Produit cartésien généralisé (tuples)

Le produit cartésien binaire se généralise aux tuples.

$$E_1 \times E_2 \times \cdots \times E_n = \{(e_1, e_2, \cdots, e_n) \mid e_i \in E_i \forall i, 1 \leq i \leq n\}$$

Exemples:

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

Tuples

Les tuples peuvent être paramètres des fonctions.

```
# let s3 (i, j, f) = i + j + int_of_float f;;
val s3 : int * int * float -> int = <fun>
# s3 (1, 2, 3.);;
- : int = 6
```

Une fonction peut retourner un tuple:

```
# let next (u, v) = v, (u + v);; (* cf Fibonacci *)
val next : int * int -> int * int = <fun>
# next (1,1);;
- : int * int = (1, 2)
# next(next (1,1));;
- : int * int = (2, 3)
# next(next(next (1,1)));;
- : int * int = (3, 5)
```

Filtrage

On peut récupérer les valeurs d'un tuple par filtrage:

```
# let tuple = 1, "deux", 3.5;;
val tuple : int * string * float = (1, "deux", 3.5)
# let i, str, f = tuple in i + int_of_float f, str;;
- : int * string = (4, "deux")
```

Requête `type`

requête `type` permet de donner un nom à un type (en général composé)

```
# type point2D = Point of float * float;;
type point2D = Point of float * float
# Point(1.2, 3.4);;
- : point2D = Point (1.2, 3.4)
```

`point2D` est le nom de type choisi. `Point` est le nom de constructeur choisi pour représenter un point. `type`, `of`, `float` sont prédéfinis en OCaML.
Le séparateur `|` correspond à une union (ou somme) de types.

```
# type int_or_infinity = Int of int | Infinity;;
type int_or_infinity = Int of int | Infinity
```

Des valeurs de ce type sont: `Int 5`, `Int (-2)`, `Infinity`.

```
type int_or_infinity = Int of int | Infinity
# let div n d =
  if d = 0 then
    if n = 0 then failwith "undefined form 0/0"
    else Infinity
  else Int(n/d);;
val div : int -> int -> int_or_infinity = <fun>
# div 3 0;;
- : int_or_infinity = Infinity
# div 3 2;;
- : int_or_infinity = Int 1
# div 0 0;;
Exception: Failure "undefined form 0/0".
# type form = Square of float | Circle of float | Rectangle
type form = Square of float | Circle of float | Rectangle
# Rectangle (10., 3.);;
- : form = Rectangle (10., 3.)
# Square(3.4);;
- : formes = Square 3.4
```

La construction `match`

La construction `match` permet d'inspecter la forme d'une valeur et de récupérer parties de la valeur dans des variables locales.

```
# let perimeter_rect w h = 2. *. (w +. h);;
val perimeter_rect : float -> float -> float = <fun>
# let perimeter_circle r = 2. *. 3.14 *. r;;
val perimeter_circle : float -> float = <fun>
# let perimeter form =
  match form with
    Rectangle(w, h) -> perimeter_rect w h
  | Circle r -> perimeter_circle r
  | Square c -> perimeter_rect c c;;
val perimeter : formes -> float = <fun>
```

Variable anomyme

Le caractère `_` (souligné ou "tiret du 8" sur un clavier AZERTY) correspond à une variable anonyme.
On peut l'utiliser

- à gauche du `=` dans un `let` ou `let ... in`

```
# let x, _, z = 1,2,3 in x, z;;
- : int * int = (1, 3)
```

- dans un motif d'un `match`

```
let est_une_figure carte =
  match carte with
    As _ -> false
  | Numero(_, _) -> false
  | _ -> true
```

- dans un fichier `.ml` pour mémoriser une expression

```
let _ = couleur_carte (Numero(7, Pique))
let _ = couleur_carte (As Coeur)
```

Regroupement des clauses d'un `match`

```
match carte with
  As c -> c
| Roi c -> c
| Dame c -> c
| Valet c -> c
| Numero(_, c) -> c
```

```
match carte with
  As c | Roi c | Dame c | Valet c | Numero(_, c) -> c
```

```
match carte with
  As _ -> false
| Numero(_,_) -> false
| _ -> true
```

```
match carte with
  As _
| Numero(_,_) -> false
| _ -> true
```

Représentation d'un point du plan par un complexe

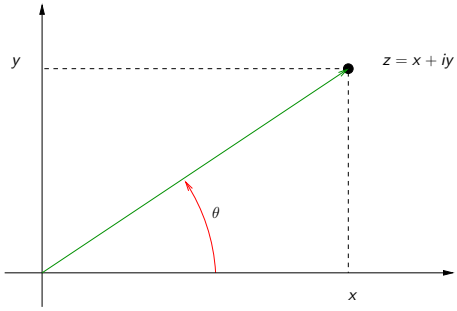

Figure: Plan complexe

plan muni d'un repère orthonormé; au point de coordonnées (x, y), on associe e nombre complexe $z = x + iy$, son affixe.
Pour représenter un point du plan, on utilise le type

```
type mycomplex = C of float * float
```

```
type point = mycomplex
```

Opérations sur les complexes

On écrira en TM les fonctions et variables suivantes:

1. fonction constructeur `make_complex` qui permet de construire un objet de type `mycomplex` fabrique un complexe à partir de ses parties réelle et imaginaire
2. accesseur `realpart` qui permet de récupérer la partie réelle d'un complexe.
3. accesseur `imagpart` qui permet de récupérer la partie imaginaire d'un complexe.
4. variable `c_origin` contenant le point de coordonnées $(0, 0)$.
5. la variable `c_i` ayant pour valeur le complexe $i = (0, 1)$,
6. opérations
   `c_abs, c_sum c1 c2, c_dif c1 c2, c_opp`,
   `c_mul c1 c2, c_abs c, c_sca lambda c, c_exp c`
   qui permettent respectivement de calculer la valeur absolue d'un complexe, la somme, la différence, l'opposé, le produit de deux complexes, la multiplication d'un complexe par un scalaire (float), l'exponentielle complexe.

Transformations dans le plan complexe

Une transformation F du plan transforme tout point P en son image $P' = F(P)$. On peut décrire la transformation F par la fonction f qui appliquée à z l'affixe de P donne z' l'affixe de P'.

$$\begin{array}{rccc} f: & \mathbb{C} & \to & \mathbb{C} \\ & z & \mapsto & z' = f(z) \end{array}$$

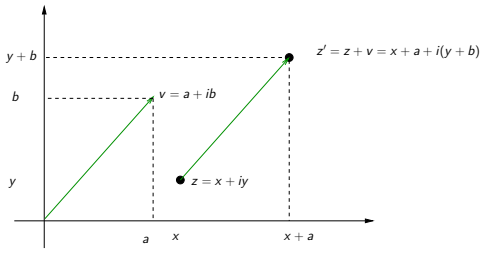

Figure: Translation de vecteur (a, b)

On écrira en TM la fonction `translate c vector`.