0

(Task 1 of 20) In this tutorial, we will learn *more* about **mutable values**, illustrated with **vectors**.

Lispy |

(Task 2 of 20) What is the result of running this program?

Lispy [Run]

```
(defvar v (mvec 51 62 73))
(defvar vv (mvec v v))
(vec-set! (vec-ref vv 1) 0 44)
(vec-ref vv 0)
```

Python

```
v = [ 51, 62, 73 ]
vv = [ v, v ]
vv[1][0] = 44
print(vv[0])
```

2

`#(44 62 73)`

3

You predicted the output correctly 🎉🎉🎉

This program first creates a three-element vector and binds it to `v`. After that, the program creates a two-element vector. Both elements refer to the first vector. After that, the `0`-th element of the three-element vector is replaced with `44`. Finally, the three-element vector is printed.

Click here to run this program in the Stacker.

Lispy |

(Task 3 of 20) What is the result of running this program?

Lispy [Run]

```
(defvar x (mvec 43 54))
(vec-set! x 0 x)
(vec-ref x 1)
```

Python

```
x = [ 43, 54 ]
x[0] = x
print(x[1])
```

5

`43`

6

The answer is `54`.

▶ Textual explanation

Lispy |

What is the result of running this program?

Lispy [Run]

```
(defvar x (mvec 84 73 69 52))
(vec-set! x 1 x)
(vec-ref x 0)
```

Pseudo

```
let x = vec[84, 73, 69, 52]
x[1] = x
print(x[0])
```

`84`

8

You predicted the output correctly 🎉🎉🎉

9

Lispy |

(Task 4 of 20) What is the result of running this program?

Lispy [Run]

```
(defvar x (mvec 74 82))
(defvar y (mvec x))
(vec-set! x 0 y)
(vec-ref x 1)
```

JavaScript

```
let x = [ 74, 82 ];
let y = [ x ];
x[0] = y;
console.log(x[1]);
```

`82`

11

You predicted the output correctly 🎉🎉🎉

12

The program binds `x` to a two-element vector and `y` to a one-element vector. After that, it replaces the `0`-th element of the two-element vector with the one-element vector. The other element of the two-element vector is still `82`.

Click here to run this program in the Stacker.

Lispy |

(Task 5 of 20) What is the result of running this program?

Lispy [Run]

```
(defvar x (mvec 53))
(defvar v (mvec 72 x))
(vec-set! x 0 72)
v
```

Python

```
x = [ 53 ]
v = [ 72, x ]
x[0] = 72
print(v)
```

`#(72 #(53))`

14

Please briefly explain why you think the answer is `#(72 #(53))`.

15

Vector inside the vector, changing the result of the outer most vector

16

The answer is `#(72 #(72))`.

▶ Textual explanation

17

What is your thought? (Feel free to skip this question.)

18

(You skipped the question.) 19

Lispy |

What is the result of running this program?

Lispy [Run]

```
(defvar m (mvec 0))
(defvar v (mvec 52 m 53))
(vec-set! m 0 71)
v
```

Scala 3

```
val m = Buffer(0)
val v = Buffer(52, m, 53)
m(0) = 71
println(v)
```

21

`#(52 #(71) 53)`

22

You predicted the output correctly 🎉🎉🎉

Lispy |

(Task 6 of 20) What is the result of running this program?

Lispy [Run]

```
(defvar x (mvec 62))
(defvar y x)
(vec-set! y 0 34)
x
```

Python

```
x = [ 62 ]
y = x
y[0] = 34
print(x)
```

24

`#(62)`

25

Please briefly explain why you think the answer is `#(62)`.

26

Clicked wrong thing, it is #(34)

27

The answer is `#(34)`.

▸ Textual explanation

Lispy |

What is the result of running this program?

Lispy [Run]

```
(defvar foo (mvec 65 48))
(defvar bar foo)
(vec-set! bar 0 55)
foo
```

JavaScript

```
let foo = [ 65, 48 ];
let bar = foo;
bar[0] = 55;
console.log(foo);
```

29

`#(55 48)`

```
#(55, 48)
```

You predicted the output correctly 🎉🎉🎉

30

Lispy |

(Task 7 of 20) What is the result of running this program?

Lispy [Run]

```
(defvar x (mvec 77))
(defvar y x)
(vec-set! x 0 34)
y
```

Scala 3

```
val x = Buffer(77)
val y = x
x(0) = 34
println(y)
```

32

```
#(34)
```

33

You predicted the output correctly 🎉🎉🎉

The program creates a one-element vector (the only element being 77) and binds it to both `x` and `y`. The `0`-th element of the vector is then replaced with `34`. So, the vector is printed as `#(34)`.

Click here to run this program in the Stacker.

Lispy |

(Task 8 of 20) What is the result of running this program?

