CS61A Notes – Week 4: Environments and Newton's Method

Environment Diagrams

We have already seen what we call the "environment model" of evaluation. An **environment frame** is a box in which we store bindings from variables to their values. A frame can *extend* another, by which we mean that the the frame has access to all the variables defined in the frame it extends, plus any of its own. We call the series of frames we are currently working with the **current environment**.

Whenever we evaluate a user-defined function, we create a new frame that extends an existing frame, set this sequence of frames as our current environment, and evaluate the body of the function in this environment.

What we will be focusing on here is the question of *which* frame we should extend when evaluating a function. There are really two simple rules to go by:

1. **When a function is defined**, the function remembers the current environment it was defined in. We show this by drawing an arrow from the top right corner of the function to the top-most (newest) frame in our current environment.
2. **When a function is evaluated**, we create a new frame and have it extend the function's remembered environment (which its arrow points to), regardless of the environment from which the function was called.

This method of extending frames is called **lexical scoping**. The guiding principle behind this method of evaluation is basically that a function is always evaluated in the same environment it was defined in. In terms of nested function definitions, this evaluation method allows the inner functions to see variables defined by the enclosing function. For instance, if the outer function has a parameter called val, any inner functions can refer to that same variable in their bodies. Try out the exercises below to verify this fact.

Lambda expressions:

So far, we have seen ways for functions to return other functions by using nested inner functions. But, what if the function you need is very short and will only be used in one particular situation? The solution is to use a `lambda`. A `lambda` expression has the following syntax:

```
lambda <args>: <body>
```

With this simple expression, you can define functions on the fly, without having to use `def` statements and without having to give them names. In other words, lambda expressions allow you to create anonymous functions. There is a catch though: The `<body>` must be a single expression, which is also the return value of the function.

One other difference between using the `def` keyword and `lambda`'s we would like to point out is that `def` is a statement, while `lambda` is an expression. Evaluating a `def` statement will have a side-effect, namely it creates a new function binding in the current environment. On the other hand, evaluating a lambda expression will not change the environment unless we do something with the

function created by the lambda. For instance, we could assign it to a variable or pass it as a function argument.

Questions:

Draw the environment diagrams for each of the following:

```
#1
>>> square = lambda x: x * x
>>> def double(f):
...     def doubler(x):
...         return f(f(x))
...     return doubler
>>> 4th_power = double(square)
>>> 4th_power(2)
16
```

#2 (also fill in the return values)

```
>>> a = 5
>>> g = lambda x: x + 3
>>> def apply(f):
...     def call(x):
...         return f(x)
...    return call
>>> f = apply(g)
>>> f(2)
__________
>>> g = lambda x: x * x
>>> f(3)
__________
```

#3

```
def make_arithmetic_generator(fn):
    return lambda x: (lambda y: fn(x,y))

>>> adder_generator = make_arithmetic_generator(add)
>>> add_4 = adder_generator(4)
>>> add_4(5)
9
```

Newton's Method

Newton's method is an algorithm that is widely used to compute the zeros of functions. It can be used to approximate the root of any continuous, differentiable function.

Intuitively, Newton's method works based on two observations:

- At a point P=(x, f(x)), the root of the function f is in the same direction relative to P as the root of the linear function L that not only passes through P, but also has the same slope as f at that point.
- Over any very small region, we can approximate f as a linear function. This is one of the fundamental principles of calculus.

Therefore, at each point, we iteratively solve for the zero of such a function L and use that as our new guess for the root of f.

Mathematically, we can derive the update equation by using two different ways to write the slope of L:

Let x be our current guess for the root, and x^* be the point want to update our guess to.

Let L be the linear function tangent to f at $(x, f(x))$.

Remember that x^* is the root of L. So, we know two L passes through, namely $(x, f(x))$ and $(x^*, 0)$.

We can write the slope of L as

$$L'(x) = \frac{0 - f(x)}{x^* - x} = \frac{-f(x)}{x^* - x}$$

We also know that L is tangent to f at x, so:

$$L'(x) = f'(x)$$

We can equate these to to get our update equation

$$\frac{-f(x)}{x^* - x} = f'(x) \Rightarrow x^* = x - \frac{f(x)}{f'(x)}$$

We know $f(x)$, and from calclulus, for some very small ϵ:

$$f'(x) = \frac{f(x+\epsilon) - f(x)}{(x+\epsilon) - x} = \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

From the above derivation, we get this algorithm:

```
def derivative(fn, x, dx=0.00001):
    return (fn(x+dx)-fn(x))/dx

def newtons_method(fn, guess=1, max_iterations=100):
    ALLOWED_ERROR_MARGIN = 0.0000001
    i = 1
    while abs(fn(x)) > ALLOWED_ERROR_MARGIN and i <= max_iterations:
        guess = guess - fn(guess) / derivative(fn, x)
        i += 1
    return guess
```

We can abstract this algorithm as a more general method of computation called iterative improvement. With this general algorithm, you start out by guessing a value and then continuously updating the guess until it is a reasonable approximation of the final value you are looking for. Here is the implementation for `iter_improve`. The `update` function takes the current guess value and returns the next guess in the iteration. The `done` function also takes the current guess and return a boolean stating whether or not the current guess is sufficiently accurate to terminate the computation.

```
def iter_improve(update, done, guess=1, max_iterations=100):
    i = 1
    while not done(guess) and i <= max_iterations:
        guess = update(guess)
        i += 1
    return guess

def newtons_method(fn, guess=1, max_iterations=100):
    def newtons_update(guess):
        return guess - fn(guess) / derivative(fn, guess)


    def newtons_done(guess):
        ALLOWED_ERROR_MARGIN= 0.0000001
        return abs(fn(guess)) <= ALLOWED_ERROR_MARGIN

    return iter_improve(newtons_update, newtons_done)
```

Questions:

#4. Write a function `cube_root` that computes the cube root of the function, ie

```
>>> cube_root(8)
2
```

#5. Newton's method converges very slowly (or not at all) if the algorithm happens to land on a point where the derivative is very small. Modify the newton's method implementation, using `iter_improve`, to return `False` if the derivative is under some threshold, say 0.001.

```
def newtons_method2(fn, guess=1, max_iterations=100):
    def newtons_update(guess, min_deriv_val=0.001):




    def newtons_done(guess):




    return iter_improve(newtons_update, newtons_done)
```