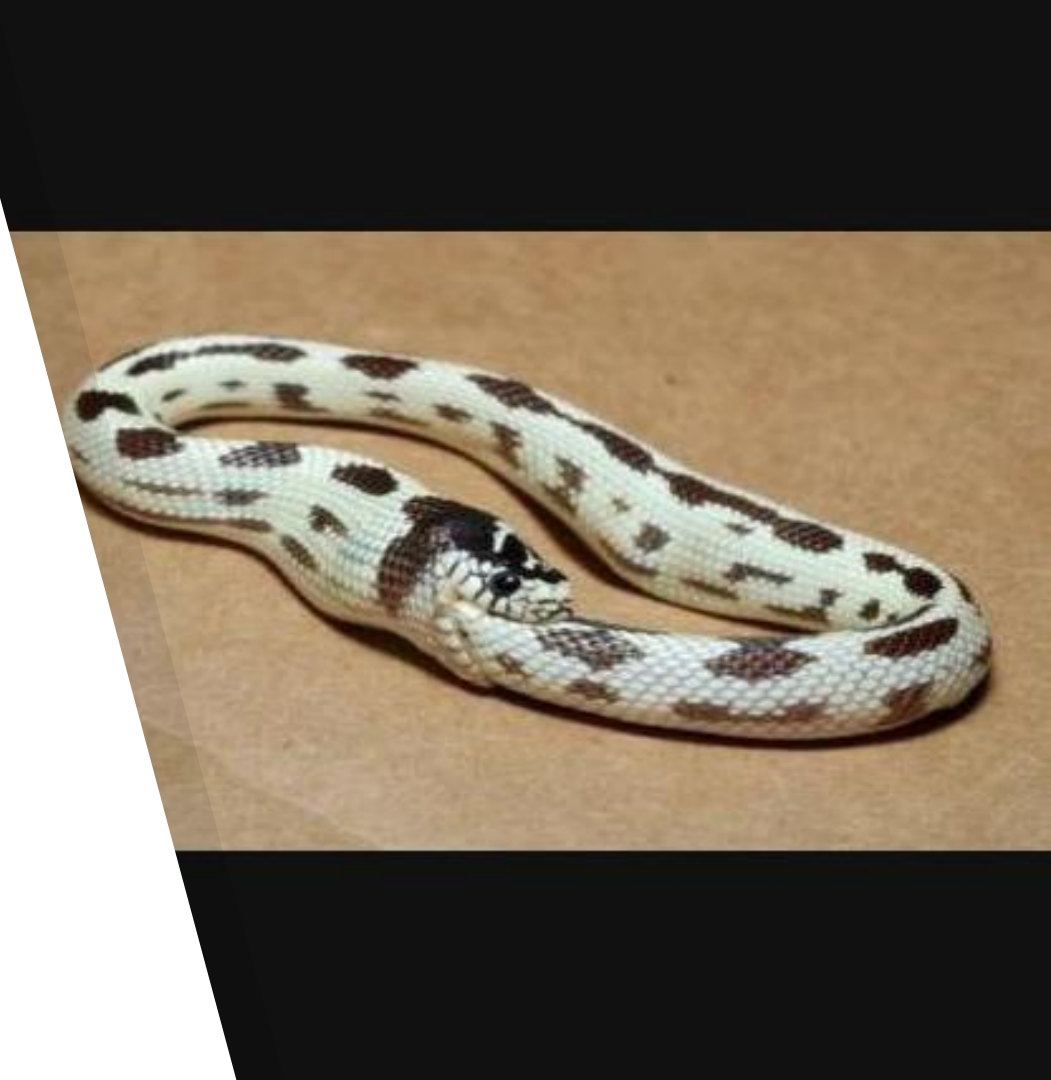
CS 61A
DISCUSSION 8

November 3, 2016



TOPICS FOR TODAY

- Interpreters
- Tail recursion

* despite the connotations of the image in the title slide, Python isn’t actually optimized for tail recursion.

ANNOUNCEMENTS

- The Scheme project has been released! Part 1 is due in a week, which means you may as well pretend that the whole project is due in a week. :)

(While you're at it, you should also pretend that the extra credit question is required.)

