Final exam, March 14th, 2014

- Duration: 3 hours. There are **3** independent exercises, plus **1** bonus question.
- Allowed documents: lecture notes, personal notes, **but no electronic devices.**
- **Mobile phones must be switched off.**
- Answers may be written in English or French.
- Universal quantification of free variables at top-level may be left implicit.

1 Repeated elements in an array

The goal of this first exercise is to design and prove correct algorithms to test whether, in a given array of integers, one of the values is repeated (i.e., occurs 2 times or more). All these algorithms are implemented as Why3 programs with the same profile.

```
function check_repetition (a:array int) (n:int) : bool
```

All functions also share the same specification, which is informally described as follows:

- all values in the array a are assumed to belong to the interval $[0..n-1]$ (inclusive).
- the function returns true if and only there are two distinct indices i, j (between 0 and $a.length-1$) such that $a[i]$ and $a[j]$ are equal.
- the array a must not be modified.

Question 1.1. State the informal specification above as a formal contract.

Question 1.2. Consider a first algorithm, implemented as shown below.

```
function check_repetition (a:array int) (n:int) : bool
  let i = ref 0 in let j = ref 0 in let res = ref False in
  while !i < a.length - 1 do
    j := !i + 1;
    while !j < a.length do
      if a[!i] = a[!j] then res := True;
      j := !j + 1;
    done;
    i := !i + 1
  done;
  !res
```

Propose suitable loop invariants for each of the two loops. Justify briefly why these invariants allow to prove the algorithm correct (max 10 lines, focusing only on important informations).

Question 1.3. Propose an algorithm similar as the one above, but that raises an exception instead of using the variable `res`, in order to exit the function as early as possible. Propose suitable loop invariants, then explain briefly why they are suitable for proving your algorithm (max 10 lines). Comment on the difference between this new algorithm and the original one.

Question 1.4. Consider now a more efficient algorithm that relies on an auxiliary array.

```
exception Break
function check_repetition (a:array int) (n:int) : bool
  let tmp = Array.create n False in
  let i = ref 0 in
  try
    while !i < a.length do
      if tmp[a[!i]] then raise Break;
      tmp[a[!i]] := True;
      i := !i + 1
    done;
    False
  with Break → True
  end
```

Propose a suitable loop invariant for the loop. Justify why this invariant allows to prove the algorithm correct (max 10 lines).

2 Circular lists

This section focuses on the formalization of circular lists and doubly linked circular lists in Separation Logic. For simplicity, we consider only lists of integers in this exercise.

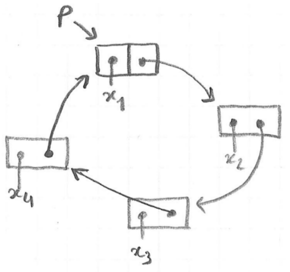
Recall the type of mutable list cells.

```
type cell = { mutable hd : int; mutable tl : cell }
```

Recall also the definition of the representation predicates $\mathsf{MlistSeg}$ and Mlist.

$$\begin{array}{lcl} p \leadsto \mathsf{MlistSeg}\, q\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \Rightarrow [p = q] \\ & & \mid x :: L' \Rightarrow \exists p'.\ p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\} \ast p' \leadsto \mathsf{MlistSeg}\, q\, L' \\ p \leadsto \mathsf{Mlist}\, L & \equiv & p \leadsto \mathsf{MlistSeg}\, \mathsf{null}\, L \end{array}$$

Question 2.1. When the last cell of the list, instead of pointing to `null`, points back to the first cell of the list, we have a circular list.



Give a definition for the representation predicate $p \leadsto \mathsf{CircList}\, L$, which asserts that, when starting at the cell p and following the pointers until reaching p again, we find the items described by the list L. Your definition should build directly on top of "$\mathsf{MlistSeg}$" (i.e., it should not mention any cell predicate of the form $q \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}y|\!\}$), and it should use the equivalence "$p = \mathsf{null} \Leftrightarrow L = \mathsf{nil}$" to handle the particular case where the list is empty.

Question 2.2. The function `read` takes as argument a pointer on a nonempty circular list, and returns the content of the item at this position.

```
let read p = p.hd
```

Give a specification for `read`, with a pre-condition of the form "$p \leadsto \mathsf{CircList}\,(v :: L)$".

Question 2.3. The function `forward` takes as argument a pointer on a nonempty circular list, and returns the pointer on the next item.

```
let forward p = p.tl
```

Specify `forward`, with a post-condition of the form "$\lambda q.\ q \leadsto \mathsf{CircList}\,K$", for some list K.

Question 2.4. Prove that the code of forward satisfies the specification that you proposed in the previous question (15 lines max). Clearly indicate where focus or defocus rules are involved in the proof (no need to prove these rules).

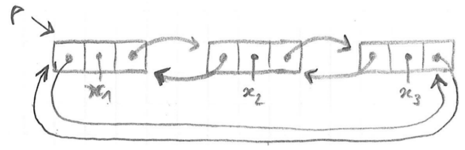
Question 2.5. We next consider a function to insert an item at the head of a circular list.

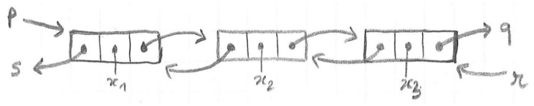
```
let insert v p =
  if p == null then begin
    let q = { hd = v; tl = null } in
    q.tl <- q;
    q
  end else begin
    let q = { hd = p.hd; tl = p.tl } in
    p.hd <- v;
    p.tl <- q;
    p
  end
```

Specify the above function, then prove its correctness (15 lines max).

