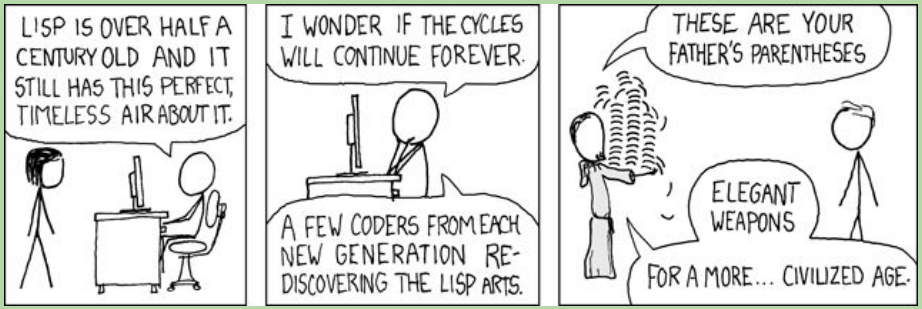
Welcome to 61A Lab!



We will begin at **5:10**!

Slides: **cs61a.bencuan.me**

Announcements

- Scheme week!!!
- Magic: The Lambdaing due Fri. for extra credit

The Plan

- Basic Scheme Syntax
  - Assignment
  - Functions and Procedures
  - Control (if, cond)

Scheme

Stop and do this now!!

- If you use VSCode, install the extensions vscode-scheme and Bracket Pair Colorizer
- This will make your life 1000x nicer when writing scheme code!!

“Why are we doing scheme?? it sucks”

- Show you that 61A concepts carry over to most other programming languages
- Makes you so much better at recursion problems
- Can make an interpreter for it in 2 weeks (proj4)
- Hopefully will appreciate later, it’s fine if you don’t like it now

Scheme, Generalized

- Only recursion, no iteration
- Everything, including operators, go **inside** parentheses!
  - f(x, y) in python becomes (f x y) in scheme
- Everything is a (linked) list
  - We will talk about this more in discussion

Scheme Resources

- go.cs61a.org/ben-scheme
- Scheme Specification: https://cs61a.org/articles/scheme-spec/
- Built-In Procedures: https://cs61a.org/articles/scheme-builtins/

Variables

Variables	Scheme	Python
Numbers	123	123
Booleans	#t, #f	True, False
Assignment	(define hippo 1) *<returns hippo>*	hippo = 1 *<returns None>*

Booleans part 1

Booleans	Scheme	Python
And	`(and (+ 1 2) 'hi)`	`(1 + 2) and 'hi'`
Or	`(or (* 3 4) '(1))`	`(3 * 4) or Link(1)`
Not	`(not (- 5 6))`	`not (5 - 6)`
Truthy Values	`0, (print 'hi), #t, (list 1), nil, '(), etc.`	`'hi', -1, [3, 5], etc.`
Falsey Values	`#f`	`0, False, [], None, etc.`

Booleans part 2

Null check	(null? duck)	duck is None
Type checks	(<TYPE>? x) *<TYPE>: list, boolean, integer, atom...*	isinstance(x, <TYPE>) *<TYPE>: str, int, list, dict...*
Even/odd	(even? 61) (odd? 61)	61 % 2 == 0 61 % 2 == 1
Equals	(= a b) *<NUMBERS ONLY>* (eq? a b) *<NUMS/BOOLS/SYMBOLS>* (equal? a b) *<LISTS/PAIRS - checks if each element is equal>*	a == b a is b *(not exact equivalence; see https://cs61a.org/articles/scheme-builtins/#general for more info)*

Functions/Procedures

Functions	Scheme	Python
Function Definitions	`(define (f x) (+ x 1))`	`def f(x):` `  return x + 1`
Lambdas	`(lambda (elephant) 7)`	`lambda elephant: 7`

Higher order functions	`(define (f x)`
	`  (define (g y) (+ x y))`
	`  g`
	`)`
		`def f(x):`
		`  def g(y):`
		`  return x + y`
		`  return g`

If and Cond

Control Statements	Scheme	Python

If	`(if (< 4 5) 'yes 'no)`	`'yes' if (4 < 5) else 'no'`
		
		- OR -
		`if 4 < 5:`
		`  return 'yes'`
		`else:`
		`  return 'no'`

Elif/Cond	`(if (< a b) 1`
	`  (if (> a b) 2 3))`
	
	- OR -
	`(cond`
	`  ((< a b) 1)`
	`  ((> a b) 2)`
	`  (else 3)`
	`)`
		`if a < b:`
		`  return 1`
		`elif a > b:`
		`  return 2`
		`else:`
		`  return 3`

Begin and Let*

*not on today's lab

Begin *(Multi-line expressions)*	(begin (print 'cs61a) (print 'is_awesome!))	print('cs61a') print('is_awesome!') *<python doesn't need begin, just type multiple lines!>*
Let *(Temporary assignment)*	(let **((**x 1) (y 2)**)** (+ x y))	(lambda x, y: x + y)(1, 2) *<not a 1-1 correlation! let doesn't exist in python>*

Lab Hints

- Parentheses in Scheme go **before** the operator, and no commas!!!
  - (operator x y z …) NOT operator(x, y, z…)
- Write solutions out in Python, then convert to Scheme
- “return a procedure” ⇒ make a lambda
- The blue hint boxes are very helpful
- In Scheme, the last expression in a define is automatically returned