# TD1 : Listes

## 1 Structure de données : liste

▷ **Exercice 1**

1. *Écrire la fonction* `deuxieme` *qui renvoie le deuxième élément d'une liste. (Ici insister sur les contraintes dans le contrat : le paramètre doit être une liste d'au moins deux éléments).*
2. *Écrire les fonctions* `n_a_zero` *et* `zero_a_n`*, telles que :*
   `n_a_zero n = [n; n-1;...; 1  ; 0]`
   `zero_a_n n = [0; 1  ;...; n-1; n]`
   *Attention : utilisation de* `append` *interdite.*
3. *Liste des indices/positions d'un élément e dans une liste l.*

▷ **Exercice 2**

*Utilisation des itérateurs obligatoire.*

1. *Écrire* `map`*, qui applique une fonction donnée à tous les éléments d'une liste, i.e.*
   `map f [a; b; c] = [f a; f b; f c]`.
2. *Écrire* `flatten` *(aplatissement d'une liste de listes).*
3. *Écrire une fonction* `fsts` *qui prend une liste de couples et renvoie la liste des premiers éléments.*
4. *Écrire une fonction* `split` *telle que :* `split [(a1,b1);...;(an,bn)]=([a1;...;an],[b1;...;bn])`.
5. *Écrire une fonction qui supprime les doublons d'une liste.*

## 2 Modules, application aux files

### 2.1 Type abstrait

Dans une spécification, on trouve plutôt des déclarations de types que des définitions. On parle alors de **types privés** ou **types abstraits**. L'absence de définition de types interdit à l'utilisateur de manipuler les valeurs "à la main" et n'importe comment, l'obligeant donc à employer les fonctions de manipulation fournies.

Si, en plus des déclarations de types et de fonctions, on trouve des équations qui spécifient la sémantique de ces fonctions (sans faire apparaître une implantation particulière), on parle alors de **types abstraits algébriques**. Bien sûr, ces équations ne sont pas supportées par les langages de programmation.

### 2.2 Un exemple : les ensembles

#### 2.2.1 Spécification : Les ensembles

```
1  (***** le type abstrait = pas de definition *****)
2  type 'a set ;;
3
4  (***** les constructeurs abstraits *****)
5     (* Fonction vide : construit l'ensemble vide *)
6  val vide          : unit -> 'a set ;;
7     (* Fonction singleton : constuit un ensemble a un element *)
8     (* Parametre : a l'element *)
9     (* Retour : l'ensemble {a} *)
10 val singleton     : 'a -> 'a set ;;
11    (* Fonction union : realise l'union de deux ensembles *)
```

```
12    (* Parametres : ens1 et ens2 deux ensembles              *)
13    (* Retour : l'union ensembliste de ens1 et ens2          *)
14  val union          : 'a set -> 'a set -> 'a set;;
15
16  (***** les accesseurs ou requetes ou selecteurs          *****)
17    (* Fonction est_vide : teste si un ensemble est vide     *)
18  val est_vide       : 'a set -> bool;;
19    (* Fonction choix : choisi un element de l'ensemble      *)
20    (* Precondition : l'ensemble ne doit pas etre vide       *)
21  val choix          : 'a set -> 'a;;
22    (* Fontion appartient :
23       teste si un element est dans l'ensemble               *)
24  val appartient     : 'a -> 'a set -> bool;;
25
26  (***** les destructeurs                                   *****)
27    (* Fonction enleve : retire un element a l'ensemble      *)
28    (* Parametre e : l'element a enlever                     *)
29    (* Parametre ens : l'ensemble                            *)
30    (* Retour : ens\{e}                                      *)
31  val enleve         : 'a -> 'a set -> 'a set;;
32    (* Fonction intersection : realise l'intersection de *****)
33    (* deux ensembles                                        *)
34    (* Parametres : ens1 et ens2 deux ensembles              *)
35    (* Retour : l'intersection ensembliste de ens1 et ens2   *)
36  val intersection : 'a set -> 'a set -> 'a set;;
```

Listing 1 – ensemble.mli

Les constructeurs “abstraits” jouent exactement le même rôle que de véritables constructeurs, sauf qu'ils ne montrent rien de l'implantation (ce sont juste des fonctions).

S'il fallait en faire un TAA, on aurait par exemple les propriétés suivantes :

```
est_vide (vide ()) = true
not (appartient x (vide ())) = true
appartient x (singleton y) = (x = y)
appartient x (union ens ens') = appartient x ens || appartient x ens'
...
```

### 2.2.2 Une implantation POSSIBLE : Les listes

```
1   (* definition du type 'a set *)
2   type 'a set = 'a list ;;
3
4   (* les constructeurs abstraits *)
5   let vide () = [];;
6
7   let singleton e = [e];;
8
9   let rec union ens1 ens2 =
10    List.fold_right (fun t tq -> if (List.mem t ens2) then tq else t::tq) ens1 ens2 ;;
11
12  (* les accesseurs ou requetes ou selecteurs *)
13  let est_vide ens = (ens=[]);;
14
15  let choix ens =
16    match ens with
17    | [] -> failwith "Ensemble vide!";
18    | t::q ->t;;
19
20  let appartient = List.mem ;;
21
22  (* les destructeurs *)
```

```
23 let rec enleve e ens =
24   match ens with
25   | [] -> []
26   |t::q -> if (e=t) then q else t::(enleve e q);;
27
28 let rec intersection ens1 ens2 =
29   List.fold_right (fun t tq -> if (appartient t ens2) then t::tq else tq) ens1 [] ;;
```

Listing 2 – ensemble.ml

## 2.3 Un exemple de TAA : Le module file

La file est une structure de données linéaire qu'on retrouve partout. On peut enfiler des éléments à une extrémité et les retirer à l'autre extrémité, comme un pipe-line où circulerait un "flot" de valeurs. La file est une structure de type FIFO (*first in, first out*).

▷ **Exercice 3**

— *Proposer une spécification des opérations du TAA file.*
— *Proposer une implantation fonctionnelle simple à base de listes.*
— *Évaluer la complexité des opérations.*

## 2.4 La complexité amortie

On a déjà vu la complexité moyenne et du pire cas, qui s'intéressent seulement à une exécution donnée d'un algorithme. La complexité amortie permet de lisser les temps de calcul des différents appels à un algorithme, supposé inséré dans une application plus grosse. C'est une moyenne temporelle sur toutes ses exécutions. Cette mesure est donc plus fiable quant au temps réel passé dans un algorithme donné. Cela permet de prouver par exemple que quelques exécutions de mauvaise complexité (du pire cas) peuvent être compensées à la longue par d'autres exécutions plus favorables.

## 2.5 Le TAA file revisité

▷ **Exercice 4**

— *Trouver une implantation efficace du TAA file à l'aide de cette représentation.*
— *Étudier la complexité des opérations.*