THE ENVIRONMENT MODEL 17

COMPUTER SCIENCE 61A
Tom Magrino and Jon Kotker

July 17, 2012

1 Computer Science in the News

Article: "The Eyes Have It: Marketers Now Track Shoppers' Retinas"

Source: Wall Street Journal

Interesting article about using computers to track consumers eyes to determine what labelling catches your eyes the fastest.

2 The Substitution Model is Not Enough!

Up until this point, we have been understanding the code through the substitution model. The way this worked is we would have something like the following code:

```
>>> def square(x):
...     return x * x
>>> square(5)
```

And the way we would explain how this code worked was to take the body of square and replace all instances of the argument `x` with 5.

```
>>> def square(5)
...     return 5 * 5
>>> square(5)
25
```

But even as we went through higher order functions, we saw that this model quickly becomes intractable and hard to read. As an example we could have the intimidating higher order function below.

```
>>> def make_operation(op):
...     def make_oper(arg1):
...         def oper(arg2):
...             return op(arg1, arg2)
...         return oper
...     return make_oper
...
>>> add = lambda x, y: x + y
>>> make_operation(add)(2)(3)
```

And this meant carefully going through and only replacing `op`, `arg1`, and `arg2` when each function with that argument is called.

Now, this was fine because the model still explained computations up until that point in the course. However, with our recent lectures we have moved into the realm of state and this causes problems for our substitution model because it does not handle the concept of identity too well.

For example, given the following code

```
>>> my_lst = [1, 2]
>>> def make_first_five(L):
...     L[0] = 5
...
>>> make_first_five(my_lst)
>>> my_lst
```

using the substitution model, we find that this doesn't really help to describe changes to the `my_lst` object, an example of state in a program, where changes *persist* over time in the program.

```
>>> my_lst = [1, 2]
>>> def make_first_five([1, 2]):
...     [1, 2][0] = 5
>>> make_first_five(my_lst)
>>> my_lst # should be [5, 2]
[1, 2]
```

So we see that our model has begun to fail and we are now left in search of something better.

3 Announcements

- HW8 due today.
- HW9 will be out by 3pm and is due Friday.
- Project 3 is due **Thursday**, July 26th.
- Midterm 2 is next Wednesday, July 25th.
  - **One** page of your own notes.
  - We will give you both the old notes sheet we provided on last midterm and an additional one for new material.
  - Group section will be at the beginning this time, 15 minutes again.
  - Material from beginning through the end of this week.
- Midterm 1 solutions (should) be posted.
  - Regrades are due by July 26th.
  - Attach an explanation to the front of your exam explaining what you think should be regraded and why.
  - We reserve the right to regrade the entire exam.
- Readings for OOP have been posted. We apologize for the delay.
- We will have readings for today's topic soon, but expect it to be extremely similar to these notes.

4 The Environment Model: A More Accurate Model

Enter the environment model, a more accurate model of computation which allows us to avoid these inconsistencies we've been running into. We'll see that it is a much more complicated procedural approach to understanding how names are mapped to values in a program but it is actually much more accurate than the substitution model we've been using thus far.

Before we begin, a couple of disclaimers. Today we will only be concerning ourselves with:

- Numbers, Strings, Tuples, Lists, and Dictionaries (and any composition of them)
- Variables
- Functions

Why? It might seem odd that we're not going to describe how objects work in this model, but we're asking you to hang tight and wait for us to describe why we never really needed a special language support for object oriented programming. We will see that we can simulate all user defined objects and classes as a series of dictionaries, functions, and other simpler tools.

So how does the environment model work? Well the basic idea is simple, instead of substituting each time we want to simulate a function call, we will make a new *frame* that tracks all of the values that are local to the function and use this as a book-keeping tool to track state. We will call the this book-keeping tool an *environment diagram*.

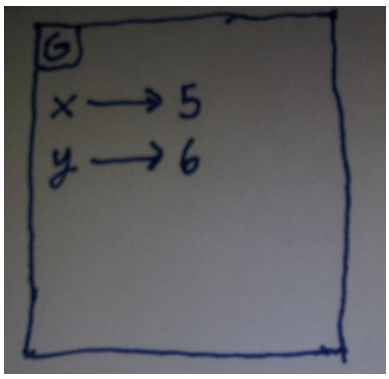
I find that learning environment diagrams happens best by starting with simple examples while building up the rules for drawing them as we go.

