

LANGAGE OCaml – TP 4

Listes

Introduction

Parmi les types vus en cours, les listes chaînées se prêtent bien à la programmation récursive. En langage Caml, leur manipulation peut se faire grâce à un certain nombre de fonctions programmées dans un module. D'autres fonctions complémentaires peuvent aussi être définies par l'utilisateur. L'accès à une fonction présente dans un module se fait suivant la syntaxe suivante : `Nom_module.nom_fonction`. Les noms des modules ont toujours leur première lettre en majuscule, afin de les distinguer des noms des variables.

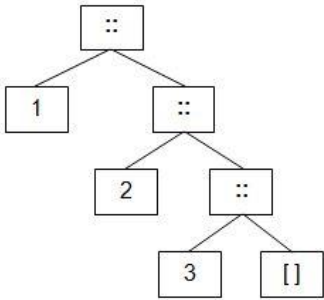
Listes en Caml

Le type correspondant aux listes chaînées est déjà implémenté en Caml.

Le module `List` dispose de certaines fonctions permettant de réaliser des opérations sur les listes chaînées.

Constructeur

OCaml permet de manipuler des listes homogènes (listes chaînées), c'est-à-dire constituées d'éléments du même type. Les listes sont des structures de données récursives définies à partir de deux constructeurs : `[]` pour la liste vide, et `::` pour l'adjonction d'un élément en tête de liste. Lors d'une construction par énumération, les éléments sont séparés par un `;`.

<pre>let liste_0 = 1 :: 2 :: 3 :: [];;</pre>	<p>Sortie écran et commentaires :</p> <pre># val liste_0 : int list = [1; 2; 3] (* Adjonction en tête de liste de plusieurs éléments *)</pre> <p>La création de cette liste peut être représentée par l'arbre suivant :</p>  <pre> graph TD A["::"] --> B["1"] A --> C["::"] C --> D["2"] C --> E["::"] E --> F["3"] E --> G["[]"] </pre>
<pre>let liste_1 = [1;2;3];;</pre>	<p>Sortie écran et commentaires :</p> <pre># val liste_1 : int list = [1; 2; 3] (* Construction par énumération *)</pre>

<pre>let liste_2 = [1,2,3];;</pre>	<p>Sortie écran et commentaires :</p> <pre># val liste_2 : (int * int * int) list = [(1, 2, 3)] (* Liste contenant un triplet*)</pre>
<pre>let liste_3 = 1 :: 'a' ::(1,2,3) [];;</pre>	<p>Sortie écran et commentaires :</p> <pre>let liste_3 = 1 :: 'a' ::(1,2,3) [];; ^^^^ Error: This expression has type char but an expression was expected of type int (* type homogène indispensable *)</pre>
<pre>let liste_4 = [];; (* Liste polymorphe à la création *) 2 :: liste_4;; 2. :: liste_4;; liste_4;; 'a' :: liste_4;; liste_4 ;;</pre>	<p>Sortie écran et commentaires :</p> <pre># val liste_4 : 'a list = [] # - : int list = [2] # - : float list = [2.] # - : 'a list = [] # - : char list = ['a'] # - : 'a list = [] (* liste restant vide *)</pre>

Accesneur

Ces différentes fonctions du module List ne sont pas à connaître.

Lors d'un examen, si elles ne sont pas rappelées dans l'énoncé ou en annexe, vous devez les reprogrammer.

<pre>let liste = [5;4;3;2;1];; let a = List.hd liste;; liste;;</pre>	<p>Commentaires et sortie écran :</p> <pre># val liste : int list = [5; 4; 3; 2; 1] # val a : int = 5 # - : int list = [5; 4; 3; 2; 1] (* hd pour head *) (* Accesneur en tête de liste sans pop *)</pre>
<pre>let l = [];; let b = List.hd l;;</pre>	<p>Commentaires et sortie écran :</p> <pre># val l : 'a list = [] # Exception: Failure "hd".</pre>
<pre>let t = List.tl liste;;</pre>	<p>Commentaires et sortie écran :</p> <pre># val t : int list = [4; 3; 2; 1]</pre>

