## 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.

- $1.1 \quad 1.0 + (0 * 5.1)$
- 1.2 if true then if false then false else true else false
- 1.3 (fun x y  $\rightarrow$  x\*(x+y)) 2 3
- 1.4 (fun x y  $\rightarrow$  x\*(x+y)) 2
- 1.5 (fun x y  $\rightarrow$  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  $\rightarrow$  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

$$n_1$$
 fois  $n_2$  fois  $n_p$  fois  $v_1; \ldots; v_1; v_2; \ldots; v_2; \ldots; v_p; \ldots; v_p; \ldots; v_p$  ]

peut donc être décrit précisément par la liste de couples  $[(v_1,n_1);\ldots;(v_p,n_p)]$  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, [('a',2);('b',1);('c',3);('a',2)] est le codage de la liste ['a';'a';'b';'c';'c';'c';'a';'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 1, alors (decode lr) = 1.
  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.

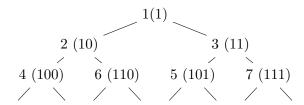
Exercice 3 (Arbres Patricia). Reprenons les arbres binaires vus en cours :

```
type 'a tree =
  | Node of 'a tree * 'a * 'a tree
  | Leaf
type pos = int (* positions via des entiers strictement positifs *)
```

Chacun des noeuds de ces arbres binaires vont être désignés ici par des positions qui seront des entiers strictement positifs. Tous les entiers manipulés ici seront supposés **strictement positifs**. La position spécifiée par un entier n dans un arbre est déterminée par l'écriture binaire de n. Pour tout arbre t:

- La racine de t est à la position 1.
- Si la position d'un noeud dans le sous-arbre gauche de t s'écrit en binaire  $b_k \dots b_0$ , la position de ce noeud dans t s'écrit  $b_k \dots b_0 0$  (i.e. le double).
- Si la position d'un noeud dans le sous-arbre droit de t s'écrit en binaire  $b_k \dots b_0$ , la position de ce noeud dans t s'écrit  $b_k \dots b_0 1$  (i.e. le double plus un).

Autrement dit, les positions dans un arbre seront paires à gauche et impaires à la racine ou à droite (on rappelle qu'en OCaml un entier n est pair si n mod 2 = 0). Et une division entière par 2 donne récursivement la position dans le sous-arbre correspondant. Voici un exemple d'arbre où les positions sont indiquées aux noeuds (avec l'écriture binaire entre parenthèses) :

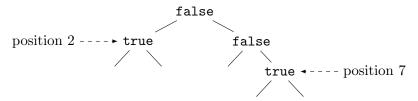


3.1 Écrire une fonction find : pos -> 'a tree -> 'a option renvoyant le contenu du nœud à une position d'un arbre. Plus précisément : si n est la position dans t d'un noeud contenant la valeur v alors find n t = Some v; et si t ne contient pas de nœud à la position n, alors find n t = None.

À l'aide de ces arbres et de ces positions, nous allons maintenant encoder des *arbres Patricia*, une structure de données permettant de représenter efficacement les ensembles finis d'entiers strictement positifs :

## type patricia = bool tree

Un arbre Patricia représente un ensemble dont les éléments sont toutes les positions des noeuds de l'arbre contenant la valeur booléenne true. L'ensemble  $\{2,7\}$  peut ainsi être représenté par l'arbre Patricia suivant :



- 3.2 Via la fonction find précédente, écrire une fonction mem : pos -> patricia -> bool testant l'appartenance d'un entier à un ensemble représenté via un arbre Patricia.
- 3.3 Écrire une fonction add : pos -> patricia -> patricia telle que si l'arbre Patricia p représente l'ensemble S, alors add n p représente l'ensemble  $S \cup \{n\}$ .
- 3.4 En utilisant la fonction précédente, écrire une fonction of\_list : pos list -> patricia construisant la représentation d'un ensemble d'entiers à partir de la liste de ces entiers.
- 3.5 Écrire une fonction union : patrica -> patricia -> patricia telle que union p1 p2 représente l'ensemble  $S_1 \cup S_2$  si p1 et p2 représentent respectivement les ensembles  $S_1$  et  $S_2$ .
- **3.6** Écrire une fonction to\_list : patricia -> pos list retournant, à partir de la représentation d'un ensemble, la liste de tous ses éléments (dans n'importe quel ordre, mais sans redondances).
- 3.7 En quoi ces arbres Patricia vous semblent-ils une représentation efficace d'ensembles de grande taille?