Assignment in Scheme

So far, we have one form of assignment: **vector-set!**

```
(let ([v (vector 1 2 3)])
  (begin
    (vector-set! v 1 72)
    v))
```

$\rightarrow\rightarrow$

```
#(1 72 3)
```

Assignment in Scheme

Scheme actually allows variables to be modified:

```
(let ([x 2])
  (begin
    (set! x 73)
    x))
→→
73
```

- *Don't write Scheme code like that*, except for HW6
- But many languages have assignment, and need it

Assignment in the Book Language

- Add a **set** expression form:

<expr> ::= **set** <id> = <expr>

Evaluating with Assignment

Can't write this, since we don't have **begin** in our language

let x = 10
  y = 12
 in (**begin set x** = +(**x**,1)
    x)

Evaluating with Assignment

Instead, use a binding for a dummy variable **d** to sequence expressions; initial environment is empty

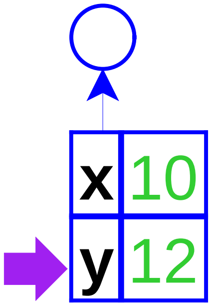
```
let x = 10
    y = 12
  in let d = set x = +(x,1)
      in x
```

Evaluating with Assignment

Eval RHS (right-hand side) of the let expression

let x = 10
y = 12
in let d = **set x** = +(**x**,1)
in x

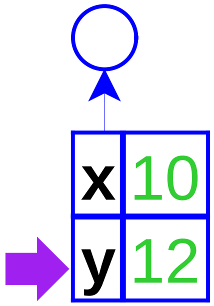
Evaluating with Assignment



Extend the current environment with **x** and **y**, and eval body

```
let x = 10
    y = 12
  in let d = set x = +(x,1)
       in x
```

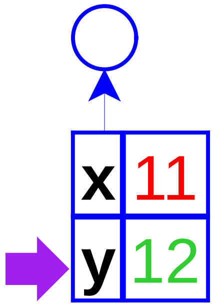
Evaluating with Assignment



Eval RHS of the let expression

let x = 10
  y = 12
 in let d = **set x** = +(**x**,1)
   in x

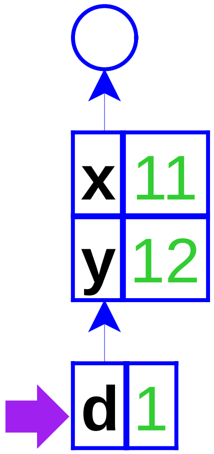
Evaluating with Assignment



It modifies the **x** in the current lexical scope; we define **set** to always return 1

let x = 10
y = 12
in let d = **set x** = +(**x**,1)
in x

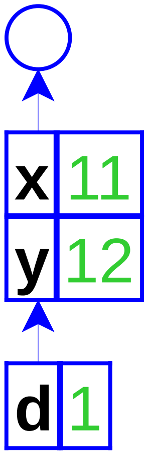
Evaluating with Assignment



Bind **d** to the result 1; to eval the body, **x**, we look it up in the environment as usual, and find 11

let x = 10
    y = 12
  in let d = **set x** = +(**x**,1)
      in **x**

Evaluating with Assignment



➤ Variables now correspond to boxes in the environment, not fixed values

let x = 10
y = 12
in let d = **set x** = +(**x**,1)
in x

Expressed and Denoted Values

<expval> ::= <num>
::= <proc>
<denval> ::= <reference>

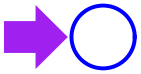
- New datatype:

```
(define-datatype reference reference?
  (a-ref (pos integer?)
         (vec vector?)))
```

- New function:

```
apply-env-ref : env sym -> ref
```

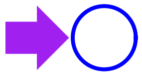
Assignment and Closures



An example with **proc**; again, we start with the empty environment

```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

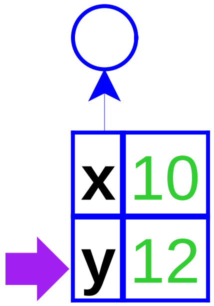
Assignment and Closures



Eval RHS of the let expression

```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

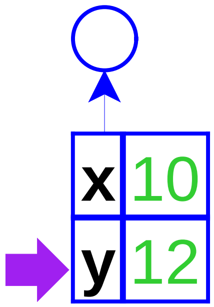
Assignment and Closures



Extend the current environment with **x** and **y**, and eval body

let x = 10
  y = 12
 in let f = **proc**(**z**)+(**z**,**x**)
  in let d = **set x** = +(**x**,1)
   in (**f** 0)

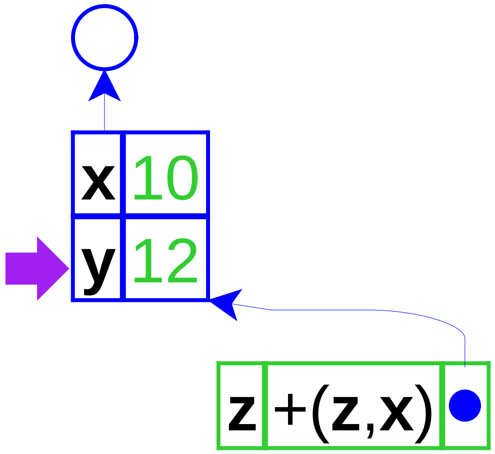
Assignment and Closures



Eval RHS of the let expression...

let x = 10
   y = 12
 in let f = **proc**(**z**)+(**z**,**x**)
   in let d = **set x** = +(**x**,1)
     in (**f** 0)

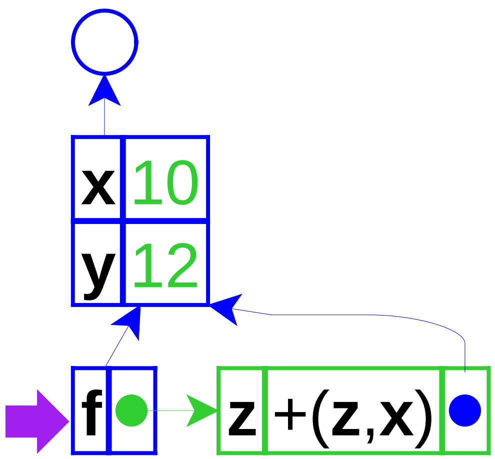
Assignment and Closures



... which creates a closure, pointing to the current environment

let x = 10
  y = 12
 in let f = **proc(z)+(z,x)**
   in let d = **set x** = +(**x**,1)
    in (**f** 0)

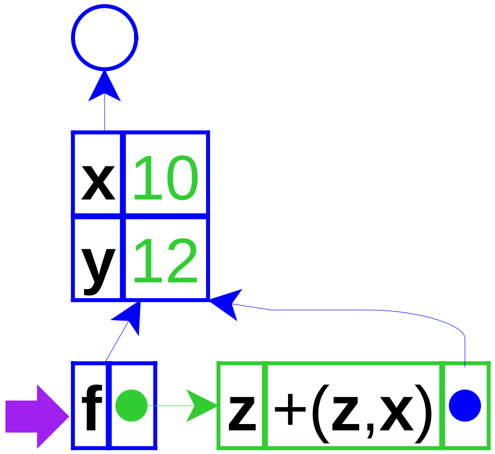
Assignment and Closures



To finish the **let**, the environment is extended with **f** bound to the closure; then evaluate the body

let x = 10
y = 12
in let f = **proc**(**z**)+(**z**,**x**)
in let d = **set x** = +(**x**,1)
in (**f** 0)

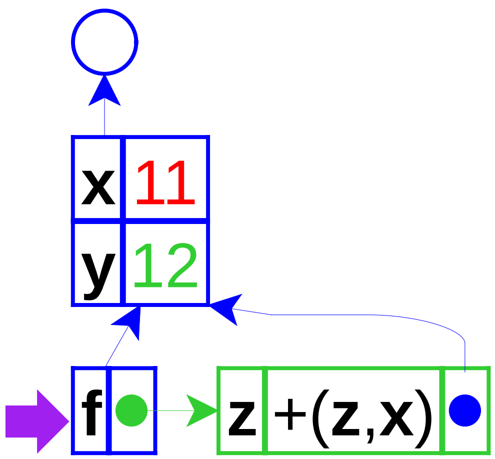
Assignment and Closures



Eval RHS of the let expression...

let x = 10
    y = 12
  in let f = **proc**(**z**)+(**z**,**x**)
      in let d = **set x** = +(**x**,1)
        in (**f** 0)

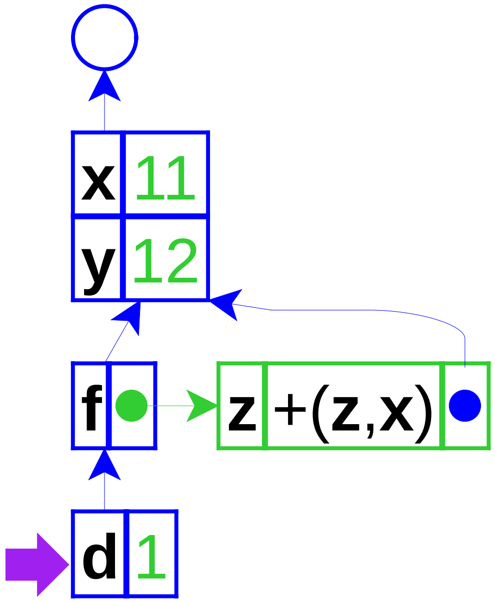
Assignment and Closures



... which changes the value of **x**, then produces 1

```
let x = 10
   y = 12
 in let f = proc(z)+(z,x)
     in let d = set x = +(x,1)
         in (f 0)
```

