Instructions

Please download lab materials lab10.zip from our QQ group if you don't have one.

In this lab, you have two tasks:

- Think about what would Scheme display in section 3 and verify your answer with Ok.
- Complete the required problems described in section 4 and submit your code to our OJ website.

The starter code for these problems is provided in `lab10.scm`.

Submission: As instructed before, you need to submit your work with Ok by `python ok --submit`. You may submit more than once before the deadline, and your score of this assignment will be the highest one of all your submissions.

Readings: You might find the following references to the textbook useful:

- Section 3.2

Review

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the next section and refer back here when you get stuck.

Streams

In Python, we can use iterators to represent infinite sequences (for example, the generator for all natural numbers). However, Scheme does not support iterators.

Let's see what happens when we try to use a Scheme list to represent an infinite sequence of natural numbers:

```
scm> (define (naturals n)
       (cons n (naturals (+ n 1))))
naturals
scm> (naturals 0)
Error: maximum recursion depth exceeded
```

Because `cons` is a regular procedure and both its operands must be evaluted before the pair is constructed, we cannot create an infinite sequence of integers using a Scheme list. Instead, our Scheme interpreter supports *streams*, which are lazy Scheme lists. The first element is represented explicitly, but the rest of the stream's elements are computed only when needed. Computing a value only when it's needed is also known as *lazy evaluation*.

```
scm> (define (naturals n)
       (cons-stream n (naturals (+ n 1))))
naturals
scm> (define nat (naturals 0))
nat
scm> (car nat)
0
scm> (cdr nat)
#[promise (not forced)]
scm> (car (cdr-stream nat))
1
scm> (car (cdr-stream (cdr-stream nat)))
2
```

We use the special form `cons-stream` to create a stream:

```
(cons-stream <operand1> <operand2>)
```

`cons-stream` is a special form because the second operand is not evaluated when evaluating the expression. To evaluate this expression, Scheme does the following:

1. Evaluate the first operand.
2. Construct a promise containing the second operand.
3. Return a pair containing the value of the first operand and the promise.

To actually get the rest of the stream, we must call `cdr-stream` on it to force the promise to be evaluated. Note that this argument is only evaluated once and is then stored in the promise; subsequent calls to `cdr-stream` returns the value without recomputing it. This

allows us to efficiently work with infinite streams like the naturals example above. We can see this in action by using a non-pure function to compute the rest of the stream:

```
scm> (define (compute-rest n)
       (print 'evaluating!)
       (cons-stream n nil))
compute-rest
scm> (define s (cons-stream 0 (compute-rest 1)))
s
scm> (car (cdr-stream s))
evaluating!
1
scm> (car (cdr-stream s))
1
```

Here, the expression `(compute-rest 1)` is only evaluated the first time `cdr-stream` is called, so the symbol `evaluating!` is only printed the first time.

When displaying a stream, the first element of the stream and the promise are displayed separated by a dot (this indicates that they are part of the same pair, with the promise as the `cdr`). If the value in the promise has not been evaluated by calling `cdr-stream`, we consider it to be not forced. Otherwise, we consider it forced.

```
scm> (define s (cons-stream 1 nil))
s
scm> s
(1 . #[promise (not forced)])
scm> (cdr-stream s) ; nil
()
scm> s
(1 . #[promise (forced)])
```

Streams are very similar to Scheme lists in that they are also recursive structures. Just like the `cdr` of a Scheme list is either another Scheme list or `nil`, the `cdr-stream` of a stream is either a stream or `nil`. The difference is that whereas both arguments to `cons` are evaluated upon calling `cons`, the second argument to `cons-stream` isn't evaluated until the first time that `cdr-stream` is called.

Here's a summary of what we just went over:

- `nil` is the empty stream
- `cons-stream` constructs a stream containing the value of the first operand and a promise to evaluate the second operand
- `car` returns the first element of the stream
- `cdr-stream` computes and returns the rest of stream

What Would Scheme Display?

As you work through these problems, remember that streams have two important components:

- Lazy evaluation – so the remainder of the stream isn't computed until explicitly requested.
- Memoization – so anything we compute won't be recomputed.

The examples here stretch these concepts to the limit. In most practical use cases, you may find you rarely need to redefine functions that compute the remainder of the stream.

Use Ok to unlock the following "What would Scheme print?" questions:

```
$ python ok -q wwsd -u
```

```
scm> (define (has-even? s)
       (cond ((null? s) #f)
             ((even? (car s)) #t)
             (else (has-even? (cdr-stream s)))))
has-even?
scm> (define (f x) (* 3 x))
f
scm> (define nums (cons-stream 1 (cons-stream (f 3) (cons-stream (f 5) nil))))
nums
scm> nums

scm> (cdr nums)

scm> (cdr-stream nums)

scm> nums

scm> (define (f x) (* 2 x))
f
scm> (cdr-stream nums)

scm> (cdr-stream (cdr-stream nums))

scm> (has-even? nums)
```

Required Problems

In this section, you are required to complete the problems below and submit your code to OJ website as instructed in lab00 to get your answer scored.

Remember, you can use `ok` to test your code:

```
$ python ok  # test functions
$ python ok -q <function>  # test single function
```

Problem 1: Fix the Bug II (100pts)

Write a procedure `filter-stream`, which takes a predicate `f` and a stream `s`, and returns a new stream containing only elements of the stream that satisfy the predicate. The output should contain the elements in the same order that they appeared in the original stream. You can assume there are as least one element satisfies `f` if `s` is an infinite stream.

Look at the following implementation of `filter-stream`, what's wrong with it?

