Les types récursifs

On utilise les constructeurs pour définir des types récursifs.

```
# type entier = Zero | Succ of entier;;
type entier = Zero | Succ of entier

# let succ x = Succ x;;
val succ : entier -> entier = <fun>
# let pred = function
     Zero -> Zero
 | Succ x -> x;;
val pred : entier -> entier = <fun>

# let rec add x y =
    match x with
       Zero -> y
   | Succ z -> Succ (add z y);;
val add : entier -> entier -> entier = <fun>

# let rec entier_fold_right f e a =
   match e with
       Zero -> a
   | Succ x -> f (entier_fold_right f x a);;
val entier_fold_right:('a ->'a ->entier->'a->'a=<fun>
```

On peut redéfinir le type liste d'entiers avec

```
# type liste_entiers =
   Nil | Cons of entier * liste_entiers;;
type liste_entiers =
     Nil | Cons of entier * liste_entiers
```


Dans cet exemple, `Nil` est un constructeur d'arité 0 et `Cons` un constructeur d'arité 2.

```
# let rec somme = function
   Nil -> Zero
| Cons(x,l) -> add x (somme l);;
val somme : liste_entiers -> entier = <fun>
```

On peut également recréer le type polymorphe des listes en définissant le type paramétré suivant.

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

# let rec liste_fold_right f l b =
     match l with
  Nil -> b
| Cons(x,reste) -> f x (liste_fold_right f reste b);;
# let rec liste_fold_right f l b =
     match l with
  Nil -> b
| Cons(x,reste) -> f x (liste_fold_right f reste b);;
val liste_fold_right :
    ('a -> 'b -> 'b) -> 'a liste -> 'b -> 'b = <fun>
```

On peut utiliser les types paramétrés pour créer par exemple des listes pouvant contenir des valeurs de types différents.

```
# type ('a,'b) liste =
   Nil
 | Acons of 'a * ('a,'b) liste
 | Bcons of 'b * ('a,'b) liste;;
type ('a, 'b) liste =
    Nil
  | Acons of 'a * ('a, 'b) liste
  | Bcons of 'b * ('a, 'b) liste
```


Définir des arbres binaires

La définition d'arbres binaires représentant des noeuds dont les étiquettes sont d'un type quelconque (mais toutes du même type) peut se faire par.

```
# type 'a arbre =
    Vide | Noeud of 'a * ('a arbre) * ('a arbre);;
type
'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
# Noeud (1, Noeud(2,Vide,Vide), Vide);;
- : int arbre = Noeud(1,Noeud(2,Vide,Vide),Vide)
```

Le constructeur `Vide` est un constructeur d'arité 0 et le constructeur `Noeud` est un constructeur d'arité 3.

```
# let rec somme = function
   Vide -> 0
 | Noeud(x,t1,t2) ->
     x + (somme t1) + (somme t2);;
val somme : int arbre -> int = <fun>

# let rec nbe = function
   Vide -> 0
 | Noeud(x,t1,t2) ->
       1 + (nbe t1) + (nbe t2);;
val nbe : 'a arbre -> int = <fun>
```

La fonction `flat` suivante donne la projection verticale d'un arbre ou l'ordre interne ou encore infixe.

```
# let rec flat = function
  Vide -> []
| Noeud(x,t1,t2) -> (flat t1)@(x::flat t2);;
val flat : 'a arbre -> 'a list = <fun>
```

On peut écrire un `flat` de meilleure complexité à l'aide d'une variable appelée accumulateur.

```
# let rec flat_accu t accu = match t with
   Vide -> accu
 | Noeud(x,t1,t2) ->
       flat_accu t1  (x::flat_accu t2 accu);;
val flat_accu : 'a arbre -> 'a list -> 'a list = <fun>
# let flat t = flat_accu t [];;
val flat : 'a arbre -> 'a list = <fun>
```

Le `flat` se fait alors en temps linéaire.

Fonctions d'itération sur les arbres

Elles sont semblables aux fonctionnelles d'itérations sur les listes mais ne sont pas prédéfinies. La fonction suivante est l'équivalent d'un `fold_right`.

```
# let rec arbre_it f t b =
 match t with
   Vide -> b
 | Noeud(x,t1,t2) ->
     f x (arbre_it f t1 b) (arbre_it f t2 b);;
val arbre_it :
 ('a ->'b ->'b ->'b) ->'a arbre ->'b ->'b=<fun>
```


Exemples d'utilisation de la fonctionnelle d'itération.

```
# let nbe t = arbre_it (fun x n1 n2 -> 1+n1+n2) t 0;;
val nbe : 'a arbre -> int = <fun>
# let ht t =
   arbre_it (fun x h1 h2 -> 1+(max h1 h2)) t 0;;
val ht : 'a arbre -> int = <fun>
# let flat t =
   arbre_it (fun x l1 l2 -> l1@(x::l2)) t [];;
val flat : 'a arbre -> 'a list = <fun>
```

Syntaxes abstraites

On définir des expressions arithmétiques, des formules logiques, ..., de la façon suivante.

```
# type expr = Const of int
            | Plus of expr * expr
            | Mult of expr * expr
            | Moins of expr * expr
            | Div of expr * expr
            | Opp of expr;;
type expr =
    Const of int
  | Plus of expr * expr
  | Mult of expr * expr
  | Moins of expr * expr
  | Div of expr * expr
  | Opp of expr
```

```
# let rec eval = function
    Const n -> n
 |  Plus (e1,e2) -> (eval e1)+(eval e2)
 |  Mult (e1,e2) -> (eval e1)*(eval e2)
 |  Moins(e1,e2) -> (eval e1)-(eval e2)
 |  Div (e1,e2)  -> (eval e1)/(eval e2)
 |  Opp e -> -(eval e);;
val eval : expr -> int = <fun>
```