Environment Diagrams

Abhinav Ashar
CS 61A: Structure and Interpretation of Computer Programs

September 15, 2019

1 Environment Diagrams

Environment diagrams give a structured and clear insight into the state of a program at any point in time. It has two main parts: frames (left side) and objects (right side). The frames indicate the current scope of variables you are working with, while the objects denote the objects and their parent frames. Variables in the frames can either be primitive (stores simple values like numbers) or be pointers to objects. In the next section, I'll outline the steps I use to tackle every environment diagram problem.

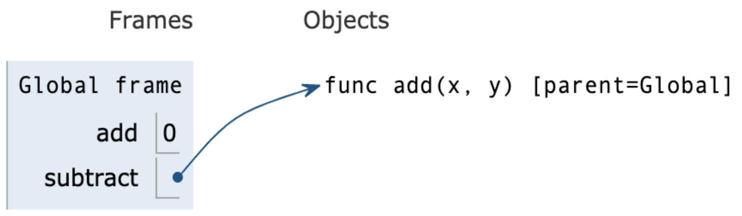
2 How to Solve an Environment Diagram Problem

1. **Assign values or pointers to function names**
   Starting from the top of the global frame, go line by line and assign variables to either values or function names. Remember that you do not actually read the code inside of the function just yet. You only establish that this function exists, but you have no idea what is actually inside of it at this point in time. For primitive numbers, you can write them in the box in the frame next to the variable name. If it is an object or function, write the following: "func *nameOfFunction* [p = *parentFrame*]". If it is a lambda, then do the same thing as for a function but just write this instead: "func $\lambda_{lineNumber}$ [p = *parentFrame*]". Then, draw a pointer from the variable in frame section to this function in the object section.

2. **When you get to a function call, do the following:**

   a) **Evaluate the operator**
   The operator is another word for the function. Look at the current frame that you are in. See if you can find the function name that is used to make the call. If not, look at one of the parent frames. When you find the variable, look at

which function object it points to on the right side. This is the **real** name for the function, not the variable name. Look at the following example:



In this environment diagram, while the variable name is *subtract*, the real name of the function is *add*. This is a common way that the test can try to trick you: using counter-intuitive or confusing names. Instead, simply take note of the function name from the **right-hand side**.

b) **Evaluate the operands**
The operands are the values inside of the parentheses. If the value is a number, there is nothing much needed to be done to evaluate it. If it is a lambda function, create the function on the right-hand side and make its parent as the frame you are currently in (not the one you are about to go to). If it is a function call, repeat step 2 again except now with this new function and arguments. When you fully complete all the steps for this inner function, you'll know what that inner function returns, so just keep that in mind.

```
1  def a(x):
2      return x + x
3  def b(x):
4      return x * x
5  b(1 + a(3))
```

In this example, we are evaluating the operand in line 5. However, the operand itself has another function call, so we open up another frame, follow the same steps using the variable *a* as our function, and after executing all the steps, we will get a return value of 6 for this example. That 6 is added to the 1, meaning that our operand evaluates to 7 at the end of the day. Just note that down for the next step.

c) **Apply the operator to the operands, create a new frame, and bind the values**
Now we apply the evaluated operator to our fully evaluated operands. First, create a new frame on the left-hand side. The name of the frame should be the **real name** of the function call that we got from 2a. Also note down the parent frame, which you can get from the objects section on the right-hand side. Next, look at all the parameters to the function. Create one box in the frame for each parameter. For example, if the function was *def a(x, y, z)*, then create a box for *x*, *y*, and *z*. Lastly, bind the values by taking the values you determined from 2b and

assigning them to these new variables. Like mentioned in step 1, add primitive values to the boxes or add pointers to objects/functions/lambdas.

d) **Optional: Make a mark on your page denoting where you are supposed to be after this function call returns a value**
It can be hard to keep track of where you last were, so I recommend just making a small mark denoting where to return.

e) **Repeat the previous steps for this new frame**
In this new frame, you'll likely have to define more variables, functions, and lambdas, and you may do even more function calls. Repeat the same instructions mentioned before. At the end, determine the return value and write that in the *RV* box in the frame. If the return value is a primitive, just write that value in the *RV* box and go back to the last location from part 2d. If the return value is an object/function/lambda, return a pointer to it. This basically means draw a pointer from *RV* to the respective object on the right-hand side. If the place you are supposed to return to in 2d has some variable equal to the result of the function call, such as "x = a(3)", then write the number for that variable in the variable's box (if primitive) or draw the pointer to the same object that the *RV* pointed to (if not primitive) in the variable's box.

3. **When referencing variables, look in the current frame and its parent frames**. When a variable is used, see if you can find it in the current frame. If not, look for it in one of the parent frames. Keep going up the parent frames until you find it. Keep in mind that the parent frame is not the frame phsyically above the current frame on your paper; instead, it is the one that you wrote down in the frame description at the top of the frame ([p = ...]).

3 Miscellaneous

- **Lexical Scoping**
Remember that in Python, a function's parent is determined by where the function was **defined**, not where it was **called**.

```
y = 5
def a(x):
    return x + y
def b(x):
    return c(x)
def c(x):
    y = 2
    return a(x)
b(3)
```

In this example, notice that *a* was defined in the global frame but was only called in frame 2. However, since it was defined in the global frame, its parent is the global frame. Thus, when you are looking for *y* in line 3, you look in the global frame for *y*, not in frame 2.

- **Return Values**
  Every function call has a return value. On one hand, there will be an explicit return statement that tells you what is returned. On the other hand, if there is no return statement, the return value is *None*.

- **Closing Frames**
  When you return a value, that specific frame is closed. That means that the content inside of the frame (the boxes on the left-hand side) will never be modified in the future. When you call the same function again, it will simply open a new, different frame rather than using this old one. However, the content inside of the frame may still be referenced and the values may be used later. It just won't be modified on the left-hand side.

- **Variable Assignment**
  Some students get confused when a variable name is assigned to a function but then is reassigned again to something else, like a primitive value.

```
def a(x):
    return x * x
a = 5
```

  This isn't an error! All you do is replace what was previously in that variable's box in that frame and replace it with this new value. So in this example, *a*'s box had a pointer to a function on the right-hand side. Once you reach line 3, cross out or erase the pointer and put a 5 in the box instead.