

# UB - Programmation Fonctionnelle.

## Cours 1.

L'algorithme et la programmation fonctionnelle font le choix de rester au plus près de la description en termes de fonction mathématique de ce qu'est traitement de données.

effets de bord: décrit les traitement de données → modification d'un état global

- fort, - statique,  
OCaml 是强静态类型的语言。

以小写字母或下划线开头

### Structures des contrôles:

Il est une expression composée permettant de choisir quelles sous-expressions sont évaluées et dans quel ordre.

Structure de données: Il est une expression composée permettant d'agréger les valeurs des sous-expressions dans une même donnée, comme des listes, des arbres, etc.

- 关于局部是与全局是：  
① 局部是只会隐藏同名的全局是；  
② 局部是并不会修改全局是；

Notion d'environnement : 环境 (environnement de calcul) 是一个有序的对·标识符, 值的列表  
被称为连接 (liaison).

例：  $B = \langle y, 7 \rangle; \langle x, 3 \rangle; \langle z, 1 \rangle$  连接 (liaison)  $B' = B; \langle u, 4 \rangle$

R.P  $B' = \langle y, 7 \rangle; \langle x, 3 \rangle; \langle z, 1 \rangle; \langle u, 4 \rangle$

在  $B$  中, valeur ( $z, 1$ ) = 1

$\exists$ : valeur ( $x, p$ )  $\Rightarrow$  未定义 (indefini)

valeur ( $x, \langle y, v \rangle; B$ )  $\Rightarrow$  = 0 si  $x = y$

$\Rightarrow$  valeur ( $x, B$ ) si  $x \neq y$ .

La conditionnelle (条件式) :

e.g. # let  $a = -3$ ;  
val  $a$ : int = -3

boolean 类型表达式, valeur de "True" 或 "False"

# let  $x = \text{if } a > 0 \text{ then } a \text{ else } -a$ ;

// "then" 和 "else" 必须是

$x$ : int = 3.

↑ ↑  
"true" "false"

同一种类型

条件式的逻辑运算符是关系运算符 (les opérateurs relationnels) 和布尔运算符 (les opérateurs booléens)

关系运算符: "< ", "> ", " = ", ">= ", "<=" , "<>"

布尔运算符: " && ", " || ", " not "

Les fonctions:

函数的规范 (specification) 由“合约”(contrat) 和测试 (test unitaires) 构成。  
任何定义的函数必须包含所有其规范。

合约 (contract) 包括:

- 名称及其类型
- 该函数的作用
- 标准情况下函数输出的预计结果及其类型
- 预计错误的列表

Définition de fonction: OCaml 是一种可将函数作为参数的语言。函数也有类型，操作范围用

e.g. # fun x → if  $x \geq 0$  then  $x$  else  $-x$ ;;

- int → int = <fun> //  $x$  称为函数的形参 (paramètre formel)

# let val valeur\_absolue = fun x → if  $x \geq 0$  then  $x$  else  $-x$ ;;

(\* 我们可以将标识符与函数关联 \*)

Les fonctions à plusieurs paramètres:

多参数函数

e.g. # let divise x y =  $y \bmod x = 0$ ;;  
val divise : int → int → bool = <fun>

c'est-à-dire



右关联 int → (int → bool)

# let divise = fun x → fun y →  $y \bmod x = 0$ ;;

# let pair x = divise x x;; (\* 函数的调用 \*)

# let carré x = x \* x;;

val carré : int → int = <fun>

# let puissance\_cinq x = x \* Carré (carré x);;

N-uplets:

元组是一种数据结构。元组是一个有序集合，其中的值可以是不同的类型。

元组

可以使用逗号连接各个值来创建一个元组。

e.g. # 1, true;;

- : int \* bool = (1, true)

元组是复合数据结构。这种类型的的数据都可以分解为部分。这些

部分可以被访问器 (accessoires) 检索。

# UB - Programmation Fonctionnelle

## Cours 1. (Sérice)

first

Second.

