Lazy Evaluation & Infinite Data

COS 326
David Walker
Princeton University

Some ideas in this lecture borrowed from Brigitte Pientka, McGill University

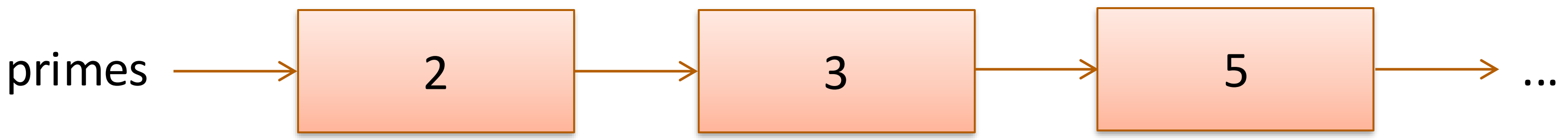
AN INFINITE DATA STRUCTURE: STREAMS

Streams

Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...

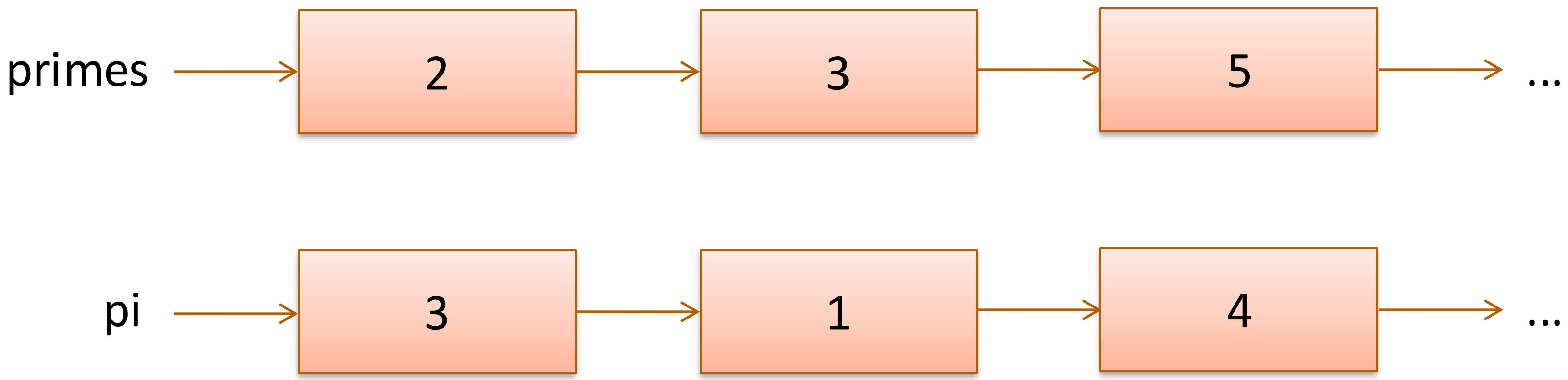
Streams

Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...



