TP3 MPSI: Récursivité et listes

ALEXIS SAURIN
http://www.eleves.ens.fr/~saurin/tp_caml

2 avril 2003

Introduction
On a vu dans le TP précédent les principaux aspects impératifs de Caml Light. On va aujourdhui étudier la récursivité, et on va notamment voir que cela rend la programmation fonctionnelle particulièrement élégante et que cela permet d'écrire des programmes très courts et plus facilement compréhensibles (et prouvables...).

Les étoiles qui précèdent certaines questions indiquent la difficultés de la question. Dans un premier temps, vous pouvez passer les questions ayant trois étoiles.[1]

1 Fonctions sur les listes.

Question 1: *Fonctions de base sur les listes*

Définir les fonctions classiques sur les listes : `longueur`, `map`, `membre`, `min_liste`, `concatene`, `renverse`, `aplati`, `assoc`...

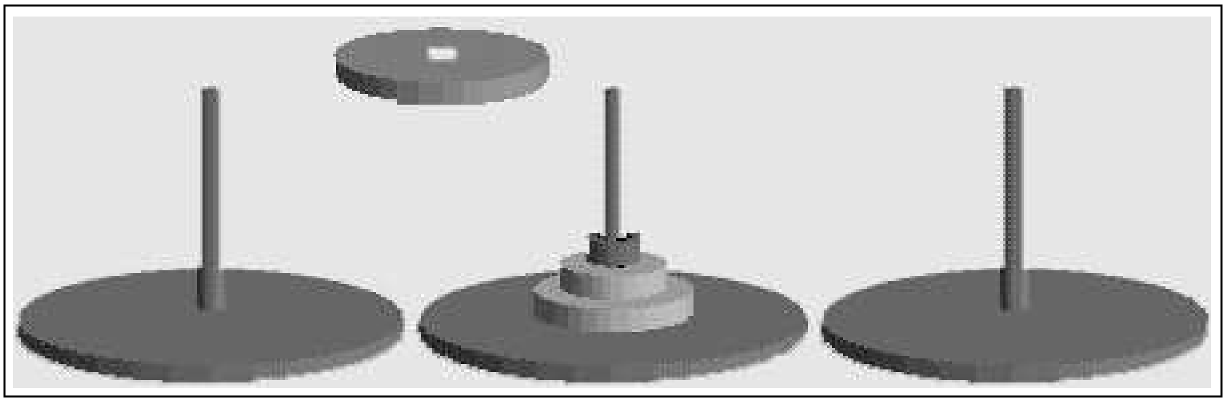
- ▷ `longueur : 'a list -> int`
- ▷ `map : ('a -> 'b) -> 'a list -> 'b list`
- ▷ `membre : 'a -> 'a list -> bool`
- ▷ `min_liste : int list -> int,` qui renvoie l'élémet minimal d'une liste
- ▷ `concatene : 'a list -> 'a list -> 'a list`
- ▷ `renverse : 'a list -> 'a list` (on fera un renversement en temps linéaire, à l'aide d'une fonction auxiliaire `rev_aux`
- ▷ `aplati : 'a list list -> 'a list` qui prend une liste de listes et renvoie une liste contenant les élements des listes.
- ▷ `assoc : 'a -> ('a*'b) list -> 'b` qui associe à un élément `x` de type `'a` le premier élément `y` de type `'b` tel que `(x, y)` est élément de la liste.

2 Un problème légendaire

Question 2: *Les tours de Hanoï*
Une légende raconte que dans un monastère, quelque part à Hanoï, sont placés trois grands piliers. Autour de ces piliers sont enfilés de gros anneaux de pierre. Il y a bien longtemps, alors que tous les anneaux (au nombre de 64) étaient placés sur le pilier de gauche les moines se virent fixer la tâche de déplacer les anneaux du pilier de gauche à celui de droite, en utilisant le pilier du milieu comme intermédiaire, et en ne déplaçant pas plus d'un anneau à la fois. Une condition essentielle était qu'à tout moment, l'ordre de taille des anneaux sur un pilier devait être respecté (du plus grand en bas au plus petit en haut). Depuis, les moines se sont mis à l'ouvrage.

