Produits cartésiens

- Soit A, B des ensembles, le **produit cartésien** $A \times B$ est

$$A \times B = \{(a, b),\ \ a \in A,\ \ b \in B\}$$

- Un élément (a, b) est appelé **couple,** a et b sont les deux composantes de ce couple.
- Plus généralement, si $A_1, A_2, ..., A_n$ sont n ensembles,

$$A_1 \times A_2 \times ...A_n = \{(a_1, a_2, ..., a_n),\ \ \forall i \in [1, n], a_i \in A_i\}.$$

- $(a_1, .., a_n)$ est appelé **n-uplet** et $\forall i \in [1, n], a_i$ est la ième composante du n-uplet.

- $\forall i \in [1, n]$ on définit la ième **projection** :

$$p_i : : \left(\begin{array}{c} A_1 \times ... \times A_n \to A_i \\ (a_1, ..., a_n) \mapsto a_i \end{array} \right.$$

Produits cartésiens en CAML

Le produit cartésien des types t_1 et t_2 est noté $t_1 * t_2$.
Plus généralement le produit cartésien des n types $t_1, \ldots t_n$ est noté

$$t_1 * t_2 * \ldots * t_n.$$

Comme en mathématiques, le constructeur de n-uplets est la virgule, un n-uplet est donc une expression de la forme

$$exp_1, exp_2, \ldots, exp_n$$

Remarque : Les parenthèses ne sont donc pas nécessaires en CAML, mais il est conseillé de les conserver pour améliorer la lisibilité, pour conserver l'habitude prise en maths, et surtout pour se protéger des surprises dues à la faible priorité de la virgule face aux autres opérateurs.

Exemple :

```
# 1,2;;
-:int*int = 1,2

# 1,true;;
-:int*bool = 1,true

# 1,(1,2);;
-:int*(int*int) = 1,(1,2)

# `a`,fun x->x+1;;
-:char*(int->int) = `a`,<fun>
```

Fonctions à plusieurs variables

Nous ne savons écrire en CAML que des fonctions à un paramètre.
Mais il nous suffira de choisir des n-uplets comme paramètre pour être en mesure d'écrire des fonctions de plusieurs variables.
Exemple :

```
# let divise = fun (n,d) ->
      if d=0 then false
      else n mod d = 0;;
```

Règles d'évaluations

- **Typage**
  Soit à typer l'expression exp :

  $$exp_1, exp_2, \ldots, exp_n.$$

  Si pour tout i$\in [1, n]$, exp_i est une expression de type t_i, alors exp est de type produit cartésien

  $$t_1 * \ldots * t_n.$$

- **Evaluation**
  Si pour tout $i \in [1, n]$, la valeur de l'expression exp_i dans l'environnement Env est v_i, alors la valeur de exp dans Env est

  $$v_1, .v_2, \ldots v_n$$

Attention! à l'ordre de priorité du aux parenthèses :

```
# 1,2,3;;
-:int*int*int = 1,2,3
```

est un triplet d'entiers alors que

```
# 1,(2,3);;
-:int*(int*int) = 1,(2,3)
```

est un couple dont la première composante est un entier et la seconde un couple d'entiers.

Projections

Pour les produits de DEUX ensembles, CAML prédéfinit les projections *fst* et *snd* qui donnent respectivement la première et la deuxième composante d'un couple.

Exemple :

```
# fst(1,2);;
-:int = 1
# fst(1,2,3);;
      ^^^^cette expression est de type int*int*int
          mais est utilisée avec le type int*int
#fst((1,2),3);;
-:int*int = (1,2)
(* Mais bien réfléchir à ... *)
#fst (1,2),3;;
-:int*int = (1,3)
```

Exercice

- Ecrire une fonction qui calcule la plus grande composante d'un couple :
  Solution 1 :

```
#let max=fun c->
if fst(c)>snd(c) then fst(c) else snd(c)
```

  pas terrible, car on a deux appels à fst et snd.
  Solution 2 :

```
#let max=fun c->
let x=fst(c) and y=snd(c) in
if x>y then x else y
```

Mais la meilleure méthode consiste à utiliser la forme (x,y) évitant tout appel à fst et snd :

