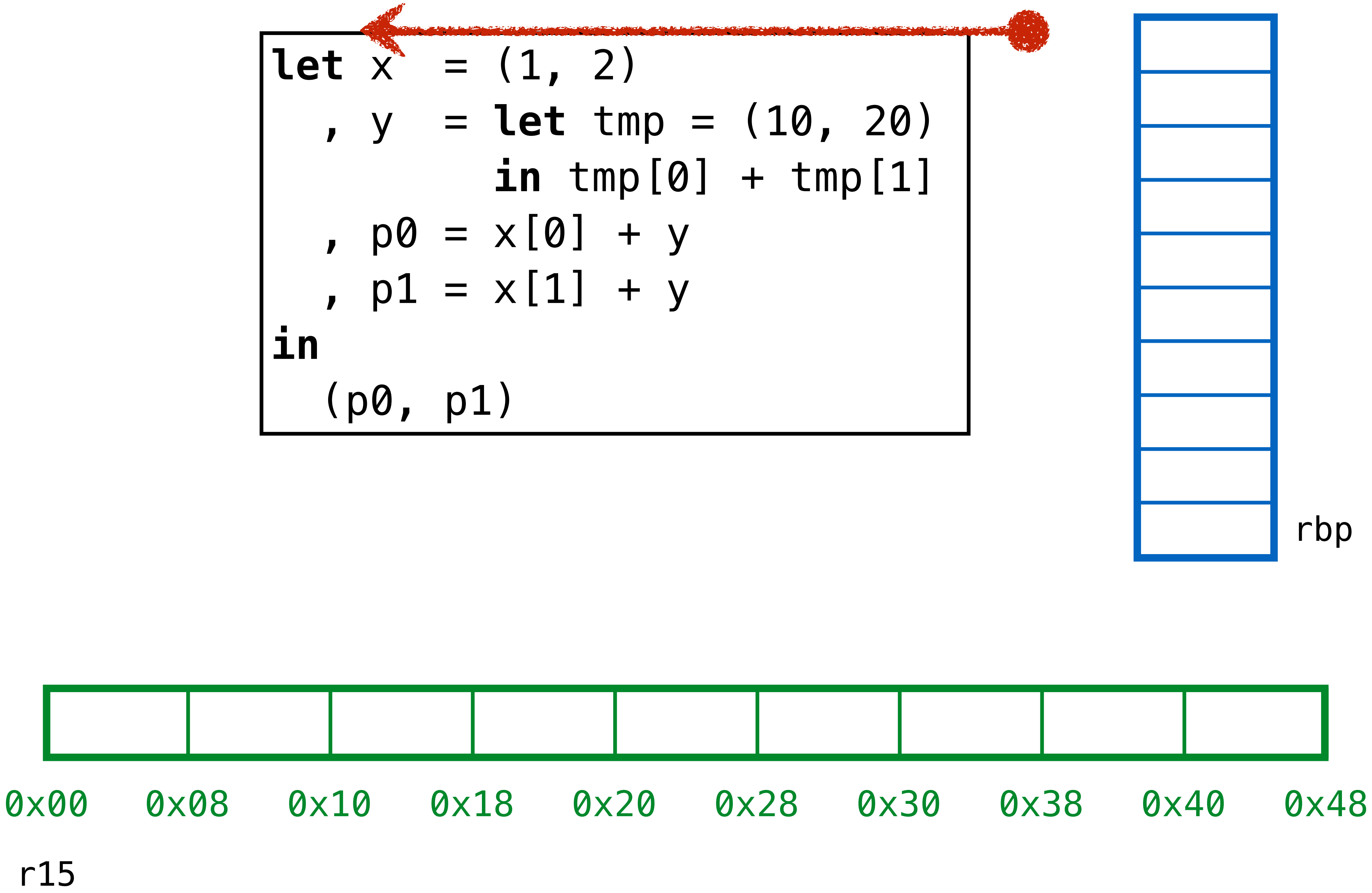
Garter

Garbage Collection

Garter / GC

Example 1

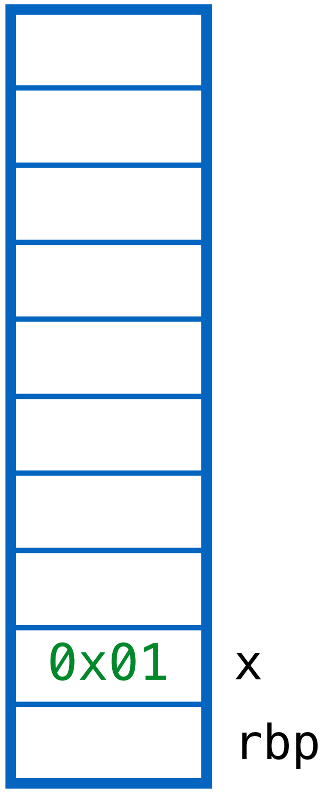
ex1: garbage at end

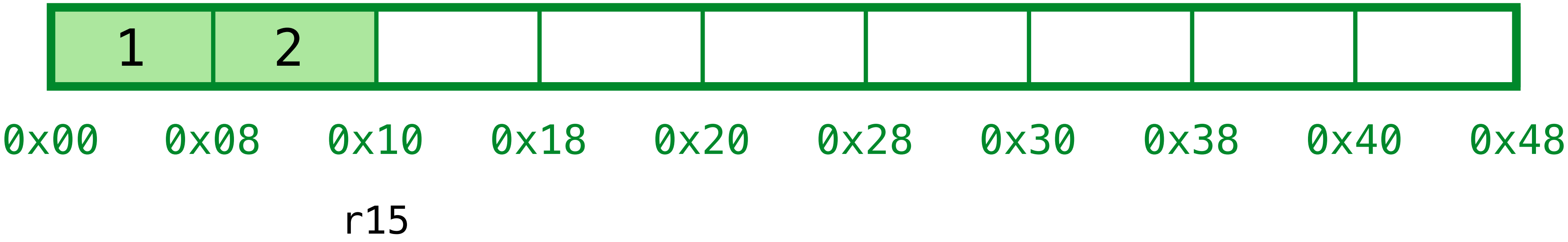


```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

ex1: garbage at end

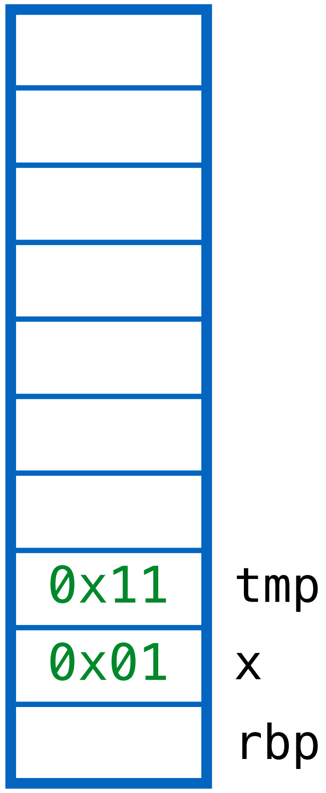
```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

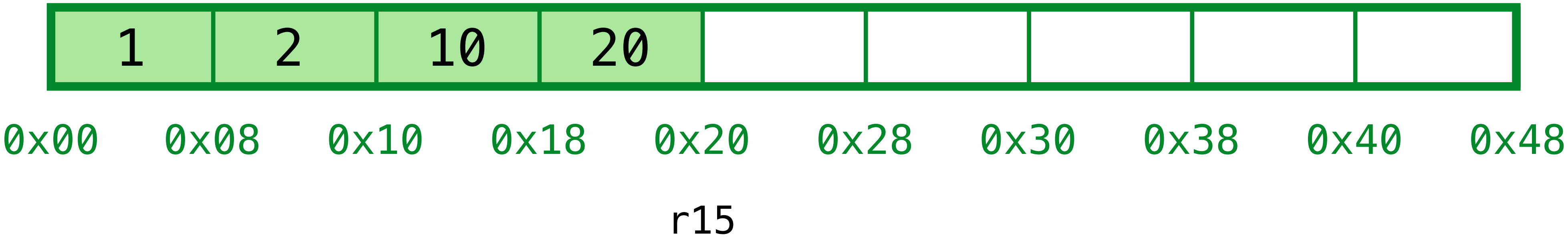




ex1: garbage at end

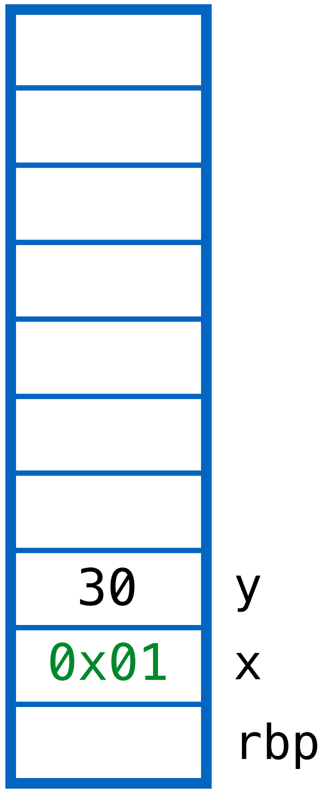
```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

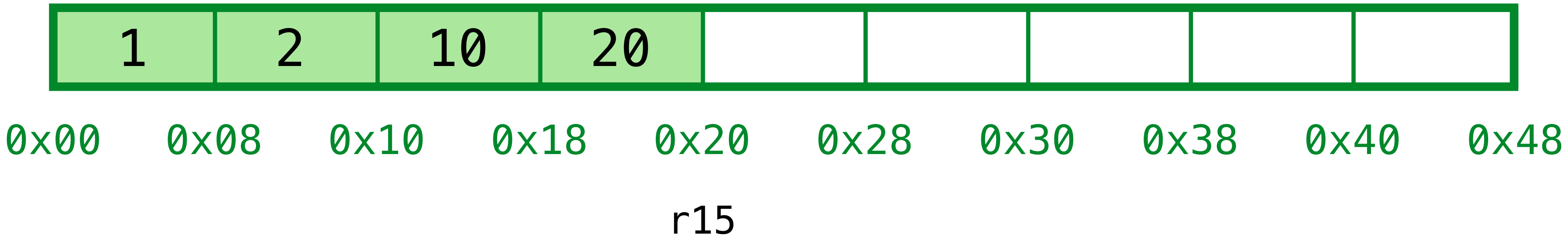




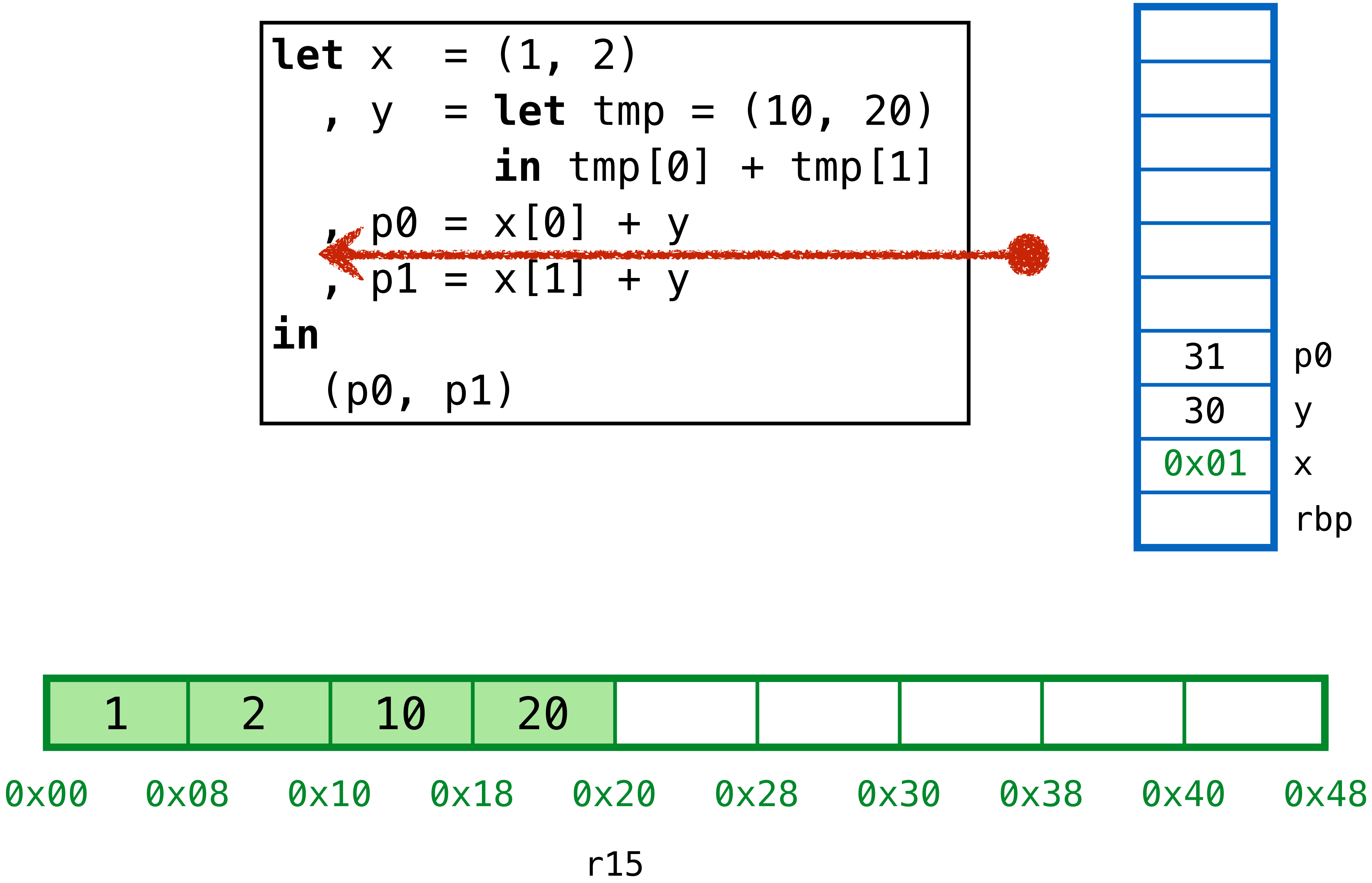
ex1: garbage at end

```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```





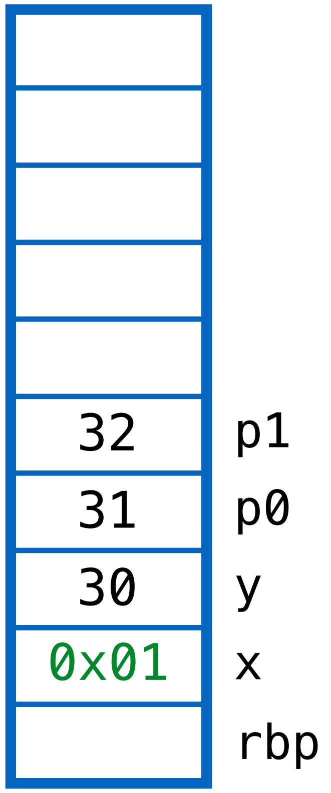
ex1: garbage at end

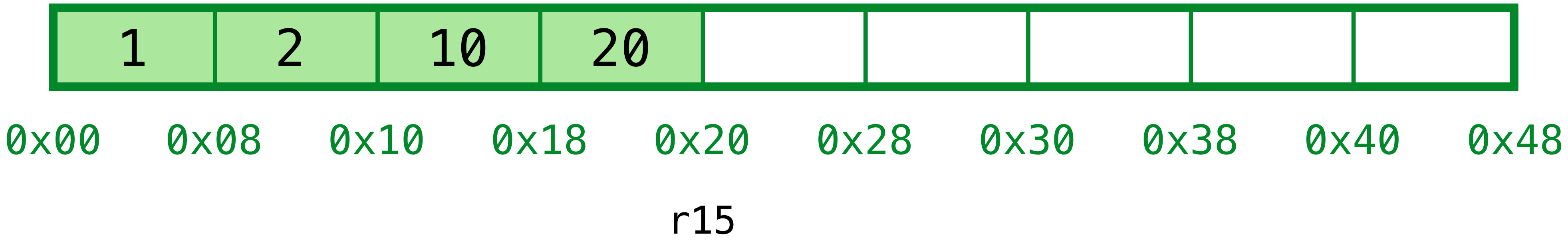


```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

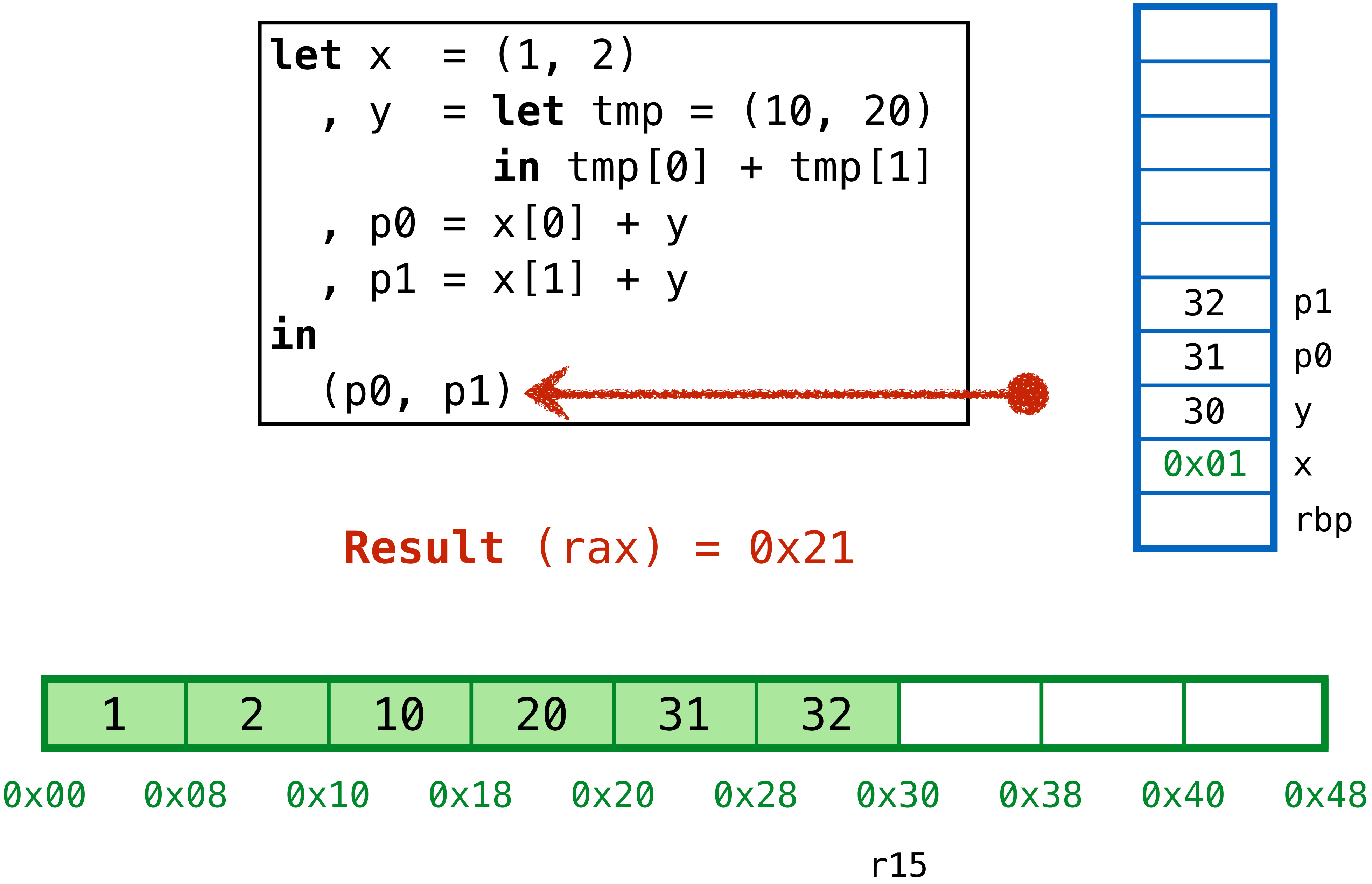
ex1: garbage at end

```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```





ex1: garbage at end



```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

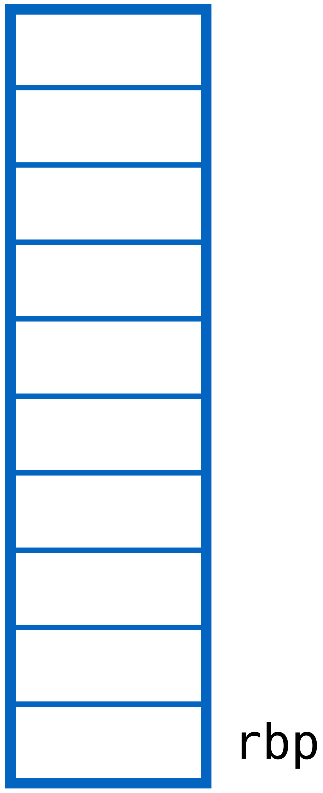
Result (rax) = 0x21

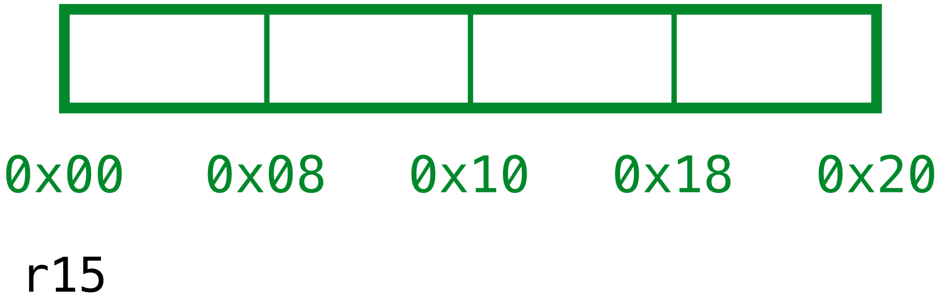
32	p1
31	p0
30	y
0x01	x
	rbp

0x00	0x08	0x10	0x18	0x20	0x28	0x30	0x38	0x40
1	2	10	20	31	32			

ex1: garbage at end

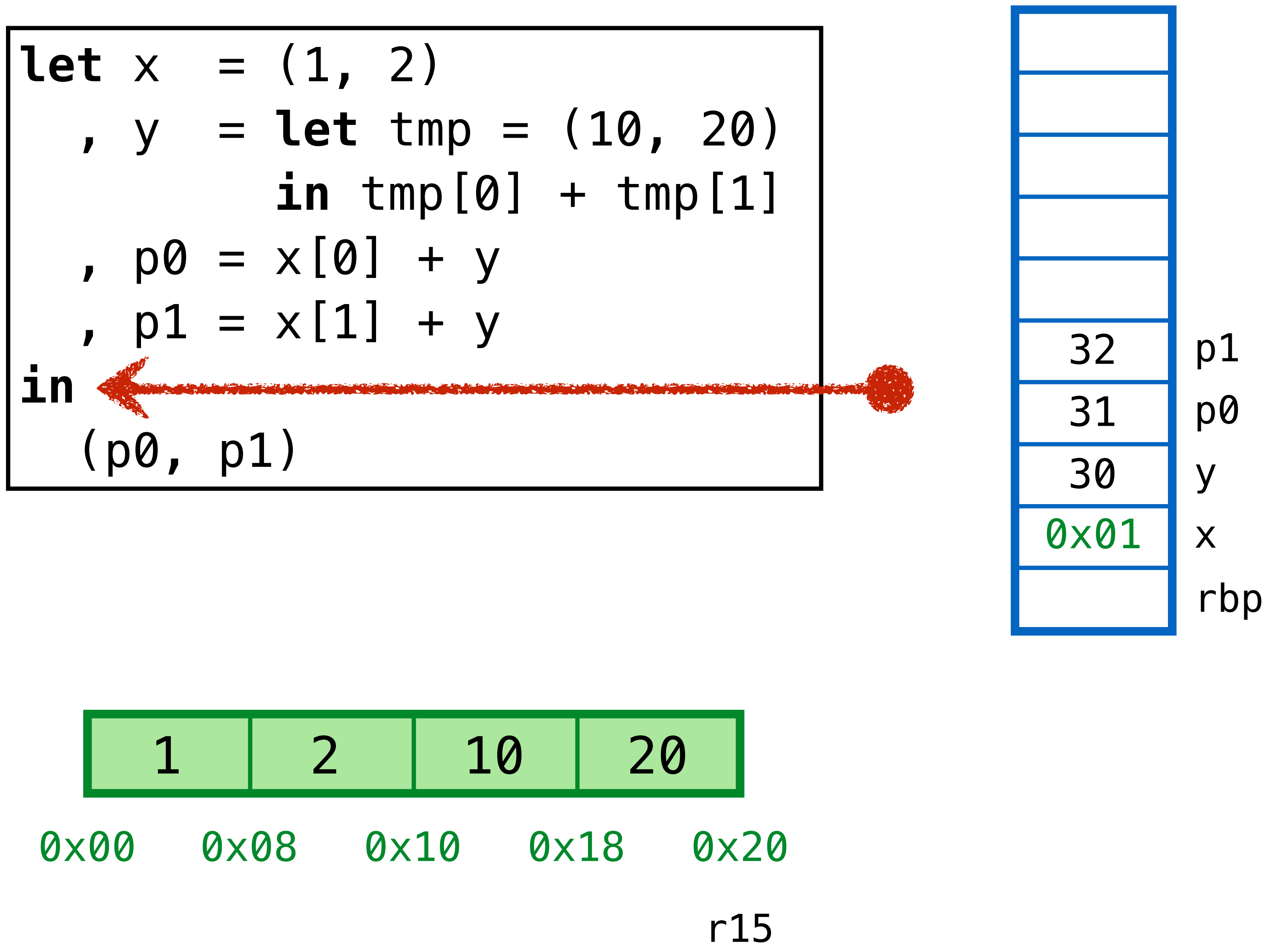
```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```





Suppose we had a smaller, 4-word heap

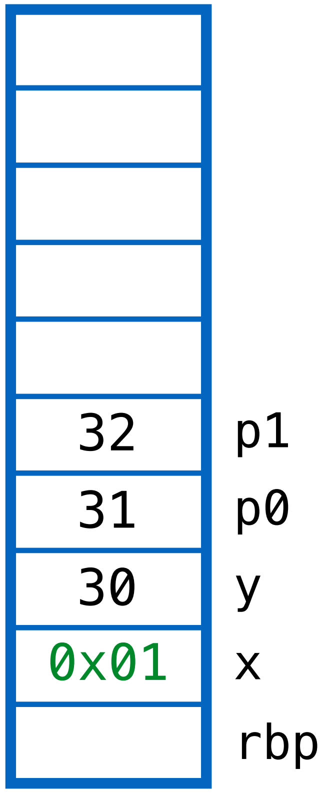
ex1: garbage at end



```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

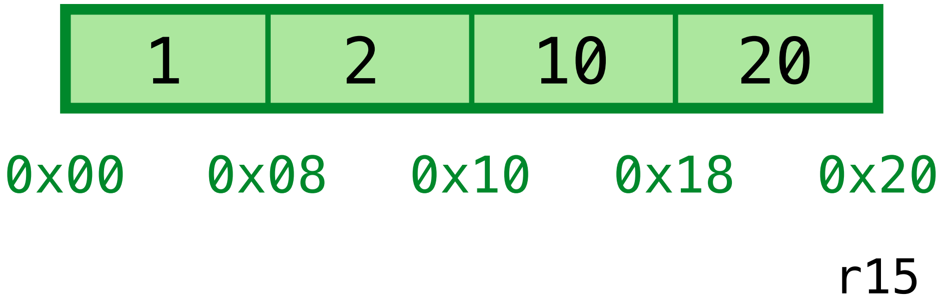
ex1: garbage at end

```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



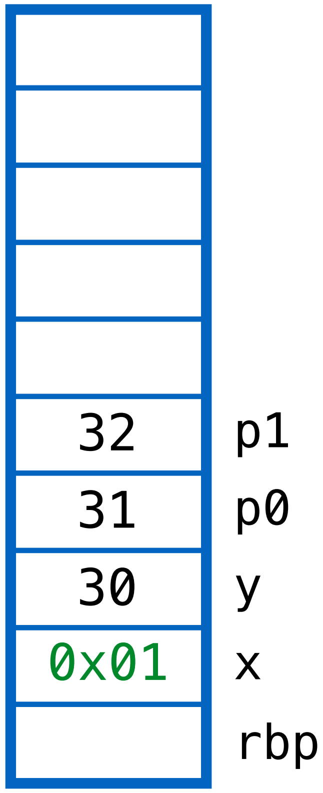
Out of memory!

Can't allocate **(p0, p1)**

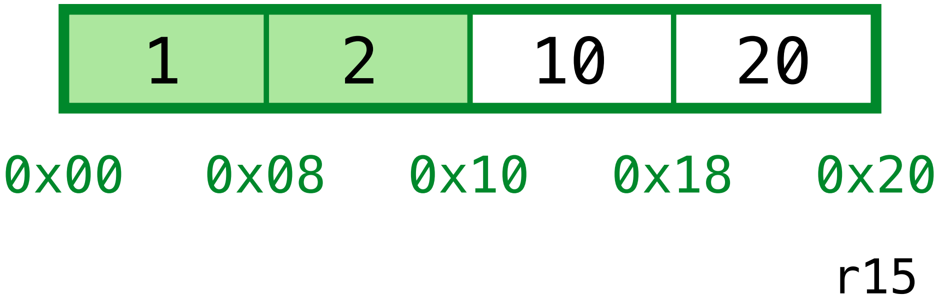


ex1: garbage at end

```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



`(10, 20)` is "garbage"

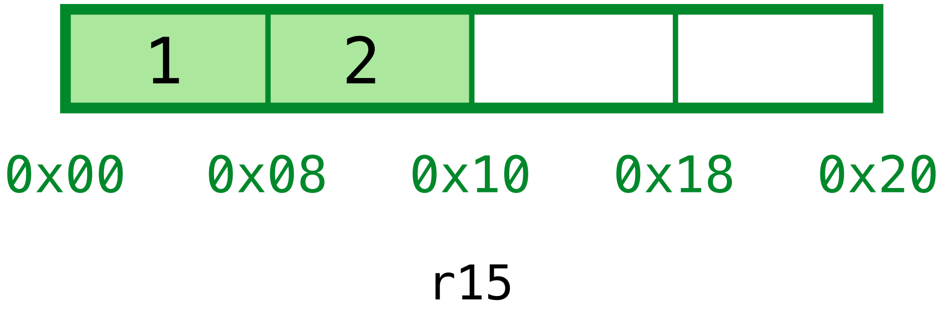


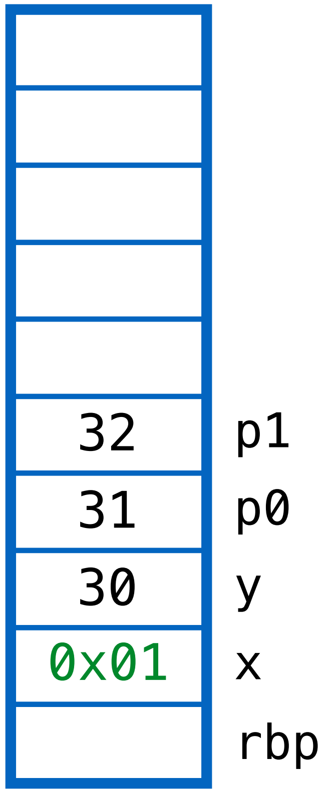
Q: How to determine if cell is garbage?

ex1: garbage at end

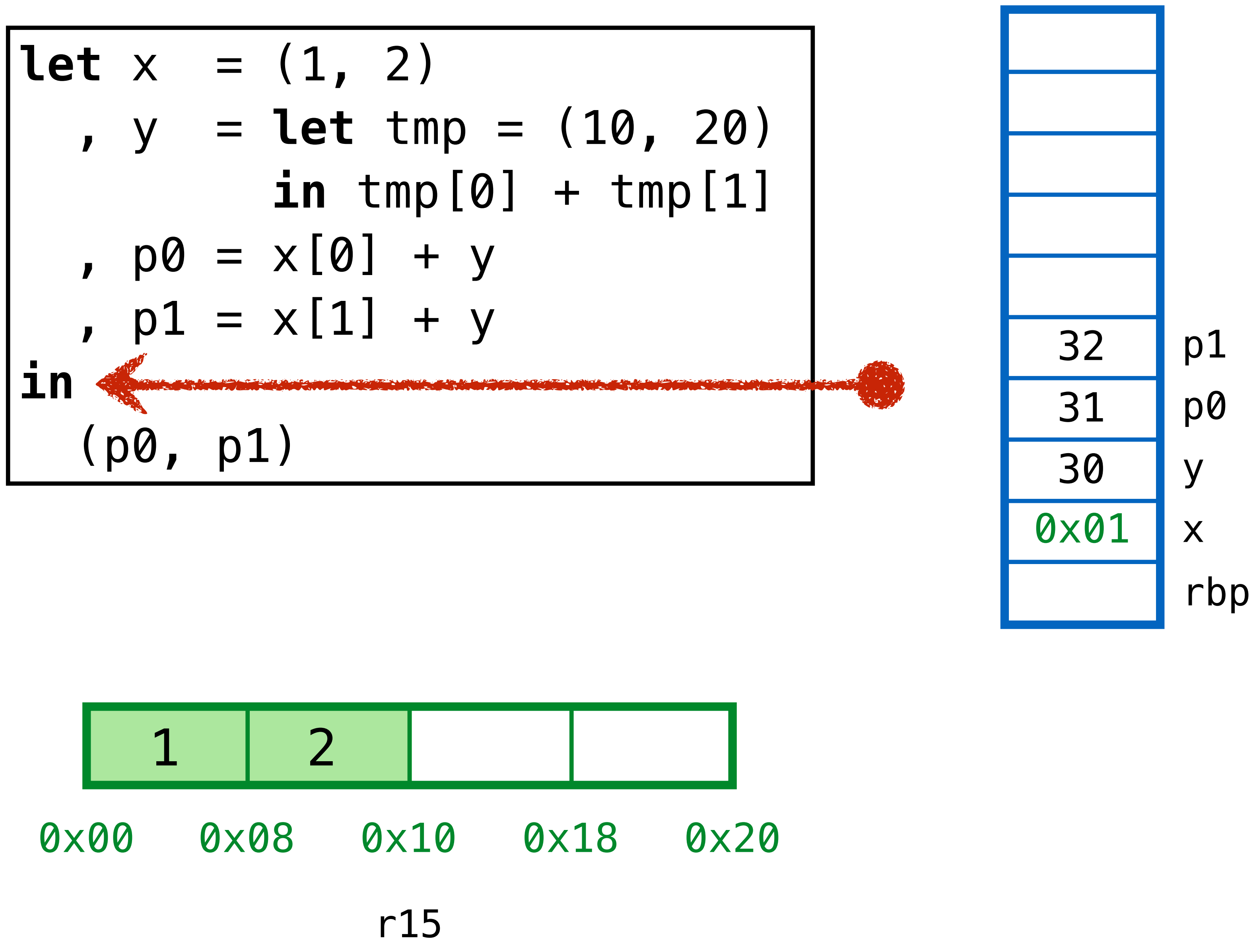
```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

(10, 20) is “garbage”





ex1: garbage at end

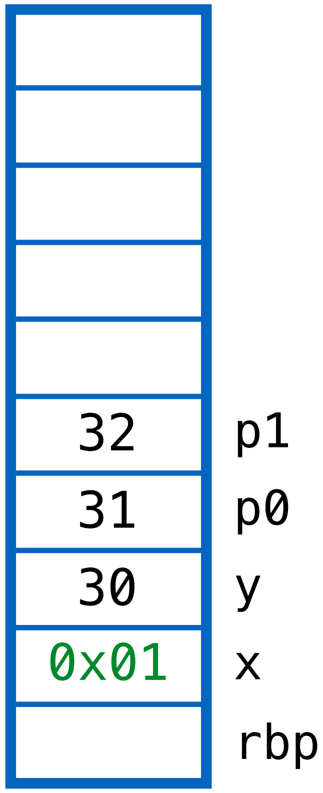


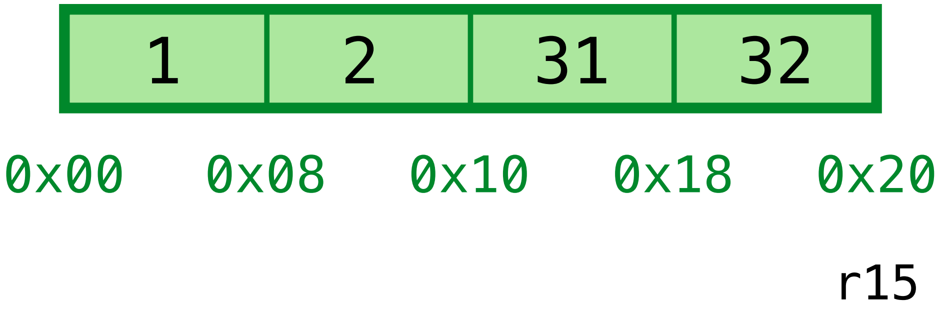
```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

ex1: garbage at end

```
let x  = (1, 2)
  , y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

Result (rax) = 0x11



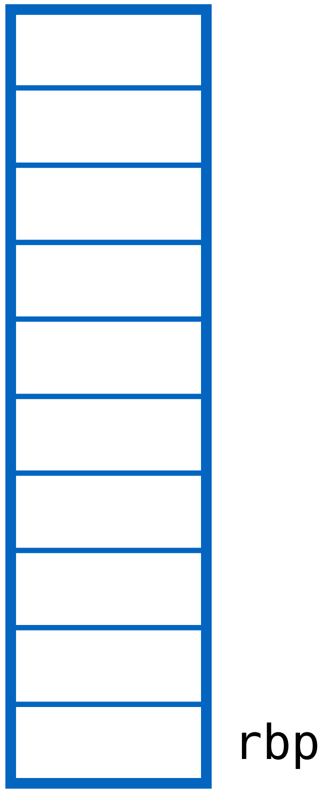


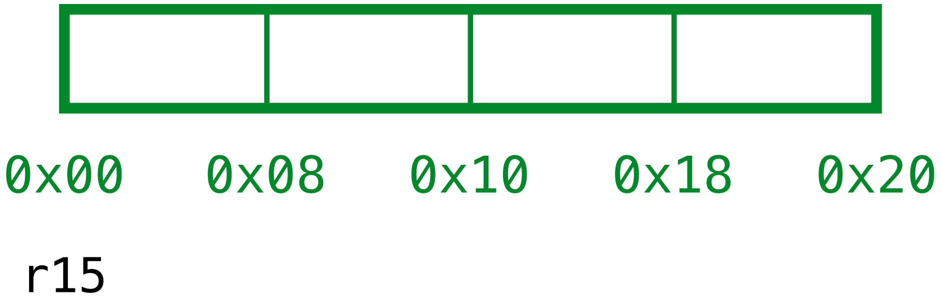
Garter / GC

Example 2

ex2: garbage in the middle

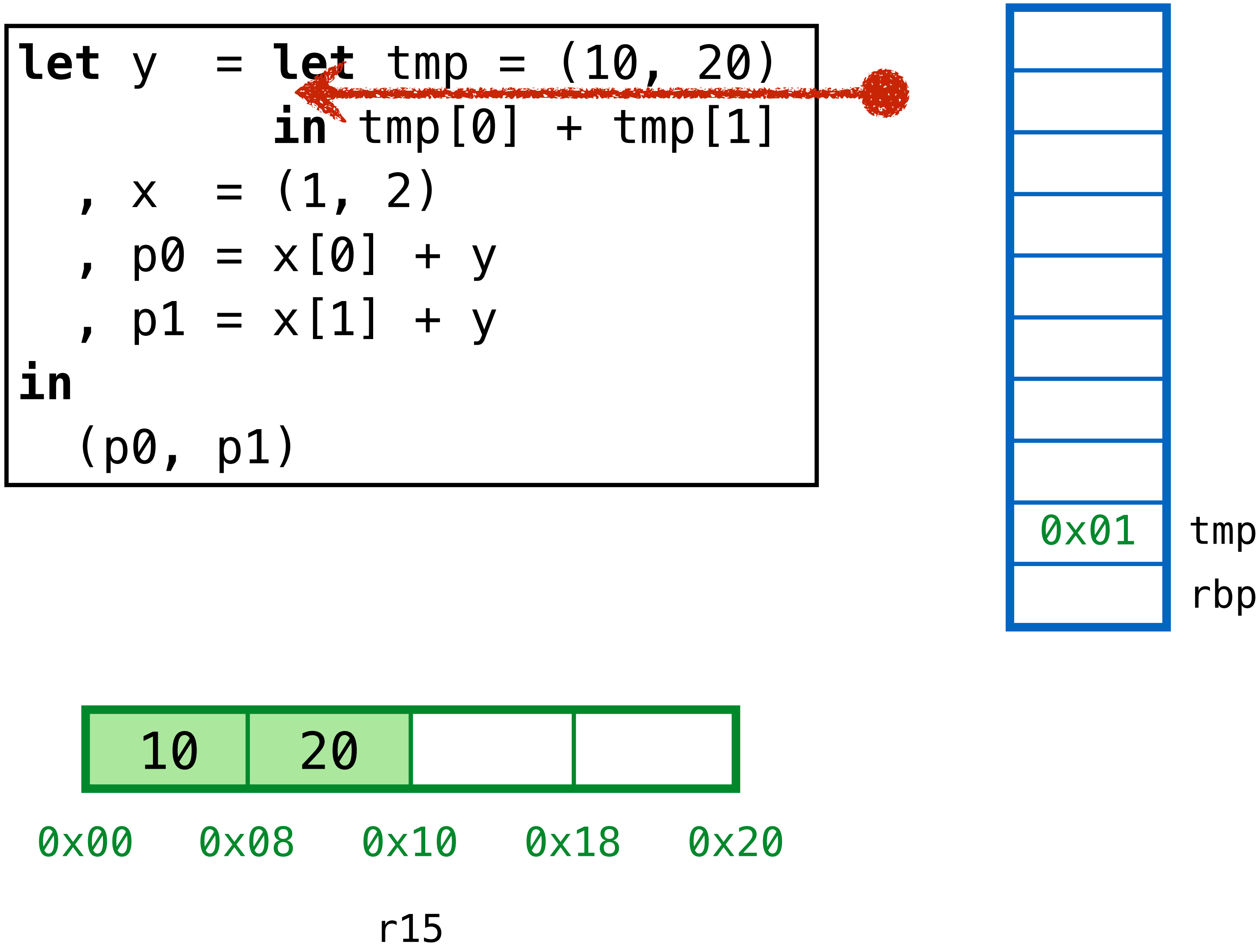
```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```





Start with a 4-word heap

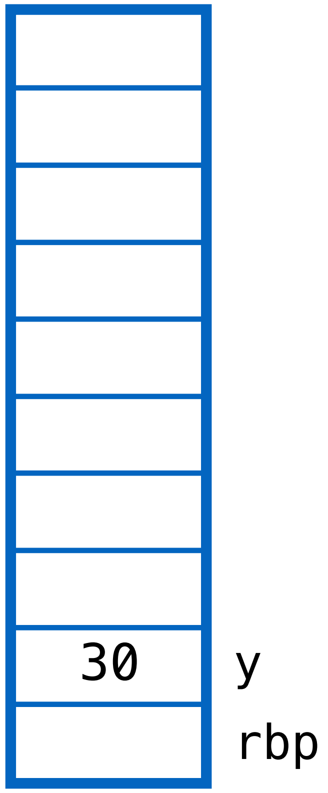
ex2: garbage in the middle

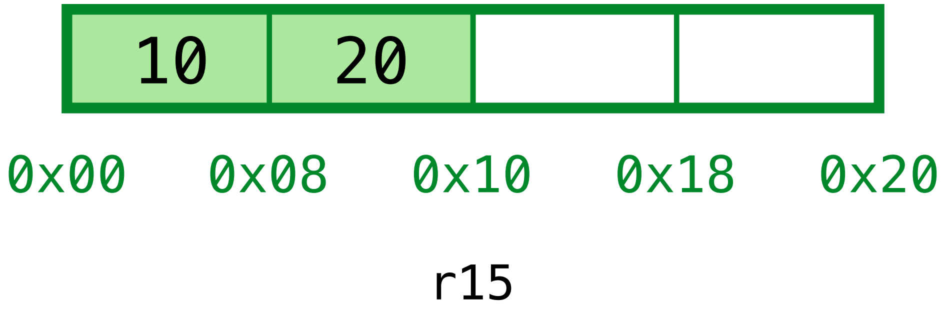


```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

ex2: garbage in the middle

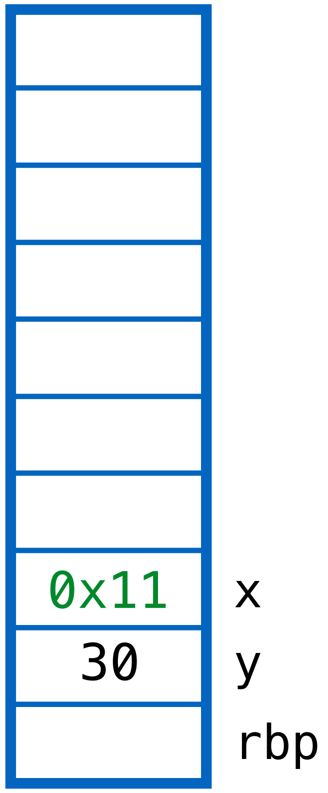
```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

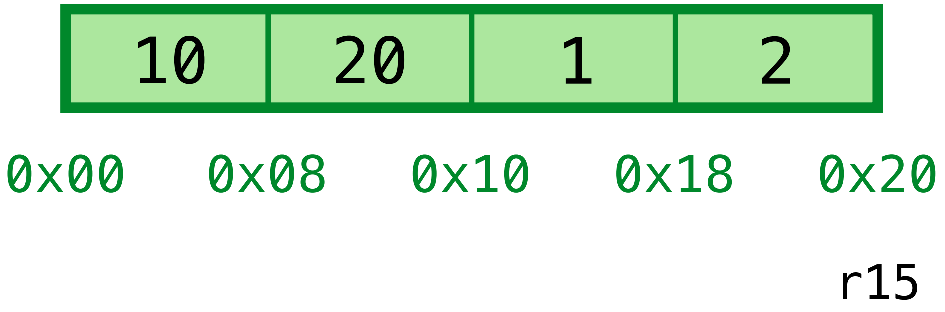




ex2: garbage in the middle

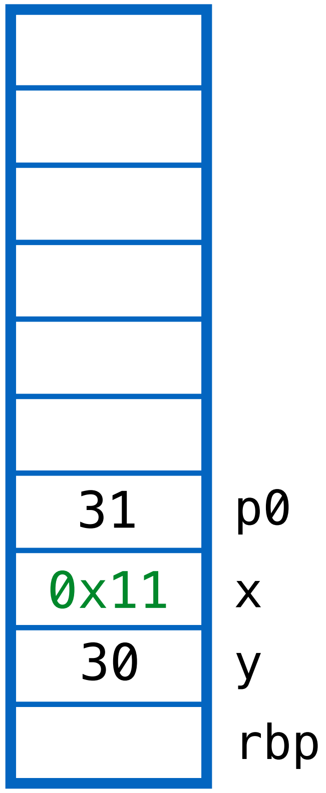
```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

