Caml : les listes

Une liste de longueur n = un n-uplet d'éléments du même type.
Exemples :

- `[99;1;-2;37]` est une `int list`
- `[true;false;false;true]` est une `bool list`
- `['p';'z';'Y']` est une `char list`
- `["pim";"pam";"poum"]` est une `string list`

D'autres exemples :

- [sin;cos;tan] est une (float -> float) list
- [2-3;4*5;6/7]
- [‘p‘;"abc".[1];char_of_int 33]
- [prefix <;prefix >;prefix =] est une ('a -> 'a -> bool) list
- [();();();();()] est une unit list

La liste vide est notée `[ ]`, sa longueur est nulle.

On peut construire des listes de listes ; exemples :

- `[[2;3];[ ];[9;-1]]` est une **`int list list`**
- `[[[ ]]]` est une **`'a list list list`**

Remarque : le type de base de cette «liste de listes de listes» ne peut être déterminé par Caml.

Plusieurs façons d'écrire une même liste :

- `[ 99;1;-2;37 ]`

- `99::1::(-2)::[ 37 ]`

- `99::1::[ -2;37 ]`

- `99::[ 1;-2;37 ]`

`::` se lit $\boxed{\text{quatre points}}$.

Fonctions de base sur les listes :

- `list_length` donne la longueur
- `hd` donne la tête (ou lève une exception si la liste est vide)
- `tl` donne la queue (ou lève une exception si la liste est vide)

Filtrage sur les listes : rédigeons `list_length`.

```
let rec list_length = function
 | [ ] -> 0
 | _::q -> 1 + list_length q ;;
```

Il est inutile de lier la tête de liste.

Les deux cas pourraient être permutés (ça irait peut-être un peu plus vite).

Exemples de motifs pour le filtrage des listes

- `[ ]` accepte la liste vide
- `[_]` accepte toute liste de longueur 1
- `[_;_]` et `_::[_]` acceptent toute liste de longueur 2
- `t::_` accepte toute liste non vide, et lie `t` à sa tête
- `_::_::q` accepte toute liste de longueur au moins 2, et lie `q` à la queue de sa queue

Rédigeons `hd` et `tl` :

```
let hd = function
 | t::_ -> t
 | [ ] -> failwith "hd" ;;

let tl = function
 | _::q -> q
 | [ ] -> failwith "tl" ;;
```

Exercice : rédigez une fonction **`dernier`** spécifiée comme suit : **`dernier z`** donne le dernier élément de la liste z si elle n'est pas vide, et lève une exception dans le cas contraire.

Analyse :

- combien de cas faut-il envisager ?

- quels sont les cas dans lesquels on peut conclure immédiatement ?

Solution :

```
let rec dernier = function
 | [ ] -> failwith "dernier"
 | [ x ] -> x
 | _::q -> dernier q ;;
```

Exercice : rédigez une fonction `avant_dernier` spécifiée comme suit : `avant_dernier z` donne l'avant-dernier élément de la liste z si sa longueur est au moins égale à 2, et lève une exception dans le cas contraire.

Analyse :

- combien de cas faut-il envisager ?

- quels sont les cas dans lesquels on peut conclure immédiatement ?

Solution :

```
let rec avant_dernier = function
 | [ ] | [_] -> failwith "dernier"
 | [x;_] -> x
 | _::q -> avant_dernier q ;;
```

Un classique : `sum` calcule la somme des éléments d'une liste d'entiers.

```
let rec sum = function
 | [ ] -> 0
 | t::q -> t + sum q ;;
```

Exercice : rédigez la fonction `prod`.

Solution :

```
let rec prod = function
 | [ ] -> 1
 | t::q -> t * prod q ;;
```

Construire une liste représentant l'intervalle discret $[\![a, b]\!]$.

```
let rec intervalle a b =
 if a>b then [ ] else a::(intervalle (a+1) b) ;;
```

Exercice : rédigez la fonction `factorielle`.

Deux solutions :

```
let rec factorielle = function
 | 0 -> 1
 | n -> n * (factorielle (n-1)) ;;

let factorielle n = prod (intervalle 1 n) ;;
```

Méditer : que se passe-t-il avec `factorielle (-1)` ?

Un autre classique : `map` applique une même fonction f à tous les éléments d'une liste.

```
let rec map f = function
 | [ ] -> [ ]
 | t::q -> (f t)::(map f q) ;;
```

Les (et) ne sont même pas nécessaires, car le découpage se fait d'abord autour de `::`. Mais la lisibilité s'en ressent.

Exemple : calcul de la somme des carrés des éléments d'une liste d'entiers.

```
let carré x = x * x ;;

let n2 v = sum (map carré v) ;;
```

Listes et prédicats : `for_all` et `exists`.

```
let rec for_all p = function
 | [ ] -> true
 | t::q -> (p t) & (for_all p q) ;;
```

Méditer : l'évaluation paresseuse des booléens.

Exercice : rédigez `exists`.

Deux solutions :

```
let rec exists p = function
 | [ ] -> false
 | t::q -> (p t) or (exists p q) ;;

let rec exists p = function
 | [ ] -> false
 | t::_ when p t -> true
 | _::q -> exists p q ;;
```

Méditer : ordre des motifs 2 et 3, dans la deuxième solution.

