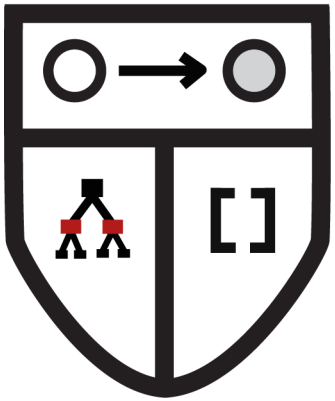

CS 3110

Streams and Laziness

Prof. Clarkson

Fall 2018

Today's music: "Lazy Days" by Enya

Attendance question

What is the type of **`f`**?

```
let rec f x = f x
```

A. Doesn't compile

B. `'a -> 'a`

C. `'a -> 'b`

D. `unit -> unit`

Review

Previously in 3110:

- Functional programming
- Modular programming

Third unit of course: Data structures

Today:

- Streams
- Laziness

INFINITE LISTS

Demo

Discussion

How can an infinite length list fit in a finite computer memory?

"Infinite" data structures

- Sequences of numbers: the naturals, primes, Fibonacci, …
- Data processed by a program: from a file, from the user, from the network
- Game tree (for some games):
  - nodes = game positions
  - edges = legal moves

Question

What does `nats` evaluate to?

```
(* [from n] is the infinite list [[n; n+1; ...]] *)
let rec from n = n :: from (n+1)

let nats = from 0
```

A. [0; 1; 2; …]
B. Never terminates (infinite loop)
C. Exception
D. Stack overflow

aka infinite lists, sequences, delayed lists, lazy lists

STREAMS

List representation

```
(** An ['a mylist] is a finite
    list of values of type
    ['a]. *)
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist
```

Stream representation?

```
(** An ['a stream] is an infinite
    list of values of type
    ['a]. *)
type 'a stream =
  | Nil
  | Cons of 'a * 'a stream
```

Stream representation?

```
(** An ['a stream] is an infinite
    list of values of type
    ['a]. *)
type 'a stream =
  | Nil
  | Cons of 'a * 'a stream
```

Stream representation?

```
type 'a stream =
   | Cons of 'a * 'a stream
```

Try coding these if possible:

- the stream of 1's
- the stream of natural numbers

Key idea of this entire lecture:

Delay evaluation

Demo

thunk

```
fun () -> (* a delayed computation *)
```

Stream representation

```
(** An ['a stream] is an infinite list
      of values of type ['a].
    AF:  [Cons (x, f)] is the stream
      whose head is [x] and tail is
      [f()].
    RI:  none *)
type 'a stream =
  Cons of 'a * (unit -> 'a stream)
```

Demo

Notation

Write

`<a; b; c; …>`

to mean stream whose first elements are `a`, `b`, `c`.

Discussion

```
(** [sum <a1; a2; ...> <b1; b2; ...>]
    is [<a1 + b1; a2 + b2; ...>] *)
let rec sum
  (Cons (h_a, tf_a))
  (Cons (h_b, tf_b))
=
  ?
```

Discussion

```
(** [map f <a; b; c; ...>] is
    [<f a; f b; f c; ...>] *)
let rec map f (Cons (h, tf)) =
  ?
```

A CUTE FIBONACCI TRICK

Fibonacci

fibs 1 1 2 3 5 8 …

Fibonacci

fibs	1	1	2	3	5	8	…
fibs	1	1	2	3	5	8	…

Fibonacci

fibs	1	1	2	3	5	8	…
tl fibs	1	2	3	5	8	13	…

Fibonacci

fibs	1	1	2	3	5	8	…
tl fibs	1	2	3	5	8	13	…
	2	3	5	8	13	21	…

fibs is
1 then
1 then
(fibs + tl fibs)

Fibonacci

```
let rec fibs =
  Cons(1, fun () ->
    Cons(1, fun () ->
      sum fibs (tl fibs)))
```

But try: `take 100 fibs`

Exponential amount of recomputation: regenerate entire prefix of `fibs`, twice, for each element produced

Solution: the Lazy module, covered in textbook

Upcoming events

- N/A

This is judiciously lazy.

THIS IS 3110