DRAWING ENVIRONMENT DIAGRAMS

COMPUTER SCIENCE 61A

September 10, 2012

0.1 Background

- A frame is a location where variable bindings are stored
- A binding is a connection between a name and a value. The name is the name of the variable, and the value can be an integer, a function, or any other data type.
- It is important to keep track of which frame is the *current frame*. The *current frame* is the frame in which code is being executed currently.
- The *frame number* is a unique identifier we give to frames if necessary. Note that we only add frame numbers to frames when it is necessary to refer back to them. Therefore, many frames will not end up with frame numbers.

0.2 Starting Your Diagram

- First draw the global frame. Designate that it is the global frame by writing "global frame" in the upper-left corner. Note that the *current frame* is now this global frame.

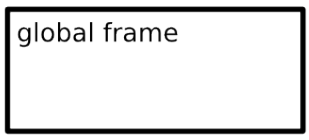

Figure 1: The global frame

- Step thorough the code evaluating one line at a time.

0.3 When you Encounter an Assignment Statement:

- Evaluate the code to the right of the "=" while continuing to follow the procedure for environment diagrams (making new frames as necessary, etc.)
- Bind the name on the left of the "=" to the result of evaluating the code to the right of the "=".
  - Write the name to the left of the equal sign in the *current frame*.
  - If the value of the right hand side is a number, string, or boolean value, write this value next to the variable name.

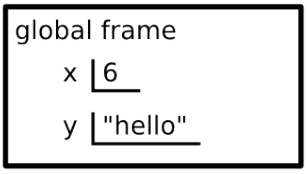

Figure 2: The result of evaluating `x, y = 6, "hello"` in the global frame

  - If the value of the right hand side is something else (for example, a function), then draw an arrow from the variable name to the value.

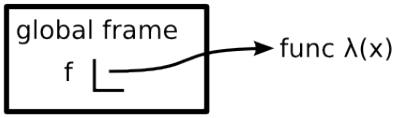

Figure 3: The result of evaluating `f = lambda x: x*x` in the global frame

0.4 When you Encounter an Import Statement:

- Bind all the imported values within the current frame.

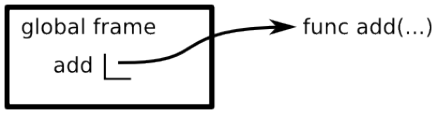

Figure 4: The result of evaluating "`from operator import add`" in the global frame.

- Note that the argument list for built in functions is usually represented as "`...`" even if the function only accepts a finite number of arguments.

0.5 When you Encounter a Function Call for a User-Defined Function:

- Evaluate the operator
- Evaluate the operands
- Create a new frame
  - Label the upper left corner of the frame with the name of the function we are calling. Leave space to the left of this label because we may need to add a *frame number* here later.
  - If the function that we're calling has a parent (i.e. the function has `[parent=f<frame number>]` written after it in its environment diagram representation), then write `[parent=f<frame number>]` in the upper-right corner of our new frame, where `<frame number>` is the frame number of the parent of the function we're calling.

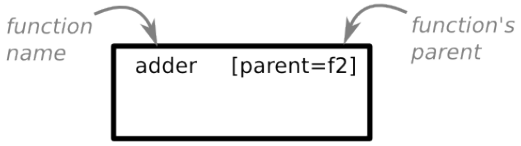

Figure 5: The frame resulting from calling a function whose name is `adder` parent is `f2` i.e. a function whose representation in the environment diagram is something like `func adder(y) [parent=f2]`. Note that in this diagram, the parameters are not yet bound

  - For each of the function's arguments, add a binding in our new frame from the argument name to the value that the operand evaluated to.
  - The *current frame* is now the frame we just created.
  - Evaluate the body of the function, stepping through the code as you had been before, but with the current frame now the new one you just created.
  - When you reach the return statement, bind "return value" to the result of evaluating the right side of the return expression.
  - Now, return from the function and take note that the *current frame* is now what was previously the *current frame* in the stack.

0.6 When you Encounter a Define Statement:

- Create the new function by writing the signature of the new function to the right.
  - This signature is of the form:
    `func <intrinsic_name>(<args1>, ... <argn>)`

where `<intrinsic_name>` is the name of the function we give it in the define statement and `<arg1> ... <argn>` are the names of the arguments this function takes

- If the *current frame* is not the global frame:
  - If the *current frame* does not yet have a *frame number*, then select a new one that has not yet been used and write "`f<frame number>:`" in the space we left in the upper-left corner of the frame when we created it.
  - Now, add "`[parent=f<frame number>]`" to the signature representing our new function on the right. This indicates that the function's parent is the frame in which the function is defined (the current frame)

- Bind the name of the function to in the *current frame* to the function signature on the right by drawing an arrow.

