LISTES

En informatique, une *structure de données* est la description d'une structure logique destinée à organiser et à agir sur des données, indépendamment de l'implémentation effective de cette structure (qui sera liée au langage utilisé). Les premières structures de données que nous étudierons sont les *listes simplement chaînées* (ou plus simplement, les *listes*).

Avant de commencer, un avertissement :

```
<b>Les "listes" Python ne sont pas des listes</b> (mais des tableaux dynamiques).
```

1 Définition et première mise en œuvre

Une *liste* est une structure de données immuable, contenant des données de même type, et obtenue à partir des opérations de construction suivantes :

- la création d'une liste vide, appelée **nil** ;
- l'ajout d'un élément t en tête d'une liste q, parfois noté **cons** (t, q).

Lorsqu'une liste est non vide, elle est alors de la forme **cons** (t, q), t est appelé la *tête* de la liste, et t sa *queue* (ou *reste*).

Pour manipuler les listes, on dispose des trois opérations suivantes :

- une opération testant si une liste est vide ou non ;
- un *asssesseur* permettant d'obtenir la tête d'une liste non vide ;
- un *asssesseur* permettant d'obtenir la queue d'une liste non vide.

Donnons une première implémentation en OCaml du type liste, à l'aide d'un type récursif et polymorphe `'a liste` :

```
type 'a liste =
  | Nil
  | Cons of 'a * ('a liste)
;;
```

```
type 'a liste = Nil | Cons of 'a * 'a liste
```

La liste `lst` contenant les entiers 4, 1 et 3 sera alors définie par :

```
let lst = Cons (4, Cons (1, Cons (3, Nil)));;
```

```
val lst : int liste = Cons (4, Cons (1, Cons (3, Nil)))
```

Les trois opérations d'accès sur les listes peuvent être décrites par les fonctions suivantes :

```
let est_vide l =
  match l with
  | Nil -> true
  | _ -> false
;;

let tete l =
  match l with
  | Cons (t, _) -> t
  | _ -> failwith "Liste vide"
;;
```

```
let queue l =
  match l with
  | Cons (_, q) -> q
  | _ -> failwith "Liste vide"
;;
```

Nous n'allons pas poursuivre avec notre type `'a liste`, car le type `'a list` existe déjà en OCaml.

2 La construction de liste en OCaml

En Caml, la liste vide se note `[]`, et **cons** (*t*,*q*) se note `t::q`.

L'opérateur `::` est associatif à droite : `x::y::z` désigne `x::(y::z)`.

```
[];;
```

```
- : 'a list = []
```

```
fun x y -> x::y;;
```

```
- : 'a -> 'a list -> 'a list = <fun>
```

```
fun x y z -> x::y::z;;
```

```
- : 'a -> 'a -> 'a list -> 'a list = <fun>
```

```
let l1 = 5::[];;
```

```
val l1 : int list = [5]
```

```
let l2 = 4::l1;;
```

```
val l2 : int list = [4; 5]
```

```
let l3 = "toto"::l2;;
```

```
      File "[9]", line 1, characters 17-19:
  1 | let l3 = "toto"::l2;;
                       ^^
  Error: This expression has type int list
         but an expression was expected of type string list
         Type int is not compatible with type string
```

On remarque :

- qu'il n'est pas possible de construire une liste avec une tête de type `string` et une queue de type `int list` ; cela est cohérent avec le fait que les éléments d'une liste doivent être de même type ;
- que OCaml utilise une notation plus agréable pour afficher les listes : plutôt que d'afficher `4::5::[]`, OCaml affiche `[4; 5]`

Cette notation est aussi acceptée en entrée :

```
let l3 = [4; 5];;
```

```
val l3 : int list = [4; 5]
```

```
l2 = l3;;
```

```
- : bool = true
```

La liste reste néanmoins construite de la même manière.

3 Filtrage

On dispose de motifs de filtrage adaptés aux listes :

- le motif `[]` filtre la liste vide ;
- le motif `t::q` filtre toute liste non vide ; dans la suite de l'évaluation, `t` prendra la valeur de la tête de la liste et `q` celle de la queue.

