CS 61A Discussion 2 Environment Diagrams / Recursion

1.

Quiz II

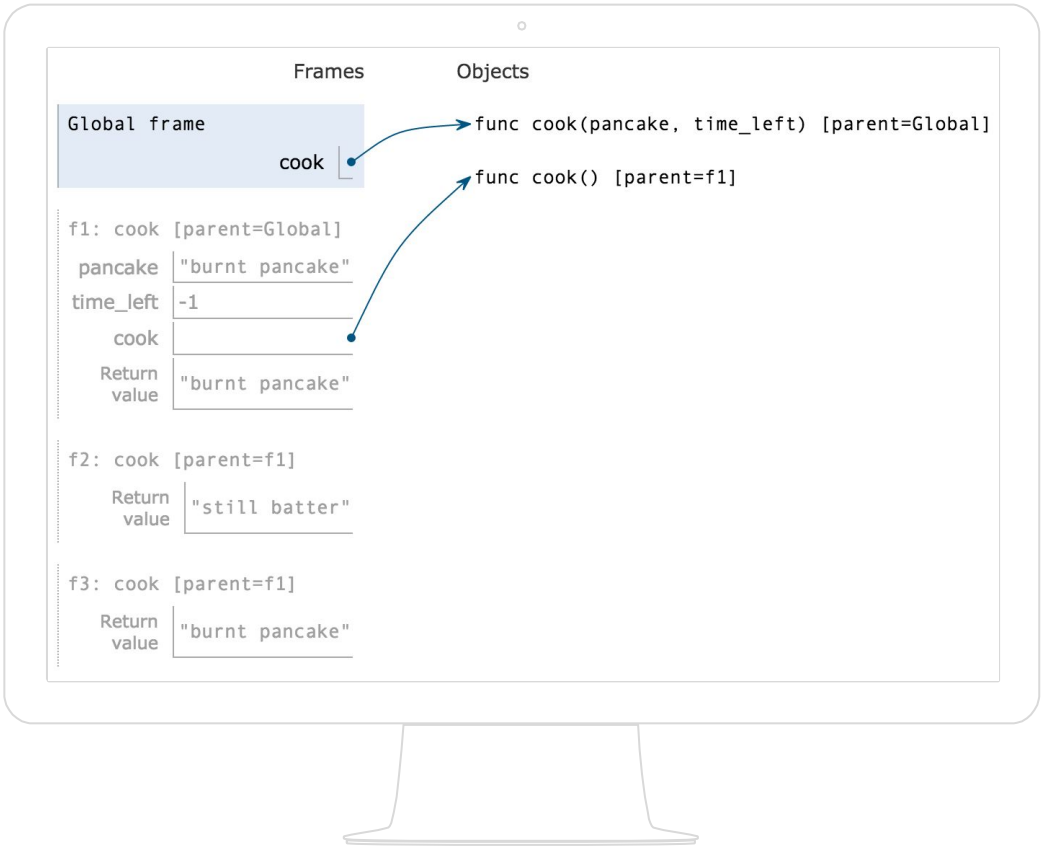
Let's see how much you already know about environment diagrams…!

Environment Diagrams

should be free points on the exams. You don't have to be creative or clever. You just have to know and follow the rules… albeit mechanically and without error. Still, there *aren't that many rules*! (Hint: they're all in the textbook!)

The best way to ensure your free points is by brutal, repetitive practice.

So let’s make a deal...



Technically speaking,

an environment diagram is just a record of all name/value bindings.

Drawing a diagram

1. MAKE A GLOBAL FRAME

This is the frame from which all of the other frames are derived.

2. GO THROUGH THE PROGRAM, UPDATING BINDINGS AS NECESSARY

And that's it. Just update them *right*.

ASSIGNMENT STATEMENTS `[x = 5]`

Evaluate the expression(s) on the right-hand side of the = sign. Then bind the value(s) on the right to the name(s) on the left, *in the current frame*.

