Hi! Welcome to 61A Discussion :)

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

Attendance form and skeleton notes:

cs61a.bencuan.me

Announcements

- Magic The Lambdaing due Friday

Agenda

- Attendance
- Looking ahead
- Scheme Review + WWSD
  - Variables, call expressions, special forms, lambdas, lists
- Scheme code writing practice

Scheme Review

Today's goals

- Help you feel more confident about Scheme
- Explore similarities (and differences) between Scheme and Python
- Give you a bunch of examples to reference later
- Do a bunch of list stuff with cons car cdr

General Scheme Tips (now with lists)

- **Everything is a list!** (except for special forms)
- Scheme lists are in the form:
  (operator operands...)
- So f(x) in python would be (f x) in scheme
  - Move all your parentheses to the front
- All manually created lists (list, cons, quote) become **linked lists**
  - Must use car and cdr to work with them

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

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>*

Q2: Virahanka-Fibonacci

Write a function that returns the `n`-th Virahanka-Fibonacci number.

```
(define (fib n)
    'YOUR-CODE-HERE
)

(expect (fib 10) 55)
(expect (fib 1) 1)
```

Scheme Lists

List Operations

List Operations ***ALL SCHEME LISTS ARE LINKED LISTS!***	**Scheme**	**Python**
Create list	`(cons first rest)`	`Link(first, rest)`
Get value	`(car lst)`	`lst.first`
Get rest	`(cdr lst)`	`lst.rest`
Empty list	`nil, '()`	`Link.empty`
Make long list	`(list 1 2 3)` *OR* `'(1 2 3)` *OR* `(quote (1 2 3))` *OR* `(cons 1 (cons 2 (cons 3 nil)))`	`Link(1, Link(2, Link(3, Link.empty)))`

Comparing Items

- =
  - Used for **numbers only!**
- eqv?
  - Equivalent to python **is**
- equal?
  - Used for **comparing lists**

Comparing Items

```
scm> (define a '(1 2 3))
a
scm> (= a a)
Error
scm> (equal? a '(1 2 3))
#t
scm> (eqv? a '(1 2 3))
#f
scm> (define b a)
b
scm> (eqv? a b)
#t
```

<- Numbers only!!

<- Not the same object

<- Is the same object

A subtle define difference (bonus)

What is the difference between:

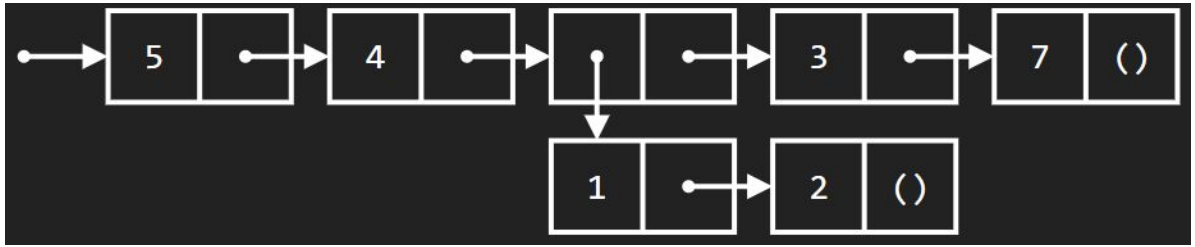
A.
```
(define x (+ 1 2 3))
```

B.
```
(define (x) (+1 2 3))
```

List practice (bonus)

Write an expression that selects the value 3 from the list below.

```
(define s '(5 4 (1 2) 3 7))
```



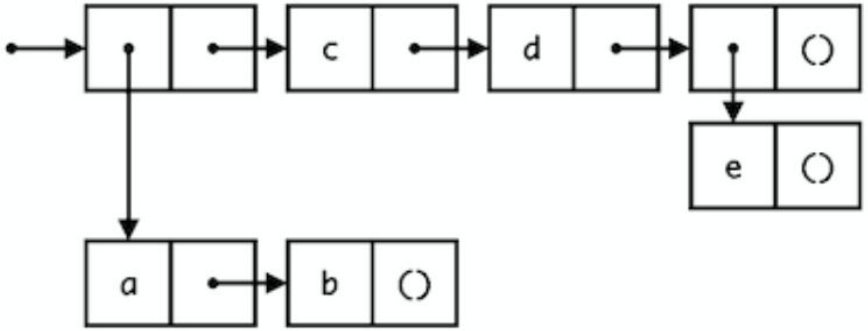
Hint: how would you do this in python (using linked lists)? What is the scheme equivalent?

List practice pt.2

Q3: List Making

Let's make some Scheme lists. We'll define the same list with `list`, `quote`, and `cons`.

The following list was visualized using the `draw` feature of code.cs61a.org.



Scheme List Skeleton

Scheme version

```
(define (f lst)
    (if (null? lst)
        ;BASE CASE

        ;RECURSIVE CASE
        (cons (do stuff to car a)
              (f (cdr lst))
        )
    )
)
```

Python version

```
def f(lst):
    if lst is Link.empty:
        # BASE CASE

    else:
        # RECURSIVE CASE
        new_first = do stuff to lst.first
        return Link(new_first, f(lst.rest))
```

Q4: List Concatenation

Write a function which takes two lists and concatenates them.

Notice that simply calling `(cons a b)` would not work because it will create a deep list. Do not call the builtin procedure `append`, since it does the same thing as `list-concat` should do.

```
(define (list-concat a b)
    'YOUR-CODE-HERE



)

(expect (list-concat '(1 2 3) '(2 3 4)) (1 2 3 2 3 4))
(expect (list-concat '(3) '(2 1 0)) (3 2 1 0))
```

Hints:

- Try it in python first (linked lists) if stuck!
- Use cons to create a new list each time

Q5: Map

Write a function that takes a procedure and applies it to every element in a given list using your own implementation *without* using the built-in `map` function.

```
(define (map-fn fn lst)
    'YOUR-CODE-HERE
)

(map-fn (lambda (x) (* x x)) '(1 2 3))
; expect (1 4 9)
```

Hints:

- Use recursion!
- What happens to (car lst)?
- What about (cdr lst)?

Q6: Remove

Implement a procedure `remove` that takes in a list and returns a new list with *all* instances of `item` removed from `lst`. You may assume the list will only consist of numbers and will not have nested lists.

> *Hint*: You might find the built-in `filter` procedure useful (though it is definitely possible to complete this question without it).
>
> You can find information about how to use `filter` in the 61A Scheme builtin specification!