UJF **POLYTECH**
RICM1 **Année 2006-07**

Langages et Programmation 2

TP2 - Listes et applications

Objectifs

Dans la première partie, on manipule les listes en OCAML et on construit des itérateurs de liste fonctionnels et procéduraux. Ce sera aussi l'occasion de faire le point sur le type `unit`.

La deuxième et troisième partie portent sur l'encodage des polynômes et des graphes à l'aide des listes. Pour les polynômes vous écrirez les fonctions nécessaires à leur manipulation, pour les graphes vous donnerez des fonctions permettant leur parcours.

Dans ces deux parties, on veillera à utiliser autant que possible les itérateurs programmés précédemment.

Il est **fortement conseillé** de tester vos fonctions avec une ou plusieurs entrées quelconques. Toutes les constructions syntaxiques qui vous seront nécessaires sont dans le cours ou dans la documentation en ligne d'OCaml.

1 Manipulations de listes

Nous avions défini les listes d'entiers dans le TP1. On utilisera à présent les listes "natives" d'OCAML qui peuvent contenir des éléments de n'importe quel type : listes de booléens, de fonctions, etc. Le type liste est donc paramétré par le type de ses élements qui peut être quelconque. En OCAML la déclaration de tels types s'écrit : `type ('a, 'b, ...) mytype = ... ('a, 'b, ...) mytype ...`, où `'a`, `'b` sont les types polymorphes paramètres du type `mytype`. Ainsi est déclaré le type des listes d'OCAML :

```
type 'a list = Vide | Liste of 'a * 'a list
```

Pour plus de commodité OCAML fournit la construction syntaxique `[]` comme alias du constructeur `Vide`, et la construction `e::l` comme alias du constructeur `Liste (e, l)`. Il est également possible de déclarer une liste comme suit `let l = [3;12;6]`.

Exercice 1 Une matrice peut être vue comme une liste de listes. Donnez le type `Matrix`.

Le module `List` regroupe un ensemble de fonctions élémentaires sur les listes. Pour les exercices qui suivent utilisez les fonctions de ce module (sauf évidemment celles que l'on vous demande d'écrire !) qui est documenté sur le site d'OCAML, section "ressources", puis "standard library documentation". Pour accéder aux fonctions d'un module, vous pouvez soit l'inclure, `open List`, soit qualifier le nom des fonctions, par exemple `List.length`.

Itérateurs fonctionnels

On testera les itérateurs suivants avec la fonction $g : x \mapsto x + 2$.

Exercice 2 Définir la fonction `map` de type `('a -> 'b) -> 'a list -> 'b list` : l'appel `map f [a1;...;an]` construit la liste `[f a1;...;f an]`. Comparer avec la fonction `List.map`. Dans les deux cas, on regardera l'ordre d'évaluation des éléments de la liste en modifiant la fonction g de façon à afficher une trace d'exécution (par exemple en utilisant `print_int`).

Exercice 3 Écrire la fonction `select` de type `('a -> bool) -> 'a list -> 'a list` : l'appel `select t [a1;...;an]` construit la liste `[...;ai;...]` des éléments vérifiant `t ai`. Comparer avec la fonction `List.filter`.

Exercice 4 Utiliser les fonctions précédentes pour contruire la sous-liste des éléments pairs d'une liste d'entiers, pour contruire la sous-liste des chaînes de caractères de taille supérieure ou égale à 3 dans une liste de chaînes de caractères.

Itérateurs procéduraux

Exercice 5 *(Bien utile pour la suite)* Écrire les lignes suivantes :

```
let x = 2+2 ; print_int 42 ;;
```

Que remarque-t-on à l'évaluation ? La variable `x` est-elle définie ? Écrire un code équivalent pour la définition de `x` en utilisant la construction `let ... in`.

Exercice 6 Écrire la fonction `dolist` de type `('a -> 'b) -> 'a list -> unit`. L'appel `dolist f [a1;...;an]` exécute l'équivalent de `f a1; f a2; ... ; ()`. Comparer le type de votre fonction avec celui de la fonction prédéfinie `List.iter`. Écrire une fonction `do_list_bis` dont le type est le même que la fonction `List.iter` et qui réalise la même chose. On pourra s'inspirer de l'exercice précédent.

Exercice 7 Écrire la fonction `reduce` de type `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` : l'appel `reduce f [a1;...;an] b` calcule `f a1 (f a2 (... (f an b) ...))`. Comparer avec la fonction `List.fold_right`. Votre fonction est-elle récursive terminale ?

Exercice 8 Utiliser les fonctions précédentes pour calculer la somme des éléments d'une liste d'entiers.

2 Manipulation de polynômes

Un polynôme $P(X) = a_nX^n + a_{n-1}X^{n-1} + \ldots + a_2X^2 + a_1X + a_0$ peut être vu comme la liste $[a_0, a_1, \ldots a_n]$ de ses coefficients.

Exercice 9 Après avoir défini le type des polynômes à coefficients entiers, définissez la fonction `monome` telle que `monome a n` construise le polynôme aX^n.

Exercice 10 Donnez les fonctions `sum` (somme de deux polynômes), `product_cst` (produit par une constante entière) et `product` (produit de deux polynômes).

Exercice 11 Définir une fonction `equal` qui teste l'égalité de deux polynômes : attention les polynômes X et $0X^2 + X + 0$ sont égaux.

3 Manipulation de graphes

On décide de coder un graphe sous la forme d'une liste d'adjacence : pour chaque sommet que l'on identifie par un entier, on associe la liste de ses successeurs.

Exercice 12 Définir le type `graph` et coder quelques exemples de graphes.

Exercice 13 Écrire une fonction `successors` qui prend un graphe et un sommet et rend la liste des successeurs du sommet.

Exercice 14 *(plus difficile)* Écrire une fonction `parcours` qui prend un graphe et une *fonction de traitement* et qui effectue cette fonction de traitement sur chacun des nœuds du graphe. Pour le parcours, on pourra utiliser une fonction auxiliaire `parcours_bis visited tobevisited` qui parcourt la liste `visited`, qui réalise l'effet de la fonction de traitement si le nœud courant n'a pas été visité, et relance récursivement la fonction `parcours_bis` sur les listes modifiées.

4 Note sur le test d'égalité

Il y a deux tests d'égalité en OCaml, le "`=`" et le "`==`". Le premier teste l'égalité structurelle, alors que le second teste l'égalité physique, c'est-à-dire l'emplacement en mémoire.

Exercice 15 Définir `type arbre = Feuille | Noeud of arbre * arbre`

Évaluer

```
let a = Feuille in
let b = Feuille in
(a=b),(a==b);;

let a = Feuille in
let b = a in
(a=b),(a==b);;

let a = Noeud(Feuille,Feuille) in
let b = Noeud(Feuille,Feuille) in
(a=b),(a==b);;

let a = Noeud(Feuille,Feuille) in
let b = a in
(a=b),(a==b);;

("aa"="aa"),("aa"=="aa");;
```

Exercice 16 Que peut-on en déduire ? Comment aurait-on pu écrire la fonction `equal` du TP précédent ?

Évaluation

Fichier commenté à envoyer à `laure.gonnord@imag.fr` avant le lundi 19 février minuit.