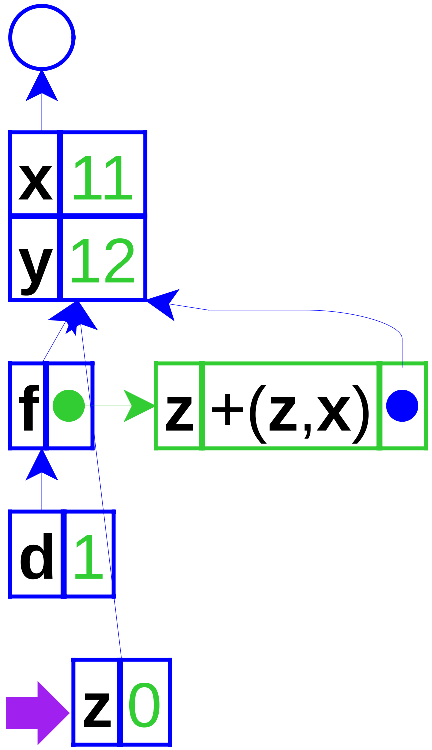
Assignment and Closures



To eval the body, (**f** 0), we look up **f** in the environment to find a closure, and evaluate 0 to 0

let x = 10
y = 12
in let f = **proc**(**z**)+(**z**,**x**)
in let d = **set x** = +(**x**,1)
in (**f** 0)

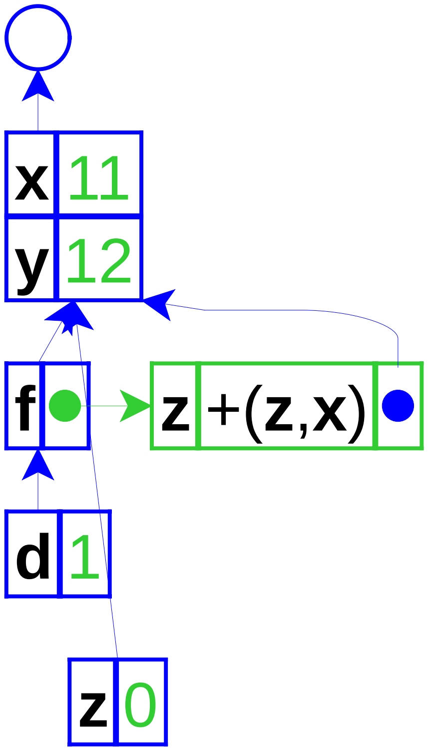
Assignment and Closures



Extend the *closure's* environment with 0 for **z**, and evaluate the closure's body in that environment; the result will be 11

let x = 10
y = 12
in let f = **proc**(**z**)+(**z**,**x**)
in let d = **set x** = +(**x**,1)
in (**f** 0)

Assignment and Closures



let x = 10
y = 12
in let f = **proc**(**z**)+(**z**,**x**)
in let d = **set x** = +(**x**,1)
in (**f** 0)

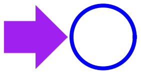
- By capturing environments, closures capture variables that may change

Assignment and Arguments

Another example with **proc**, but with the **let** inside the **proc**

```
let f = proc(z)
            let x = 10
             in let d = set x = +(x,z)
                 in x
 in +((f 1), (f 9))
```

