



## Problem with conventional interface approach

- Sum-palindromes-2 is easier to understand
- But it is grossly inefficient if `b - a` is large, e.g, `a = 100` and `b = 100,000,000`
- Finding the first palindrome number can also be inefficient:

```
(car (filter palindrome?  
      (enumerate-interval 123 100000000)))
```

## Key property of streams

- Streams allow us to generate the elements of a list incrementally as they are needed
- We want to think of streams as lists, so we'll need the equivalent of `()`, `null?`, `car`, `cdr`, `cons` and other procedures for building and manipulating lists

## Some basics

```
(define the-empty-stream ())  
(define (stream-null? x) (null? x))
```

We need a `cons`:

```
(cons-stream <value>  
            <procedure-call promise>)
```

## Delay and force

- **Delay** macro generates promises
  - **Force** converts promises into procedure calls
- ```
(define test-promise  
  (delay (display 'hi)))  
(force test-promise) --> hi
```

## Implementing cons-stream

(In DrScheme version 205)

```
(define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream value-expr call-expr)
     (cons value-expr (delay call-expr))))))
(In DrScheme version 103)
(define-macro cons-stream
  (lambda (val expr)
    (list 'cons val (list 'delay expr)))))
```

Expression (cons-stream <value> <call-expr>) expands into (cons <value> (delay <call-expr>)) which is then evaluated

## Selector functions

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
(define (stream-ref stream n)
  (if (= n 0)
      (stream-car stream)
      (stream-ref (stream-cdr stream) (- n 1))))
```

## Stream version of enumerate-interval

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
        low
        (stream-enumerate-interval (+ low 1)
                                    high)))))
```

## Equivalent to writing ...

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons
        low
        (delay (stream-enumerate-interval (+ low 1)
  high))))))
```

## Stream-filter

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

## Finding palindrome numbers efficiently

```
(define palnums
  (stream-filter
   palindrome?
   (stream-enumerate-interval
    123
    100000000)))
(stream-car palnums) --> 131
(stream-ref palnums 99) --> 2222
(stream-ref palnums 1000) --> 92329
```

## Some useful stream procedures

```
(define (stream-map proc stream)
  (if (stream-null? stream)
      the-empty-stream
      (cons-stream
       (proc (stream-car stream))
       (stream-map proc (stream-cdr stream)))))
```

## Stream-for-each

```
(define (stream-for-each proc stream)
  (if (stream-null? stream)
      'done
      (begin
       (proc (stream-car stream))
       (stream-for-each proc
                        (stream-cdr stream)))))
```

## Display-stream

```
(define (display-stream stream)
  (stream-for-each
   (lambda (x)
     (newline)
     (display x))
   stream))
```

## Implementing delay

- Delay needs to produce something that can evaluate to a procedure call later
- Lambda expressions to this
- We can expand (delay <procedure call>) to (lambda () <procedure call>)

## Simple implementation

```
(In DrScheme version 205)
(define-syntax delay
  (syntax-rules ()
    ((delay expr) (lambda () expr))))
(In DrScheme version 103)
(define-macro delay
  (lambda (expr) (list 'lambda () expr)))
```

## Implementing force

```
(define (force delayed-proc-call)
  (delayed-proc-call))
```

## Making delay more efficient

It would be more efficient if the delayed procedure call were evaluated only once and the result stored for future reuse

## Memoization

```
(define (memo-proc proc)
  (let ((already-run? #f) (result #f))
    (lambda ()
      (if already-run?
          result
          (begin (set! result (proc))
                  (set! already-run? #t)
                  result))))))
```

## A better delay

```
(In DrScheme version 205)
(define-syntax delay
  (syntax-rules ()
    ((delay expr) (memo-proc (lambda () expr)))))
(In DrScheme version 103)
(define-macro delay
  (lambda (expr) (list 'memo-proc
                       (list 'lambda () expr)))))
```

## Infinite streams

```
(define (integers-from n)
  (cons-stream n (integers-from (+ n 1))))
(define large-integers (integers-from 100))
(define large-palindromes
  (stream-filter palindrome? large-integers))
(stream-ref large-palindromes 11) --> 212
```

## Stream of Fibonacci numbers

```
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))
(stream-ref fibs 10) --> 55
```

## Recursively defined streams

```
(define lucky (cons-stream 7 lucky))
```

## Stream addition

```
(define (add-stream stream1 stream2)
  (stream-map + stream1 stream2))
```

(See Exercise 3.50 for generalized `stream-map`)

## Recursive Fibonacci stream (theory)

- $\text{Fib}(n+2) = \text{Fib}(n+1) + \text{Fib}(n)$
- In terms of fibs stream:  
`(stream-cdr (stream-cdr fibs)) =`  
`(add-stream (stream-cdr fibs) fibs)`
- (In general, when the index is  $(n+k)$ , `stream-cdr` has to be applied  $k$  times)

## In other words ...

```
fibs = 0 1 1 2 3 5 8 13 21 ...  
+ (stream-cdr fibs) = 1 1 2 3 5 8 13 21 34 ...  
= 1 2 3 5 8 13 21 34 55 ...  
= (stream-cdr (stream-cdr fibs))
```

hence

```
fibs =  
  (cons-stream  
   0  
   (cons-stream  
    1  
    (stream-cdr (stream-cdr fibs))))
```

## Recursive definition

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

## Why it works

- By the time `(add-streams (stream-cdr fibs) fibs)` is evaluated, the first two values in `fibs` are already explicitly in the stream (memoized), so first value in the stream addition of `(stream-cdr fibs)` and `fibs` can be calculated
- Once the first value in the addition stream is calculated, it is made the third value in `fibs`
- The second and third values in `fibs` are then available to calculate the second value in the addition stream
- This second value in the addition stream is made the fourth value in `fibs`
- And so forth

## Scalar multiplication of streams

```
(define (scale-stream stream factor)  
  (stream-map (lambda (x) (* x factor))  
              stream))
```

## Recursive stream of powers

```
(define triple  
  (cons-stream 1  
    (scale-stream triple  
      3)))
```