```
(*
                              CS51 Lab 14
              Lazy Programming and Infinite Data Structures
                    Implementing laziness as user code
 *)

(* This lab provides practice with delayed (lazy) computations,
implemented as user code using the naturally lazy behavior of OCaml
functions. (In the next lab we explore OCaml's built-in `Lazy`
module.)

In this lab, you will use an infinite data structure, the *stream*. *)

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

(*=====================================================================
Part 1: Programming with lazy streams

Recall the lazy `stream` type and associated functions from the
reading, here packaged up into a module. *)

module LazyStream =
  struct

    type 'a stream_internal = Cons of 'a * 'a stream
     and 'a stream = unit -> 'a stream_internal ;;

    (* head strm -- Returns the first element of `strm`. *)
    let head (s : 'a stream) : 'a =
      let Cons (h, _t) = s () in h ;;

    (* tail strm -- Returns a stream containing the remaining elements
       of `strm`. *)
    let tail (s : 'a stream) : 'a stream =
      let Cons (_h, t) = s () in t ;;

    (* first n strm -- Returns a list containing the first `n`
       elements of the `strm`. *)
    let rec first (n : int) (s : 'a stream) : 'a list =
      if n = 0 then []
      else head s :: first (n - 1) (tail s) ;;

    (* smap fn strm -- Returns a stream that applies the `fn` to each
       element of `strm`. *)
    let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
      fun () -> Cons (f (head s), smap f (tail s)) ;;

    (* smap2 fn strm1 strm2 -- Returns a stream that applies the `fn`
       to corresponding elements of `strm1` and `strm2`. *)
    let rec smap2 f s1 s2 =
      fun () -> Cons (f (head s1) (head s2),
                      smap2 f (tail s1) (tail s2)) ;;
  end ;;

(* We open the module for ease of access throughout this lab. *)

open LazyStream ;;

(* Here, recalled from the reading, is the definition of an infinite
stream of ones. *)

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

(* Now you'll define some useful streams. Some of these were defined
```

```
in the reading, but see if you can come up with the definitions
without looking them up. *)

(*.....................................................................
Exercise 1: An infinite stream of the integer 2. As usual, for this
and all succeeding exercises, you shouldn't feel beholden to how the
definition is introduced in the skeleton code below. (We'll stop
mentioning this now, and forevermore.)
.....................................................................*)

let twos = fun () -> failwith "twos not implemented" ;;

(*.....................................................................
Exercise 2: An infinite stream of threes, built by summing the ones
and twos.
.....................................................................*)

let threes = fun () -> failwith "threes not implemented" ;;

(*.....................................................................
Exercise 3: An infinite stream of natural numbers (0, 1, 2, 3, ...).
Try working this out on your own before checking out the solution in
the textbook.
.....................................................................*)

let nats = fun () -> failwith "nats not implemented" ;;

(*.....................................................................
Exercise 4: Create a function `alternate`, which takes two streams and
'alternates' them together; `alternate` should output a single stream
created by alternating the elements of the two input streams starting
with an element of the first stream.

For example, 'alternating' infinite streams of ones (1,1,1,1....) and
twos (2,2,2,2....) would look like this:

     # first 10 (alternate ones twos) ;;
     - : int list = [1; 2; 1; 2; 1; 2; 1; 2; 1; 2]

and alternating the natural numbers (0,1,2,3,4,...) and ones would
look like this:

     # first 10 (alternate nats ones) ;;
     - : int list = [0; 1; 1; 1; 2; 1; 3; 1; 4; 1]
.....................................................................*)

let alternate : 'a stream -> 'a stream -> 'a stream =
  fun _ -> failwith "alternate not implemented";;

(* Now some new examples. For these, you should build them from
previous streams (`ones`, `twos`, `threes`, `nats`) by making use of
the stream mapping functions (`smap`, `smap2`). *)

(*.....................................................................
Exercise 5: Generate two infinite streams, one of the even natural
numbers, and one of the odds.
.....................................................................*)

let evens _ = failwith "evens not implemented" ;;
let odds _ = failwith "odds not implemented" ;;

(* In addition to mapping over streams, we should be able to use all
the other higher-order list functions you've grown to know and love,
like folding and filtering. So let's implement some. *)
```

```
(*.....................................................................
Exercise 6: Define a function `sfilter` that takes a predicate (that
is, a function returning a `bool`) and a stream, and returns the
stream that contains all the elements in the argument stream that
satisfy the predicate. Here's an example -- generating a stream of
even numbers by filtering the natural numbers for the evens:

   # let evens = sfilter (fun x -> x mod 2 = 0) nats ;;
   val evens : int stream = <fun>
   # first 10 evens ;;
   - : int list = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18]
.....................................................................*)

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

(*.....................................................................
Exercise 7: Now redefine `evens` and `odds` (as `evens2` and `odds2`)
by using `sfilter` to filter over `nats`.
.....................................................................*)

let evens2 _ = failwith "evens with sfilter not implemented" ;;
let odds2 _ = failwith "odds with sfilter not implemented" ;;

(*=====================================================================
Part 2: Eratosthenes' Sieve

Eratosthenes' sieve is a method for generating the prime numbers.
Given a list (or stream) of natural numbers starting with 2, we filter
out those in the tail of the list not divisible by the head of the
list and then apply the sieve to that tail. The first few steps go
something like this: We start with the natural numbers (in the example
here, just a prefix of them).

    2 3 4 5 6 7 8 9 10 11 12 13 14 15...

The first element, 2, is prime. Now we remove numbers divisible by 2
from the tail of the list (marking here with a | the boundary between
the first element and the tail we're currently working on:

    2  |  3 5 7 9 11 13 15...

and apply the sieve to the tail:

    2 3  |  5 7 11 13

and again:

    2 3 5  |  7 11 13
    2 3 5 7  |  11 13
    ...
    2 3 5 7 11 13

Here's the process of sieving a stream of numbers in more detail:

    1. Retrieve the head and tail of the stream. The head is the first
       prime in the result stream; the tail is the list of remaining
       elements that have not been sieved yet. For instance,

       head      | tail
       2         | 3 4 5 6 7 8 9 10 11 ...

    2. Filter out all multiples of the head from the tail.

       head      | filtered tail
       2         | 3 5 7 9 11 ...
```

3. Sieve the filtered tail to generate all primes starting with the first element of the tail.

```
head      | sieved filtered tail
2         | 3 5 7 11 ...
```

4. Add the head on the front of the sieved results.

```
2 3 5 7 11 ...
```

5. Of course, this whole series of computations (1 through 4) should be delayed, and only executed when forced to do so.

```
.....................................................................
Exercise 8: Implement Eratosthenes sieve to generate an infinite
stream of primes.  Example:

# primes = sieve (tail (tail nats)) ;;
# first 4 primes ;;
- : int list = [2; 3; 5; 7]

(You probably won't want to generate more than the first few primes
this way; it'll take too long, depending on how your other stream
functions were implemented. Here are some timings from the solution
code on my laptop:

  n      time for nth prime (seconds)
  1 --   0.00000405
  2 --   0.00001597
  3 --   0.00006604
  4 --   0.00105000
  5 --   0.00343299
  6 --   0.04916501
  7 --   0.19323015
  8 --   3.12322998

Just generating the first eight primes takes over three seconds --
longer if a less efficient `sfilter` is used.  You'll address this
performance problem in the next lab.)

In defining the `sieve` function, the following function may be
useful: *)

(* not_div_by n m -- Predicate returns true if `m` is not evenly
   divisible by `n` *)
let not_div_by (n : int) (m : int) : bool =
  m mod n <> 0 ;;
(*.....................................................................*)

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