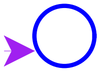
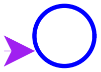


call-by-value

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Starting call-by-value...

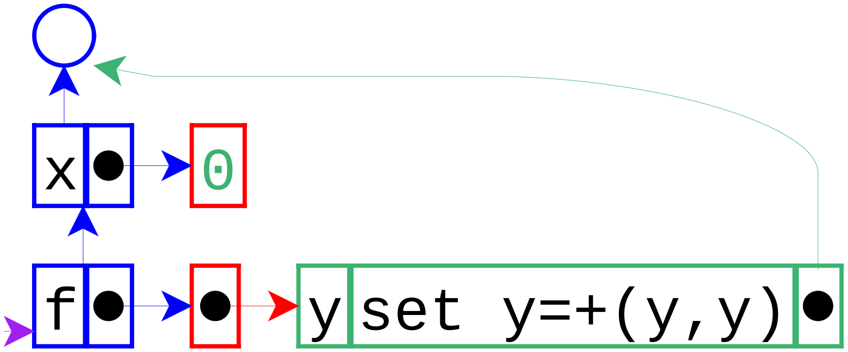


call-by-value

```
 let x = 0
     f = proc(y) set y=+(y,y)
  in let z = { set x=+(x,1) ; x }
      in { (f z) ;  z }
```

- Eval RHSs

call-by-value

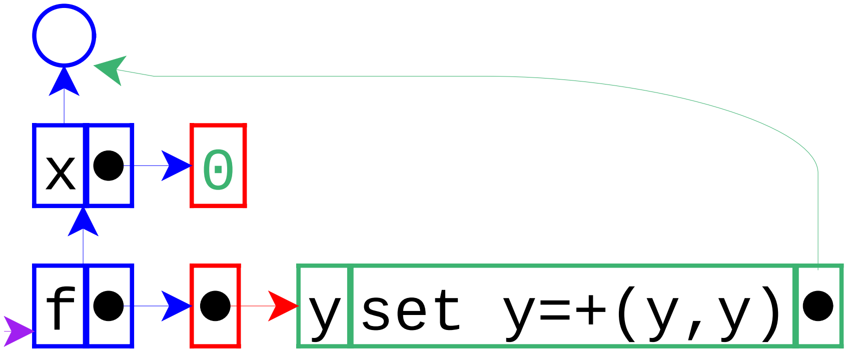


*technically, should be one frame with both x and f

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

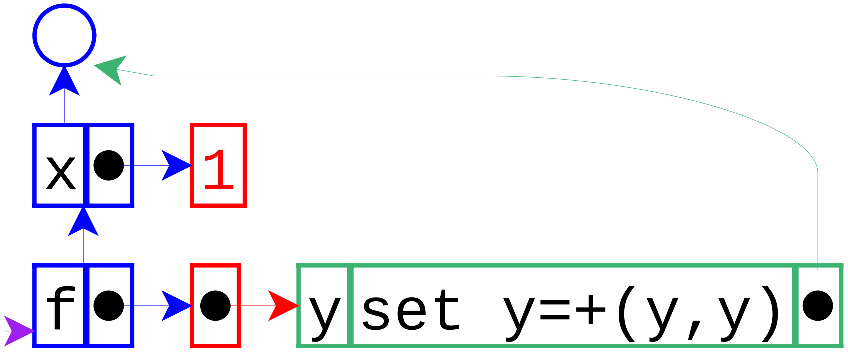
- Bind x and f to 0 and closure, respectively

call-by-value



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Eval RHS for z

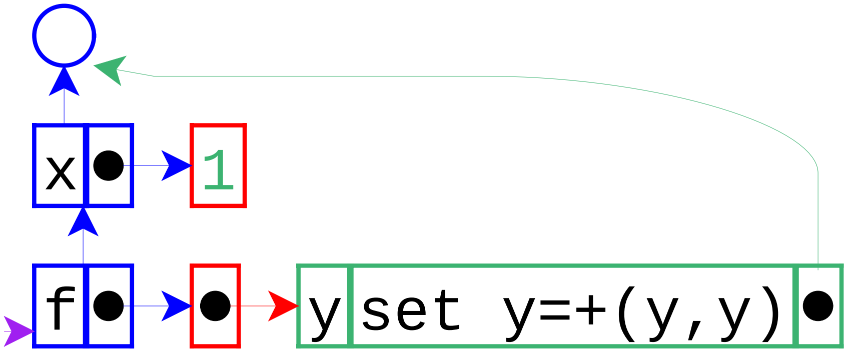


call-by-value

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Value for x changed to 1

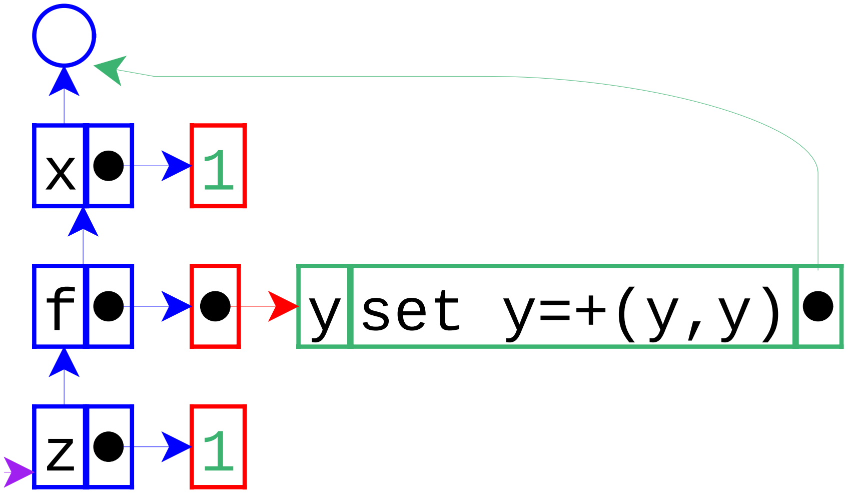
call-by-value



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

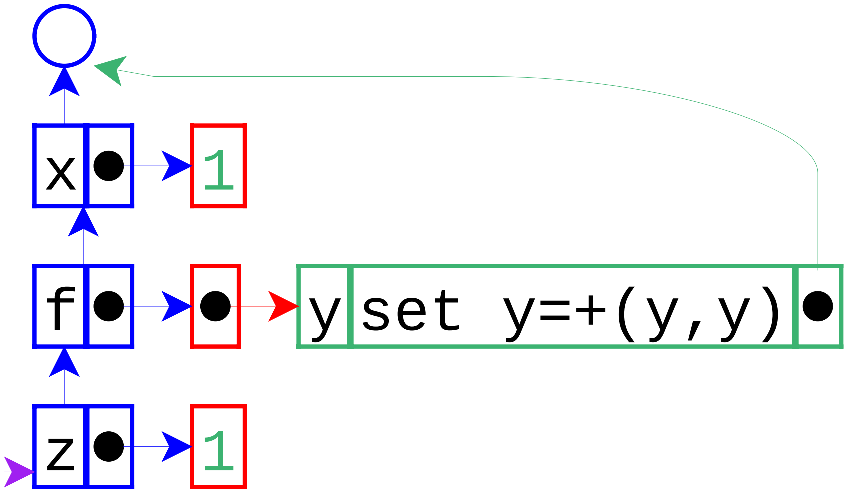
- Return x...

call-by-value



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- ... and bind z to the result, 1

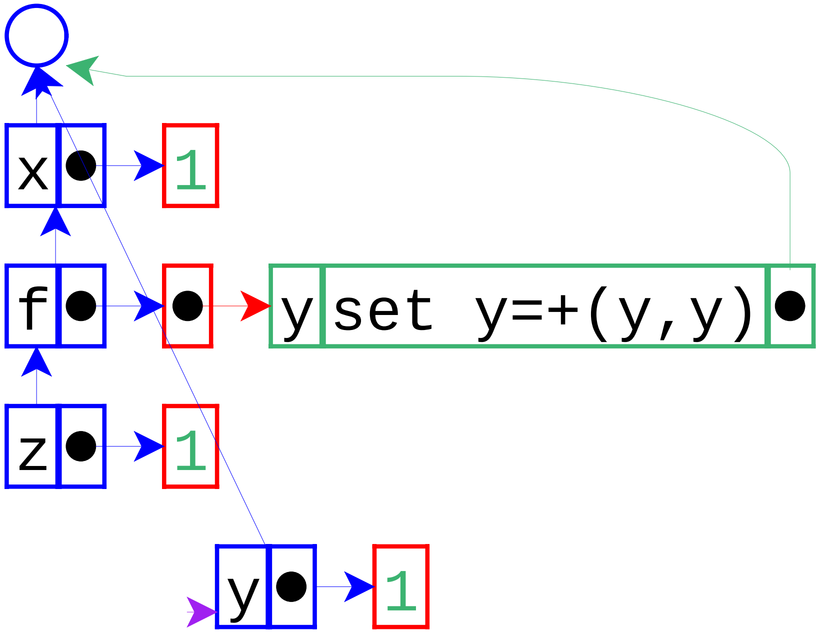


call-by-value

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Call f with z

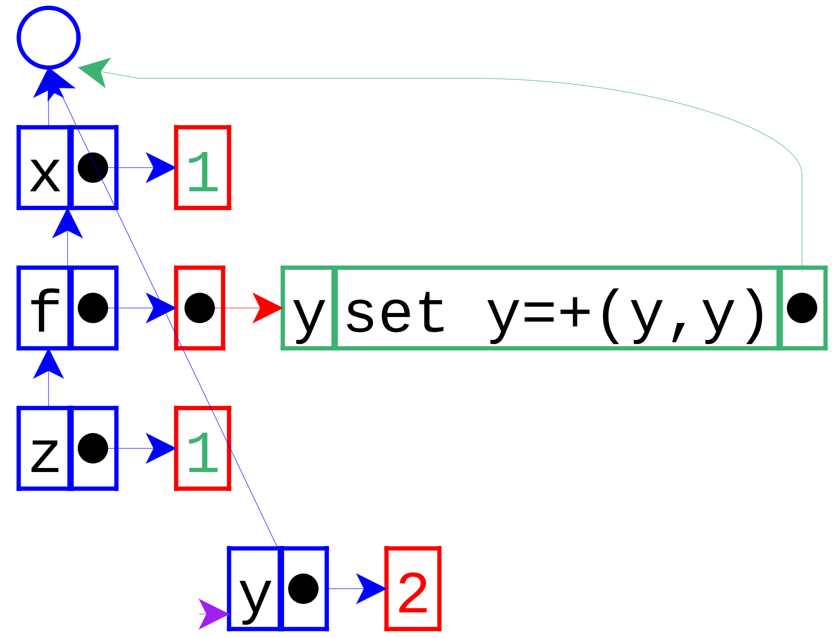
call-by-value



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Call-by-value creates a new location for y

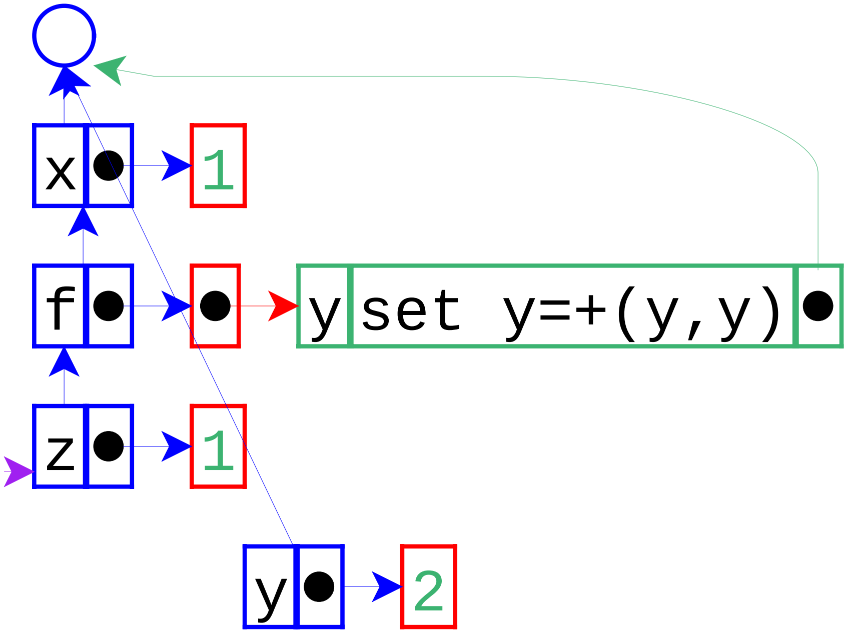
call-by-value



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Value for y changed to 2

call-by-value



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Result is the current value of z: 1

