CS 3110

# Streams and Laziness

Nate Foster

Spring 2018

# Review

**Previously in 3110:**

- Functional programming
- Modular programming

**Third unit of course:** Data structures

**Today:**

- Streams
- Laziness

# What is this?

```
let rec ones = 1 :: ones
```

# Infinite list

```
let rec ones = 1 :: ones

tl ones
-->
tl (1 :: ones)
-->
ones
```

# Infinite list

```
let rec a = 0 :: b
    and b = 1 :: a
```

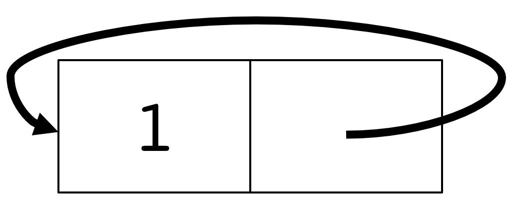
```
a = [0;1;0;1;...]
b = [1;0;1;0;...]
```

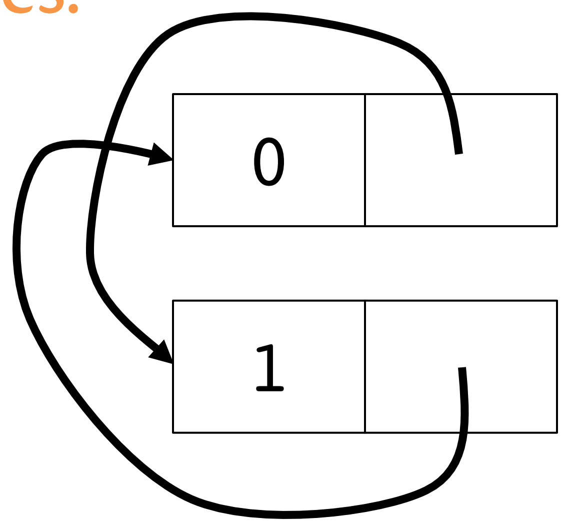
# Infinite list

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

**A:** It can't.

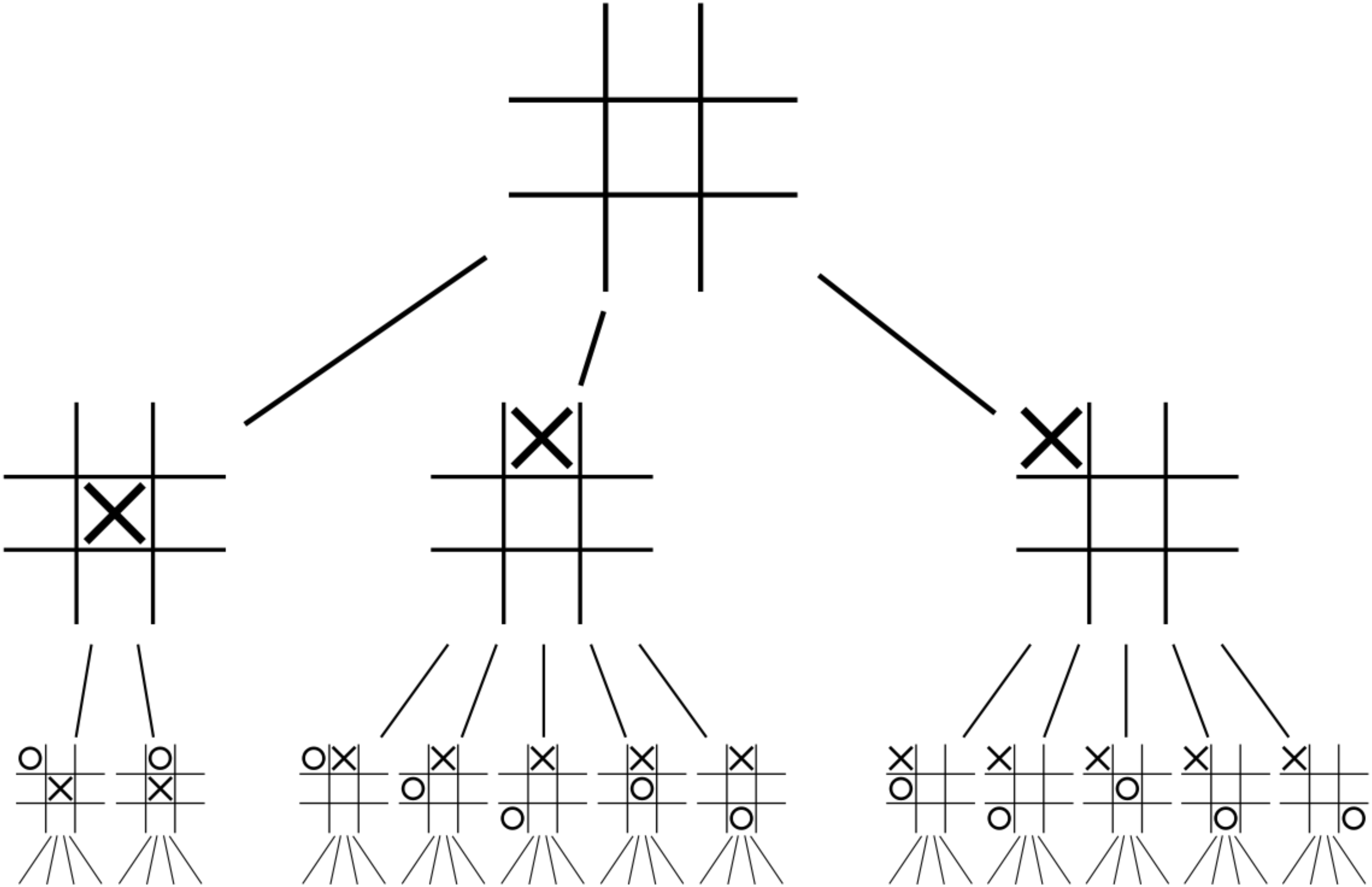
But linked lists can have cycles!



# 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



(game tree is actually finite for tic-tac-toe)

# Infinite list

Q: Could we use *recursive values* to define the infinite list of natural numbers?

```
# let rec nats = 0 :: (* [1;2;3;...] *);;
```

`nats` should be `[0;1;2;3;...]`

so

`List.map (fun x -> x+1) nats`
should be
`[1;2;3;4;...]`

# Infinite list

Q: Could we use *recursive values* to define the infinite list of natural numbers?

```
# let rec nats = 0 :: List.map (fun x -> x+1) nats;;
Error: This kind of expression is not allowed as right-
hand side of let rec
```

A: No. ☹

Why?

*Simple reason: it's not just a cycle in memory.*

*Real reason: can't* use *recursive value before finished* defining *it*

- `List.map` *will try to take apart* `nats`*, but* `nats` *isn't finished being defined yet.*
- *Whereas with* `ones`*, nothing ever tried to take* `ones` *apart as part of definition.*

aka infinite lists, sequences, delayed lists, lazy lists

# STREAMS

# Stream representation

```
type 'a mylist =
   | Nil
   | Cons of 'a * 'a mylist
```

# Stream representation

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

# Stream representation

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

# Stream representation

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

Can construct infinite list of ones:

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

But still can't construct the naturals:

```
let rec from n =
  Cons (n, from (n+1))
let nats = from 0 (* stack overflow *)
```

Need to prevent OCaml from evaluating entire infinite list
Instead produce finite parts of it on demand

# Delaying evaluation

```
let f1 = failwith "oops"
let f2 = fun x -> failwith "oops"
```

- defining `f1` immediately raises exception
- defining `f2` does not
- Dynamic semantics:
  - functions are already values
  - don’t evaluate inside body until function is applied