Note: We do **NOT** provide ok tests for this problem, but you can write your own tests in `tests/filter-stream.py`.

```
(define (filter-stream f s)
  (if (null? s) nil
      (let ((rest (filter-stream f (cdr-stream s))))
        (if (f (car s))
            (cons-stream (car s) rest)
            rest))))
```

Problem 2: Slice (100 pts)

Write a function `slice` which takes in a stream `s`, a non-negtive integer `start`, and a non-negtive integer `end`. It should return a Scheme list that contains the elements of `s` between index `start` and `end`, not including end. If the stream ends before `end`, you can return the whole part after `start`; if the stream ends before `start`, return `nil`. You can assume `end` may not less than `start`.

```
(define (slice s start end)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (define nat (naturals 0)) ; See naturals procedure defined earlier
; nat
; scm> (slice nat 4 12)
; (4 5 6 7 8 9 10 11)
; scm> (define a (cons-stream 1 (cons-stream 2 nil)))
; a
; scm> (slice a 1 114514)
; (2)
```

Problem 3: Make Some Streams (300pts in total)

Since streams only evaluate the next element when they are needed, we can combine infinite streams together for interesting results! Use it to define a few of our favorite sequences. We've defined the function `combine-with` for you below, as well as an example of how to use it to define the stream of even numbers.

```
(define (combine-with f xs ys)
  (if (or (null? xs) (null? ys))
      nil
      (cons-stream
        (f (car xs) (car ys))
        (combine-with f (cdr-stream xs) (cdr-stream ys)))))
```

```
scm> (define evens (combine-with + (naturals 0) (naturals 0)))
evens
scm> (slice evens 0 10)
(0 2 4 6 8 10 12 14 16 18)
```

For these problems, you may use the `naturals` stream in addition to `combine-with`. **We highly recommend you do not define any other helper functions.** Such problems are likely to appear on final exam.

Problem 3.1: Factorials (100 pts)

For this problem, you may use the `naturals` stream in addition to `combine-with` defined above. **We highly recommend you do not define any other helper functions.** Such problems are likely to appear on final exam.

Write `factorials`, which is a stream where the `n`th term represents $n!$.

```
(define factorials
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (slice factorials 0 10)
; (1 1 2 6 24 120 720 5040 40320 362880)
```

Problem 3.2: Fibonacci (100 pts)

For this problem, you may use the `naturals` stream in addition to `combine-with` defined above. **We highly recommend you do not define any other helper functions.** Such problems are likely to appear on final exam.

Write `fibs`, which is a stream where the `n`th term represents the `n`th Fibonacci number.

```
(define fibs
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (slice fibs 0 10)
; (0 1 1 2 3 5 8 13 21 34)
```

Problem 3.3: Exp (100 pts)

For this problem, you may use the `naturals`, `combine-with`, `factorials` defined above and built-in function `expt`. **We highly recommend you do not define any other helper functions.** Such problems are likely to appear on final exam.

Write function `exp`, which returns a stream where the `n`th term represents the degree-n polynomial expantion for e^x, which is $\sum_{i=0}^{n} \frac{x^i}{i!}$.

```
(define (exp x)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (slice (exp 2) 0 5)
; (1 3 5 6.333333333 7)
```

Problem 4: Non Decreasing (100 pts)

Define a function `nondecrease`, which takes in a scheme stream of numbers and outputs a stream of lists, which overall has the same numbers in the same order, but grouped into lists that are non-decreasing.

For example, if the input is a stream containing elements

```
(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1)
```

the output should contain elements

```
((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1))
```

Your solution may handle infinite streams correctly, which means if an infinite streams is always non-decreasing after the `n`th element, and from the `0`th to the `n - 1`th element can group to `m` non-decreasing sublists, your solution can output the first `m` sublists correctly.

```
(define (nondecrease s)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (define s (list-to-stream '(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1))) ; a
helper function to make stream from list
; s
; scm> (slice (nondecrease s) 0 8)
; ((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1))
```

Just for fun Problems

This section is out of scope for our course, so the problems below is optional. That is, the problems in this section **don't** count for your final score and **don't** have any deadline. Do it at any time if you want an extra challenge!

To check the correctness of your answer, you can submit your code to ok.

Problem 5: My `cons-stream` (optional, 0pts)

Define `my-cons-stream`, which returns a pair of elements, where the second element is not evaluated until `my-cdr-stream` is called on it. Also define the procedure `my-car` and `my-cdr-stream`, which take in a stream returned by `my-cons-stream` and return the first element and the result of evaluating the second element in the stream pair, respectively.

Unlike the streams we've seen in lecture, if you repeatedly call `my-cdr-stream` on a stream returned by `my-cons-stream`, you may evaluate an expression multiple times.

```
(define (my-cons-stream first second) ; Does this line need to be changed?
  'YOUR-CODE-HERE
)

(define (my-car stream)
  'YOUR-CODE-HERE
)

(define (my-cdr-stream stream)
  'YOUR-CODE-HERE
)

;;; Tests
; scm> (define a (my-cons-stream 1 (my-cons-stream (print 2) nil)))
; a
; scm> (my-car a)
; 1
; scm> (define b (my-cdr-stream a))
; 2
; b
; scm> (define c (my-cdr-stream a))
; 2
; c
; scm> (my-cdr-stream b)
; ()
```

Problem 6: Primes (optional, 0pts)

We can even represent the sequence of all prime numbers as an infinite stream! Define a function `sieve`, which takes in a stream of increasing numbers and returns a stream containing only those numbers which are not multiples of an earlier number in the stream. We can define `primes` by sifting all natural numbers starting at 2. Look online for the Sieve of Eratosthenes if you need some inspiration.

Hint: You might find using `filter-stream` as defined earlier helpful.

```
(define (sieve s)
  'YOUR-CODE-HERE
)

(define primes (sieve (naturals 2)))

;;; Tests
; scm> (slice primes 0 10)
; (2 3 5 7 11 13 17 19 23 29)
```