Travaux Pratiques sur Caml Light n°3

Cours d'Informatique de l'École Centrale d'Électronique

—Ingénieur Deuxième Année—

Ce TP traite des différents points suivants :

- le *pattern matching* ;
- la définition de types énumérés ;
- les types récursifs ;
- les types polymorphes ;
- les fonctionnelles d'itération.

▸ **Exercice 1.** *On souhaite réaliser un programme simulant un jeu de cartes. Pour cela, on a besoin de définir un type servant à représenter l'ensemble des cartes. On choisit la représentation suivante :*

```
#type suit = Heart | Diamond | Club | Spade;;
Type suit defined.
#type card = Ace of suit
           | King of suit
           | Queen of suit
           | Jack of suit
           | Plain of suit*int;;
Type card defined.
```

1. *Définissez un type* `color` *représentant les deux couleurs* `Red` *et* `Black`*.*
2. *Donnez une fonction* `color_of_suit` *qui associe à un objet de type* `suit` *sa couleur.*
3. *En déduire une fonction* `color_of_card` *qui retourne la couleur d'une carte.*
4. *On souhaite désormais compter les points dans une partie de belote. Donnez une fonction* `points_of_card` *prenant en argument un objet de type* `suit` *et retournant la valeur d'une carte passée en deuxième argument. Les règles sont les suivantes : un as vaut* 11 *points, un roi* 4*, une reine* 3*, un valet* 20 *ou* 2 *points selon que sa couleur* (`suit`) *correspond ou non au premier argument de la fonction (couleur de l'atout), un* 10 *vaut* 10 *points, un* 9 *vaut* 14 *ou* 0 *selon que sa couleur* (`suit`) *correspond ou non au premier argument de la fonction et toutes les autres cartes (jusqu'au* 7*) ne valent rien.*

On souhaite désormais redéfinir notre propre type "liste d'objets de type α". Pour cela, on définit le type `mylist` comme un type paramétré par `'a` et qui plus est récursif :

```
#type 'a mylist = Nil | Cons of 'a * 'a mylist;;
Type mylist defined.
```

Ce type nous dit qu'un objet de type `mylist` est soit une liste vide correspondant au constructeur `Nil`, soit un couple dont la première composante est un élément de type α et la seconde est une liste d'éléments de type α. Voici un exemple de liste :

```
#let l1 = Cons(false,Cons(true,Cons(true,Nil)));;
l1 : bool mylist = Cons (false, Cons (true, Cons (true, Nil)))
```

▸ **Exercice 2.** *Donnez une fonction* `mylist_of_list` *transformant une* `'a list` *en une* `'a mylist`.

▸ **Exercice 3.** *Donnez une fonction* `list_of_mylist` *transformant une* `'a mylist` *en une* `'a list`.

▸ **Exercice 4.** *Donnez une fonction d'insertion* `mylist_insert` *d'un élément dans une liste triée de type* `mylist`. *Cette fonction prendra en premier argument une fonction* `comp` *de comparaison (prédicat) de deux objets de type α.*

▸ **Exercice 5.** *Définissez une fonctionnelle d'itération* `mylist_it` *sur le type* `mylist`. *Cette fonctionnelle aura le type suivant :*

```
mylist_it : 'a -> ('b -> 'a -> 'a) -> 'b mylist -> 'a = <fun>
```

▸ **Exercice 6.** *Déduisez de ce qui précède la fonction* `mylist_insertsort` *de tri par insertion d'une liste de type* `mylist`. *La fonction prendra bien entendu un prédicat* `comp` *de comparaison en premier argument.*

On s'intéresse désormais au type des arbres binaires génériques, ayant des éléments de type `'a` au niveau des feuilles et des éléments de type `'b` au niveau des nœuds internes. Le type est donc paramétré par le couple `('a,'b)` de sorte que :

```
#type ('a,'b) bintree = Leaf of 'a
                      | Node of 'b * ('a,'b) bintree * ('a,'b) bintree;;
Type bintree defined.
```

À titre d'exemple, la fonction calculant la hauteur d'un tel arbre est donnée par :

```
#let rec bintree_height = function
       Leaf(_) -> 0
 | Node(_,l,r) -> 1 + max (bintree_height l) (bintree_height r);;
bintree_height : ('a, 'b) bintree -> int = <fun>
```

Ce type générique nous sert à définir un type plus particulier représentant les expressions arithmétiques usuelles. Pour cela, on procède comme suit. On définit un type des opérateurs :

```
#type op = Plus | Minus | Mult | Div;;
Type op defined.
```

puis un type des expressions :

```
#type expr = Expr of (int,op) bintree;;
Type expr defined.
```

Ainsi, une expression sera représentée par un arbre binaire dont les feuilles seront des entiers et les nœuds internes des opérateurs. Par exemple, `5 + (6 * (8 - 3))` sera codé par :

```
#let ex1 = Expr(Node(Plus, Leaf(5), Node(Mult, Leaf(6),
                                    Node(Minus, Leaf(8), Leaf(3)))));;
ex1 : expr = Expr (Node (Plus, Leaf 5, Node (Mult, Leaf 6,
                   Node (Minus, Leaf 8, Leaf 3))))
```

▸ **Exercice 7.** *Donnez une fonction* `int_of_expr` *permettant d'évaluer une expression de type* `expr`*. Par exemple, on aura :*

```
#int_of_expr ex1;;
- : int = 35
```

▸ **Exercice 8.** *Réaliser un parcours préfixe de l'arbre représentant l'expression infixe* `4 / 2` *signifie écrire cette expression en mettant l'opérateur* `/` *devant les parcours préfixes des opérandes gauche et droit. Dans notre exemple, on obtient simplement* `/ 4 2` *et pour l'exemple de la formule* `5 + (6 * (8 - 3))`*, on obtient* `+ 5 * 6 - 8 3`*. On considère le type* `ops` *représentant soit un opérateur, soit un opérande :*

```
#type ops = Op of op | Val of int;;
Type ops defined.
```

Donnez une fonction `prefix_of_expr` *retournant une* `ops list` *correspondant au parcours préfixe de l'expression passée en argument. On aura par exemple :*

```
#prefix_of_expr ex1;;
- : ops list = [Op Plus; Val 5; Op Mult; Val 6; Op Minus; Val 8; Val 3]
```

On se donne une fonctionnelle d'itération sur les objets de type `bintree` :

```
#let bintree_it f1 f2 = bintree_it_rec where rec bintree_it_rec = function
       Leaf(x) -> f1 x
 | Node(x,l,r) -> f2 x (bintree_it_rec l) (bintree_it_rec r);;
bintree_it : ('a -> 'b) -> ('c -> 'b -> 'b -> 'b) ->
             ('a, 'c) bintree -> 'b = <fun>
```

▸ **Exercice 9.** *Redéfinissez la fonction* `int_of_expr` *à l'aide de* `bintree_it`*.*

▸ **Exercice 10.** *Redéfinissez la fonction* `prefix_of_expr` *de parcours préfixe d'une expression à l'aide de la fonctionnelle* `bintree_it`*.*