Discussion 08:
Interpreters and Tail Recursion

TA: **Jerry Chen**
Email: **jerry.c@berkeley.edu**
TA Website: **jerryjrchen.com/cs61a**

Agenda

1. Attendance
2. Feedback
3. Announcements
4. Tail Calls
5. Calculator

Attendance

Sign in at bit.do/jerrydisc

OR

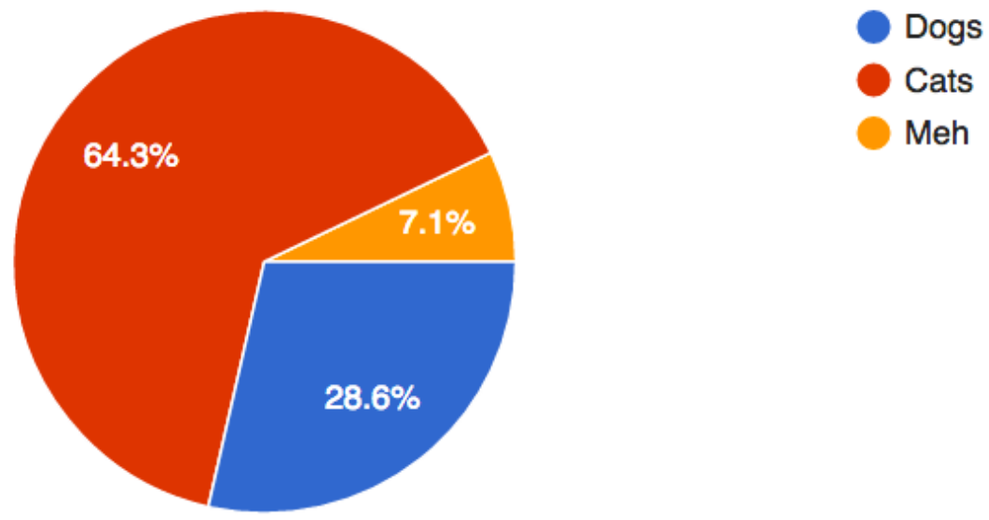
Come to me for check-in

Feedback

Thanks to those who filled it out!

- Some of you asked for harder discussion questions — I'll try to work those in but the majority of responses like the current difficulty

- A bit more time on worksheet vs lecture



Announcements

MT 2 Grades released

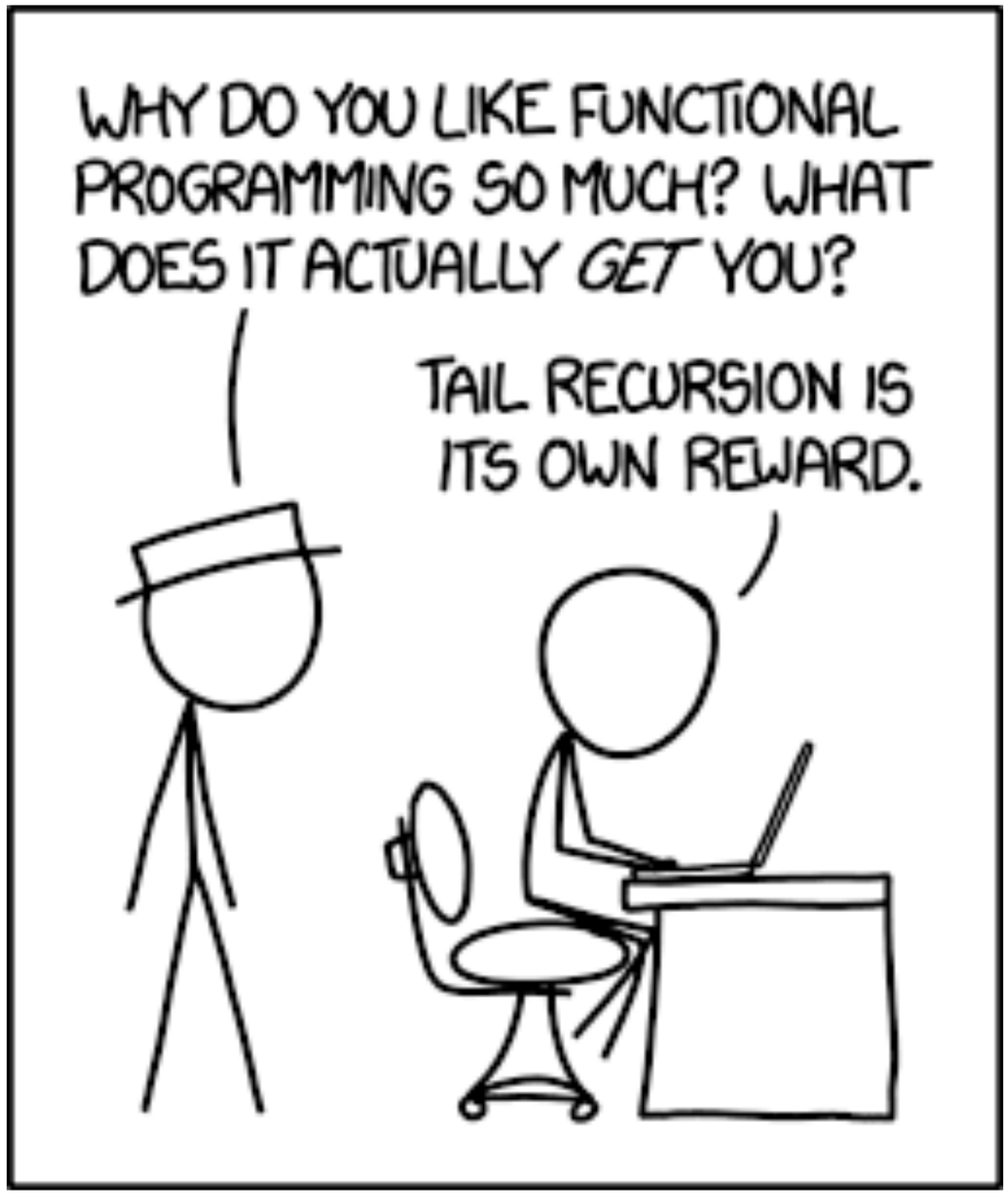
- As always, feel free to email me to chat!
- My OH are only 4-5pm this week and next

