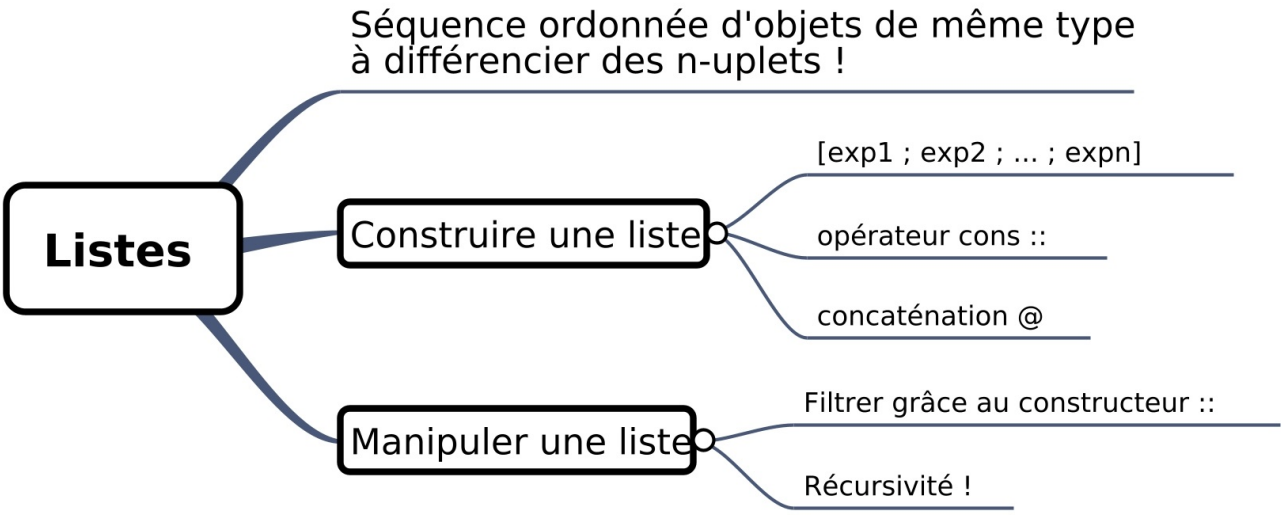
Institut National Universitaire Champollion

Chapitre 6 - Les listes

En bref



1. Les listes, généralités

1.1. Liste VS n-uplets

Il est parfois nécessaire de disposer d'une 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 1

En Caml, une **liste** est une séquence (=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 l'inconvénient, est que l'on est contraint, dans une liste, de n'avoir que des objets du même type.
En effet, le type d'une liste d'objets d'un même type `t` est :

```
t List
```

Exemple 1

```
#[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]
```

1.2. Comment construire une liste en Caml ?

Il existe deux manières principales de construire les listes.

- **Soit on connaît 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 :

  `[exp_1 ; exp 2 ; ... ; exp_n]`
- **Soit de manière évolutive.** Nous partons alors d'une liste existante (éventuellement **la liste vide** notée `[ ]` et appelée "nil") et nous lui ajoutons des éléments **en tête de liste** à l'aide du **constructeur** de liste noté `: :` et appelé "cons".

  Exemple 2

  ```
  #[] ; ;
  -  : 'a list = []    (* Noter que comme la liste est vide,
  Caml utilise le type polymorphe 'a *)
  #6 : :[1 ;2 ;3] ; ;
  -  : int list = [6 ; 1 ; 2 ; 3]
  #2 : :1 : :[] ; ;
  -  : int list = [2 ; 1]
  ```

Remarque : Nous disposons également d'une opération interne, la **concaténation** de liste noté . Dans la mesure du possible, on veillera toutefois à éviter de l'utiliser.

Exemple 3

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

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

2. Evaluation d'une liste

2.1. Type d'une liste

Comme nous l'avons évoqué précédemment, si nous devons évaluer la liste `[exp_1 ; exp_2 ; ... ; exp_n]`, les types des expressions `exp_i` sont évalués.

– S'ils ne sont pas tous égaux, il y a une erreur de type
– S'ils sont tous du même type `t`, la liste est de type `t list`

Exemple 4

```
# [1 ;1.0] ; ;
Error  : This expression has type float but an expression was expected of type int
```

2.2. Valeur d'une liste

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

```
[v1 ; v2  ; ...  ; vn]
```

2.3. 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 `[v1  ; v2  ; ...  ; vn]` alors le résultat a pour valeur `[v ;v1 ;v2 ;... ; vn]`.

2.4. 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 `[v1  ; v2  ; ...  ; vn]` et $L2$ a pour valeur `[w1  ; w2  ; ...  ; wp]` alors le résultat a pour valeur `[v1  ; v2  ; ...  ; vn  ; w1  ; w2  ; ...  ; wp]`.