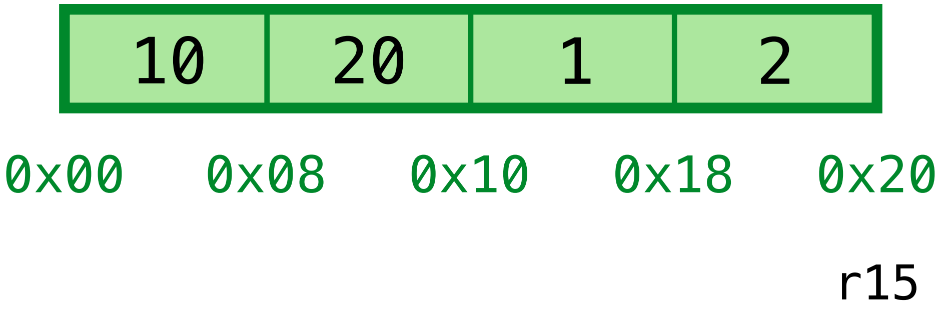




ex2: garbage in the middle

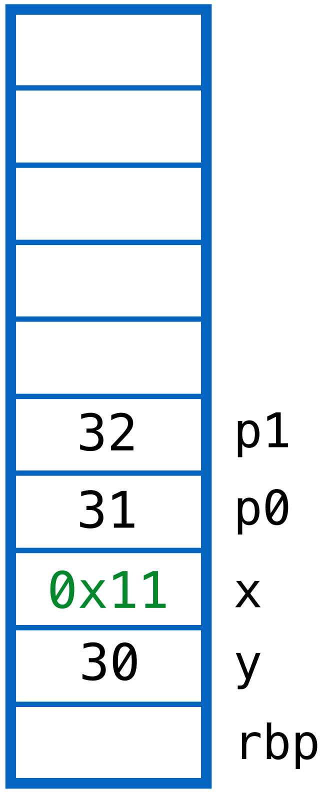
```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



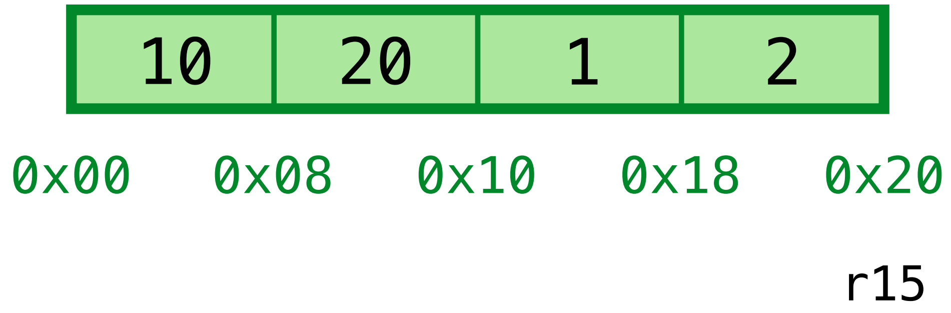


ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



32	p1
31	p0
0x11	x
30	y
	rbp

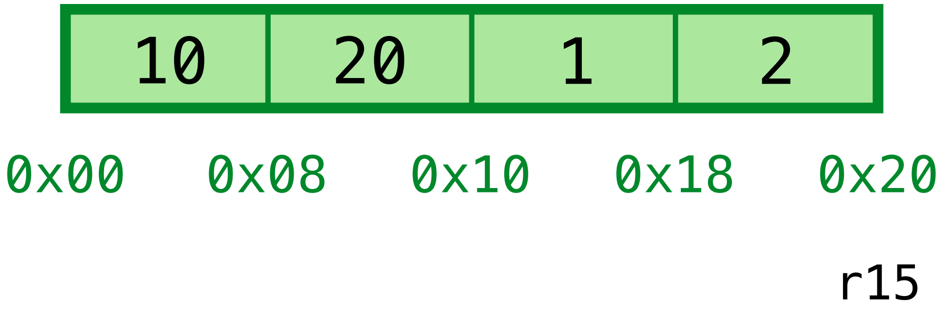


10	20	1	2	
0x00	0x08	0x10	0x18	0x20
				r15

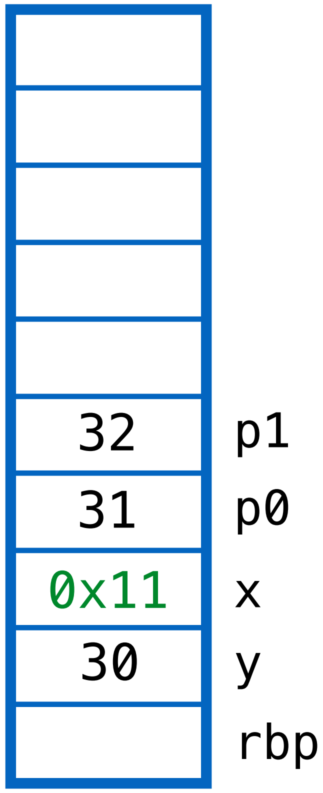
ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

Out of memory!
Can’t allocate **`(p0, p1)`**



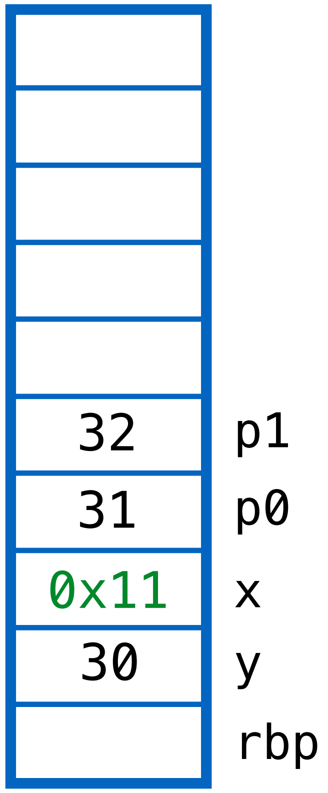
0x00	0x08	0x10	0x18	0x20
10	20	1	2	
				r15



32	p1
31	p0
0x11	x
30	y
	rbp

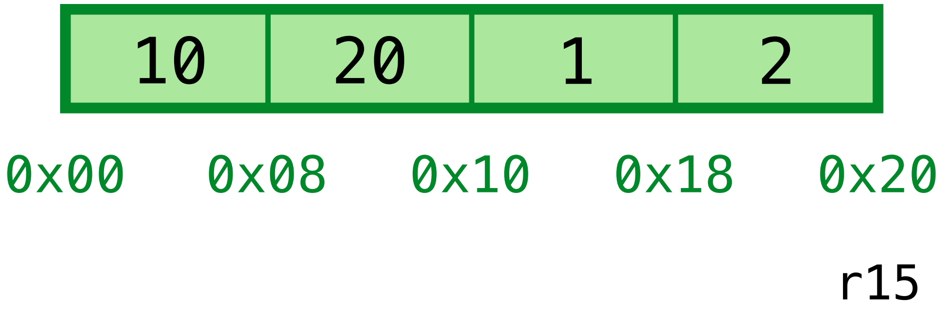
ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



32	p1
31	p0
0x11	x
30	y
	rbp

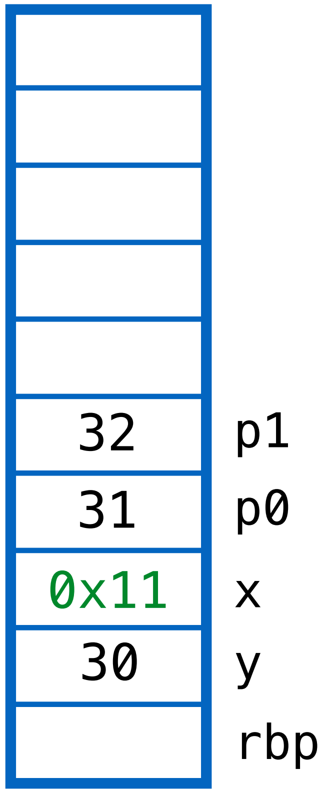
Lets reclaim & recycle garbage!



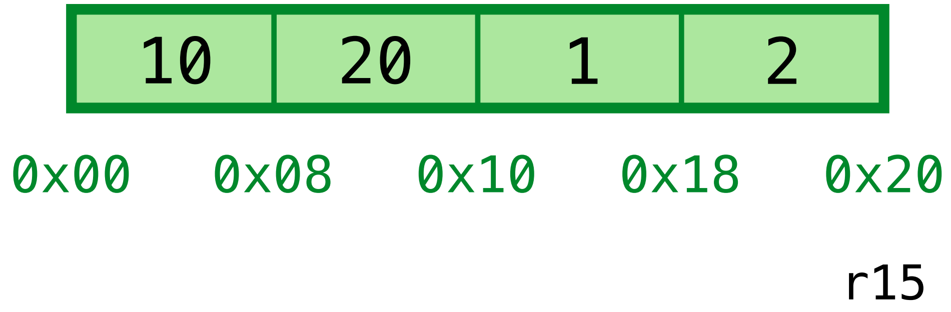
10	20	1	2	
0x00	0x08	0x10	0x18	0x20
				r15

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



Lets reclaim & recycle garbage!

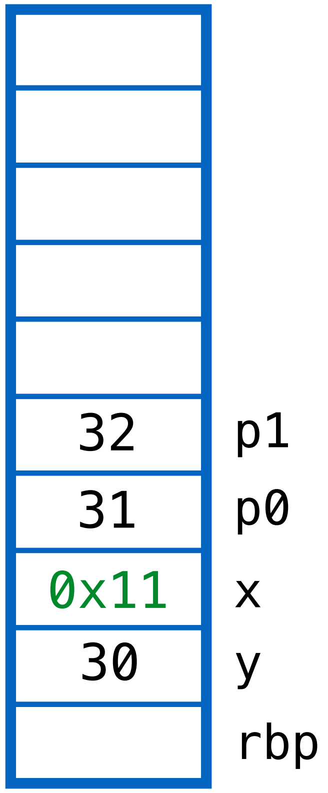


QUIZ: Which cells are garbage?

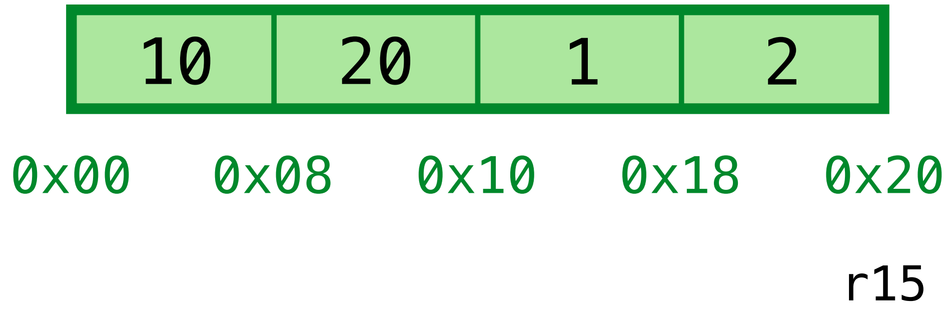
(A) 0x00, 0x08 (B) 0x08, 0x10 (C) 0x18, 0x20 (D) None (E) All

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



Lets reclaim & recycle garbage!

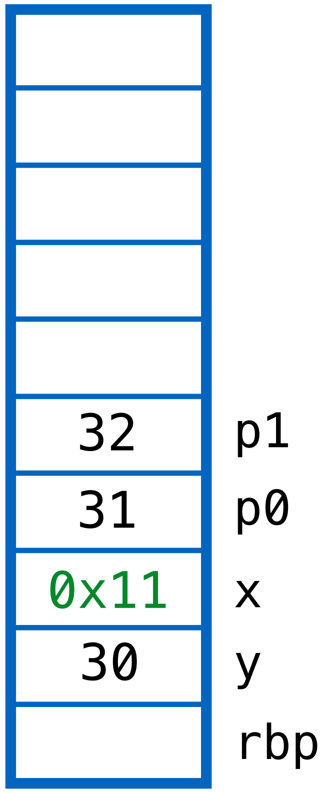


QUIZ: Which cells are garbage?

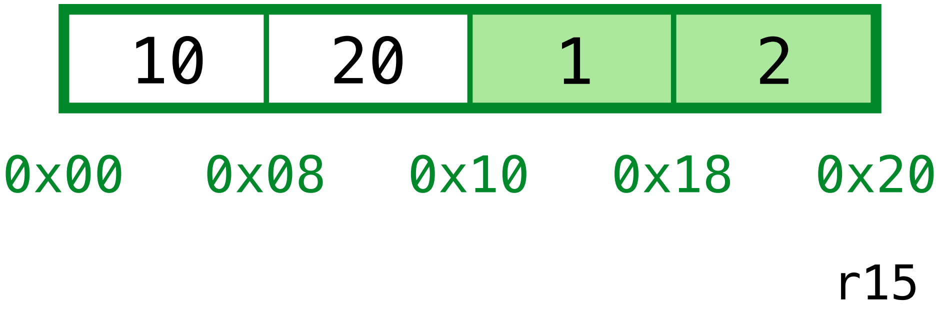
Those that are *not reachable from stack*

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



Lets reclaim & recycle garbage!

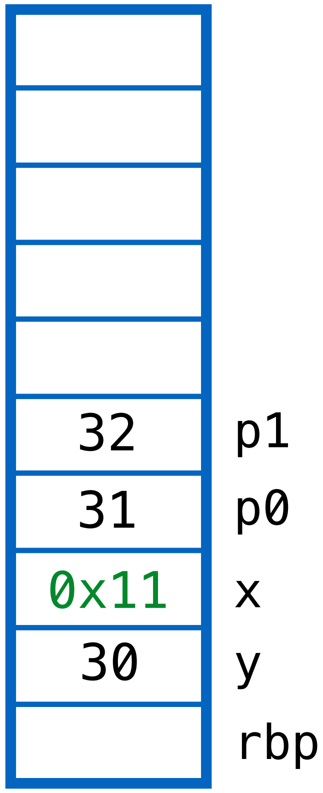


QUIZ: Which cells are garbage?

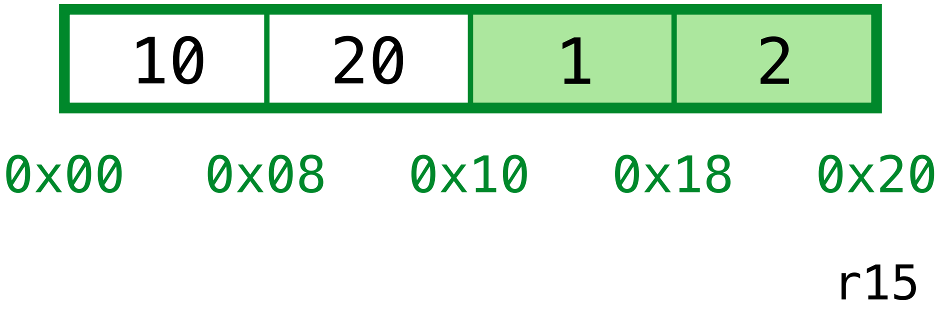
Those that are *not reachable from stack*

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



Lets reclaim & recycle garbage!

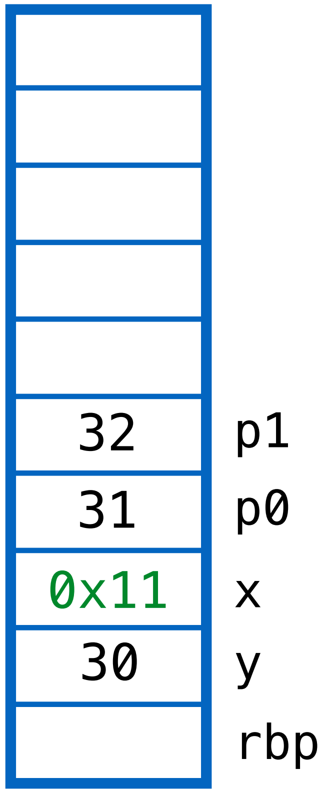


Q: How to reclaim space?

Why is it not enough to rewind `r15`?

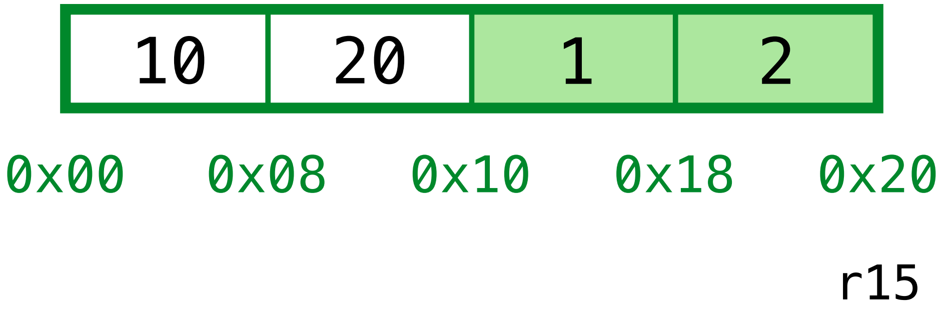
ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



32	p1
31	p0
0x11	x
30	y
	rbp

Lets reclaim & recycle garbage!

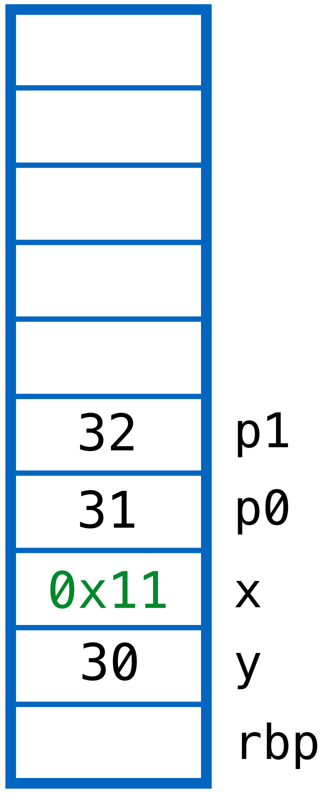


Why is it not enough to rewind `r15`?

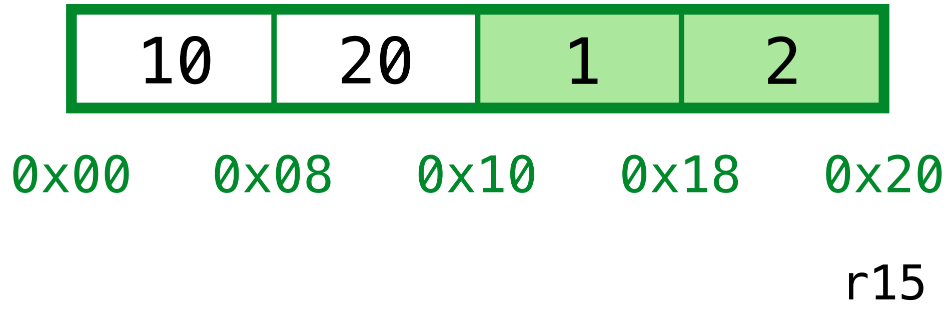
Want free space to be *contiguous* (i.e. go to end of heap)

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



Lets reclaim & recycle garbage!

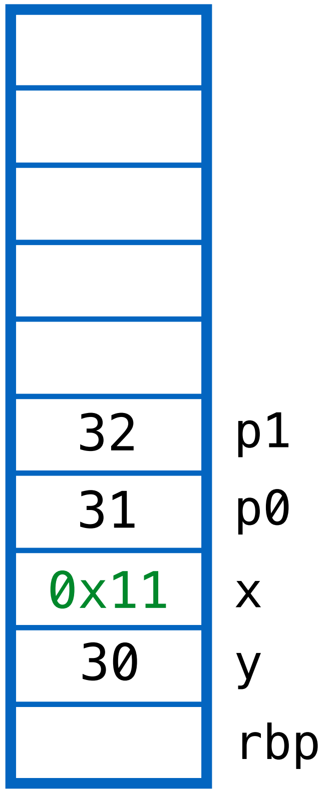


Solution: Compaction

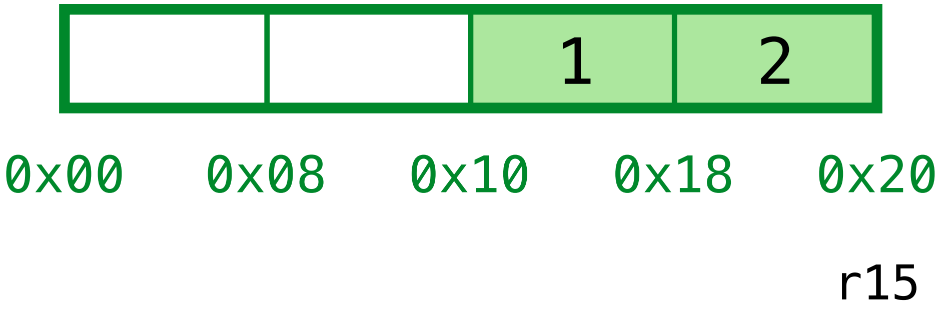
Copy “live” cells into “garbage” …

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



Lets reclaim & recycle garbage!

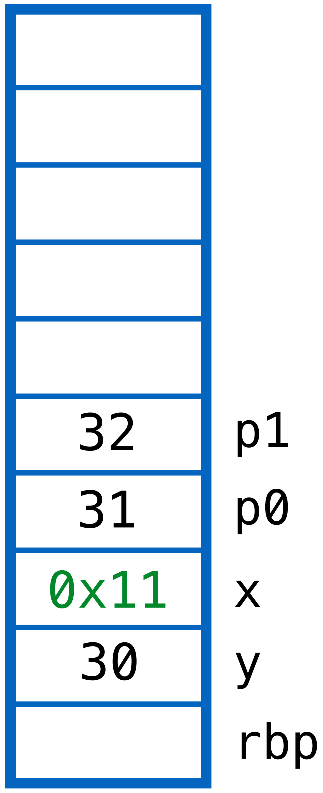


Solution: Compaction

Copy “live” cells into “garbage” …

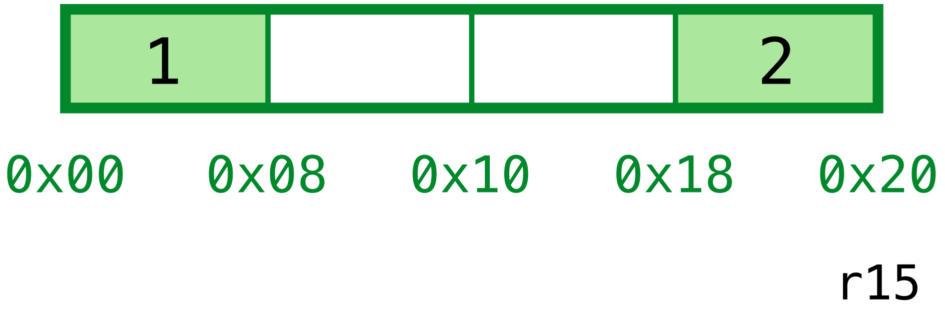
ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



32	p1
31	p0
0x11	x
30	y
	rbp

Lets reclaim & recycle garbage!



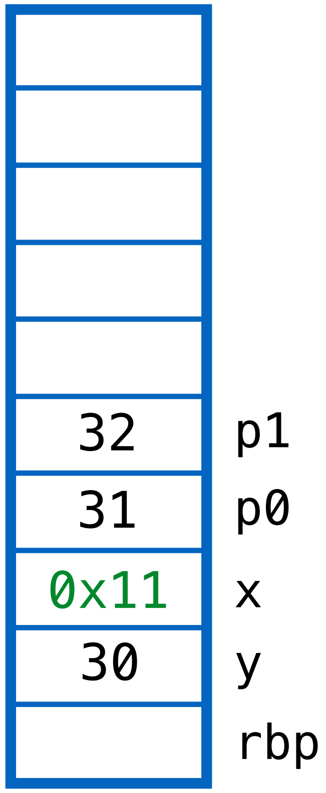
1			2

Solution: Compaction

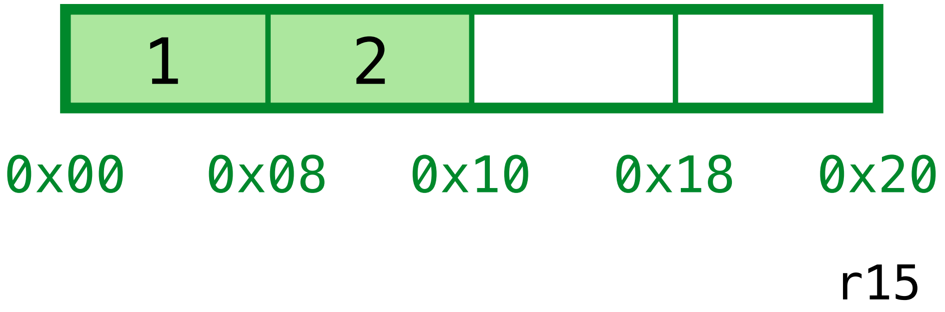
Copy "live" cells into "garbage" ...

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



Lets reclaim & recycle garbage!

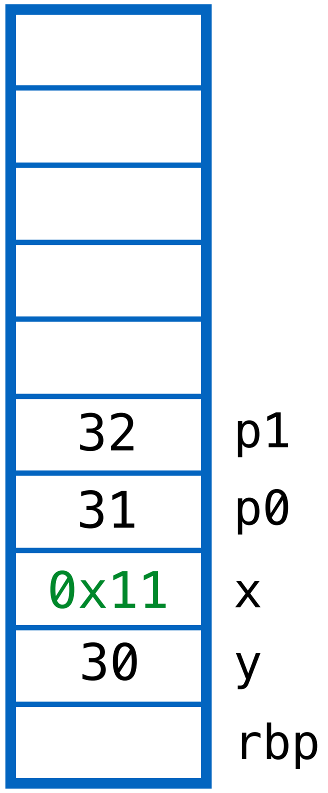


Solution: Compaction

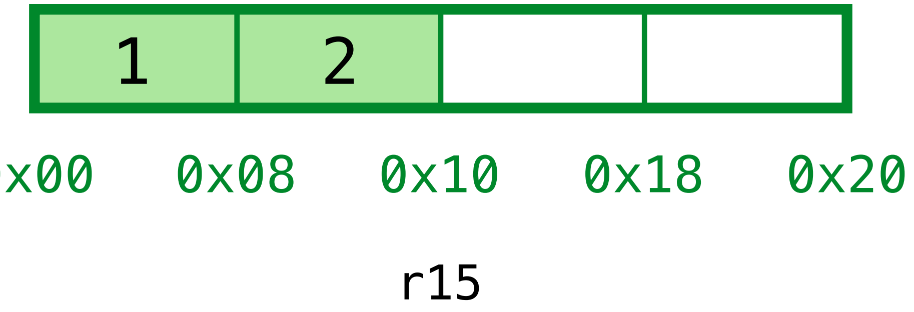
Copy “live” cells into “garbage” ...

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



Lets reclaim & recycle garbage!

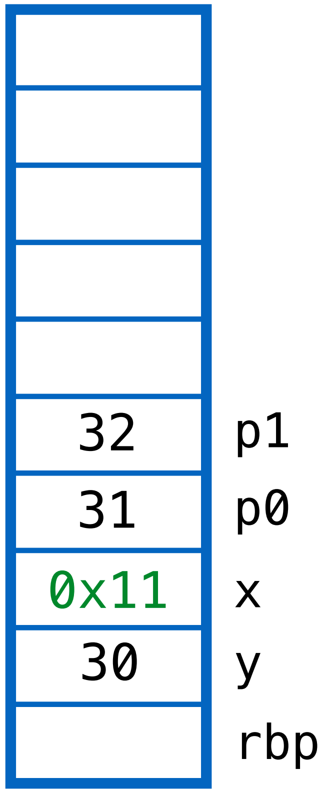


Solution: Compaction

Copy “live” cells into “garbage” … ***and then*** … rewind `r15`!

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



32 p1

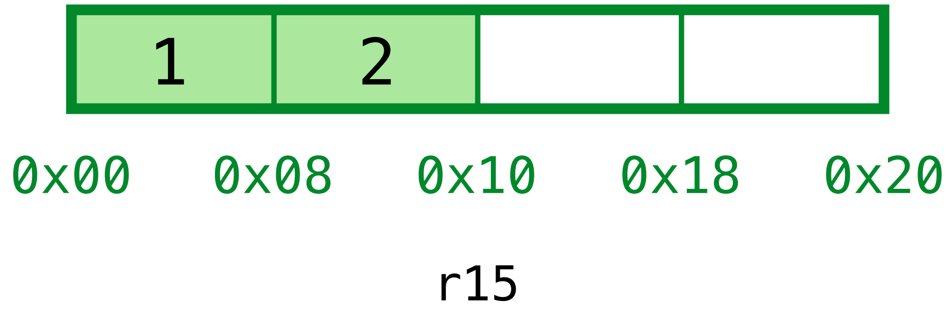
31 p0

0x11 x

30 y

rbp

Yay! Have space for `(p0, p1)`



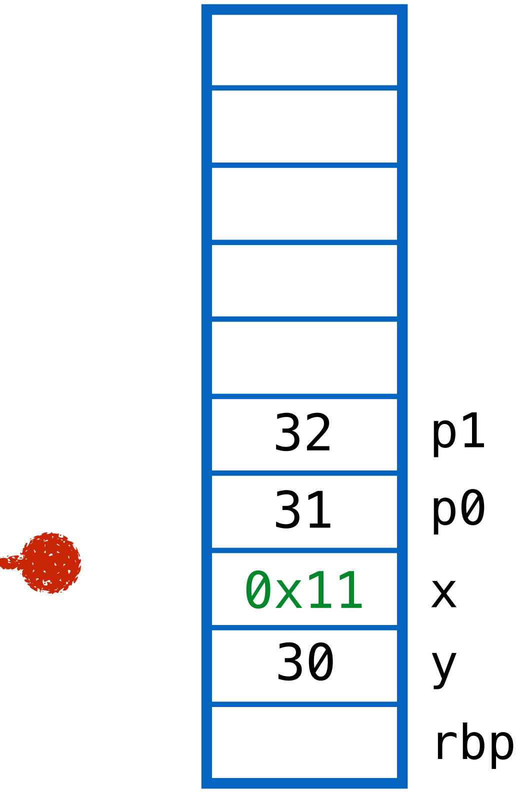
1 2

0x00 0x08 0x10 0x18 0x20

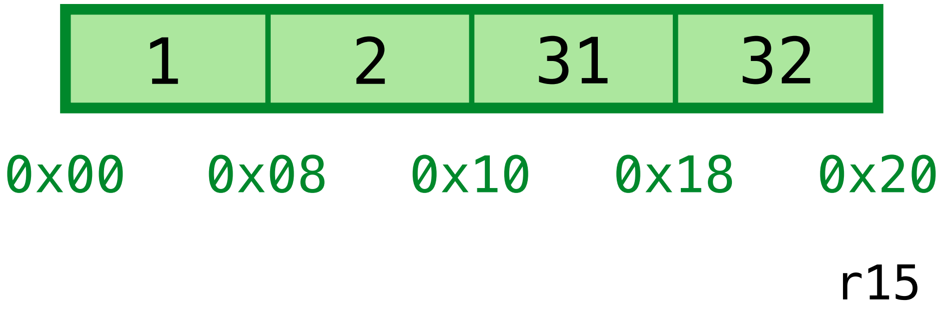
r15

ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```



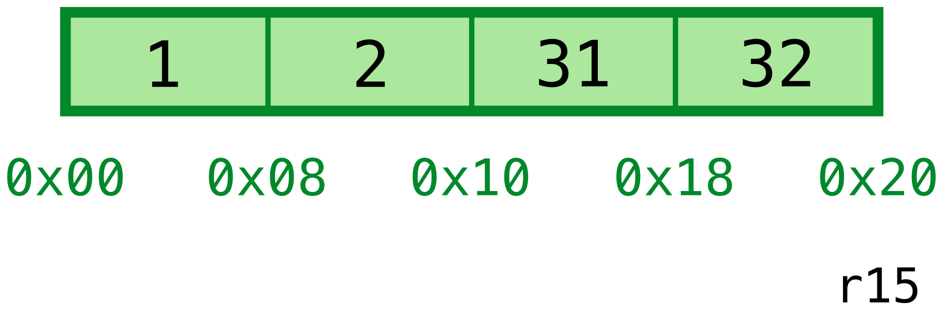
Yay! Have space for `(p0, p1)`



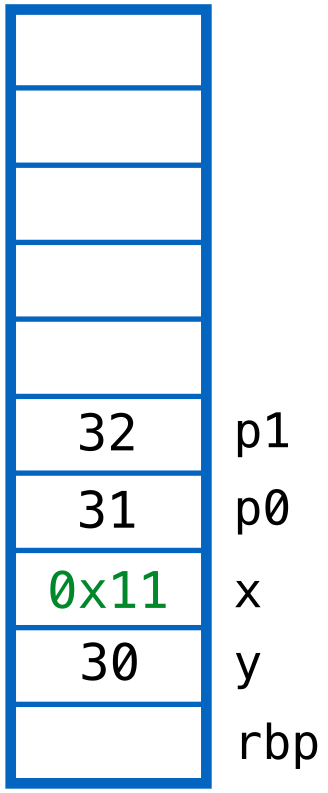
ex2: garbage in the middle

```
let y  = let tmp = (10, 20)
         in tmp[0] + tmp[1]
  , x  = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in
  (p0, p1)
```

Result (rax) = 0x09



1	2	31	32



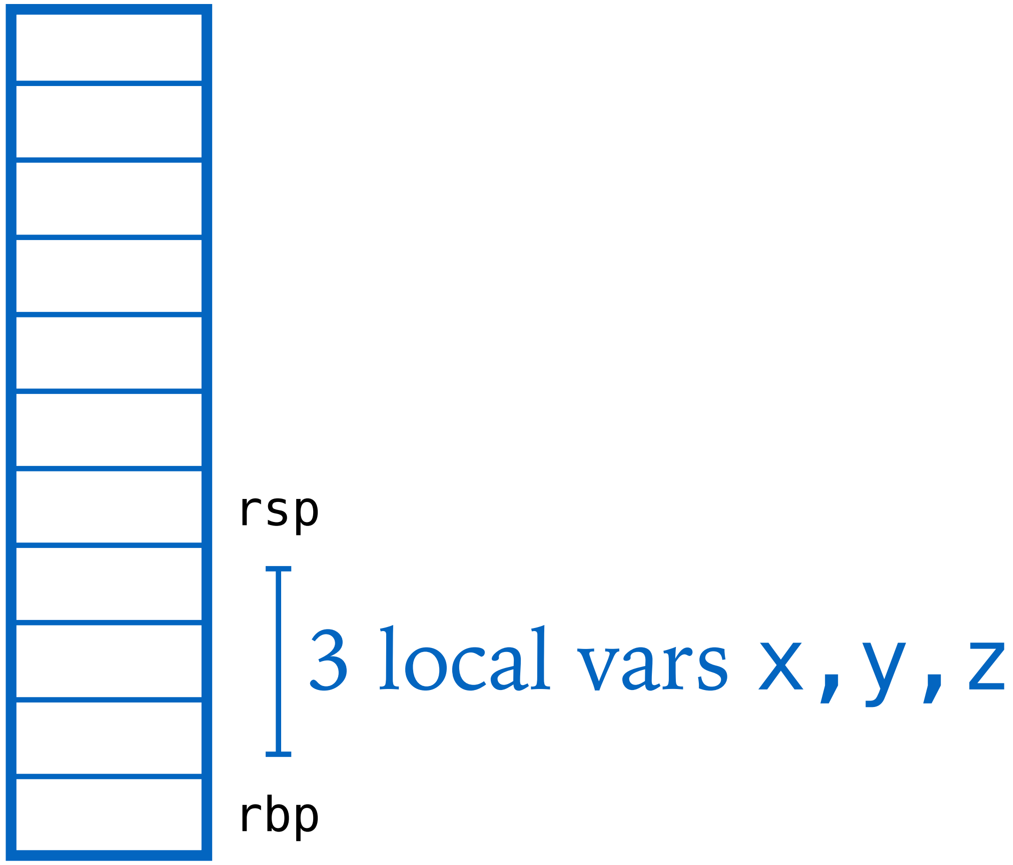
32	p1
31	p0
0x11	x
30	y
	rbp

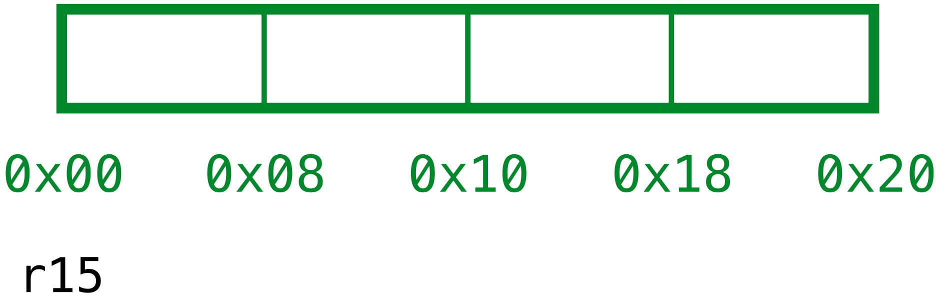
Garter / GC

Example 3

ex3: garbage in the middle (with stack)

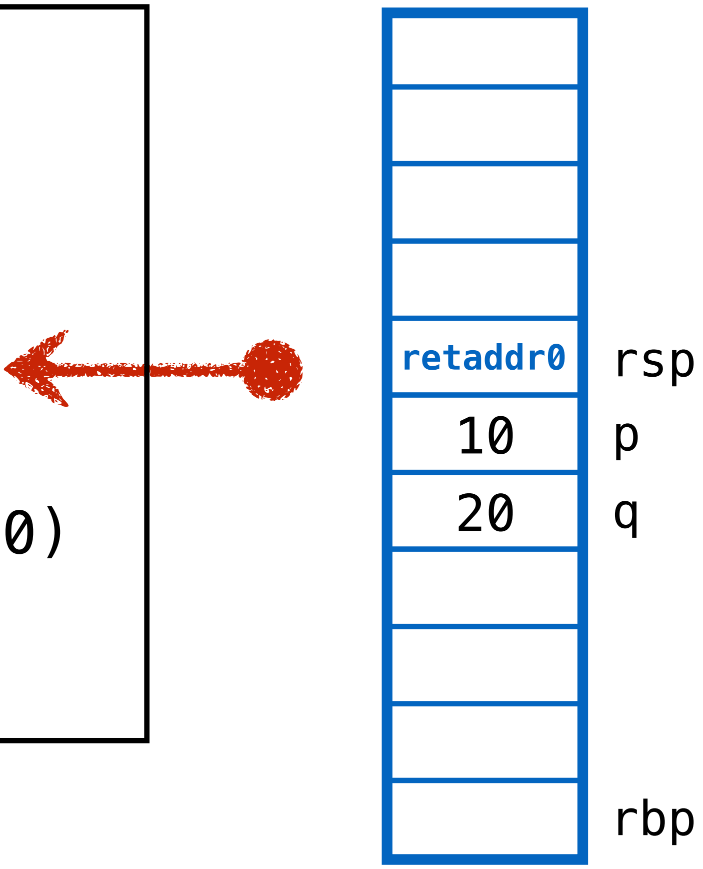
```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + y + z
```

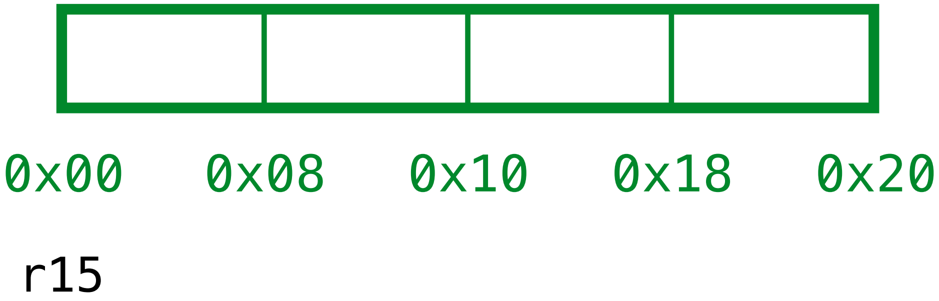




ex3: garbage in the middle (with stack)

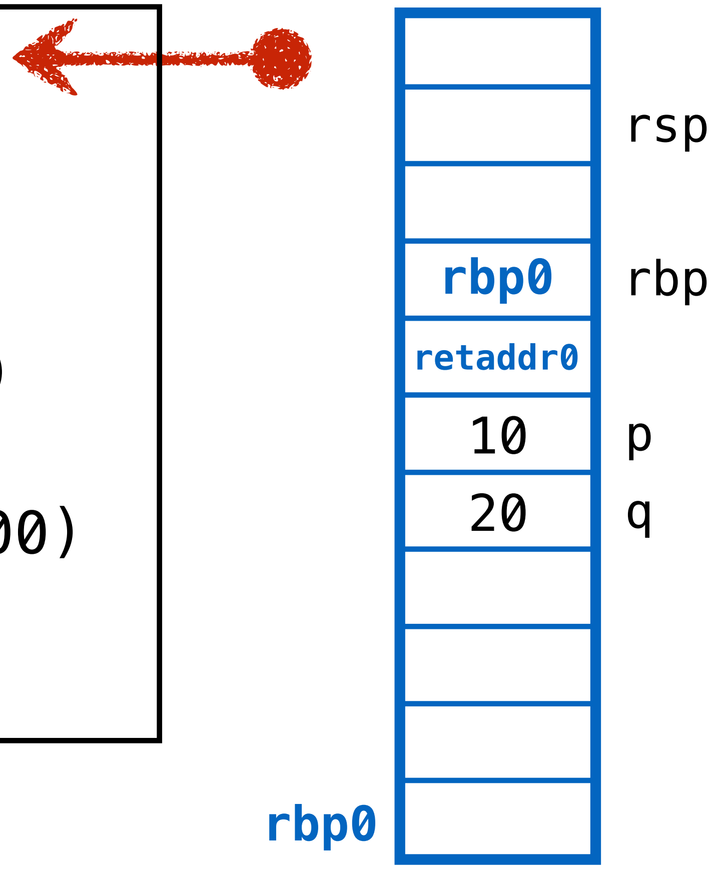
```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + y + z
```

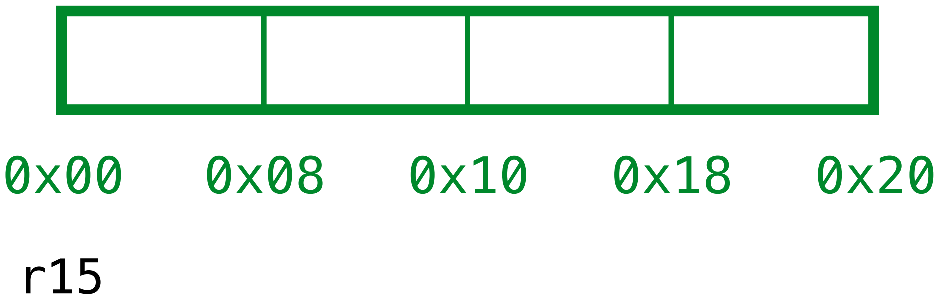




ex3: garbage in the middle (with stack)

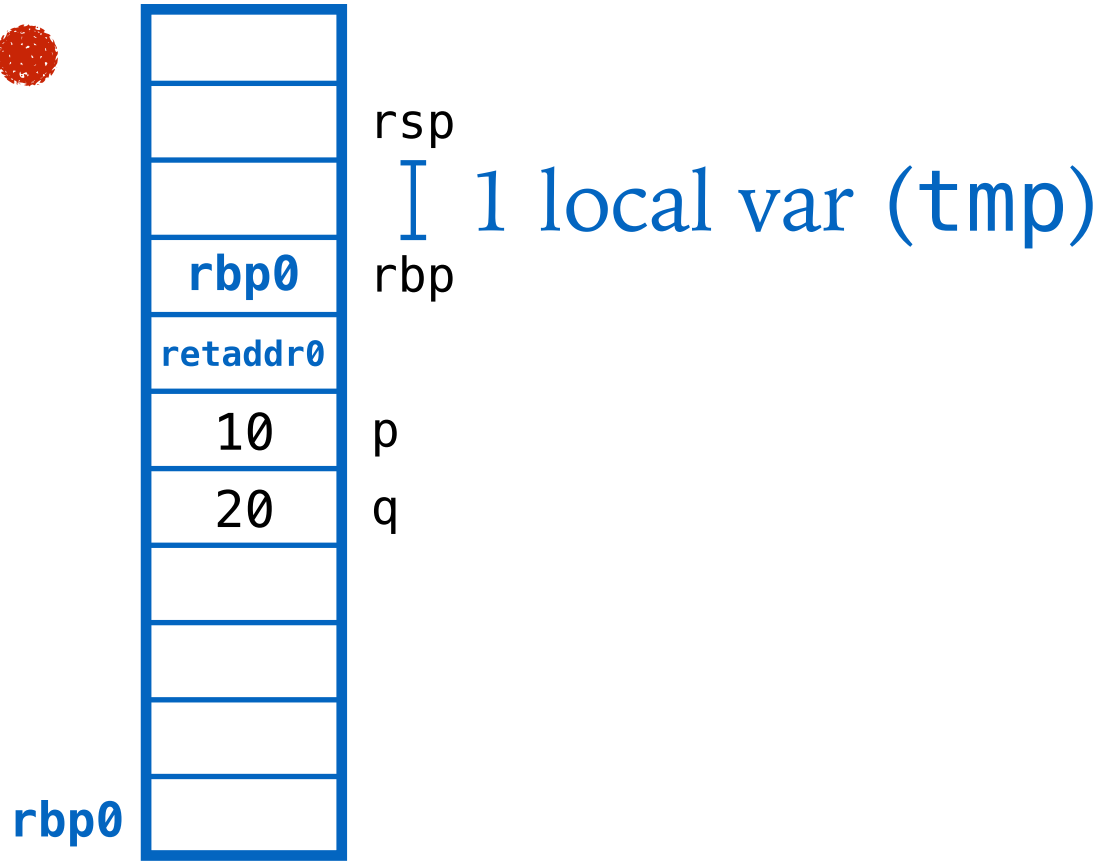
```
def foo(p, q):
   let tmp = (p, q)
   in tmp[0] + tmp[1]

let y  = foo(10, 20)
   , x  = (y, y + 1)
   , z  = foo(100, 200)
in
   x[0] + y + z
```

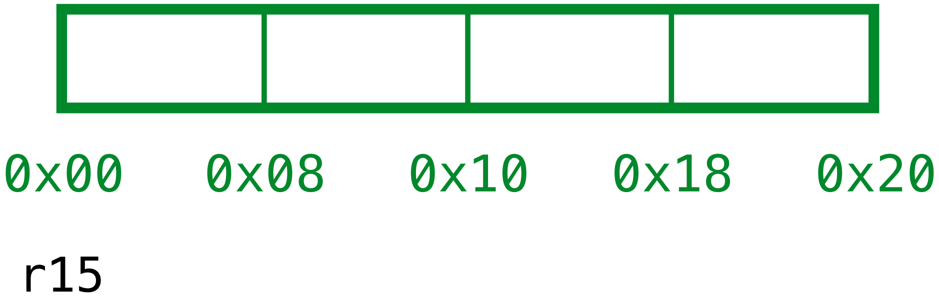




ex3: garbage in the middle (with stack)

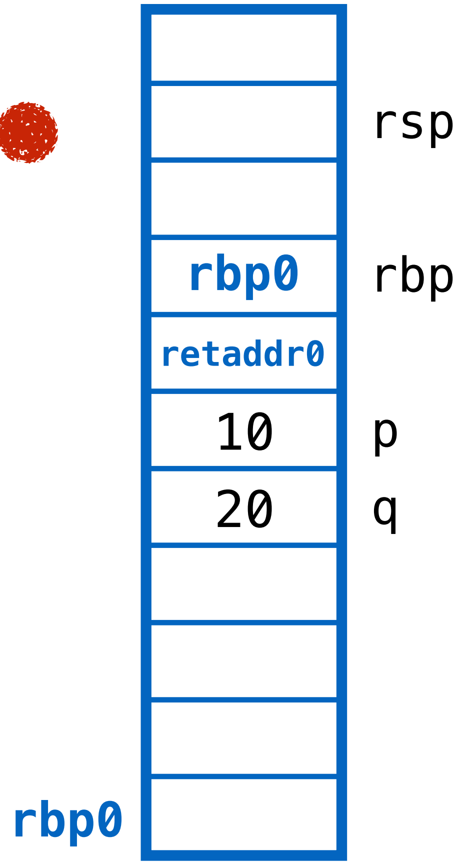
```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + y + z
```





ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + y + z
```



rsp

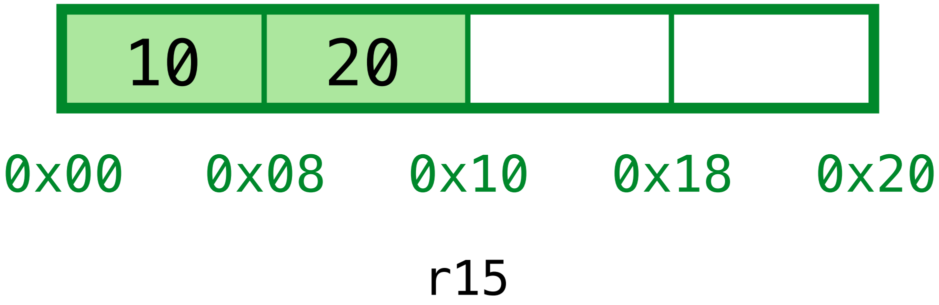
rbp0 rbp

retaddr0

10 p

20 q

rbp0



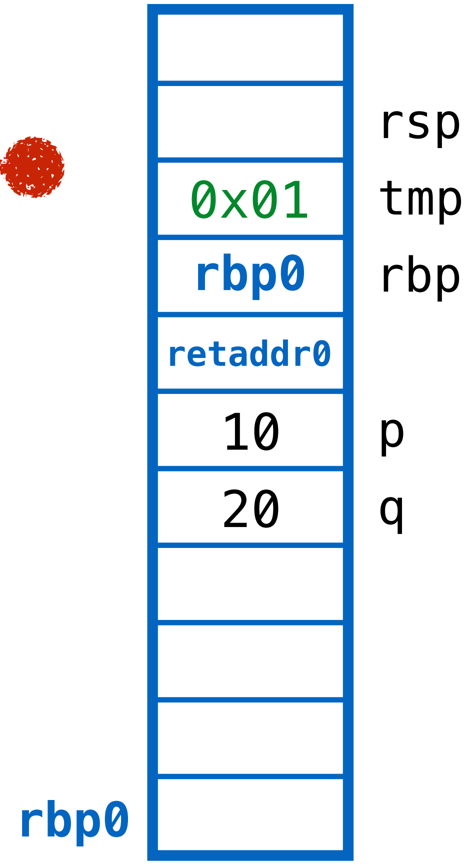
10	20		

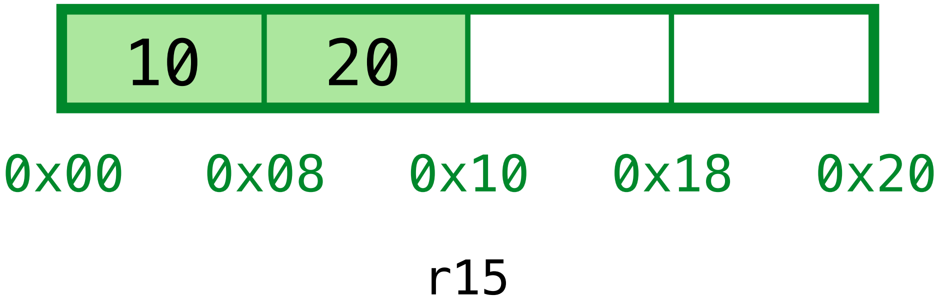
0x00 0x08 0x10 0x18 0x20

r15

ex3: garbage in the middle (with stack)

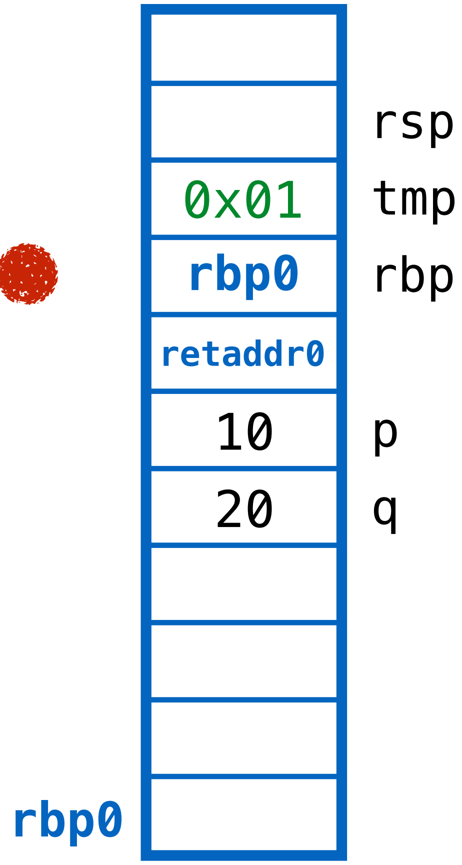
```
def foo(p, q):
   let tmp = (p, q)
   in tmp[0] + tmp[1]

let y  = foo(10, 20)
   , x  = (y, y + 1)
   , z  = foo(100, 200)
in
   x[0] + y + z
```





ex3: garbage in the middle (with stack)

```
def foo(p, q):
   let tmp = (p, q)
   in tmp[0] + tmp[1]

let y  = foo(10, 20)
   , x  = (y, y + 1)
   , z  = foo(100, 200)
in
   x[0] + y + z
```



rsp

0x01 tmp

rbp0 rbp

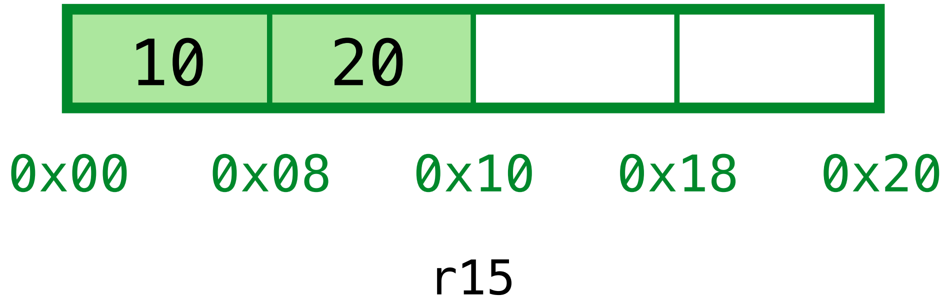
retaddr0

10 p

20 q

rbp0

Return (rax) = 30



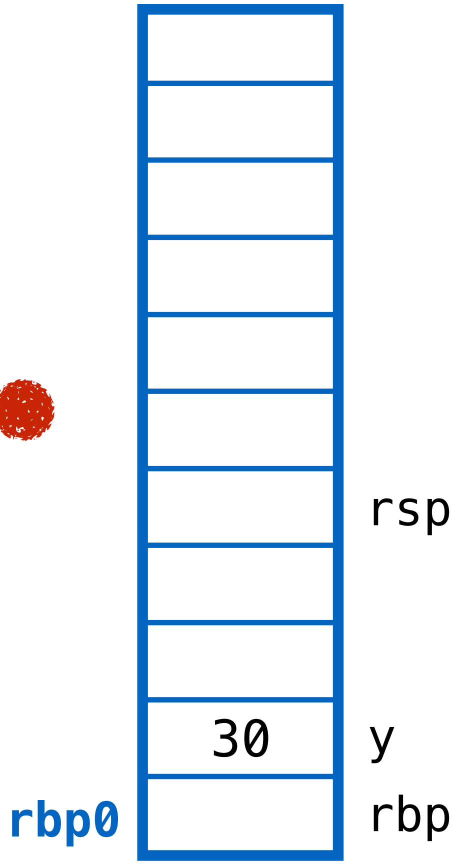
10 | 20

0x00 0x08 0x10 0x18 0x20

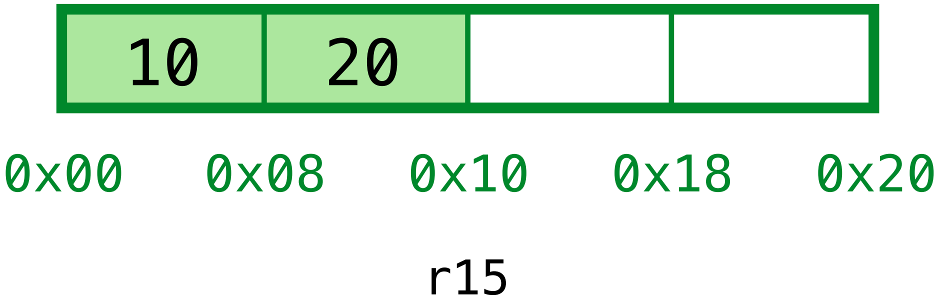
r15

ex3: garbage in the middle (with stack)

```
def foo(p, q):
   let tmp = (p, q)
   in tmp[0] + tmp[1]

let y   = foo(10, 20)
   , x   = (y, y + 1)
   , z   = foo(100, 200)
in
   x[0] + z
```

