

Streams

Tiffany Perumpail from cs61a

Lazy Evaluation in Scheme

Streams are similar to lists, except that the tail of a stream is not evaluated until we asked to do it. This allows streams to be used to represent infinitely long lists.

Lazy evaluation

- In Python, iterators and generators allowed for **lazy evaluation**

Can represent large or infinite sequences

```
def ints(first):  
    while True:  
        yield first  
        first += 1
```

```
>>> s = ints(1)  
>>> next(s)  
1  
>>> next(s)  
2
```

-
- Scheme doesn't have iterators.
How about a list?

Second argument to cons is **always evaluated**

```
(define (ints first)  
  (cons first (ints (+ first 1))))
```

```
scm> (ints 1)  
maximum recursion depth exceeded
```

Streams

Instead of iterators,
Scheme uses **streams**

```
(define (ints first)
  (cons first
        (ints (+ first 1))))
```

```
scm> (ints 1)
maximum recursion depth exceeded
```

```
(define (ints first)
  (cons-stream first
              (ints (+ first 1))))
```

```
scm> (ints 1)
(1 . #[promise (not forced)])
```

Lazy evaluation, just like
iterators in Python

Streams

```
scm> (define s (cons-stream 1 (cons-stream 2 nil)))
```

```
s
```

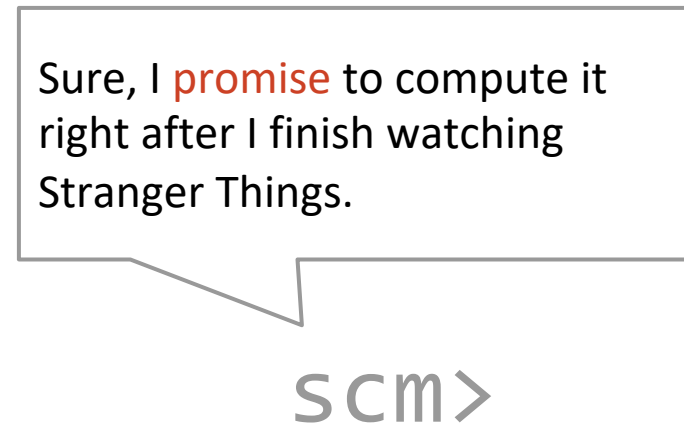
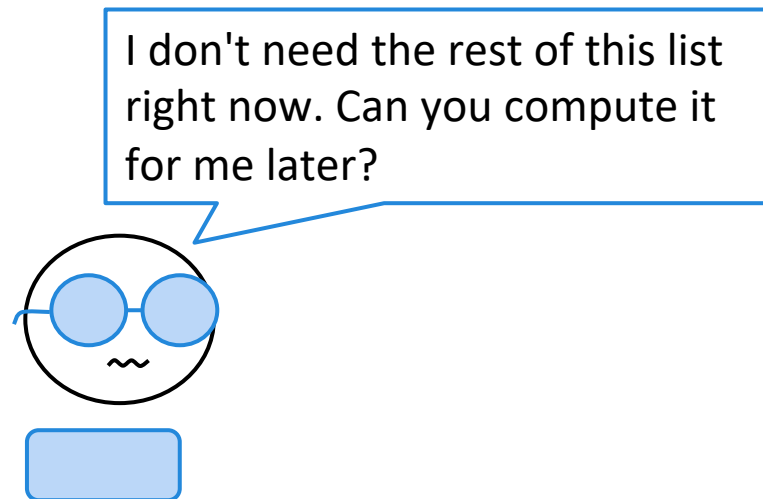
```
scm> s
```

```
(1 . #[promise (not forced)])
```

```
scm> (car s)
```

```
1
```

- **Stream**: (linked) list whose **rest** is lazily evaluated
 - A **promise** to compute



Streams

```
scm> (define s (cons-stream 1 (cons-stream 2 nil)))
```

```
s
```

```
scm> s
```

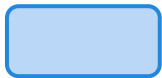
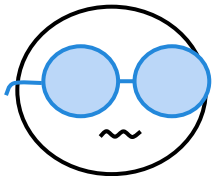
```
(1 . #[promise (not forced)])
```

```
scm> (cdr s)
```

```
#[promise (not forced)]
```

- `cdr` returns the rest of a list
 - For normal lists, the rest is another list
 - The rest of a stream is a **promise to compute the list**

I want the `cdr` of the list now.



