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.

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

>>> dogs[1] = list(dogs)
>>> dogs[1]
[[1, 2, 23], None, [1, 2, 23]]

>>> dogs[0].append(2)
>>> cats
[1, 2, 23, 2]

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

>>> dogs
[[1, 2, 23, 2], [[1, 2, 23, 2], None, [1, 2, 23, 1, 3]], [1, 2, 23, 1, 3]]
```
1.2 (Fall 2013) Draw the environment diagram for the following code.

```python
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 \texttt{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.

\textbf{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!

\begin{verbatim}
def merge(s1, s2):
    """ Merges two sorted lists
    >>> merge([1, 3], [2, 4])
    [1, 2, 3, 4]
    >>> merge([1, 2], [])
    [1, 2]
    """
    if len(s1) == 0:
        return s2
    elif len(s2) == 0:
        return s1
    elif s1[0] < s2[0]:
        return [s1[0]] + merge(s1[1:], s2)
    else:
        return [s2[0]] + merge(s1, s2[1:])

if ____________________________________________________________________________:
    return s2
elif ____________________________________________________________________________:
    return s1
elif ____________________________________________________________________________:
    return ________________________________________________________________________
else:
    return ________________________________________________________________________
\end{verbatim}
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:

```python
>>> 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
```

```python
def subset_sum(seq, k):
    if ____________________________________________________________:
        return False
    elif ________________________________________________________:
        return True
    else:
        return ___________________________________________________

if len(seq) == 0:
    return False
elif k in seq:
    return True
else:
    return subset_sum(seq[1:], k - seq[0]) or \
    subset_sum(seq[1:], k)
```
3 Trees

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

```python
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

def sum_helper(t):
    total, count = t.label, 1
    for b in t.branches:
        b_total, b_count = sum_helper(b)
        total += b_total
        count += b_count
    return total, count

total, count = sum_helper(t)
return total / count
```
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.

```scheme
(define (merge s1 s2)
  (if _________________________________________________________________
      _________________________________________________________________
      _________________________________________________________________)

(define (merge s1 s2)
  (if (< (car s1) (car s2))
      (cons-stream (car s1) (merge (cdr-stream s1) s2))
      (cons-stream (car s2) (merge s1 (cdr-stream s2)))))
```

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:** 

```scheme
(define (cycle start)
  (cons-stream start (cycle (modulo (+ start 2) 5))))
```

```scheme
(define (cycle start)
  (cons-stream start (cycle (modulo (+ start 2) 5))))
```
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.

```python
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)

    total = next(it)
    yield total
    for element in it:
        total = f(total, element)
        yield total
```
5.2 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.

```python
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:
    __________________________________________________________________________
    __________________________________________________________________________

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

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

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

This solution does not work. The value \( g \) changes with each iteration so the bodies of the lambdas yielded change as well.
6 SQL

6.1 You’re trying to re-organize your music library! The table tracks below contains song titles and the corresponding album. Create another table tracklist with two columns: the album and a comma-separated list of all songs from that album in alphabetical order.

```
CREATE TABLE tracks AS
    SELECT "Human" AS title, "The Definition" AS album UNION
    SELECT "Simple and Sweet", "The Definition" UNION
    SELECT "Paper Planes", "Translations Through Speakers";

CREATE TABLE tracklist AS
    WITH songs(album, total) AS (
        ________________________________________________________________________________
    ),
    ________________________________________________________________________________ AS (  
        ________________________________________________________________________________
        ________________________________________________________________________________
        ________________________________________________________________________________
        ________________________________________________________________________________
    )
    SELECT _____________________________________________________________________________
    WHERE _____________________________________________________________________________;

CREATE TABLE tracklist AS
    WITH songs(album, total) AS (  
        SELECT album, COUNT(*) FROM tracks GROUP BY album
    ),
    listing(album, songs_so_far, last_song, num_songs_so_far) AS (  
        SELECT album, title, title, 1 FROM tracks UNION
        SELECT l.album, songs_so_far || ', ' || title, title, num_songs_so_far + 1
        FROM tracks AS t, listing AS l
        WHERE l.album = t.album AND title > last_song
    )
    SELECT l.album, l.songs_so_far
    FROM songs AS s, listing AS l
    WHERE s.album = l.album AND total = num_songs_so_far;
```
sqlite3> SELECT * FROM tracklist ORDER BY album;
The Definition|Human, Simple and Sweet
Translations Through Speakers|Paper Planes