III - Listes

26 janvier 2024

En informatique, une *structure de données* est la description d'un ensemble organisé d'objets, et des opérations qui lui sont associées. On distingue :

— Le type de données abstrait, qui correspond à la description mathématique de la structure de données et de ses opérations. Il ne dépend pas du langage de programmation utilisé.
— L'implémentation, qui correspond à sa réalisation technique, et dépend du langage de programmation.

Les opérations usuelles des structures de données sont la *création*, la *consultation* est la *modification* (ajout/suppression).

Lors de la création d'une structure de données, on prend en compte différents éléments : - L'espace mémoire utilisé ; si l'espace alloué est de taille fixe, on dit que la structure de donnée est *statique*. Sinon, on dit qu'elle est *dynamique* ; - La complexité temporelle des différentes opérations ; - les opérations de modifications autorisées : si le contenu est modifiable, la structure est dite *mutable*. Une structure statique et non mutable est dite *persistante* : on doit créer une nouvelle instance de la structure si on souhaite la modifier. Cela offre notamment la possibilité de garder en mémoire les différentes modifications effectuées. Dans le cas contraire, on parle de structure *impérative*.

La première structure de données que nous étudierons est celle des *listes simplement chaînées* (ou plus simplement, les *listes*).

Avant de commencer, un avertissement :

Les « listes » Python ne sont pas des listes, mais des tableaux dynamiques. Elles sont dynamiques et mutables. Les liste chaînées ne sont pas mutables. Les tableaux que nous étudierons plus tard OCaml, sont quant à eux mutables mais pas 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).

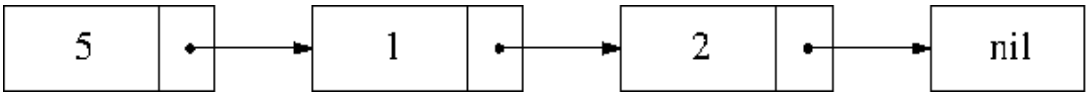
Lorsqu'une liste est non vide, elle est alors de la forme **cons** (t, q), t est appelé la *tête* de la liste, et q sa *queue* (ou *reste*).

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.

Plus précisément, cette structure est appelée *liste chaînée* : chaque élément de la liste contient en fait une donnée (la *valeur* de l'élément) mais aussi un pointeur, qui indique quel est l'élément suivant dans la liste. Une liste n'étant pas infinie, le dernier élément pointé est **nil**, ce qui indiqe la fin de la liste. Voici par exemple une représentation de `cons(5,cons(1,cons(2,nil)))` :



Il n'est possible d'accéder directement qu'à un seul élément d'une liste chaînée : sa tête.

Pour démontrer qu'un résultat est vrai pour toute liste, on prouvera par induction qu'il est vrai pour la liste vide, et que s'il est vrai sur une liste, il le reste par adjonction d'un nouvel élément en tête de cette liste.

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

[1] :
```
type 'a liste =
  | Nil
  | Cons of 'a * ('a liste)
 ;;
```

[1] :
```
type 'a liste = Nil | Cons of 'a * 'a liste
```

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

[2] :
```
let lst = Cons (4, Cons (1, Cons (3, Nil))) ;;
```

[2] :
```
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 :

[3] :
```
let est_vide l =
  l = Nil
 ;;

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

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

[3] :
```
val est_vide  : 'a liste -> bool = <fun>
val tete  : 'a liste -> 'a = <fun>
val queue  : 'a liste -> 'a liste = <fun>
```

```
[4] : queue lst ; ;
```

```
[4] : -  : int liste = Cons (1, Cons (3, Nil))
```

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

2 La construction de listes 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)`.

```
[5] : [] ; ;
```

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

```
[6] : fun x y -> x : :y ; ;
```

```
[6] : -  : 'a -> 'a list -> 'a list = <fun>
```

```
[7] : fun x y z -> x : :y : :z ; ;
```

```
[7] : -  : 'a -> 'a -> 'a list -> 'a list = <fun>
```

```
[8] : let lst1 = 5 : :[] ; ;
```

```
[8] : val lst1  : int list = [5]
```

```
[9] : let lst2 = 4 : :lst1 ; ;
```

```
[9] : val lst2  : int list = [4 ; 5]
```

```
[10] : let lst3 = "toto" : :lst2 ; ;
```

```
Line 1, characters 19-23 :
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 les opérations permettant de renvoyer la tête ou la queue d'une liste se font en temps constant. C'est aussi le cas de l'opération `t : :q` : elle ne recopie pas une liste de n élements dans une autre plus grande, elle ne fait que créér un nouvel élément, dont le pointeur indique la tête de `q` ;
- 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 :

```
[11] : let lst3 = [4 ; 5] ; ;
```

```
[11] : val lst3 : int list = [4 ; 5]
```

```
[12] : lst2 = lst3 ; ;
```

```
[12] : - : 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` :

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

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

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

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

```
[15] : queue [4 ; 6 ; 7 ; 9]
```

```
[15] : - : int list = [6 ; 7 ; 9]
```

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

- `-`, qui filtre tout sans qu'on puisse l'utiliser
- 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.

[16] :
```
let sont_egaux x y =
    match x, y with
    | n, n -> true
    | _ -> false
 ;;
```

```
Line 3, characters 9-10 :
Error : Variable n is bound several times in this matching
```

4 Exercices divers

4.1 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` i.e `x : :(y : :z)`

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

4.2 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.

4.3 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 $n = 0$).

Première méthode : Ecrire une fonction qui calcule la liste des entiers de n à 1 et une fonction qui renverse une liste

Deuxième méthode : Ecrire une fonction à deux paramètres p et n qui calcule la liste des entiers de p à n, et l'utiliser pour p=1.

Troisième méthode : Écrire une fonction qui prend en argument une liste et qui renvoie la liste précédée du prédécesseur de son premier élément si celui-ci ne vaut pas 1.

4.4 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`.

4.5 Exercice 5

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

4.6 Exercice 6

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

4.7 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 $[a_n; a_{n-1}; \ldots; a_0]$ 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.

4.8 Exercice 8

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).