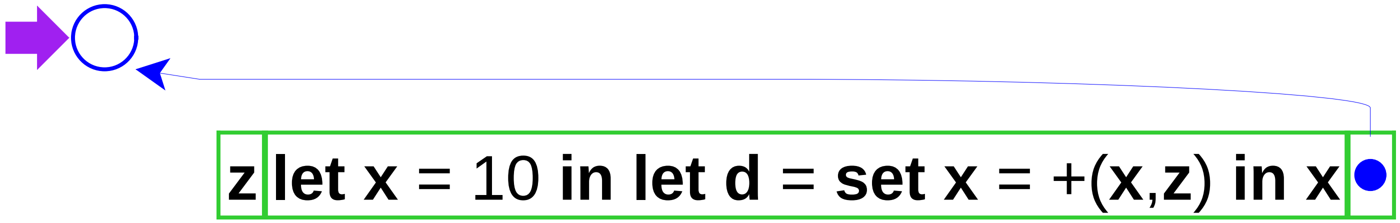
Assignment and Arguments



Eval RHS of the let expression...

```
let f = proc(z)
           let x = 10
            in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

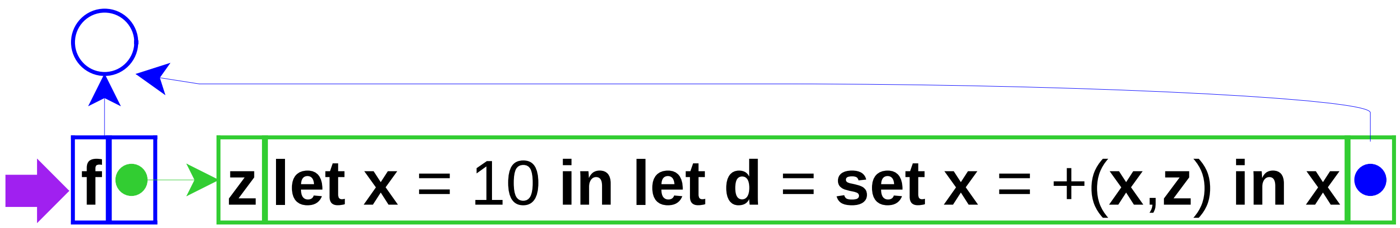
Assignment and Arguments



... which creates a closure, pointing to the current environment

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

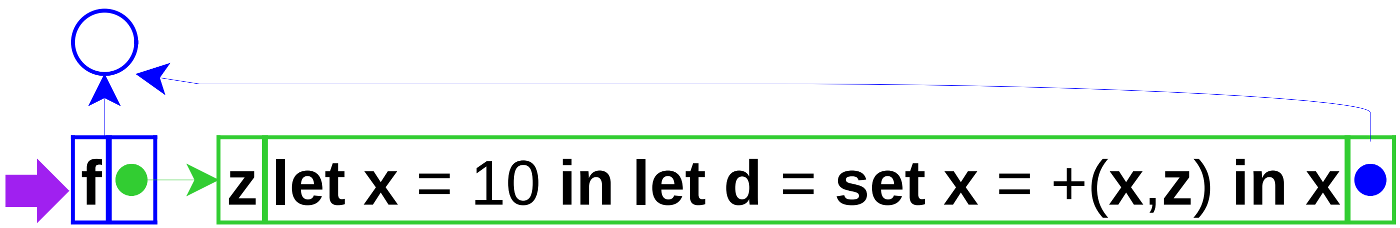
Assignment and Arguments



Bind the closure to **f** and eval the body

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

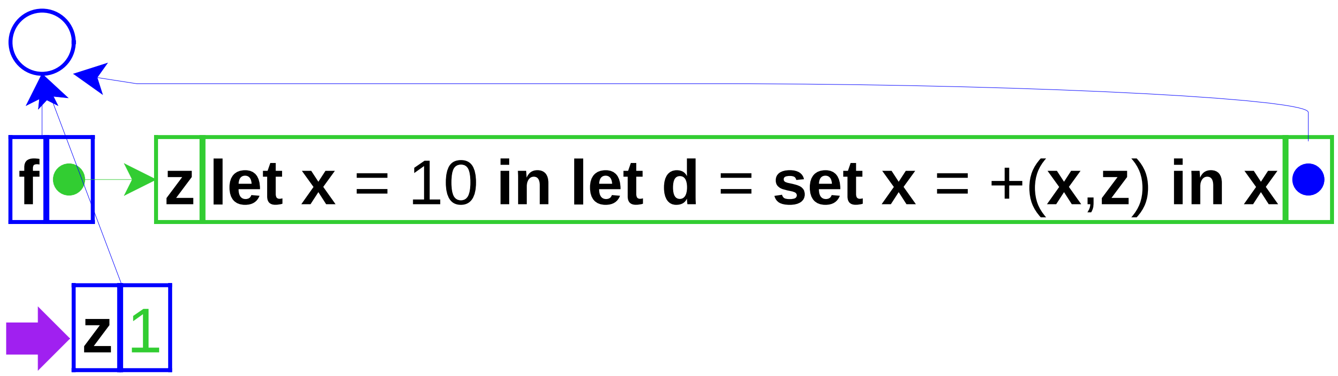
Assignment and Arguments



Evaluate the first operand, (**f** 1)

let f = **proc**(**z**)
let x = 10
in let d = **set x** = +(**x**,**z**)
in x
in +((**f** 1), (**f** 9))

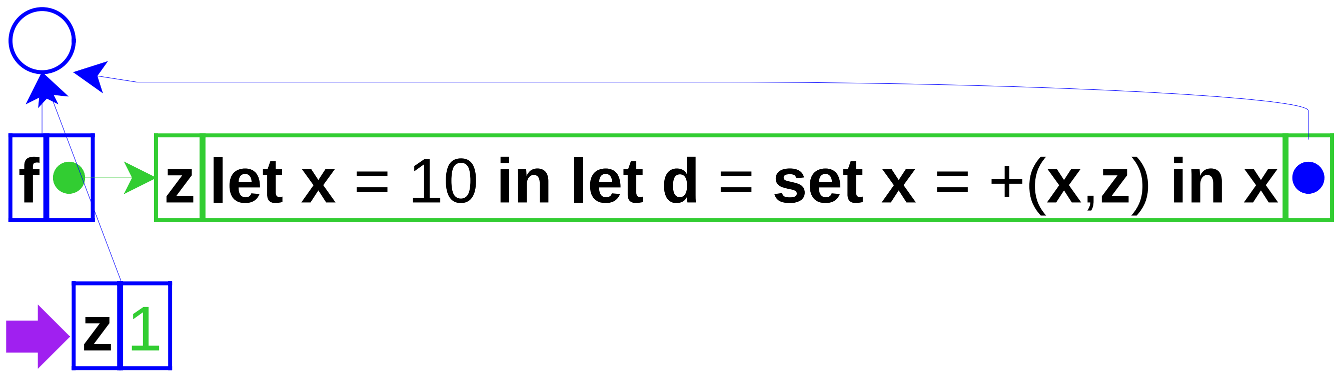
Assignment and Arguments



Take the closure for **f**, extend its environment with a binding for **z**, and eval the closure's body

let f = **proc**(**z**)
let x = 10
in let d = **set x** = +(**x**,**z**)
in x
in +((**f** 1), (**f** 9))

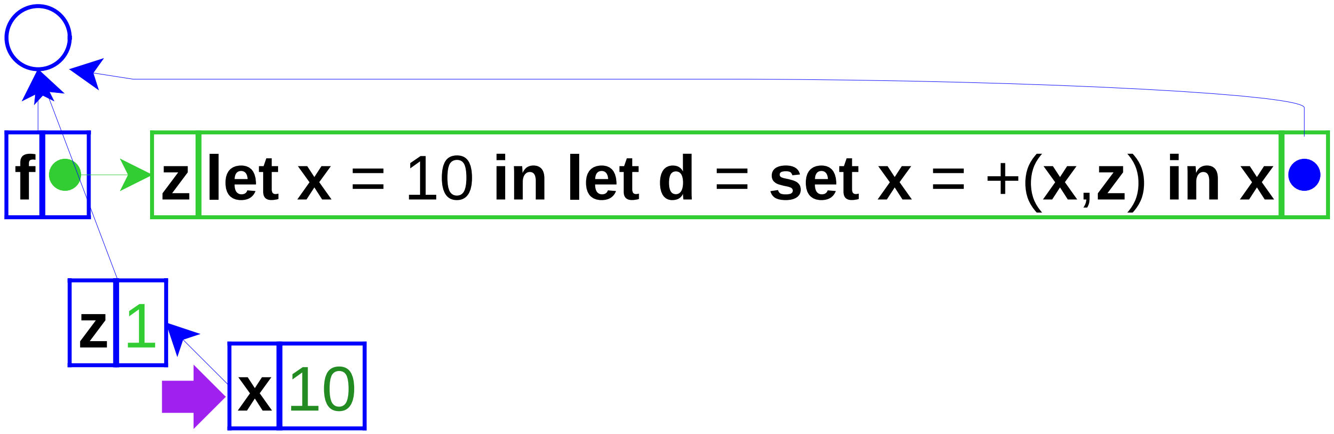
Assignment and Arguments



Eval the RHS

let f = **proc**(**z**)
let x = 10
in let d = **set x** = +(**x**,**z**)
in x
in +((**f** 1), (**f** 9))

