

Examen du mardi 5 janvier 2021 – Durée 2 heures

Cet énoncé a 2 pages. Tout document papier est autorisé. Les ordinateurs et les téléphones portables doivent être éteints et rangés, ainsi que tout autre moyen de communication.

Les fonctions demandées doivent être rédigées en fonctionnel pur : ni références, ni tableaux, ni boucles `for` ou `while`, ni champs mutables. Chaque question ci-dessous peut utiliser les fonctions prédéfinies et/ou les fonctions des questions précédentes. À titre indicatif, toutes les fonctions demandées peuvent s'écrire en moins de dix lignes.

Exercice 1 (Expressions, types, valeurs). Pour chacune des expressions suivantes, indiquez si OCaml l'accepte, et donnez dans ce cas le type et la valeur calculés par OCaml. Si une valeur fonctionnelle est présente dans le résultat, la noter `<fun>` sans détailler plus. Si OCaml signale une erreur, la décrire succinctement. On ne demande pas alors le message d'erreur exact, mais l'idée essentielle, par exemple "ceci est de type `string`, mais `int` était attendu ici". On rappelle que l'exception `Not_found` est prédéfinie.

- 1.1 `1.0 +. (0 * 5.1)`
- 1.2 `if true then if false then false else true else false`
- 1.3 `(fun x y -> x*(x+y)) 2 3`
- 1.4 `(fun x y -> x*(x+y)) 2`
- 1.5 `(fun x y -> x*(x+y)) 2 3 4`
- 1.6 `let x = 2*2 in let x = x*x in x`
- 1.7 `List.map (fun x -> x*x) [1; 2; 3]`
- 1.8 `List.map (fun x y -> x*y) [1; 2; 3]`
- 1.9 `if 1 < 2 then 42 else raise Not_found`
- 1.10 `let r = ref 42 in r := !r + !r; !r`

Exercice 2 (Listes de répétitions). On étudie ici une version simple de l'algorithme RLE, qui permet de compacter des données ayant beaucoup de répétitions. Ainsi, toute liste commence par un certain nombre de répétitions d'une première valeur, puis un certain nombre de répétitions d'une autre valeur, et ainsi de suite jusqu'à la fin de la liste. Le contenu d'une liste de la forme

$$[\underbrace{v_1; \dots; v_1}_{n_1 \text{ fois}}; \underbrace{v_2; \dots; v_2}_{n_2 \text{ fois}}; \dots; \underbrace{v_p; \dots; v_p}_{n_p \text{ fois}}]$$

peut donc être décrit précisément par la liste de couples $[(v_1, n_1); \dots; (v_p, n_p)]$ signifiant : "la liste contient n_1 fois la valeur v_1 , suivi de n_2 fois la valeur v_2 , ..., suivi de n_p fois la valeur v_p ". Cette description sera appelée une *liste de répétitions* si elle satisfait en outre les conditions suivantes : (a) chaque n_i est non nul ; (b) chaque v_{i+1} est différent de v_i . Dans les questions suivantes, on appellera *codage* d'une liste sa description sous forme de liste de répétitions. Par exemple, $[(\text{'a'}, 2); (\text{'b'}, 1); (\text{'c'}, 3); (\text{'a'}, 2)]$ est le codage de la liste $[\text{'a'}; \text{'a'}; \text{'b'}; \text{'c'}; \text{'c'}; \text{'c'}; \text{'a'}; \text{'a'}]$.

- 2.1 Écrire une fonction `decode : ('a * int) list -> 'a list` reconstruisant une liste à partir de son codage. Autrement dit, si `lr` encode la liste `l`, alors `(decode lr) = l`.
Question bonus : implémenter `decode` de façon récursive terminale.
- 2.2 Écrire une fonction `add : 'a -> int -> ('a * int) list -> ('a * int) list` de telle sorte que `(add v n lr)` ajoute `n` répétitions de la valeur `v` en tête de la liste de répétitions `lr`. Veillez en particulier à respecter les conditions (a) et (b) dans la liste produite.
- 2.3 En utilisant la fonction précédente, écrire une fonction `encode : 'a list -> ('a * int) list` qui construit, à partir d'une liste, son codage en tant que liste de répétitions.
- 2.4 Écrire une fonction `code_map : ('a -> 'b) -> ('a * int) list -> ('b * int) list` de sorte que `(code_map f lr)` encode la liste `(List.map f l)` lorsque `lr` encode la liste `l`. Cette fonction ne doit pas reconstruire `l`. Veillez en particulier à respecter la condition (b) dans la liste produite.

