GENERATORS AND STREAMS

COMPUTER SCIENCE MENTORS 61A

April 16 to April 18, 2018

1 Iterators and Generators

1. What does the following code block output?

```
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())== [0, 1, 2, . . . , 9]`? (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:
        yield a
                                a += 1
```

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.

```
def hailstone_sequence(n):
    """
    >>> hs_gen = hailstone_sequence(10)
    >>> next(hs_gen)
    10
    >>> next(hs_gen)
    5
    >>> for i in hs_gen:
            print(i)
    16
    8
    4
    2
    1
    """
```

Solution:

```
    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
```

Alternate solution using yield from:

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

2 Streams

1. What are the differences between streams and scheme lists? What's the advantage of using a stream over a linked list?

   Solution: Scheme lists have another list as their second element, streams have another stream as the second argument to their constructor. Streams also wrap their second element in a promise. This means that in order to calculate the second element, we have to force that promise. Therefore, streams are lazily evaluated, and we need to use cons-stream in order to access the second element (instead of just being able to use cons, which do not force a promise). Streams are good because of lazy evaluation. We only evaluate up to what we need at any given point, allowing us to change the environment or save space between calculating elements.

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

   Solution: Infinity

3. When is the next element actually calculated?

   Solution: Only when it's requested (and hasn't already been calculated)

4. What Would Scheme Display?

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

Solution: `x`

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

Solution: `p`

(c) `scm> p`

Solution: `#[promise (unforced)]`

(d) `scm> (force p)`

Solution: `2`

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

Solution: `foo`

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

Solution: `bar`

(g) `scm> (car bar)`

Solution: `11`

(h) `scm> (cdr bar)`

Solution: `#[promise (unforced)]`

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

Solution: `foo`

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

Solution: `(3 . #[promise (unforced)])`

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

Solution: `foo`

(l) `scm> (car bar)`

Solution: `11`

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

Solution: `(3 . #[promise (unforced)])`

3 Code Writing for Streams

1. Implement `double_naturals`, which is 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 (eq? #t 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))))
)
```

4 Challenge Question

1. **(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.

```
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 _______________:

        yield ___________________

    for ______________________:

        for ________________________________:

            ___________________________________
```

Solution:

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