Today: define/lambda assignmet

... or... how does lambda actually work?

(CoK on Wed)

```
(lambda (x)
   (+ x 1))
```

lambda returns a closure

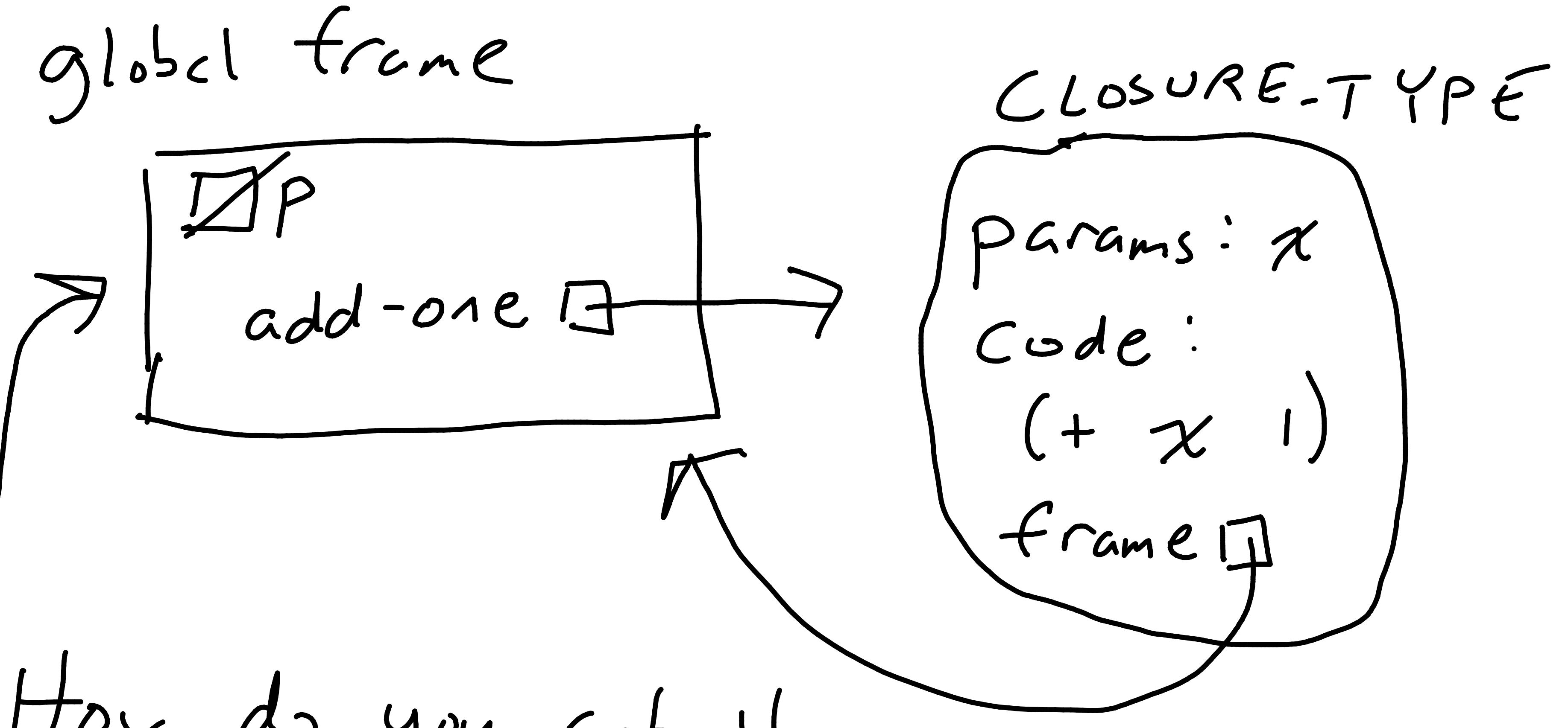
- a struct that contains 3 things

① a list of the parameters
   e.g. x

② the code in the lambda
   e.g. (+ x 1)

③ a pointe to the frame that was active when you called lambda
   - this is how we get static scoping to work