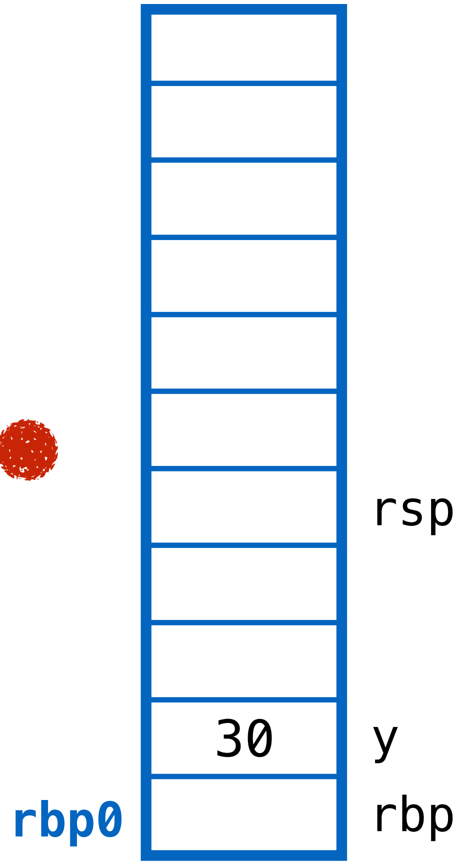


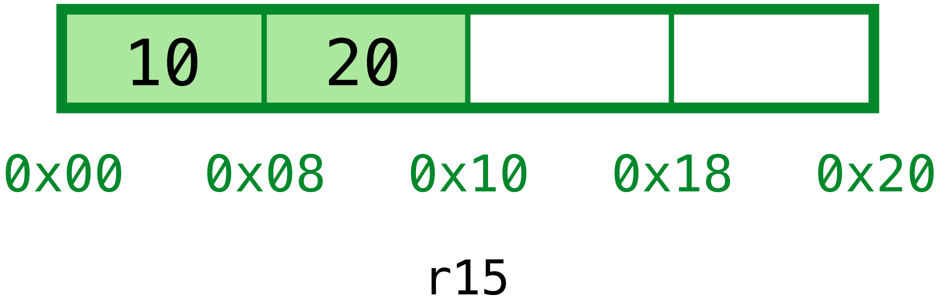
Return (rax) = 30



ex3: garbage in the middle (with stack)

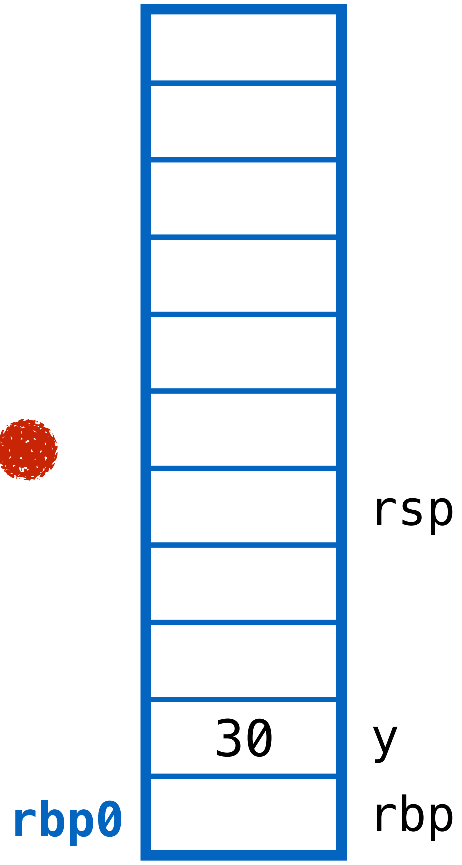
```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

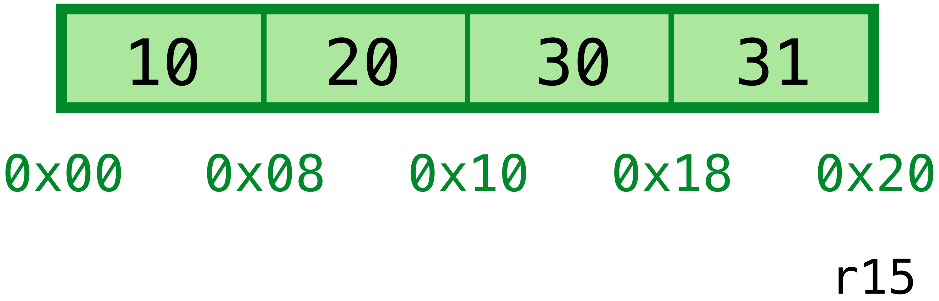




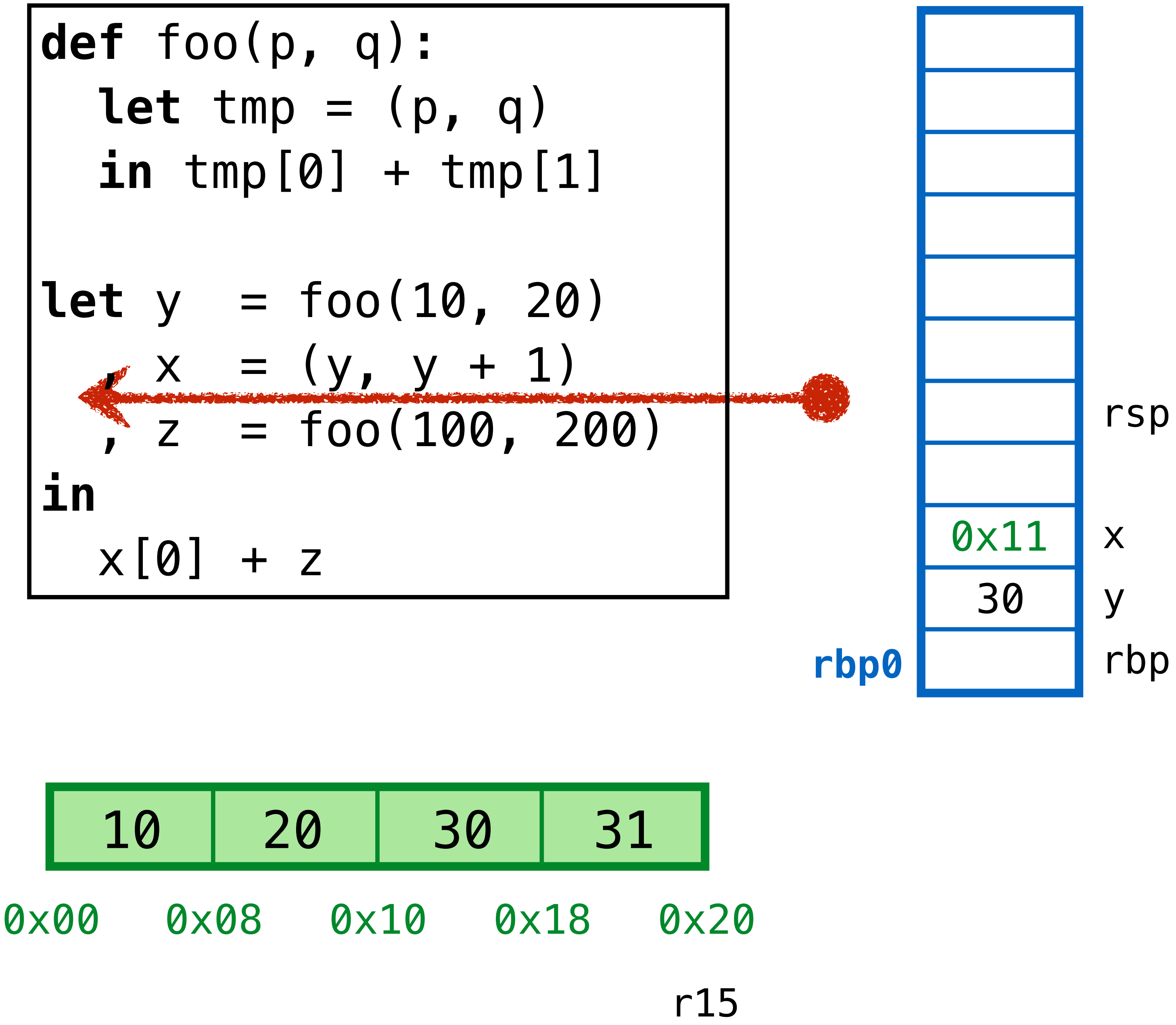
ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```





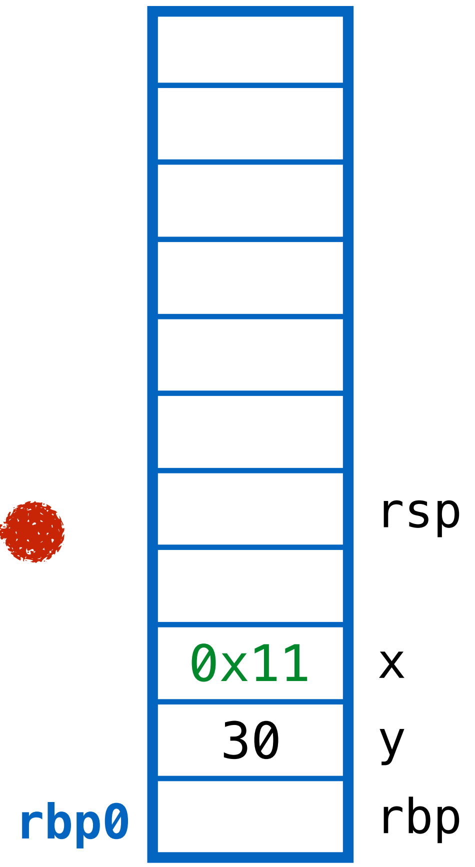
ex3: garbage in the middle (with stack)

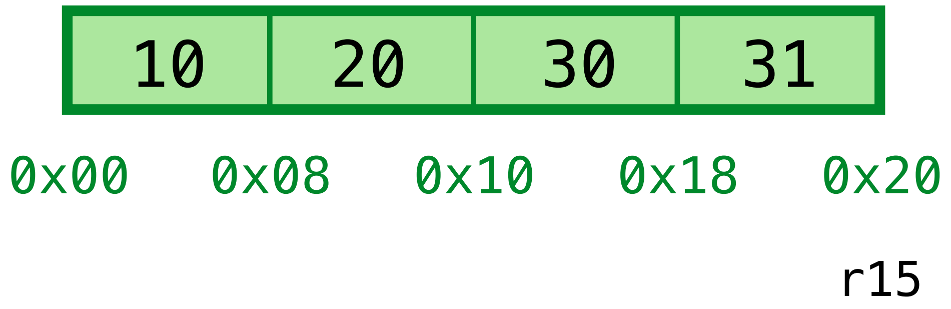


```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

ex3: garbage in the middle (with stack)

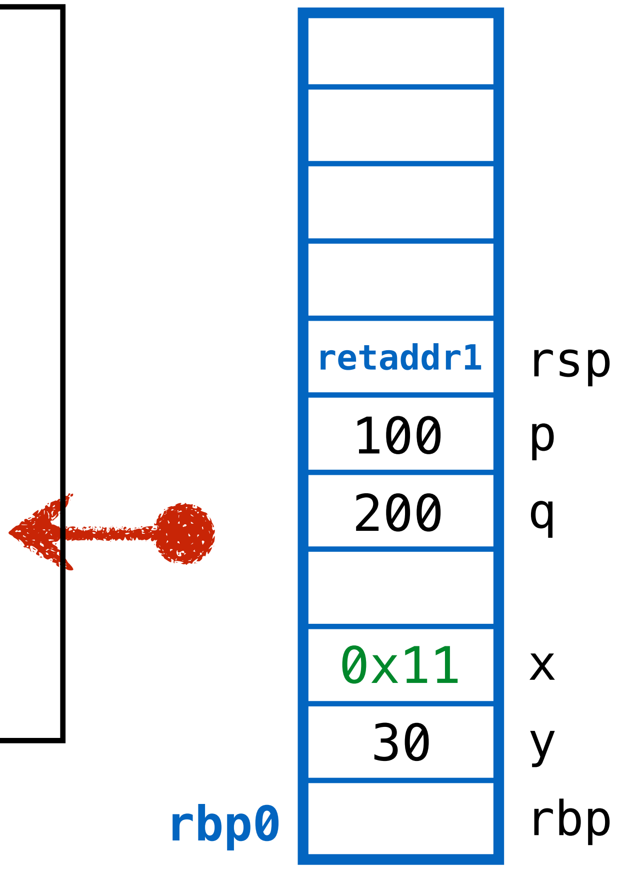
```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

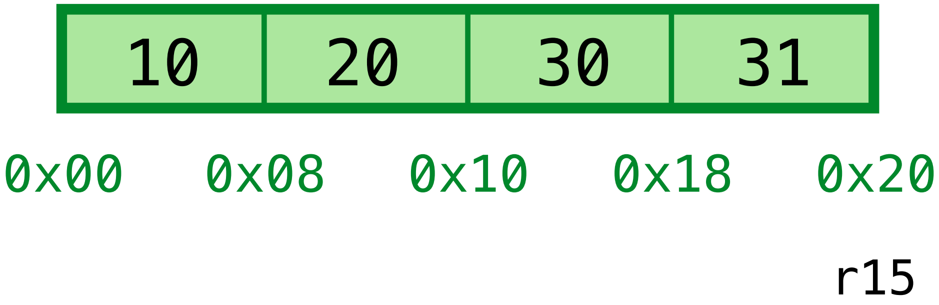




ex3: garbage in the middle (with stack)

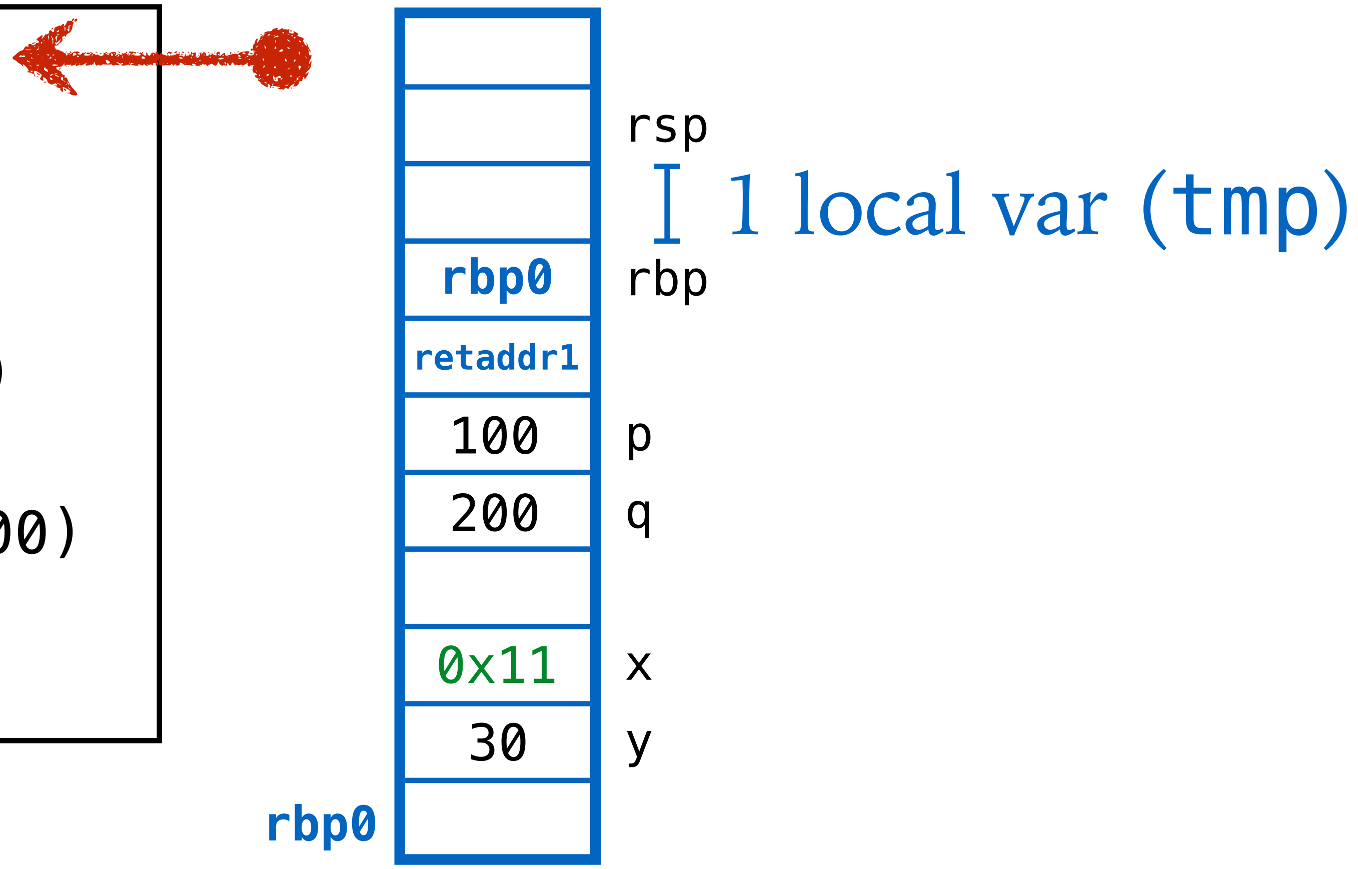
```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```





ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



rsp

[1 local var (tmp)

rbp0 rbp

retaddr1

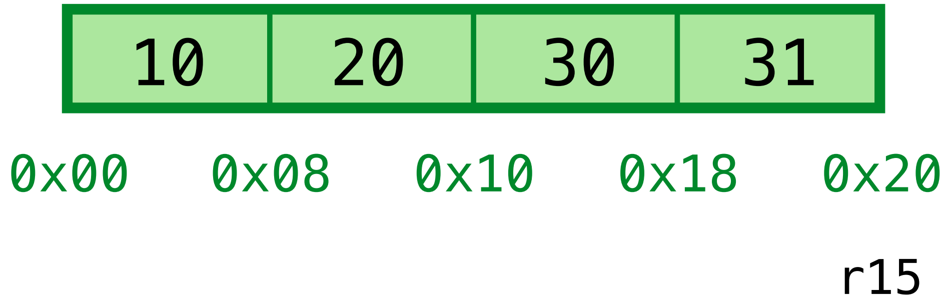
100 p

200 q

0x11 x

30 y

rbp0



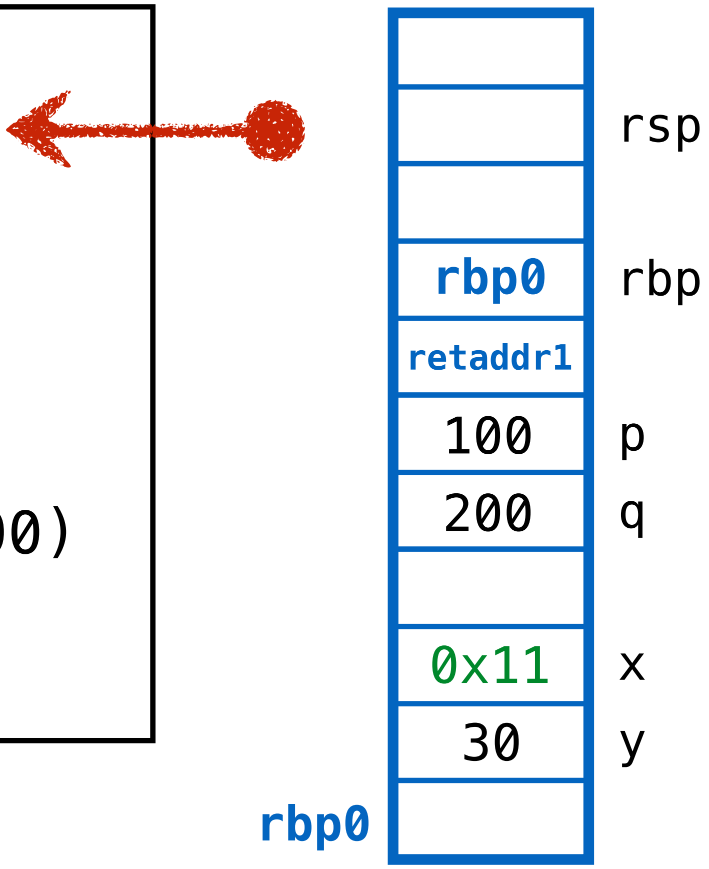
10 20 30 31

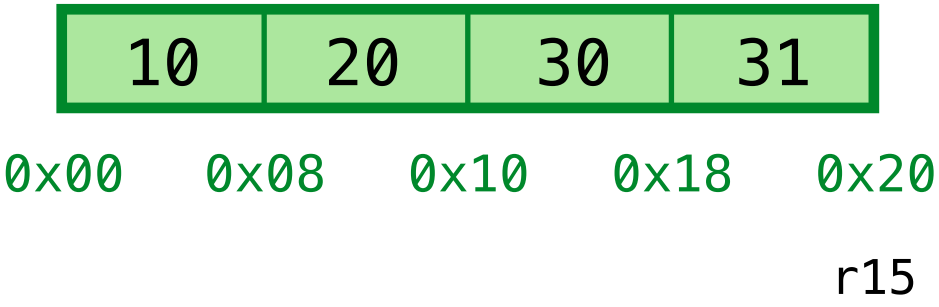
0x00 0x08 0x10 0x18 0x20

r15

ex3: garbage in the middle (with stack)

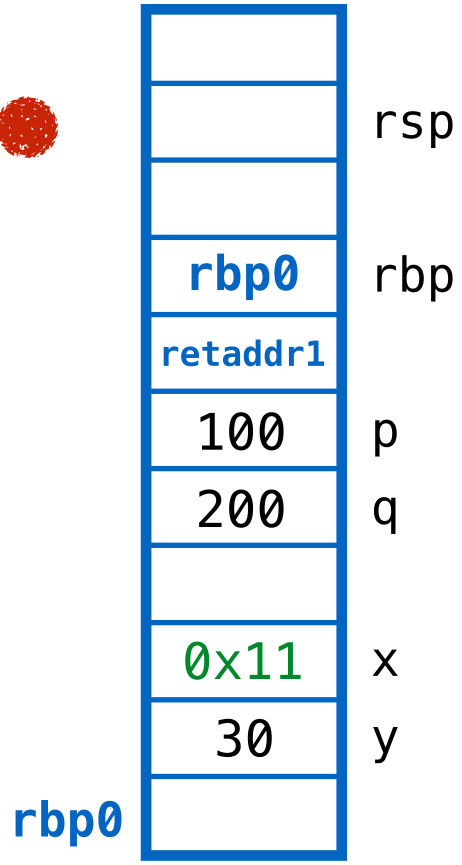
```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



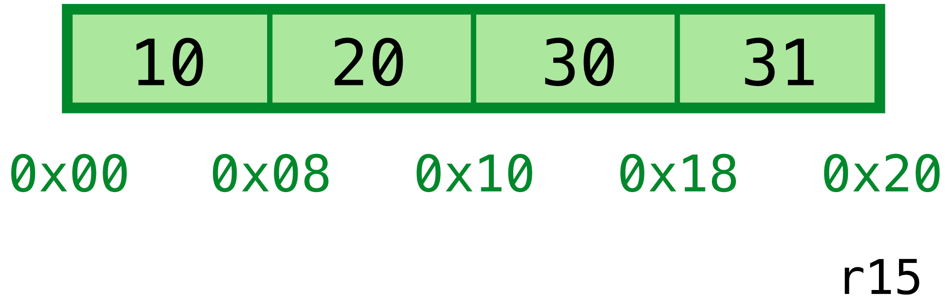


ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

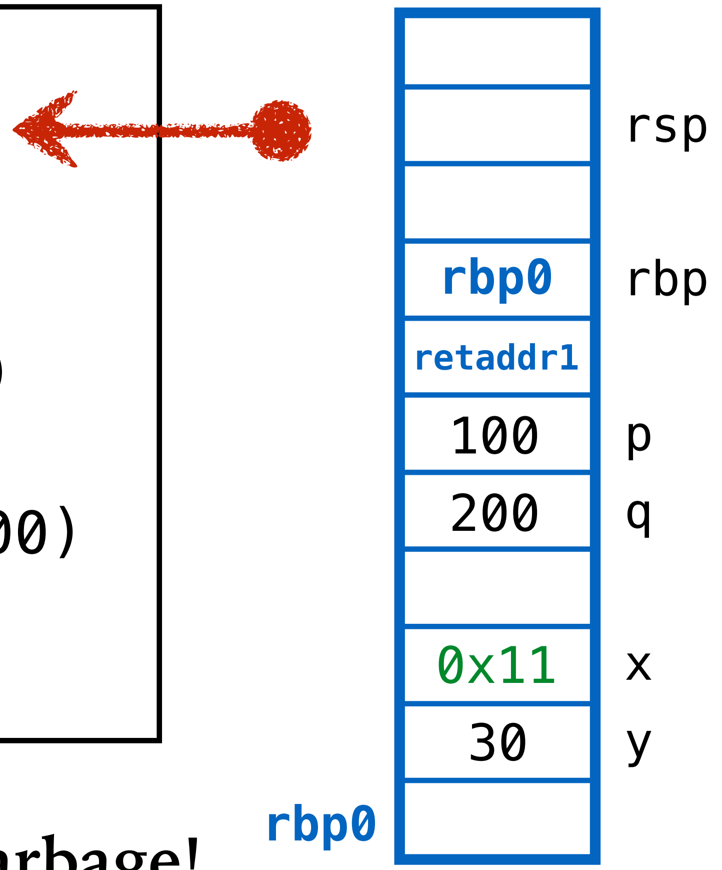


Lets reclaim & recycle garbage!

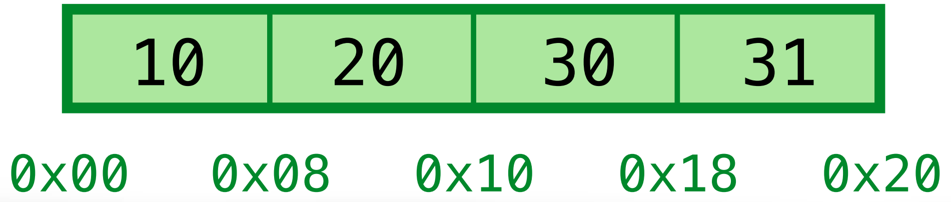


ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



Lets reclaim & recycle garbage!

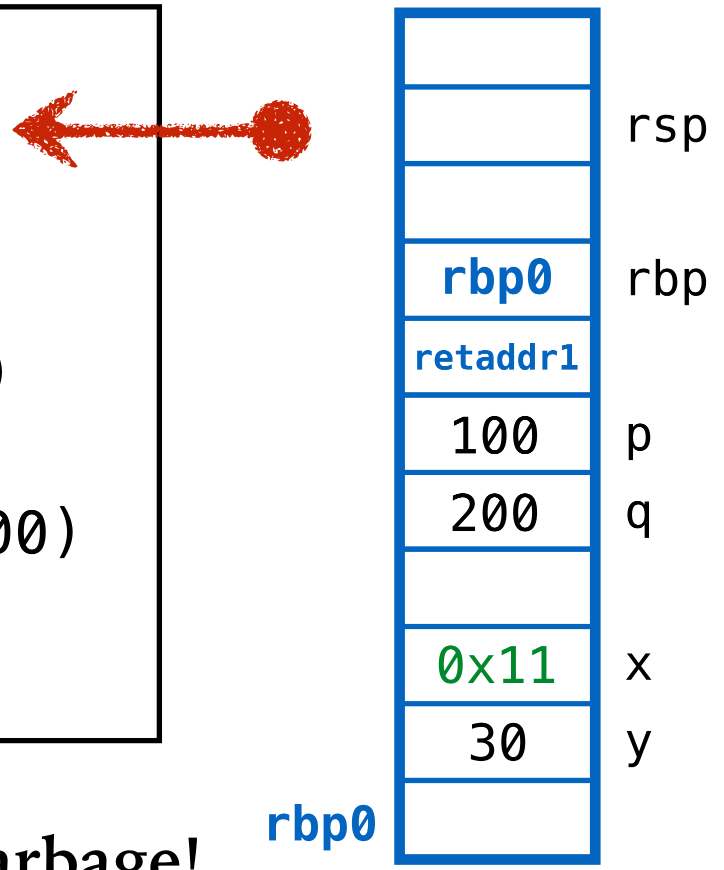


QUIZ: Which cells are garbage?

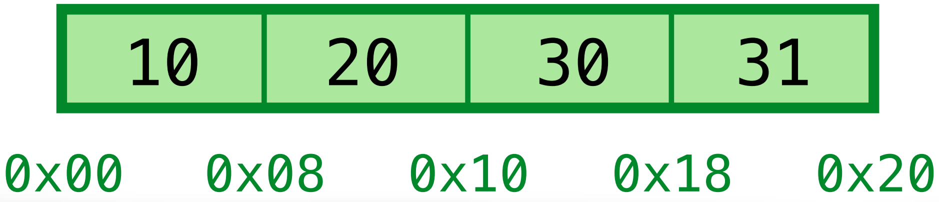
(A) 0x00, 0x08 (B) 0x08, 0x10 (C) 0x10, 0x18 (D) None (E) All

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



Lets reclaim & recycle garbage!

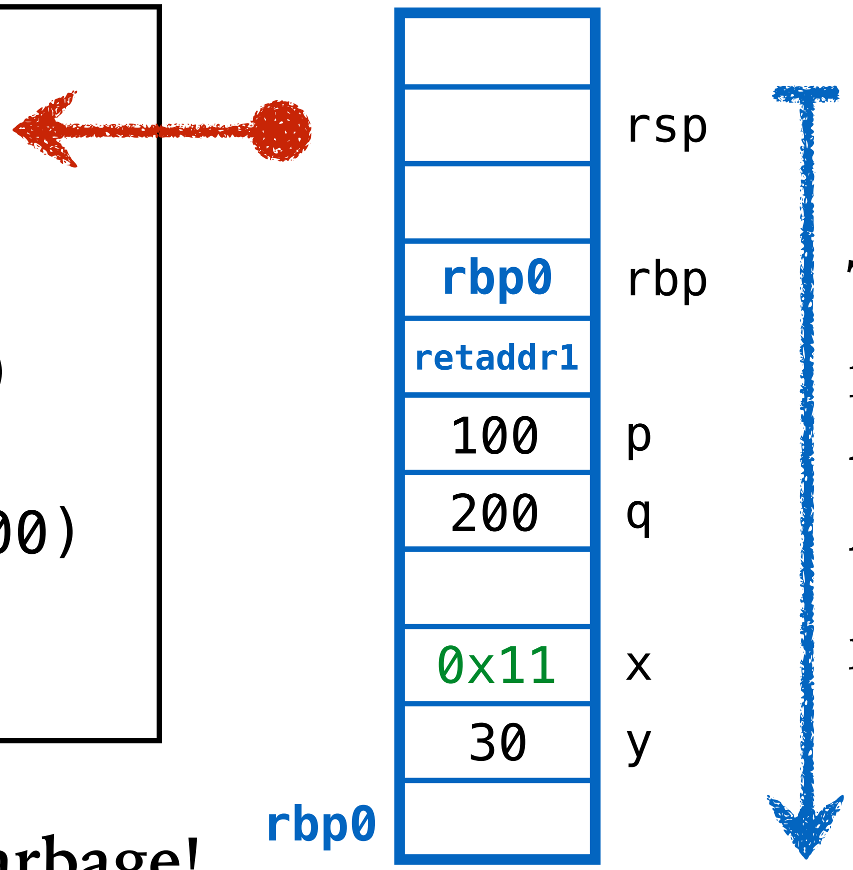


QUIZ: Which cells are garbage?

Those that are *not reachable from any stack frame*

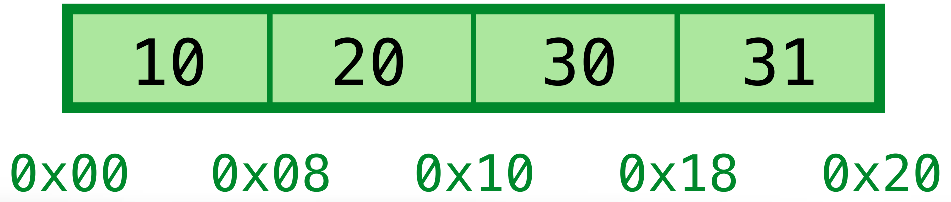
ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



Traverse Stack
from top (`rsp`)
to bottom (`rbp0`)
to mark
reachable cells.

Lets reclaim & recycle garbage!

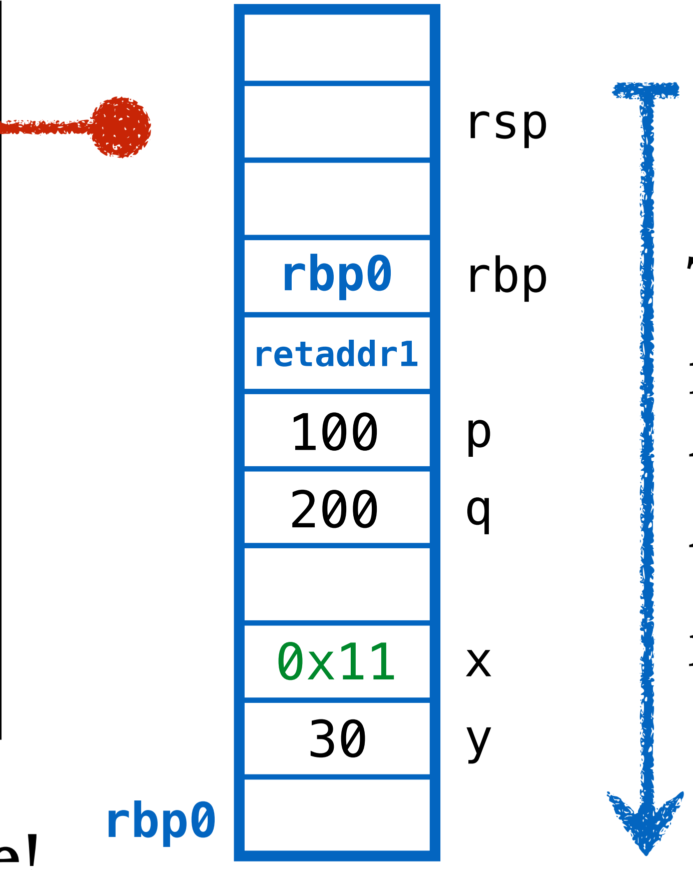


QUIZ: Which cells are garbage?

Those that are *not reachable from any stack frame*

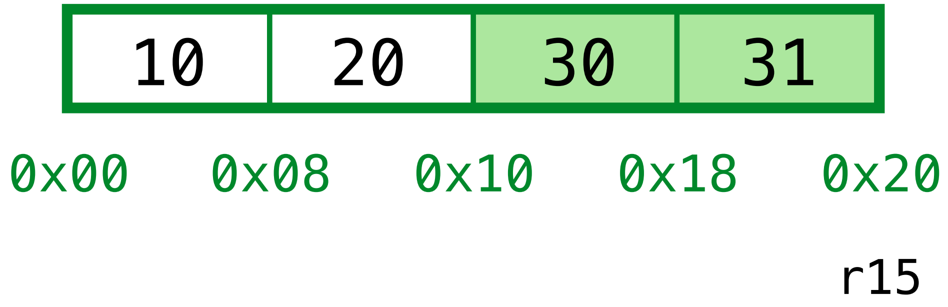
ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



Traverse Stack
from top (`rsp`)
to bottom (`rbp0`)
to mark
reachable cells.

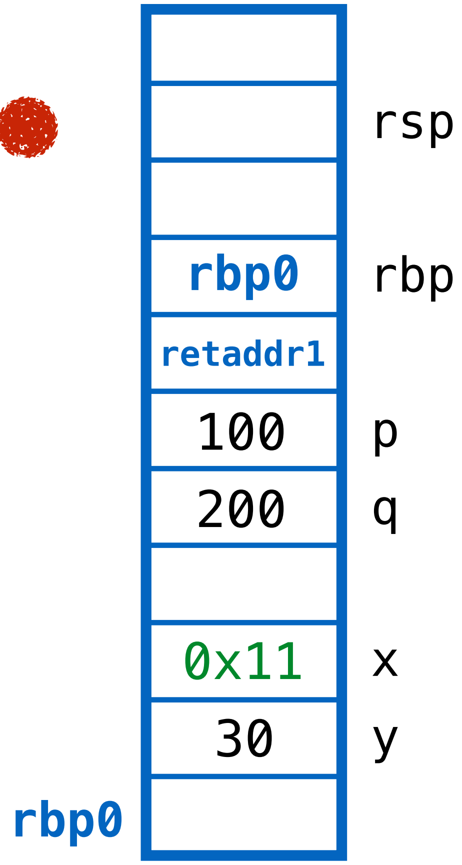
Lets reclaim & recycle garbage!

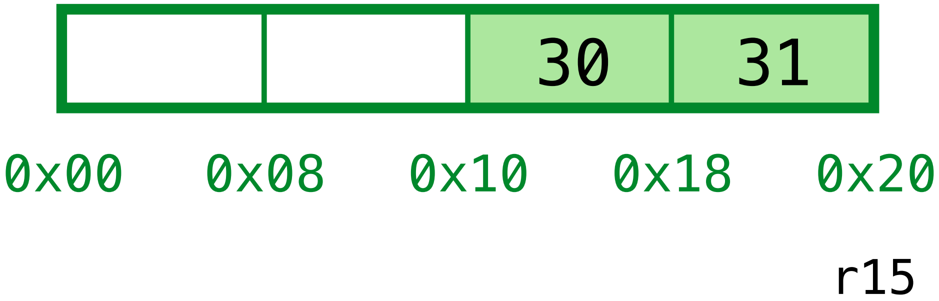


Which cells are garbage?

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

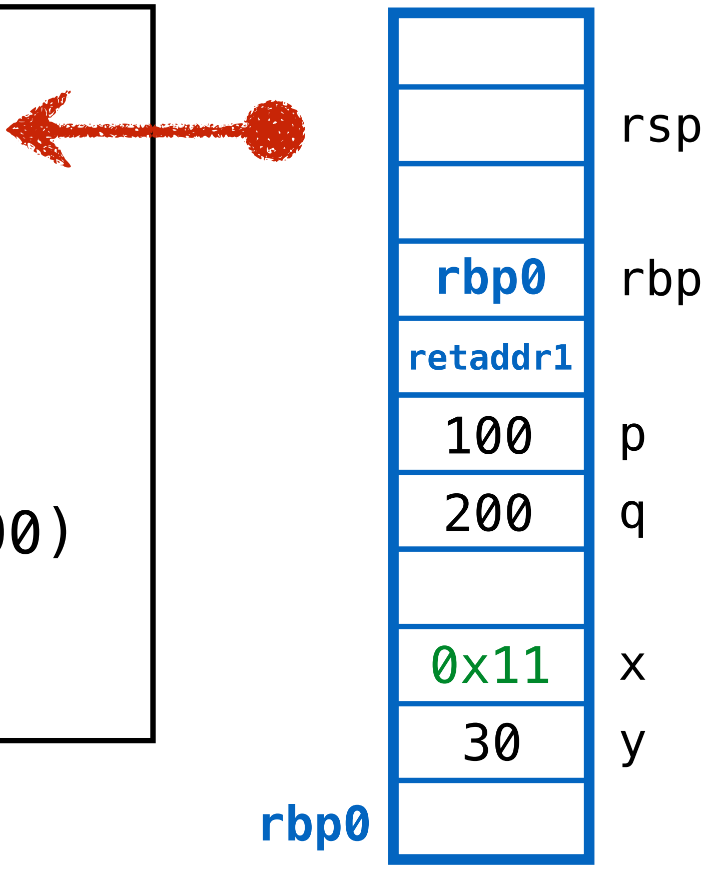


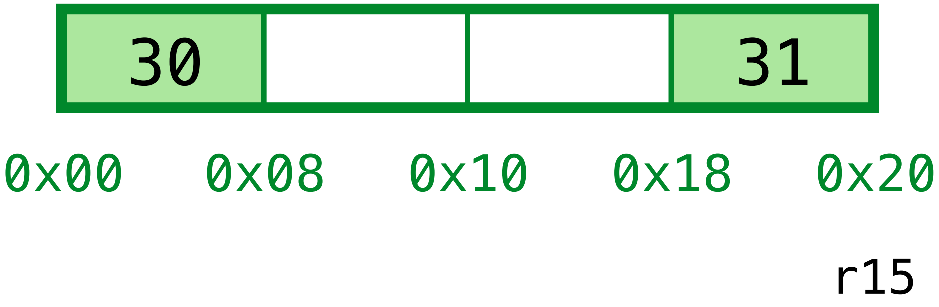


Compact the live cells

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

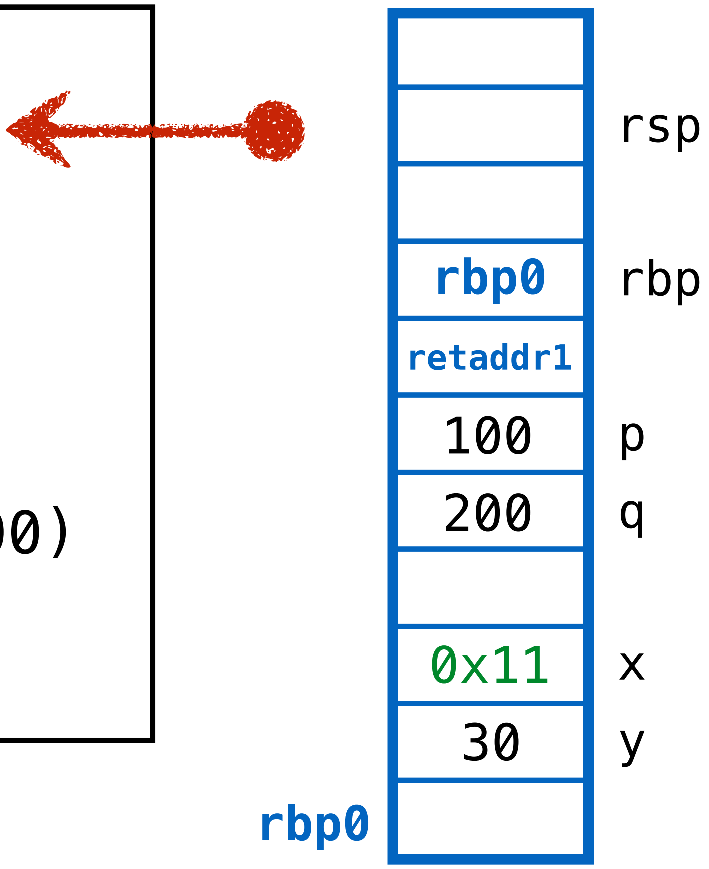




Compact the live cells

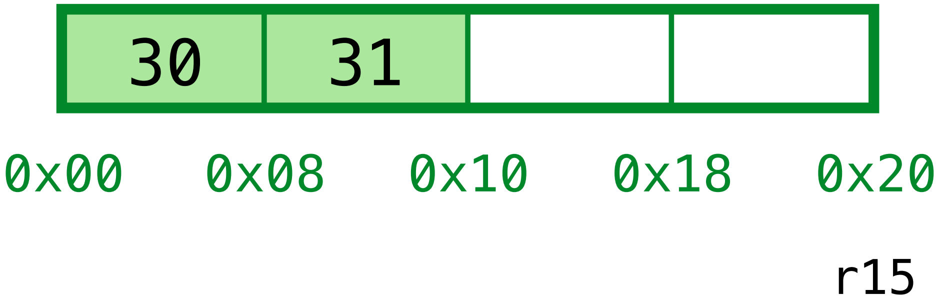
ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



	rsp
rbp0	rbp
retaddr1	
100	p
200	q
0x11	x
30	y

rbp0



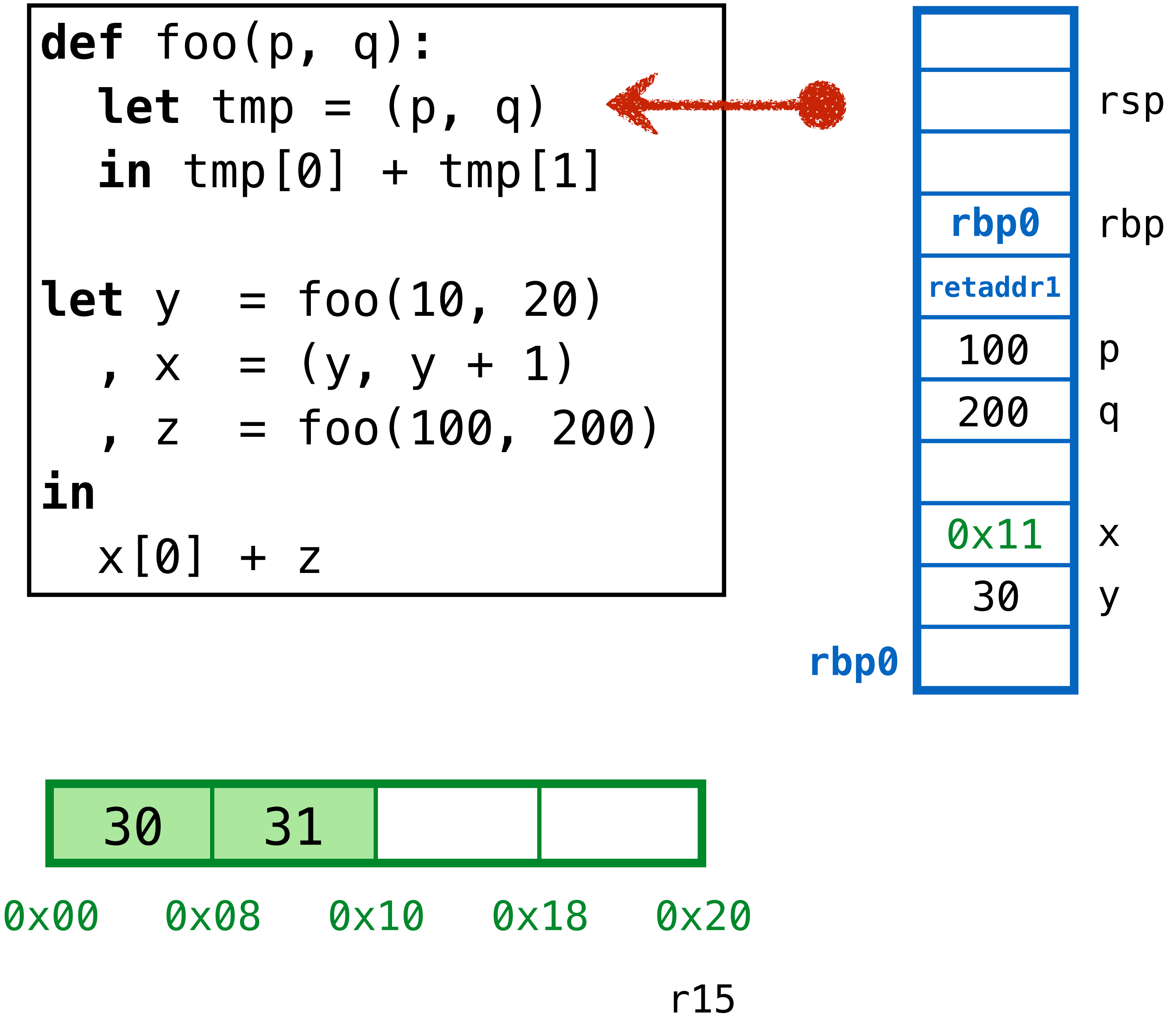
30	31		

0x00 0x08 0x10 0x18 0x20

r15

Compact the live cells

ex3: garbage in the middle (with stack)

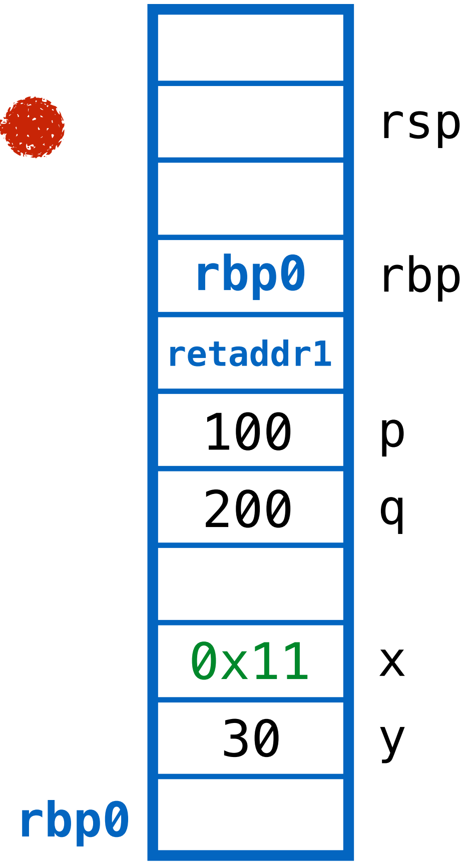


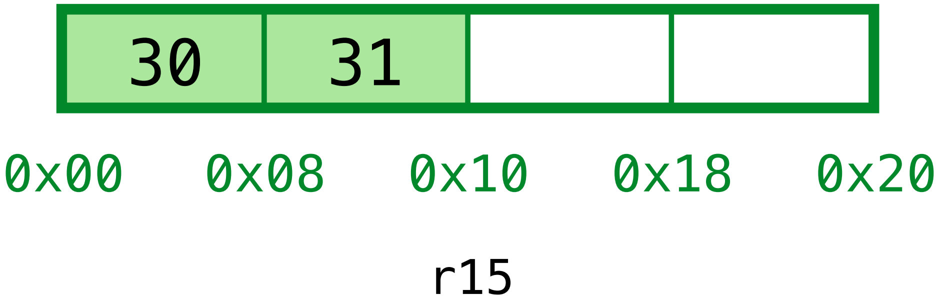
```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

Compact the live cells … then rewind `r15`

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

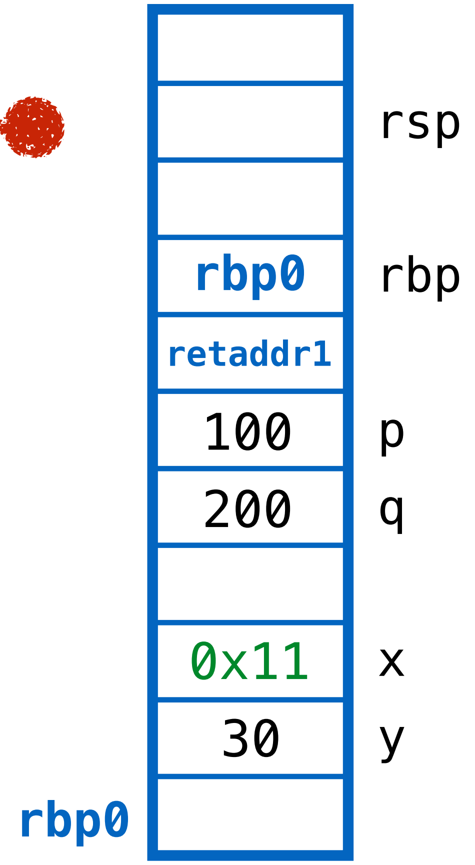


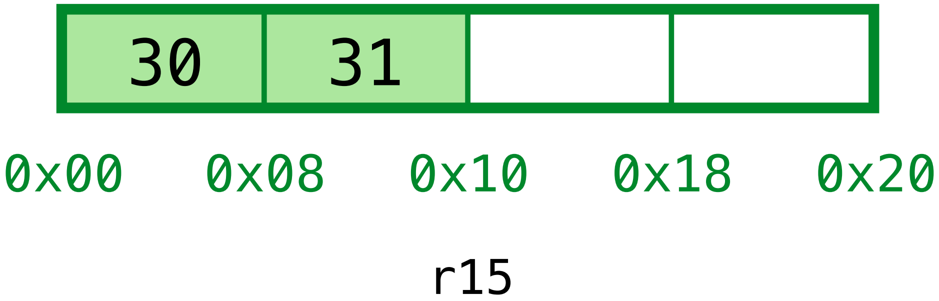


Compact the live cells … then rewind r15

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

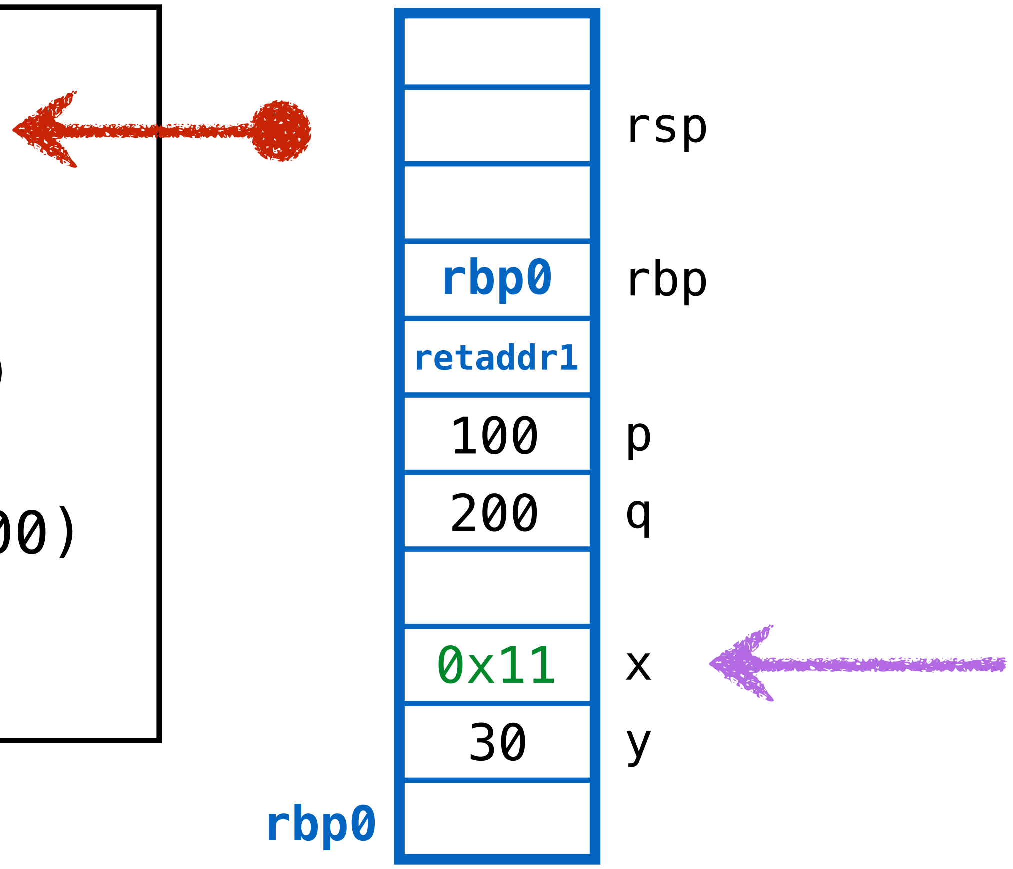


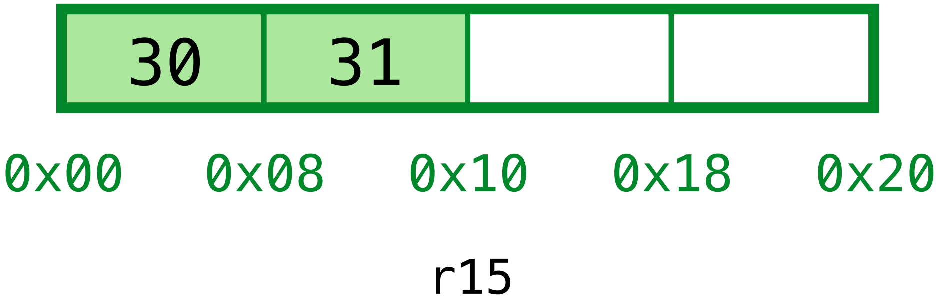


Problem???