Exercice . TD - Exercice A

Donner les résultats des évaluations suivantes, vérifier ensuite avec Caml.

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

Exercice . TD - Exercice B

Par quoi faut-il remplacer "?" pour obtenir les résultats souhaités ? Vérifier ensuite en Caml.

```
#[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]
```

3. Faire usage des listes en Caml

3.1. Sélection d'un élément dans une liste

Il existe deux fonctions prédéfinies dans la librairie `List` de OCaml permettant d'extraire (on dit sélectionner) des éléments d'une liste :

- **List.hd (head)** sélectionne le premier élément d'une liste non vide (la tête). Le résultat est un élément de type `t`.
- **List.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 provoquent une erreur.

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

Remarque : Ces fonctions peuvent être utiles au début pour mieux comprendre comment fonctionnent les listes. Mais en pratique, nous n'utiliserons pas ces fonctions, car nous préfèrerons le filtrage, voir paragraphe suivant.

3.2. Sélection d'un élément par filtrage

Important

Nous utiliserons plus généralement, un filtrage par motif sur la liste. Il est en effet possible d'utiliser le constructeur :: dans un filtre.

Exemple 5

Le filtre $a :: b :: q$ permet de filtrer toutes les listes s'écrivant un élément, un deuxième élément, puis une liste quelconque potentiellement vide. Ce filtre filtre donc toutes les listes d'au moins deux éléments, et nous pouvons facilement disposer des deux premiers éléments de la liste notés ici a et b.

Exercice . TP - Exercice 1 - Premières fonctions non récursives sur les listes

Écrire les fonctions suivantes :

1. `deuxieme : 'a list -> 'a` qui extrait le deuxième élément d'une liste.
2. `auMoinsTrois : 'a list -> bool` testant si une liste a au moins trois éléments.
3. `sommeTrois : int list -> int` qui fait la somme des trois premiers éléments d'une liste d'entiers
4. `troisEstPair : int list -> bool` testant si le troisième élément d'une liste est pair.
5. `ajoutDeuxFois : 'a * 'a list -> 'a list` qui ajoute deux fois un élément en tête d'une liste
6. `permute : 'a list -> 'a list` qui échange les deux premiers éléments d'une liste.

3.3. Liste et récursivité

Les listes se prêtent particulièrement bien à un traitement récursif.
En effet, l'ensemble des listes, ordonné partiellement par l'inclusion, est un ensemble bien ordonné d'élément minimal : la liste vide. Il est donc naturel de se placer dans le cas des hypothèses du théorème de terminaison, en ramenant un problème sur une liste, à une liste strictement incluse dans la première, obtenue bien souvent en lui enlevant le ou les premiers éléments.

Exercice . TP - Exercice 2

Écrire les fonctions suivantes :

1. `construitListe : int -> int list` qui, à un entier n, associe la liste $[n;n-1;\ldots;1;0]$.
2. `longueur : 'a list -> int` qui calcule la longueur d'une liste.
3. `dernier : 'a list -> 'a` qui donne le dernier élément d'une liste.
4. `somme : int list -> int` qui calcule la somme des éléments d'une liste donnée.
5. `taillePaire : 'a list -> bool` qui teste (sans utiliser la fonction longueur) si une liste possède un nombre pair d'éléments. **Réaliser cette fonction de deux**

manières différentes (avec donc deux idées récursives différentes)

6. `rangImpair : 'a list -> 'a list` permettant de construire la liste des éléments de rang impair d'une liste donnée.

Exercice . TP - Exercice 3

Écrire les fonctions suivantes :

1. `appartient : 'a * 'a list -> bool` qui teste si un élément appartient à une liste.
2. `maximum : 'a list -> 'a` qui calcule le maximum d'une liste d'entiers.
3. `occurrences : 'a * 'a list -> int` qui, étant donnés un élément et une liste, calcule le nombre d'occurrences de l'élément dans la liste.
4. `fois2 : int list -> int list` qui, étant donnée une liste d'entiers, construit la liste des doubles des entiers de la liste.
5. `insere : int * int list -> int list` qui insère un entier dans une liste d'entiers ordonnée dans l'ordre croissant.

Exercice . TP - Exercice 4

Écrire les fonctions suivantes :

1. `ieme : int * 'a list -> 'a`
2. `prendre : int * 'a list -> 'a list` qui, étant donnés un entier p et une liste l, retourne la liste des p premiers éléments de l.
3. `enleve : int * 'a list -> 'a list` qui, étant donnés un entier p et une liste l, retourne la liste privée des p premiers éléments de l.
4. `melange : 'a list * 'b list -> ('a * 'b) list` qui prend une paire de listes et retourne la liste des paires correspondantes. Si les deux listes n'ont pas la même longueur, on s'arrêtera à la liste la plus courte.

