Functional Abstractions over Imperative Infrastructure
and
Lazy Evaluation

COS 326

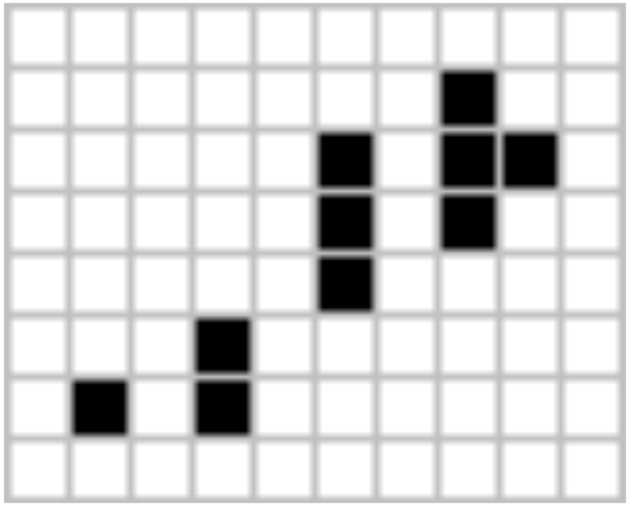
David Walker

Princeton University

– *Abstractions involve using your imagination*

LoStrangolatore

2, 3, 5, 7, 11, 13, 17, 19 …



ConwayLife

Welcome to the Infinite!

```
module type INFINITE =
  sig
    type ‘a stream                        (* an infinite series of values *)

    val const : ‘a -> ‘a stream           (* an infinite series – all the same *)

    val head : ‘a stream -> ‘a            (* get next value – there always is one! *)
    val tail : ‘a stream -> ‘a stream            (* get all the rest *)

    val map : (‘a -> ‘b) -> ‘a stream -> ‘b stream
    val nats : () -> int stream                  (* all of the natural numbers *)
    ...
end

module Inf : INFINITE = ... ?
```

How would you implement this data structure?

```
module type INFINITE =
  sig
    type ‘a stream                    (* an infinite series of values *)

    val const : ‘a -> ‘a stream       (* an infinite series – all the same *)

    val head : ‘a stream -> ‘a        (* get next value – there always is one! *)
    val tail : ‘a stream -> ‘a stream          (* get all the rest *)

    val map : (‘a -> ‘b) -> ‘a stream -> ‘b stream
    val nats : () -> int stream                (* all of the natural numbers *)
    …
end

module Inf : INFINITE = … ?
```

Consider this definition:

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

We can write functions to extract a stream's head and tail:

```
let head(s:'a stream):'a =
  match s with
  | Cons (h,_) -> h

let tail(s:'a stream):'a stream =
  match s with
  | Cons (_,t) -> t
```

But there's a problem…

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

How do I build a value of type 'a stream?

attempt: Cons (3, ______) …. Cons (3, Cons (4, ___))

There doesn't seem to be a base case (e.g., Nil)

Since we need a stream to build a stream,
what can we do to get started?

