CS 61A Challenge Problems: Midpoint Review

Solutions at https://alextseng.net/teaching/cs61a/

Alex Tseng

1 Recursion

(a) *Challenge* Define a function `parens` that takes in a positive integer `n` and returns the list of all possible nesting patterns of parentheses:
`parens(2) ---> ["()()", "(())"]`
`parens(3) ---> ["()()()", "()(())", "(())()", "(()())", "((()))"]`
Order in the list doesn't matter, but be wary of repeats.

(b) *Challenge* Given a string `s`, write a function `perms` that returns a list of all the possible permutations of characters. You may assume all characters are unique:
`perms("61a") ---> ["61a", "6a1", "16a", "a61", "1a6", "a16"]`
Order in the list doesn't matter.

(c) *Challenge* A tarsier spies a delicious cricket at the top of a ladder with `n` rungs. Due to a tarsier's short arms and legs, it can only climb 1 or 2 rungs at a time. How many ways can the tarsier climb the ladder to retrieve its reward? Define a function `climbs` to determine the answer:
`climbs(4) ---> 5 (1111, 12, 21, 22)`

(d) Recall Pascal's Triangle. Write a function `pascal` that takes in `i,j` and returns the entry of Pascal's Triangle in the `i`th row and `j`th entry in that row. `i` and `j` are both 0-indexed. It may help to look up the precise definition of the triangle:

```
pascal(4, 0) ---> 1
pascal(4, 2) ---> 6
pascal(4, 3) ---> 4
pascal(5, 3) ---> 10
```

2 Mutable Data and Functions

(a) Draw the environment diagram of the following:

```
def butter(fly):
    cater = 20
    pillar = 10
    def chrysalis(mystery):
        nonlocal cater
        cater = mystery(cater)
        pillar = mystery(fly)
        return [cater, pillar]
    return chrysalis
pupa = butter(4)
a = pupa(lambda x: x / 2)
b = pupa(lambda x: x / 2)
bugs = list([a, b]) + list(a)
```

(b) *Challenge* Write a function to compute the `n`th Fibonacci number. The first time it is called on `n`, it should take $\Theta(n)$ time. All later calls to the function for the same `n` should take $\Theta(1)$ time.
Hint: Use higher order functions and memoization.

3 Hierarchical Data Structures

As a reminder, here are the definitions of `Link` and `Tree`:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is Link.empty:
            rest = ""
        else:
            rest = ", " + repr(self.rest)
        return "Link({0}{1})".format(self.first, rest)

class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        self.branches = branches

    def is_leaf(self):
        return len(self.branches) == 0

    def __repr__(self):
        if self.branches:
            return "Tree({0}, {1})".format(self.entry, repr(self.branches))
        else:
            return "Tree({0})".format(repr(self.entry))
```

(a) Write a function `reverse_print` that prints out the elements of the linked list in reverse order. Do not create new Links or modify the existing list.

(b) *Challenge* Write a function `k_from_end` that returns the `k`th element from the end of a linked list. A value of `k = 0` corresponds to the last element. Try your best to do this in $\Theta(n)$ time where n is the length of the list.
`k_from_end(Link(1, Link(2, Link(3, Link(4)))), 2) ---> 2`

(c) *Challenge* Write a function `reverse_k` that reverses every `k` elements of a linked list *in place*. Assume that the length of the list is a perfect multiple of `k`:

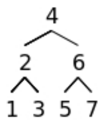
```
x = Link(1, Link(2, Link(3, Link(4, Link(5, Link(6, Link(7, Link(8, Link(9)))))))))
y = reverse_k(x, 3)
y ---> Link(3, Link(2, Link(1, Link(6, Link(5, Link(4, Link(9, Link(8, Link(7)))))))))
```

Note that since `reverse_k` changes the first link, it will return a pointer to the new head of the list. This means that `x` will no longer point to the head of the list. Instead, `x` will still be pointing to the 1.

(d) A binary search tree is a tree with two very special properties:

1. Each subtree (including the main one) has exactly 0 or 2 children.
2. For every subtree `t`, every element in `t`'s left branch is smaller than `t.entry`, and every element in `t`'s right branch is larger than `t.entry`.

Here is an example of a valid binary search tree. This example is particularly balanced, but it is not necessary that the depth is the same on each side.



Write a function `is_bst` that takes in a tree and returns whether or not it is a valid binary search tree. Hint: It may be helpful to keep track of the minimum and maximum allowable values for a subtree.