Chapitre 3

Types

Option informatique MP1, MP2 & MP3

OCaml permet de créer facilement des types nouveaux.
C'est un outil qui augmente la lisibilité des programmes en permet un contrôle fin des variables. En particulier le pattern matching, dans le cas des types sommes, permet une structuration plus simple des études des différents cas.

I Types produits

Un type produit (ou enregistrement) est un ensemble de rubriques nommées (ou champs) qui ont chacune un type fixé.

```
type personne = {nom : string; age : int};;
```

- Les différents champs peuvent ne pas être du même type.
  Les champs ne sont par ordonnés, contrairement au couples et tuples.
- On définit un enregistrement en renseignant chacune des champs :

  ```
  let ed = {age = 63; nom = "Détrez"};;
  ```

- On atteint la valeur du champs `label` de l'enregistrement `nom` par `nom.label`.

  ```
  ed.nom;;
  #- : string = "Détrez"
  ```

- Si on fait précéder le type d'un champs du mot clé `mutable` alors la valeur de la rubrique est modifiable. On modifie un champs par `nom.label <- nouveau`. La modification d'un champs mutable est une instruction de résultat `unit` sans retour de valeur. C'est un effet de bord.

  ```
  type etudiant = {nom : string;
                   mutable classe : string};;
  let ed = {nom = "Detrez"; classe = "HX4"};;
  ed.classe <- "M'1";;
  ed;;
  #- : etudiant = {nom = "Detrez"; classe = " M'1"}
  ```

- Le comportement des enregistrements ressemble fortement à celui des tableaux, en particulier la modification d'un champs mutable fait perdre l'ancienne valeur.
- La définition d'une variable référencée (avec le mot-clé `ref`) crée en fait un enregistrement mutable dont le seul champs est `content`.
- On peut aussi définir soi-même un type d'entier mutable.

```
type intRef = {mutable valeur : int};;
# type intRef = { mutable valeur : int; }

let n = {valeur = 1};; (* remplace let n = ref 1;; *)
# val n : intRef = {valeur = 1}

let incremente n = n.valeur <- n.valeur + 1;;
# val incremente : intRef -> unit = <fun>
```

I.1 Types sommes

Un type somme est formé d'une énumération de cas possibles pour une valeur de ce type, chaque cas comporte un nom de cas, le **constructeur**, et un (éventuel) type associée.
Les cas sont séparés par |.
Il faut écrire en majuscule la première lettre du nom d'un constructeur de valeur.

```
type couleur = Pique | Coeur | Carreau | Trefle;;
type carteTarot = |Roi of couleur
                  |Dame of couleur
                  |Cavalier of couleur
                  |Valet of couleur
                  |Numero of int*couleur
                  |Atout of int
                  |Excuse;;
```

Lorsque l'on veut définir une fonction sur un type somme il faut en général donner un comportement distinct selon les différents cas.
On peut, comme dans le cas des listes, utiliser le **pattern matching**.

```
let bout carte =
  match carte with
  |Excuse -> true
  |Atout k -> k =1 || k = 21
  |_ -> false;;
```

```
let points carte =
  match carte with
  |Excuse -> 4.5
  |Atout k when k =1 || k = 21 -> 4.5
  |Roi c -> 4.5
  |Dame c -> 3.5
  |Cavalier c -> 2.5
  |Valet c -> 1.5
  |_ -> 0.5;;
```

Un type peut être défini de manière récursive.
Par exemple on peut définir les listes avec

```
type 'a liste = Vide | Cons of 'a * 'a liste;;
```

Pour gérer la possibilité de l'absence de réponse il existe le type **optionnel** :

```
type 'a opt = None | Some of 'a;;
```