Programmation Fonctionnelle en OCaML

Chargé de cours:	Jean-Marc Talbot,	
Chargés de TD/TM	Irène Durand, Stefka Gueorguieva,	
	Alice Rixte, Augusta Mukam, Jean-Marc Talbot	
Cours, 7 x 1h20		S36 (x2), 37-39, 41, 43
TM en présentiel, 13 séances de 2h40		S36-43, 45-49
Devoir surveillé: S46 ?		
TP noté: S50		
Travail individuel		4h par semaine

https://moodle.u-bordeaux.fr/course/view.php?id=1208

MCC: Session1: 0.5EX1+ 0.5CC, Session2: 0.5EX2 + 0.5CC
CC: DS 1h30 40%, TP noté 1h30 40%, exos Moodle 20%
Examens 1h30

Objectifs de ce cours

- Acquérir les bases de la programmation fonctionnelle
- Appliquer les principes généraux de programmation
  - Lisibilité du code
  - Réutilisabilité
  - Maintenabilité
  - Efficacité (quand elle ne nuit pas à la lisibilité) $\Rightarrow$ Complexité
  - Test

Paradigmes de programmation

Un paradigme correspond à un style de programmation se basant sur un ensemble de concepts.

- impératif
- fonctionnel
- procédural
- objet
- logique

Un langage peut offrir au programmeur plusieurs paradigmes. Un programme peut utiliser de plusieurs paradigmes.

- `Python`: ?
- `Common Lisp`: ?
- `OCaML` ?
- `C`: ?
- `C++`: ?
- `Java`: ?
- `Prolog`: ?

Paradigmes de programmation

Un paradigme correspond à un style de programmation se basant sur un ensemble de concepts.

- impératif
- fonctionnel
- procédural
- objet
- logique

Un langage peut offrir au programmeur plusieurs paradigmes. Un programme peut utiliser de plusieurs paradigmes.

- `Python`: impératif, procédural, objet, fonctionnel
- `OCaML`: fonctionnel, impératif, procédural, objet
- `C`: impératif, procédural
- `C++`: impératif, procédural, objet
- `Java`: impératif, objet, fonctionnel
- `Prolog`: logique

Paradigme impératif

- un programme a un état: la valeur de l'ensemble de ses variables
- l'exécution d'une instruction fait passer le programme d'un état à un autre
- instruction de base est l'affectation
- structure de contrôle de base est la boucle tant-que

Paradigme impératif

- un programme a un état: la valeur de l'ensemble de ses variables
- l'exécution d'une instruction fait passer le programme d'un état à un autre
- instruction de base est l'affectation
- structure de contrôle de base est la boucle tant-que

- source de nombreuses difficultés et bugs
- produit des programmes plus difficiles à comprendre
- voir bugs célèbres aux conséquences désastreuses

Paradigme fonctionnel vs paradigme impératif

Le paradigme fonctionnel s'oppose au paradigme impératif

- La fonction est un objet de base (comme les entiers, flottants, caractères, ..).
  On dit que c'est un objet de première classe; elle peut être:
  - passée en paramètre d'une autre fonction
  - créée et retournée par une fonction
- pas d'effets de bord
  - pas d'affectation
  - on se contente d'appeler des fonctions
  - la structure de contrôle de base est la récursivité.

Quand utiliser le paradigme fonctionnel?

- quand il y a des enjeux de sécurité (programmes plus sûrs et même dans certains cas prouvés)
- quand on veut développer rapidement un prototype

L'inconvénient de la programmation fonctionnelle sera principalement la performance; mais ce défaut diminue avec des compilateurs de plus en plus optimisés.

Propriétés d'un langage

Un langage de programmation peut-être

- interactif (`Python`) /non interactif (`C`)
- compilé (`C`, `Java`) / interprété (`Python`)
- typé dynamiquement (`Python`) / statiquement (`C`, `Java`)
- typé faiblement (`C`) /fortement (`Java`)
- avec un typage déclaré (`C`, `Java`) ou inféré (`TypeScript`, `Rust`)

Le typage statique fort permet d'éliminer des bugs à la compilation

L'inférence de type allège l'écriture d'un programme

Langage support: `OCaml`

- développé à l'INRIA `caml.inria.fr`
- disponible sur de nombreuses architectures (Linux, Windows, MacOS X, ...)

OCaML est un langage

- fonctionnel avec des traits impératifs
- compilé soit vers du bytecode, soit vers du code natif
- utilisable en mode interactif ou en batch
- fonctionnel, statiquement et fortement typé avec de l'inférence de type

Prise de contact avec `OCaml`

- Mode **interactif**:
  - Dans un terminal

```
$ ocaml
OCaml version 5.4.0
# 1 + 2;;
- : int = 3
#
```

  - Avec `utop` dans un terminal
  - sous Emacs, modes Tuareg et utop
  - sous VSCode nouveau pluging permettant l'interactivité
- Mode non interactif (`ocamlc`, `ocamlopt`)
  - dans un terminal
  - sous vs-code

Boucle REPL

`OCaML` est un langage interactif. Quand on le lance, on se trouve dans une boucle `REPL`[1], dans laquelle on peut taper des requêtes qui sont soit des expressions soit des définitions.

