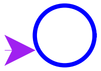
```
let x = 10
    y = 12
  in set x = +(x,1);
     x
```

Can’t write this, since we don’t have `;` in our language.

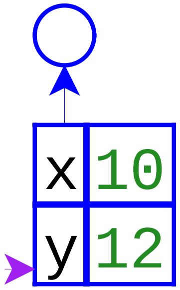
```
let x = 10
    y = 12
  in let d = set x = +(x,1)
      in x
```

Instead, use a binding for a dummy variable d to sequence expressions. Initial environment is empty.



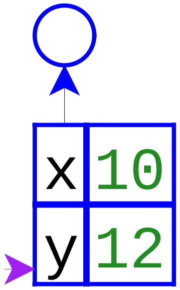
```
let x = 10
    y = 12
  in let d = set x = +(x,1)
      in x
```

Eval RHS (right-hand side) of the let expression. Purple part of program shows the current expression. Top area shows environments, with purple arrow to the current one.



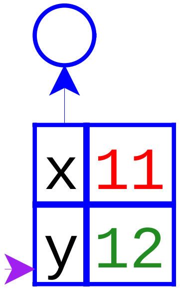

```
let x = 10
    y = 12
  in let d = set x = +(x,1)
       in x
```

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



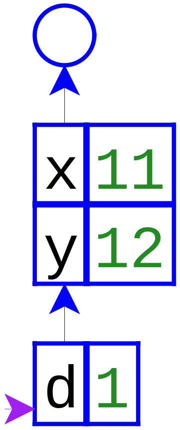

```
let x = 10
    y = 12
  in let d = set x = +(x,1)
      in x
```

Eval RHS of the let expression.



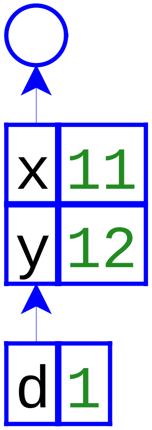

```
let x = 10
    y = 12
  in let d = set x = +(x,1)
       in x
```

It modifies the `x` in the current lexical scope. We define `set` to always return `1`.



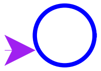

```
let x = 10
    y = 12
  in let d = set x = +(x,1)
      in x
```

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



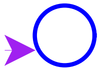

```
let x = 10
    y = 12
  in let d = set x = +(x,1)
      in x
```

The Point: Variables now correspond to boxes in the environment, not fixed values.



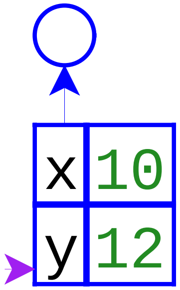
```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

An example with `proc`. Again, we start with the empty environment.



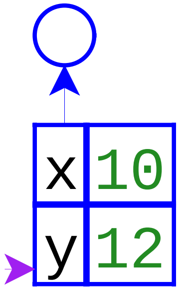
```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

Eval RHS of the let expression.



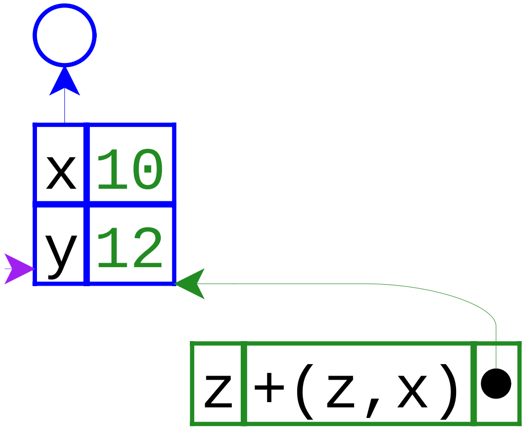

```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

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



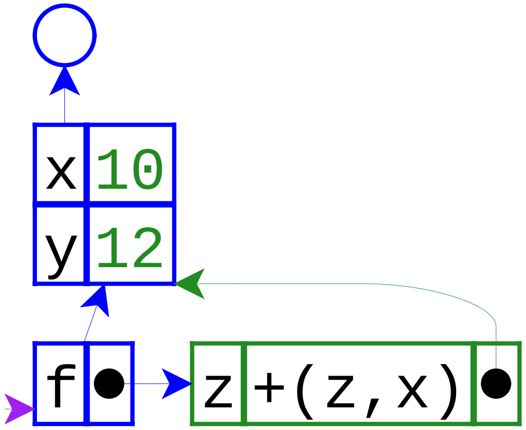

```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

Eval RHS of the let expression...



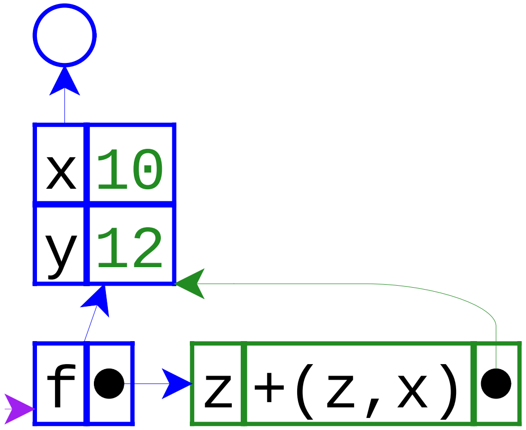

```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

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



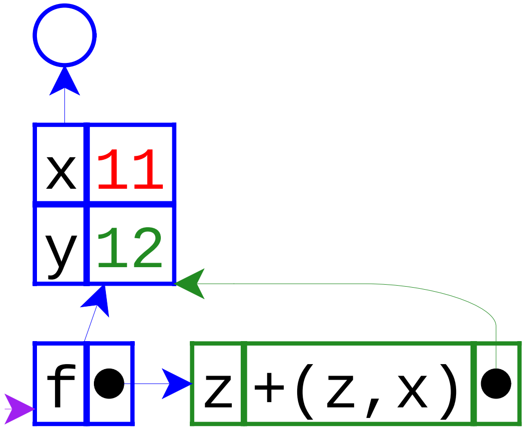

```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

To finish the `let`, the environment is extended with `f` bound to the closure. Then evaluate the body.



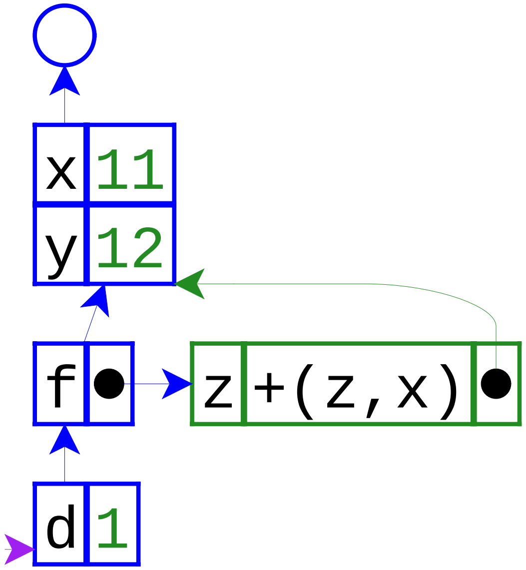

```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

Eval RHS of the let expression...



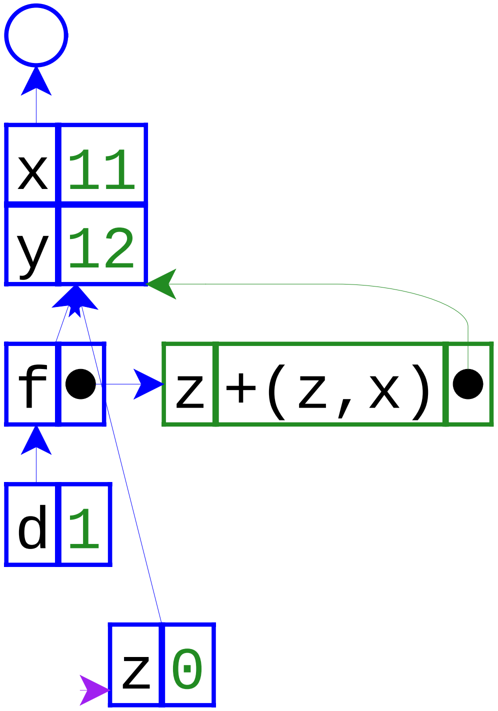

```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

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



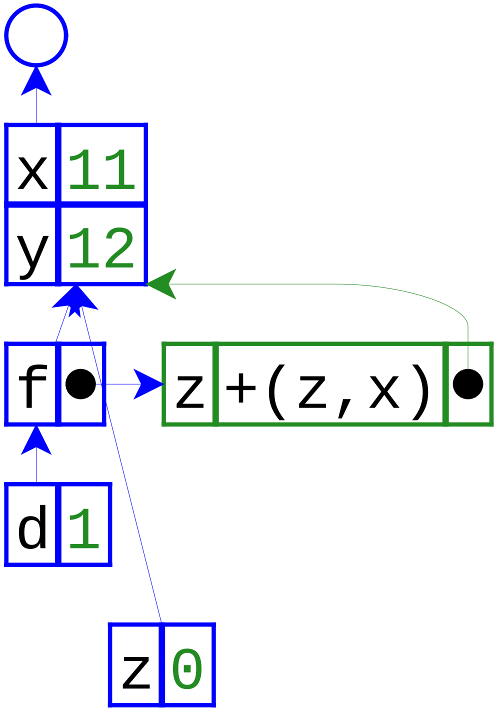

```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

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)
```

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

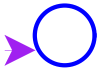



```
let x = 10
    y = 12
  in let f = proc(z)+(z,x)
      in let d = set x = +(x,1)
          in (f 0)
```

The Point: By capturing environments, closures capture variables that may change.

