Programming Abstractions

Lecture 32: Streams 2

Stephen Checkoway

Streams in Racket

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

Simplest infinite-length stream: A stream of all zeros

```
(define all-zeros
  (stream-cons 0 all-zeros))
```

Note that we couldn't do this with a list

```
(define all-zeros-lst
  (cons 0 all-zeros-lst))
```

```
Error: all-zeros-lst: undefined;
       cannot reference an identifier before its definition
```

Why does

```
(define all-zeros
  (stream-cons 0 all-zeros))
```

work when the list-version does not?

A. Streams are magic

B. Streams are lazy so the stream-cons doesn't run until all-zeros is accessed for the first time

C. Streams are lazy so although the stream is constructed by `stream-cons`, its "first" and "rest" part aren't evaluated until forced by `stream-first` and `stream-rest`

D. Racket treats streams specially so it knows this construction is okay `(stream-length s)` is a standard Racket stream function that returns the length of the stream

What is the result of this code?

```
(define all-zeros
  (stream-cons 0 all-zeros))
(stream-length all-zeros)
```

A. 0

B. +inf.0 (which is how Racket spells positive infinity)

C. +nan.0 (which is how Racket spells Not a Number (NaN))

D. Infinite loop

E. Error

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))
```

Primes as a stream

```
(define (prime? n) …) ; Returns #t if n is prime

(define (next-prime n)
  (cond [(prime? n) (stream-cons n (next-prime (+ n 2)))]
        [else (next-prime (+ n 2))]))

(define (primes)
  (stream-cons 2 (next-prime 3)))
```

Fibonacci numbers as a stream

Recall the Fibonacci numbers are defined by $f_0 = 0$, $f_1 = 1$ and $f_n = f_{n\text{-}1} + f_{n\text{-}2}$

```
(define (next-fib m n)
  (stream-cons m (next-fib n (+ m n))))

(define fibs (next-fib 0 1))
```

Building streams from streams

Let's write a procedure to add 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)))]))
```

Fibonacci numbers as a stream: take 2

$f_0 = 0$, $f_1 = 1$ and $f_n = f_{n\text{-}1} + f_{n\text{-}2}$

We can build our Fibonacci sequence directly from that definition (this is silly)

```
(define fibs
  (stream-cons
   0
   (stream-cons
    1
    (stream-add fibs (stream-rest fibs)))))
```

Write some infinite-length streams

- `(constant-stream x)`
  Returns a stream containing an infinite number of x
  `(stream->list (stream-take (constant-stream 'ha) 10))`
  `=> '(ha ha ha ha ha ha ha ha ha ha)`
- `(define abc ...)`
  Define an infinite-length stream (not a function) consisting of 'A, 'B, 'C repeating in order. [Hint: `(stream* ...)` makes this short]
  `(stream->list (stream-take abc 12))`
  `=> '(A B C A B C A B C A B C)`
- `(stream-cycle s)`
  Returns an infinite-length stream consisting of the elements of s repeating in order. E.g., the abc stream could be rewritten as
  `(stream-cycle (stream 'A 'B 'C))`

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

Write more stream procedures

Write the following procedures that act like their list counterparts, but operate lazily on streams; in particular, do not covert them to lists!

- `(stream-take s num)`
  Returns a stream containing the first `num` elements of `s`, make sure this is lazy
- `(stream-drop s num)`
  Returns a stream containing all of the elements of s in order *except* for the first `num`
- `(stream-filter f s)`
  Returns a stream containing the elements `x` of s for which `(f x)` returns true

Multi-argument stream-map

`(stream-map f s ...)`

Racket has stream-map built-in but unlike its list counterparts, it only takes a single stream

Generalize it to take any number of streams where the length of the returned string is the minimum length of any of the stream arguments (i.e., return `empty-stream` if any of the streams becomes empty); you'll want to use `ormap`, `map` and `apply`

- `(define (stream-map f . ss) …)`