Le système boucle sur les trois opérations suivantes:

- lit (READ) une expression ou une définition (et la met sous une forme interne)
- évalue (EVAL) la forme interne
- affiche (PRINT) le résultat sous forme lisible par l'utilisateur

[1] Read Eval Print Loop

Expressions et définitions

Une expression a toujours une valeur et toute valeur a un type.

- Il existe des types de base comme `int`, `float`, `bool`, `char`, `...`.
- il existe des types composés construits à l'aide d'opérateurs qu'on peut nommer.

Le type d'une expression est le type de sa valeur.

Une définition réalise un effet de bord.

Expressions

Une expression est

- soit un objet de base (nombre, caractère, booléen, chaîne, fonction, ...),
- soit une expression prédéfinie (`if then else`, `let .. in` `match with`, ...)
- soit l'application d'une fonction à des arguments qui sont eux-mêmes des expressions[2].

langage interactif $\Rightarrow$ pas de programme principal
n'importe quelle fonction peut être appelée (au sein d'une expression)

[2] Noter la définition récursive d'une expression

Application d'une fonction

Notation par défaut: préfixe

la fonction *f* est placée avant ses arguments: *f e1 e2 . . .*.

Ne **pas** séparer les arguments par une virgule, comme en C.

L'application de fonction est plus prioritaire que les opérateurs usuels.

```
# max 20 12;;
- : int = 20
# max 30 12 * 2;;
- : int = 60
# max 30 (12 * 2);;
- : int = 30
```

Parenthésage

par défaut: `f e1 e2 ... en` équivaut à `(((f e1) e2) ... en)`.

Pour un autre parenthésage, il faut le préciser:
`sqrt (max (cos 3.1415) (sin 3.1415))`.

Opérateurs binaires

opérateurs binaires courants prédéfinis, en particulier opérateurs arithmétiques binaires $\Rightarrow$ notation infixe: opérateur entre les deux opérandes.

```
# 2 * (1 + 3);;
- : int = 8
```

Infixe $\Rightarrow$ préfixe
Version préfixe d'un opérateur infixe: le mettre en parenthèse:

```
# 5 + 2;;
- : int = 7
# 5 - 2;;
- : int = 3
# (+) 5 2;;
- : int = 7
# (-) 5 2;;
- : int = 3
```

Nombres et Expressions numériques

types de base: `int` (entiers), `float` (flottants)
Entiers: `3, 4, 1000`
Flottants `2.1, 3.14e-3, 17.`

Opérateurs arithmétiques

Syntaxe proche du langage mathématique (notation infixe)

À cause du typage, pas de surcharge des opérateurs ⇒ un jeu d'opérateurs pour chaque type:

`int`: `+, -, *, /, mod, abs, succ, pred, ...`

`float`:
`+., -., *., /., **, abs_float, truncate, sqrt, ...`.

```
# 2.1 +. 4.5;;
-: float 6.6
# abs_float  (-2.1);;
-: float 2.1
```

Conversions

Pas de conversion de type implicite

```
# 2.2 +. 4;;
Error:  ...
```

```
float_of_int, int_of_float, string_of_int, string_of_bool,
```

```
# 2.2 +. (float_of_int 4);;
- : float = 6.2
```

Expressions booléennes

- Type booléen `true, false`
- Opérateurs booléens
  - `&&, ||, not`
  - `=` (égalité de valeurs), `<, <=, >, >=`.

```
# 2 * 2 < 3 || 2 = 1 + 1;;
- : bool = true
# not (2 * 2 < 3 || 2 = 1 + 1);;
- : bool = false
```

Commentaires

délimités par les caractères `(*` et `*)`
pas de commentaire de ligne

```
(* ceci est un commentaire *)
```

Expressions conditionnelles

```
(* nombre de solutions d'une équation du 1er degré *)
(* ax + b = 0 *)
if a = 0 then
  if b = 0 then -1
  else 0
else 1
```

Les expressions conditionnelles[3] doivent avoir une valeur $\Rightarrow$ la partie `else` est obligatoire.