Streams

```
scm> (define s (cons-stream 1 (cons-stream 2 nil)))
```

```
s
```

```
scm> s
```

```
(1 . #[promise (not forced)])
```

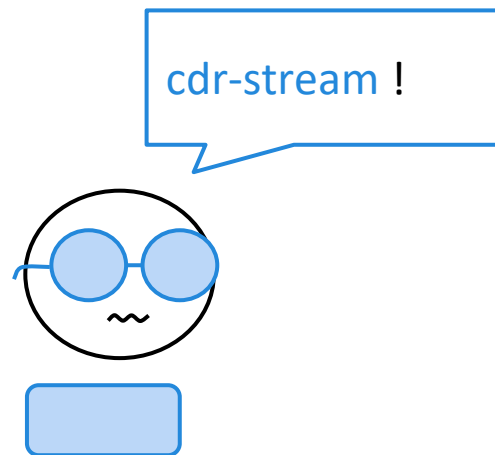
```
scm> (cdr-stream s)
```

```
(2 . #[promise (not forced)])
```

```
scm> (cdr-stream (cdr-stream s))
```

```
()
```

- `cdr-stream` forces Scheme to compute the rest



Streams

Remember, a **stream** is just a regular Scheme pair whose
second element is a **promise**

```
scm> (define s (cons-stream 1 (cons-stream 2 nil)))  
s  
scm> (cdr s)  
#[promise (not forced)]  
scm> (cdr-stream s)  
(2 . #[promise (not forced)])  
scm> (cdr-stream (cdr-stream s))  
()
```


Promises: delay

- Promise: an object that delays evaluation of an expression
 - The `delay` special form creates promises

```
scm> (print 5)  
5
```

(print 5) is immediately
evaluated

```
scm> (delay (print 5))  
#[promise (not forced)]
```

(print 5) is not
evaluated yet

Promises: force

- The `delay` special form creates promises
- The `force` procedure evaluates the expression inside the promise

```
scm> (define x (delay (print 5)))
```

```
x
```

```
scm> x
```

```
#[promise (not forced)]
```

```
scm> (force x)
```

```
5
```

```
scm> x
```

```
#[promise (forced)]
```

(print 5) is not
evaluated yet

Evaluates (print 5)

Error, in our interpreter

Promises

`cons-stream` and `cdr-stream` are syntactic sugar.
Achieve the same effect with `delay` and `force`

```
scm> (define s (cons-stream 1 nil))
s
scm> s
(1 . #[promise (not forced)])
scm> (cdr-stream s)
()

scm> (define s (cons 1 (delay nil)))
s
scm> s
(1 . #[promise (not forced)])
scm> (force (cdr s))
()
```

Recursively Defined Streams - Constant Stream

Let's start with the constant stream. A constant stream is an infinitely long stream with a number repeated.

```
(define (constant-stream i)
  (cons-stream i (constant-stream i)))
```

```
scm> (define ones (constant-stream 1))
scm> (car ones)
1
scm> (car (cdr-stream ones))
1
```

Check Your Understanding: Natural Number Stream

Let's define the naturals stream which is an infinitely long stream with the natural numbers starting at start.

```
(define (nats start)
```

Demo_1

```
-----)
```

```
scm> (define s (nats 0))
scm> (car s)
0
scm> (car (cdr-stream s))
1
scm> (car (cdr-stream (cdr-stream s)))
2
```

Natural Number Stream

```
(define (nats start)
  (cons-stream start (nats (+ start 1))))
```

Add-Stream and Ints-Stream

Let's write a function that will add two infinite streams together and return a new stream.

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

Let's see it in action! Let's first define the ones stream again.

```
(define ones (cons-stream 1 ones))
```

This is the same as (constant-stream 1).

Let's use the ones stream and our new add-stream function to define the ints stream. This is the same as (nats 1). How do we do this?

```
(define ints (cons-stream 1 (add-stream ? ?)))
```

Demo_2

Ints-Stream Solution

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

ones:	1	1	1	1	1	...
	+	+	+	+		
ints:	1	2	3	4	5	...

We can use infinite streams to build other infinite streams. This is the power of lazy evaluation, our current stream stays one step ahead of itself!

Examples: map-stream

- Implement `(map-stream fn s)`:
 - `fn` is a one-argument function
 - `s` is a stream
- Returns a new stream with `fn` applied to elements of `s`

```
(define (map-stream fn s)  
  'YOUR-CODE-HERE  
)
```

Demo_3

Examples: map-stream

- How would you implement `map-list`?

```
(define (map-list fn s)
  (if (null? s)
      nil
      (cons (fn (car s))
            (map-list fn (cdr s)))))
```

- How about `map-stream`?

```
(define (map-stream fn s)
  (if (null? s)
      nil
      (cons-stream (fn (car s))
                   (map-stream fn (cdr-stream s)))))
```

What happens if you change this to `cons`?

Examples: stream-to-list

- Implement `(stream-to-list s num-elements)`:
 - `s` is a stream
 - `num-elements` is a non-negative integer
- Returns a Scheme list containing the first `num-elements` elements of `s`

```
scm> (stream-to-list (ints 1) 10)
(1 2 3 4 5 6 7 8 9 10)
```

```
(define (stream-to-list s num-elements)
  'YOUR-CODE-HERE
)
```

Demo_4

Examples: stream-to-list

```
(define (stream-to-list s num-elements)
  (if (or (null? s) (= num-elements 0))
      nil
      (cons (car s)
            (stream-to-list (cdr-stream s)
                           (- num-elements 1))))
)
```