

Types en OCAML

Type : = type de données

Exemple: bool, int, char, string, float, ...
" double en réalité"

On peut créer ses types "conteneurs"

- listes
- tableaux
- :

$h \bmod 5$ ~~~ modulo

$1 | s| 32$ ~~~ multiplie 1 32 fois par 2

$1 | \text{and } 4$ ~~~ divise 1 4 fois par 2

0b0110 permet d'écrire un nombre sous forme binaire.

Oxat...

Forme hexadécimale

Dans un match & with

| ...

| ...

Les types de sorte doivent être identiques
Ne pas oublier le cas général...

Pour les fonctions de la librairie standard:

Char.code

nom de module nom de fonction

String.make 10 'a' → "aaaaaaaaaa"
let s = "abc" in s.[1] } → 'b'
(⇒ let s = "abc" in String.get s 1) } → 'b'

↳ Quels sont les types plus riches

1) Pairs ou n-uplets

$\{ \dots * \dots \}$ ↓
 $(\text{inr} * \text{bool})^n$ $* \text{inr}$

Les parenthèses sont conseillées

Sur les paires, si y a 2 fonctions:

fst((0, 2)) → 0

snd((true, 1)) → true

let fstTripl r =
 match r with
 | (x, _, _) → x
 → 'a * 'b * 'c → 'a

Fonction qui prend
1 argument (un
triplet) et renvoie
le premier élément
du triplet.

On ne peut pas comparer un n-uplet à un m-uplet. ($n \neq m$)

On peut imbriquer les n-uplets:

$((1, 2), 3, \text{True}, ("ab", "bc"))$

Pour aller plus vite dans l'écriture:

Exemples:

let for $\underbrace{(a, b)}_{\text{Pattern matching}} = a;;$

Pattern matching
intégré à
l'expression.

let sum_rec $(a, b, c) =$
 $a + b + c$

→ Somme d'un tuple d'entier.

2) Les records

Type user = {
name: string;
surname: string;
power: int
};;

let luke = {
name = "luke";
surname = "sky...";
power = 11;
};

```
let docth_my_father = {  
    luke with name = "anakin";  
};
```

3) Types énumérés:

```
type color =  
| Red  
| Black  
| White;;
```

```
let change_color x =  
match x with  
| Black → White  
| White → Red  
| Red → Black;;
```

On peut ajouter des valeurs aux types énumérés

```
type number =  
| Float of float  
| Integer of int;;
```

ajoute un n-uplet
ou un 1-uplet
ajoute un entier pour Integer

```
Float (3.14);;
```

```
let opposite n =  
  match n with  
  | Float f → Float(-.f)  
  | Integer i → Integer(-i);;  
Il faut gérer  
tous les types  
de l'enum Number.
```

opposite Integer(123);;
→ Integer(-123)

```
let add n m =  
  match (n, m) with  
  | (Float n, Float m) → Float(n +. m)  
  | (Integer n, Integer m) → Integer(n + m)  
  | (Integer n, Float m) | (Float m, Integer n) →  
    Float(Float.of_int(n) +. m);;
```

Type Number =

:
| Fraction of int * int;;

```
let calc_frac =  
  match frac with  
  | Fraction(x, y) → Float.of_int(x) /. Float.of_int(y);;
```

Type 'a option =
| None
| Some of 'a

```
let predecessor n =  
  if n = 0 then None else Some(n - 1);;
```

On peut créer des types récursifs.

Type my_list_t =
| Empty
| Element of int * my_list_t;;

[1,2,3] ← let l1 = Element(1, Element(2, Element(3, Empty)));
[] ← let l2 = Empty;;

let rec length l =
match l with
| Empty → 0
| Element(_, l) → 1 + length(l);;

let rec sum l =
match l with
| Empty → 0
| Element(n, l) → n + sum(l);;

Type btree =
| Leaf
| Node of btree * int * btree;;

let alone = Node(Leaf, 1, Leaf);;

Les liaisons prédefinies en Ocaml:

C'est en fait des 'à list'

| @ | → concaténation de listes

:: | → i = val courante de l, l dépend de la suite.

```
let rec length l =  
  match l with  
  | [] → 0  
  | _ :: l → 1 + length(l);;
```