Les types définis par l'utilisateur

Nous disposons pour l'instant des types

- Elémentaires :

int, float, bool, char, string

- Construits :

fonctions, n-uplets, listes

et de toutes les combinaisons récursives de ceux-ci.

Nous allons voir dans ce chapitre comment :

- Définir et nommer nos propres types.
- Comment définir des types regroupant les différentes formes que peut prendre un objet. Ce seront **les types sommes.**
- Etendre la notion de n-uplet pour obtenir des types regroupant différents champs de types différents. Ce seront **les types produits.**

Les types somme

Les types sommes ou type "ou“ sont définis lorsqu'un objet peut prendre plusieurs formes (d'un type "ou“ d'un autre...)

Exemple

- On aimerai bien regrouper les entiers et les réels sous un même type *nombre*.
- La solution d'une équation du second degré peut être un réel, deux réels distincts ou deux complexes conjugués. On peut souhaiter regrouper ces différentes formes sous le même type *solution*.

- Un polynôme peut être représenté par
  - Son degré et la liste de ses coefficients $(n, [a_0, ..a_n])$ de type *int*float list*. On parle de représentation pleine des polynômes.
  - C'est maladroit par exemple pour $P = X^{17} - 1$ pour lequel on préfère la représentation creuse par une liste de couples (k, a_k) représentant chaque monôme. P est alors représenté par $[(0, -1.); (17, 1.)]$ de type *(int*float) list*.

  On veut regrouper les deux représentations possibles sous un même type *polynome*.

Nous allons voir trois façons de définir des types somme.

Les types énumérateurs de constantes

Définition

La syntaxe générale permettant de définir un type ne prenant qu'un nombre fini de valeurs constantes est :

```
type id=Cst1|...|Cstn;;
```

Attention les noms de valeur doivent commencer par une majuscule.

Exemple

- Le type booléen ne prend que deux valeurs constantes. Il peut être énuméré :

```
#type booleen = True| False;;
# let t= True;;
val t : booleen = True
```

- Les couleurs d'un jeu de carte :

```
# type couleur_carte = pique | coeur | carreau | trefl
Characters 27-28:
Syntax error
# type couleur_carte = Pique | Coeur | Carreau | Trefl
type couleur_carte = Pique | Coeur | Carreau | Trefle
# Pique;;
- : couleur_carte = Pique
```

Les constructeurs de type somme

Définition

Si on veut définir un type de nom id pour représenter un objet pouvant prendre plusieurs formes de types différents. On donne un nom à chacune des formes possibles et on utilise la syntaxe :

```
type id = Nom_1 of t1
        |...
        |Nom_n of tn;;
```

Un objet de ce type est alors utilisé sous la forme :

```
Nom_i exp
```

où nom_i est le nom de la ième forme et exp une expression de type t_i. ATTENTION : les noms de forme doivent obligatoirement commencer par une majuscule.

Exemples

- On peut enfin regrouper les entiers et les réels dans un même type :

```
#type nombre = N of int| R of float;;
#N 4;;
-:nombre=4
#R 4.;;
-:nombre=4.
```

• On définit le type solution pour les équations du second degré :

```
#type solution =
      Double of float
    | DeuxRéels of float*float
    | DeuxComplexes of (float*float)*(float*float);;
```

• On définit le type polynome regroupant les représentation pleines et creuses :

```
#type polynome = Plein of int*(float list)
               | Creux of (int*float) list;;
#let p=Plein (4,[1.;1.;0.;2.;3.]);;
val p : polynome = Plein (4, [1; 1; 0; 2; 3])
```

• On peut définir un complexe par sa forme géométrique $\rho e^{i\theta}$ ou par son écriture algébrique $a + ib$:

```
#   type complexe = Geo of (float*float)
                  | Alg of (float*float);;

  type complexe = Geo of (float * float) | Alg of (float *
```

• Un marchand de peinture dispose de trois couleurs toutes prêtes et peut commander n'importe quelle peinture à l'aide d'un code. On peut définir un type somme regroupant des énumrations de constantes et un constructeur de forme par :

```
#type peinture = Bleu|Blanc|Vert| Code of int;;
type peinture = Bleu | Blanc | Vert | Code of int
#Bleu ;;
-: peinture = bleu;;
#let c= Code 120 ;;
val c : peinture = Code 120
```

• On énumère les têtes possibles d'un jeu de carte :

```
#type tete_carte=As|Roi|Dame|Valet;;
```