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

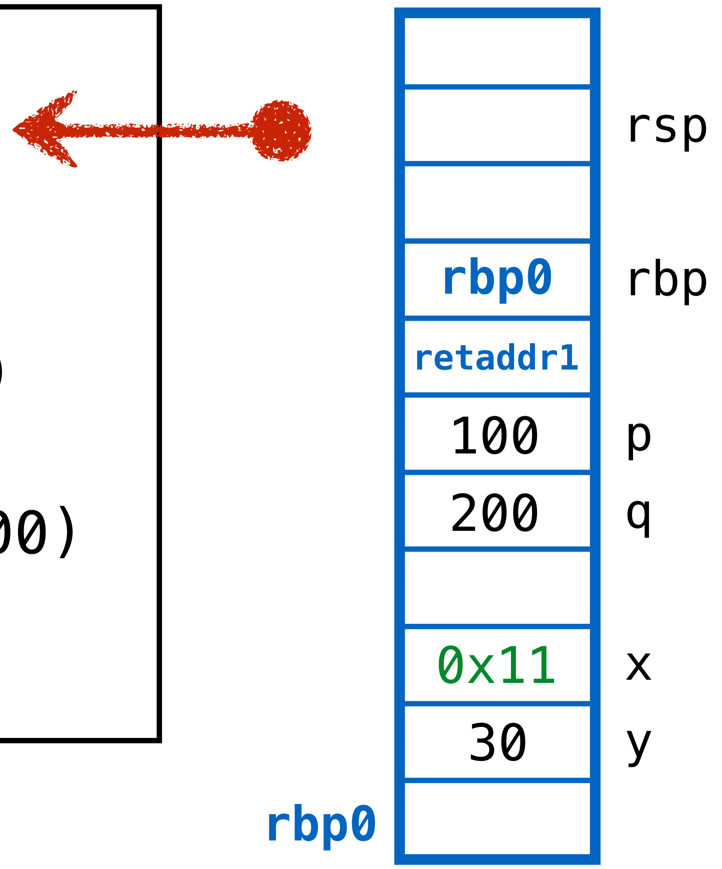


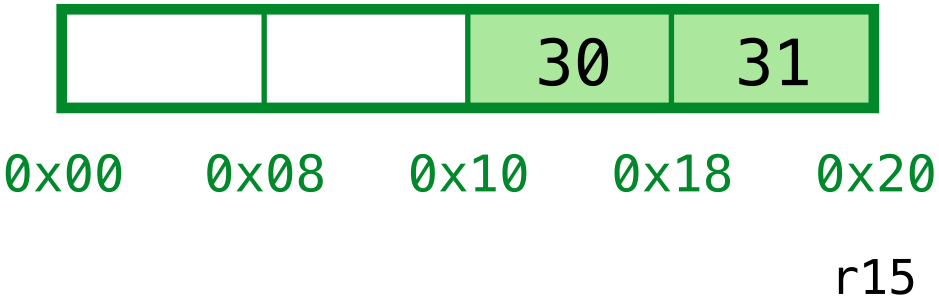


Problem! Have to REDIRECT existing pointers

ex3: garbage in the middle (with stack)

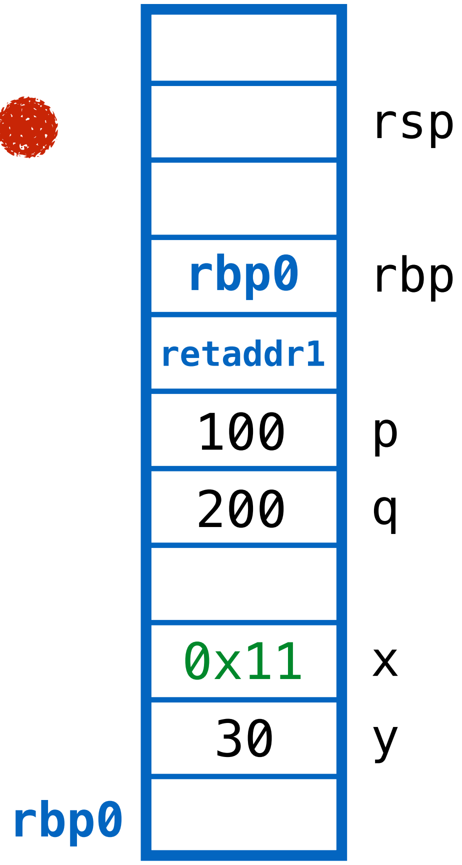
```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

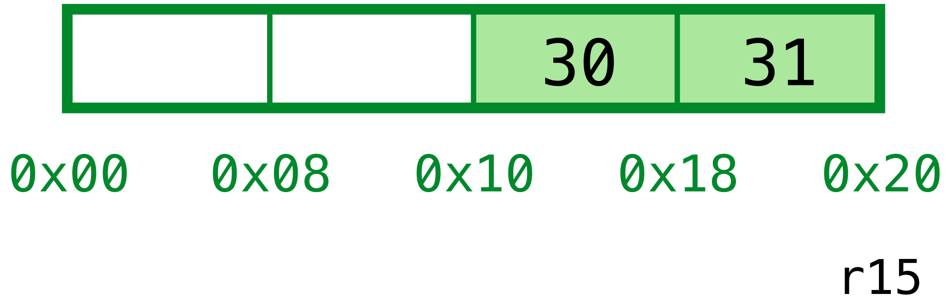




ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

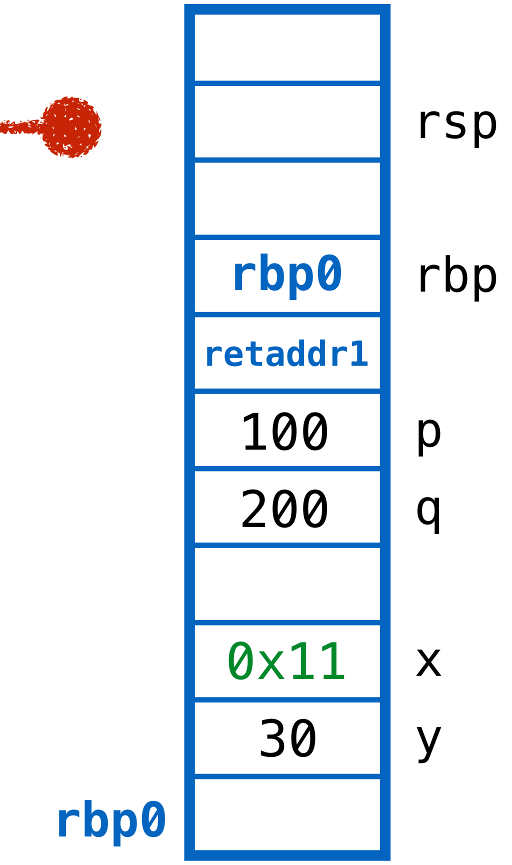


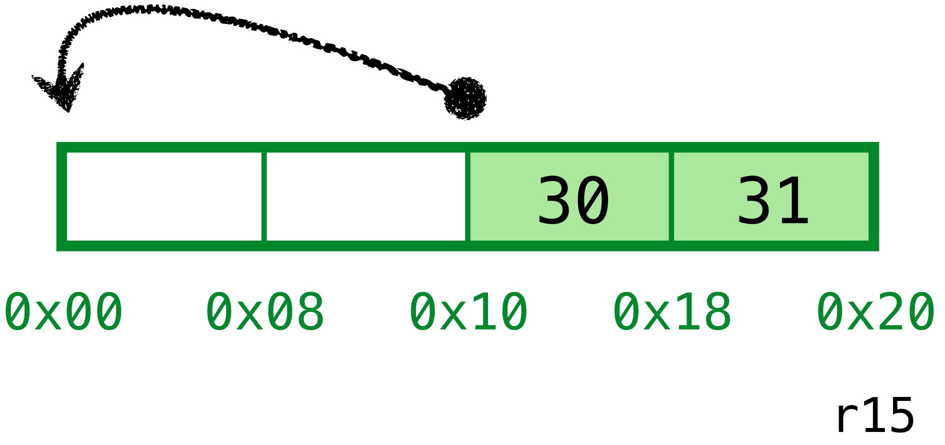


1. Compute FORWARD addrs (i.e. new compacted addrs)

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



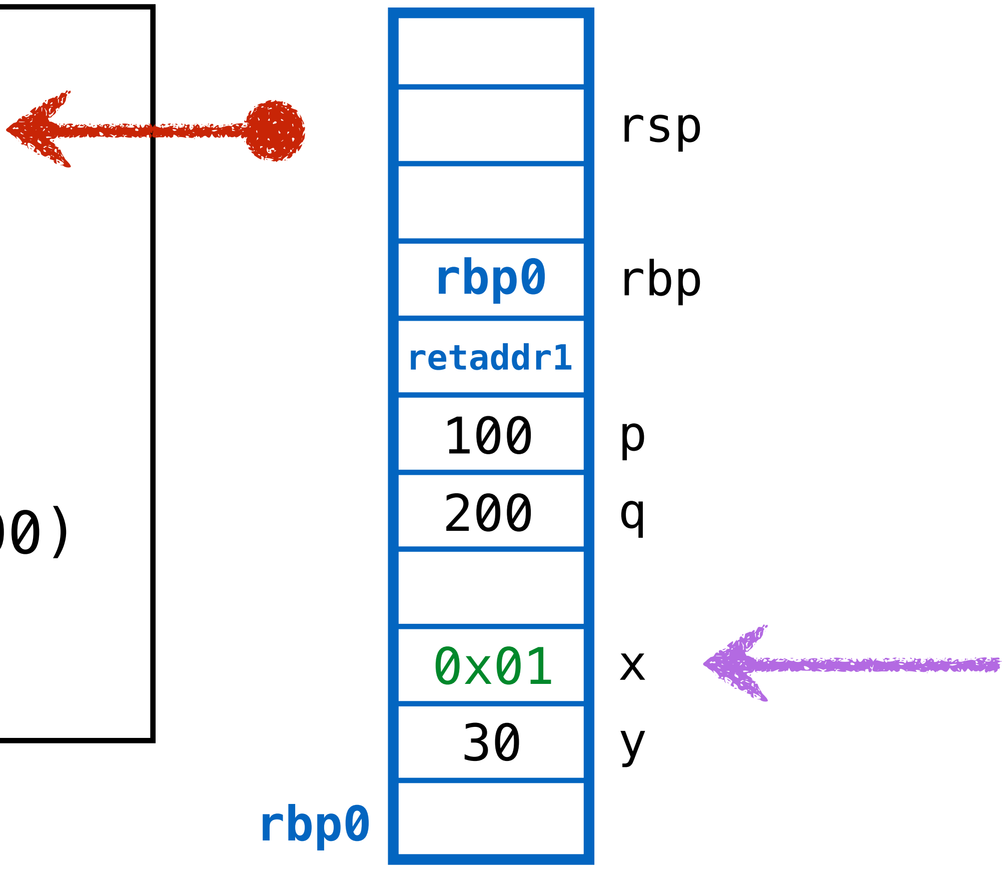


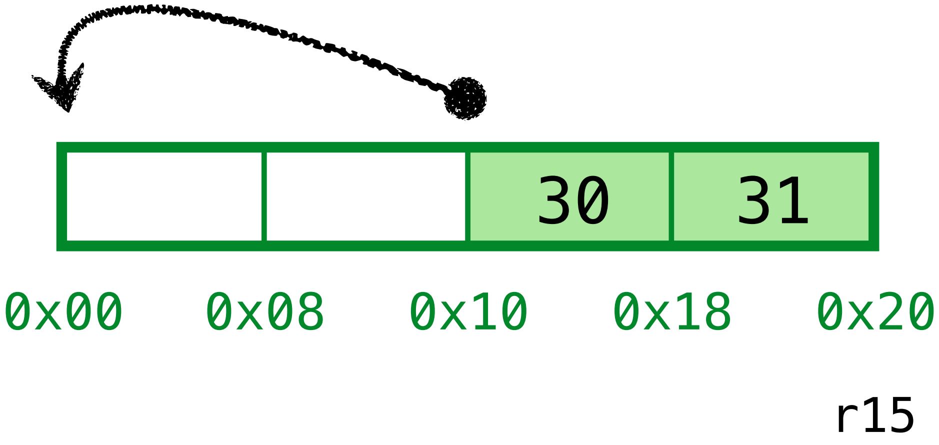
1. Compute FORWARD addrs

   e.g. 0x11 —> 0x01

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

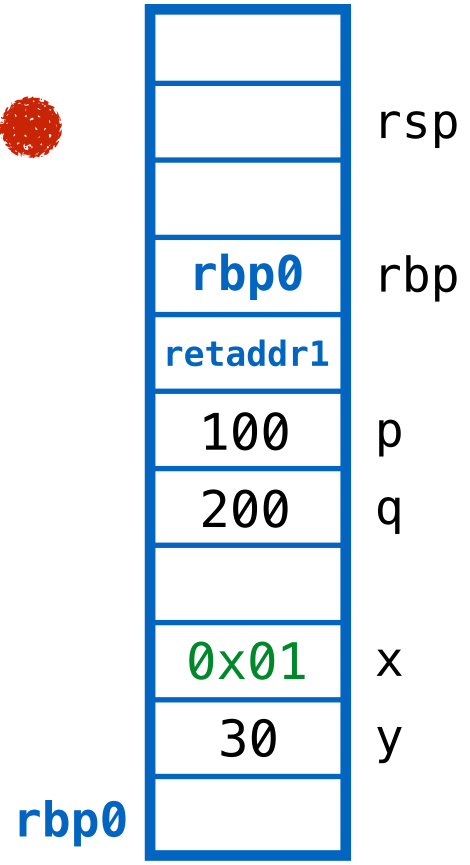


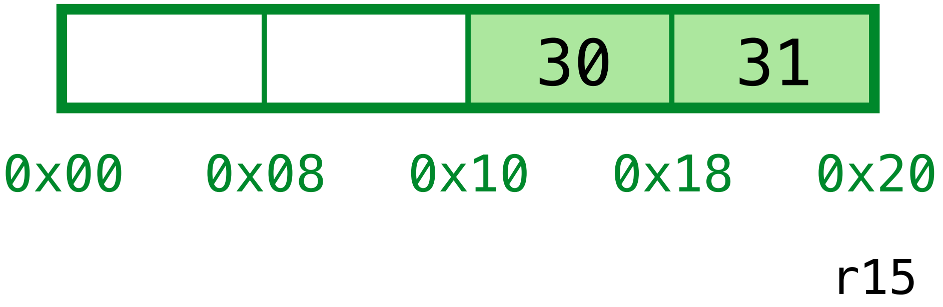


1. Compute FORWARD addrs
   e.g. 0x11 —> 0x01
2. REDIRECT addrs on stack

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```





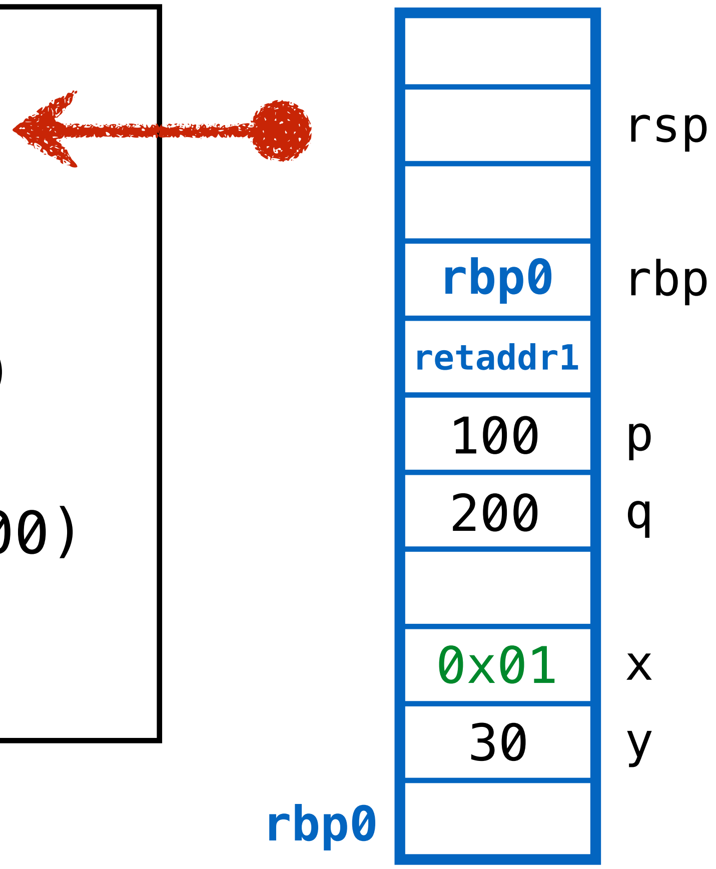
1. Compute FORWARD addrs
   e.g. 0x11 —> 0x01

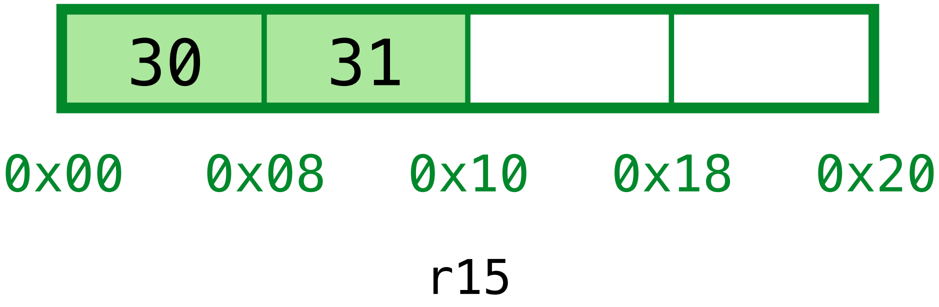
2. REDIRECT addrs on stack

3. COMPACT cells on heap

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```





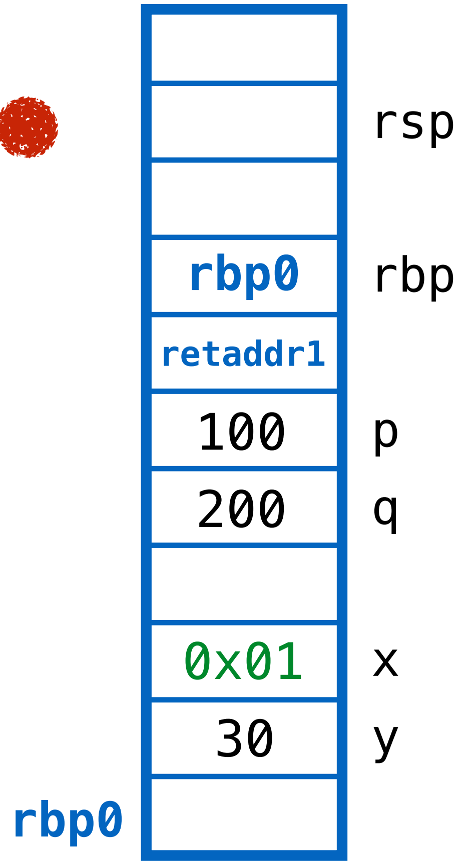
1. Compute FORWARD addrs
   e.g. 0x11 —> 0x01

2. REDIRECT addrs on stack

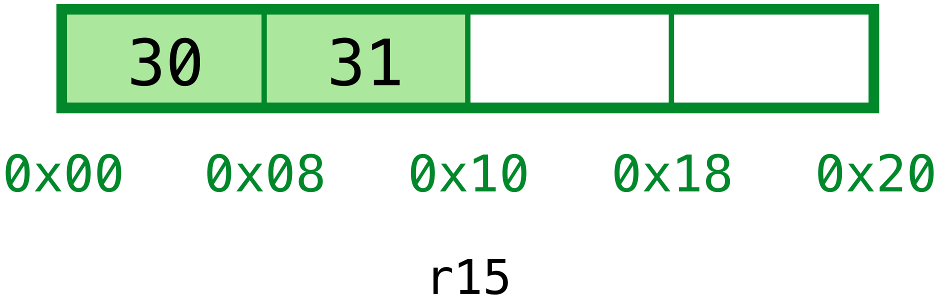
3. COMPACT cells on heap

ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

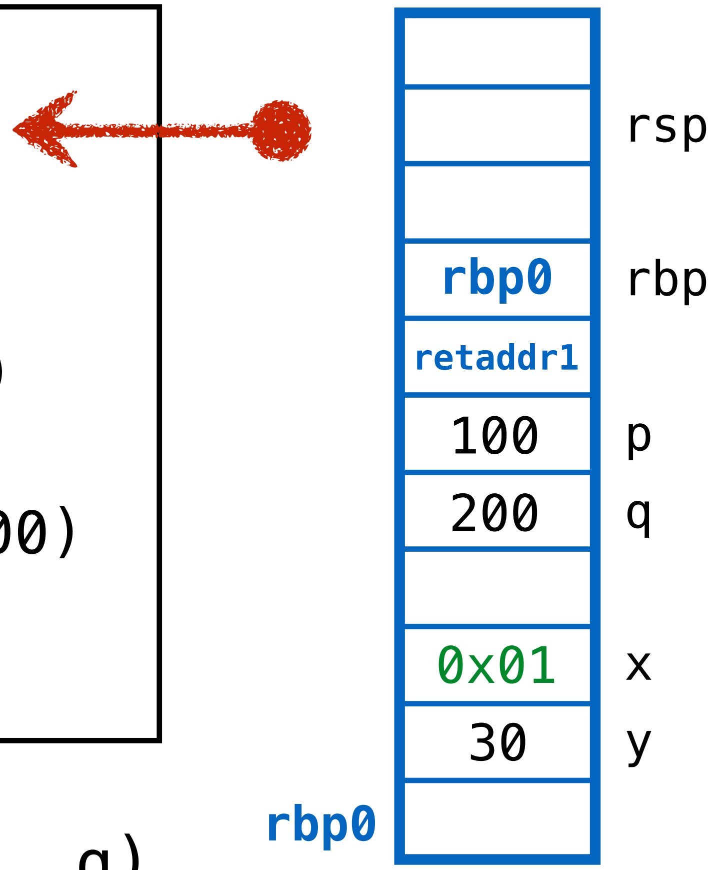


Yay! Have space for `(p, q)`

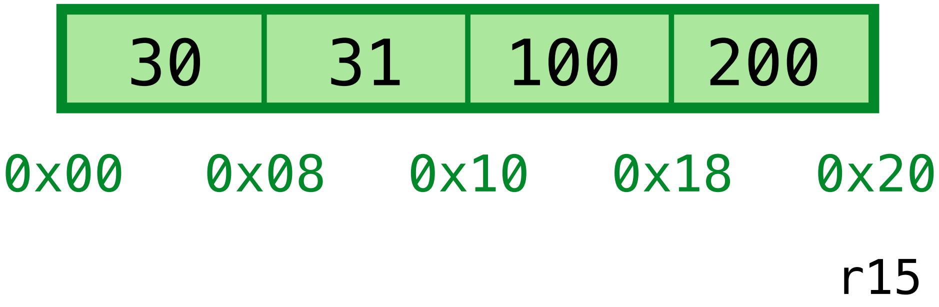


ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

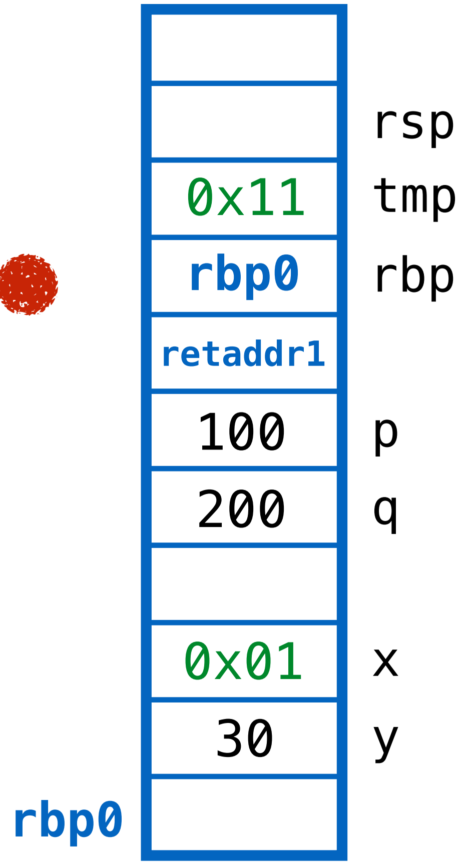


Yay! Have space for (p, q)



ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



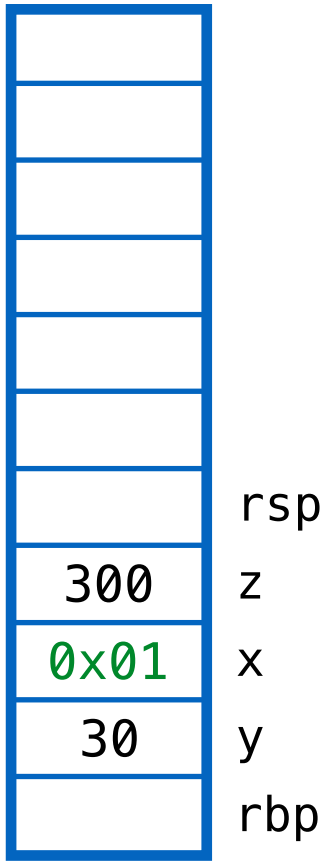
		rsp
	0x11	tmp
	rbp0	rbp
	retaddr1	
	100	p
	200	q
	0x01	x
	30	y
rbp0		

Return (rax) = 300

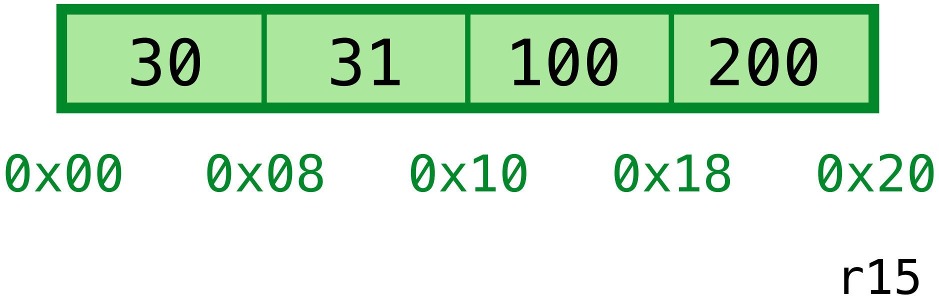
30	31	100	200	
0x00	0x08	0x10	0x18	0x20
				r15

ex3: garbage in the middle (with stack)

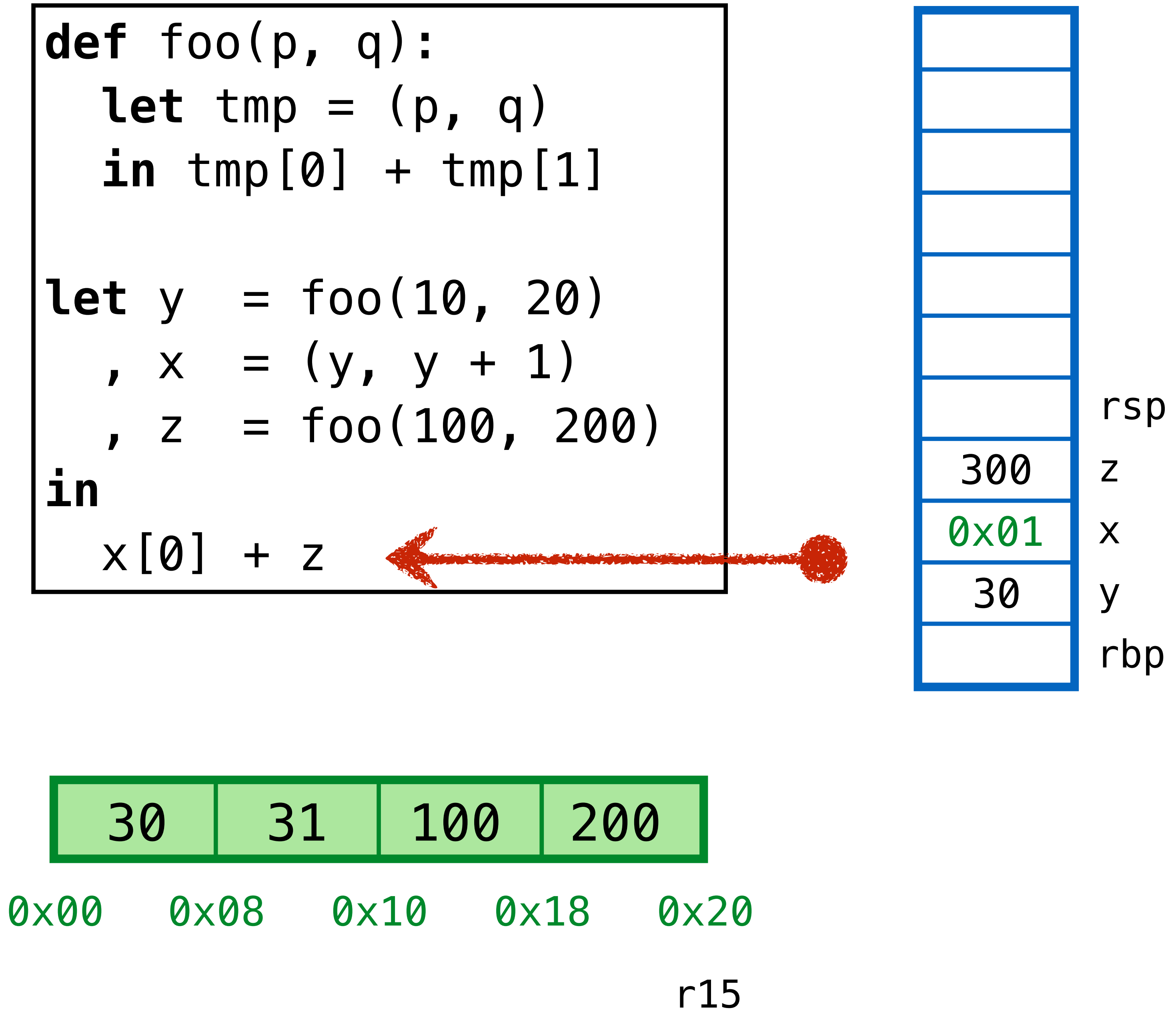
```
def foo(p, q):
   let tmp = (p, q)
   in tmp[0] + tmp[1]

let y  = foo(10, 20)
   , x  = (y, y + 1)
   , z  = foo(100, 200)
in
   x[0] + z
```



Return (rax) = 300



ex3: garbage in the middle (with stack)



```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```

rsp

300	z
0x01	x
30	y
	rbp

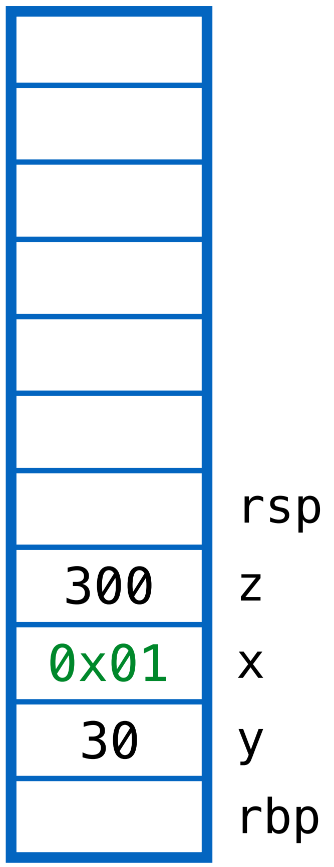
30	31	100	200

0x00 0x08 0x10 0x18 0x20

r15

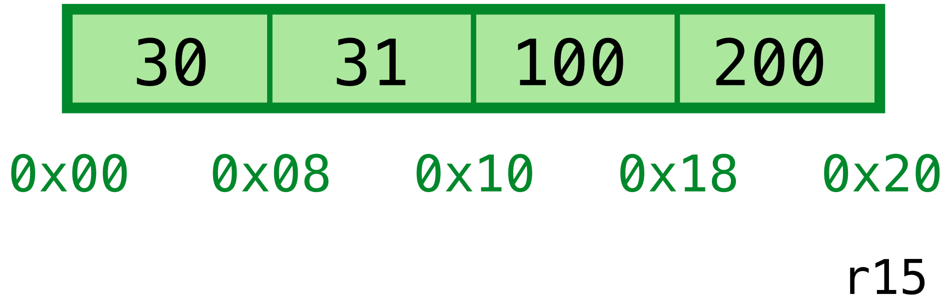
ex3: garbage in the middle (with stack)

```
def foo(p, q):
  let tmp = (p, q)
  in tmp[0] + tmp[1]

let y  = foo(10, 20)
  , x  = (y, y + 1)
  , z  = foo(100, 200)
in
  x[0] + z
```



	rsp
300	z
0x01	x
30	y
	rbp

Return (rax) = 30+300 = 330



30	31	100	200

0x00 0x08 0x10 0x18 0x20

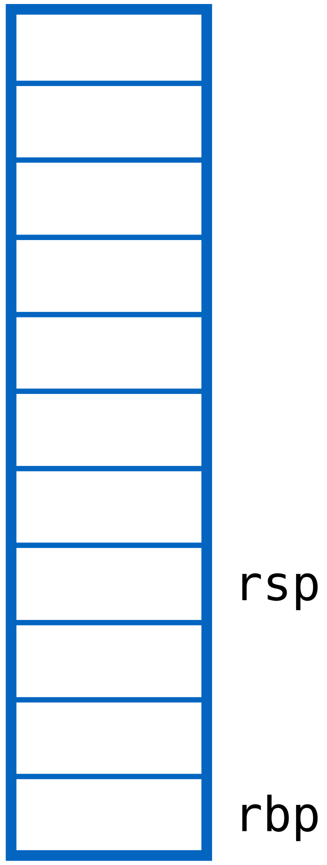
r15

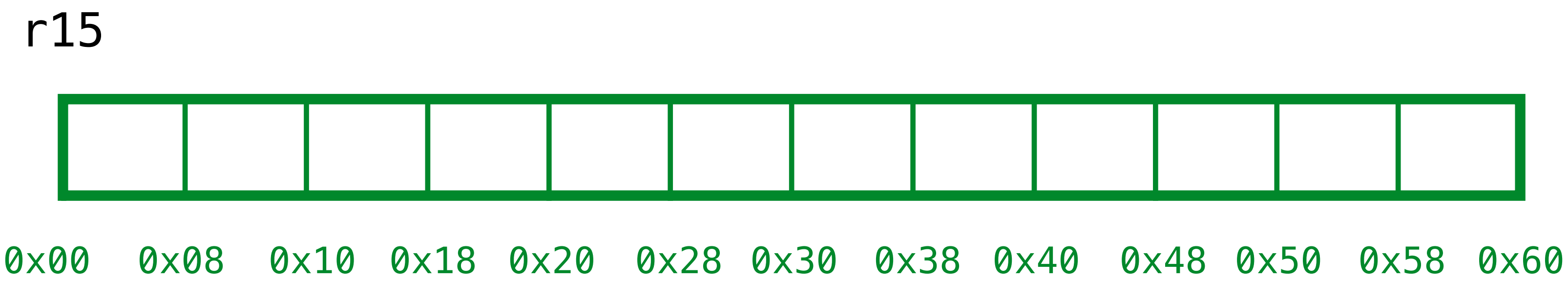
Garter / GC

Example 4

ex4: recursive data

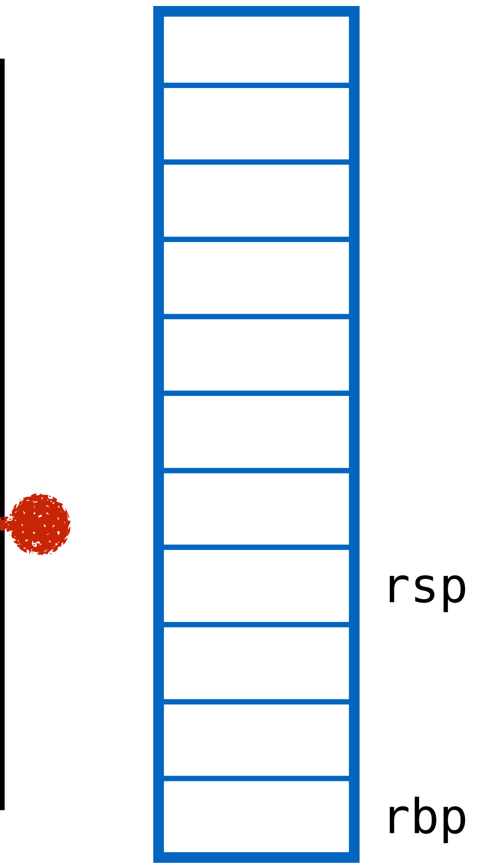
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



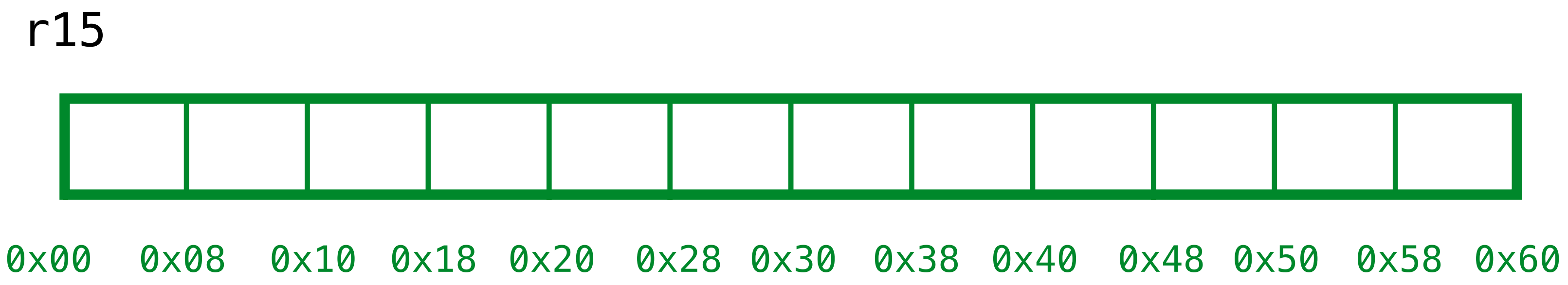


ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

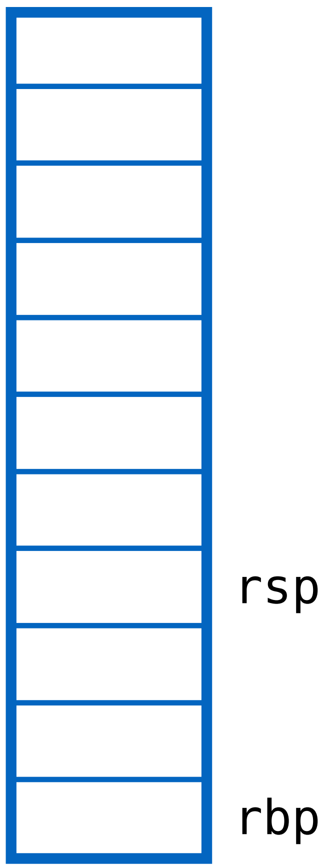


call range(0, 3)



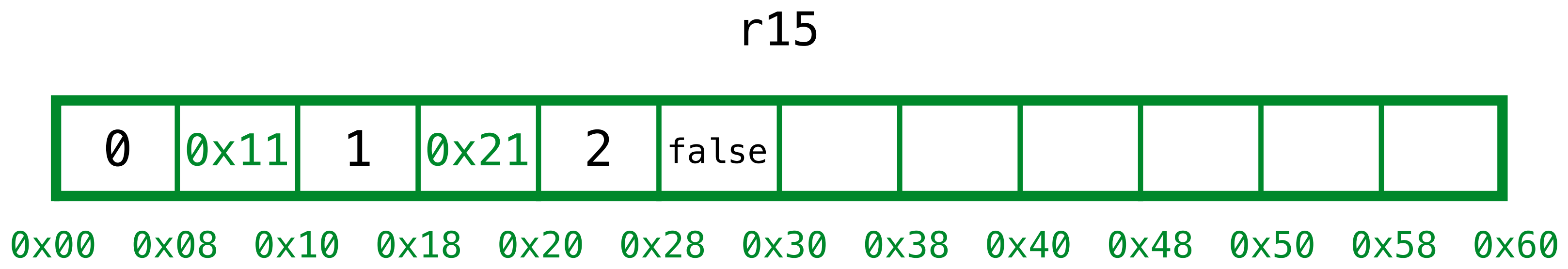
ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

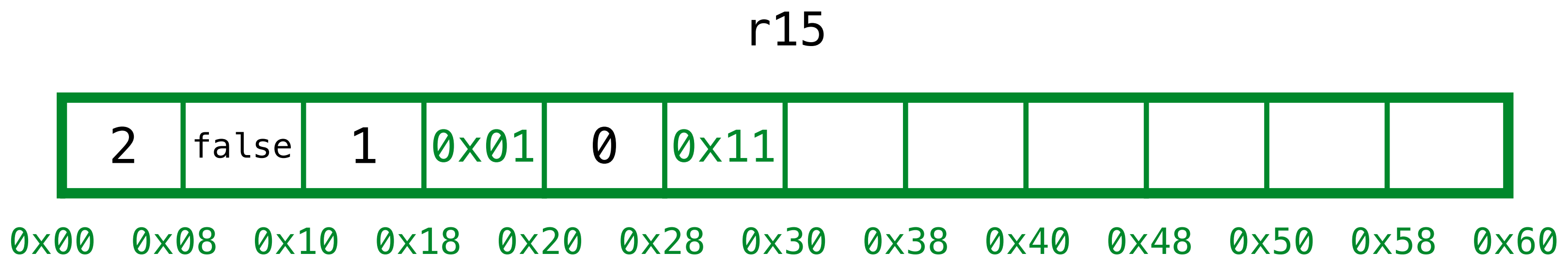


QUIZ: What is heap when `range(0,3)` returns?