Accès aux composantes d'une paire: 函数 "fst" 和 "snd" 分别允许访问第 1 和第 2 部分。

e.g. # fst (1, "boom");; -: int = 1

# snd (1, (2, "boom")); -: int \* float string = (2, "boom")

当函数有多个参数时, 可以分别访问, 也可以组合成一个元组, 但最好还是避免处理元组, 除非元组中的参数意义相同。

Filtrage (过滤器): 对于除对 (pairs) 以外的元组, 没有访问器, 我们必须使用另一种机制: 过滤器 (filtering).

简单过滤器: 将多个参数的元组 → 由多个对组成的元组

e.g. # let t = 1, (1, 2);; - t: int \* (int \* int) = (1, (1, 2))

# let (x, y) = t - x: int = 1 - y: int = (1, 2)

Filtrage partial (部分过滤): 用下划线 "\_" 来执行, 不会创建链接 (环境中没有添加任何东西)

e.g. # let (x, \_, z) = (1, "toto", (5, b));;

- x: int = 1 - z: int \* int = (5, b)

\* 注意: 元组要和表达式类型匹配。

如 # let (x, y) = (1, "toto");;

type 'a \* 'b \* 'c - type int \* string.

"match" ... "with" ... :

# UB - Programmation Fonctionnelle

## Cours 2.

La récursivité: 函数可以是递归的，即函数标识符可以在自己的定义中。

(递归)

e.g. # let rec fact n = (\* 所有的递归是义\*)  
if n=0 then 1 else n \* fact(n-1);;

\* 为了让标识符出现在自己的定义中，必须使用关键字“rec”。

L'algorithme de récursivité: But = 定义一系列基本操作以有效地从数据 D 处理为结果 R。  
(递归算法)

Méthodologie: décomposition de problèmes (分解问题)

- Méthode descendante (自上而下的方法)
- Méthode ascendante (自下而上的方法)
- Souvent un mélange des deux.

问题的功能分解基于已经看到的控制结构的引入：定义、条件、过滤、函数的调用/组合，尤其是递归分析。

当我们有一个递归模式作为规范，我们可以简单的把模式(schemas)转换为递归函数，如阶乘、斐波那契等。

Termination: 编写递归函数的主要问题之一就是确保它可以被终止。  
(终止)

我们有一个↑  
严格的递归的整数列，以确保可以终止。

cas terminaux =  $n=0$

cas généraux =  $n-1$

Complexité des algorithmes: 复杂度是算法 A 执行时间的复杂度。  
(算法复杂度)

Intérêts: • 预测执行时间或所需内存 • 知道数据是否可处理。

• 如果算法使用次数不多或用在小数据上，则优先选择易于编写、调试。

维护的算法

• 如果算法在大规模数据或在有限时间内执行多次，则优先选择可以快及执行或低内存占用的算法。

Pour une donnée de taille  $n$ :  $C_{\min}^A(n)$  : 在最好的情况下

$C_{\text{avg}}^A(n)$  : 平均情况下

$C_{\max}^A(n)$  : 在最坏情况下

我们一般采用最坏情况下的复杂度作为算法复杂度。

Optimalité: 如果没有一个算法的复杂度低于当前算法复杂度，我们称当前  
(最优性) 算法为最优算法。

# WB - Programmation Fonctionnelle

## Cours 3.

### Structure de données : Liste

Une  $\alpha$ -liste est :

- soit la liste vide
- soit  $a::l$ ,  $a$  est un  $\alpha$  et  $l$  est une  $\alpha$ -liste.

- 注意：
- 所有的元素都有相同的类型  $\alpha$ .
  - 非空列表总是采用  $head :: queue$  形式。对于表没有直接索引访问。
  - 我们可以“添加”或“删除”元素。

"[]" 表示空列表

Ainsi à la tête et

à la queue de la liste = 访问头部 =  $List.head$  或 读取首元素.

