TP 1 - Itérateurs, programmation dynamique

Jean-Baptiste Rouquier

27 septembre 2005

1 Préliminaires

En vrac :
- http://perso.ens-lyon.fr/jean-baptiste.rouquier
- Si vous ne connaissez pas Caml, commencez tout de suite par OCaml (que vous pourrez gardez après la prépa), et n'hésitez pas à me solliciter pour les petites différences avec Caml.
- Le manuel de Caml est consultable sur la page web officielle, http://caml.inria.fr. Les sections les plus utiles seront les deux premières («The core library» et «The standard library») dans la quatrième partie «The {Objective Caml, Caml Light} library». Je vous invite en particulier à consulter les fonctions sur les tableaux et listes ainsi que les modules printf, queue, random et stack pour éviter de réinventer la roue.
- Si vous souhaitez que je relise le code (commenté) que vous avez écrit pendant ce TP, appelez-le ⟨votre nom⟩_tp⟨numéro du tp, à deux chiffres⟩.ml et envoyez-le moi par mail le soir même (`prenom.nom@ens-lyon.fr`). Soignez le style et respectez le point suivant.
- Testez systématiquement vos fonctions, et laissez le code des tests dans le fichier.
- Mon job est (entre autres) de répondre à vos questions.
- Il vaut mieux implémenter la fonction demandée avec une mauvaise complexité que ne pas l'implémenter du tout. C'est valable en concours.

2 Mise en jambes : un peu de fonctionnel bien abstrait

La fonction

```
let plus x y z = x+y+z;;
```

est dite *curryfiée* car elle prend tout ce dont elle a besoin sous la forme de plusieurs arguments. À l'opposé, la fonction

```
let plus (x,y,z) = x+y+z;;
```

ne l'est pas car elle prend ce dont elle a besoin sous la forme d'un unique tuple.

Question 2.1. Écrire les fonctions `curryfie` et `decurrifie`.

Question 2.2. Écrire une fonction `compose` telle que `compose f g` renvoie la fonction $f \circ g$. Tester en composant trois fonctions (sous la forme $f \circ g \circ h$), avec le minimum de parenthèses.

Bonus : observer la syntaxe suivante :

```
let divise x y =
  y mod x = 0;;
#infix "divise";;
if 5 divise 10 then "ouf" else "bug";;
```

et définir l'opérateur `o` tel que `(f o g) x` soit bien $f \circ g(x)$.

Question 2.3. Écrire une fonction `iter : ('a -> unit) -> 'a list -> unit` qui prend une fonction `f` et l'applique à tous les éléments d'une liste. Noter que `f` renvoie `()` : on ne veut pas construire la liste des résultats de `f` (qui serait une liste de `unit`) mais simplement renvoyer `()`. En d'autre termes, cette fonction n'est pas `map`.

S'en servir pour définir la fonction `print_int_list`.

3 Encore du fonctionnel : `it_list` et `list_it`

Félicitations si vous connaissiez déjà `do_list`. Sinon, allez voir ce qu'elle fait dans le manuel. Allez voir aussi si vous ne connaissez pas `it_list` ou `list_it`, que nous allons maintenant utiliser. Ces fonctions un peu délicates à apprivoiser sont très utiles pour écrire vite du code sans erreurs et efficace.

Voici un exemple d'utilisation de `it_list` :

```
let somme_list liste = it_list (fun x y -> x+y) 0 liste
```

Question 3.1. Écrire la fonction `rev_map` (combinaison des fonctions `rev` et `map`) à l'aide de `it_list`. Indication : commencer éventuellement par réécrire `rev` à l'aide de `it_list`.

Écrire `map_option : ('a -> 'b option) -> 'a list -> 'b list` à l'aide de `list_it` : elle fonctionne comme `map` mais ignore les éléments pour lesquels la fonction donnée en argument renvoie `None`. Indication : commencer éventuellement par réécrire `map` à l'aide de `list_it`. Le type `option` est déjà défini en Caml par

```
type 'a option = None | Some of 'a
```

Il n'est donc pas nécessaire de le redéfinir. C'est un type somme classique, que l'on utilise à coups de filtrages :

```
let ma_fonction mon_option = match mon_option with
  | None -> ...
  | Some quelquechose -> ...
```

Bien comprendre la différence entre

- `it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` (la plus souvent utilisée) et
- `list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`.

Question 3.2. Écrire une fonction `aiguillage : 'a list -> 'a list * 'a list` qui partage une liste en deux listes de longueurs identiques à un près (l'ordre n'a pas d'importance).

Réécrire cette fonction en utilisant `it_list` si ce n'est pas déjà le cas.

Bonus : écrire cette fonction en 27 tokens (exemples de tokens : un nom de valeur comme `aiguillage`, « `let` », « `fun` », « `->` », « `=` », « `[]` », « `::` », « `;;` », « `,` », « `(` », « `)` »).

Question 3.3. Écrire une fonction récursive `fusion` qui prend deux listes triées et les fusionne en une liste triée contenant l'union des éléments des listes données en argument.

Question 3.4. Finir d'écrire la fonction `tri`.

Bonus (surtout si vous avez déjà écrit le tri fusion) : paramétrer le tout par une fonction de comparaison fournie par l'utilisateur. Ainsi on pourra définir

```
let tri_croissant = tri (fun x y -> x < y);;
let tri_decroissant = tri (fun x y -> x > y);;
let tri_case_insensitive =
  tri (fun x y -> string_lowercase x < string_lowercase y);;
```

Tester ces trois dernières fonctions. Il faut écrire la fonction `string_lowercase`. Pour cela, on peut transformer une majuscule non accentuée en minuscule par

```
let lowercase c = char_of_int (int_of_char c +32);;
```

Question 3.5. Quel est l'algorithme mis en œuvre par le code suivant ?

```
let tri liste =
  it_list
    (fun la lb -> fusion la lb)
    []
    (map (fun x -> [x]) liste);;
```

Les meilleurs élèves n'ayant jamais utilisé `it_list` ni `list_it` devraient arriver ici en deux heures.

4 La plus longue sous-suite commune

On se donne deux `'a vect` pour représenter deux suites finies de `'a` et l'on cherche à déterminer la plus longue sous-suite (ou suite extraite, par opposition à facteur qui est constitué d'éléments consécutifs) commune. On commence par déterminer simplement la longueur de cette sous-suite.

Question 4.1. On note $a_1, \ldots, a_n$ et $b_1, \ldots, b_m$ les deux suites, $\ell(i,j)$ la longueur de la plus longue sous-suite commune de $a_1, \ldots, a_i$ et $b_1, \ldots, b_j$. Compléter et justifier les équations suivantes :

$$\begin{aligned} \ell(i,0) \ = \ \ell(0,j) &= 0 \\ \ell(i,j) &= 1 + \ell(i-1,j-1) && \text{si } \ldots \\ \ell(i,j) &= \max(\ell(i-1,j), \ell(i,j-1)) && \text{sinon} \end{aligned}$$

On cherche donc $\ell(n,m)$.

Question 4.2. Écrire `lcss_ij` telle que `lcss_ij ell a b i j` inscrive $\ell(i,j)$ dans la case `ell.(i).(j)`. On supposera que $i, j > 0$ et que les cases voulues sont déjà remplies.

Question 4.3. Écrire `lcss` (*longest common subsequence*). On fera attention à n'appeler `lcss_ij` que lorsque ses hypothèses sont satisfaites.

Question 4.4 (bonus). Renvoyer la sous-suite et non simplement sa longueur.