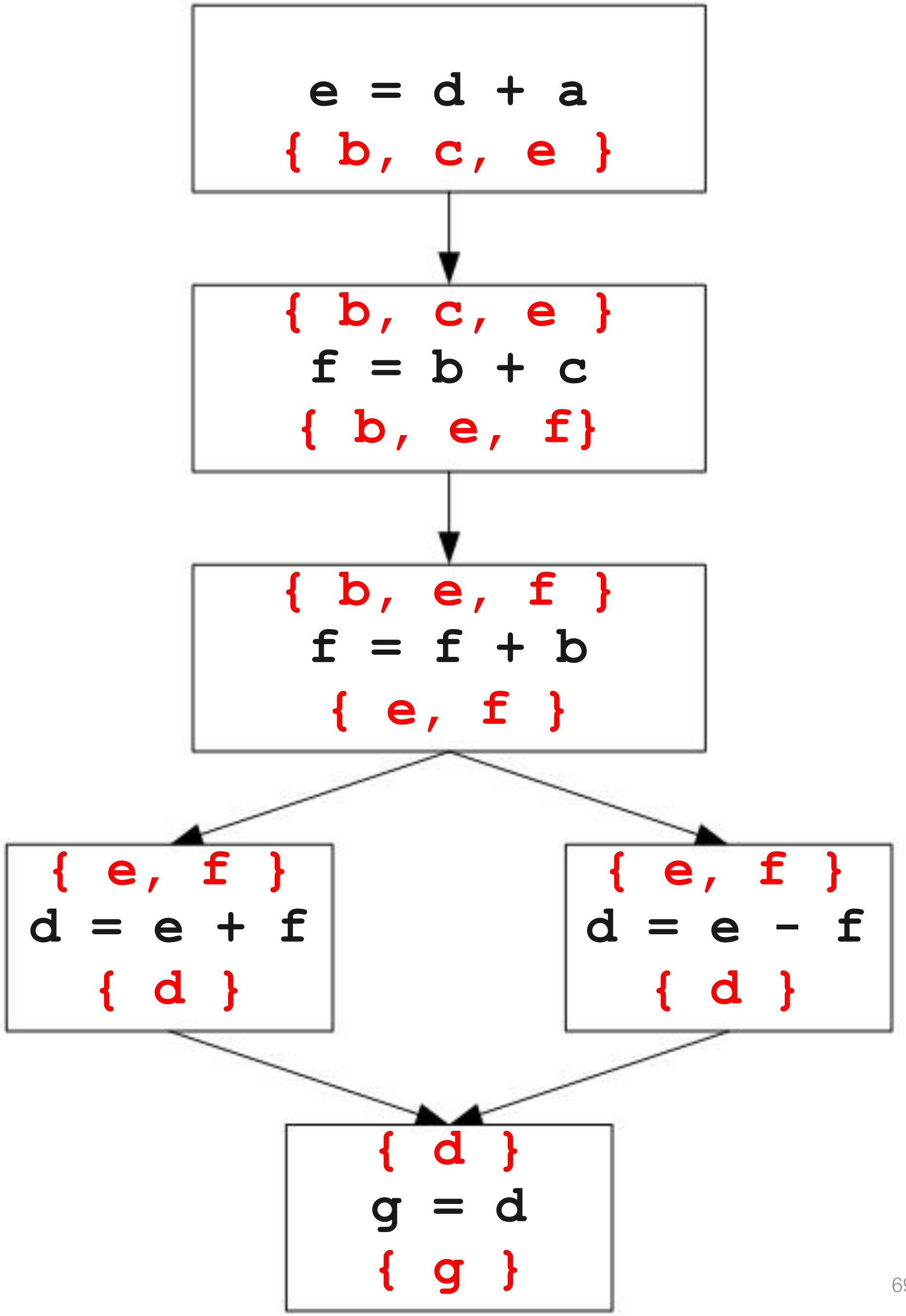
Live Ranges and Live Intervals

```
    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

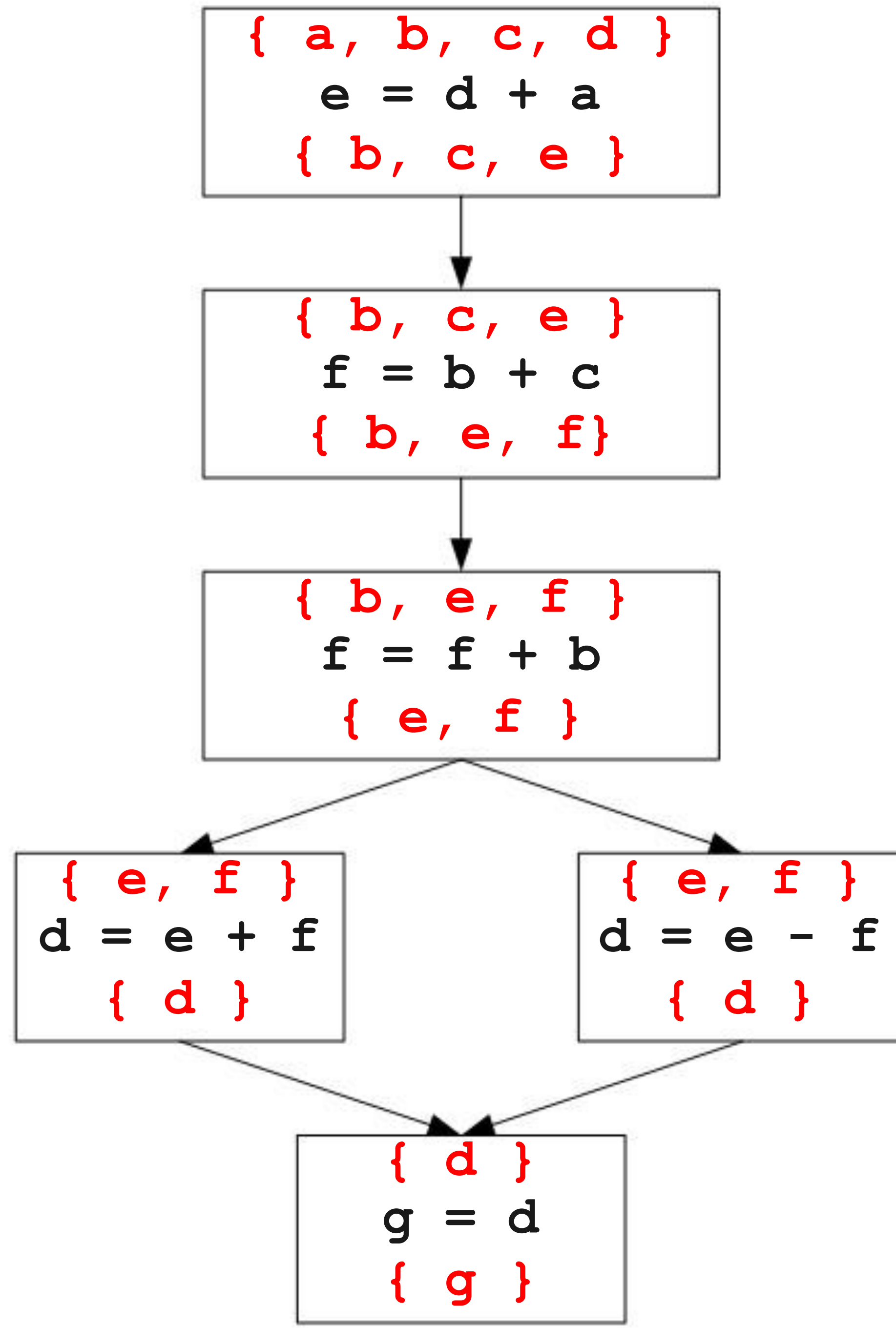
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



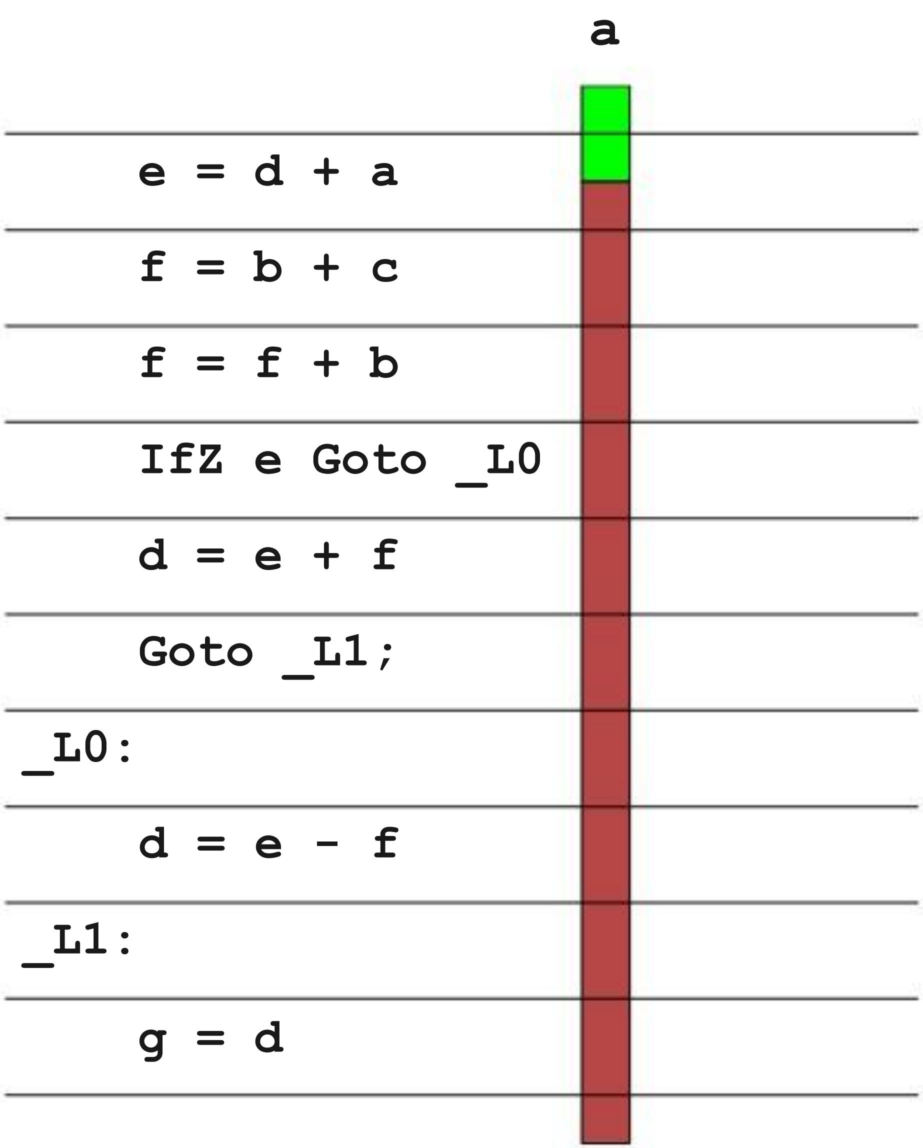
