ENVIRONMENT MODEL: NONLOCAL STATE, DISPATCHING 19

COMPUTER SCIENCE 61A
Tom Magrino and Jon Kotker

July 19, 2012

1 Computer Science in the News

Title: Sky high success for Raspberry Pi computer

Source: http://www.bbc.com/news/technology-18900862

Raspberry Pi makes another appearance in the news! Dave Ackerman uses the chip to take pictures as it rose up to nearly 25 miles in the air on a balloon. The pictures taken were the highest altitude shots ever taken with an amateur device!

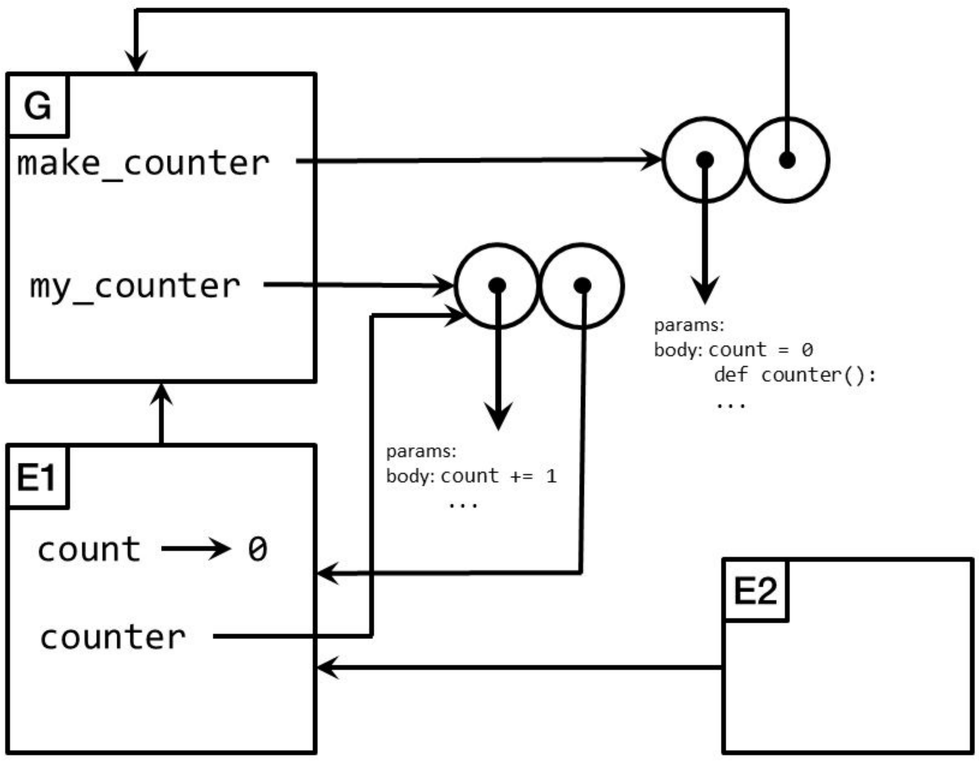
2 Modifying Nonlocal Variables

2.1 What Didn't Work

Suppose we wanted to make a function that kept track of the number of times it was called. What we would really like to do is to have this code work:

