Prep: bring ID, watch, jacket, a pencil, glasses, lots of paper, pen, water and this set of notes
You got this!

Scheme

Arithmetic Operations

(+ [num] …)	(abs <num>)
scm> (+ 15 6 2) 23	scm> (abs -22) 22
(* [num] …)	(/ <dvdend> [dvsor] …)
scm> (* 3 2 3) 18	scm> (/ 16 2 2 2 2) 1
(- <num> …)	(expt <base> <power>)
scm> (- 1) -1 scm> (- 16 7) 9	scm> (expt 3 2) 9 scm> (expt 3 (- 1)) 0.333333333
(quotient <dvdend> <dvsor>)	(remainder <dvdend> <dvsor>)
scm> (quotient 7 3) 2 scm> (quotient -4 3) -1	scm> (remainder 7 3) 1 scm> (remainder -8 3) -2 scm> (remainder -9 3) 0

Type Check

(null? <arg>)	(boolean? <arg>)
scm> (null? ()) #t scm> (null? nil) #t scm> (null? 0) #f	scm> (boolean? ()) #f scm> (boolean? #t) #t scm> (boolean? 1) #f
(list? <arg>)	(procedure? <args>)
scm> (list? ()) #t	scm> (procedure? abs) #t

List Manipulation

(car <list>)	(cdr <list>)
scm> (define x '(+ 1 2 3)) scm> (car x) +	scm> (cdr x) (1 2 3)
(length <args>)	(list <item>)
scm> (= (length '()) 0) #t	scm> (list 1 2 3) (1 2 3)
(cons <first> <rest>)	(map <procedure> <list>)
Equivalent to `Pair(first, rest)`, `rest` should be a list. There is a `nil` at the end (!) scm> (cons 1 2) (1 . 2) scm> (cons 1 (cons 2 (cons 3 nil))) (1 2 3)	<proc> one argument procedure scm> (map (lambda (x) (* x x)) '(1 2 3)) (1 4 9) scm> (map abs '(1 -1 3)) (1 1 3)
(append [list] …)	(reduce <combiner> <lst>)
ALL arguments are lists! Useful for adding to end of lists scm> (append '(1 2 3) '(4 5 6)) (1 2 3 4 5 6) scm> (append) () scm> (append '(1 2 3) '(a b c) '(foo bar baz)) (1 2 3 a b c foo bar baz)	<combiner> Two args procedure Left (combined value) to right scm> (reduce (lambda (x y) (+ x y)) '(1 6 5 7)) 19 scm> (reduce (lambda (x y) (+ x y)) '(1)) 1 scm> (reduce (lambda (x y) (+ x y)) '()) Error
(filter <pred> <lst>)	(apply <proc> <list>)
<pred> one argument procedure scm> (filter (lambda (x) (<= x 5)) '(1 6 5 7)) (1 5)	scm> (apply + '(1 2 3)) 6
(set-car! <pair> <val>) (set-cdr! <pair> <val>)	(eval <expression>)
scm> (define x '(1 2)) x scm> (set-car! x 3) scm> x (3 2) scm> (set-cdr! x 3) scm> x (3 . 3)	scm> (eval '(cons 1 (cons 2 nil))) (1 2) scm> (eval (list 'quotient 10 2)) 5

Boolean Operations

(eq? <a> <b>)	(equal? <a> <b>)
#t if a, b are same num, bool, symbol, str or object in memory scm> (eq? '(1 2) '(1 2)) #f scm> (define x '(1 2)) scm> (eq? x x) #t	Checks if a and b are equivalent scm> (equal? '(1 2) '(1 2)) #t scm> (equal? op '*) #t
(not <arg>)	(= <a> <b>) (< <a> <b>) (<= <a> <b>)
scm> (not 0) #f scm> (not #f) #t	scm> (= 2 2) #t
(even? <num>)	(odd? <num>)
scm> (even? 22) #t	scm> (odd? 22) #f

Special Forms

(define <name> <expr>) (define (<name> [param] …) <body> …)	(if <pred> <conseq> [alt])

scm> (define x '(1 2))
x
scm> (define op '*)
op
 	
(define (f x) (+ x 1))
 	
(define (f x)
  (define (g y) (+ x y))
  g
)
	(if (null? nums) 0
	(length nums))
	
	(if (equal? op '*)
	  (list 'mul (cons 'list rest))
	  (cons op rest)))))

(or [test] …)	(and [test] …)
#f if no args scm> (or #f #f #f) #f scm> (or #f 2 3) 2	#t if no args scm> (and (> x 10) (< x 20))
(cond <clause> …)	(let ([binding] …) <body> …)

<clause> is of the form
(<test> [expression] …)
(else [expression] …)
If all clauses are #f, undefined.
 	