Live Ranges and Live Intervals

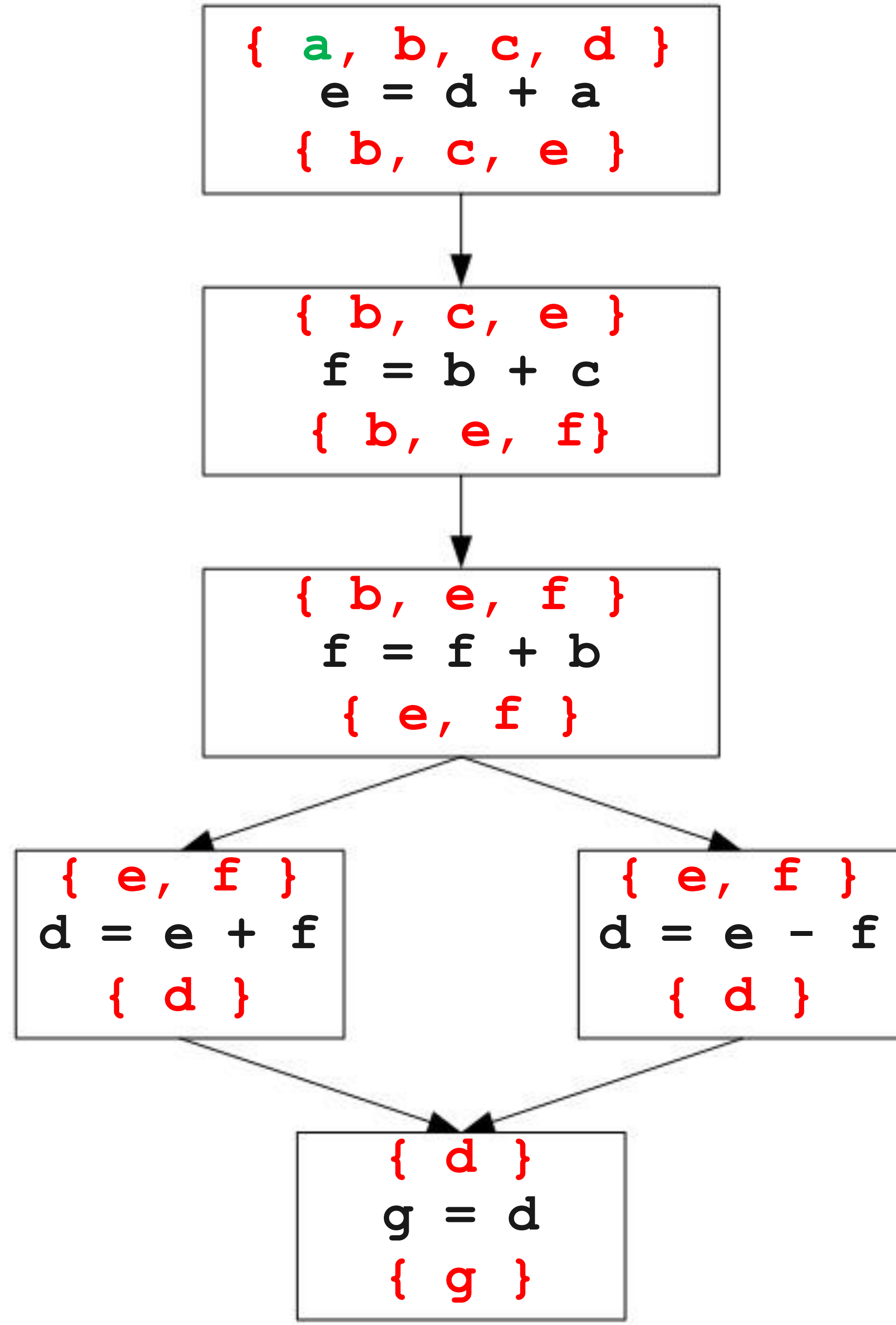
Finding Live Intervals

```
    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
```

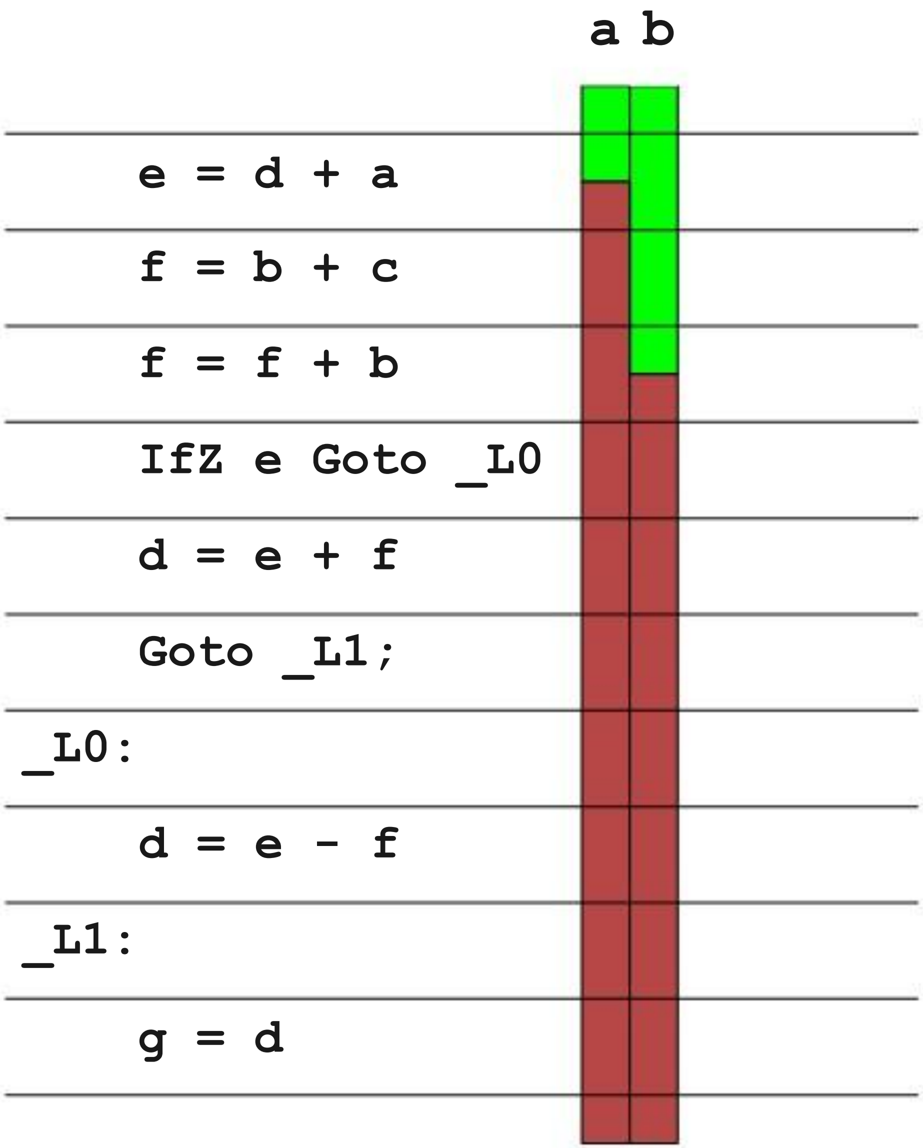


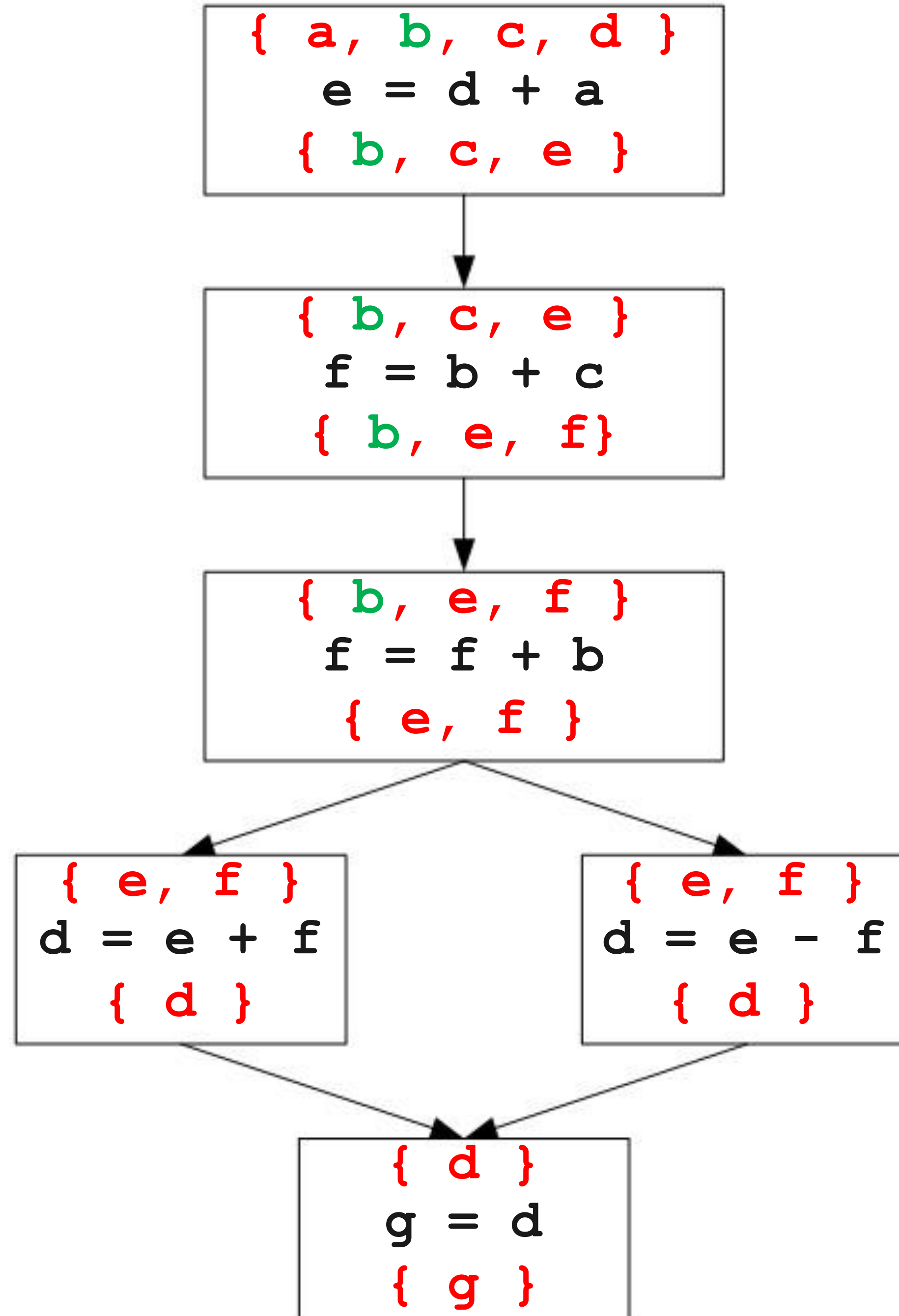