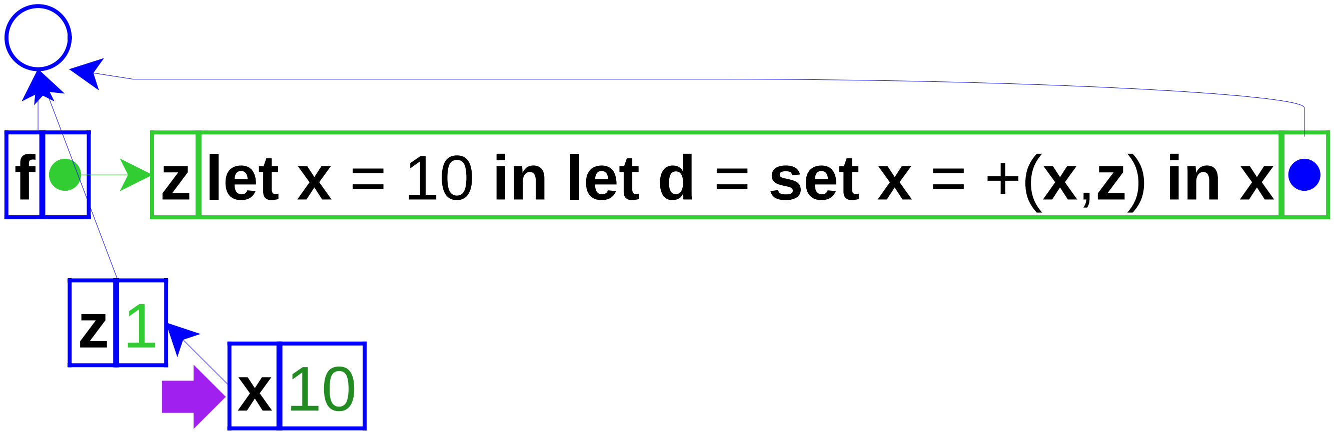
Assignment and Arguments



Add the binding for **x** and eval the inner body

let f = **proc**(**z**)
let x = 10
in let d = **set x** = +(**x**,**z**)
in x
in +((**f** 1), (**f** 9))

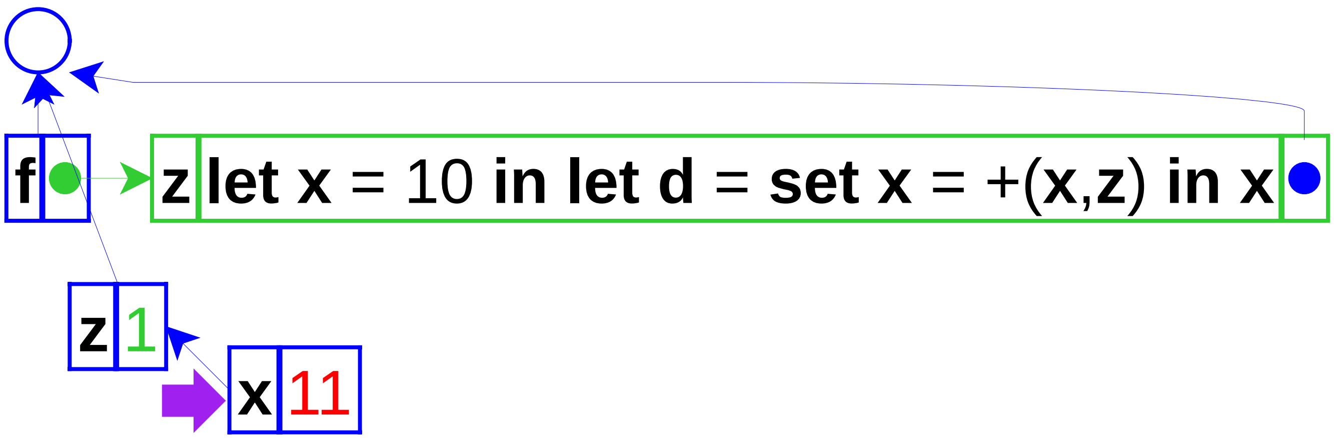
Assignment and Arguments



Eval RHS...

let f = **proc**(**z**)
let x = 10
in let d = **set x = +(x,z)**
in x
in +((**f** 1), (**f** 9))

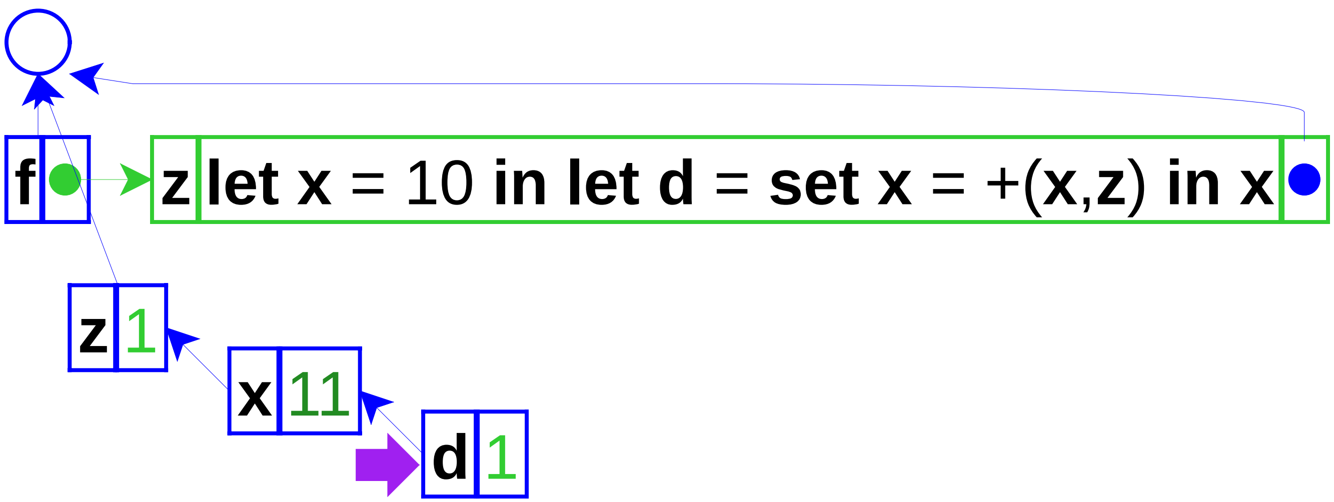
Assignment and Arguments



... which modifies the value of **x**

let f = **proc**(**z**)
let x = 10
in let d = **set x** = +(**x**,**z**)
in x
in +((**f** 1), (**f** 9))

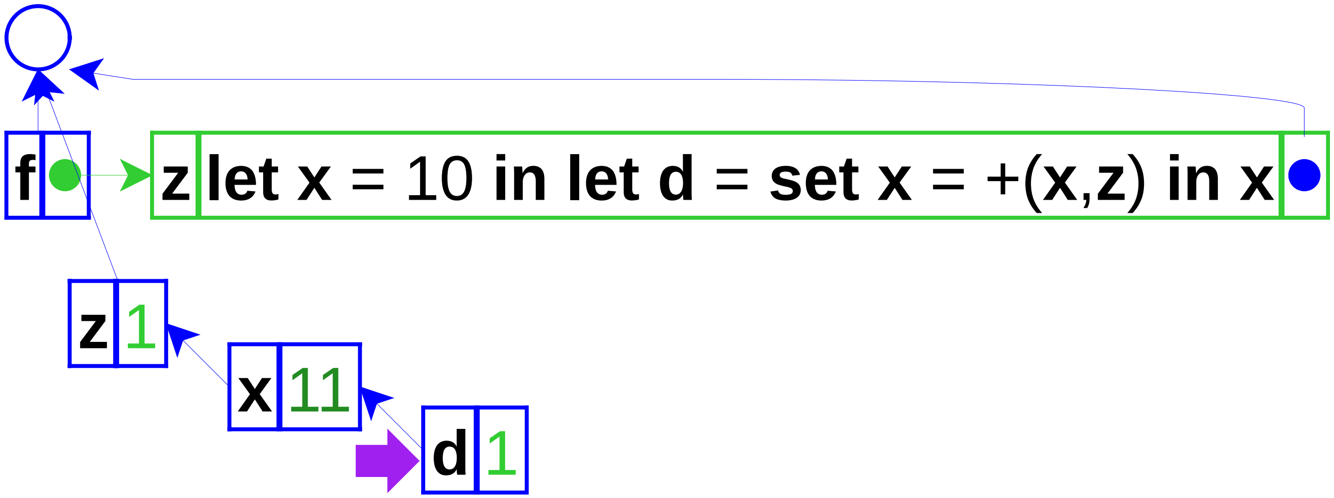
Assignment and Arguments



Bind **d** to 1 and evaluate **x**, which produces 11

let f = **proc**(**z**)
    let x = 10
    in let d = **set x** = +(**x**,**z**)
      in x
in +((**f** 1), (**f** 9))

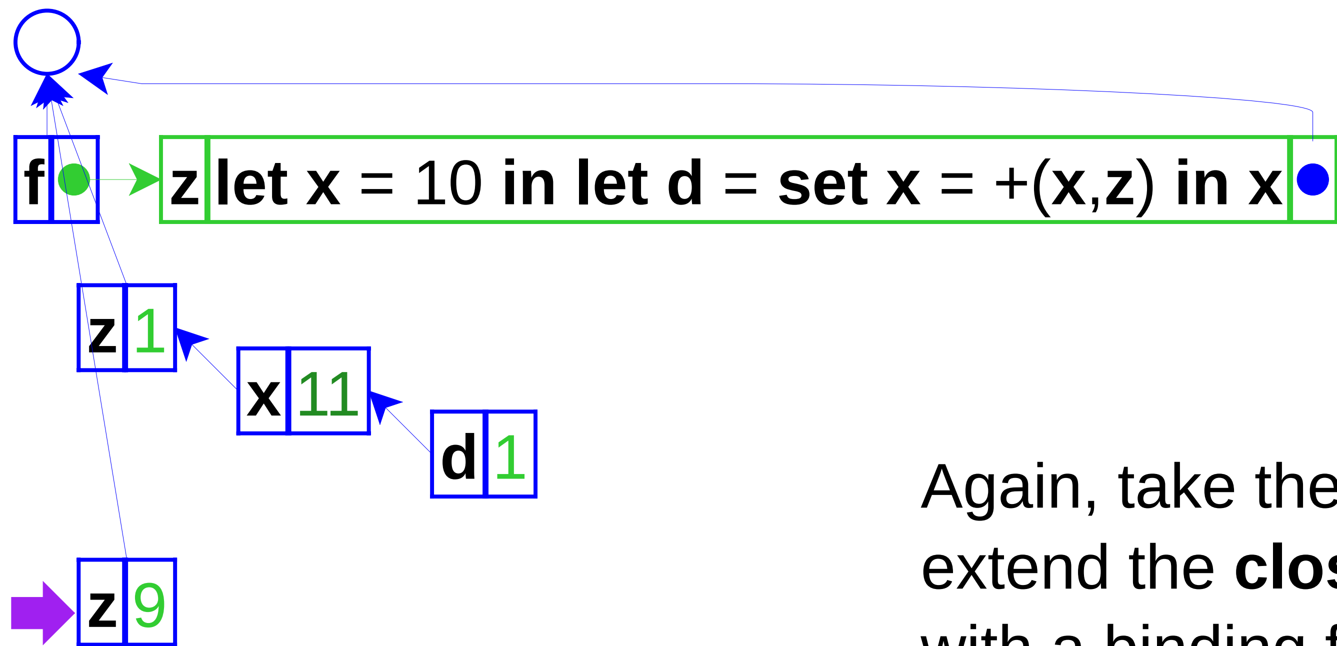
Assignment and Arguments



First operand is 11; now evaluate the second operand, (**f** 9)

let f = **proc**(**z**)
    let x = 10
    in let d = **set x** = +(**x**,**z**)
        in x
in +((**f** 1), (**f** 9))

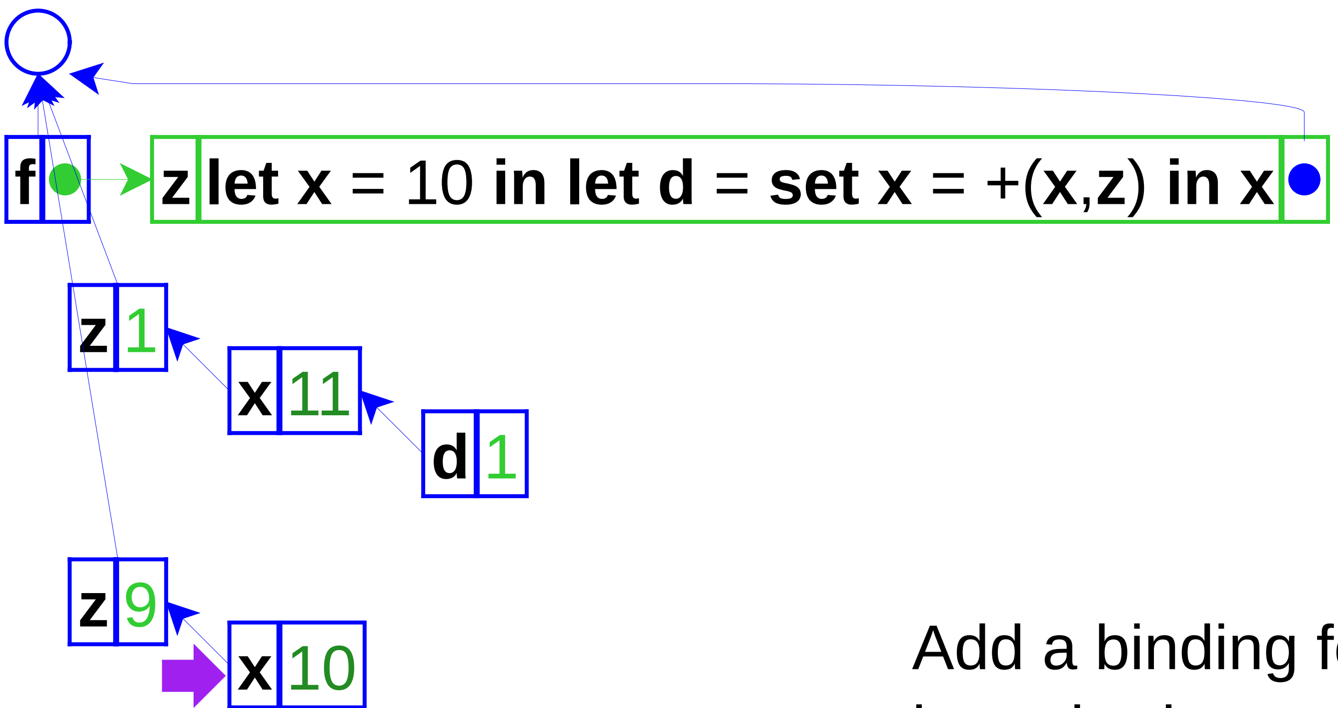
Assignment and Arguments



Again, take the closure for **f**, extend the **closure's** environment with a binding for **z**, and eval the closure's body

