Functional Programming

Streams and lazy evaluation

Exercise 1. My first impulse was to define lazy list functions as here:

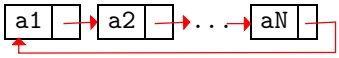
```
let rec wrong_lzip = function
 | LNil, LNil -> LNil
 | LCons (a1, lazy l1), LCons (a2, lazy l2) ->
     LCons ((a1, a2), lazy (wrong_lzip (l1, l2)))
 | _ -> raise (Invalid_argument "lzip")

let rec wrong_lmap f = function
 | LNil -> LNil
 | LCons (a, lazy l) -> LCons (f a, lazy (wrong_lmap f l))
```

What is wrong with these definitions – for which edge cases they do not work as intended?

Exercise 2. Cyclic lazy lists:

1. Implement a function `cycle : 'a list -> 'a llist` that creates a lazy list with elements from standard list, and the whole list as the tail after the last element from the input list.

   `[a1; a2; ...; aN]` ↦ 

   Your function `cycle` can either return `LNil` or fail for an empty list as argument.

2. Note that `inv_fact` from the lecture defines the power series for the $\exp(\cdot)$ function ($\exp(x) = e^x$). Using `cycle` and `inv_fact`, define the power series for $\sin(\cdot)$ and $\cos(\cdot)$, and draw their graphs using helper functions from the lecture script `Lec7.ml`.

Exercise 3. * Modify one of the puzzle solving programs (either from the previous lecture or from your previous homework) to work with lazy lists. Implement the necessary higher-order lazy list functions. Check that indeed displaying only the first solution when there are multiple solutions in the result takes shorter than computing solutions by the original program.

Exercise 4. *Hamming's problem.* Generate in increasing order the numbers of the form $2^{a_1}3^{a_2}5^{a_3}...p_k^{a_k}$, that is numbers not divisible by prime numbers greater than the kth prime number.

- In the original Hamming's problem posed by Dijkstra, $k = 3$, which is related to http://en.wikipedia.org/wiki/Regular_number.

Starter code is available in the middle of the lecture script `Lec7.ml`:

```
let rec lfilter f = function
 | LNil -> LNil
 | LCons (n, ll) ->
     if f n then LCons (n, lazy (lfilter f (Lazy.force ll)))
     else lfilter f (Lazy.force ll)

let primes =
 let rec sieve = function
     LCons(p,nf) -> LCons(p, lazy (sieve (sift p (Lazy.force nf))))
   | LNil -> failwith "Impossible! Internal error."
 and sift p = lfilter (function n -> n mod p <> 0)
in sieve (lfrom 2)

let times ll n = lmap (fun i -> i * n) ll;;
```

```
let rec merge xs ys = match xs, ys with
  | LCons (x, lazy xr), LCons (y, lazy yr) ->
     if x < y then LCons (x, lazy (merge xr ys))
     else if x > y then LCons (y, lazy (merge xs yr))
     else LCons (x, lazy (merge xr yr))
 | r, LNil | LNil, r -> r

let hamming k =
 let pr = ltake k primes in
 let rec h = LCons (1, lazy (
   <TODO> )) in
 h
```

Exercise 5. Modify `format` and/or `breaks` to use just a single number instead of a stack of booleans to keep track of what groups should be inlined.

Exercise 6. Add **indentation** to the pretty-printer for groups: if a group does not fit in a single line, its consecutive lines are indented by a given amount `tab` of spaces deeper than its parent group lines would be. For comparison, let's do several implementations.

1. Modify the straightforward implementation of `pretty`.
2. Modify the first pipe-based implementation of `pretty` by modifying the `format` function.
3. Modify the second pipe-based implementation of `pretty` by modifying the `breaks` function. Recover the positions of elements – the number of characters from the beginning of the document – by keeping track of the growing offset.
4. * Modify a pipe-based implementation to provide a different style of indentation: indent the first line of a group, when the group starts on a new line, at the same level as the consecutive lines (rather than at the parent level of indentation).

Exercise 7. Write a pipe that takes document elements annotated with linear position, and produces document elements annotated with (line, column) coordinates.

Write another pipe that takes so annotated elements and adds a line number indicator in front of each line. Do not update the column coordinate. Test the pipes by plugging them before the `emit` pipe.

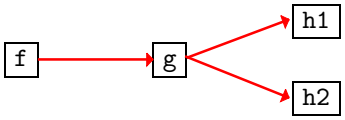
```
1: first line
2: second line, etc.
```

Exercise 8. Write a pipe that consumes document elements `doc_e` and yields the toplevel subdocuments `doc` which would generate the corresponding elements.

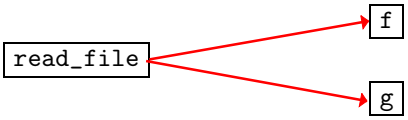
You can modify the definition of documents to allow annotations, so that the element annotations are preserved (`gen` should ignore annotations to keep things simple):

```
type 'a doc =
  Text of 'a * string | Line of 'a | Cat of doc * doc | Group of 'a * doc
```

Exercise 9. * Design and implement a way to duplicate arrows outgoing from a pipe-box, that would memoize the stream, i.e. not recompute everything "upstream" for the composition of pipes. Such duplicated arrows would behave nicely with pipes reading from files.



Does not recompute `g` nor `f`.



Reads once and passes all content to `f` and `g`.