Il est facile à titre d'exemple de définir les fonctions `tete` et `queue` :

```
let tete l =
  match l with
  | [] -> failwith "Liste vide"
  | t::q -> t
;;
```

```
val tete : 'a list -> 'a = <fun>
```

```
let queue l =
  match l with
  | [] -> failwith "Liste vide"
  | t::q -> q
;;
```

```
val queue : 'a list -> 'a list = <fun>
```

Rappelons les motifs rencontrés jusqu'à présent : un motif est une des formes suivantes :

- `_`
- constante flottante, entière, booléenne
- variable
- `[]`
- *motif* `::` *motif*
- *motif*, . . . , *motif*
- `Constructeur` *motif*

On dit qu'un motif *filtre* une valeur v lorsque, en remplaçant toutes les variables du motif et chaque occurrence de _ par des valeurs bien choisies, on obtient v.

Remarque : OCaml refuse tout motif dans lequel une même variable apparaît plus d'une fois.

Exercices divers

Exercice 1

- Décrire en français courant les listes reconnues par les motifs suivants : `[x]` ; `x::[]` ; `x::2::[]` ; `[1; 2; x]` ; `1::2::x` ; `x::y::z`

- `_::0::1::_` est-il un motif ? Si oui, décrire en français les listes reconnues.

Exercice 2

- Écrire une fonction testant si une liste est non vide.
- Écrire une fonction testant si une liste a exactement deux éléments.
- Écrire une fonction testant si une liste possède deux éléments ou moins.
- Écrire une fonction testant si le premier élément d'une liste de booléens vaut `true`.
- Écrire une fonction testant si le premier élément d'une liste de booléens vaut `false` et le deuxième vaut `true`.
- Écrire une fonction renvoyant l'avant-dernier élément d'une liste, s'il existe.

Exercice 3

- Écrire une fonction calculant la liste des carrés des éléments d'une liste d'entiers.
- Écrire une fonction sommant les éléments d'une liste d'entiers.
- Écrire une fonction admettant un entier n comme argument et qui renvoie la liste des entiers de 1 à n (et la liste vide si $n = 0$).

Exercice 4

- Écrire une fonction `length : 'a list -> int`, qui prend en argument une liste et qui renvoie le nombre d'éléments de la liste. Quelle est sa complexité ?
- Écrire une fonction `mem : 'a -> 'a list -> bool` testant l'appartenance d'un élément à une liste. Quelle est sa complexité ?
- Écrire une fonction `map : ('a -> 'b) -> 'a list -> 'b list`, qui prend en argument une fonction `f` de type `'a -> 'b` et une liste `[a1; ...; an]` d'éléments de type `'a` et qui renvoie la liste `[f a1; ...; f an]`
- Écrire une fonction `filter : ('a -> bool) -> 'a list -> 'a list` qui prend en argument une fonction `f : 'a -> bool` et une liste `l : 'a liste` et qui renvoie la liste des éléments `x` de `l` tels que `f x` soit vrai.

Ces quatre fonctions sont en fait déjà implémentées dans le module `List`.

Exercice 5

Écrire une fonction calculant la concaténation de deux listes. Quelle est sa complexité ?

Exercice 6

Écrire une fonction qui prend en argument une liste d'entiers de longueur au moins 2 et qui retourne le couple constitué des deux plus petits entiers de la liste (éventuellement égaux).

Exercice 7

On représente un polynôme à coefficients entiers par la liste de ses coefficients en puissances décroissantes.

Écrire une fonction `evalue : int list -> int -> int` qui prend en argument un polynôme $[a_n; a_{n-1}; \dots; a_0]$ et un entier x et qui retourne l'entier $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$ avec seulement n multiplications.

Exercice 7

Écrire une fonction qui retourne le nombre de changements de signes d'une suite d'entiers (les zéros ne comptent pas et la fonction retournera 0 si la liste est vide).