CSE 305: Infinite Lists

Lukasz Ziarek

Due: May 7th, at 11:59 pm

1 Overview

In this homework you will be exploring infinite lists in OCaml. This homework is divided into 3 parts, each of which has an associated point total. The total of all parts combines to 100 points.

Part 1: Streams (85 Points)
Part 2: Generalized Printing (15 Points)

Note: Please adhere to the type signatures that we request you to follow as you will be graded exactly based on them. If you do not follow the type signatures you will be assigned 0 points.

2 Requirements and Submission

Submissions must be made on Autolab. For infinite lists the grader will call your functions. For generalized printing, your program must open an out channel, write to it, and close the file.

For both parts, you must include the contents of `streams.ml` provided on Piazza.

Autolab will grade your submission as follows:
Part 1 - Checks for correct `even/odd/squares/fibs/evenFibs/oddFibs/primes`. Your submission does not need to print them.
Part 2 - Checks for correct `printGenList/printList/printPairList`. The grader will call your `printGenList` function. `printList` and `printPairList` must open an out_channel, print to the out_channel, and then close the out_channel. Printing to stdout or stderr will not be given any points. Autolab will check the file for correct output.

3 Streams

Start with the following datatype and function bindings given in `streams.ml` under the General Resources tab in Piazza:

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

exception Subscript
exception Empty
```

```
let head (s :'a stream) : 'a =
  match s () with
    Cons (hd,tl) -> hd
  | Nil -> raise Empty

let tail (s :'a stream) : 'a stream =
  match s () with
    Cons (hd,tl) -> tl
  | Nil -> raise Empty

let null (s : 'a stream) =
  match s () with
    Nil -> true
  | _ -> false

let rec take (n: int) (s: 'a stream) : 'a list =
  match n with
    n when n > 0 -> head s :: take (n - 1) (tail s)
  | 0 -> []
  | _ -> raise Subscript

let rec nth (n: int) (s: 'a stream) : 'a =
  match n with
    n when n > 0 -> nth (n - 1) (tail s)
  | 0 -> head s
  | _ -> raise Subscript

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

let rec filter (s: 'a stream) (f: 'a -> bool) : 'a stream =
  if f (head s)
  then fun () -> Cons (head s, filter (tail s) f)
  else filter (tail s) f

let rec sieve (s: int stream) : int stream =
  fun () -> Cons(head s, sieve (filter (tail s) (fun x -> x mod (head s) <> 0)))

let rec fromn (n: int) = fun () -> Cons (n, fromn (n + 1))
let rec fib n m = fun () -> Cons (n, fib m (n+m))

(* implement the streams and functions below *)

let even : int -> bool = fun x -> true
let odd  : int -> bool = fun x -> true

let squares : int stream = fun () -> Nil
let fibs : int stream = fun () -> Nil
let evenFibs : int stream = fun () -> Nil
let oddFibs : int stream = fun () -> Nil
let primes : int stream = fun () -> Nil

let rev_zip_diff : 'a stream -> 'b stream -> ('b * 'a -> 'c ) -> ('b * 'a * 'c) stream =
  fun a b f -> fun () -> Nil

let rec printGenList : 'a list -> ('a -> unit) -> unit =
  fun l f -> ()

let rec printList : int list -> string -> unit =
  fun l f ->
    let oc = open_out f in
    close_out oc

let rec printPairList : (int * int) list -> string -> unit =
  fun l f ->
    let oc = open_out f in
    close_out oc
```

*Note that a `Nil` constructor has been included to express an empty stream or empty infinite list.

To start, write a function called `even` which returns true if the integer argument is even, false otherwise. Write a function called `odd` which returns true if the integer argument is odd, false otherwise.

Now create five streams: `squares`, `fibs`, `evenFibs`, `oddFibs`, and `primes`. `squares` contains the sequence of perfect squares starting from 1. You may find the `map` function helpful to create such a stream. Then, create a stream called `fibs` that contains the Fibonacci sequence. `evenFibs` and `oddFibs` are similar but they only contain the even Fibonacci numbers and odd Fibonacci numbers, respectively. The last stream is `primes`, which has the sequence of prime numbers.

Write a function called `rev_zip_diff` which will zip together two streams in reverse order (hint: take a look at how the zip function works that we went over in class) and leverage an arbitrary difference function to obtain the difference of every two elements. `rev_zip_diff` should have the following type signature:
`val zip :  'a stream -> 'b stream -> ('b * 'a -> 'c) -> ('b * 'a * 'c) stream`
You may wish to test `rev_zip_diff` by zipping together `evenFibs` and `oddFibs` and a function that gets the integer difference, and then create a concrete list with the `take` function.

4 Generalized Printing

You are tasked with creating generalized list printing functions. Write a function called `printGenList` which takes a list l and a printing function f and applies the function to each element of the list recursively. The function should have the following type signature:
`val printGenList :  'a list -> ('a -> unit) -> unit`

Create a function called `printList` that will pretty print an integer list. You will find the string concatenation operator (ˆ) and the `string_of_int` function useful. `printList` will take an integer

list followed by a string. The string is the name of the output file that you will write the list to. This function should leverage `printGenList` and provide an anonymous function (the `fun ... ->...` construct) that will do the appropriate pretty printing to the output file. This anonymous function should print the element of the list and then a space character. printList should have the following type signature:
`val printList : int list -> string -> unit`

Create a function called `printPairList` that will pretty print a list consisting of integer pairs. The function will take an (int * int) list followed by a string. The string is the name of the output file that you will write the list to. `printPairList` should leverage `printGenList` and provide an anonymous function (the `fun ... ->...` construct) that will do the appropriate pretty printing. This anonymous function should print an open parenthesis, the first element of the pair, a comma, a space, the second element of the pair, and then a close parenthesis followed by a space. `printPairList` should have the following type signature:
`val printPairList :  (int * int) list -> string -> unit`

5 Example output for the entire homework

Note that printList and printPairList must print to the output file, we show the contents of the file as string for example only.

```
even 2;;
- : bool = true
even 3;;
- : bool = false
odd 2;;
- : bool = false
odd 3;;
- : bool = true
take 10 squares;;
- : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
take 10 fibs;;
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34]
take 10 evenFibs;;
- : int list = [0; 2; 8; 34; 144; 610; 2584; 10946; 46368; 196418]
take 10 oddFibs;;
- : int list = [1; 1; 3; 5; 13; 21; 55; 89; 233; 377]
take 10 primes;;
- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29]
take 5 (rev_zip_diff evenFibs oddFibs (fun (x,y) -> x - y));;
- : (int * int * int) list =
[(1, 0, 1); (1, 2, -1); (3, 8, -5); (5, 34, -29); (13, 144, -131)]
printGenList ["how"; "the"; "turntables"] (fun s -> print_string (s ^ " "));;
how the turntables - : unit = ()
printList [2; 4; 6; 8] "printList.txt";; (* "2 4 6 8 " *)
printPairList [(2, 1); (3, 2); (4, 3)] "printPairList.txt";; (* "(2, 1) (3, 2) (4, 3) " *)
```