CS 61A Structure and Interpretation of Computer Programs
Summer 2014 Midterm 3 Solutions

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, and closed electronics, except two 8.5” × 11” cheat sheets, and The Environment Diagram Rules.
- Mark your answers ON THE EXAM ITSELF. Answers outside of the space allotted to problems will *not* be graded. If you are not sure of your answer you may wish to provide a *brief* explanation.

Full name	
SID	
Login	
TA & section time	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

0. (1 points) Your thoughts? What is something that makes you strong?

1. **(8 points) What will Python output?**

Include all lines that the interpreter would display. If it would display a function, then write Function. If it would cause an error, write Error. Assume that you have started Python 3 and executed the following. **These are entered into Python exactly as written.**

```
def mystery(x):
    if x[3:] or (x - 5):
        return x
    return 'Evil input'

class Laptop:
    laptops = []
    def __init__(self, battery):
        self.power = lambda: battery
        self.keyboard = 'qwerty'
        self.laptops = self.laptops + [ self.keyboard ]
    def turn_on(self):
        return self.power()
    def destroy(self):
        self.power = lambda: 'Destroyed!'

x = Laptop(lambda: 10)
```

Expression	Interactive Output
`print("Ducks are cool!")`	Ducks are cool!
`mystery('inc')`	Error
`mystery([1, 2, 3])`	Error
`mystery('detectives')`	'detectives'
`x.turn_on()`	Function
`x.laptops`	['qwerty']
`print(x.destroy())`	None
`Laptop.laptops`	[]
`x.turn_on()`	'Destroyed!'

2. (12 points) Environment Diagrams

(a) (6 pt) Fish are friends not food

Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* You may want to keep track of the stack on the left, but this is not required.

A complete answer will:

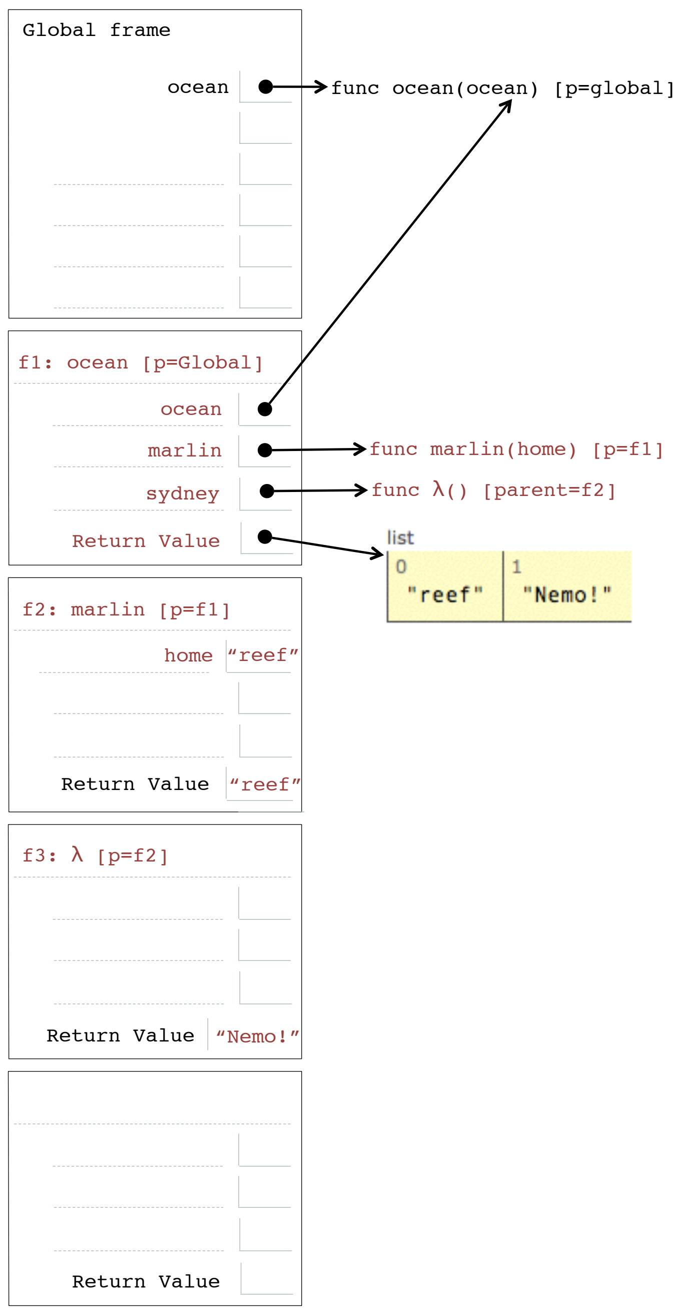
- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.
- The first function created by `lambda` should be labeled λ_1, the next one should be λ_2, and so on.

