```
(*
                              CS51 Lab 15
                Lazy Programming and Infinite Data Structures
                      Using OCaml's native lazy module
 *)

(* In the last lab you became familiar with an *explicit*
implementation of laziness, using functions to delay computation. In
this lab you will delve further into lazy evaluation and look at how
laziness is implemented *natively* in OCaml with the `Lazy` module,
which we use to implement a `NativeLazyStreams` module. *)

open CS51Utils ;;       (* for access to timing functions *)

  (* Digression: We've just opened `CS51Utils` above, so we can make
     use of its components (like `Absbook.call_reporting_time`) in
     this file. But if you want to use it in the REPL, you'll need to
     make it accessible there as well, e.g.,

        # #require "CS51Utils" ;;
        # open CS51Utils ;;                                          *)

(*======================================================================
Part 1: Using OCaml's Lazy module

All of the recomputation going on behind the scenes with the
stream-based solutions in the previous lab is prohibitive. Chapter 17
of the textbook describes the use of *memoizing* to eliminate the
recomputation, and showed an implementation of memoizing *thunks* in
terms of refs. As described in Section 17.3, that functionality is
actually already available in OCaml through its `Lazy` module. The
`Lazy` module introduces a new polymorphic type `'a Lazy.t` of delayed
values of type 'a, and a new function `Lazy.force : 'a Lazy.t -> 'a`
that forces a delayed computation to occur, saving the result if this
is the first time the value was forced and simply returning the saved
value on later requests. For instance, suppose we've defined the
Fibonacci function "eagerly" as: *)

let rec fib (n : int) : int =
  if n < 2 then n
  else (fib (n - 1)) + (fib (n - 2)) ;;

(* Then a delayed computation of the 42nd Fibonacci number would be *)

let fib42 : int Lazy.t =
  lazy (fib 42) ;;

(* Here, we force the computation twice in a row, timing the two calls:

    # Absbook.call_reporting_time Lazy.force fib42 ;;
    Elapsed time: 13.380860
    - : int = 267914296
    # Absbook.call_reporting_time Lazy.force fib42 ;;
    Elapsed time: 0.000000
    - : int = 267914296

The first time through takes 13 seconds, the second less than a
microsecond. Try it on your own computer. *)

(*......................................................................
Exercise 1. The `NativeLazyStreams` module, found in the file
`nativeLazyStreams.ml`, is an incomplete reimplementation of the
`LazyStreams` module from the previous lab, but using OCaml's native
`Lazy` module. Complete this implementation by implementing `smap`,
`smap2`, and `sfilter` in that file. You may want to refer to the last
```

```
lab, especially the `LazyStreams` module provided there and the lab
solution's discussion of its exercise on `sfilter`.
.............................................................*)

(* Now we can redo the Fibonacci example from the textbook. First, we
open the `NativeLazyStreams` module so we can use its components more
easily. *)

open NativeLazyStreams

  (* Digression redux: We've just opened `NativeLazyStreams` above, so
     we can make use of its components in this file. But if you want
     to use it in the REPL, you'll need to make it accessible there as
     well, e.g.,

        # #mod_use "nativeLazyStreams.ml" ;;
        # open NativeLazyStreams ;;

    or

        # #use "nativeLazyStreams.ml" ;;                              *)

(* We implement the Fibonacci sequence as a
`NativeLazystreams.stream`. *)

let rec fibs : int stream =
  lazy (Cons (0, lazy (Cons (1, smap2 (+) fibs (tail fibs))))) ;;

(* We run it twice, generating the first 50 Fibonacci numbers:

    # Absbook.call_reporting_time (first 50) fibs ;;
    time (msecs): 0.029087
    - : int list =
    [0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233; 377; 610; 987; 1597;
     2584; 4181; 6765; 10946; 17711; 28657; 46368; 75025; 121393; 196418; 317811;
     514229; 832040; 1346269; 2178309; 3524578; 5702887; 9227465; 14930352;
     24157817; 39088169; 63245986; 102334155; 165580141; 267914296; 433494437;
     701408733; 1134903170; 1836311903; 2971215073; 4807526976; 7778742049]

    # Absbook.call_reporting_time (first 50) fibs ;;
    time (msecs): 0.006914
    - : int list =
    [0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233; 377; 610; 987; 1597;
     2584; 4181; 6765; 10946; 17711; 28657; 46368; 75025; 121393; 196418; 317811;
     514229; 832040; 1346269; 2178309; 3524578; 5702887; 9227465; 14930352;
     24157817; 39088169; 63245986; 102334155; 165580141; 267914296; 433494437;
     701408733; 1134903170; 1836311903; 2971215073; 4807526976; 7778742049]

This version is much faster than the one implemented using the
`LazyStreams` module from the last lab, even the first time
around. Why? *)

(*.............................................................
