streams

Dr. Kimberle Koile
(define (cons-my-list first rest) (cons first rest))

(define ints-from-1-to-4 (cons-my-list 1 '(2 3 4)))

(define ints-from-1 ???)

(define (cons-stream first (rest (lazy-memo) (cons first rest)))

ints-from-1

\[ \text{thunk for ints from 2} \]
(define (ints-from-n n n)
  (cons-stream n (ints-from (+ n 1))))

(define ints (ints-from 1))
streams summary

Key ideas:

> streams are delayed lists

> represent a stream as a cons-stream, pair-like object with lazy cdr

> define a stream by figuring out first element, then how to compute rest

Examples:

> integers 1, 2, 3, 4, 5 ....

> factorials 1, 2, 6, 24, 120 ...
another way to think about streams

ints

first element?

rest?

(define ints
  (cons-stream 1 (add-streams ints ones)))

facts

first element?

rest?

(define facts
  (cons-stream 1 (mult-streams facts
                  (stream-cdr ints)))))
summary of examples of defining streams

finite

(stream-interval 1 1x10^100)  
(define (stream-interval a b)
  (cons-stream a (stream-interval (+ a 1) b)))

infinite (indefinite)

explicit

(define (ints-from n)
  (cons-stream n (ints-from (+ n 1))))
(define ints (ints-from 1))

implicit

(define ints (cons-stream
  1
  (add-streams ints ones)))

+1 +1 +1 +1
① 2 3 4 5 ...
useful stream procedures

stream-filter | add-streams
stream-map | mult-streams
  . 2 args
  . variable number args
stream-ref

(define (any? test l)
  (cond ((null? l) #f
            ((test (car l)) #t)
            (else (any? test (cdr l)))))

(define (map proc . args)
  (if (null? args)
      nil
      (if (any? null? args)
          nil
          (cons (apply proc (map car args))
                (apply map (cons proc (map cdr args)))))))

(define (stream-map proc . args)
  (if (null? args)
      nil
      (if (any? null? args)
          nil
          (cons-stream (apply proc (map stream-car args))
                      (apply stream-map (cons proc (map stream-cdr args)))))))

Does this procedure work?  yes  first elt

(define (add-streams s1 s2)
  (cons-stream (+ (stream-car s1) (stream-car s2))
               (add-streams (stream-cdr s1) (stream-cdr s2))))
another example

What value is printed in response to the last expression in this sequence of expressions?

\[
\text{(define evens (cons-stream 2 (stream-map (lambda (x) (+ x 2)) evens))}
\]

\[
\text{(stream-car}
\text{(add-streams evens (stream-cdr (stream-cdr evens))))))))
\]

\[
2 \quad 4 \quad 6 \quad 8 \quad 10 \quad \ldots
\]

\[
\underline{2} \quad 4 \quad 6 \quad \ldots
\]

\[
\underline{8} \quad 12 \quad 16
\]
Problem 1

1. Write mult-stream which takes two streams and returns a new stream that is the product of the two streams.

   (define (mult-streams s1 s2)
     (cons-stream (* (stream-car s1) (stream-car s2))
                 (mult-streams (stream-cdr s1) (stream-cdr s2))))
Problem 2

2. Write stream-ref, modeled after list-ref, which takes a stream and a number n and returns the nth element of the stream.

(define (list-ref x n)
  (if (= n 0)
      (car x)
      (list-ref (cdr x (- n 1)))))

(define (stream-ref x n)
  (if (= n 0)
      (stream-car x)
      (stream-ref (stream-cdr x) (- n 1)))))
3. Write list->stream, which turns a list into a stream.

(define  (list->stream l)
  (cons-stream (car l) (list->stream (cdr)))))
Problem 4 (modified from a previous final exam problem)

4. Assume that the following have been evaluated:

   (define ones (cons-stream 1 ones))

   (define (add-streams s1 s2)
      (cons-stream (+ (stream-car s1) (stream-car s2))
      (add-streams (stream-cdr s1) (stream-cdr s2)))))

Consider the expression:

   (define integers (add-streams ones integers))

For each of the following, put an X in the box if the statement applies to the above scenario:

☐ The expression evaluates to a stream of integers.

☐ The interpreter goes into an infinite loop when (stream-cdr integers) is evaluated.

☐ An "unbound variable" error occurs when the above expression defining ones is evaluated.