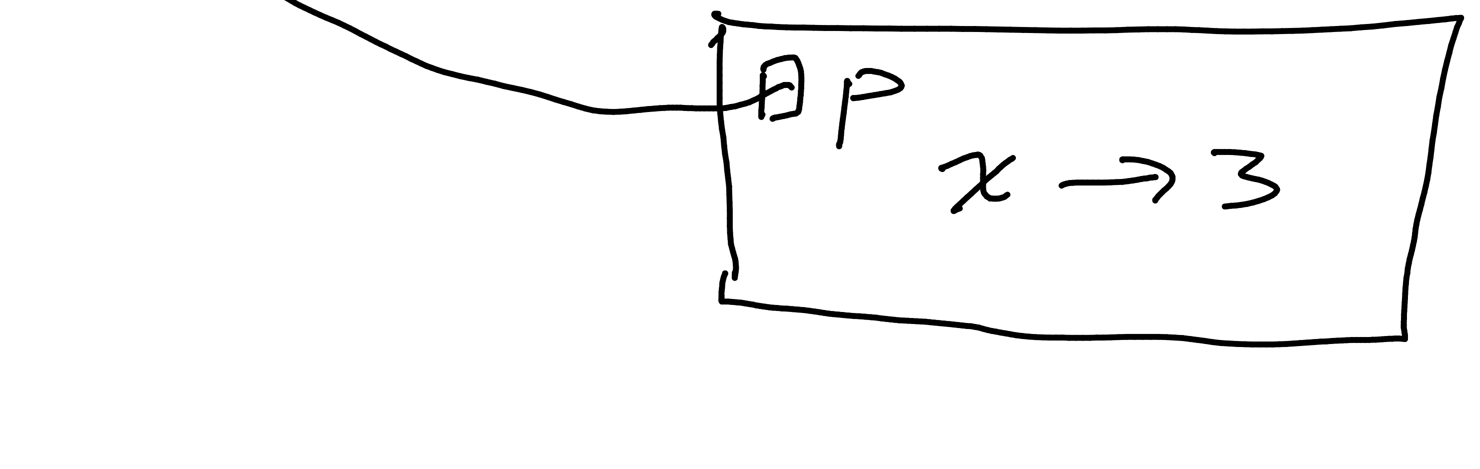
```
(define add-one
  (lambda (x)
    (+ x 1)))
```



How do you actually call a fuction? <u>apply</u>

e.g. (add-one 3)

① Make a new frame to hold paremeters. Parent frame is the one the closure refers to.



