

## CS61A Notes – Week 11: Streams (solutions)

### Streaming Along

#### QUESTIONS

1. Define a procedure (`ones`) that, when run with no arguments, returns a cons pair whose `car` is 1, and whose `cdr` is a procedure that, when run, does the same thing.

```
(define (ones) (cons 1 (lambda () (ones)))) , or, just
(define (ones) (cons 1 ones))
```

2. Define a procedure (`integers-starting n`) that takes in a number `n` and, when run, returns a cons pair whose `car` is `n`, and whose `cdr` is a procedure that, when run with no arguments, does the same thing for `n+1`.

```
(define (integers-starting n)
  (cons n (lambda () (integers-starting (+ n 1)))))
```

---

### Constructing Streams

QUESTIONS: Describe what the following expressions define.

1. `(define s1 (add-stream (stream-map (lambda (x) (* x 2)) s1) s1))`

Infinite loop! We didn't specify a first element. Even the define statement will go into an infinite loop.

2. `(define s2
 (cons-stream 1
 (add-stream (stream-map (lambda (x) (* x 2)) s2) s2)))`

```
1
+ 2 6 18 ...
  1 3 9 ...
=====
1 3 9 27 ... powers of 3
```

3. `(define s3
 (cons-stream 1
 (stream-filter (lambda (x) (not (= x 1))) s3)))`

Infinite loop! `stream-filter` will keep trying to look for a number that's not 1. Or, more specifically, `stream-filter`, failing to find a non-1 element in `stream-car`, will call `stream-filter` again, which will call `stream-filter` again, and so on.

4. `(define s4
 (cons-stream 1
 (cons-stream 2
 (stream-filter (lambda (x) (not (= x 1))) s4))))`

```
(1 2 2 2 2...)
```

Rather counter-intuitive, but...well, we know that it starts with 1 and 2, since we said so. Then, the `stream-cdr` will be a stream that is produced by the `stream-filter`. `stream-filter` returns a stream whose first element is the first non-1 element of `s4` (namely, 2), and whose promise is `(stream-filter pred? (stream-cdr s))`, where `pred?` is the lambda, and `s` is `s4`. What's `(stream-cdr s4)`? Well, it's a stream containing the element 2 and a promise to evaluate `(stream-filter pred? s4)`. And we already know what that returns - a stream starting with 2, with a promise to evaluate `(stream-filter pred? (stream-cdr s))`, etc.

5. `(define s5  
 (cons-stream 1  
 (add-streams s5 integers)))`

```

1
  1 2 4 7 ...
+  1 2 3 4 ...
=====
1 2 4 7 11 ... starting from 1, add 1, 2, 3, etc.

```

6. Define facts without defining any procedures; the stream should be a stream of 1!, 2!, 3!, 4!, etc. More specifically, it returns a stream with elements (1 2 6 24 ...)

```

(define facts
  (cons-stream 1
  (stream-map * (stream-cdr integers) facts)))

```

7. (HARD!) Define powers; the stream should be 1<sup>1</sup>, 2<sup>2</sup>, 3<sup>3</sup>, ..., or, (1 4 16 64 ...). Of course, you cannot use the exponents procedure. I've given you a start, but you don't have to use it.

```

(define powers (helper integers integers))
(define (helper s t)
  (cons-stream (stream-car s)
  (helper (stream-map * (stream-cdr s) (stream-cdr t))
  (stream-cdr t))))

```

## Constructing Streams Through Procedures

### QUESTIONS

1. Define a procedure, `(list->stream ls)` that takes in a list and converts it into a stream.

```

(define (list->stream ls)
  (cond ((null? ls) the-empty-stream)
  (else (cons-stream (car ls) (list->stream (cdr ls))))))

```

2. Define a procedure `(lists-starting n)` that takes in  $n$  and returns a stream containing  $(n)$ ,  $(n\ n+1)$ ,  $(n\ n+1\ n+2)$ , etc. For example, `(lists-starting 1)` returns a stream containing (1) (1 2) (1 2 3) (1 2 3 4)...

```

(define (lists-starting n)
  (cons-stream (list n)
  (stream-map (lambda (ls) (cons n ls)) (lists-starting (+ n 1)))))

```

3. Define a procedure `(chocolate name)` that takes in a name and returns a stream like so:

```

(chocolate `chung) =>
(chung really likes chocolate chung really really likes chocolate chung really
really really likes chocolate ...)