(A)

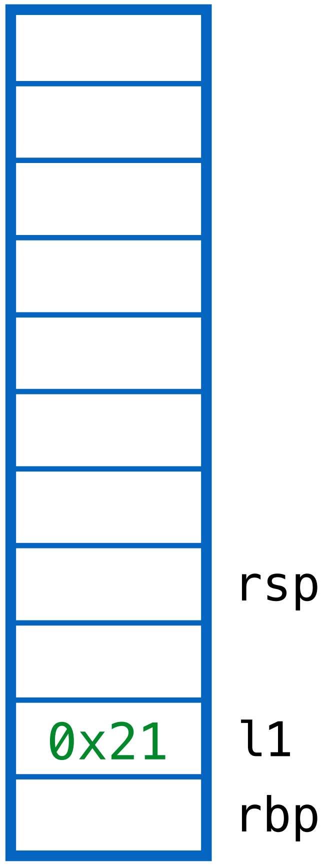


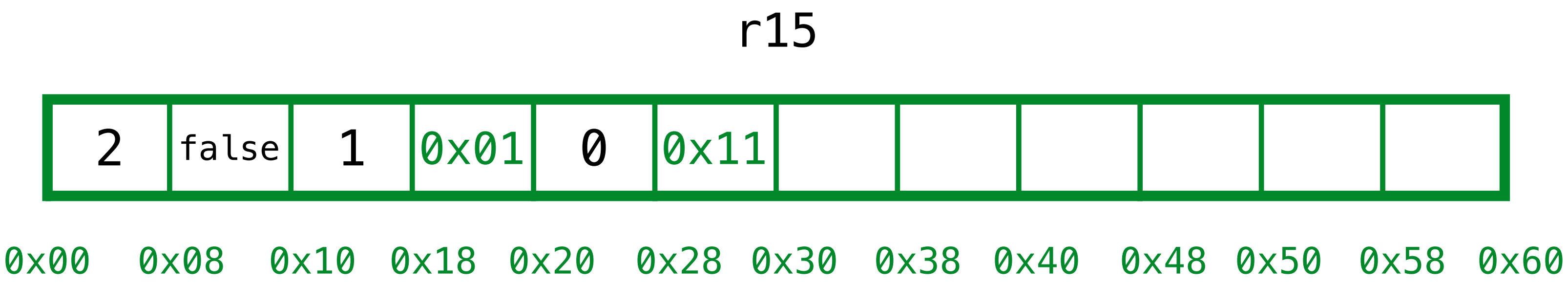
(B)



ex4: recursive data

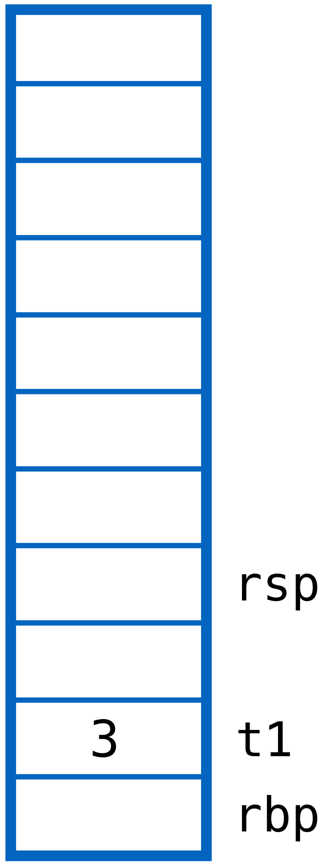
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



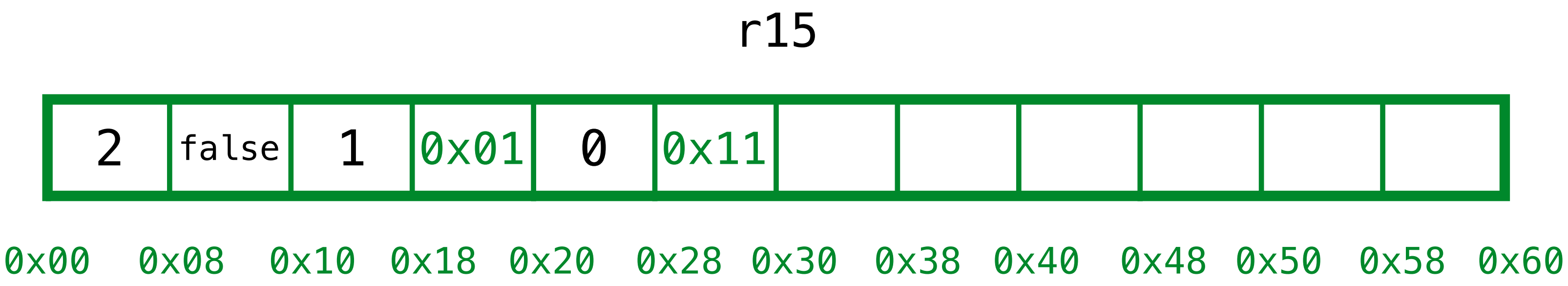


ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

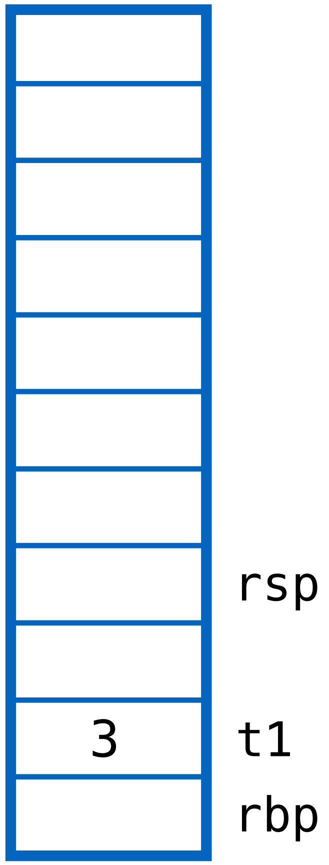


Result sum(0x11) = 3



ex4: recursive data

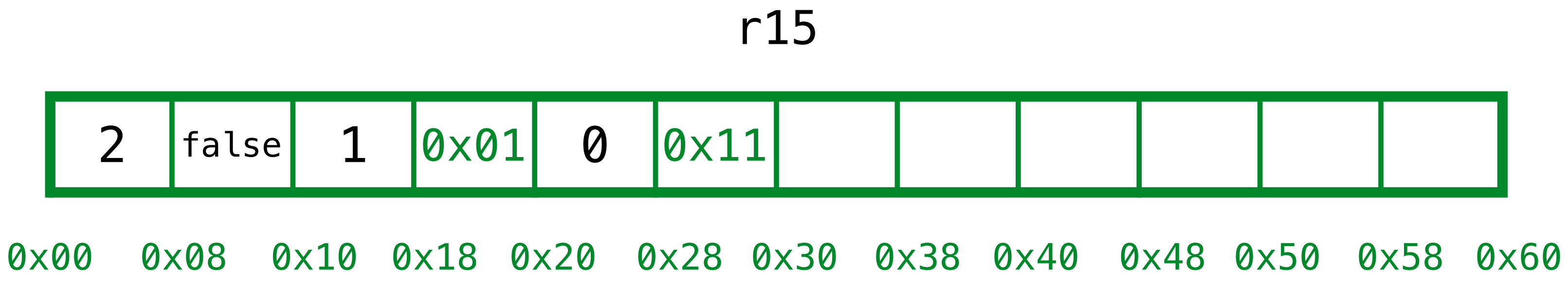
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
       let l1 = range(0, 3)
       in sum(l1)
  , t  = range(t1, t1 + 3)
in
  (1000, l)
```



rsp

3 t1

rbp



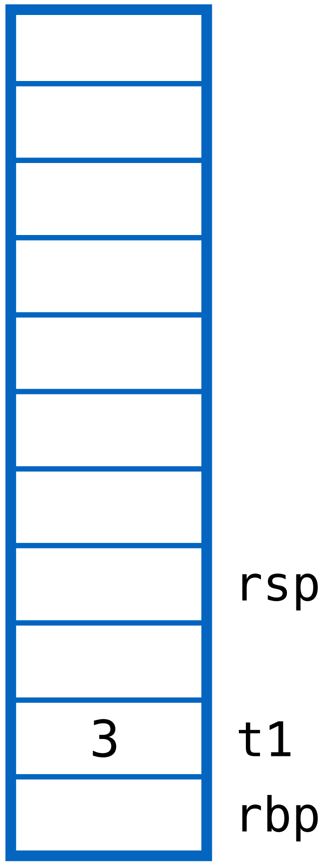
r15

2	false	1	0x01	0	0x11						

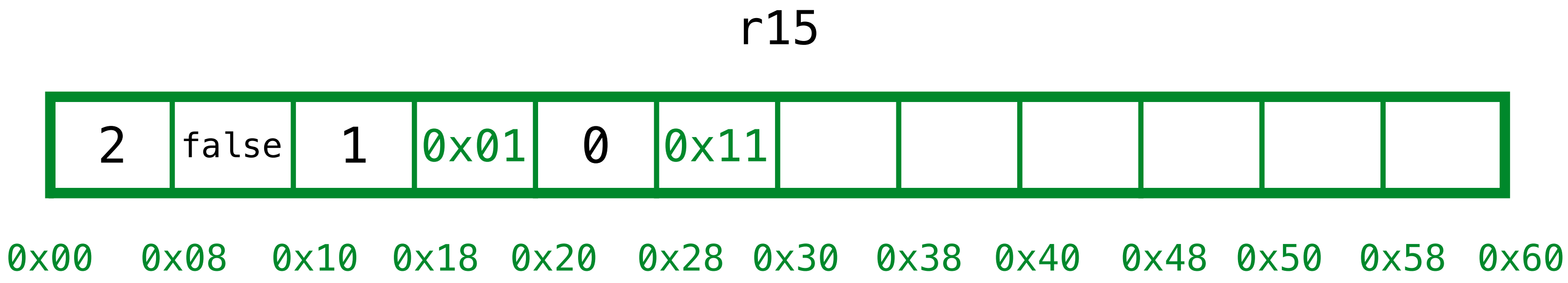
0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58 0x60

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

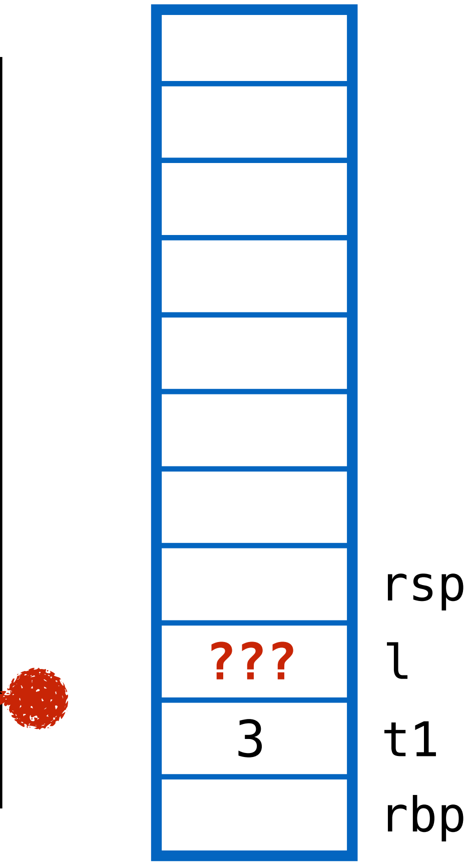


call range(3,6)

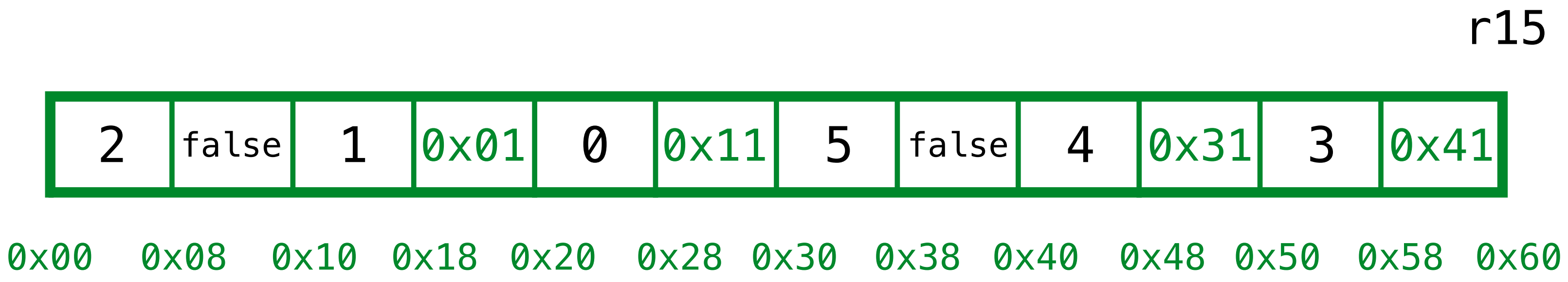


ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
       let l1 = range(0, 3)
       in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



call range(3,6)

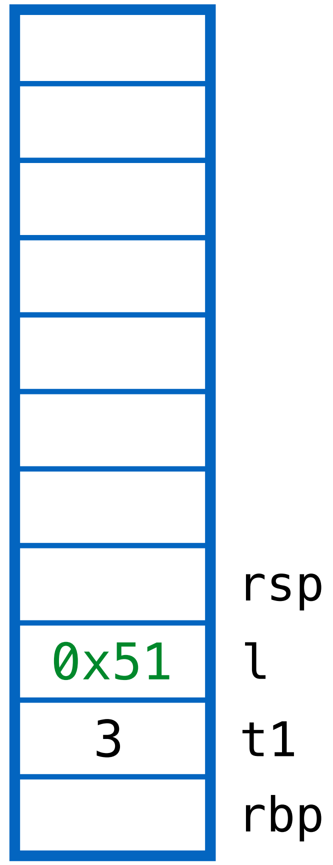


QUIZ: What is the value of `l`?

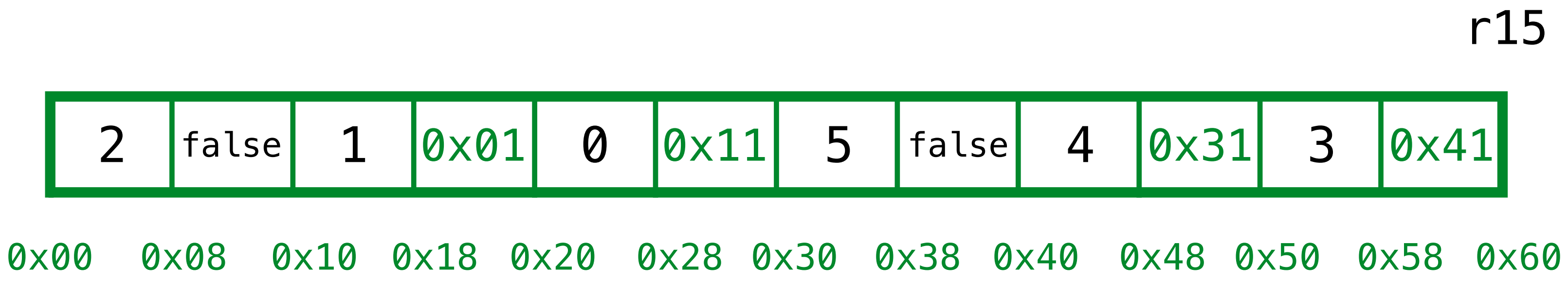
(A) 0x30 (B) 0x31 (C) 0x50 (D) 0x51 (E) 0x60

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



Yikes! Out of Memory!



2	false	1	0x01	0	0x11	5	false	4	0x31	3	0x41

0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58 0x60

ex4: recursive data

QUIZ: Which cells are "live" on the heap?

(A) 0x00

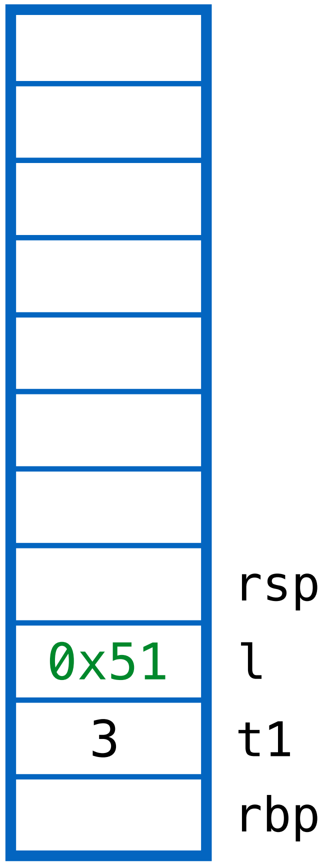
(B) 0x10

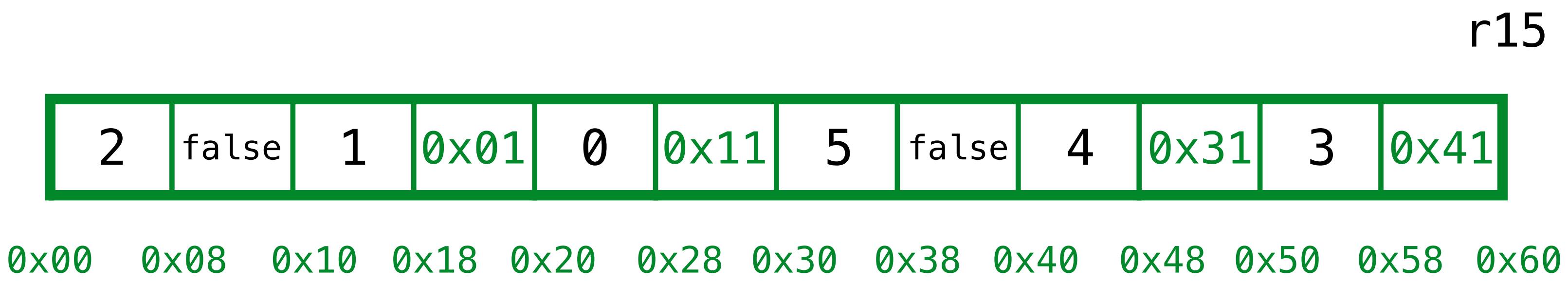
(C) 0x20

(D) 0x30

(E) 0x40

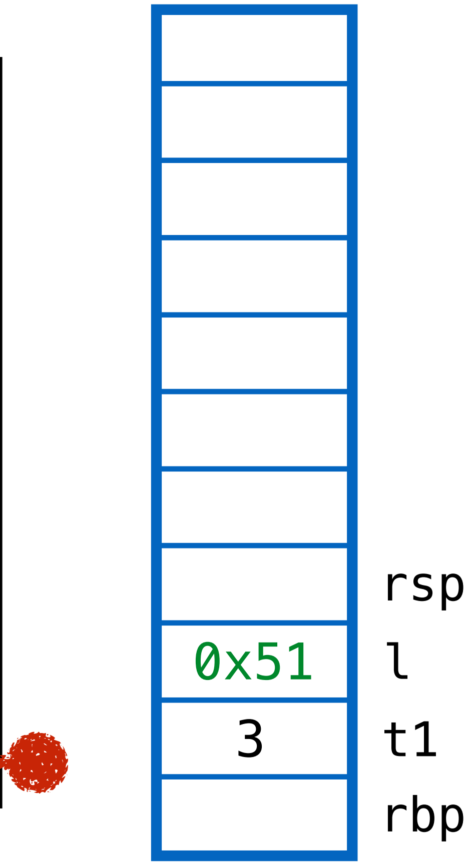
(F) 0x50





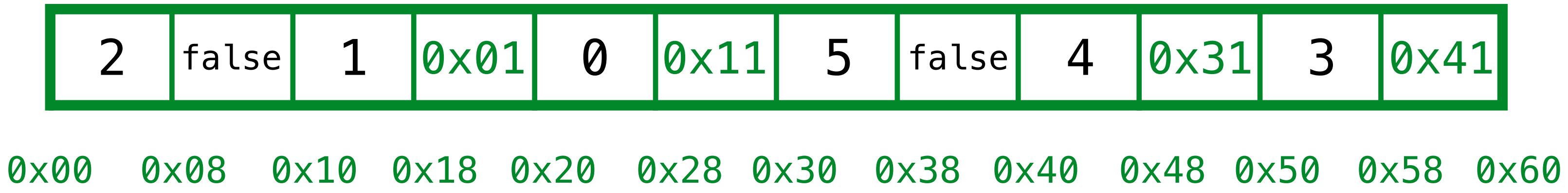
ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



	rsp
0x51	l
3	t1
	rbp

r15

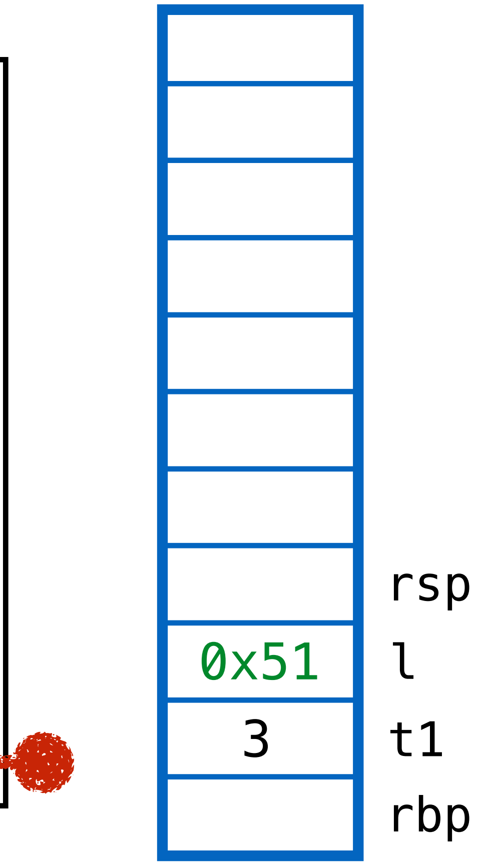


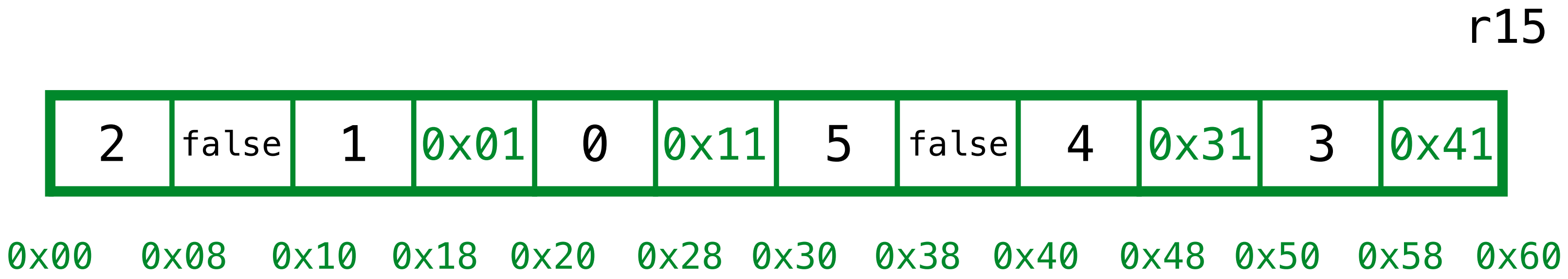
0x00	0x08	0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58	0x60
2	false	1	0x01	0	0x11	5	false	4	0x31	3	0x41	

1. MARK live addrs
2. Compute FORWARD addrs
3. REDIRECT addrs on stack
4. COMPACT cells on heap

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

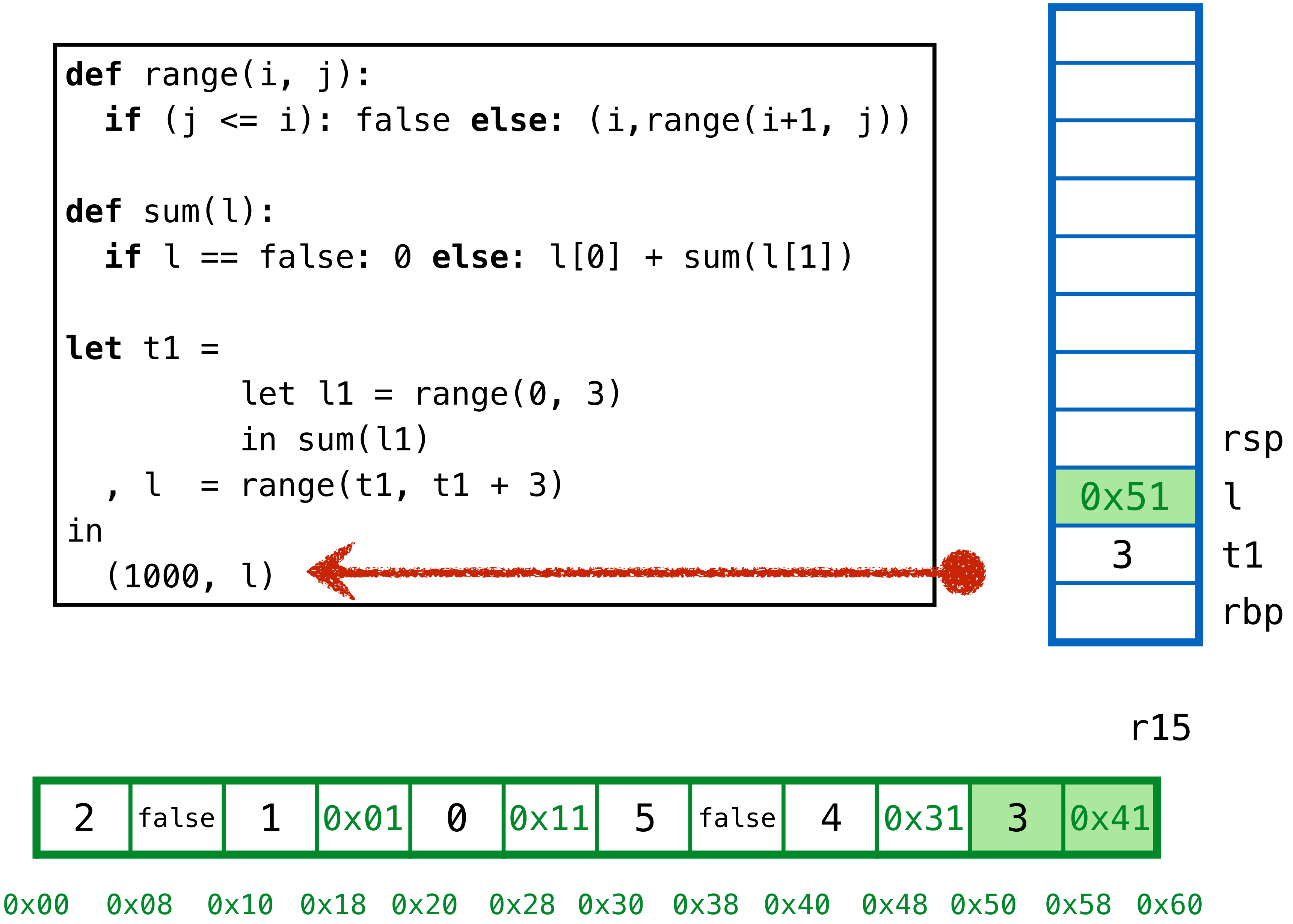




1. MARK live addrs

reachable from stack

ex4: recursive data

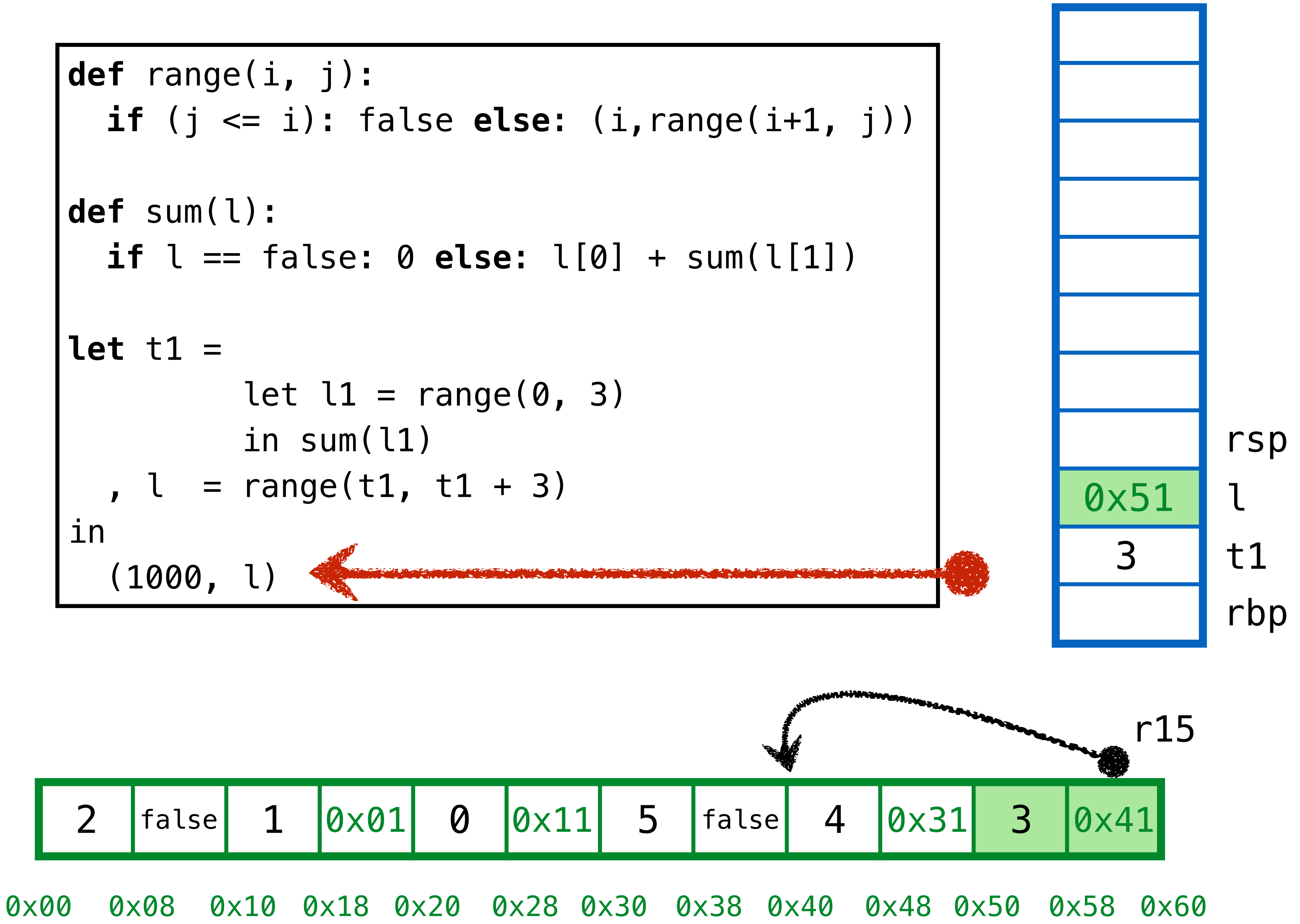


```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

1. MARK live addrs

reachable from stack

ex4: recursive data



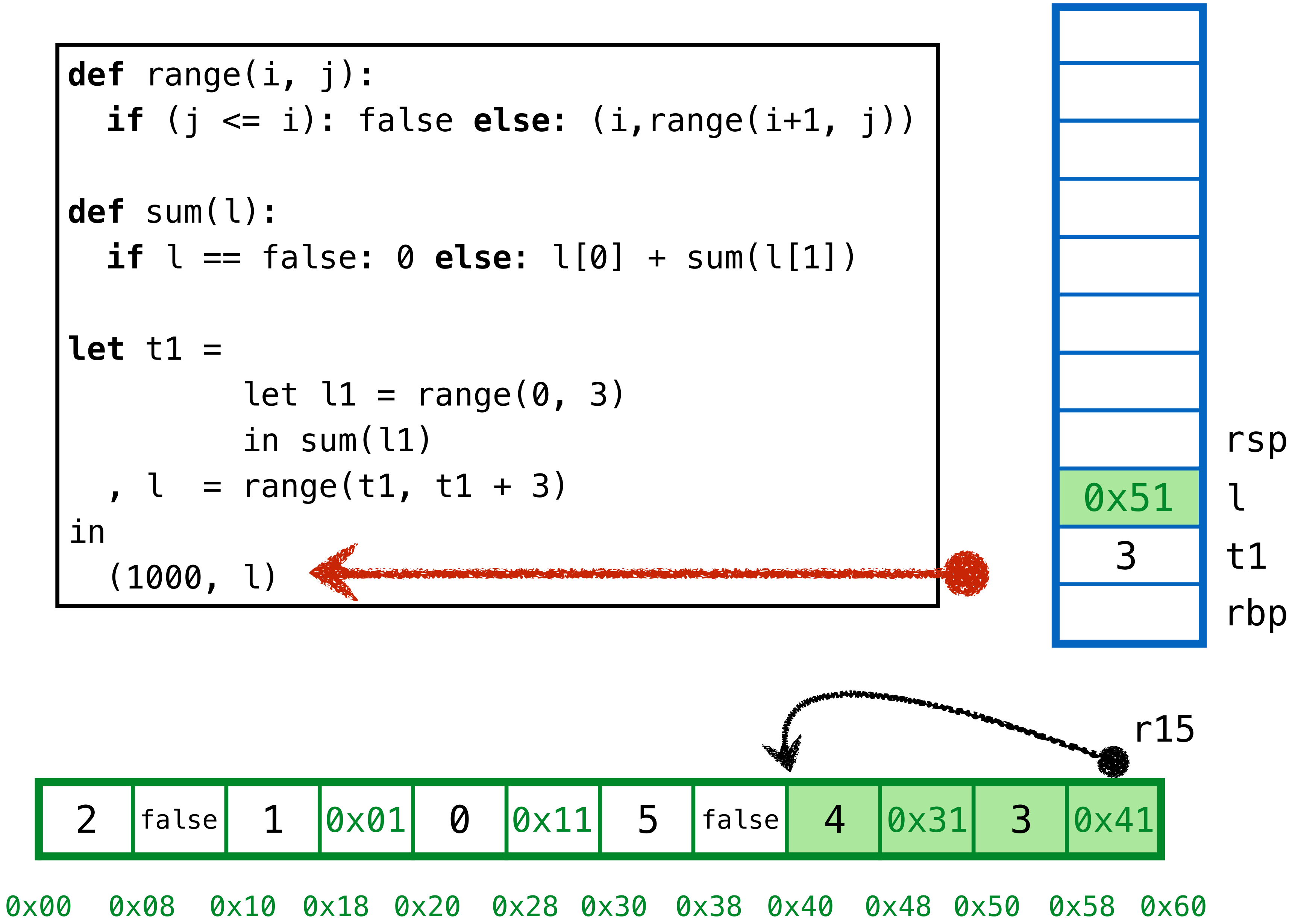
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

1. MARK live addrs

reachable from stack

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



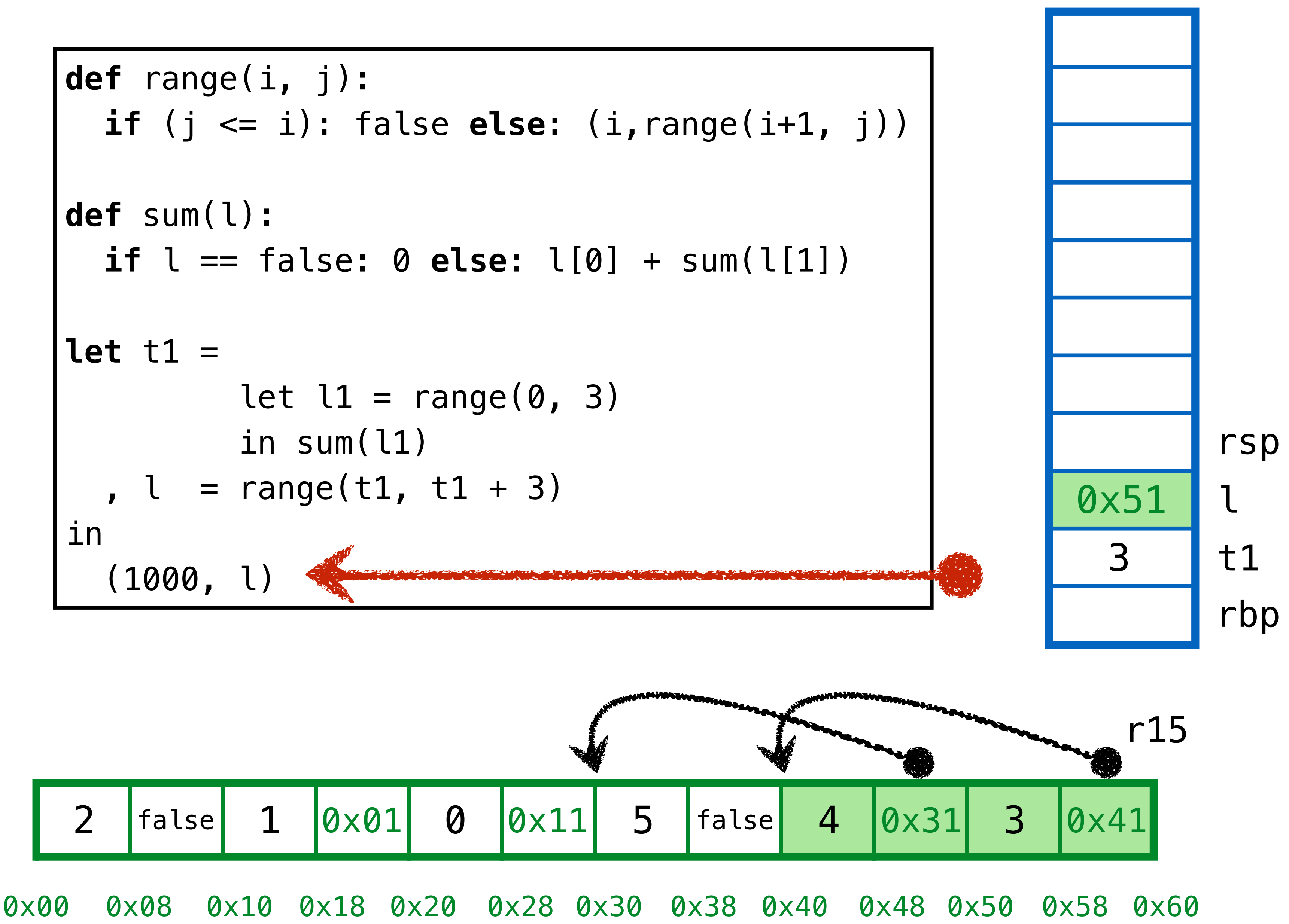
2	false	1	0x01	0	0x11	5	false	4	0x31	3	0x41

0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58 0x60

1. MARK live addrs

reachable from stack

ex4: recursive data



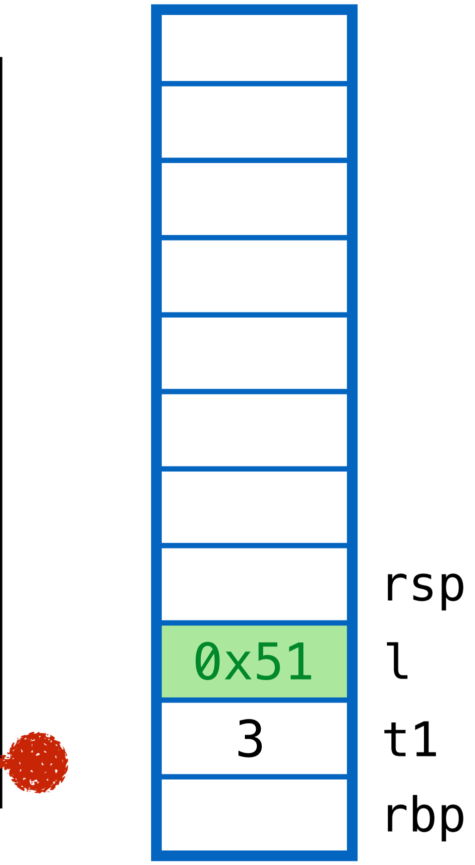
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

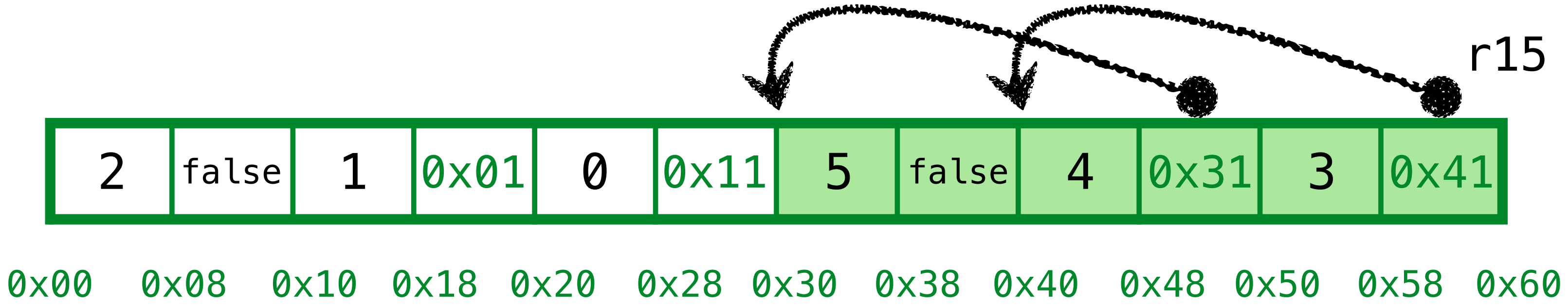
1. MARK live addrs

reachable from stack

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



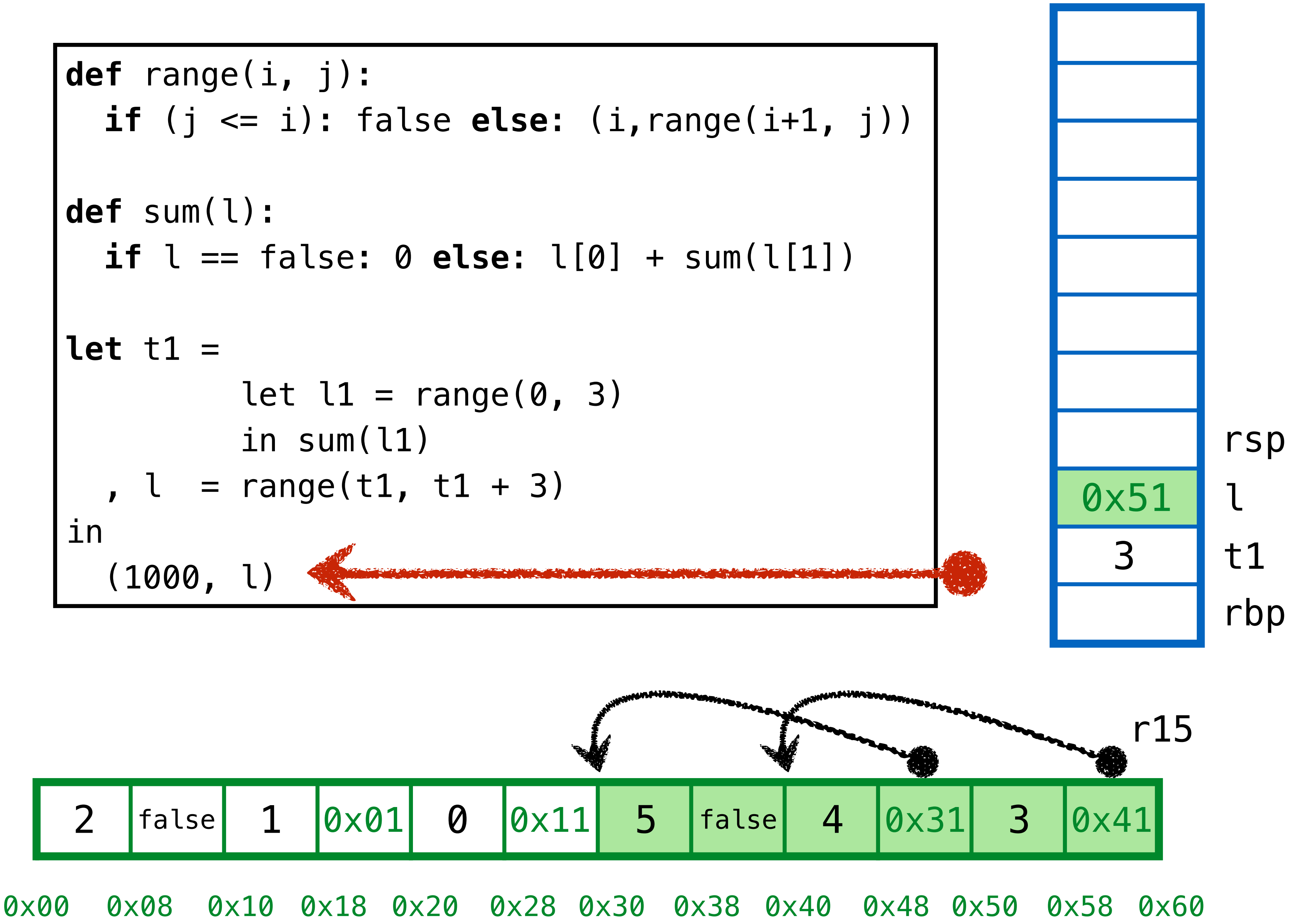


1. MARK live addrs

reachable from stack

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



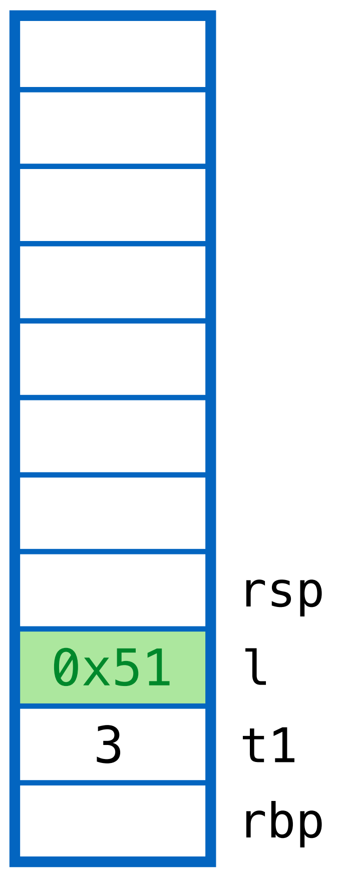
rsp
0x51
l
3
t1
rbp
r15
0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58 0x60

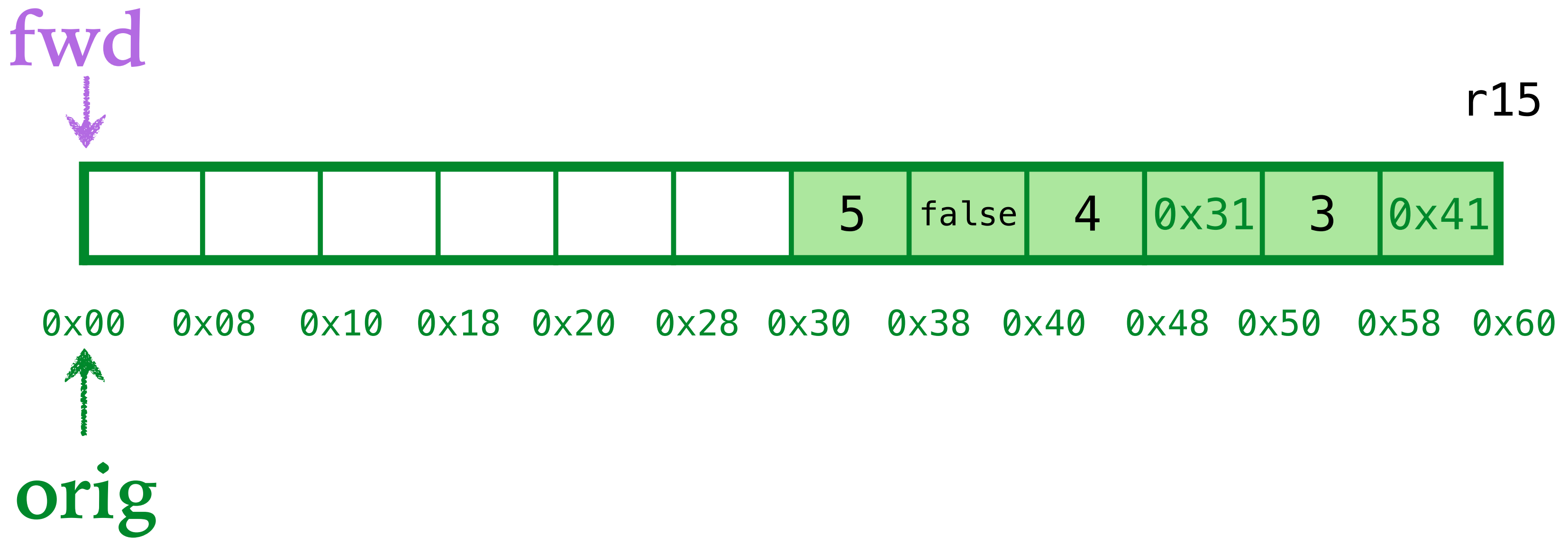
2	false	1	0x01	0	0x11	5	false	4	0x31	3	0x41

Done!

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

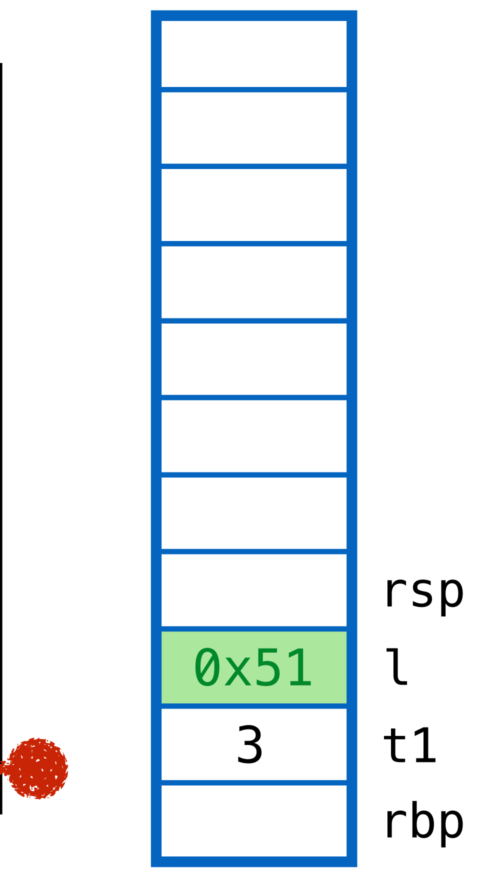


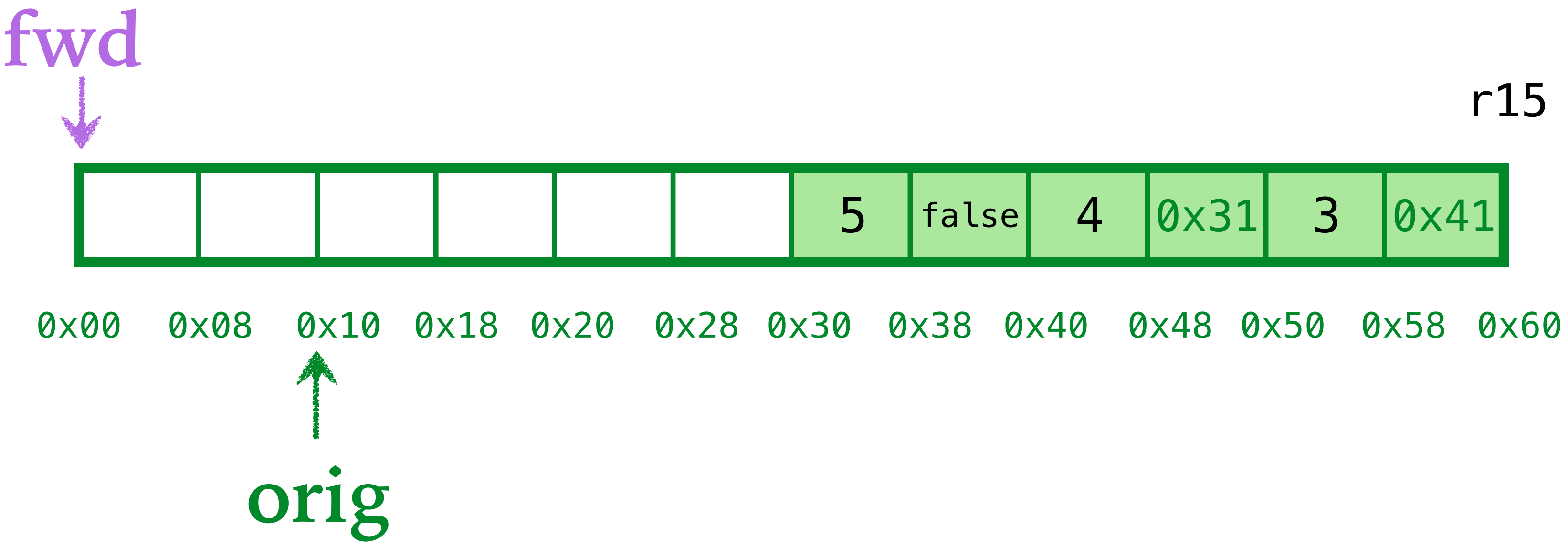


2.Compute FORWARD addrs

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

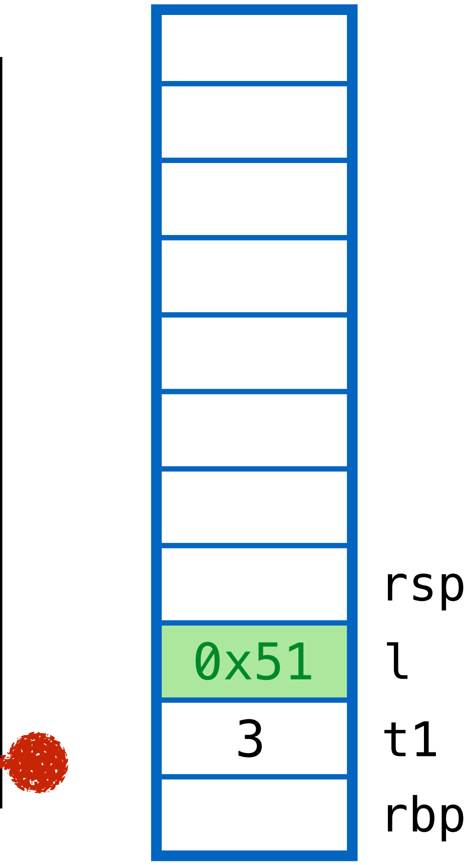


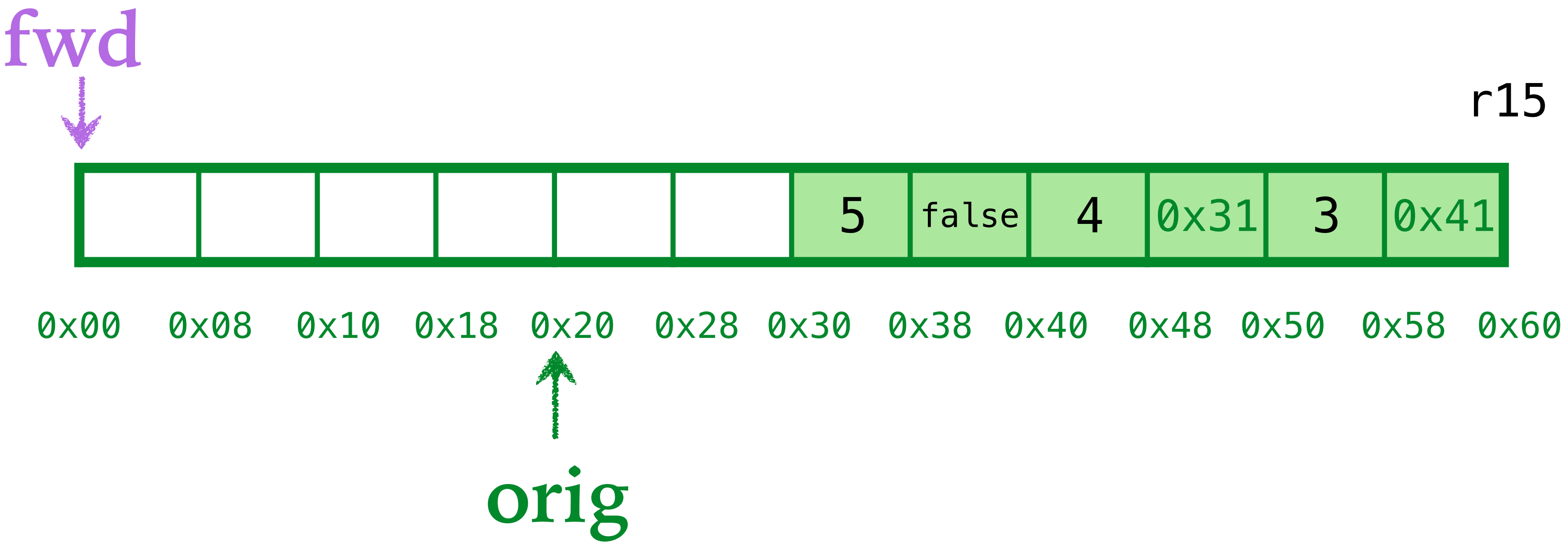


2. Compute FORWARD addrs

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

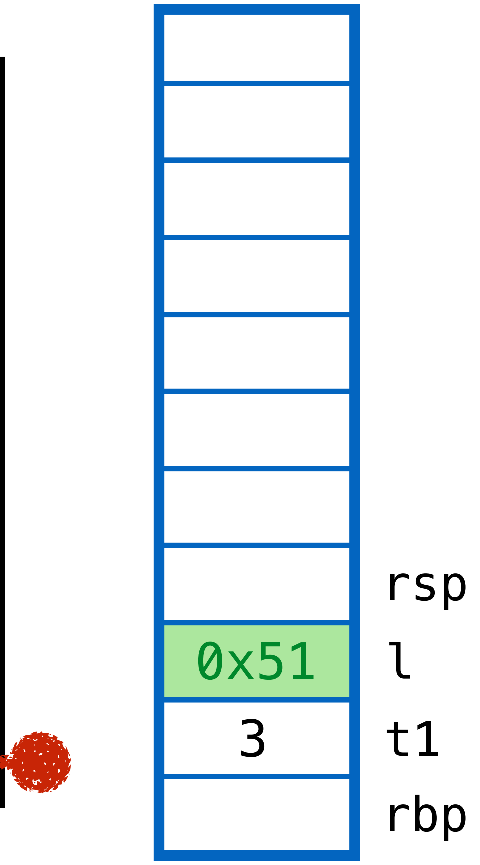


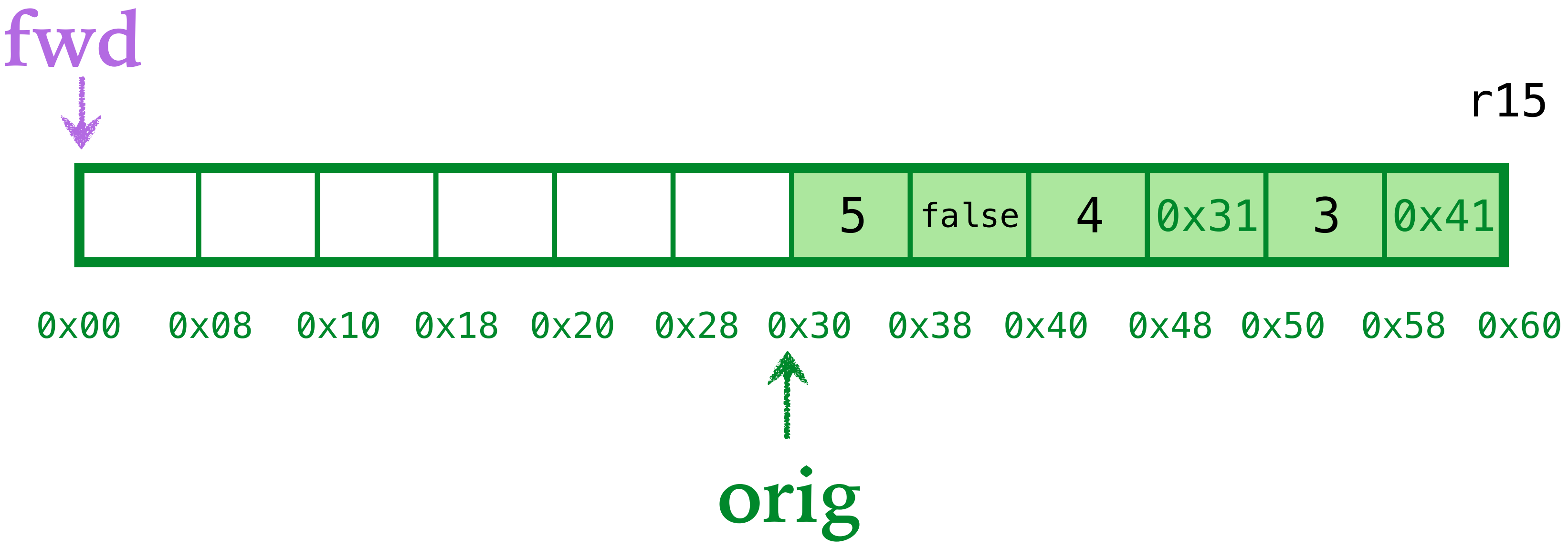


2.Compute FORWARD addrs

ex4: recursive data

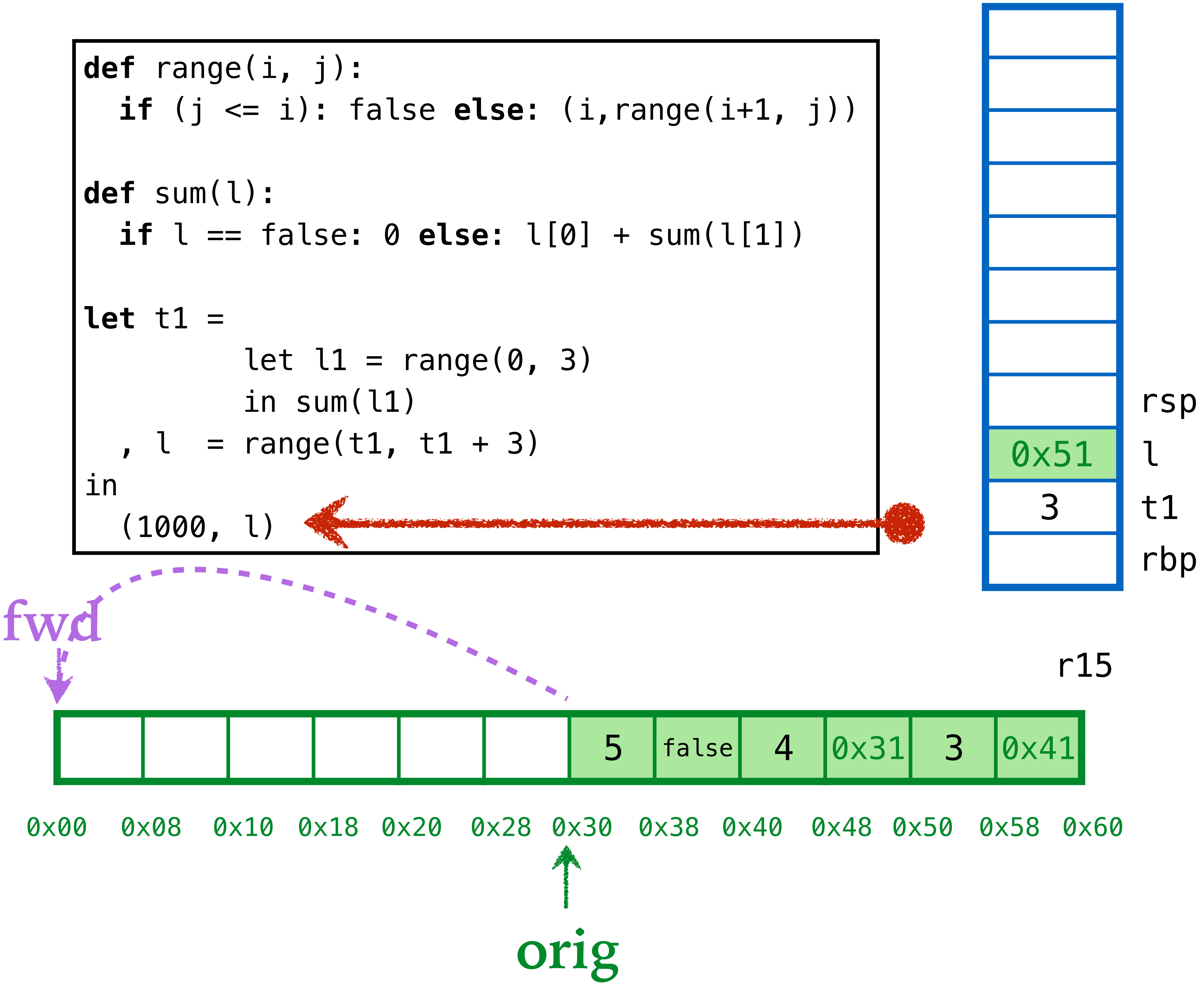
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```