(cond
  ((< a b) 1)
  ((> a b) 2)
  (else 3)
)
	[binding] in the form of
	(<name> <expression>)
	
	(let ((x 1) (y 2))
	  (+ x y))
	
	(let ((x 5) (y 10))
	  (print x)
	  (print y)
	  (+ x y))

(begin <expression> …)	(lambda ([param] …) <body> …)

(define (sortedpair a b)
 (if (> a b)
  (begin
  (print a)
  (cons a (cons b nil))
)
  (begin
  (print b)
  (cons b (cons a nil))
)
)
)
	(lambda (jz) 7)
	
	Immediate calling of `lambda` functions
	
	scm> ((lambda (x y) (+ (* x x) (* y y))) 3 4)
	25
	scm> ((lambda () 4))
	4

(quote <expr>)	(quasiquote <expr>)
equivalent to '<expr> scm> (quote (1 2 3)) (1 2 3) scm> (quote (abs 3)) (abs 3) scm> (quote '(abs 3)) (quote (abs 3))	scm> (define (make-adder n) `(lambda (d) (+ d ,n))) make-adder scm> (make-adder 2) (lambda (d) (+ d 2)) scm> ((eval (make-adder 2)) 3) 5
(unquote <expr>)	mu (dynamic scoping)
equivalent to ,<expr> <see quasiquote for example>	**Dynamic scoping**: parent of the frame is the frame in which the procedure is called.

Scheme (continued)

List equivalence

'(1 2 3)	(list 1 2 3)	(quote (1 2 3))
(cons 1 (cons 2 (cons 3 nil)))		

Reversing a list

```
(define (reverse lst curlst)
  (if (null? lst)
    curlst
    (reverse (cdr lst) (cons (car lst) curlst)))

(define (deep-reverse lst)
  (cond ((list? lst) (reverse (map deep-reverse lst) nil)) (else lst))
```

Tricks

Symbols	Dot '.'
scm> ((lambda (x) `(,x 2)) 4) (4 2)  scm> ((lambda (x) `(x 2)) 4) (x 2)	If latter is list, then equivalent to cons scm> '(1 . (2)) (1 2) scm> '(1 . (2 3)) (1 2 3)

Backus-Naur Form (BNF)

Can parse more things.

```
rstring: "r\"" regex* "\""
?regex: character | word | group | pipe |
class | range | plus_quant | star_quant |
num_quant

character: LETTER | NUMBER
word: WORD
group: "(" regex ")"
class: "[" (character*range*)* "]"
plus_quant: (group | character | class) "+"
star_quant: (group | character | class) "*"
num_quant: (group | character | class) "{"
interval "}"
?interval: (NUMBER+ ",") | ("," NUMBER+) |
("NUMBER+ "," NUMBER+)

%import /\s+/
%import common.LETTER
%import common.NUMBER
%import common.WORD
```

Calculator

```
def calc_eval(exp):
  if isinstance(exp, (int, float)):
    return exp
  elif isinstance(exp, Pair):
    arguments = exp.rest.map(calc_eval)
    return calc_apply(exp.first, arguments)
  else: raise TypeError

def calc_apply(operator, args):
  if operator == '+':
    return reduce(add, args, 0)
  elif operator == '-':
    # logic
  elif operator == '*':
    # logic
  elif operator == '/':
    # logic
  else: raise TypeError
```

Regex (`import re`)

- `re.match(<pattern>, <str>)`

Anchored at the start. Searches from the start of `<str>`, returns `True` if matches any prefix.

```
>>> x = re.match("jz", "jz)
<re.Match object; span=(0, 2), match='jz'>
>>> bool(x)
True
```

- `re.search(<pattern>, <str>)`

Not anchored. Searches and matches any part of `<str>`

```
>>> m = re.search("(jz).*(app)", "jz app")
<re.Match object; span(0, 6), match='jz app'>
```

>>> bool(m)
2
>>> m.groups()
('jz', 'app')
	>>> m.group(0)
	'jz app'
	>>> m.group(1)
	'jz'