Remember that arguments are evaluated from left to right when creating a list.

```
def ocean(ocean):
    def marlin(home='reef'):
        nonlocal sydney
        sydney = lambda: 'Nemo!'
        return home
    sydney = 'city'
    return [marlin(), sydney()]
ocean(ocean)
```

Stack

global
f1
f2
f3

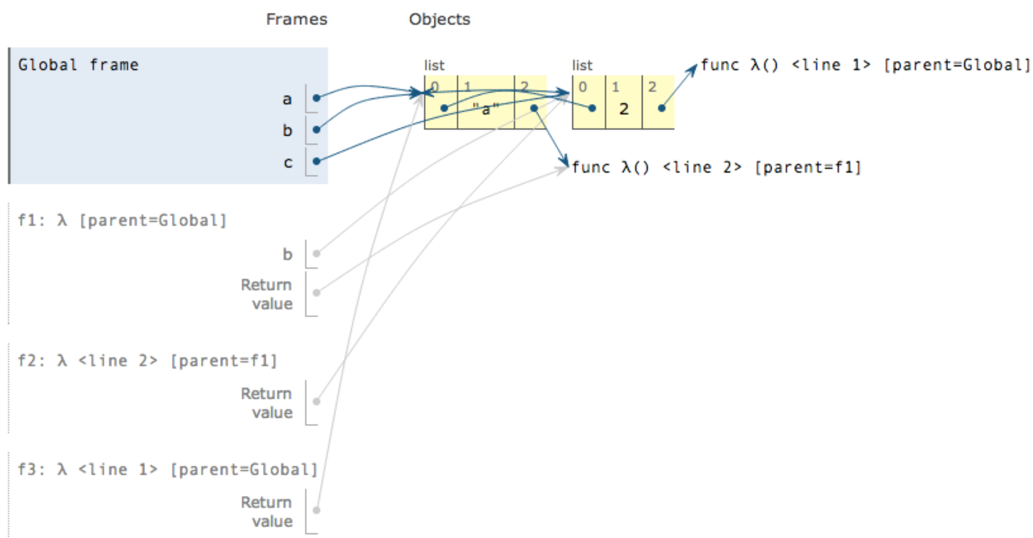


(b) (6 pt) Box and Pointer

Fill in the enviroment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution. *This may include more box-and-pointer diagrams.*
- Show the return value for each local frame.
- The first function created by `lambda` should be labeled λ_1, the next one should be λ_2, and so on.

```
a = [1, 2, lambda: a]
b = [a, 'a', (lambda b: lambda: b)(a)]
a = b[:]
b = a
c = a[2]()
c[0] = b[0][2]()
```



3. (6 points) Stealing All My Money

We are given the `Store` class below:

```
class Store:
    def __init__(self, products):
        self.products = products
        self.revenue = 0
    def sell(self, product):
        self.revenue += self.products[product]
```

Implement the `ComputerStore` and `AppleStore` classes. All `ComputerStore`s should enforce a tax of 10% on any product being sold, and all `AppleStore`s should add a `markup` of 100. *Make use of inheritance wherever possible.*

```
class ComputerStore(Store):
    """
    >>> ibm_store = ComputerStore()
    >>> ibm_store.products
    {'Phone': 220.0, 'Laptop': 770.0, 'Tablet': 550.0}
    >>> ibm_store.sell('Phone')
    >>> ibm_store.revenue
    220.0
    """
    electronics = {'Phone': 200, 'Laptop': 700, 'Tablet': 500}
    tax = 0.1

    def __init__(self):
        Store.__init__(self, self.electronics)
        after_tax = {}
        for product in self.products.keys():
            orig_cost = self.products[product]
            after_tax[product] = orig_cost * (1 + ComputerStore.tax)
        self.products = after_tax

class AppleStore(ComputerStore):
    """
    >>> apple_store = AppleStore()
    >>> apple_store.products
    {'iPhone': 320.0, 'iLaptop': 870.0, 'iTablet': 650.0}
    """
    markup = 100

    def __init__(self):
        ComputerStore.__init__(self)
        pricey = {}
        for product in self.products.keys():
            orig_cost = self.products[product]
            pricey['i' + product] = orig_cost + self.markup
        self.products = pricey
```

