Cours numéro 3: Programmation impérative

LI213 – Types et Structures de données

Christophe Gonzales – Pierre-Henri Wuillemin

Licence d'Informatique – Université Paris 6

Les tableaux (ou vecteurs) (1/3)

Le type `tableau` est une des réponses à la question :

comment agréger un très grand nombre de données de même type ?

- Représenter un point 3D (N-uplet – cf. cours4) :

```
# let p3D=(3.,2.,3.);;
```

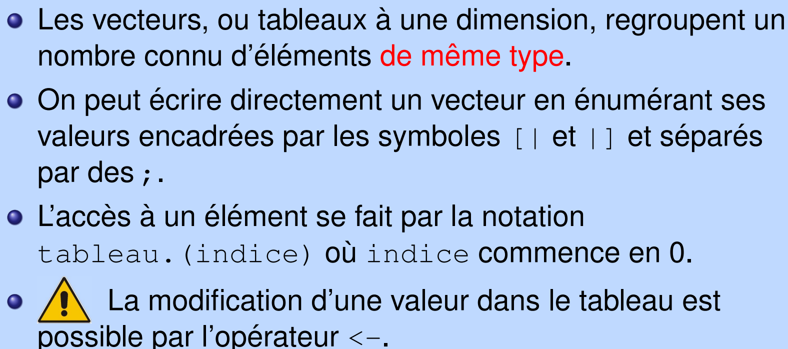
  Comment représenter un polygone défini par un grand nombre de points ?

- Représenter un descripteur de fichier (enregistrement cf. cours 4) :

```
# type fichier =
  { nom :string; taille :int;...};;
```

  Comment représenter le contenu d'un répertoire ?

Les tableaux (2/3)



définition

- Les vecteurs, ou tableaux à une dimension, regroupent un nombre connu d'éléments de même type.
- On peut écrire directement un vecteur en énumérant ses valeurs encadrées par les symboles `[|` et `|]` et séparés par des `;`.
- L'accès à un élément se fait par la notation `tableau.(indice)` où `indice` commence en 0.
- ⚠ La modification d'une valeur dans le tableau est possible par l'opérateur `<-`.

(cf. enregistrement mutable / cours4)

Les tableaux (3/3)

0	1	2	3	4
1.0	1.5	~~3.2~~ 3.0	2.9	56.6

```
# let a=[| 1.; 1.5; 3.2; 2.9; 56.6 |];;
val a : float array = [|1.; 1.5; 3.2; 2.9; 56.6|]
# a.(2);;
- : float = 3.2
# a.(5);;
Exception : Invalid_argument "index out of bounds".
# a.(2)<-2. *. a.(1)
```

pas de let ;;

```
- : unit = ()
# a;;
- : float array = [|1.; 1.5; 3.; 2.9; 56.6|]
```

Tableaux : valeur ou variable ?

- Le tableau est une valeur.

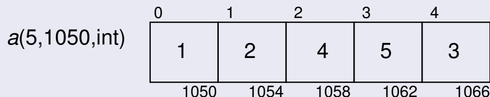
```
# let a= [|1;2;4;5;3|];;
```

- pourtant on peut changer son contenu.

```
# a.(2)<-5;;
```

(instruction d'affectation <- et non le =)

Petite digression “à la C” : adresses (et pointeurs)



- `a` : «zone de mémoire ; début :1050 ; taille :5×TINT»
- `a.(i)` : «entier situé en début+i×TINT ; i<taille»
- `a.(i)<-...` : «changer la valeur en mémoire, à l'adresse début+i×TINT»

Manipulation sur les tableaux

Les fonctions dédiées à la manipulation des tableaux font partie du module Array de la bibliothèque standard.