Lispy [Run]

```
(defvar x (mvec 71 86))
(deffun (f y)
  (vec-set! y 0 34))
(f x)
x
```

Python

```
x = [ 71, 86 ]
def f(y):
    y[0] = 34
    return
print(f(x))
print(x)
```

35

```
error
```

36

The answer is `#(34 86)`.

▶ Textual explanation

Lispy |

What is the result of running this program?

Lispy [Run]

```
(defvar a (mvec 55 17))
(deffun (foobar b)
  (vec-set! b 0 52))
```

JavaScript

```
let a = [ 55, 17 ];
function foobar(b) {
  b[0] = 52;
```

```
(foobar a)
a
```

```
  return;
}
console.log(foobar(a));
console.log(a);
```

```
#(52 17)
```

38

You predicted the output correctly 🎉🎉🎉

39

Lispy |

(Task 9 of 20) What is the result of running this program?

Lispy [Run]

```
(defvar x (mvec 99 83))
(deffun (f y)
  (vec-set! x 0 34)
  y)
(f x)
```

Pseudo

```
let x = vec[99, 83]
fun f(y):
  x[0] = 34
  return y
end
print(f(x))
```

```
#(34 83)
```

41

You predicted the output correctly 🎉🎉🎉

42

The program creates a vector and binds it to `x`. The function call `(f x)` binds `y` to the same vector. The function replaces the `0`-th element of the vector with `34` and then returns the vector. So, the final result is the vector, which is printed as `#(34 83)`.

Click here to run this program in the Stacker.

(Task 10 of 20) What did you learn about vectors from these programs?

43

Vectors are mutable and each element can be manipulated individually, as well as vectors can contain other vectors.

44

(Task 11 of 20) A vector can be referred to by more than one variable and even by other vectors (including itself). Referring to a vector does not create a copy of the vector; rather, they share the same vector. Specifically

45

- Binding a vector to a new variable does not create a copy of that vector.
- Vectors that are passed to a function in a function call do not get copied.

- Creating new vectors that refer to existing ones does not create new copies of the existing vectors.

The references share the same vector. That is, vectors can be **aliased**.

Any feedback regarding these statements? Feel free to skip this question. 46

(You skipped the question.) 47

(Task 12 of 20) Now please scroll back and select 1-3 programs that make the above points. 48

You don't need to select *all* such programs.

(*You selected 2 programs*) 49

Okay. How do these programs (23,28) support the point? 50

We define it as foo, then in the function name it bar based on the input arguments, then when the vector is passed in, we can modify it inside the function, and this is reflected after the vector exits the scope of the function 51

Lispy |

(Task 13 of 20) Reconsider these two programs that you might have seen. The only difference is in `(defvar y ___)`.

Lispy [Run] Scala 3

```
(defvar x (mvec 52 96))   val x = Buffer(52, 96)
(defvar y x)              val y = x
(vec-set! x 0 34)         x(0) = 34
y                         println(y)
```

Lispy [Run] Scala 3

```
(defvar x (mvec 52 96))   val x = Buffer(52, 96)
(defvar y (mvec 52 96))   val y = Buffer(52, 96)
(vec-set! x 0 34)         x(0) = 34
y                         println(y)
```

It is tempting to describe the variables as

- `x` is bound to `#(52 96)`
- `y` is bound to `#(52 96)`

This description does not help us to understand the program because it can't explain why `(vec-set! x 0 34)` mutates `y` in one case, and does not in the other.

What is a better way to describe the bindings that help solve this problem?

In the first case, we bind y to x, which itself is bound to the vector, where in the second case x and y are bound to their own vectors, which are separate memory addresses [53]

We can say, in the first program [54]

- `x` is bound to `@100`
- `y` is bound to `@100`

where

- `@100` is `#(52 96)`

While for the other program

- `x` is bound to `@100`
- `y` is bound to `@200`

where

- `@100` is `#(52 96)`
- `@200` is `#(52 96)`

(Task 14 of 20) In SMoL, each vector has its own unique **heap address** (e.g., `@100` and `@200`). The mapping from addresses to vectors is called the **heap**. [55]

(**Note**: we use `@ddd` (e.g., `@123`, `@200`, and `@100`) to represent heap addresses. Heap addresses are *random*. The numbers don't mean anything.)

Any feedback regarding these statements? Feel free to skip this question. [56]

(You skipped the question.) [57]

(Task 15 of 20) Which choice best describes the status of the heap at the end of the following program? Lispy |

Lispy [Run]

```
(defvar x 3)
(defvar v (mvec 1 2 x))
```

Pseudo

```
let x = 3
let v = vec[1, 2, x]
```

59

A. `@1 = #(1 2 3)`
B. `@1 = #(1 2 x)`
C. `There is nothing in the heap.`

60

`@1 = #(1 2 x)`

61

The answer is `@1 = #(1 2 3)`.