Pendant ce temps, chaque matin, un petit oiseau vient prendre dans son bec quelques

[1] Parfois, des indications sont portées en notes de bas de page ; n'hésitez pas à vous y reporter !



cailloux au sommet de l'Everest et va les déposer au loin.

La légende affirme que la fin du monde arrivera dès que tous les anneaux se trouveront placés sur le pilier de droite ou que l'Everest aura disparu grâce aux efforts de l'oiseau.

Selon vous, l'Everest sera-t-il encore là lorsque le monde finira ?

Question 3: *Un peu de programmation tout de même* (∗∗)

Programmez une fonction `hanoi` qui prend en argument un entier n, imprime les mouvements effectués pour déplacer n anneaux du pilier de gauche au pilier de droite, et rend le nombre de mouvement effectués. Indice en bas de page[2]

3 Des suites numériques

3.1 Suite de Fibonacci

La suite de Fibonacci, $(u_n)_{n\in\mathbb{N}}$, est définie par la relation de récurrence suivante :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_{n+2} = u_{n+1} + u_n \end{cases}$$

On va voir plusieures manières de programmer cette suite comme une fonction Caml Light récursive `fib : int -> int`.

3.2 Calcul en temps exponentiel

Question 4: *Calcul de `fib` en temps exponentiel* (∗)

Programmez une fonction `fib` telle que `fib n` calcule le n^e terme de la suite de Fibonacci en s'inspirant directement de la définition de la suite.

Testez votre programme, d'abord sur de petits entiers (pour vérifier qu'il est correct), puis sur un entier plus grand, par exemple 30. Expliquez pourquoi le calcul est si long. Indication[3]

3.3 Calcul en temps linéaire

Question 5: *Calcul de `fib` en temps linéaire* (∗∗)

Il n'est pas très difficile de voir que le programme précédent fait beaucoup de travail inutile. Écrivez une version plus efficace de `fib` en utilisant une fonction auxiliaire `fib_aux` définie récursivement telle que `fib_aux n` retourne la paire (u_n, u_{n+1}).

[2] Indice : on définira une fonction auxiliaire qui prend quelques arguments en plus... de manière à construire une fonction récursive.

[3] Si vous ne voyez pas pourquoi, vous pouvez taper : `trace "fib";;` dans l'interpréteur, ce qui tracera l'exécution de la fonction et vous permettra de voir les appels de fonctions effectués. Pour arrêter de tracer de la fonction, tapez `untrace "fib";;`.

Testez maintenant votre programme sur de grands entiers...

3.4 Encore mieux : en temps logarithmique !

Ce n'est pas fini : on va faire encore mieux.

Question 6: *Une nouvelle relation de récurrence* (**)

Trouvez une relation de récurrence reliant u_{p+q} à u_{p+1}, u_p, u_q et u_{q-1}. Vous démontrerez par récurrence sur q que cette relation de récurrence est vraie pour tout entier naturel p. Indice en bas de page[4]

Question 7: *Un programme en temps logarithmique* (**)

Déduisez de l'expression précédente une expression de u_{2k}, u_{2k+1} et u_{2k+2} en fonction de u_k et u_{k+1}. En déduire une fonction `fib_aux2` telle que `fib_aux2 n` retourne le couple (u_n, u_{n+1}). Comparez la vitesse d'exécution de cette dernière version par rapport à la précédente pour de très grandes valeurs du paramètre n.

3.5 Fonction d'Ackermann

La fonction d'Ackermann est définie comme suit :

$$\begin{cases} Ack(0,n) = n+1 \\ Ack(m,0) = Ack(m-1,1) \\ Ack(m,n) = Ack(m-1, Ack(m,n-1)) \end{cases}$$

Question 8: *Implémentation* (*)

Définir une fonction `Ack` qui calcule la fonction d'Ackermann. La tester sur quelques exemples.

Question 9: *Terminaison* (**)

La définition de la fonction d'Ackermann présentée ci-dessus n'est pas une définition par récurrence habituelle. En particulier, il n'est pas clair que l'on ne fasse appel qu'à des éléments "plus petits" pour définir Ack(m,n), et d'ailleurs, que veut vraiment dire "plus petit" ici puisqu'on parle de couples...