Solution 3 :

```
#let max=fun (x,y)->if x>y then x else y
```

Exercices

1) Expressions de type produit :
Donner le type et la valeur expressions suivantes :

```
#(1,false);;
#(2.,1+4,1=1,"salut");;
#((`a`,1),(`b`,2),(`c`,3));;
#(1,2,3);;
#(1,(2,3));;
#((1,2),3);;
#(1,(2,3))=(1,2,3);;
#(1,2,3)=(2,1,3);;
#fst(2,false);;
#snd(2,false);;
#fst((1,2),3);;
```

Fonctions définies sur un produit cartésien :

- Ecrire une fonction min calculant le minimum de deux entiers.
- Ecrire une fonction norme qui calcule la somme des carrés des éléments d'un triplets d'entiers (on utilisera une définition locale d'une fonction carree).
- Ecrire une fonction inflex qui réalise la comparaison lexicographique de deux couples de nombres entiers.

Fonctions définies par filtrage

La notion de filtrage est une généralisation de la notion de paramètre.
Un filtre permet de préciser les différentes valeurs que peut prendre un paramètre.
On en rencontre souvent en informatique et en mathématiques :
Quand on tape la commande MS-DOS :

$$copy \quad *.ml \quad a :$$

$*.ml$ est un filtre permettant de selectionner les fichiers qui seront soumis à la commande copy.

Filtres en maths...

1. $\forall x \in \mathbb{R}$ est un filtre qui filtre toutes les valeurs réelles.
2. "Considérons un élément de la forme $(x, 0)$ où $x \in \mathbb{R}$..."
3. "Les solutions sont les fonctions de la forme :“

$$\lambda cos(x) + \mu e^{-x}$$

4. Comme en CAML, on s’en sert aussi pour définir des fonctions :

$$f \ : \left(\begin{array}{c} \mathbb{R} \rightarrow \mathbb{R} \\ x \neq 0 \mapsto \frac{sin(x)}{x} \\ 0 \mapsto 1 \end{array} \right.$$

Qui s'écrira en Caml...

```
Let f= fun
  0.->1.0
| x-> sin(x)/.x;;

f : float->float=<fun>
```

Définition d'un filtre en CAML

- En CAML, un filtre est une expression constituée de constantes, d'identificateurs, du symbole souligné et des constructeurs de n-uplets (..,..,). Les filtres remplacent et généralisent les paramètres formels de fonctions.
- Les constantes ne laissent passer qu'elles-mêmes,
- les identificateurs et le symbole souligné laissent tout passer (nous verrons plus loin ce qui les distingue)
- les constructeurs de n-uplets permettent de définir la forme de l'objet à filtrer.

Exemples de filtre

1 est un filtre qui ne laisse passer que la constante 1
n est un filtre qui laisse passer n'importe quelle valeur de n'importe quel type.
(0,x) et (0,_) sont des filtres qui laissent passer tous les couples de première composante nulle.

Utilisation des filtres en Caml

On utilise les filtrages pour affiner la définition des fonctions :
Syntaxe :

```
Let ident = fun
  F1->exp1
| F2->exp2
    .
    .
    .
| Fn->expn;;
```

Exemples

```
Let f= fun
  1->0
| n-> n+1;;
f : int->int=<fun>

Let zero= fun
  0->true
| _-> false;;
f : int->int=<fun>

Let divise= fun
(_,0)  -> false
|(n,d) -> n mod d =0;;
f : int->bool=<fun>
```

Syntaxe

- Les filtres F_i sont des expressions d'un même type t_1 correspondant à différentes valeurs du paramètre formel.
- Les expressions exp_i sont différents corps de la fonction, correspondant aux valeurs à prendre en fonction du filtre.
  Remarque : Dans un filtre le nom du paramètre formel n'a aucune influence, mais à l'intérieur d'un même filtre, on ne peut utiliser qu'une fois un même paramètre formel. Par exemple le filtre $(x, 0, x)$ est interdit.

Règles d'évaluation

Typage :
Si tous les filtres sont de type t_1 et que tous les corps de la fonction sont de type t_2 alors la fonction définie est de type

$$t_1 \to t_2.$$

Dans tous les autres cas il y a erreur de type.
Evaluation :
On n'écrira jamais complètement la fermeture d'une fonction définie par filtrage. Cela se fait exactement comme pour une fonction classique, en particulier il faudra se souvenir de l'environnement de définition.