Les expressions des branches `then` et `else` doivent être du même type

```
 # if true then 2.0 else 4;;
Error: ...
```

[3] ...dont l'évaluation des sous-expressions terminent

Expression `match ... with ...`

La construction `match` permet de filtrer une valeur selon des motifs tout en affectant tout ou partie de cette valeur à des variables locales

```
match x with
  0 -> "vaut 0"
| 1 -> "vaut 1"
| y -> (string_of_int y)^" est different de 0 et de 1"
```

- Cette expression a pour valeur le membre droit du premier motif (dans l'ordre d'apparition) qui filtre la valeur.
- La variable `y` est locale à ce membre droit
- Warning si `OCcaml` estime qu'il est possible qu'aucun filtre ne s'applique
- Tous les membres droits ont le même type

Particulièrement utile pour les valeurs de type composé.

Combiner un filtre et une condition `when`

Il est possible de raffiner un cas de motif de filtre afin une condition notamment sur les variables capturées

```
match x with
  0 -> "vaut 0"
| 1 -> "vaut 1"
| y -> if y <0 then (string_of_int y)^" est négatif"
          else (string_of_int y)^" est positif et pas 0,1"
```

Plus simple et plus lisible,

```
match x with
  0 -> "vaut 0"
| 1 -> "vaut 1"
| y when y <0 ->  (string_of_int y)^" est négatif"
| y -> (string_of_int y)^" est positif et pas 0,1"
```

Expression `let in`

L'expression `let in` permet de mémoriser temporairement la valeur d'une ou plusieurs expressions dans des variables temporaires. Ceci permet d'éviter

- la duplication du code
- l'évaluation multiple d'une expression

```
# let x = 10 in x + 9;;
- : int = 19
```

On peut donner des valeurs à plusieurs variables en parallèle à l'aide du mot-clé `and`.

```
# let x = 1 and y = 2 in x + y;;
- : int = 3
```

Pour des affectations séquentielles, on utilise le `let in` en cascade.

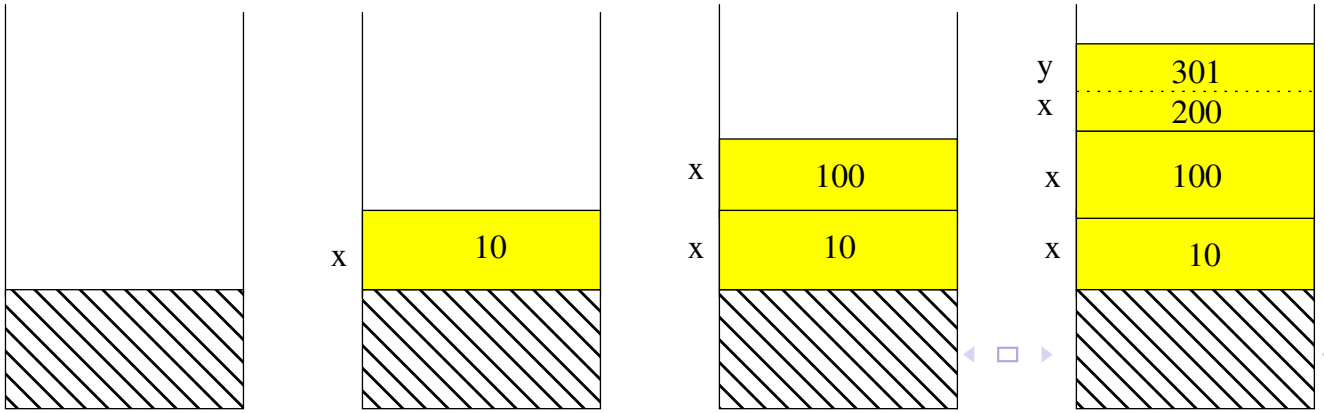
```
# let x = 2 in
  let y = x * x in y + 1;;
- : int = 5
```

Expression `let in` - portée et évaluation

```
# (let x = 10 in x + 9) + x;;
Error: Unbound value x
```

- la portée des variables lors de l'évaluation d'une expression `let x = ... in` utilise une pile
- la variable lexicale `x` créée sur la pile le temps de l'exécution du `let ... in`

```
# let x = 10 in
      let x = x * x in
      let x = 2 * x
      and y = 3 * x + 1 in
      x* y;;
- : int  = 60200
```



Définitions

Les définitions sont des requêtes qui sont, comme les expressions, tapées dans la REPL. Elles permettent de faire des opérations non purement fonctionnelles (qui modifient l'état du système).

- définition `let` d'une variable
- définition `type` d'un type
- définition `module` d'un module ou de sa signature

Définition `let`

- liaison entre une variable et une valeur (donne un nom à une valeur)
- permet de définir des variables globales (indispensable pour enregistrer données et fonctions)
- effet de bord (affecte la valeur à la variable) et affiche de la variable, de la valeur et de son type
- ne pas confondre avec l'expression `let ... in`.

```
# let x = 3 + 4;;
val x : int = 7
# x;;
- : int 7
# match x with
  0 -> "vaut 0"
| 1 -> "vaut 1"
| y -> (string_of_int y)^" est different de 0 et de 1";;
- : string = "7 est different de 0 et de 1"
```