- **Array.create** `taille valeur`

```
# let monTableau=Array.create 3 2.0;;
val monTableau : float array = [|2.; 2.; 2.|]
```

- **Array.length** `tableau`

```
# Array.length monTableau;;
- : int = 3
```

- **Array.copy** `tableau`

```
# let a=Array.copy monTableau;;
val a : float array = [|2.; 2.; 2.|]
```

- **Array.append** `tableau1 tableau2`

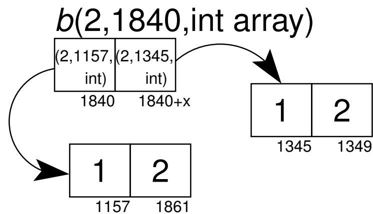
```
# let b=Array.append monTableau a;;
val b : float array = [|2.; 2.; 2.;2.; 2.; 2.|]
```

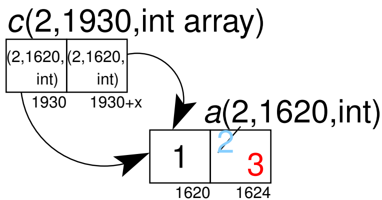
Pas de restrictions sur les types des cellules d'un tableau...

```
# let pascal = [| [| 1; 1 |];
                  [| 1; 2; 0 |];
                  [| 1; 3; 3; 1 |];
                  [| 1; 4; 6; 4; 1 |]
                |];;
val pascal : int array array = ...
# pascal.(0);;
- : int array = [| 1; 1 |]
# pascal.(1).(1);;
- : int = 2
# pascal.(1)<- [| 1; 2; 1 |];;
# pascal.(1).(2)<-1;;
```

Tableaux de tableaux (2/2)

```
# let a=[|1; 2 |];;
val a : int array = [|1; 2|]
# let b=[| [|1; 2 |]; [|1; 2 |] |];;
val b : int array array = [|[|1; 2|]; [|1; 2|]|]
# let c=[| a; a |];;
val c : int array array = [|[|1; 2|]; [|1; 2|]|]
```






```
# a.(1)<- 3;;
# c;;
```

```
val c : int array array =
  [|[|1; 3|]; [|1; 3|]|]
```

Le type string

Les chaînes de caractères peuvent être considérées comme un cas spécial de tableaux de caractères. Néanmoins, leur type est spécialisé et l'accès aux éléments possède une syntaxe particulière.

Chaîne de caractères

```
# let s = "hello world!";;
val s : string = "hello world!"
# s.[0];;
- : char = 'h'
# s.[5]<-' !';;
- : unit = ()
# s;;
- : string ="hello!world!"
# String.length s;;
- : int = 12
```

Programmation fonctionnelle et impérative

Programmation impérative : modèle de Turing

- Programme = données + traitements/transformations.
- Mécanismes de portées (locales/globales) et d'agrégations (procédure, OO) : simplification de la lecture des programmes.
- Une partie des données servent à décrire l'état du programme.
- Une procédure peut modifier des données de portée supérieure à celle de la procédure : effet de bord.

Programmation fonctionnelle

- Programme = application d'une série de fonctions sur les données d'entrée pour obtenir les sorties escomptées.
- Pas d'état pour le programme. Pas de modification de valeur. Pas d'effet de bord.

Pourquoi s'intéresser à la prog. fonctionnelle ?

Inconvénients de la programmation impérative

- Effets de bord : difficultés et bugs assurés
- État du programme : difficultés et bugs assurés

Avantages de la programmation fonctionnelle

- Assurance "mathématique" : le résultat ne dépend que des données d'entrée.
- **Transparence référentielle** : le résultat ne change pas, pour des valeurs d'entrées égales.
- Maintenance, test unitaire, parallélisation facilités par tout ça.

De l'eau dans son vin

Tous les langages fonctionnels modernes intègrent des traits d'impératif, pour faciliter certaines expressions.

- Les valeurs mutables et les références permettent quand même les effets de bord locaux, si besoin.
- Jeu d'instructions de type impératif (séquence, boucle `for`, boucle `while`, ...)

Récursif un peu bête

```
# let affiche v =
  let rec aff i v =
   if (i<length v) then (
    print_int v.(i) ;
    print_string " ";
    aff (i+1) v
   ) else ()
  in aff 0 v;;
```

Impératif un peu correct

```
# let affiche v =
  for i=0 to
      (length v - 1) do
   (
    print_int v.(i) ;
    print_string " "
   )
  done;;
```

Fonctions et instructions

En OCAML(langage fonctionnel), tout est expression. Donc toute compilation aboutit à une évaluation d'un résultat.

```
# 3+3;;
# let x=4.;;
# let f x=x.[1];;
```

```
- : int = 6
val x : float =4.
val f : string -> char
    = <fun>
```

Que faire des instructions ?

affectation	`# v.(5)<-4.3;;`
Entrée/sortie	
affichage d'une chaîne	`# print_string "hello!";;`
affichage d'un entier	`# print_int 3;;`
affichage d'un flottant	`# print_float 2.12;;`

La valeur ()

Le type `unit` n'a que `()` comme valeur possible.

() comme résultat

```
# x.(3)<-4.;;
- : unit = ()
```

L'opération est effectuée. Pas de résultat.

La valeur () peut également servir comme paramètre (indiquant une fonction sans paramètre)

() comme paramètre

```
# print_newline;;
- : unit->unit = <fun>
# print_newline ();;

- : unit = ()
```

() comme paramètre

```
# let f = 3;;
val f : int = 3
# let f () = 3;;
val f : unit->int
     = <fun>
```

unit->[type] et [type]->unit

Exemples de fonction `unit->[type]` ? et `[type]->unit` ?

unit -> [type] : input

```
# read_line;;
- : unit -> string=<fun>
# read_int;;
- : unit -> int=<fun>
# read_float;;
;- : unit -> float=<fun>
```

unit -> unit : output

```
# print_newline;;
- : unit -> unit=<fun>
```

[type]->unit : output

```
# print_char;;
- : char -> unit=<fun>
# print_string;;
- : string -> unit=<fun>
# print_int;;
- : int -> unit=<fun>
# print_float;;
- : float -> unit=<fun>
# print_endline;;
- : string -> unit=<fun>
```

La séquence

Définition

- La séquence est une expression composée d'une suite d'expressions, séparées par des " `;` " et qui seront évaluées l'une après l'autre (de gauche à droite).
- Le type et la valeur de la séquence sont ceux de la dernière expression de la liste.
- Toute autre expression de la liste devrait être de type `unit`.
- Séquence parenthésée par des `()` ou par `begin / end`.

```
# print_string "bonjour, 3+3="; 3+3;;
bonjour, 3+3 = - : int = 6
# 2+2;3+3;;
Warning : this expression should have type unit.
: - int = 6
```

La boucle «for»

Boucle for

- **for** nom = expr1 **to** expr2 **do** expr3 **done**
- **for** nom = expr1 **downto** expr2 **do** expr3 **done**

expr1 et expr2 sont des int.
expr3 est un unit.
nom est une valeur nommée dans l'environnement de expr3 ! !

```
# for i=1 to 10 do print_int i; print_string " " done;
     print_newline();;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
# for i=10 downto 1 do print_int i done;
     print_newline();;
10987654321
- : unit = ()
```

La boucle « `for` »

Boucle `for`

for `nom = expr1` **to** `expr2` **do** `expr3` **done**

Question : quand sont évaluées `expr1` et `expr2` ?

Boucle et séquence

Lorsqu'une expression est nécessaire, une séquence de même type peut être utilisée !

```
# for i = print_string "A "; 1
    to print_string "B ";5 do
     print_int i
   done;;
A B 12345- : unit = ()
```

Et le `tant que` ?

En algorithmique, ces 2 programmes sont équivalents :

Boucle `for`

```
pour i = 1 jusqu'à 5
faire
  écrire i
fin pour
```

```
# for i = 1 to 5 do
    print_int i
  done;;
```

Boucle `tant que`

```
i = 1
tant que i<=5 faire
  écrire i
  i=i+1
fin tant que
```

?

```
while expr1 : bool do
  expr2   : unit
done
```

Nécessite que `expr2` modifie le contexte de `expr1` ...

Effet de bord

(Enfin !) des effets de bord

Question : peut-on faire des effets de bord maintenant ?
Réponse : **oui !** Grâce à séquence + valeurs mutables
valeurs mutables = cellule de tableaux (ou champs mutables cf. cours 4)

Effet de bord

```
# let a = [|1|];;
val a : int array = [|1|]
# for i=1 to 10 do
      a.(0)<-a.(0)+2;print_int i
  done;;
12345678910- : unit = ()
# a;;
- : int array = [|21|]
```

Référence

- On sait faire des valeurs mutables (création, accès, modification).

```
# let a = [|1|];;
val a : int array = [|1|]
# a.(0);;
- : int = 1
# a.(0)<-5;;
- : unit = ()
```

Plus direct : les références

Création	`let x =[|12|]`	`let x = ref 12`
Accès	`x.(0)`	`!x`
Modification	`x.(0)<-x.(0)+1`	`x :=!x + 1`

La boucle «while»

Boucle `while`

while `expr1` **do** `expr2` **done**

`expr1` est de type `bool`.

`exp2` est de type `unit`.

```
#  for i = 1 to 10 do
     print_int i;
     print_string " "
   done ;;
```

```
#  let i = ref 1 in
     while !i <=10 do
       print_int !i;
       print_string " ";
       i :=!i + 1
     done;;
```

Programmation fonctionnelle ou impérative ?

- Privilégier la programmation fonctionnelle
- Utiliser la programmation impérative si besoin (`séquence` ou `for`)
- Savoir mixer si besoin

Générateur d'identifiants uniques

But : Écrire `initId : ()->()` et `getId : ()->string` qui, à chaque appel, fournit un identifiant unique :

```
# initId();;
- : unit = ()
# getId();;
- : string = "i1"
# getId();;
- : string = "i2" ...
```

1. Version purement fonctionnelle ?
   Impossible : utilise un effet de bord.
2. Version impérative alors :

besoin d'une variable globale !

```
# let idValue = ref 0;;
```

```
# let initId () = idValue := 0;;
val initId : unit -> unit = <fun>
```

```
# let getId () = idValue :=!idValue+1 ;
        "i" ^ (string_of_int !idValue);;
val getId : unit -> string = <fun>
```

Mais c'est vraiment très laid !
`idValue :=...` est modifiable pour tous ! !

initId() et getId() : version mixte

but : utiliser une référence (prog. impérative) mais utiliser la prog. fonctionnelle pour *encapsuler* la référence.

```
# let idFactory () =
    let idValue = ref 0  in
       let initId() = idValue :=0
       and getId() = idValue :=!idValue+1;
                      "i" ^ (string_of_int !idValue)
       in (initId,getId);;
val idFactory : unit -> (unit -> unit) *
                (unit -> string) = <fun>
```

```
# let (iA,gA) = idFactory();;
```

```
# let (iB,gB) = idFactory();;
```

```
# iA ();gA();;
- : string = "i1"
```

```
# iB ();gA();;
- : string = "i2"
```

```
# gA() ^ gB();;
- : string = "i3i1"
```

```
# let f x y = x / y;;
val f : int -> int -> int = <fun>
```

Quelle est la valeur de `f 3 0` ou de `f (-1) 0` ?
On aimerait retourner $\pm\infty$ mais aucun `int` n'encode $+\infty$ ou $-\infty$

Quelle est la valeur de `f 0 0` ?
On aimerait retourner une «valeur indéfinie» mais aucun `int` n'encode cette valeur

Problème : que doit-on faire ?
Solution :

1. ne surtout pas retourner de valeur `int`
2. trouver un mécanisme indiquant que l'on ne peut retourner de valeur «valide»

$\Longrightarrow$ c'est le mécanisme des exceptions.

