Université Paris Diderot
M1 Info

Programmation Fonctionnelle Avancée
Année 2017-2018

Examen session 1

Mercredi, 10 janvier 2018

Tout document papier est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints et rangés. Le temps à disposition est de 3 heures.

Les exercices doivent être rédigés en fonctionnel pur : ni références, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements à champs mutables. Chaque fonction ci-dessous peut utiliser les fonctions prédéfinies (sauf indication contraire), et/ou les fonctions des questions précédentes.

Cet énoncé a 3 pages.

1 Compte à 100 et découpe de listes

Rappel :

```
List.concat [[1;2];[];[3]] = [1;2;3]
String.concat "," ["a";"b";"c"] = "a,b,c" .
```

Exercice 1 On cherche à résoudre en OCaml la devinette suivante :

> *Prendre les nombres de 1 à 9 dans cet ordre, et mettre entre chaque soit un + soit un * de telle manière que le résultat du calcul soit 100.*

On ne peut pas utiliser de parenthèses, et comme d'habitude une multiplication est plus prioritaire qu'une addition. Par exemple, une solution possible est `1*2*3+4+5+6+7+8*9`. On choisit ici de représenter ce genre d'expression par le type OCaml `int list list`, la liste extérieure représentant les additions et les listes intérieures représentant les multiplications. Ainsi l'expression solution précédente est codée `[[1;2;3];[4];[5];[6];[7];[8;9]]`.

1. Écrire une fonction `printexpr : int list list -> string` qui transforme le codage précédent en la chaîne de caractères associée (ici `"1*2*3+4+5+6+7+8*9"`).
2. Écrire une fonction `evalexpr : int list list -> int` qui calcule la valeur de l'expression (ici 100).

Un *découpage* d'une liste `l` est une liste de listes `ll` dont tous les éléments sont des listes non-vides, et telle que la concaténation de tous ces éléments `List.concat ll` redonne la liste de départ `l`. Par exemple `[[1;2];[3]]` et `[[1];[2];[3]]` sont deux découpages possibles de `[1;2;3]`, et il y en a deux autres.

3 Écrire une fonction `splits : 'a list -> 'a list list list` qui produit la liste de tous les découpages possibles de la liste donnée en entrée. Ces découpages peuvent être listés dans l'ordre de votre choix.

4 Utiliser cette fonction `splits` pour générer la liste de toutes les expressions que l'on peut obtenir en insérant des + et des * entre les nombres de 1 à 9, puis la liste des expressions donnant 100 comme résultat. On ne demande pas d'effectuer ce calcul !

2 Types

Exercice 2 Soit les deux définitions de type suivantes :

```
type t1 = [`A | `B ]
type t2 = [`A | `B | `C ]
```

Dire pour chacune des expressions suivantes si elle est bien typée ou pas. Si vous pensez que c'est le cas donnez son type, dans le cas contraire justifiez brièvement pourquoi l'expression n'est pas bien typée.

1. ```
   let f (x:t2) = match x with
     | `A -> `B
     | `B -> `C
     | `C -> `A
   ```
2. `let h x = [ f x; `D]`
3. `let ff x = f (f x)`
4. `let g (x:t1) = f x`
5. `let g (x:t1) = f(x:>t2)`
6. `let g (x: t1) = ((f x) :> t2)`

Exercice 3 Soit la définition de type suivante :

```
type 'a t =
  | MkB: bool -> bool t
  | MkI: int -> int t
```

Dire pour chacune des expressions suivantes si elle est bien typée ou pas. Si vous pensez que c'est le cas donnez son type, dans le cas contraire justifiez brièvement pourquoi l'expression n'est pas bien typée.

1. ```
   let extract = function
     | MkB b -> b
     | MkI i -> i
   ```
2. `let f i = MkB (MkI i = MkI i)`
3. `let f i = if i<0 then MkB true else MkI i`

3 Évaluation paresseuse

Exercice 4 Soit la définition suivante :

```
type digit = Zero | One
type real = cell Lazy.t
 and cell = Cons of digit * real
```

Le type `real` contient les flots infinies de symboles `Zero` et `One`. Un tel flot $(b_1, b_2, b_3, \ldots)$ représente une valeur réelle entre 0 et 1 qu'on peut calculer comme $b_1 * \frac{1}{2^1} + b_2 * \frac{1}{2^2} + b_3 * \frac{1}{2^3} + \ldots$. Par exemple, le flot $(1, 0, 1, 0, 0, 0, 0, \ldots)$ représente la valeur réelle $0.5 + 0.125 = 0.625$. Cette conversion des flots en des valeurs réelles nous sert seulement pour comprendre le sens des opérations sur les flots, à aucun moment on ne vous demande de programmer une conversion des flots vers des valeurs flottantes de OCaml.

1. Définir en OCaml le flot `zero` qui consiste en des symboles `Zero` seulement (ce flot représente la valeur réelle 0), le flot `one` qui consiste en des symboles `One` seulement (ce flot représente la valeur réelle 1), ainsi que le flot $(1, 0, 1, 0, 1, \ldots)$ qui alterne entre des symboles `One` et `Zero`.
2. Définir en OCaml une fonction `is_zero: int -> real -> bool` telle que `is_zero n f` teste si les n premiers symboles du flot f sont `Zero`. Écrire de même une fonction `is_one : int -> real -> bool`.
3. Définir en OCaml une fonction `max : real -> real -> real` telle que `max f1 f2` envoie le flot qui représente la valeur réelle qui est le maximum des valeurs réelles représentées par `f1` et `f2`. Remarquez qu'on n'a pas besoin d'une opération de comparaison de deux *flots* pour réaliser cette fonction.
4. Définir en OCaml une fonction `syntactic_equal : int -> real -> real -> bool` telle que `syntactic_equal n f1 f2` teste si les n premiers symboles de `f1` et `f2` sont les mêmes.
5. L'égalité syntaxique entre deux flots n'est pas suffisante pour décider l'égalité des valeurs représentées. Par exemple, les deux flots $(1, 0, 0, 0, 0, \ldots)$ et $(0, 1, 1, 1, 1, \ldots)$ sont syntaxiquement différents mais représentent tous les deux la même valeur réelle qui est $\frac{1}{2}$. En fait, quand un flot f_1 commence par `Zero` et un flot f_2 commence par `One`, ils sont syntaxiquement différents mais représentent la même valeur quand le reste de f_1 consiste en des `One` seulement et le reste de f_2 consiste en des `Zero` seulement.
   Écrire une fonction `real_equal : int -> real -> real -> bool` telle que `real_equal n f1 f2` teste si les valeurs représentées par les deux flots sont les mêmes. Le paramètre n vous sert à simplement couper la comparaison après n étapes : si on atteint la profondeur n dans les flots sans d'avoir trouvé de différence on considère les deux flots comme égaux ; il s'agit donc d'une égalité approximée.