Exercice . TP - Exercice 5 - Suite de Fibonacci

Vous connaissez tous la suite de Fibonacci : $(1,1,2,3,5,8,13,...)$ qui commence par $1,1$ et dont chaque terme est la somme des deux précédents.

L'objectif est d'écrire une fonction `fibonacci : int -> int list` qui associe à l'entier n la liste des n premiers termes de la suite de Fibonacci.
Toutefois, commencez par écrire une fonction auxiliaire `fibo : int * int * int -> int list` qui prend en paramètre un triplet d'entiers (a,b,n) et qui renvoie la liste de Fibonacci commençant par a et b : $[a;b;a+b;\ldots]$ de longueur n.
En déduire, la fonction `fibonacci`. Arrangez votre programme pour que la fonction auxiliaire `fibo` soit définie localement.

```
Fibonacci 8 ;;
[1 ;1 ;2 ;3 ;5 ;8 ;13 ;21]
```

Exercice . TP - Exercice 6 - Fusion de listes croissantes

1. Écrire une fonction `estCroissante : 'a list -> bool` qui indique si les éléments consécutifs d'une liste sont bien ordonnés dans l'ordre croissant.
2. Écrire une fonction `fusion : 'a list * 'a list -> 'a list` qui, étant données deux listes supposées croissantes, renvoie la fusion de ces deux listes, toujours ordonnée croissante.

Exercice . TP - Exercice 7 - Nombres premiers par le crible d'Ératosthène

Nous souhaitons écrire un programme permettant d'obtenir une liste de couples de nombres premiers jumeaux, c'est à dire des couples (a,b) avec a et b premiers et $b-a=2$. par exemple, $(3,5)$ et $(5,7)$ sont des couples de nombres premiers jumeaux.

1. Écrire une fonction `generer : int -> int list` qui, étant donné un entier n, construit la liste ordonnée des entiers de 2 à n.
2. Écrire une fonction `eliminer : int * int list -> int list` qui, étant donnés une liste et un entier a, élimine tous les multiples de l'entier a dans la liste.
3. Écrire une fonction `eratos : int -> int list` qui, étant donné un nombre entier n, construit la liste des nombres premiers inférieurs ou égaux à n. Pour cela, on utilisera la méthode du crible d'Ératosthène : partant de la liste $[2;\ldots;n]$, son premier élément est premier mais on doit éliminer tous les multiples de cet élément qui eux ne le sont pas. On itère ce procédé jusqu'à la fin de la liste.
4. Écrire une fonction `jumeaux : int list -> (int * int) list` qui, étant donnée une liste de nombres premiers, construit la liste des couples de nombres premiers jumeaux qu'elle contient.
5. Écrire enfin une fonction `listeJumeaux : int -> (int * int) list` permettant de donner la liste des couples de nombres premiers jumeaux inférieurs à une limite n fixée.

Exercice . TP - Exercice 8 - Représentation des ensembles par des listes

On choisit de représenter un ensemble fini par la liste Caml de ses éléments. Cette liste ne contient aucune répétition. Ainsi l'ensemble $\{1,2,3\}$ est représenté par une liste contenant 1,2 et 3 dans un ordre quelconque. [1;2;3], [1;3;2], [2;3;1] ou [3;1;2] sont donc des représentations acceptables de $\{1,2,3\}$.

Écrire les fonctions définissant les opérations ensemblistes suivantes :

1. `appartient : 'a * 'a list -> bool` qui détermine si un élément appartient à un ensemble.
2. `union : 'a list * 'a list -> 'a list` et `intersection : 'a list * 'a list -> 'a list` qui calcule respectivement l'union et l'intersection de deux ensembles.
3. `inclus : 'a list * 'a list -> bool` qui détermine si un premier ensemble est inclus dans un second.
4. `disjoint : 'a list * 'a list -> bool` qui détermine si deux ensembles sont

disjoints.

5. `egaux : 'a list * 'a list -> bool` qui détermine si deux ensembles sont égaux.
6. `complement : 'a list * 'a list -> 'a list` qui calcule le complémentaire d'un premier ensemble dans un second.
7. `ensemble : 'a list -> 'a list` qui, à partir d'une liste quelconque, construit une liste contenant les mêmes éléments mais privés de leurs redondances
8. `parties : 'a list -> 'a list list` qui construit l'ensemble des parties d'un ensemble

Remarque. Certaines fonctions que nous avons écrites dans ces exercices préexistent en OCaml, dans la bibliothèque `List`, par exemple `appartient` peut s'obtenir avec la fonction `List.mem`. Toutefois, tant que nous n'avons pas vu le chapitre 10 Fonctionnelles, il n'est pas conseillé d'y avoir recours.