PFITA – Cours/TD 7–8

UPS – L3 Info – Semestre 5

1 Rappels sur les listes

En OCaml, les listes sont homogène (1), c.-à-d. tous les éléments d'une liste ont le même type. De plus, les listes sont immuables (2), c.-à-d. on ne peut pas modifier en place une valeur de type liste.

Pour « modifier une liste », il faut donc utiliser le filtrage et la récursion (3) pour parcourir cette liste, et en construire une nouvelle.

`[x; y]` est l'écriture abrégée de l'expression `x::y::[]` (4). Le type des listes de listes de booléens s'écrit bool lst list (5).

`[1, true; 2, true]` et `[(1, true); (2, true)]` sont deux expressions équivalentes ayant pour type (int * bool) list (6)

`(1, [true])` est une expression ayant pour type int * bool list (7)
= int * (bool list)

2 Exercices sur les types utilisateur

Soit le type utilisateur suivant :

```
type t = C | B of bool * bool
```

#|t| = 1 + 2×2

Combien ce type a-t-il de valeurs différentes possibles ? 5 (8)

Quel est le type et la valeur renvoyés par OCaml pour la phrase suivante ?

```
let b = B (not false, C = C)
val b : t = B (true, true)
```

Définition de types

Définir le type `couleur` avec les 3 couleurs bleu, blanc et rouge.

```
type couleur = Bleu | Blanc | Rouge
```

⚠ aux majuscules

Définition d'expressions

Définir un objet `bleu` du type `couleur`

```
let bleu = Bleu
val bleu : couleur = Bleu
```

Définir le type `carte` à jouer avec les 3 figures (roi, dame et valet) et les cartes numérotées de 1 à 10 :

```
type carte =
  | Roi
  | Dame
  | Valet
  | Valeur of int;;
```

```
let valeur n =
  if n >= 1 && n <= 10
  then n
  else failwith "invalid number"
```

Vous proposerez une version où la valeur de la carte entre 1 et 10 est un paramètre entier du constructeur (possiblement en dehors de la plage), et définirez un "smart constructor" `valeur : int -> carte` qui vérifiera la **validité de l'entier** avant de construire la carte.

Définir les objets `valet` et `dix` de type `carte` (en utilisant le "smart constructor" si nécessaire.

```
let valet = Valet and dix = valeur 10;;
val valet : carte = Valet
val dix : carte = Valeur 10
```

Définir le type `jour` des jours de la semaine (vous pourrez utiliser des constructeurs à 2 lettres).

```
type jour = Lu | Ma | Me | Je | Ve | Sa | Di
```

Définir le type `nombre` comme étant soit un entier, soit un réel, soit un complexe (couple de 2 réels).

```
type nombre =
  | Entier of int
  | Reel of float
  | Cplxe of float * float
```

Définir le type `resultat` comme étant soit un résultat entier, soit une erreur.

```
type resultat =
  | Entier of int
  | Erreur
```

Définir le type `'param option` comme étant soit rien (`None`), soit un élément du type `'param`. Ce type est prédéfini dans la bibliothèque standard.

```
type 'p option = None | Some of 'p
```

Définir le type `nfa` à 1 constructeur (automates finis non déterministes), correspondant aux triplets (état initial, fonction de transition associant à un état et un symbole la liste des états successeurs, prédicat "état final ?").

```
type ('e, 's) nfa = Co of 'e * ('e -> 's -> 'e list)
                        * ('e -> bool)

nfa => neo déterministe, finite, automaton
```

Définir l'abbréviation de type `monAutomate`, instanciation du type générique précédent, avec des entiers pour représenter les états et des caractères pour représenter les symboles.

```
type monAutomate = (int, char) nfa
```

Définir un complexe `z` et un réel `r` du type `nombre`.

```
let z = Cplx (0.1, 2.3);;
val z : nombre = Cplx(0.1, 2.3)
let r = Reel 4.
val r : nombre = Reel 4.
```

Définir le type `ab` des arbres binaires comme étant soit une feuille contenant une information, soit un nœud avec 2 sous-arbres.

```
type 'a ab = Feuille of 'a | Noeud of ('a ab * 'a ab)
```

Définir le type `'a arbreNaire` des arbres n-aires comme étant un nœud contenant une information `'a` et un nombre quelconque d'arbres n-aires comme fils.

```
type 'a arbreNaire = Noeud of 'a * ((('a arbreNaire) list));;
```
parenthèses facultatives