☒ An "unbound variable" error occurs when the above expression defining integers is evaluated.

Because the second arg to add-streams is not delayed
Problem 5  (from a previous final exam)

5. What value is printed in response to the last expression in this sequence of expressions?

(define s (cons-stream 1 (stream-map (lambda (x) (* x 2)) s))

(stream-car
(stream-cdr
(stream-cdr
(add-streams s (stream-cdr (stream-cdr s)))))))

\[\begin{array}{c}
1 \\
2 \\
4 \\
8 \\
16 \\
32 \\
64 \\
\vdots \\
\end{array}\]

\[\begin{array}{c}
1 \\
2 \\
4 \\
8 \\
16 \\
\vdots \\
\end{array}\]

\[\begin{array}{c}
[10] \quad [20] \\
40 \\
80 \\
\end{array}\]
Problem 6

Consider the sequence of expressions:

\[
\begin{align*}
\text{(define (stream-enumerate-interval low high)} & \text{)} \\
& \text{(if (> low high)} \\
& \quad \text{the-empty-stream} \\
& \quad \text{(cons-stream low (stream-enumerate-interval (+ low 1) high))}) \\
\text{(define sum 0)} \\
\text{(define (accum x)} \\
& \quad \text{(set! sum (+ x sum))} \\
& \quad \text{sum)} \\
\text{(define seq (stream-map accum (stream-enumerate-interval 1 10))} \\
\text{(define y (stream-filter even? seq))} \\
\text{(define z (stream-filter (lambda (x) (= (remainder x 5) 0)) seq))}
\end{align*}
\]

What is the printed response to evaluating the following expressions. Assume print-stream prints out stream elements inside [], e.g. [1 2 3]

6a. (print-stream y) 
\[
[6 \ 10 \ 28 \ 36]
\]

6b. (stream-ref y 3) 
36

6c. (print-stream z) 
\[
[10 \ 15 \ 45 \ 55]
\]
Assume that we're interested in the partial sums of a stream. Given a stream \( S \), for example, a stream of partial sums for \( S \) is the stream \( S_0, S_0 + S_1, S_0 + S_1 + S_2, \ldots \).

7a. Write an expression that defines a stream that is the partial sum of integers. For example, \( (\text{partial-sums }\ \text{integers}) \) should be the stream \( 1, 3, 6, 10, 15 \ldots \)

\[
\text{(define ints (cons-stream 1 (add-streams ints ones)))}
\]

\[
\text{(define int-partial-sums}
\]
\[
\quad (\text{cons-stream 1 (add-streams ints ones)))
\]
\[
\)

7b. Write a procedure \( \text{partial-sums} \) that takes a stream as an argument, and returns the stream \( S_0, S_0 + S_1, S_0 + S_1 + S_2 \ldots \). For example, \( (\text{partial-sums }\ \text{integers}) \) should be the same stream as in part a.

\[
\text{(define (partial-sums s)}
\]
\[
\quad (\text{cons-stream (stream-car s)}
\]
\[
\quad \quad (\text{add-streams (partial-sums s)}
\]
\[
\quad \quad \quad (\text{stream-car s}))
\]
\[
\)

8. Suppose you are given two streams and you need to produce a stream that contains both. Translating append, which works on lists, into an append-stream procedure by changing the data abstraction selectors and constructor will not work if the streams are indefinite in length: "appending" the infinite stream $S_1, S_2, S_3 \ldots$ and a second infinite stream $T_1, T_2, T_3 \ldots$ results in the stream $S_1 S_2, S_3, \ldots, T_1, T_2, T_3, \ldots$ which is effectively the same as the first stream. The solution is to merge the two streams instead of appending them. Write a procedure called alternate-streams that consumes two streams and returns a single one that contains elements alternating from the two inputs.

Remember that the data abstraction for streams uses stream-null?, stream-car, stream-cdr, null-stream, and cons-stream.

```scheme
(define alternate-streams s1 s2)

(if (stream-null? s1)
   s2
   (cons-stream (stream-car s1)
                (alternate-streams s2 (stream-cdr s2))))
```

```scheme
(define alternate-streams s1 s2)

(if (stream-null? s1)
   s2
   (cons-stream (stream-car s1)
                (cons-stream (stream-car s2)
                             (alternate-streams (stream-cdr s1) (stream-cdr s2))))))
)```