call-by-reference

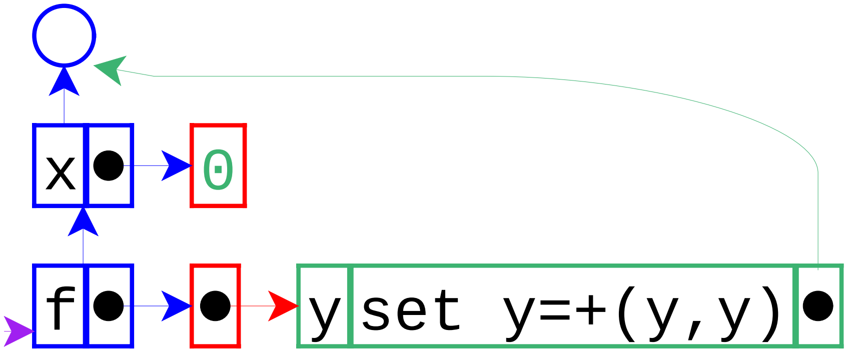
```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Starting call-by-reference...

call-by-reference

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Eval RHSs

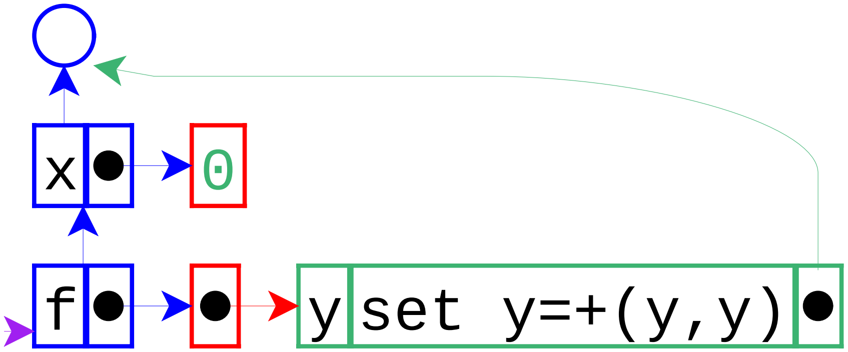


call-by-reference

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

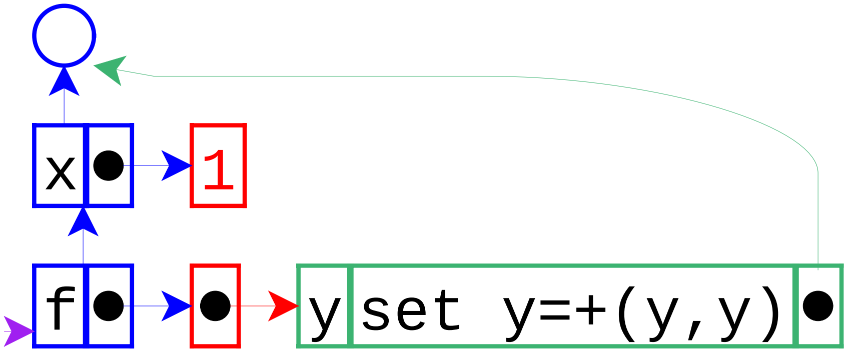
- Bind x and f to 0 and closure, respectively

call-by-reference



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Eval RHS for z

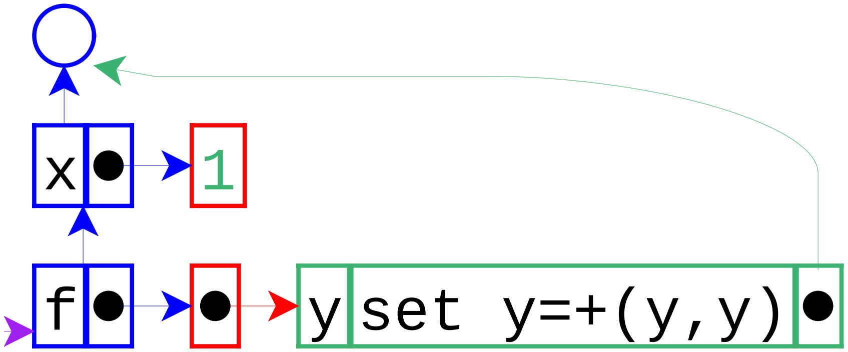


call-by-reference

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

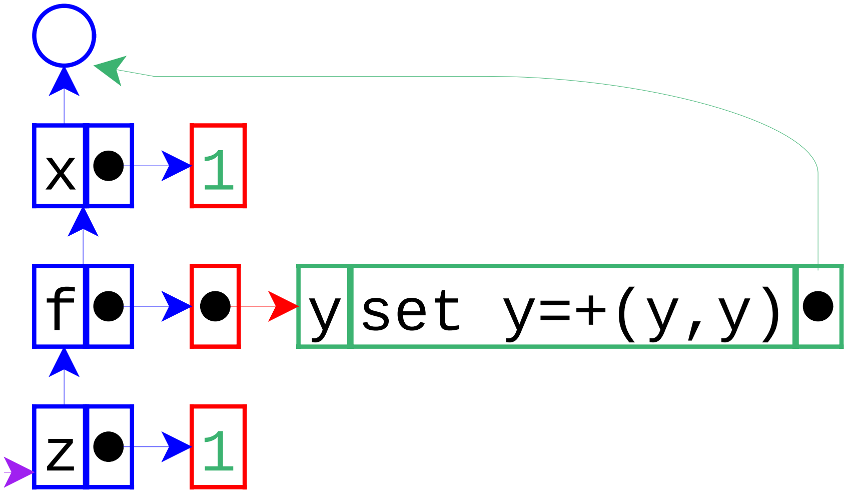
- Value for x changed to 1

call-by-reference



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Return x...

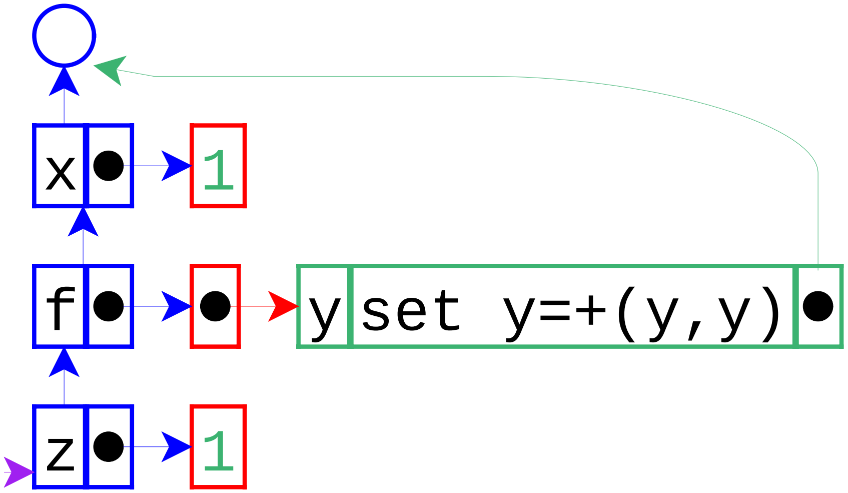


call-by-reference

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- ... and bind z to the result, 1

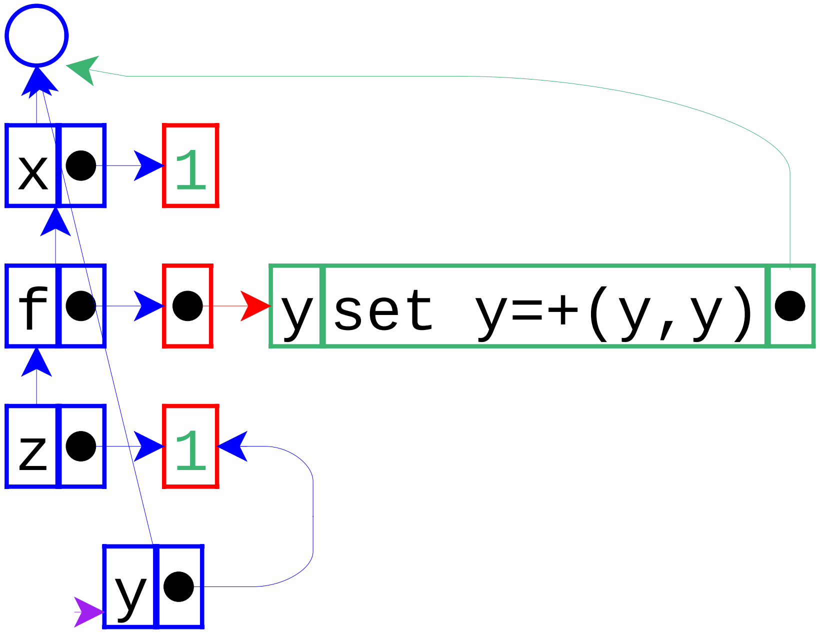
call-by-reference



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Call f with z

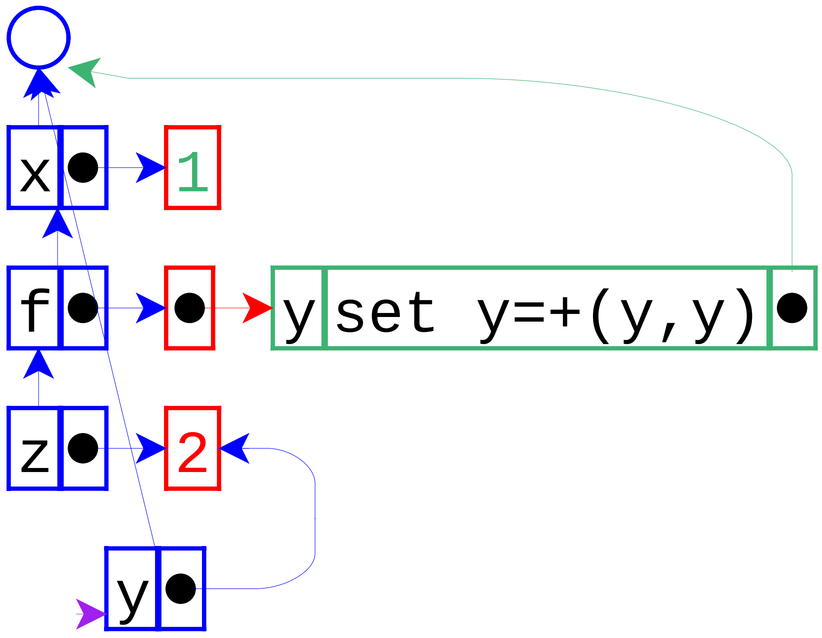
call-by-reference



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

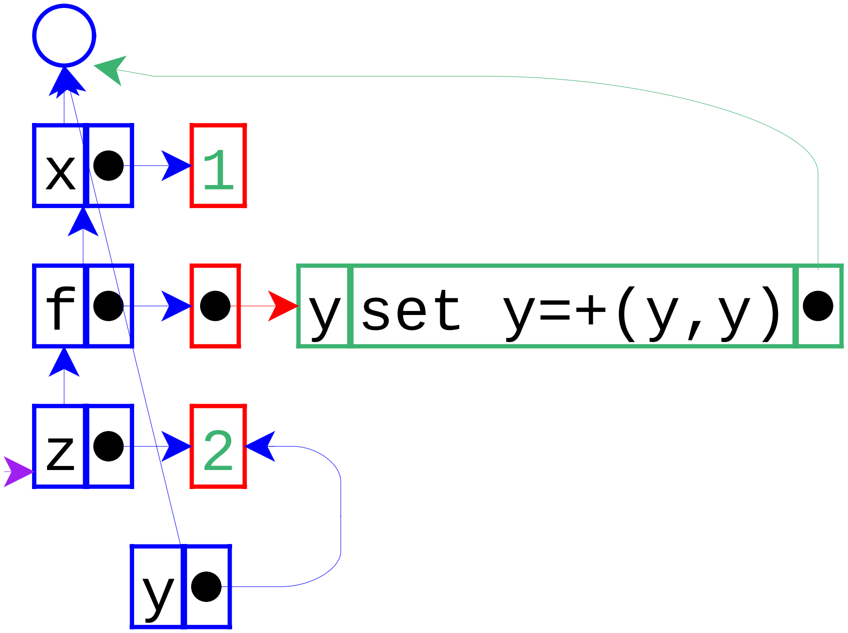
- Call-by-reference shares location for z with y

call-by-reference



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Value for y (and therefore z) changed to 2



call-by-reference

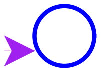
```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Result is the current value of z: 2

call-by-name

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Starting call-by-name...

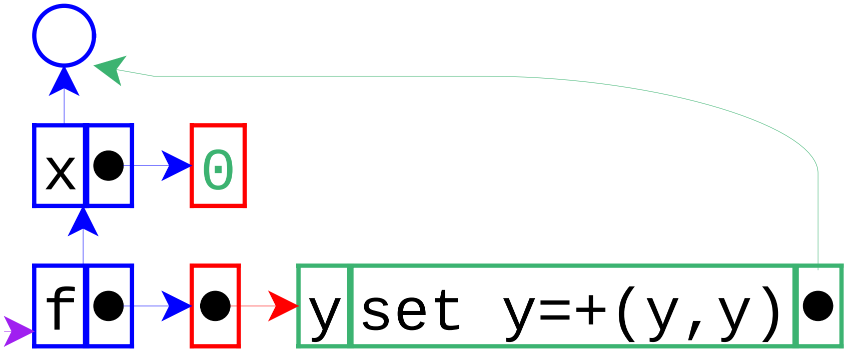


call-by-name

```
 let x = 0
     f = proc(y) set y=+(y,y)
  in let z = { set x=+(x,1) ; x }
      in { (f z) ;  z }
```

- Eval RHSs

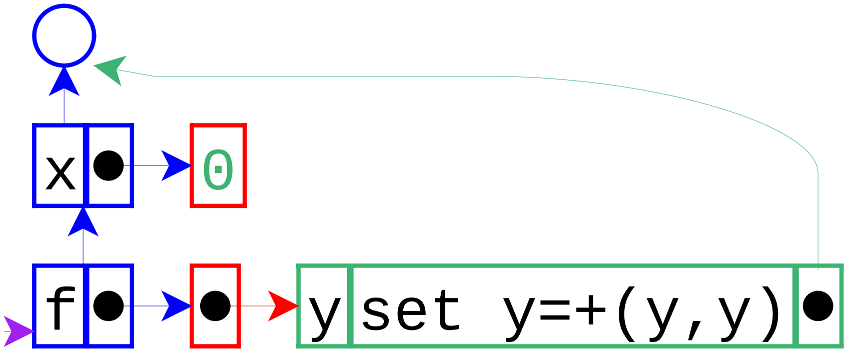
call-by-name



```
    let x = 0
        f = proc(y) set y=+(y,y)
     in let z = { set x=+(x,1) ; x }
         in { (f z) ;  z }
```

- Simple expressions: bind x and f to 0 and closure, respectively

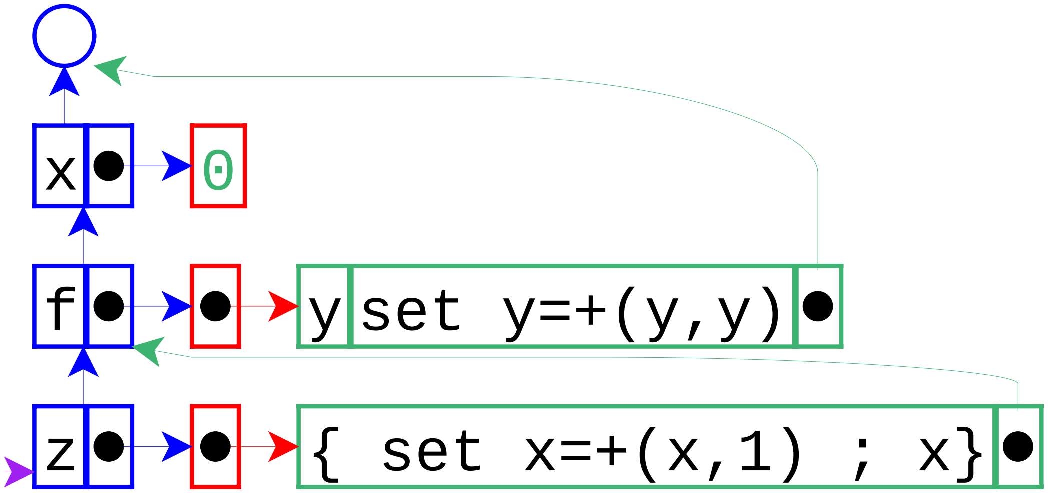
call-by-name



```
  let x = 0
      f = proc(y) set y=+(y,y)
   in let z = { set x=+(x,1) ; x }
       in { (f z) ;  z }
```

- Handle RHS of z...