访问尾部 =  $List.tail$

e.g. # let rec somme-liste liste =  
match liste with

| [] → 0

(\*基本情况\*)

| hd :: tl → hd + somme-liste tl; (\*归约情况\*)

val somme-liste : int list → int = fun

# somme-liste [1; 2; 3];;

- : int = 6

hd tl

somme-liste [1; 2; 3]

= 1 + (somme-liste [2; 3])

hd tl

= 1 + (2 + (somme-liste [3]))

hd tl

= 1 + (2 + (3 + (somme-liste [])))

o

= 1 + (2 + (3 + (0))) = 6

List.map :  $List.map$  会取一个列表，另外取一个函数用来转换该列表元素，然后

2 Lists  $\rightarrow$  1 List 返回另一个列表。

e.g. # List.map ~f: String.length ["Hello"; "World!"];;  
- : int list = [5; 6]

List.map2-exn : 与  $List.map$  类似，它取两个列表和一个函数作为参数，这个函数用于结合这两个列表。  
2  $\rightarrow$  1. 当列表长度不一致时会报错。

e.g. # List.map2-exn ~f: Int.max [1; 2; 3] [3; 2; 1];;  
- : int list = [3; 2; 3]

\*如果两个列表长度不同，会抛出一个异常。

List.fold :  $List.fold$  取3个参数：

- 处理的列表
- 一个初始的累加器值
- 用来更新这个累加器的函数

`List.fold` 会从左到右遍历这个列表，每一步都会更新累加器，并返回完成更新时累加器的最终值。

`String.concat` 和 `\`：`String.concat` 处理字符串列表，`\`` 是一个成对处理字符串的操作符。  
避免用`\``连接大量字符串，因为每次运行时，它都会分配一个新字符串。

# UG - Programmation Fonctionnelle

## Cours 4

<1>

Définition de  
type (类型定义)

- 如果要是的数据类型只有一种情况，则可以避免使用构造函数。  
e.g. 由树对列表建立的 file:

type 'a !! file!! = ['a list \* ''a list];

- 常量构造体:

e.g. type color = [Blue | Red | Green];  
↑ "标识符" 小写开头    ↑ "构造体" 大写开头.

type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche;

let chance (j, d) = match j, d with | Not chance (j, d),  
| Vendredi, 13 → true  
| \_ → false;;  
j = Vendredi &  
d = 13;;

val chance : jour \* int → bool;;

- 带参数的构造体:

e.g. # type num = Entier of int | Flottant of float;;  
+ 参数 argument ↪

# num Entier 1;;

-: num = Entier 1;; <有一个值> <没有解决多值性>

\* option 类型是一种数据结构，用于指定正常情况和异常情况的选择。

可以将其与其他数据结构并行： - une paire encode 2 valeurs.

- un n-uplet encode n valeurs. - une liste encode de 0 à n valeurs.

- une option encode 0 ou 1 valeur.

语法: type 'a option = | None | Some of 'a;;

Arbre binaire

二叉树

我们可以有一个经典的二叉树类型，它包括：节点，分支，叶子  
<node> <branche> <feuille>

type 'a standard-tree =

| Empty

左子树

右子树

| Node of ['a \* ['a standard-tree] \* ['a standard-tree]]

Exemple: Node ("x1",

Node ("x2",

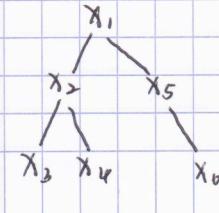
Node ("x3", Empty, Empty),

Node ("x4", Empty, Empty)),

Node ("x5", Empty,

Node ("x6", Empty, Empty))

,



&lt;2&gt;

type 'a edge-tree =

根据例子树的  
分支中:

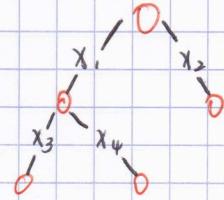
| Empty

| Edges of ('a \* 'a edge-tree) \* ('a \* 'a edge-tree);

Example: Edges((x1,

Edges((x3, Empty), (x4, Empty)),

(x2, Empty));



type 'a leaf-tree =

| Empty

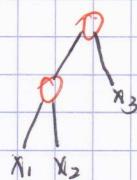
| Node of ('a leaf-tree \* 'a leaf-tree);

Example:

Node (Node (Leaf "x1",

Leaf "x2"),

Leaf "x3");



计算树的大小

&lt;Cardinal&gt; :

(\* Cardinal : 'a standard-tree  $\rightarrow$  int \*)

