(Task 1 of 14) In this tutorial, we will learn more about **variable assignments** and **mutable variables**. 0

Lispy |

(Task 2 of 14) What is the result of running this program?

Lispy [Run] Python

```
(defvar x 12)   x = 12
(deffun (f)     def f():
  x)                return x
(deffun (g)     def g():
  (set! x 0)        global x
  (f))              x = 0
(g)                 return f()
(set! x 1)      print(g())
(f)             x = 1
                print(f())
```

```
0 1
```
2

You predicted the output correctly 🎉🎉🎉 3

There is exactly one variable `x`. The `x` in `(set! x 0)` refers to that variable. Calling `f` evaluates `(set! x 0)`, which mutates `x`. When the value of `x` is eventually printed, we see the new value.

Click here to run this program in the Stacker.

Lispy |

(Task 3 of 14) What is the result of running this program?

Lispy [Run] JavaScript

```
(defvar x 1)    let x = 1;
(deffun (f n)   function f(n) {
  (+ x n))        return x + n;
(set! x 2)      }
(f 30)          x = 2;
                console.log(f(30));
```

```
32
```
5

You predicted the output correctly 🎉🎉🎉 6

The program binds `x` to `1` and then defines a function `f`. `x` is then bound to `2`. So, `(f 30)` is `(+ x 30)`, which is 32.s

Click here to run this program in the Stacker.

Lispy |

(Task 4 of 14) What is the result of running this program?

Lispy [Run]	JavaScript

`(set! foobar 2)`
`foobar`
	`foobar = 2;`
	`console.log(foobar);`

8

`2`

9

The answer is `error`.

▼ Textual explanation

You might think `(set! foobar 2)` defines `foobar`. However, it errors. In SMoL, variable assignments mutate existing bindings and never create new bindings.

Click here to run this program in the Stacker.

Lispy |

What is the result of running this program?

Lispy [Run]	JavaScript

`(set! foo 42)`
`foo`
	`foo = 42;`
	`console.log(foo);`

11

`error`

12

You predicted the output correctly 🎉🎉🎉

13

(Task 5 of 14) What did you learn about variable assignment from these programs?

14

You cannot set a var without defining it first

15

(Task 6 of 14) - Variable assignments mutate existing bindings and do not create new bindings.

- Functions refer to the latest values of variables defined outside their definitions. That is, functions do not remember the values of those variables from when the functions were defined.

Any feedback regarding these statements? Feel free to skip this question. 16

(You skipped the question.) 17

(Task 7 of 14) Please scroll back and select 1-3 programs that make the following point: 18

> Variable assignments mutate existing bindings and do not create new bindings.

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

(*You selected 1 programs*) 19

Okay. How does this program (10) support the point? 20

It errors otherwise 21

(Task 8 of 14) Please scroll back and select 1-3 programs that make the following point: 22

> Functions refer to the latest values of variables defined outside their definitions. That is, functions do not remember the values of those variables from when the functions were defined.

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

(*You selected 1 programs*) 23

Okay. How does this program (4) support the point? 24

It simply uses the new defined x = 0 that is defined after the initial definition 25

Lispy |

(Task 9 of 14) Although we keep saying "this variable is mutated", the variables themselves are *not* mutated. What is actually mutated is *the binding between the variables and their values*.

Blocks have been a good way to describe these bindings. However, they can't explain variable mutations: a block is a piece of source code, and we can't change the source code by mutating a variable. Besides, blocks can't explain how parameters might be bound differently in different function calls. Consider the following program:

Lispy [Run ▶] JavaScript

```
(deffun (f n)    function f(n) {
  (+ n 1))         return n + 1;
(f 2)            }
(f 3)            console.log(f(2));
                 console.log(f(3));
```

In this program, `n` is bound to `2` in this first function call, and `3` in the second. Blocks can't explain how `n` is bound differently because the two calls share the same block: the body of `f`.

What is a better way to describe the binding between variables and their values?

A variable is a reference to a heap allocated value that exists in an environment. The binding is simply the link, with the variable having a name in an environment and the value existing on the heap. 27

(Task 10 of 14) **Environments** (rather than blocks) bind variables to values. 28

Similar to vectors, environments are created as programs run.

Environments are created from blocks. They form a tree-like structure, respecting the tree-like structure of their corresponding blocks. So, we have a primordial environment, a top-level environment, and environments created from function bodies.

Every function call creates a new environment. This is very different from the block perspective: every function corresponds to exactly one block, its body.

Any feedback regarding these statements? Feel free to skip this question. 29

(You skipped the question.) 30

(Task 11 of 14) Which language construct(s) create new bindings? 31

32
- ✓ Definitions
- ☐ Variable mutations (e.g., `(set! x 3)`)
- ☐ Variable references
- ☐ Function calls

You should also have chosen Function calls 33

Both definitions and function calls create bindings. Variable assignment mutates existing bindings but doesn't create new bindings.

(Task 12 of 14) Which language construct(s) mutate bindings? 34

35

- ☐ Definitions
- ✓ Variable mutations (e.g., `(set! x 3)`)
- ☐ Variable references
- ☐ Function calls

You gave the correct answer 🎉🎉🎉 36

Lispy | 🎲

(Task 13 of 14) Here is a program that confused many students 37

Lispy [Run ▶] Scala 3

```
(defvar x 5)       var x = 5
(deffun (f x y)    def f(x : Int, y : Int) =
  (set! x y))        x = y
(f x 6)            println(f(x, 6))
x                  println(x)
```

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;

https://www.cs.unb.ca/~bremner/teaching/cs4613/stacker/?syntax=Lispy&randomSeed=smol-tutor&hole=%E2%80%A2&nNext=2&program=%0A%28defvar+x+5%29%0A%28deffun+%28f+x+y%29%0A++%28set%21+x+y%29%29%0A%28f+x+6%29%0Ax%0A&readOnlyMode= 38

(Task 14 of 14) Please write a couple of sentences to explain how your configuration explains the result(s) of the program. 39

When we call set, we set the binding of x for only the x that exists in the function environment, as a new x is defined in the arguments of the function 40

Let's review what we have learned in this tutorial. 41

- Variable assignments mutate existing bindings and do not create new bindings.
- Functions refer to the latest values of variables defined outside their definitions. That is, functions do not remember the values of those variables from when the functions were defined.

Environments (rather than blocks) bind variables to values.

Similar to vectors, environments are created as programs run.

Environments are created from blocks. They form a tree-like structure, respecting the tree-like structure of their corresponding blocks. So, we have a primordial environment, a top-level environment, and environments created from function bodies.

Every function call creates a new environment. This is very different from the block perspective: every function corresponds to exactly one block, its body.

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: 1737762399560