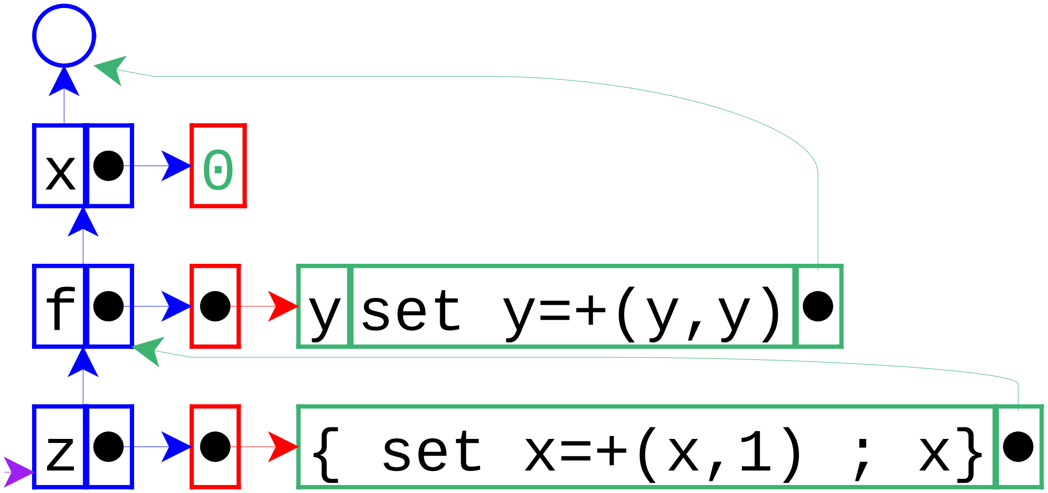
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- ... by creating a thunk for z

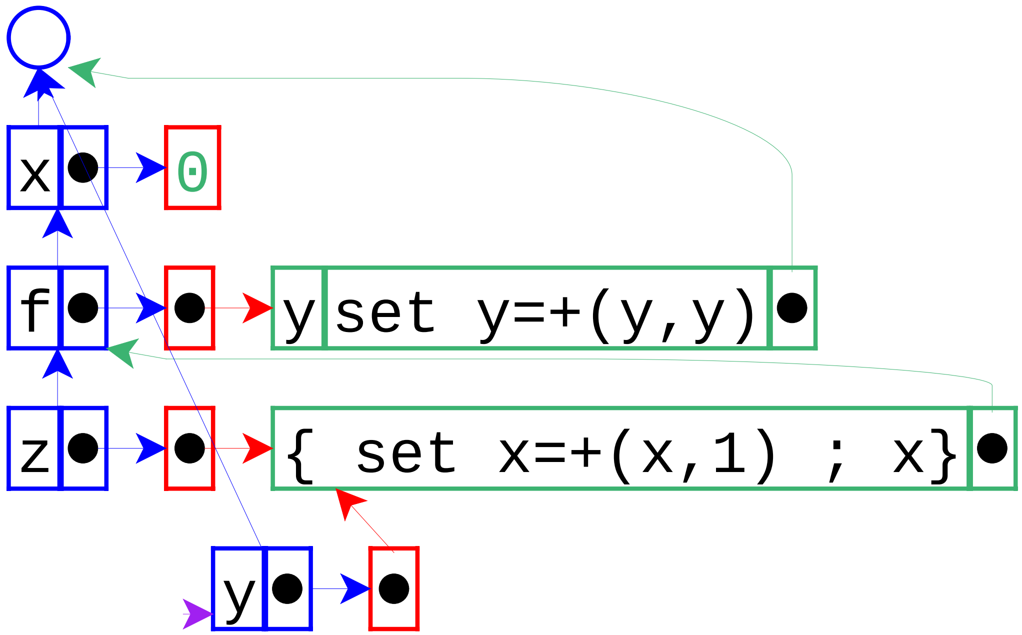
call-by-name



```
  let x = 0
      f = proc(y) set y=+(y,y)
   in let z = { set x=+(x,1) ; x }
       in { (f z) ;  z }
```

- Call f with z

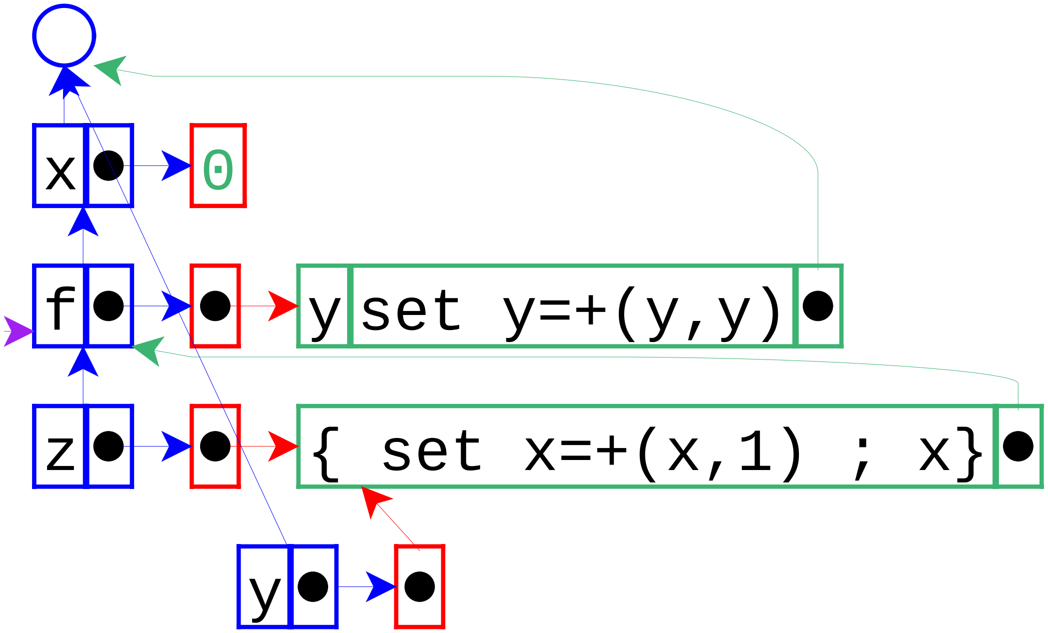
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Not by-reference; y gets a new location, containing the same thunk as z’s location

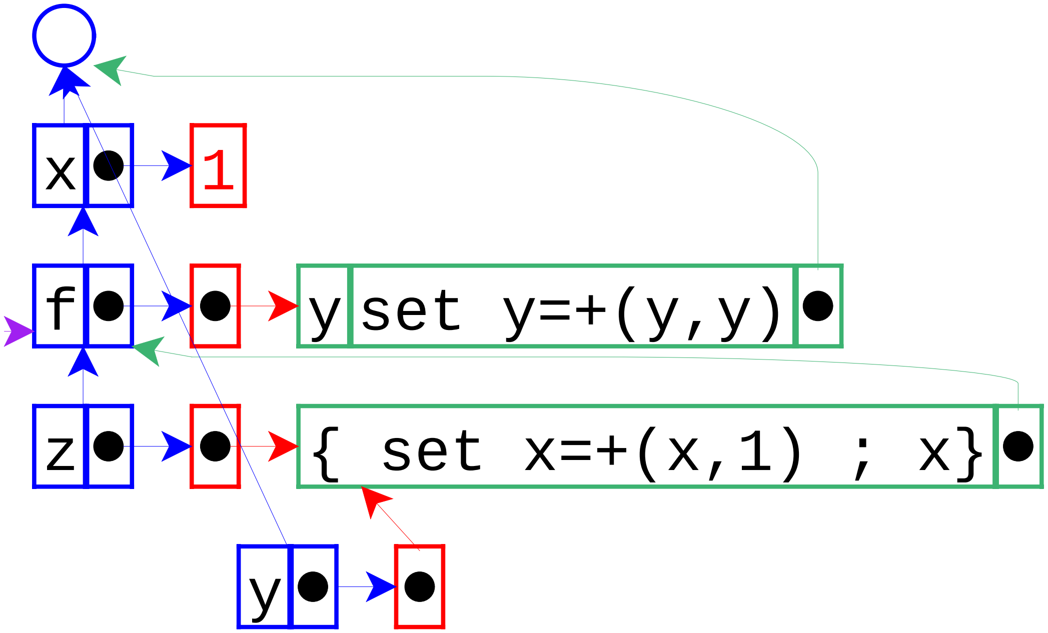
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Use of y means we eval the thunk

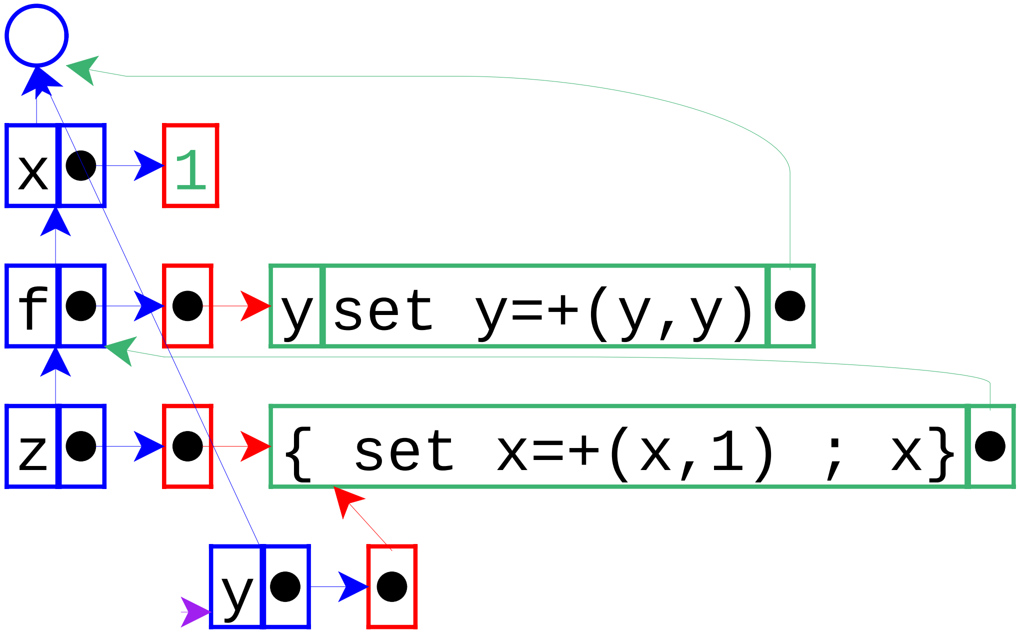
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Thunk changes value of x to 1

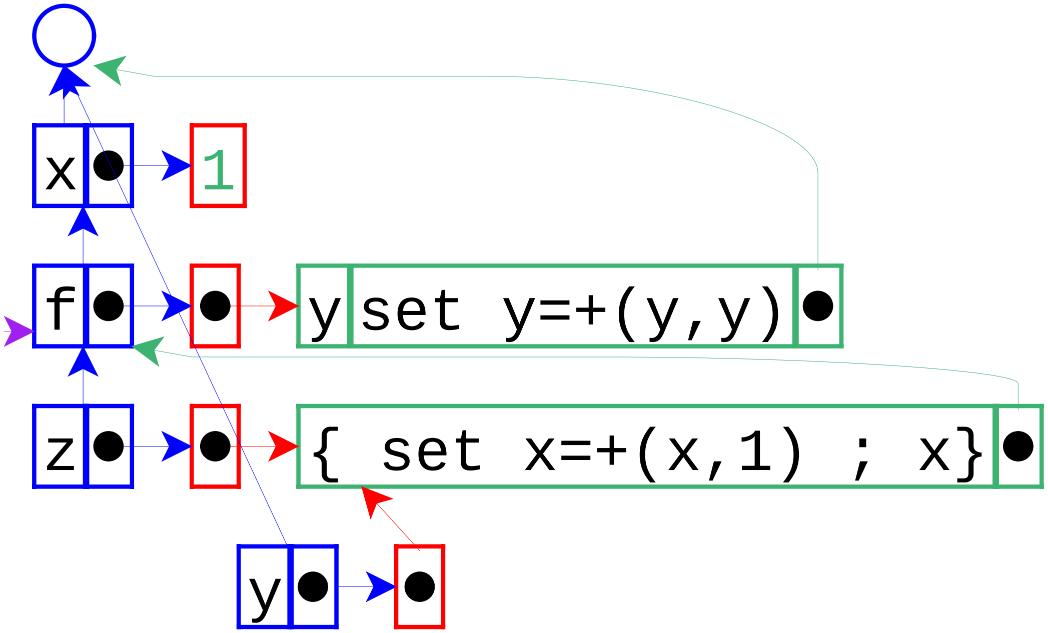
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Result for first use of y is 1

