Caml-TP1 : Révisions, typage, tris,...

Paul Melotti*

19 septembre 2013

1 Que diable allez-vous faire dans cette mémoire ?

Pour une chaîne de caractères `u` on note $\tilde{u}$ sa chaîne miroir (par exemple si `u` = "`abcd`", $\tilde{u}$ = "`dcba`").

a) Écrire la fonction qui à `u` associe $\tilde{u}$.

b) Écrire la fonction qui teste si un mot est un palindrome, i.e. si `u` = $\tilde{u}$.

c) Que penser de la solution suivante ?

```
let echange u i j = let m=u.[i] in u.[i]<-u.[j] ; u.[j]<-m ;;
let miroir u = let n=string_length u in
    for i=0 to n/2-1 do
        echange u i (n-1-i)
    done ;
    u ;;
let palindrome u = let v = miroir u in u=v ;;
```

d) Et si l'on remplace `miroir` par celle-ci ?

```
let miroir u = let n=string_length u in
    let v=u in
    for i=0 to n/2-1 do
        echange v i (n-1-i)
    done ;
    v ;;
```

2 Récursivité

On considère la fonction suivante :

```
let rec f n = match n with
    | 0 -> 2
    | _ -> (f (n-1)) * (f (n-1)) ;;
```

Que calcule-t-elle ? Quelle est sa complexité ? Proposer une amélioration.

3 Matching

Henriette a l'idée saugrenue d'écrire une fonction qui teste si deux éléments sont égaux. Voici sa solution.

```
let egal x y = match y with
    | x -> true
    | _ -> false ;;
```

Qu'en pensez-vous ? Quel est le type de cette fonction ? Que renvoie `egal 1 2` ?

*Vous pouvez bien entendu me contacter par mail à `paul.melotti@ens.fr`, même en cas de début de question : N'HÉSITEZ PAS.

4 Manipulation de listes

Écrire les fonctions :

a) `card : 'a list -> int`
qui renvoie le cardinal d'une liste.

b) `estpresent : 'a -> 'a list -> bool`
qui dit si un élément est présent dans une liste.

c) `miroir : 'a list -> 'a list`
qui retourne une liste. Essayez d'obtenir une bonne complexité.

d) `applique : ('a -> 'b) -> 'a list -> 'b list`
qui étant donné une fonction `f` et une liste $[a_1;\ldots;a_n]$ doit renvoyer $[fa_1;\ldots;fa_n]$.
Cette fonction existe en Caml, il s'agit de `map`.

e) `itere : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
qui étant donné une fonction `f`, un élément a_0 et une liste $[b_1;\ldots;b_n]$ doit renvoyer $f\,(\ldots(f\,(f\,a_0\,b_1)\,b_2)\ldots)\,b_n$.
Cette fonction existe en Caml, il s'agit de `it_list`.

f) Utiliser `it_list` pour écrire une fonction qui calcule la somme d'une liste d'entiers.

g) (*) Utiliser `map` et/ou `it_list` pour écrire une fonction qui calcule le produit cartésien de deux listes.

5 Tris

a) Le tri par insertion consiste à insérer les éléments un à un dans une liste triée initialement vide, comme un joueur de cartes qui reçoit ses cartes une à une.
Implémenter le tri par insertion pour des listes. On pourra commencer par écrire une fonction `insere : 'a -> 'a list -> 'a list` qui insère un élément `x` au bon endroit dans une listé triée `l`.

b) Écrire un tri fusion pour des listes.
Rappel : on pourra écrire une fonction `partition : 'a list -> 'a list * 'a list` qui partitionne une liste en deux listes de même taille (ou presque), puis une fonction `fusion : 'a list * 'a list -> 'a list` qui fusionne deux listes triées en une grande liste triée.

c) Quel est à votre avis le meilleur de ces deux tris ?

d) Comment pourrait-on réaliser un tri en temps linéaire, ne fonctionnant pas par comparaison des éléments (mais éventuellement très coûteux en mémoire) ?
Indication : comment faire si les éléments à trier ne prennent qu'un nombre fini de valeurs ?

6 Exceptions

Les exceptions sont des instructions qui demandent l'arrêt du programme. Vous en avez déjà rencontré : ce sont par exemple les erreurs du style "Invalid_argument : vect_item " que vous obtenez lorsque vous essayez d'accéder à une case inexistante d'un tableau. Le programme s'est arrêté en cours de route pour vous lancer ce signal.

Une exception doit toujours être déclarée au préalable par la commande `exception <nom>`. Une exception peut aussi être rattachée à un type (par exemple, un entier : on renvoie l'indice du tableau pour lequel on a eu une erreur,...), dans ce cas, on le précisera dans la déclaration : `exception <nom> of <type>`.

Pour pouvoir gérer les exceptions, nous devrons commencer le programme par la commande `try`. À l'intérieur du programme, on déclenchera l'exception par la commande `raise <nom>`, ou `raise <nom> <donnée>` si l'exception a un type. L'exception sera alors récupérée par un matching final qui indiquera ce qu'il faut faire.

La syntaxe est donc :

```
try <expr> with
  <exception1> -> <faire ceci>
  | <exception2> -> <faire cela>
  | ...
```

Caml va évaluer `<expr>`. Si aucune exception n'est déclenchée, on continue normalement en ignorant le matching sur les exceptions. Si une exception est déclenchée, Caml la recherche dans le matching pour savoir ce qu'il doit faire.

Exemple : on cherche dans un vecteur `v` un élément `x`. La fonction `cherche` doit renvoyer l'indice d'un élément de `v` égal à `x`, ou `-1` si on n'en trouve pas.

```
exception trouve of int;;
let cherche x v = try
  for i=0 to (vect_length v)-1 do
  if v.(i)=x then raise (trouve i)
  done;
  -1
with trouvé k -> k;;
```

Exercice : écrire une fonction qui teste si une liste d'entiers est croissante, d'abord en style itératif (i.e. avec des boucles), puis récursif (i.e. avec des fonctions récursives). Le faire sans exceptions, puis avec.

7 Entiers de Church

Dans certains domaines de la logique et de l'informatique théorique, on raisonne essentiellement avec des fonctions. On peut avoir besoin de construire, dans cette théorie, les entiers à partir des fonctions. Une façon de le faire est d'utiliser les entiers de Church : on assimile l'entier `n` à la fonction qui à `f` associe son itérée `n` fois, f^n. [1]

a) Quel serait en Caml le type d'un entier de Church ?

b) Écrire des fonctions `church_of_int` et `int_of_church` qui effectuent les conversions entre entiers usuels et entiers de Church.

c) (*) Écrire les fonctions d'addition, multiplication et exponentiation pour les entiers de Church (sans repasser par les entiers usuels !).

1. Pour plus d'informations, chercher du côté du lambda-calcul, ou me demander.