Question 2.6. In order to be able to efficiently navigate the circular lists backwards, we extend it with reverse pointers. A cell is now represented as follows.

```
type cell = {
  mutable prev : cell;
  mutable item : int;
  mutable next : cell; }
```





The goal is to define a predicate "$p \leadsto \mathsf{DbList}\,L$" to characterize such a doubly linked list. The elements are described by the list L, when starting at the cell at address p and following the next pointers until reaching p again. Note that the predicate should use the equivalence "$p = \mathsf{null} \Leftrightarrow L = \mathsf{nil}$" to handle the particular case where the list is empty.

Hint: Start by defining a predicate of the form "$p \leadsto \mathsf{DbListSeg}\,q\,r\,s\,L$" that describes a segment of a doubly linked list storing the elements L. As shown in the rightmost picture above, the first cell, at address p, stores the address s in its prev field, and the last cell, at address r, stores the address q in its next field. When L is empty, enforce $p = q$ and $r = s$.

3 Filtering

In this exercise, we are interested in specifying and verifying, in Separation Logic, functions that filters particular elements from a list. We assume the existence of a predicate called

Even, which characterizes even integers. We also assume the existence of a logical function Filter, which applies to a predicate of type "$A \to \mathsf{Prop}$" and to a list of type "$\mathsf{list}\, A$". For example, "$\mathsf{Filter}\,\mathsf{Even}\, L$" filters the even elements from a list L, in the logic.

Question 3.1. We first consider a purely-functional implementation of filter on a pure list of integers, specialized to filtering even numbers.

```
let rec list_filter_even l =
  match l with
  | [] -> []
  | x::t -> let q = list_filter_even t in
            if x mod 2 = 0 then x::q else q
```

Give a specification for this function, using a Separation Logic triple.

Question 3.2. We now adapt the above function to mutable lists as follows:

```
let rec mlist_filter_even p =
  if p == null then null else begin
    let q = mlist_filter_even p.tl in
    if p.hd mod 2 = 0
      then (p.tl <- q; p)
      else q
```

Give a Separation Logic specification for this function, making use of the predicate Mlist.

Question 3.3. Prove that `mlist_filter_even` satisfies the specification that you claimed in the previous question (in 20 lines max).

Question 3.4. We generalize the function so that it may take as argument a filter function `f`, of type `int` → `bool`.

```
let rec mlist_filter f p =
  if p == null then null else begin
    let q = mlist_filter p.tl in
    if f p.hd
      then (p.tl <- q; p)
      else q
```

Give the specification of this function, assuming that `f` does not perform any visible side effect. For this question, quantify explicitly all the variables involved.

Question 3.5. We are now interested in the generalization to mutable lists containing items of arbitrary types. For this purpose, we need in particular to quantify over the representation predicate R associated with the items stored in the list, and use Mlistof instead of Mlist.

Moreover, we assume that the function `f` may read (but not write) values in the heap; in particular, it may not alter any of the heap invariants. So, the function `f`, in addition to being able to access the representation of the items that it receives as argument, should also be able to read in other parts of the heap. These pieces of heap should be described by an invariant, call it "I", that should appear in both the pre- and the post-condition of `mlist_filter`.

Give the corresponding specification of `mlist_filter`.

Question 3.6. Consider the application of `mlist_filter` to a mutable list "`p`" made of possibly-aliased reference cells, each of these reference storing an integer. The goal is to filter the reference cells that store an even number. Using the "Group" representation predicate to describe the possibly-aliased reference cells, state a triple that specifies the behavior of the following expression, which you may refer to as "*expr*".

```
mlist_filter (fun r -> !r mod 2 = 0) p
```

Question 3.7. Prove that the function "`fun r -> !r mod 2 = 0`" satisfies the hypothesis made on `f` in the specification of `mlist_filter` in question 3.5. Clearly state the focus rule that you exploit (max 12 lines for the proof).

4 Bonus: xor-based doubly linked lists

This bonus question is an extension of the exercise on circular lists. You should only consider this exercise when done with all the other questions.

Question 4.1. There exists an alternative representation of doubly linked lists that improves the space efficiency. The idea is to replace the prev and next fields with a single field that stores the xor value of the addresses of the previous and next cells in the list. By maintaining at all time the address of the current cell and that of the previous cell, and by exploiting properties of the xor operation, it is possible to navigate the list in both directions in constant time. More precisely, the implementation relies on the following properties of the xor operation:

$$z = x \,\mathsf{xor}\, y \quad\Rightarrow\quad x = z \,\mathsf{xor}\, y \quad\wedge\quad y = z \,\mathsf{xor}\, x$$

This representation of doubly linked lists allows saving 1/3 of the space required for representing the list, at the cost of performing only a few extra arithmetic operations. It was commonly used back in the days where the amount of memory available was the main bottleneck. A formalization of this tricky representation appears in Reynold's seminal paper on Separation Logic (2002).

The implementation is as follows. Note that xor is written `lxor` in OCaml; we assume this function to operate on values of any type (and that it does not interfer with the tags used by the garbage collector).

```
type iterator = {
  mutable prev : cell;
  mutable cur : cell; }

type cell = {
  mutable item : int;
  mutable sum : cell; } (* xor of prev and next *)

let read (i:iterator) =
  i.cur.item

let forward (i:iterator) =
  let c = i.cur in
  let p = i.prev in
  let n = c.sum lxor p in (* xor of cur.sum and prev *)
  i.cur <- n;
  i.prev <- c

let backward (i:iterator) = ... (* similar *)
```

Define the representation predicate "$i \rightsquigarrow \mathsf{DbXorList}\, L$", which asserts that the iterator i holds a doubly linked circular list implemented using xor and that L describe the list of elements, starting from the cell at address cur. When L is empty, the pointer i should be null. Otherwise, the representation predicate should describe both the iterator record and the set of all the cells involved in the list structure. Then, based on this definition, specify and prove the code of the function forward.