CSCI 275: Programming Abstractions Lecture 23: Streams (cont.) Spring 2025

Stephen Checkoway Slides from Molly Q Feldman





Reminder: Better Evaluation in Built-in Racket

(delay exp) returns an object called a promise, without evaluating exp

returns its value

- If the promise's exp has not been evaluated yet, it is evaluated and cached; otherwise, the cached value is returned • A promised expression is evaluated only once, no matter how many
- times it is forced!

(force promise) evaluates the promised expression and



Promises in action!

> (define prime-lst (primes)) > prime-lst '(2 . #<promise>) > (force (cdr prime-lst)) '(3 . #<promise>) > (force (cdr (force (cdr prime-lst)))) '(5 . #<promise>) > prime-lst

This worked, but it was a bit annoying if we wanted to process the whole list!

'(2 . #<promise!(3 . #<promise!(5 . #<promise>)>))



Available Stream Procedures These are already built-in, so we don't need to write them!

(require racket/stream) (stream exp ...) ; Works like (list exp ...) (stream? v) (stream-cons head tail) (stream-first s) (stream-rest s) (stream-empty? s) empty-stream (stream-ref s idx)

And several others

Constructing an Infinite Length Stream

Write a procedure which

- returns a stream constructed via stream-cons
- where the tail of the stream is a recursive call to the procedure

Call the procedure with the initial argument

(define (integers-from n) (stream-cons n (integers-from (add1 n))))

(define positive-integers (integers-from 0))

Handy testing function for streams

(define (sp s) (stream->list (stream-take s 10)))

out as normal

- This returns a list of the first 10 elements in the stream which DrRacket will print

the numbers of the Collatz sequence starting with $x_1 = n$

The rest of the sequence is defined by $x_{i+1} = x_i / 2$ if x_i is even $x_{i+1} = 3x_i + 1$ if x_i is odd

Examples sequences with different starting values: 1 4 2 1 4 2 1 4 ... 3 10 5 16 8 4 2 1 ... 10 5 16 8 4 2 1 4 ... -1 -2 -1 -2 -1 -2 -1 -2 ...

A. Vote for anything when done

Write a Racket function (collatz n) that returns a stream containing

Building streams from streams

How to write a procedure that adds two streams together • Use stream-cons to construct the new stream • Use stream-first on each stream to get the heads • Recurse on the tails via stream-rest

(define (stream-add s t) (cond [(stream-empty? s) empty-stream] [(stream-empty? t) empty-stream] [else (stream-cons (+ (stream-first s))

- (stream-first t))
- (stream-add (stream-rest s)
 - (stream-rest t)))]))



Fun example with laziness

- element and the previous element
- elements of fibs to the elements of (stream-rest fibs)
- 0 1 1 2 3 5 ... fibs 1 1 2 3 5 8 ... (stream-rest fibs) 2 3 5 8 13 ...

We get a new stream that's just (stream-rest (stream-rest fibs))

Consider the Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, ... where we start with 0 and 1 and then to get the next element in the sequence we add the current

If fibs is that sequence as a stream then consider what happens if we add the

Fun example with laziness continued

and then followed by adding fibs and (stream-rest fibs)? (define fibs (stream-cons 0 (stream-cons 1

A. We get the Fibonacci sequence B. We get an error before running because fibs is used before it is defined C. We get an error at run time because we're accessing elements of fibs before they're defined

What happens if we define fibs to be the stream starting with 0 and 1

(stream-add fibs (stream-rest fibs))))

Higher-order stream functions

streams

element of s

for which f does not return false

init as the initial accumulator value

- Like their list counterparts, we can write higher-order functions that operate on
 - (stream-map f s) Returns a stream that contains f applied to each
 - (stream-filter f s) Returns a stream that contains the elements of s
 - (stream-fold f init s) Folds (f acc elem) over s starting with

If we have a stream of numbers s, how can we construct a new stream whose elements are twice the value of the elements in s?

A. (list->stream $(map (\lambda (x) (* x 2) (stream->list s)))$ B. (stream-map (λ (x acc) S) C. (stream-map (λ (x) (* x 2)) s) D. (map (λ (x) (stream (* x 2)) s)

(stream-cons (* x 2) acc))

both finite-length streams and infinite-length streams

Hint: Think about how you'd implement the map function for lists

A. Vote for anything when done

Implement the (stream-map f s) function. Make sure it works for

using basic recursion with empty?, empty, cons, first, and rest

Stream Procedures

Implement (stream-filter f s) which returns a stream containing the elements of s (in order) such that applying f to the element returns anything other than #f

Bonus: You can prevent your implementation from evaluating f on elements of the stream **at the time you call stream-filter** by wrapping your implementation in a call to stream-lazy

Compare (stream-filter (λ (x) #f) (collatz 1)) when stream-filter is not implemented with stream-lazy to when it is

Write some more stream procedures

(stream-double s) Returns a stream containing each element of s twice (stream-double (stream 1 2 3)) =>(stream 1 1 2 2 3 3)

(stream-interleave s t) Returns a stream that interleaves elements of s and t => (stream 1 'a 2 'b 3 'c 'd)

(stream-interleave (stream 1 2 3) '(a b c d))