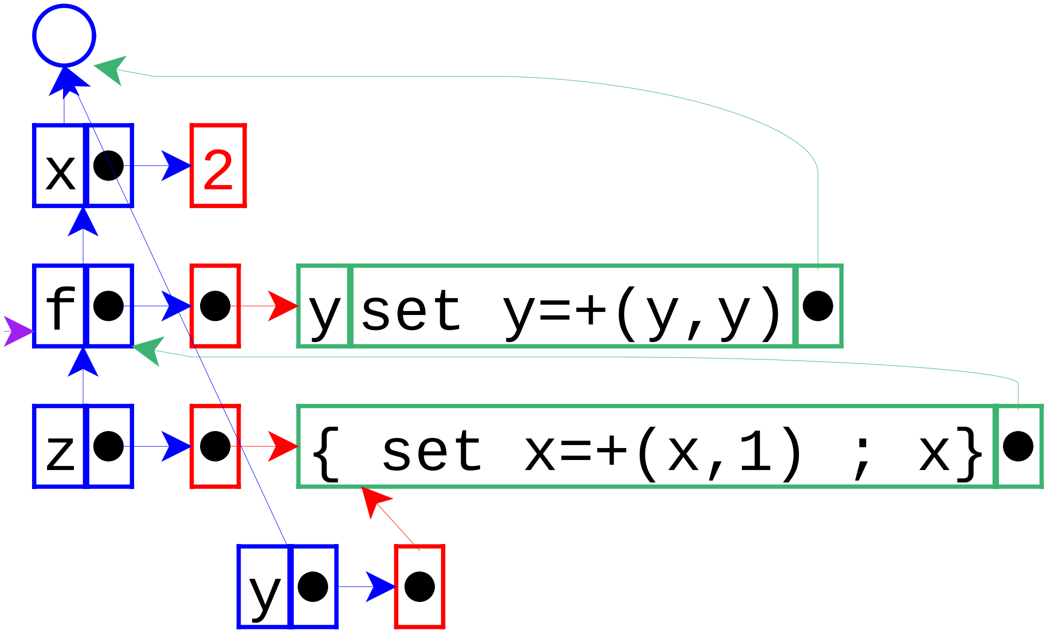
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Another use of y means we eval the thunk again

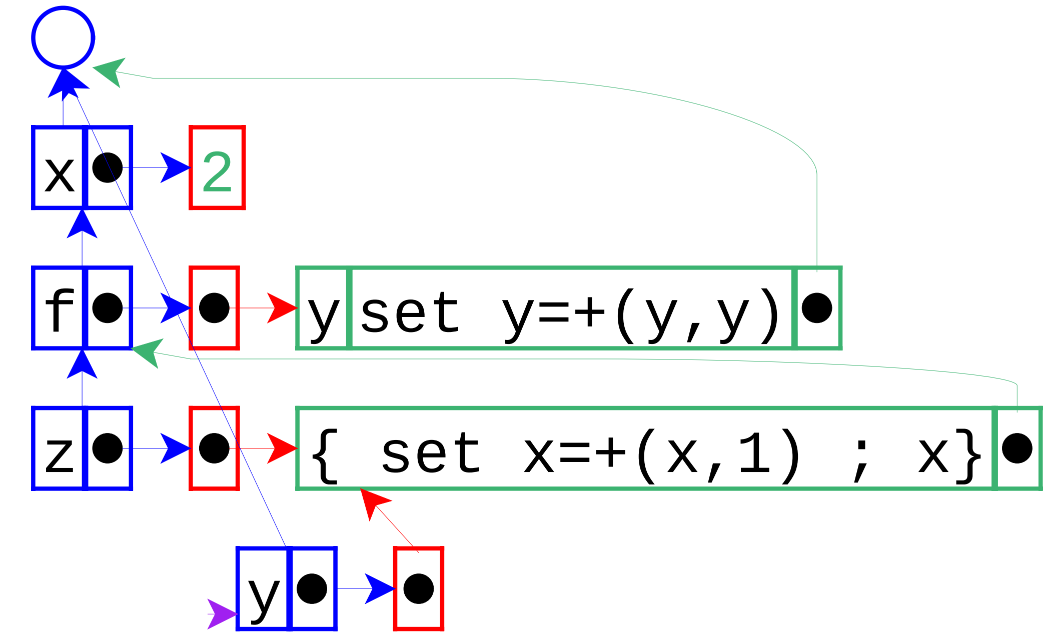
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

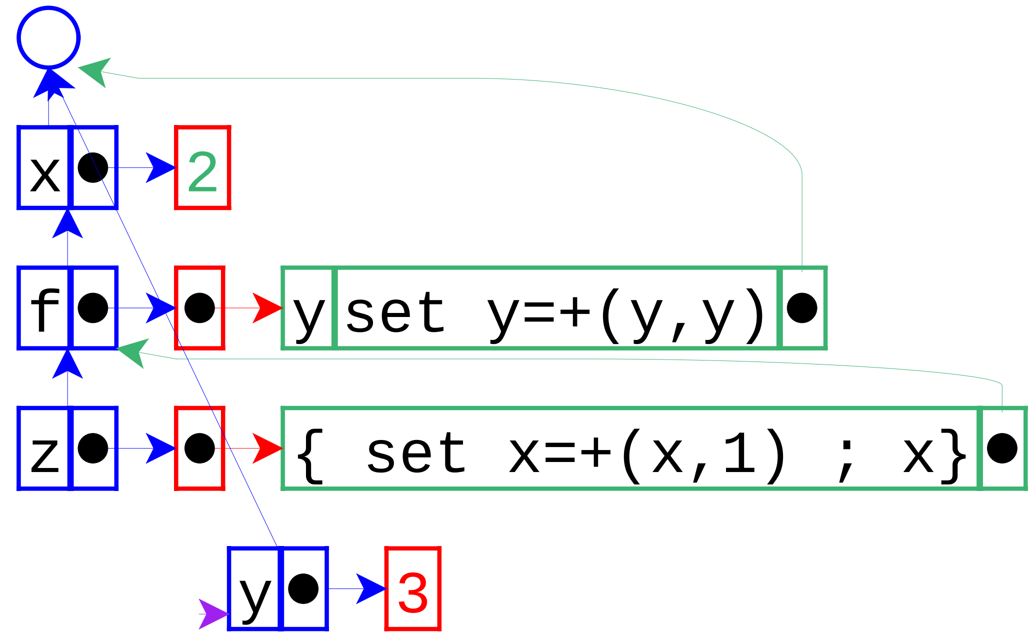
- Thunk changes value of x to 2

call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Result for second use of y is 2

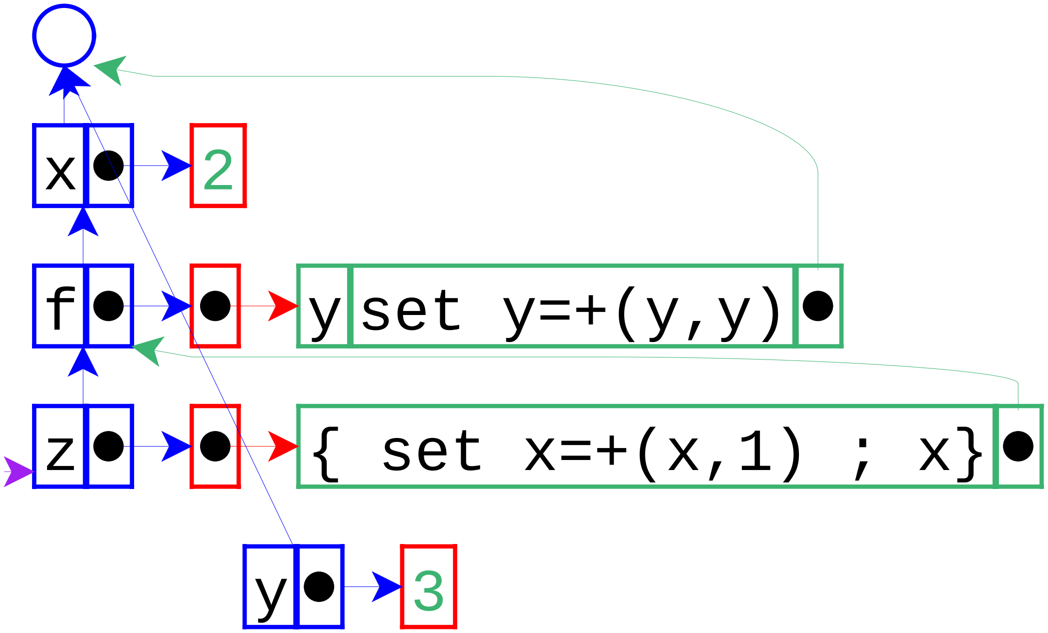


call-by-name

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Value for y changed to 3 (= 1 + 2)

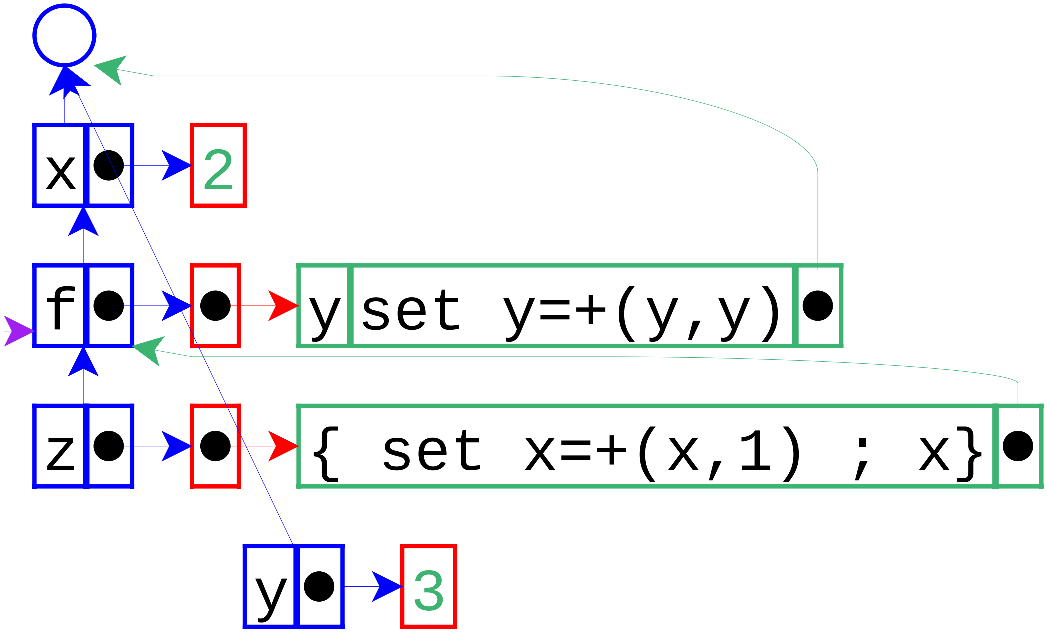
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Result is the value of z...

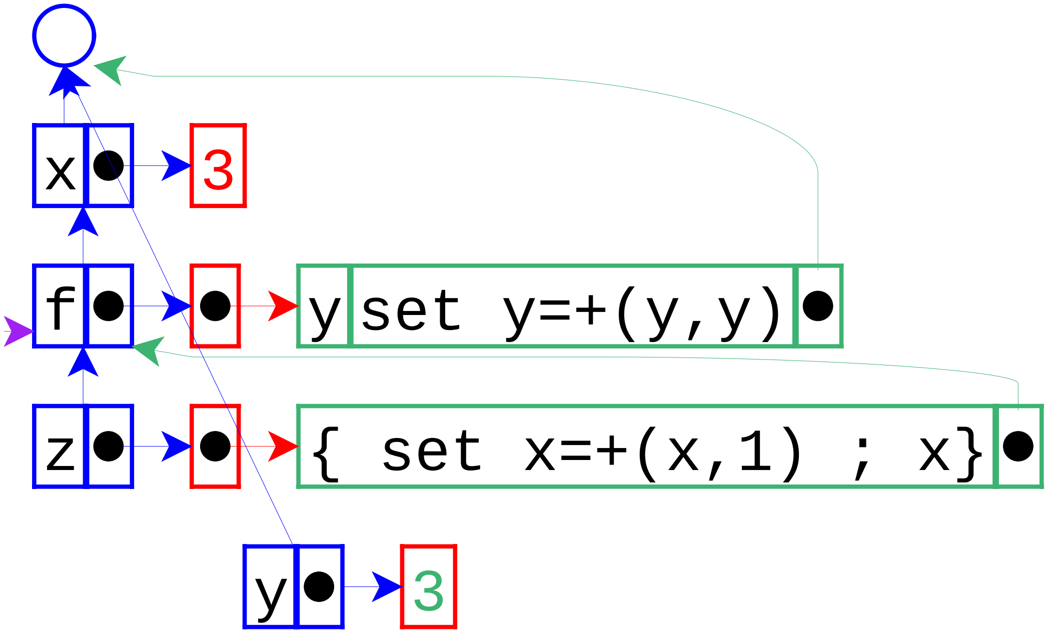
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- ... which means eval the thunk again