Live Ranges and Live Intervals



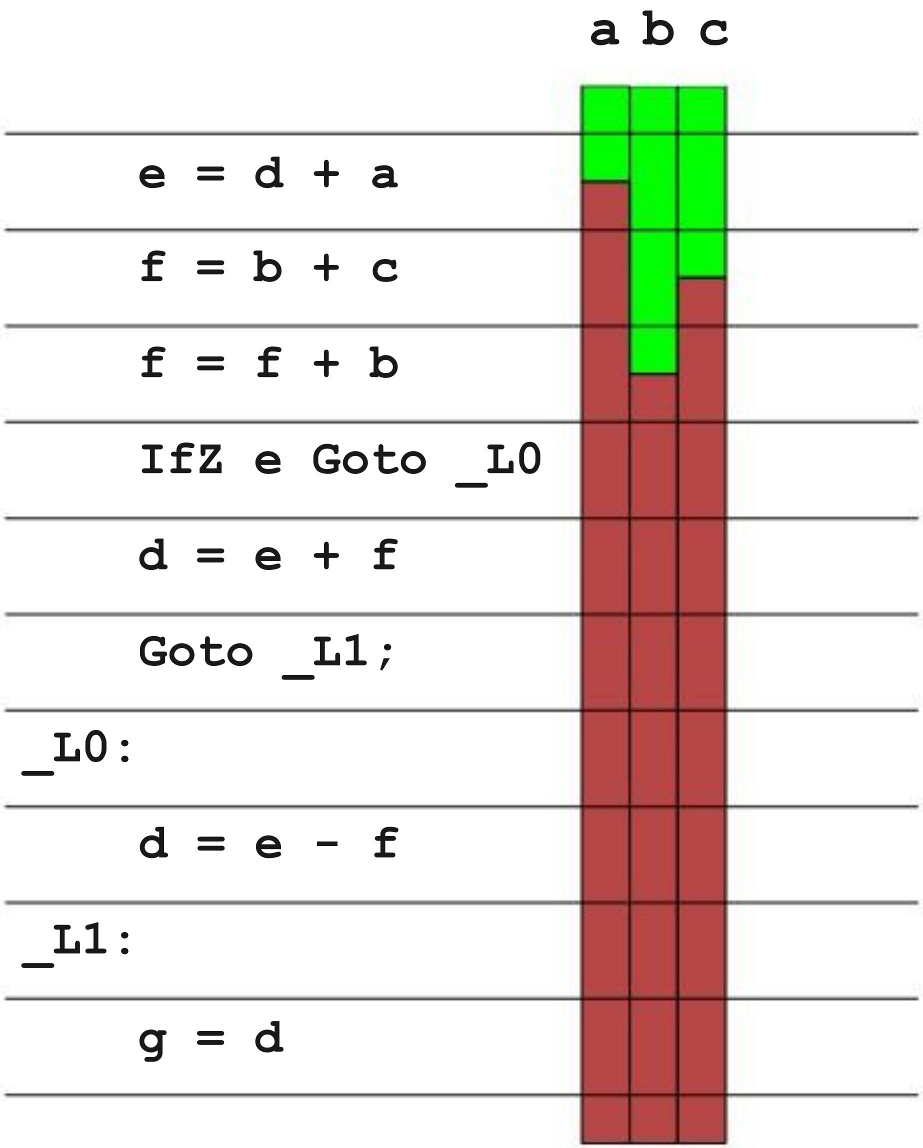


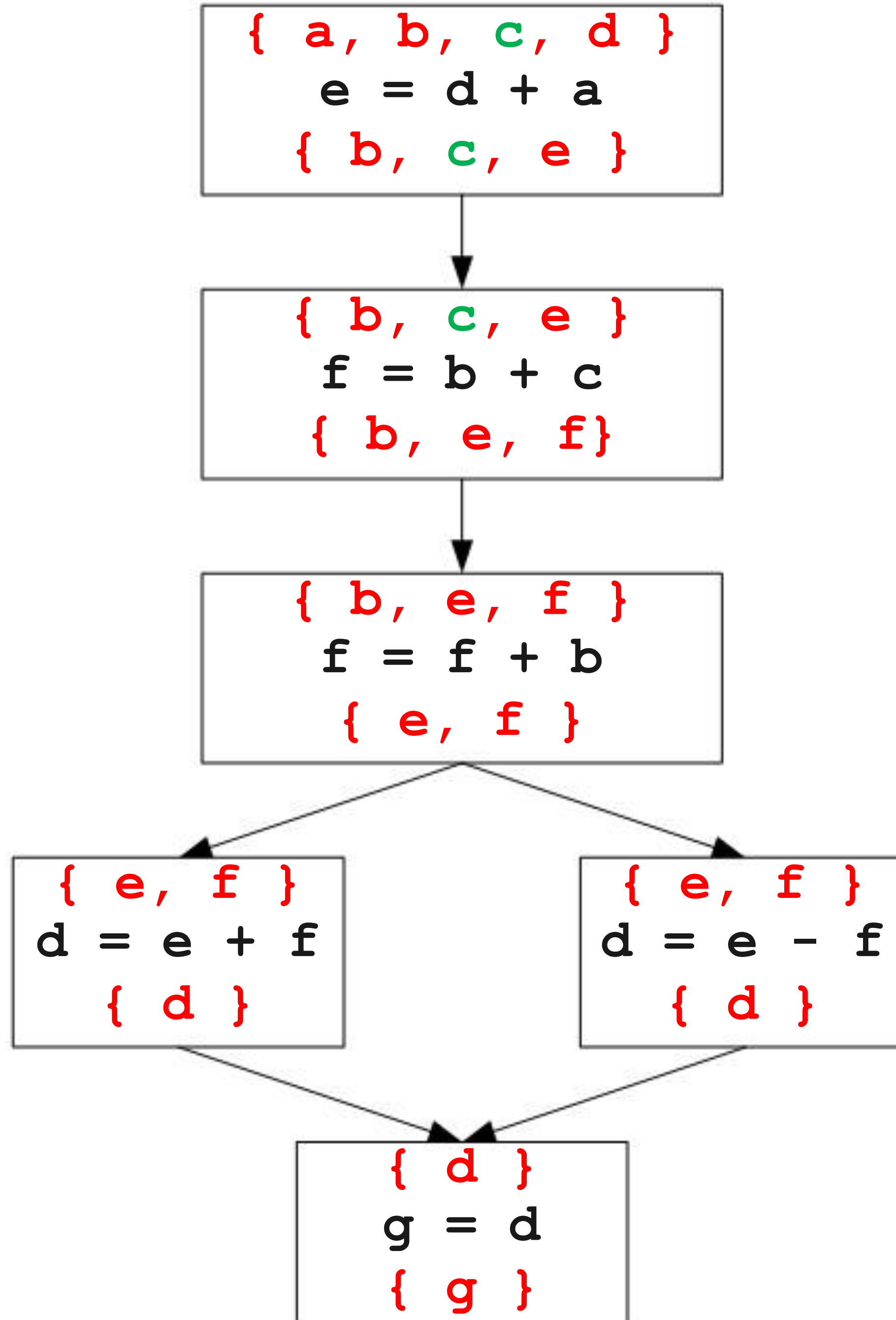
Live Ranges and Live Intervals



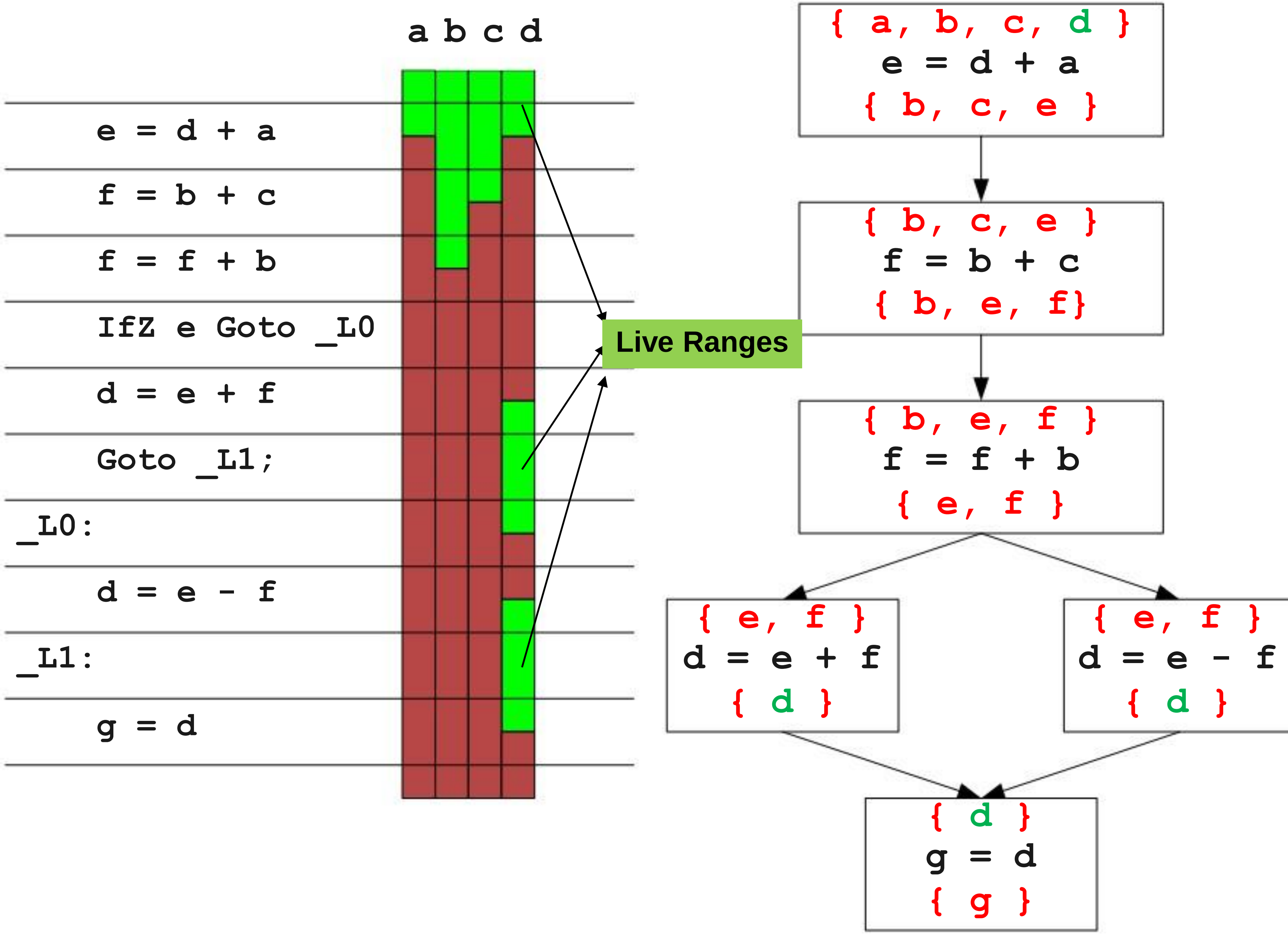


Live Ranges and Live Intervals

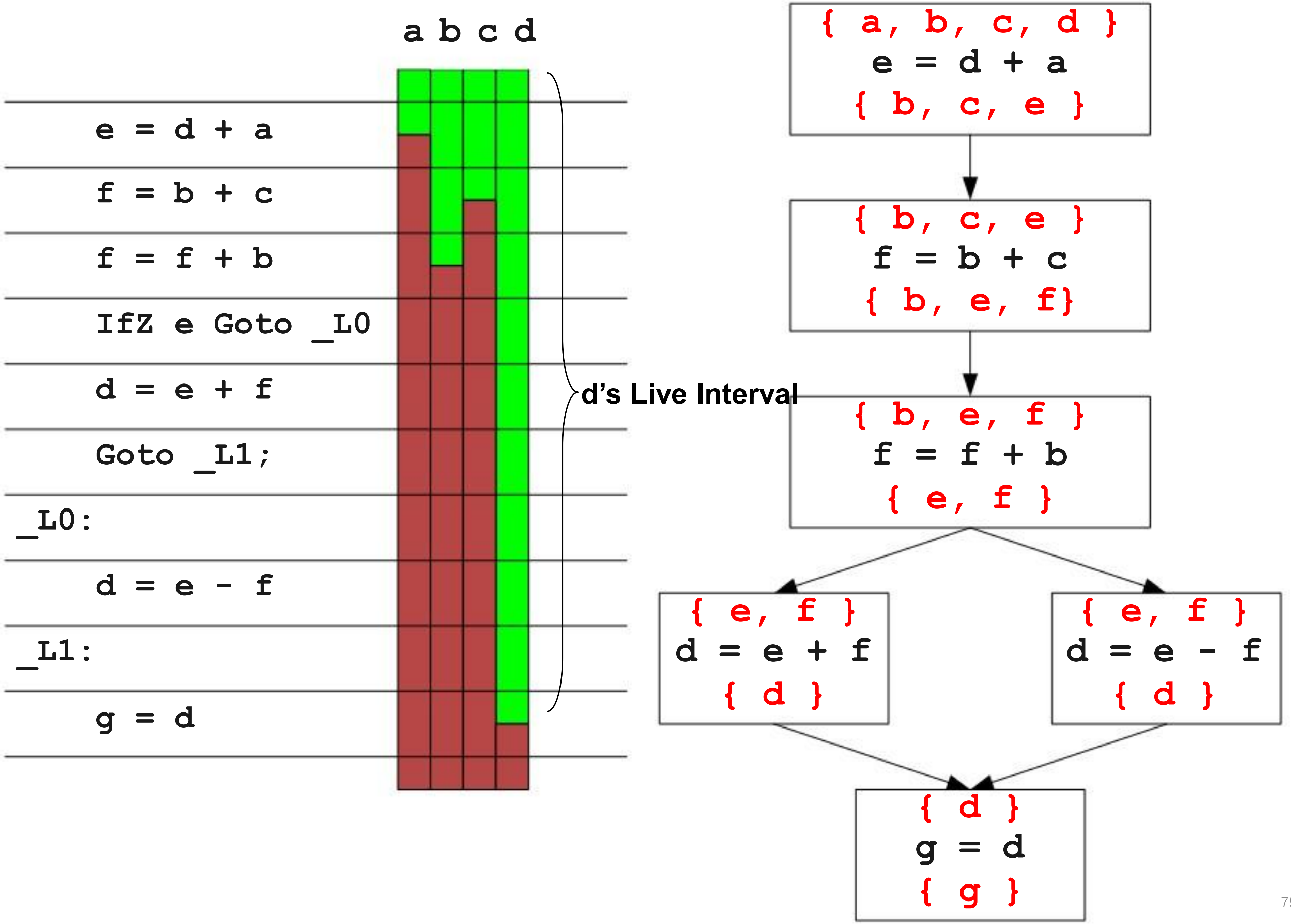




