CS 61A
Spring 2018

Final Review

Discussion 11: April 25, 2018

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


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


>>> dogs
```

1.2 (Fall 2013) Draw the environment diagram for the folowing code.

```
def miley(ray):
    def cy():
        def rus(billy):
            nonlocal cy
            cy = lambda: billy + ray
            return [1, billy]
        if len(rus(2)) == 1:
            return [3, 4]
        else:
            return [cy(), 5]
    return cy()[1]

billy = 6
miley(7)
```

2 Recursion

2.1 Write a procedure `merge(s1, s2)` which takes two sorted (smallest value first) lists and returns a single list with all of the elements of the two lists, in ascending order. Use recursion.

Hint: If you can figure out which list has the smallest element out of both, then we know that the resulting merged list will have that smallest element, followed by the merge of the two lists with the smallest item removed. Don't forget to handle the case where one list is empty!

```
def merge(s1, s2):
    """ Merges two sorted lists
    >>> merge([1, 3], [2, 4])
    [1, 2, 3, 4]
    >>> merge([1, 2], [])
    [1, 2]
    """
    if ____________________________________________________________________________:


        return s2


    elif ____________________________________________________________________________:


        return s1


    elif ____________________________________________________________________________:


        return ____________________________________________________________________


    else:


        return ____________________________________________________________________
```

2.2 Consider the subset sum problem: you are given a list of integers and a number k. Is there a subset of the list that adds up to k? For example:

```
>>> subset_sum([2, 4, 7, 3], 5)         # 2 + 3 = 5
True
>>> subset_sum([1, 9, 5, 7, 3], 2)
False
>>> subset_sum([1, 1, 5, -1], 3)
False
```

```
def subset_sum(seq, k):


    if ____________________________________________________________________________:


        return False


    elif ____________________________________________________________________________:


        return True


    else:


        return ____________________________________________________________________
```

3 Trees

3.1 Assuming that every value in `t` is a number, define `average(t)`, which returns the average of all the values in `t`. You may not need to use all the provided lines.

```
def average(t):
    """
    Returns the average value of all the nodes 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
    """
    def sum_helper(t):

        total, count = ____________________________________________________________


        for ____________________________________________________________________:


            ____________________________________________________________________


            ____________________________________________________________________


            ____________________________________________________________________


        return total, count


    total, count = ____________________________________________________________

    return total / count
```

4 Macros

4.1 Consider 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 subexpressions are evaluated in order and the value of the last expression is returned.
- Otherwise, the entire `when` expression evaluates to `okay`.

Create this new special form using a macro. You may do the parts below in any order.

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

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

  (list ____________________________________________________________________________))

scm> (do-when-form (= 1 0) ((/ 1 0) 'error))
okay
scm> (do-when-form (= 1 1) ((print 61) (print 'a) 'final-review))
61
a
final-review
```

(b) Now, implement the macro using quasiquotes.

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

  `(if ____________________________________________________________________________))

scm> (do-when-form (= 1 0) ((/ 1 0) 'error))
okay
scm> (do-when-form (= 1 1) ((print 61) (print 'a) 'final-review))
61
a
final-review
```

5 Streams

5.1 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 ___________________________________________________________________

        ___________________________________________________________________

        ___________________________________________________________________))
```

5.2 (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)

    ___________________________________________________________________)
```

6 Generators

6.1 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 ____________________________________________________________________________:


        ____________________________________________________________________________


        ____________________________________________________________________________
```

6.2 Write a generator function that yields functions that are repeated applications of a one-argument function `f`. The first function yielded should apply `f` 0 times (the identity function), the second function yielded should apply `f` once, etc.

```
def repeated(f):
    """
    >>> double = lambda x: 2 * x
    >>> funcs = repeated(double)
    >>> identity = next(funcs)
    >>> double = next(funcs)
    >>> quad = next(funcs)
    >>> oct = next(funcs)
    >>> quad(1)
    4
    >>> oct(1)
    8
    >>> [g(1) for _, g in
    ...  zip(range(5), repeated(lambda x: 2 * x))]
    [1, 2, 4, 8, 16]
    """


    g = ____________________________________________________________________

    while True:


        ____________________________________________________________________


        ____________________________________________________________________
```

6.3 Ben Bitdiddle proposes the following alternate solution. Does it work?

```
def ben_repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = lambda x: f(g(x))
```

7 SQL

7.1 You're starting a new job at an animal shelter, and you've been tasked with keeping track of all the cats that are up for adoption!

We'll start with an empty table:

```
CREATE TABLE cats(name, weight DEFAULT 1, notes DEFAULT "meow");
```

(a) What would SQL display?

```
sqlite> INSERT INTO cats(name) VALUES ("Tom"), ("Whiskers");
sqlite> SELECT * FROM cats;
```

```
sqlite> INSERT INTO cats VALUES
   ...>   ("Mittens", 2, "Actually likes shoes"),
   ...>   ("Rascal", 4, "Prefers to associate with dogs"),
   ...>   ("Magic", 2, "Expert at card games");
sqlite> SELECT * FROM cats ORDER BY weight, name;
```

```
sqlite> UPDATE cats SET notes = "A cat" WHERE notes = "meow";
sqlite> SELECT name FROM cats WHERE notes = "A cat";
```

(b) Cats of different weights require different quantities of food. We have the following table:

```
CREATE TABLE food AS
    SELECT 1 AS cat_weight, 0.5 AS amount UNION
    SELECT 2              , 2.5           UNION
    SELECT 3              , 4.0           UNION
    SELECT 4              , 4.5;
```

Write a query that calculates the total amount of food required to feed all the cats (this should work for any table of cats, not just the one we created above). In our example, we have two cats of weight 1, two cats of weight 2, and one cat of weight 4. The total food required is $2 \times 0.5 + 2 \times 2.5 + 1 \times 4.5 = 10.5$.

```
SELECT _____________________________________________________________________

  FROM _____________________________________________________________________

 WHERE _____________________________________________________________________;
```