FUNCTION DEFINITIONS `[def foo(...)]`

Create a new function object, written as `func <fn name>(<params>) [p=<parent>]`. Bind it to the name `<fn name>` in the current frame. The parent is just the current frame.

FUNCTION CALLS `[foo(...)]`

[Step 1: evaluate the operator (the function), then evaluate the operands (the arguments) in order.] Create a new frame. Always! (And *only* when there's a function call, incidentally)! Title the frame `f<curr #>: <intrinsic fn name> [p=<parent>]`. The intrinsic name and the parent **should be obtained from the appropriate function object on the right**. Bind the arguments to the formal parameter names in the newly created frame. Finally, run through the body (also in the new frame). When the function returns, go back to whichever frame you were in before.

LOOKUP `[somevar]`

Check the current frame for the name you're looking for. If it's there, use its value. If not, keep following the chain of parents and doing the same thing. If you get to the global frame and the name still isn't there, it's an error.

Common Confusions

- Arguments are evaluated in the *calling* frame
- Everything on the right of an assignment statement is evaluated before ANYTHING on the left
- When titling a new frame, use the *intrinsic name* of the function (just copy the name and the parent from the right section of the diagram)
- If you have `x = var`, copy the value of `var` and put it in `x`'s box. If the value of `var` is an arrow pointing to a function, then `x`'s value should be an arrow pointing to the same function.

2.

Recursion

Defining functions in terms of themselves

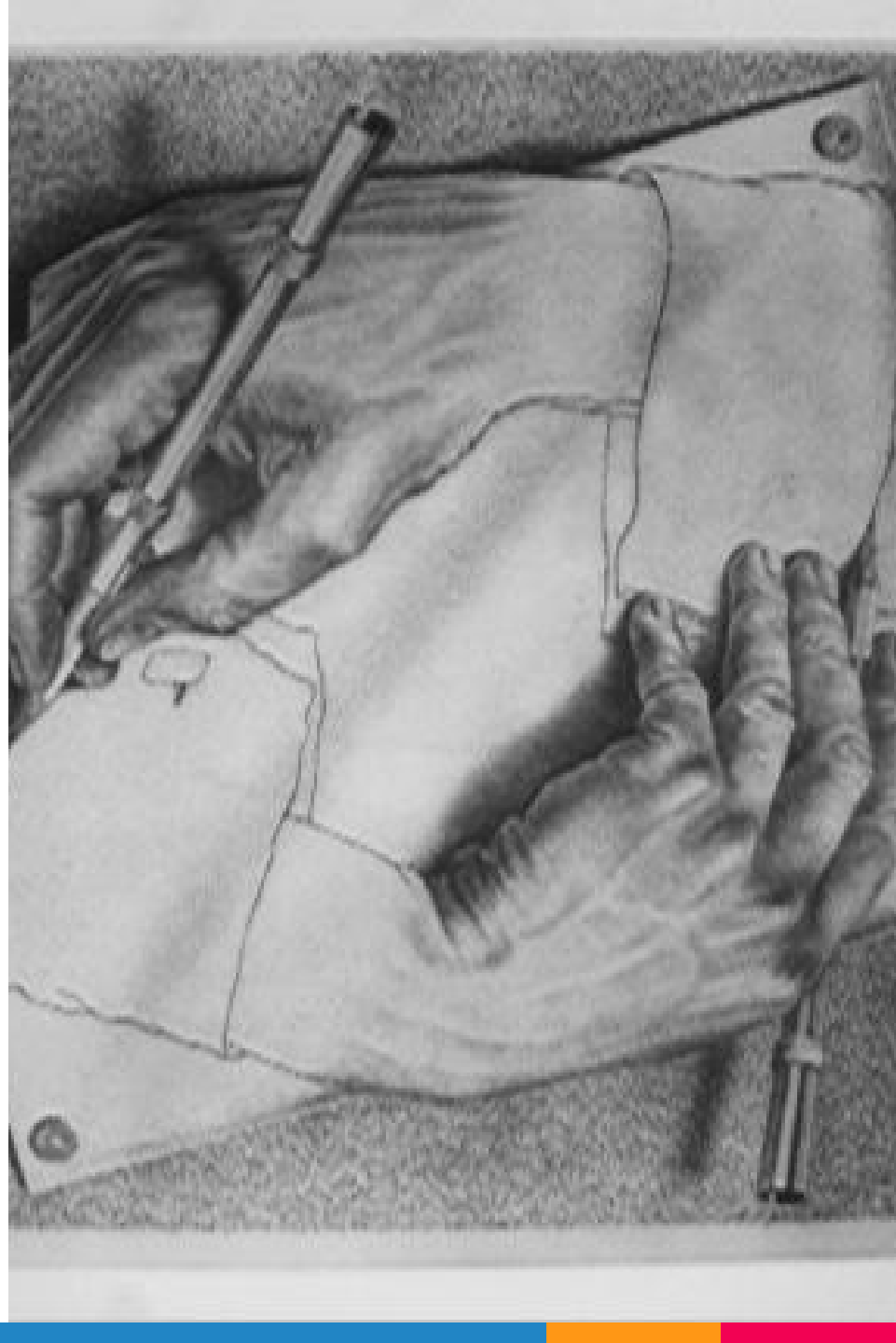
“To understand recursion, one must first understand recursion” - Darth Vader