	<pre>(* Attention : le reste de la liste est tout ce qu'il y a après la tête de liste *) (* tl pour tail *)</pre>
<pre>let liste = [5;4;6;2;1;3];; let a = List.nth liste 4;; let b = List.nth liste 7;;</pre>	<p>Même si cette fonction existe, il est très très fortement déconseillé de l'utiliser.</p> <p>Il n'est pas dans l'esprit de la manipulation de liste d'aller chercher le nième composant.</p> <p>Par ailleurs, cette fonction présente un coût temporel non négligeable.</p> <p>Commentaires et sortie écran :</p> <pre># val liste : int list = [5; 4; 6; 2; 1; 3] # val a : int = 1 (* indexation à partir de 0 *) # Exception: Failure "nth". (* out of range *)</pre>
<pre>let liste_5 = [[1;2];[1;2;3];[1;2;3;4;5;6]];;</pre>	<p>Commentaires et sortie écran :</p> <pre># val liste_5 : int list list = [[1; 2]; [1; 2; 3]; [1; 2; 3; 4; 5; 6]] (* type homogène : listes *)</pre>

Transformateur : ajout en tête de liste

<pre>let liste_6 = [];; let liste_6 = 5 :: liste_6;; let liste_6 = 2 :: liste_6;; liste_6;;</pre>	<p>Sortie écran et commentaires :</p> <pre># val liste_6 : 'a list = [] # val liste_6 : int list = [5] # val liste_6 : int list = [2; 5] # - : int list = [2; 5] (* Attention : type homogène *) (* Adjonction en tête de liste *)</pre>
---	--

Transformateur : concaténation de deux listes (à connaître)

<pre>let l1 = [1;2;3];; let l2 = [4;5;6];; let l3 = [7;8;9];; let l = List.append l2 l1;;</pre>	<p>Sortie écran et commentaires :</p> <pre># val l1 : int list = [1; 2; 3] # val l2 : int list = [4; 5; 6] # val l3 : int list = [7; 8; 9] # val l : int list = [4; 5; 6; 1; 2; 3]</pre>
<pre>let l1 = l @ l3;;</pre>	<p>Sortie écran et commentaires :</p>

	<pre># val ll : int list = [4; 5; 6; 1; 2; 3; 7; 8; 9] (* autre manière de concaténer deux listes *)</pre>
--	---

Autres opérations du module List

<pre>let l_1 = List.map (function x -> x*x) [1;2;3;4];;</pre>	<p>Sortie écran et commentaires :</p> <pre># val l_1 : int list = [1; 4; 9; 16]</pre> <p>(fonction mentionnée dans le programme mais pas à connaître)</p>
<pre>let a = List.length [1;2;3;4];;</pre>	<p>Sortie écran et commentaires :</p> <pre># val a : int = 4</pre> <p>(fonction mentionnée dans le programme et à connaître)</p>

Listes et filtrage

De nombreuses opérations sur les listes sont disponibles dans le module List. La puissance des listes en Caml vient de la possibilité de construction par cas sur la forme d'une liste, en utilisant le filtrage. En effet, une liste peut être considérée soit comme une liste vide, soit comme formée d'un premier élément suivi d'une liste : c'est un type récursif.

L'architecture d'une fonction récursive sur les listes est donc généralement de la forme :

```
let rec fonction liste =
  match liste with
  | [ ] -> expression
  | x :: tl -> expression ;;
```

On peut, par exemple, écrire une fonction calculant la somme des éléments d'une liste d'entiers de la manière suivante :

<pre>let rec somme liste = match liste with [] -> 0 x :: r -> x + somme r ;; let liste = [1;2;3];; let s = somme liste;;</pre>	<p>Sortie écran :</p> <pre># val somme : int list -> int = <fun> # val liste : int list = [1; 2; 3] # val s : int = 6</pre>
--	--

Les listes peuvent donc être passées en argument ou retourner par des fonctions.

Exercices

Exercice 1 : détermination de la longueur d'une liste chaînée

Ecrire une fonction récursive longueur de signature `val longueur : 'a list -> int = <fun>` déterminant la longueur d'une liste chaînée.

```
let rec longueur liste =  
  match liste with  
  | [] -> 0  
  | x :: q -> 1 + longueur q ;;
```

```
let len = longueur [1;2;3;4;5];;
```

Sortie écran :

```
# val longueur : 'a list -> int = <fun>  
# val len : int = 5
```

Exercice 2 : concaténation de deux listes

Ecrire une fonction récursive concatene de signature `val concatene : 'a list -> 'a list -> 'a list = <fun>` réalisant la concaténation de deux listes.

```
let rec concatene liste1 liste2 =  
  match liste1 with  
  | [] -> liste2  
  | x :: q -> x :: concatene q liste2;;
```

```
concatene [3;2;1] [6;5;4];;
```

Sortie écran :

```
# val concatene : 'a list -> 'a list -> 'a list = <fun>  
# - : int list = [3; 2; 1; 6; 5; 4]
```

Exercice 3 : avant-dernier élément d'une liste

Écrire une fonction récursive de signature `val avant_dernier : 'a list -> 'a = <fun>` qui retourne l'avant dernier élément d'une liste, s'il existe.

L'instruction `failwith` sera utilisée pour éviter les erreurs irrattrapables (liste vide ou liste comprenant un seul élément).

```
let rec avant_dernier lst = match lst with  
  | [] -> failwith "liste vide"  
  | t::[] -> failwith "liste à un élément"  
  | p::s::[] -> p  
  | t::q -> avant_dernier q;;
```

```
(*Tests : *)
avant_dernier [4;3;2;8;9];;
avant_dernier [4;3;2;8;9;10];;
```

Exercice 4 : liste des préfixes d'une liste

Écrire une fonction de signature `val prefixes : 'a list -> 'a list list = <fun>` calculant la liste de tous les préfixes d'une liste donnée.

Par exemple :

```
prefixes [1 ;2 ;3 ;4] ;;
```

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

```
let rec prefixes lst =
  let rec ajout_a_chaque x lst2 =
    (*Ajoute x à chaque composante d'une liste de listes*)
    match lst2 with
    | [] -> []
    | t2::q2-> (x::t2)::(ajout_a_chaque x q2)
    (* decompose lst2 en t2 + le reste
       Concatène x avec t2 : donc constitution d'une liste [x;t2]
       [x;t2] sera ajouté comme un élément à la liste renvoyée par
       l'appel récursif de x et du reste de lst2 *)
    (* Donc si x et lst2 sont 1 [1;2;3], on aura
       1::[2;3] qui donnera [1;1] à conc avec appel 1 [2;3]
       [1;1] à conc avec 1 pour x et 2::[3] pour lst2,
       Soit [1,1]::[1,2] a conc avec appel de 1 [3]
       Soit [1,1]::[1,2]::[1,3] à conc avec appel de 1 [] qui renvoie []
       Soit [1,1]::[1,2]::[1,3]::[] donc [[1,1];[1,2];[1,3]]
       Tracer l'arbre d'appels *)

  in
  match lst with
  | [] -> []
  | t::q -> [t]::(ajout_a_chaque t (prefixes q))
  (* idem : faire tracer la liste obtenue pour une lst simple *)
  (* Donc, pour lst = [1;2;3;4]
     On aura [1] :: (ajout 1 (prefixes [2 ;3 ;4]))
     [1]::(ajout 1 ( [2] :: (ajout 2 (prefixes [3 ;4]) ) ) )
     [1]::(ajout 1 ( [2] :: (ajout 2 ([3] :: (ajout 3 prefixes [4]) ) ) ) )
     [1]::(ajout 1 ( [2] :: (ajout 2 ([3] :: (ajout 3 [4] :: (ajout 4 []))))))
     [1]::(ajout 1 ( [2] :: (ajout 2 ([3] :: (ajout 3 ::(4 :: []))))))
     [1]::(ajout 1 ( [2] :: (ajout 2 ([3] :: [[3;4]])))
     [1]::(ajout 1 ( [2] :: (ajout 2 ([3];[[3;4]])))
     [1]::(ajout 1 ( [2] :: [[2;3];[[2;3;4]]])
     [1]::[[1;2];[[1;2;3];[[1;2;3;4]]]]
     [[1] ;[1;2];[1;2;3];[1;2;3;4]]]]
;;
```

```
(*TEST*)
*prefixes [1;2;3;4];;
```

Exercice 5 : mise à plat d'une liste de listes

Écrire une fonction `flatten : 'a list list -> 'a list` aplatissant une liste de listes.