- `re.findall(<pattern>, <str>)`

Returns `list` of all matches

```
txt = "The rain in Spain"
x = re.findall("ai", txt) # ['ai', 'ai']
y = re.findall("in$", txt) # ['in']
m = re.findall("(jz).*(app)", "jz app")
# [('jz', 'app')]
```

- `re.fullmatch(<pattern>, <str>)`

Anchored at start and end. Must remember to `bool()`

```
>>> bool(re.fullmatch(r'\s', '\n'))
True
>>> bool(re.fullmatch(r'[^A-Za-z]', 'b'))
False
>>> bool(re.fullmatch(r'hi|hello', 'hello'))
True
```

Regex	Explanation
[abc]	'a' or 'b' or 'c'
[a-z]	any char between 'a' and 'z' (incl)
[^A-Z]	any char not between 'A' and 'Z' (incl), applies to the whole []
\w	equiv to [A-Za-z0-9_]
\d	equiv to [0-9]
\s	any whitespace char, including ' ', '\t', '\n', '\r'
Combinations	
AB	Concatenation (A followed by B)
A|B	Or (Either pattern A or pattern B)
()	Capturing group
(?:)	Non capturing group
Anchors	
$	Start of line character
^	End of line character
\b	Word boundary (zero-length) - Before first char if it is \w - After last char if it is \w - Between two char, one \w, one ^\w
Special Characters (12 of them)	
'\', '{', '[', '(', ')', '+', '*', '?', '$', '^', '.', '|'	
Quantifiers (apply to previous)	
?	0 or 1
*	0 or more
+	1 or more
{1,3}	Range from 1 to 3 inclusive

Examples

r'-?\d+'	Matches any number (negative too)
r'.*'	Matches any string

SQL

General Structure

```
SELECT _____ FROM _____ [WHERE _____] GROUP BY
_____ HAVING _____ ORDER BY _______ [ASC|DESC]
```

Order of Operations:

1. WHERE to filter
2. GROUP by
3. HAVING to filter the groups
4. ARITHMETIC OPERATIONS IN SELECT like MAX(), COUNT(*) and ORDER BY

Arithmetic Functions (in SELECT)

Function	Description / Use case
ROUND(col)	ROUND(release_year/10.0)
AVG(col)	Average value of numeric column
SUM(col)	Total sum of numeric column
MAX(col)	Maximum entry of numeric column
MIN(col)	Minimum entry of numeric column
COUNT(*)	Number of rows in table; preserves duplicate row; include rows with null.
COUNT(col)	Number of non-null entries in col
COUNT (DISTINCT col)	Number of distinct entries in col

Creating Table Examples

```
CREATE TABLE table1 AS
  SELECT col1, col2 FROM table WHERE
col1="jz" AND col2="jj";
```

```
CREATE TABLE table2 AS
  SELECT col1 FROM table GROUP BY smallest
HAVING COUNT(smallest)=1 ORDER BY smallest
```

```
CREATE TABLE table3 AS
  SELECT a.col1 FROM table1 AS a, table2 AS
b WHERE a.col2=7 AND b.'7'='True';
```

```
CREATE TABLE table4 AS
  SELECT ROUND(AVG(ABS(a.number –
b.number))) FROM students AS a;
```

```
CREATE TABLE table5 AS
  SELECT name FROM dino WHERE name <> "t-
rex" GROUP BY weight/legs HAVING COUNT(*)>1
```

Common Phrases

```
HAVING SUM(weight/legs) > 0;
```

```
SELECT MAX(legs) – MIN(legs) AS diff FROM
animals GROUP BY weight ORDER BY diff DESC
LIMIT 1;
```

```
SELECT "Thank you, " || name FROM players
WHERE SUBSTR(name, 5) = "Chase";
```

```
SELECT team, SUM(points) FROM scoring,
players
WHERE player=name GROUP BY team;
Cal 24
Stanford 20
```

```
SELECT DISTINCT col1, col2 FROM table;
```
Returns pairwise distinct col1, col2 entries.

Trampolining

```
def trampoline(f, *args):
  v = f(*args)
  while callable(v):
    v = v()
  return v

def f_thunked(n, k):
  if n == 0:
    return k
  else:
    return lambda: f_thunked(n – 1, k * n)
```

Exception

- Try-except blocks

```
try:
  # logic
except <exp> as e:
  # logic
```

```
try:
  quot = 10/0
except ZeroDivisionError as e:
  print('Error, type(e))
  quot = 0
```

Example: `except StopIteration:`

- Assert

```
assert <expr>, <string>
```
raises exception of the type `AssertionError`

- Raise

```
raise <expr> # e.g. raise TypeError('bad arg')
```

Tail Recursion

- Last body sub-expression in a lambda expression
- Sub-expressions 2 and 3 in a tail context if expression
- All non-predicate subexpression in a tail context cond
- The last subexpression in a tail context and, or, begin or let

Bottom line: last evaluated function of the function is either terminal or the function itself.