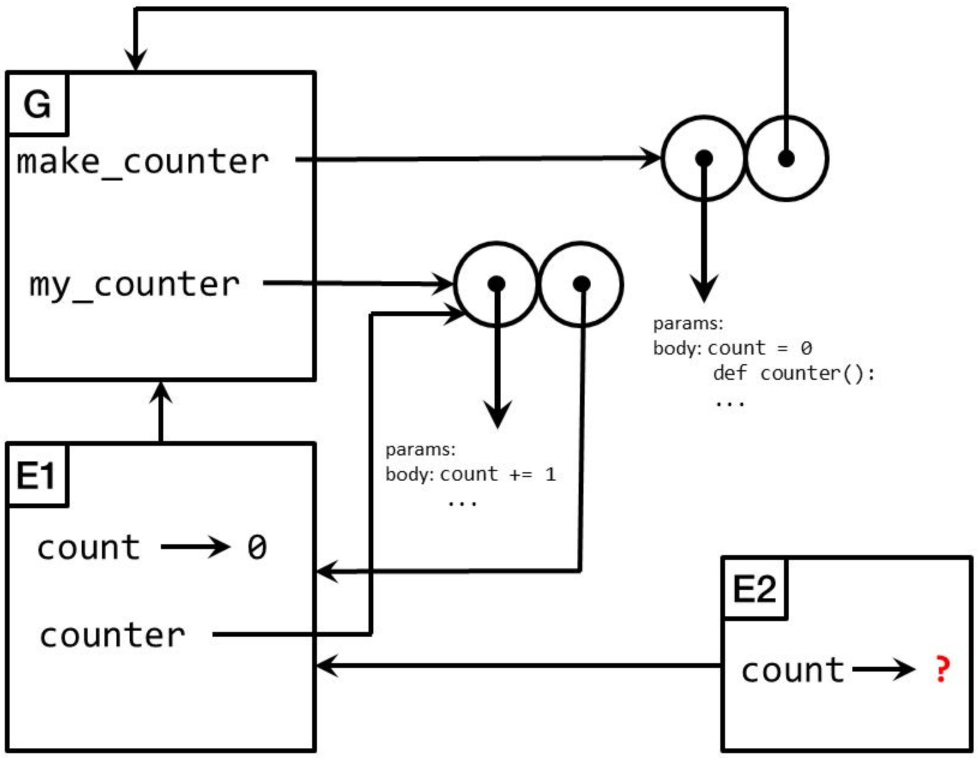
```
>>> def make_counter():
...     count = 0
...     def counter():
...         count += 1
...         return count
...     return counter
...
>>> my_counter = make_counter()
>>> my_counter()
1
>>> my_counter()
2
```

However, if you type this into the interpreter, you get the error `UnboundLocalError: local variable 'count' referenced before assignment`. Why did this happen? Well the problem is in the first line of the nested function counter, which says `count += 1` which translates to `count = count + 1`. Now, if we draw the environment diagram up until the error on that line we'll see this:



So what's Python crying about? The line says to take the value of `count`, add 1 to it, and store that in a variable named `count`. Sure, maybe it wouldn't do the "right" thing and it would make a new variable in the current frame E2 which binds count to 1, but why does it crash?

Well as it turns out, our environment model isn't *quite* what Python does under the hood and this is because Python's trying to stop you from accidentally writing code with hard to track bugs and be as efficient as it can while still being the great language we know. What do I mean by that? Well it turns out Python does this extra step before evaluating a function body where it goes through and determines all of the various local variables the function might end up defining and makes initial bindings to this mysterious "unbound" value before walking through the code. So we can imagine that Python sees an environment diagram that's more like the image below when it first steps into the E2 frame:



Now, I should make the following disclaimer: **DO NOT WRITE UNBOUND VALUES INTO YOUR FINAL ENVIRONMENT DIAGRAMS**, we are simply looking at what Python "actually does" for the moment. In most dynamic languages, this does not happen and we can simulate all of the effects of Python without having to include this step in the process of drawing these diagrams.

So Python does this step and it follows the rule that "it is an error if I (Python) ever try to use a variable that is still bound to the 'unbound' value." Thus, we can see why it decides to throw a fit over the line `count += 1` because in Python's eyes we just told it to use the forbidden "unbound" value.

Again, why does Python do this? Well, as you'll learn in later courses when you start paying attention to the physical memory your program uses, it turns out to be more efficient for Python to go ahead and try to plan for how much memory you'll need for the entire duration of your function call.

Furthermore, efficiency aside, programmers don't typically want to modify a variable in a different scope (akacontext/environment/frame) when assigning to a variable in *most* situations. What you typically want is to either update an already existing local value or create a new locally defined variable within your current environment.

So what gives? Sometimes we *would* like to modify variables in frames that are not the current frame we're working in. Seems like we're stuck here and Python starts to look like it has a real annoying shortcoming.