One idea

```
type ‘a stream =
  Cons of ‘a * (‘a stream)

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

What happens?

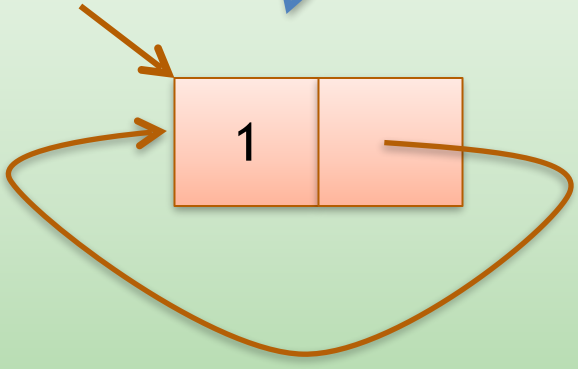
```
# let rec ones = Cons(1,ones);;
val ones : int stream =
  Cons (1,
   Cons (1,
    Cons (1,
     Cons (1, …
))))
#
```

One idea

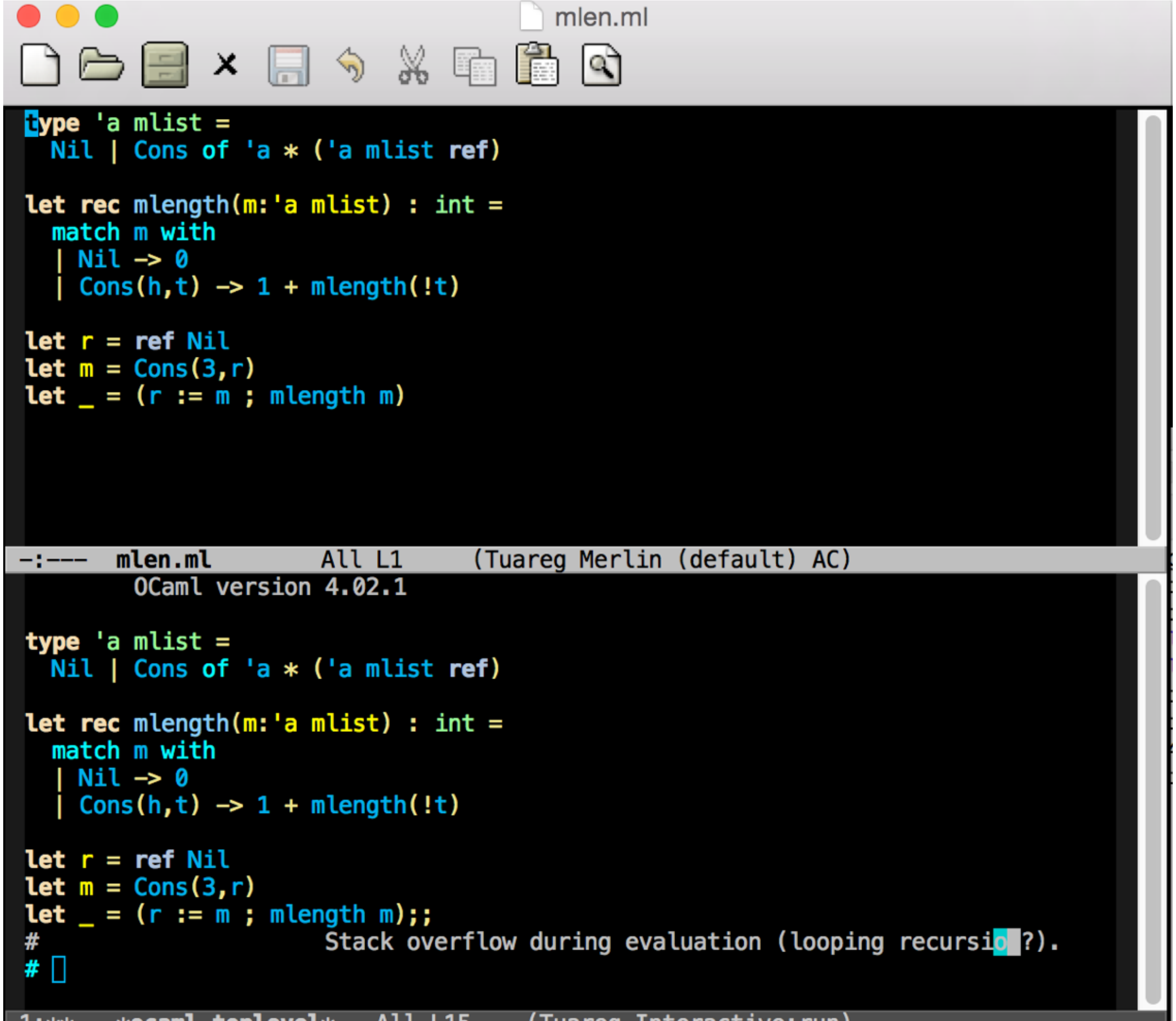
```
type ‘a stream =
  Cons of ‘a * (‘a stream)