Exercise 2. As practice in using `NativeLazyStreams`, implement a
function `geo : float -> float -> float stream` that returns a
geometric sequence as an infinite stream, where the first argument is
the initial value in the stream, and each successive value is
multiplied by the second argument. For example:

    # first 10 (geo 1. 2.) ;;
    - : float list = [1.; 2.; 4.; 8.; 16.; 32.; 64.; 128.; 256.; 512.]

    # first 10 (geo 0.5 0.5) ;;
    - : float list =
    [0.5; 0.25; 0.125; 0.0625; 0.03125; 0.015625; 0.0078125;
```

```
     0.00390625; 0.001953125; 0.0009765625]

For more information on geometric sequences, see
<https://en.wikipedia.org/wiki/Geometric_progression>.
.....................................................................*)

let geo _ = failwith "geo not implemented" ;;

(*======================================================================
Part 2. Eratosthenes' sieve revisited

We return to the Eratosthenes' sieve example from the last lab, which
used the `LazyStreams` module. For reference, here are the
implementations of `nats`, `sieve`, and `primes` from the lab
solution.

    let rec nats =
      fun () -> Cons (0, smap ((+) 1) nats) ;;

    let not_div_by (n : int) (m : int) : bool =
      m mod n <> 0 ;;

    let rec sieve (s : int stream) : int stream =
      let Cons (h, t) = s () in
      fun () -> Cons (h, sieve (sfilter (not_div_by h) t)) ;;

    let primes : int stream = sieve (tail (tail nats)) ;;
 *)

(*.....................................................................
Exercise 3. Reimplement Eratosthenes' sieve now using the
`NativeLazyStreams` module by completing the values and functions
below.
.....................................................................*)

let rec nats = lazy (failwith "nats native not implemented") ;;

let rec sieve s = failwith "sieve native not implemented" ;;

let primes = lazy (failwith "primes native not implemented") ;;

(*.....................................................................
Exercise 4. How much further can you get in computing primes now that
the recomputation problem is solved?  Implement a function to find the
nth element in a stream (indexed starting from 0), and use it to find
out the 2000th prime.
.....................................................................*)

let rec nth (s : 'a stream) (n : int) : 'a =
  failwith "nth native not implemented" ;;


(*======================================================================
Part 3: Series acceleration with infinite streams

In the `Pi` module (see the file `pi.ml`), we provide the definitions
of lazy streams using OCaml's native `Lazy` module, up to and
including code for approximating pi through partial sums of the terms
in a Taylor series. In the next problem, you'll use streams to find
approximations for pi much faster. (For your reference, the first few
digits of pi are:

    3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510  )

Recall from the reading the use of streams to generate approximations
```

of pi of whatever accuracy. Try it. You should be able to reproduce the following in the REPL:

```
# #mod_use "nativeLazyStreams.ml" ;;
...
# #use "pi.ml" ;;
...
# within 0.1 pi_sums ;;
- : int * float = (19, 3.09162380666784)
# within 0.01 pi_sums ;;
- : int * float = (199, 3.13659268483881615)
# within 0.001 pi_sums ;;
- : int * float = (1999, 3.14109265362104129)
# within 0.0001 pi_sums ;;
- : int * float = (19999, 3.14154265358982476)
```

We've recorded the number of steps required in a table below. Notice that it takes about 2000 terms in the Taylor series to get within .001 of the value of pi. This method converges quite slowly. But we can increase the speed dramatically by **averaging adjacent elements in the approximation stream**. *)

```
(*.....................................................................
Exercise 5: Implementing average on streams

Write a function `average` that takes a `float stream` and returns
another `float stream` each of which is the average of adjacent values
in the input stream. For example:

# first 5 (average (to_float nats)) ;;
- : float list = [0.5; 1.5; 2.5; 3.5; 4.5]
.....................................................................*)

let average (s : float stream) : float stream =
  failwith "average not implemented" ;;

(* Now instead of using the stream of approximations in `pi_sums`, you
can instead use the stream of averaged `pi_sums`, which converges much
more quickly. Test that it requires far fewer steps to get within,
say, 0.001 of pi. Record your results below. *)

(*.....................................................................
Exercise 6: Testing the acceleration

Fill out the following table, recording how many steps are needed to
get within different epsilons of pi using the averaged stream.

    ------------------------------------------
    epsilon   |  pi_sums   |  averaged method  |
    ------------------------------------------
    0.1       |         19 |                   |
    ------------------------------------------
    0.01      |        199 |                   |
    ------------------------------------------
    0.001     |       1999 |                   |
    ------------------------------------------
    0.0001    |      19999 |                   |
    ------------------------------------------
.....................................................................*)
```