TP : Définition de types

En plus des types prédéfinis en Caml (`int, float, string, 'a list, 'a array...`), il est possible de définir de nouveaux types. Selon ce qu'on veut, il y a plusieurs façons de le faire.

1 Types sommes

On peut tout d'abord définir des types ayant un nombre fini d'éléments, avec la syntaxe `type montype = V1 | V2 | ... | Vn`.

Par exemple, on peut créer un type des couleurs

```
type couleur = Bleu | Vert | Jaune | Rouge | Orange | Blanc | Noir
```

Pour définir une fonction sur ce type de types, on utilise le pattern matching. Par exemple, si je associer des triplets RGB à mes couleurs, on aura

```
let rgb col =
  match col with
  |Bleu -> 0,0,255
  |Vert -> 0,255,0
  |Jaune -> 255,255,0
  |Rouge -> 255,0,0
  |Orange -> 255,165,0
  |Blanc -> 255,255,255
  |Noir -> 0,0,0
```

Cette fonction a pour type `couleur -> int*int*int`.

On peut aussi utiliser cette syntaxe pour définir des réunions de types : `type montype = V1 of type1 | V2 of type2 | ... |Vn of typen`. Dans ce cas, les `Vi` sont des fonctions de type `typei -> montype`. Par exemple, on peut créer un type des nombres :

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

De la même façon, on peut définir des fonctions par pattern matching

EXERCICE

Écrire une fonction `somme : nombre -> nombre`.

On note qu'avec cette syntaxe, on peut définir des types récursivement.

```
type naturel = Zero | Succ of naturel
```

2 Types produits

Un type produit représente un produit cartésien de type. On pourrait utiliser directement la syntaxe `type montype = type1 * type2 * ... *typen`, mais on a une méthode plus explicite, en donnant un nom à chacune des composantes.

La syntaxe est `type montype = nom1 : type1 ; nom2 : type2 ; ... ; nomn : typen`. On peut alors définir un objet `obj` de type `montype` en tapant `let obj = nom1 = valeur1 ; ... ; nomn = valeurm`.

On peut aussi récupérer les informations de `obj` en tapant `obj.nomi` pour avoir la valeur de `nomi`.

Par exemple, le type des rectangles peut se définir par

```
type rectangle = {longueur : float; largeur : float}
```

Écrivons maintenant une fonction qui calcule l'aire d'un rectangle.

```
let aire rect =
  rect.longueur *. rect.largeur
```

Si on souhaite pouvoir modifier un champ d'un objet d'un type produit, il faut faire précéder ce champ du mot-clef `mutable`.

Regardons par exemple le type suivant, représentant un numéro de client ainsi qu'un nombre de visites.

```
type client = {num : int ; mutable visite : int};;

alice = {num = 1 ; visite = 0};;
let augmente_visite p =
  p.visite <- p.visite + 1;;

augmente_visite alice;;
```

3 Type polymorphes

Notons que dans les cas précédents, on peut rendre un type *polymorphe*, c'est-à-dire dépendant d'un ou plusieurs types quelconques en le précédant de `'a`, `('a,'b)`, *etc.*

Par exemples, si on voulait redéfinir les couples, on pourrait écrire

```
type ('a,'b) couple = {premier : 'a ; second : 'b}
```

On pourra ensuite considérer les couples du type `(float,int) couple` pour avoir des couples dont la première composante est réelle, et la seconde entière.

4 Exercices

Exercice 1. On considère le type suivant

```
type couleur =
  |Rouge
  |Vert
  |Bleu
  |Melange of couleur*couleur
```

Écrire une fonction donnant le code RGB de toute couleur.

Exercice 2. On utilise le type suivant pour définir les polynômes :

```
type fonction =
|Const of int
|Somme of fonction*fonction
|Produit of fonction*fonction
|Puiss of fonction*int
```

Écrire une fonction `image : fonction -> int -> int` qui calcule l'image d'un entier par une fonction

Écrire une fonction `derive : fonction -> fonction` qui calcule la dérivée d'une fonction.

Écrire des fonctions `degre : fonction -> int` et `cd : fonction -> int` qui calculent le degré et le coefficient dominant d'un polynôme.

Exercice 3. Définir un type produit correspondant aux nombres complexes. Implémenter ensuite les fonctions usuelles sur les complexes : somme, produit, quotient, module, conjugaison.

Exercice 4. Définir un type produit `etudiant` qui contient un nom, un prénom, un âge et une liste de notes.

Écrire une fonction `anniversaire` qui augmente l'âge d'un étudiant d'un an.

Écrire des fonctions pour ajouter une note à un étudiant, calculer sa moyenne, et enlever les deux notes extrèmes (la plus haute et la pus basse).

Écrire une fonction `etudiant list -> float` qui calcule la moyenne de la classe.