Final exam, March 3rd, 2016

- Duration: 3 hours. There are **3** independent exercises.
- Allowed documents: lecture notes, personal notes, **but no electronic devices.**
- **Mobile phones must be switched off.**
- Answers may be written in English or French.
- All variables must be bound or quantified explicitly.

Reminder Exercises 2 and 3 involve mutable lists. Recall that mutable lists are implemented using cells with a head and a tail field, and that the empty list is represented using the null pointer.

```
type 'a cell = { mutable hd : 'a; mutable tl : 'a cell } (* or null *)
```

Recall that a mutable linked list is described by the heap predicate $p \rightsquigarrow \mathsf{Mlist}\, L$, where p denotes the location of the first cell (or null), and where L describes the values stored in the head fields of the cells.

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

1 Binomial Coefficients

Definition The binomial coefficients $C(n,p)$, for $0 \leqslant p \leqslant n$, are defined recursively as follows.

$$
\begin{array}{lcll}
C(n,0) & = & 1 & (0 \leqslant n) \\
C(n,n) & = & 1 & (0 \leqslant n) \\
C(n,p) & = & C(n-1,p-1) + C(n-1,p) & (0 < p < n)
\end{array}
$$

Program A The code below computes the binomial coefficients for arguments up to some value n, storing them into a matrix called M. The matrix M is declared globally, and has dimension N by N, for some fixed constant N.

```
let triangle (n:int) : unit
  requires { 0 ⩽ n < N }
  writes  { M }
  ensures { forall i,j. 0 ⩽ j ⩽ i ⩽ n → M[i,j] = C(i,j) }
= for i = 0 to n do
    M[i,0] ← 1;
    M[i,i] ← 1;
    for j = 1 to i-1 do
      M[i,j] ← M[i-1,j] + M[i-1,j-1];
    done
  done
```

Question 1.1. Propose loop invariants for the above code, both for the external loop and the internal loop. Explain why your invariants are sufficient to prove the code correct (max 10 lines).

Program B The code below is a variant of the previous program that computes the rows iteratively, in-place inside a single array of length $n+1$. As suggested by the keyword `downto`, the internal loop iterates downwards, meaning that j takes the values $i-1, i-2, ..., 1$.

```
let binomial (n:int) : int
  requires { 0 ⩽ n }
  ensures { forall i. 0 ⩽ i ⩽ n, result[i] = C(n,i) }
= let a = Array.make (n+1) 0 in  (* a fresh array of length n+1, initialized with 0 *)
  for i = 0 to n do
    a[i] ← 1;
    for j = i-1 downto 1 do
```

```
    a[j] ← a[j] + a[j-1];
  done
done;
a
```

Question 1.2. The general form of a "`for .. downto`" loop is:

$$t \quad \equiv \quad \texttt{for}\ i = v_1\ \texttt{downto}\ v_2\ \texttt{do invariant}\ \{\ I\ \}\ e\ \texttt{done}$$

Propose a formula for the weakest precondition $\mathrm{WP}(t, Q)$, where the term t denotes such a loop and where Q denotes a postcondition. Justify every parts of the formula that you propose (max 10 lines).

Question 1.3. Propose loop invariants for program B, both for the external loop and the internal loop. Explain why your invariants are sufficient to prove the code correct (max 10 lines).

Program C Consider the program B with the body of the function modified as follows.

```
let a = Array.make (n+1) 1 in  (* a fresh array of length n+1, initialized with 1 *)
for i = 1 to n do
  for j = i-1 downto 1 do
    a[j] ← a[j] + a[j-1];
  done
done;
a
```

Question 1.4. Is the code of program C provable with the same loop invariants as those of program B? If yes, explain why. If no, propose alternative invariants for program C, and explain why these new invariants suffice (max 10 lines).

2 Exceptions in Separation Logic

Representation of exceptions We consider an extension of Separation Logic to support exceptions. For simplicity, we here assume that an exception always carries an integer value. The result of the evaluation of a term is described by the inductive data type Res, defined as follows.

```
Inductive Res : Type → Type := | Val : A → Res A | Exn : int → Res A.
```

In this context, a triple is written $\{H\}\ t\ \{Q\}$, where the pre-condition H has type Hprop (i.e. Heap → Prop) and the post-condition Q has type $\mathsf{Res}\,A \to \mathsf{Hprop}$, where A corresponds to the type of t. We define the following entailment relations:

$$\begin{array}{lcll} H \rhd H' & \equiv & \forall (h : \mathsf{Heap}). & H\,h \Rightarrow H'\,h \\ Q \rhd Q' & \equiv & \forall (A : \mathsf{Type})(r : \mathsf{Res}\,A). & Q\,r \rhd Q'\,r \\ Q \rhd_{\mathsf{val}} Q' & \equiv & \forall (A : \mathsf{Type})(v : A). & Q\,(\mathsf{Val}\,v) \rhd Q'\,(\mathsf{Val}\,v) \\ Q \rhd_{\mathsf{exn}} Q' & \equiv & \forall (n : \mathsf{int}). & Q\,(\mathsf{Exn}\,n) \rhd Q'\,(\mathsf{Exn}\,n). \end{array}$$

