Plan

Présentation du langage CAML

thierry.montaut@univ-jfc.fr

Les listes

Il est parfois nécessaire de disposer de structure de données permettant de traiter des suites d'objets **de taille variable.** C'était la principale limitation des n-uplets qui doivent avoir une taille constante fixée à l'avance.

Définition

On appelle **liste** *en CAML une suite finie, ordonnée, éventuellement vide, d'objets de même type.*

L'avantage sur les n-uplets est donc que la taille de la suite est variable mais comme on n'a rien sans rien dans cette vallée de larmes, on est contraint dans le cas des listes de n'utiliser que des objets de même type.

Le type d'une liste d'objets d'un même type *t* est :

t List

Le type *t* est un type quelconque connu par CAML :

Exemple :

```
#[1;2;3];;
- : int list = [1; 2; 3]
#[1+2;2+3];;
- : int list = [3; 5]
#["a";"bonjour";"c"];;
- : string list = ["a"; "bonjour"; "c"]
#[(1,2);(2,3);(3,1)];;
- : (int * int) list = [1, 2; 2, 3; 3, 1]
```

Construction de listes en CAML

Il existe deux manières principales de construire les listes.

- Soit on connait la taille et les éléments de la liste et on la définit par extension comme sur les exemples précédents. La syntaxe générale d'une telle définition de liste est alors :

  [exp1 ;exp2 ; ... ;exp3]

- Soit de manière évolutive. On part alors d'une liste existante (éventuellement la liste vide notée [] et appelée "nil") et on lui ajoute des éléments **en tête de liste** à l'aide du constructeur de liste noté : : et appelé "cons".

```
#[];;
- : 'a list = []
#6::[1;2;3];;
- : int list = [6; 1; 2; 3]
#2::1::[];;
- : int list = [2; 1]
```

On dispose également d'une opération interne, la concaténation de liste noté @ .

```
#[1;2]@[3;4;5];;
- : int list = [1; 2; 3; 4; 5]
#[1;2]@[];;
- : int list = [1; 2]
```

Type d'une liste

Soit à évaluer la liste

$$[exp1\ ;exp2\ ;...\ ;expn]$$

Les types des expression exp_i sont évalués.

- S'il ne sont pas tous égaux, il y a erreur de type,
- s'ils sont tous d'un même type t, la liste est de type t list.

```
#Entrée interactive:
>[1;1.0];;
>^^^^^^^^
Cette expression est de type float list,
mais est utilisée avec le type int list.
```

Valeur d'une liste

Les expressions exp_i sont évaluées dans l'environnement actif Env. Si la valeur de exp_i est v_i pour tout i alors la valeur de la liste dans l'environnement Env est

$$[v_1; v_2; \ldots; v_n].$$

Evaluation du constructeur cons

La syntaxe générale est :

$$exp :: L$$

où exp est une expression et L une liste.
Si exp est de type t et L de type t list alors le résultat est de type t list.
Dans les autres cas il y a une erreur de typage.
Si exp a pour valeur v et L a pour valeur $[v_1; v_2; \ldots; v_n]$ alors le résultat a pour valeur

$$[v; v_1; v_2; \ldots; v_n].$$

Evaluation de l'opérateur de concaténation

La syntaxe générale est :

$$L1 \ @@ \ L2$$

où L1 et L2 sont des listes.
Si L1 et L2 sont de même type t list alors le résultat est de type t list.
Dans les autres cas il y a une erreur de typage.
Si L1 a pour valeur $[v_1; v_2; \ldots; v_n]$ et L2 a pour valeur $[w_1; w_2; \ldots; w_p]$ alors le résultat a pour valeur

$$[v_1; v_2; \ldots; v_n; w_1; w_2; \ldots; w_p].$$

Sélection d'un élément dans une liste

Il existe deux fonctions prédéfinies dans le noyau de CAML permettant d'extraire (on dit sélectionner) des éléments d'une liste :

- hd (head) sélectionne le premier élément d'une liste non vide (la tête). Le résultat est un élément de type *t*.
- tl (tail) extrait la sous-liste privée de son premier élément si elle est non vide. Le résultat est donc de type *t* list.