4.1 Baby Steps: Simple Variable Assignment and the Global Frame

Let's start with the simplest (interesting) code imaginable and take a look at the environment diagram:

```
>>> x = 5
>>> y = 6
>>> x + y
11
```

So, as we said, we're going to have a diagram of *frames* to keep track of the values associated with each name for a given context in our program. The first frame we will always draw is the *global frame*, or *global environment*, which holds all the names that are globally accessible within the program (unless you make a local variable in a function with the same name). We denote this in the following way:



The way we draw this is:

1. Draw the global frame
2. Go through the code line by line.

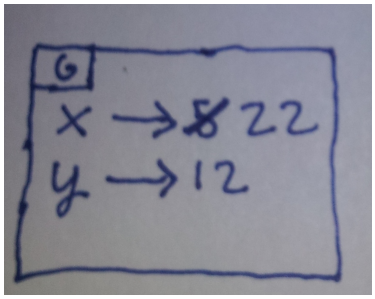
(a) If you see an assignment to a variable, draw a *binding* for the variable to the value.

Notice that we didn't write anything for the last line, we only read off values from the diagram. This is intentional, the environment diagram is simply a way to keep track of values assigned to names in various contexts.

Let's examine something slightly more complicated:

```
>>> x = 5
>>> y = x + 7
>>> x = 22
>>> x + y
34
```

And the resulting diagram is such:



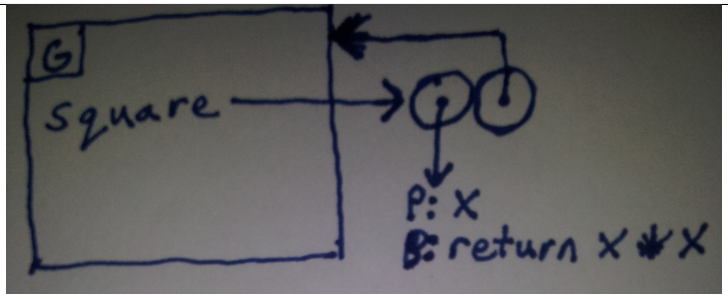
We see that if a variable changes value at some point in the program, we simply update the binding in the frame. Notice that the computation does exactly what we expect, `y` is correctly evaluated to 12 despite it being computed using `x` which later changes. This is because **bindings are always between variable names and values**, we never point to an expression and so we never have to worry about pointing to another variable name and having that name's value change.

4.2 Ramping Up: Defining Functions

So far we've only been handling simple variables and arithmetic, which, let's be honest, is really really boring. Let's move on to incorporating user-defined functions into our diagram. Suppose I had the following code in my program:

```
>>> square = lambda x: x * x
```

We already know how to handle bindings, but how do we represent a function in our diagram? In this class, we will use "double bubble" notation. The resulting diagram is the following:



Double bubbles are sort of funny, but hopefully they will make sense in due time. The first bubble is straight forward, it points to a list of argument names and a copy of the instructions found in the body of the function. We are going to discuss the second bubble later in greater detail, but the short version is that this will point to where the function will look up names that it does not define locally within the body of the function. As it turns out, this frame is the same as the frame we defined the function in, so we can think of this second bubble as pointing back to its "birthplace." We'll see what I mean later in the lecture.

We're going to go ahead and write the "omitted" return statement for the lambda, which helps generalize this notation for any function definition. This is reflected in the diagram.

So you might be saying, "That's great, but I normally use `def` to define my functions." Right you are! Turns out that it's essentially the same thing. To see why let's consider any `def` statement:

```
def name(arg1, arg2):
    <body>
```

Well, in reality, most functions can be rewritten into an equivalent lambda expression (admittedly with a lot of really really really ugly notation and hard work). We can, in our minds, essentially convert the definition statement to the following assignment statement:

```
name = lambda arg1, arg2: <body>
```

Yes, I know it's not EXACTLY the same, but for our purposes we can act like it is. So, as you can see, we would get the same exact environment diagram as we previously got for the following code if we had instead done:

```
>>> def square(x):
...     return x * x
...
```

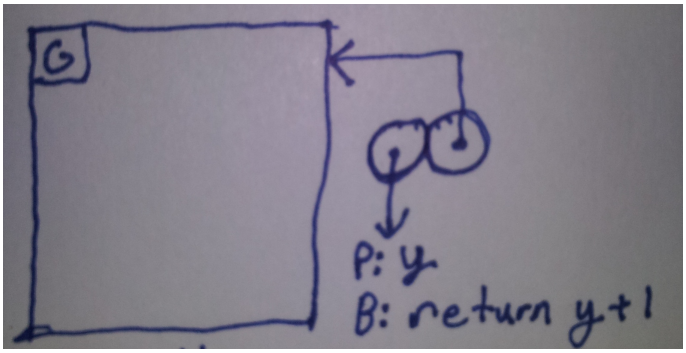
This is because we are doing the exact same operations, defining a function object and then binding a name to that function.