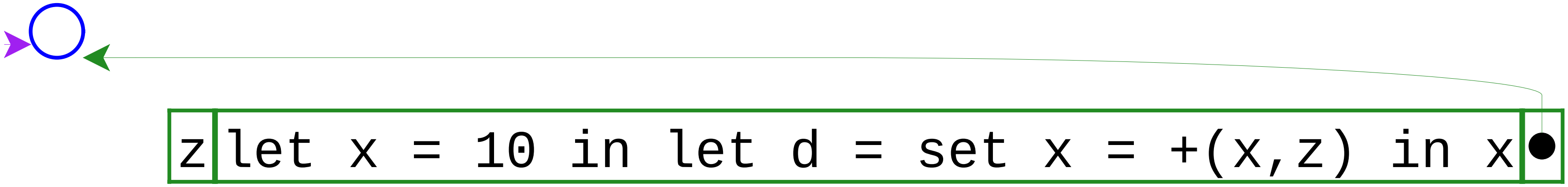
```
let f = proc(z)
         let x = 10
          in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

Another example with `proc`, but with the `let` inside the `proc`.



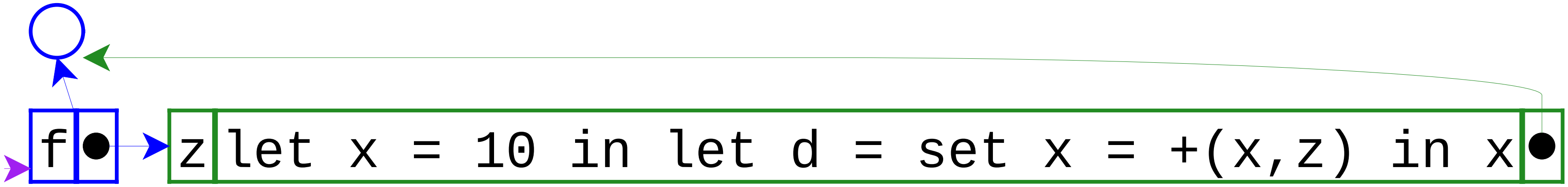
```
let f = proc(z)
         let x = 10
          in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

Eval RHS of the let expression...



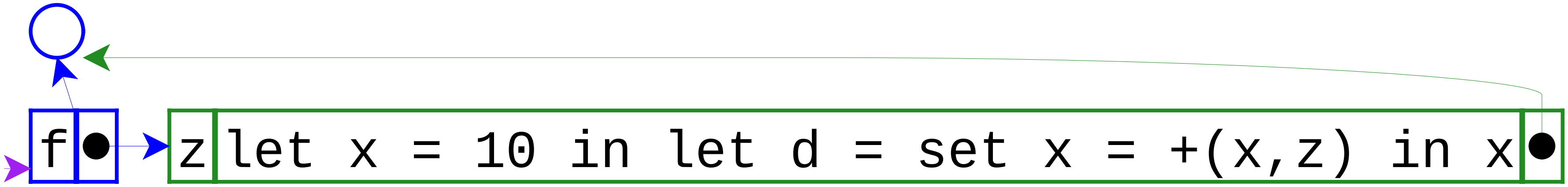

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

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



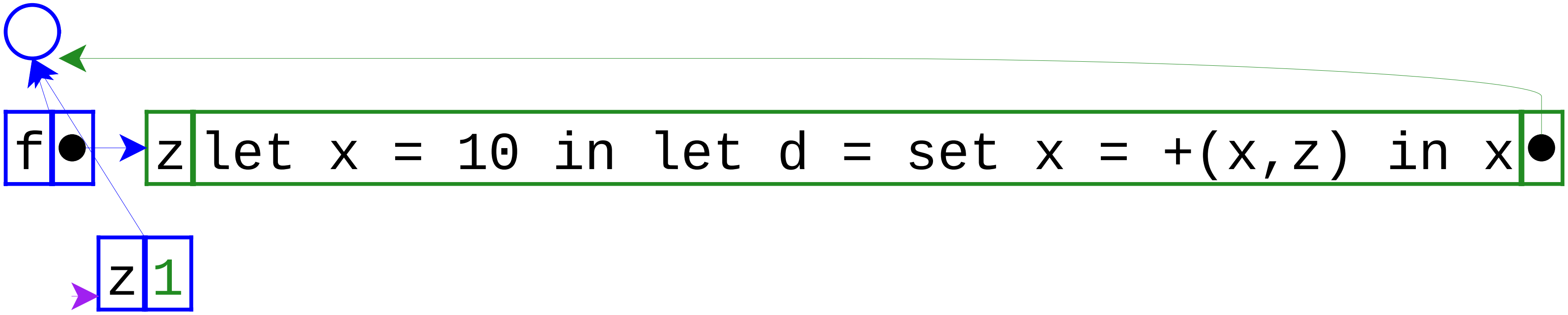

```
let f = proc(z)
         let x = 10
          in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

Bind the closure to `f` and eval the body.



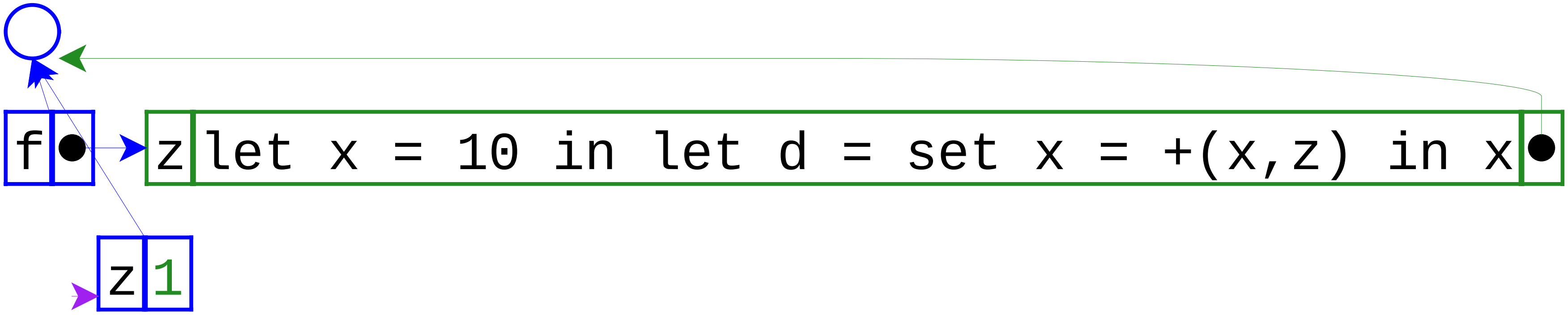

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

Evaluate the first operand, `(f 1)`.



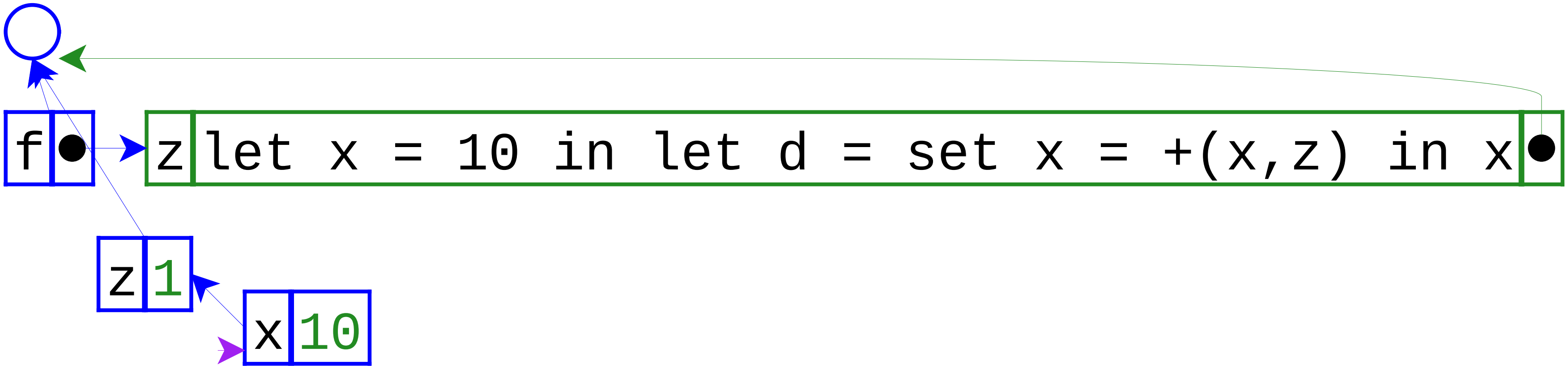