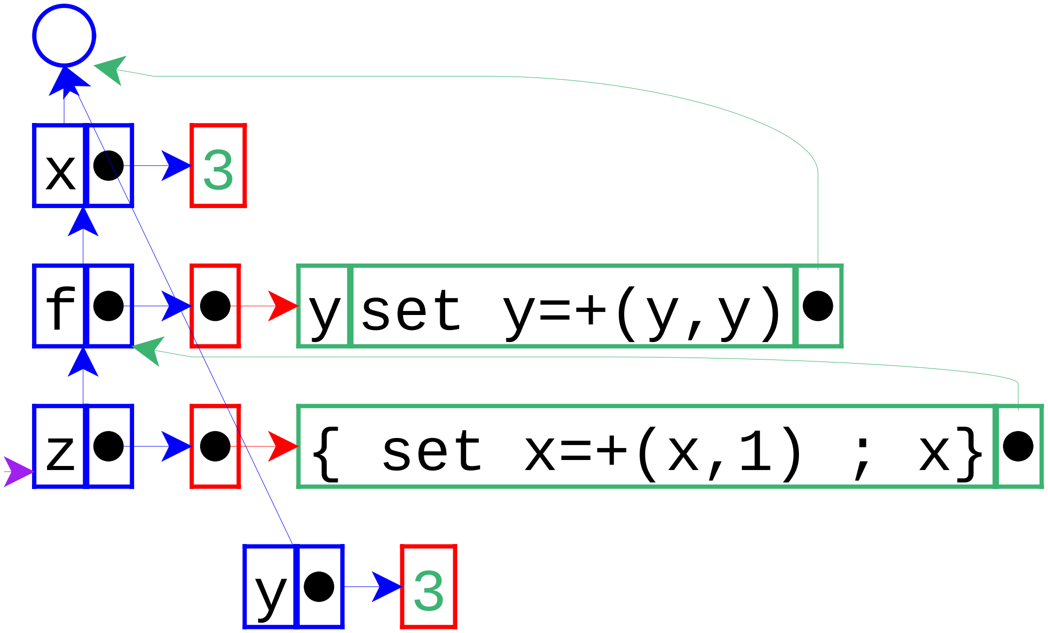
call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Thunk changes value of x to 3

call-by-name



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- So 3 is the final result

call-by-need

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

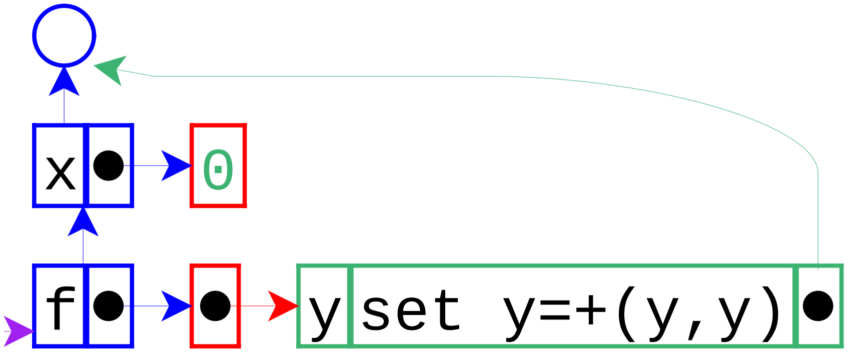
- Starting call-by-need...

call-by-need

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Eval RHSs

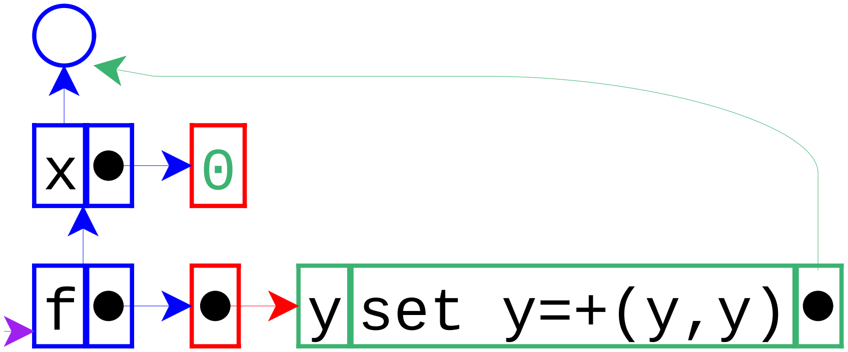
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Simple expressions: bind x and f to 0 and closure, respectively

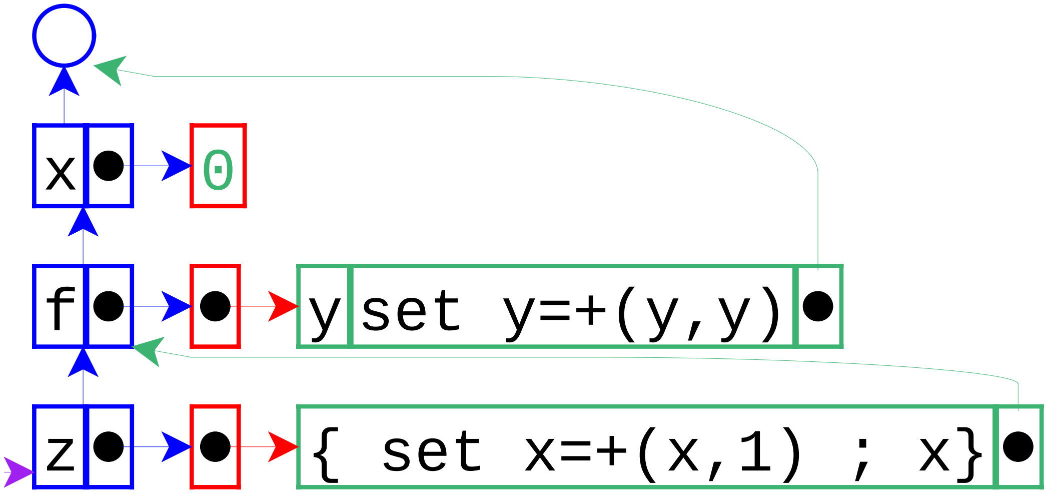
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Handle RHS of z...

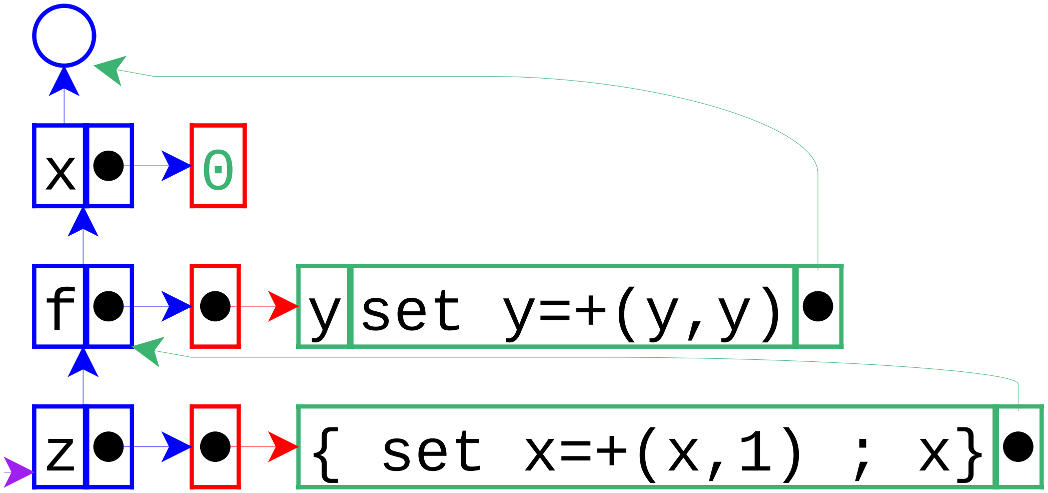
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- ... by creating a thunk for z

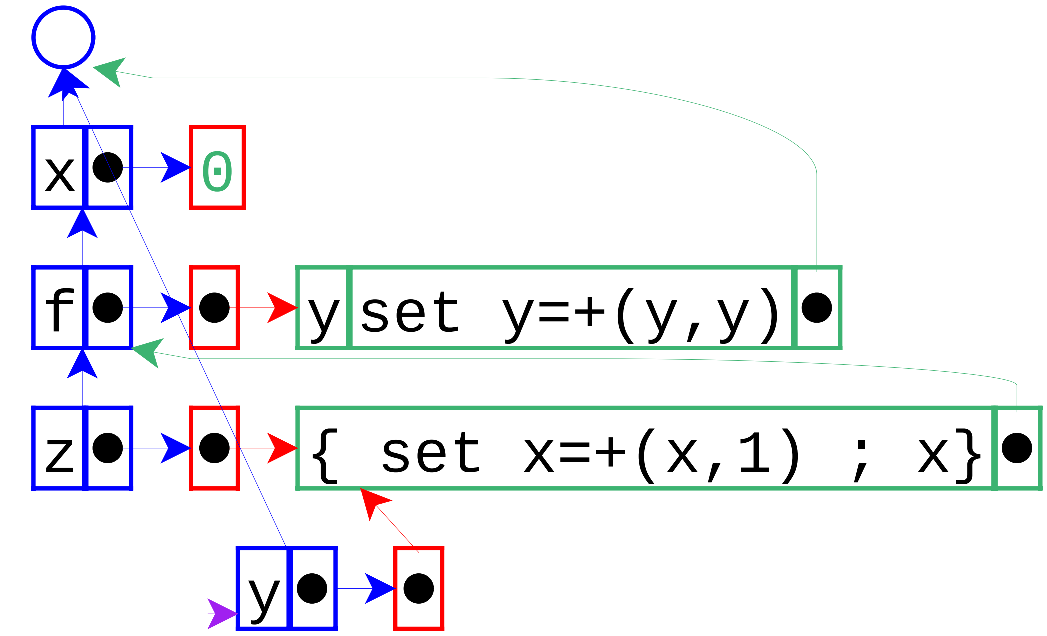
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Call f with z

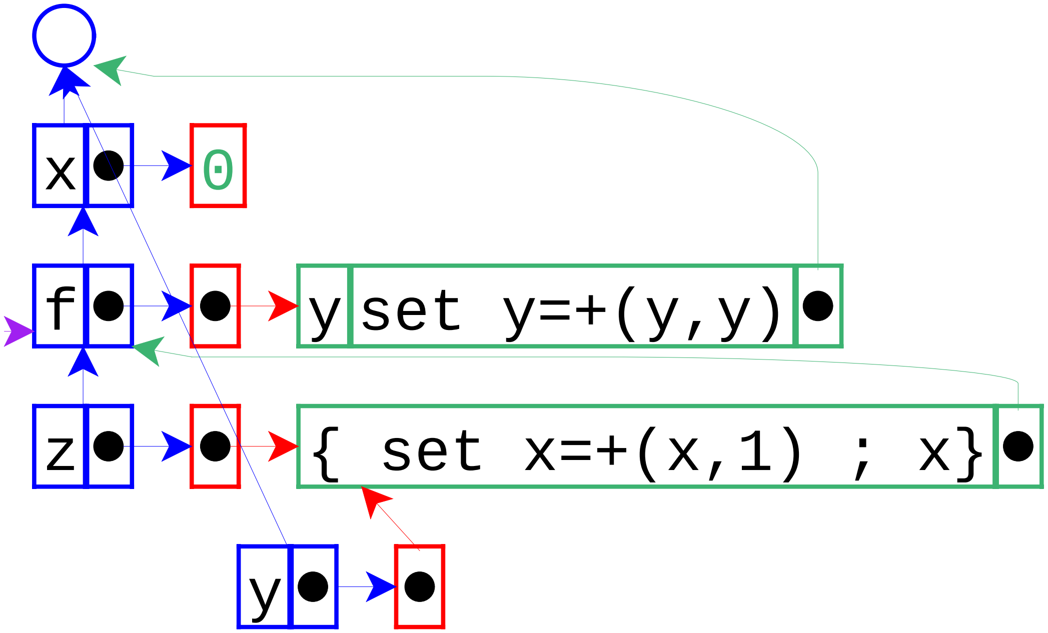
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Not by-reference; y gets a new location, containing the same thunk as z’s location

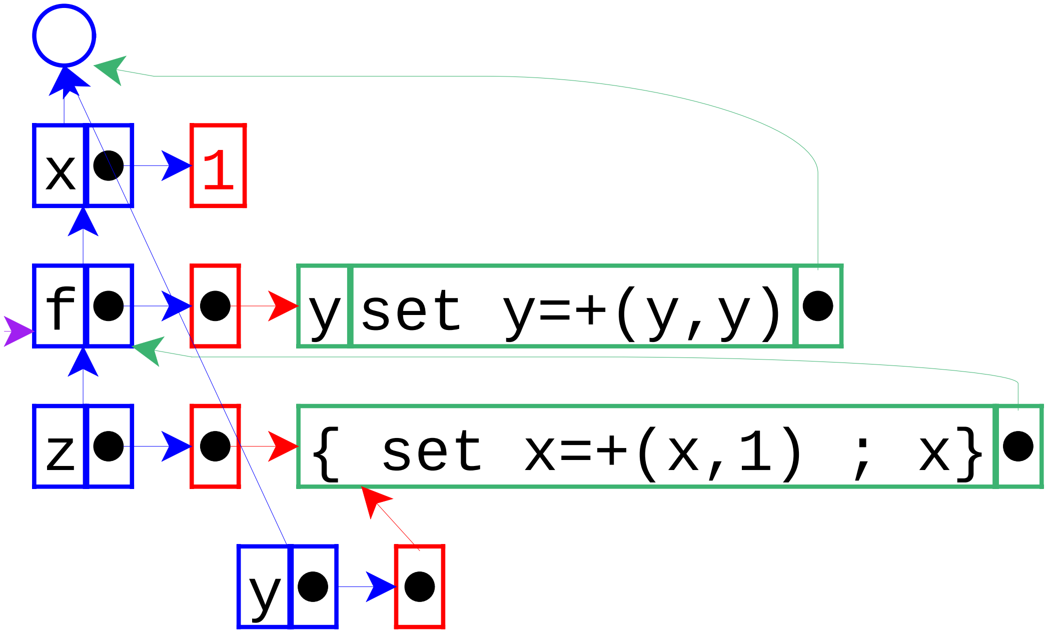
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Use of y means we eval the thunk