Now, we know from earlier in the semester that we can write lambdas without a corresponding name being given. Can we represent this in our environment diagram? Of

course! We just don't bind a name to the double bubble (the function object). So if we had:

```
>>> lambda y: y + 1
```

We would end up with the following diagram:



4.3 Getting Fancy: Calling Functions

So, great, we have the ability to represent assignments in the global frame. Both to numbers and to functions. We also now have a special notation for representing functions. So now, let's call some functions!

First, let's call a built-in function.[1] We write the following code:

```
>>> max(1, 3, 22, 4)
22
```

As it turns out, the environment diagram for this code is really boring:

It's blank! Why? Well, we COULD represent this the same way as user defined functions. However, this turns out to be both clunky and really unhelpful. We instead opt to represent this operation with nothing and treat it like "magic." This allows us to both avoid

[1] Or a function that was imported into the program, it's the same thing.

clutter in our diagrams and focus on the part of the program we actually care to keep track of: names that we've made and their associated values.

Fine, that was anti-climactic. What about functions that aren't built in? Well this is where we reach the real meat of environment diagrams.[2]

Let's recall the point of an environment diagram: we are supposed to be keeping track of contexts in our program with the various names accessible to us and their associated values. Each specific context of names is represented as a *frame*. We know that functions contain names that are not globally accessible and so this tells us that we need to make a new context, a new *frame*, to hold bindings from names to values. However, we also know that in the context of a function body, we can still access names that are defined in a context outside the body. So we must have a way of representing that we can still reach some names outside the context of the function body!

Let's stop talking about this in abstract terms and see what I mean. Suppose I wrote the following code:

```
>>> y = 6
>>> def square_and_add_y(x):
...     x_squared = x * x
...     return x_squared + y
>>> w = square_and_add_y(5)
```

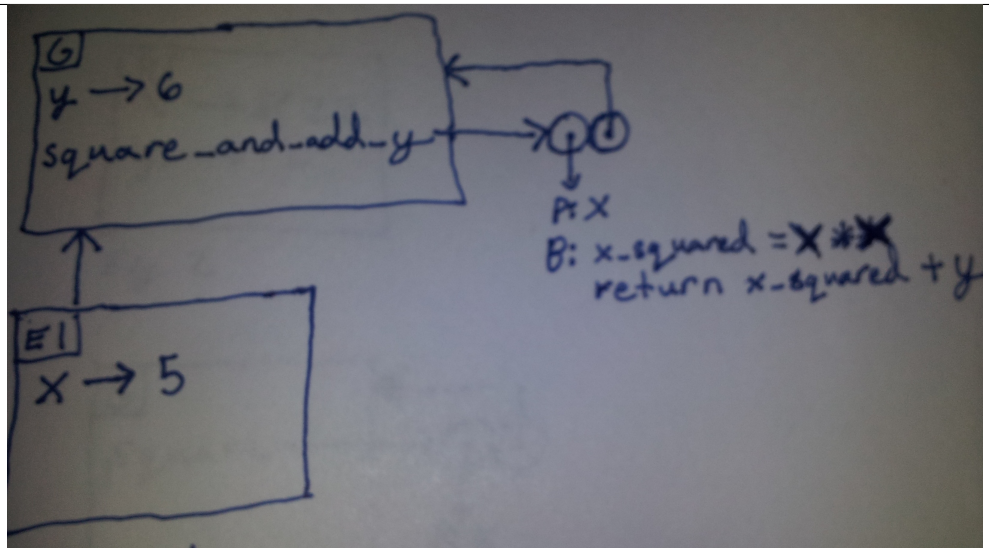
Well, I know how to handle the `def` statement, so I do the same thing as we saw before. Now how do I handle the function call? Well, first thing's first, I know this means that I will begin executing in a new *local* context for the body of the function. This means I need a new frame.

Performing a function call and making a new frame is a multiple-step process.

1. Fully evaluate the function to get a double bubble.
2. Fully evaluate all the arguments.
3. Draw a new frame.
4. In this new frame, bind the argument names to the values we computed.
5. Take the double bubble for the function we're evaluating, locate the frame the second bubble points to. Point an arrow from the newly created frame to the frame that the second bubble points to. This process is called *extending* the old frame. We say that our new frame *extends* the old one.
6. Evaluate the body of the function (pointed to by the first bubble) using this new frame.