```
let f = proc(z)
         let x = 10
          in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

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



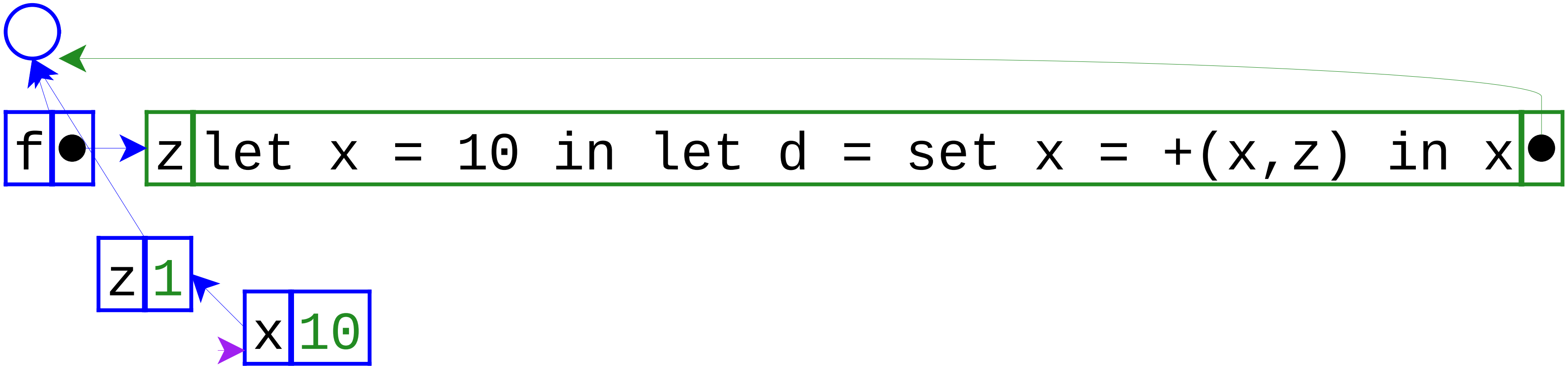

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

Eval the RHS.



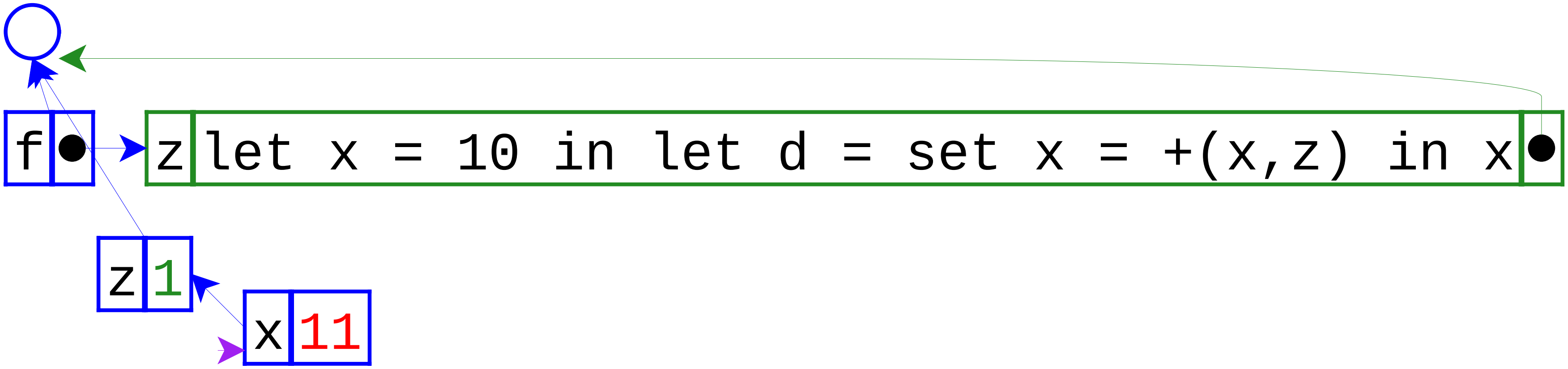

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

Add the binding for x and eval the inner body.



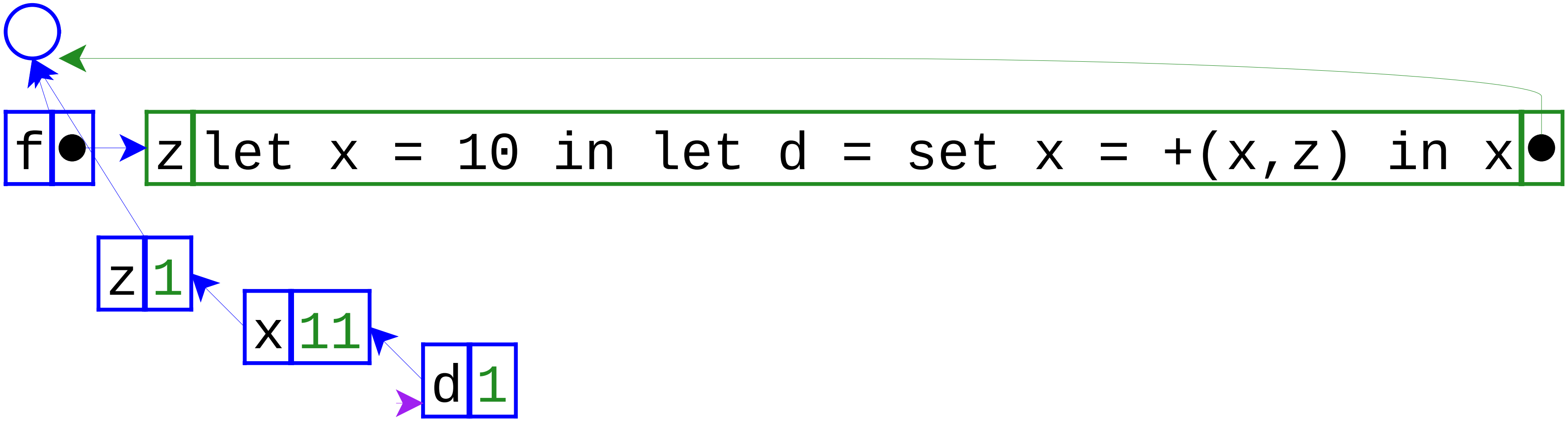

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

Eval RHS...



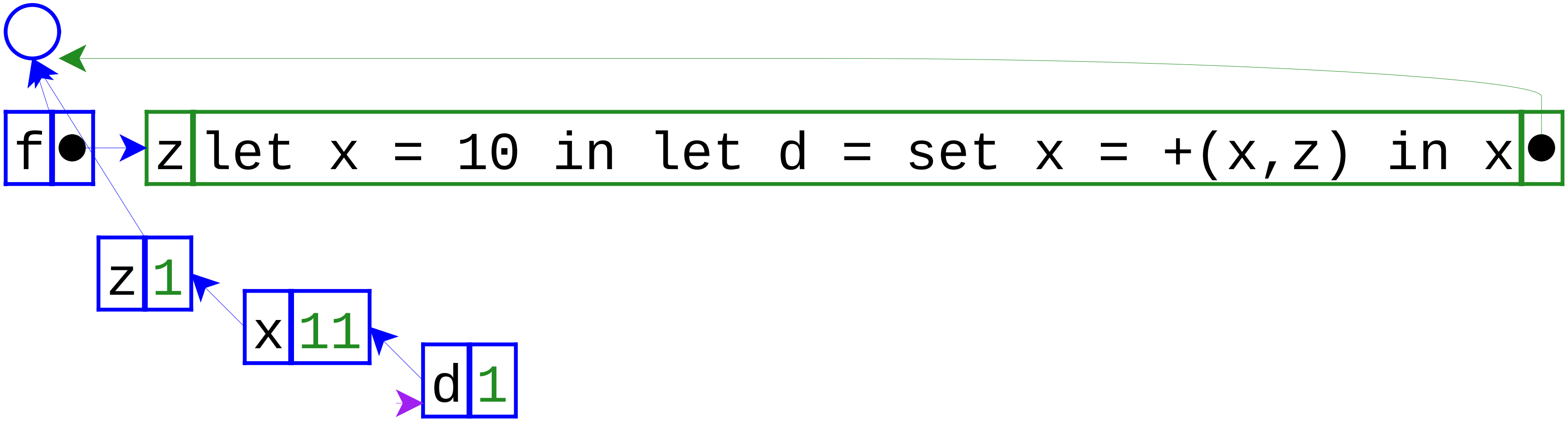

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

... which modifies the value of x.