2.Compute FORWARD addrs

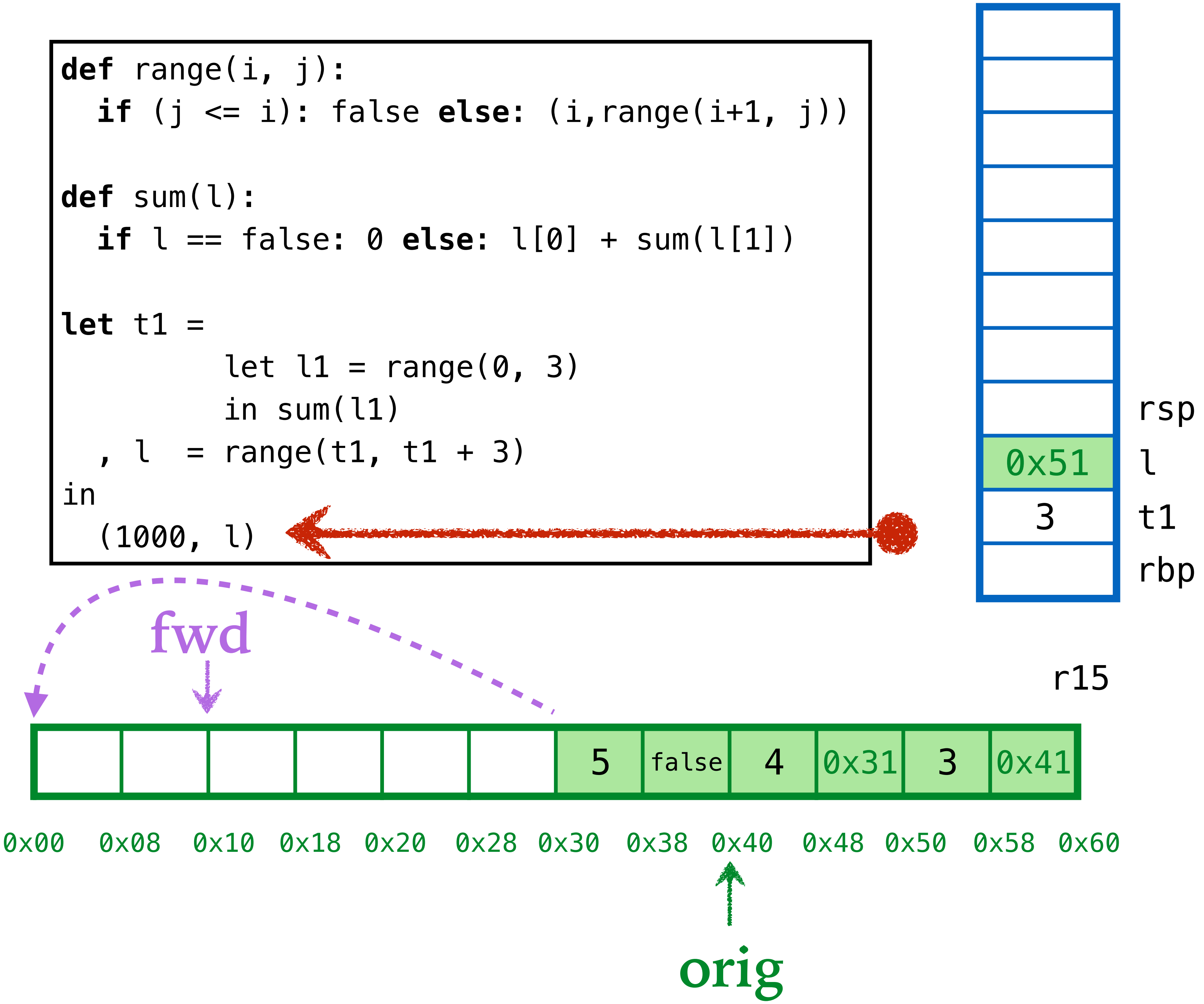
ex4: recursive data



```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

2. Compute FORWARD addrs

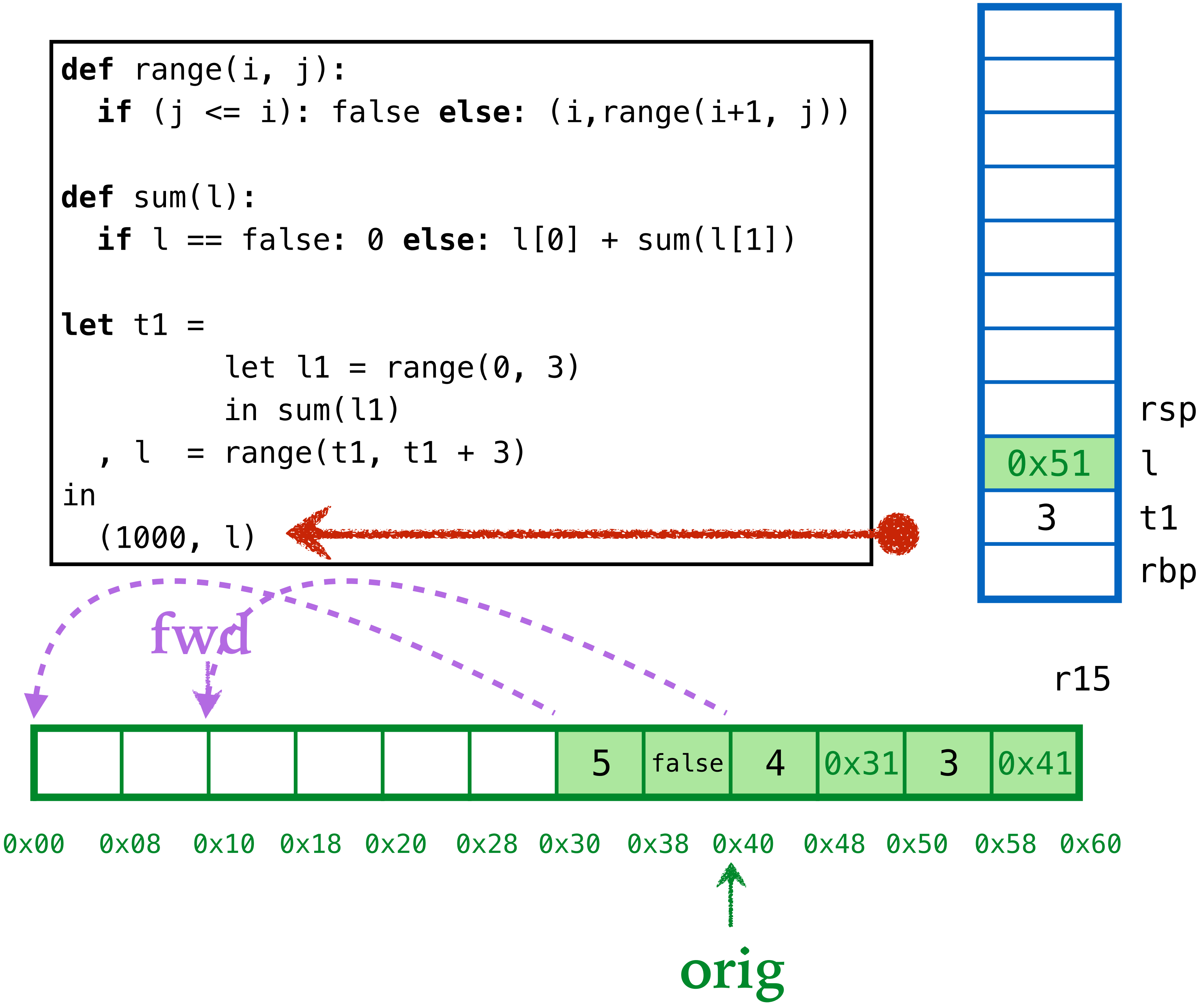
ex4: recursive data



```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

2.Compute FORWARD addrs

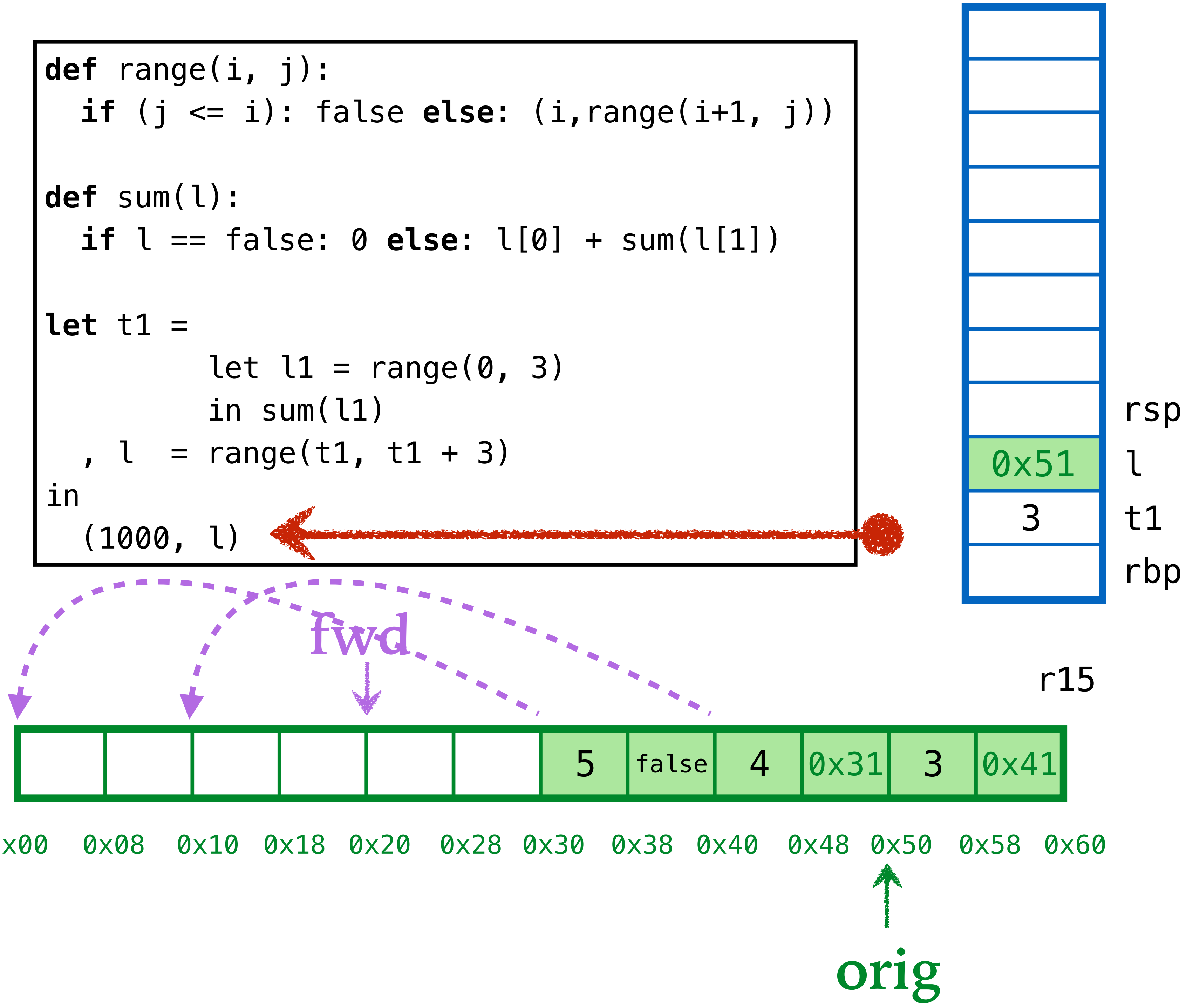
ex4: recursive data



```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

2. Compute FORWARD addrs

ex4: recursive data



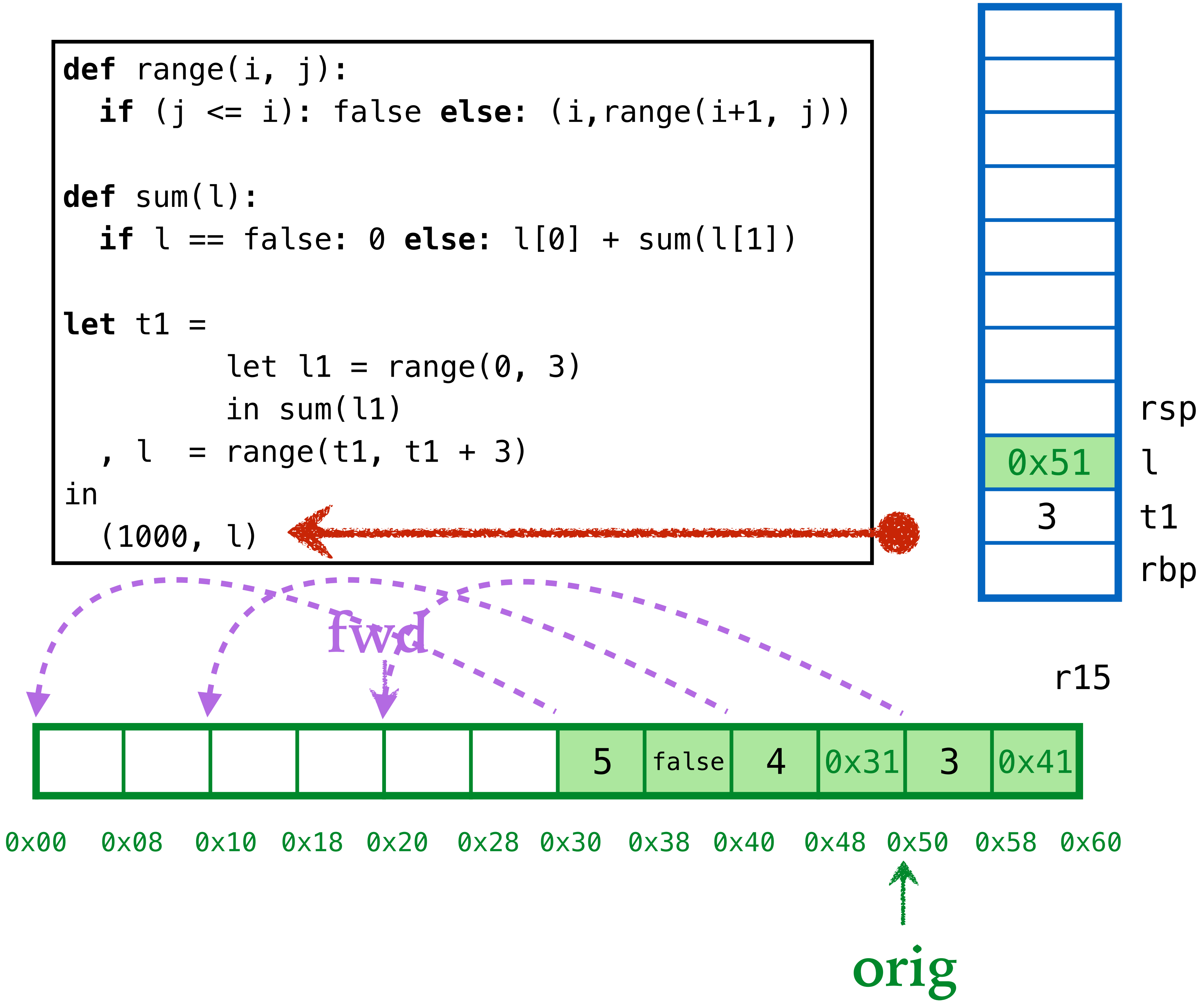
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

0x00	0x08	0x10	0x18	0x20	0x28	0x30	0x38	0x40	0x48	0x50	0x58
						5	false	4	0x31	3	0x41

2.Compute FORWARD addrs

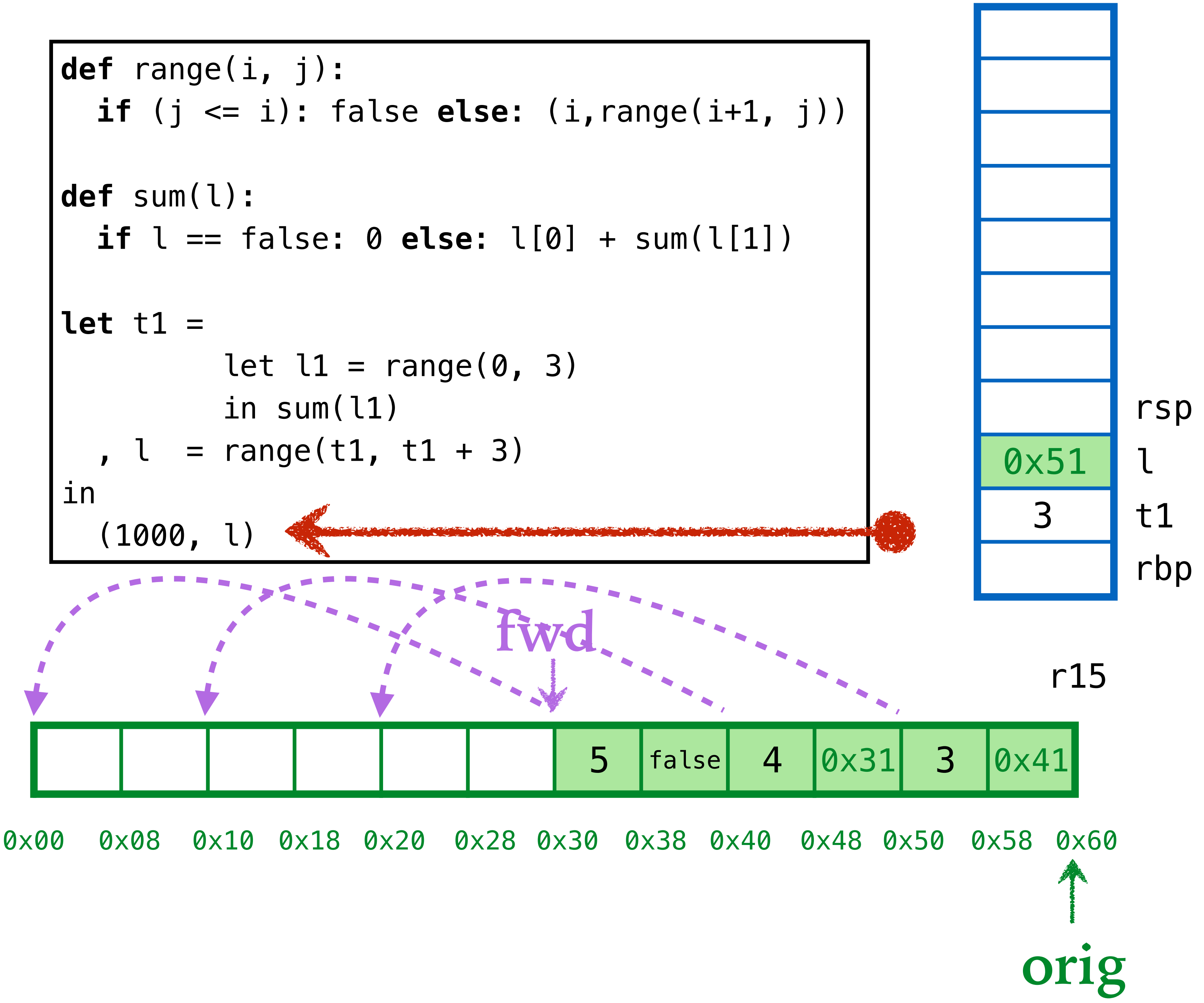
ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



2.Compute FORWARD addrs

ex4: recursive data

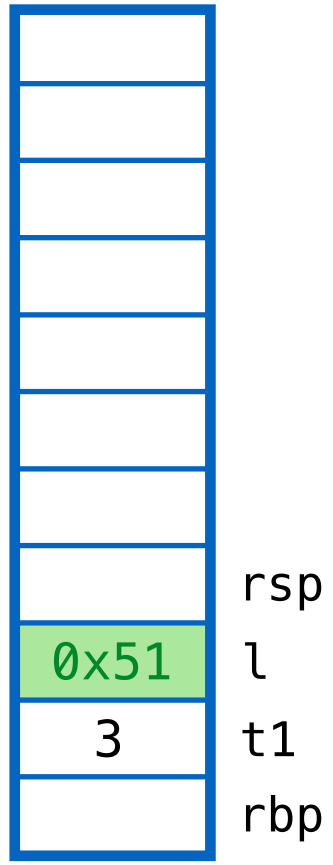


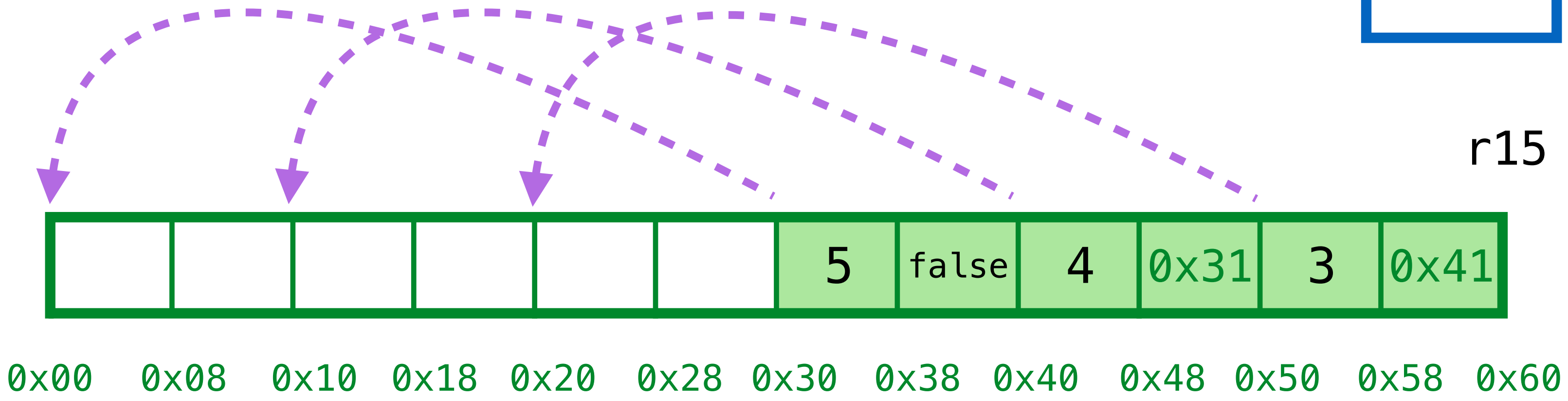
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

2.Compute FORWARD addrs

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



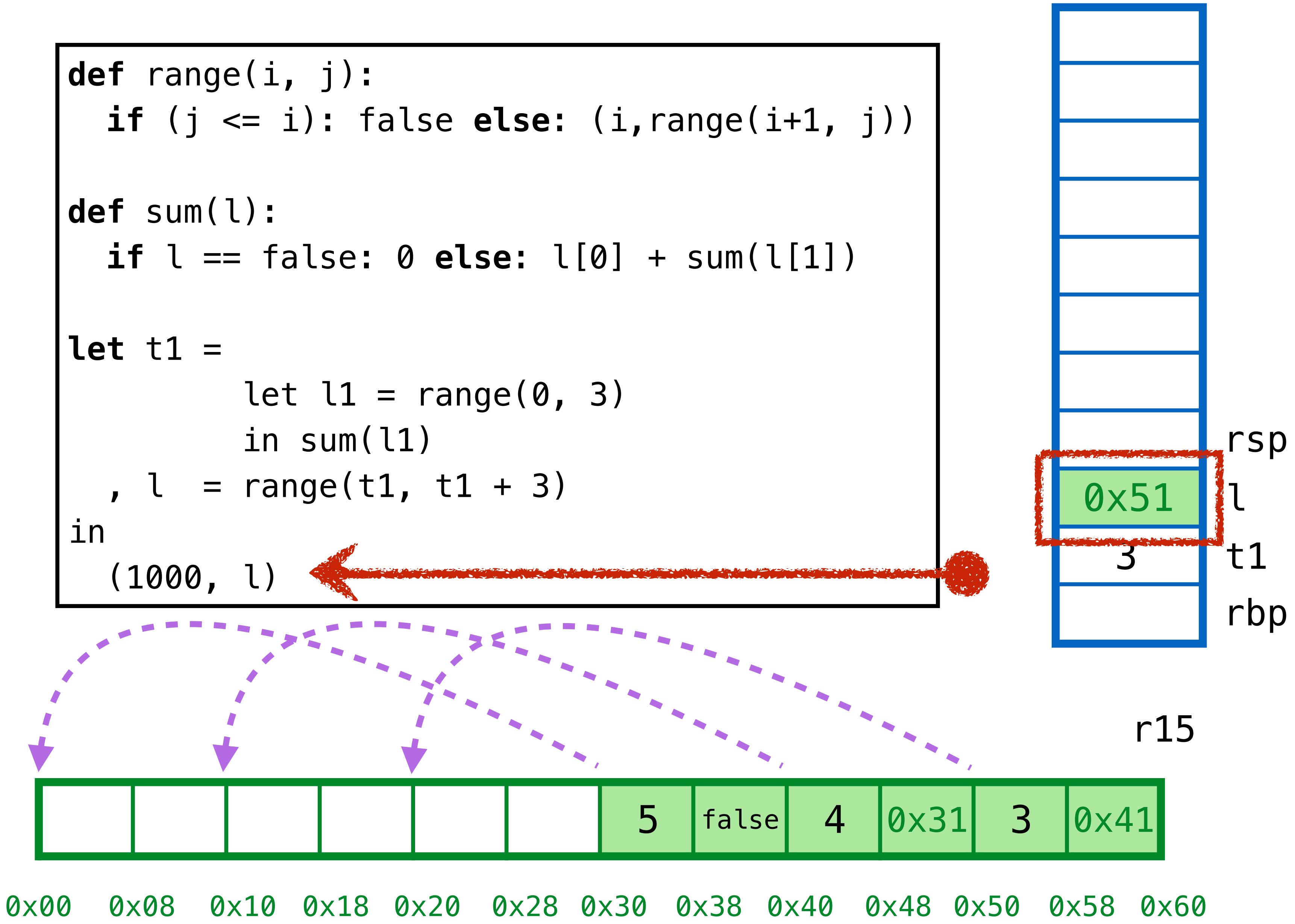


2.Compute FORWARD addrs

Where should we store the forward addrs?

ex4: recursive data

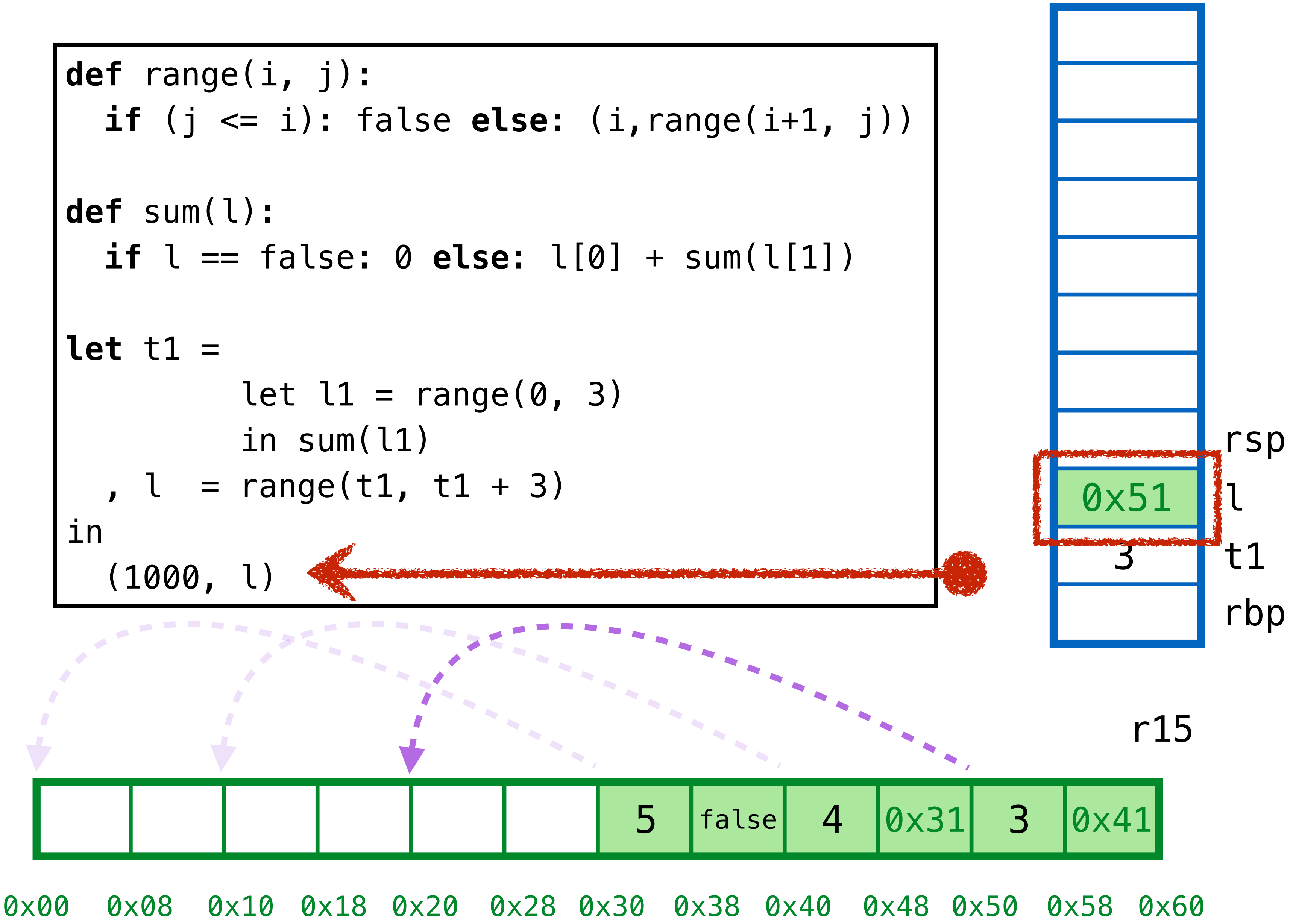
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



3. REDIRECT addrs on stack

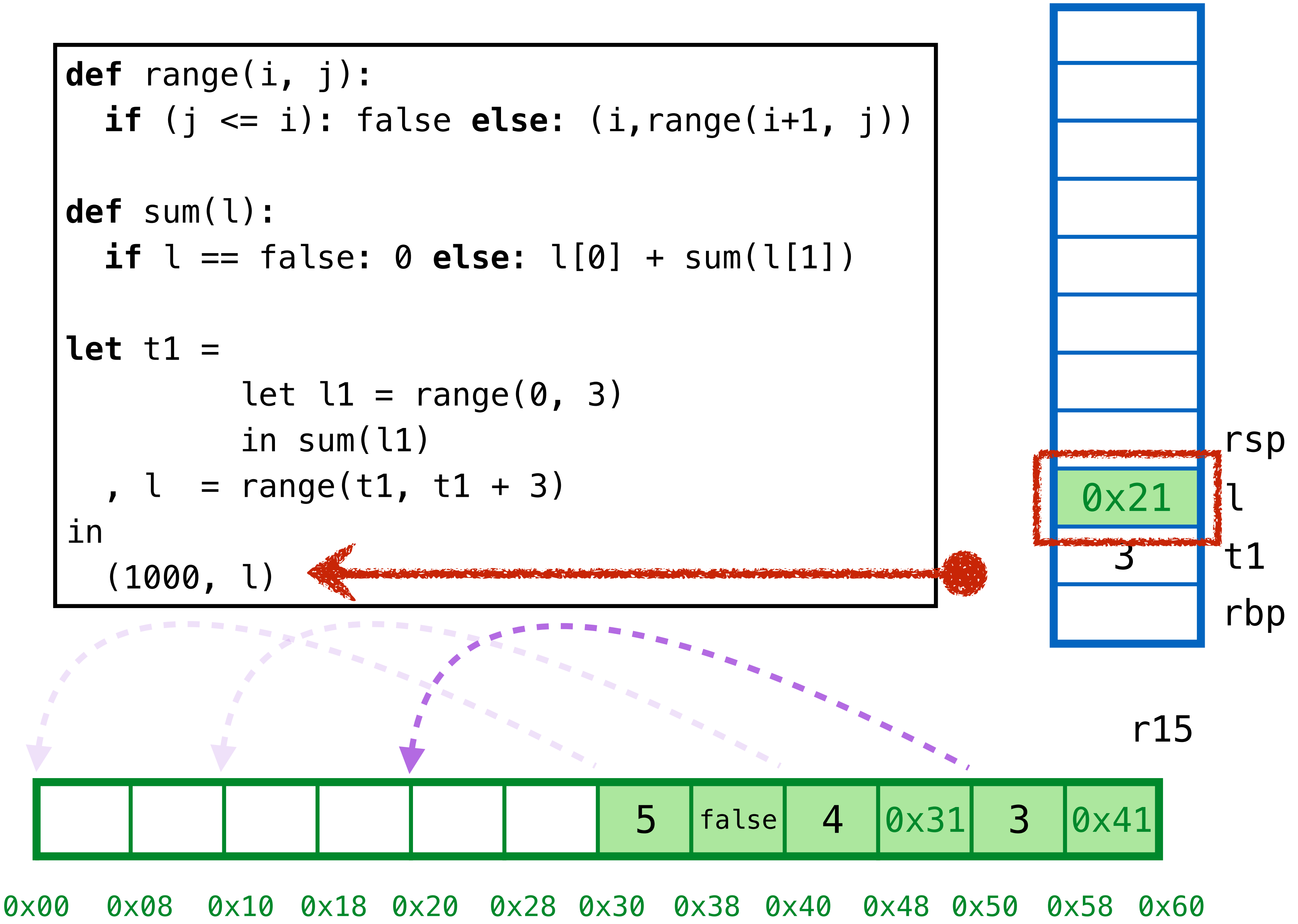
ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



3. REDIRECT addrs on stack

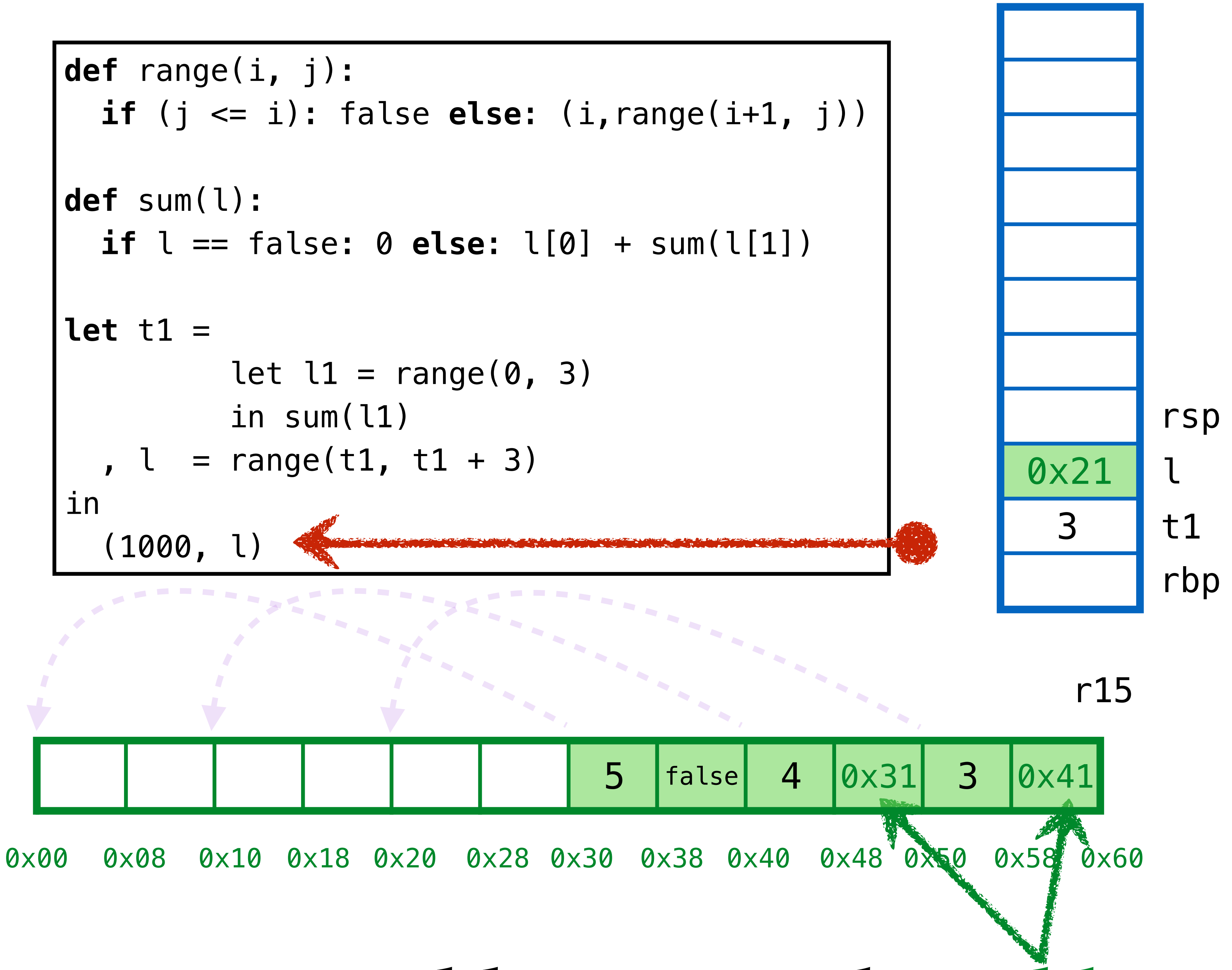
ex4: recursive data



```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

3. REDIRECT addrs on stack

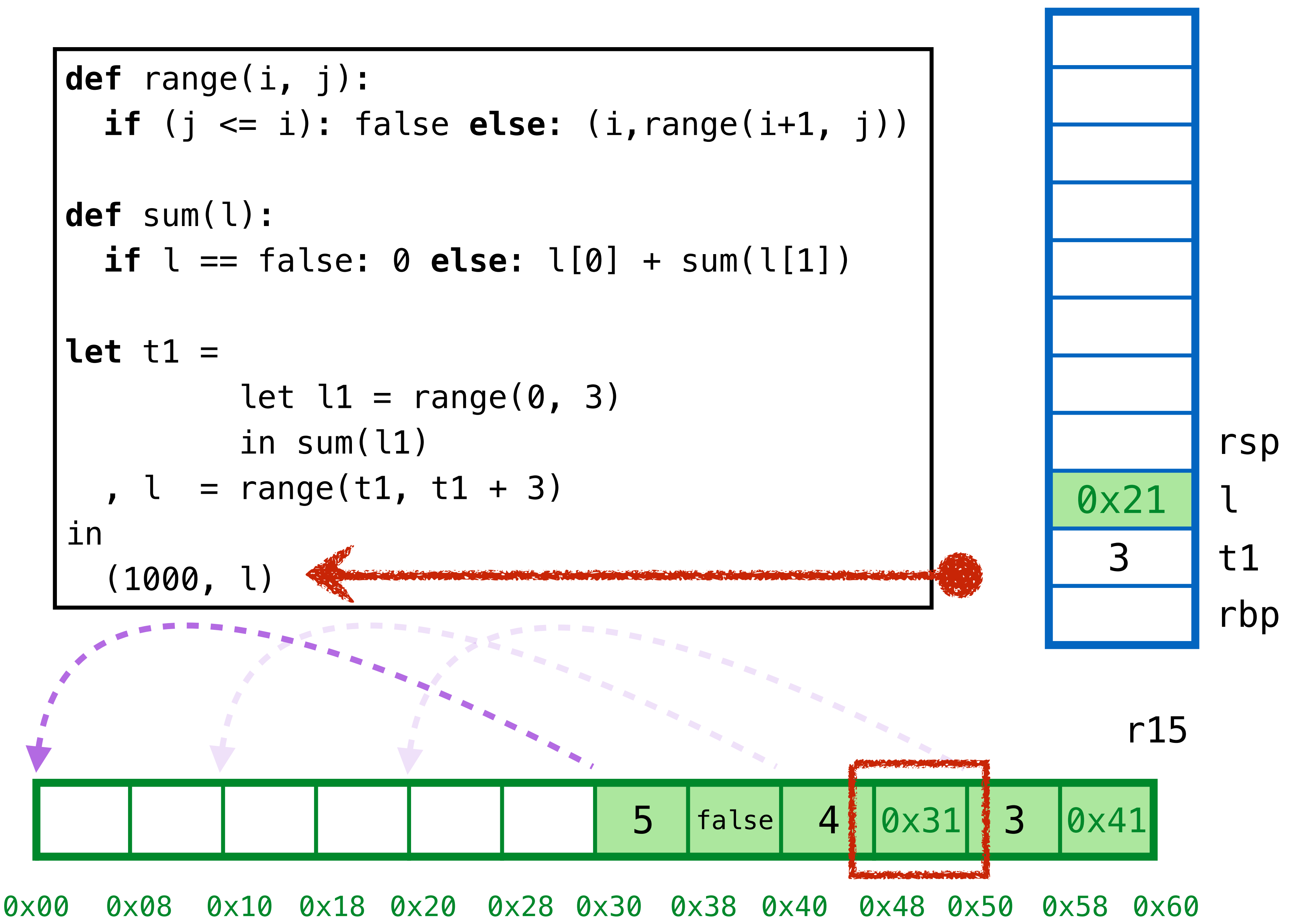
ex4: recursive data



```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

3. REDIRECT addrs on stack and heap!

ex4: recursive data

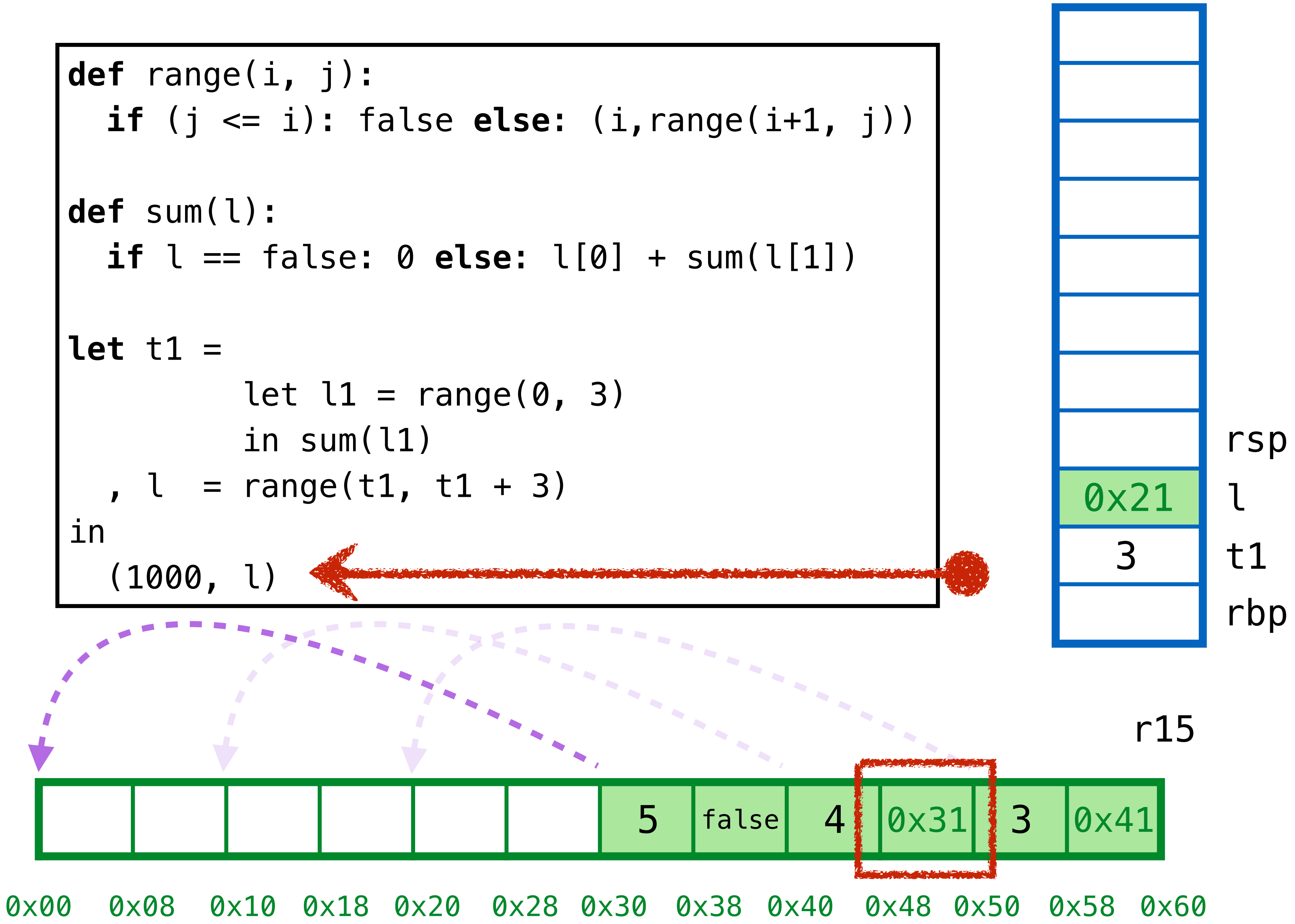


```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

3. REDIRECT addrs on stack and heap!

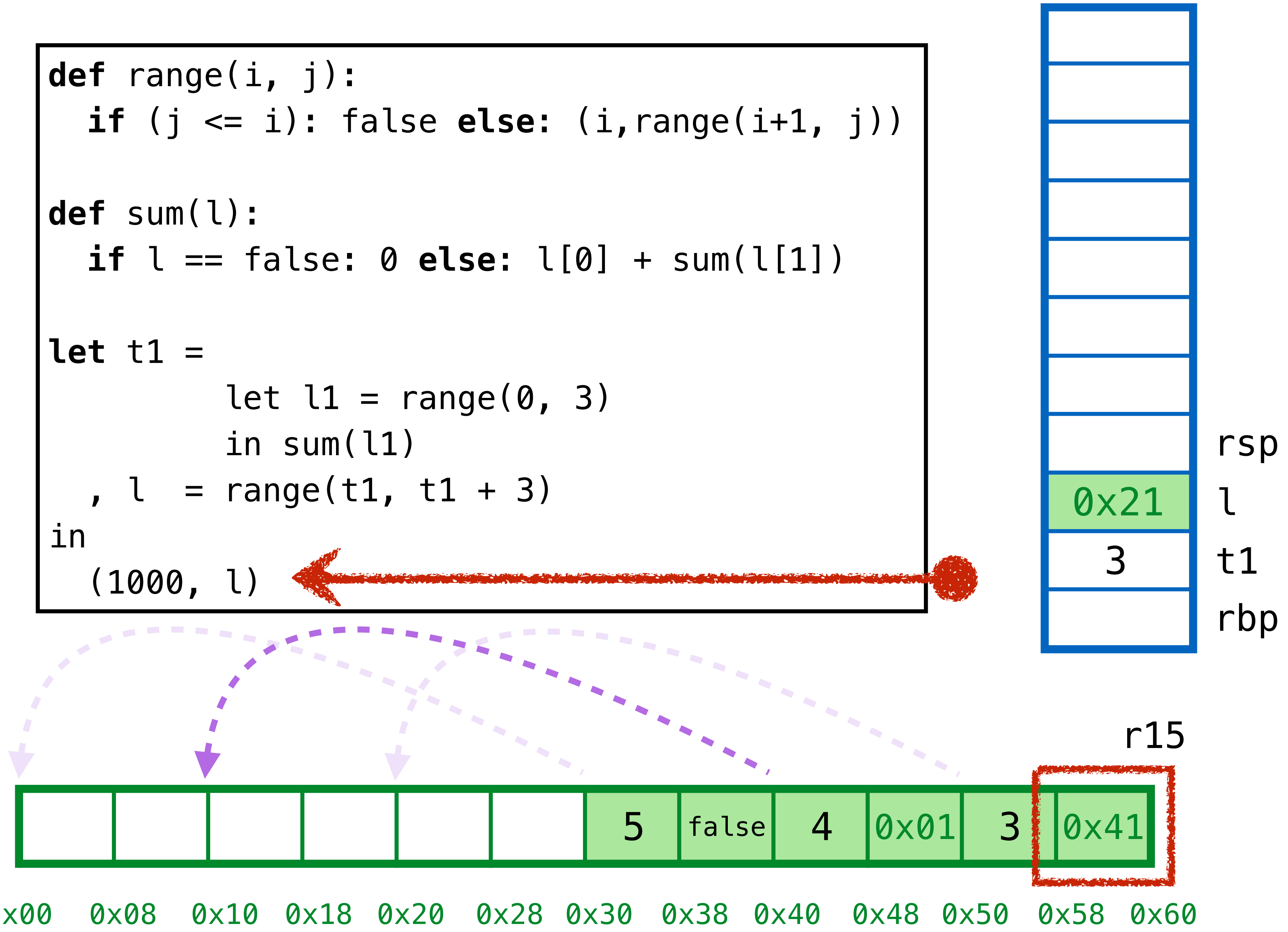
ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