2.2 Using the Nonlocal Statement

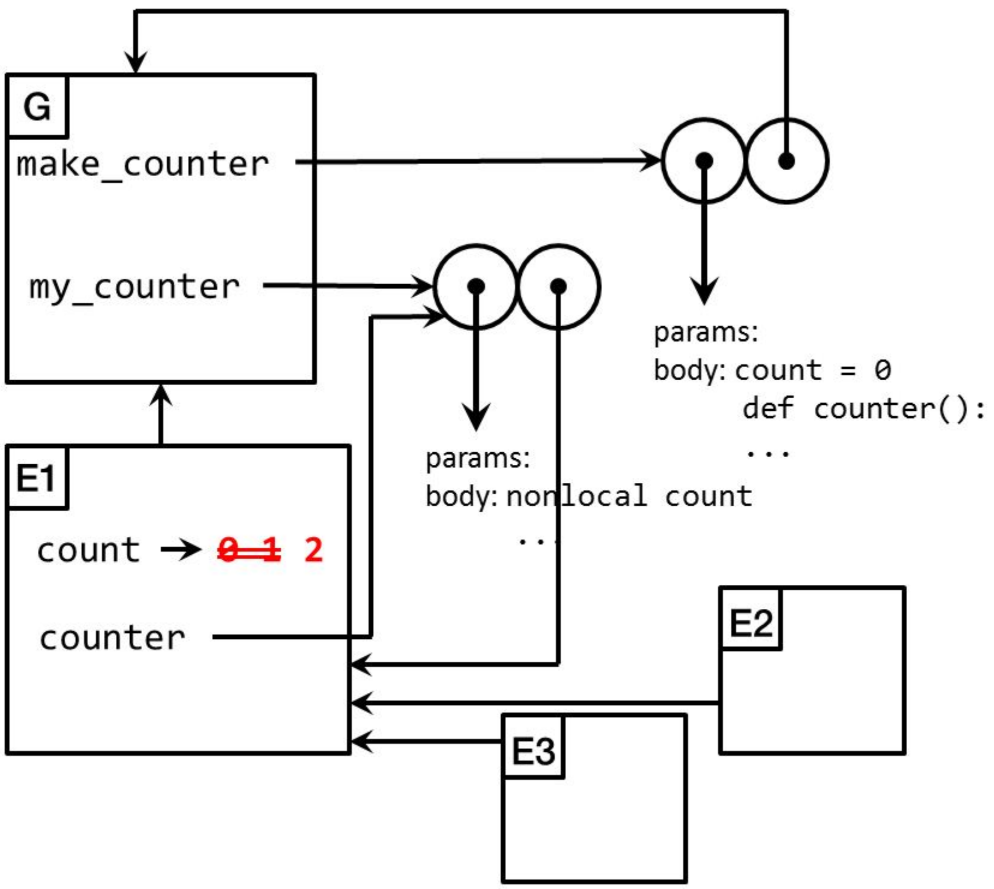
Well, as you hopefully saw in lab yesterday, there is actually a way to write `make_counter` so that it works the way we intended. The way we do this is by using a new kind of statement which tells Python, "Hey, Python, do me a favor, will you? When I mention these variables and assign to them, I actually want to modify the ones that were already present in the environment in other frames." This is the job of the `nonlocal` *statement*.

The easiest way to see how this works is to examine 1 or 2 examples of it in action. Let's first see how we could use it to get our original example to work:

```
>>> def make_counter():
...     count = 0
...     def counter():
...         nonlocal count
...         count += 1
...         return count
...     return counter
...
>>> my_counter = make_counter()
>>> my_counter()
1
>>> my_counter()
2
```

If we try this out in the interpreter, we see that I'm not pulling your leg, this stuff *actually* works!

So how does this work in environment diagrams? Well, as I said, the `nonlocal` statement tells Python that the variable(s) specified should not have new bindings in the current frame when we assign to them. Instead, Python should go search for a pre-existing binding in the environment (in one of the enclosing frames) and update that one instead. Cool, so that means we'll end up drawing this diagram using that rule:



As a result, we need to slightly modify the rules for assignment in our environment diagrams "manual." We've already taken the liberty of updating the rules, which should appear at the end of these notes.

2.3 Some Details

So, now it's time to be a bit more formal in talking about what exactly `nonlocal` does and does not do. Here's the syntax:

```
nonlocal x[, y, ...]
```

So we can specify a list of variable names, separating them with commas in a line. Traditionally, we do this in one line (or a few consecutive lines, if we would have had a really long line) at the top of the function.

Now for the most part, this works as advertised, it tells Python to not make a local binding for these names and instead update some other binding in the environment in another frame. There is, however, one exception to this rule. `nonlocal` does **not** work for variables defined in the global scope.