4. **(4 points) Interleave** Write a generator function that takes as input two iterators. It should return a generator that interleaves the elements returned by the two iterators. *It should yield* `'Done!'` *as soon as either of the two iterators runs out of elements.* Hint: You should not need any `for` loops in your solution.

```
def interleave(iter1, iter2):
    """
    >>> gen = interleave(iter(range(10)), iter(['s', 'u']))
    >>> next(gen)
    0
    >>> for elem in gen:
    ...   print(elem)
    ...
    s
    1
    u
    2
    Done!
    """

    try:
        while True:
            yield next(iter1)
            yield next(iter2)
    except StopIteration:
        yield 'Done!'
```

5. **(3 points) Orders of Growth**

(a) **(1 pt)** Considering the function definition shown below, what is the order of growth for a call to `spam(n)`?

```
def spam(n):
    if n == 3:
        return n
    return spam(n - 1) * spam(n - 1)
```

$\Theta(2^n)$

(b) **(1 pt)** Now consider the function definition shown below, what is the order of growth for a call to `foo(n)`?

```
def foo(n):
    if n < 0:
        return n
    elif n > 500:
        return foo(n - 2)
    return foo(n // 2)
```

$\Theta(n)$

(c) **(1 pt)** Now consider the function definition shown below, what is the order of growth for a call to `bar(n)`?

```
def bar(n):
    for i in range(n):
        lst = [j for j in range(i) if j != 1]
        if i == 3:
            bar(3)
```

$\Theta(n^2)$

6. (5 points) Higher Order Shuffling We would like to implement the `evens`, `odds` and `split` functions that work as follows:

```
>>> evens(['c', 's', 6, 1, 'a'])
['c', 6, 'a']
>>> odds(['c', 's', 6, 1, 'a'])
['s', 1]
>>> split(['c', 's', 6, 1, 'a'])
['c', 6, 'a', 's', 1]
>>> split(list(range(10)))
[0, 2, 4, 6, 8, 1, 3, 5, 7, 9]
```

You are given the following code:

```
evens = make_stepper(0, 2)
odds = make_stepper(1, 2)
split = adder(evens, odds)
```

Define the `make_stepper` and `adder` functions such that `evens`, `odds` and `split` work as expected. *Make sure that you consider domain and range.* Hint: You should use slicing in order to implement `make_stepper`.

```
def make_stepper(start, step):
    return lambda x: x[start::step]

def adder(f, g):
    return lambda x: f(x) + g(x)
```

7. (6 points) The Hexes Have Spread

Write the function `inhexing`, which takes in a Scheme list of numbers `lst`, a function `hex`, and an integer `n`, and returns a new list where every n^{th} element is replaced by the result of calling `hex` on that element. Note that this is *not* a deep list.

```
STk> (inhexing '(1 2 3 4 5) (lambda (x) 'poof!) 2)
(1 poof! 3 poof! 5)
STk> (inhexing '(2 3 4 5 6 7 8) (lambda (x) (+ x 10)) 3)
(2 3 14 5 6 17 8)

(define (inhexing lst hex n)

  (define (helper lst counter)
    (cond ((null? lst) '())
          ((= counter n)
           (cons (hex (car lst)) (helper (cdr lst) 1)))
          (else
           (cons (car lst) (helper (cdr lst) (+ counter 1))))))
  (helper lst 1))
```

8. (5 points) Tree Recursive Reductions

Part of the `Tree` class is given below:

```
class Tree:
    def __init__(self, datum, children=[]):
        self.datum = datum
        self.children = children
```

We want to write a `reduce` method for `Tree`s. The `reduce` method takes a function `fn` as an argument. **`fn` must be a function that takes in an element, a list of elements, and combines them in some way.** For example, in the first `reduce` doctest below, the function is supposed to be called 5 times (once for each node in the tree):

```
fn(2, [])         --> 2
fn(3, [])         --> 3
fn(5, [3])        --> 8
fn(6, [])         --> 6
fn(4, [2, 8, 6]) --> 20
```

Finish the implementation of the `reduce` method.

```
    def reduce(self, fn):
        """
        >>> t = Tree(4, [Tree(2), Tree(5, [Tree(3)]), Tree(6)])
        >>> t.pretty_print()
        4
        |__2
        |__5
        |  |__3
        |__6
        >>> t.reduce(lambda datum, child_sums: datum + sum(child_sums))
        20
        >>> t.reduce(lambda datum, child_results: [datum] + child_results)
        [4, [2], [5, [3]], [6]]
        """

        child_results = []
        for child in self.children:
            child_results.append(child.reduce(fn))
        return fn(self.datum, child_results)
```