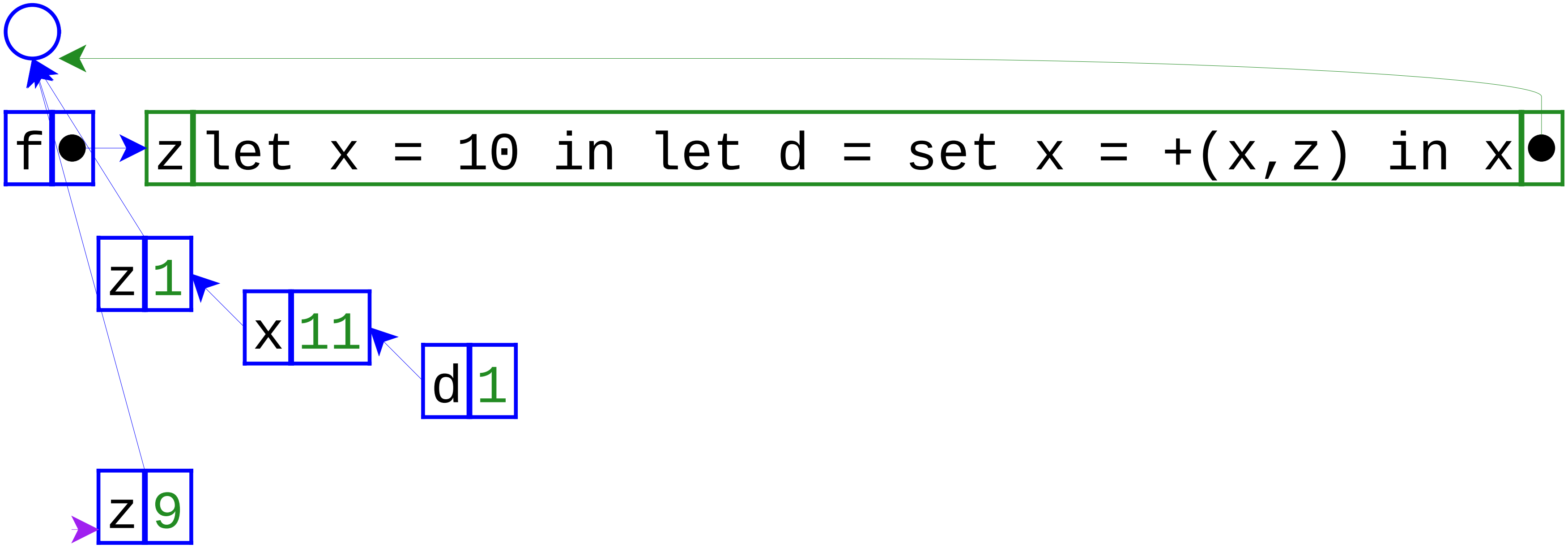

```
let f = proc(z)
         let x = 10
          in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

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



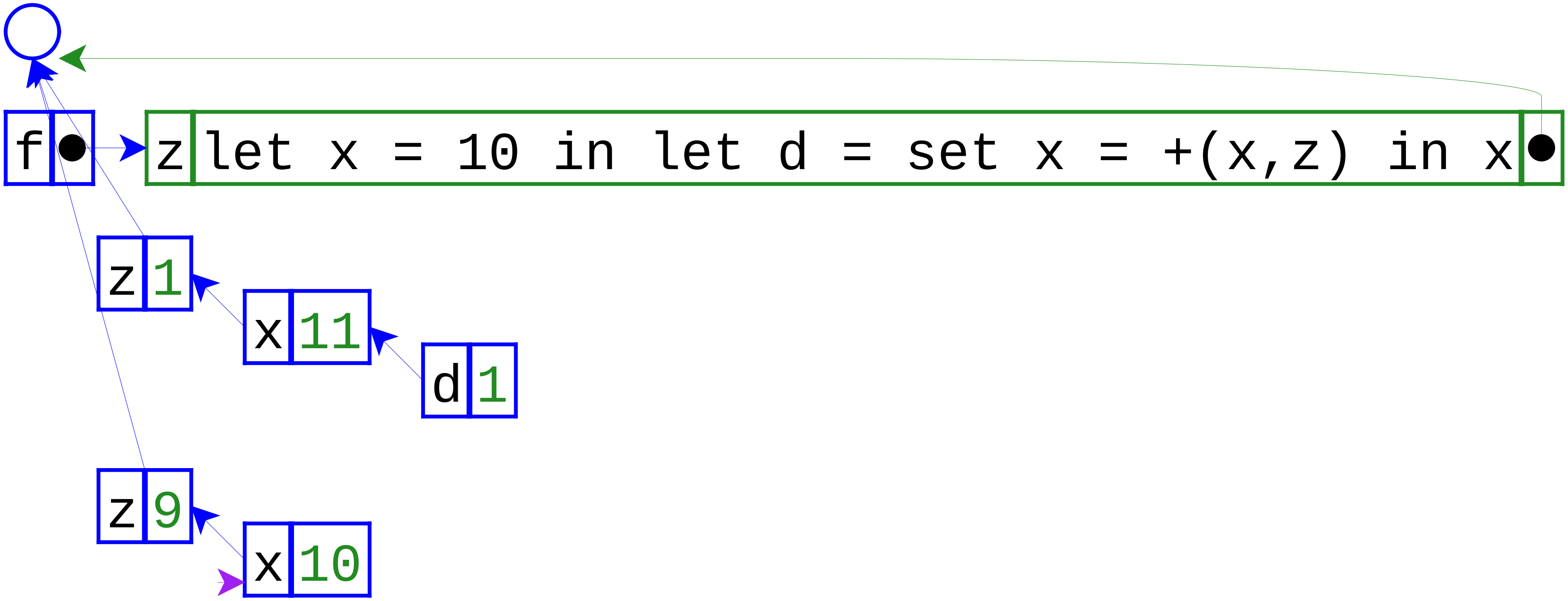

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

First operand is `11`. Now evaluate the second operand, `(f 9)`.



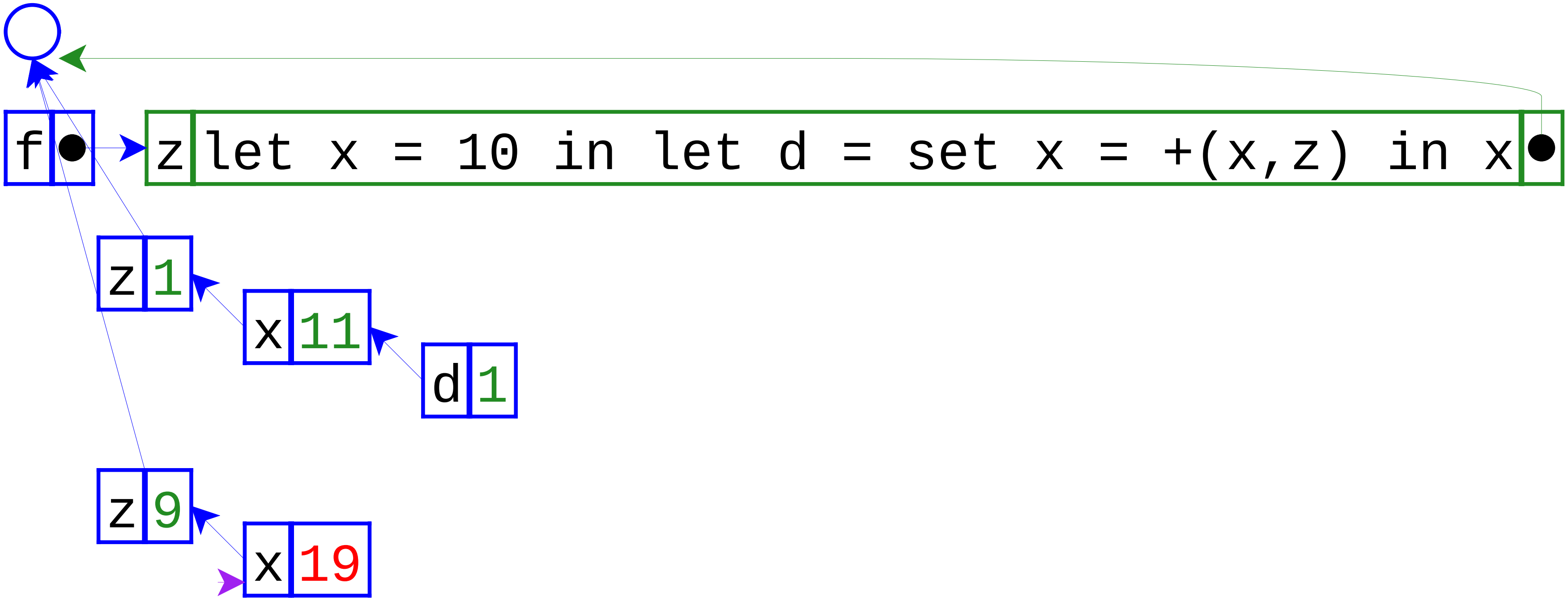

```
let f = proc(z)
         let x = 10
          in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

