Devoir surveillé #2

Vendredi 2025-12-12 ; durée : trois heures

Dans tout le sujet, vous pouvez (sauf mention du contraire) réutiliser au sein **d'un même exercice** une fonction demandée à une question précédente, même si la question n'a pas été traitée.
Sauf mention du contraire, vous **ne pouvez pas** utiliser de fonctions de la bibliothèque standard OCaml.

Petits exercices

Exercice 1. *Lecture/écriture de types OCaml*

Donnez sans justification les **types OCaml** des expressions suivantes (aucune phrase n'est attendue) :

1. `12`
2. `true, false`
3. `[None]`
4. `fun p -> let x, y = p in x, x, y`

Pour chacun des types suivants, écrivez une expression OCaml de ce type (aucune phrase n'est attendue) :

5. `bool`
6. `unit`
7. `int list`
8. `('a -> 'b -> 'c) -> ('a * 'b) -> 'c`

Exercice 2. `log2`

1. Écrivez une fonction OCaml `log2 : int -> int` telle que pour tout `(x:int)` strictement positif, `log2 x` s'évalue en la plus grande puissance de deux ℓ telle que $2^\ell \leq x$ (et a un comportement indéfini sinon).
   Par exemple, `log2 1` doit s'évaluer à `0`, `log2 2` et `log2 3` à `1`, et `log2 65536` à `16`.

Exercice 3. `compare_length`

On donne les spécifications suivantes pour la fonction OCaml `compare_length` de type :
`'a list -> 'b list -> int`
Pour toutes listes `x`, `y`, `compare_length x y` s'évalue en :

— `-1` si `x` contient strictement moins d'éléments que `y` (`x` est strictement plus courte que `y`)
— `0` si `x` contient le même nombre d'éléments que `y` (les deux listes ont la même longueur)
— `1` si `x` contient strictement plus d'éléments que `y` (`x` est strictement plus longue que `y`)

Par exemple :

— `compare_length [] [1]` s'évalue à `-1`
— `compare_length [bool] [1]` s'évalue à `0`
— `compare_length [1; 2] [1]` s'évalue à `1`

1. Écrivez une fonction OCaml `compare_length` qui est totalement correct pour les spécifications ci-dessus. (On ne demande pas de prouver que c'est bien le cas.) Votre fonction doit également être la plus efficace possible, mais aucune analyse de coût n'est demandée.
   Votre fonction peut être récursive, mais **elle ne doit faire appel à aucune autre fonction** (que celle-ci existe dans la bibliothèque standard ou que vous l'ayez définie vous-même).

Exercice 4. `prod` & `init` & `fact`

On donne les spécifications suivantes pour la fonction OCaml `prod` de type :
`int list -> int`
Pour tout `(x:int list)`, `prod x` s'évalue en 1 si `x` est la liste vide, et en le produit de ses éléments sinon.

1. Écrivez une fonction OCaml `prod` qui est totalement correcte pour les spécifications ci-dessus. (On ne demande pas de prouver que c'est bien le cas). Votre fonction doit de plus être « la plus simple possible ».

On donne les spécifications suivantes pour la fonction OCaml `rev_init` de type :
`int -> (int -> 'a) -> 'a list`
Pour tout `(x:int)` positif ou nul et toute fonction `(f:int -> 'a)`, `rev_init x f` s'évalue en une liste de longueur `x` (c'est à dire possédant `x` éléments) dont le `i`ème élément en partant de la queue de liste est égal à `f i`, où l'on commence la numérotation à `0`. Autrement dit, l'on a informellement que `rev_init x f` s'évalue à `[f (x - 1); f (x - 2); ... f 0]`.

2. Écrivez une fonction OCaml `rev_init` qui est totalement correcte pour les spécifications ci-dessus. Votre fonction peut être récursive (s'appeler elle-même) et appeler son argument `f`, mais **elle ne doit faire appel à aucune autre fonction** (que celle-ci existe dans la bibliothèque standard ou que vous l'ayez définie vous-même).
3. Écrivez une fonction OCaml `(fact : int -> int)` telle `fact x` s'évalue en $x!$ pour tout `(x:int)` positif tel que $x!$ est représentable par une valeur de type `int` (et a un comportement indéfini sinon). Cette fonction **ne devra pas** être récursive et **devra** appeler les fonctions des questions précédentes.

Exercice 5. `boxed_mem`

On définit le type `struct boxed_int` comme :
```
struct boxed_int
{
    int v;
};
```
et donne les spécifications suivantes pour une fonction `boxed_mem` de signature :
```
struct boxed_int *
boxed_mem(size_t n, struct boxed_int *a[n], struct boxed_int x)
```
Soit `a` un tableau de longueur `n`, de pointeurs vers le type `struct boxed_int` et `x` un objet de type `struct boxed_int`, alors `boxed_mem(n, a, x)` renvoie :

— une valeur de type `struct boxed_int *` égale à un élément de `a` et qui pointe vers un objet de type `struct boxed_int` dont le champ `v` est égal au champ `v` de `x`, si une telle valeur existe ;
— une valeur de pointeur nul (par ex. `NULL`) sinon.

1. Écrivez une fonction C `boxed_mem` totalement correcte pour les spécifications ci-dessus. (On ne demande pas de prouver que c'est bien le cas.)

Pour tester la fonction `boxed_mem`, on souhaite écrire une fonction `test_m` de signature :
`bool test_m(void)`
qui fait les choses suivantes :

— elle définit un tableau de longueur 10 dont le $i$ème élément contient un objet dont le type est `struct boxed_int *` pointant vers un objet de type `struct boxed_int` dont le champ `v` vaut `i` ;
— elle appelle deux fois `boxed_mem` sur ce tableau, une fois avec un argument `x` tel que `boxed_mem` doit renvoyer un élément du tableau, et une fois tel qu'elle doit renvoyer une valeur de pointeur nul ;
— elle renvoie une valeur booléenne égale à `true` si les deux tests se sont déroulés avec succès, et `false` sinon.

De plus, on souhaite que cette fonction utilise `malloc` pour l'allocation mémoire des objets de type `struct boxed_int` pointés par les éléments du tableau, et qu'elle traite correctement les cas d'erreurs possibles lors des allocations.
Enfin, cette fonction ne doit avoir aucune fuite mémoire, **y compris en cas d'éventuelle erreur lors d'une allocation mémoire**.

2. Écrivez une fonction C `test_m` qui satisfait les spécifications ci-dessus.

Problèmes

Exercice 6. *Correction de* `partition`

On considère la fonction C suivante :

```
1  void partition(size_t n, int a[n]) {
2      int p = a[0];
3      int pi = 0;
4      size_t i = 1;
5      for (; i < n; i++) {
6          if (a[i] < p) {
7              a[pi] = a[i];
8              a[i] = a[pi + 1];
9              a[pi + 1] = p;
10             pi = pi + 1;
11         }
12     }
13 }
```

Le but de cet exercice va être de prouver qu'elle est totalement correcte pour les spécifications suivantes : Soit `a` un tableau d'`int` de longueur `n` strictement positive, et soit `p` la valeur de `a[0]`, alors l'exécution de `partition(n, a)` ne cause pas d'accès hors-borne dans le tableau `a`, et à l'issue de celle-ci les éléments de `a` :

- (**a**) forment une permutation des éléments de `a` avant appel (*cf.* ci-dessous pour une définition un peu plus formelle de cette condition) ;
- (**b**) sont tels qu'étant donné un indice p_i où se trouve une valeur égale à `p` dans `a` :
  - les éléments d'indice $i \in [\![0, p_i - 1]\!]$ sont tous strictement inférieurs à `p` (pour l'ordre naturel sur les `int` ;
  - les éléments d'indice $i \in [\![p_i + 1, n - 1]\!]$ sont tous supérieurs ou égaux à `p` (pour l'ordre naturel sur les `int`).

On peut représenter cela schématiquement de la façon suivante :

- l'état de `a` avant appel à `partition` est de la forme :

  `p x x x x x x x x ...`

  où `x` dénote un élément de valeur inconnue (relativement) à `p`
- et après appel, il est de la forme :

  `s s s s s s p b b b ...`

  où `s` (resp. `b`) dénote un élément de valeur strictement inférieure (resp. supérieure ou égale) à `p`.

1. Prouvez succinctement la terminaison de `partition`.
2. Prouvez l'invariant de boucle $\mathcal{I}_2 : p_i \in [\![0, i-1]\!]$ et $i \in [\![p_i + 1, n]\!]$ (ou dit autrement : $0 \leq p_i < i \leq n$), où p_i et i désignent respectivement les valeurs des variables `pi` et `i` en début d'itération.
3. Prouvez que `partition` effectue uniquement des accès valides à `a` (qui ne sont pas « hors-borne »).
4. Prouvez l'invariant de boucle $\mathcal{I}_4 : p = a[p_i]$, où p, p_i, $a[p_i]$ désignent respectivement les valeurs des variables `p`, `pi`, et de l'élément du tableau `a` d'indice `pi` en début d'itération.

Pour deux tableaux x, y de même longueur, on définit (informellement) le prédicat $\pi(x, y)$ (« x est une permutation de y ») qui s'évalue à *vrai* ssi. x peut être obtenu par une suite d'échange de deux éléments de y ; ou de façon équivalente (admise) que x et y possèdent les mêmes éléments avec même multiplicité (si e est un élément de x et apparaît n_e fois dedans, alors il apparaît exactement n_e fois dans y). On pourra de plus admettre que π est commutatif ($\pi(x, y) = \pi(y, x)$) et transitif ($\pi(x, y) \wedge \pi(y, z) \Rightarrow \pi(x, z)$).

5. On note A la valeur du tableau `a` avant appel à `partition`, et a sa valeur en début d'une itération de la boucle de la ligne 5. Prouvez l'invariant de boucle $\mathcal{I}_5 : \pi(A, a)$.

On définit maintenant les deux ensembles suivants :

- $\mathcal{S} := \{a[j],\ 0 < j < p_i\}$
- $\mathcal{B} := \{a[j],\ p_i < j < i\}$

où i et p_i désignent respectivement les valeurs des variables `i` et `pi`, et $a[j]$ (pour j un paramètre libre) celle de l'élément de `a` d'indice j (s'il existe).

6. Prouvez l'invariant de boucle $\mathcal{I}_6$: les éléments de $\mathcal{S}$ sont tous strictement inférieurs à p, et ceux de $\mathcal{B}$ lui sont tous supérieurs ou égaux. (Vous pouvez éventuellement vous aider d'un dessin.)
7. Conclure. (On attend ici que vous identifiez clairement quelles questions ou invariants permettent de prouver chacun des aspects des spécifications de `partition`, mais vous pouvez vous contenter de justifications rapides).

N.B. L'algorithme étudié dans cet exercice est une version un peu naïve de celui se trouvant au cœur de l'algorithme du *tri rapide* (« *quicksort* »), dans sa version dégradée non probabiliste.

Exercice 7. *Zipper sur les listes*

Soit `(x : 'a list)` une liste OCaml de longueur n, on définit son *zipper à l'indice* i (avec $0 \leq i \leq n$) comme le couple ordonné de deux listes `z1` et `z2` telles que si l'on note `[x1; x2; ...; xn]` les n éléments de `x`, l'on a `z1` égal à `[xi; ...; x1]` et `z2` égal à `[x(i+1); ...; xn]`.
Autrement dit, `z1` contient les i premiers éléments de `x` stockés « en miroir », et `z2` contient les $n-i$ derniers éléments de `x`.
Par exemple, pour `x = [1; 2; 3; 4; 5; 6]` :

— son zipper à l'indice 0 vaut `[], [1; 2; 3; 4; 5; 6]`
— son zipper à l'indice 6 vaut `[6; 5; 4; 3; 2; 1], []`
— son zipper à l'indice 2 vaut `[2; 1], [3; 4; 5; 6]`

1. Écrivez une fonction OCaml :
   `list2zipper : 'a list -> 'a list * 'a list`
   telle que `list2zipper x` s'évalue au zipper de `x` à l'indice 0.
2. Écrivez une fonction OCaml :
   `move_right : 'a list * 'a list -> 'a list * 'a list`
   telle que pour `z` le zipper à l'indice i d'une certaine liste `x`, `move_right z` s'évalue en `z` si i est égal à n la longueur de x, et sinon s'évalue en le zipper de `x` à l'indice $i+1$.
3. Écrivez une fonction OCaml :
   `zipper2list : 'a list * 'a list -> 'a list`
   telle que pour `z` un zipper de `x` à un indice (valide) quelconque, `zipper2list z` s'évalue en `x`.
4. Écrivez une fonction OCaml :
   `insert_left : 'a list * 'a list -> 'a -> 'a list * 'a list`
   telle que pour `z` un zipper de `x` à un indice i valide, et `(v : 'a)` quelconque, `insert_left z v` s'évalue en un zipper à l'indice $i+1$ de la liste `[x1; ...; xi; v; ...; xn]`

On s'intéresse maintenant au problème suivant : soit `(x : int list)` une liste d'entiers positifs, distincts et triés par ordre croissant, et soit `m` le dernier élément de cette liste (qui est donc son élément maximum) on souhaite construire une nouvelle liste contenant tous les entiers positifs de 0 à `m`, une unique fois, dans l'ordre croissant. Par exemple, étant donnée `x = [1; 3]`, on souhaite construire la liste `[0; 1; 2; 3]`.
Une solution aisée à ce problème serait de parcourir toute la liste `x` pour trouver la valeur de `m`, puis de construire la liste voulue ; cependant, on cherche ici une solution spécifiquement à base de zipper (pourquoi pas ?).