Wrapping an expression with a function will delay its evaluation

# Stream representation

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

let rec from n =
  Cons (n, from (n+1))

let nats = from 0
```

# Stream representation

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

let rec from n =
  Cons (n, fun x -> from (n+1))

let nats = from 0
```

delay evaluation

# Stream representation

type must change

```
type 'a stream =
  Cons of 'a * (? -> 'a stream)

let rec from n =
  Cons (n, fun x -> from (n+1))

let nats = from 0
```

# Stream representation

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

let rec from n =
  Cons (n, fun () -> from (n+1))

let nats = from 0
```

Function that takes unit as argument is called a *thunk*.

# 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)
```

# Accessing finite parts of stream

```
(* [hd s] is the head of [s] *)
let hd (Cons (h, _)) = h

(* [tl s] is the tail of [s] *)
let tl (Cons (_, tf)) = tf ()

(* [take n s] is the list of the first [n] elements of [s] *)
let rec take n s =
  if n=0 then []
  else hd s :: take (n-1) (tl s)

(* [drop n s] is all but the first [n] elements of [s] *)
let rec drop n s =
  if n = 0 then s
  else drop (n-1) (tl s)
```

Applying the thunk to unit *forces* evaluation to resume

# Notation

For documentation examples, write

```
<a; b; c; ...>
```

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

# Arith. operations on streams

```
(* [square <a;b;c;...>] is [<a*a;b*b;c*c;...]. *)
let rec square (Cons (h, tf)) =
  Cons (h*h, fun () -> square (tf ()))

(* [sum <a1;b1;c1;...> <a2;b2;c2;...>] is
 * [<a1+b1;a2+b2;a3+b3;...>] *)
let rec sum (Cons (h1, tf1)) (Cons (h2, tf2)) =
  Cons (h1+h2, fun () -> sum (tf1 ()) (tf2 ()))
```

# Map on streams

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

let square' = map (fun n -> n*n)
let rec nats = Cons(0, fun () -> map (fun x -> x+1) nats)
```

now recursive value definition succeeds

```
(* [map2 f <a1;b1;c1;...> <a2;b2;c2;...>] is
 * [<f a1 b1; f a2 b2; f a3 b3; ...>] *)
let rec map2 f (Cons (h1, tf1)) (Cons (h2, tf2)) =
  Cons (f h1 h2, fun () -> map2 f (tf1 ()) (tf2 ()))

let sum' = map2 (+)
let mult = map2 ( * )
```

# LAZINESS

# 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 1 (fibs + tl fibs)

# Fibonacci

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

But try: `take 100 fibs`

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

We'd like OCaml to remember the results of forcing a thunk, instead of recomputing: aka caching or memoization

# Lazy

OCaml module for

- delaying evaluation
- remembering results once computed

```
module Lazy :
  sig
    type 'a t = 'a lazy_t
    val force : 'a t -> 'a
  end
```

# Lazy

- Syntax: **`lazy`** `e`

- Static semantics:
  if `e:u` then **`lazy`** `e : u Lazy.t`

- Dynamic semantics:
  **`lazy`** `e` does not evaluate e to a value.
  Instead, **`lazy`** `e` evaluates to a *delayed value* that, when forced for the first time, will cause the evaluation of `e` to a value `v`, and if forced again, will simply return `v` without evaluating `e` again

# Lazy fib

```
let fib30long =       (* long time to compute *)
  take 30 fibs |> List.rev |> List.hd

let fib30lazy =       (* short time to compute *)
  lazy
   (take 30 fibs |> List.rev |> List.hd)

let fib30 =           (* long time to compute *)
  Lazy.force fib30lazy

let fib30fast =       (* short time to compute *)
  Lazy.force fib30lazy
```

# Laziness

- OCaml's usual evaluation is eager aka strict:
  - always evaluate argument before function application
  - have to ask for laziness

- Haskell is lazy by default:
  - pleasant when programming with infinite data
  - but harder to reason about space and time
  - and has bad interactions with side-effects

# Upcoming events

- [Friday] A2 due
- [next Tuesday] Prelim I
- [Thursday, 7-9pm] Review Session, Gates G01
- [Sunday, 12-2pm] Review Session, Gates G01