Algorithmique et Approche Fonctionnelle de la Programmation

Cours 1 : Premiers Pas

10 septembre 2009

Louis Mandel
Université Paris-Sud 11
louis.mandel@lri.fr

D'après des transparents de Sylvain Conchon

Calendrier

- Début
  - TP : le lundi 21 septembre
  - TD : le mercredi 23 septembre

- Au total : 10 cours de 1h30, 10 TD de 2h et 8 TP de 2h

- Toutes les nouvelles fraîches sur :

  `http://www.lri.fr/~mandel/ensiegnement/aaf`

Objectif de ce cours

- comprendre les différences entre l'approche fonctionnelle et l'approche impérative de la programmation
- savoir exploiter au mieux les atouts de la programmation fonctionnelle dans vos applications futures
- acquérir de bons principes de programmation (indépendants du langage de programmation)

La programmation fonctionnelle, qu'est-ce que c'est ?

- un style de programmation qui utilise comme concepts essentiels
  - la définition de fonctions
  - l'application de fonctions
- Les fonctions sont des valeurs comme les autres. Elles peuvent être :
  - passées en argument à des fonctions
  - rendues en résultats d'autres fonctions
- Avec ce style de programmation on n'utilise plus de boucles `for` ou `while`
  - on utilise uniquement des fonctions (récursives)

Trois questions légitimes

Les étudiants se posent souvent ces trois questions :

1. pourquoi s'intéresser à ce style de programmation ?
2. peut-on résoudre les mêmes problèmes qu'avec le langage C ?
3. pourquoi ce style de programmation n'est-il pas connu du grand public ?

Réponse à la première question (1/2)

Tout bon programmeur doit avant tout se fixer comme objectif d'écrire des programmes

1. corrects
2. lisibles
3. facilement réutilisables ou modifiables

ainsi,

> les progrès de la programmation sont donc étroitement liés aux constructions, concepts et méthodes qu'apportent les langages de programmation

Réponse à la première question (2/2)

Les avantages de l'approche fonctionnelle reposent sur le fait que ses concepts de programmation sont très proches des mathématiques et de la logique, d'où

- des langages de programmation de haut niveau permettant de s'abstraire de l'architecture des machines
- une clarté et une concision des programmes (conserve la simplicité de la notation mathématique)
- une plus grande sûreté des programmes (notamment grâce à un typage fort et l'absence d'effets de bord)

au final,

rapidité de développement et programmes plus sûrs

Réponse à la deuxième question

- La théorie de la calculabilité apporte une réponse à cette question
- Au début du 20^{e} siècle, mathématiciens et logiciens étudient les problèmes qui peuvent être résolus par calcul ; ils proposent plusieurs modèles de calculs théoriques dont :
  - ▷ la machine de Turing (A. Turing, 1936), modèle abstrait du fonctionnement d'un ordinateur
  - ▷ le λ-calcul (A. Church, 1930), langage de programmation fonctionnel théorique (fondement des langages fonctionnels modernes)
- la thèse de **Church-Turing** dit que
  1. le λ-calcul et la machine de Turing sont équivalents (on peut les simuler l'un dans l'autre)
  2. tout algorithme peut être calculé par une machine de Turing

Réponse à la troisième question

Comme les langages impératifs, les langages fonctionnels sont nés au début de l'informatique et ils n'ont cessé de se développer depuis ...

langages impératifs	langages fonctionnels
machine de Turing 1936	λ-calcul 1930
assembleurs	-
FORTRAN 1954-1958	LISP 1958
BASIC 1964	ISWIM 1965
C 1973	ML 1973
C++, 1981 - 1986	CAML 1984
Java 1994	OCAML 1996

Réponse à la troisième question

Les conditions du succès grand public d'un langage de programmation sont diverses

- Ada a été imposé par décision administrative
- Fortan survit grâce à une bibliothèque importante qui ne peut être remplacée sans un très gros investissement financier et beaucoup de temps
- C s'est imposé car il a servi a développer le système Unix
- Java est devenu populaire grâce aux applets
- etc.

