lec10

November 5, 2018

0.1 6.009 Week 10 Lecture

In lab 8A/8B, we are implementing an interpreter for a dialect of a programming language called LISP.

Python:

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Scheme (LISP):

```
(define (fib n)
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))
  )
)
```

Why bother writing interpreters?

- It's just *so cool*
- It can help you understand the semantics of languages you already know
- There is something powerful about the fact that an interpreter is just another program

Why LISP?

- LISP is weird/cool
  - "A language that doesn't affect the way you think about programming, is not worth learning." -Alan Perlis
- LISP has very minimal syntax (spend less time thinking about parsing and more time on evaluation)
- MIT and LISP have a long history
  - LISP invented here in 1958
  - one widely-used dialect (Scheme) invented here as well, used in 6.001 from ~1980-2007

0.1.1 Today: More on Python Variable BInding/Lookup

```
In [ ]: # brief example: HTML tagger

        tagger_functions = {}
        for tag in ('b', 'i', 'u', 'marquee', 'blink'):
            tagger_functions[tag] = lambda x: '<%s>%s</%s>' % (tag, x, tag)


        print(tagger_functions['b']('this is bold!'))
```

0.1.2 Python Variable Scoping and Closures

In lab 8, we'll be implementing LISP variable scoping semantics. It turns out that these semantics are almost exactly equivalent to Python's, including how function calls are handled.

In particular, we'll introduce a few mechanisms for affecting variable bindings outside of the current local scope (which will have analogues in LISP).

0.1.3 1. Python local variables shadow variables in surrounding scope

```
In [ ]: x = 0
        def outer():
            x = 1
            def inner():
                print("inner:", x)

            inner()
            print("outer:", x)

        outer()
        print("global:", x)
```

0.1.4 2. How does python know a variable is local?

- If you **assign** a variable in a local scope, it's local and shadows the variable in the surrounding scope.
- If you **access** a variable before it is assigned in the local scope, the surrounding variable is used. Note, however, that once you've accessed the variable in the surrounding scope, you can't assign to it!

```
In [ ]: x = 0
        def outer():
            x = 1
            def inner():
                print("inner:", x)

            inner()
            print("outer:", x)
```

```
outer()
print("global:", x)
```

0.1.5 3. Sharing Information: What if we want to assign a variable in the global scope?

- Python **global** declaration tells python to use the variable in the global scope (skipping over any intervening nonlocal/surrounding scoped variables).

```
In [ ]: x = 0
        def outer():
            x = 1
            def inner():
                global x     ## THIS WAS CHANGED
                x = 2
                print("inner:", x)

            inner()
            print("outer:", x)

        outer()
        print("global:", x)
```

0.1.6 4. What if we want to assign a variable in the surrounding scope?

- Python **nonlocal** declaration tells python to use the variable in the nearest surrounding scope (except global scope).

```
In [ ]: x = 0
        def outer():
            x = 1
            def inner():
                nonlocal x     ## HERE IS THE CHANGE
                x = 2
                print("inner:", x)

            inner()
            print("outer:", x)

        outer()
        print("global:", x)
```

- Note that a **nonlocal** variable needs to be present in a surrounding scope, not including the global scope.

0.2 Python Closures Example

The idea of a "closure" is that a function definition remembers the environment in which it was defined, so that later when the function is called the function has access to the variables in the enclosing environment.

```
In [ ]: def derivative(func, delta=1e-6):
            def _fprime(x):
                return (func(x+delta)-func(x-delta))/(2*delta)
            return _fprime

        def f(x):
            return 4*x**3

        fp = derivative(f)
        fp(2)
```

0.2.1 Closures enable object-oriented programming styles

In python, we have classes, methods, instances, etc., so those are usually most convenient. However, we could build something similar using the power of closures and local state. One such style is often referred to as **message passing**:

```
In [ ]: def make_account(balance):
            pass
```