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



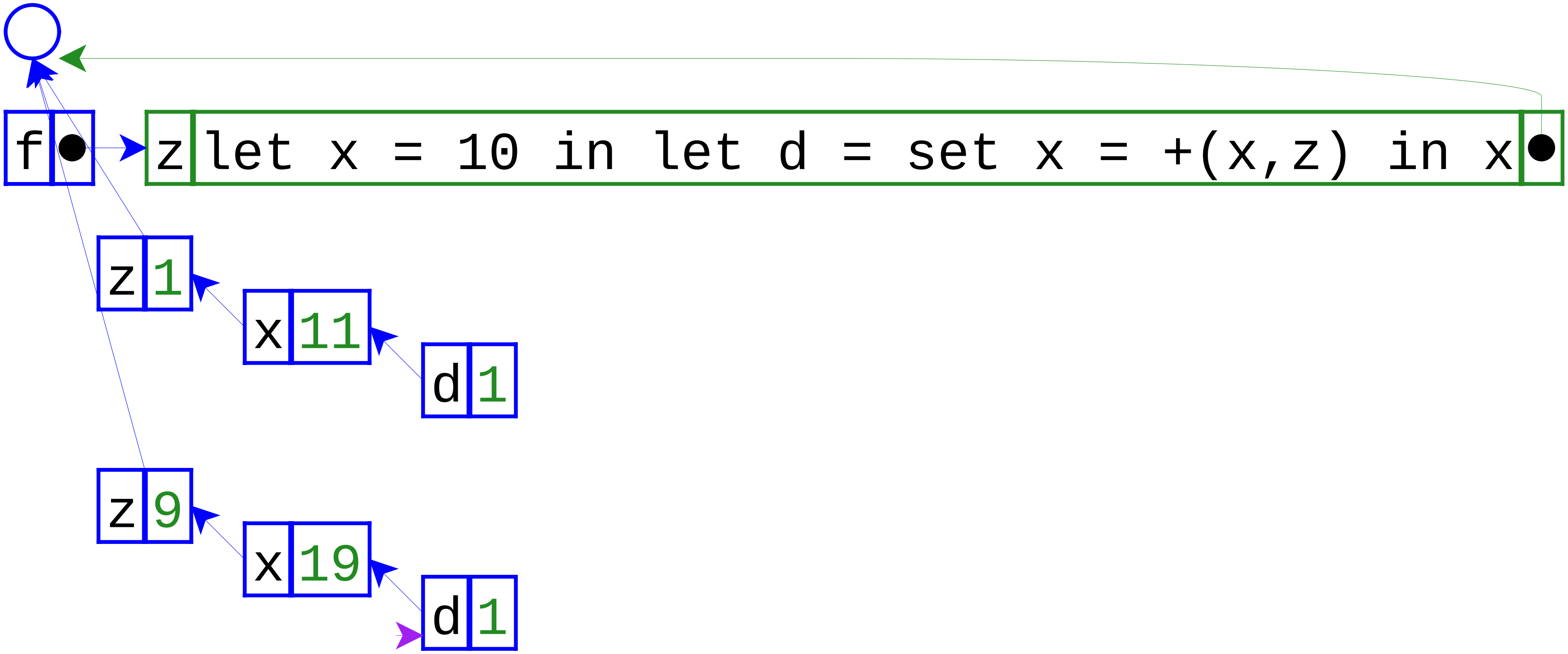

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

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



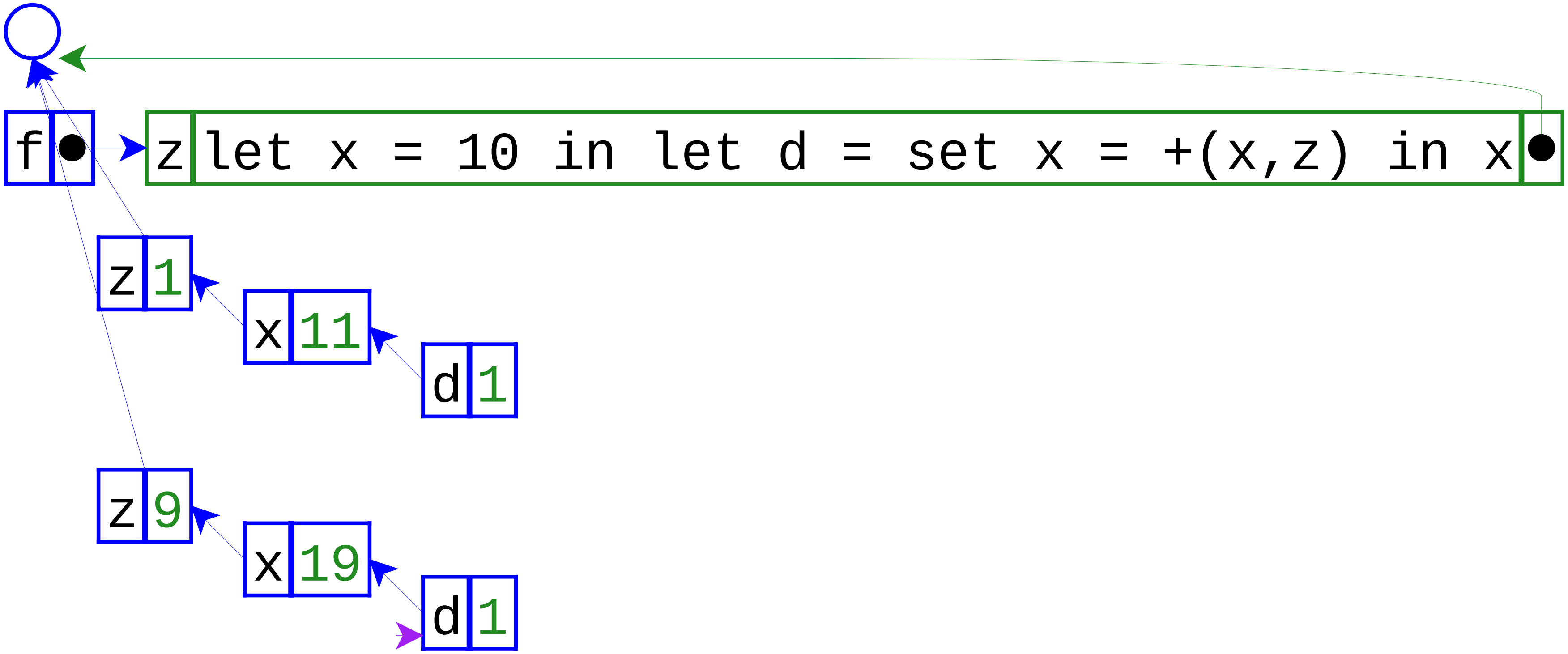

```
let f = proc(z)
         let x = 10
          in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

Again the dRHS modifies the value of x, but using the new z and x.



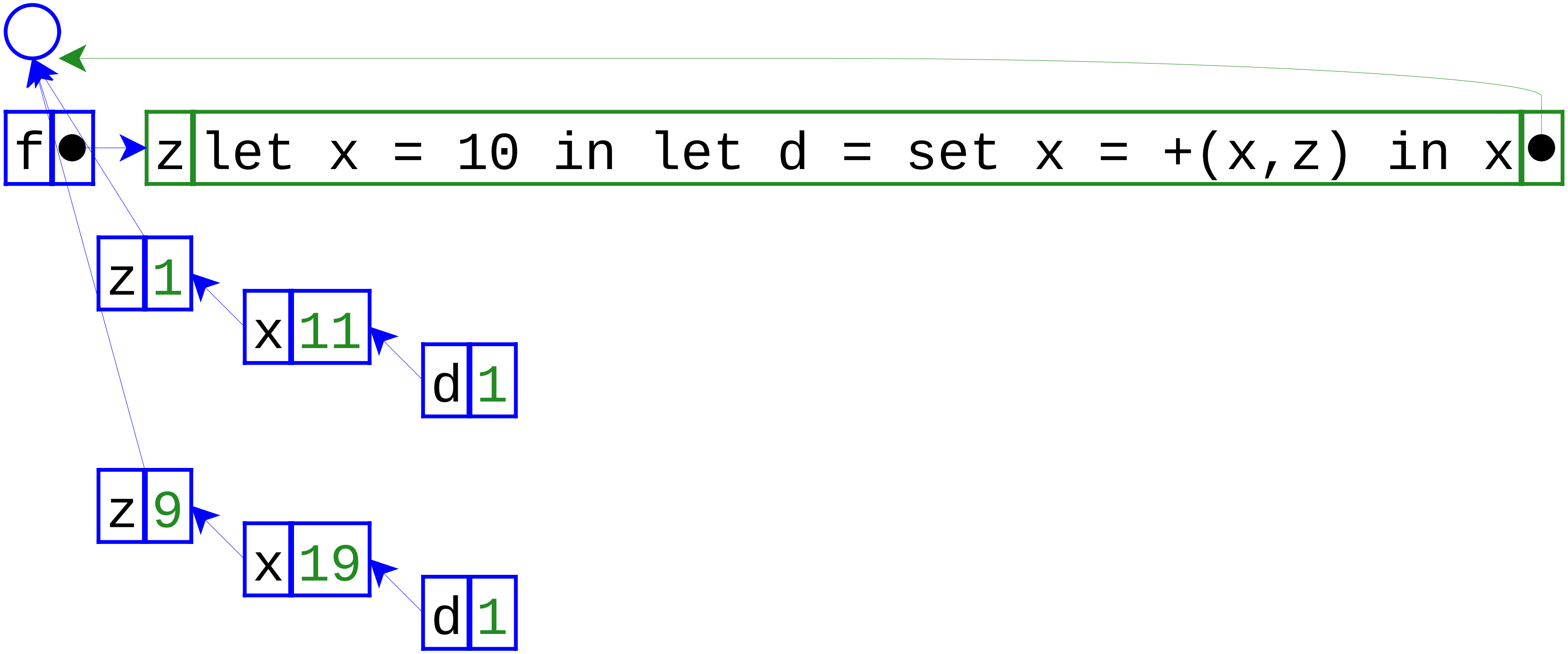

```
let f = proc(z)
          let x = 10
           in let d = set x = +(x,z)
               in x
 in +((f 1), (f 9))
```

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



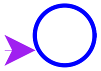

```
let f = proc(z)
         let x = 10
          in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

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



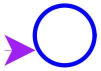

```
let f = proc(z)
         let x = 10
          in let d = set x = +(x,z)
              in x
 in +((f 1), (f 9))
```

The Point: Every evaluation of a binding expression creates a new variable (box).