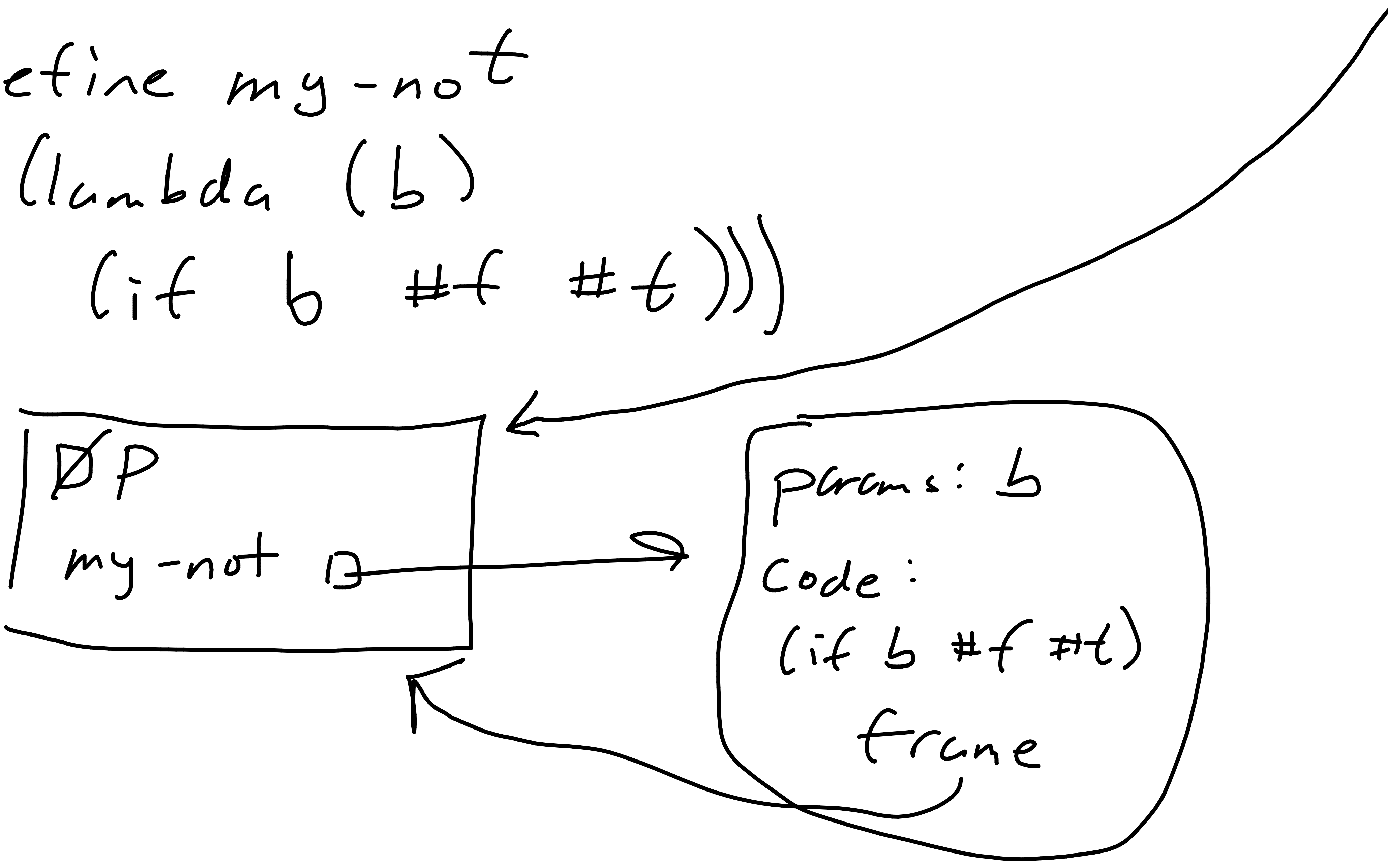
② Evaluate the code from the closure in the context of that new frame.

