IPRF – Introduction, Syntaxe de OCaml

Christophe Mouilleron

ensiie
école nationale supérieure d'informatique
pour l'industrie et l'entreprise

Plan

Plan

Organisation du cours

4 séances de Cours-TP $4 \times 3h30$

Sujets qui pourront être abordés :

1. noyau fonctionnel de OCaml
2. fonctions récursives
3. structures de données (AVLs, associations, graphes)
4. programmation dynamique et mémoïsation
5. backtracking

Évaluation en contrôle continu :

- rendus de certains TPs
- projet à rendre pour le **vendredi 1 février 2019 à 23h59**

Fonctionnel VS impératif VS objets ?

Trolleur « C'est <insérer un paradigme ici> le mieux ! »

Fonctionnel VS impératif VS objets ?

Trolleur	« C'est <insérer un paradigme ici> le mieux ! »
Théoricien	« Tant que le langage est Turing-complet, on peut faire tout ce qu'il est possible de faire avec. »

Fonctionnel VS impératif VS objets ?

Trolleur	« C'est <insérer un paradigme ici> le mieux ! »
Théoricien	« Tant que le langage est Turing-complet, on peut faire tout ce qu'il est possible de faire avec. »
Pragmatique	« Si il y avait un paradigme meilleur que les autres, on n'utiliserait plus que celui-là ! »

Fonctionnel VS impératif VS objets ?

Trolleur	« C'est <insérer un paradigme ici> le mieux ! »
Théoricien	« Tant que le langage est Turing-complet, on peut faire tout ce qu'il est possible de faire avec. »
Pragmatique	« Si il y avait un paradigme meilleur que les autres, on n'utiliserait plus que celui-là ! »
Ingénieur	« Pour un problème donné, je choisis la bonne structure de données et je conçois un bon algorithme. Ensuite, je code avec le paradigme/langage le plus adapté. »

La programmation fonctionnelle

Tout est valeur :

- fonction = valeur comme une autre
- passage de fonctions comme arguments

Persistance :

- une valeur définie n'est JAMAIS modifiée
- très utile en cas de retour en arrière (undo, backtrack)

Programmation proche d'un raisonnement mathématique

- abstraction $\rightsquigarrow$ code compact et clair
- modélisation et preuve formelle plus aisées

Le langage OCaml

Langage d'origine française `https://ocaml.org/`

Caractéristiques :

- Noyau fonctionnel + traits impératifs + couche objet
- Fortement typé, avec typage implicite
- Évaluation stricte — calcul des arguments avant appel de fonction
- Garbage Collector
  - gestion de la mémoire "par le langage"
  - rend la persistance efficace

Plan

Programme OCaml

Expression

- quantité évaluée à l'exécution
- résultat = valeur

Déclaration

- nommage d'une quantité
- résultat = nouvelle fonction / variable globale ou nouveau type

programme OCaml = suite de déclarations et expressions

séparateur = `;;`

commentaires = texte entre `(*` et `*)`

Évaluation d'un programme OCaml

1. Interprétation avec le toplevel :
   - commande Unix `ocaml`
   - mode interactif

2. Compilation
   - commande Unix `ocamlc` ou `ocamlopt`
   - fonctionnement similaire à celui de `gcc`

3. Interprétation avec le mode Tuareg d'Emacs
   - très pratique en TP

Plan

Types de bases

- int = entiers `0, 1, -4`
- float = “réels” `3.14, infinity`
- char = caractères `'c', '\n'`
- string = chaînes de caractères `"une chaine"`
- bool = booléens `true, false`
- le type spécial unit à 1 élément `()`

Autres types

Les tuples = collection d'expressions séparées par ,

```
# 17, "42", 3 < 5 ;;
- : int * string * bool = (17, "42", true)
```

Les listes = suite d'expressions de même type

```
# [] ;;
- : 'a list = []
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
# 1 :: 'b' :: [] ;;
Error: This expression has type char but an
         expression was expected of type int
```

Opérateurs usuels

- opérateurs logiques `||`, `&&`
- arithmétique entière `+`, `-`, `*`, `/`, `mod`
- arithmétique réelle `+.`, `-.`, `*.`, `/.`, `**`
- comparaisons `=`, `==`, `<`, `<>`, `!=`, etc.
- concaténation de listes `@`

Opérateurs usuels (suite)

ATTENTION

Les opérateurs ont un type fixe ...

```
# 3.14 +. 1 ;;
Error: This expression has type int but an
         expression was expected of type float
# 3.14 +. float_of_int 1 ;;
- : float = 4.1400000000000057
```

Opérateurs usuels (suite)

ATTENTION

Les opérateurs ont un type fixe ...

```
# 3.14 +. 1 ;;
Error: This expression has type int but an
         expression was expected of type float
# 3.14 +. float_of_int 1 ;;
- : float = 4.1400000000000057
```