Start the Scheme project early!

November 8th

Next week's discussion (11/10) in 380 Soda. Priority to those officially enrolled in my section

Tail Calls



http://xkcd.com/1270/

Tail Calls

Scheme is **recursion only**

- Usually, recursive calls will take up space (think extra frames in the env diagram)
- **Tail calls** allow **recursion using constant space => efficiency of iteration!**
- **Tail recursion** is recursive calls performed at the end ("tail") of a function

Tail Calls

Big idea: with a valid tail call setup, **a recursive call** does not need anything **from the current frame** after it returns

- Put another way, after we do the recursive call, we **do not need to return for any computation**

- This is important because it means we can **reuse the current frame!** (might still need info for lookups)

Tail Calls

```
(define (fact n)

  (if (= n 0)

      1

      (* n (fact (- n 1)))))
```

Tail Calls

```
(define (fact n)

  (define (fact-tail n result)

    (if (= n 0)

      result

      (fact-tail (- n 1) (* n result))))

(fact-tail n 1))
```

Tail Calls

Usually use a **helper** function to **track state**

Recursive call must be in a **tail context** to be a valid tail call

Tail Context

Tail contexts are essentially places we know a function terminates from ("tail end")

There's a list of them in the discussion handout. Think about why they make sense!

Tail Calls

Summary

- **Tail calls let us use constant space** for recursive calls

- To do a tail call, must perform **recursive calls in a valid tail context**

- Valid tail contexts are at certain "tails" of expressions, and **must not require addl. work** after the recursive call

The End (of Tail Recursion)

Calculator

```
>>> (+ 1 (* 5 2))
  File "<stdin>", line 1
    (+ 1 (* 5 2))
               ^
SyntaxError: invalid syntax
>>> screw it, i'm going back to Scheme
  File "<stdin>", line 1
    screw it, i'm going back to Scheme
              ^
SyntaxError: invalid syntax
>>>
```

Calculator

The humble **Calculator** langage:

- Polish-prefix notation
- Math only
- (Scheme… but less impressive)

Calculator

Supports argument nesting, and the 4 basic arithmetic operations:

```
> (+ (* 4 500) (- 26 (/ 20 2)))
2016
```

Calculator

Expressions are Pairs… seem familiar?

Calculator expressions structured (mostly) the same as Scheme expressions

Pair is the Python data structure equivalent for Scheme cons

Calculator

Recall: evaluating call expressions

- **Evaluate** the operator
- **Evaluate** the operands
- **Apply** the operator to the operands