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
```

[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]
```
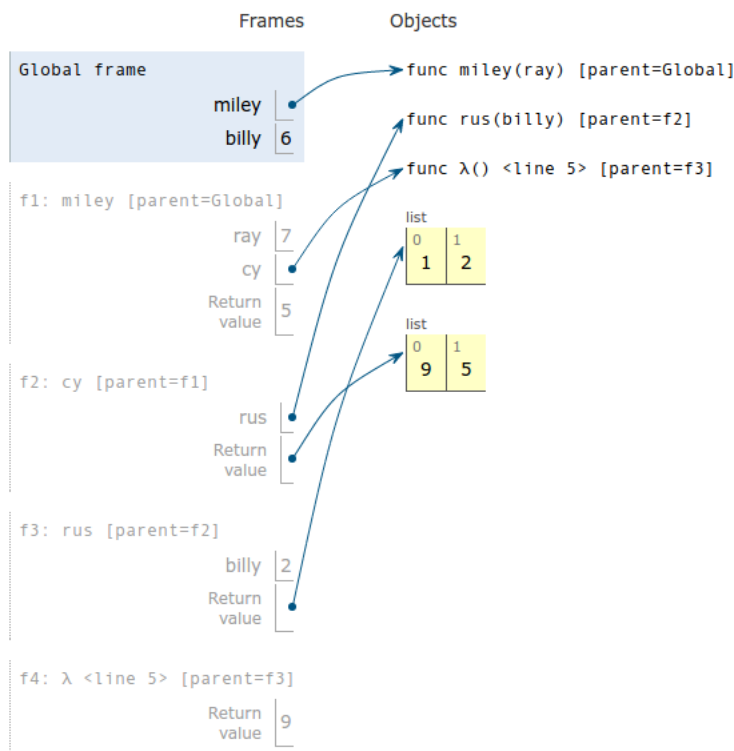
Index Error

```
>>> 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 folowing 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)
```

Frames   Objects

Global frame   → func miley(ray) [parent=Global]

    miley •

    billy 6   ↗ func rus(billy) [parent=f2]

               func λ() <line 5> [parent=f3]

f1: miley [parent=Global]

    ray 7

    cy •

    Return value 5

list
0	1
1	2

list
0	1
9	5

f2: cy [parent=f1]

    rus •

    Return value •

f3: rus [parent=f2]

    billy 2

    Return value •

f4: λ <line 5> [parent=f3]

    Return value 9

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 _____


    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:])
```

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 _____


    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 `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
```

Video walkthrough

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


(define-macro (do-when-form condition exprs)
       (list 'if condition (cons 'begin exprs) ''okay))
```

(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


(define-macro (do-when-form condition exprs)
  `(if ,condition ,(cons 'begin exprs) 'okay))
```

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 _____


        _____


        _____))
```

```
(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)))))
```

Video walkthrough

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)


    _____)
```

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

Video walkthrough

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.

```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)


    _____


    _____


    for _____:


        _____


        _____


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

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.

```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:


        _____


        _____
```

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

Video walkthrough

6.3   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 x: f(g(x))
```

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

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;


Tom|1|meow
Whiskers|1|meow

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;


Tom|1|meow
Whiskers|1|meow
Magic|2|Expert at card games
Mittens|2|Actually likes shoes
Rascal|4|Prefers to associate with dogs

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


Tom
Whiskers
```

(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 _____;

Specifying the table name in the WHERE clause here is not necessary and was added just for clarity.

SELECT SUM(amount) **FROM** cats, food **WHERE** cats.weight = food.cat_weight;