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

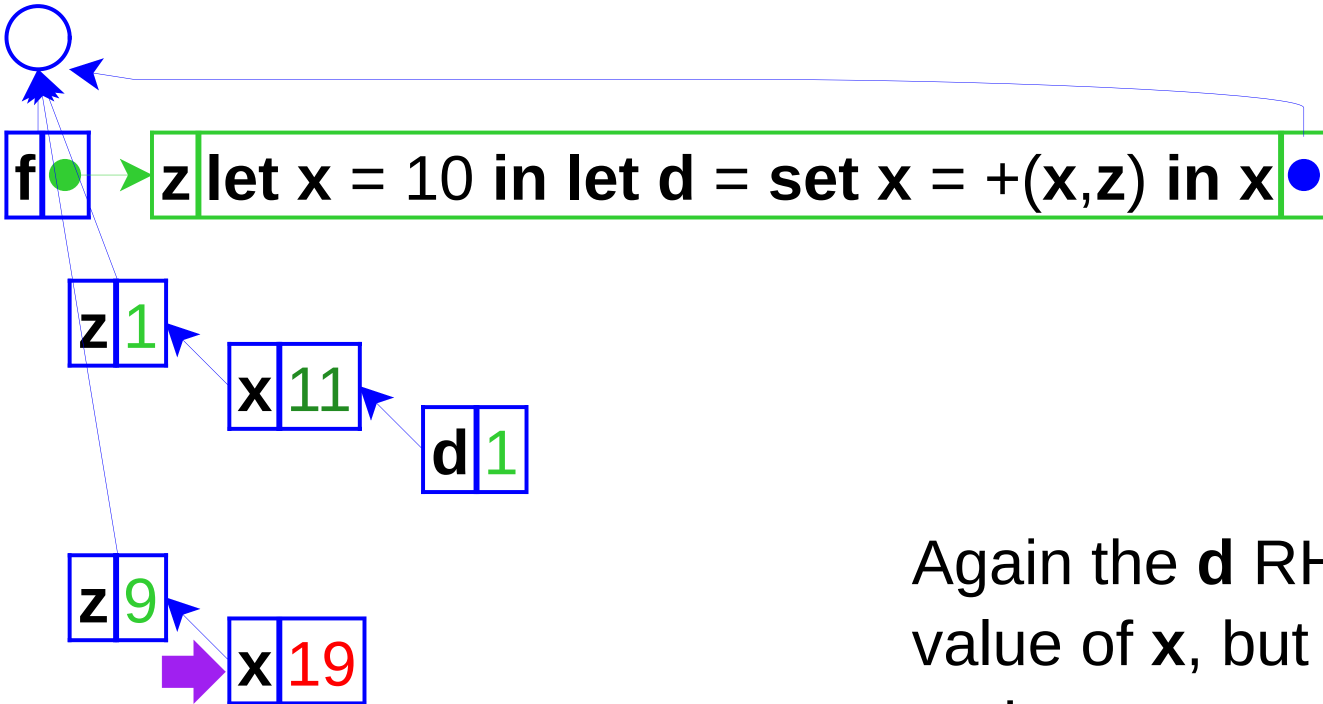
Assignment and Arguments



Add a binding for **x**, then eval the inner body

let f = **proc**(**z**)
let x = 10
in let d = **set x** = +(**x**,**z**)
in x
in +((**f** 1), (**f** 9))

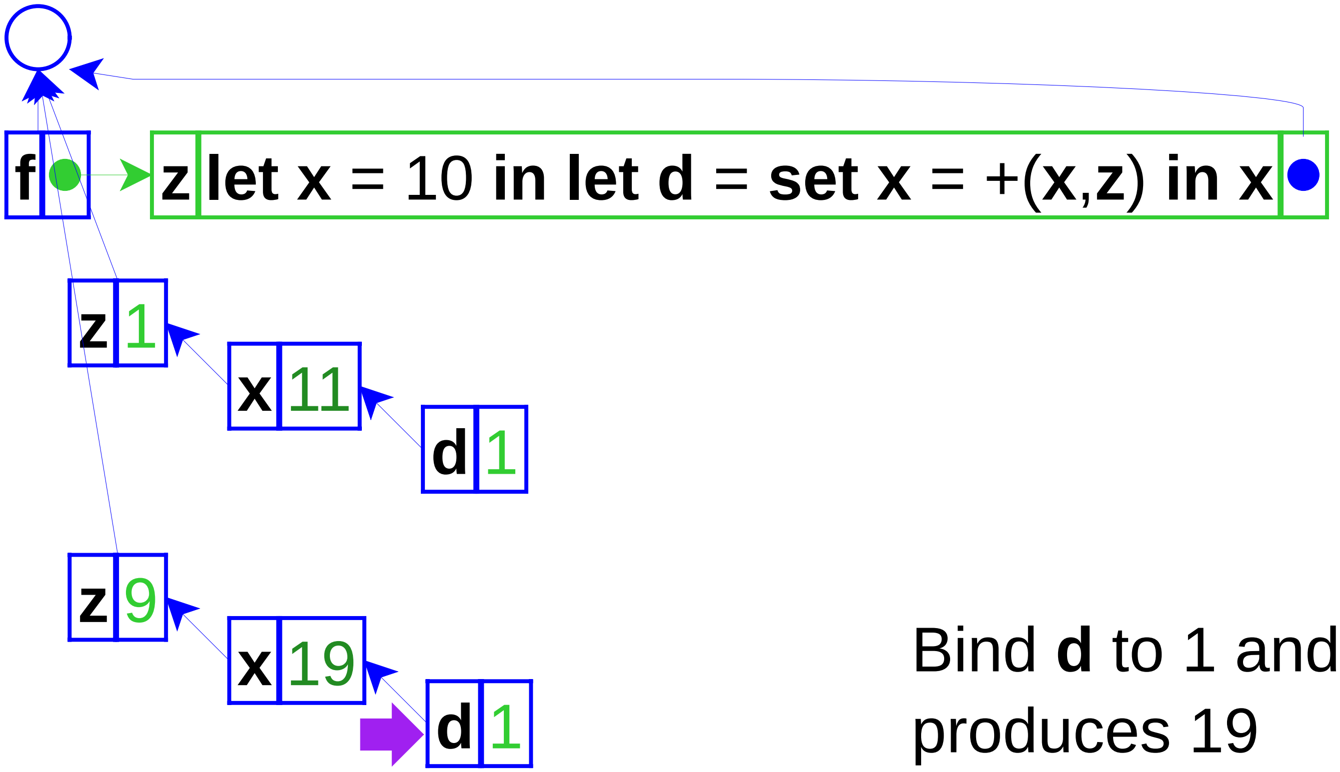
Assignment and Arguments



Again the **d** RHS modifies the value of **x**, but using the new **z** and **x**

let f = **proc**(**z**)
let x = 10
in let d = **set x = +(x,z)**
in x
in +((**f** 1), (**f** 9))

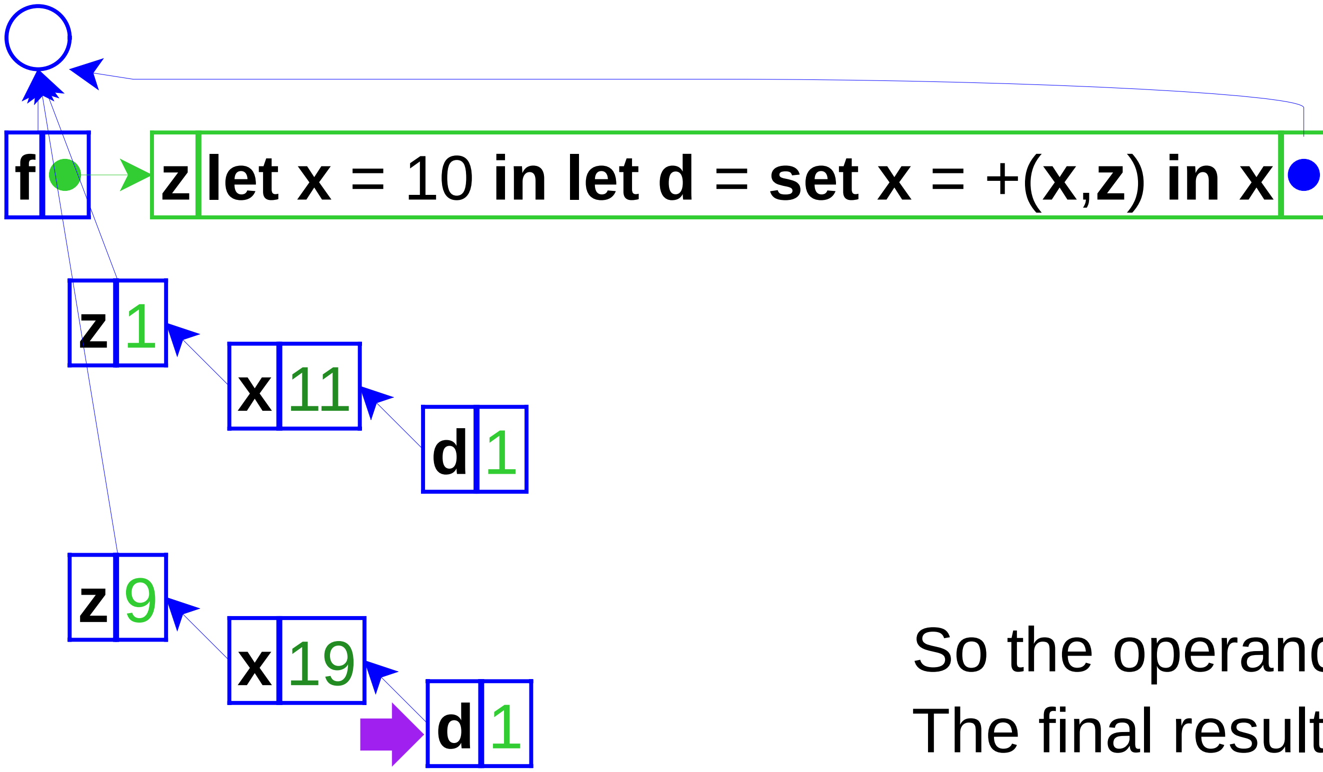
Assignment and Arguments



Bind **d** to 1 and evaluate **x**, which produces 19

let f = **proc**(**z**)
let x = 10
in let d = **set x** = +(**x**,**z**)
in x
in +((**f** 1), (**f** 9))

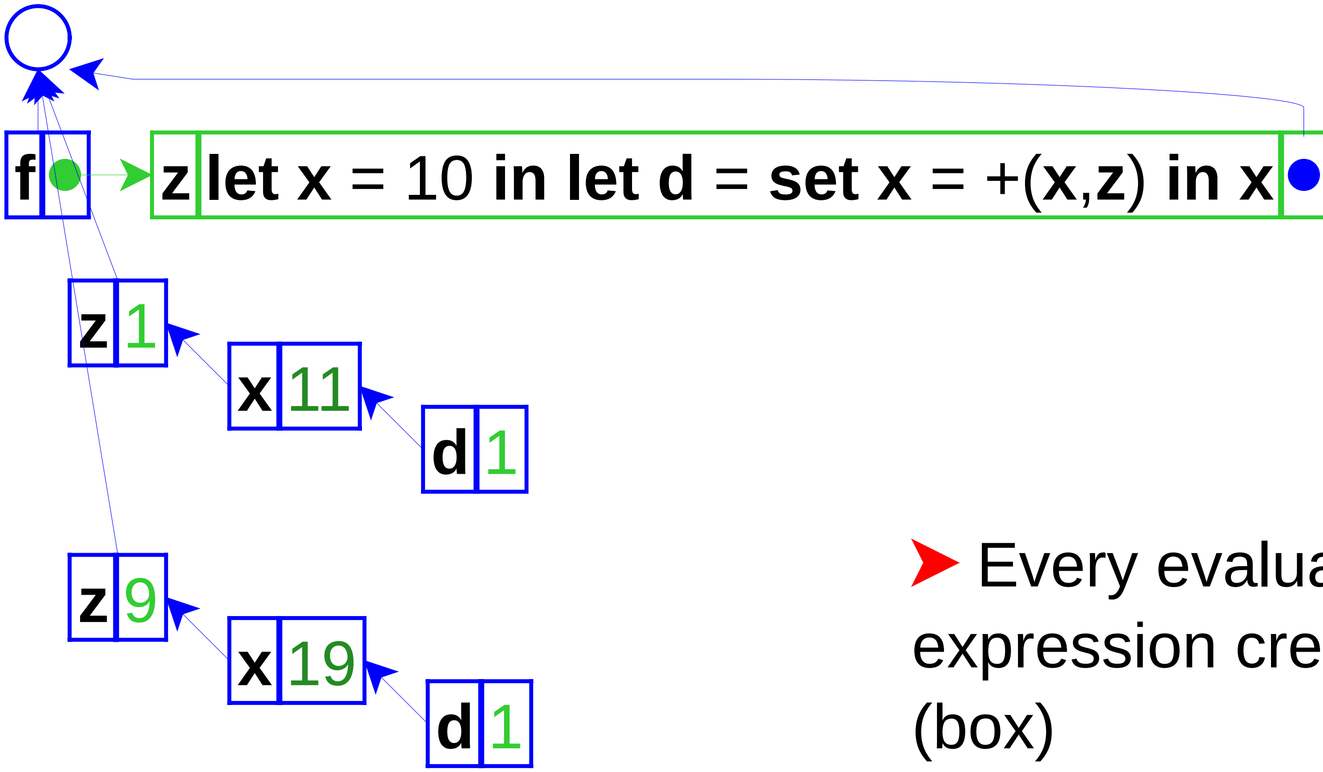
Assignment and Arguments



So the operands are 11 and 19;
The final result is 30

let f = **proc**(**z**)
let x = 10
in let d = **set x** = +(**x**,**z**)
in x
in +((**f** 1), (**f** 9))

Assignment and Arguments



- Every evaluation of a binding expression creates a new variable (box)

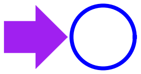
```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

