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 `list(foo()) == [1, 2, 3, . . . , 10]`? (It's okay if the program prints along the way.)

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

```
def foo():
    a = 0
    while a < 10:
        a += 1
        yield a
```
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
    """

    while n != 1:
        yield n
        if n % 2 == 0:
            n = n // 2
        else:
            n = n*3 + 1
        yield n
```
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.

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]
    """

    Solution:
    yield t.label
    for branch in t.branches:
        for value in tree_sequence(branch):
            yield value
1. What’s the advantage of using a stream over a linked 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 first and rest? What are their types?

Solution: First is a value, rest is another stream (either a method to calculate it, or an already calculated stream). In the case of Scheme, this is called a promise.

4. When is the next element actually calculated?

Solution: Only when it’s requested (and hasn’t already been calculated)
5. For each of the following lines of code, write what Scheme would output.

```scheme
(scm> (define x 1))

Solution: x
```

```scheme
(scm> (if 2 3 4))

Solution: 3
```

```scheme
(scm> (define p (delay (+ x 1))))

Solution: p
```

```scheme
(scm> p)

Solution: #[promise]
```

```scheme
(scm> (force p))

Solution: 2
```

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

Solution: foo
```

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

Solution: bar
```
scm> (car bar)

**Solution:** 11

scm> (cdr bar)

**Solution:** #'[promise]

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

**Solution:** foo

scm> (cdr-stream bar)

**Solution:** (3 . #'[promise])

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

**Solution:** foo

scm> (car bar)

**Solution:** 11

scm> (cdr-stream bar)

**Solution:** (3 . #'[promise])
6. Write out `double_naturals`, which is a stream that evaluates to the sequence 1, 1, 2, 2, 3, 3, etc.

```scheme
(define (double_naturals)
  (double_naturals_helper 1 0))

(define (double_naturals_helper first go-next)
  (if (= 1 go-next)
      (cons-stream first (double_naturals_helper (+ 1 first) 0))
      (cons-stream first (double_naturals_helper first 1))))
```

; Alternative Solutions
```scheme
(define (double_naturals_helper first go-next)
  (cons-stream first (double_naturals_helper (+ go-next first) (- 1 go-next))))
```
7. Write out `interleave`, which returns a stream that alternates between the values in `stream1` and `stream2`. Assume that the streams are infinitely long.

```scheme
(define (interleave stream1 stream2)
  (cons-stream (car stream1)
               (interleave stream2 (cdr-stream stream1))))

; Alternative solution
(define (interleave stream1 stream2)
  (cons-stream (car stream1)
               (cons-stream (car stream2)
                            (interleave (cdr-stream stream1)
                                        (cdr-stream stream2))))
)
```
8. (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: