MIDTERM REVIEW 6

COMPUTER SCIENCE 61A

July 10, 2014

1 What Would Python Output?

Consider the following definitions and assignments, and determine what Python would output for each of the calls below *if they were evaluated in order*.

```
>>> def andrew(rohin):
...     return lambda andrew: rohin(shah)
...
>>> def rohin(andrew):
...     return lambda rohin: andrew(huang)
...
>>> huang, shah = andrew, rohin
```

1. `>>> rohin("shah") == rohin("shah")`

   Solution: `False`

   With every call to `rohin`, Python will create a new lambda function. Although they may do the same thing, Python won't know that, and just duly notes that they're not the same function object.

2. `>>> andrew("Elephants are an abstraction")`

   Solution: `<function <lambda> at 0x...>`

3. `>>> andrew("Don't break")("abstraction barriers")`

Solution: `TypeError: 'str' object is not callable`

4. `>>> rohin(lambda x: x == andrew)("I love CS 61A!")`

Solution: `True`

2 Environment Diagrams

1.
```
fly = 100
def back(home):
    frog = lambda stroke: stroke + 200
    return frog

def im(free):
    fly = back
    turn = 200
    egerszergi = lambda turn: -turn + fly(400)(free)
    return egerszergi(turn)

free = im(fly)
```

Solution: See Python Tutor

2.
```
def cookie(m):
    return lambda a, b, c: m(a, b) * c

def cupcake():
    return 5

cookie(lambda a, b: a + b)(3, cupcake(), 4)
```

Solution: See Python Tutor

3 Higher Order Functions

1. Here is a simple cipher algorithm, called a *Caesar cipher*. Given a word, it will return the word with each letter shifted by a certain amount. Let us consider a Caesar cipher where you shift every letter by 2:

   ```
   >>> two shifter(’apple’) # We pretend two_shifter exists
   ’crrng’
   ```

   We get the result `’crrng’` because ”a” shifts over by two letters in the alphabet to get ”c”, ”p” shifts over by two letters to get ”r”, ”l” to ”n”, and ”e” to ”g”.

   One important part of the Caesar cipher is that it wraps around the alphabet:

   ```
   >>> two_shifter(’yoyo’)
   ’aqaq’
   ```

   So when we shift ”y”, we wrap around the alphabet and get ”a”.

   We are going to generalize a 2-shifter, since we want to be able to Caesar cipher by any amount. Write a function `make_caesar_cipher` that accepts as its argument the shift amount and returns a function that accepts a word that Caesar ciphers by that amount.

   You can use the following definitions to help you write your function as well.

   ```
   alphabet = ’abcdefghijklmnopqrstuvwxyz’

   def find_index(letter):
       index = 0
       for char in alphabet:
           if letter == char:
               return index
           index += 1
   ```

   Don’t forget that you’re also able to index into strings as you would lists:

   ```
   >>> alphabet[0]
   ’a’
   >>> alphabet[25]
   ’z’
   ```

```
def make_caesar_cipher(shift_amt):
    """Creates a Caesar cipher that shifts a word by a number
    of characters.

    >>> one_shifter = make_caesar_cipher(1)
    >>> one_shifter('apple')
    'bqqmf'
    >>> one_shifter('abcxyz')
    'bcdyza'
    >>> make_caesar_cipher(5)('applez')
    'fuuqje'
    """
```

Solution:

```
    def caesar_cipher(word):
        cipher = ''
        for char in word:
            new_index = find_index(char) + shift_amt
            cipher += alphabet[new_index % len(alphabet)]
        return cipher
    return caesar_cipher
```

4 Linked Lists and Recursion

1. Given `lst`, a flat linked list, return a new copy of the linked list with an element `el` inserted at index `n`. Assume that `n` is a valid index!

```
def insert_nth_place(lst, el, n):
    """
    >>> x = link(1, link(2, link(3)))
    >>> y = insert_nth_place(x, 42, 0)
    >>> print_linked_list(y)
    < 42 1 2 3 >
    >>> print_linked_list(insert_nth_place(y, 10, 2))
    < 42 1 10 2 3 >
    """
```

Solution:

```
    if n == 0:
        return link(el, lst)
    return link(first(lst), insert_nth_place(rest(lst),
                  el, n - 1))
```