Evaluation d'un appel de fonction définie par filtrage

Lors d'un appel :

```
ident(arg);;
```

1) L'argument arg est évalué dans l'environnement d'appel, on cherche alors à filtrer sa valeur V en parcourant les différents filtres dans leur ordre d'écriture.

- 2a) Si aucun filtre ne correspond à l'argument, l'évaluation de la fonction échoue. On dit dans ce cas que le filtrage n'est pas exhaustif (certain arguments ne sont filtrés par aucun filtre). Le système CAML détecte ces cas lors de la définition de la fonction et affiche une mise en garde :

```
#let f= fun 1->0|2->3;;
Attention : Ce filtrage n'est pas exhaustif

#f(1);;
-:int=0

#f(3);;
exception non rattrapée : match failure
```

N.B : Comme en maths, ce n'est pas une erreur d'écrire une fonction non exhaustive, mais s'en est une que de l'utiliser hors de son ensemble de définition.

- 2b) Sinon : le filtrage prend fin sur le premier filtre (dans l'ordre d'écriture) correspondant à l'argument :

  ```
  |Fi->expi
  ```

La encore distinguons deux cas :

a) Si Fi contient des paramètres formels :

3a) Il y a création d'un environnement temporaire contenant des liaisons entre les paramètres formels de Fi et la valeur correspondante dans l'argument, ajouté à l'environnement de définition de la fonction.

4a) expi est alors évalué dans cet environnement et c'est le résultat de la fonction.

5a) L'environement temporaire est détruit.

Exemple :

```
#let f = fun
0->1
|n->2*n-3;;

f(4);;
-:int=5
```

4 n'est pas filtré par 0
4 est filtré par n
1)Création de

```
EnvT=[(n,4)><Env-def]
```

2) évaluation de 2*n-3 dans EnvT : 5
3)destruction de EnvT

Attention! L'ordre d'écriture des filtres est donc très important :
Dailleurs Caml y prend garde :

```
#let f=fun
n->2*n-3
|0->1;;
Attention : ce cas de filtrage est inutile.
#f(0);;
-:int=-3
```

Regardons un nouvel exemple :

```
#let f=fun
1->"un"
|n->"plusieurs";;

f(4)
```

Conduit à la création de l'environnement temporaire

```
EnvT=[(n,4)><Env_def]
```

pour y évaluer l'expression constante "beaucoup". La liaison est donc complètement inutile et on cherchera à l'éviter.

b) Si l'expression n'utilise pas le paramètre formel il est inutile de créer la liaison correspondante. C'est dans ce but que l'on filtrera par le caractère souligné plutôt que par un identificateur. En effet ce symbole filtre toute les valeurs mais ne provoque aucune liaison dans l'environnement temporaire.

Remarque : Si l'on veut rendre exhaustive une fonction qui ne l'est pas et personnaliser le message d'erreur à afficher lorsqu'on appelle la fonction hors de son ensemble de définition, on peut utiliser la commande failwith

Exemple :

```
let f = fun
1->"un"
|2->"deux"
|_ -> failwith "f n'est définie qu'en 1 et 2."
```

C'est le moment de filtrer tous les cas non traités par le symbole souligné.

Exercices

- Attention aux erreurs fréquentes : Un filtre ne peut distinguer que des valeurs particulières de l'argument et pas des conditions telles que : si l'argument est un entier ou si l'argument est positif etc.
  On est parfois amener à mélanger filtres et sélection pour gérer ces cas.
  On rappelle également qu'un même identificateur ne peut être présent deux fois dans un même filtre :

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

```
let pasDe0=fun
(0,_)->false
|(_,0)->false
|_->true;;

let lexicopositif =fun
(0,n)->if n>=0 then true else false
|(n,_)->if n>0 then true else false;;
```

- Soit la fonction :

```
#let f= fun
  (0,_)->0
|(_,0)->5
|(1,x)->x+50
|(x,1)->x+35
| _ ->3;;
```

Expliquer les calculs de f(0,1), f(1,0), f(1,4), f(4,1), f(25,5)

- Ecrire les fonctions booléennes et, ou, et xor .
- Ecrire des fonctions ord2 et ord3 qui ordonnent les éléments d'un couple puis d'un triplet d'entiers.