Evaluate (+ x 1) in the new frame

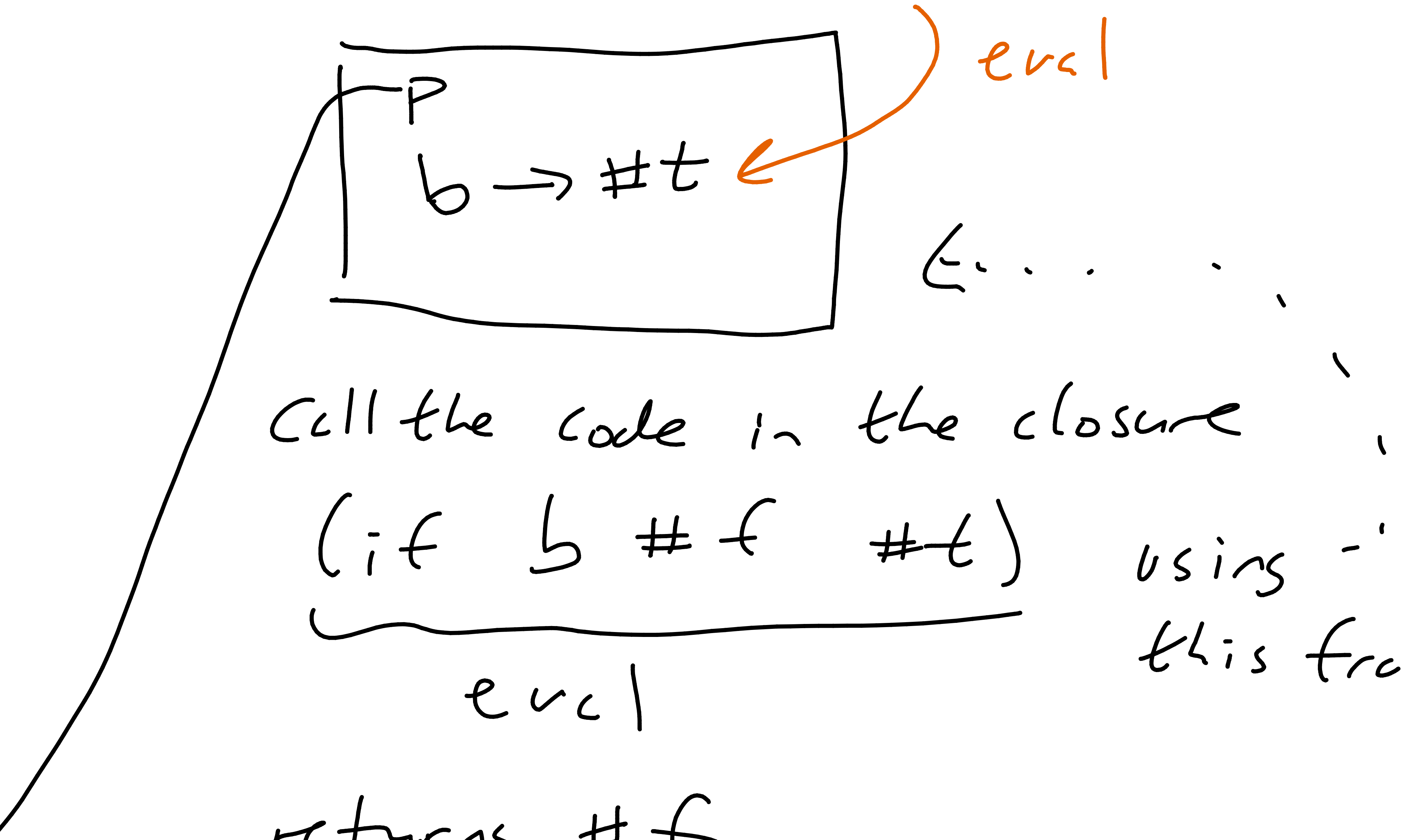
$\rightarrow 4$

... return that

```
(define my-not
  (lambda (b)
    (if b #f #t)))
```



invoke (my-not #t)

