


(Task 1 of 7) In this tutorial, we will learn *more* about using functions as values.

0

(Task 2 of 7) What is the result of running this program?

Lispy | 

Lispy [Run 	Scala 3
<pre>(defvar x 1) (deffun (f) (deffun (addx y) (+ x y)) addx) (defvar g (f)) (set! x 2) (g 0)</pre>	<pre>var x = 1 def f = def addx(y : Int) = x + y addx var g = f x = 2 println(g(0))</pre>

2

2

Please briefly explain why you think the answer is 2.

3

Super convoluted at to bind x to 2 and then add the argument to it

4

You predicted the output correctly 🎉🎉🎉🎉


5

x is bound to 1. g is bound to the function addx (set! x 2) binds x to 2. So, the value of (g 0) is the value of (addx 0), which is the value of (+ 2 0), which is 2.

Click [here](#) to run this program in the Stacker.

(Task 3 of 7) What is the result of running this program?

Lispy | 

Lispy [Run 	Pseudo
<pre>(deffun (foo) (defvar n 0) (deffun (bar) (set! n (+ n 1)) n) bar) (defvar f (foo)) (defvar g (foo)) (f) (f) (g)</pre>	<pre>fun foo(): let n = 0 fun bar(): n = n + 1 return n end return bar end let f = foo() let g = foo() print(f()) print(f()) print(g())</pre>

1 2 1

7

You predicted the output correctly 🎉🎉🎉

8

Every time `foo` is called, it creates a *new* environment frame that binds `n`. So, `f` and `g` have different bindings for the `n` variable. When `f` is called the first time, it mutates its binding for the `n` variable. So, the second call to `f` produces `2` rather than `1`. `g` has its own binding for the `n` variable, which still binds `n` to `0`. So, `(g)` produces `1` rather than `3`.

Click [here](#) to run this program in the Stacker.

(Task 4 of 7) What is the result of running this program?

Lispy | 🍷

Lispy [Run]	Pseudo
<code>(deffun (bar y)</code>	<code>fun bar(y):</code>
<code>(deffun (addx x)</code>	<code>fun addx(x):</code>
<code>(+ x y))</code>	<code>return x + y</code>
<code>addx)</code>	<code>end</code>
<code>(defvar f (bar 2))</code>	<code>return addx</code>
<code>(defvar g (bar 4))</code>	<code>end</code>
<code>(f 2)</code>	<code>let f = bar(2)</code>
<code>(g 2)</code>	<code>let g = bar(4)</code>
	<code>print(f(2))</code>
	<code>print(g(2))</code>

4 6

10

You predicted the output correctly 🎉🎉🎉

11

The value of `(bar 2)` is the function `addx` defined in an environment where `y` is bound to `2`. The value of `(bar 4)` is *another* `addx` defined in an environment where `y` is bound to `4`. The two `addx` functions are *different* values. So, the value of `(f 2)` is `4`, while the value of `(g 2)` is `6`.

Click [here](#) to run this program in the Stacker.

(Task 5 of 7) What did you learn about functions from these programs?

12

You can setup environments that persist values by returning an inner function to the one defined inside it

13

(Task 6 of 7) Functions remember the environment in which they are defined. That is, function bodies are "enclosed" by the environments in which the function values are created. So, function values are called *closures*.

14

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

15

(You skipped the question.)

16

(Task 7 of 7) Please scroll back and select 1-3 programs that make the point that

17

Functions remember the environment in which they are defined.

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

(You selected 2 programs)

18

Okay. How do these programs (6,9) support the point?

19

We setup a var in the environment that is local to a bound variable to a function, which is used in the function calls

20

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

21

Functions remember the environment in which they are defined. That is, function bodies are "enclosed" by the environments in which the function values are created. So, function values are called *closures*.

You have finished this tutorial 🎉🎉🎉

Please 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: 1737765210222