Par exemple

```
flatten [[3 ;4] ;[5 ;7 ;9] ;[0] ;[6 ;8]]
```

doit renvoyer `[3 ;4 ;5 ;7 ;9 ;0 ;6 ;8]`.

```
let rec flatten lst = match lst with
| []          -> []
| []::s       -> flatten s
| (t::q1)::s  -> t::(flatten (q1::s));;
```

```
(* TEST *)
flatten [[3;4];[2];[5;6;8];[];[4]];
```

Déroulement

```
Flatten [[3;4];[2];[5;6;8];[];[4]]
Flatten [ 3 :: [4]] :: [[2];[5;6;8];[];[4]]
Flatten 3 :: flatten [[4];[2];[5;6;8];[];[4]]
Flatten 3 :: flatten [4 :: []] :: [[2];[5;6;8];[];[4]]
Flatten 3 :: 4 :: flatten [[]] :: [[2];[5;6;8];[];[4]]
Flatten 3 :: 4 :: flatten [[2];[5;6;8];[];[4]]
Flatten 3 :: 4 :: flatten [2 :: []] :: [5;6;8];[];[4]]
Flatten 3 :: 4 :: 2 :: flatten [] :: [5;6;8];[];[4]]
Flatten 3 :: 4 :: 2 :: flatten [[5;6;8];[];[4]]
Flatten 3 :: 4 :: 2 :: flatten [[5;6;8];[];[4]]
Flatten 3 :: 4 :: 2 :: flatten [5 :: [6;8]] :: [[];[4]]
Flatten 3 :: 4 :: 2 :: 5 :: flatten [6;8];[];[4]]
Flatten 3 :: 4 :: 2 :: 5 :: flatten 6::[8] :: [[];[4]]
Flatten 3 :: 4 :: 2 :: 5 :: 6 flatten [[8];[];[4]]
Flatten 3 :: 4 :: 2 :: 5 :: 6 flatten [8 :: []] :: [[];[4]]
Flatten 3 :: 4 :: 2 :: 5 :: 6 :: 8 :: flatten [[] ;[];[4]]
Flatten 3 :: 4 :: 2 :: 5 :: 6 :: 8 :: flatten [[];[4]]
Flatten 3 :: 4 :: 2 :: 5 :: 6 :: 8 :: flatten [[4]]
Flatten 3 :: 4 :: 2 :: 5 :: 6 :: 8 :: flatten [4 :: []]
Flatten 3 :: 4 :: 2 :: 5 :: 6 :: 8 :: 4 :: flatten[]
Flatten 3 :: 4 :: 2 :: 5 :: 6 :: 8 :: 4 :: []
[3 ;4 ;2 ;5 ;6 ;8 ;4]
```

Sortie :

```
# val flatten : 'a list list -> 'a list = <fun>
# - : int list = [3; 4; 2; 5; 6; 8; 4]
```

Exercice 6 : occurrence du minimum d'une liste

Écrire une fonction `ocsmin : 'a list -> 'a * int` qui renvoie le couple formé par l'élément minimum d'une liste et le nombre de fois où il apparaît. Un `failwith` sera utilisé pour envisager le cas d'une liste vide.

```
let rec ocsmin lst = match lst with
| []      -> failwith "liste vide"
| [t]     -> (t,1)
| t::q    -> match ocsmin q with
| (min,nbocc) when t<min -> (t,1)
| (min,nbocc) when t=min -> (t,(nbocc + 1))
| (min,nbocc)             -> (min,nbocc);;
```

```
(* TEST *)
```

```
ocsmin [4;3;2;8;9;2];;
```

Sortie écran :

```
# val ocsmin : 'a list -> 'a * int = <fun>
# - : int * int = (2, 2)
```