Infinite Streams/Infinite Trees

Bookkeeping...

(define (nth n s)
  (if (= n 1)
      (stream-car s)
      (nth (- n 1) (stream-cdr s)))))
;Value: nth

(define (show a s)
  (map (lambda (n) (nth n s)) a))
;Value: show

(define (peek s)
  (show '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15) s))
;Value: peek

Stream building blocks...

(define (add s t)
  (cons-stream (+ (stream-car s) (stream-car t))
               (add (stream-cdr s) (stream-cdr t)))))
;Value: add

(define (map-stream fn s)
  (cons-stream (fn (stream-car s))
               (map-stream fn (stream-cdr s)))))
;Value: map

(define (filter-stream fn s)
  (if (fn (stream-car s))
      (cons-stream (stream-car s) (filter-stream fn (stream-cdr s)))
      (filter-stream fn (stream-cdr s)))))
;Value: filter
Some infinite streams...

(define ints
  (cons-stream 1 (map-stream 1+ ints)))
;Value: ints

(peek ints)
;Value: (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

(define (ndiv n)
  (lambda (x) (not (= (remainder x n) 0))))
;Value: ndiv

(define odds (filter-stream (ndiv 2) ints))
;Value: odds

(peek odds)
;Value: (1 3 5 7 9 11 13 15 17 19 21 23 25 27 29)

(define (sieve s)
  (cons-stream (stream-car s)
                (sieve
                 (filter-stream (ndiv (stream-car s))
                                (stream-cdr s)))))
;Value: sieve

(define primes (sieve (stream-cdr ints)))
;Value: primes

(peek primes)
;Value: (2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)

(define fibs
  (cons-stream 0
               (cons-stream 1
                            (add fibs (stream-cdr fibs)))))
;Value: fibs

(peek fibs)
;Value: (0 1 1 2 3 5 8 13 21 34 55 89 144 233 377)
Here's how to implement a bank account without state, using streams instead:

\[
\text{(define (account balance A)} \nonumber \\
\quad \text{(cons-stream balance)} \nonumber \\
\quad \quad \quad \text{(account (- balance (stream-car A)))} \nonumber \\
\quad \quad \quad \quad \quad \text{(stream-cdr A))}) \nonumber 
\]

Now: what about a *shared account*? Suppose Anne and Harry share an account—in the state world, we could do the following:

\[
\text{(define (make-account balance)} \nonumber \\
\quad \text{(lambda (w)}} \nonumber \\
\quad \quad \quad \text{(set! balance (- balance w))}} \nonumber \\
\quad \quad \quad \quad \quad \text{balance})} \nonumber \\
\quad \text{;Value: make-account} \nonumber 
\]

\[
\text{(define anne (make-account 1000))} \nonumber \\
\quad \text{;Value: anne} \nonumber 
\]

\[
\text{(define harry anne)} \nonumber \\
\quad \text{;Value: harry} \nonumber 
\]

\[
\text{(anne 10)} \nonumber \\
\quad \text{;Value: 990} \nonumber 
\]

\[
\text{(harry 50)} \nonumber \\
\quad \text{;Value: 940} \nonumber 
\]

How do we model this in the streams world? Problem—we don't know *who goes first*...
Suppose for simplicity that Anne always withdraws $1, and Harry always withdraws $2. Describe them by streams:

(define anne (cons-stream 1 anne))  
;Value: anne

(define harry (cons-stream 2 harry))  
;Value: harry

Streams are just delayed lists. Consider instead an implementation using delayed trees—the left and right branches are both represented by procedures that describe how to “expand” the tree. In this case, we want to consider all possible merges of withdrawals between the joint holders of an account:

(define (joint balance A H)
  (list balance
    (lambda () (joint (- balance (stream-car A))
      (stream-cdr A)
      H))
    (lambda () (joint (- balance (stream-car H))
      A
      (stream-cdr H))))

;Value: joint

(define (top dtree) (car dtree))  
;Value: top

(define (left dtree) ((cadr dtree)))  
;Value: left

(define (right dtree) ((caddr dtree)))  
;Value: right
Represent *all possible merges* of Anne and Harry’s behavior by a delayed tree:

```
(define tree (joint 1000 anne harry))
;Value: tree
tree
;Value: (1000 #[compound-procedure 14] #[compound-procedure 15])
(left tree)
;Value: (999 #[compound-procedure 17] #[compound-procedure 18])
(right tree)
;Value: (998 #[compound-procedure 20] #[compound-procedure 21])
(right (right tree))
;Value: (996 #[compound-procedure 23] #[compound-procedure 24])
(left (right tree))
;Value: (997 #[compound-procedure 26] #[compound-procedure 27])
```

Let’s choose a *random path* through the tree...

```
(define (rstream) (cons-stream (random 2) (rstream)))
;Value: rstream
(peek (rstream))
;Value: (0 1 0 0 1 0 1 0 0 0 0 0)
(peek (rstream))
;Value: (1 1 0 1 0 1 1 1 0 0 1 0)
(define (random-path stream dtree)
  (cons-stream (top dtree)
    (random-path
      (stream-cdr stream)
      ((if (zero? (stream-car stream)) left right) dtree))))
;Value: random-path
(peek (random-path (rstream) tree))
;Value: (1000 999 997 995 994 993 991 989 988 987 986 985 984 982 980)
```

**A Weird Thought**

Procedures are just *delayed infinite data structures*. For example, *we’ve been lying to you*—*square* is just a stream, and when you evaluate *(square 10)*, you’re extracting the 10th thing in the stream.