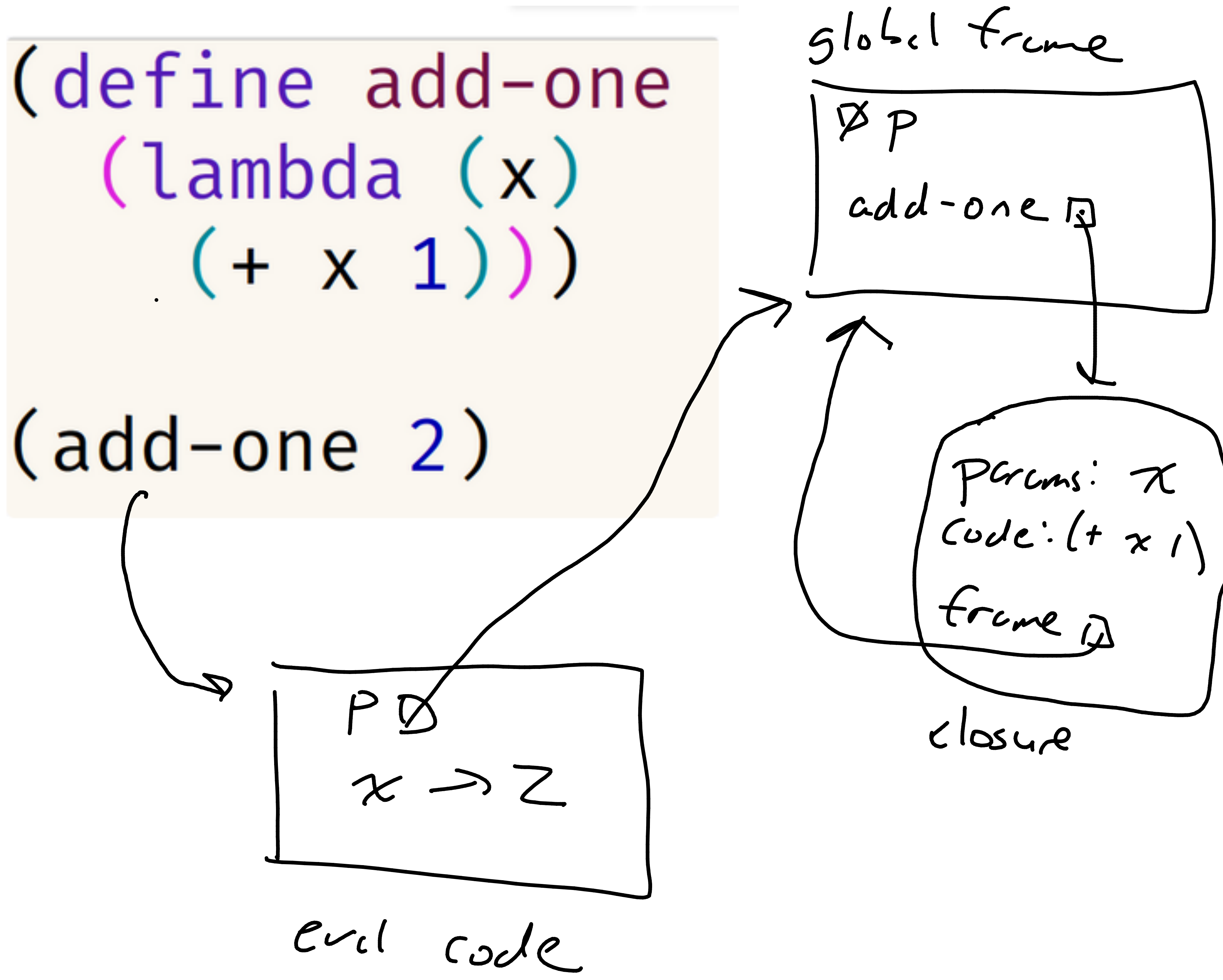


call the code in the closure

$$\underbrace{\texttt{(if b \#f \#t)}}_{\text{eval}}$$

using this frame

