CS61A FINAL "CHEAT" SHEET CS61A-1B HINA SAKAZAKI

Remember:
- Focus
- Look again and doubt your answers
- Env. diagrams.
  - → tuples & lists
  - → think indexing
  - → CAREFUL w/ LEXICAL/DYNAMIC

Use your brain!

when drawing env. diagrams, pay attention to nesting [ ]

lambda: variables defined in env. which procedure was defined / parent of frame is env. where it was called

→ NONLOCAL? looks up every frame in parent instead of new var. until global → error if cant find

```
def oracle(a,b,c):
    if a==42:
        return b(a)
    return c
```
if c errors → error.

ALWAYS KNOW WHAT FUNCTION TAKES IN AND RETURNS

like (lambda x, y: x**2 + y)(3,4) → 13

- seq.append(elem)
- seq.extend(seq)
- seq.remove(4) → removes '4' (first instance of it)
- seq.pop(i) → at index i, or last elem if ()

NEWTON'S METHOD / iter_improve:

- iter_improve takes in (update, done, guess=1, max_updates=1000)

```
k=0
while not done(guess) and k < max_updates:
    guess = update(guess)
    k += 1
return guess
```

- "___" improve takes in (a), to find ___ of a (like cube root of a)

```
def update(guess): return ___ update(guess, a)
def done(guess): return guess * guess * guess == a
return iter_improve(update, done) cube root
```

- approx_derivative (f, x, delta 1e-5)
- newton_update(f) → does thing w/ slopes, returns update fct.
- def find_root(f, guess=1):
  `return iter_improve(newton_update(f), lambda x: f(x)==0, guess)`
- square_root_newton(a) finds sq. root of a:
  `return find_root(lambda x: pow(x,2) - a)`

MT1:
```
def invert(f):
    def g(x):
        return find_root(lambda y: f(y) - x)
    return g
```

```
def cube_root(x):
    def f(y):
        return y*y*y - x
    return newtons_method(f)
```

```
def make_derivative(f, h=1e-5):
    def derivative(a):
        df = f(a+h) - f(a)
        return df/h
    return derivative
```
h≠0

Object Oriented Programming

```
>>> class Account(object):
        def __init__(self, holder):
            self.balance = 0
            self.holder = holder
        def deposit(self, amount):
            self.balance += amount
            return self.balance
        interest = 0.02
```

- class statement Account; when calling need Account().
- CONSTRUCTOR: `__init__`
- instance attributes: self.balance, self.holder
- like >>> a = Account('Hina') >>> a.balance 0 >>> a.balance = 4000 >>> a.deposit(300)
- methods on Account objects
- interest = 0.02 ← class attribute; can be changed >>> Account.interest = 0.07 >>> a.interest 0.07 ← changed

```
>>> class CheckingAccount(Account):
        deposit_charge = 1
        interest = 0.01
        def deposit(self, amount):
            return Account.deposit(self, amount + self.deposit_charge)
```
INHERITANCE

multiple inheritance:

```
>>> class AsSeenOnTV(CheckingAccount, SavingsAccount):
        def __init__(self, holder):
            self.holder = holder
            self.balance = 1
```

```
def take_all(self): # prints color every time it takes
    for i in range(len(self.skittles)):
        if self.skittles[i].color == color:
            skittle = self.skittle[i]
            self.skittles = self.skittles[i] + self.skittles[i+1:]
            return skittle
```

THINK ABOUT INHERITANCE!
DON'T MIX UP! WHAT SELF IS IT REFERRING TO?

Huge().medium(4)(5) problem
self refers to huge.

BTW... SLICING DOES NOT MUTATE original list

== vs is
```
ls = [1,2,3,4]
list = [1,2,3,4]
ls == list  T
ls is list  F
```

MapReduce
input # the tempest → mapper (line_count.py) 'line' 1 'line' 1 → sort → sorted → reducer (sum.py) 2768 → vals

Sets: unordered, no duplicates
```
>>> s = {3,2,1,4,4}
>>> s
{1,2,3,4}
```

```
def foo(lst):
    return [elem**2 for elem in lst if elem % 0 == 0]
```
map exp. / name / sequence / filter

Iters/Streams/Gen.

