smol/hof

How to read this file?

```
(a smol program)
```

- **Correct answer**
- **Other answer 1**
- **Other answer 2**

Why the correct answer is correct

fun returns lambda

```
(deffun (f x)
  (lambda (y) (+ x y)))

((f 2) 1)
```

- **3**
- **Error**

The lambda expression is created in the scope of x, so it can use x.

filter gt

```
(filter (lambda (n) (> 3 n)) '(1 2 3 4 5))
```

- **'(1 2)**
- **'(4 5)**

This program keeps numbers that 3 is greater than (not that is greater than 3).

fun and state 1/4

```
(defvar x 1)
(defvar f
  (lambda (y)
    (+ x y)))
(set! x 2)
```

```
(f x)
```

- **4**
- **3**

Every time f is called, it looks up the value of x again.

fun and state 2/4

```
(defvar x 1)
(deffun (f y)
  (+ x y))
(set! x 2)
(f x)
```

- **4**
- **3**

Same as fun and state 1/4

fun and state 3/4

```
(defvar x 1)
(defvar f
  (lambda (y)
     (+ x y)))
(let ([x 2])
  (f x))
```

- **3**
- **4**

The x in the definition of f is the global x, which is a variable different from the x in let.

fun and state 4/4

```
(defvar x 1)
(deffun (f y)
  (+ x y))
(let ([x 2])
  (f x))
```

- **3**
- **4**

Same as fun and state 3/4.

eval order

```
(deffun (f x) (+ x 1))
(deffun (new-f x) (* x x))

(f (begin
     (set! f new-f)
     10))
```

- **11**
- **100**
- **Error**

Function application first computes the value first computes the value of the operator, then the values of operands (i.e. actual parameters) from left to right. So when the set! happened, f had been resolved to its initial value.

counter

```
(deffun (make-counter)
  (let ([count 0])
    (lambda ()
      (begin
        (set! count (+ count 1))
        count))))
(defvar f (make-counter))
(defvar g (make-counter))

(f)
(g)
(f)
(f)
(g)
```

- **1; 1; 2; 3; 2**

- **1; 1; 1; 1; 1**
- **1; 1; 2; 3; 4**

Every time the function make-counter is called, it returns a function that returns 1 when called the first time, 2 the second time, etc. Each (lambda () …) has its own local variable count.

hof + set!

```
(defvar y 3)
(+ ((lambda (x) (set! y 0) (+ x y)) 1)
   y)
```

- **1**
- **7**
- **4**
- **Error**

Because function applications compute their parameters from left to right, set! happened before resolving the last y.

filter

```
(defvar l (list (ivec) (ivec 1) (ivec 2 3)))
(filter (lambda (x) (vlen x)) l)
```

- **'(#() #(1) #(2 3))**
- **‘(0 1 2)**
- **'(#(1) #(2 3))**
- **Error**

Recall that all values other than #f are considered truthy. The filter is effectively creating a copy of l.

eq? fun fun 1/3

```
(eq? (λ (x) (+ x x))
     (λ (x) (+ x x)))
```

- **#f**
- **#t**

Lambda expressions are similar to mvec in the sense that everytime we compute the value of a lambda expression, a new value is created. eq? returns true only when the two values are the same (i.e. identical).

eq? fun fun 2/3

```
(deffun (f x) (+ x x))
(deffun (g x) (+ x x))
(eq? f g)
```

- #f
- **#t**

(deffun (f x) …) can be viewed as (defvar f (lambda (x) …)).

eq? fun fun 3/3

```
(deffun (f x) (+ x x))
(deffun (g) f)
(eq? f (g))
```

- #t
- **#f**

The function value associated with f is computed exactly once when f is defined. (g) is just looking up the value of f.

equal? fun fun

```
(deffun (f) (lambda () 1))
(equal? (f) (f))
```

- #f
- **#t**

Two function values are equal if and only if they are eq. f computes (lambda () …) everytime it is called. So (f) is not equal to another (f).