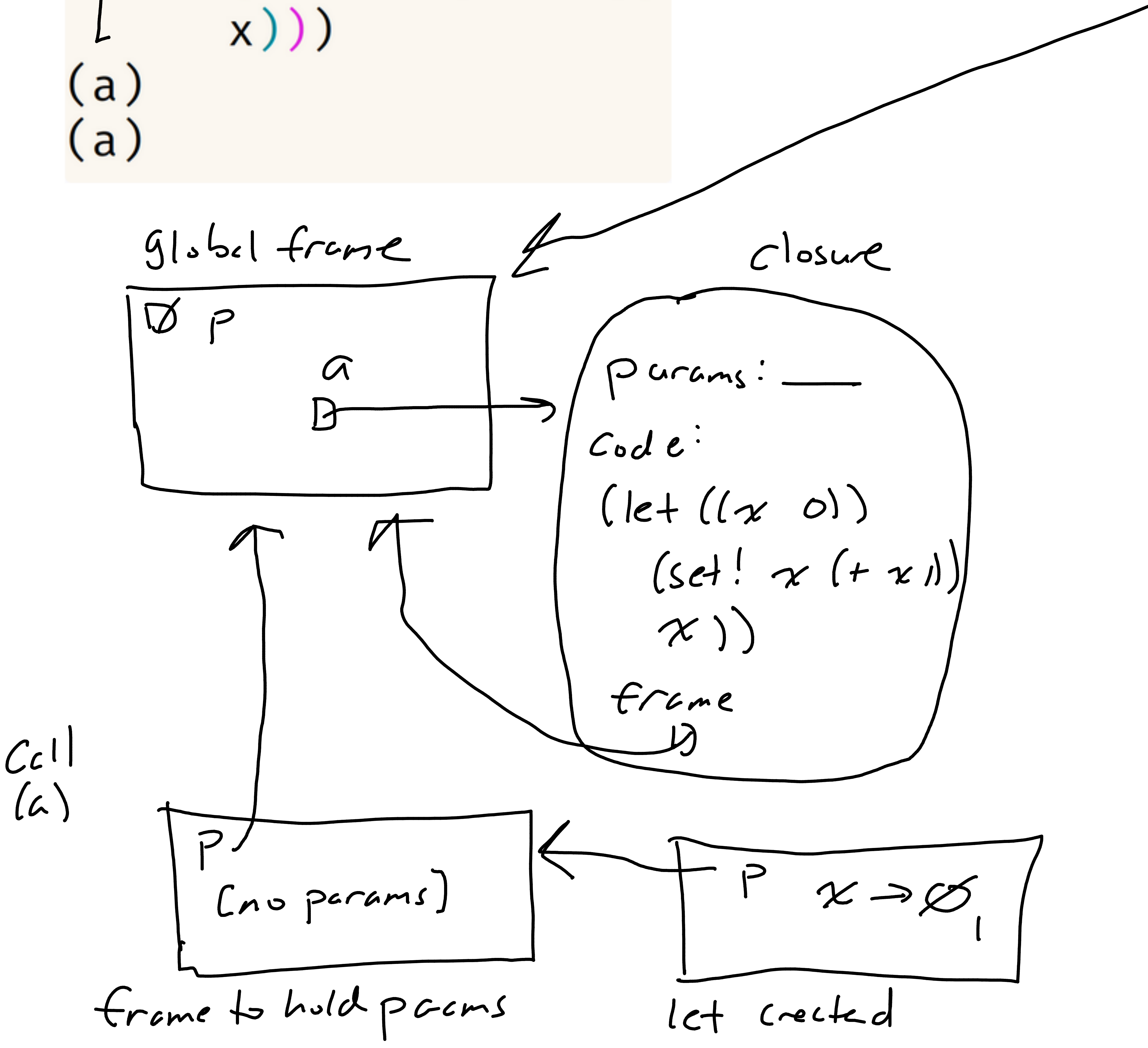
returns #f



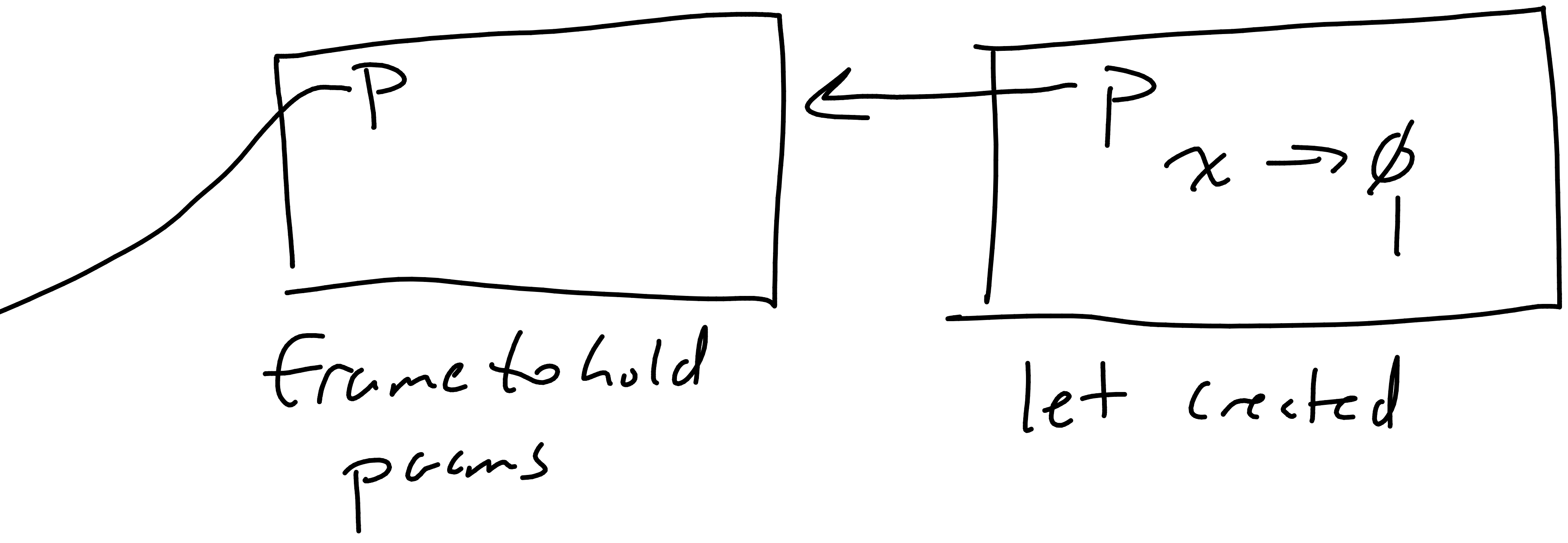
```
(define add-one
  (lambda (x)
    (+ x 1)))

(add-one 2)
```