Why? It's actually not as interesting as you would think. Turns out that Python didn't have `nonlocal` before version 3. Instead we only had the close cousin of `nonlocal`, the `global` statement, which allowed us to say "Python, when I assign to this variable, update the binding in the global frame." The syntax is the exact same, replace the word `nonlocal` with `global`. This existed in older versions of the language and, for backwards compatibility, this still exists in Python 3 as the way to update global bindings. I should note however, it is typically considered **VERY** poor style to use global variables like this in your program, which is why we're not focusing on it too much.

Why doesn't `nonlocal` just do what it already does *and* the job of `global`? As far as I can tell, it might be because it was easier to implement like that and we already had a way to update global bindings. Otherwise, I'm not quite sure, I didn't design the language.

So, in short, `nonlocal` will let us update any binding that is not in the current frame or the global frame. If we want to use the local frame, we just write code as we did before. If we want to update a binding in the global frame, we use `global` instead.

2.4 Another Example

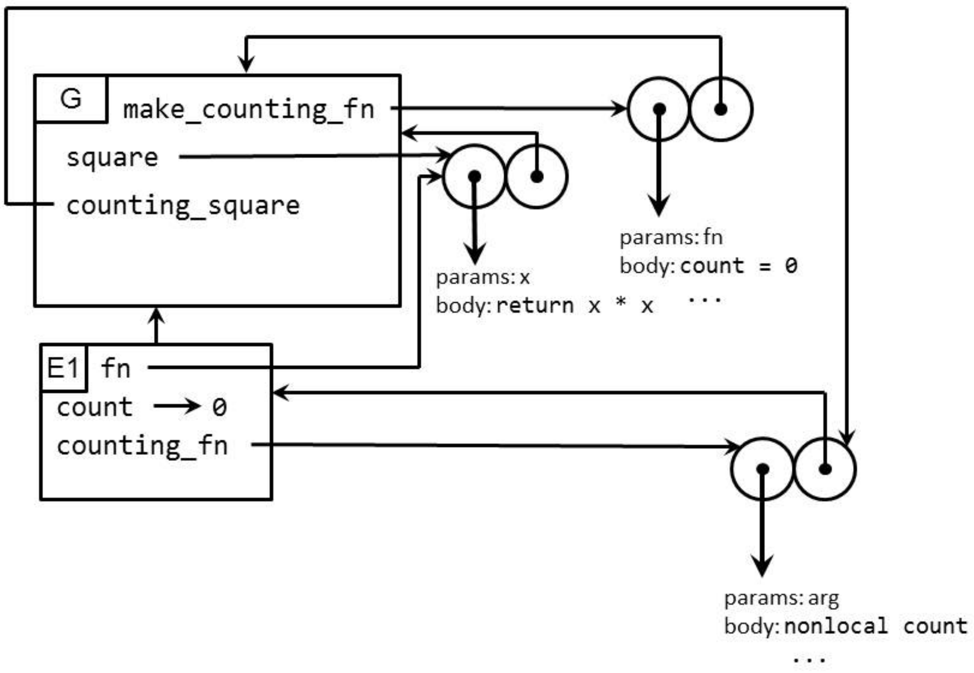
Note: I might skip this example in class if I think we're going to be running short on time.

Let's see how we can use nonlocal to do something even cooler than making counters. Let's write a higher order function which takes another function and produces a new function that, before evaluating the original function, prints the number of times you've called the new function. Here's the code:

