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)
   ```

2. How can we modify `foo` so that it satisfies the following doctests?
   ```python
>>> a = list(foo())
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```
3. Define `filter_gen`, a generator that takes in iterable `s` and one-argument function `f` and yields every value from `s` for which `f` returns `True`

```python
def filter_gen(s, f):
    """
    >>> list(filter_gen([1, 2, 3, 4, 5],
                      lambda x: x % 2 == 0))
[2, 4]
    >>> list(filter_gen((1, 2, 3, 4, 5), lambda x: x < 3))
[1, 2]
    """
```
4. Define `tree_sequence`, a generator that iterates through a tree by first yielding the root value and then yielding the values from each branch. Use the object-oriented representation of trees in your solution.

```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]
    """
```

5. **(Optional)** Write a generator that takes in a tree and yields each possible path from root to leaf, represented as a list of the values in that path. Use the object-oriented representation of trees in your solution.

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

1. What’s the advantage of using a stream over a scheme list?

2. What’s the maximum size of a stream?

3. What’s stored in the car and cdr of a stream? What are their types?

4. When is the next element actually calculated?
5. What Would Scheme Display?

(a) ```
    scm> (define (foo x) (+ x 10))
```  

(b) ```
    scm> (define bar (cons-stream (foo 1) (cons-stream (foo 2) bar)))
```  

(c) ```
    scm> (car bar)
```  

(d) ```
    scm> (cdr bar)
```  

(e) ```
    scm> (define (foo x) (+ x 1))
```  

(f) ```
    scm> (cdr-stream bar)
```  

(g) ```
    scm> (define (foo x) (+ x 5))
```  

(h) ```
    scm> (car bar)
```  

(i) ```
    scm> (cdr-stream bar)
```  

(j) ```
    scm> (cdr bar)
```
3 Code Writing for Streams

1. Implement \texttt{double-naturals}, which is a returns a stream that evaluates to the sequence 1, 1, 2, 2, 3, 3, etc.
   \begin{verbatim}
   (define (double-naturals)
     (double-naturals-helper 1 #f)
   )
   (define (double-naturals-helper first go-next)
   \end{verbatim}

2. Implement \texttt{interleave}, which returns a stream that alternates between the values in \texttt{stream1} and \texttt{stream2}. Assume that the streams are infinitely long.
   \begin{verbatim}
   (define (interleave stream1 stream2)
   \end{verbatim}
4 Tail Recursion

1. Consider the following function:
   
   ```scheme
   (define (count-instance lst x)
     (cond ((null? lst) 0)
           ((equal? (car lst) x) (+ 1 (count-instance (cdr lst) x)))
           (else (count-instance (cdr lst) x))))
   ```

   What is the purpose of `count-instance`? Is it tail recursive? Why or why not?
   Optional: draw out the environment diagram of this sum-list with \( \text{lst} = (1\ 2\ 1) \) and \( \text{x} = 1 \).
2. Rewrite count-instance to be tail recursive.

\texttt{(define (count-tail lst x)}

\texttt{)}

\texttt{)}

3. Implement \texttt{filter}, which takes in a one-argument function \texttt{f} and a list \texttt{lst}, and returns a new list containing only the elements in \texttt{lst} for which \texttt{f} returns true. Your function must be tail recursive.

You may wish to use the built-in \texttt{append} function, which takes in two lists and returns a new list containing the elements of the first list followed by the elements of the second.

\texttt{; Doctests}

\texttt{scm> (filter (lambda (x) (> x 2)) '(1 2 3 4 5))}

\texttt{(3 4 5)}

\texttt{(define (filter f lst)}

\texttt{)}

\texttt{)}