(\* Renvoie le nb des éléments d'un arbre \*)

let rec cardinal tree =

match tree with

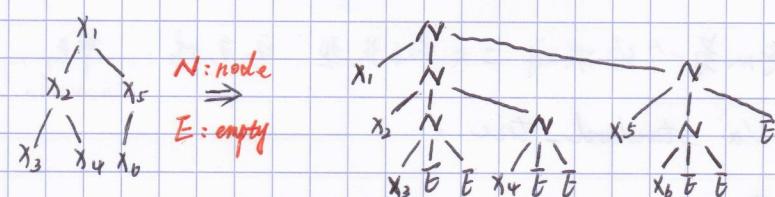
| Empty  $\rightarrow$  0| Node (l, r)  $\rightarrow$  1 + cardinal l + cardinal r ;(\* tree-map : ('a  $\rightarrow$  'b)  $\rightarrow$  'a standard-tree  $\rightarrow$  'b standard-tree)

let rec tree-map f tree = match tree with

| Empty  $\rightarrow$  Empty| Node (n, l, r)  $\rightarrow$  Node(f n, (tree-map f l), (tree-map f r));

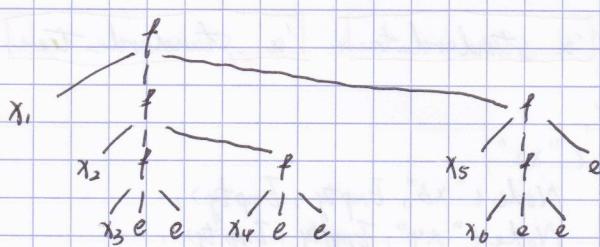
tree-map :

&lt;#List.map&gt;



tree-fold :

&lt;#List.fold-right&gt;



# UG - Programmation Fonctionnelle

## Cours 4 < Snode >

<3>

tree-fold  
< Snode >

\* tree-fold :  $c \rightarrow 'b \rightarrow 'b \rightarrow 'b \rightarrow 'a$  standard-tree  $\rightarrow 'b$  \*

let rec tree-fold f e arb = match arb with

- | Empty  $\rightarrow$  e
- | Node (n, l, r)  $\rightarrow$  f n (tree-fold f e l) (tree-fold f e r);

\* 这段 f 接受三个匹配的参数：- un contenu du noeud  
- un fils gauche "déjà traité" - un fils droite "déjà traité"

计算树的大小

< 基于 fold >

let cardinal arb =

tree-fold (fun cardinal-g cardinal-d  $\rightarrow$  1 + cardinal-g + cardinal-d  
 累加器初值  
 | arb);

↓  
n 的性质为 "累加器" ->

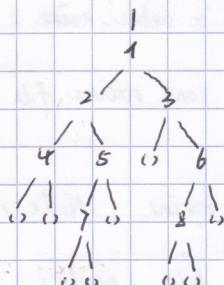


不懂： Cours 4 Partie 3: Arbre Binaire à Gauche

Parcours d'arbres binaires: 前序 (Parcour préfixe): 根在其子级之前被处理，根  $\rightarrow$  左子树  $\rightarrow$  右子树  
< 树的遍历 >

中序 (Parcour infixe): 左子树  $\rightarrow$  根  $\rightarrow$  右子树

后序 (Parcour postfixe): 根在其子级之后处理，左子树  $\rightarrow$  右子树  $\rightarrow$  根



前序遍历: [1; 2; 4; 5; 7; 3; 6; 8]

中序遍历: [4; 2; 7; 5; 1; 3; 8; 6]

后序遍历: [4; 7; 5; 2; 8; 6; 3; 1]

对应的函数定义:

前序: let parcour-profondeur-prefixe tree =

tree-fold (fun racine ppp-g ppp-d  $\rightarrow$  racine :: (ppp-g @ ppp-d)) []

中序: let parcour-profondeur-infixe tree =

tree-fold (fun racine ppi-g ppi-d  $\rightarrow$  ppi-g @ (racine :: ppi-d)) [] tree;

后序: let parcour-profondeur-postfixe tree =

tree-fold (fun racine ppp-g ppp-d  $\rightarrow$  ppp-g @ (ppp-d @ [racine])) [] tree;