... sauf les opérateurs de comparaison.

```
# (=) ;;
- : 'a -> 'a -> bool = <fun>
```

Égalités logique et physique

Autre subtilité : différence entre = et ==

```
# [1;2] = [1;2] ;;
- : bool = true
# [1;2] == [1;2] ;;
- : bool = false
```

Égalités logique et physique

Autre subtilité : différence entre = et ==

```
# [1;2] = [1;2] ;;
- : bool = true
# [1;2] == [1;2] ;;
- : bool = false

# [1] <> [1] ;;
- : bool = false
# [1] != [1] ;;
- : bool = true
```

logique : = et <> $\rightsquigarrow$ à utiliser de préférence

physique : == et !=

Plan

Branchements

Syntaxe :

```
if test then e1 else e2
```

Exemples :

```
# if 32 <= 52 then "ok" else "soucis" ;;
- : string = "ok"

# 1 + (if 32 <= 52 then 32 else 52) ;;
- : int = 33
```

Branchements (suite)

ATTENTION

1. `test` doit être de type `bool`.
2. Les expressions du **`then`** et du **`else`** doivent être de même type.

```
# if 1 = 1 then "toto" else 42;;
Error: This expression has type int but an
         expression was expected of type string
```

Conséquence : **`if`** toujours accompagné d'un **`else`**

Fonctions anonymes

Syntaxe :

```
fun var -> expr
```

Exemples :

```
# fun x -> 2 * x ;;
- : int -> int = <fun>

# fun x -> fun y -> 2 * x + y ;;
- : int -> int -> int = <fun>
```

Remarque : Les fonctions sont univariés.

Fonctions anonymes (suite)

Passage d'argument :

(**fun** var **->** expr) arg

Exemples :

```
# (fun x -> 2 * x) 21;;
- : int = 42

# (fun x -> fun y -> x + y) 32 52;;
- : int = 84

# (fun x -> fun y -> x + y) 1;;
- : int -> int = <fun>

# (fun x -> 2 * x) 10 + 1;;
- : int = 21
```

Filtrages

Syntaxe :

```
match e with pattern1 -> e1 | ... | patternk -> ek
```

avec `e1`, `...`, `ek` de même type.

Exemples :

```
# match 42::[] with [] -> 0 | x::[] -> 1 | _ -> 2 ;;
- : int = 1

# match 32::52::[] with [] -> 0 | x::_ -> x ;;
- : int = 32
```

note : `_` = joker.

Branchement VS Filtrage

Branchement et filtrage sont deux choses différentes.

if = test sur les valeurs

- utile par exemple pour tester la valeur d'un entier

match = test sur la structure d'une expression

- utile par exemple pour raisonner sur la longueur d'une listes
- permet d'extraire des valeurs en fonction de la structure

Exceptions

Syntaxe :

```
try e with pattern1 -> e1 | ... | patternk -> ek
```

avec `e`, `e1`, `...`, `ek` de même type.

Exemple :

```
# 1 / 0 ;;
Exception: Division_by_zero.

# try 1 / 0 with Division_by_zero -> 0 ;;
- : int = 0
```

Plan

Variables et fonctions globales

Syntaxe :

```
let nom = expr;;
```

Exemples :

```
# let v = 17 + 11 ;;
val v : int = 28

# let inc = fun x -> x + 1 ;;
val inc : int -> int = <fun>

# inc v ;;
- : int = 29
```

Variables et fonctions locales

Syntaxe :

```
let nom = expr1 in expr2;;
```

Exemple (suite du transparent précédent) :

```
# let v = 41 in inc v ;;
- : int = 42

# v ;;
- : int = 28
```

Variables et fonctions locales

Syntaxe :

```
let nom = expr1 in expr2;;
```

Exemple (suite du transparent précédent) :

```
# let v = 41 in inc v ;;
- : int = 42

# v ;;
- : int = 28
```

Note : dans `fun x -> expr`, le `x` est une variable locale à `expr`.
$\rightsquigarrow$ dès qu'on fournit une valeur pour `x`, on déduit celle de `expr`.

Autres usages de `let`

Déclarations simultanées

```
# let x = 17 and y = 42 ;;
val x : int = 17
val y : int = 42
```

Déstructuration

```
# let paire = 1, 'a';;
val paire : int * char = (1, 'a')

# let a, b = paire ;;
val a : int = 1
val b : char = 'a'

# let x :: y :: _ = 1 :: 2 :: 3 :: [] ;;
```

Définitions récursives

En mathématiques, on définit la factorielle de $n \in \mathbb{N}$ par :

$$0! = 1$$
$$n! = n \times (n-1)! \text{ si } n > 0$$

En OCaml :

```
# let rec fact = fun n ->
      if n = 0 then 1
               else n * fact  (n-1) ;;
val fact : int -> int = <fun>

# fact 6 ;;
- : int = 720
```

Attention à ne pas oublier le **rec** !!!