RECURSION

Recursion is where you call a function from its own body.

```
def stack(overflow):
    return stack(overflow)
```

It's technically recursion… but what is wrong with the function above?



Your base case

is the smallest or simplest case.

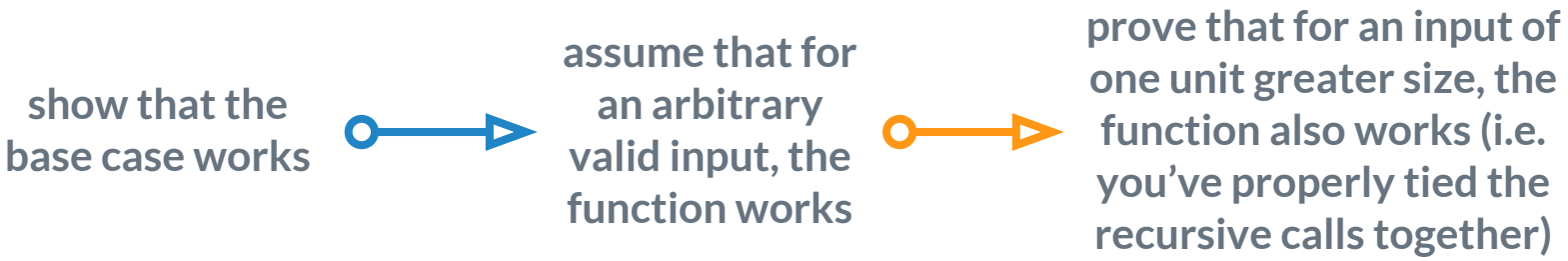
Your recursive case

defines the problem in terms of a simpler problem.

Tie the two together

by working toward the base case in your recursive calls.

Why does recursion work? Let's prove the correctness of one example using induction...



