Lec 20 Lazy

* Dealing with infinite data structure.

Consider `f` vs `(fn x => f x)`. Are they the same?

→ `f` could be expression, not value, but `(fn x => f x)` can only be value

```
fun loop x = loop x
```

`loop ()` ← loops `(fn x => (loop ()) x)` ← just waits there

Suspension

Def A **suspension** of type `t` is a func of type `unit -> t`

Let `e : t` (any expression) `(fn () => e) : unit -> t` is a "**t suspension**"

"**lazy**" representation of `e`

Calling a suspension on `()` is called "**forcing**" the suspension

Def A **stream** is some infinite data structure.

Sig for stream

```
Signature STREAM =
  sig
    type 'a stream          (* abstract *)
    datatype 'a front = Empty | Cons of 'a * 'a stream
    val expose : 'a stream -> 'a front
    val delay : (unit -> 'a front) -> 'a stream
```

```
    val  empty : 'a stream
    val  null : 'a stream → bool
    val  take : 'a stream * int → 'a list
    val  map : ('a → 'b) → 'a stream → 'b stream
    val  filter : ('a → bool) → 'a stream → 'a stream
    ⋮
  end

Structure Stream :> STREAM =
  struct
    datatype 'a stream = Stream of (unit → 'a front)      ← suspension of front
    and 'a front = Empty | Cons of 'a * 'a stream
    fun expose (Stream d) = d()
    fun delay d = Stream d
    val empty = delay (fn () => Empty)
    fun null s = (case expose s of Empty => true | _ => false)
          ↑ could loop!
    ⋮
  end

Structure S = Stream
```

Example usage

```
fun ones' () = S.Cons (1, S.delay (fn () => ones' ()))      : ones' : unit → int S.front
val ones = S.delay (fn () => ones' ())                      : int S.stream

fun natsFrom' x = S.Cons (x, S.delay (fn () => natsFrom' (x+1)))
fun natsFrom x = S.delay (fn () => natsFrom' x)
val nats = natsFrom 0
```

More jargons

Def Streams X, Y are eeq if

S.take(X, n) ≅ S.take(Y, n) for all n

Def Stream X is productive if S.expose X returns

1. Empty or
2. Cons(x, X') with productive X'

Def Stream X is infinite if it's productive and we never get Empty

Map

```
(* map: ('a → 'b) → ('a Stream) → ('b Stream)  *)
fun map f s = delay (fn () ⇒ map' f (expose s))
and map' f Empty = Empty
  | map' f (Cons(x, s')) = Cons(f x, map f s')
```

map': ('a → 'b) → 'a front → 'b front

```
(* filter : ('a → bool) → 'a Stream → 'a Stream  *)
fun filter p s = delay (fn () ⇒ filter' p (expose s))
and filter' p Empty = Empty
  | filter' p (Cons(x, s')) =
      if p x then
        Cons(x, filter p s')
      else
        filter' p (expose s')
```

Example usage

```
val evens = S.map (fn x => 2*x) nats
val [0,2,4] = S.take (evens, 3)
val ns = S.filter (fn x => x<0) nats      ← almost instantaneous
val _ = S.expose ns                       ← loops forever.
                                            Notice ns is empty stream
                                            but the code cannot tell.
```

Code to compute all prime


```
fun sieve s = S.delay (fn () => sieve' (S.expose))
and sieve' S.Empty = S.Empty
  | sieve' (S.Cons (p, s)) =
        S.Cons (p, sieve (S.filter (fn x => x mod p <> 0) s))
val primes = sieve (natsFrom 2)
```