Les exceptions

```
# let f x y = x / y;;
  let y = f 10 3;;
  let z = f 3 0;;
val f : int -> int -> int = <fun>
y : int = 3
Exception : Division_by_zero
```

Remarque : pas de « `z :` » devant `Exception`

⟹ pas de liaison `(z,Division_by_zero)`

⟹ `Division_by_zero` n'est pas une valeur retournée

Description d'une exception

- une valeur de type `exn`.
- on ne la retourne pas : on la lève (`raise`)
- on ne crée pas de liaison dont elle est la valeur : on la rattrape (`try... with`)

Le mécanisme des exceptions (1/3)

Description du mécanisme

1. Quand une exception est levée (via un `raise`), tous les calculs en suspens sont arrêtés.
2. On remonte récursivement les instructions qui ont entraîné ces calculs jusqu'à rencontrer un `try... with` qui englobe ces instructions et récupère l'exception.
3. Si on ne trouve pas de `try... with`, le programme se termine.

```
# let f x y = print_int (x / y);;
  f 4 0;;
  f 10 2;;
```

```
val f : int -> int -> unit = <fun>
Exception : Division_by_zero.
```

pas de `try... with` $\Longrightarrow$ on n'exécute pas `f 10 2`

Le mécanisme des exceptions (2/3)

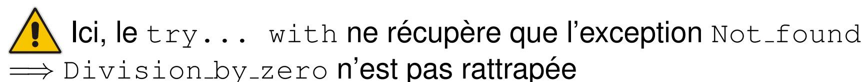
```
# let f x y = x / y;;
  let g x y = 1 + f x y;;
  let h x y = try print_int (g x y) with
     Division_by_zero ->
           print_string "erreur\n";;
  h 1 0;;
val f : int -> int -> int = <fun>
val g : int -> int -> int = <fun>
val h : int -> int -> unit = <fun>
erreur
- : unit = ()
```

1. `(g 1 0)` appelle `f 1 0` qui lève l'exception
2. exception non rattrapée (pas de `try... with`) $\Longrightarrow$ on remonte (sans les effectuer) les calculs de `f` en `g` puis en `h`
3. on remonte au `try... with Division_by_zero`
4. exception rattrapée $\Longrightarrow$ exécution du `print_string`

Le mécanisme des exceptions (3/3)

```
# let f x y = (x / y);;
  let g x y = 1 + f x y;;
  let h x y = try print_int (g x y) with
     Not_found -> print_string "erreur\n";;
  h 1 0;;
```

```
val f : int -> int -> int = <fun>
val g : int -> int -> int = <fun>
val h : int -> int -> unit = <fun>
Exception : Division_by_zero.
```



Ici, le `try... with` ne récupère que l'exception `Not_found`
$\Longrightarrow$ `Division_by_zero` n'est pas rattrapée

Créer une nouvelle exception

Déclarer une nouvelle exception sans argument

Il suffit d'utiliser le mot-clé `exception` :

```
# exception Mon_exception_de_la_mort_qui_tue;;
exception Mon_exception_de_la_mort_qui_tue
```

L'exception peut alors être levée/rattrapée

Tout nom d'exception doit commencer par une majuscule

Déclarer une nouvelle exception avec argument

`exception` nom_exception `of` type_argument

```
# exception Exn_with_arg of int;;
exception Exn_with_arg of int
```

Lever une exception

Comment lever une exception sans argument

`raise` nom_exception :

```
# let f x y =
    if y = 0 then raise Division_by_zero
    else x / y;;
val f : int -> int -> int = <fun>
```

Comment lever une exception avec argument

`raise (`nom_exception argument`)` :

```
# let f x y =
    if y = 0 then raise (Failure "message")
    else x / y;;
val f : int -> int -> int = <fun>
```

`raise : exn -> 'a` $\Longrightarrow$ `(raise nom_exception)` est de type `'a` $\Longrightarrow$ peut être utilisé dans n'importe quelle fonction

Le failwith

Une exception très utilisée : *Failure*

```
# raise (Failure "texte");;
```

Trop compliqué à écrire pour un informaticien

$\Longrightarrow$ remplacé par la fonction `failwith` :

```
# let failwith texte = raise (Failure texte);;
```

gain à utiliser `failwith` : on gagne 8 caractères par rapport au `raise`

Si vous tapez à 240 caractères / minute et que vous écrivez 10 `failwith` par jour, alors après 1 an de programmation, vous aurez gagné 2 heures ! ! ! !