LANGAGE OCaml – TP 4
Récursivité – 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.

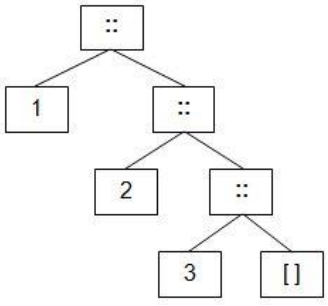
Listes en Caml

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 ;.

`let liste_0 = 1 :: 2 :: 3 :: [];;`	Sortie écran et commentaires : La création de cette liste peut être représentée par l'arbre suivant : 
`let liste_1 = [1;2;3];;`	Sortie écran et commentaires :

`let liste_2 = [1,2,3];;`	Sortie écran et commentaires :
`let liste_3 = 1 :: 'a' ::(1,2,3) [];;`	Sortie écran et commentaires :

`let liste_4 = [];;`
`(* Liste polymorphe à la création *)`
 	
`2 :: liste_4;;`
`2. :: liste_4;;`
`liste_4;;`
`'a' :: liste_4;;`
`liste_4 ;;`
	Sortie écran et commentaires :

Accesseur

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.

`let liste = [5;4;3;2;1];;`
`let a = List.hd liste;;`
`liste;;`
	Commentaires et sortie écran :

`let l = [];;` `let b = List.hd l;;`	Commentaires et sortie écran :
`let t = List.tl liste;;`	Commentaires et sortie écran :

`let liste = [5;4;6;2;1;3];;`
`let a = List.nth liste 4;;`
`let b = List.nth liste 7;;`
	Même si cette fonction existe, il est très très fortement déconseillé de l'utiliser.
	
	Il n'est pas dans l'esprit de la manipulation de liste d'aller chercher le nième composant.
	
	Par ailleurs, cette fonction présente un coût temporel non négligeable.

`let liste_5 =` `[[1;2];[1;2;3];[1;2;3;4;5;6]];;`	Commentaires et sortie écran :

Transformateur : ajout en tête de liste

`let liste_6 = [];;`
`let liste_6 = 5 :: liste_6;;`
`let liste_6 = 2 :: liste_6;;`
`liste_6;;`
	Sortie écran et commentaires :

Transformateur : concaténation de deux listes (à connaître)

`let l1 = [1;2;3];;`
`let l2 = [4;5;6];;`
`let l3 = [7;8;9];;`
`let l = List.append l2 l1;;`
	Sortie écran et commentaires :

`let ll = l @ l3;;`	Sortie écran et commentaires :

Autres opérations du module List

`let l_1 = List.map (function x -> x*x)[1;2;3;4];;`	Sortie écran et commentaires : (fonction mentionnée dans le programme mais pas à connaître)
`let a = List.length [1;2;3;4];;`	Sortie écran et commentaires : (fonction mentionnée dans le programme et à connaître)

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 :

`let rec somme liste =` `  match liste with` `	[] -> 0` `	x :: r -> x + somme r ;;`  `let liste = [1;2;3];;` `let s = somme liste;;`	Sortie écran :

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. Proposer une version récursive non terminale et une version récursive terminale pour cette fonction.

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.

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).

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]]
```

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].`

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.