Sur une liste vide ces deux fonctions provoque une erreur.

```
#tl([1;2;3]);;
- : int list = [2; 3]
#hd([1;2;3]);;
- : int = 1
#hd([]);;
Exception non rattrapée: Failure "hd"
```

Sélection d'un élément par filtrage

Mais une solution plus idiomatique consite à utiliser un filtrage par motif sur la liste. Il est en effet possible d'utiliser le constructeur : : dans un filtre.
Dans ce cas le filtre doit obligatoirement être écrit entre parenthèse.
Par exemple le filtre (a : :b : :l) permet de filtrer toutes les listes s'écrivant un élément, un deuxième puis une lste quelconque, donc toutes les listes d'au moins deux éléments et il est alors facile de disposer de *b* le deuxième élément de la liste.

Exercices : manipulations non récursives de listes

Compléter les sessions suivantes :

```
#[1;2;3];;
#["ceci est une liste de chaînes"];;
#[("dupont","jean",15);("martin","pierre",65);("durand","m
#hd[("dupont","jean",15);("martin","pierre",65);("durand",
#tl[("dupont","jean",15);("martin","pierre",65);("durand",
#[1;"a";true];;
#"a"::["b";"cd"];;
```

Exercices : manipulations non récursives de listes

Donner les fonctions permettant d'accéder à l'élément 3 des listes suivantes (on précisera leur type) :

```
#let l1=[1;2;3;4];;
#let l2 = [[1;2];[3;4]];;
#let l3 = [[1];[2];[3];[4]];;
#let l4 = [(1,2);(3,4)];;
#let l5 = [[[1];[2];[3];[4]]];;
```

Exercices : manipulations non récursives de listes

Donner les résultats des évaluations suivantes :

```
#let l1=[1;2];;
#let l2 = [3;4];;
#let l3=l2;;
#hd l1 :: tl l1;;
#[l1;l2;l3];;
#l1::[];;
#l1::l2;;
#l1::l2::[];;
#l1 @ l2;;
#l2::l3::[];;
```

Exercices : manipulations non récursives de listes

Donner les expressions donnant les résultats suivants :

```
#[1;2]?[3];;
- : int list = [1; 2; 3]
#[1;2]?[[3]];;
- : int list list = [[1; 2]; [3]]
#[1]?[];;
- : int list list = [[1]]
#[1;2]?[3]?[];;
- : int list list = [[1; 2]; [3]]
#[]?[[1]];;
- : int list list = [[]; [1]]
#[1;2]?[3]?[[]];;
- : int list list = [[1; 2]; [3]; []]
#[1;2]?[];;
- : int list = [1; 2]
```

Exercices : manipulations non récursives de listes

- Ecrire la fonction deuxième qui extrait le deuxième élément d'une liste.
- Ecrire une fonction testant si une liste a au moins trois éléments.
- Ecrire une fonction qui fait la somme des trois premiers éléments d'une liste d'entiers.
- Ecrire une fonction testant si le troisième élément d'une liste est pair.
- Ecrire une fonction qui ajoute deux fois un élément en tête d'une liste.
- Ecrire une fonction permute qui échange les deux premiers éléments d'une liste.

Exercices : Liste et récursivité

Les listes se prètent particulièrement bien à un traitement récursif. En effet l'ensemble des listes ordonnée partiellement par inclusion est un ensemble bien ordonné d'élement minimal la liste vide. On est donc "naturellement" placé dans les hypothèses de TFT en ramenant un problème sur les liste à une liste strictement incluse dans la première, obtenue bien souvent en lui enlevant le ou les premiers éléments.

Exercices : Liste et récursivité

1. Ecrire une fonction qui à un entier n associe la liste

$$[n; n-1; \ldots; 1; 0].$$

2. Ecrire une fonction qui calcule la longueur d'une liste
3. Ecrire (sans utiliser la fonction longueur) une fonction qui teste si une liste possède une nombre pair d'éléments.
4. Ecrire une fonction permettant de construire la liste des éléments de rang impair d'une liste donnée.
5. Donner le dernier élément d'une liste.
6. Calculer la somme des éléments d'une liste donnée.

Encore ?