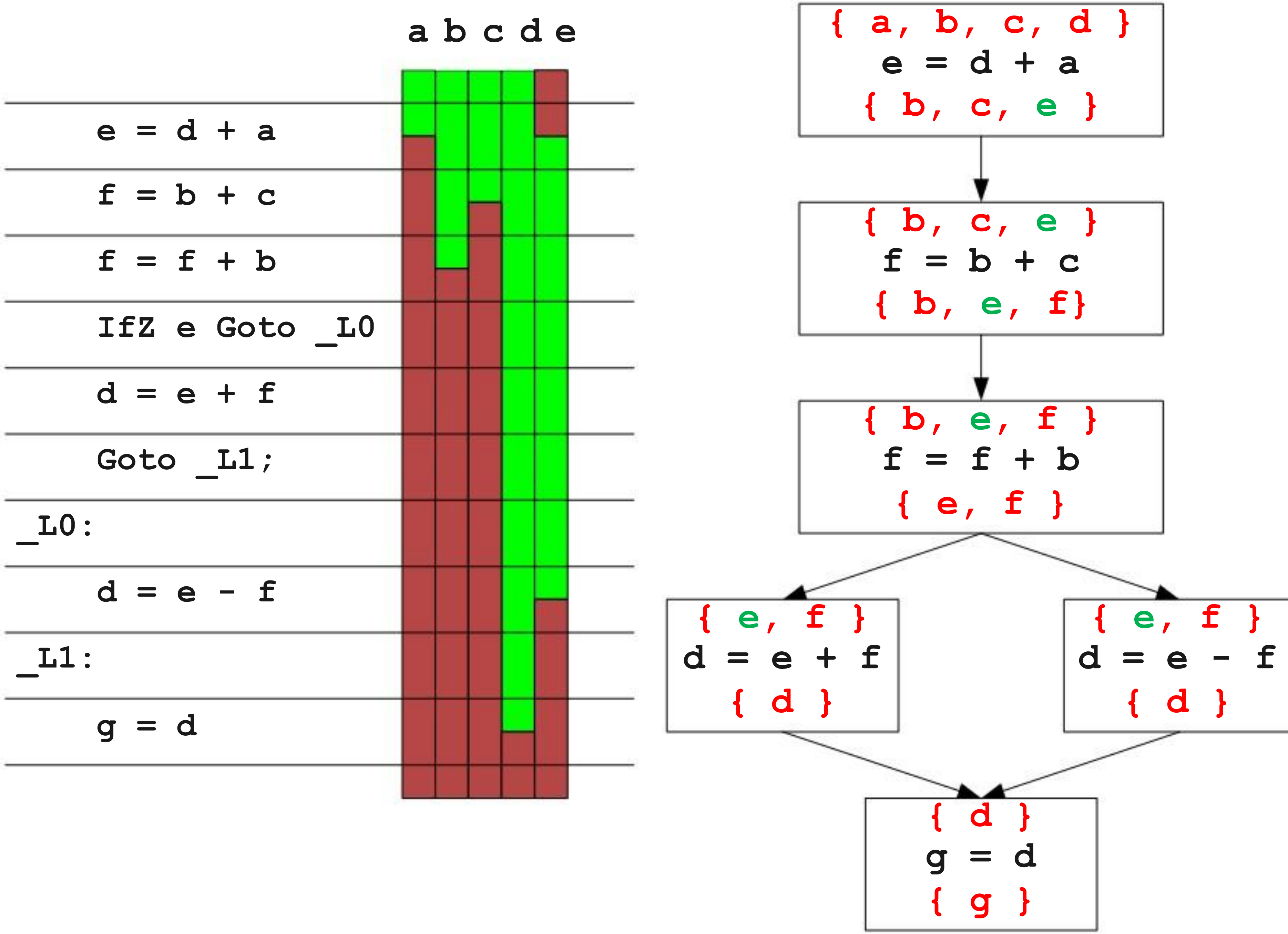
Live Ranges and Live Intervals



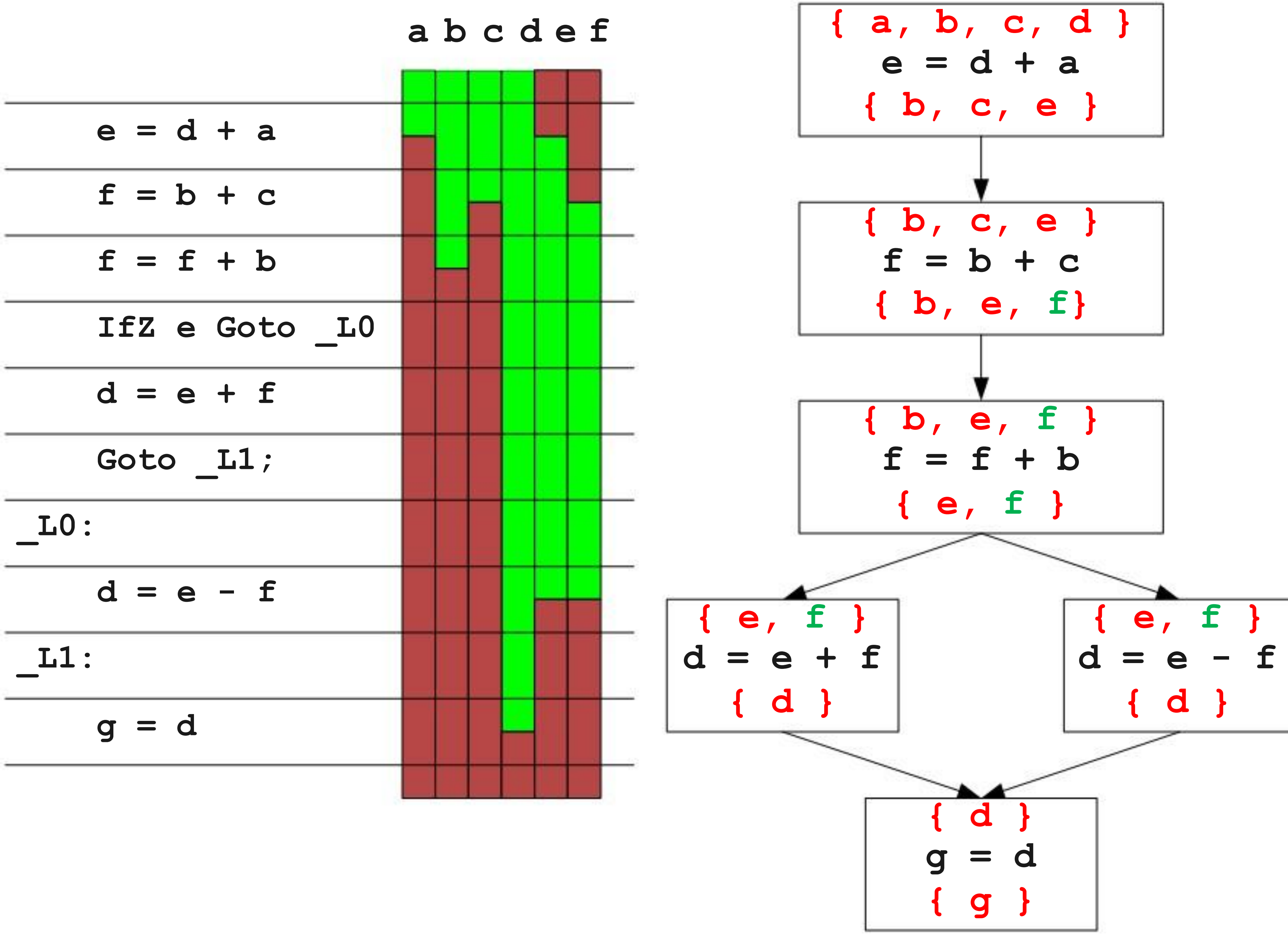
Live Ranges and Live Intervals



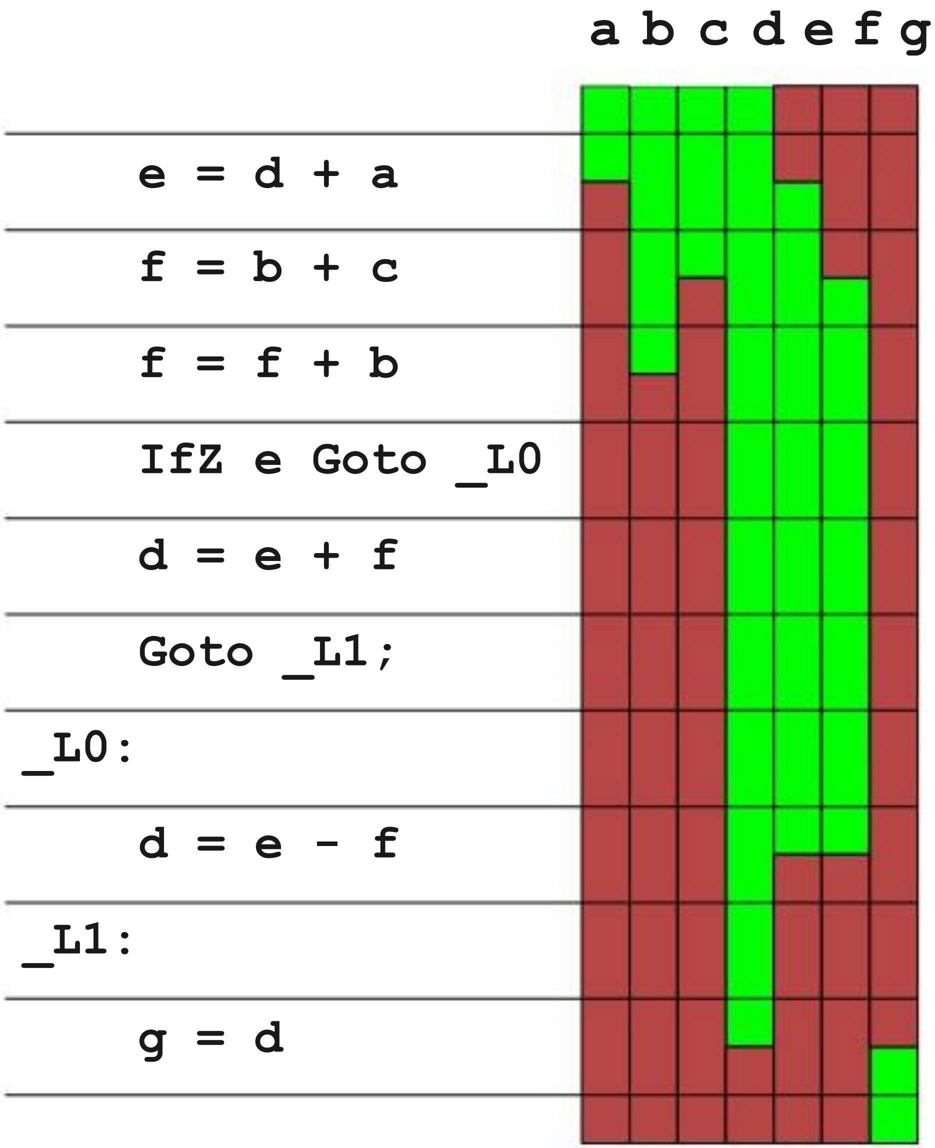
Live Ranges and Live Intervals

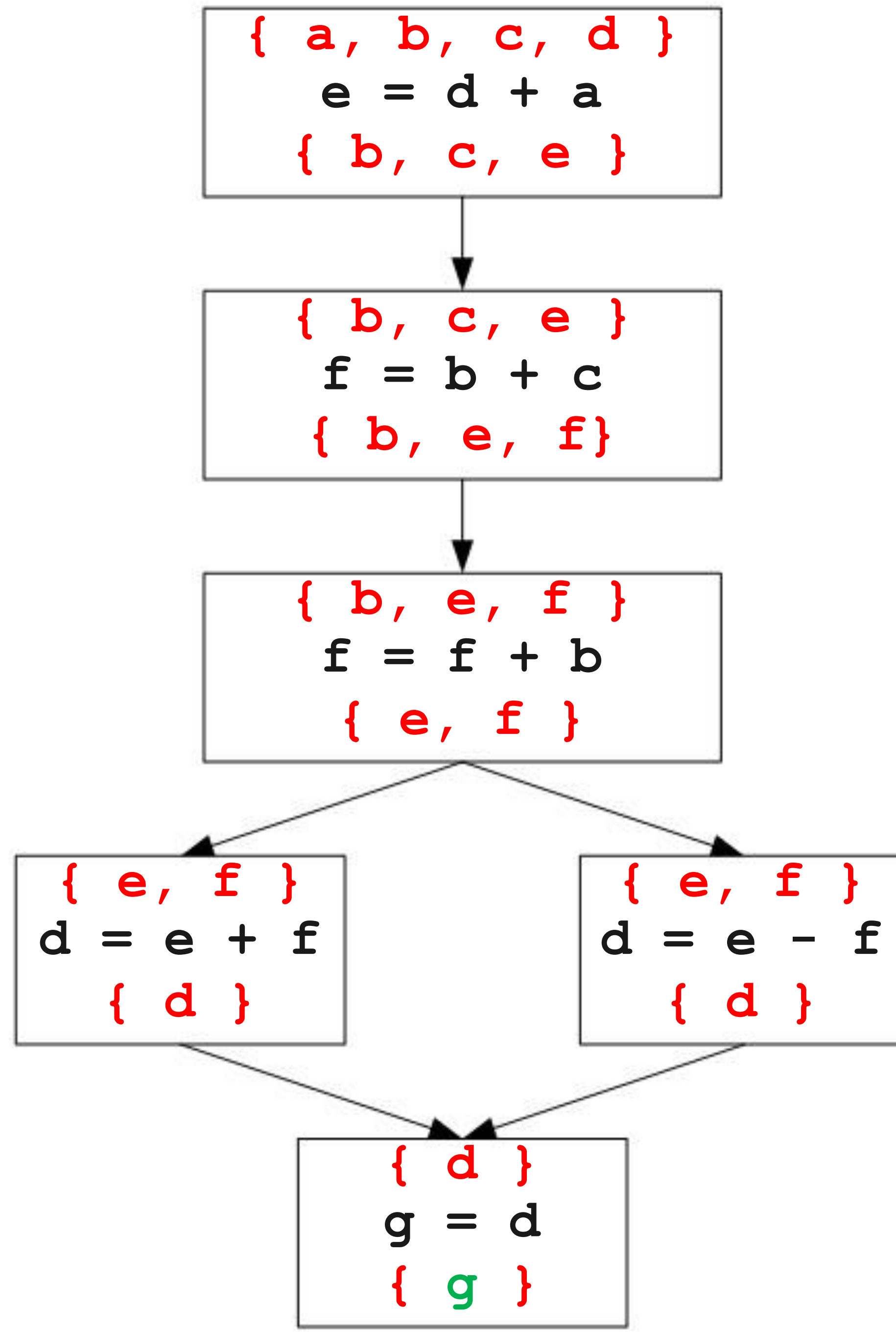