2. Given a *deep* linked list, define `max_depth`, a function which returns the maximum number of nested lists inside the list. For example, the list `<<<<3>>>2>` would have a `max_depth` of 3, because the list `<<<3>>>` is nested inside the larger list. Elements in a flat list are all at depth 0. Assume you have a function `is_linked_list(obj)`, which returns `True` if `obj` is a linked list and `False` otherwise.

```
def max_depth(lst):
    """
    >>> x = link(link(1, link(link(2, empty), empty)),
                 link(3, link(4, empty)))
    >>> max_depth(x)
    2
    >>> y = link(1, link(2, link(3, empty)))
    >>> max_depth(y)
    0
    """
```

Solution:

```
    if lst == empty:
        return 0
    elif is_linked_list(first(lst)):
        return max(1 + max_depth(first(lst)),
                   max_depth(rest(lst)))
    return max_depth(rest(lst))
```

5 Data Abstraction

1. Given the following constructor and selector functions for linked lists,

```
def link(first, rest):
    return [first] + rest

def first(lst):
    return lst[0]

def rest(lst):
    return lst[1:]

empty = []
```

Find and correct the data abstraction violations:

```
def get_item(lst, n):
    while n > 0:
        lst, n = lst[1:], n - 1
    return first(lst)

def reverse(lst):
    new_lst = []
    while lst != []:
        new_lst = link(get_item(lst, 0), new_lst)
        lst = lst[1:]
    return new_lst

def append(lst1, lst2):
    if not lst1:
        return lst2
    return [lst1[0]] + append(rest(lst1), lst2))
```

Solution: Changes are in all caps.

```
def get_item(lst, n):
    while n > 0:
        lst, n = REST(LST), n - 1
    return first(lst)

def reverse(lst):
```

```
    new_lst = EMPTY
    while lst != EMPTY:
        new_lst = link(get_item(lst, 0), new_lst)
        lst = REST(LST)
    return new_lst

def append(lst1, lst2):
    if LST1 == EMPTY:
        return lst2
    return LINK(FIRST(LST1), APPEND(REST(LST1), LST2))
```

6 Sequences

1. Define a function common_prefix, which takes in two Python lists and returns a list of elements found at the beginning of each list.

```
def common_prefix(lst1, lst2):
    """
    Returns the common prefix of lst1 and lst2.
    >>> common_prefix([1, 2, 3, 4], [1, 2, 1, 4])
    [1, 2]
    >>> common_prefix([9, 8, 7, 6], [9, 8])
    [9, 8]
    """
```

Solution:

```
    prefix = []
    while lst1 and lst2 and lst1[0] == lst2[0]:
        prefix += [lst1[0]]
        lst1, lst2 = lst1[1:], lst2[1:]
    return prefix
```

7 Trees

Here's a particular implementation of the tree data structure. Remember that calling children on a tree returns *a Python list of trees*!

```
def tree(node, children=[]):
    def new_tree(dispatch):
        if dispatch == 'node':
            return node
        else:
            return children
    return new_tree

def datum(tree):
    return tree('node')

def children(tree):
    return tree('children')
```

1. Define a function `make_even` which takes in a `tree` of integers, and returns a new tree in which all the odd numbers are increased by 1 and all the even numbers remain the same.

```
def make_even(t):
    """
    >>> t = tree(1, [tree(2, [tree(3)]), tree(4, [tree(5)])])
    >>> print_tree(make_even(t))
      2
     / \
    2   4
    |   |
    4   6
    """
```

Solution:

```
    new_children = []
    for child in children(t):
        new_children += [make_even(child)]
    if datum(t) % 2 == 1:
        return tree(datum(t) + 1, new_children)
    return tree(datum(t), new_children)
```

8 Orders of Growth

1. What is the runtime of foo?

```
def foo(n):
    if n == 0:
        return True
    return foo(n // 2) or foo(n % 2)
```

Solution: $\Theta(\log n)$ - even though this function appears to be tree recursive, because `n // 2` is always evaluated first, the function will recurse down to `n == 0` and immediately return True because `or` short-circuits once it hits a true value.

2. What is the runtime of bottles?

```
def bottles(n):
    if n == 1:
        return 'on the wall'
    elif n >= 100:
        return bottles(99)
    return bottles(n - 1)
```

Solution: $\Theta(1)$ - the function bottles is called at *most* 99 times for any value of `n`. So even if we had called `bottles(1000000)`, it would only make 99 more calls to `bottles`.