Plan

Calcul de 2^n

Comment faire ?

- pas de boucle . . . mais on a la récursivité $\rightsquigarrow$ **let rec**

$$2^0 = 1$$
$$2^n = 2 \times 2^{n-1} \quad \text{si } n > 0$$

Calcul de 2^n

Comment faire ?

- pas de boucle . . . mais on a la récursivité $\rightsquigarrow$ **let rec**

$$2^0 = 1$$
$$2^n = 2 \times 2^{n-1} \quad \text{si } n > 0$$

Solution :

```
let rec pow2 = fun n ->
    if n = 0 then 1
             else 2 * pow2 (n-1)
  ;;
```

Longueur d'une liste

Comment faire ?

Longueur d'une liste

Comment faire ?

- raisonnement sur la taille d'une liste $\rightsquigarrow$ **match**
- définition récursive $\rightsquigarrow$ **let rec**

$$\text{longueur } [] = 0$$
$$\text{longueur } (h :: t) = 1 + \text{longueur } t$$

Longueur d'une liste

Comment faire ?

- raisonnement sur la taille d'une liste $\rightsquigarrow$ **match**
- définition récursive $\rightsquigarrow$ **let rec**

$$\text{longueur } [] = 0$$
$$\text{longueur } (h :: t) = 1 + \text{longueur } t$$

Solution :

```
let rec longueur = fun l -> match l with
  | [] -> 0
  | _::t -> 1 + longueur t
  ;;
```

`h` = head = début, `t` = tail = suite

Maximum d'une liste d'entiers

Comment faire ?

- définition récursive $\rightsquigarrow$ **let rec**
- raisonnement sur la taille d'une liste $\rightsquigarrow$ **match**
- maximum de `[]` ? ? ?

Maximum d'une liste d'entiers

Comment faire ?

- définition récursive $\rightsquigarrow$ **let rec**
- raisonnement sur la taille d'une liste $\rightsquigarrow$ **match**
- maximum de `[]` ? ? ?

Une solution :

```
(* l doit être non vide *)
let rec maximum = fun l -> match l with
  | [] -> assert false
  | h :: [] -> h
  | h :: t -> let m = maximum t in
                  if h > m then h else m
  ;;
```

Exemple de fonctions mutuellement récursives

Définition mathématique :

- 0 est pair,
- 0 n'est pas impair,
- $n > 0$ est pair lorsque $n - 1$ est impair,
- $n > 0$ est impair lorsque $n - 1$ est pair.

Exemple de fonctions mutuellement récursives

Définition mathématique :

- 0 est pair,
- 0 n'est pas impair,
- $n > 0$ est pair lorsque $n - 1$ est impair,
- $n > 0$ est impair lorsque $n - 1$ est pair.

En OCaml :

```
# let rec even = fun n -> if n = 0 then true
                          else odd (n-1)
      and odd  = fun n -> if n = 0 then false
                          else even (n-1)
  ;;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
```

Plan

Plan

Quelques fonctions prédéfinies

- fonctions sur les entiers `abs`, `succ`, `pred`
- fonctions mathématiques sur les réels `sqrt`, `log`, `sin`, etc.
- quelques constantes `max_int`, `epsilon_float`, etc.
- opérations bits à bits `lor`, `lsl`, `asr`, etc.
- fonctions liées à la comparaison `min`, `max`, `compare`
- négation d'un booléen `not`
- manipulation des couples `fst`, `snd`

Plan

Le module `Printf`

Fournit la fonction `Printf.printf` de syntaxe similaire à celle de `printf` en C.

```
# Printf.printf "%f" (log 2.) ;;
0.693147- : unit = ()

# Printf.printf "%d\n" (32 * 52) ;;
1664
- : unit = ()
```

Le module `Printf`

Fournit la fonction `Printf.printf` de syntaxe similaire à celle de `printf` en C.

```
# Printf.printf "%f" (log 2.) ;;
0.693147- : unit = ()

# Printf.printf "%d\n" (32 * 52) ;;
1664
- : unit = ()
```

Astuce : pour flusher, ajouter `%!`

La fonction `read_line`

Permet de récupérer une ligne de texte.

```
# let line = read_line () ;;
bonjour
val line : string = "bonjour"

# Printf.printf "%s %s\n" line line ;;
bonjour bonjour
- : unit = ()
```

Ne pas oublier l'argument `()` pour provoquer l'exécution de la fonction.

Le module `Scanf`

Fournit la fonction `Scanf.scanf` similaire au `scanf` du C **mais** le dernier argument est une fonction.

```
# let nb = Scanf.scanf "%d" (fun x -> x) ;;
17
val nb : int = 17

# let _ = Scanf.scanf "%c" (fun x -> x) ;;
- : char = '\n'

# Scanf.scanf "%d" (fun x ->
      Printf.printf "%d * %d = %d\n" x x (x*x)) ;;
17
17 * 17 = 289
- : unit = ()
```