NFP119 : feuille d'exercices 4

María-Virginia Aponte

26 mars 2010

Exercice 1

Fonctionnelles
Expliquez les réponses données par Ocaml.

```
# let double_du_succ(f,y) = 2*f(y+1);;
val double_du_succ : (int -> int) * int -> int = <fun>

# let f1 (x) = x-2;;
val f1 : int -> int = <fun>

#  double_du_succ(f1,3);;
- : int = 4

# let g1(z) = z*z;;
val g1 : int -> int = <fun>

#  double_du_succ(g1,3);;
- : int = 32

# let compare_res(f,g,x) = f(x+2) > g(x)+2;;
val compare_res : (int -> int) * (int -> int) * int -> bool = <fun>

# compare_res(f1,g1,2);;
- : bool = false
```

Exercice 2

Fonctionnelles
Écrivez une fonctionnelle récursive `sigma` qui prend une fonction f et un entier n et calcule :

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) \ldots + f(n)$$

En utilisant cette fonctionnelle, calculez la somme des n premier entiers, et la somme des carrés de n premiers entiers.

Exercice 3

Fonctionnelles sur les listes

Écrire la fonctionnelle `pour_tous` qui teste si tous les éléments d'une liste satisfont une condition. Pour cela, inspirez-vos du code de la fonctionnelle `existe` de la feuille d'ED précédente (et qui pour une condition *cond*, et une liste l passées en argument, teste s'il existe au moins un élément de la liste qui satisfait cette condition).

Testez ces deux fonctionnelles sur plusieurs exemples répresentatifs, puis comparez leur fonctionnement avec les fonctionnelles correspondantes du module `List` : `List.exists` et `List.for_all`.

Exercice 4

Fonctionnelles sur les listes
Dans cet exercice vous devez écrire des fonctions à l'aide des fonctionnelles vues en cours.

1. À l'aide de `List.filter` : écrire la fonction qui extrait d'une liste de dates toutes celles dont l'année est bissextile.
2. À l'aide de `List.map` : écrire la fonction qui extrait la liste de noms d'une liste d'employés, la fonction `remplace_tous` de l'exercice 4.

Tester toutes ces fonctions en TP sur plusieurs exemples.

Exercice 5

Fonctionnelles sur les listes

1. Écrire la fonction `est_trie` qui prend en argument une liste de n'importe quel type et teste si elle est triée dans l'ordre croissant.
2. Modifier cette fonction afin d'obtenir une fonctionnelle `est_trie_gen` qui prend en argument une liste de n'importe quel type et et une fonction qui correspond à un test d'ordre entre deux éléments de la liste. Cette fonction de test doit être de type `'a -> 'a -> bool`. La fonction `est_trie_gen` teste si la liste est triée selon l'ordre passé en argument.
3. A l'aide le fonction précédente, écrire les fonctions :
   – la fonction qui teste si une liste est triée dans l'odre décroissant,
   – la fonction qui teste si une liste des dates est triée dans l'odre croissant.
   – la fonction qui teste si une liste de vols est triée par l'ordre croissant des prix et décroissant des disponibilités.

Tester toutes ces fonctions en TP sur plusieurs exemples.

Exercice 6

Fonctionnelles sur les listes

1. Écrire la fonction `merge` qui prend deux liste supposées triés dans l'ordre croissant et calcule une nouvelle liste triée composée des éléments des deux arguments. Tester en TP sur plusieurs exemples.
2. Généraliser cette fonction pour un ordre quelconque passé en argument. Tester en TP sur plusieurs exemples.

Exercice 7

Exercice d'un ancien sujet

Partie I

Un *contexte de typage* décrit les noms des types pour les variables d'un programme. Il s'agit d'une liste de paires (x, Ty) où x est un nom de variable et Ty est le nom de son type. Les noms des variables sont représentés par des `string` et leurs types par le type `nomType` donné plus bas, qui répresente 2 sortes de types : les entiers et les booléans. La variable `c` est un exemple de contexte de typage pour deux variables, x et a.

```
type nomType = Int | Bool;;
let c = [("x", Int); ("a", Bool)];;
typeOfVar "a" c;;                          (* doit renvoyer => Bool *)
```

1. Ecrivez la fonction `typeOfVar` qui prend un contexte de typage et un nom de variable et qui retourne son type, ou qui échoue si la variable n'est pas dans le contexte.
2. Donner une nouvelle version **non récursive** de `typeOfVar`, utilisant cette fois la fonctionnelle `List.assoc`.