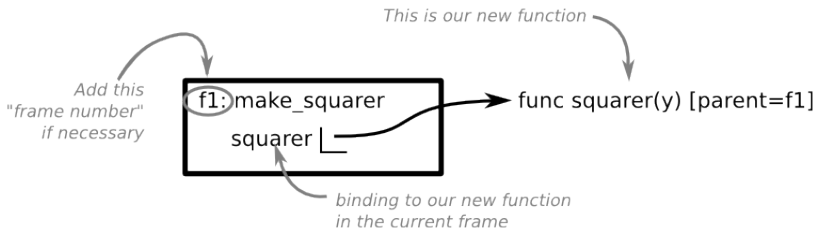

Figure 6: The result of evaluating a function definition for a function named squarer from within a frame named `make_squarer`.

0.7 When you encounter a lambda expression:

- Create the new function by writing the signature of the new function to the right.
  - This signature is of the same form as it is for a define statement, except the `<intrinsic_name>` is replaced with λ, because lambda functions don't have intrinsic names.
- Note that evaluating the lambda function itself does not involve binding any names.

0.8 Looking up a name:

- Start in the current frame.
- If the name you are looking for is not in the current frame, then continue looking in the current frame's parent. Note that if no parent is listed for a frame, then it is implied that the parent is the global frame.
- Continue this process until you have arrived at the global frame. If you still can't find the name you are looking for then a `NameError` exception is thrown.

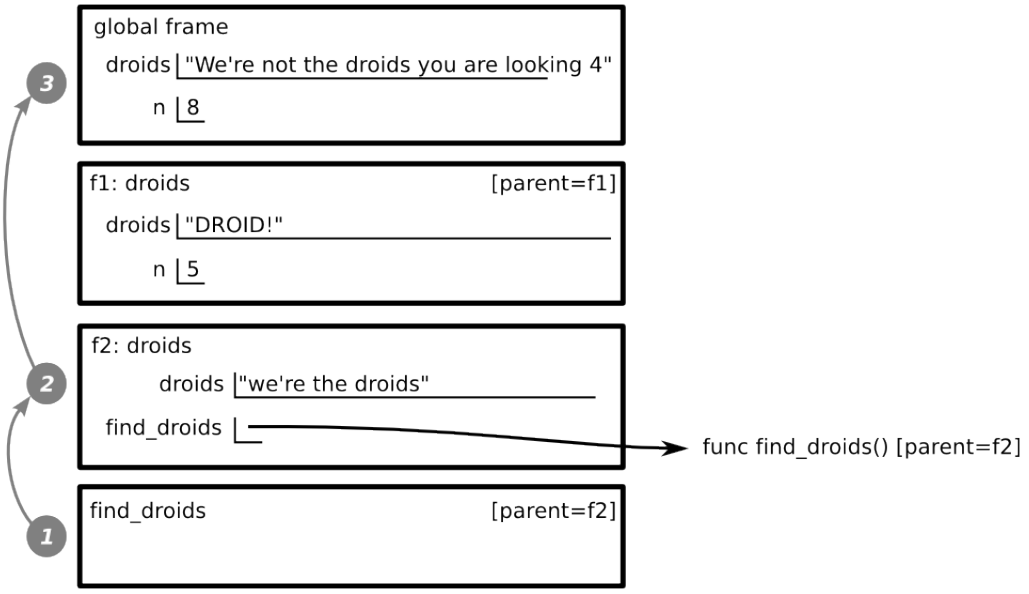

Figure 7: Note that in this diagram the result of looking up `droids` in the `find_droids` frame is `we're the droids`, and the result of looking up `n` in the same frame is `8`. The lookup order passes from parent to child moving from the bottom frame to f2 to the global frame.

0.9 Dos and Don'ts

- Never draw an arrow from one variable name to another.
- It is not necessary to draw environments for built in functions such as `sum(...)` or **`print`**`(...)`.
- Remember to follow the sequence of frames from parent to child—not simply from bottom to top.
- Remember to write down the parent of every function or frame whenever this parent is not the global frame.