In the presence of exceptions, the reasoning rule for values can be stated as follows.

$$\frac{H \rhd Q\,(\mathsf{Val}\,v) \qquad Q \rhd_{\mathsf{exn}} [\mathsf{False}]}{\{H\}\ v\ \{Q\}}\ \textsc{val}$$

Question 2.1. A term of the form "$\mathsf{raise}\,n$" is used to throw an exception carrying the integer value n. State a rule for exceptions with a conclusion of the form $\{H\}\ (\mathsf{raise}\,n)\ \{Q\}$.

Question 2.2. State a reasoning rule for let-bindings, following the template given below.

$$\frac{\{...\}\ t_1\ \{...\} \qquad (\forall x.\ \{...\}\ t_2\ \{...\}) \qquad ... \rhd_{\mathsf{exn}} ...}{\{H\}\ (\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}\ \textsc{let}$$

Question 2.3. The try-catch construct, written "$\mathsf{try}\,t_1\,\mathsf{with}\,x \mapsto t_2$", evaluates as follows. If t_1 returns a value v, then the construct returns v. Otherwise, if t_1 throws an exception carrying n, then the term t_2 is executed, with x bound to n. State a reasoning rule for the try-catch construct.

Remark: the next two questions are independent from the previous ones.

Question 2.4. Give a (most-general) specification for the function `nth`, which returns the i-th element of a pure list, and whose code is shown below. Hint: state the post-condition in the form "$\lambda r.$ match r with ...", and decompose l as $l_1 +\!\!+ x :: l_2$ in case of normal termination.

```
let rec nth (i:int) (l:'a list) : 'a =
  if i < 0 || l = [] then raise 3
  else if i = 0 then head l
  else nth (i-1) (tail l)
```

Question 2.5. Give a (most-general) specification for the function `mnth_cell`, which returns the *address* of i-th cell of a mutable list (and not the *contents* of this cell), and whose code is shown below. Hint: the post-condition must, in all cases, return the ownership of all the cells of the list.

```
let rec mnth_cell (i:int) (p:'a cell) : 'a cell =
  if i < 0 || p == null then raise 3
  else if i = 0 then p
  else mnth_cell (i-1) p.tl
```

3 Hashsets

We assume the existence of a polymorphic hash function h, which, when applied to an argument x, returns a non-negative integer called the *hash value* of x, written $h(x)$.

```
val h : 'a -> int
```

A hashset is a data structure that represents a set of elements using an array of *buckets*. If the array has length n (for some $n > 0$), then an element x from the set is stored in the bucket at index "$h(x)$ mod n".

Representation 1 First, we assume that each bucket is represented as a pure list, as shown below.

```
type 'a hashset1 = ('a list) array
```

Such a data structure is described by a heap predicate of the form $p \rightsquigarrow \mathsf{Array}\, L$. Here, the list L has type $\mathsf{list}\,(\mathsf{list}\, a)$, where a denotes the type of the elements. Thus, $L[i]$ denotes the contents of the bucket at index i in the main array. Thereafter, we write $|L|$ for the length of the list L, thus we have: $n = |L|$.

Question 3.1. To describe such a hashset, we define the following heap predicate:

$$p \rightsquigarrow \mathsf{HashSet1}\, E \;\equiv\; \exists L.\; p \rightsquigarrow \mathsf{Array}\, L \star [\mathsf{Repr1}\, L\, E].$$

Give the definition of Repr1 (of type $\mathsf{list}\, a \to \mathsf{set}\, a \to \mathsf{Prop}$), which should assert that the elements of E are stored in the appropriate buckets, and that the buckets contain only elements from the set E.

Question 3.2. Consider the remove function shown below. Give a specification for this function in terms of HashSet1. Make sure to properly quantify all variables.

```
let remove1 (p:'a hashset1) (a:'a) : unit =
  let k = h(a) mod (Array.length p) in
  p.(k) <- List.filter (fun x -> x <> a) p.(k)
```

Question 3.3. Give a specification for the OCaml function `List.filter`, which filters elements from a pure list l, in order to describe the simple case where the filter function is applied to a function f that does not perform any side effects. You may assume the existence of a logical filter function called Filter, of type $\forall a.\ (a \to \mathsf{Prop}) \to \mathsf{list}\, a \to \mathsf{list}\, a$.