Final Checks

Scheme

- Check all operators `(- x 1)` instead of `(x-1)` and `(< 1 2)` instead of `(1 < 2)`
- `cons` always 2 arguments (even `nil`)
- Cannot have a list of integers that are not in quotes (`Error: int is not callable`)
- Check each variable: list or procedure?

BNF

- Empty terminals not allowed – leave blank

Regex

- Remember anchors `(\b, $, ^)`
- Did you escape special characters?
- Use non-capturing groups for safety

SQL

- Remember the ';'
- Eliminate double counting (`a.id < b.id;`)
- Use single quotes
- Check: referenced correct table and cols.

General

For high-value problems, check and recheck the answers - can't afford loss of points

WWPD

```
>>> "hi"
'hi'
>>> 'hi'
'hi'
>>> 1 == True
True
>>> 0 == False
True
```

```
>>> repr([1, 2, 3])
'[1, 2, 3]'
>>> x = {}
>>> x["jz"] = "rn"
>>> repr(x)
"{'jz':'rn'}"
```

```
>>> repr("jianzhi")
"'jianzhi'"
>>> str("jianzhi")
'jianzhi'
>>> "jianzhi"
'jianzhi'
```

```
>>> repr(2)
'2'
>>> repr(None)
'None'
>>> str(None)
'None'
```

Tree, Link

- Link

Class Var	Constructor
`Link.empty`	`l = Link(1, Link(2))`
Instance Var	**Methods**
`l.first` `l.rest # must be Link instance`	

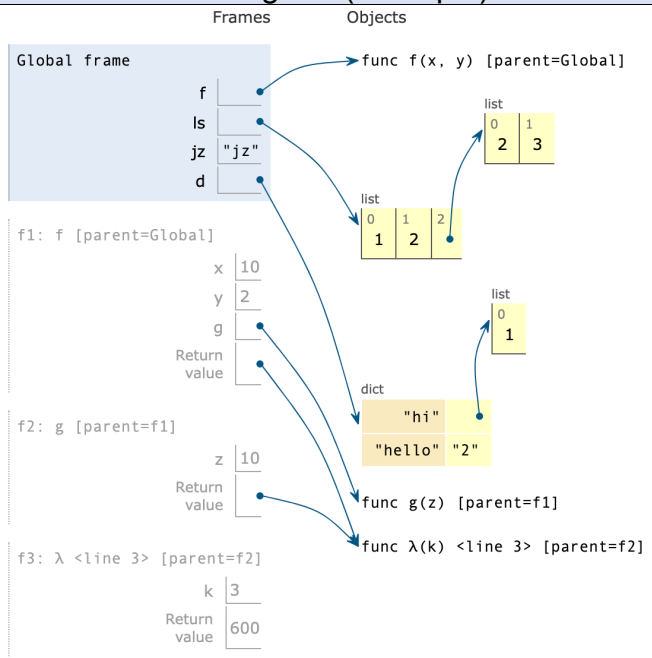
- Tree

Class Var	Constructor

`None`	`t = Tree(label, branches=[])`
	
	`t = Tree(2)`

Instance Var	**Methods**
`t.label` `t.branches`	`t.is_leaf() # FUNCTION`

Environmental Diagram (Example)



Data Structures

1. List

Method	Remarks
`ls.append(x)`	Returns `None`
`ls.insert(1, "a")`	Returns `None`
`ls.pop()`	Returns `<element>`
`ls.pop(1)`	Returns `<element>`
`ls.reverse()`	Returns `None`
`ls.remove(x)`	Returns `None`
`ls.extend([1])`	Returns `None`
`ls.copy()`	Returns new list
`ls.index(x)`	Returns index of `x`
`ls.count(x)`	Returns number of `x`
`ls[:]`	Returns new list

List Comprehensions

```
[x for x in range(1, 12) if 12 % x == 0]
[x if 12 % x == 0 else 1 for x in range(1, n)]
```