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


What happens?
```

```
# let rec ones = Cons(1,ones);;
val ones : int stream =
 Cons (1,
  Cons (1,
   Cons (1,
    Cons (1, …
))))
#
```

OCaml builds this!



Fraught with Peril



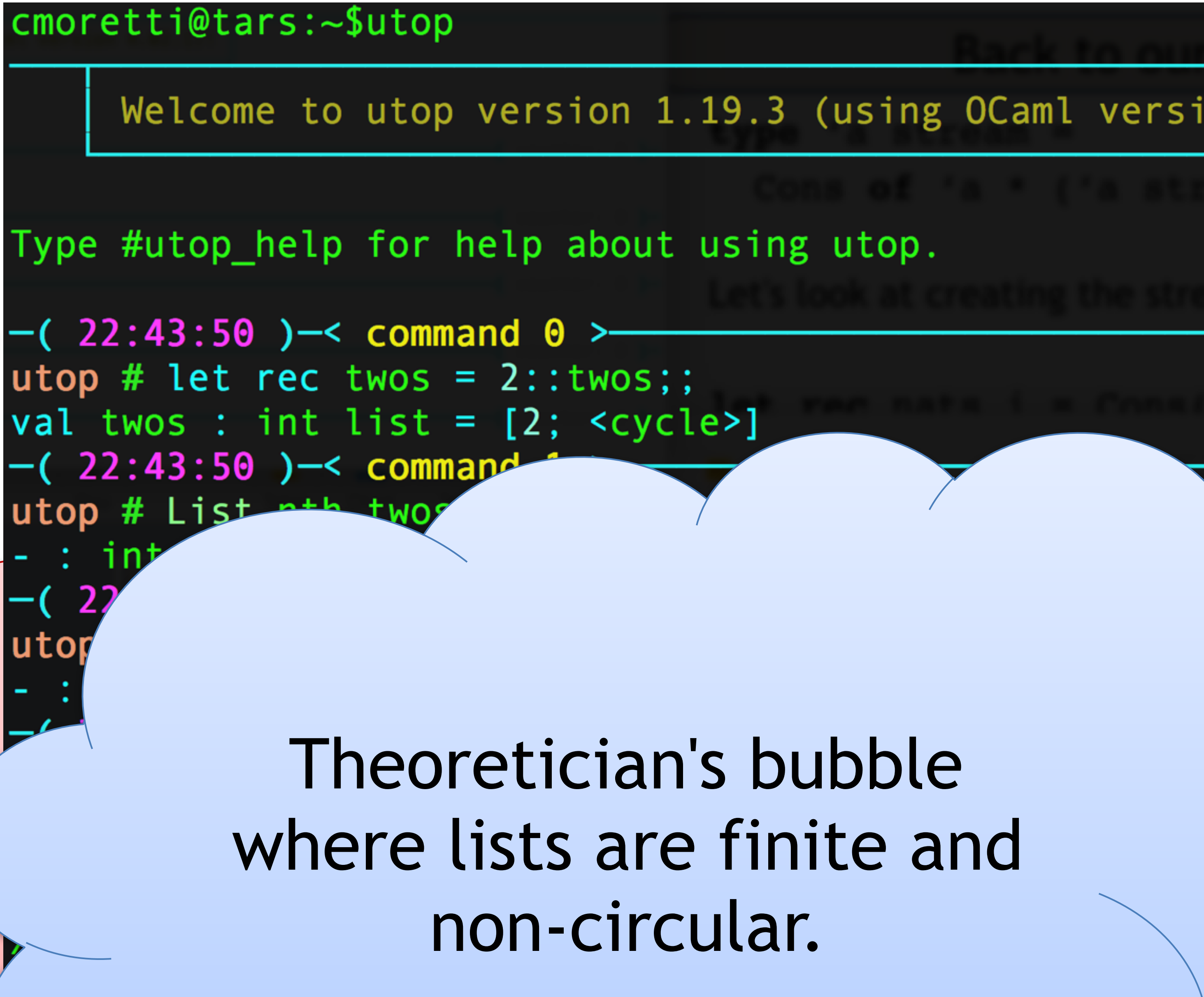
```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + mlength(!t)

let r = ref Nil
let m = Cons(3,r)
let _ = (r := m ; mlength m)
```

```
        OCaml version 4.02.1

type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + mlength(!t)

let r = ref Nil
let m = Cons(3,r)
let _ = (r := m ; mlength m);;
#                     Stack overflow during evaluation (looping recursion?).
# 
```

Oops, I lied … big time

It bugs me that you can do this in OCaml.

WHY????



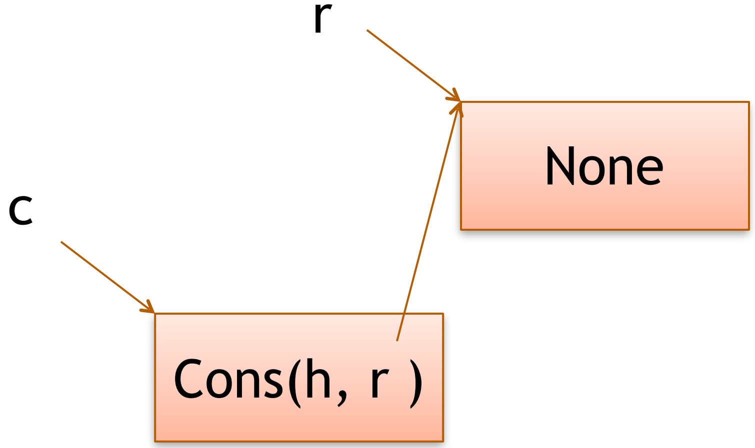
```
cmoretti@tars:~$utop
─────────────────────────────────────────
     │ Welcome to utop version 1.19.3 (using OCaml versi
     └───────────────────────────────────