Assignment and Locals within Procedures

An example with a procedure in a procedure

```
let mk = proc(x) proc(z)
                 let d = set x = +(x,z) in x
 in let f = (mk 10)
    in let g = (mk 12)   in ...
```

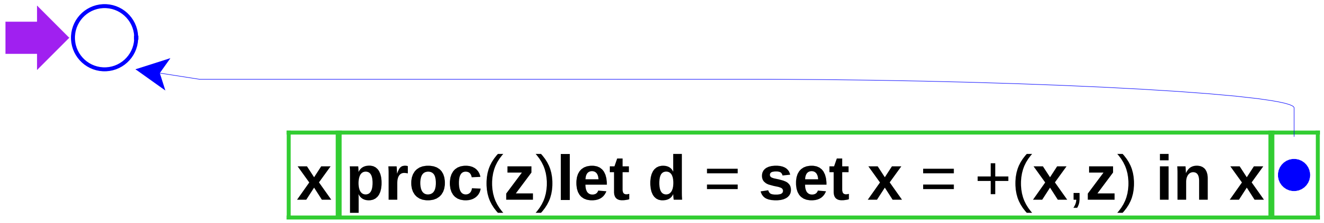
Assignment and Locals within Procedures



Eval RHS of the let expression...

```
let mk = proc(x) proc(z)
             let d = set x = +(x,z) in x
 in let f = (mk 10)
    in let g = (mk 12)   in ...
```

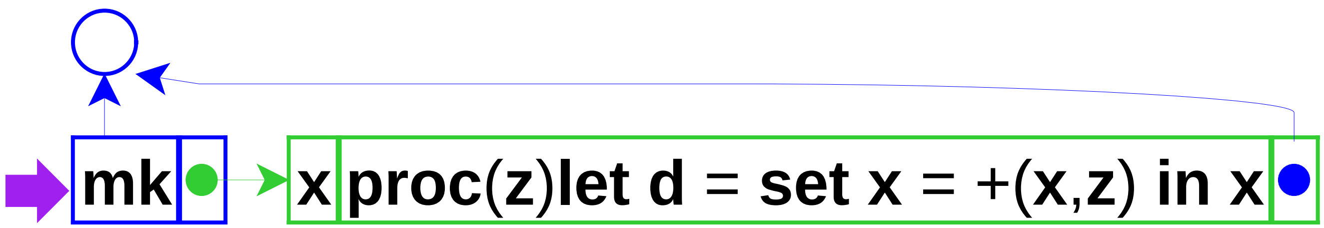
Assignment and Locals within Procedures



... which creates a closure, pointing to the current environment

let mk = **proc(x) proc(z)**
let d = **set x** = **+(x,z) in x**
in let f = (**mk** 10)
in let g = (**mk** 12) **in ...**

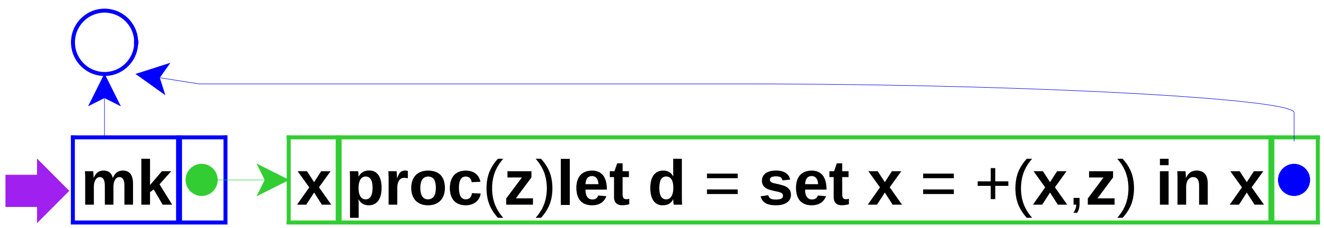
Assignment and Locals within Procedures



To finish the **let**, the environment is extended with **mk** bound to the closure, then evaluate the body

let mk = **proc**(**x**) **proc**(**z**)
let d = **set x** = +(**x**,**z**) **in x**
in let f = (**mk** 10)
in let g = (**mk** 12) **in ...**

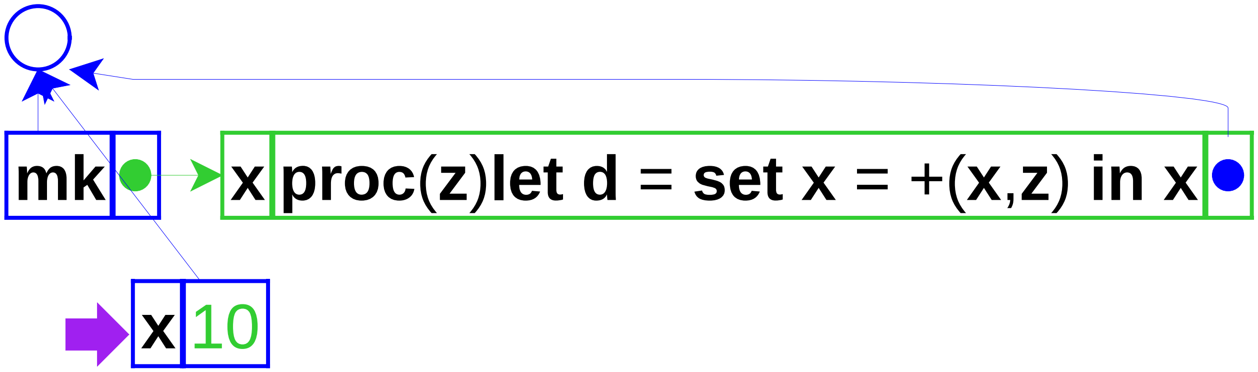
Assignment and Locals within Procedures



Eval RHS, a function call; look up **mk**...

let mk = **proc**(**x**) **proc**(**z**)
let d = **set x** = +(**x**,**z**) **in x**
in let f = (**mk** 10)
in let g = (**mk** 12) **in ...**