Live Ranges and Live Intervals



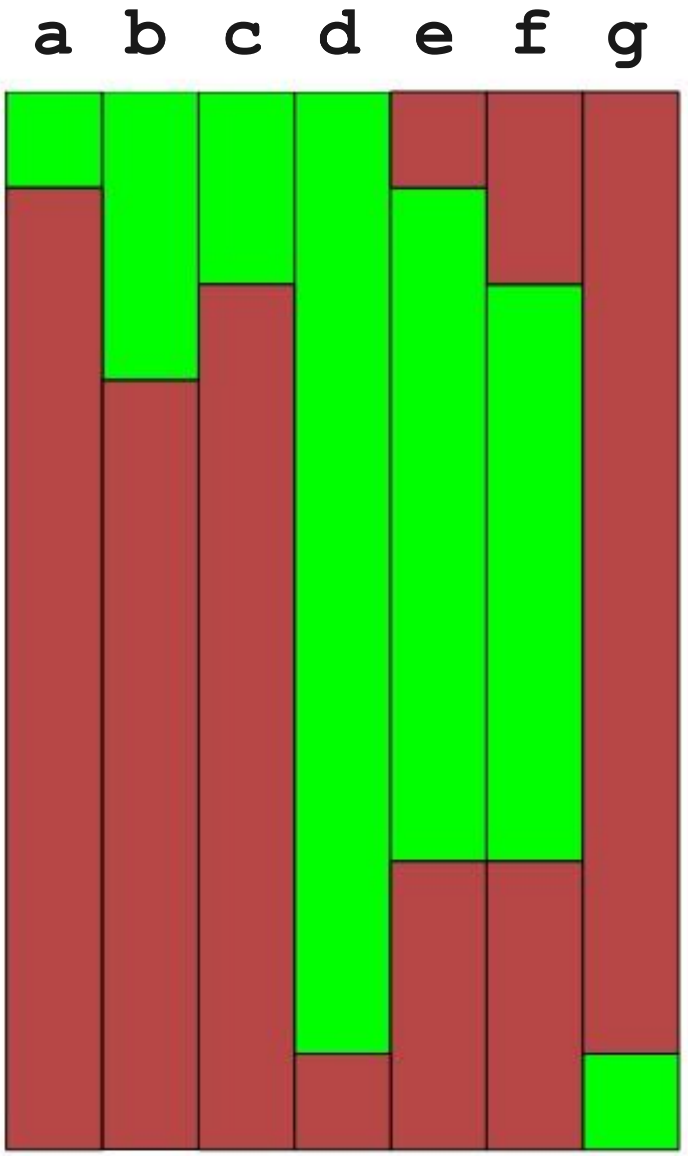
Live Ranges and Live Intervals



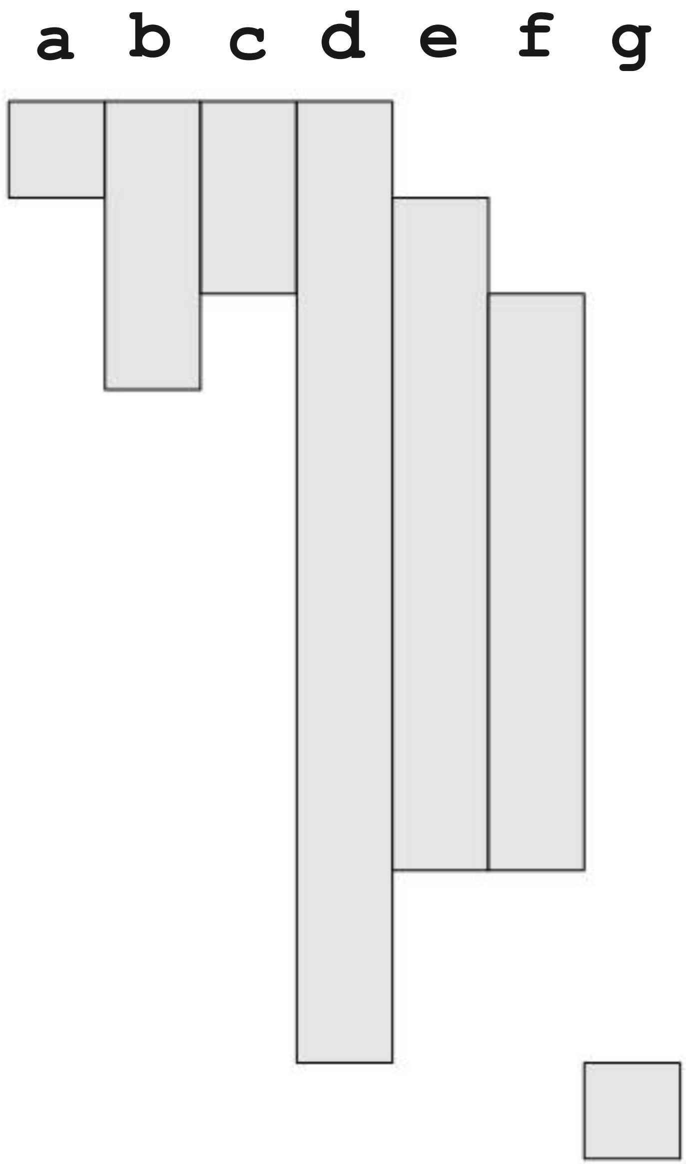


Register Allocation with Live Intervals

- 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.