• On peut alors définir un type carte utilisant les types énumérés tete et couleur-carte :

```
#type carte =
    Tete of tete_carte*couleur_carte
  | Petite of int*couleur_carte;;

#Tete (Roi,Carreau);;
-:carte=tete (Roi,Carreau)

#Petite (9,Carreau);;
-:carte=petite (9,Carreau)
```

Fonctions définies sur un type somme

Le plus efficace est de les définir par filtrage en fonction des constructeurs :

```
let moduleC= fun
(Geo(x,y))->x
| (Alg (a,b))->sqrt(a*.a+.b*.b);;
val moduleC : complexe -> float = <fun>
#let partie_reelle= fun
(Geo(x,y))->x*.cos(y)
| (Alg (a,b))->a;;
val partie_reelle : complexe -> float = <fun>
```

On ajoute donc à la définition des filtres la possibilité d'utiliser des constantes de type sommes et des constructeurs de type somme (entre parenthèse).

Complément sur les filtres :*match ... with*

Nous avons vu au premier semestre l'intérêt d'utiliser un filtrage des valeurs lors de la définition d'une fonction :

```
#let liste_vide = fun
 []->true
|_->false;;
liste_vide : 'a list -> bool = <fun>
```

Définition

La structure match...with permet un filtrage dit explicite, c'est-à-dire nommant le paramètre de la fonction :

```
#let liste_vide = fun  l-> match l with
 []->true
|_->false;;
val liste_vide : 'a list -> bool = <fun>
```

On peut l'utiliser pour filtrer un élément d'un n-uplet...

```
#let rec concatene = fun  (a,b)-> match a with
 []->b
|x::reste -> x::concatene (reste,b);;
val concatene : 'a list * 'a list -> 'a list = <fun>
```

...ou plusieurs :

```
#let rec meme_longueur = fun  (a,b)-> match (a,b) with
 ([],[])->true
|(_::r,_::l) -> meme_longueur(r,l)
| _   -> false ;;
val meme_longueur : 'a list * 'b list -> bool = <fun>
```

Complément sur les filtres : nommage d'une valeur filtrée

Lors d'un filtrage, il est parfois pratique de nommer tout ou partie de la valeur filtrée.

Définition

Le mot-clef as permet de nommer tout ou partie d'un filtre.

Exemple : Le calcul du minimum de deux rationnels

```
let minRat =  fun
(((n1,d1) as r1),((n2,d2) as r2))-> if n1*d2<n2*d1 then r1

  val minRat : (int * int) * (int * int) -> int * int = <f
```

Complément sur les filtres : le filtrage gardé

Définition

Le filtrage gardé filtre **when** *cond est une syntaxe allégée pour le cas fréquent ou un filtre est immédiatement suivi par une expression conditionnelle.*

```
let egalNonNul =  fun
(0,0)-> false
|(x,y)-> if x=y then true else false;;
```

peut ainsi s'écrire :

```
let egalNonNul =  fun
(0,0)-> false
|(x,y) when x=y -> true
|(x,y) -> false;;
```

Exercices

Calculons la valeur d'une carte à jouer à la belotte en fonction de l'atout :

Exercices

Calculons la valeur d'une carte à jouer à la belotte en fonction de l'atout :

```
#let valeur = fun  atout -> fun
(Tete(As,_))->11
|(Tete (Roi,_))->4
|(Tete (Dame,_))->3
|(Tete (Valet,c))->if c=atout then 20 else 2
|(Petite (10,_))->10
|(Petite (9,c))->if c=atout then 14 else 0
|_->0;;
val valeur : couleur_carte -> carte -> int = <fun>
```

Attention! La définition d'un nouveau type utilisant un nom de constructeur déjà utilisé peut provoquer des erreurs.
Attention! Lorsqu'on définit un nouveau type il faut définir toutes les opérations sur ce type même si elles vous paraissent banales :

```
#type nombre = N of int | R of float;;
#let p = N 4;;
val p : nombre = N 4
#p+1;;
>^
Cette expression est de type nombre,
mais est utilisée avec le type int.
```

```
#let plus = fun  k-> fun
(N n)->N (k+n)
|(R r)->R (r+.float_of_int(k));;
val plus : int -> nombre -> nombre = <fun>

#plus 1 p;;
- : nombre = N 5
```

C’est en particulier le cas sur les listes :

```
#type truc = L of int list;;

#let t= L [1;2;3];;
val t : truc = L [1; 2; 3]
#4:: t;;
>    ^
Cette expression est de type truc,
mais est utilisée avec le type int list.
#let cons = fun
(a,L l)-> L (a::l);;
val cons : int * truc -> truc = <fun>
#cons(4,t);;
- : truc = L [4; 1; 2; 3]
```