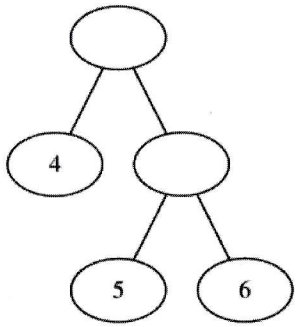
Node → pour différencier du précédent

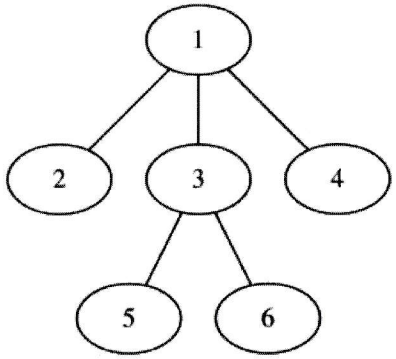
Définir le type `action` du langage LOGO, sachant qu'une action peut être : tourner d'un certain nombre de degrés, avancer d'une certaine distance, lever ou poser le crayon, une séquence de 2 actions ou la répétition d'une action un certain nombre de fois.

```
type action = Tourner of int | Avancer of int
            | Lever | Poser | Seq of action * action
            | Repeter of int * action
```

Définir un arbre binaire `t` (ci-dessous à gauche) du type `ab` contenant les entiers 4, 5 et 6. Donnez son type.

```
let t = Noeud (Feuille 4, Noeud (Feuille 5, Feuille 6));;
val t : int ab = "même écriture que au dessus"
```





Définir l'arbre n-aire `t6` (ci-dessus à droite). Donner son type.

```
let t6 = let f i = Node (i, []) in
  Node (1, [f 2; Node(3, [f 5; f 6]); f 4]);;

val t6 : int arbreNaire = _
```

Définir un objet `carre` de type `actionLogo` qui dessine un carré de 10 cm de côté à l'endroit où se trouve le crayon :

```
let carre = Seq (Poser,
            Seq (Repeter (4, Seq (Avancer (10), Tourner (90))),
                 Lever ));;
```

Filtrage et récursion pour manipuler des objets de type utilisateur

Ecrire la fonction associant à une couleur 1, 2 ou 3. Donner son type.

```
let rang coul = match coul with
    |Bleu -> 1
    |Blanc -> 2
    |Rouge -> 3
```

val rang : couleur -> int = <fun>

Ecrire la fonction `lendemain` (d'un jour de la semaine).

```
let lendemain j = match j with
    Lu -> Ma | ------ | Di -> Lu

val lendemain : jour -> jour = <fun>
```

Ecrire la `somme` des points d'une liste de cartes (les figures valent 10).

```
let rec somme l = match l with
    | [] -> 0
    | c :: l' -> somme l' + match c with
                    |Roi|Dame|Valet -> 10
                    |Valeur n -> n ;;
```

Ecrire la fonction `sommeNombres` qui additionne 2 objets de type `nombre` : soit la somme de 2 entiers, 2 réels, ou 2 complexes, soit c'est une erreur.

```
let sommeNombre n1 n2 = match n1, n2 with
    | Entier e1, Entier e2 -> Some (Entier (e1 + e2))
    | Reel e1, Reel e2 -> Some (Reel (e1 +. e2))
    | Cplx (x1, y1), Cplx (x2, y2) -> Some (Cplx (x1 +. x2,
                                                  y1 +. y2))
    | _ -> None

val sommeNombre : nombre -> nombre -> nombre option
```

Ecrire la fonction qui teste l'égalité de 2 objets du type arbre binaire, en respectant l'ordre des fils, puis modifiez cette fonction pour qu'elle ignore l'ordre des fils.

```
let rec egAbV1 a1, a2 = match a1, a2 with
    | Feuille f1, Feuille f2 -> f1 = f2
    | Noeud (g1, d1), Noeud (g2, d2)
                -> egAbV1 g1 g2 && egAbV1 d1 d2
    | _ -> false

egAbV2 ---
    | Noeud, Noeud -> egAbV2 g1 g2 && egAbV2 d1 d2 ||
                      egAbV2 g1 d2 && egAbV2 g2 d1
```

Ecrire la fonction renvoyant la liste des informations d'un un arbre n-aire parcouru dans l'ordre préfixe (vous pouvez utiliser la récursion mutuelle :)

```
let rec prefix arbre = match arbre with
    | Node (e, l) -> e :: prefix_list l

and prefix_list l = match l with
    | [] -> []
    | a :: l' -> prefix a @ prefix_list l' ;;
```

Ecrire la fonction donnant, pour un objet du type `action` (LOGO), l'état du crayon (levé = true / baissé = false), à partir de l'état initial et après exécution de l'action passée en argument :