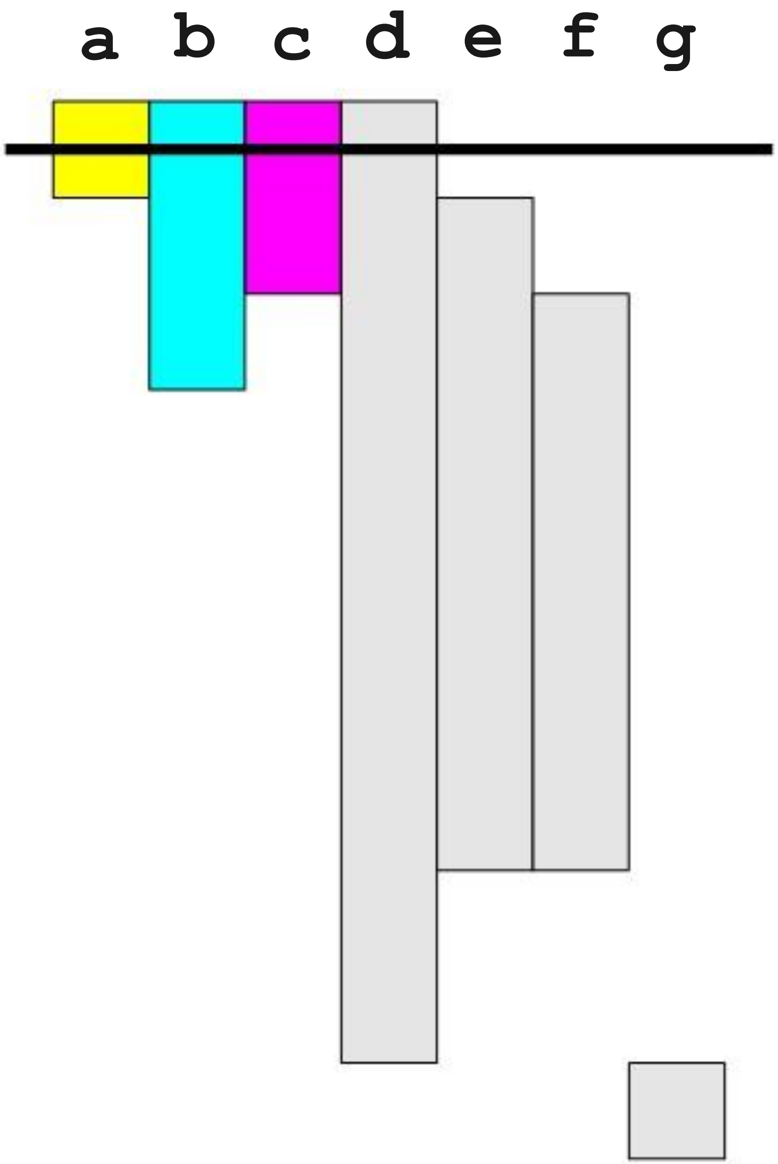
Register Allocation with Live Intervals



Free Registers

Greedy algorithm

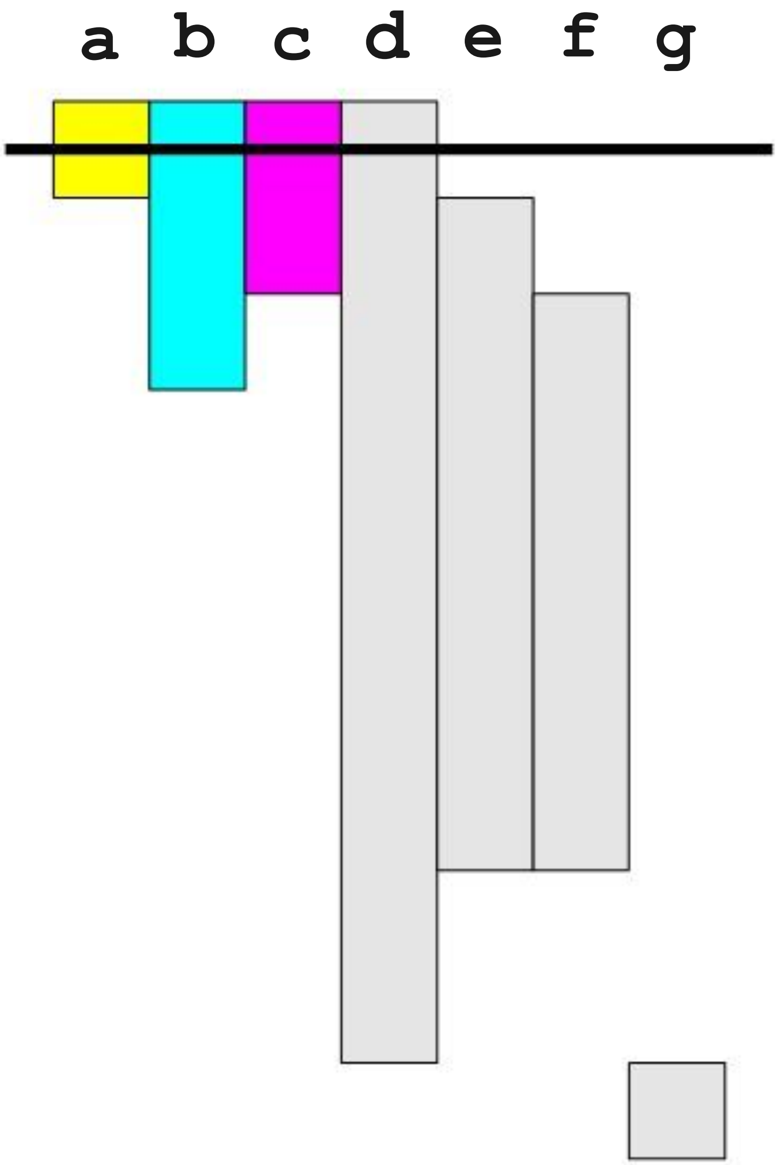
Register Allocation with Live Intervals



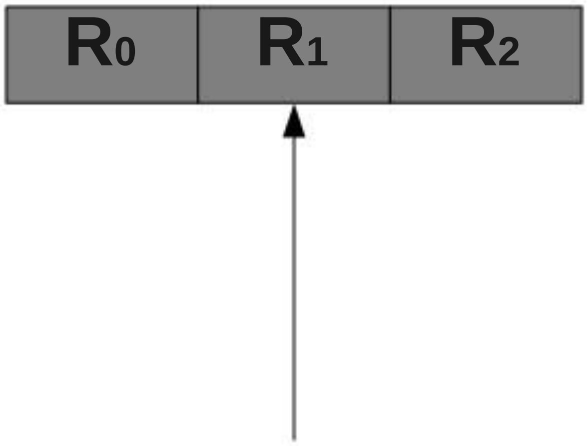
Free Registers

R_0	R_1	R_2

Register Allocation with Live Intervals



Free Registers

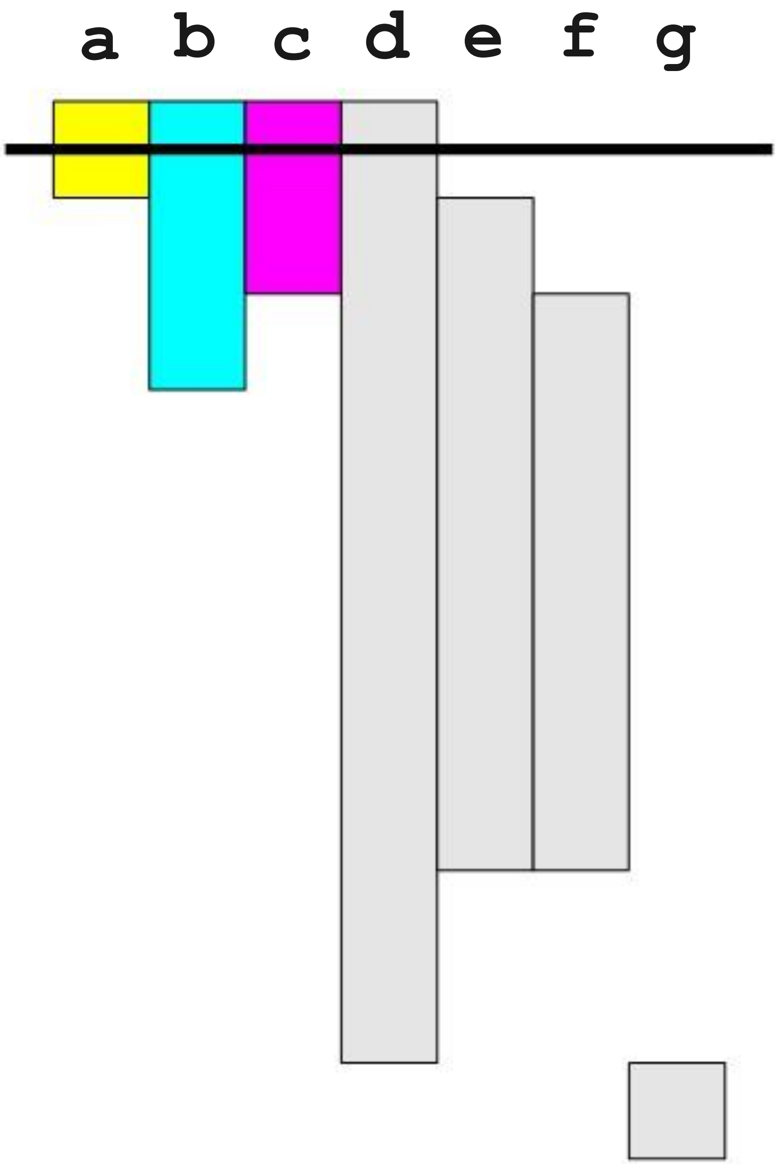


What should we do now?
(We have run out of registers!)

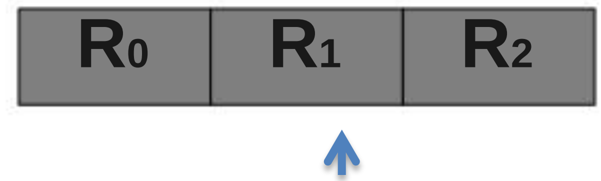
Register Spilling

- 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.**
- Note: Some register allocation algorithms can handle spilling intelligently.
- Spilling is slow, but sometimes necessary.