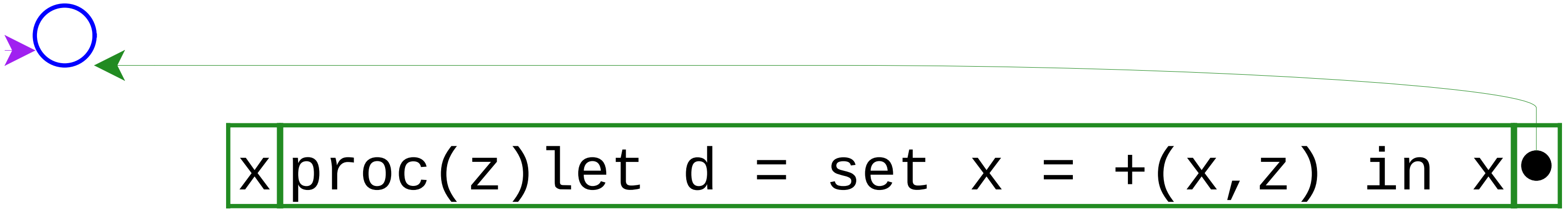
```
let mk = proc(x) proc(z)
                  let d = set x = +(x,z)
                   in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

An example with a procedure in a procedure.



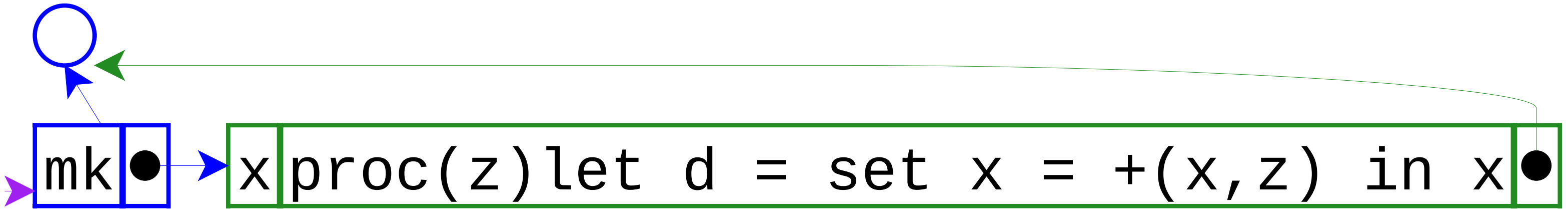
```
let mk = proc(x) proc(z)
                   let d = set x = +(x,z)
                    in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

Eval RHS of the let expression...



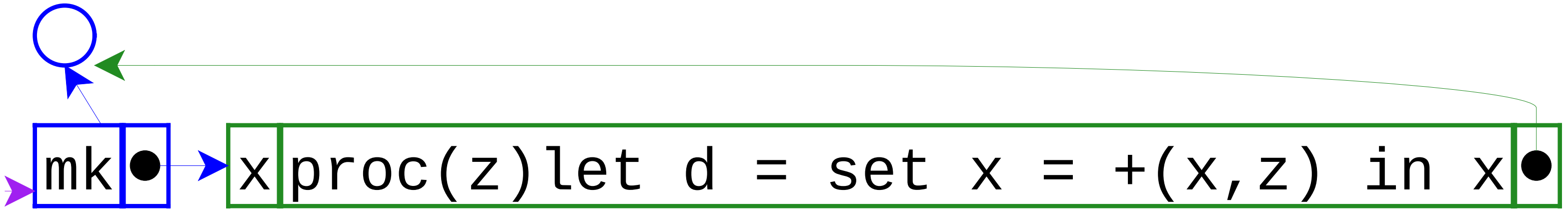

```
let mk = proc(x) proc(z)
                  let d = set x = +(x,z)
                   in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

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



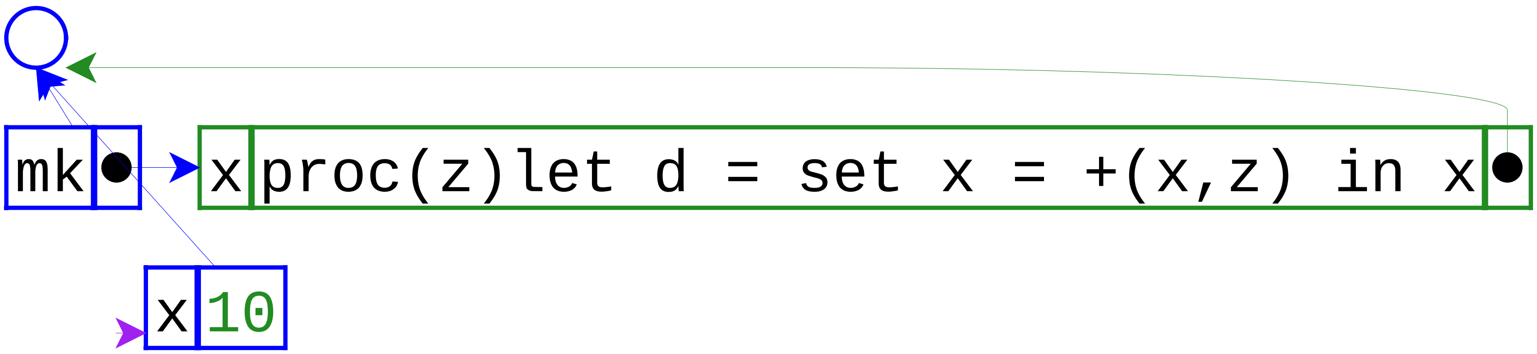

```
let mk = proc(x) proc(z)
                  let d = set x = +(x,z)
                   in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