3. REDIRECT addrs on stack and heap!

ex4: recursive data

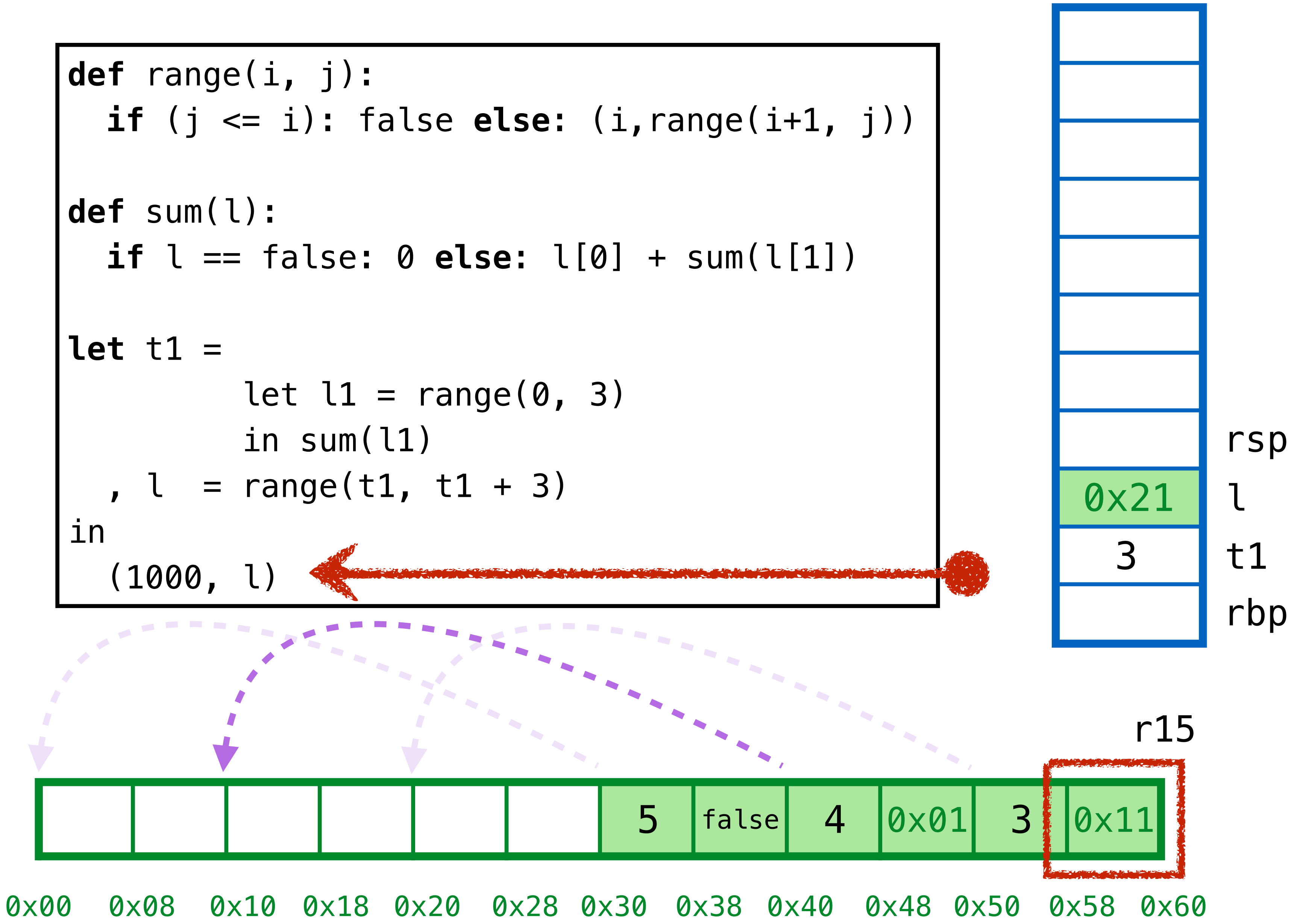


```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

3. REDIRECT addrs on stack and heap!

ex4: recursive data

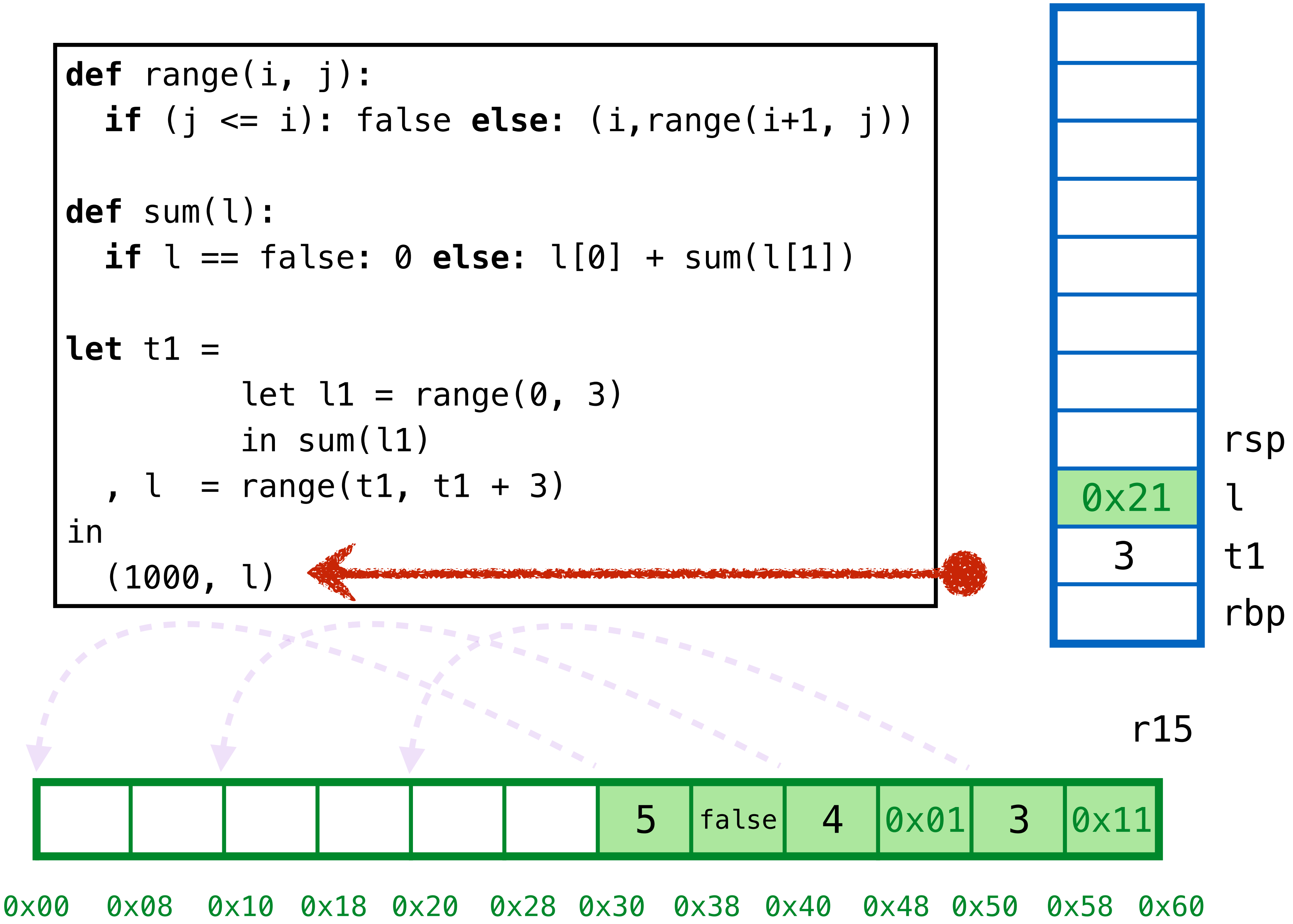
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



3. REDIRECT addrs on stack and heap!

ex4: recursive data

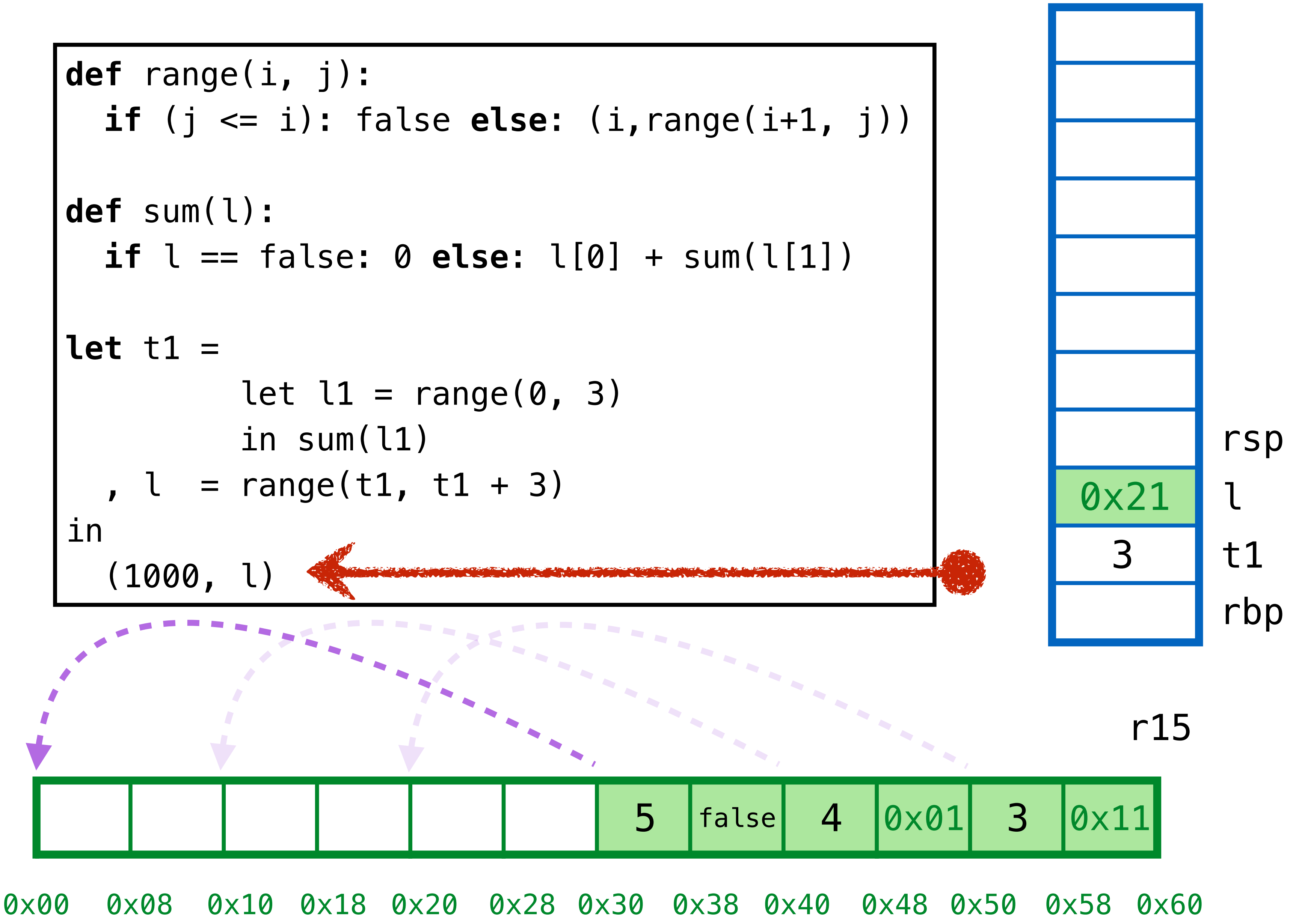
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



4. COMPACT cells on heap

Copy cell to forward addr!

ex4: recursive data

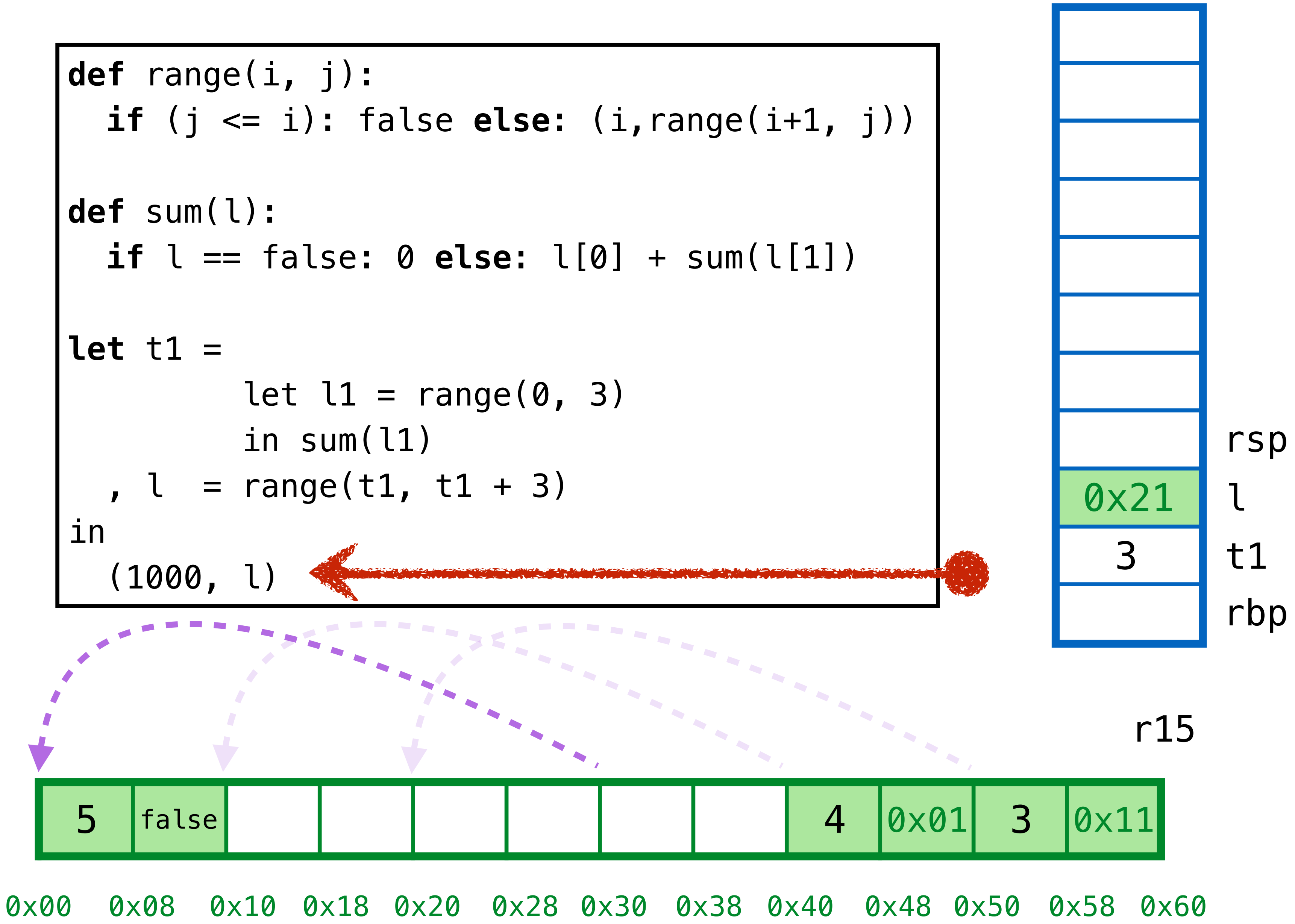


```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

4. COMPACT cells on heap

Copy cell to forward addr!

ex4: recursive data

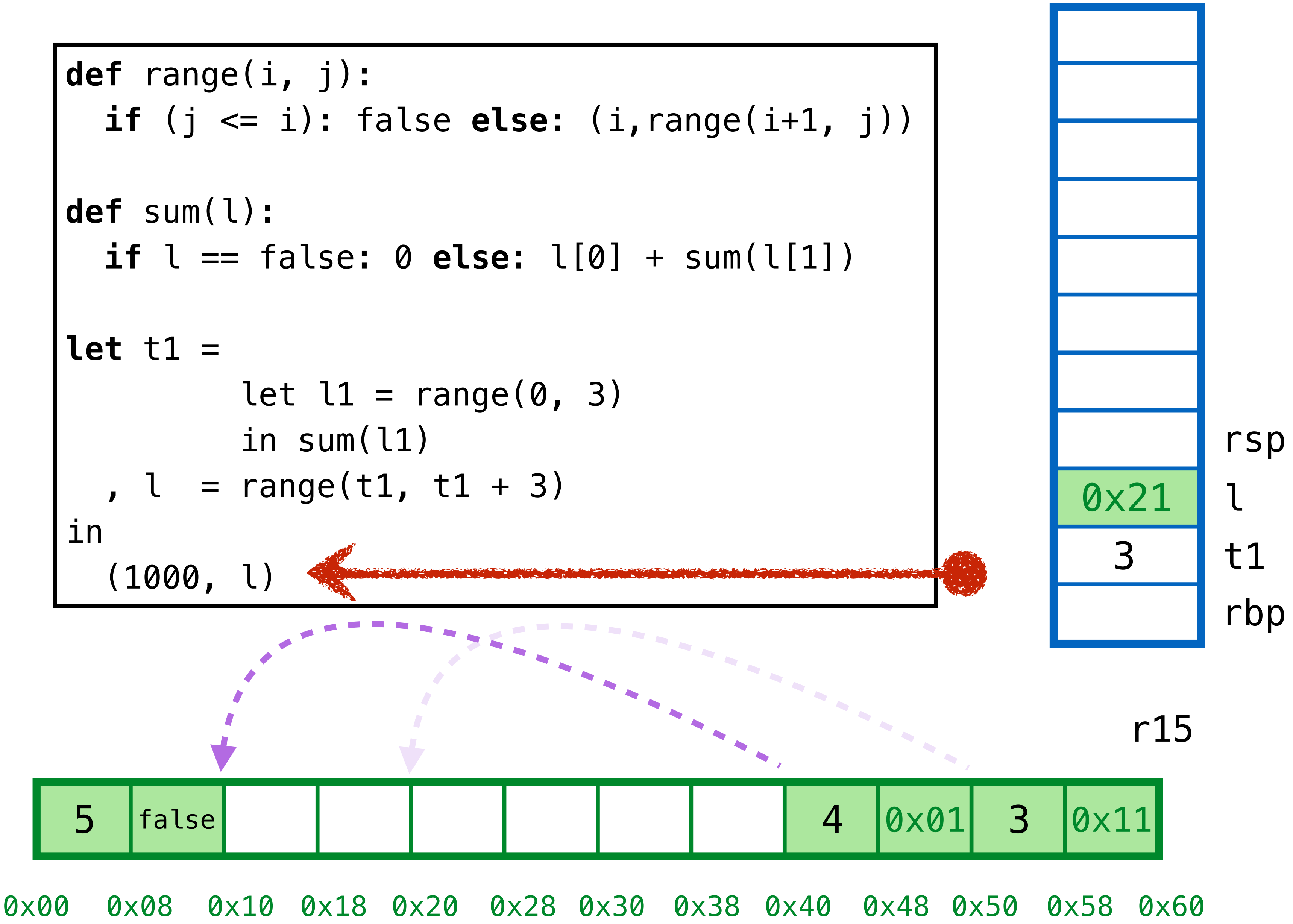


```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

4. COMPACT cells on heap

Copy cell to forward addr!

ex4: recursive data

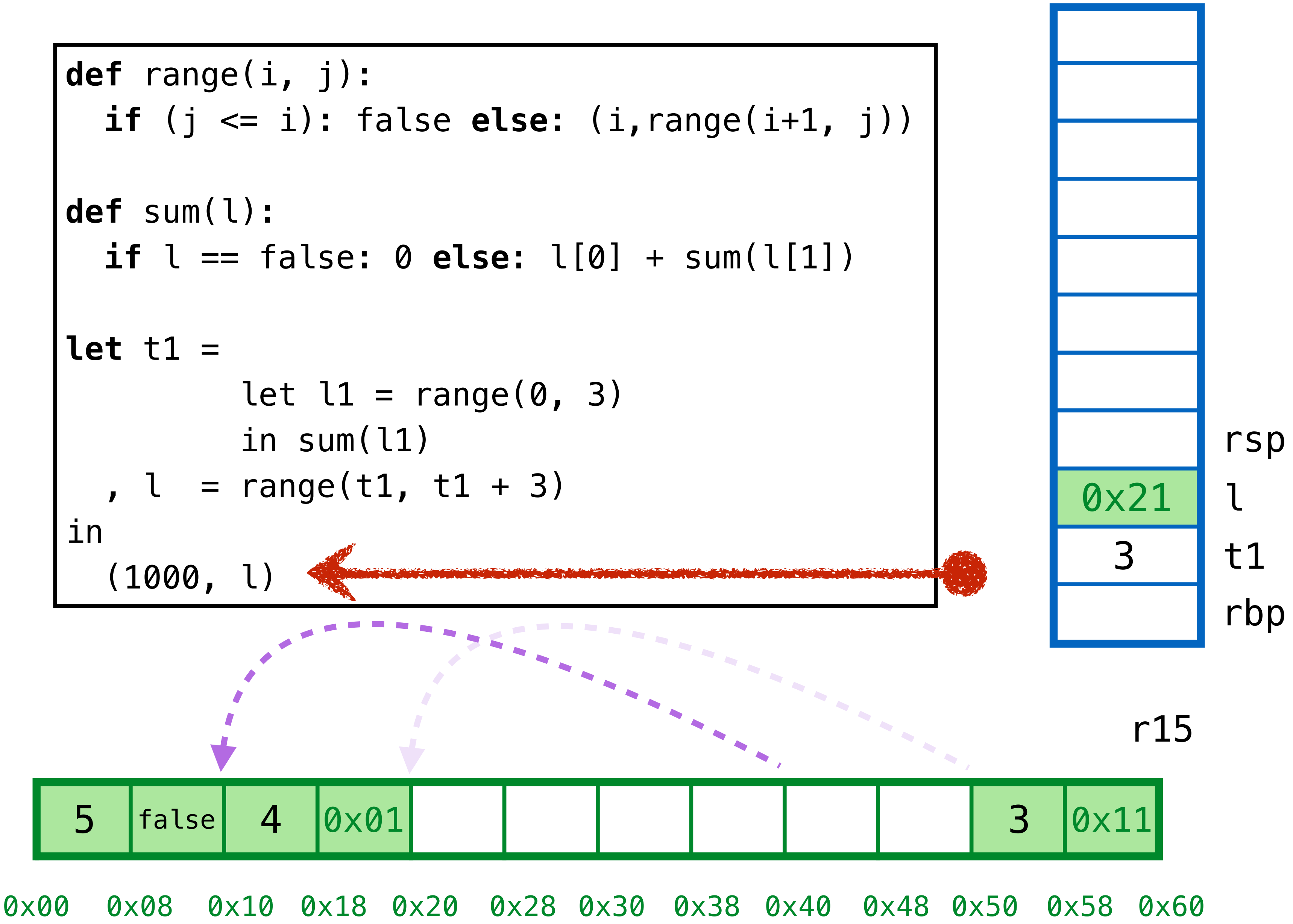


```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

4. COMPACT cells on heap

Copy cell to forward addr!

ex4: recursive data

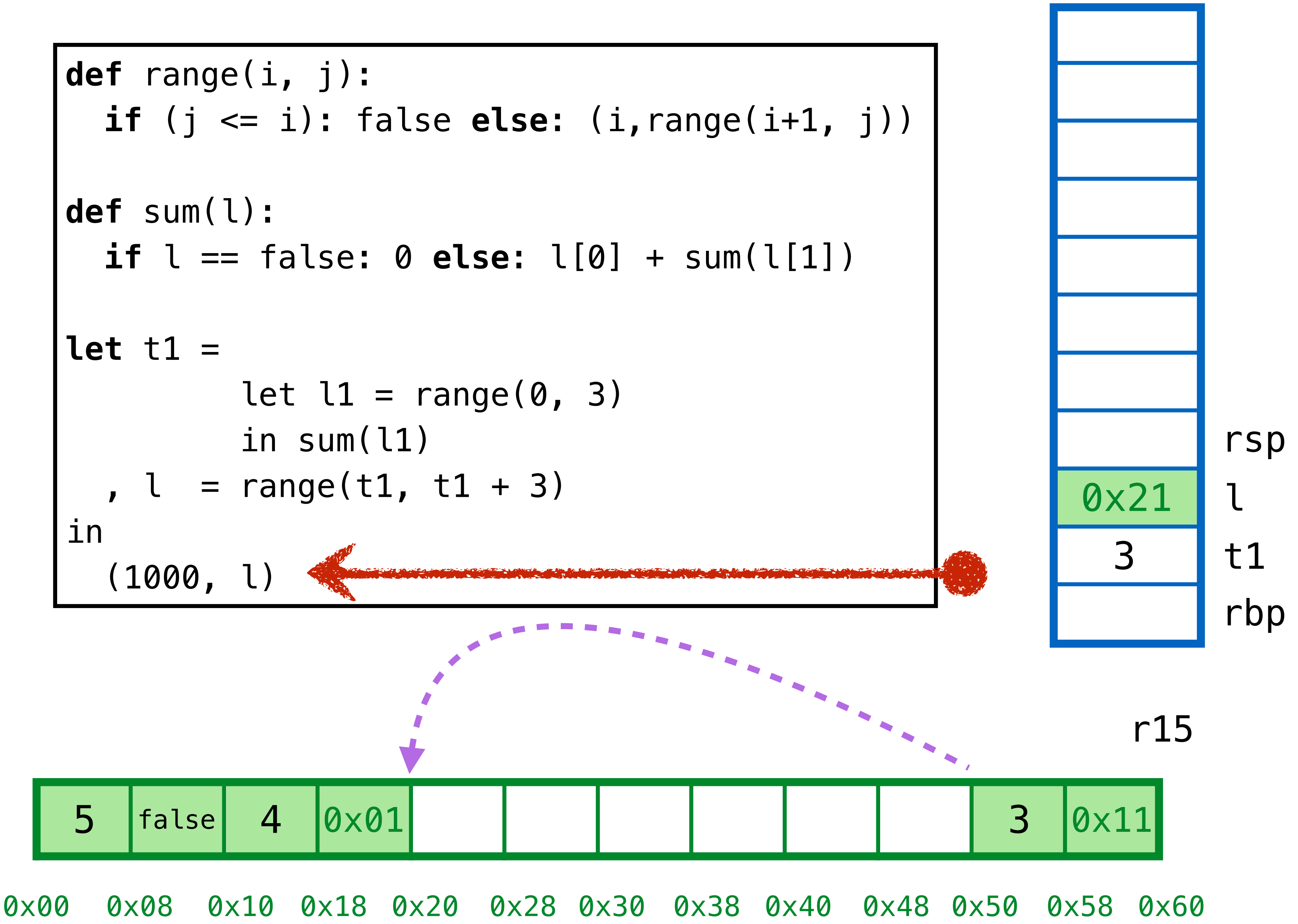


```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

4. COMPACT cells on heap

Copy cell to forward addr!

ex4: recursive data

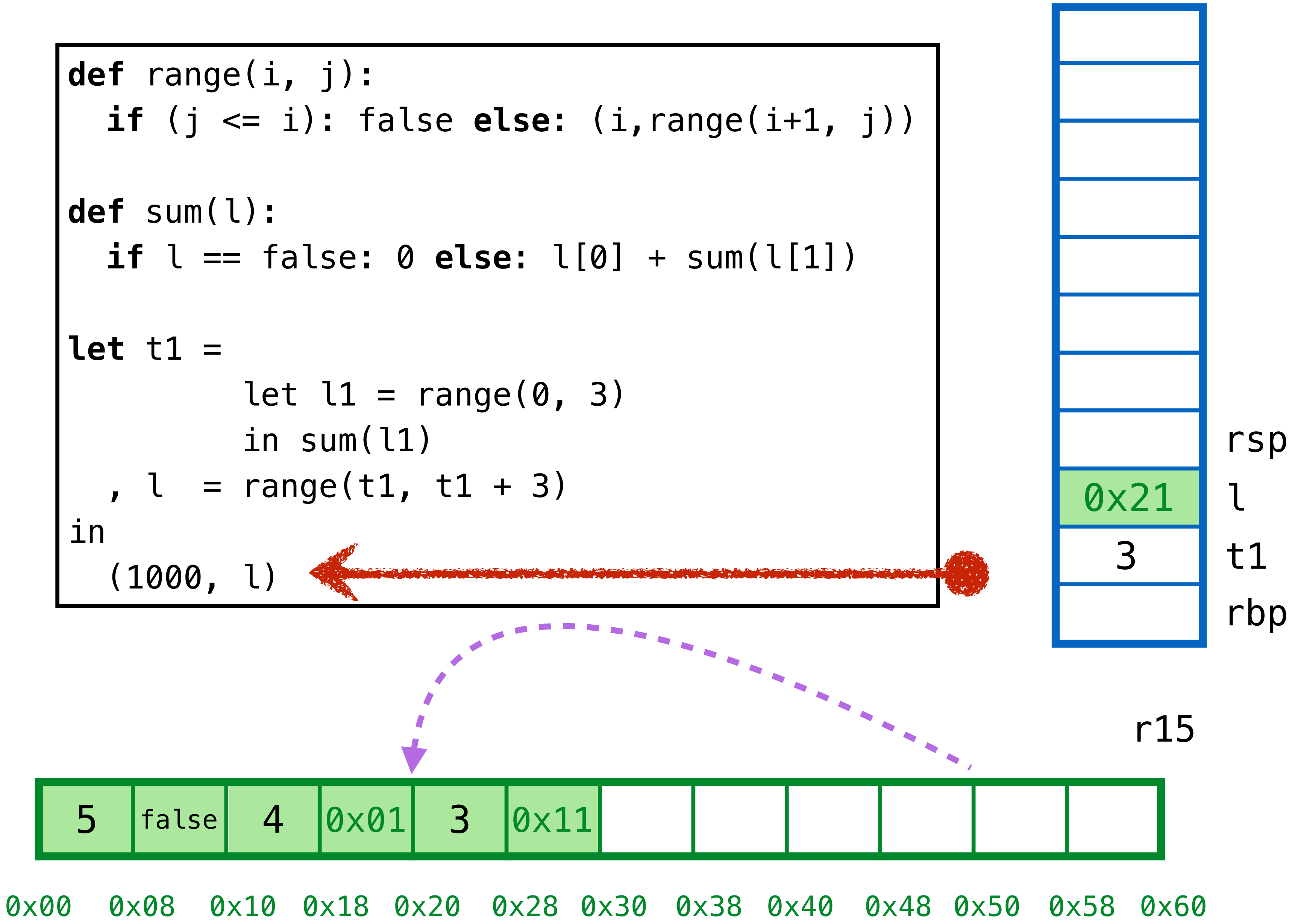


```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

4. COMPACT cells on heap

Copy cell to forward addr!

ex4: recursive data



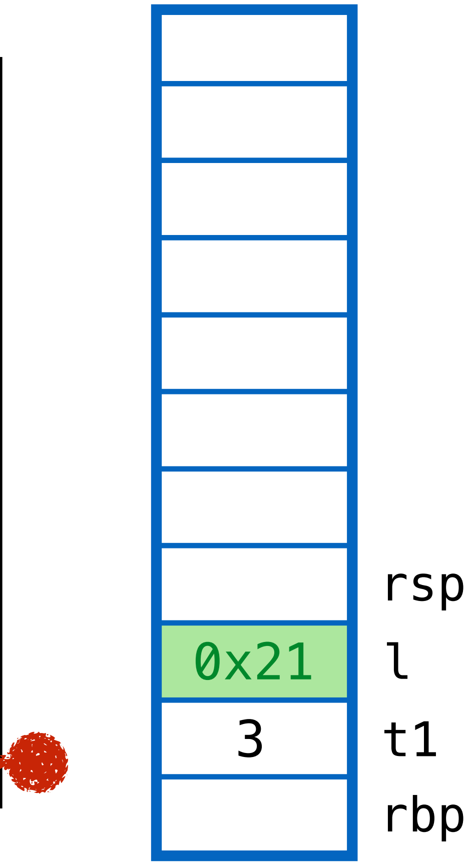
```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

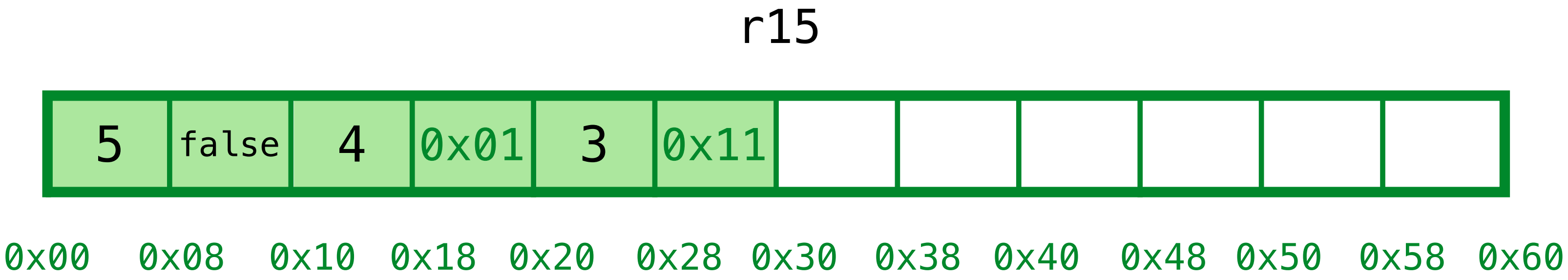
4. COMPACT cells on heap

Copy cell to forward addr!

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
          let l1 = range(0, 3)
          in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```



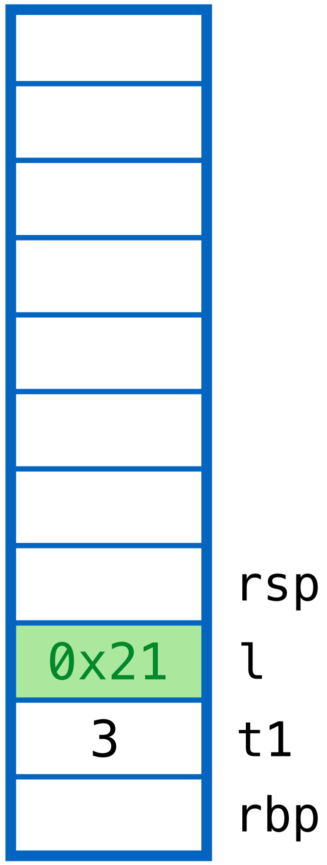


GC Complete!

Have space for `(1000, l)`

ex4: recursive data

```
def range(i, j):
  if (j <= i): false else: (i,range(i+1, j))

def sum(l):
  if l == false: 0 else: l[0] + sum(l[1])

let t1 =
         let l1 = range(0, 3)
         in sum(l1)
  , l  = range(t1, t1 + 3)
in
  (1000, l)
```

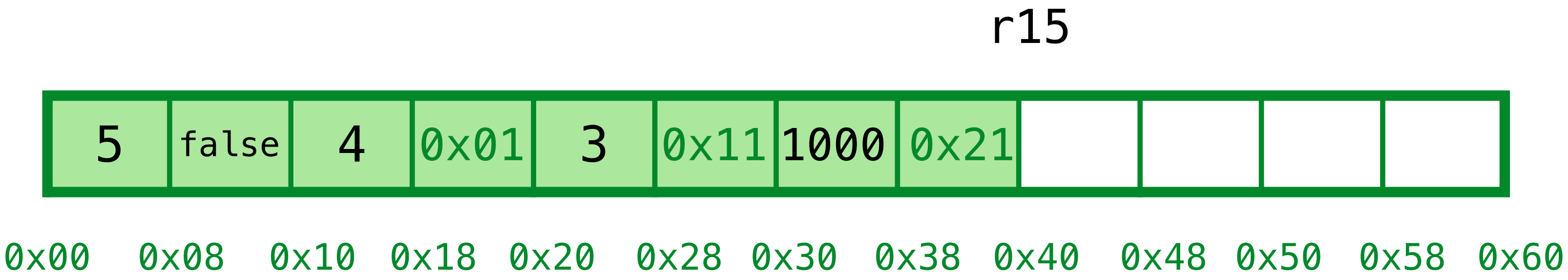


rsp

0x21 l

3 t1

rbp



r15

5	false	4	0x01	3	0x11	1000	0x21				

0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58 0x60

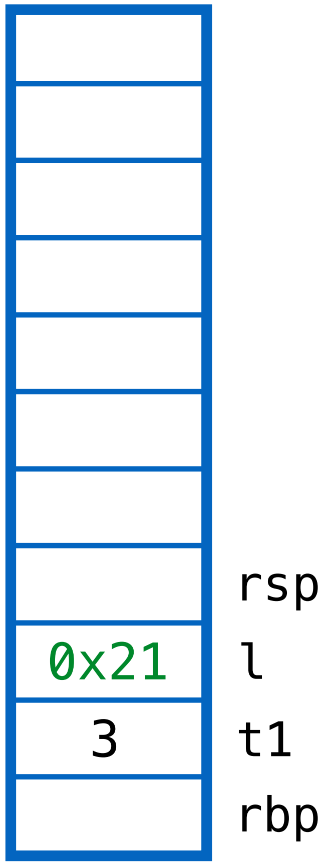
GC Complete!

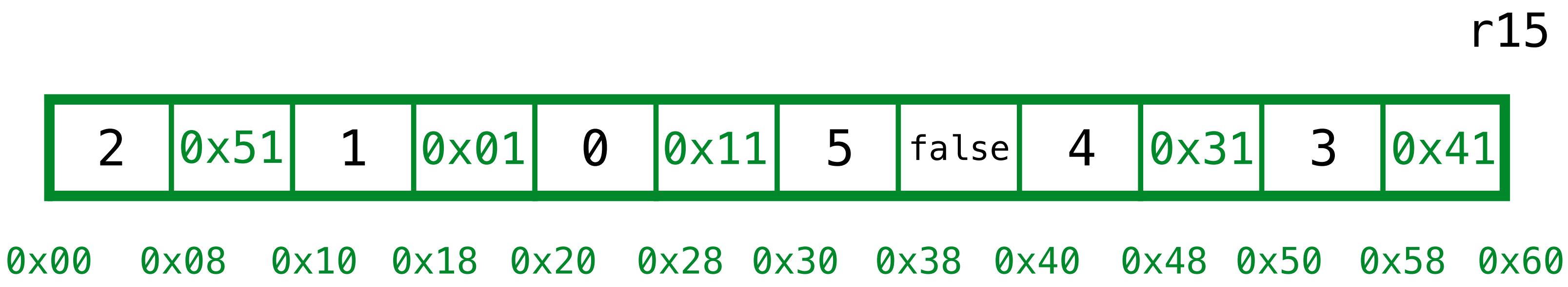
Have space for `(1000, l)`

ex4: recursive data

QUIZ: What should `print(0x21)` show?

