MUTABLE LINKED LISTS, MUTABLE TREES, INTERFACES, AND ITERATORS

COMPUTER SCIENCE 61A

July 25 to July 30, 2015

1 Mutable Linked Lists

1. Draw the box-and-pointer diagram for each of the following linked lists:

```
>>> delphine = Link(1, Link(2, Link(3, Link(4))))
>>> delphine.rest.first = 5
```

```
>>> joseph = Link(7, Link(11))
>>> delphine.rest.rest = joseph
```

```
>>> albert = delphine.rest.rest
>>> albert is joseph    # True or False?
```

```
>>> robert = Link(7, link(11))     # True or False?
>>> robert is joseph
```

2. Implement the `double_up` method for the `Link` class, which mutates a linked list by duplicating every element. See the doctest for an example:

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

    def double_up(self):
        """
        >>> john = Link(1, Link(3, Link(5)))
        >>> john.double_up()
        >>> john
        Link(1, Link(1, Link(3, Link(3, Link(5, Link(5))))))
        """
```

2 Mutable Trees

1. Implement `make_even`, which takes a `Tree` and mutates it in the following way: for each element,
   - if the element is even, leave it as is
   - if the element is odd, add 1 to it to make it even

```
class Tree:
    def __init__(self, entry, subtrees=[]):
        self.entry = entry
        self.subtrees = list(subtrees)
    def is_leaf(self):
        return not self.subtrees

def make_even(t):
```

3 Binary Trees and Binary Search Trees

1. How is a `BinaryTree` different from a `Tree`?

2. What is a binary search tree?

3. Implement `bst_to_sorted_list`, which takes a binary search tree and returns a list containing all of the elements of the binary search tree in sorted order.

```
class BinaryTree:
    empty = ()
    def __init__(self, entry, left=empty, right=empty):
        self.entry = entry
        self.left = left
        self.right = right

def bst_to_sorted_list(bst):
```

4 Interfaces

1. What is an interface? What is it in the context of OOP?

2. What is a Python magic method?

3. Implement the `__contains__` method for the `Tree` class. The `__contains__` method allows you to use the built-in **`in`** operator to check if an element is in your `Tree`.

```
class Tree:
    ...
    def __contains__(self, value):
```

5 Iterators and Generators

1. What is the difference between an iterable and an iterator?

2. What is a generator function?

3. Implement `every_other`, a generator function that takes an iterable and yields all of the even-indexed elements (0-based indexing).

```
def every_other(s):
    """
    >>> mystery = every_other('CASE 2601-A')
    >>> classy = ''
    >>> for letter in mystery:
    ...     classy += letter
    >>> classy
    'CS 61A'
    """
```

4. Implement `evens`, a generator function that takes an iterable of numbers and yields all of the elements that are even numbers.

```
def evens(s):
    """
    >>> appreciate = evens([2, 11, 6, 5, 4, 13, 8, 9])
    >>> for num in appreciate:
    ...     print(num)
    2
    6
    4
    8
    """
```