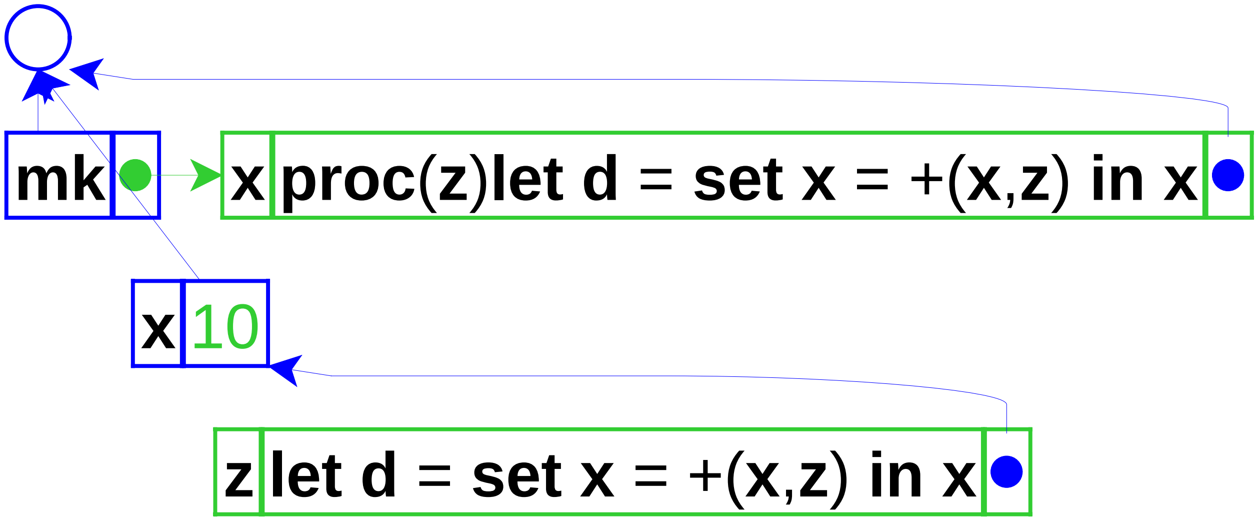
Assignment and Locals within Procedures



It's a closure, so extend the closure's environment with 10, and eval the closure's body

let mk = **proc**(**x**) **proc(z)**
let d = set x = +(x,z) in x
in let f = (**mk** 10)
in let g = (**mk** 12) **in ...**

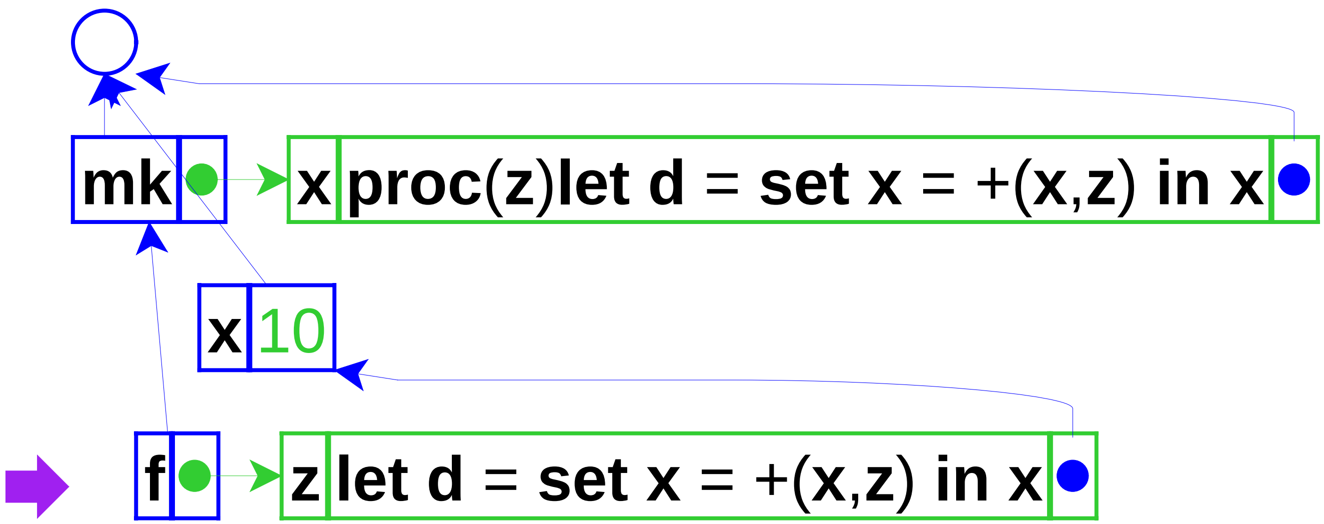
Assignment and Locals within Procedures



Note that the variable **x** is in the closure's environment

let mk = **proc**(**x**) **proc(z)**
let d = **set x** = +(**x**,**z**) **in x**
in let f = (**mk** 10)
in let g = (**mk** 12) **in ...**

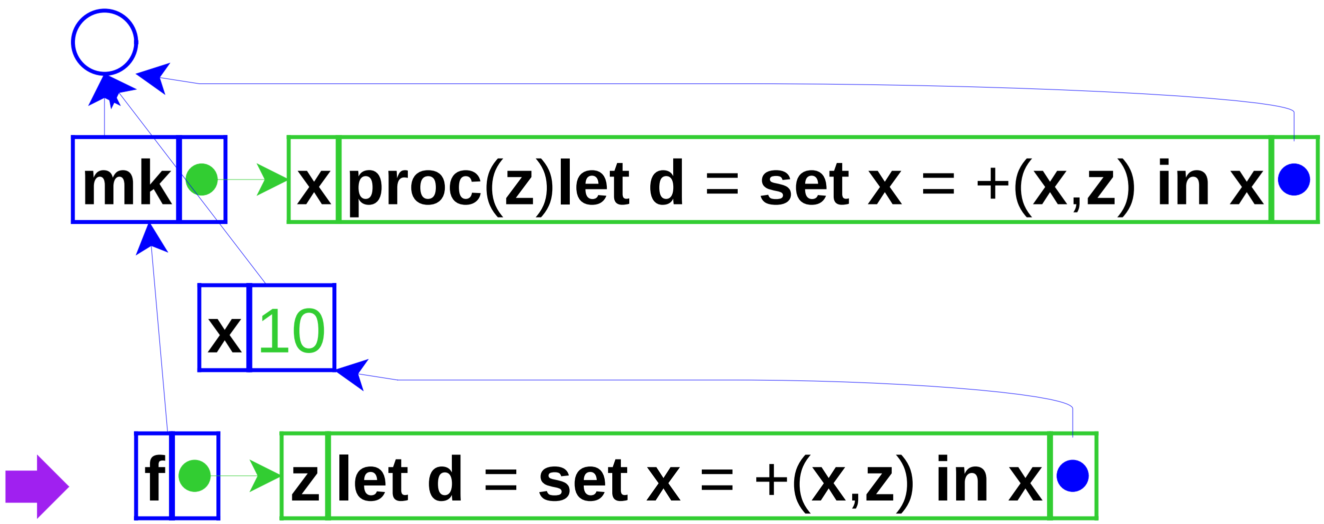
Assignment and Locals within Procedures



Bind **f** to the closure, and evaluate the body

let mk = **proc**(**x**) **proc**(**z**)
let d = **set x** = +(**x**,**z**) **in x**
in let f = (**mk** 10)
in let g = (**mk** 12) **in ...**

