Announcements:

- PS #6 due Friday December 2, 11:59PM
    - Submissions open until grading starts, no earlier than Monday night.
- Final quiz on Thursday?
- FINAL EXAM on Monday December 12
- Tournament and review session the weekend before the final exam, probably both will be Sunday.

Streams and lazy evaluation

- Introduction : the if construct

- We have already seen that in OCaml, `if true then e1 else e2` evaluates to `e1`, while `if false then e1 else e2` evaluates to `e2`.

- Actually, in `if true then e1 else e2`, the value of `e2` is never evaluated: this is called a lazy evaluation.

- We say that if *eagerly* evaluates condition expression to true or false, and *lazily* evaluates `e1` and `e2`.

- In OCaml, function arguments are eagerly evaluated: the function arguments are evaluated before the function is called.
- This is not true for every language; for instance in Haskell function arguments are lazily evaluated.
- Also in OCaml, function bodies are lazily evaluated: `fun x->e` is considered a value,
    - and no attempt is made to evaluate the value of `e`: function bodies are not evaluated until the function is applied.

- The example below shows that trying to reprogram an if construct using an eager evaluation always fails: the if construct needs a lazy evaluation!

```
let rec factorial (n : int) : int =
  if n <= 0 then 1 else n * factorial (n - 1)

let my_if ((b, t, f):bool * 'a * 'a) : 'a =
  if b then t else f

(* does not terminate :
   get to factorial2(0), then tries to
   evaluate factorial2(-1), factorial2(-2), etc. *)

let rec factorial2 (n : int) : int =
  my_if (n <= 0, 1, n * factorial2 (n - 1))

let if_funs ((b, t, f):bool * (unit->'a) * (unit->'a)): 'a=
  if b then t() else f()

(* factorial2 fixed *)
let rec factorial3 (n : int) : int =
  if_funs (n <= 0, (fun () -> 1), (fun () -> n * factorial3 (n - 1)))

(* factorial2 NOT fixed: need for macros! *)
let rec factorial4 (n : int) : int =
  if_funs (n <= 0, lazy(1)), lazy(n * factorial4 (n - 1))
```

Call by value and call by name

- Let us now look at let constructs and function evaluations.
- In an eager language like OCaml, these are evaluated using a call by value semantics:
  - `let x=v in e2 --> e2{v/x}` and `(fun x->e2) v --> e2{v/x}`;
  - the value bound to `x` is evaluated eagerly before the body `e2`.

- In a lazy language like Haskell however, they are evaluated using a call by name semantics:
  - `let x=e1 in e2 --> e2{e1/x}` and `(fun x->e2) e1 --> e2{e1/x}`:
  - `e1` is not evaluated until `x` is used, and a variable can stand for an unevaluated expression.

- However a question arises: what if `x` occurs 10 times in `e2`? Should we evaluate it 10 times?
- In Haskell this is solved by a thunk-like mechanism, where `x` is evaluated only the first time it is used, and then its value is remembered.

Thunks

- We already know that `let f=e` evaluated `e` right away; on the other hand, `let f=fun () -> e` evaluates `e` every time, but not until `f` is called. Here we introduce thunks, for which if we write `let f = Thunk.make (fun ()->e)`, `e` is evaluated once, but not until we use it by calling `Thunk.apply f`.
- The implementation here has to use a `ref`, so that it is possible to do something different the second time `Thunk.apply` is called.

```
module type THUNK =
  sig
    (* A 'a thunk is a lazily
     * evaluated expression e of type
     * 'a. *)
    type 'a thunk
    (* make(fn()=>e) creates a thunk
     * for e *)
    val make : ( unit -> 'a ) -> 'a thunk
    (* apply(t) is the value of t's expression,
     * which is only evaluated once *)
    val apply : 'a thunk -> 'a
  end

module Thunk : THUNK =
  struct
    type 'a thunkPart =
      Done of 'a
    | Undone of (unit -> 'a)

    type 'a thunk =  ('a thunkPart) ref

    let make (f : unit -> 'a) : 'a thunk =
      ref (Undone f)

    let apply (th : 'a thunk) : 'a =
      match !th with
       Done x -> x
     | Undone f ->
         let ans = f()
         in (th := Done ans;
             ans)
end
```

- Thunk example:

```
(* A silly way to take a long time to do something. *)
let rec slow_add1 (x: int) =
  let rec slow_id ((x, y):(int * int)) =
    if(y = 0) then x else slow_id(x, y-1)
  in
    (slow_id (x,100000000)) + 1

(* Returns immediately *)
let seven_thunk = Thunk.make (fun () -> slow_add1 6)
```

Streams

Streams

A stream is a possibly infinite list; for example,

- the stream of all natural numbers `[0; 1; 2; 3; 4; ...]`
- the stream of all Fibonacci numbers `[1; 1; 2; 3; 5; 8; 13; ...]`
- the stream of all primes `[2; 3; 5; 7; 11; 13; ...]`.

It is actually possible to create some infinite OCaml lists, but only *regular* (or *ultimately periodic*) ones, and they are not too useful. We mostly use only finite OCaml lists. Recall that if lists had not been built into OCaml, we might have defined them as

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

This is actually how it is done. Finite lists are built inductively from right to left, starting with `Nil`and `Cons`'ing a new head onto an already evaluated tail.

However, we can get infinite streams by *deferring* the creation of the tail using thunks. Thus we create the tail only when we need it.