Type #utop_help for help about using utop.

─( 22:43:50 )─< command 0 >──────────────
utop # let rec twos = 2::twos;;
val twos : int list = [2; <cycle>]
─( 22:43:50 )─< command
utop # List nth twos
- : int
─( 22
utop
- :
```

OCAML –1!
Java -12
C –200

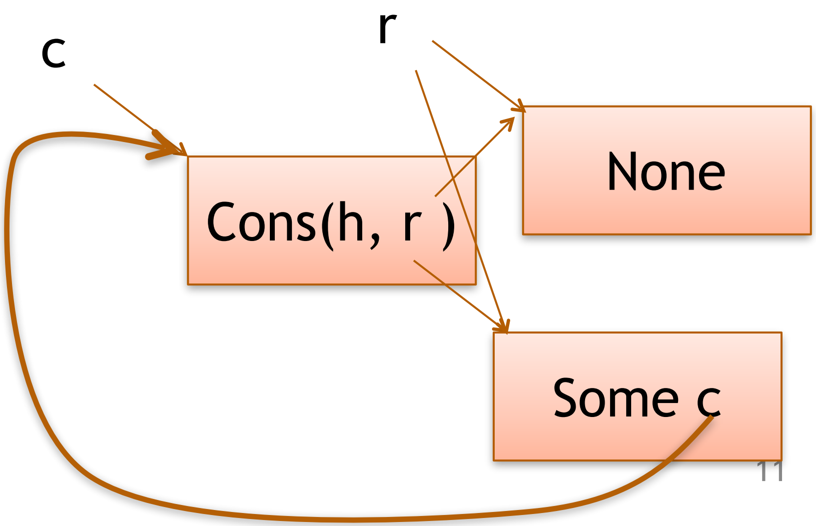
Theoretician's bubble where lists are finite and non-circular.

An alternative would be to use refs

```
type ‘a stream =
   Cons of ‘a * (‘a stream) option ref

let circular_cons h =
   let r = ref None in
   let c = Cons(h,r) in
   (r := (Some c); c)
```



This works …
but has a serious drawback



An alternative would be to use refs

```
type 'a stream =
  Cons of 'a * ('a stream) option ref

let circular_cons h =
  let r = ref None in
  let c = Cons(h,r) in
  (r := (Some c); c)
```

This works …
but has a serious drawback:
  when we try to get out the tail, it may not exist.

Back to our earlier idea

```
type ‘a stream =
  Cons of ‘a * (‘a stream)
```

Let's look at creating the stream of all natural numbers:

```
let rec nats i = Cons(i,nats (i+1))
```

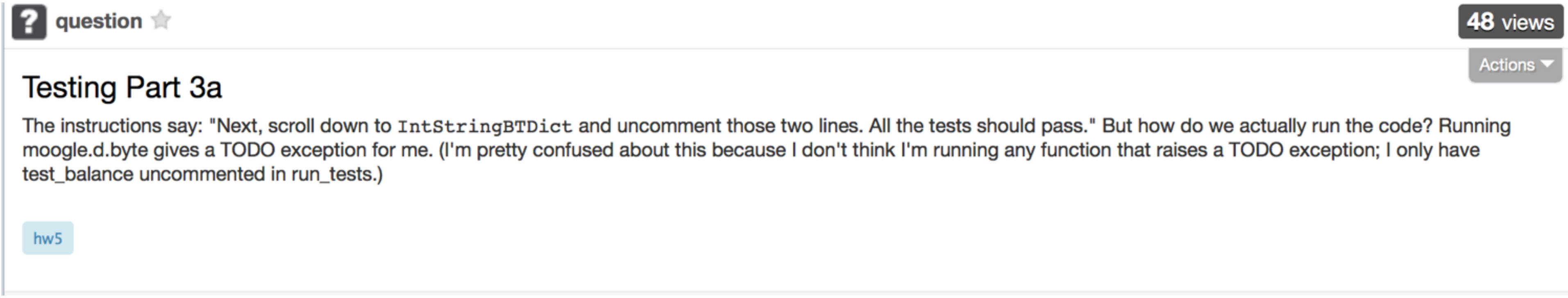
```
# let n = nats 0;;
Stack overflow during evaluation (looping recursion?).
```

OCaml evaluates our code just a little bit too *eagerly*.
We want to evaluate the right-hand side only when necessary...

Be Less Eager

How can we prevent OCaml from evaluating an expression immediately when it is defined?

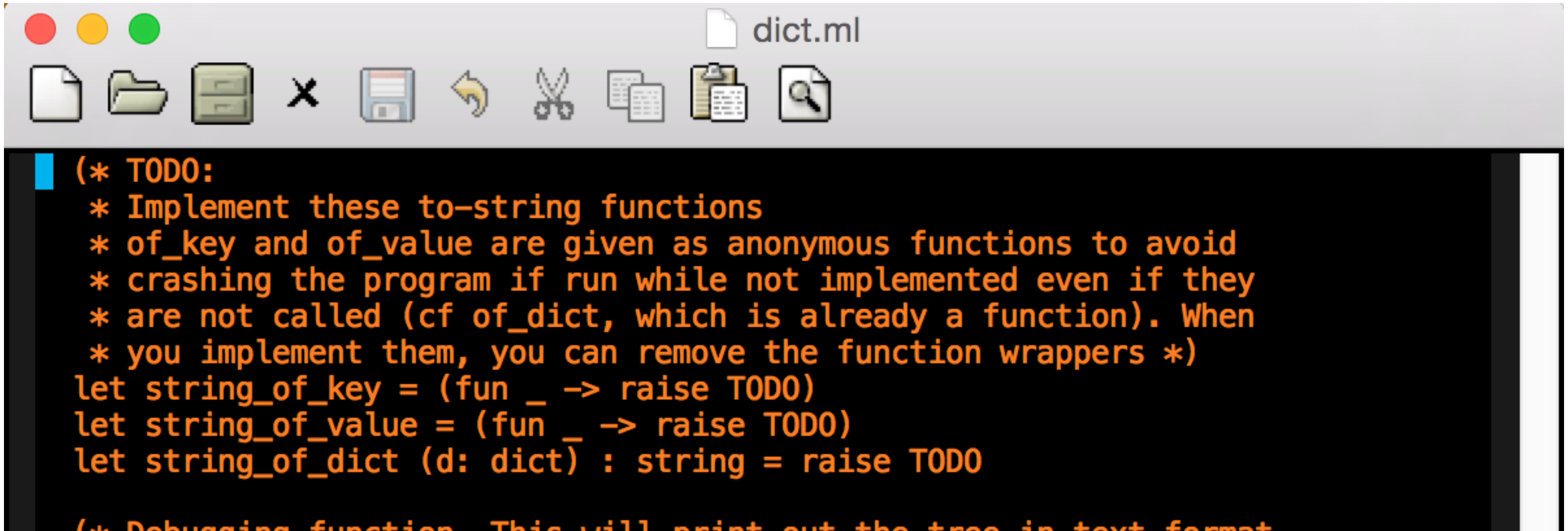
Wait, this sounds familiar …



Testing Part 3a

The instructions say: "Next, scroll down to `IntStringBTDict` and uncomment those two lines. All the tests should pass." But how do we actually run the code? Running moogle.d.byte gives a TODO exception for me. (I'm pretty confused about this because I don't think I'm running any function that raises a TODO exception; I only have test_balance uncommented in run_tests.)

hw5



```
(* TODO:
 * Implement these to-string functions
 * of_key and of_value are given as anonymous functions to avoid
 * crashing the program if run while not implemented even if they
 * are not called (cf of_dict, which is already a function). When
 * you implement them, you can remove the function wrappers *)
let string_of_key = (fun _ -> raise TODO)
let string_of_value = (fun _ -> raise TODO)
let string_of_dict (d: dict) : string = raise TODO

(* Debugging function. This will print out the tree in text format
```

Another idea

One way to implement “waiting” is to wrap a computation up in a function and then call that function later when we want to.

Another attempt:

```
type ‘a stream = Cons of ‘a * (‘a stream)

let rec ones =
  fun () -> Cons(1,ones)

let head (x) =
  match x () with
    Cons (hd, tail) -> hd
;;

head (ones);;
```

Are there any problems with this code?

Darn. Doesn’t type check!
ones is a function with type
unit -> int stream
not just int stream

Functional Implementation

What if we changed the stream definition one more time?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str
```

What we had before.