Ecrire les fonctions Caml suivantes :

1. Tester si un élément appartient à une liste.
2. Calculer le maximum d'une liste d'entiers.
3. *nbocc* qui, étant donnés un élément et une liste, calcule le nombre d'occurrences de l'élément dans la liste.
4. *fois2* qui, étant donnée une liste d'entiers, construit la liste des doubles des entiers de la liste.
5. *inserer* un entier dans une liste d'entiers ordonnée dans l'ordre croissant.

Encore ?

Ecrire les fonctions Caml suivantes :

1. *trier* une liste d'entiers dans l'ordre croissant par une méthode simple.
2. *ieme* qui retourne le ieme élément d'une liste.
3. *prendre* qui, étant donnés un entier p et une liste l retourne la liste des p premiers éléments de l.
4. *enleve* qui, étant donnés un entier p et une liste l retourne la liste privée des p premiers éléments de l.
5. *melange* qui prend une paire de listes et retourne la liste des paires correpondante. Si les deux listes n'ont pas même longueur, on s'arretera à la liste la plus courte.

Algorithmes de tri des listes

- Les méthodes de tri sont très importantes en pratique, de nombreux problèmes (notamment en informatique de gestion) se ramenant à des tris de listes.
  Nous en verrons des applications en théorie des graphes (L3 Inf).
- Le tri est également un bon exemple de problème pour lequel il existe de nombreux algorithmes très différents les uns des autres. Il sert donc souvent de base pour présenter les techniques d'analyse de performance ou d'optimisation (cf complexité des algorithmes en L3 Inf).
- Le but est ici de présenter une première fois les différentes stratégies classiques de tri de listes afin de commencer à vous familiariser avec ces techniques.

Conventions - Hypothèses

Le problème à résoudre est le suivant :
Etant donné une liste L d'éléments appartenant à un type totalement ordonné, écrire une fonction donnant la liste L' contenant les mêmes éléments mais triés dans l'ordre croissant.

- On peut considérer (ou pas) que les éléments de la liste initiale sont différents.
- Des modifications mineures permettent de trier les listes dans l'ordre décroissant.
- On se restreindra dans la suite au cas d'une liste d'entiers relatifs que l'on souhaite trier pour l'ordre usuel de $\mathbb{N}$.

On pourra distinguer les tris "en place“ qui ne nécessitent pas de listes auxilliaires mais réordonnent les éléments au sein de la liste *L*.

Tri naïf (par sélection du minimum)

Cette première méthode naïve de tri consiste à calculer le minimum de la liste L, à l'échanger avec l'élément de tête de la liste et à trier récursivement le reste de la liste.

Indications...

Cette solution nécessite l'écriture :

- d'une fonction *min* permettant de calculer le minimum d'une liste non vide.
- d'une fonction *enleve* permettant d'enlever un élément d'une liste (sachant qu'il y est présent).

L

’idée récursive est la suivante :
$naif(l)$ est obtenu par $min(l)$ :: $naif(enleve(min(l), l))$

Solution Caml

```
(* Tri naif par calcul de minimum *)
#let rec min = fun
 [a]-> a
| (a::l) -> let b= min(l) in if a<b then a else b;;
Attention: ce filtrage n'est pas exhaustif.
min : 'a list -> 'a = <fun>

#let rec enleve = fun
(a,(b::l))->if a= b then l else b::enleve(a,l)
|(a, [])->[];;
enleve : 'a * 'a list -> 'a list = <fun>

#let rec naif = fun
[] ->[]
| l -> let a=min(l) in a::naif(enleve(a,l));;
naif : 'a list -> 'a list = <fun>
```

Tri par insertion

C'est l'algorithme de tri du joueur de carte. On considère les éléments un par un et on les insère dans une liste annexe, initialement vide et qui doit toujours être maintenue triée.

Indications...

1) Ecrire une fonction *insertion* permettant d'insérer un élément dans une liste triée.
2) Ecrire la fonction récursive *tri_insere* dont l'idée récursive consiste à insérer l'élément de tête dans le reste de la liste, une fois celui-ci trié.