Réponse à la troisième question

Quelques succès industriels des langages fonctionnels

- **Ericsson** utilise le langage fonctionnel ERLANG pour programmer des commutateurs téléphoniques
- **Lexifi Technologies** et **Jane Street Capital** utilisent OCAML pour programmer des logiciels destinés aux marchés financiers
- **Microsoft Research** utilise OCAML pour réaliser des logiciels de vérification de drivers
- **BlueSpec, Inc.** développe des outils d'aide à la conception de micro-processeurs en HASKELL
- **Aérospatiale** et **Airbus** utilisent des outils de vérification de programmes (C ou Java) écrits en OCAML
- etc.

Le langage Ocaml

- Nous illustrerons l'approche fonctionnelle de la programmation à l'aide du langage OCAML, ce langage :
  - ▷ met en oeuvre les principes de la programmation fonctionnelle
  - ▷ est facile d'apprentissage et d'emploi
  - ▷ dispose d'un compilateur performant générant du code efficace
- OCAML est développé à l'INRIA (Institut National de Recherche en Informatique et Automatique)

`http://caml.inria.fr`

Premier programme

Compilateur

- fichier source `hello.ml` :

```
(* Ceci est notre premier programme *)
let _ = Printf.printf "hello world!\n";;
```

- compilation :

```
--> ocamlc -o hello hello.ml
```

- exécution :

```
--> ./hello
```

Premier programme

Boucle d'interaction

- exécution du "toplevel" :

```
--> ocaml
        Objective Caml version 3.11.1
#
```

- interaction :

```
# let _ = Printf.printf "hello world!\n";;
hello world!
- : unit = ()
```

Remarque : le caractère # est le prompt inséré automatiquement par le toplevel

Survol du langage

Déclarations : constantes

```
# let une_constante = 1;;
val une_constante : int = 1
# let _ = 1;;
- : int = 1
# let _ =
    let une_constante_locale = 1.5 in
    une_constante_locale +. 2.7;;
- : float = 4.2
```

Les phrases OCaml se terminent par un `;;` qui est optionnel.

Attention : il y a deux `let` qui sont différents.

Les erreurs de typages

```
# let _ = 1 + "coucou";;
Error: This expression has type string but an expression was expected of type
         int
```

La cohérence des types est vérifiée à la compilation

Déclarations : fonctions

```
# let carre = function x -> x * x;;
val carre : int -> int = <fun>
# let carre = fun x -> x * x;;
val carre : int -> int = <fun>
# let carre x = x * x;;
val carre : int -> int = <fun>
```

```
# let _ = carre (5);;
- : int = 25
# let _ = carre 25;;
- : int = 625
```

Types produits

```
# let tuple = (9, 9.0, '9', "9");;
val tuple : int * float * char * string = (9, 9., '9', "9")
```

```
# let (prem, sec) = (false, "aaa");;
val prem : bool = false
val sec : string = "aaa"
```

Les parenthèses sont optionnelles

```
# let prem, sec = false, "aaa";;
val prem : bool = false
val sec : string = "aaa"
```

Types produits et polymorphisme

```
# let first = fun (x, y) -> x;;
val first : 'a * 'b -> 'a = <fun>
# let _ = first  (1, 2);;
- : int = 1
# let _ = first ("aaa", "bbb");;
- : string = "aaa"
```

Types produits

```
# let somme = fun (x, y) -> x + y;;
val somme : int * int -> int = <fun>
# let diagonale = fun x -> (x, x);;
val diagonale : 'a -> 'a * 'a = <fun>
# let _ = (diagonale 2, diagonale true);;
- : (int * int) * (bool * bool) = ((2, 2), (true, true))
```