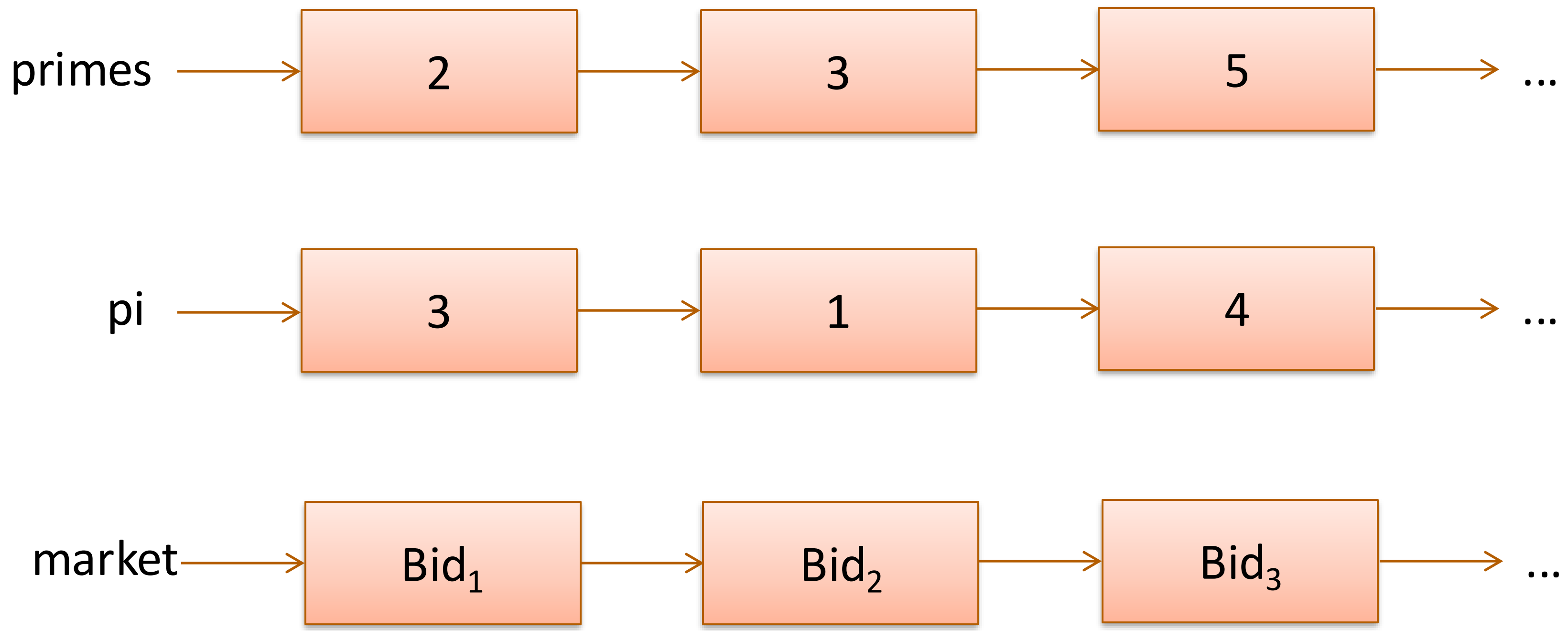
Streams

Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...



Streams

Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...



Consider this definition:

```
type 'a stream =
  Cons of 'a * ('a stream)
```

We can write functions to extract the head and tail of a stream:

```
let head(s:'a stream):'a =
  match s with
  | Cons (h,_) -> h

let tail(s:'a stream):'a stream =
  match s with
  | Cons (_,t) -> t
```

But there's a problem…

```
type 'a stream =
  Cons of 'a * ('a stream)
```

But how do we build a value of type 'a stream?

Cons (3, ___)

Cons (3, Cons (4, ___))

But there's a problem…

```
type 'a stream =
  Cons of 'a * ('a stream)
```

But jow do we build a value of type 'a stream?

Cons (3, ___)

Cons (3, Cons (4, ___))

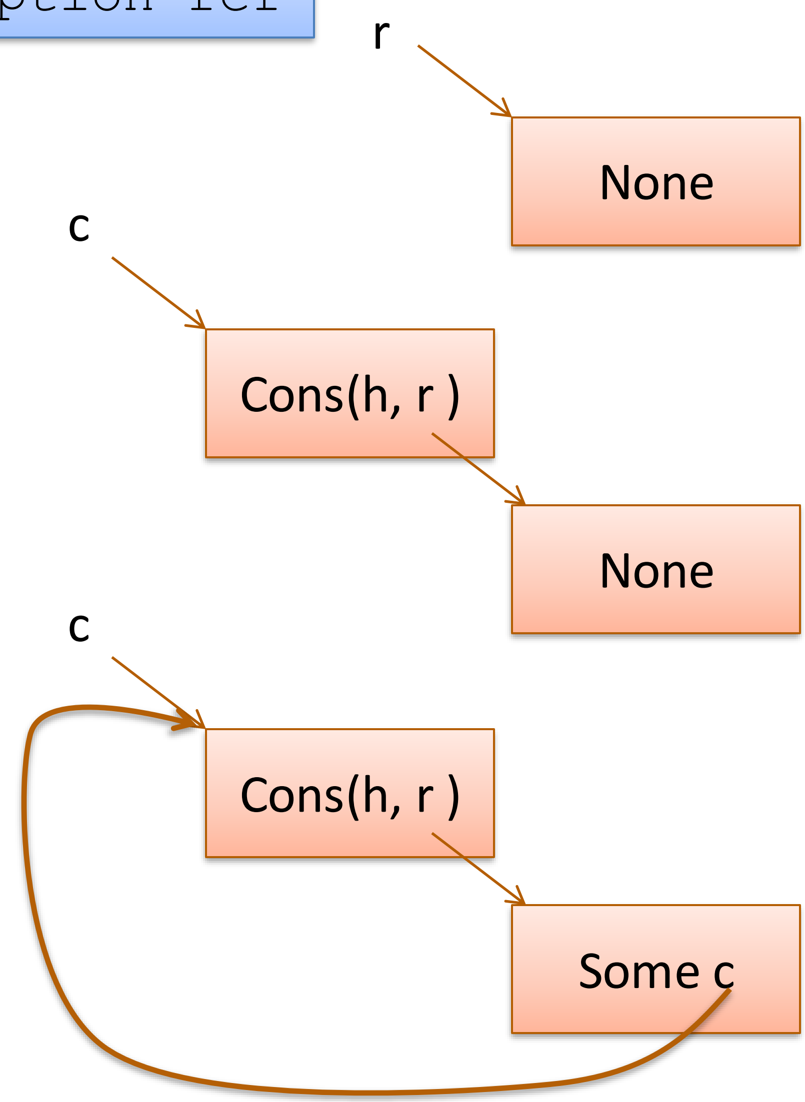
There doesn't seem to be a base case (e.g., Nil or [])