使用堆栈 pile 定义:

let parcour-prefixe tree =

let rec parcour pile = match pile with

| []  $\rightarrow$  []

| Empty :: tl  $\rightarrow$  parcour tl

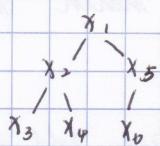
| Node (n, g, d) :: tl  $\rightarrow$  n :: parcour g :: d :: tl

in parcour [tree];

&lt;4&gt;

广度优先遍历 (Parcours en largeur) : “逐行”遍历树的节点

tree :



list :

$$[x_1; x_2; x_5; x_3; x_4; x_6]$$

&lt;1&gt; 根出现在子树之前

&lt;2&gt; 子树的根在子树的根之后

&lt;3&gt; 子树的节点在子树的根之后。

函数定义：

let parcour\_largeur tree =

let rec parcour file = match file with

| []

→ []

| Empty :: tl

→ parcour tl

| Node (n, g, d) :: tl

→ n :: parcour (tl @ [g; d])

in parcour [tree];

Arbre n-aire

(N叉树)

拥有任意数量 (n) 子树。

Arbre-naïre . mli :

type 'a arbre-naïre

val cons : 'a → 'a arbre-naïre list  
→ 'a arbre-naïre

val racine : 'a arbre-naïre → 'a

val fils : 'a arbre-naïre →  
'a arbre-naïre listval fold : ('a → 'b list → 'b) →  
'a arbre-naïre → 'bval map : ('a → 'b) → 'a arbre-naïre  
→ 'b arbre-naïre

Arbre-naïre . ml

type 'a arbre-naïre = | Node of 'a \* 'a arbre-naïre

let cons racine fils = Node (racine, fils)

let racine (Node (racine, \_)) = racine

let fils (Node (\_, fils)) = fils

ref

let fold fNode (Node (racine, fils)) =  
fNode racine (List.map (fold fNode) fils)let rec map f (Node (racine, fils)) =  
Node (f racine, List.map (map f) fils)

on

let map f tree =  
fold (fun racine list-fils →  
Node (f racine, list-map-fils)) tree

计算树的大小

&lt;Cardinal&gt;

let cardinal tree =

fold ( - list-cardinal-fils ) → 1 + List.fold-right (+) list-cardinal-fils 0

List.fold-right 里的函数,

↑ 累加

tree;;

# ÜB - Programmation Fonctionnelle

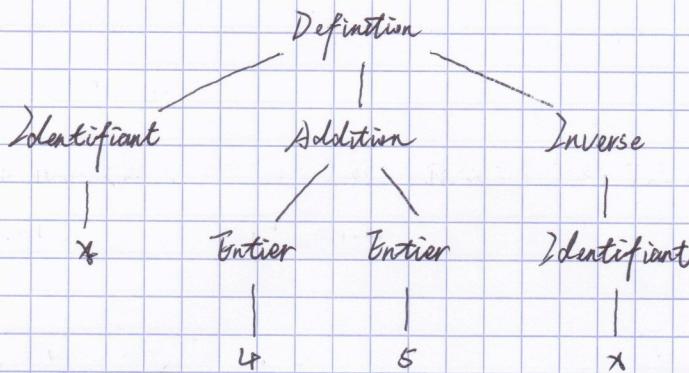
Cours 4

< Suisse >

< 5 >

Evaluateur  
d'expression:

表达式: let  $x = 4 + 5$  in  $-x$  语义树表示:



type expression =

- | Definition of string \* expression \* expression
- | Addition of expression \* expression
- | Subtraction of expression \* expression
- | Inverse of expression
- | Identifiant of string
- | Entier of int

let evalue expression =

let rec aux expression env = match expression with

- | Definition (i, def, e)  $\rightarrow$  aux e ((i, aux def env) :: env)
- | Addition (e1, e2)  $\rightarrow$  (aux e1 env) + (aux e2 env)
- | Subtraction (e1, e2)  $\rightarrow$  (aux e1 env) - (aux e2 env)
- | Inverse e  $\rightarrow$  - (aux e env)
- | Identifiant id  $\rightarrow$  List\_assoc id env.
- | Entier i  $\rightarrow$  i

in aux expression [];;