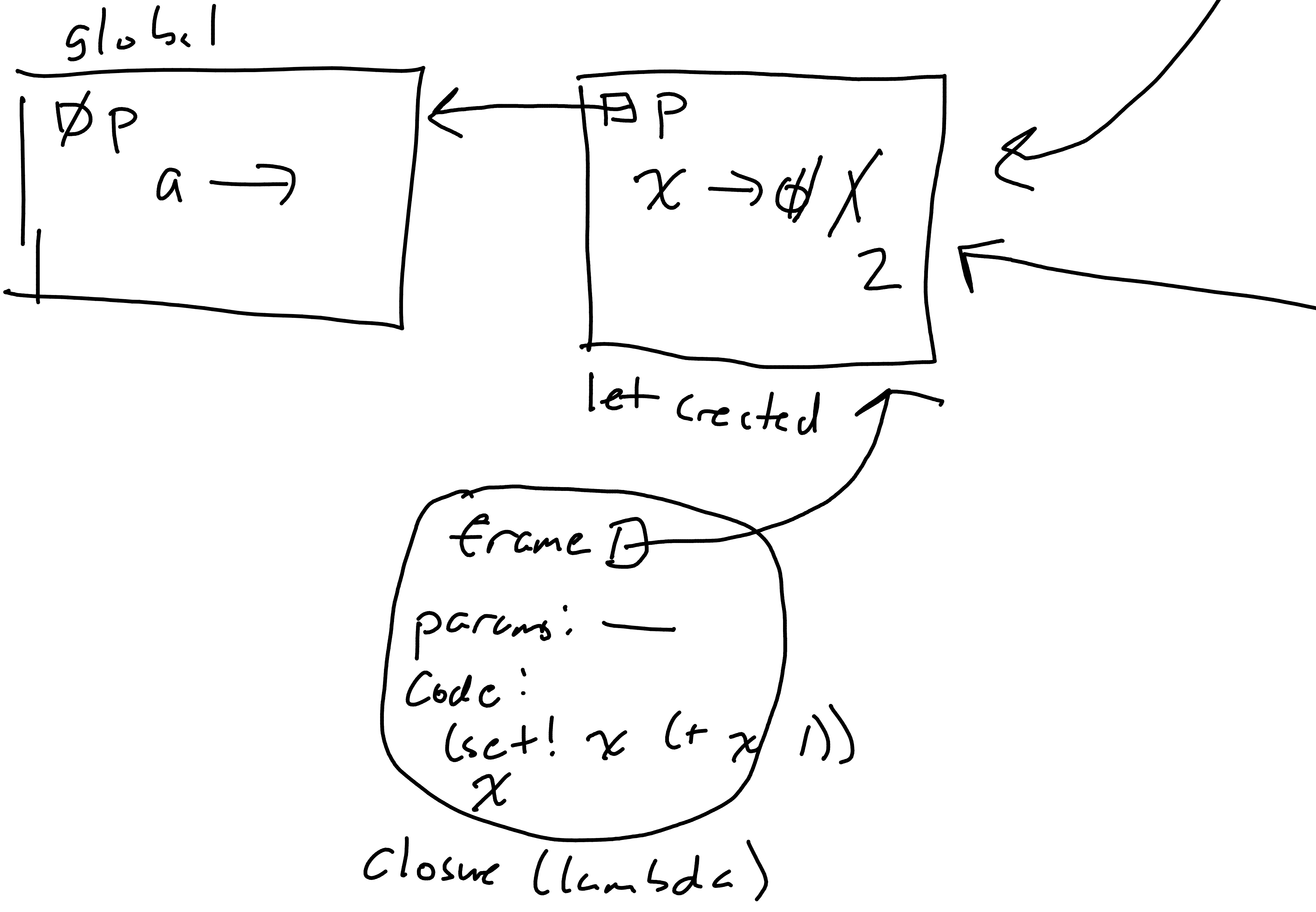
```
(define a
  (lambda ()
    (let ((x 0))
      (set! x (+ x 1))
      x)))
(a)
(a)
```