Montrez que la définition donnée ci-dessus est bien licite, en ce sens qu'il existe un ordre bien fondé sur $\mathbb{N}^*\mathbb{N}$ tel que la définition de la valeur de la fonction en un point ne fasse appel qu'à des points plus petits pour cet ordre.

4 Un petit interpréteur d'expressions arithmétiques

On se donne un nouveau type défini comme suit :

```
type expression =
  Entier of int
 |Add of expression * expression
 |Mult of expression * expression
 |Moins of expression * expression
 |Puissance of expression * expression
 |Var of string;;
```

C'est ce qu'on appelle un type somme. Il faut le comprendre comme suit : un élément de ce type sera soit de la forme `(Entier n)` où n est de type `int`, soit de la forme `Add (E1, E2)` où `E1` et `E2` seront des éléments de type `expression`...

Question 10: *Interpréteur d'expressions arithmétiques* (**)

Définir une fonction `evalue` de type `expr -> (string*int) list -> int` où la liste de type `(string*int) list` est un contexte d'évaluation des variables de l'expression, ou

[4]Indice : $u_{p+1} = u_{p+1}u_1 + u_p u_0$ (pour l'initialisation) et $u_{(p+q)+1} = u_{(p+1)+q}$ (pour la propagation).

encore une valuation des variables (c'est-à-dire les valeurs entières que désignent ces variables).[5]

Question 11: *Impression d'une expression arithmétique* (∗∗)

Définir une fonction qui imprimera une expression arthmétique de type `expression`. On veut donc une fonction de type `expression -> unit`. Attention à bien mettre des parenthèses, et si possible à ne pas en mettre trop !

Question 12: *Développement d'une expression arithmétique* (∗∗∗)

Définir une fonction `developpe` qui prend une expression E et rend une expression qui est la forme développée de E.

Question 13: *Et avec des fonctions ?* (∗∗∗)

On complique un peu les choses en enrichissant le type `expression` avec la construction suivante :

```
| Fun_Appli
  of string * expression * expression
```

où Fun_Appli ("x", Add(Var "x", Entier 1), Mult(Entier 3, Entier 4)) désigne l'application de la fonction qui à x associe x+1 à l'expression Mult (Entier 3, Entier 4).

Définir la nouvelle fonction d'évaluation.

5 Tris avec la méthode "Diviser pour Régner"

On étudie ici les algorithmes de tris de listes appelés tri fusion et tri rapide, qui sont plus efficaces que les méthodes de tri à bulles, tri par sélection ou par insertion.

5.1 Tri fusion

Le principe du tri fusion est très simple : On partage la liste en deux liste de tailles similaires, on trie récursivement les deux listes avec un tri fusion, on fusionne les deux listes.

Question 14: *Implémentation du tri fusion* (∗∗)

Définir un fonction `partage` qui prend un liste et rend un couple de listes tel que les éléments en position impaire sont dans la liste de gauche et les éléments en position paire sont dans la liste de droite.

Définir ensuite une fonction `fusion` qui fusionne deux listes triées.

Définir enfin une fonction `tri_fusion` qui effectue le tri.

5.2 Tri rapide

Question 15: *Tri rapide* (∗∗)

Le principe du tri rapide est proche de celui du tri fusion même s'il est un peu plus intelligent dans sa manière de faire le partage : Dans le tri rapide, l'opération de partage des listes va être un peu plus compliquée que pour le tri fusion mais l'opération de fusion sera simplement une concaténation de listes...

Pour partager la liste l, on compare les éléments à répartir avec le premier élément de la liste, les éléments plus petits sont mis dans une liste l_1 (qui sera triée en l_2), les éléments plus grand sont mis dans une liste l'_1 (qui sera triée récursivement en une liste l'_2). Le résultatdu tri est la concaténation de l_2, du premier élément de l (à ne pas oublier, sinon il ne restera plus beaucoup d'éléments dans votre liste résultat...), et de l'_2.

Définir une fonction `tri_rapide` prenant une liste et la triant selon l'algorithme du tri rapide.

[5]Remarque : on supposera que les variables sont toujours bien définies dans le contexte, ou bien on prévoira de lever une exception au cas où on ne trouve pas la variable dans la liste d'association.