CS 61A
Summer 2019

Final Review

Discussion 12: August 13, 2019

1 Mutation

1.1 For each row below, fill in the blanks in the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and expressions may affect later expressions.

```
>>> cats = [1, 2]
>>> dogs = [cats, cats.append(23), list(cats)]
>>> cats
```

```
>>> dogs[1] = list(dogs)
>>> dogs[1]
```

```
>>> dogs[0].append(2)
>>> cats
```

```
>>> cats[1::2]
```

```
>>> cats[:3]
```

```
>>> dogs[2].extend([list(cats).pop(0), 3])
>>> dogs[3]
```

```
>>> dogs
```

2 Recursion

2.1 (Adapted from Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns a list of paths, where each path is a list containing steps to reach `y` from `x` by repeated incrementing or doubling For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9, so one path is `[3, 4, 8, 9]`

```
def paths(x, y):
    """Return a list of ways to reach y from x by repeated
    incrementing or doubling.
    >>> paths(3, 5)
    [[3, 4, 5]]
    >>> sorted(paths(3, 6))
    [[3, 4, 5, 6], [3, 6]]
    >>> sorted(paths(3, 9))
    [[3, 4, 5, 6, 7, 8, 9], [3, 4, 8, 9], [3, 6, 7, 8, 9]]
    >>> paths(3, 3) # No calls is a valid path
    [[3]]
    """
    if _________________________:

        return ______________________________________________

    elif _______________________:

        return ______________________________________________

    else:

        a = _________________________________________________

        b = _________________________________________________

        return ______________________________________________
```

3 Trees

3.1 Implement `long_paths`, which returns a list of all *paths* in a tree with length at least `n`. A path in a tree is a linked list of node values that starts with the root and ends at a leaf. Each subsequent element must be from a child of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are listed in order from left to right. See the doctests for some examples.

```
def long_paths(tree, n):
    """Return a list of all paths in tree with length at least n.

    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])])])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    <0 1 2>
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 6 9>
    <0 11 12 13 14>
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 11 12 13 14>
    >>> long_paths(whole, 4)
    [Link(0, Link(11, Link(12, Link(13, Link(14)))))]
    """
```

4 Streams

4.1 Write a function merge that takes 2 sorted streams s1 and s2, and returns a new sorted stream which contains all the elements from s1 and s2.
Assume that both s1 and s2 have infinite length.

```
(define (merge s1 s2)


    (if ___________________________________________________________________________


        ___________________________________________________________________________


        ___________________________________________________________________________))
```

4.2 (Adapted from Fall 2014) Implement cycle which returns a stream repeating the digits 1, 3, 0, 2, and 4, forever. Write cons-stream only once in your solution!
Hint: `(3+2) % 5 == 0`.

```
(define (cycle start)


    _______________________________________________________________________________)
```

5 Generators

5.1 Implement `accumulate`, which takes in an `iterable` and a function `f` and yields each accumulated value from applying `f` to the running total and the next element.

```
from operator import add, mul

def accumulate(iterable, f):
    """
    >>> list(accumulate([1, 2, 3, 4, 5], add))
    [1, 3, 6, 10, 15]
    >>> list(accumulate([1, 2, 3, 4, 5], mul))
    [1, 2, 6, 24, 120]
    """
    it = iter(iterable)


    ______________________________________________________________________________


    ______________________________________________________________________________


    for __________________________________________________________________________:


        __________________________________________________________________________


        __________________________________________________________________________
```

5.2 Implement `sum_paths_gen`, which takes in a Tree instance `t` and and returns a generator which yields the sum of all the nodes from a path from the root of a tree to a leaf.

You may yield the sums in any order.

```
def sum_paths_gen(t):
    """
    >>> t1 = Tree(5)
    >>> next(sum_paths_gen(t1))
    5
    >>> t2 = Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(9)])
    >>> sorted(sum_paths_gen(t2))
    [6, 7, 10]
    """

    if ___________________________:

        yield ____________________

    for __________________________:

        for __________________________:

            yield ____________________
```

6 Macros

6.1 Using macros, let's make a new special form, `when`, that has the following structure:

```
(when <condition>
      (<expr1> <expr2> <expr3> ...))
```

If the condition is not false (a truthy expression), all the subsequent operands are evaluated in order and the value of the last expression is returned. Otherwise, the entire `when` expression evaluates to `okay`.

```
scm> (when (= 1 0) ((/ 1 0) 'error))
okay
scm> (when (= 1 1) ((print 6) (print 1) 'a))
6
1
a
```

(a) Fill in the skeleton below to implement this without using quasiquotes.

```
(define-macro (when condition exprs)

  (list 'if_______________________________________________________________________________))
```

(b) Now, implement the macro using quasiquotes.

```
(define-macro (when condition exprs)

  `(if ___________________________________________________________________________________))
```

6.2 Write a macro that takes in a call expression and strips out every other argument. The first argument is kept, the second is removed, and so on. You may find it helpful to write a helper function.

```
(define-macro (prune-expr expr)
```

```
scm> (prune-expr (+ 10))
10
scm> (prune-expr (+ 10 100))
10
scm> (prune-expr (+ 10 100 1000))
1010
scm> (prune-expr (prune-expr (+ 10 100) 'garbage))
10
```