Suspension

`f` v.s. `(fn x => f x)`

`f`: may not be valuable (if is exp)

`(fn x => f x)`: `f` is not evaluated until the lambda expression is called on an argument and the body `f x` is evaluated; always valuable

eg. `fun g x = g x`

`f` is the exp `(g 3)`

- `g 3` → loops `g : int -> 'a`
- `(fn x => (g 3)) x` → a value (closure) `(g 3) : 'a -> 'b`

Suspension A suspension of type τ is a function of type `unit -> τ`

when apply to `()`, the suspension is forced.

eg. `e : t`, `fn () => e` is a suspension of type `t`

Stream

Base case: empty stream

Inductive case: a suspension of "a single element consed onto another stream"

Co-Induction no base case

↳ suspension allow us to do so

```
signature STREAM =
sig
  type 'a stream (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream
```

the result of performing just enough computation to expose the first element of a stream, and obtain the rest

```
  val expose : 'a stream -> 'a front
```

performing just enough computation to expose the first element of a stream

returns the corresponding front

⚠ involves computation, may not terminate

```
  val delay : (unit -> 'a front) -> 'a stream
```

delay (fn () => e), suspension enables lazy evaluation

```
  val empty : 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val null : 'a stream -> bool
```

~~test~~ whether a stream is empty

involves stream exposure, may not terminate

```
  val take : ('a stream * int) -> 'a list
```

return the first n elements of stream s, raise Subscript if encounter Empty

```
  val map : ('a -> 'b) -> 'a stream -> 'b stream
```

lazy!! map f s → a stream s', but ✗ apply f to any element of s not until someone exposes s'

```
  val filter : ('a -> bool) -> 'a stream -> 'a stream
end
```

```
structure Stream : STREAM =
struct
    datatype 'a stream = Stream of unit -> 'a front
    and 'a front = Empty | Cons of 'a * 'a stream
```

mutually recursive

```
    fun delay (d) = Stream (d)
    fun expose (Stream (d)) = d ()
    val empty = Stream (fn () => Empty)
    fun cons (x,s) = Stream (fn () => Cons (x,s))
    fun null
    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')

    fun filter f s = delay (fn () => filter' f (expose s))
    and filter' f Empty = Empty
      | filter' f (Cons (x,s')) = if (p x) then Cons (x, filter p s')
                                  else filter' p (expose s')
```

Lazyness !

Lazyness !

stream (fn () => Cons (x, s))

the stream's element ← t + stream type

+ front

+ front suspension

+ stream

```
structure S = Stream
```

① implement an infinite stream, all elements are 1 :

```
fun ones' () = S.Cons (1, S.delay ones')
val ones = S.delay ones'
```

ones' : unit -> int S.front

ones : unit -> int S.stream ?

② implement an infinite stream consisting of all ℕ

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

val S.Cons (x, rest) = S.expose nats
val S.Cons (y, _) = S.expose rest
```

} ⇒ 0/x, 1/y, rest is a stream consisting of all ≥ 2

nat' : int -> unit -> int S.front

③ `val evens = S.map (fn n => 2 * n) nats` : int S.stream

`val [0,2,4] = S.take (evens, 3)`

④ `val ns = S.filter (fn n => n < 0) nats` : int S.stream

is a value, computes instantaneously

S.expose ns loops forever

⑤ All the primes (the Sieve of Eratosthenes)

```
fun notDivides p q = (q mod p <> 0)
fun sieve s = S.delay (fn () => sieve' (S.expose s))
and sieve' (S.Empty) = S.Empty
  | sieve' (S.Cons (p, s)) = S.Cons (p, sieve (S.filter (notDivides p) s))
```

Stream Equivalence $X \cong Y$ if take $(X, n) \cong$ take (Y, n) $\forall n \in \mathbb{N}$

productive Stream.expose (s) → Empty
Cons (,)