(\* test \*) (\* let  $x = 4 + 5$  in  $-x$  \*)

# evalue (Definition ("x"), Addition ((Entier 4), (Entier 5))), Inverse (Identifiant

"x")));;

-: int = -9

# UG - Programmation Fonctionnelle

## Cours 5: Typage avancé

Types fantômes: un type fantôme est un type paramétrisé dont au moins un des paramètres n'apparaît pas dans les types des valeurs de ce type.

e.g. type 'a t = int

- 表示内部状态或隐藏状态。

若我们要强制读取文件的第一个字符，我们定义 FichierLecture1Car 为：

```
module type FichierLecture1Car =
```

```
sig
```

```
type debut
```

```
type fin
```

```
type - fichier.
```

(+类型的参数 - fichier, 取值 debut 和 fin, 定义 fichier 的内部

状态)

```
val open: string → debut fichier
```

```
val read: debut fichier → char * fin fichier
```

```
val close: fin fichier → unit
```

```
end.
```

Réalisation:

```
module Impl: FichierLecture1Car =
```

```
struct
```

```
type debut = unit
```

```
type fin = unit
```

```
type - fichier = in-channel
```

```
end.
```

词法OCaml 中的文件操作

```
let open mem = open-in mem
```

```
let read f = cinput-char f, f,
```

```
let close f = close-in f
```

Types uniques: un type unique est associé à une valeur dans tout le programme. Un type unique et sa valeur sont créés ensemble.

值

type 的定义

e.g. type unique = Unique = 'a → unique;; (+ GADT &)

用处：表示或隔离单独（unique）的数据。

通常和 type fantôme 一起用。

\* 密钥 Pk 的 RSA 密钥

Types non uniformes: 可以看作是统一的类型。

用处：表示“被确定之后的结构不变”。 meilleure specification

e.g. type ('a, 'b) alt-list = | Nil | Cons of 'a \* ('a, 'b) alt-list 交替列表

```
let rec to-alt-list: ('a + 'b) list → ('a, 'b) alt-list =  
function | [] → Nil
```

```
| ('a, b)::tl → Cons ('a, Cons ('b, to-alt-list tl));;
```

```
let rec from-alt-list: ('a, 'b) alt-list → 'a list * 'b list =  
function | Nil → ([], [])
```

```
| Cons ('a, q) → let (q1, q2) = from-alt-list q in ('a::q1, q2);;
```

Types algébriques généralisés : Generalized Algebraic Data Type 允许我们自由选择类型的参数.

e.g. type - repr =

$$\begin{array}{l} | \text{Int} : \text{int} \rightarrow \text{int repr} \\ | \text{Add} : (\text{int} \rightarrow \text{int} \rightarrow \text{int}) \text{ repr} \end{array} \xrightarrow{\text{réaliser}} \begin{array}{l} | \text{Int } i \rightarrow i \\ | \text{Add} \rightarrow (\text{fun } a \ b \rightarrow a + b) \end{array}$$

好处： + 非统一类型的泛化  
+ 可以表示单性的类型  
+ 可以表示运行时的类型信息 Run Time Type Information

# ÜB - Programmation Fonctionnelle

## Cours 6 : Structure monadiques

bind : val bind : 'a option  $\rightarrow$  ('a  $\rightarrow$  'b option)  $\rightarrow$  'b option

val (>>=) = let bind e f

List. flatten ([ [1]; [2;3]; [4] ]) = [1;2;3;4]

">=>" : val (>=>) = (a  $\rightarrow$  'b t)  $\rightarrow$  ('b  $\rightarrow$  'c t)  $\rightarrow$  'a  $\rightarrow$  'c t

e.g. id >=> ifm (r)  $\Rightarrow$  return (f r)) = map f

monade "listie" : type 'a t = 'a list

let map f l = List.map f l

let return e = [e]

let rec (>>=) & f = match l with

| []  $\Rightarrow$  []

| t :: q  $\Rightarrow$  t f @ (q >>= f)

let bind f l f = List.flatten (List.map f l);;