Register Allocation with Live Intervals

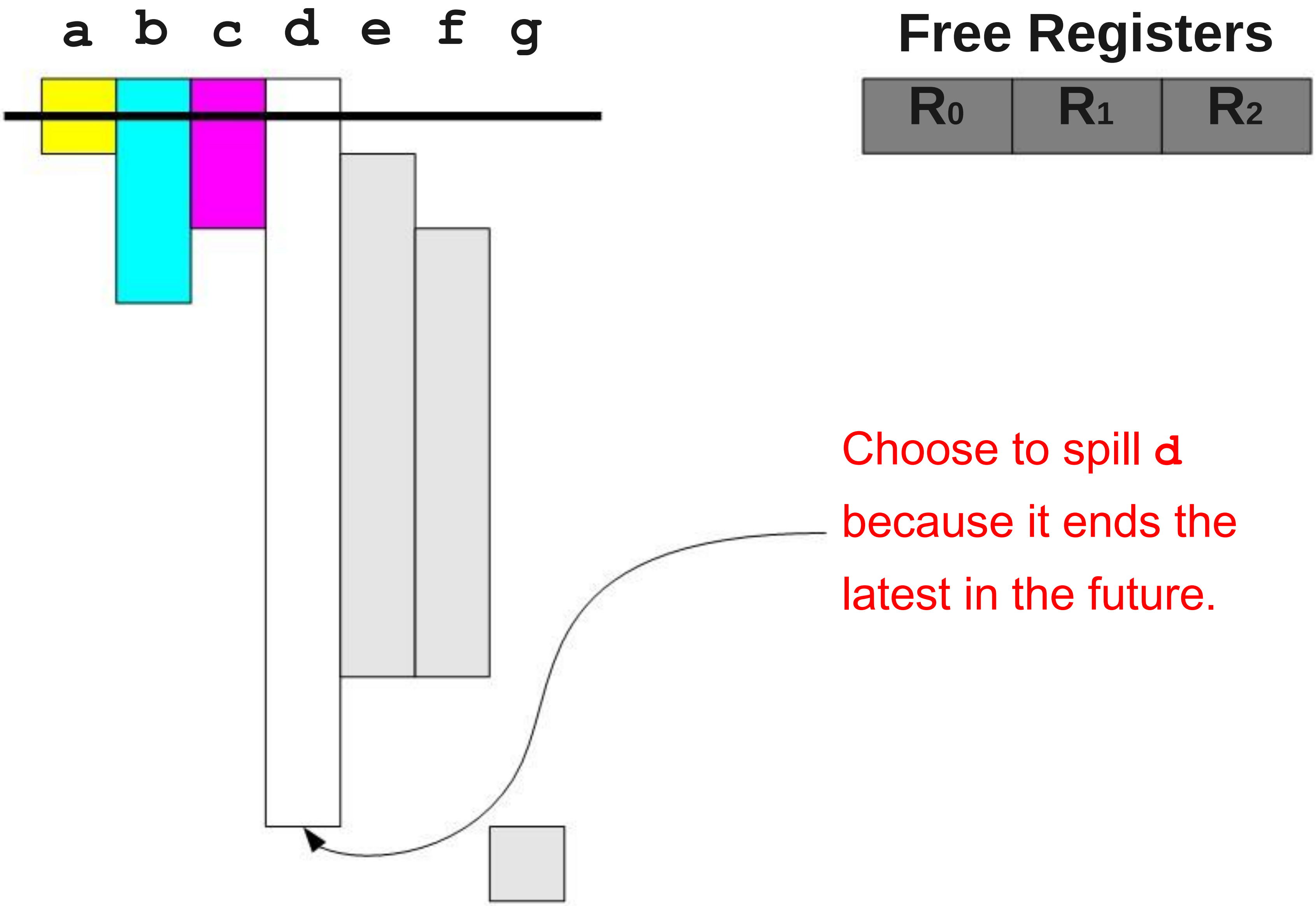


Free Registers



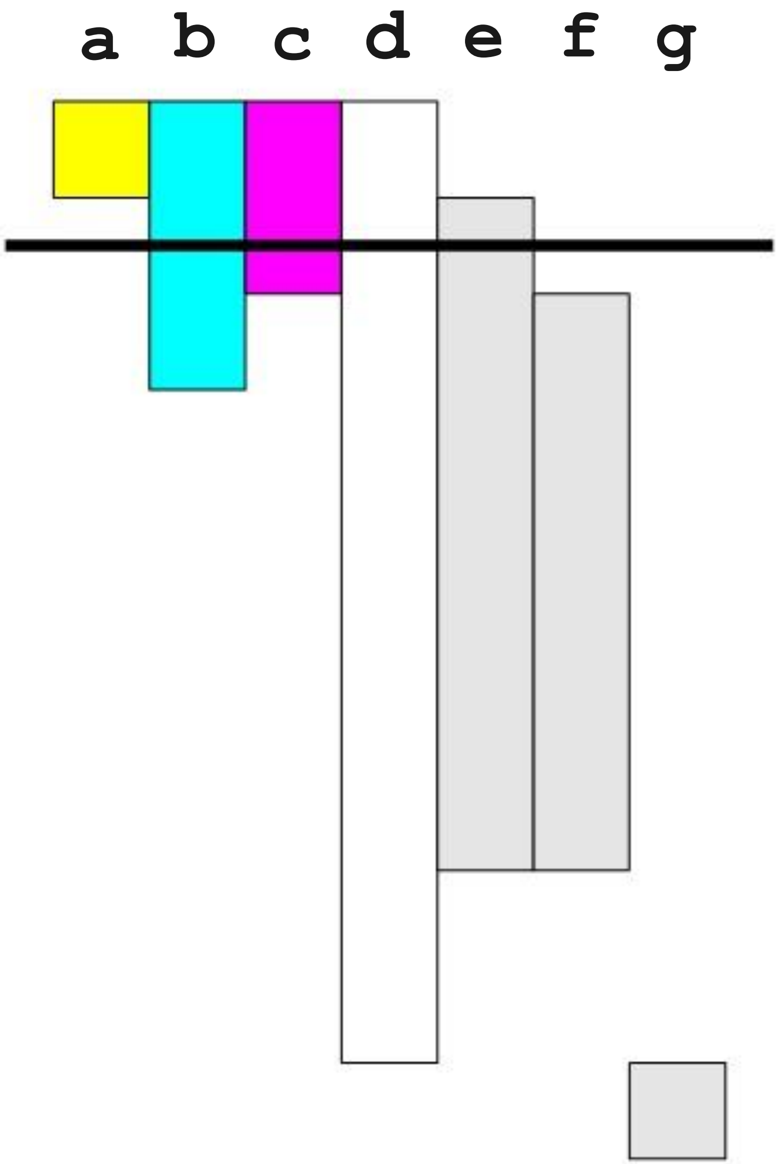
Grey color means the register is already occupied.

Register Allocation with Live Intervals



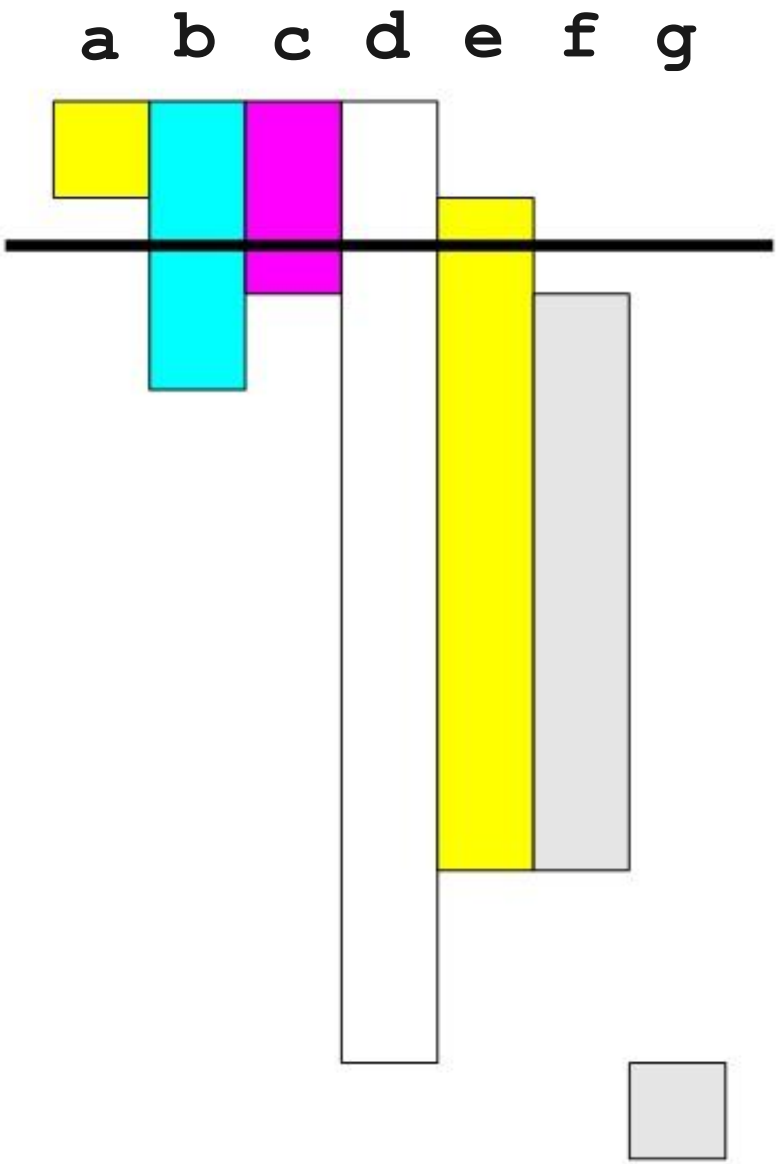
Choose to spill `d` because it ends the latest in the future.