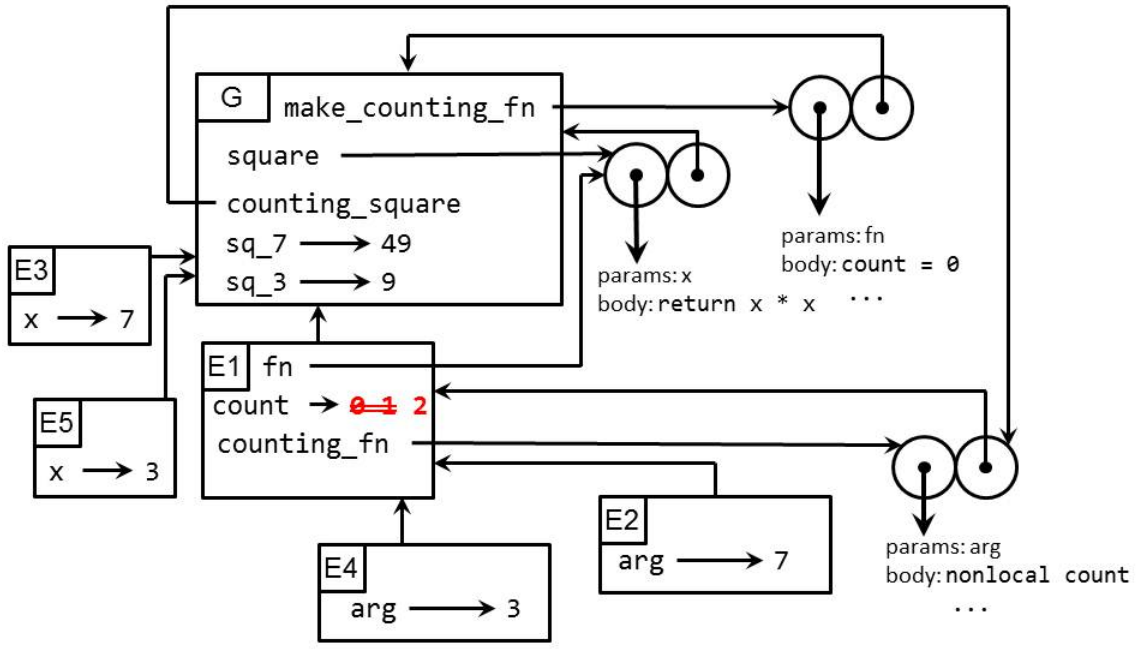
```
>>> def make_counting_fn(fn):
...     count = 0
...     def counting_fn(arg):
...         nonlocal count
...         count += 1
...         print(count)
...         return fn(arg)
...     return counting_fn
...
>>> def square(x):
...     return x * x
...
>>> counting_square = make_counting_fn(square)
>>> sq_7 = counting_square(7)
1
>>> sq_7
49
>>> sq_3 = counting_square(3)
```

```
2
>>> sq_3
9
```

If we draw the environment diagram for the code up to and include the part where we define `counting_square` we have:



So we see that `counting_square` is a function which will have access to the count variable, because it will extend E1, the frame which holds the `count` variable. Since we mark `count` as `nonlocal` in the body of `counting_square`, we can see that it is now able to update and use the `count` each time. After the two calls to `counting_square`, we will have the following diagram:



3 Announcements

- HW9 is due Tomorrow.
- HW10 will be out tonight and due Tuesday.
- Project 3 is due **Thursday**, July 26th.
- Midterm 2 review session **TOMORROW** 6pm - 9pm in 120 Latimer.
- 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 are posted.

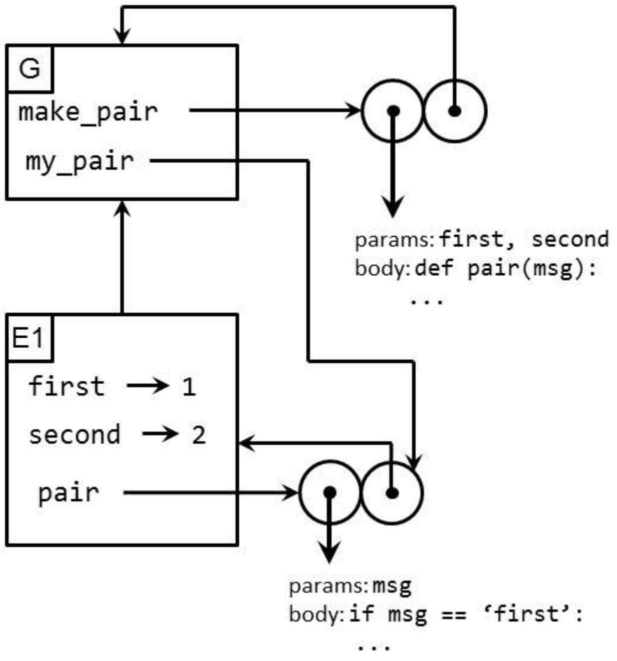
- Regrades are due by July 26th.
- Attach an explanation to the front of your exam explaining what you think should be regraded and why.
- We reserve the right to regrade the entire exam.