Shallow Copy

`list(ls)`	`[ls[0], ls[1], …, ls[n]]`	
`ls + []`	`ls[:]`	`ls.copy()`

2. String

```
" ".join(<iterable>)
" ".join(["b", "luv", "z"]) # delimiter
s.split("e") # split by delimiter into list
s.split() # default " "
```

3. Dictionary

Method	Return
`d.get("x", 0)`	`d[x] if "x" in d else 0`
`d.pop("key")`	`d["key"] else KeyError`
`list(d.keys())`	list of keys
`list(d.items())`	list of `(key, item)`

`dd = d.copy()`	returns a shallow copy
	`dd == d # True`
	`dd is d # False`

`d.clear()`	clears the dictionary
`d.popitem()`	returns the item last inserted and removes it
`d.setdefault("x", v)`	returns `d["x"]` if `x` in `d`, else return `d["x"] = v`

Iteration

`for i in d.keys():` `  print(i)`	`for k, v in d.items():` `  print(k, v)`
`iter(prices.keys())`	`iter(prices.items())`

Dictionary Comprehensions

```
x = {i : s[i] for i in range(len(s)) if s[i]
!= w}
```

Iterators and Generators

Iterables: `[]`, `()`, `"string"`, `range(1, 21)`, `{"dict": 1}`

Functions on Iterable

```
list(<iterable>)
list(<iterator>)
tuple(<iterable>)
tuple(<iterator>)
```

Iterable Functions:

- `list(<iterable>)`
- `type(<iterable>) # == tuple, list, dict, str`
- `sorted(<iterable>)`

```
if (type([1, 2, 3]) == list):
    # logic
```

Iterator Functions:

- `iter(<iterable>)`
- `iter(<iterator>)` returns itself
- `next(<iterator>)`
- `list(<iterator>) # exhausts the iterator`

```
for i in <iterator>: # python calls next()
    # logic
```

If called on an iterator, will exhaust it.
`StopIteration exception`

Generator:

- Key words: `yield`, `yield from`
- Usually comes with `while` loop
- If you do a `yield from`, remember to `return`

```
def gen(x):
  while x > 0:
    yield x
    x -= 1
```

```
def g(x):
  while x > 0:
    yield x
    yield from g(x//2)
    return
```

Generator Functions:

- `yield from <iterable>`
- `yield from <generator>`
- `list(<generator_with_args>)`

```
for i in <generator_function>:
    # logic
```

- reversed(<iterable>)

```
<reversed object at 0x7fe37836dc70>
# same as reversed iterator
it = reversed(["jz", "doing", "cs61a"])
next(it) # 'cs61a'
```

- all(<iterable>)

```
all(<iterable>) # returns a boolean
all([]) == True
```

- any(<iterable>)

```
any(<iterable>) # returns a boolean
any([]) == False
```

- max(<iterable>) / min(<iterable>)

```
max(["jz", "hi", "wang"]) = "wang" #
lexigraphic
max([]) # ValueError
max(["jianzhi", "hi", "wang"], key = lambda x:
len(x)) # "jianzhi"
```

- sum(<iterable>, <initial value>)

```
sum([]) == 0
sum([1, 2, 3]) == 6
>>> sum([[x] for x in [1, 2, 3]], [])
[1, 2, 3]
>>> sum([[[x]] for x in [1, 2, 3]], [])
[[1], [2], [3]]
sum([[t.label + x for x in f(y)] for y in
z], []) # double list summation
```

- string
  - `text.lower()`
  - `text.upper()`
  - `text.split()`

OOP and Inheritance

Method Resolution Order (MRO)

```
class A(B, C):
    # logic
    # searches A then B then C
```

Calling super's methods:

```
super().__init__(title, author)
super().__str__()
# calls the __str__(self) in super class
```

```
class Car:
    wheels = 4
    def __init__(self, name):
        self.name = name
        self.fuel = 0
    def add_fuel(self, x):
        self.fuel += x
x = Car("Tesla")
x.add_fuel(10) # equiv to Car.add_fuel(x, 10)

class FastCar(Car):
    def __init__(self, name, brand):
        super().__init__(name)
        self.brand = brand

    def add_fuel(self, x):
        super().add_fuel(2*x)
```

__Dunder__()

`str()` and `repr()`

```
>>> repr(x)
'<probably with quotes>'
```

`repr()` calls `__repr__()`, returns a **string**

```
>>> print(obj)
<str(obj) - probably no quotes>
```