ITERATORS
- `__next__` method: calculates next vals, checks if any vals left
- `__iter__` method: returns another — if object has both `__next__` & `__iter__`, iter only returns self

GENERATORS
generators are special kind of Python iterator that uses YIELD (instead of return) ← contains an implicit `__next__` method. By definition, is a generator.

```
def generate_naturals():
    current = 0
    while True:
        yield current
        current += 1
>>> gen = gen_naturals()
>>> gen
<generator object ....>
>>> next(gen)
0
```

```
class Naturals():
    def __init__(self):
        self.current = 0
    def __iter__(self):
        while True:
            yield self.current
            self.current += 1
```
the nats = Naturals() is NOT iterator. it's an object. nats.iter is generator. doesn't need `__next__` b/c implicit

STREAMS
Streams are similar to rlists

```
def make_fib_stream():
    return fib_stream_generator(0,1)
def fib_stream_generator(a,b):
    def compute_rest():
        return fib_stream_generator(b, a+b)
    return Stream(a, compute_rest)
```

s.rest returns Stream(5, <...>)

usually → just have compute_rest → return func(s.rest)
like map_stream(fn, s.rest)
return Stream(fn(s.first), compute_rest)

compose (f, g): return lambda x: f(g(x))
repeated_in_stream(fn): def compute_rest(): return stream_map(lambda x: ..., repeated_in_stream(fn))

Recursion

Tail recursion: CONSTANT SPACE

```
(define (last s))
  (= (cdr s) nil)
  (car s)
  (last (cdr s)))
```

helper fcts are helpful

```
(define (reverse-iter s result)
  (if null? s) result
  (reverse-iter (cdr s) (cons (car s) result))))
```

```
def Remove_Vowels(s):
    if len(s)==0:
        return s
    return "" if s[0] in ('a','e',...) else
        s[0] + remove-Vowels(s[1:])
```

```
def merge(s1, s2):
    if len(s1)==0:
        return s2
    if len(s2)==0:
        return s1
    elif s1[0] < s2[0]:
        return [s1[0]] + merge(s1[1:], s2]
    else:
        return [s2[0]] + merge(s1, s2[1:])
```

r_lists:

r = rlist(1, rlist(2, rlist(3, empty_rlist)))

r.first, r[0] r.rest, r[1] → returns rlist obj. r.rest
r[1] returns 2

range = ending val - starting val
tuple(range(0,4)) → (0, 1, 2, 3)

gcd(a,b) == gcd(b, a%b)

```
def gcd-rec(a,b):
    if a<b: return gcd-rec(b, a)
    if a>b and not a%b==0:
        return gcd-rec(b, a%b)
    return b
```

```
def gcd-iter(a,b):
    if a<b:
        return gcd-iter(b,a)
    while a>b and not a%b==0:
        a,b = b, a%b
    return b
```

SCHEME LISTS: (1.2), valid list of scm tokens

EVAL/READ: Read-Eval-Print Loop.
- Read: tokenize & parse
- Eval: data structures into values
- Print: prints

scm> '(1 2 3)
Pair('quote', Pair(Pair(1, Pair(2, Pair(3, nil))), nil))

(cons (cons 1 2) (cons 3 (cons 4 5))) → ((1.2) 3 4 . 5)

scheme eval:
takes in (expr, env)
returns do_du_form or scheme-apply ↓ returns procedure

requires env for lookup

EVAL
- Base cases:
  - primitive values (#s)
  - look up values bound to symbols
- Recursive calls:
  - Eval (operands) of call exp.
  - Apply (op, arg)
  - Eval (sub-exp) of special forms

APPLY
- Base cases:
  - Built in primitive procedures
- Rec. calls:
  - Eval (body) of user defined proc.

new env. for user defined

returns apply-primitive, scheme_eval (procedure, — procedure)

→ mutually recursive. Eval required if call exp. encountered. Apply uses eval to evaluate operand exp. into args, and evaluate body of user-defined procedures. the recursion ends with language primitives.