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

Now we can define some infinite streams.

```
(* an infinite stream of 1's *)

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



(* the natural numbers *)

let rec from (n : int) : int stream =

  Cons (n, fun () -> from (n + 1))

let naturals = from 0
```

What have we just created? The head of stream `ones` is 1 and its tail is itself, namely `ones`. Thus, an infinite stream of 1's. But where are all those 1's? The computer is finite. The answer is that they are not created yet. They will only be created when we need them.

Let's define some operations on streams.

```
(* head of a stream *)

let hd (s : 'a stream) : 'a =

  match s with

    Nil -> failwith "hd"

  | Cons (x, _) -> x

(* tail of a stream *)

let tl (s : 'a stream) : 'a stream =

  match s with

    Nil -> failwith "tl"

  | Cons (_, g) -> g () (* get the tail by evaluating the thunk *)

(* n-th element of a stream *)

let rec nth (s : 'a stream) (n : int) : 'a =

  if n = 0 then hd s else nth (tl s) (n - 1)

(* make a stream from a list *)

let from_list (l : 'a list) : 'a stream =

  List.fold_right (fun x s -> Cons (x, fun () -> s)) l Nil

(* make a list from the first n elements of a stream *)

let rec take (s : 'a stream) (n : int) : 'a list =

  if n <= 0 then [] else

  match s with

    Nil -> []

  | _ -> hd s :: take (tl s) (n - 1)
```

Let's try these out.

```
# hd ones;;

- : int = 1

# hd (tl ones);;

- : int = 1

# nth ones 10;;

- : int = 1

# nth ones 10000000;;

- : int = 1

# take ones 20;;

- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1]

# let five = from_list [1; 2; 3; 4; 5];;

val five : int stream = Cons (1, <fun>)

# take five 2;;

- : int list = [1; 2]

# take five 10;;

- : int list = [1; 2; 3; 4; 5]

# take naturals 10;;

- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Now we can operate on streams as if they existed in their entirety. For example, we can define the usual list operations `map` and `filter`:

```
let rec map (f : 'a -> 'b) (s : 'a stream) : 'b stream =
  match s with Nil -> Nil
  | _ -> Cons (f (hd s), fun () -> map f (tl s))


let rec filter (f : 'a -> bool) (s : 'a stream) : 'a stream =
  match s with Nil -> Nil
  | Cons (x, g) ->
      if f x then Cons (x, fun () -> filter f (g ()))
      else filter f (g ())


let rec map2 (f: 'a -> 'b -> 'c)
             (s : 'a stream) (t : 'b stream) : 'c stream =
  match (s, t) with
    (Nil, Nil) -> Nil
  | (Cons (x, g), Cons (y, h)) ->
       Cons (f x y, fun () -> map2 f (g ()) (h ()))
  | _ -> failwith "map2"
```

Let's try these out.

```
# let square n = n * n;;

val square : int -> int = <fun>

# take (map square naturals) 20;;

- : int list =

[0; 1; 4; 9; 16; 25; 36; 49; 64; 81; 100; 121; 144; 169; 196; 225; 256;
289;

 324; 361]

# let even = fun n -> n mod 2 = 0;;

val even : int -> bool = <fun>

# take (filter even naturals) 20;;

- : int list =

[0; 2; 4; 6; 8; 10; 12; 14; 16; 18; 20; 22; 24; 26; 28; 30; 32; 34; 36; 38]
```

Now for something a little fancier:

```
(* the Fibonacci sequence *)

let fib1 : int stream =

  let rec fibgen (a : int) (b : int) : int stream =

    Cons(a, fun () -> fibgen b (a + b))

  in fibgen 1 1


# take fib1 20;;

- : int list =

[1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233; 377; 610; 987; 1597; 2584;

 4181; 6765]

# nth fib1 43;;

- : int = 701408733


(* another version - this one is a lot slower *)

let rec fib2 : int stream =

  let add = map2 (+) in

  Cons (1, fun () -> Cons (1, fun () -> add fib2 (tl fib2)))


# take fib2 20;;

- : int list =

[1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233; 377; 610; 987; 1597; 2584;

 4181; 6765]
```

```
(* delete multiples of p from a stream *)

let sift (p : int) : int stream -> int stream =

  filter (fun n -> n mod p <> 0)



(* sieve of Eratosthenes *)

let rec sieve (s : int stream) : int stream =

  match s with Nil -> Nil

  | Cons (p, g) -> Cons (p, fun () -> sieve (sift p (g ())))



(* primes *)

let primes = sieve (from 2)



# take primes 20;;

- : int list =

[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67;
71]

# nth primes 1000;;

- : int = 7927
```

Streams are actually useful in real life. Some applications:

- compilers reading source file from text
- network sockets
- audio and video signal processing
- voice recognition
- approximating solutions to equations using convergent series

One last example: merging and splitting streams.

```
(* merge two streams into one, taking elements alternately *)
```

```
let rec merge (s : 'a stream) (t : 'a stream) : 'a stream =
  match s with Nil -> t
  | Cons (x, g) -> Cons (x, fun () -> merge t (g ()))


(* split a stream into two streams - inverse of merge *)
let rec split (s : 'a stream) : 'a stream * 'a stream =
  match take s 2 with
    [] -> (Nil, Nil)
  | [x] -> (Cons (x, fun () -> Nil), Nil)
  | x :: y :: _ ->
     let t = tl (tl s) in
     (Cons (x, fun () -> fst (split t)), Cons (y, fun () -> snd (split t)))
```