Partie II

A l'aide du type `expr` donné plus bas, nous définissons un petit langage d'expressions, permettant de représenter les expressions constantes entierès, true, false, l'addition et le `if`. **Exemples** : la variable `e` correspond à l'expression `1 + 2`; alors que `i` correspond à `if true then 1 else false` (**qui est une expression mal typée**).

```
type expr =   Num of int                        (* Nombres   *)
            | CBool of bool                     (* Boooleans *)
            | Sum of expr * expr                (* Addition  *)
            | If of expr * expr * expr;;        (* If        *)

let e = Sum ((Num 1), (Num 2));;            (* 1 + 2                        *)
let i = If ( (CBool true),                  (* if true then 1 else false *)
             (Num 1),
             (CBool false));;

typeOfExpr(e)                           (* doit renvoyer => Int   *)
typeOfExpr(i)                           (* doit renvoyer => Bool  *)
```

Ces expressions sont de type entier ou booléean. Nous voulons écrire une fonction `typeOfExpr` permettant de déterminer le nom du type (autrement dit, le `nameType` de la partie I) d'une expression ou de lever une exception si l'expression est mal typée.
Exemples : le type de `Sum( (Num 1), (Num 2))` sera `Int`, celui de `CBool(true)` sera `Bool`, alors que `Sum( (Num 1), (CBool true))` est mal typée et doit ainsi lever une exception.

Ecrivez la fonction `typeOfExpr` qui prend une `expr` en argument et renvoie son `nomType` ou qui échoue si elle est mal typée.

Partie III

Nous allons ajouter les noms de variables aux expressions `expr`, ce qui nous permettra de représenter des expressions telles que `1 + x`. Pour typer ce genre d'expression on doit connaître le type de ces variables. On va donc utiliser les contextes de typage de la partie I. Voici la nouvelle définition du type `expr`.

```
type expr =   Num of int                        (* Nombres   *)
            | CBool of bool                     (* Boooleans *)
            | Var of string                     (* Variables *)
            | Sum of expr * expr                (* Addition  *)
            | If of expr * expr * expr;;        (* If        *)

let e = Sum ((Num 1), (Var "x"));;          (* 1 + x                  *)
let i = If ( (Var "b"),                     (* if b then 1+x else y *)
             (Sum ((Num 1), (Var "x"))),
             (Var "y"));;

typeOfExpr c e                          (* doit renvoyer =>  Int       *)
typeOfExpr [("x",Bool)]  e              (* doit lever une exception    *)
varsInExpr i                            (* => renvoit ["b";"x";"y"]    *)
listUnbound [("x",Int)] e               (* => renvoit []               *)
listUnbound [("y",Int)] i               (* => renvoit [("x";"b"]       *)
```

Vous devez réécrire la fonction `typeOfExpr` de manière à prendre en argument supplémentaire un contexte de typage.

1. Ecrivez la nouvelle fonction `typeOfExpr:(string * nameType) list -> expr -> nameType`.

2. Ecrivez la fonction `varsInExpr` qui prend en argument une expression et qui renvoie en résultat la liste de tous les noms de variables qui apparaissent dans l'expression. Par exemple, pour l'expression `i` plus haut, elle doit retourner la liste `["b"; "x"; "y"]`.
3. On voudrait écrire une fonction qui determine si une expression contient des variables non définies (absentes) dans son contexte de typage. Par exemple, `e` ne peut pas être typée dans le contexte `[(Var "w",Int)]` car il ne contient pas la variable "x". Ecrire la fonction `unboundVarsInExpr` qui prend un contexte de typage c, une expression e, et donne la liste de toutes les variables de e qui ne sont pas définies dans c. Vous **devez utiliser** la fonction `varsInExpr` de la question précédente. On peut écrire une solution **non récursive** avec les fonctionnelles List.filter, List.map et List.mem. Attention : la solution **sans** ces fonctionnelles est récursive et beaucoup plus longue.

