TP #13 — Toujours du OCaml varié

Exercice 1. *Un peu d'exceptions :* `List.hd`

1. Effectuez un appel à `List.hd` qui lève une exception.
2. Effectuez ce même appel au sein d'un filtrage `try with` qui rattrape l'exception et affiche un message quelconque (en utilisant la fonction `print_string`).
3. Écrivez une fonction OCaml :

   `head_opt : 'a list -> 'a option`

   qui utilise `List.hd` et rattrape son éventuelle exception afin que `head_opt x` s'évalue en `Some h` avec `h` la tête de `x` si celle-ci est non vide, et `None` sinon.
   (Cette fonction ne devra utiliser aucun filtrage `match with`, aucun test, et aucun appel à une autre fonction que `List.hd`.)
4. Testez.
5. Écrivez une fonction OCaml :

   `head : 'a list -> 'a`

   qui utilise `head_opt` et qui quand appliquée à `x` lève une exception `Failure` si `x` est vide, et sinon s'évalue en la tête de `x`.
   (Cette fonction ne devra utiliser aucun filtrage sur son argument, aucun test, et aucun appel à une autre fonction que `head_opt`.)
6. Testez.

Exercice 2. *Un peu d'exceptions :* `List.nth`

1. Écrivez une fonction OCaml :

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

   telle que `nth x n` s'évalue au *n*ième élément de `x` (en partant de 0) s'il existe, lève une exception `Invalid_argument` si `n` est strictement négatif, et une exception `Failure` si `x` ne possède pas de *n*ième élément.
2. Testez.

N.B. On peut constater le coût important de cette fonction, qui fait que l'on essaye en général de l'utiliser le moins possible. (Dans un exercice (TP, TD, DS, concours…), utiliser `nth` (ou toute fonction équivalente) mènera quasiment toujours à une solution sous-optimale.)

Exercice 3. *Un peu d'exceptions : trouver des trucs*

1. Définissez une exception `Found` d'un argument `int`
2. Écrivez une fonction OCaml :
   `_find : 'a -> ('a * int) list -> 'b`
   telle que pour `(k: 'a)`, `(x: ('a * int) list)`, `_find k x` lève une exception `Found` d'argument le premier `v` tel que `k, v` apparaît dans `x`, et `Not_found` si aucun tel `v` n'existe.
   N.B. On peut constater de par son type que l'évaluation de `_find` ne termine jamais.
3. Testez.
4. Écrivez une fonction OCaml :
   `find : 'a -> ('a * int) list -> int option`
   de mêmes spécifications (à la spécialisation du type près) que la fonction du même nom du dernier TD, qui utilise `_find` et un filtrage d'exception adapté.
5. Testez.
6. Écrivez une fonction OCaml :
   `_exists2 : (int -> bool) -> int list -> 'a`
   telle que pour `(p: int -> bool)`, `(x: int list)`, `_exists2 p x` lève une exception `Found` d'argument le premier `v` dans `x` tel que `p v` s'évalue à `true`, et `Not_found` s'il n'y en a pas.
7. Testez.
8. Écrivez une fonction :
   `exists2 : (int -> bool) -> int list -> int option`
   qui utilise `_exists2` et un filtrage d'exception adapté afin que `exists2 p x` s'évalue à `Some v` avec `v` le premier élément de `x` tel que `p v` s'évalue à `true` si un tel `v` existe, et `None` s'il n'y en a pas.
9. Testez.

N.B. Il serait pratique de disposer d'exceptions polymorphes, afin de généraliser l'approche ci-dessus à des fonctions `find` et `exists2` qui puissent respectivement opérer sur des `('a * 'b) list` ou des `'a list`. polymorphe comme au dernier TD/TP. Ceci est en fait possible (dans une certaine mesure), mais pas pour nous…

Exercice 4. *Faux tri rapide*

On propose dans cet exercice d'implémenter une variante dégradée de l'algorithme du *tri rapide* ("quicksort"), dû à Hoare.
Celui-ci se base sur l'approche suivante : pour trier une liste x :

— on choisit un *pivot* p parmi les éléments de x
— on *partitionne* les éléments **restants** de x en deux sous-listes $x_<$ et $x_\geq$ contenant respectivement les éléments de x strictement inférieurs (resp. supérieurs ou égaux) à p ; ces listes ne sont pas forcément triées

— on trie récursivement $x_<$ et $x_\geq$; le résultat s'obtient en les concaténant (sans oublier de rajouter p au bon endroit)

