Université de Paris — Programmation Fonctionnelle
L3 Informatique — Année 2021-2022

Examen du mercredi 5 janvier 2022 – Durée 2 heures

Cet énoncé a 3 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".

1.1 `(1.0 +. 3.14, 1 + 3)`

1.2 `if true then if false then false`

1.3 `(fun x y -> x+x*y) 1 2`

1.4 `(fun x y -> x+x*y) 1`

1.5 `(fun x y -> x+x*y) 1 2 3`

1.6 `let f x = x*x in f 2 + 2`

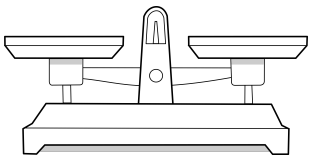
1.7 `List.map (fun x -> x+x) [1; 2; 3]`

1.8 `(fun x -> x) (fun y -> y) 5`

1.9 `(fun f -> f 1) (fun y -> y+3)`

1.10 `try List.assoc 1 [(2,"yes")] with Not_found -> "no"`

Exercice 2 (**Balance de Roberval).** Une *balance à deux plateaux* ou balance de Roberval indique si ses deux plateaux sont équilibrés et contiennent donc la même masse totale de chaque côté.



On considère une *boîte de poids*, c'est-à-dire un certain nombre de poids de référence. Tous les poids que nous utilisons ici sont des entiers. Une boîte de poids sera représentée ici par une `int list` donnant la liste des poids disponibles, dans l'ordre croissant, avec redondances possibles. Par exemple, la boîte `[2;2;5]` fournit deux exemplaires du poids 2 et un exemplaire du poids 5. Pour une pesée avec la balance, on utilise certains des poids disponibles dans la boîte de poids considérée.

Dans un premier temps, on considère seulement des pesées dites *simples*, avec l'objet à peser sur un plateau et certains poids pris dans la boîte de l'autre côté. Le poids que l'on mesure est donc l'addition des poids de référence utilisés à ce moment-là. Note : 0 est mesurable (en n'utilisant aucun poids).

2.1 Écrire `pesee_max : int list -> int` recevant une boîte de poids en entrée et donnant en sortie le poids maximal pesable à l'aide de cette boîte. Par exemple `pesee_max [2;2;5] = 9`.

Dans ce qui suit, une liste est dite *triée selon* `<` si elle est strictement croissante selon l'ordre `<` d'OCaml (et en particulier sans redondances).

2.2 Écrire la fonction auxiliaire `fusion : 'a list -> 'a list -> 'a list` prenant en entrée deux listes triées selon `<` et produisant l'union de ces deux listes, triée également selon `<`. Si ces listes d'entrée ont des éléments communs, n'en garder qu'un seul exemplaire.

Par exemple `fusion [1;3;5] [1;2;3] = [1;2;3;5]`.

2.3 En utilisant `fusion`, écrire `pesees_simples : int list -> int list` recevant une boîte de poids en entrée et donnant en sortie la liste de toutes les mesures de pesées simples réalisables avec cette boîte, triées selon `<`.

Par exemple `pesees_simples [] = [0]` et `pesees_simples [2;2;5] = [0;2;4;5;7;9]`.

On considère maintenant des pesées complexes, où on peut placer des poids sur chacun des deux plateaux, même du côté de l'objet que l'on pèse. On peut ainsi former des additions et des soustractions. Par exemple, la boîte `[1;3;9;27]` permet de mesurer $2 = 3 - 1$ ou $20 = 27 - 9 + 3 - 1$. On ignore les totaux négatifs. Les pesées simples sont en particulier des exemples de pesées complexes.

2.4 Écrire `pesees_complexes : int list -> int list` recevant une boîte de poids en entrée et donnant en sortie la liste triée selon `<` de toutes les sommes de pesées complexes réalisables avec cette boîte. Par exemple `pesees_complexes [2;3] = [0;1;2;3;5]`.

2.5 Une boîte `b` est dite *complète* si elle permet de mesurer tous les nombres entiers entre 0 et `pesee_max b` via des pesées complexes. Écrire `complete : int list -> bool` qui détermine si une boîte `b` est complète. Par exemple `complete [1;3] = true` mais `complete [2;3] = false` (car 4 n'est pas pesable avec cette boîte). Indice : considérer la longueur de `pesees_complexes b`.

Remarque mathématique : on peut démontrer que la boîte complète à n poids ayant la plus grande masse totale est `[1;3;9;...;`3^{n-1}`]`.

Exercice 3 (**Arbres-séquences).** Ce qu'on appelle *arbre-séquence* est une représentation des suites finies non vides $(x_0, \ldots, x_{n-1})$ d'éléments indexées par les entiers naturels. En terme de complexité, cette représentation va offrir un compromis intéressant entre les listes usuelles et les tableaux. En particulier, l'accès aux éléments des suites représentées se fera en temps logarithmique, de même que l'ajout ou la suppression d'éléments au début ou à la fin de la suite.

