

Streeeeeeeeeeeeeeeeeeeeeeams

Echo Recitation, X, 28 March 2007

- Stay hydrated, mmmkay?

- This function is impractical:

```
fun useful (weekday,  
            trillionthPrime:int,  
            trillionthDigitOfPi:int) =  
  if weekday = Tuesday  
    then trillionthPrime  
    else trillionthDigitOfPi
```

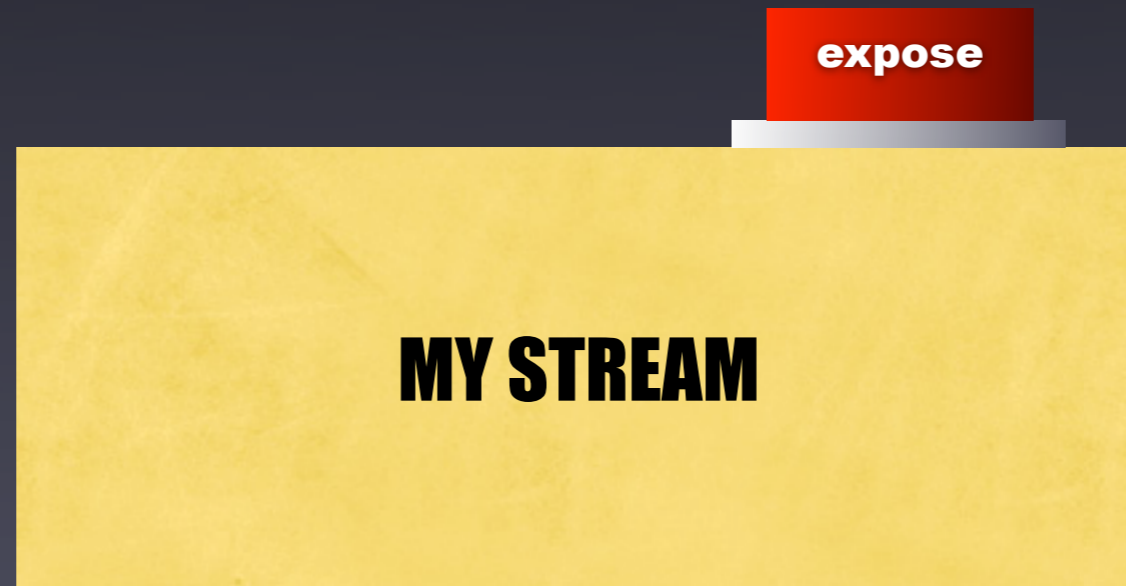
- You have to do a lot of computation before you can call it.
- Half of the results of the computation is always discarded.
- What can you do about this?

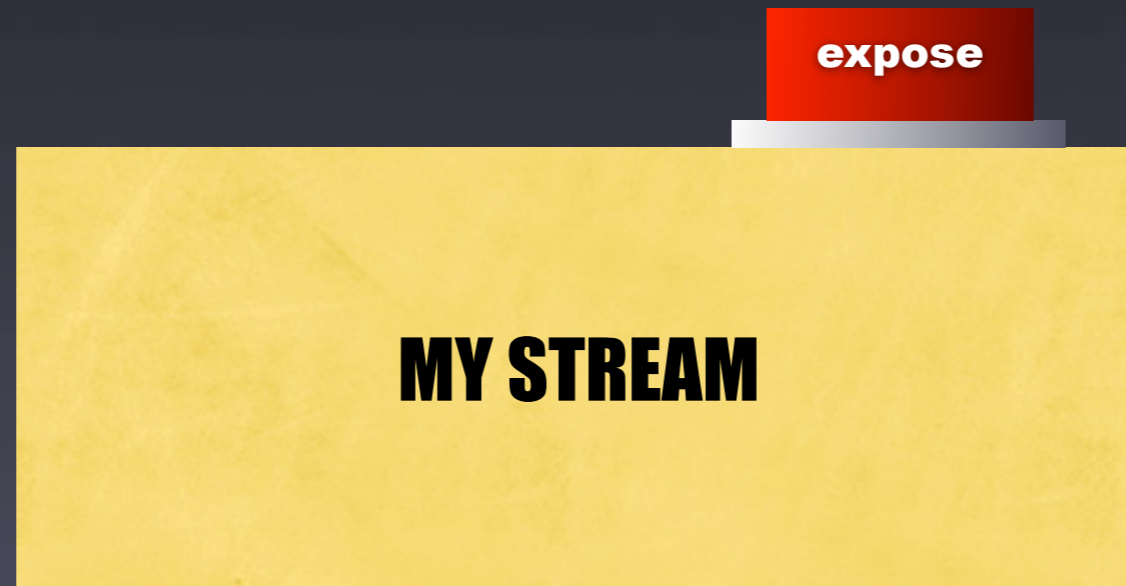
- Solution: delay computation till you're sure you need it.

```
fun useful (weekday,  
            findTrPrime:unit->int,  
            findTrDigitOfPi:unit->int) =  
  if weekday = Tuesday  
  then findTrPrime()  
  else findTrDigitOfPi()
```

- Pass in computation as “thunks”—argumentless functions.
- Only compute the value (evaluate the thunk) you need.
- win!

- Stream: a potentially infinite list-like data structure with nothing revealed.