Since we need a stream to build a stream,
what can we do to get started?

One Possibility: Use Refs

```
type 'a stream =
  Cons of 'a * ('a stream) option ref
```

```
let circular_cons h =
  let r = ref None in
  let c = Cons(h,r) in
  (r := (Some c); c)
```



An alternative would be to use refs

```
type 'a stream =
  Cons of 'a * ('a stream) option ref
```

```
let circular_cons h =
  let r = ref None in
  let c = Cons(h,r) in
  (r := (Some c); c)
```

Problem: The type system can't guarantee that the tail of the list exists. And every time we get the tail of the stream, we have to check it is there.

Using functions to build streams

```
type 'a stream =
  Cons of 'a * ('a stream)
```

This function can create the stream of all natural numbers:

```
let rec nats i = Cons(i,nats (i+1))
```

```
# let n = nats 0;;
Stack overflow during evaluation (looping recursion?).
```

OCaml evaluates our code just a little bit too *eagerly*.
We want to evaluate the right-hand side *only when necessary* ...

Another idea

One way to implement "waiting" is to wrap a computation up in a function and then call that function later when we want to.

Another attempt:

```
type 'a stream =
  Cons of 'a * ('a stream)
```

```
let rec ones =
  fun () -> Cons(1,ones)
```

```
let head x =
  match x () with
    Cons (hd, tail) -> hd
```

Are there any problems with this code?

Darn. Doesn't type check!

ones : unit -> int stream

not int stream

Functional Implementation

What if we changed the definition of streams one more time?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

mutually recursive
type definition

```
let rec ones : int stream =
  fun () -> Cons(1,ones)
```

Or, the way we'd normally write it:

```
let rec ones () = Cons(1,ones)
```

Functional Implementation

How would we define head, tail, and map of an ’a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let head(s:'a stream):'a =
...
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let head(s:'a stream):'a =
 match s() with
 | Cons(h,_) -> h
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let head(s:'a stream):'a =
 match s() with
  | Cons(h,_) -> h
```

```
let tail(s:'a stream):'a stream =
  ...
```

Functional Implementation

How would we define head, tail, and map of an ’a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let head(s:'a stream):'a =
 match s() with
 | Cons(h,_) -> h
```

```
let tail(s:'a stream):'a stream =
 match s() with
 | Cons(_,t) -> t
```

Functional Implementation

How would we define head, tail, and map of an ’a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec map (f:'a->'b) (s:'a stream) : 'b stream =
...
```

Functional Implementation

How would we define head, tail, and map of an ’a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec map (f:'a->'b) (s:'a stream) : 'b stream =
  Cons(f (head s), map f (tail s))
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec map (f:'a->'b) (s:'a stream) : 'b stream =
  Cons(f (head s), map f (tail s))
```

Rats!

Infinite looping!

