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   Implement a function to solve 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?

```python
def subset_sum(lst, k):
    """
    >>> subset_sum([], 0)
    True
    >>> subset_sum([], 4)
    False
    >>> 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
    """

    if _____:


        return True


    elif _____:


        return False


    else:


        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.

```python
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 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:


        _____


        _____
```

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

6 SQL

6.1 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:

```sql
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:

```sql
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 _____;

7   Macros

7.1   Write the `let` special form as a macro called `let-macro`. Recall that `let` takes in a list of bindings and a body expression. It creates a temporary frame containing the given bindings, and returns the result of evaluating the body in this temporary frame. Do not use the `let` special form in your solution.

You may use the provided `cadr` procedure in your solution.

Hint: The built-in `map` procedure takes in a one-argument function and a list and returns the result of mapping the function to every element in the list.

```
(define-macro (let-macro bindings body)




)

(define (cadr lst) (car (cdr lst)))

scm> (define x 3)
x
scm> (let-macro ((x 1) (y 2)) (+ x y))
3
scm> (let-macro ((x 2) (y x)) (* x y))
6
```

7.2   Write a macro called `zero-cond` that takes in a list of clauses, where each clause is a two-element list containing two expressions, a predicate and a corresponding result expression. All predicates evaluate to a number. The macro should evaluate each predicate and return the value of the expression corresponding to the first true predicate, *treating 0 as a false value.*

```
scm> (zero-cond
             ((0 'result1)
              ((- 1 1) 'result2)
              ((* 1 1) 'result3)
              (2 'result4)))
result3

(define-macro (zero-cond clauses)
    (cons 'cond

          (map        _____


                  _____


                  _____)))
```