Exercice 8

Deux exercices d'un ancien sujet

Partie I

On souhaite modéliser le découpage en sections, sous-sections et sous-sous-sections de la table de matières d'un un document. On utilise pour cela une structure d'arbre, où chaque noeud correspond à une section, identifiée par son titre (une chaîne), et auquel on associe la liste de ses sous-sections, elles mêmes pouvant avoir des sous-sections. Le type `section` permet de représenter ceci, où le constructeur `FinalSect` correspond à une section terminale dans sa hiérarchie. Une table de matières est donc une liste de sections.

```
type section = FinalSect of string
             | Section of string * section list;;

type tableMat = Table of section list;;

let d = Table  [Section ("Amon-Re",
                      [Section("Baa",
                            [FinalSect "Canope"; FinalSect "Delphes"]);
                       FinalSect "Euphrates"]);
                FinalSect "Fin"];;
```

1. Ecrire une fonction `afficheTable` capable d'afficher la hiérarchie des sections et sousections d'une table de matières. Ainsi, la table `d` de l'exemple, devra être affiché sous la forme :

```
# afficheDoc d;;
1 section Amon-Re
  1.1 subsection Baa
    1.1.1 subsection Canope
    1.1.2 subsection Delphes
  1.2 subsection Euphrates
2 section Fin
- : unit = ()
```

2. Ecrire une fonction `numerotationDe` qui prend un titre de section et une table de matières, et renvoie la numérotation sous forme de chaîne de caractères, correspondant à ce titre dans la table. Exemple : `numerotationDe "Canope" d` renvoie la chaîne `"1.1.1"`.

Partie II

On veut modéliser, à l'aide des listes, l'indexation d'un ensemble de documents selon leurs mot-clés. Dans cet index, chaque document est représenté par une paire $(t, lMots)$, où t est son titre, et $lMots$ est sa liste de mots-clés.

```
type index =  (string * string list) list;;

let  i = [("Amon-Re",["Egypte"; "dieux"]);
```

```
("Voyager en Egypte",["Egypte"; "voyages"]);
("Typeful Programming", ["typage"; "programmation";"informatique"]);
("Java pour les nuls", ["programmation";"informatique";"Java"])];;
```

Afin d'exploiter ces index vous devez écrire les fonctions de recherche suivantes :

1. la fonction `chercheUnMot` qui prend un index i et un mot-clé m, et qui renvoie la liste de tous les titres (et uniquement les titres) de documents dont les mots clés contiennent m. La fonction `List.mem` peut-être utile ici.
2. la fonction `chercheTousLesMots` qui prend un index i et une liste de mots clés m, et qui renvoie la liste de tous les titres (et uniquement les titres) de documents qui contiennent tous les mots de m. Les fonctions `List.for_all` et `List.mem` peuvent être utiles ici.
3. la fonction `pertinenceDe` qui prend une liste de mots l et une paire (t, lm) composé d'un titre et de sa liste de mots clés. La fonction retourne le nombre d'éléments de l qui sont dans lm, ce qui donne une mesure de la pertinence du document t par rapport à la liste de mots-clés l. Par exemple, `pertinenceDe ["Egypte"; "dieux"]` appelée sur `("Amon-Re",["Egypte"; "dieux"])` renvoie 2, et sur `("Voyager en Egypte",["Egypte"; "voyages"])` renvoie 1. `List.mem` est utile ici.
4. Ecrivez la fonction `chercheParPertinence` qui prend une liste de mots-clés lm, un index i et renvoie le document le plus pertinent de l'index, à savoir, celui dont la liste de mots clés contient le plus de mots recherchés. Pour cela, on peut commencer par calculer la liste de toutes les paires (t, p) de titres t avec leur pertinence p pour cette liste de mots-clés (on utilisera List.map). Ensuite, il suffit de choisir le titre avec la plus forte pertinence dans cette liste.