Syntax: Lispy JS PY Scala 3

In this tutorial, we will learn even more about **mutable values**, illustrated with **lists**.

Syntax: Lispy JS PY Scala 3

Which choice best describes the heap at the end of the following program?

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

```
x = [2]
x[0] = 33
print(x)
```

Run ▶

Syntax: Lispy JS PY Scala 3

A. `@100 = #(2); @200 = #(33)`
B. `@200 = #(2)`
C. `@200 = #(33)`
D. `@200 = 33`

Syntax: Lispy JS PY Scala 3

`@200 = [33]`

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

Exactly one list was created. So, there must be at most one list on the heap. This rules out **A**.

For now, the heap maps addresses only to lists. This rules out **D**.

The heap looks like **B** after evaluating `[2]`. However, the subsequent mutation changes the list. So, the correct answer is **C**.

Syntax: Lispy JS PY Scala 3

Which choice best describes the heap at the end of the following program?

```
m = [1, 2]
m[1] = [3, 4]
```

Run ▶

Syntax: Lispy JS PY Scala 3

A. `@100 = #(@200 1); @200 = #(3 4)`
B. `@100 = #(1 @200); @200 = #(3 4)`
C. `@100 = #(1 #(3 4))`
D. `@100 = #(1 #(3 4)); @200 = #(3 4)`
E. `@100 = #(1 2)`
F. `@100 = #(1 2); @200 = #(3 4)`

Syntax: Lispy JS PY Scala 3

`@100 = [1, [3, 4]]`

Syntax: Lispy JS PY Scala 3

The answer is `@100 = #(1 @200); @200 = #(3 4)`.

Two lists are created. So, there must be at least two things on the heap. This rules out **C** and **E**.

D is wrong because lists refer values (e.g., `1`, `2`, and `@200`). `[3, 4]` is not itself a value; it's the *printing* of the value that resides at `@200`. **F** can be the heap after the second list is created. However, the subsequent mutation changes `@100`. So, **F** is wrong.

The mutation replaces the 1-th (i.e., second) element rather than the 0-th element, so **B** is correct, while **A** is wrong.

Syntax: Lispy JS PY Scala 3

Which choice best describes the heap at the end of the following program?

```
x = [3]
v = [1, 2, x]
x[0] = 4
print(v)
```

Run ▶

Syntax: Lispy JS PY Scala 3

A. `@100 = #(3); @200 = #(1 2 @100)`
B. `@100 = #(3); @200 = #(1 2 @300); @300 = #(4)`
C. `@100 = #(4); @200 = #(1 2 @100)`
D. `@100 = #(4); @200 = #(1 2 #(3))`
E. `@100 = #(4); @200 = #(1 2 #(4))`
F. `@100 = #(4); @200 = #(1 2 3)`
G. `@100 = #(4); @200 = #(1 2 4)`

Syntax: Lispy JS PY Scala 3

`@100 = #(4); @200 = #(1 2 @100)`

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

D and **E** are wrong because lists refer values. `[3]` and `[4]` are not values, although they can be the printed representation of a list.

F and **G** are wrong because the 2-th element of the 3-element list must be a list. The 3-element list is created by `[1, 2, x]`. The value of `x` is a list at that moment. This 3-element list is never mutated.

B is wrong because only two lists are created. There must be at most two lists on the heap.

A can be the heap after the two lists are created. However, the subsequent mutation changes the shorter list. So, **C** is the correct answer.

Syntax: Lispy JS PY Scala 3

Which choice best describes the heap at the end of the following program?

```
mv = [4, 5]
print(mv[0])
```

Run ▶

Syntax: Lispy JS PY Scala 3

A. `@100 = #(@200 @300); @200 = 4; @300 = 5`
B. `@100 = #(4 5)`
C. `@100 = #(4 5); @200 = 4`

Syntax: Lispy JS PY Scala 3

```
@100 = [4, 5]
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

For now, the heap maps addresses only to lists. This rules out **A** and **C**.

So, **B** is correct.

Syntax: Lispy JS PY Scala 3

Which choice best describes the heap at the end of the following program?

```
v = [1, 2, 3]
vv = [v, v]
vv[1][0] = 6
print(vv[0])
```

Run ▶

Syntax: Lispy JS PY Scala 3

A. `@100 = #(1 2 3); @200 = #(@100 @100)`
B. `@100 = #(1 2 3); @200 = #(@100 @300); @300 = #(6 2 3)`
C. `@100 = #(1 2 3); @200 = #(#(1 2 3) #(6 2 3))`
D. `@100 = #(1 2 3); @200 = #(1 2 3); @300 = #(6 2 3); @400 = #(@200`
E. `@100 = #(1 2 3); @200 = #(6 @100)`
F. `@100 = #(6 2 3); @200 = #(@100 @100)`
G. `@100 = #(6 2 3); @200 = #(#(1 2 3) #(6 2 3))`

Syntax: Lispy JS PY Scala 3

```
@100 = #(6 2 3); @200 = #(@100 @100)
```

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

Lists refer values (e.g., `1` and `@200`). This rules out **C** and **G**.

Two lists are created. So, there must be two lists on the heap. This rules out **B** and **D**.

`vv` is bound to a 2-element list, and the 1-th element of the 2-element list must be the 3-element list. The mutation replaces the 0-th element in the 3-element list with `6`. So, **F** is the correct answer, while **A** does not reflect the effect of the mutation, and **E** mutates the wrong list.

Syntax: Lispy JS PY Scala 3

Which choice best describes the heap at the end of the following program?

```
x = [1, 0, 2]
x[1] = x
print(len(x))
```

Run ▶

Syntax: Lispy JS PY Scala 3

A. `@100 = #(1 @100 2)`
B. `@100 = #(1 @200 2); @200 = #(1 0 2)`
C. `@100 = #(1 #(1 0 2) 2)`
D. `@100 = #(1 0 2)`

Syntax: Lispy JS PY Scala 3

`@100 = #(1 @100 2)`

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

C is wrong because lists refer values. `[1, 0, 2]` is not a value, although it can be some list values printed.

B is wrong because only one list is created. There must not be two lists on the heap.

D can be the heap after the list is created. But the subsequent mutation replaces the 1-th element of `@100` with `@100`. So, **A** is the correct answer.

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: 1711098542560