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec map (f:'a->'b) (s:'a stream) : 'b stream =
  Cons(f (head s), map f (tail s))
```

Doesn't type check!
Cons (x,y) is a **str** not a **stream**

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec map (f:'a->'b) (s:'a stream) : 'b stream =
  fun () -> Cons(f (head s), map f (tail s))
```

Importantly, map must return a function, which delays evaluating the recursive call to map.

Functional Implementation

How would we define head, tail, and map of an ’a stream?

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec map (f:'a->'b) (s:'a stream) : 'b stream =
  fun () -> Cons(f (head s), map f (tail s))
```

```
let rec ones = fun () -> Cons(1,ones)

let inc x = x + 1

let twos = map inc ones
```

Functional Implementation

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec map (f:'a->'b) (s:'a stream) : 'b stream =
  fun () -> Cons(f (head s), map f (tail s))
```

```
let rec ones = fun () -> Cons(1,ones)
let twos = map (fun x -> x+1) ones
```

head twos
--> head (map inc ones)
--> head (fun () -> Cons (inc (head ones), map inc (tail ones)))
--> match (fun () -> ...) () with Cons (hd, _) -> h
--> match Cons (inc (head ones), map inc (tail ones)) with Cons (hd, _) -> h
--> match Cons (inc (head ones), fun () -> ...) with Cons (hd, _) -> h
--> ... --> 2

Functional Implementation

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec zip f s1 s2 =
  fun () ->
   Cons(f (head s1) (head s2),
        zip f (tail s1) (tail s2))
```

Functional Implementation

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec zip f s1 s2 =
  fun () ->
   Cons(f (head s1) (head s2),
        zip f (tail s1) (tail s2))
```

```
let threes = zip (+) ones twos
```

Functional Implementation

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

```
let rec zip f s1 s2 =
  fun () ->
   Cons(f (head s1) (head s2),
        zip f (tail s1) (tail s2))
```

```
let threes = zip (+) ones twos
```

```
let rec fibs =
  fun () ->
   Cons(0, fun () ->
           Cons (1,
                 zip (+) fibs (tail fibs)))
```

Unfortunately

This is not very efficient:

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

Every time we want to look at a stream (e.g., to get the head or tail), we have to re-run the function.

Unfortunately

This is not very efficient:

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```

Every time we want to look at a stream (e.g., to get the head or tail), we have to re-run the function.

```
let x = head s
let y = head s
```

rerun the *entire* underlying function as opposed to fetching the first element of a list

```
let head(s:'a stream):'a =
 match s() with
 | Cons(h,_) -> h
```

Unfortunately

This is really, really inefficient:

```
let rec fibs =
  fun () ->
   Cons(0, fun () ->
           Cons (1,
                zip (+) fibs (tail fibs)))
```

So when you ask for the 10th fib and then the 11th fib, we are re-calculating the fibs starting from 0...

If we could *cache* or *memoize* the result of previous fibs...

LAZY EVALUATION

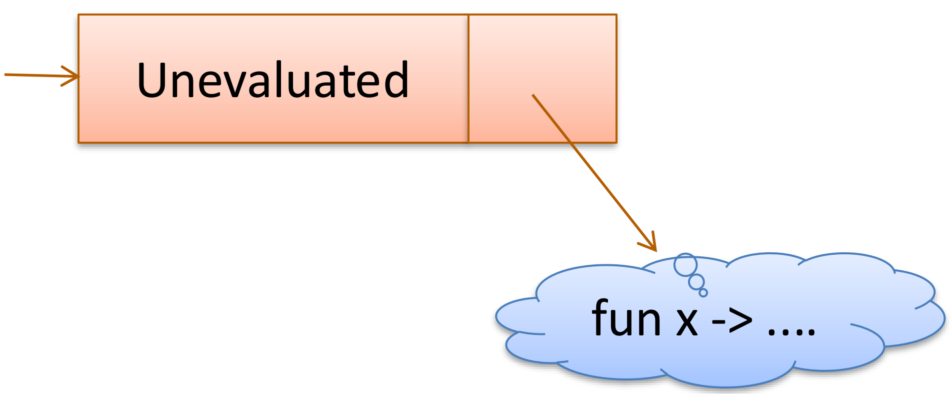
Lazy Data

We can take advantage of mutation to memoize:

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
```

```
type 'a lazy = 'a thunk ref
```

initially:



after evaluating once:

Lazy Data

We can take advantage of mutation to memoize:

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
```

```
type 'a lazy = 'a thunk ref
```

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) lazy_t
```

Lazy Data

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
```

```
type 'a lazy = 'a thunk ref
```

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) lazy
```

```
let rec head(s:'a stream):'a =
```

Lazy Data

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
```

```
type 'a lazy = 'a thunk ref
```

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) lazy
```

```
let rec head(s:'a stream):'a =
  match !s with
   | Evaluated (Cons(h,_)) ->
   | Unevaluated f ->
```

Lazy Data

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
```

```
type 'a lazy = 'a thunk ref
```

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) lazy
```

```
let rec head(s:'a stream):'a =
  match !s with
   | Evaluated (Cons(h,_)) -> h
   | Unevaluated f ->
```

Lazy Data

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
```

```
type 'a lazy = 'a thunk ref
```

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) lazy
```

```
let rec head(s:'a stream):'a =
  match !s with
   | Evaluated (Cons(h,_)) -> h
   | Unevaluated f ->
      let x = f() in (s := Evaluated x; head s)
```

Lazy Data

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
```

```
type 'a lazy = 'a thunk ref
```

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) lazy
```

```
let rec tail(s:'a stream) : 'a stream =
  match !s with
   | Evaluated (Cons(_,t)) -> t
   | Unevaluated f ->
     (let x = f () in s := Evaluated x; tail s)
```

Lazy Data

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
```

```
type 'a lazy = 'a thunk ref
```

```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) lazy
```

```
let rec tail(s:'a stream) : 'a stream =
  match !s with
  | Evaluated (Cons(_,t)) -> t
  | Unevaluated f ->
     let x = f() in (s := Evaluated x; tail s)
```

```
let rec head(s:'a stream):'a =
  match !s with
  | Evaluated (Cons(h,_)) -> h
  | Unevaluated f ->
     let x = f() in (s := Evaluated x; head s)
```

Lazy Data

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
```

```
type 'a lazy = 'a ... r
```

```
type 'a ... Con
and 'a
```

```
|
...ail s)
```

Common pattern!

Dereference & check if evaluated:

- If so, take the value.
- If not, evaluate it & take the value

```
...'a
  | Eva...d (...s(h,_)) -> h
  | Une...uated f ->
     let x = f() in (s := Evaluated x; head s)
```

Memoizing Streams

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a
type 'a lazy_t = ('a thunk) ref

type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) lazy

let rec force(t:'a lazy_t):'a =
  match !t with
  | Evaluated v -> v
  | Unevaluated f ->
      let v = f() in
      (t:= Evaluated v ; v)

let head(s:'a stream) : 'a =
  match force s with
  | Cons(h,_) -> h

let tail(s:'a stream) : 'a stream =
  match force s with
  | Cons(_,t) -> t
```

Memoizing Streams

```
type 'a thunk =
  Unevaluated of unit -> 'a | Evaluated of 'a

type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) thunk ref

let rec ones =
  ref (Unevaluated (fun () -> Cons(1,ones)))
```

Memoizing Streams

```
type 'a thunk =
  Unevaluated of unit -> 'a | Evaluated of 'a

type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) thunk ref

let lazy f = ref (Unevaluated f)

let rec ones =
  lazy (fun () -> Cons(1,ones))
```

Our Interface

```
type 'a lazy

val  lazy : (unit -> 'a) -> 'a lazy

val  force :  'a lazy -> 'a
```

Laziness Built-in to OCaml

Our interface

```
let y = lazy (fun () -> print "hello"; 3)

              …

let _ = force y             (* prints now *)
```

don't need to wrap in function

OCaml built-in support for type 'a lazy_t

```
let y = lazy (print "hello"; 3)

              …
let _ = Lazy.force y      (* prints now *)
```

Summary

By default, OCaml (and Java, C, etc) is an eager language

- but you can use thunks or “lazy” to suspend computations
- use “force” to run the computation when needed

By default, Haskell is a lazy language

- the implementers (eg: Simon Peyton Jones) would probably make it eager by default if they had a do-over
- working with infinite data is generally more pleasant
- but difficult to reason about space and time

Lazy evaluation makes it possible to build *infinite data structures*.

- can be modelled using functions
- but adding refs allows memoization

END