`print(obj)` implicitly calls `str()` on it, and then removes one layer of quotes

```
>>> obj
<repr(obj) - probably no quotes>
```

Python calls `repr()` method on it, obtain `res`, calls print on `res`, which removes quotes.

```
>>> str("a")
'a'
```

Just strings

```
# called during print(), which removes quotes
# called during str()
# if not defined, defaults to __repr__
def __str__(self):
    return f'hi'
```

```
        # equiv to Car.add_fuel(self, 2*x)

Car.wheel #4
x.wheel #4
```

CANNOT to modify class var through instance

```
>>> getattr(obj, "name") # returns value or AttributeError
>>> hasattr(obj, "name") # returns boolean
```

```
# called during >>> obj
def __repr__(self):
    return f"PaperReam('{self.x}')"

def __iter__(self):
    return iter(self.leds)

def __iter__(self):
    for led in self.leds:
        yield led
```

Misc

Range (exclusive)

- `range(start=0, stop, step=1)`
- `range(start, stop)`
- `range(stop)`

zip(*iterables)

```
>>> zip([1, 2, 3], [7, 8, 9]) # iterator
<zip object at 0x7f810aa4f440>
for i, j in x: # next(x) returns tuple
    print(i, j)
```

map(func, iterable, …)

```
map(lambda x: x*2, [1, 2, 3])
# returns iterator on resultant list
>>> print(map(lambda x: x*2, [1, 2, 3]))
<map object at 0x7fa31936dc70>
list(map(lambda x: x*2, [1, 2, 3]))
```

filter(func, iterable)

```
>>> filter(lambda x: x % 2 == 0, [1, 2, 3, 4])
<filter object at 0x7fdb8bc52c10>
# returns iterator on resultant list
list(filter(lambda x: n%x==0, range(1, n+1)))
```

reduce(lambda, iterable)

```
from functools import reduce
total = reduce(lambda a, b: a+b, [1, 2, 3, 4])
```

- fstring (Do **NOT** mix ‘ and “)

```
return f'Current {self.item} stock: {self.stock}'
```

Last Resorts

1. Python ternary:

```
return x if <condition> else y
```

2. Tuple assignment:

```
x, y = y, x + y
```

3. Chaining assignment (just use tuple)

```
L = L.rest = L.rest.rest
```

Line of Attack

- Eye power, intuition
- Iteration, recursion (break into subproblems)
- Get hints from previous functions in the same problem; Inspect arguments & return value
- Exploit other values
- Lambda (particularly when # of args doesn't fit or need a placeholder), double lambda; Arguments can be functions too!
- Look at lambda variants below
- Recursion (`__str__` is sort of recursive)
- Whack, try test cases, whatever works
- Check and verify examples
- Generators (`for` and list comprehensions)

λ Functions

- 2 Recursive

```
def d(f):
    def g(i):
        # logic
        # calls parent function with args
        # i == x is memorized
        return (lambda x: f(x) or i == x)
    return g
```

- 3-recursive (typically consists of initiation function, then a 2-recursive)

```
def maxer(smoke):
    # initiation function
    def fire(y):
        # 2-recursive parent
        def haze(z):
            # 2-recursize child
            # logic
            return fire(z);
        return haze
    return fire

def compose(n):
    # returns g(x) = f1(f2(…fn(x)…))
    if n == 1:
        return lambda f: f
    def call(f):
        def on(g):
            return compose(n – 1)(lambda x: f(g(x)))
        return on
    return call
```

Final Checks

- **`n // 10`** vs `n / 10`
- `float("-inf")`
- `print(str(i) + " this is a number")`
- `pow()` returns float when any argument is float, use `round()`
- `range(x, y)` is exclusive
- Check base cases (`[[]]` in 19 Summer)
- **`self`**`.var / label()` vs `label`
- Read doc tests and problem statement
- Check all methods and variables of objects (what's their purpose?)
- Did you remember to recursive call? (`dfs(*it)`)
- Check args and RV: Is the argument a list? What does function return? Boolean? (Midterm 2)
- `.append` and `.extend` returns `None`

- Multiple lambdas

```
def multiadder(n):
        # >>> multiadder(3)(5)(6)(7)
        # 18
        if n == 1:
                return lambda x: x
        else:
                # essentially lambda, since one argument given
                return lambda x: lambda y: multiadder(n – 1)(x + y)

zipper = lambda x: x
helper = lambda f, g: lambda x: f(g(x))
zipper = helper(f1, zipper)
```