PARSING reads string
Scheme_read
→ returns valid scheme tokens

read> 42
42
read> '(1 2 3)
(quote (1 2 3))
read> nil
()
read> '()
(quote ())
read> (1(2 3) (4(5)))
(1(2 3)(4 (5)))
read> (hi there cs.(student))
(hi there cs student)

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len-rlist(contents)
        elif message == 'getitem':
            return get-item(contents, val)
        elif message == 'push first':
            contents = make-rlist(val, contents)
        elif message == 'pop-first':
            f = first(contents)
            contents = rest(contents)
            return f
        elif message == 'str':
            return str(contents)
    return dispatch
```

mutable objects can change over execution of program

complex facts:
(fact ('conclusion')
('hypothesis1')
('hypothesis2'))

```
(define (insert n s)
  (define split s-t)
    (if (or (null? s) (< n (car s))) (list t s)
    (split (cdr s) (cons (car s) t))))
  (let ((s-and-t (split s nil)))
  (let ((bigger (car s-and-t)))
       (smaller-reversed (car (cdr s-and-t))))
  (reverse-iter smaller-reversed (cons n bigger)))))
```

```
def are-siblings(t, s1, s2):
    if is-leaf(t):
        return False
    else:
        childnames = [item.data(entry) for c in item.children]
        if s1 in childnames and s2 in childnames:
            return True
        else:
            for child in item.children:
                if are-siblings(child, s1, s2):
                    return True
            return False
```

calculator:
(+ 2 4 6 8) evals 5, apply: 1
(+ 2 (* 4 (- 6 8))) eval: 7, apply: 3
NOT SAME AS SCHEME

```
def safe-square(x):
    try:
        return x * x
    except TypeError:
        print('incorrect arg type')
```

Trees

Tree(3, Tree(4), Tree(-2, Tree(8), Tree(3)))

ROOT → 3 — nodes (point in tree w/value)
4, -2; 8, 3 — LEAF

```
def find-path(t, entry):
    if t is None or (t.is-leaf and t.entry != entry):
        return False
    elif t.entry == entry:
        return (entry,)
    else:
        left-path = find-path(t.left, entry)
        if left-path:
            return (t.entry,) + left-path
        right-path = find-path(t.right, entry)
        if right-path:
            return (t.entry,) + right-path
        return False
```

simple tree recursive fct:

```
def fib(n):
    if n==1:
        return 0
    elif n==2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

$\Theta(b^n)$

fib(4)
fib(3) fib(2)
fib(2) fib(1)

BST: Binary Search Trees

```
def right_is_bigger(bst):
    if not bst.left._is_empty and
       not bst.right.is_empty:
        return bst.right.size >
               bst.left.size
    return bst.left == empty_bst
```

```
def coerce(tree):
    "dispatch dict. repping tree"
    if tree is None: return None
    return {'entry': tree.entry, 'left': coerce(tree.left),
            'right': coerce(tree.right)}
```

Env. diagram Tips:
- when copying integers, just their values.
- if lists, just copy the arrow to point to that object (same thing)
- lambda - no binding
- nonlocal → no new val

when assign something like
a = fun(15)
then CALL a('sleepy')(15)
earlier goes in Global Frame

ORDERS OF GROWTH
- Tree recursion is usually exponential $\Theta(b^n)$ - recursive fibs
- Incrementing problem scales $R(n)$ by factor $\Theta(n^b)$ for... for... for...
- factorial(n):
  if n==0:
  return 1
  return factorial(n) + factorial(n-2) → $\Theta(n^2)$
- linear growth $\Theta(n)$ if just scale w/n
- fast-exp(b, n):
  if n==0:
  return 1 → Log $\Theta(\log n)$
  if n%2==0:
  return sq(fast-exp(n/2))
  else:
  return b * fast_exp(b, n-1)
- while with in while like def foo(n):
  while i<=n:
  sum += bar(n)
  i += 1
  return sum
  def bar(n):
  while ...
  $\Theta(n^2)$
- def subsets(n):
  if n==0:
  return [[]]
  else:
  result = subsets(n-1)
  for subset in result[:]:
  result.append([n]+subset)
  return result
  → $\Theta(n^{2^n})$