Augmented as a *mutually recursive* type definition

```
let rec ones : int stream =
  fun () -> Cons(1,ones)
```

Or, the way we’d normally write it:

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

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let head(s:’a stream):’a =
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let head(s:’a stream):’a =
 match s() with
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let head(s:’a stream):’a =
 match s() with
  | Cons(h,t) ->
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let head(s:’a stream):’a =
 match s() with
  | Cons(h,_) -> h
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let head(s:’a stream):’a =
 match s() with
 | Cons(h,_) -> h

let tail(s:’a stream):’a stream =
 match s() with
 | Cons(_,t) -> t
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let rec map (f:’a->’b) (s:’a stream) : ’b stream =
  Cons(f (head s), map f (tail s))
```

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let rec map (f:’a->’b) (s:’a stream) : ’b stream =
  Cons(f (head s), map f (tail s))
```

Rats!

Infinite looping!

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let rec map (f:’a->’b) (s:’a stream) : ’b stream =
  Cons(f (head s), map f (tail s))
```

But we don’t infinite loop, because the typechecker saves us: Cons (x,y) is a str not a stream

Functional Implementation

How would we define head, tail, and map of an 'a stream?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let rec map (f:’a->’b) (s:’a stream) : ’b stream =
  fun () -> Cons(f (head s), map f (tail s))
```

Importantly, map must return a function, which delays evaluating the recursive call to map.

Functional Implementation

Now we can use map to build other infinite streams:

```
let rec map(f:'a->'b)(s:'a stream):'b stream =
  fun () -> Cons(f (head s), map f (tail s))

let rec ones = fun () -> Cons(1,ones) ;;
let inc x = x + 1
let twos = map inc ones ;;
```

head twos
--> head (map inc ones)
--> head (fun () -> Cons (inc (head ones), map inc (tail ones)))
--> match (fun () -> …) () with Cons (hd, _) -> h
--> match Cons (inc (head ones), map inc (tail ones)) with Cons (hd, _) -> h
--> match Cons (inc (head ones), fun () -> …) with Cons (hd, _) -> h
--> … --> 2

Another combinator for streams:

```
let rec zip f s1 s2 =
  fun () ->
   Cons(f (head s1) (head s2),
        zip f (tail s1) (tail s2)) ;;

let threes = zip (+) ones twos ;;


let rec fibs =
  fun () ->
   Cons(0, fun () ->
            Cons (1,
                  zip (+) fibs (tail fibs)))
```

Unfortunately

This is not very efficient:

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str
```

Every time we want to look at a stream (e.g., to get the head or tail), we have to re-run the function.

So when you ask for the 10th fib and then the 11th fib, we are re-calculating the fibs starting from 0, when we could *cache* or *memoize* the result of previous fibs.

LAZY EVALUATION

Memoizing Streams

We can take advantage of refs to memoize:

```
type ‘a thunk =
  Unevaluated of (unit -> ‘a) | Evaluated of ‘a

type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = (‘a str) thunk ref
```

When we build a stream, we use an Unevaluated thunk to be lazy. But when we ask for the head or tail, we remember what Cons-cell we get out and save it to be re-used in the future.

Memoizing Streams

```
type ‘a thunk =
  Unevaluated of (unit -> ‘a) | Evaluated of ‘a
type ‘a lazy_t = (‘a thunk) ref ;;

type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = (‘a str) lazy_t;;

let rec head(s:’a stream):’a =
  match !s with
  | Evaluated (Cons(h,_)) -> h
  | Unevaluated f ->
     let x = f() in (s := Evaluated x; head s)
```

Memoizing Streams

```
type ‘a thunk =
  Unevaluated of (unit -> ‘a) | Evaluated of ‘a
type ‘a lazy_t = (‘a thunk) ref ;;

type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = (‘a str) lazy_t;;

let rec tail(s:’a stream) : ’a stream =
  match !s with
  | Evaluated (Cons(_,t)) -> t
  | Unevaluated f ->
    (s := Evaluated (f()); tail s) ;;
```

Memoizing Streams

