Homework 3
COMP 302 Programming Languages and Paradigms

Francisco Ferreira and Brigitte Pientka
McGill University: School of Computer Science

Due Date: 3 November 2017

Your homework is due at the beginning of class on Nov 3, 2017. All code files must be submitted electronically using my courses. **Your program must compile.** Solutions should correct (i.e. produce the correct result), be elegant and compact, and take advantage of OCaml's pattern matching. Please consult the style guides posted on the course website to get more information regarding what constitutes good style.

This assignment contains several optional questions, these are meant to let you practice, the model solutions will contain their answers. It also allows you to get some extra credit. If you have questions when solving them you can always ask during office hours! Solving them should be a great way of learning!

Q1. Unfolding is like ɡnibloℲ (25pts)

In this question we will practice with higher-order functions by discussing the function `unfold`. The function `unfold` takes a value that will generate a list. This is in contrast to `fold_left` and `fold_right` which both take lists to generate a value. The relation between folds and unfolds is the beginning of a wonderful tale in computer science. But let's not get ahead of ourselves.

The function `unfold` takes three arguments: 1) a function `f` that from a seed value generates a new element of the list together with the seed for the next element, 2) a function `stop` to stop the generation, and 3) the current seed value `b`.

The implementation is straightforward. Unless the `stop` function indicates us to stop, use the generator function `f` to generate a new element together with the next seed to recursively generate elements of the list.

```
let rec unfold (f : 'seed -> ('a * 'seed)) (stop : 'b -> bool) (b : 'seed) : 'a list =
  if stop b then []
  else let x, b' = f b in
       x :: (unfold f stop b')
```

With this function it is easy to generate all the natural numbers up to a certain point. The generator function simply returns the seed, and increments the next seed in the following manner:

```
let nats max = unfold (fun b -> b, b + 1) (fun b -> max < b) 0
```

A couple of warm-up questions. Using `unfold` write the following functions:

Q1.1 (5 points) **let** `evens (max : int): int list = ...` returns a list of successive even numbers starting with 0.

Q1.2 (10 points) The Fibonacci sequence is a sequence of integers that starts with two 1s, and then each successive number is the addition of the previous two. The beginning of the sequences is: $1, 1, 2, 3, 5, 8, 12, \ldots$. Write a function **let** `fib (max : int): int list = ...` that returns a list of Fibonacci numbers that are smaller than `max`.

Pascal's triangle is a number triangle with numbers arranged in staggered rows such that it contains binomial coefficient. We show Pascal's triangle below in Figure 1.

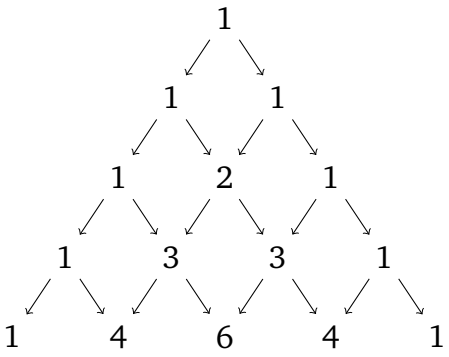


Figure 1: Pascal's triangle

In general, a row n in Pascal's triangle may be constructed from row $n-1$ in the following manner: the first element in row n is simply the first element of row $n-1$, or more precisely $0+1$; the second element in row n can be obtained by adding the first and second element in row $n-1$; etc. In general, the i-the element in row n is obtained by adding the $(i-1)$-th element and the i-th element in row $n-1$.

Q1.3 (10 points) Write a function **let** `pascal (max : int): int list list =. ..` that returns the list of rows of the Pascal triangle that are shorter than `max`.

Q1.4 – Optional [Extra Credit 10 points] And, finally, consider the function `zip` that takes two lists and returns a tuple of elements from each list. It is precisely defined by this recursive function:

```
let rec zip (l1 : 'a list) (l2 : 'b list) : ('a * 'b) list =
match l1, l2 with
| [], _ -> []
| _, [] -> []
| x::xs, y::ys -> (x, y):: zip xs ys
```

There is a nice short implementation of this function using unfold, write that function.

Q2. Fleeting memories of values past (30pts)

In this question we will practice higher-order functions, references, and closures. As we saw in class, in OCaml functions are values, and as such they can be passed as parameters and returned from functions. Let's consider, as in the notes, very slow computations. In particular the ugly computation implemented in this function:

```
let ugly x =
  let rec ackermann m n = match (m , n) with
    | 0 , n -> n+1
    | m , 0 -> ackermann (m-1) 1
    | m , n -> ackermann (m-1) (ackermann m (n-1))
  in
  ackermann 3 x
```

This computation grows very quickly, and it gets slow very soon. In this question we will write a higher-order function that tries to mitigate that.

The idea is to write a function that accepts the slow function as a parameter, and caches the recently used input values and results. Let's consider the trivial function that caches exactly zero results:

```
let memo_zero (f : 'a -> 'b) : 'a -> 'b = f

let ugly' = memo_zero ugly
```

If we call that function with `ugly` (that we defined above) it returns a function that behaves exactly like `ugly`. In fact this function does not help at all. The purpose of this exercise is to write a function that remembers the last result of the computation, and if it is called with the same input it returns the cached result instead of recomputing it. This is often referred to as memoization.

Q2.1 (15 points) Write the function `memo_one`. It takes a function `f` as a parameter and returns a function that when given an input `x` computes `f x` and stores in a reference both the current input `x` together with the return value `f x`, i.e. it store the pair consisting of `x` and the return value `f x`. Subsequently, it tries to avoid recomputing the value by returning the cached value when called on repeated inputs.

> **UPDATED: 31 Oct 2017:** If `memo_one` is given an input `y` which is different from the value `x`, it will update the current input-output pair to store the new current input `y` together with the new output value `f y`. See example output below.

The caching should work as follows:

```
let ugly' = memo_one ugly

let u1 = ugly' 3 (* this one calls ugly with 3 *)
let u2 = ugly' 3 (* this one uses the stored value *)
let u3 = ugly' 1 (* the stored value is for 3 so it calls ugly *)
let u4 = ugly' 2 (* the stored value is for 1 so it calls ugly *)
```

```
let u5 = ugly' 10 (* the stored value is for 2 so it calls ugly and takes a couple of seconds *)
let u6 = ugly' 10 (* the one uses the stored value and returns immediately *)
```

Having only one value stored is not enough. Next, we want to generalize to remembering an arbitrary number of past results.

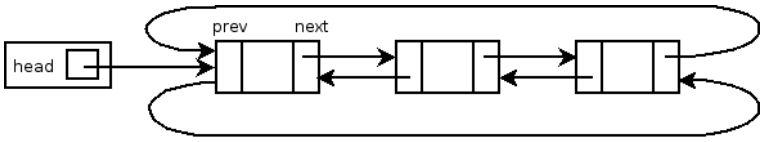
Q2.2 (15 points) Write `let memo_many (n : int)(f : 'a -> 'b): 'a -> 'b = ...` that behaves like `memo_one` except that it remembers the last `n` values. To store the values you can use a list of references of the appropriate length. If you feel adventurous you can learn and use OCaml arrays that are described in https://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html.

> **UPDATED: 31 Oct 2017:** Please use a first-in-first-out strategy to manage the buffer which stores `n` input-output pairs.

Q3. Ouroboros, alchemy and archetype (45pts)

The notes describe reference lists, that is a list that contains a mutable reference to the tail of the list. In this question we explore a doubly linked list, that is a list with forward and backward pointers. Having both pointers helps by making the traversal of the list easier. Taking advantage of the double links we want to write a circular list where the last node points to the first and vice-versa. These kinds of data structures are common in audio processing libraries, where audio buffers are often implemented as circular buffers. Additionally, representing some board games might take advantage of circular lists. Consider representing a Monopoly board for example!

To illustrate, we can draw doubly-linked circular list as follows:



We define circular lists using a record with three fields. The field `p` contains the location of the previous cell, the field `data` contains the data we store, and the field `n` contains the location of the next cell. We declare both the field `p` and `n` to be **mutable** which means these fields can be updated by assigning them a new value.

```
type 'a cell = { mutable p : 'a cell; data : 'a ; mutable n : 'a cell}

type 'a circlist = 'a cell option
```

Let's start with some examples. Empty lists are very simple, and they are just setting the option type to `None`:

```
(* An empty circular list *)
let empty :'a circlist = None
```

Another important function is the one that creates the singleton list. Observe how we recursively define `pointer` by setting both `p` and `n` to itself!

```
(* A singleton list that contains a single element *)
let singl (x : 'a) : 'a circlist =
  let rec pointer = {p = pointer ; data = x ; n = pointer} in
  Some pointer
```

Because of the circularity, we can always follow the pointers in the next and previous directions. The following two functions return a list that starts with the next and previous nodes respectively:

```
(* Move to the next element *)
let next : 'a circlist -> 'a circlist = function
  | None -> None
  | Some cl -> Some (cl.n)

(* Move to the previous element *)
let prev : 'a circlist -> 'a circlist = function
  | None -> None
  | Some cl -> Some (cl.p)
```

Note that we access the field `p` (or `n`) of a record `c1` by simply writing `c1.p` (or `c1.n`). Since both fields are declared mutable, we can update them. For example, to write the location of `c1.n` into the field `c1.p`, we write `c1.p <- c1.n`.

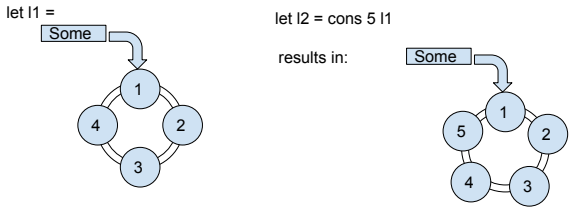


Figure 2: Adding an element to a circular list

Q3.1 (15 points) Write the function **let** `cons (x : 'a)(xs : 'a circlist): 'a circlist =...` that adds an element to the circular list. This element should be inserted right before current head of `xs` (i.e.: using the previous pointer in the node). Figure 2 contains an illustration of how `cons` modifies a circular list. This is the analogous of the `::` operator for regular lists.

> **UPDATED: 31 Oct 2017:** Once, you have implemented `cons` you can use the provided function `from_list` to turn a list of elements into a circular list. Note however, that `from_list` builds up the circular list in reverse order.

Q3.2 (10 points) Write the function **let rec** `length : 'a circlist -> int =...` that returns the number of cell nodes in a list.

Q3.3 (10 points) Write the function **let** `to_list : 'a circlist -> 'a list =...` that converts a circular list into a regular OCaml list.

> **UPDATED: 31 Oct 2017:** Note that `from_list`, the function we provide to turn a list into a circular list, builds up the circular list in reverse order. Hence, `to_list (from_list [1;2;3;4])` returns `[4;3;2;1]`.

Q3.4 (10 points) Write the function **let** `rev (l : 'a circlist): 'a circlist =...` that reverses the directions of the circular list. As a result the lists produced by `to_list` should be reversed.

Q3.5 OPTIONAL [Extra Credit 5 points]
Write the function **let** `map (f : 'a -> 'b): 'a circlist -> ' b circlist =...` that maps a function to a circular list in an analogous way to `List.map`.

Q3.6 OPTIONAL [Extra Credit 15 points]
Write the function **let** `eq (l1 : 'a circlist)(l2 : 'a circlist): bool =...` that compares two lists. In particular we want the following properties to hold of equality:

1. Given a list `l` it is always the case that `eq l (next l)=` **true**
2. Given a list `l` it is always the case that `eq l (prev l)=` **true**
3. Lists with different lengths may be equal if their contents go together one by one.

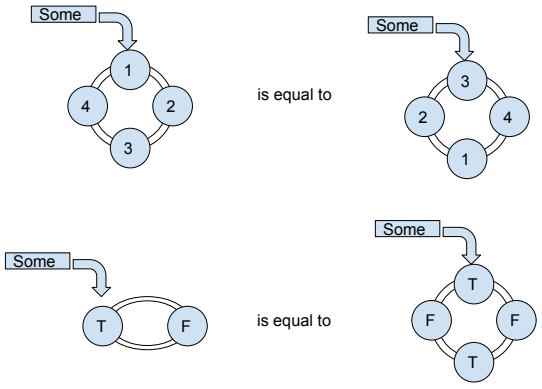


Figure 3: Equality of repeating lists

See diagrams in Figure 3 for reference. The idea is that equality is independent or rotation, and independent or repeating patterns as long as they correspond one to one. This is a challenging question, you can consider starting without the third constraint for equality.

Remember that all these functions need to preserve the invariant that lists are circular, so following the pointers in either direction should eventually bring you back to the same node. All functions need to be implemented using the circular list representation, for example you are not allowed to use the functions `to_list` and `from_list`.

Acknowledgments: Many thanks to François Thiré and his assignments from ENS Cachan that were an inspiration for these questions!