La description ci-dessus n'est pas complète car on n'a pas précisé comment choisir le pivot p. Le *vrai* algorithme quicksort choisit celui-ci aléatoirement parmi les éléments de x, ce qui permet de garantir un bon coût *espéré* (il existe également des variantes faisant un choix « intelligent » du pivot, mais elles ne sont pas forcément plus efficaces en pratique). Dans notre cas nous nous contenterons (dans un premier temps) de prendre la tête de x pour pivot (ceci ruine les bonnes garanties de l'algorithme mais a un impact pédagogique plus modéré). Le vrai quicksort est aussi typiquement implémenté *en place* sur des tableaux, plutôt qu'avec des listes…

1. Donnez une liste des algorithmes de tri que vous connaissez (vous devriez pouvoir en nommer au moins cinq).
2. Écrivez une fonction OCaml :
   ```
   partition : 'a list -> 'a list * 'a list * 'a list
               -> 'a list * 'a list
   ```
   telle que `partition x (xless, xmore) p` s'évalue en une paire de listes `xless', xmore'` où `xless'` (resp. `xmore'`) contient *dans un ordre quelconque* les éléments de `x` strictement inférieurs (resp. supérieurs ou égaux) à `p` ainsi que les éléments de `xless` (resp. `xmore`), en conservant les multiplicités.
3. Testez votre fonction.
4. Quel est le coût de votre fonction ? S'il n'est pas linéaire en la longueur de son premier argument, cela veut dire qu'elle est trop inefficace, et vous devez dans ce cas recommencer à la question 2.
5. Écrivez une fonction :
   ```
   fake_qs : 'a list -> 'a list
   ```
   qui utilise votre fonction `partition` pour trier son argument avec l'algorithme du faux tri rapide décrit ci-dessus.
6. Testez votre fonction.
7. Si votre version d'OCaml le permet, écrivez une fonction :
   ```
   qs : 'a list -> 'a list
   ```
   qui choisit ses pivots aléatoirement (uniformément parmi tous les éléments possibles de la liste à partitionner) à chaque étape (on pourra supposer ici que l'argument de `qs` est de longueur strictement inférieure à 2^{30}). Ceci peut se faire par exemple en utilisant les fonctions `self_init` et `int` du module Random.
8. Testez votre fonction.

Exercice 5. *Tri fusion*

1. Implémentez toutes les fonctions de l'exercice 8 du dernier DS. Testez.

Exercice 6. *Nombre d'inversions*

Soit une suite d'entiers $x = x_1, \ldots, x_n$, on définit le *nombre d'inversions* $\sigma(x)$ de x comme le nombre de couples (i, j) tels que $1 \leq i < j \leq n$ et $x_i > x_j$.

1. Déterminez la valeur maximum que peut prendre $\sigma(x)$ en fonction de n.

Dans tout ce qui suit, on représente la suite x par une `int list`.

2. Écrivez une fonction OCaml :
   `inv_nav : 'a list -> int`
   qui utilise une approche naïve et s'évalue en le nombre d'inversions de son argument.
3. Testez votre fonction.
4. Exprimez le coût de votre fonction `inv_nav` en fonction de n.

On cherche maintenant à écrire une fonction `inv_dc` plus efficace que `inv_nav`, qui utilise une approche « diviser pour régner ». On va pour cela modifier une fonction de tri fusion, pour qu'en plus de « trier son argument » elle calcule également son nombre d'inversions ; on verra que ceci peut se faire essentiellement sans surcoût, ce qui donnera un coût global inférieur à `inv_nav`.
L'observation principale est que si `x1` et `x2` sont triées, le nombre d'inversions de `x1 @ x2` est égal à la somme des nombres d'inversions de `x1` et `x2` (tous deux nuls puisque `x1` et `x2` sont triées), plus le nombre d'inversions entre éléments de `x1` et éléments de `x2`. Ce dernier peut alors se calculer en fusionnant `x1` et `x2` en la liste triée contenant les même éléments que `x1 @ x2` : quand un élément de `x1` est « ajouté » au résultat, il est impliqué dans autant d'inversions que le nombre d'éléments de `x2` déjà « présents » dans le résultat.

5. Écrivez une fonction OCaml
   `merge_inv : 'a list -> 'a list -> 'a list * int`
   telle que pour `x1`, `x2` triées, `merge_inv x1 x2` s'évalue en une liste triée contenant les même éléments que `x1` et `x2`, et un entier donnant le nombre d'inversions de `x1 @ x2`. Il est ici fortement conseillé d'utiliser une fonction auxiliaire intermédiaire.
6. Testez votre fonction.
7. Écrivez une fonction OCaml
   `inv_dc : 'a list -> int`
   qui calcule le nombre d'inversions de son argument en suivant l'approche décrite ci-dessus.
8. Testez votre fonction.