C is wrong because the `(mvec 1 2 x)` creates a vector. Every vector is stored on the heap.

B is wrong because vectors refer values, while `x` is not a value.

Lispy |

(Task 16 of 20) Which choice best describes the status of the heap at the end of the following program?

Lispy [Run]

```
(defvar mv (mvec 3))
(defvar mv2 (mvec mv mv))
(vec-set! (vec-ref mv2 0) 0 42)
```

Pseudo

```
let mv = vec[3]
let mv2 = vec[mv, mv]
mv2[0][0] = 42
```

63

A. `@100 = #(3); @200 = #(@100 @100)`
B. `@100 = #(3); @200 = #(@300 @100); @300 = #(42)`
C. `@100 = #(3); @200 = #(#(42) #(3))`
D. `@100 = #(42); @200 = #(@100 @100)`
E. `@100 = #(42); @200 = #(#(42) #(42))`
F. `There is nothing in the heap.`

64

`@100 = #(3); @200 = #(#(42) #(3))`

65

The answer is `@100 = #(42); @200 = #(@100 @100)`.

E and **C** are wrong. Vectors refer values. `#(42)` and `#(3)` are not values, although some vector values are printed like them. So, `@200 = #(#(42) #(42))` and `@200 = #(#(42) #(3))` can not be valid.

`(mvec 3)` creates a 1-element vector, `@100`. The only element is `3`. `(mvec mv mv)` creates a 2-element vector, `@200`. Both elements of `@200` are `@100`. No more vectors are created, which means **B** must be wrong. So, then, the correct answer must be **A** or **D**.

However, the subsequent mutation changes `@100` (the first element of `@200`). The 0-th element of `@100` is mutated to `42`. So, **D** is the correct answer.

Lispy |

(Task 17 of 20) The following program defines two variables but creates nothing on the heap.

Lispy [Run]

```
(defvar x 2)
(defvar y 3)
x
y
```

JavaScript

```
let x = 2;
let y = 3;
console.log(x);
console.log(y);
```

The following program defines no variables but creates two things on the heap.

Lispy [Run]

```
(mvec 1 (mvec 2 3))
```

JavaScript

```
console.log([ 1, [ 2, 3 ] ]);
```

What did you learn from this pair of programs?

Variables are not heap allocated while vectors are

67

(Task 18 of 20) - Creating a vector does not inherently create a binding.

68

- Creating a binding does not necessarily alter the heap.

Any feedback regarding these statements? Feel free to skip this question.

69

(You skipped the question.)

70

Lispy |

(Task 19 of 20) Here is a program that confused many students

Lispy [Run]

```
(defvar v (mvec 1 2 3 4))
(defvar vv (mvec v v))
(vec-set! (vec-ref vv 1) 0 100)
vv
```

Pseudo

```
let v = vec[1, 2, 3, 4]
let vv = vec[v, v]
vv[1][0] = 100
print(vv)
```

Please

1. Run this program in the stacker by clicking the green run button above;
2. The stacker would show how this program produces its result(s);
3. Keep clicking ⏭ Next until you reach a configuration that you find particularly helpful;

4. Click 🔗 Share This Configuration to get a link to your configuration;
5. Submit your link below;

72

https://www.cs.unb.ca/~bremner/teaching/cs4613/stacker/?syntax=Lispy&randomSeed=smol-tutor&hole=%E2%80%A2&nNext=0&program=%0A%28defvar+v+%28mvec+1+2+3+4%29%29%0A%28defvar+vv+%28mvec+v+v%29%29%0A%28vec-set%21+%28vec-ref+vv+1%29+0+100%29%0Avv%0A&readOnlyMode=

73

(Task 20 of 20) Please write a couple of sentences to explain how your configuration explains the result(s) of the program.

74

We can see that the replace vector is operating in the context of vv, and we can see which element it is operating on in the heap

75

Let's review what we have learned in this tutorial.

A vector can be referred to by more than one variable and even by other vectors (including itself). Referring to a vector does not create a copy of the vector; rather, they share the same vector. Specifically

- Binding a vector to a new variable does not create a copy of that vector.
- Vectors that are passed to a function in a function call do not get copied.
- Creating new vectors that refer to existing ones does not create new copies of the existing vectors.

The references share the same vector. That is, vectors can be **aliased**.

In SMoL, each vector has its own unique **heap address** (e.g., `@100` and `@200`). The mapping from addresses to vectors is called the **heap**.

(**Note**: we use `@ddd` (e.g., `@123`, `@200`, and `@100`) to represent heap addresses. Heap addresses are *random*. The numbers don't mean anything.)

- Creating a vector does not inherently create a binding.
- Creating a binding does not necessarily alter the heap.

You have finished this tutorial 🎉🎉🎉

Please print the finished tutorial to a PDF file so you can review the content in the future. **Your instructor (if any) might require you to submit the PDF.**

Start time: 1737146449319