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



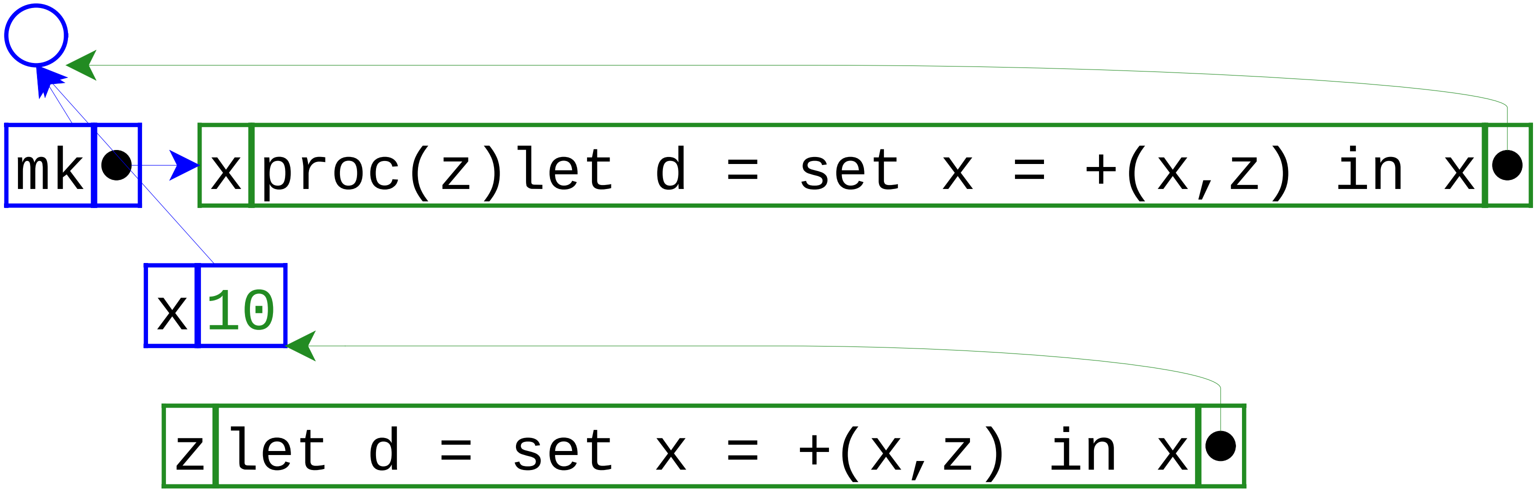

```
let mk = proc(x) proc(z)
                  let d = set x = +(x,z)
                   in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

Eval RHS, a function call. Look up `mk`...



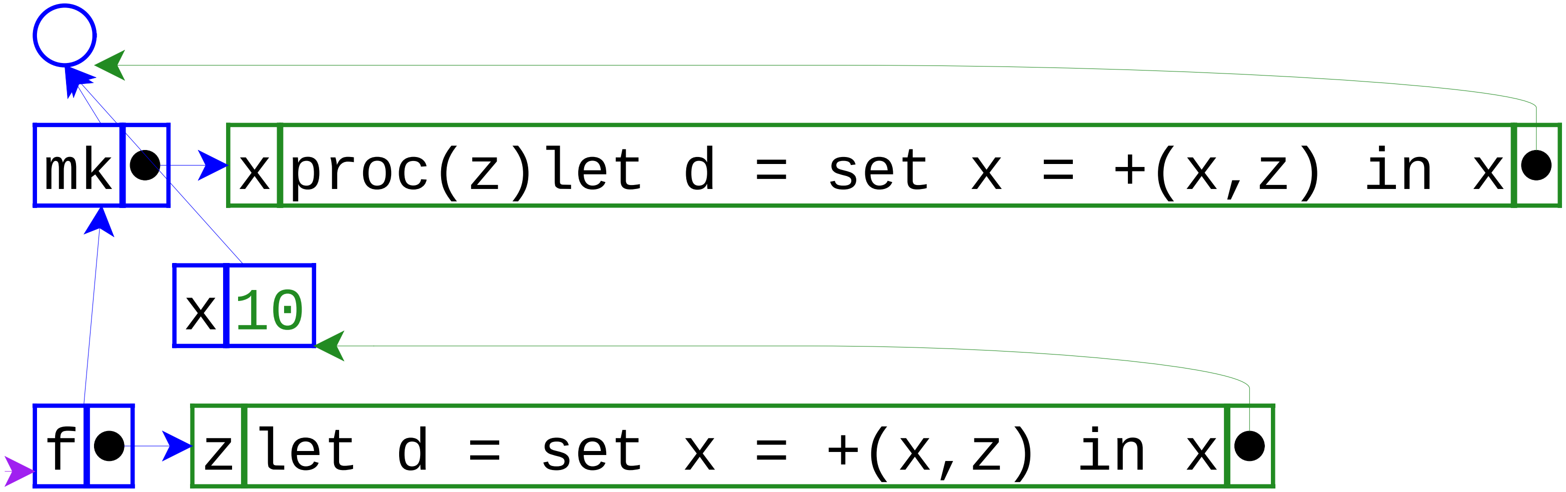

```
let mk = proc(x) proc(z)
                  let d = set x = +(x,z)
                   in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

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



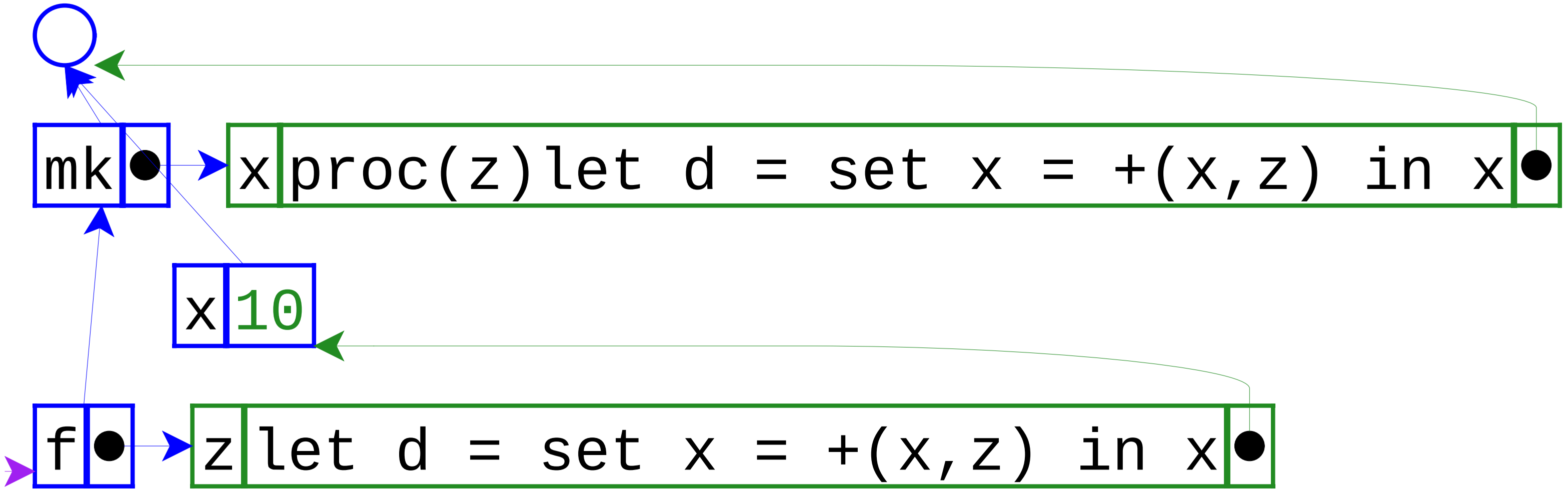

```
let mk = proc(x) proc(z)
                  let d = set x = +(x,z)
                   in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

The body is a `proc` expression, so we create another closure. Note that the variable `x` is in the closure's environment.



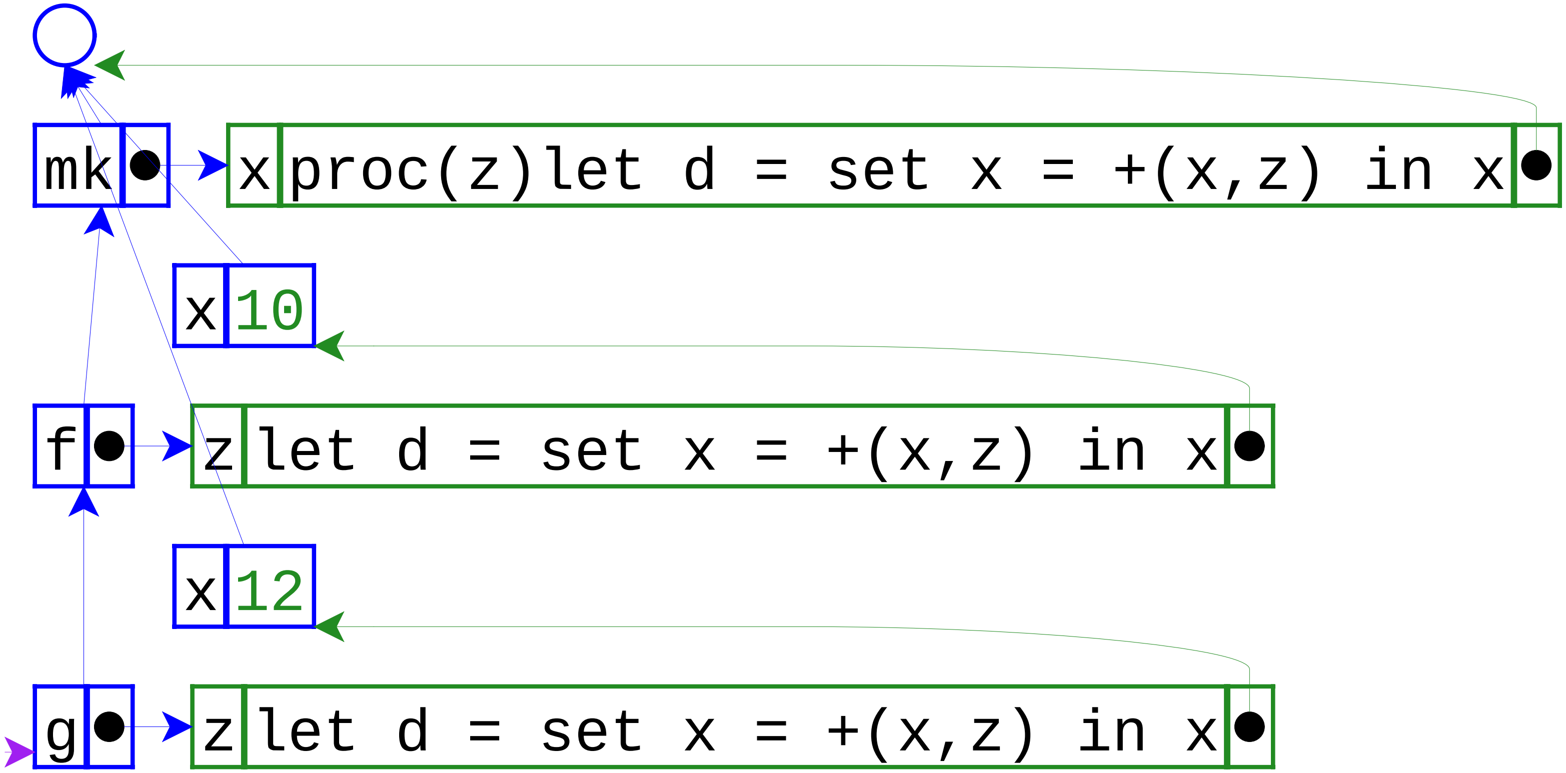

```
let mk = proc(x) proc(z)
                   let d = set x = +(x,z)
                    in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

Bind `f` to the closure, and evaluate the body.