Solution Caml :

```
(* Tri par insertion *)

#let rec insere = fun
(x,[])->[x]
|(x,(y::l))-> if x<=y then x::y::l else y::insere (x,l);;
insere : 'a * 'a list -> 'a list = <fun>

#let rec tri_insere = fun
(x::l)->insere(x ,tri_insere(l))
|[]->[];;
tri_insere : 'a list -> 'a list = <fun>
```

Tri par fusion

Le tri par fusion (ou merge sort) consiste à partager la liste de taille n à trier en deux listes de taille $\frac{n}{2}$, à trier chacune d'elle et à fusionner les deux listes triées.
Il s'agit d'une technique de programmation appartenant à la vaste famille des algorithmes de type "diviser pour régner.“
(divide and conquer).

Indications...

1) Ecrire une fonction *divise* permettant de diviser une liste en deux.
2) Ecrire une fonction *fusion* permettant de fusionner deux listes triées.
3) Ecrire la fonction *tri_fusion* dont l'idée récursive est de diviser la liste en deux, puis trier chaque moitié, puis enfin fusionner les deux listes triées obtenues.

Solution Caml :

```
(* Tri fusion *)
#let rec divise = fun
[]->([],[])
|[x]->([x],[])
|(a::b::l)->let (l1,l2)=divise(l) in (a::l1,b::l2);;

divise : 'a list -> 'a list * 'a list = <fun>


#let rec fusion = fun
(l,[])->l
|([],l)->l
|((a::r),(b::s))-> if a<b then a::fusion(r,(b::s))
                    else b::fusion((a::r),s);;
fusion : 'a list * 'a list -> 'a list = <fun>
```

Tri à bulle

On va parcourir la liste à trier par couples d'éléments. Si un couple n'est pas ordonné, on échange les deux éléments. l'algorithme s'arrête lorsqu'un parcours n'a pas provoqué d'échange.

Indications...

Cet algorithme est plutôt fait pour une programmation impérative. La solution fonctionnelle récursive n'est pas très naturelle.
1) Ecrire une fonction récursive *parcours* parcourant toute la liste, inversant les couples qui sont mal ordonnés et mettant à jour un booléen *inversion* valant vrai si le parcours à provoqué au moins une inversion.
2) Ecrire une fonction récursive *bulle* auxilliaire permettant de répéter les parcours tant que ceux-ci provoquent des inversions.
3) Il faut enfin une fonction *tri_bulle* pour provoquer l'appel initial à *bulle*.

Solution en Caml :

```
#let rec parcours = fun
 ((a::b::l),inv) ->
   if a>b then
         let (ll,i)=parcours(a::l,inv) in (b::ll,true)
   else let (ll,i)=parcours(b::l,inv) in (a::ll,i)
| ([a],inv)->([a],false) | ([],inv)->([],false);;

parcours : 'a list * 'b -> 'a list * bool = <fun>

#let rec bulle_aux =  fun
(l,false) ->l
|(l,true) -> bulle_aux(parcours(l,true));;

bulle_aux : 'a list * bool -> 'a list = <fun>

#let bulle = fun l-> bulle_aux(l,true);;
```

Tri rapide

Au contraire du tri bulle, ce tri se décrit très bien de manière récursive.
Le principe est le suivant :
Si L possède au moins deux éléments, on considère son premier élément que l'on appelle le pivot p. On partitionne alors L en deux sous-listes L_1 contenant les éléments de L strictement inférieurs à p et L_2 contenant les éléments de L supérieurs au pivot.
On poursuit récursivement les tris de L_1 et L_2.

Indications...

1) Ecrire une fonction *partition* qui à un pivot et à une liste donnée associe les deux sous-listes définies comme précédemment.
2) Ecrire une fonction *quick* mettant en oeuvre l'idée récursive : partitionner la liste en prenant comme pivot son premier élément, trier chacune des deux moitiés puis les concaténer.

Remarques :

- On reconnaît une stratégie de type "diviser pour régner.“
- Nous verrons au L3 que cette technique très simple à mettre en oeuvre est également très efficace.

Solution Caml :

```
(* Tri rapide *)
#let rec partition = fun
(_,[])->([],[])
| (e,d::r)-> let l1,l2 = partition (e,r)
          in if d<e then (d::l1,l2) else (l1,d::l2);;
partition : 'a * 'a list -> 'a list * 'a list = <fun>


#let rec quick = fun
[x]->[x]
|(e::r)->let l1,l2 = partition (e,r)
          in (quick l1)@(e::quick(l2))
|[]-> [];;
quick : 'a list -> 'a list = <fun>
```