smol/state

How to read this file?

```
(a smol program)
```

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

Why the correct answer is correct

circularity

```
(defvar x (mvec 2 3))
(set-right! x x)
(set-left! x x)
x
```

- **x='#(x x) or something similar. Both (left x) and (right x) are x itself.**
- **'#(#(2 #(2 3)) #(2 3))**
- **Error**

set-right! makes x a pair whose left is 2 and whose right is the pair itself. set-left! makes a pair whose both components are x itself.

eval order

```
(defvar x 0)
(ivec x (begin (set! x 1) x) x)
```

- **'#(0 1 1)**
- **'#(0 1 0)**
- **'#(1 1 1)**

When computing the value of `(ivec …)`, we first compute x, which is 0 at that moment, then `(begin …)`, which mutates x to 1 and returns 1, and finally the last x, which is now 1.

mvec as arg

```
(defvar x (mvec 1 2))
(deffun (f x)
  (vset! x 0 0))
(f x)
x
```

- `'#(0 2)`
- **`'#(1 2)`**

f was given the *same* mutable vector. When two mutable values are the same, updates to one are visible in the other.

var as arg

```
(defvar x 12)
(deffun (f x)
  (set! x 0))
(f x)
x
```

- `12`
- **`0`**

The global variable x and the parameter x are different variables. Changing the binding of the parameter x will not change the binding of the global x.

seemingly aliasing a var

```
(defvar x 5)
(deffun (set1 x y)
  (set! x y))
(deffun (set2 a y)
  (set! x y))
(set1 x 6)
x
(set2 x 7)
x
```

- `5; 7`

- **6; 7**
- **5; 5**

Similar to the last question (var as arg), calling the function set1 will not change the global x. The other function (set2), however, is using the global x.

mutable var in vec

```
(defvar x 3)
(defvar v (mvec 1 2 x))
(set! x 4)
v
x
```

- **'#(1 2 3); 4**
- **'#(1 2 4); 4**

The mutable vector stores the *value* of x (i.e. 3) rather than the *binding* (i.e. the information that x is mapped to 3). So the later set! doesn't affect v.

aliasing mvec in mvec

```
(defvar v (mvec 1 2 3 4))
(defvar vv (mvec v v))
(vset! (vref vv 1) 0 100)
vv
```

- **'#(#(100 2 3 4) #(100 2 3 4))**
- **Error**
- **'#(#(1 2 3 4) #(1 2 3 4))**
- **'#(#(1 2 3 4) #(100 2 3 4))**

Both components of vv are identical to v. That is, the left of vv, the right of vv, and v are the same vector.

vset! in let

```
(defvar x (mvec 123))
(let ([y x])
  (vset! y 0 10))
x
```

- `'#(10)`
- **`'#(123)`**

y and x are bound to the same vector.

set! in let

```
(defvar x 123)
(let ([y x])
  (set! y 10))
x
```

- **`123`**
- **10**

y and x are different variables. The set! re-binds y to 10. This won't affect x.

seemingly aliasing a var again

```
(defvar x 10)
(deffun (f y z)
  (set! x z)
  y)
(f x 20)
x
```

- **`10; 20`**
- `20; 20`

At first, x is bound to 10. When f is called, y is bound to the value of x, which is 10, and z is bound to the value of 20, which is 20 itself. Then f re-binds x to the value of z, which is 20. After that f returns the value of y, which is 10. Finally, the program computes the value of x, which is now 20 because f has rebound x.