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)
```

**Solution:**
Hello
0
World
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]
```

**Solution:** Change the `if` to a `while` statement, and make sure to increment `a`. This looks like:

```python
def foo():
    a = 0
    while a < 10:
        a += 1
        yield a
```
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]
    """
    for x in s:
        if f(x):
            yield x
```

Solution:

```python
for x in s:
    if f(x):
        yield x
```
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]
    """
    yield t.label
    for branch in t.branches:
        for value in tree_sequence(branch):
            yield value

Alternate solution:
    yield t.label
    for branch in t.branches:
        yield from tree_sequence(branch)
```

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]]
    """
    if t.is_leaf():
        yield [t.label]
    for b in t.branches:
        for subpath in all_paths(b):
            yield [t.label] + subpath
```

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

**Solution:** Lazy evaluation. We only evaluate up to what we need.

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

**Solution:** Infinity

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

**Solution:** First is a value, rest is another stream encapsulated in a promise. The promise in the cdr of a stream may be forced (evaluated) or unforced (yet to be evaluated)

4. When is the next element actually calculated?

**Solution:** Only when it’s requested (and hasn’t already been calculated)
5. What Would Scheme Display?

(a) \texttt{scm> (define (foo x) (+ x 10))}

\textbf{Solution:} foo

(b) \texttt{scm> (define bar (cons-stream (foo 1) (cons-stream (foo 2) bar)))}

\textbf{Solution:} bar

(c) \texttt{scm> (car bar)}

\textbf{Solution:} 11

(d) \texttt{scm> (cdr bar)}

\textbf{Solution:} #\{promise (not forced)\}

(e) \texttt{scm> (define (foo x) (+ x 1))}

\textbf{Solution:} foo

(f) \texttt{scm> (cdr-stream bar)}

\textbf{Solution:} (3 . #\{promise (not forced)\})

(g) \texttt{scm> (define (foo x) (+ x 5))}

\textbf{Solution:} foo

(h) \texttt{scm> (car bar)}

\textbf{Solution:} 11

(i) \texttt{scm> (cdr-stream bar)}

\textbf{Solution:} (3 . #\{promise (not forced)\})
(j) scm> (cdr bar)

**Solution:** #[promise (forced)]
3 Code Writing for Streams

1. Implement double-naturals, which is a returns a stream that evaluates to the sequence 1, 1, 2, 2, 3, 3, etc.
   
   (define (double-naturals)
     (double-naturals-helper 1 #f))

   Solution:
   
   (define (double_naturals_helper first go-next)
     (if go-next
       (cons-stream first (double_naturals_helper (+ 1 first) #f))
       (cons-stream first (double_naturals_helper first #t))))

2. Implement interleave, which returns a stream that alternates between the values in stream1 and stream2. Assume that the streams are infinitely long.

   Solution:
   
   (define (interleave stream1 stream2)
     (cons-stream (car stream1)
       (interleave stream2 (cdr-stream stream1)))
   )

   (define (interleave stream1 stream2)
     (cons-stream (car stream1)
       (cons-stream (car stream2)
         (interleave (cdr-stream stream1) (cdr-stream stream2))))
   )
1. Consider the following function:

\[
\text{(define (count-instance lst x)}
\text{(cond ((null? lst) 0)}
\text{((equal? (car lst) x) (+ 1 (count-instance (cdr lst) x)))
\text{(else (count-instance (cdr lst) x)))))}
\]

What is the purpose of \text{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}.

\text{Solution: \text{count-instance} returns the number of time \text{x} appears in \text{lst}. It is not tail recursive. The call to \text{count-instance} appears as one of the arguments to a function call, so it will not be the final thing we do in every frame (we will have to apply + after evaluating it.)}
2. Rewrite count-instance to be tail recursive.
   (~def~ ?~COUNT-TAIL~ ~LST~ ?X~)

\[
\begin{array}{l}
\text{Solution:}\\
\text{(define (count-tail lst x)}\\
\hspace{1cm}\text{(define (count-helper lst x instances)}\\
\hspace{2cm}\text{(cond ((null? lst) instances)}\\
\hspace{3cm}((equal? (car lst) x) (count-helper (cdr lst) x (+ instances 1)))\\
\hspace{3cm}~\text{else~(count-helper (cdr lst) x instances)))))}\\
\hspace{1cm}(count-helper lst x 0))
\end{array}
\]
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}

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

(define (filter f lst)
    ;; Solution:

(define (filter f lst)
    (define (filter-tail f lst so-far)
        (cond ((null? lst) so-far)
              ((f (car lst)) (filter-tail f (cdr lst)
                                (append so-far (list (car lst))))))
              (else (filter-tail f (cdr lst) so-far)))
    (filter-tail f lst nil))

\end{verbatim}