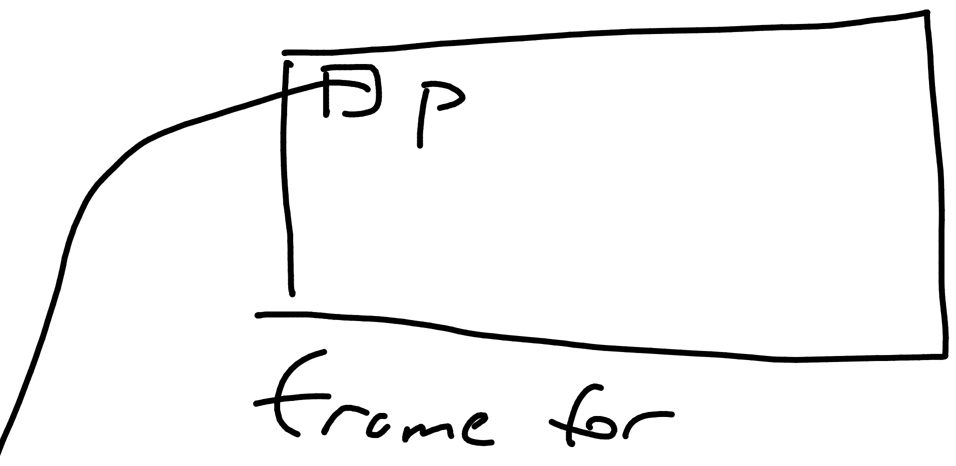
After second call to (a)



```scheme
(define a
  (let ((x 0))
    (lambda ()
      (set! x (+ x 1))
      x)))
(a)
(a)
```



Call (a)

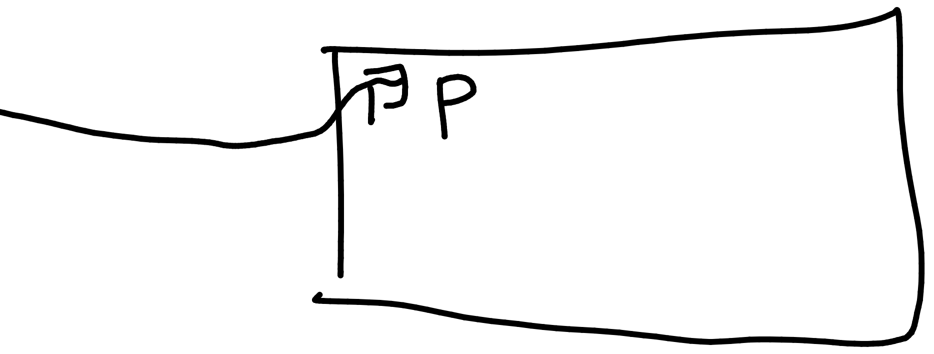


frame for
params

execute code

```
(set! x (+ x 1))
```

Call (c)



```
(set! x (+ x 1))
```