

STREAMS 10

COMPUTER SCIENCE 61AS

Basics of Streams

1. What is a stream?

Solution: A stream is an element and a promise to evaluate the rest of the stream. Streams are a clever idea that allows one to use sequence manipulations without incurring the costs of manipulating sequences as lists. A stream is implemented as a pair whose `car` is forced and whose `cdr` is a promise.

2. How does memoization work?

Solution: The first time the memoized procedure is run, it saves the computed result. On subsequent evaluations, it simply returns the result.

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

Solution: Yes. `cons-stream` will not evaluate its second argument, making it 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`.

Solution:

```
(define (ones) (cons 1 (lambda () (ones))))
```

or

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

Solution:

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

3. Describe what the following expressions define:

- a.

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

Solution: infinite loop! We didnt specify a first element. Even the define statement will go into infinite loop.

- b.

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

Solution:

```
this is:
1
2 6 18 ...
+ 1 3 9 ...
=====
1 3 9 27 ...
So powers of 3.
```

- c.

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

Solution:

Remember:

```
(define (stream-filter pred? s)
  (cond ((stream-null? s) the-empty-stream)
        ((pred? (stream-car s))
         (cons-stream (stream-car s)
                       (stream-filter pred? (stream-cdr s))))
        (else (stream-filter pred? (stream-cdr s)))))
```

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.

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

Solution:

```
(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-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 pred? (stream-cdr s))
```

, where pred? is the lambda, and s is s4. Whats (stream-cdr s4)? Well, its 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
```

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

Solution:

Well, so,

```
1
```

```
1 2 4 7 ...
```

```
+ 1 2 3 4 ...
```

```
=====
```

```
1 2 4 7 11 16 22 29 ...
```

So, starting from 1, add 1, add 2, add 3, etc.

4. Define `facts` without defining any procedures; the stream should be a stream of $1!, 2!, 3!, 4!, \dots$. More specifically, it returns a stream with elements `(1 2 6 24 ...)`. Hint: use the `integers` stream.

Solution:

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

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.

Solution:

```
(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))))
```

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))
```

Solution:

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

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`

Solution:

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

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 ...)
```

You'll want to use helper procedures.

Solution:

```
(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

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 ...)

```

Solution:

```

(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 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.

Solution:

```
(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 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)))
```

Solution:

```
(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. You may use procedures that you have defined above.

Solution:

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

Or, you couldve 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))))))
```

4. What happens when we do:

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

Solution: 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)))`

Solution: 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.

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

Solution: 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.

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.

Solution: 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 it evaluates 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.

Note: this is the same as SICP 3.68. This is a pretty hard problem, but if you understand this you should be fine.