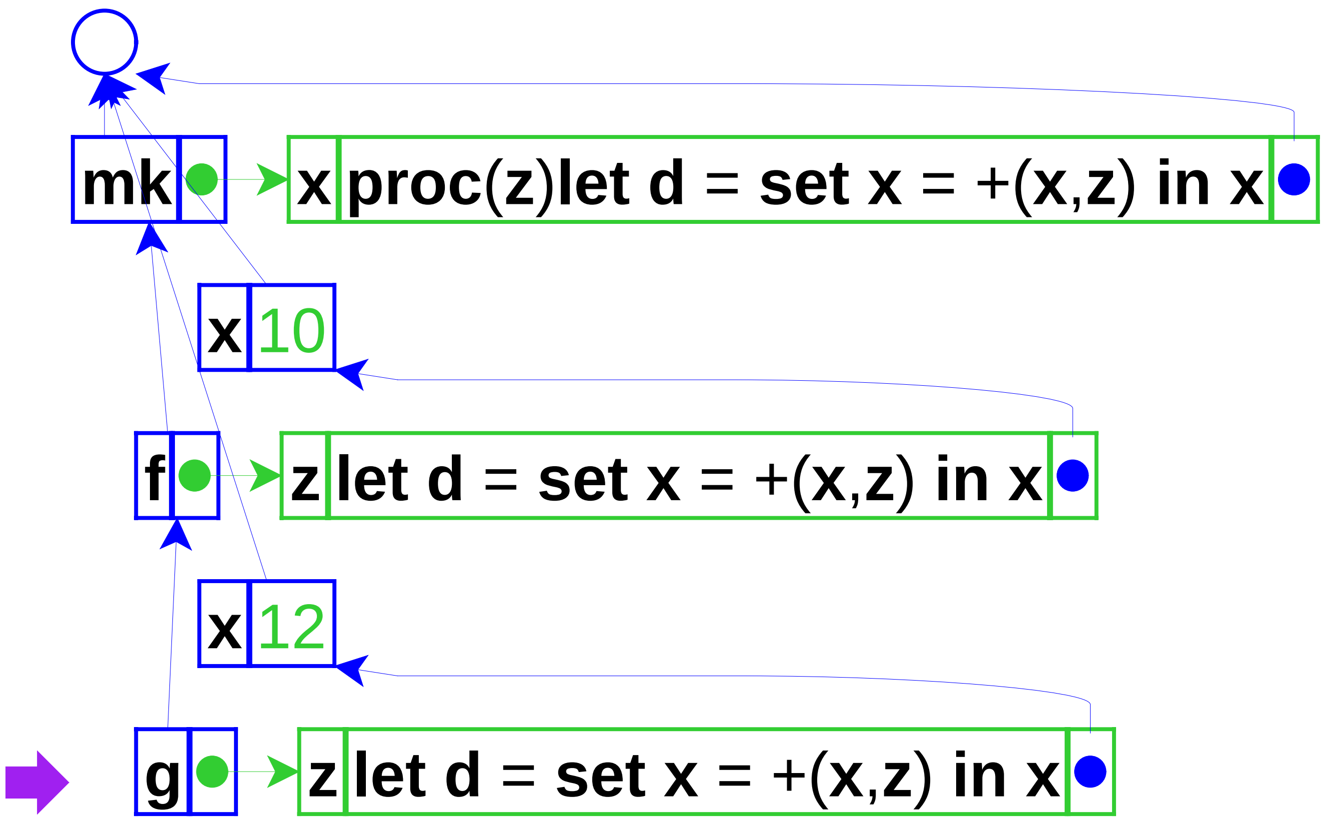
Assignment and Locals within Procedures



Eval RHS of the let expression, another call to **mk**; same as before...

let mk = **proc**(**x**) **proc**(**z**)
let d = **set x** = +(**x**,**z**) **in x**
in let f = (**mk** 10)
in let g = (**mk** 12) **in ...**

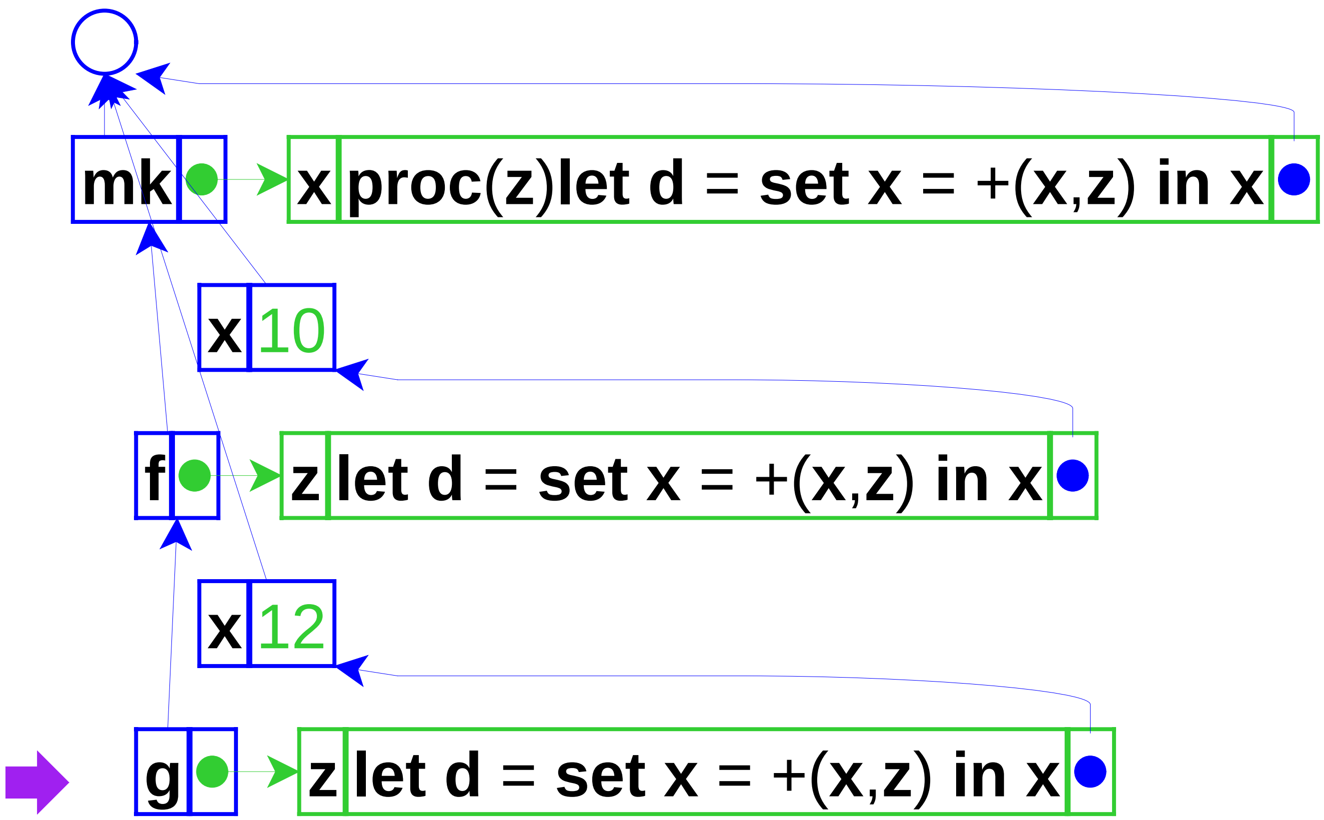
Assignment and Locals within Procedures



Extend **mk**'s env with a new **x** and get a closure, this time bound to **g**

let mk = **proc**(**x**) **proc**(**z**)
let d = **set x** = +(**x**,**z**) **in x**
in let f = (**mk** 10)
in let g = (**mk** 12) **in** ...

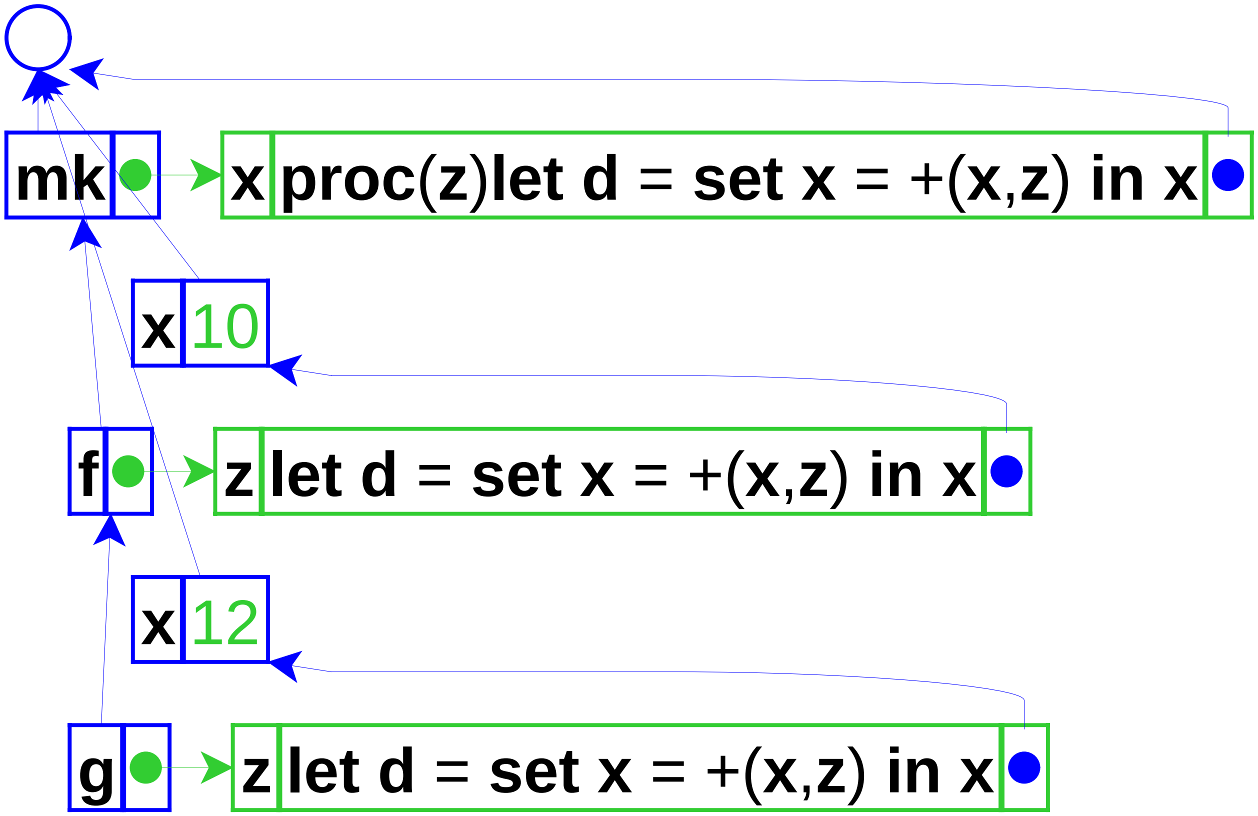
Assignment and Locals within Procedures



At this point, **f** and **g** have private versions of **x**

let mk = **proc**(**x**) **proc**(**z**)
let d = **set x** = +(**x**,**z**) **in x**
in let f = (**mk** 10)
in let g = (**mk** 12) **in** ...

Assignment and Locals within Procedures



let mk = **proc**(**x**) **proc**(**z**)
let d = **set x** = +(**x**,**z**) **in x**
in let f = (**mk** 10)
in let g = (**mk** 12) **in ...**

- Closures can capture generated variables, effectively getting private state

Assignment Summary

- Variables now denote references (a.k.a. locations), not values
- Lexical scope still works