- Stream: a potentially infinite list-like data structure with nothing revealed.
- Front: a potentially infinite list-like data structure with only the first element revealed



- Stream: a potentially infinite list-like data structure with nothing revealed.
- Front: a potentially infinite list-like data structure with only the first element revealed

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

- Streams look like `Stream(fn () => ...:'a front)`
- Fronts look like `Empty` or `Cons(x:'a, s:'a stream)`

```
Stream(fn () => Cons(x:'a, rest:'a stream))
```



Suspension

has type `unit -> 'a front`

```
Stream(fn () => Cons(x:'a, rest:'a stream))
```

Suspension

has type `unit -> 'a front`

- Applying a suspension to `()`: forcing the suspension.
 - `fun delay(susp) = Stream(susp) (* Delay *)`
 - `fun expose(Stream(susp)) = susp() (* FORCE *)`

- Natural numbers
- `map`
- `exists`
- `append`
- `filter`
- folding (`fold`s)

- Tandem pairs of functions: one with a delayed suspension (which is a stream) and one with a front.

```
fun foobar args = delay ( fn () => foobar' ..args... )  
and foobar' args = Cons(..., foobar ..args...)
```

- Example: counting upwards/natural numbers

```
fun countup x = delay ( fn () => countup' x )  
and countup' x = Cons(x, countup(x+1))  
  
val nat = countup 0 (* Natural numbers *)
```

- Just like lists: map applies a function to elements in the stream. E.g.:

```
val evens = map (fn x => 2*x) nat
```

- How do we achieve with lazy computation?

```
fun map f s = delay( fn () => map' _____ )  
and map' _____ Empty = Empty  
    | map' _____ = Cons(_____, _____)
```

- Just like lists: map applies a function to elements in the stream. E.g.:

```
val evens = map (fn x => 2*x) nat
```

- How do we achieve with lazy computation?

```
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)
```

- What if map didn't delay? (note: conceptual, does not typecheck)

```
fun map f s = map' f (expose s)
and map' f      Empty = Empty
    | map' f Cons (f, s) = Cons (f x, map f s)
```

- **exists**: returns true if an element satisfying the predicate **p** exists in the stream. Type: ('a -> bool) -> 'a stream -> bool

```

fun exists p s = _____
and exists' p Empty = _____
    | exists' p (Cons(x,s)) = _____
    
```

- **filter**: returns a stream with only the elements that satisfy **p**. Type: ('a -> 'b) -> 'a stream -> 'b stream

```

fun filter p s = _____
and filter' p Empty = _____
    | filter' _____ =
        if _____ then _____
        else _____
    
```

- **exists**: returns true if an element satisfying the predicate **p** exists in the stream. Type: ('a -> bool) -> 'a stream -> bool

```
fun exists p s = exists' p (expose s)
and exists' p Empty = false
    | exists' p (Cons(x,s)) = (p x) orelse exists p s
```

- **filter**: returns a stream with only the elements that satisfy **p**. Type: ('a -> bool) -> 'a stream -> 'a stream

```
fun filter p s =
    _____
and filter' p Empty = _____
    | filter' _____ =
        if _____ then _____
          else _____
```

- `exists`: returns true if an element satisfying the predicate `p` exists in the stream. `Type: ('a -> bool) -> 'a stream -> bool`

```
fun exists p s = exists' p (expose s)
and exists' p Empty = false
    | exists' p (Cons(x,s)) = (p x) orelse exists p s
```

- `filter`: returns a stream with only the elements that satisfy `p`. `Type: ('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)
```

- The problem with folding streams: they're (maybe) infinite!
- Strategy: *delay* the rest of the fold.
- Normal `fold1` for lists takes functions of this type:
 - `('a * 'b -> 'b)`
- `fold`s takes a function of this type:
 - `('a * (unit -> 'b) -> 'b)`
- Note the thunk: accumulator computation is delayed!
- This lets you handle infinite streams.


```
- fun sumtoten (item, accfn) = if item > 10 then 0  
else item + accfn();
```

```
val sumtoten = fn : int * (unit -> int) -> int
```

```
- fun sumtoten (item, accfn) = if item > 10 then 0  
else item + accfn();
```

```
val sumtoten = fn : int * (unit -> int) -> int
```

```
- folds sumtoten 0 nat;
```

```
- fun sumtoten (item, accfn) = if item > 10 then 0  
else item + accfn();
```

```
val sumtoten = fn : int * (unit -> int) -> int
```

```
- folds sumtoten 0 nat;
```

```
val it = fn : unit -> int
```

```
- fun sumtoten (item, accfn) = if item > 10 then 0  
else item + accfn();
```

```
val sumtoten = fn : int * (unit -> int) -> int
```

```
- folds sumtoten 0 nat;
```

```
val it = fn : unit -> int
```

```
- it();
```

```
- fun sumtoten (item, accfn) = if item > 10 then 0  
else item + accfn();
```

```
val sumtoten = fn : int * (unit -> int) -> int
```

```
- folds sumtoten 0 nat;
```

```
val it = fn : unit -> int
```

```
- it();
```

```
val it = 55 : int
```


- Type of `foldl` (for lists):

`('a * 'b -> 'b) -> 'b -> 'a list -> 'b`

- Type of `fold`s:

`('a * (unit -> 'b) -> 'b) -> 'b -> 'a stream -> (unit -> 'b)`

- Type of `foldl` (for lists):

```
('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

- Type of `folds`:

```
('a * (unit -> 'b) -> 'b) -> 'b -> 'a stream -> (unit -> 'b)
```

- Implementation of `folds`:

```
fun folds f b s = (fn () => folds' f b (expose s))
and folds' f b Empty = b
  | folds' f b (Cons(x,s)) = f (x, folds f b s)
```

- And our function that uses it...

```
fun sumtoten (item, accfn) = if item > 10 then 0 else
item + accfn();
```