1  Eval and Apply

1.1  Circle the number of calls to scheme_eval and scheme_apply for the code below.

\[\text{(define } (\text{square } x) (* x x))\]
\[\text{(+ } (\text{square } 3) (- 3 2))\]

<table>
<thead>
<tr>
<th>Calls to scheme_eval</th>
<th>2 5 14 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls to scheme_apply</td>
<td>1 2 3 4</td>
</tr>
</tbody>
</table>

2  Tail Calls

2.1  (a) What is tail context, tail calls, and tail recursive functions?

(b) Why are tail calls useful for recursive functions?

2.2  \(\text{(define } (\text{sum-list } \text{lst}))\)

\[\text{if } (\text{null? lst}) \quad \text{0}\]
\[\quad (\text{+ } (\text{car lst}) (\text{sum-list } (\text{cdr lst})))\]

(a) Why is sum-list not a tail call? Optional: draw out the environment diagram of this sum-list with list: (1 2 3). When do you add 2 and 3?

(b) Rewrite sum-list in a tail recursive context.
3 Iterators

3.1 What is difference between an iterator and an iterable?

3.2 Write an iterator that takes in a list and returns the sum of the list thus far.

```python
>>> accu = Accumulator([1, 2, 3, 4, 5, 6])
>>> for a in accu:
...    print(a)
1
3
6
10
15
21
```

3.3 Is this an iterator or an iterable or both?

3.4 Write Accumulator so it works if it takes in any iterable, not just a list
4 Generators

4.1 What does the following code block output?

```python
def foo():
a = 0
    if a < 10:
        print("Hello")
yield a
        print("World")
    for i in foo():
        print(i)
```

4.2 How can we modify `foo` so that `list(foo()) == [1, 2, 3, . . . , 10]`? (It’s okay if the program prints along the way.)
4.3 Define `hailstone_sequence`, a generator that yields the hailstone sequence. Remember, for the hailstone sequence, if \( n \) is even, we need to divide by two, otherwise, we multiply by 3 and add by 1.

```python
def hailstone_sequence(n):
    """
    >>> hs_gen = hailstone_sequence(10)
    >>> hs_gen.__next__()
    10
    >>> next(hs_gen) # equivalent to previous
    5
    >>> for i in hs_gen:
    >>>     print(i)
    16
    8
    4
    2
    1
    """
```

4.4 Define `tree_sequence`, a generator that iterates through a tree by first yielding the root value and then yielding each branch.

```python
def tree_sequence(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(5)]), Tree(3, [Tree(4)])])
    >>> print(list(tree_sequence(t)))
    [1, 2, 5, 3, 4]
    """
```