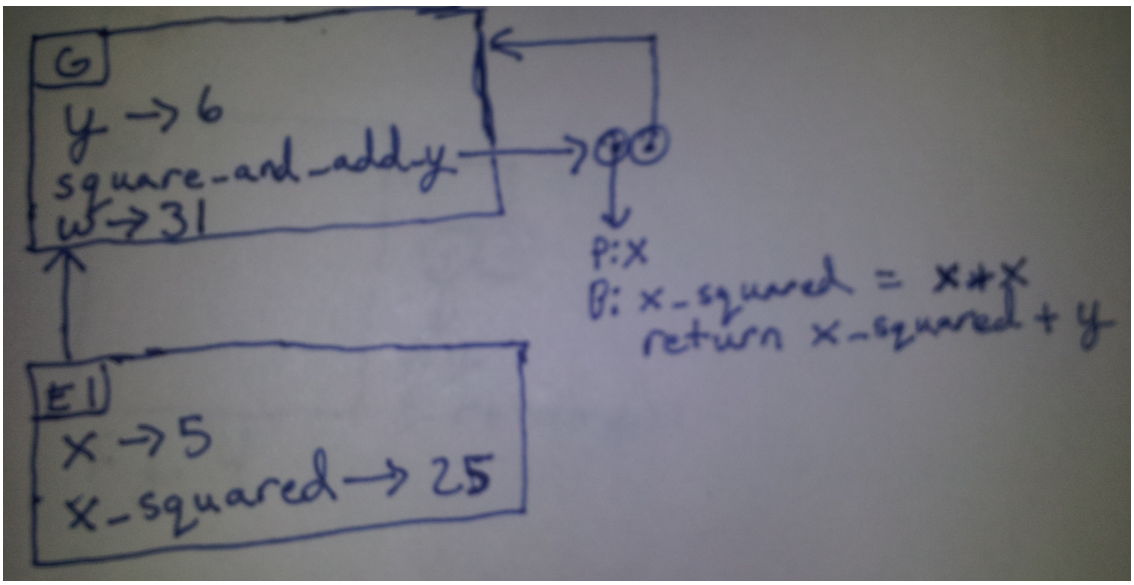
So before I start evaluating the function body, I end up with the following diagram:

[2] As well as the part that generally leads to errors if you're not careful!



Now, let's evaluate the body of the function! I do the same thing as I would have done in the global environment for the most part, I look up bindings for values, I make new bindings whenever I define a new variable name, and I evaluate each line one step at a time.

However, we have something new here, I'm using a variable outside the current frame! What do I do?!?! Well, this is where that arrow coming out of the frame comes into play, whenever we need to look up a name that is not in our current frame, we follow the arrow from our frame to the *parent frame*. Once we've finished evaluating the body of `square_and_add_y`, we return the value and we can finally make a binding for `w`. The final diagram is the following:



4.4 Recap: Where's the Manual?

Except for a way to represent some of the built-in data structures, we've seen basically all of the major pieces to environment diagrams.[3]

Let's review the steps to drawing environment diagrams that we've learned so far. In these instructions, we refer to the *current frame*, which is whatever context we are evaluating in at a given time.

- Looking Up Variables
  1. Start with the current frame.
  2. Look in the frame we have for a binding from the name to a value.
  3. If it's not in our current frame, repeat the search in the parent frame.
  4. Once we have a value, return to the current frame.
- Variable Assignment
  1. Evaluate the right hand side, continue drawing the environment diagram as you evaluate.
  2. Create a binding in the current frame from the variable name to the value of the right hand side.
- Function Creation
  1. Create a double bubble.
  2. Point the first bubble to a list of parameters and the body of the function.
  3. Point the second bubble to the current frame.
  4. If this was a `def` statement, create a binding from the function name to the double bubble in the current frame.
- Calling Functions
  1. Find the value of the function.
     - If this was a built-in function, treat it like "magic" (where the built-in function is defined in global and not shown).
     - Otherwise, locate the double bubble for the user defined function.
  2. Find the value of the arguments.
  3. Depending on whether the function was built-in or not
     - If this was a built-in function, treat it like a black box and find the resulting value returned.

[3] There's actually 1 more bit worth talking about, but we're tabling that discussion until the next lecture.

– Otherwise, if it is a user defined function, do the following:

(a) Draw a new frame.

(b) Create bindings in the new frame from the argument names to the values we found.

(c) Have the new frame extend the frame that the second bubble points to.

(d) Make this new frame the current frame (this is called "stepping into" the new frame) and evaluate the function body pointed to by the first bubble.

(e) Once the function body is evaluated, return with the value and restore the current frame to what it was before we stepped into the new frame.

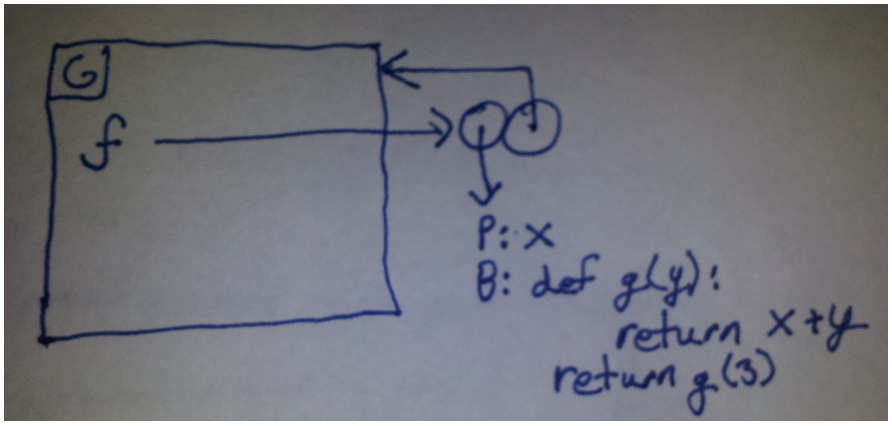
5 Take 'er Out for a Spin

5.1 Nested Functions

Let's do something a bit more complicated. Suppose I wanted to write the environment diagram for the following code:

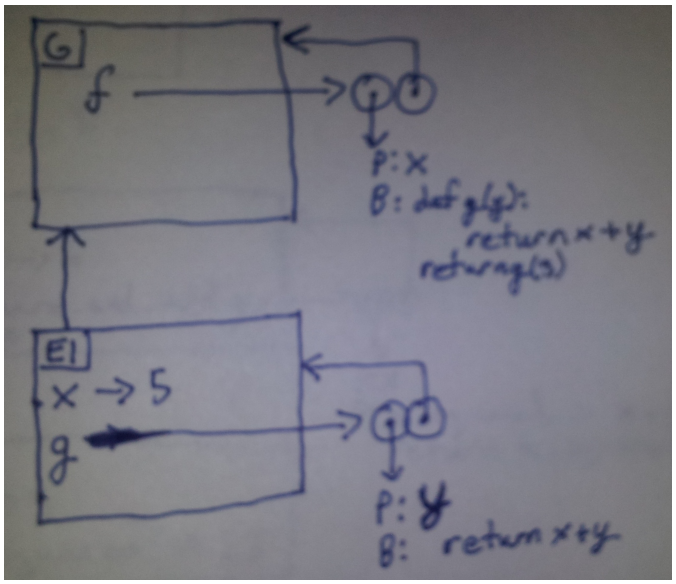
```
>>> def f(x):
...     def g(y):
...         return x + y
...     return g(3)
>>> f(5)
8
```

Now I'm defining a local function inside `f`. We actually know how to do this already, we just follow the same rules we've established. First we define `f`.

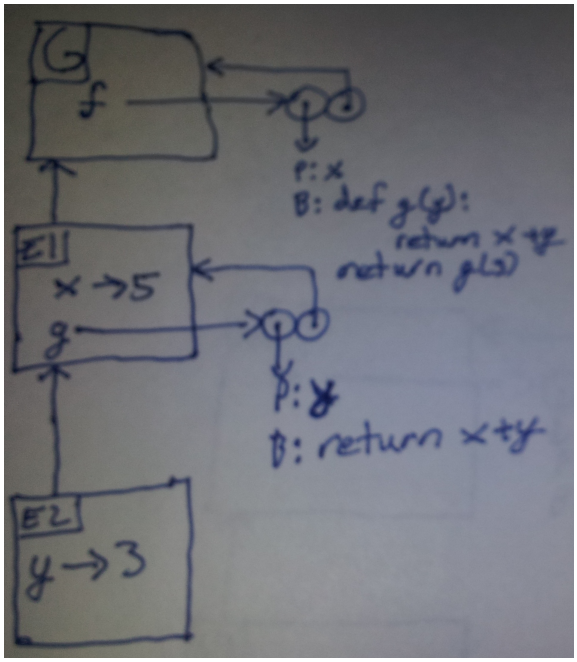


Notice that I didn't make a double bubble for `g`! This is because we don't evaluate the body of a function until we call it.

Next we call `f` and the first line of the body is to define the function `g`. After defining `g`, we end up with the following diagram:



Now here's the bit that is important to get right. We call `g` on the next line, which frame do we extend? Well, if you follow the rules, it says we extend the frame that we created for the current call to `f`, because that's where the second bubble of `g`'s double bubble points to. So we evaluate `g(5)` and we have the resulting final environment diagram:

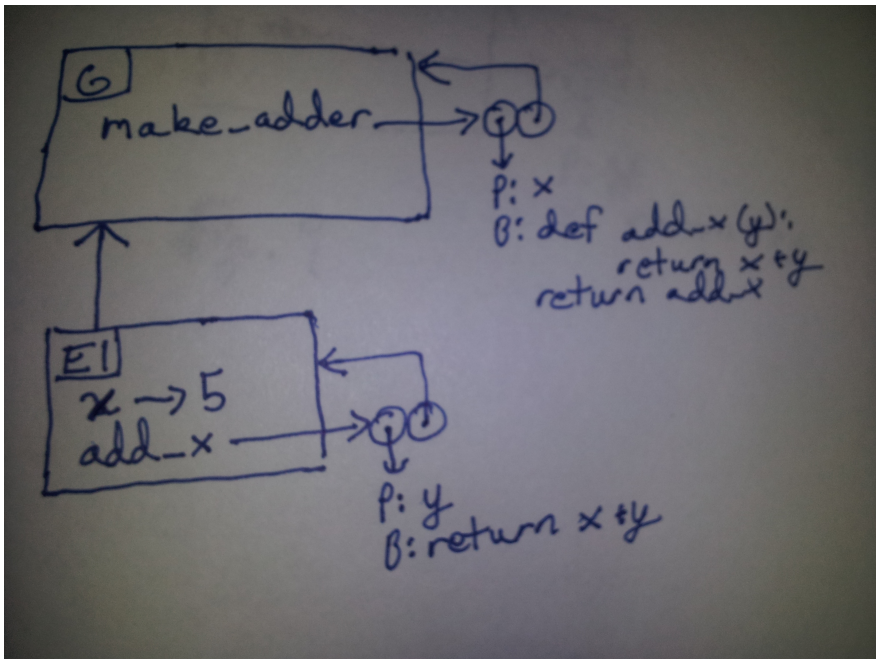


5.2 Higher Order Functions: Functions that Return Functions

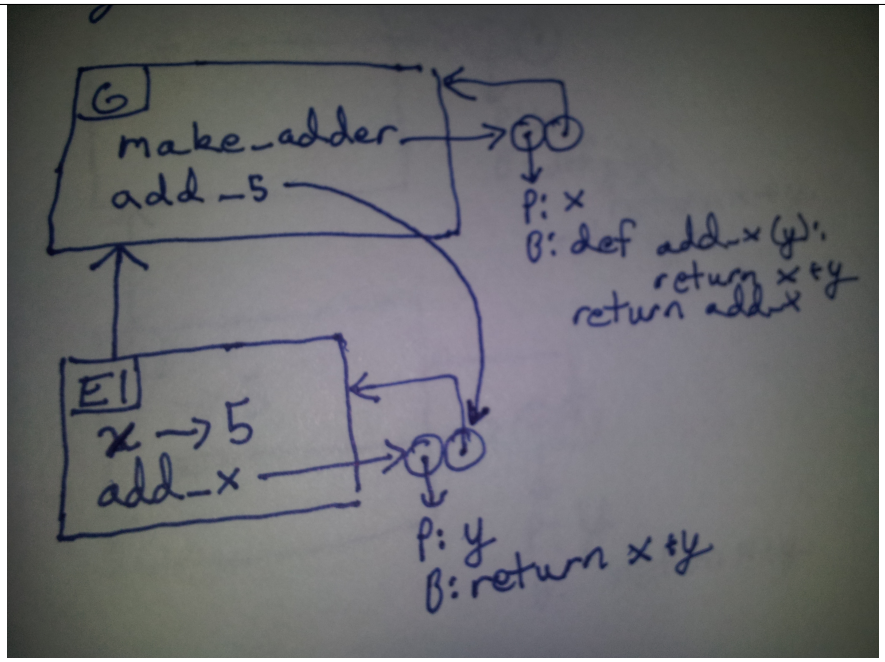
Okay, so now we've seen an example of nested functions. Using that, we can extrapolate the rules *even further* to draw an environment diagram for the following code:

```
>>> def make_adder(x):
...     def add_x(y):
...         return x + y
...     return add_x
>>> add_5 = make_adder(5)
>>> add_5(7)
12
```

So as in the last example, we'll first define `make_adder`, then we'll call it and start by defining `add_x`. Before we evaluate the return statement at the end of `make_adder` we will have the following diagram:

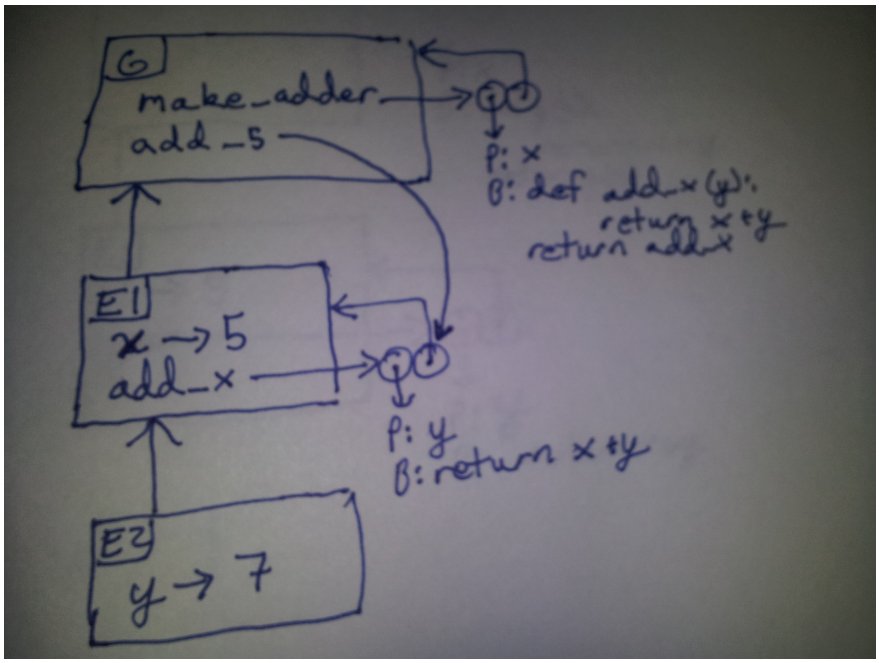


Now, it says to return the **value** of `add_x`. Well, according to our diagram, this is the double bubble that `add_x` points to, so we're returning that. So now we can complete the assignment statement for `add_5` which is to point it to the value being returned, the double bubble that was pointed to by `add_x`! We now have the following diagram:



Notice that we now have two names pointing to the same function double bubble. This isn't as weird as you might initially think. It's exactly what the code is doing, two names are bound to the same function value!

So now, we evaluate `add_5(7)`, which results in a new frame for the function call. Again, we have our new frame extend the frame that the second bubble points to. Once we've finished evaluating we end up with this final diagram:

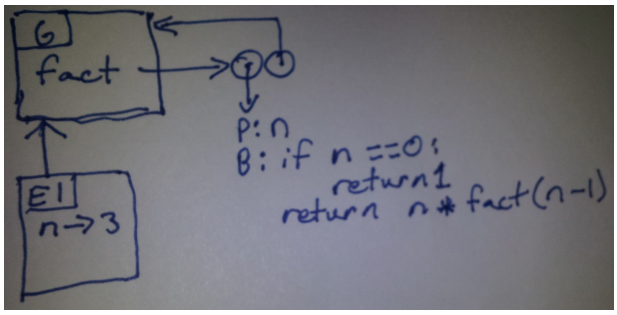


5.3 Recursion

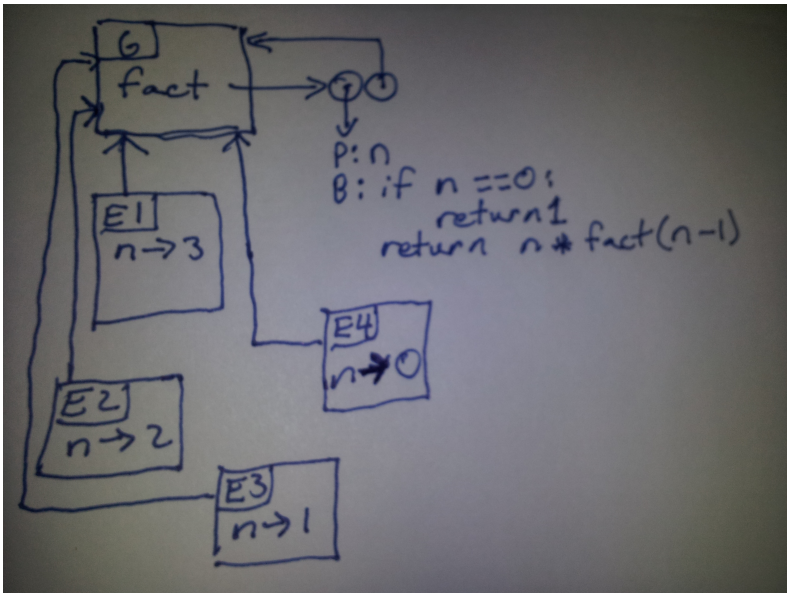
Let's examine an example with recursion, where we have more than one call to the same function in the diagram. Let's use the following code:

```
>>> def fact(n):
...     if n == 0:
...         return 1
...     return n * fact(n - 1)
>>> fact(3)
6
```

By this point, we know what happens up until the recursive call. Right before the recursive call we have the following diagram:



So how does a recursive call work? Really, we treat it as if we were calling some other function and make the frames for the recursive calls the same way as we did before. We look up the function, make a new frame that extends the frame pointed to by the second bubble, and evaluate. After we're done, we'll have:

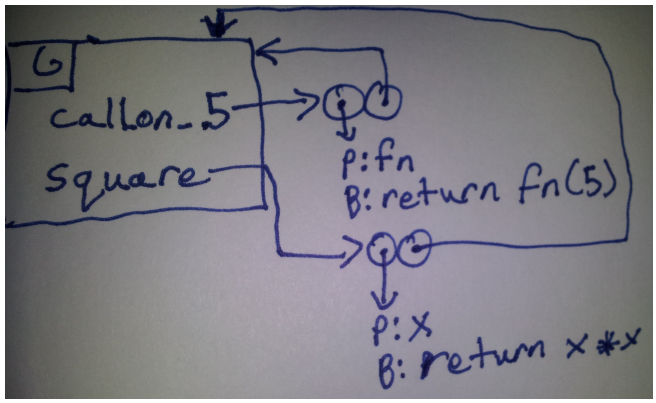


5.4 Higher Order Functions: Functions that take Functions as Arguments

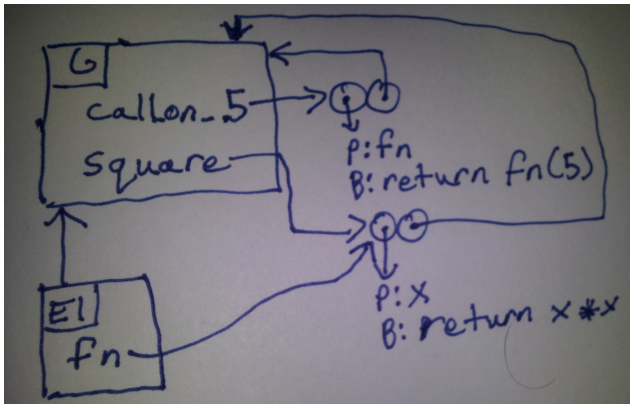
Another thing we did with higher order functions was that we took functions as arguments to other functions. Let's try that with our new model. Let's use the following example:

```
>>> def call_on_5(fn):
...     return fn(5)
>>> def square(x):
...     return x * x
>>> call_on_5(square)
25
```

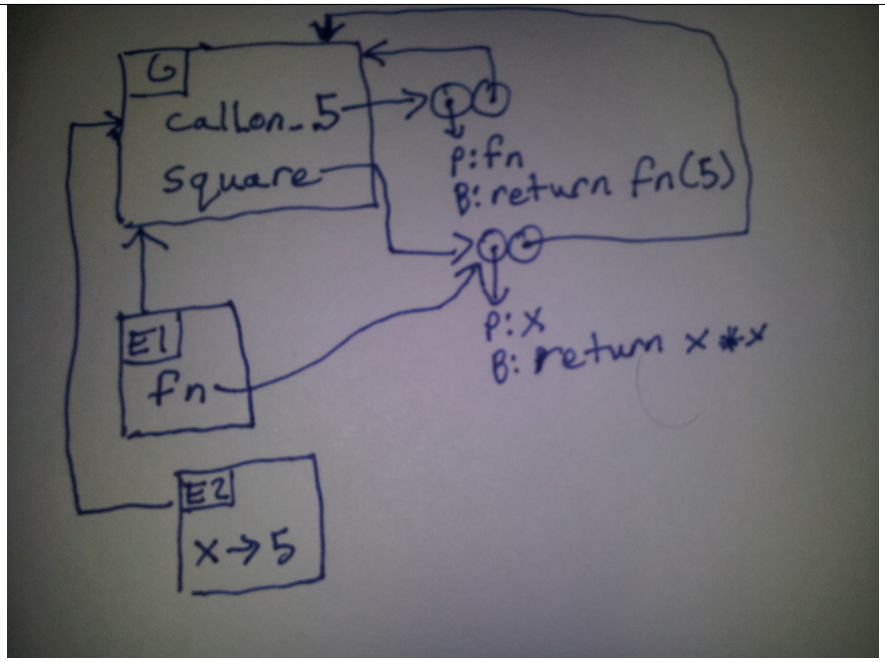
First we evaluate the definitions and we'd have the following environment diagram:



Now, we perform the call to `call_on_5`. What happens to the function being passed as an argument? Well the same thing as if it were any other piece of data, we point the argument to be bound to the same value, the double bubble. We'll end up with the following diagram as a result:



And now we can complete the diagram, again using the rules we've established as before and we end up with:



6 Conclusion

So what's the bottom line? Well, first, don't think this is over we still have a bit more to go. We still need to talk about built-in data structures and we have a new piece of python syntax that we'll want to use environment diagrams to understand.

What we've seen today is an *accurate* way to model state in our code, by diagramming the different contexts in which code runs and can see various values for names. Admittedly, it's much more involved than substitution, but substitution wasn't quite right and one of the major goals of this course is to develop tools for understanding how code behaves.

The important thing to remember is that environment diagrams are **not** a complete picture by any means. It simply helps us track the various variables and their values within our program at any given point. I know it's not the most satisfying thing in the world, but the important thing to understand is that it's not the job of an environment diagram to show everything, and you shouldn't ask it to.

Next Lecture: We'll talk about lists, tuples, and dictionaries. Furthermore, we'll introduce a new piece of Python syntax that allows us to update variables outside our current frame and see how that plays out in environment diagrams.