4 Dispatching: Passing Messages to Make Data Structures

If you recall, in an “extras” slide in one of the previous lectures, we mentioned that we could have actually implement 2-tuples and other data structures using just functions. The code we wrote was something like this:

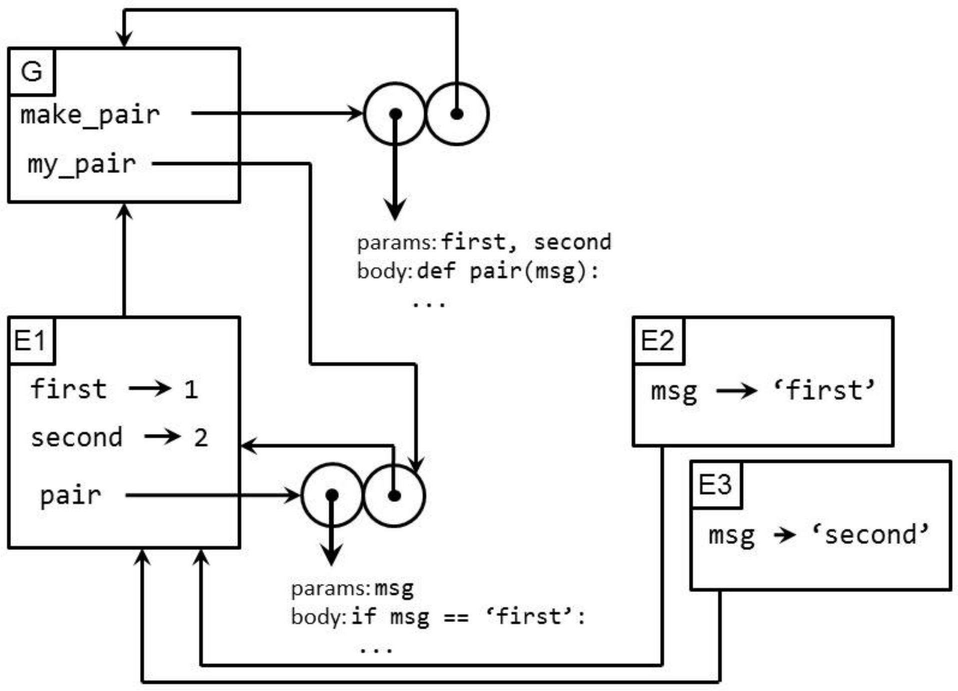
```
>>> def make_pair(first, second):
...     def pair(msg):
...         if msg == 'first':
...             return first
...         elif msg == 'second':
...             return second
...         else:
...             return "ERROR! Not sure what you meant."
...     return pair
...
>>> my_pair = make_pair(1, 2)
>>> my_pair('first')
1
>>> my_pair('second')
2
```

This is an example of what we call “dispatching” where we make data structures operate by passing them messages indicating what we’d like to get from them. How does this work? Well let’s draw the environment diagram for everything up to and including making the variable `my_pair`.



So, we see that E1 holds the data we'd like to get back from our pair and that our pair happens to be a function that, when called, will extend the frame E1 and will have access to the information we want. So, in a sense, the `pair` function we made and returned is our "backdoor" to get the information we left laying around in the frame E1, which existed when we called the constructor.

So now we can see, using environment diagrams, why the selectors work! If we continue drawing the environment diagram for the rest of the code, we get:



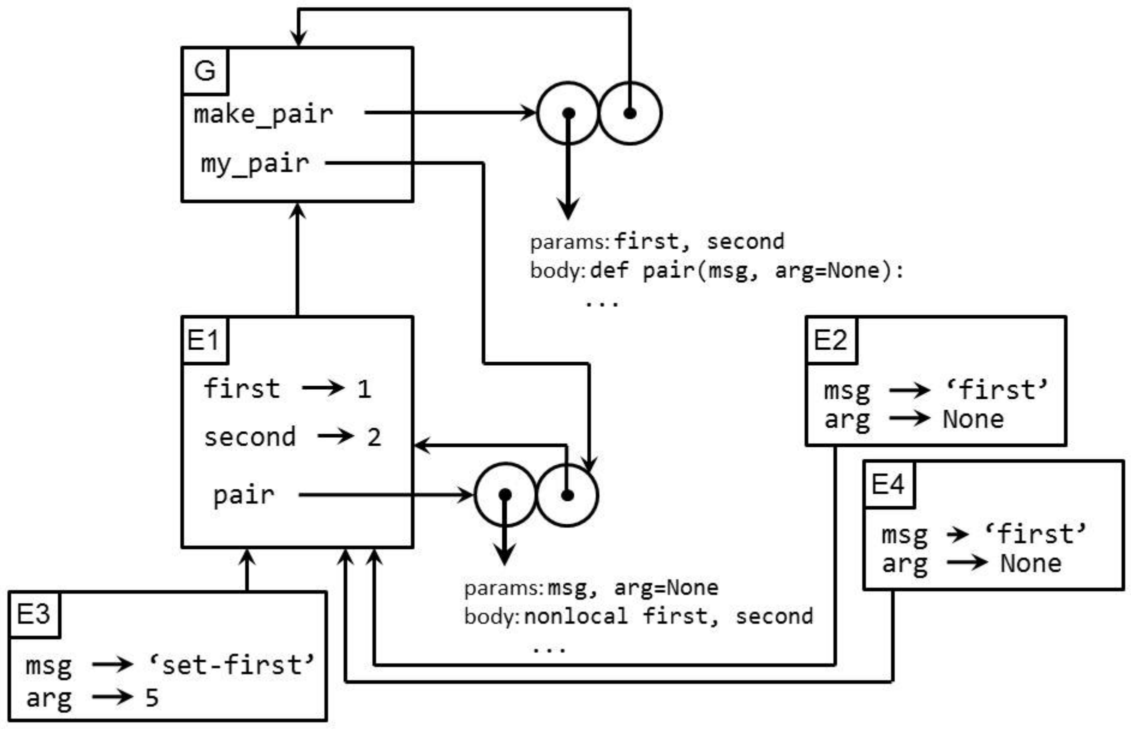
E2 was the frame created for the call `my_pair('first')` and E3 was the frame created for the call to `my_pair('second')`. These frames had access to E1 where the data is stored and this is how we're able to retrieve the data stored in our pair.

So, in reality, we never *needed* a single data structure to be built-in in our programming language if we can just use higher order functions. We know that we can use pairs to implement recursive lists, which provide the same functionality as tuples. Furthermore, we saw that we could implement immutable dictionaries with any sequence data structure (we used tuples, but we could have just as easily have used recursive lists).

"Wait!" you say, "We still couldn't implement mutable lists using tuples, so how do we get *mutable* data structures?" I'm glad you asked! This is where `nonlocal` can be really handy for us. Let's implement a *mutable* pair:

```
>>> def make_pair(first, second):
...     def pair(msg, arg=None):
...         nonlocal first, second
...         if msg == 'first':
...             return first
...         elif msg == 'second':
...             return second
...         elif msg == 'set-first':
...             first = arg
...         elif msg == 'set-second':
...             second = arg
...         else:
...             return "ERROR! Not sure what you meant."
...     return pair
...
>>> my_pair = make_pair(1, 2)
>>> my_pair('first')
1
>>> my_pair('set-first', 5)
>>> my_pair('first')
5
```

Again, we can draw the environment diagram and see what's going on:



And we see that in the call `my_pair('set-first', 5)` that we again used the fact that `my_pair` is a function which will have access to `first` and, because we marked `first` and `second` as `nonlocal`, we can now update them through a call to `my_pair`.

So, again, we see that we didn't need Python's built in (mutable) data structures, with higher order functions and `nonlocal` we already have all we need to create the same functionality!

5 Conclusion

Today we learned about:

- `nonlocal`, a new piece of Python syntax that let's us update values that are not defined in the current frame.
- Dispatching, which is the idea of making data structures where we pass them messages to retrieve data stored in the structure.
- We saw that with `nonlocal`, we really could have implemented any data structure that Python provides us.

Environment Model Rules

- Looking Up Variables
  1. Start with the current frame.
  2. Look in the frame for a binding from the variable name to a value.
  3. If it is not in the current frame, repeat the search in the parent frame.
  4. Once a value is found, return to the current frame.
- Variable Assignment
  1. Evaluate the right hand side, and 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, only if there is no binding for the variable in the current frame. Otherwise, update the binding in the current frame.

     Special case: If the variable is `nonlocal`, update the binding in a parent frame, if there is a binding in a parent frame that is not the global frame. Otherwise, make a binding in the current frame.

     Special case: If the variable is `global`, update the binding in the global frame, if there is a binding in the global frame. Otherwise, it is an error.
- 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. – If this was a built-in function, treat it like a black box and find the resulting value returned.
     - Otherwise, if it is a user defined function:

(a) Draw a new frame.

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

(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.