STREAMS 10

COMPUTER SCIENCE 61AS

Basics of Streams

1. What is a stream?

2. How does memoization work?

3. Is a `cons-stream` a special form?

Practice with Streams

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. Do NOT use `cons-stream`.

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`. Again, do NOT use `cons-stream` for this part.

3. Describe what the following expressions define:

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

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

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

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

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

4. 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 ...)`. Hint: use the `integers` stream.

5. (HARD!) Define `powers`; the stream should be $1^1, 2^2, 3^3$ or, `(1 4 16 64 ...)`. You cannot use the exponents procedure.

Practice with Streams

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

2. Define a procedure, `(list->stream ls)` that takes in a list and converts it into a stream. Remember, streams don't have to be infinite, and finite streams end with `the-empty-stream`

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

```
STk>(chocolate 'chung)
(chung really likes chocolate chung really really likes chocolate ...)
```

Youll want to use helper procedures.

Stream-processing

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

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

2. Define a procedure `(make-alternating s)` that takes in a stream of positive numbers and alternate their signs. So

```
STk>(make-alternating ones)
(1 -1 1 -1 1 -1 ...)
```

and

```
STk>(make-alternating integers)
(1 -2 3 -4 ...)
```

. Assume `s` is an infinite stream.

My Bodys Floating Down The Muddy Stream

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

```
(interleave ones (interleave twos (interleave threes fours)))
```

2. Construct a stream `all-integers` that includes 0 and both the negative and positive integers. You may use procedures that you have defined above.

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))))))
```

4. What happens when we do:

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

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

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

5. Louis Reasoner thinks that building a stream of pairs from three parts is unnecessarily complicated. Instead of separating the pair (S_0, T_0) from the rest of the pairs in the first row, he proposes to work with the whole first row, as follows:

```
(define (pairs s t)
  (interleave
   (stream-map (lambda (x) (list (stream-car s) x))
               t)
   (pairs (stream-cdr s) (stream-cdr t))))
```

Does this work? Consider what happens if we evaluate `(pairs integers integers)` using Louis's definition of pairs.