```
type 'a thunk =
  Unevaluated of (unit -> 'a) | E   luated of 'a
type 'a lazy_t = ('a t   nk) re

type 'a str            c
and 'a stre

let re
  ma

  | Un
                       ail s)  ;
```

Common pattern!

Dereference & check if evaluated:

- If so, take the value.
- If not, evaluate it & take the value

Memoizing Streams

```
type ‘a thunk =
  Unevaluated of (unit -> ‘a) | Evaluated of ‘a
type ‘a lazy_t = (‘a thunk) ref

type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = (‘a str) lazy_t

let rec force(t:’a lazy_t):’a =
  match !t with
  | Evaluated v -> v
  | Unevaluated f ->
      let v = f() in
      (t:= Evaluated v ; v)

let head(s:’a stream) : ’a =
  match force s with
  | Cons(h,_) -> h

let tail(s:’a stream) : ’a stream =
  match force s with
  | Cons(_,t) -> t
```

Memoizing Streams

```
type ‘a thunk =
  Unevaluated of (unit -> ‘a) | Evaluated of ‘a

type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = (‘a str) thunk ref;;

let rec ones =
  ref (Unevaluated (fun () -> Cons(1,ones))) ;;
```

Memoizing Streams

```
type ‘a thunk =
  Unevaluated of (unit -> ‘a) | Evaluated of ‘a

type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = (‘a str) thunk ref;;

let thunk f = ref (Unevaluated f)

let rec ones =
  thunk (fun () -> Cons(1,ones))
```

What’s the interface?

```
type ‘a lazy

val  thunk : (unit -> ‘a) -> ‘a lazy

val  force:  ‘a lazy -> ‘a
```

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = (‘a str) lazy

let rec ones =
  thunk(fun () -> Cons(1,ones))
```

OCaml’s Builtin Lazy Constructor

If you use Ocaml’s built-in lazy_t, then you can write:

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

and this takes care of wrapping a “ref (Unevaluated (fun () -> ...))” around the whole thing.

So for example:

```
let rec fibs =
  lazy (Cons(0,
        lazy (Cons(1,zip (+) fibs (tail fibs)))))
```

The whole example at once

```
type 'a str = Cons of 'a * 'a stream
and 'a stream = ('a str) Lazy.t;;

let rec zip f (s1: 'a stream) (s2: 'a stream) : 'a stream =
 lazy (match Lazy.force s1, Lazy.force s2 with
        Cons (x1,r1), Cons (x2,r2) ->
                 Cons (f x1 x2, zip f r1 r2));;

let tail (s: 'a stream) : 'a stream =
 match Lazy.force s with Cons (x,r) -> r;;

let rec fibs : int stream =
  lazy (Cons(0, lazy (Cons (1, zip (+) fibs (tail fibs)))));;

let rec g n s =
 if n>0 then
  match Lazy.force s with Cons (x,r) ->
(print_int x; print_string "\n"; g (n-1) r)
 else ();;

g 10 fibs;;
```

More Examples: Pi

```
(* pi is approximated by the Taylor series:
 *   4/1 - 4/3 + 4/5 - 4/7 + ...
 *)
let rec alt_fours =
  lazy (Cons (4.0,
  lazy (Cons (-4.0, alt_fours))));;

let pi_series = zip (/.) alt_fours (map
  float_of_int odds);;

let pi_up_to n =
  List.fold_left (+.) 0.0
      (first n pi_series) ;;
```

A note on laziness

- By default, OCaml is an eager language, but you can use the “lazy” features to build lazy datatypes.

- Other functional languages, notably Haskell, are lazy by default. *Everything* is delayed until you ask for it.
  - generally much more pleasant to do programming with infinite data.
  - but harder to reason about space and time.
  - and has bad interactions with side-effects.

- The basic idea of laziness gets used a lot:
  - e.g., Unix pipes, TCP sockets, etc.

Summary

You can build *infinite data structures*.

- Not really infinite – represented using cyclic data and/or lazy evaluation.

Lazy evaluation is a useful technique for delaying computation until it's needed.

- Can model using just functions.
- But behind the scenes, we are *memoizing* (caching) results using refs.

This allows us to separate model generation from evaluation to get "scale-free" programming.

- e.g., we can write down the routine for calculating pi regardless of the number of bits of precision we want.
- Other examples: geometric models for graphics (procedural rendering); search spaces for AI and game theory (e.g., tree of moves and counter-moves).