Une solution expéditive pour `exists` :

```
let exists p l =
 let non_p x = not(p x) in
 not(for_all non_p l) ;;
```

Explication :

$$\exists x : p(x) \iff \neg(\forall x : \neg p(x))$$

Listes et prédicats : `filtre`.

```
let rec filtre p = function
 | [ ] -> [ ]
 | t::q when p t -> t::(filtre p q)
 | _::q -> filtre p q ;;
```

`filtre` construit une nouvelle liste, dans laquelle ne figurent que les éléments qui satisfont le prédicat.

Remarque : le nom `filtre` n'est peut-être pas bien choisi (risque de confusion avec le filtrage offert par Caml). On peut préférer `tamis`.

Test d'appartenance : `mem`.

```
let rec mem x = function
 | [ ] -> false
 | t::q -> t=x or mem x q ;;
```

Méditer : on ne peut utiliser le motif `x::q`.

Test de présence de doublons : dans une liste, un même élément peut apparaître plusieurs fois.

Rédigez une fonction

```
sans_doublon : 'a list -> bool
```

spécifiée comme suit : **`sans_doublon z`** indique si la liste z est sans doublon.

Solution proposée :

```
let rec sans_doublon = function
 | [ ] -> true
 | t::q when mem t q -> false
 | _::q -> sans_doublon q ;;
```

Remarque : le coût est en $\mathcal{O}(|z|^2)$. Variante avec deux cas, sans garde, et un booléen paresseux :

```
let rec sans_doublon = function
 | [ ] -> true
 | t::q -> not(mem t q) & sans_doublon q ;;
```

Utile : enlever les doublons. Rédigez une fonction

```
uniq : 'a list -> 'a list
```

spécifiée comme suit : `uniq z` construit une liste dans laquelle chaque élément de z apparaît une fois et une seule.

Solution :

```
let rec uniq = function
 | [ ] -> [ ]
 | t::q when mem t q -> uniq q
 | t::q -> t::(uniq q) ;;
```

Remarque : ici encore, le coût est en $\mathcal{O}(|z|^2)$. Pour chaque ≪paquet≫ d'éléments identiques, on ne garde que le dernier.

Intersection de deux listes : rédigez une fonction

```
intersection : 'a list -> 'a list -> 'a list
```

qui construit la liste (sans doublons) des éléments appartenant simultanément à deux listes données.

Solution : on dédoublonne u, puis on ne garde de u que les éléments qui appartiennent à v.

```
let intersection u v =
 let p x = mem x v and u' = uniq u
 in filtre p u' ;;
```

Le coût est un $\mathcal{O}(|u|^2 + |u'| \times |v|)$. En permutant au besoin les deux arguments, on peut se placer dans la situation favorable, où $|u| < |v|$.

Solution plus efficace : on inverse l'ordre des deux actions.

```
let intersection u v =
 let u' = filtre (function x -> mem x v) u
 in uniq u' ;;
```

Le coût est un $\mathcal{O}(|u| \times |v| + |u'|^2)$. La liste à laquelle on applique `uniq` ne sera pas plus longue que la plus courte des deux listes ; et on peut espérer qu'elle sera nettement plus courte. . .

Plus subtil : `répète`.

```
let rec répète = function
 | [ ] -> [ ]
 | t::q -> t::t::(répète q) ;;
```

Remarque : en général, on met en premier le motif `[ ]` (habitude venue des mathématiques : dans une récurrence, on regarde d'abord le cas initial).

Exercice : rédigez `permute` qui, appliquée à une liste $(x_1, x_2, \dots, x_n)$ rend la liste $(x_2, x_1, x_4, x_3, \dots)$: chaque élément d'indice pair est échangé avec celui qui le précède.

Analyser la situation : combien de cas de filtrages ?

Solution :

```
let rec permute = function
 | [ ] -> [ ]
 | [x] -> [x]
 | x::y::q -> y::x::(permute q) ;;
```

Exercice : rédigez `sans_doublet` qui, appliquée à une liste z rend `true` ssi il n'existe pas deux éléments consécutifs égaux :

$$z_i \neq z_{i+1} \qquad \text{pour } 0 \leqslant i < |z| - 1$$

Solution :

```
let rec sans_doublet = function
 | [ ] | [ _ ] -> true
 | x::y::q when x = y -> false
 | _::q -> sans_doublet q ;;
```

La suite auto-descriptive (Conway) :

$$\begin{array}{llll} u_0 = 1 & u_1 = 11 & u_2 = 21 & u_3 = 1211 \\ u_4 = 111221 & u_5 = 312211 & u_6 = 13112221 & u_7 = 1113213211 \end{array}$$

Chaque terme est obtenu en «épelant» le terme précédent.

Question : le chiffre 4 apparaît-il dans un terme de cette suite ?

Exercice : rédigez `épèle` qui, appliquée à une liste $(x_1, x_2, \ldots, x_n)$ rend l'épellation (?) de cette suite, si elle ne contient pas quatre termes consécutifs égaux ; dans le cas contraire, une exception sera levée.

Solution :

```
let rec épèle = function
 | [ ] -> [ ]
 | x::y::z::t::q when x=y & y=z & z=t -> failwith "épèle"
 | x::y::z::q when x=y & y=z -> 3::x::(épèle q)
 | x::y::q when x=y -> 2::x::(épèle q)
 | x::q -> 1::x::(épèle q) ;;
```

Exercice : rédigez `conway`, de type `int -> int list` qui, appliquée à n, calcule le terme d'indice n de la suite de Conway.

Solution : `itère n f x` permet de calculer $f^n(x)$.

```
let rec itère n f x =
 if n = 0 then x else itère (n-1) f (f x) ;;

let conway n = itère n épèle [1] ;;
```