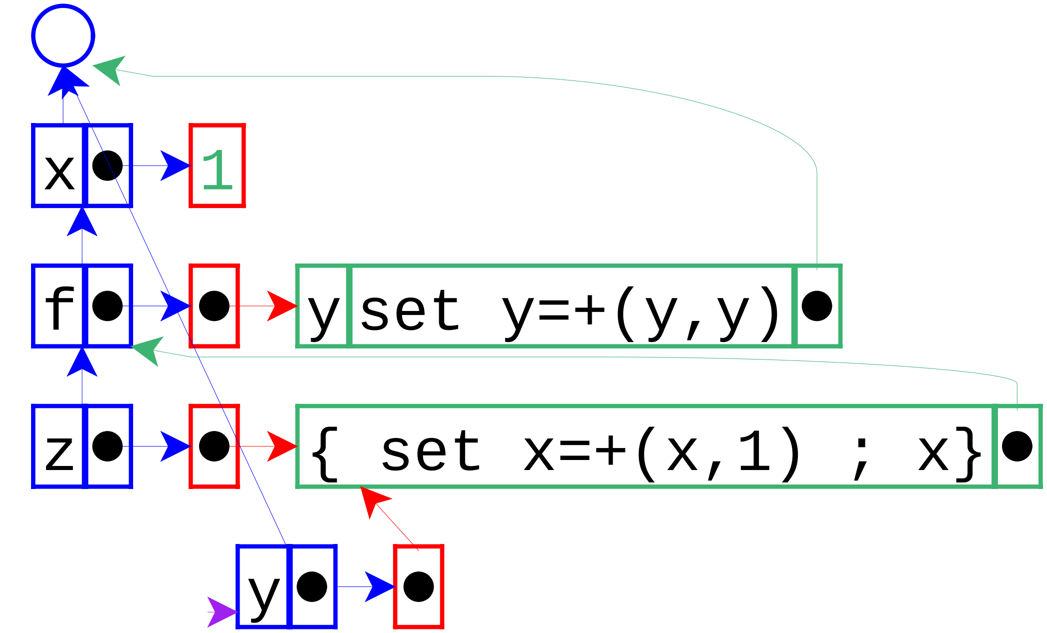
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Thunk changes value of x to 2

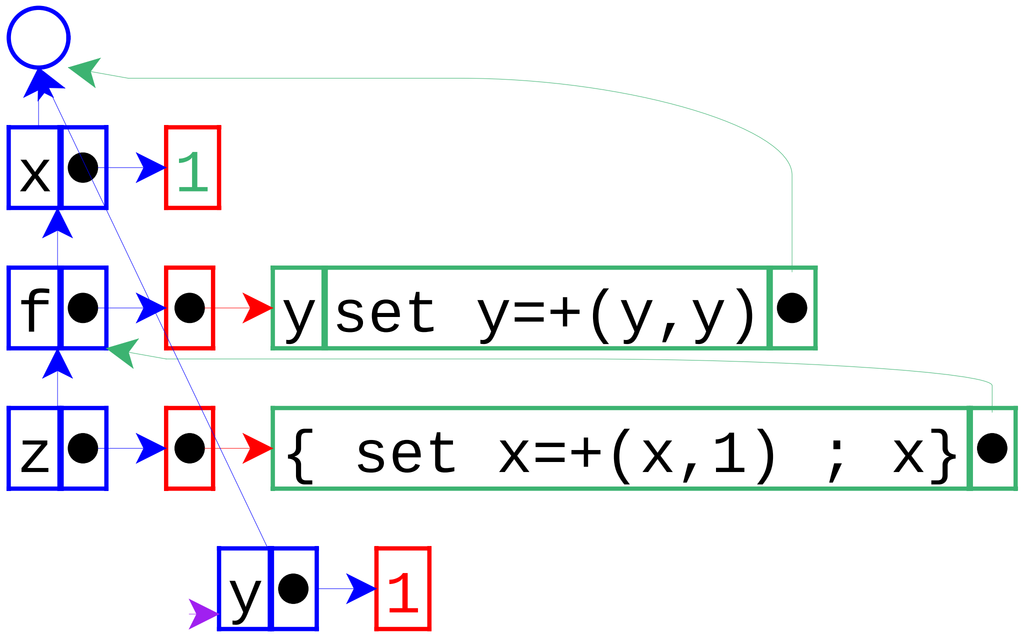
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Result from first use of y was 1

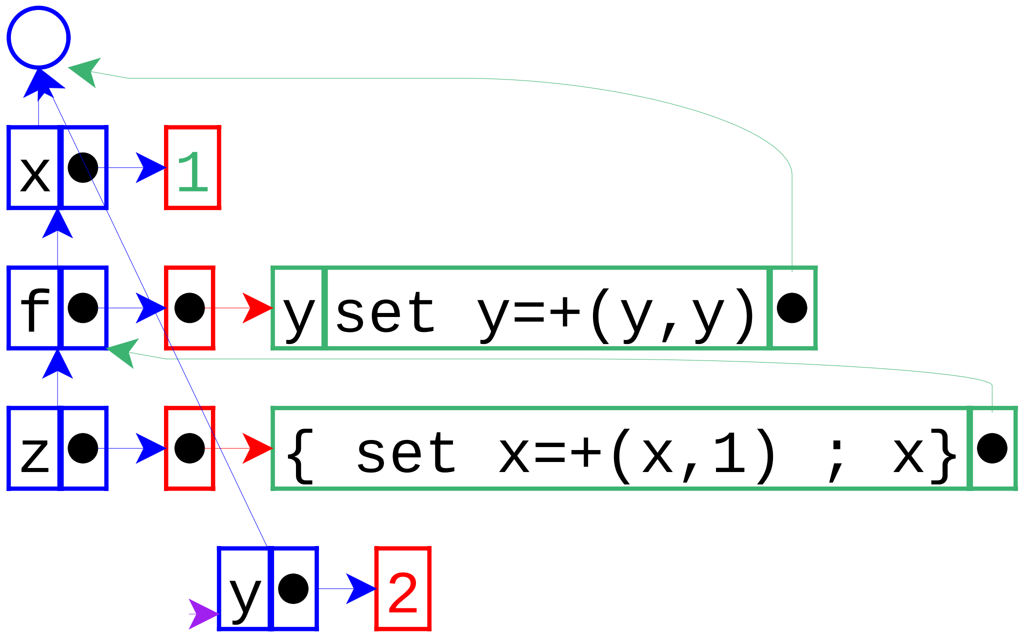
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Since this is call-by-need, install the 1 into y

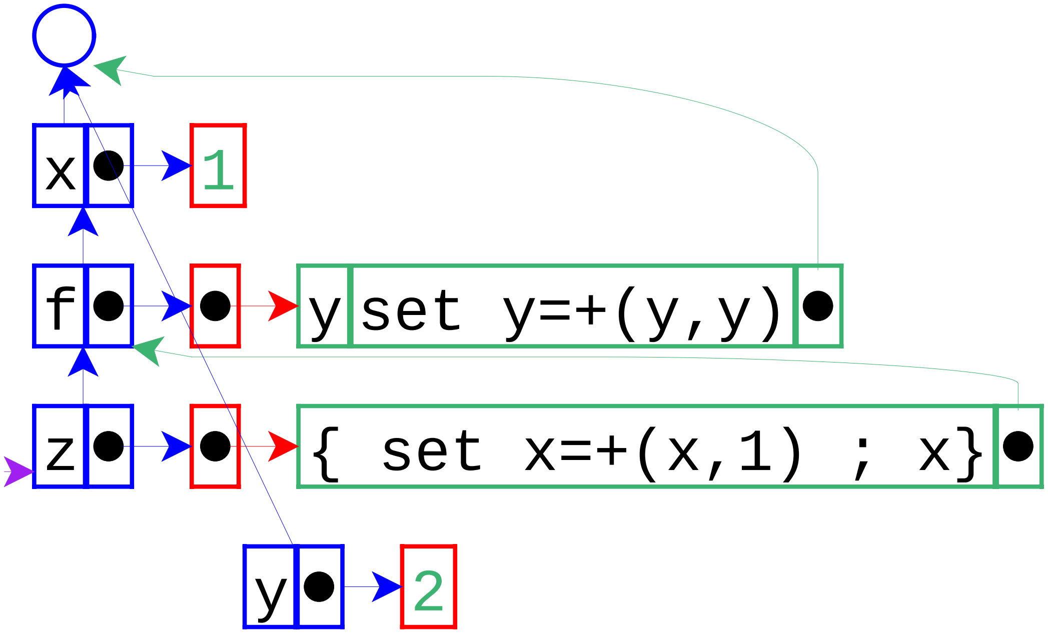
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Second use of y gets 1, set y to 2 (= 1+1)

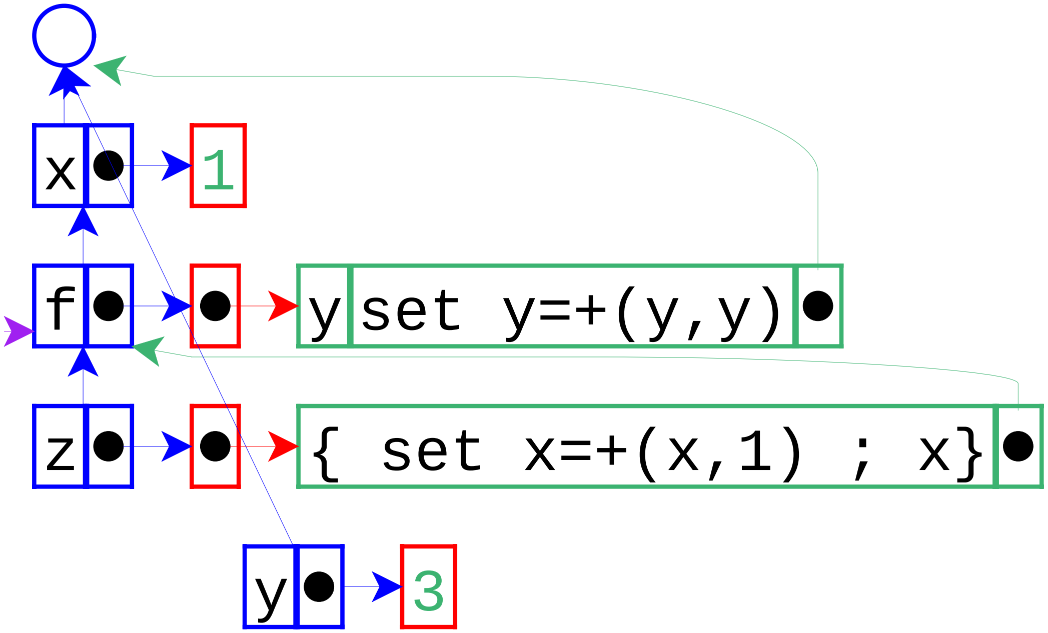
call-by-need



```
  let x = 0
      f = proc(y) set y=+(y,y)
   in let z = { set x=+(x,1) ; x }
       in { (f z) ;  z }
```

- Result is value of z...

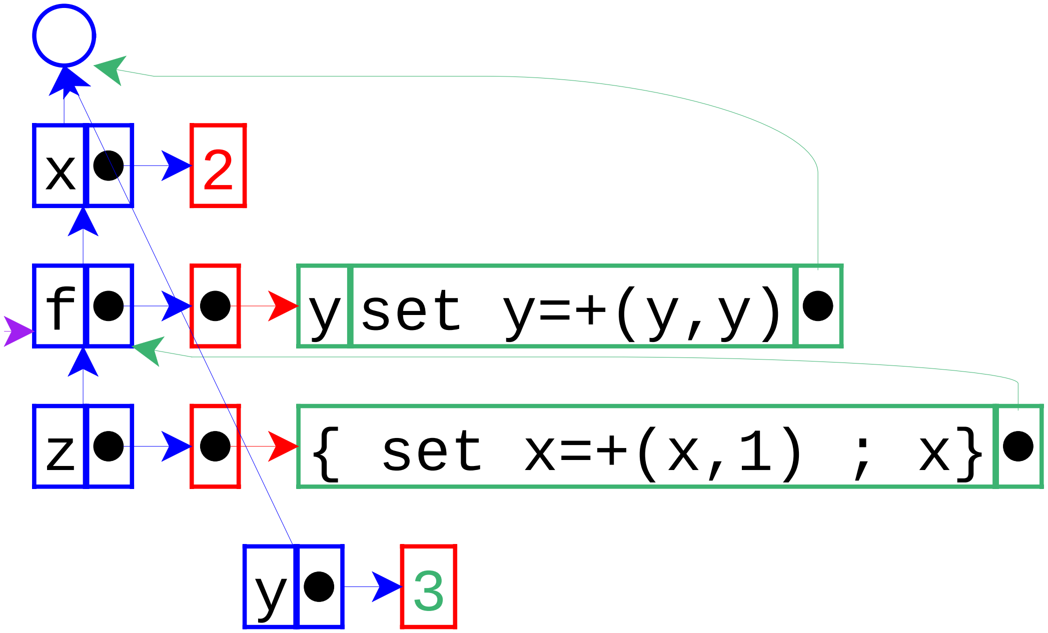
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- ... which means eval the thunk again (see note at end of this section)

