QUESTIONS

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

\[
\text{(define \( \text{ones} \) \( \text{(cons 1 \text{(lambda () \( \text{ones} \))})} \), or, just \\
\text{(define \( \text{ones} \) \( \text{(cons 1 \( \text{ones} \))})}\)}
\]

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

\[
\text{(define \( \text{integers-starting n} \) \( \text{(cons n \text{(lambda () \( \text{integers-starting (+ n 1))})})} \)}
\]

Using Stream Operators

Here are some that we'll be using quite a bit:

\[
\text{(stream-map \text{proc \( s \) ...}) -- works just like list \text{map}; can take in any number of streams} \\
\text{(stream-filter \text{proc \( s \)) -- works just like list \text{filter}} \\
\text{(stream-append \( s1 \) \( s2 \)) -- appends two finite streams together (why not infinite streams?)} \\
\text{(interleave \( s1 \) \( s2 \)) -- interleaves two streams into one, with alternating elements from \( s1 \) and \( s2 \)}
\]

Constructing Streams

QUESTIONS: Describe what the following expressions define.

In Class:

1. \( \text{(define \( s1 \) \( \text{(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. \( \text{(define \( s2 \) \( \text{(cons-stream 1} \\
\text{\( \text{(add-stream (stream-map (lambda (x) (* x 2)) \( s2 \)) \( s2 \))} \)} \)

\[
\begin{array}{c}
1 \\
2 & 6 & 18 & \ldots \\
+ & 1 & 3 & 9 & \ldots \\
\text{===================}
\end{array}
\]

\[
\begin{array}{c}
1 & 3 & 9 & 27 & \ldots \\
\text{powers of 3}
\end{array}
\]

3. \( \text{(define \( s4 \) \( \text{(cons-stream 1} \\
\text{\( \text{(cons-stream 2} \\
\text{\( \text{(stream-filter (lambda (x) (not (= x 1))) \( s4 \))})} \)} \)

\[
\begin{array}{c}
1 & 2 & 2 & 2 & \ldots \\
\text{Rather counter-intuitive, but...well, we know that it starts with 1 and 2, since we said so. Then, the stream-cddr 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 \text{pred? \( (\text{stream-cdr s}) \))}, where \text{pred?} \text{ is the lambda, and}
\end{array}
\]
4. Define \textbf{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 \((1 \ 2 \ 6 \ 24 \ ...)\)

\[
\begin{align*}
\text{(define facts} & \text{ (1 2 6 24 ...))} \\
\text{Extra Practice:} & \\
5. & \text{(define s5} \\
& \text{(1 2 4 7 ...} \\
& + \text{ 1 2 3 4 ...} \\
& \text{=====================} \\
& \text{1 2 4 7 11 ... starting from 1, add 1, 2, 3, etc.}) \\
6. & \text{(define s3} \\
& \text{(1 2 4 7 ...} \\
& \text{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.}) \\
7. & \text{(HARD!) Define powers; the stream should be } 1^1, 2^2, 3^3, \ldots, \text{ or, } (1 \ 4 \ 16 \ 64 \ ...). \text{ Of course, you cannot use the exponents procedure. I've given you a start, but you don't have to use it.} \\
& \text{(define powers (helper integers integers))} \\
& \text{(define (helper s t} \\
& \text{(cons-stream} \\
& \text{(cons-stream (stream-car s) \\
& \text{(helper (stream-map * (stream-cdr s) (stream-cdr t)) \\
& \text{(stream-cdr t))))}) \\
\text{QUESTIONS} \\
1. Define a procedure \text{(lists-starting n)} that takes in \(n\) and returns a stream containing \((n), (n+1), (n+1+n+2), \text{ etc. For example, (lists-starting 1) returns a stream containing (1) (1 2) (1 2 3) (1 2 3 4)...} \\
\text{(define (lists-starting n)} \\
& \text{(cons-stream (list n) \\
& \text{(stream-map (lambda (ls) (cons n ls)) (lists-starting (+ n 1))))})
2. 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))))))
Extra Practice:

My Body's Floating Down the Muddy Stream

MORE QUESTIONS (extra practice)

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.

5. **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))))))

CS61A Summer 2011—Eric, Stephanie, Kevin, Hamilton, Phill—Notes Courtesy of Justin Chen/Chung Wu