FINAL REVIEW 13

COMPUTER SCIENCE 61A

August 4, 2016

1 Mutable Sequences

1. Write a function that takes in two values `x` and `el`, and a list, and adds as many `el`'s to the end of the list as there are `x`'s.

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
```

Solution:

```
    count = 0
    for element in lst:
        if element == x:
            count += 1
    while count > 0:
        lst.append(el)
        count -= 1
```

2. Given a deep dictionary `d`, replace all occurences of `x` *as a value (not a key)* with `y`. Hint: You will need to combine iteration and recursion.

```
def replace_all_deep(d, x, y):
    """
    >>> d = {1: {2: 3, 3: 4}, 2: {4: 4, 5: 3}}
    >>> replace_all_deep(d, 3, 1)
    >>> d
    {1: {2: 1, 3: 4}, 2: {4: 4, 5: 1}}
    """
```

Solution:

```
    for key in d:
        if d[key] == x:
            d[key] = y
        elif type(d[key]) == dict:
            replace_all_deep(d[key], x, y)
```

2 Object-Oriented Programming

1. Assume these commands are entered in order. What would Python output?

```
>>> class Foo:
...     def __init__(self, a):
...         self.a = a
...     def garply(self):
...         return self.baz(self.a)
>>> class Bar(Foo):
...     a = 1
...     def baz(self, val):
...         return val
>>> f = Foo(4)
>>> b = Bar(3)
>>> f.a
```

Solution: `4`

```
>>> b.a
```

Solution: `3`

```
>>> f.garply()
```

Solution: `AttributeError: 'Foo'object has no attribute 'baz'`

```
>>> b.garply()
```

Solution: `3`

```
>>> b.a = 9
>>> b.garply()
```

Solution: `9`

```
>>> f.baz = lambda val: val * val
>>> f.garply()
```

Solution: `16`

3 Mutable Linked Lists and Trees

3.1 Linked Lists

Here is the implementation of the linked list class:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
    def __len__(self):
        return 1 + len(self.rest)
    def __repr__(self):
        if self.rest is Link.empty:
```

```
        return 'Link({})'.format(self.first)
    else:
        return 'Link({}, {})'.format(self.first,
                                     repr(self.rest))
```

1. Write a recursive function `flip_two` that takes as input a linked list `lnk` and mutates `lnk` so that every pair is flipped.

```
def flip_two(lnk):
    """
    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))
    """
```

Solution:

```
    if lnk == Link.empty or lnk.rest == Link.empty:
        return
    lnk.first, lnk.rest.first = lnk.rest.first, lnk.first
    flip_two(lnk.rest.rest)
```

3.2 Trees

```
class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children
```

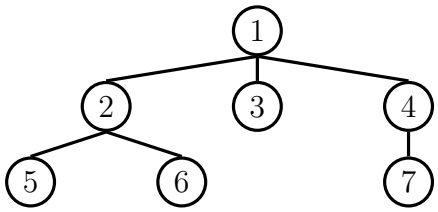
1. Assuming that every entry in `t` is a number, let's define `average(t)`, which returns the average of all the entries in `t`.

```
def average(t):
    """
    Returns the average value of all the entries in t.
    >>> t0 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
    >>> average(t0)
    1.5
    >>> t1 = Tree(8, [t0, Tree(4)])
    >>> average(t1)
    3.0
    """
```

Solution:

```
    def sum_helper(t):
        sum_entries, count = t.entry, 1
        for c in t.children:
            child_sum, child_count = sum_helper(c)
            sum_entries += child_sum
            count += child_count
        return sum_entries, count
    sum_entries, count = sum_helper(t)
    return sum_entries / count
```

2. Write a program `flatten` that given a Tree `t`, will return a linked list of the elements of `t`, ordered by level. Entries on the same level should be ordered from left to right. For example, the following tree will return the linked list `<1 2 3 4 5 6 7>`.




```
def flatten(t):
```

Solution:

```
        def flatten_helper(queue):
                if not queue:
                        return Link.empty
                curr = queue.pop(0)
```

```
            for c in curr.children:
                queue.append(c)
            return Link(curr.entry, flatten_helper(
                queue))
    return flatten_helper([t])
```

4 Scheme

1. Write a Scheme function that, when given an element, a list, and an index, inserts the element into the list at that index.

```
(define (insert element lst index)
```

Solution:

```
   (if (= index 0)
       (cons element lst)
       (cons (car lst) (insert element (cdr lst) (- index
          1))))
                                                          )
```

2. Define `deep-apply`, which takes a nested list and applies a given procedure to every element. `deep-apply` should return a nested list with the same structure as the input list, but with each element replaced by the result of applying the given procedure to that element. Use the built-in `list?` procedure to detect whether a value is a list. The procedure `map` has been defined for you.

```
(define (map fn lst)
  (if (null? lst)
    nil
    (cons (fn (car lst)) (map fn (cdr lst)))))
(define (deep-apply fn nested-list)
```

Solution:

```
  (if (list? nested-list)
      (map (lambda (x) (deep-apply fn x)) nested-list)
      (fn nested-list))
                                                        )
```

```
scm> (deep-apply (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
scm> (deep-apply (lambda (x) (* x x)) '(1 ((4) 5) 9))
(1 ((16) 25) 81)
scm> (deep-apply (lambda (x) (* x x)) 2)
4
```

4.1 Streams

1. What would Scheme display?

```
scm> (define (has-even? s)
       (cond ((null? s) False)
             ((even? (car s)) True)
             (else (has-even? (cdr-stream s)))))
has-even?
scm> (define ones (cons-stream 1 ones))
```

Solution:
```
ones
```

```
scm> (define twos (cons-stream 2 twos))
```

Solution:
```
twos
```

```
scm> ones
```

Solution:
```
(1 . #[promise (not forced)])
```

```
scm> (cdr ones)
```

Solution:
```
#[promise (not forced)]
```

```
scm> (cdr-stream ones)
```

Solution:
```
(1 . #[promise (forced)])
```

```
scm> (has-even? ones)
```

Solution:
```
# Runs forever
```

```
scm> (has-even? twos)
```

Solution:
```
True
```

5 Logic

1. Write facts for `match`, a relation between two lists if and only if the two lists are identical.

```
> (query (match (i am so cool) (i am . ?you)))
Success!
you: (so cool)
```

Solution:
```
(fact (match ?() ?()))
(fact (match (?f0 . ?r0) (?f0 . ?r1))
      (match ?r0 ?r1))

(query (match (i am so cool) (i am . ?you)))
```

6 Generators

1. Write a generator function that returns all subsets of the positive integers from 1 to `n`. Each call to this generator's `__next__` method will return a list of subsets of the set `[1, 2, ..., n]`, where `n` is the number of times `__next__` was previously called.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]
    """
```

Solution:

```
    subsets = [[]]
    n = 1
    while True:
        yield subsets
        subsets = subsets + [s + [n] for s in subsets]
        n += 1
```