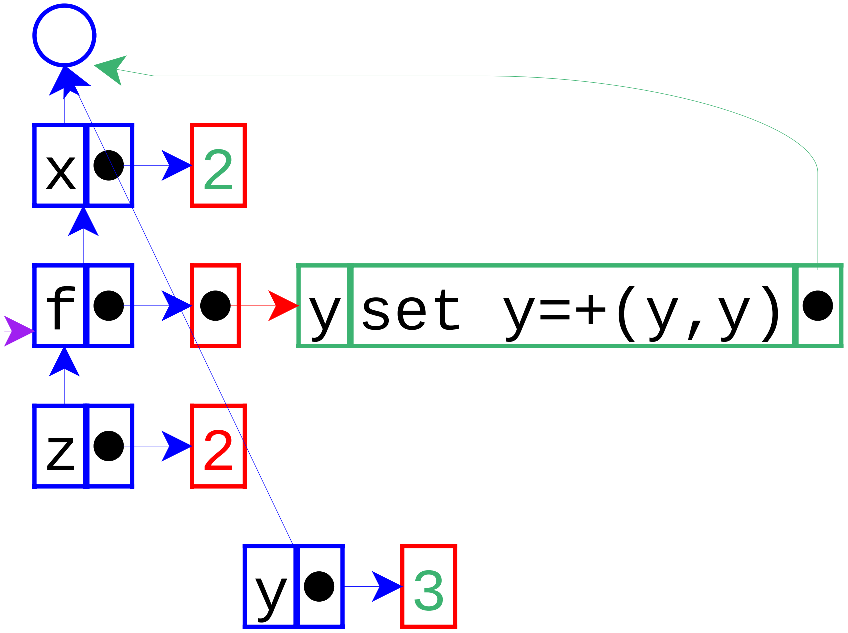
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
   in { (f z) ;  z }
```

- Thunk changes value of x to 2

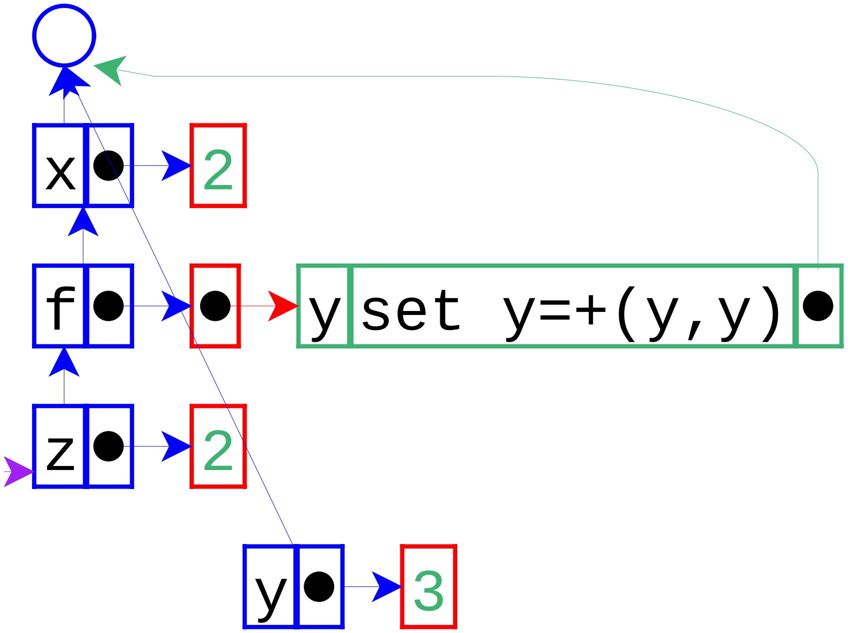
call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Result of thunk is 2; install result into z

call-by-need



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Fnal result is from z: 2

Note:

- Our interpreter implements a strange kind of call-by-need, where using a variable in a function call can cause a thunk to be evaluated multiple times.
- This strangeness is an artifact of supporting call-by-reference, where we alway treating variable arguments specially (even in the call-by-value case).

call-by-name/ref

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

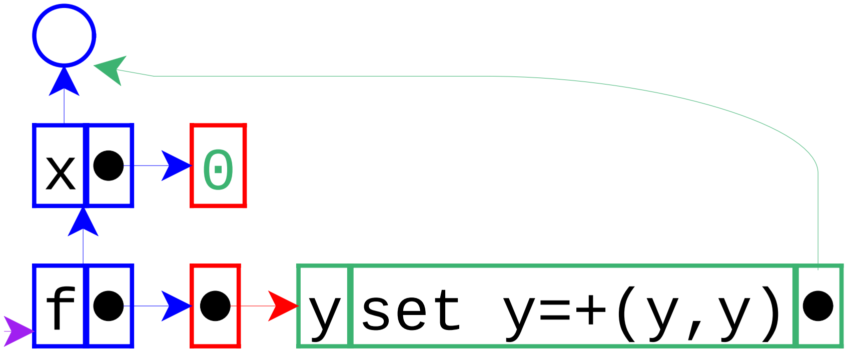
- Starting call-by-name combined with call-by-reference...

call-by-name/ref

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Eval RHSs

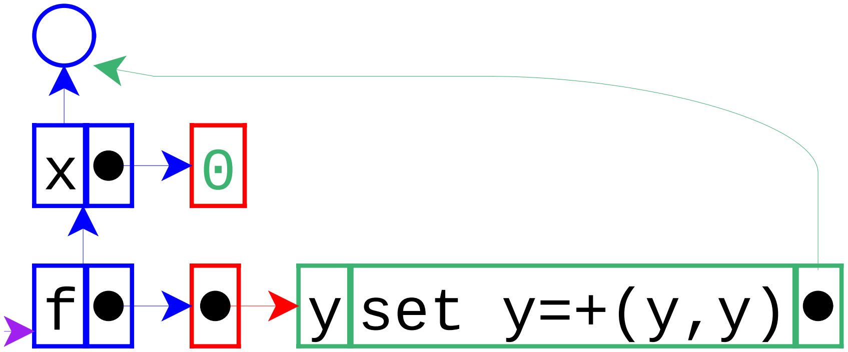
call-by-name/ref



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Simple expressions: bind x and f to 0 and closure, respectively

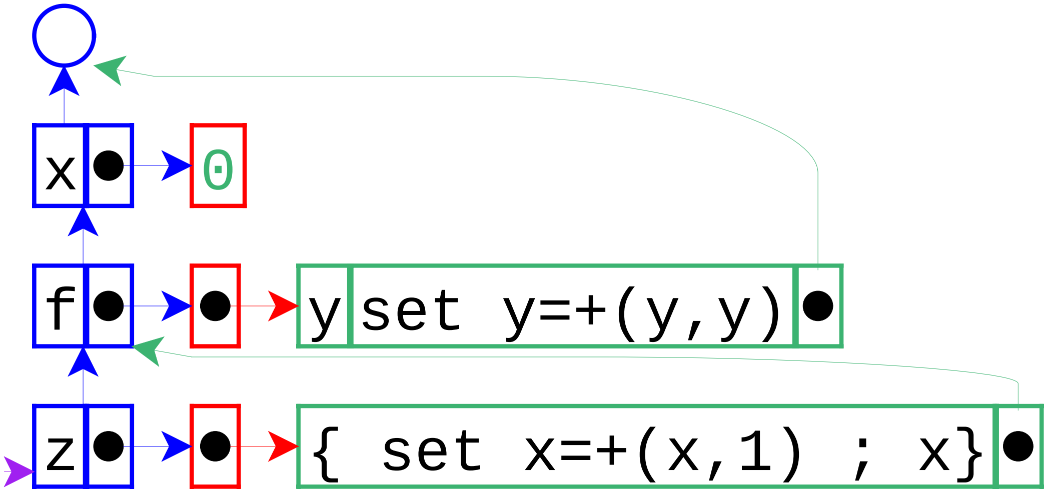
call-by-name/ref



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Handle RHS of z...

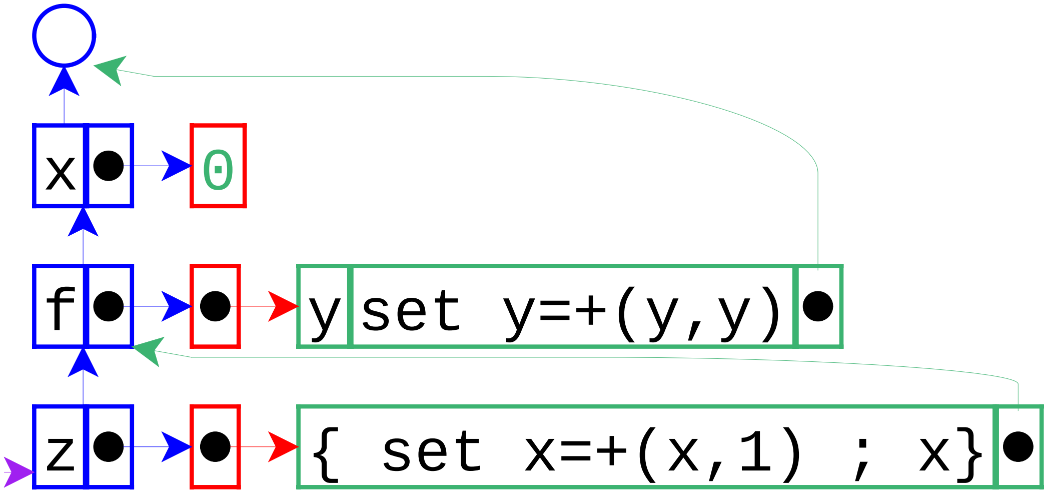
call-by-name/ref