Question 3.4. Give another specification for the function `List.filter`, this time to describe the case where the filter function is applied to a function f that might perform side effects on the global state. Hint: introduce an invariant J of type $\mathsf{list}\, a \to \mathsf{list}\, a \to \mathsf{Hprop}$.

Representation 2 We now consider buckets represented using disjoint mutable lists.

```
type 'a hashset2 = ('a cell) array
```

Question 3.5. The recursive function `mremove` shown below removes all occurences of an element from a mutable list. Give a specification for this function in terms of the heap predicate Mlist. Here again, you may assume the existence of a logical filter function called Filter, of type $\forall a.\ (a \to \mathsf{Prop}) \to \mathsf{list}\, a \to \mathsf{list}\, a$.

```
let rec mremove (a:'a) (s:'a cell) : 'a cell =
  if s == null then s else begin
    let t = mremove a s.tl in
    if s.hd = a then t else begin
      s.tl <- t;
      s
    end
  end
```

Question 3.6. Prove that the code of `mremove` is correct with respect to the specification proposed in question 3.5. Your proof should clearly distinguish the various cases. Make sure to mention the names of the structural rules of the logic being used. (Expected answer: 15 lines.)

Question 3.7. Consider the function `remove2` shown below.

```
let remove2 (p:'a hashset2) (a:'a) : unit =
   let k = h(a) mod (Array.length p) in
   in p.(k) <- mremove a p.(k)
```

State a specification for this function where the precondition is: $p \rightsquigarrow \mathsf{Arrayof}\,\mathsf{Mlist}\,L$, and where the post-condition takes the form: $p \rightsquigarrow \mathsf{Arrayof}\,\mathsf{Mlist}\,L'$.

Question 3.8. Prove the correctness of the specification proposed at the previous question for `remove2`, by exploiting the specification of `mremove` from question 3.5. (Expected answer: 12 lines.) You may assume the following equality to be already established:

$$p \rightsquigarrow \mathsf{Arrayof}\,R\,L \;=\; p \rightsquigarrow \mathsf{Arraysize}\,|L| \;\star\; p \rightsquigarrow \mathsf{Cellof}\,R\,i\,(L[i]) \;\star \bigstar_{k\in[0,|L|-1]\setminus\{i\}} p \rightsquigarrow \mathsf{Cellof}\,R\,k\,(L[k])$$

where: $p \rightsquigarrow \mathsf{Cellof}\,R\,i\,V \;=\; \exists v.\ (p[i] \mapsto v) \;\star\; (v \rightsquigarrow R\,V)$.

Question 3.9. Assume a strict order relation on elements, written $x < y$. We write "$\mathsf{sorted}\,(<)\,L$" the assertion stating that a logical list L is sorted with respect to $(<)$. Consider the function `minsert` shown below, which inserts an element in a sorted mutable list. This element is assumed to not already be in the list.

```
let minsert (a:'a) (s:'a cell) : 'a cell =
  if s == null || a < s.hd then { hd = a; tl = s } else begin
    let r = ref s in
    while true do
      let p = !r in
      let q = p.tl in
      if q == null || a < q.hd then begin
        p.tl <- { hd = a; tl = q };
        break;
      end;
      r := q;
    done;
    s
  end
```

Assume the initial state to be described by $s \rightsquigarrow \mathsf{Mlist}\,L$. For the code above: (1) State an invariant that is true at the *beginning* of every iteration of the loop (but not necessarily after the *break*). (2) Describe the state just before the beginning of the loop, and explain how it entails the loop invariant. (3) Describe the state that you obtain just before the *break* instruction, and explain how it allows deriving $s \rightsquigarrow \mathsf{Mlist}\,L'$, for some list L' that is sorted and that stores the elements of L plus the element a. (4) Briefly justify why the invariant is preserved in case the loop steps to the next iteration (i.e. does not *break*), focusing only on the justification of the purely logical facts from the invariants. (5) Justify termination of the loop. (Expected answer: 25 lines. Make sure to be very precise about all the details in the formulae that you write. If you have an invariant but know that it is not strong enough, indicate it explicitly.)

Question 3.10. The representation predicate of type `hashset2` is defined as shown below:

$$p \rightsquigarrow \mathsf{HashSet2}\,E \;\equiv\; \exists L.\ p \rightsquigarrow \mathsf{Arrayof}\,\mathsf{Mlist}\,L \;\star\; [\mathsf{Repr1}\,L\,E].$$

Recall the specification of `remove1` from question 3.2. The function `remove2` admits the same specification, except with the representation predicate HashSet2 instead of HashSet1. Prove the correctness of `remove` with respect to this specification, by exploiting the specification already established for the function `remove2` in question 3.7. (Expected answer: 15 lines.)