Le type OCaml que nous utiliserons pour représenter les arbres-séquences est celui des arbres binaires dont les feuilles (et seulement les feuilles) sont étiquetées par des valeurs :

```
type 'a tree =
 | Leaf of 'a
 | Node of 'a tree * 'a tree
```

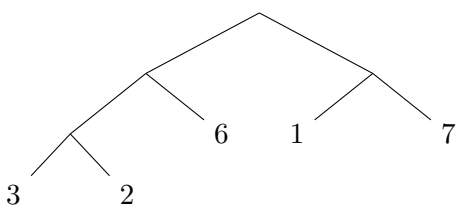
La représentation dans ce type d'une suite non vide $(x_0, \ldots, x_{n-1})$ de valeurs d'un même type, où n est le nombre d'éléments de la suite, se fait de la manière suivante :

- La représentation d'une suite réduite à un seul élément (x_0) est `Leaf` x_0.
- Si $n > 1$, la représentation de $(x_0, \ldots, x_{n-1})$ est `Node(pairs,impairs)` où :
  - `pairs` est la représentation de la sous-suite des éléments d'indices pairs $(x_0, x_2, \ldots)$,
  - `impairs` est la représentation de la sous-suite des éléments d'indices impairs $(x_1, x_3, \ldots)$,

Par exemple :

(1) `let exemple1 = Node (Node (Leaf 'a', Leaf 'c'), Leaf 'b')`
représente la suite de caractères `('a','b','c')`.

(2) La suite d'entiers $(3, 1, 6, 7, 2)$ est représentée par :
`let exemple2 = Node (Node (Node (Leaf 3, Leaf 2), Leaf 6), Node (Leaf 1, Leaf 7))`
autrement dit par l'arbre-séquence suivant :



Noter la représentation de la sous-suite $(3, 6, 2)$ côté pair, qui se décompose elle-même en les deux sous-suites $(3, 2)$ et (6).

Ce que nous appellerons *élément d'indice i* dans un arbre-séquence `t` est l'élément d'indice i dans la suite qu'il représente (ces indices commençant à 0). La *taille* de l'arbre est le nombre d'éléments de sa suite associée, *i.e.* le nombre de feuilles de l'arbre, calculable par la fonction suivante :

```
let rec size = function
| Leaf _ -> 1
| Node(pairs,impairs) -> size pairs + size impairs
```

Il est important de noter qu'un arbre-séquence de la forme `t = Node(pairs,impairs)` satisfait les propriétés suivantes (où `/` est ici la division entière d'OCaml) :

- `size pairs = (1 + size t)/2` et `size impairs = (size t)/2`
- `size pairs = size impairs` lorsque `size t` est pair,
- `size pairs = 1 + size impairs` lorsque `size t` est impair.

Les arbres reçus par vos fonctions ci-dessous pourront supposer ces propriétés, tandis que les arbres produits par vos fonctions devront les satisfaire. Sauf indication contraire explicite, toutes vos fonctions devront procéder directement sur les arbres, sans utiliser de listes ou autres structures intermédiaires.

3.1 Écrire `get : 'a tree -> int -> 'a` tel que `get t i` renvoie l'élément d'indice `i` dans `t`, ou lève l'exception de votre choix si `i` n'est pas un indice valide dans `t`.

Exemple : `get exemple1 2 = 'c'`.

3.2 Écrire `set : 'a tree -> int -> 'a -> 'a tree` tel que `set t i x` renvoie l'arbre-séquence obtenu en remplaçant dans `t` l'élément d'indice `i` par `x`, ou lève l'exception de votre choix si `i` n'est pas un indice valide dans `t`.

Exemple : `set exemple1 1 'z' = Node (Node (Leaf 'a', Leaf 'c'), Leaf 'z')`.

3.3 Écrire `tail : 'a tree -> 'a tree` tel que `tail t` renvoie la suite représentée par `t` privée de son premier élément, ou lève l'exception de votre choix si `t` est réduit à une feuille.

Exemple : `tail exemple1 = Node (Leaf 'b', Leaf 'c')`.

3.4 À l'aide de `get` et `tail`, écrire `to_list : 'a tree -> 'a list` renvoyant la liste formée des mêmes éléments que la suite représentée par `t`. Cette fonction doit évidemment manipuler des listes.

Exemple : `to_list exemple2 = [3; 1; 6; 7; 2]`.

3.5 Écrire `cons_left : 'a -> 'a tree -> 'a tree` tel que `cons_left x t` représente la suite formée de `x` suivi des éléments de la suite représentée par `t`.

Exemple : `cons_left 'a' (Node (Leaf 'b', Leaf 'c')) = exemple1` .

3.6 Écrire `cons_right : 'a tree -> 'a -> 'a tree` tel que `cons_right x t` représente la suite formée des éléments de la suite représentée par `t`, suivi de `x`. La fonction `size` pourra être utilisée librement ici, sans se soucier de sa complexité.

Exemple : `cons_right (Node (Leaf 'a', Leaf 'b')) 'c' = exemple1` .

3.7 Définir un nouveau type `'a sequence` réutilisant le type `'a tree`, et permettant :

- de représenter n'importe quelle suite finie, y compris la suite vide
- de stocker la taille des suites représentées (au moins celles non-vides), afin de pouvoir consulter cette taille en temps constant (i.e. sans recursivité).

Comment adapter la fonction `cons_right` de la question précédente pour en faire une version sur vos `'a sequence` qui fonctionne en temps logarithmique (en particulier sans utiliser la fonction `size` fournie) ?