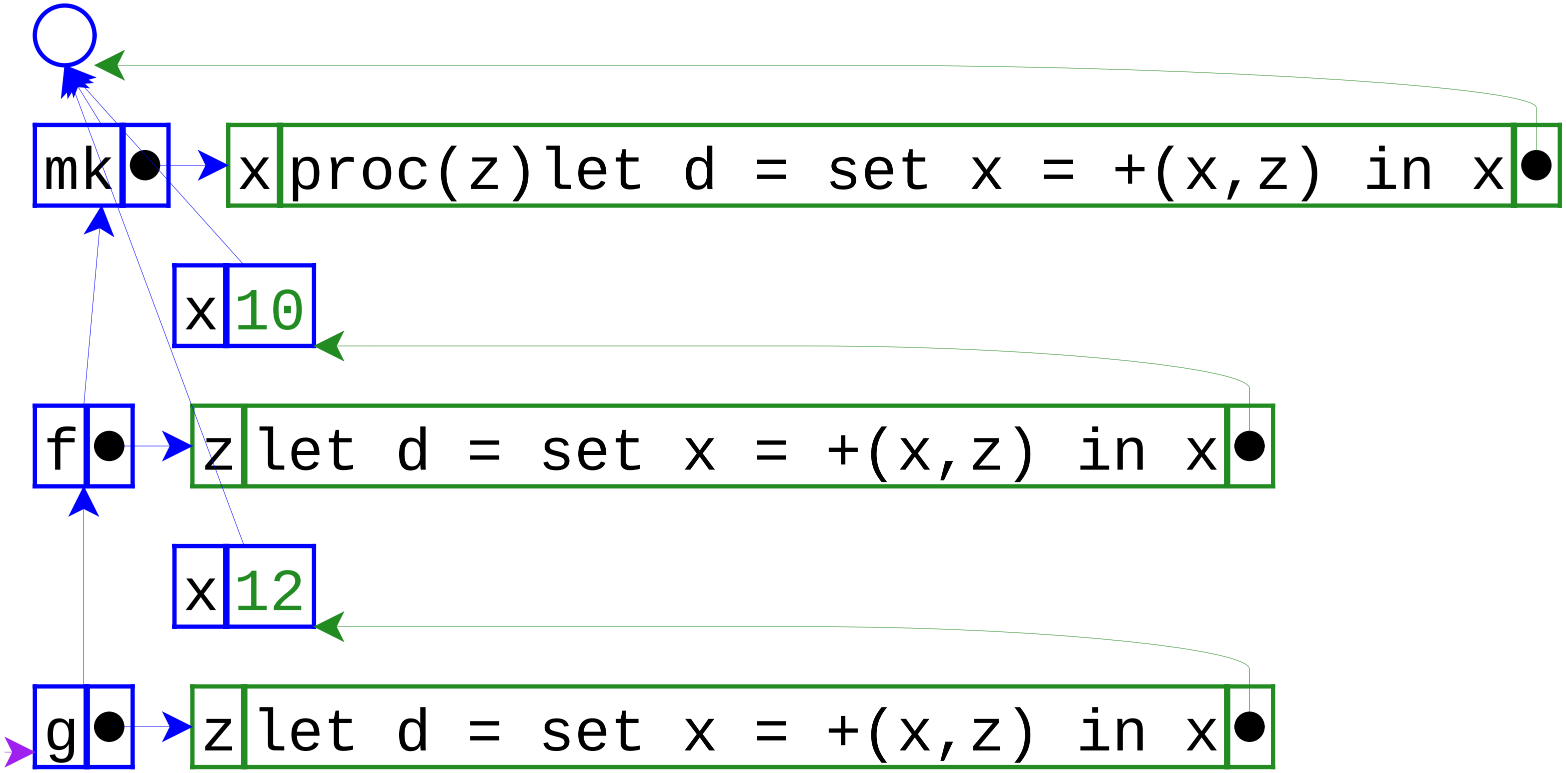

```
let mk = proc(x) proc(z)
                  let d = set x = +(x,z)
                   in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

Eval RHS of the let expression, another call to `mk`. Do the same thing as before...



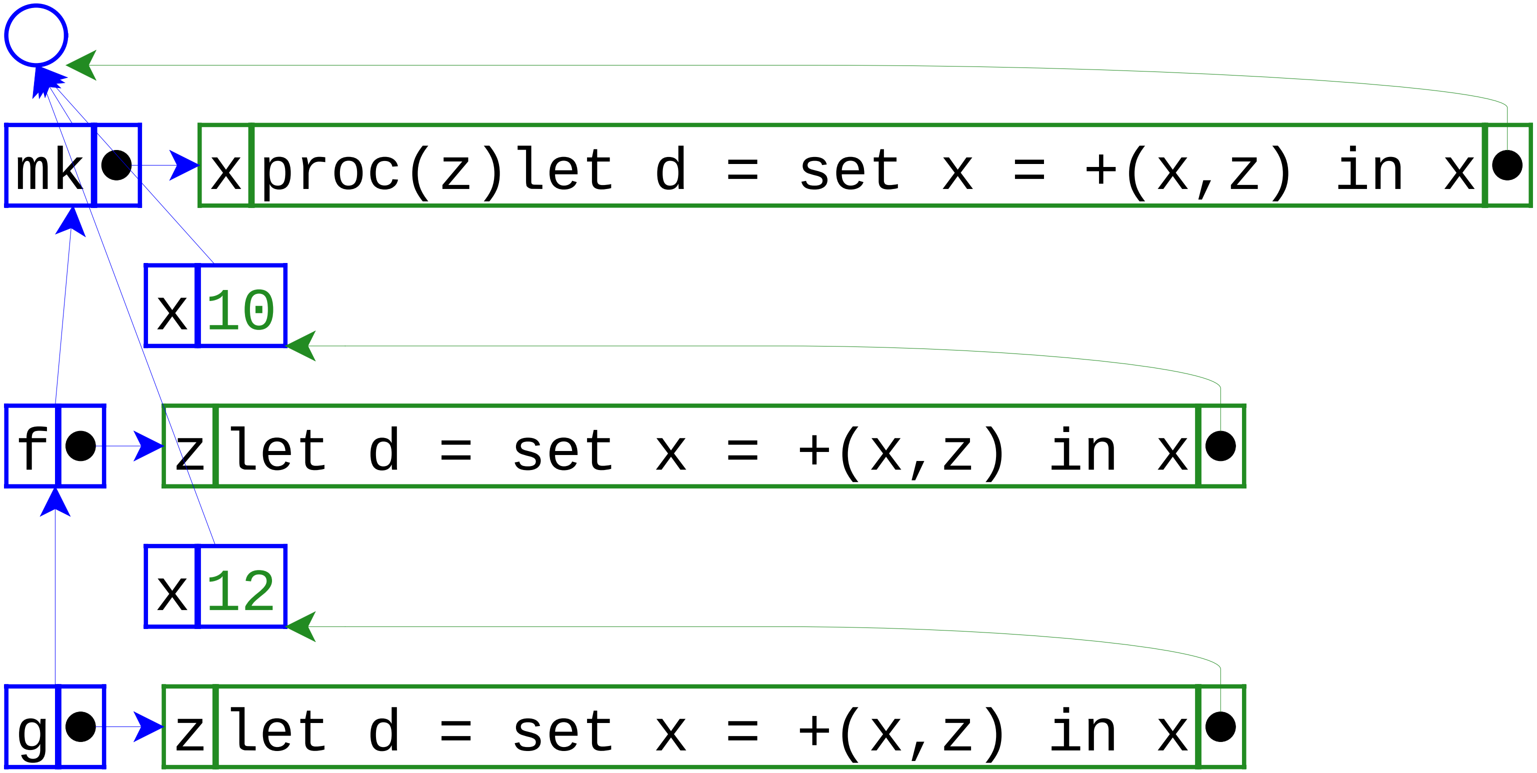

```
let mk = proc(x) proc(z)
                   let d = set x = +(x,z)
                    in x
 in let f = (mk 10)
     in let g = (mk 12)
         in ...
```

Just as before, we extend `mk`'s environment 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 ...
```

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 ...
```

The Point: Closures can capture generated variables, effectively getting private state.

Summary:

- Variables now denote locations, not values.
- Lexical scope still works.