```
  let x = 0
      f = proc(y) set y=+(y,y)
   in let z = { set x=+(x,1) ; x }
       in { (f z) ;  z }
```

- ... by creating a thunk for z

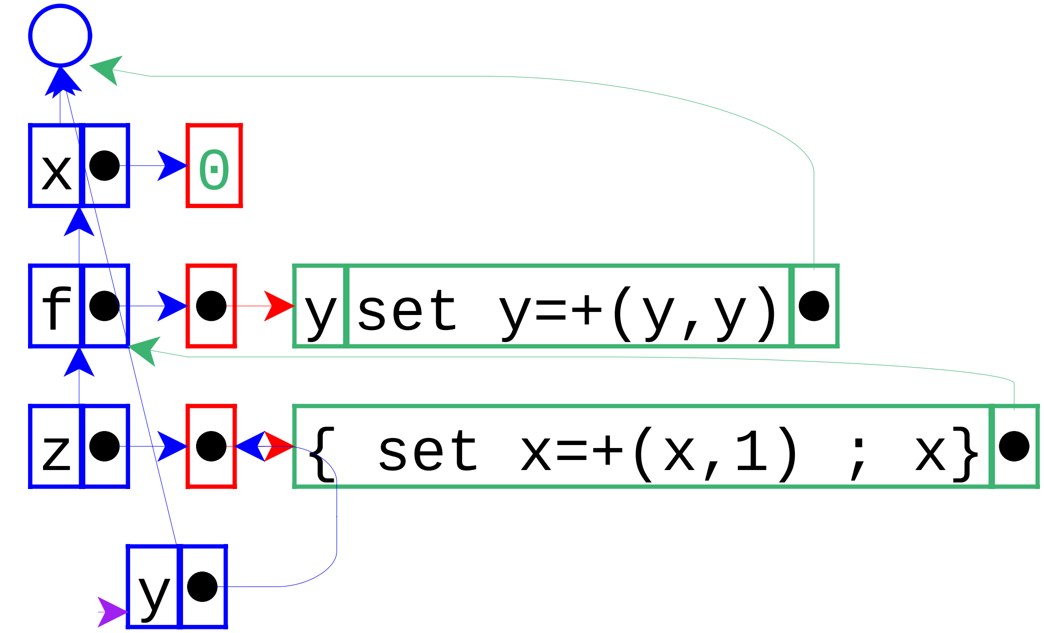
call-by-name/ref



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Call f with z

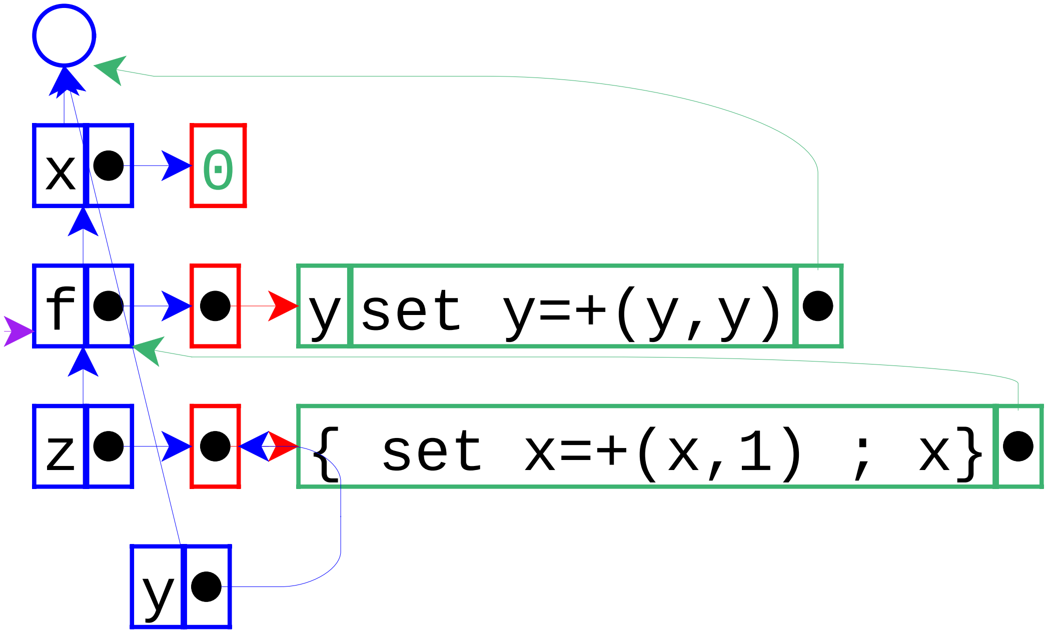
call-by-name/ref



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Call-by-reference shares location for z with y

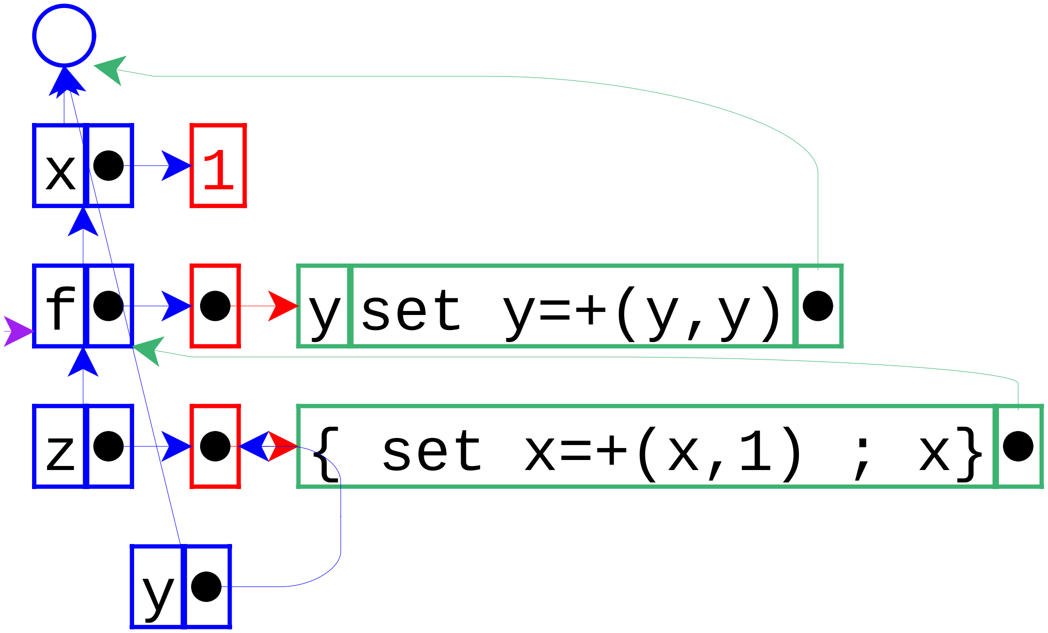
call-by-name/ref



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

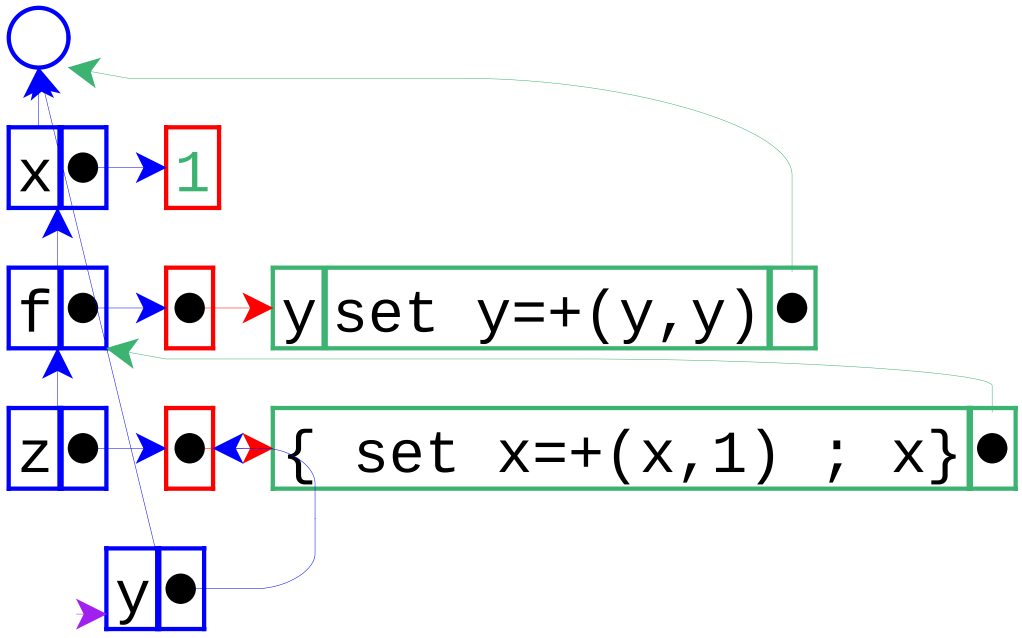
- First use of y triggers evaluation of the thunk

call-by-name/ref



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Thunk changes value of x to 1

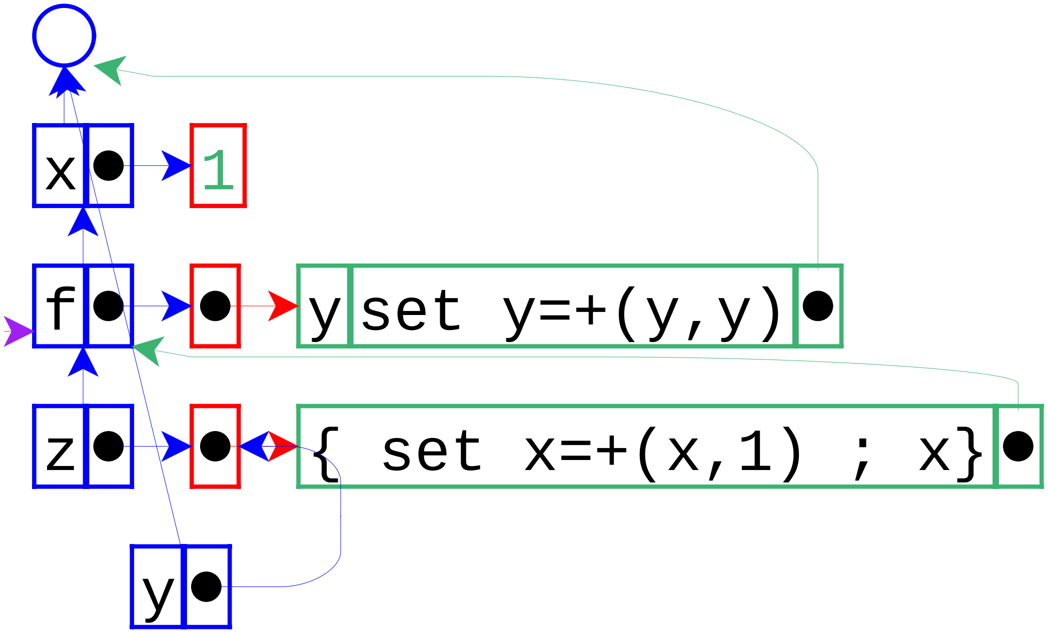


call-by-name/ref

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Result for first y is 1

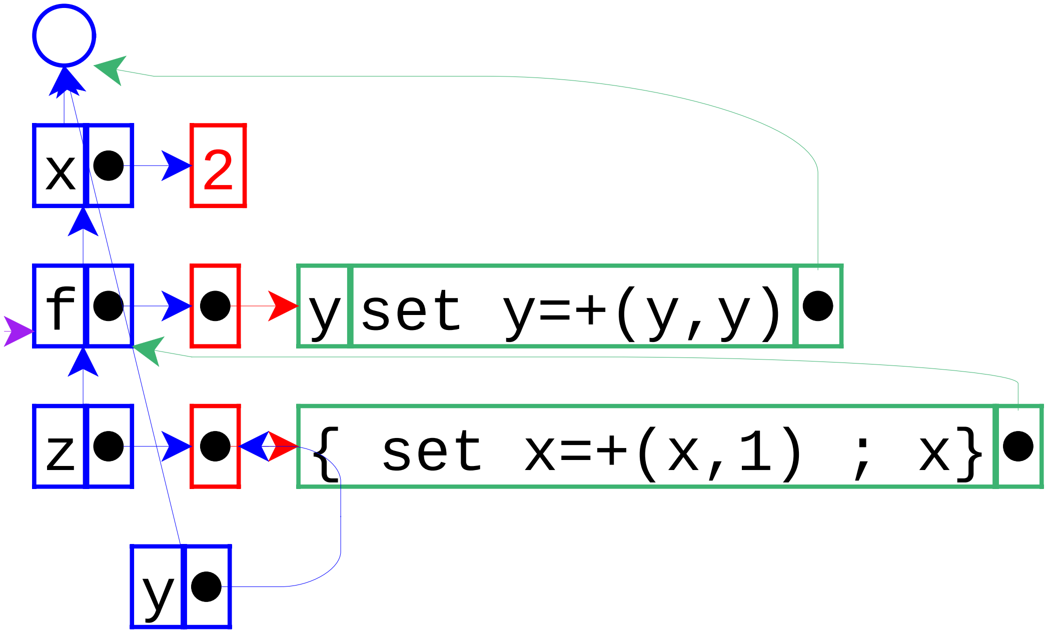
call-by-name/ref



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
      in { (f z) ;  z }
```

- Second use of y triggers evaluation of the thunk

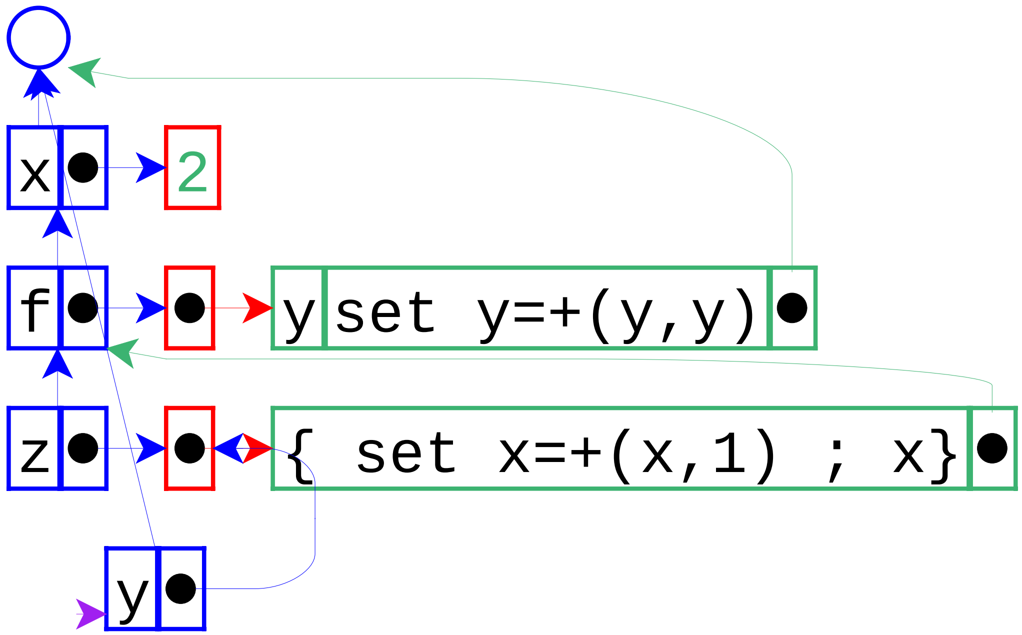
call-by-name/ref



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

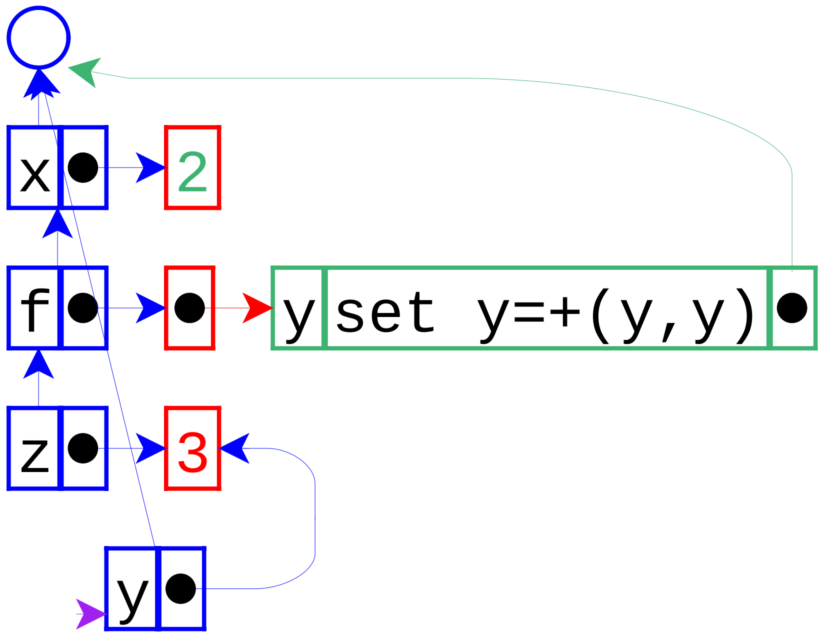
- Thunk changes value of x to 2

call-by-name/ref



```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

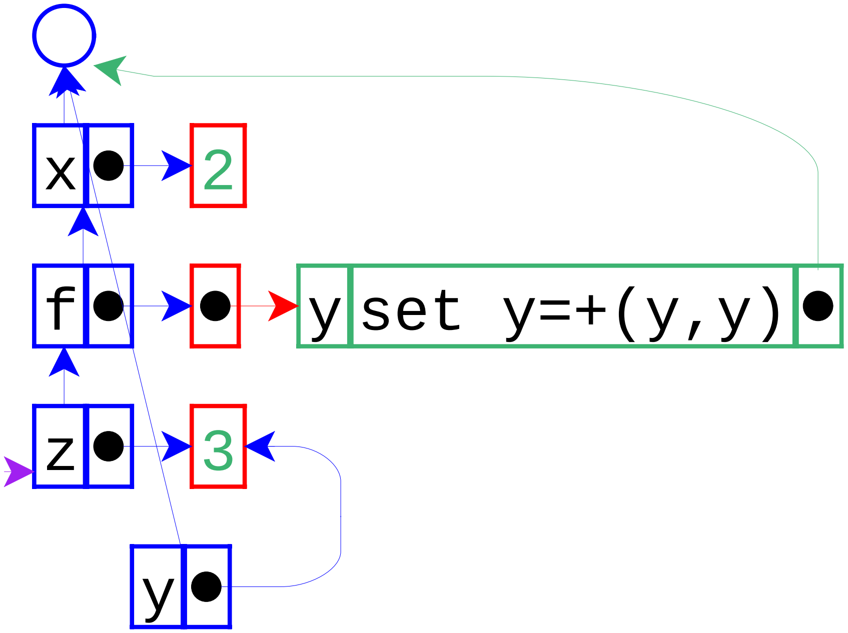
- Result for second y is 2



call-by-name/ref

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Set value of y to 3 (= 1+2)



call-by-name/ref

```
let x = 0
    f = proc(y) set y=+(y,y)
 in let z = { set x=+(x,1) ; x }
     in { (f z) ;  z }
```

- Final result is the value of z: 3