```

You'll want to use helper procedures.

```

(define (chocolate name)
  (define (helper n)
    (cons-stream name
    (stream-append (really n) (helper (+ n 1)))))
  (define (really n)
    (cond ((= n 0)
      (cons-stream `likes
      (cons-stream `chocolate the-empty-stream))
      (else (cons-stream `really (really (- n 1)))))
    (helper 1))

```

---

## Stream Processing

### QUESTIONS:

1. Define a procedure, `(stream-censor s replacements)` that takes in a stream `s` and a table `replacements` and returns a stream with all the instances of all the car of entries in `replacements` replaced with the cadr of entries in `replacements`:

```
(stream-censor (hello you weirdo ...) ((you I-am) (weirdo an-idiot))) =>
(hello I-am an-idiot ...)
```

```
(define (stream-censor s replacements)
  (if (stream-null? s)
      the-empty-stream
      (let ((match (assoc (stream-car s) replacements)))
        (if match
            (cons-stream (cadr match)
                          (stream-censor (stream-cdr s) replacements))
            (cons-stream (stream-car s)
                          (stream-censor (stream-cdr s) replacements))))))
```

2. Define a procedure `(make-alternating s)` that takes in a stream of positive numbers and alternates their signs. So `(make-alternating ones) => (1 -1 1 -1 ...)` and `(make-alternating integers) => (1 -2 3 -4 ...)`. Assume `s` is an infinite stream.

```
(define (make-alternating s)
  (cons-stream (stream-car s)
               (cons-stream (* -1 (stream-car (stream-cdr s)))
                             (make-alternating (stream-cdr (stream-cdr s))))))
```

or, a cooler way:

```
(define (make-alternating s)
  (cons-stream (stream-car s)
               (stream-map (lambda (x) (* -1 x))
                             (make-alternating (stream-cdr s)))))
```

---

## My Body's Floating Down the Muddy Stream

### MORE QUESTIONS

1. Given streams `ones`, `twos`, `threes`, and `fours`, write down the first ten elements of:  
`(interleave ones (interleave twos (interleave threes fours)))`

```
(interleave threes fours) ==> (3 4 3 4 3 4 ...)
(interleave twos threes-fours) ==> (2 3 2 4 2 3 2 4 ...)
(interleave ones twos-threes-fours) ==> (1 2 1 3 1 2 1 4 1 2 1 3 ...)
```

2. Construct a stream `all-integers` that includes 0 and both the negative and positive integers.

```
(define all-integers
  (interleave (make-alternating (integers-starting 0))
              (make-alternating (integers-starting 1))))
```

Or, you could've interleaved the positives and the negatives.

3. Suppose we were foolish enough to try to implement `stream-accumulate`:

```
(define (stream-accumulate combiner null-value s)
  (cond ((stream-null? s) null-value)
        (else (combiner
                  (stream-car s)
                  (stream-accumulate combiner null-value (stream-cdr s))))))
```

What happens when we do:

a. `(define foo (stream-accumulate + 0 integers))`

The `define` statement goes into an infinite loop. When we evaluate `stream-accumulate`, we'll go into the `else` clause, and have to call `stream-accumulate` again on the `stream-cdr` of `integers`, which does the same thing again. The problem is, NOTHING IS DELAYED.

b. `(define bar (cons-stream 1 (stream-accumulate + 0 integers)))`

The `define` statement is fine (since `stream-accumulate` is delayed). But when you call `stream-cdr` on `bar`, all hell breaks loose again.

c. `(define baz (stream-accumulate
 (lambda (x y) (cons-stream x y))
 the-empty-stream integers))`

So the question is, does THIS delay anything? It looks like it does. If the combiner uses `cons-stream`, then it seems that we'll delay the evaluation of `y`, which is the next call to `accumulate`. Alas, that's making the same mistake as believing `new-if` would work. Whereas `cons-stream` is a special form, the combiner is NOT, and so it will evaluate both of its arguments - including the call to `accumulate` - before evaluating its body. So the problem persists.

4. SICP ex. 3.68, page 341. If you understand this, you'll be fine.

This doesn't work. Let's try `(pairs integers integers)`. We start with a call to `interleave`. Well, `interleave` is not a special form, so evaluate both arguments. What's the first argument, the call to `stream-map`? It returns a stream starting with `(1 1)`. What's the second argument, the call to `pairs`? Well, what's `(pairs (stream-cdr integers) (stream-cdr integers))`? It's a call to `interleave`. The first argument to `interleave` is `(2 2)`, and the second argument is a call to `pairs` again...and so on.