5. Écrivez une fonction OCaml :
   `fillz : int list * int list -> int -> int list * int list`
   telle que pour `z` un zipper à l'indice v d'une liste `x` triée dont les v premiers éléments sont les entiers de 0 à $v-1$, `fillz z v` s'évalue en un zipper d'une liste triée égale à `x` où l'on a (si besoin) « inséré » tous les entiers inférieurs à la valeur maximum s'y trouvant.
   Cette fonction devra être la plus efficace possible (mais l'on ne demande pas d'en analyser le coût).
6. Donnez un exemple non trivial d'appel de `fillz` ainsi que du résultat produit.
7. Écrivez une fonction OCaml :
   `fill_list : int list -> int list`
   qui résout le problème ci-dessus.

Exercice 8. *Tri fusion*

On se propose dans cet exercice d'implémenter (une version de) l'algorithme du *tri fusion* en OCaml.

1. Écrivez une fonction OCaml :

   `merge : 'a list -> 'a list -> 'a list`

   telle que pour `(x1:'a list)` et `(x2:'a list)` triées par ordre croissant (pour l'opérateur < standard), `merge x1 x2` construit et s'évalue en une liste triée par ordre croissant contenant les même éléments que `x1` et `x2` avec somme de leur multiplicité (c'est à dire que pour toute valeur `(e:'a)`, soit $x_1(e)$ et $x_2(e)$ le nombre d'occurrences de `e` dans `x1` et `x2` respectivement, alors `e` apparaît $x_1(e)+x_2(e)$ fois dans la liste construite par `merge x1 x2`).
2. Donnez un exemple non trivial d'appel de `merge` ainsi que du résultat produit.
3. Écrivez une fonction OCaml :

   `split : 'a list -> int -> 'a list * 'a list`

   telle que `split x 0` s'évalue en la paire de listes `[], x`, et pour `n` strictement positif `split x n` s'évalue (sans erreur) en une paire de listes `x1, x2` telle que :

   — `x1 @ x2 = x` (où `@` désigne l'opérateur de concaténation entre listes ; autrement dit, `x1 @ x2` est formé des éléments de `x1` suivis de ceux de `x2`) ;
   — `x1` contient au plus `n` éléments, et n'en contient moins que si `x2` est la liste vide
4. Donnez un exemple non trivial d'appel de `split` ainsi que du résultat produit.
5. Écrivez une fonction OCaml récursive :

   `_ms : 'a list -> int -> 'a list`

   telle que pour `x` une liste de longueur `n` on a :

   — Cas de base : `_ms x 0 = x` ; `_ms x 1 = x`
   — Cas récursifs : `_ms x n` s'appelle récursivement deux fois, une sur chacune des listes `x1` et `x2` construites de façon à avoir `x1 @ x2 = x` et une différence des longueurs de `x1` et `x2` au plus égale à 1 (en valeur absolue), puis combine le résultat de sorte qu'il contienne les mêmes éléments (avec somme des multiplicités) que `x1` et `x2` et soit trié pour l'ordre croissant.

   Vous devez justifier brièvement (en quelques phrases) que votre fonction est correcte pour les spécifications ci-dessus.
6. Écrivez une fonction OCaml :

   `ms : 'a list -> 'a list`

   qui construit une liste contenant les mêmes éléments que son argument (avec multiplicité) qui soit triée pour l'ordre croissant. **Dans cette question uniquement**, vous pouvez si vous le souhaitez utiliser la fonction `List.length` qui s'évalue en la longueur de son argument de type `'a list`.
7. Donnez un exemple non trivial d'appel de `ms` ainsi que du résultat produit.

N.B. L'algorithme du tri fusion est l'un des exemples canoniques de la stratégie algorihmique dite « diviser pour régner » (en anglais : *divide and conquer*) : on résout le problème récursivement sur des petites instances, et l'on construit une solution complète en combinant ces solutions.