Register Allocation with Live Intervals



Free Registers

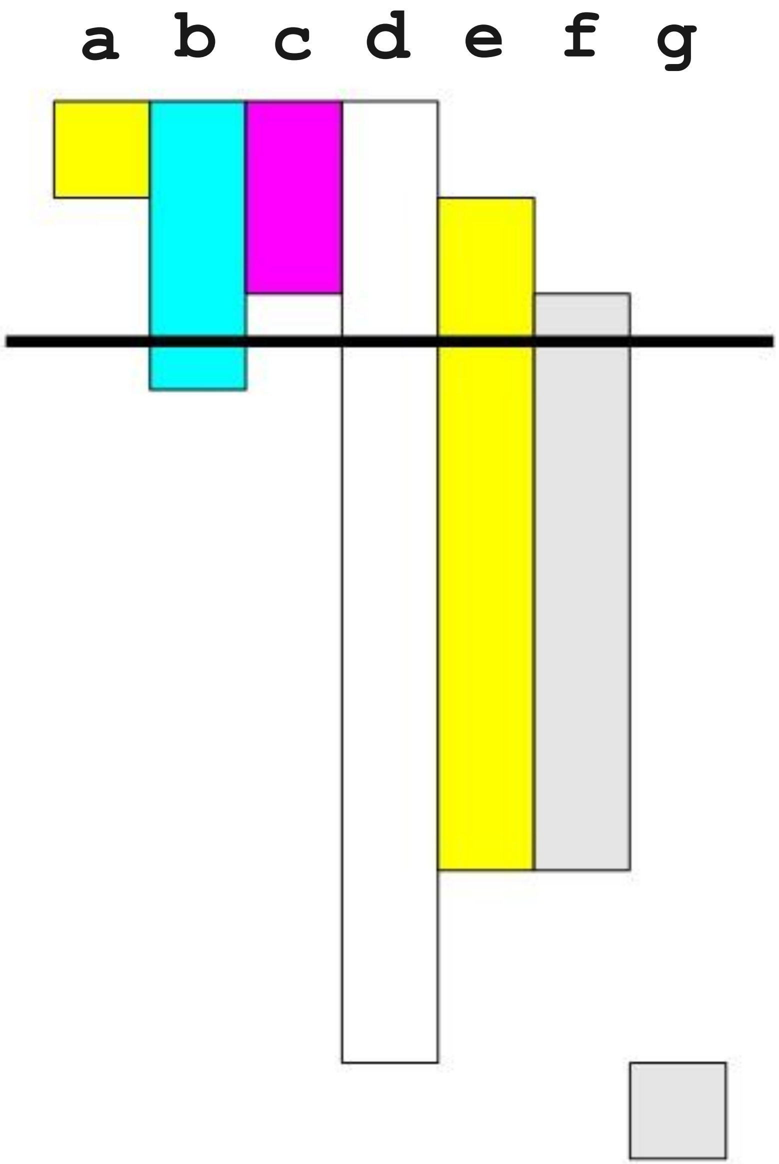
Register Allocation with Live Intervals



Free Registers

R_0	R_1	R_2

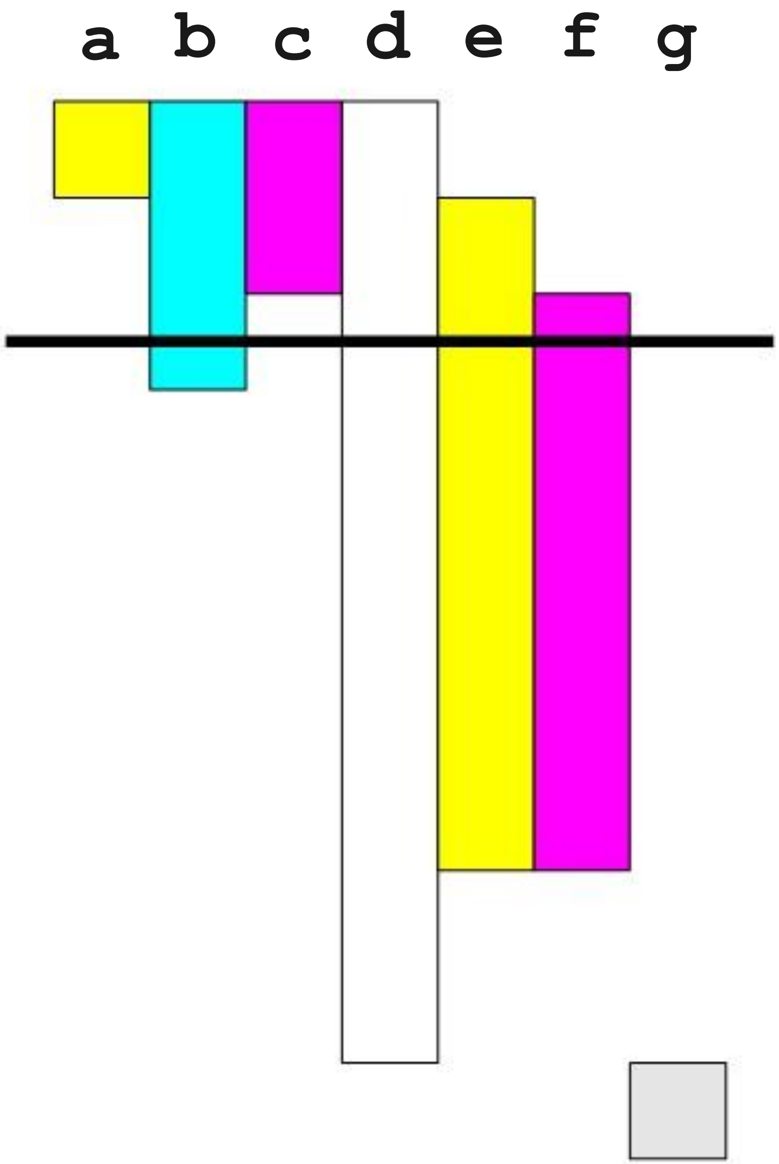
Register Allocation with Live Intervals



Free Registers

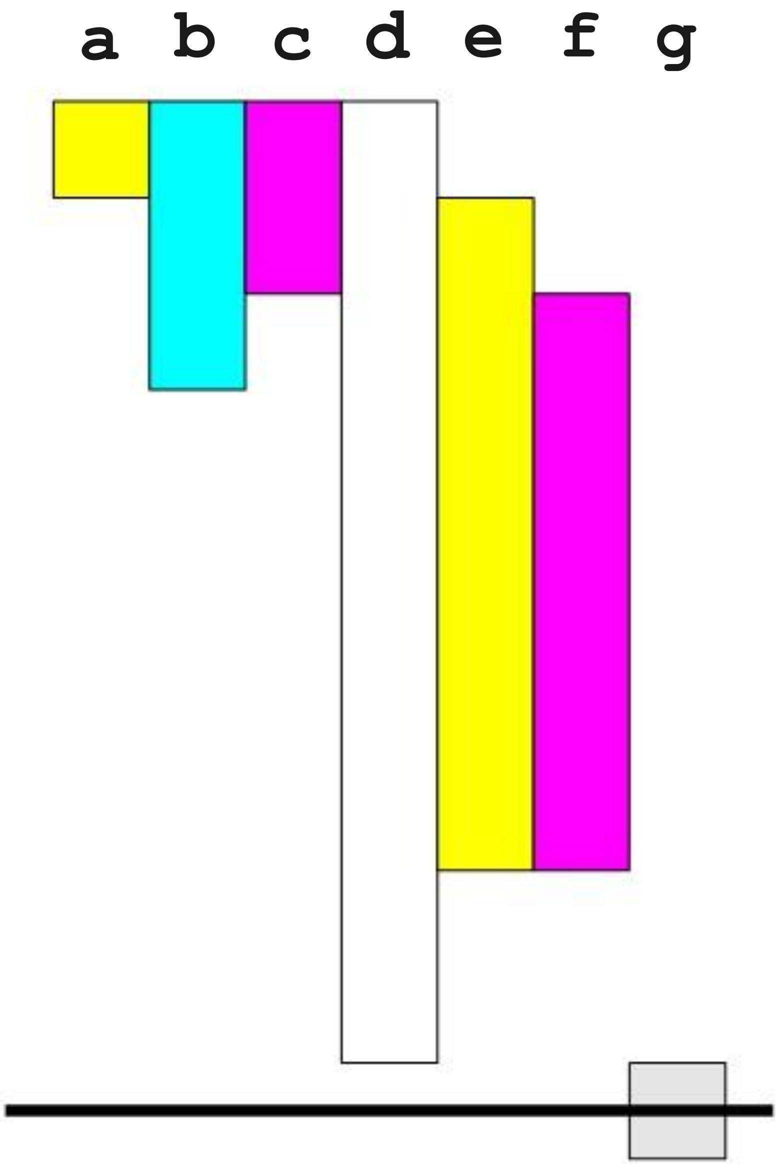
R_0	R_1	R_2

Register Allocation with Live Intervals



Free Registers

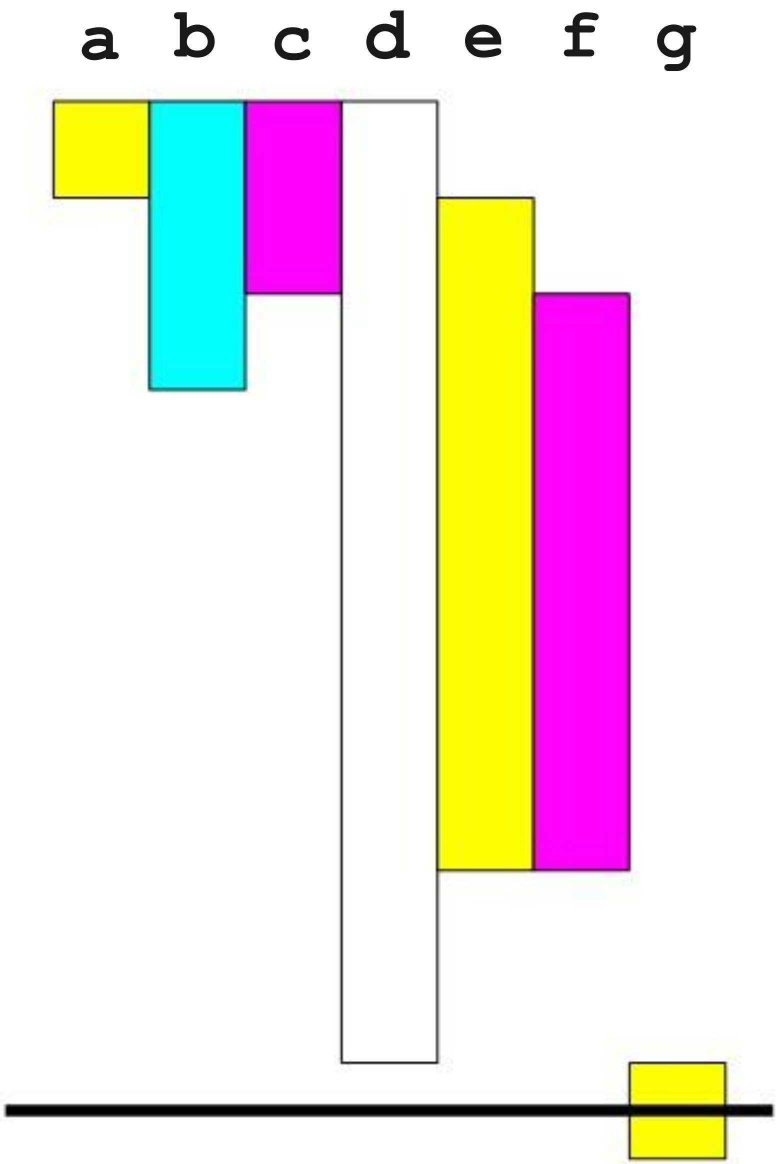
Register Allocation with Live Intervals



Free Registers

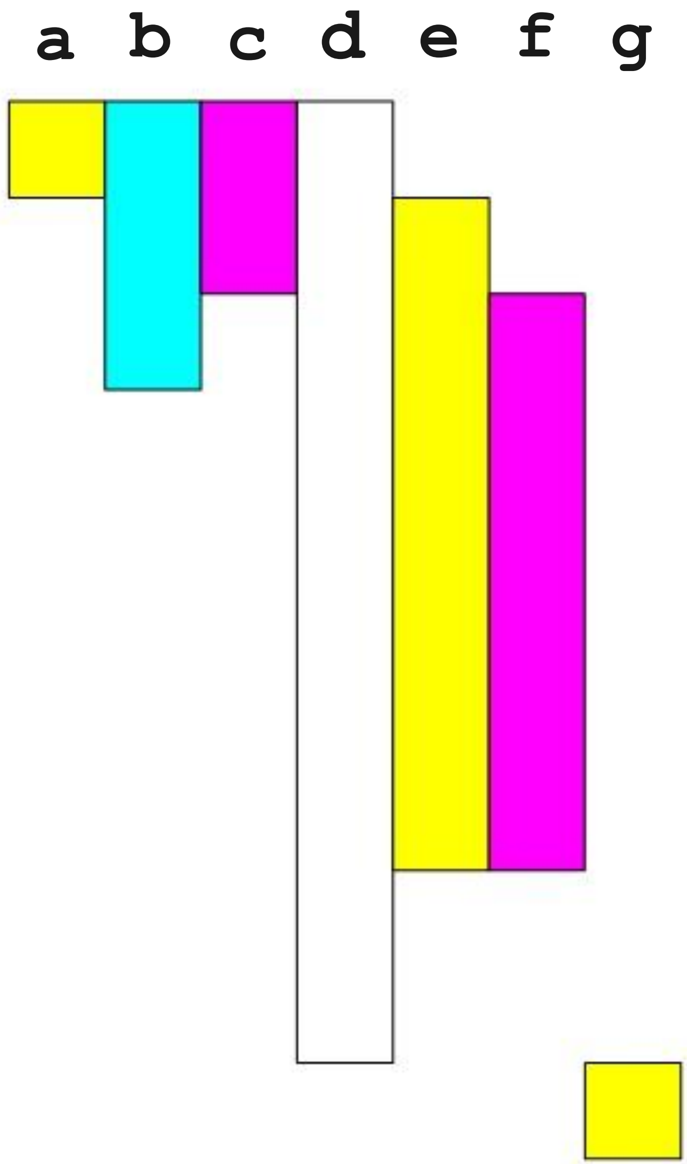
R_0	R_1	R_2

Register Allocation with Live Intervals



Free Registers

Register Allocation with Live Intervals



Next time : continue register spilling +

Other techniques for register allocation