INTERPRETERS

Interpreters

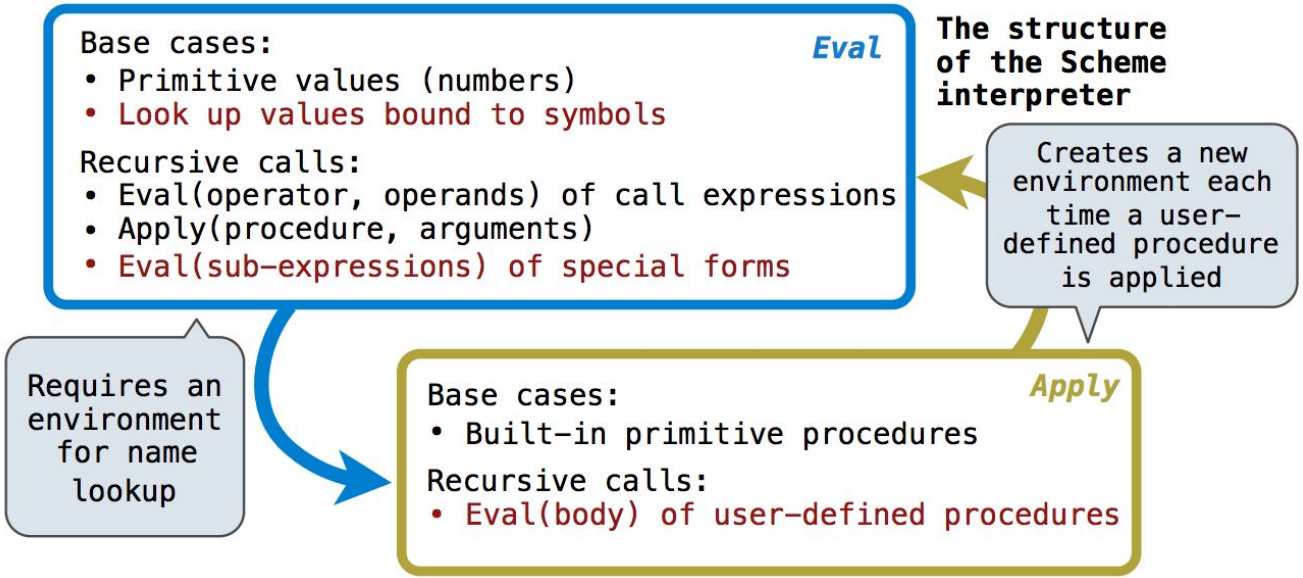
tl;dr An interpreter is a program that reads, evaluates, and executes other programs (i.e code) on a line-by-line basis.

Basically, it takes in code and evaluates it ("applying" functions to arguments when necessary). How? As in the lab:

- **Step 1**: Read the code (treat it as a string) and turn it into whatever format the interpreter wants to work with
  - Generally means **tokenizing** the code (breaking it into pieces) and sticking it into data structures in the underlying interpreter language
  - e.g. Reading in Scheme code and turning it into `Pair` objects in Python (remember, Scheme code is really just a bunch of primitives and lists)

Interpreters

- **Step 2**: Once the code is in an acceptable format, continually "evaluate" and/or "apply" the expressions until everything's been executed.



TAIL RECURSION

Tail recursion explained

tl;dr It's a technique that increases spatial efficiency during recursion.

How? By having the recursive call be the last thing to happen in the function body. If this is the case, then *the frame from which the recursive call sprung is now redundant* and we can kick it off the stack! (None of the frame's information will ever be needed, since the recursive call concluded the execution of its associated function body.)

Possible Q: What if there are nonlocal variables and some kind of multi-function recursion setup?
A: That s*** isn't in Scheme, and Python doesn't support tail recursion anyway.

Tail recursion rephrased

Tail recursion means turning all of your recursive calls into **tail calls**. (*What is a tail call*? It's technically defined as a call in a tail context – but if you find that confusing, just think of it as a call that's the last thing to happen in the function body.)

In a tail-optimized language implementation, tail calls let us reuse frames. Since the tail call is the last thing that happens in a function body, we don't need to retain data from a frame that makes a tail call. Thus, when we execute the tail call we can just overwrite the old frame in memory – we won't have to create a new one.

This means constant space!
(/no stack overflows from excessive recursion depth)

Tail contexts

[Disclaimer: You can memorize this if you want, but as far as I'm concerned just know that it's a tail context if it's a spot in the function body after which the function is finished, i.e. *nothing else will happen in the function body* (this *function body, at least*) *afterward*.]

- **last subexpression** in a lambda or a `let`'s body
- the **second** or **third expression** in an `if` form
- **any** of the **non-predicate subexpressions** in a `cond` form
- the **last subexpression** in an `and` / `or` form
- the **last subexpression** in a `begin`'s body

If the expression in any of those contexts is a procedure call, it's a tail call. It might not be a recursive tail call (the kind that saves space), but it's a tail call.

Defining tail-recursive procedures

Normally you just create a helper function and pass along all the information you need as extra arguments (for example, the return value you're building up).

```
(define (factorial n)
  (define (factorial-helper n result)
    (if (= n 0) result
        (factorial-helper (- n 1) (* n result))
    )
  )
  (factorial-helper n 1)
)
```

DISCUSSION ATTENDANCE

http://tiny.cc/threeeggs

QUIZ 8