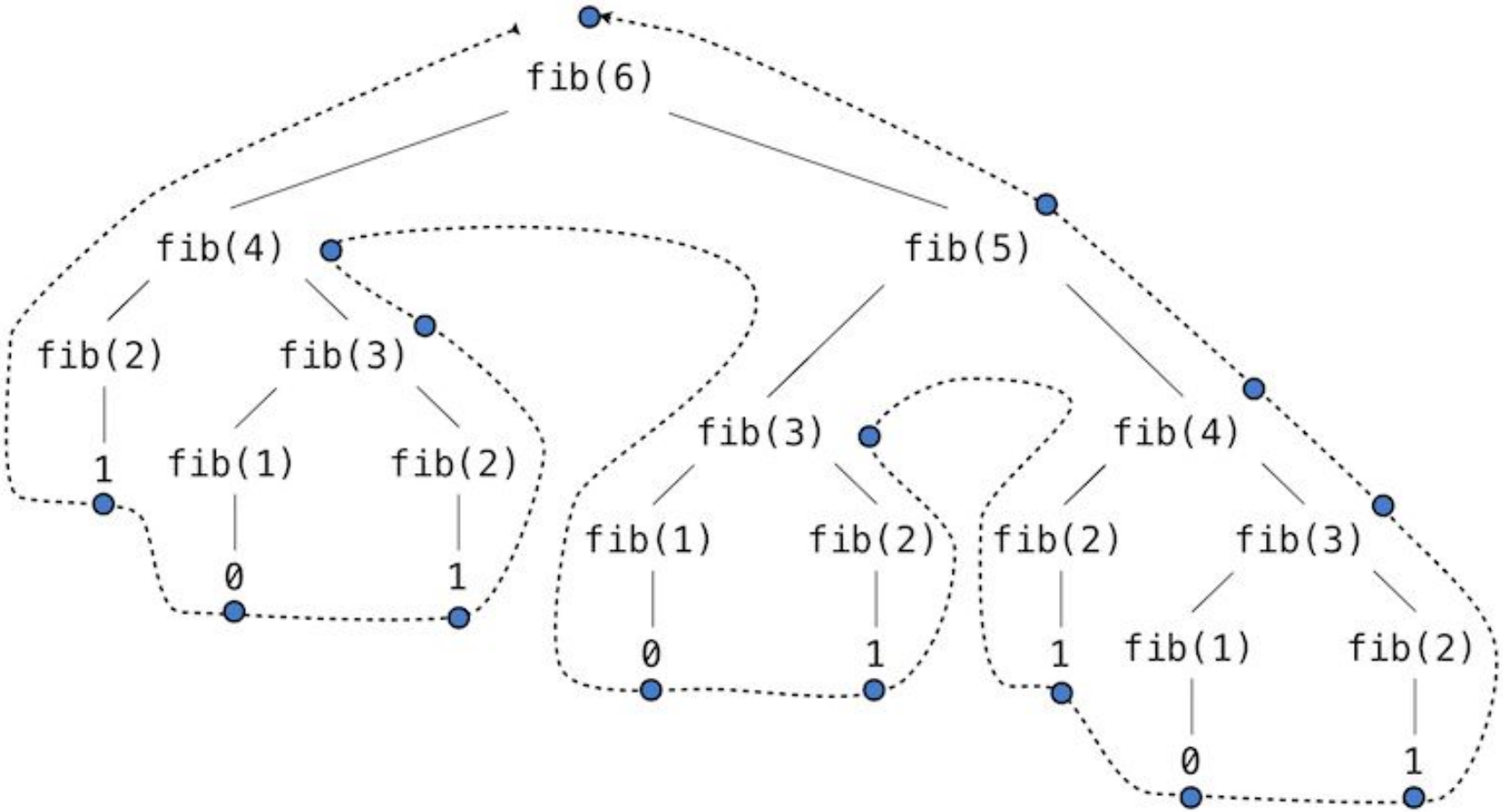
If we can prove the above, we've proved correctness for all inputs. Thus, in practice we can take the "recursive leap of faith" under security of these types of proofs.

TREE RECURSION

Tree recursion is where you make *more than one recursive call* in a single function call.

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + \
           fib(n - 2)
```

(See how there are two recursive calls in the body above!)



The Fibonacci recursion tree. The blue dots depict the order in which the function calls return.

DEMO + CHALLENGE

Deep Blue and Gold

また来週！

Thanks, everyone...