```
(A) (0, (1, (2, false)))
(B) (3, (4, (5, false)))
(C) (0, (1, (2, (3, (4, (5, false))))))
(D) (3, (4, (5, (0, (1, (2, false))))))
(E) (2, (1, (0, (3, (4, (5, false))))))
```





ex4: recursive data

QUIZ: Which cells are "live" on the heap?

(A) 0x00

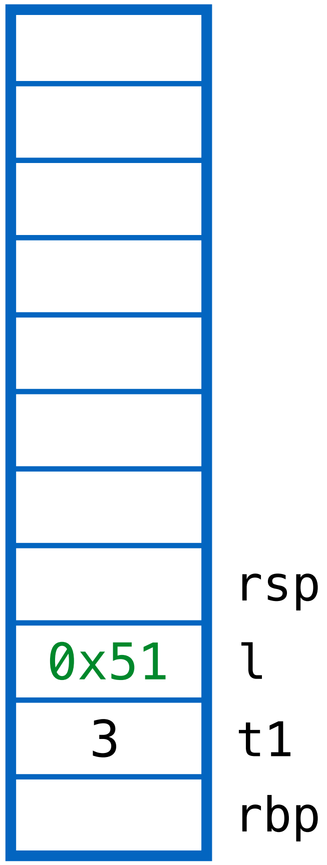
(B) 0x10

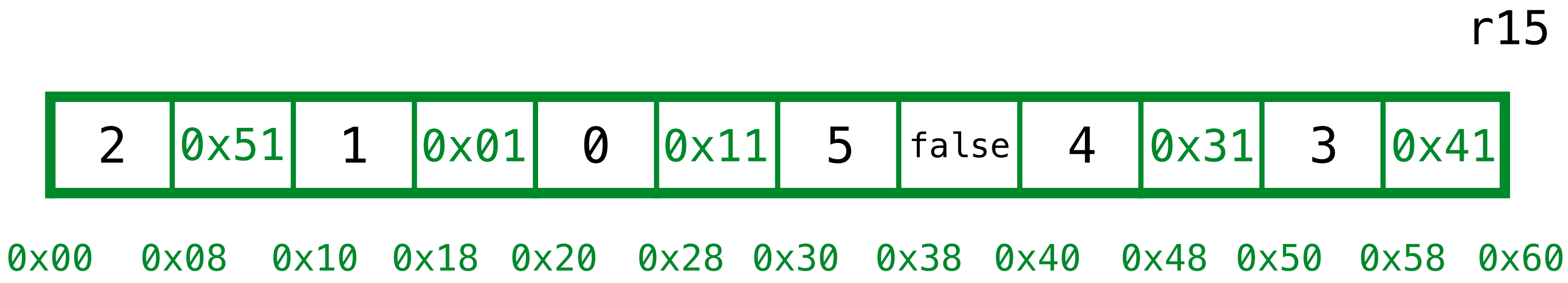
(C) 0x20

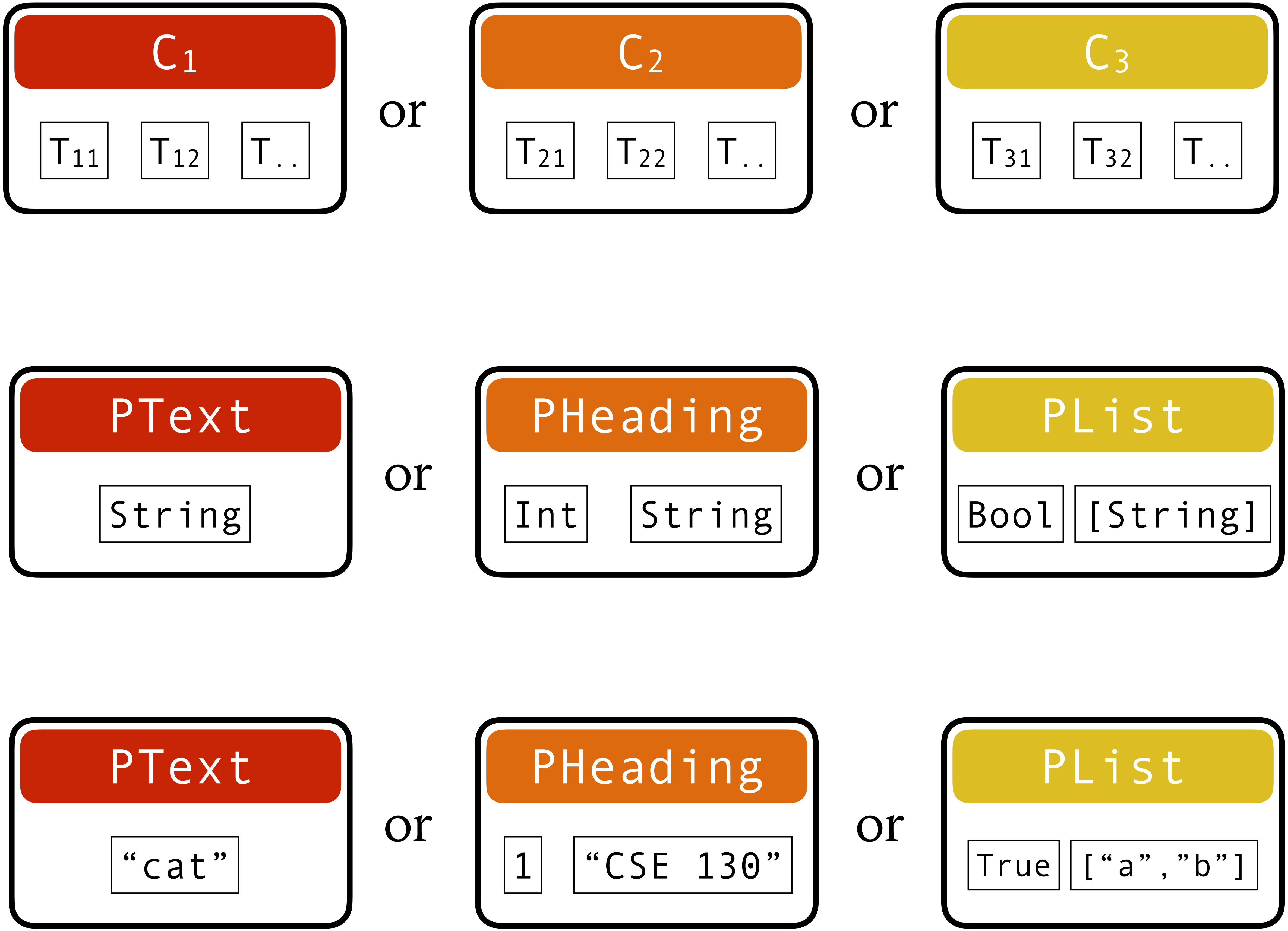
(D) 0x30

(E) 0x40

(F) 0x50



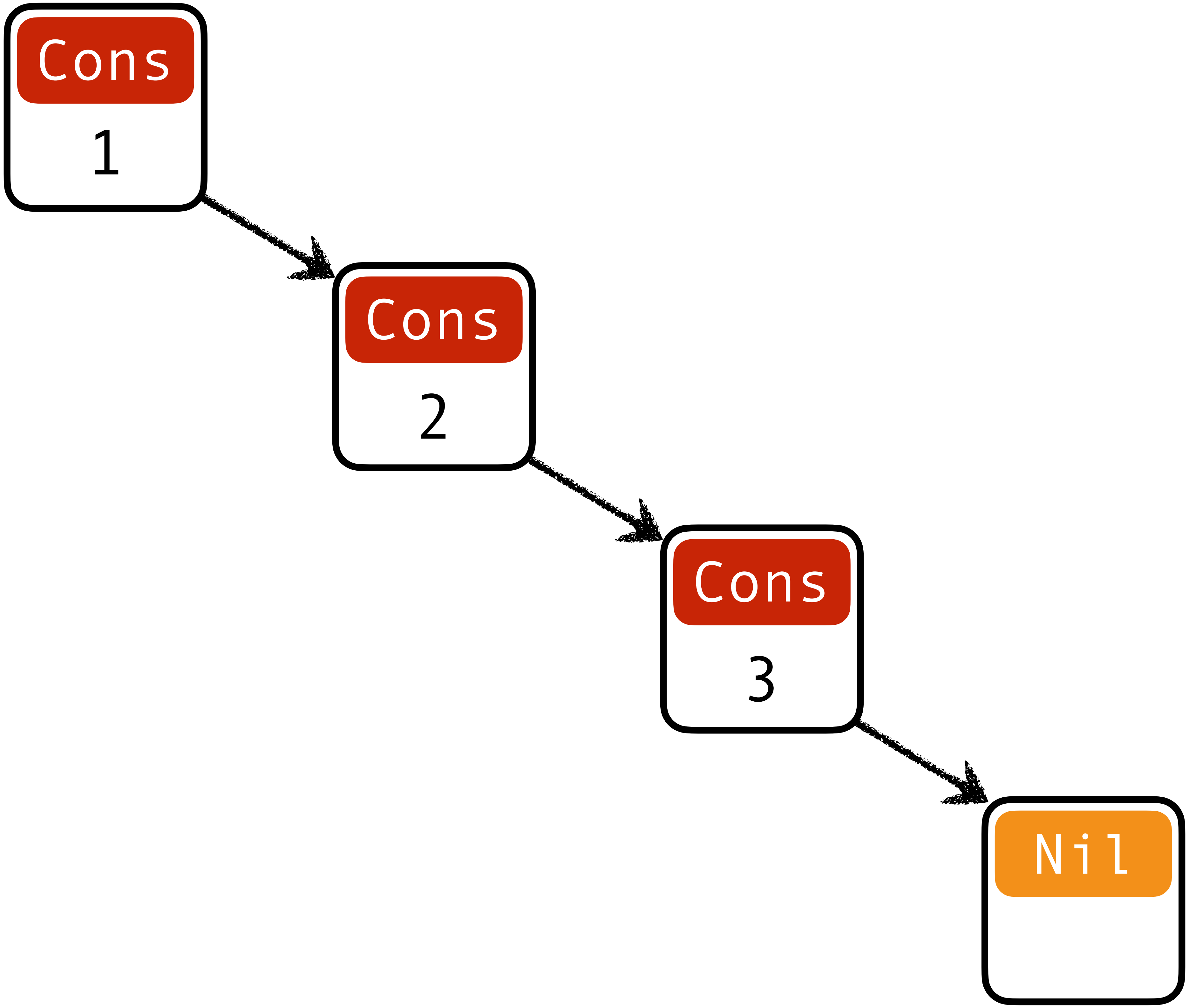




C_1 T_{11} T_{12} T.. or C_2 T_{21} T_{22} T.. or C_3 T_{31} T_{32} T..

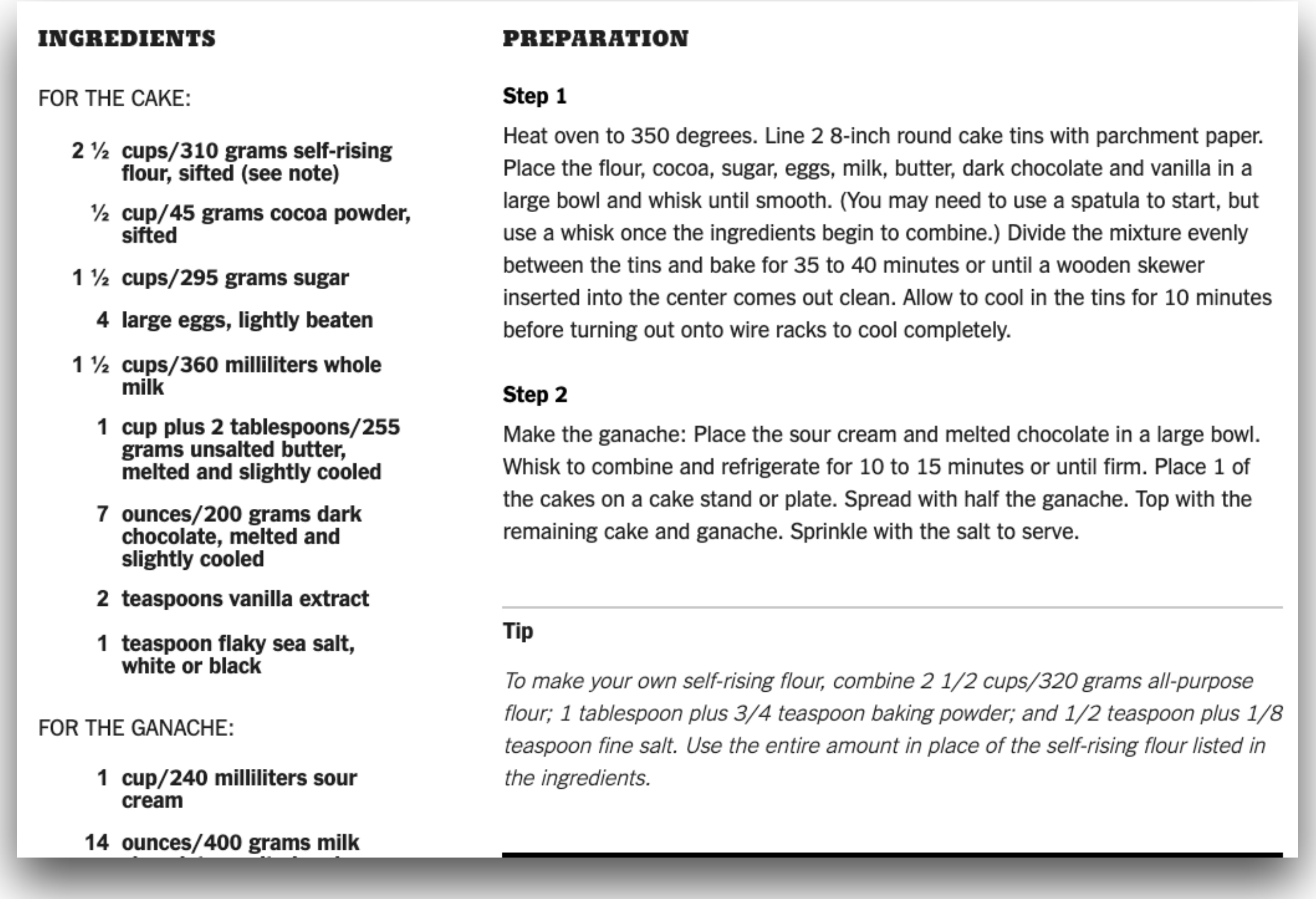
PText String or PHeading Int String or PList Bool [String]

PText “cat” or PHeading 1 “CSE 130” or PList True [“a”,”b”]



Cons
1
Cons
2
Cons
3
Nil

vs.



INGREDIENTS

FOR THE CAKE:

2 ½	cups/310 grams self-rising flour, sifted (see note)
½	cup/45 grams cocoa powder, sifted
1 ½	cups/295 grams sugar
4	large eggs, lightly beaten
1 ½	cups/360 milliliters whole milk
1	cup plus 2 tablespoons/255 grams unsalted butter, melted and slightly cooled
7	ounces/200 grams dark chocolate, melted and slightly cooled
2	teaspoons vanilla extract
1	teaspoon flaky sea salt, white or black

FOR THE GANACHE:

1	cup/240 milliliters sour cream
14	ounces/400 grams milk

PREPARATION

Step 1

Heat oven to 350 degrees. Line 2 8-inch round cake tins with parchment paper. Place the flour, cocoa, sugar, eggs, milk, butter, dark chocolate and vanilla in a large bowl and whisk until smooth. (You may need to use a spatula to start, but use a whisk once the ingredients begin to combine.) Divide the mixture evenly between the tins and bake for 35 to 40 minutes or until a wooden skewer inserted into the center comes out clean. Allow to cool in the tins for 10 minutes before turning out onto wire racks to cool completely.

Step 2

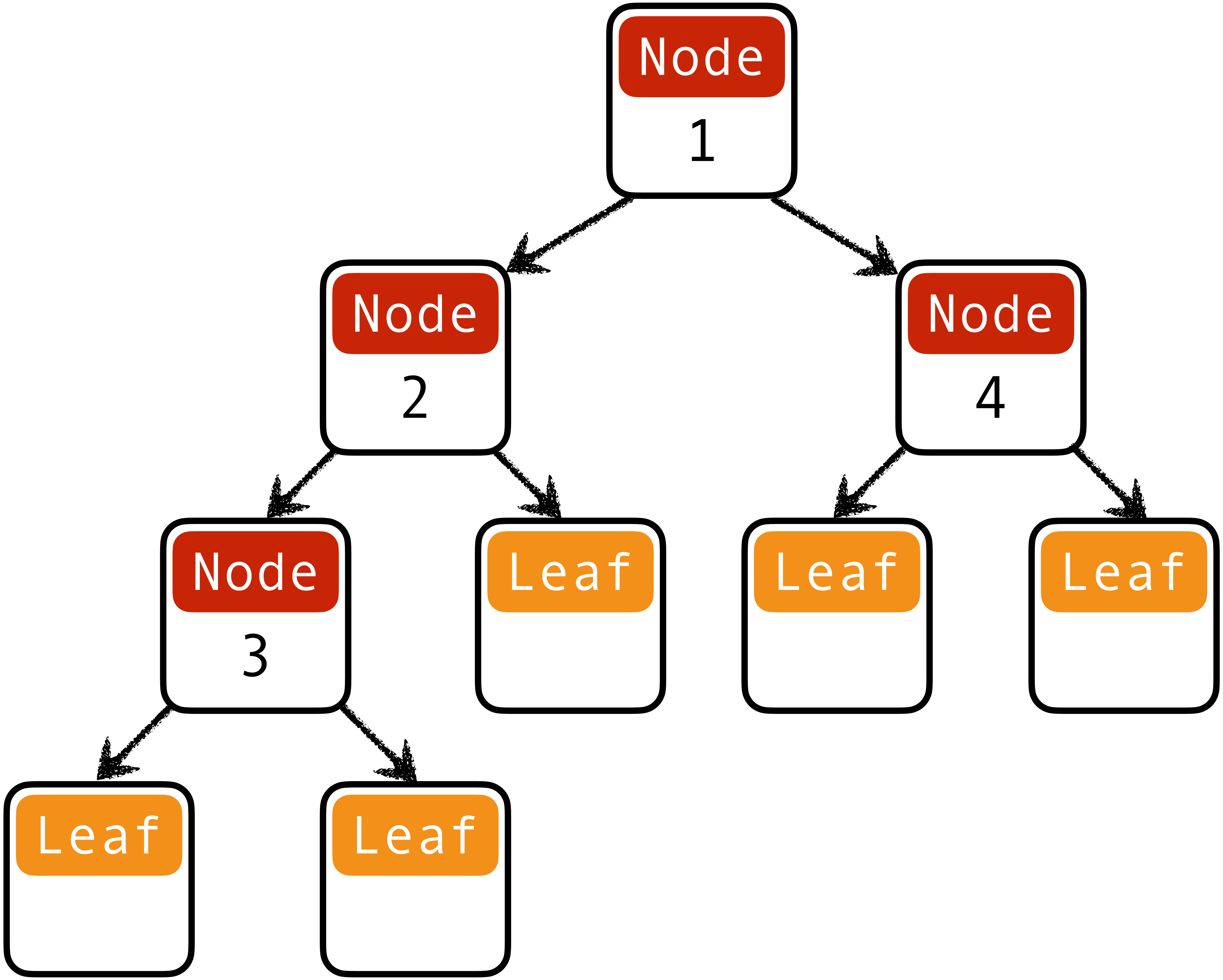
Make the ganache: Place the sour cream and melted chocolate in a large bowl. Whisk to combine and refrigerate for 10 to 15 minutes or until firm. Place 1 of the cakes on a cake stand or plate. Spread with half the ganache. Top with the remaining cake and ganache. Sprinkle with the salt to serve.

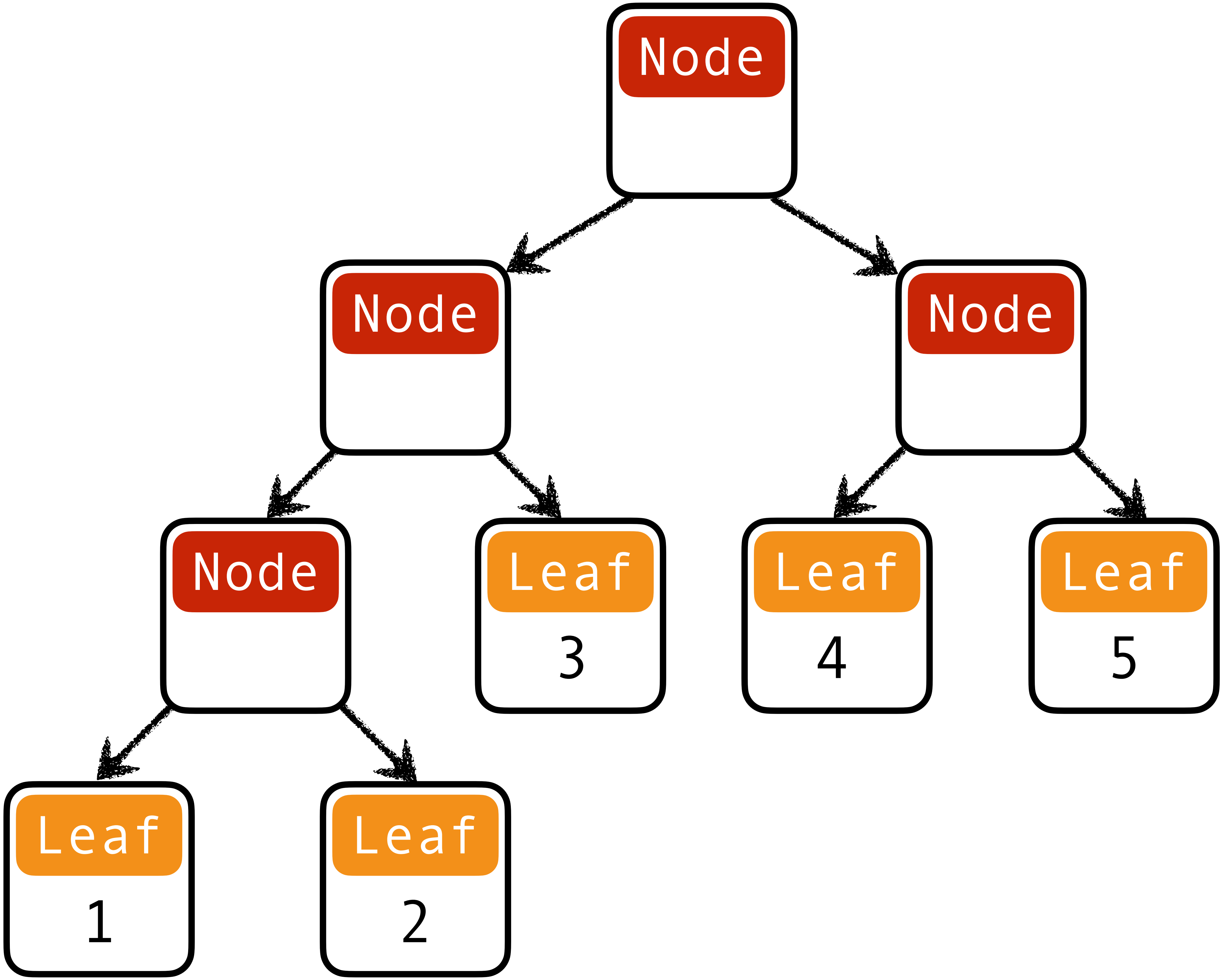
Tip

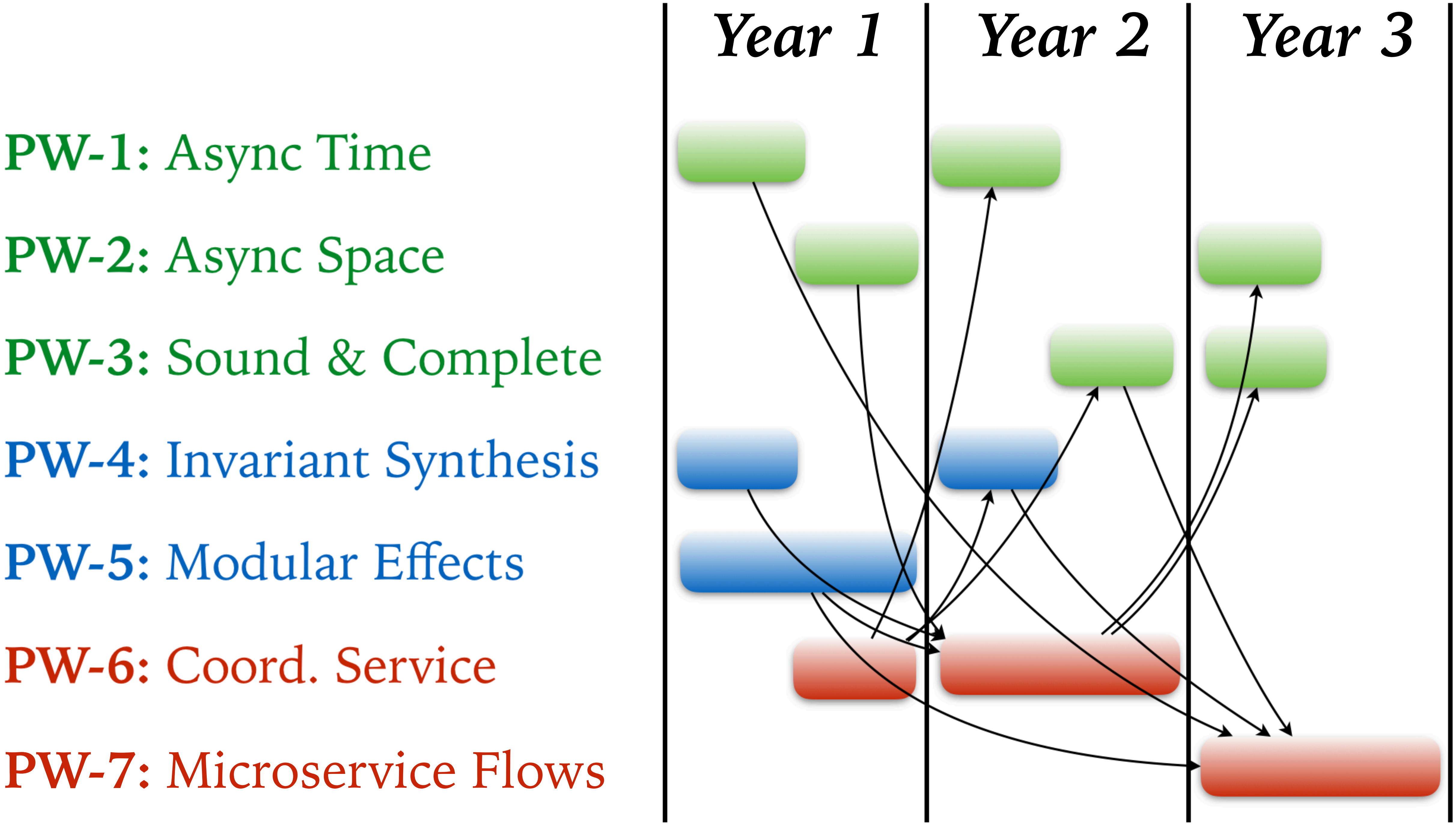
To make your own self-rising flour, combine 2 1/2 cups/320 grams all-purpose flour; 1 tablespoon plus 3/4 teaspoon baking powder; and 1/2 teaspoon plus 1/8 teaspoon fine salt. Use the entire amount in place of the self-rising flour listed in the ingredients.

saltDarkChoco :: Cake

howToSaltChoco :: Recipe Cake


Node
1
Node
2
Node
4
Node
3
Leaf
Leaf
Leaf
Leaf
Leaf


Node
Node
Node
Node
Leaf
3
Leaf
4
Leaf
5
Leaf
1
Leaf
2



Year 1 *Year 2* *Year 3*

PW-1: Async Time

PW-2: Async Space

PW-3: Sound & Complete

PW-4: Invariant Synthesis

PW-5: Modular Effects

PW-6: Coord. Service

PW-7: Microservice Flows

```
let rec wwhile (f, b) =
  let (b', c') = f b in
  if c' = true then wwhile (f b')
               else b'
```

RITE: (f, b')

SEMINAL: ((f b'); [[...]])

```
let rec clone x n =
  if n <= 0 then [] else
    x :: clone (n-1)
```

RITE: clone (n-1) n

SEMINAL: clone [[...]] (n-1)

```
let sqsum xs =
  let f a x = a + (x ** 2) in
  let base = 0             in
  List.fold_left f base xs
```

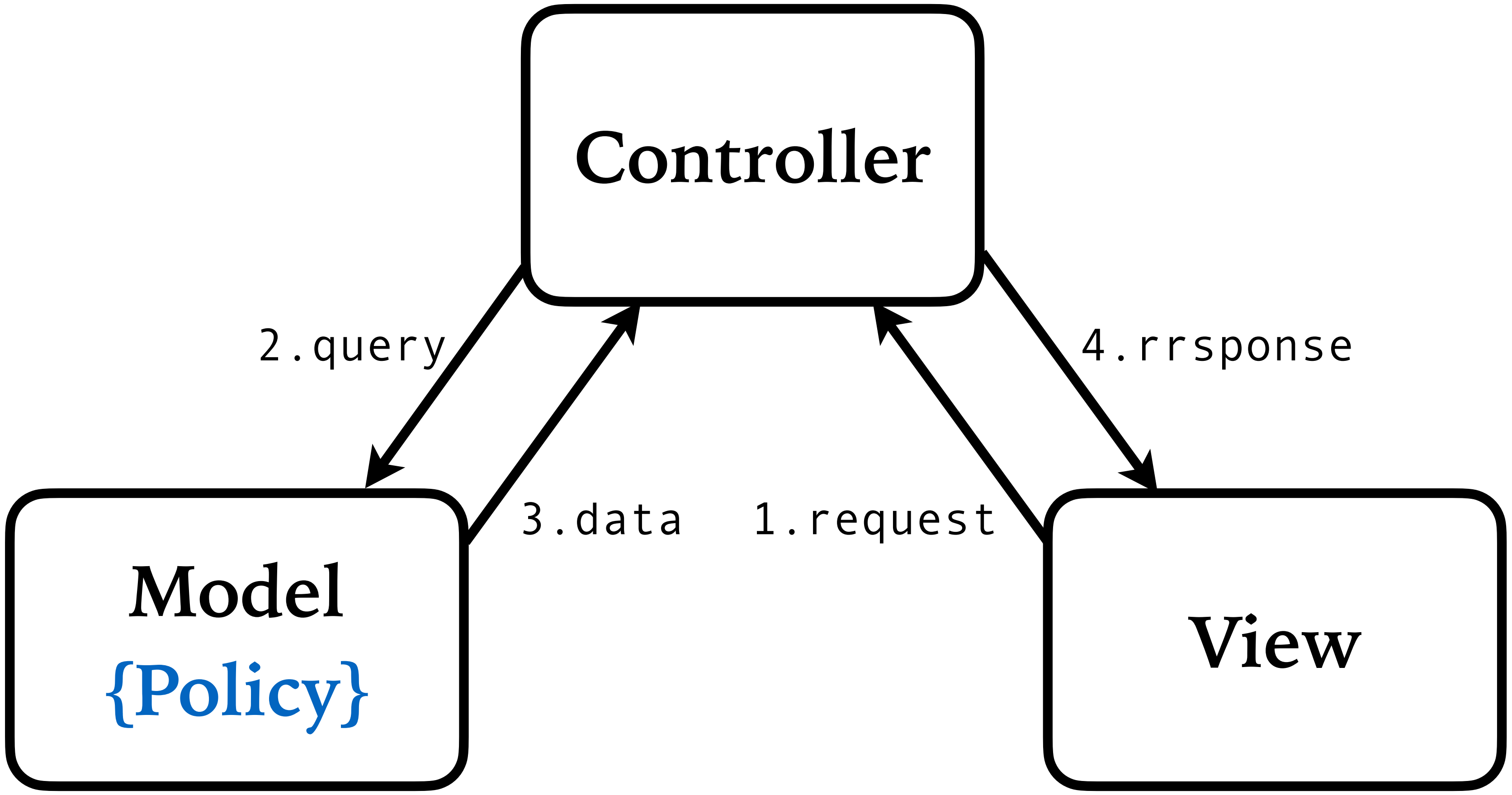
RITE: (x * x)

SEMINAL: (x + 2)

```
let rec clone x n =
  if n <= 0 then [] else
    x :: clone (n-1)
```

RITE: clone (n-1) n

SEMINAL: clone [[...]] (n-1)



Controller

2.query

3.data

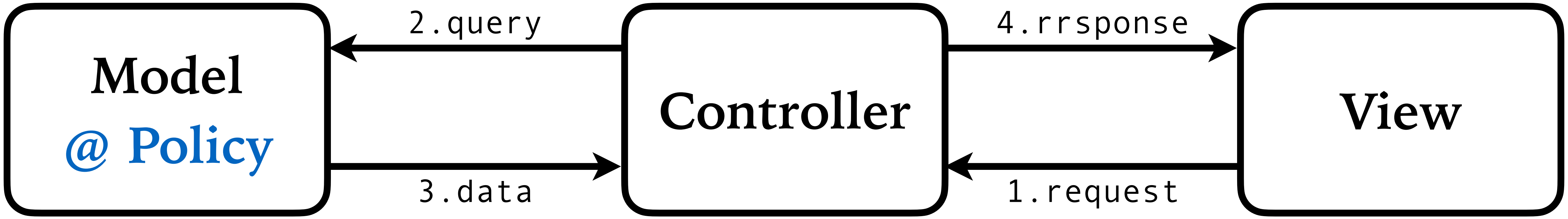
1.request

4.rrsponse

Model

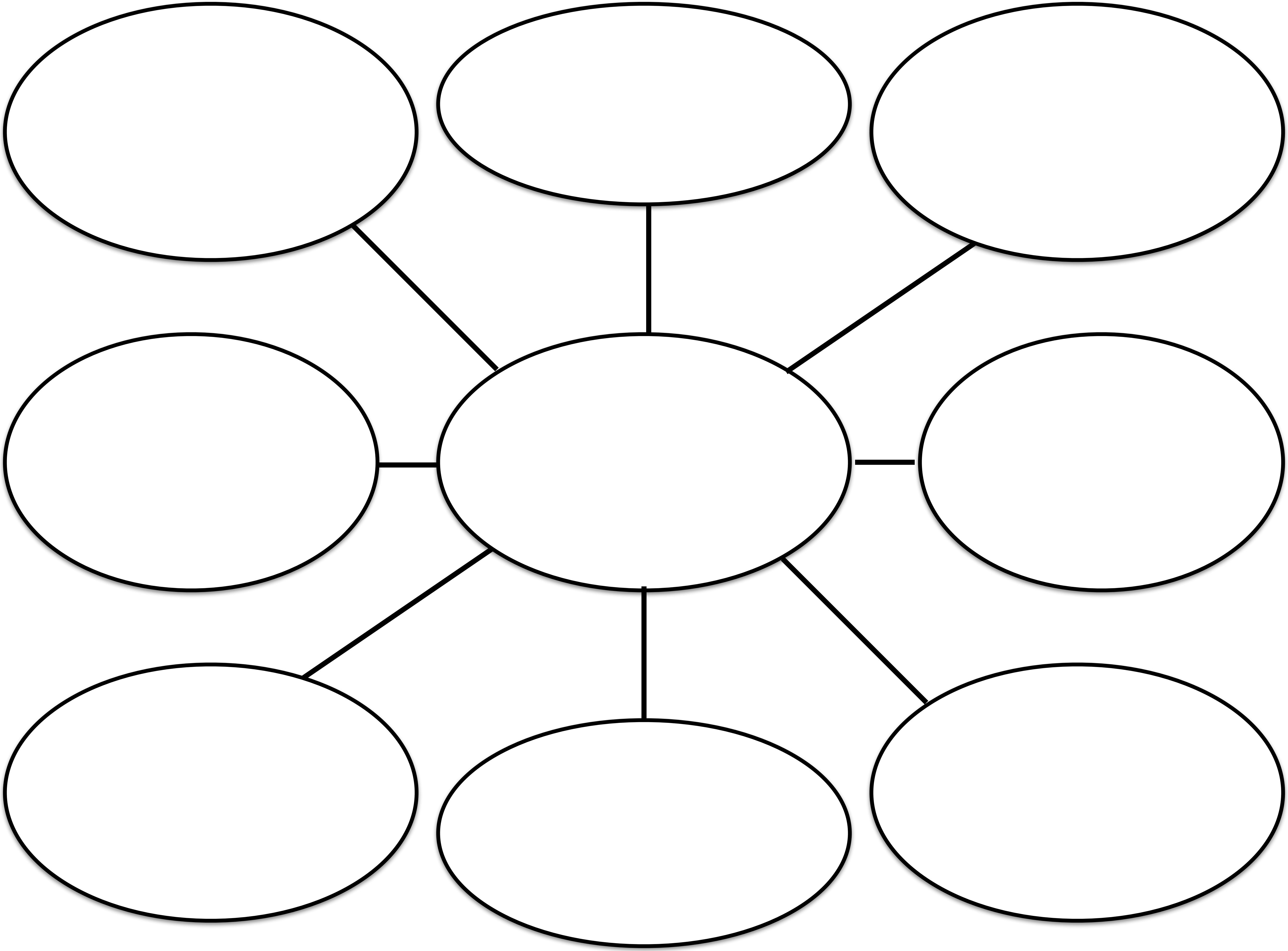
{Policy}

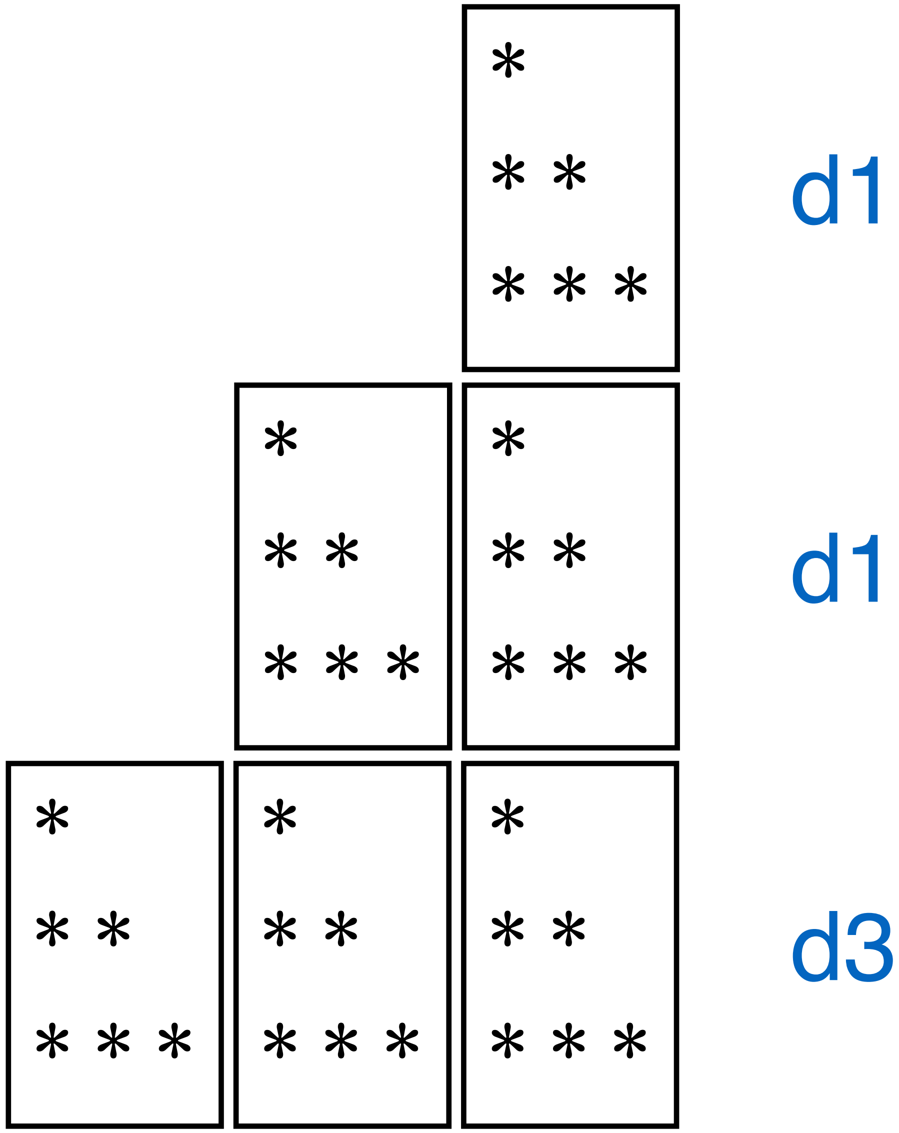
View

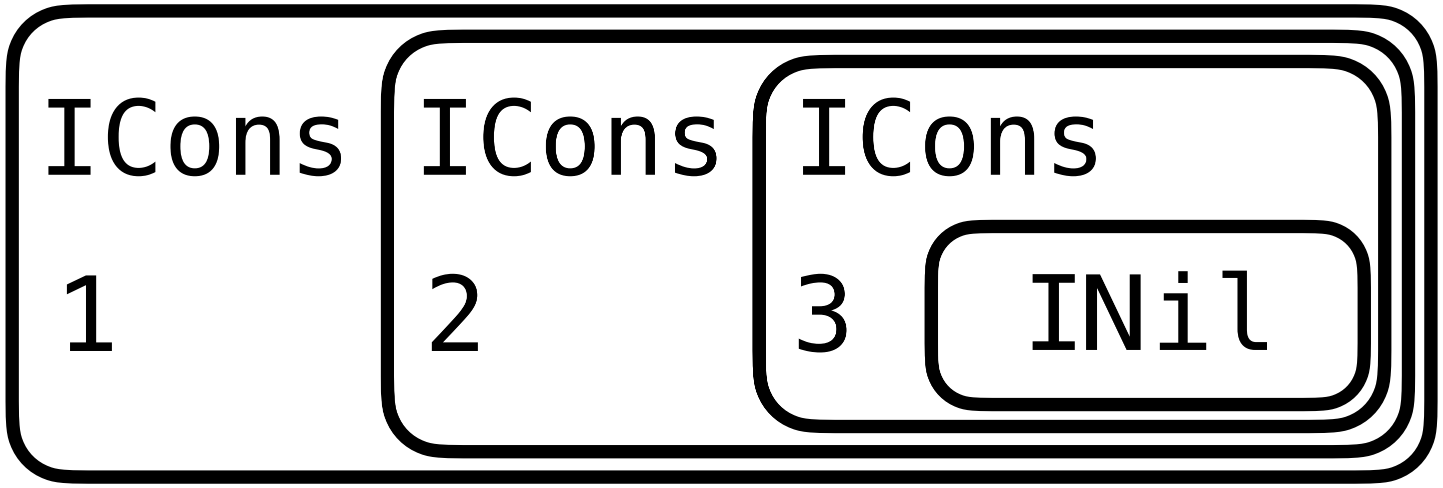

Model
@ Policy
2.query
3.data
Controller
4.rrsponse
1.request
View

The question then is, is it possible to use (overapproximate) static analyses to precisely report that the target location is reachable, *without* actually finding a feasible path to it? Intuitively, the code through the for-loop is irrelevant to the reachability of the error location. In other words, if we can reason that *there exists some path* from the start to the end of the loop, *i.e.,* from location `3:` to `5:`, and along such a path, the variables `x`, `a` are not modified, then we are guaranteed that the location `ERR:` can be reached.

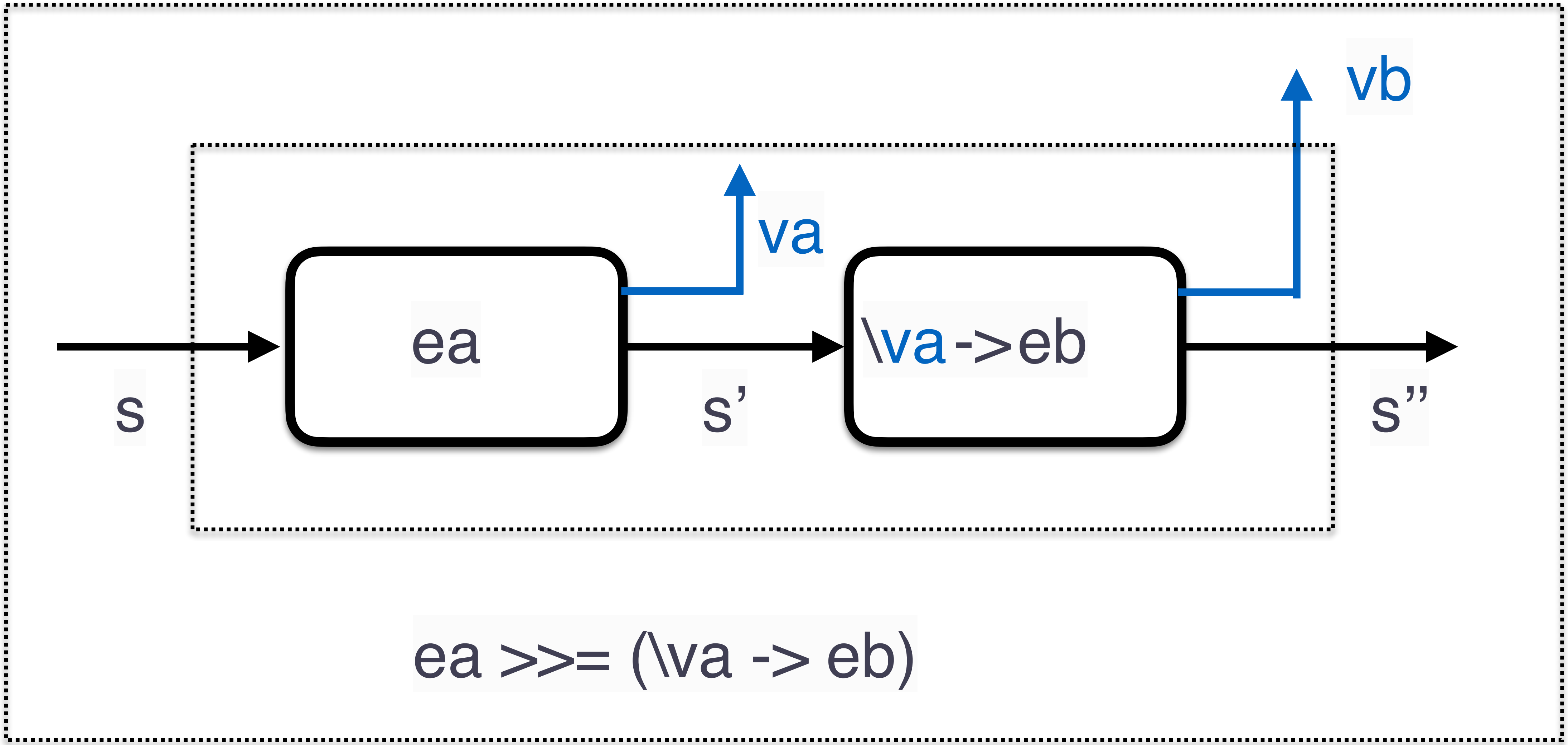
30 Lecture (4/3)




d1
d1
d3



ICons
1
ICons
2
ICons
3
INil



s

ea

s’

va

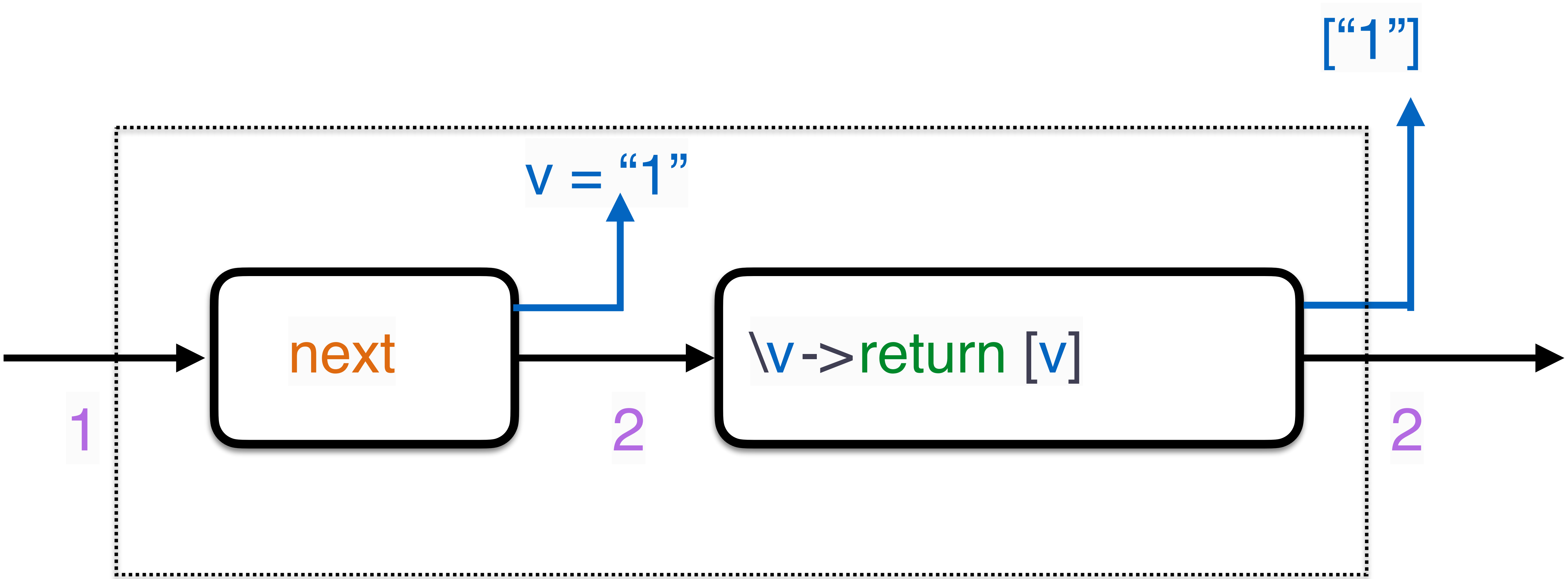
\va->eb

s’’

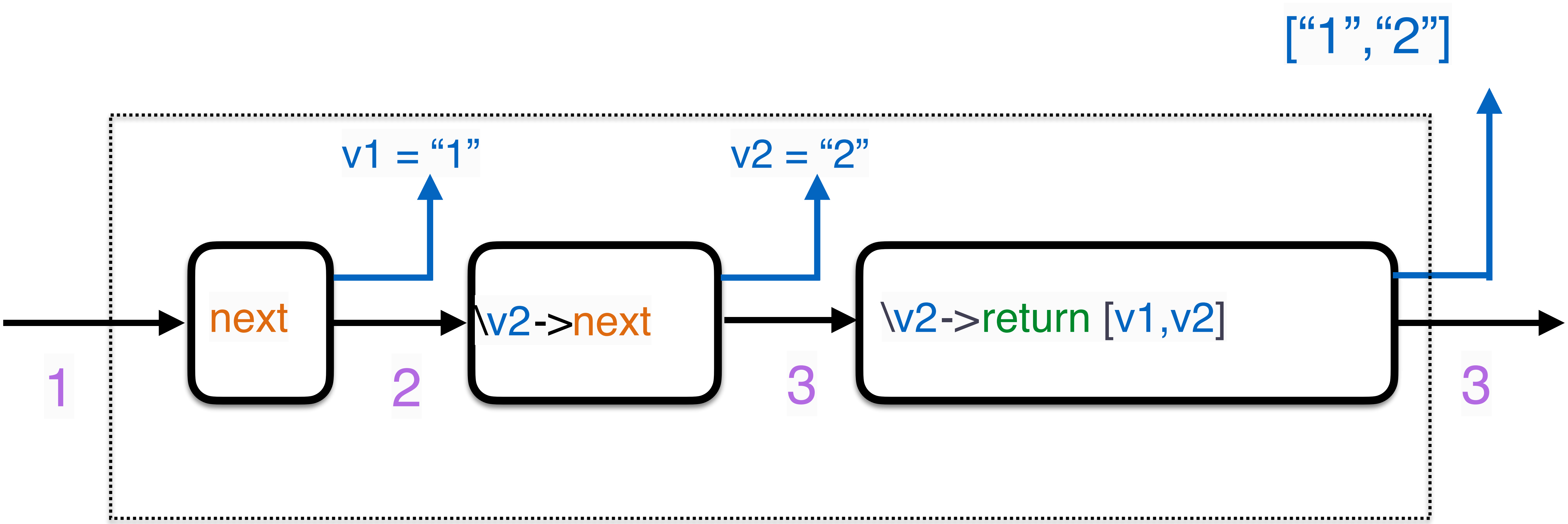
vb

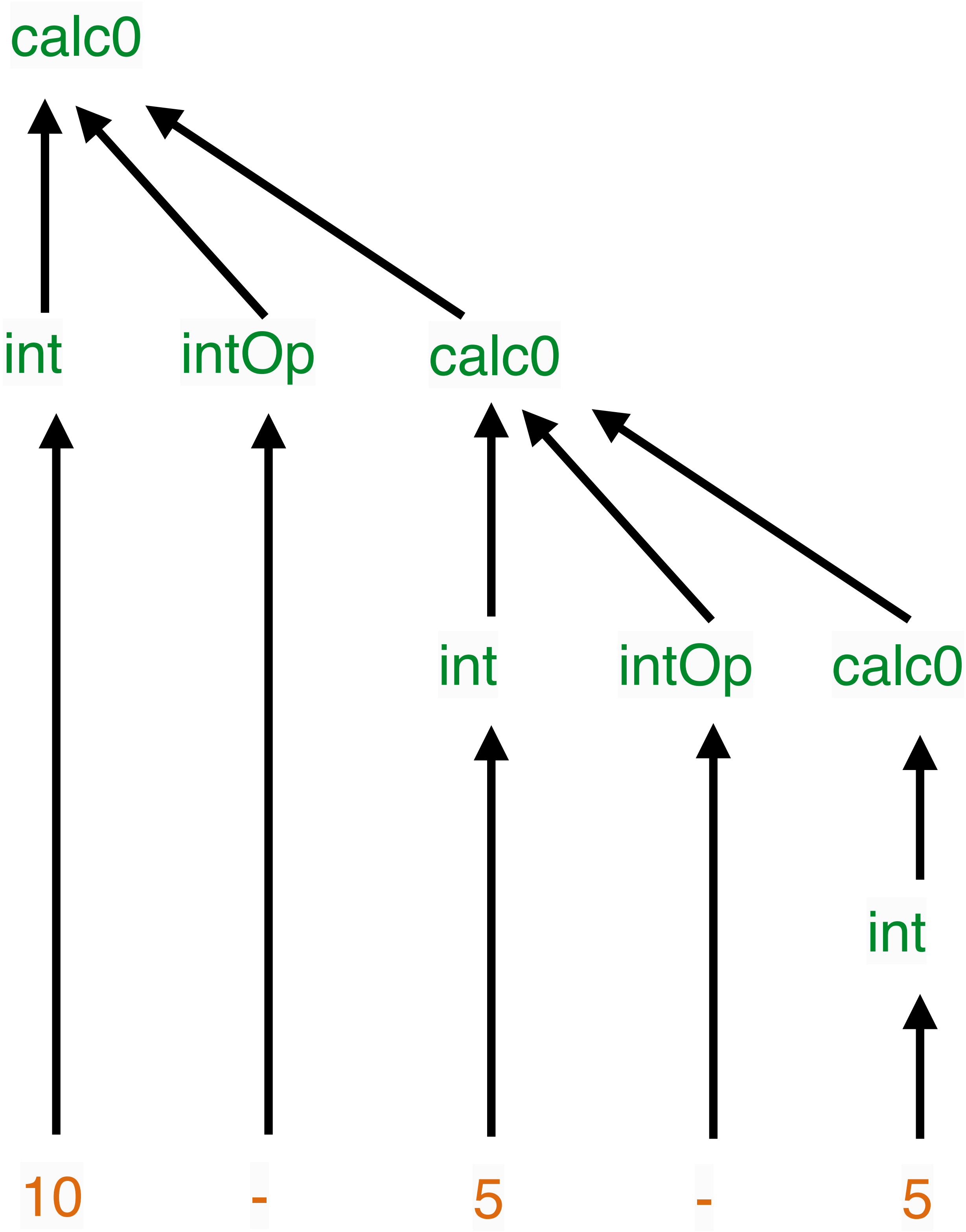
ea >>= (\va -> eb)

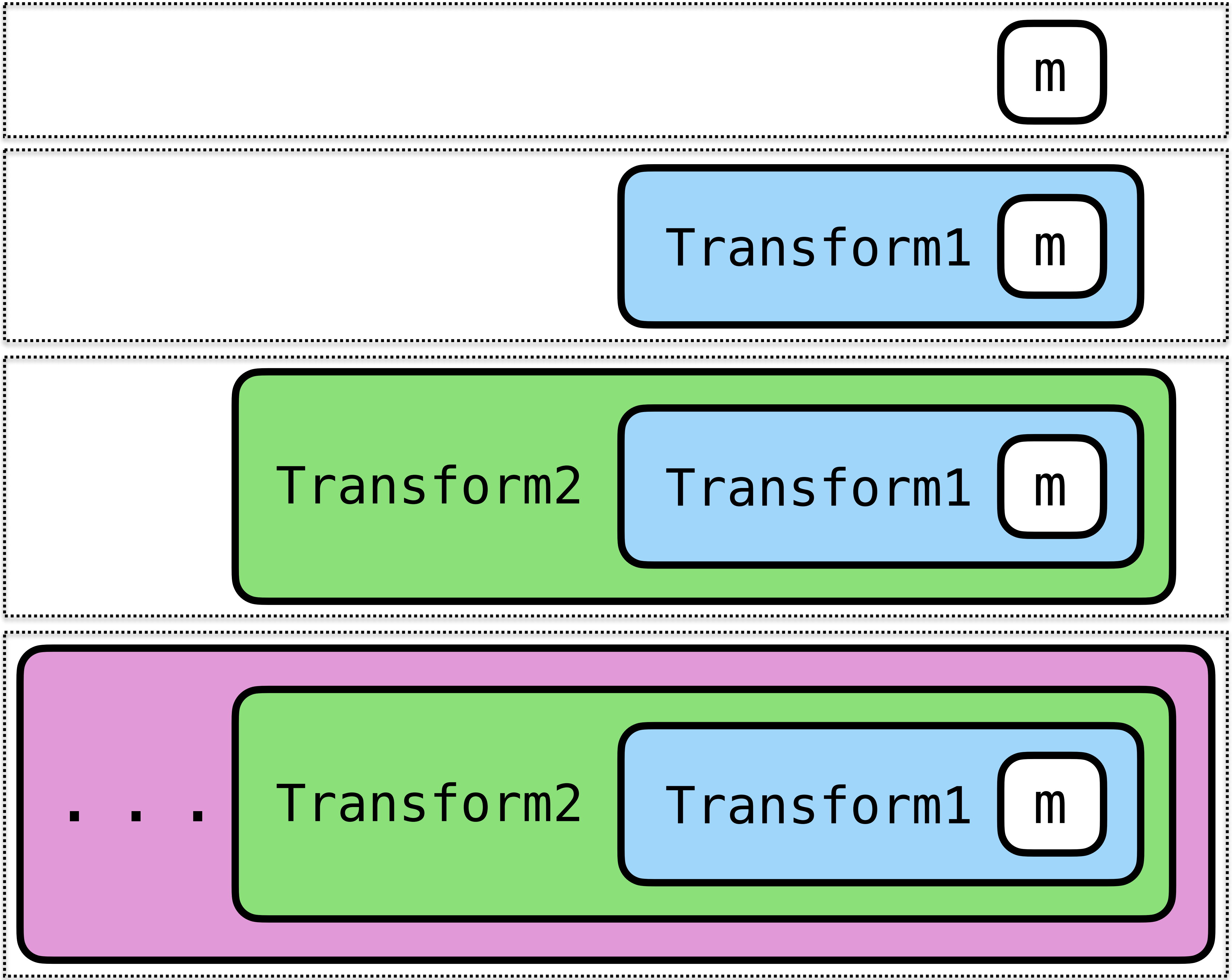
```haskell
next :: ST0 String
next = ST0C (\s -> (s+1, show s))

wtf :: ST0 [String]
wtf  = next >>= (\v -> return [v])

quiz = evalState wtf 1
```

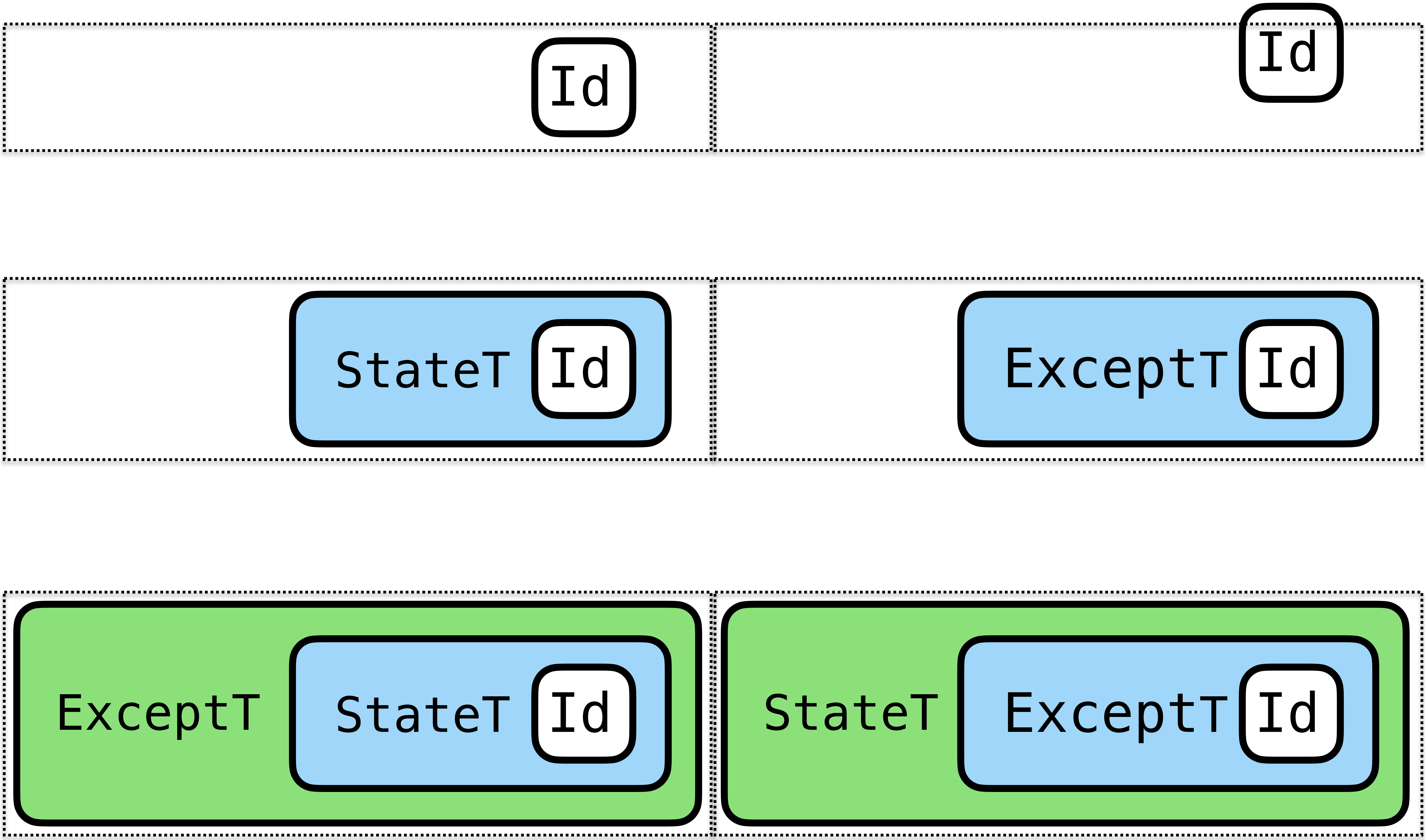


```
next :: ST0 String
next = ST0C (\s -> (s+1, show s))

wtf :: ST0 [String]
wtf  = next >>= (\v1 -> next >>= (\v2 -> return [v1, v2]))

quiz = evalState wtf 1
```




calc0
int
intOp
calc0
int
intOp
calc0
int
10
-
5
-
5


m
Transform1
m
Transform2
Transform1
m
. . .
Transform2
Transform1
m



Id

Id

StateT Id

ExceptT Id

ExceptT StateT Id

StateT ExceptT Id