Types enregistrements

```
# type personne = { nom: string; age: int; };;
type personne = { nom : string; age : int; }
# let lambda = { nom = "LA"; age = 21; };;
val lambda : personne = {nom = "LA"; age = 21}
```

```
# let anniv x =
    { nom = x.nom; age = x.age + 1; };;
val anniv : personne -> personne = <fun>
# let _ = anniv lambda;;
- : personne = {nom = "LA"; age = 22}
# let _ = lambda;;
- : personne = {nom = "LA"; age = 21}
```

Types sommes

```
# type couleur = Trefle | Pique | Coeur | Carreau;;
type couleur = Trefle | Pique | Coeur | Carreau
```

```
# type carte =
    | Dame of couleur
    | Petite_carte of int * couleur;;
type carte = Dame of couleur | Petite_carte of int * couleur
```

```
# let dame_pique = Dame Pique;;
val dame_pique : carte = Dame Pique
```

Types sommes et filtrage de motifs

```
# let points c =
    match c with
    | Dame _ -> 20
    | Petite_carte (10, _) -> 10
    | Petite_carte (v, coul) ->
        if v = 9 && coul = Coeur then 11 else 0;;
val points : carte -> int = <fun>
```

```
# let _ = points dame_pique;;
- : int = 20
# let _ = points (Petite_carte (7, Trefle));;
- : int = 0
```

Le langage OCaml plus en détails

Types et expressions élémentaires

- Le type `int`
  - représente les entiers compris entre -2^{30} et $2^{30} - 1$
  - les opérations sur ce type sont notamment `+`, `-`, `*`, `/`, `mod` (modulo), ...
  - exemple : `4 / 2 * 2 mod 3 - 1`

- Le type `float`
  - représente les nombres réels (ou flottants) à l'aide d'une mantisse `m` et d'un exposant `n` (pour coder le nombre $\mathtt{m} \times 10^{\mathtt{n}}$)
  - les types `int` et `float` sont disjoints
  - les opérations élémentaires sur ce type sont suivies d'un point `+.`, `-.` `*.`, `/.`, `sqrt`, ...
  - on passe d'un flottant à un entier à l'aide de la fonction `float_of_int`
  - exemple : `1.0 +. (float_of_int 2) *. 2.3e4`

Types et expressions élémentaires

- Le type `bool`
  - représente les valeurs de vérité (ou valeurs booléennes) `true` (vrai) et `false` (faux)
  - ces valeurs peuvent être combinées avec les opérations élémentaires `not` (négation), `&&` (conjonction) et `||` (disjonction)
  - les opérateurs de comparaison (=, <, >, <=, >=) retournent des valeurs booléennes
  - l'expression `if - then - else -` utilise ces valeurs
  - exemples :

```
1<=2 && (2=3 || 4>5) && not(2<2)

if 2<0 then 2.0 else (4.6 *. 1.4)
```

Types et expressions élémentaires

- Le type `string`
  - représente les chaînes de caractères
  - ces valeurs sont encadrées par deux guillemets `"`
  - l'opération de concaténation est `^`
  - exemple : `"bonjour à" ^ " tous\n`
- Le type `char`
  - représente les caractères
  - ces valeurs sont encadrées de deux apostrophes `'`
  - exemples : `'a'`, `'\n'`
- Le type `unit`
  - représente les expressions sans réelle valeur (ex. fonctions d'affichage)
  - une seule valeur a ce type, elle est notée `()`
  - c'est l'équivalent du type `void` en C

Les déclarations globales

$$declaration \quad ::= \quad \texttt{let}\ pattern\ \texttt{=}\ expression\ [\texttt{;;}]$$

Un programme est une suite de déclaration de valeurs

- exemple :

```
let x = 1 + 2;;
let _ = Printf.printf "%i" x;;
let y = x * x;;
let _ = Printf.printf "%i" y;;
let a, b = ("bonjour", 6);;
```

- les « ;; » sont optionnels.

Notion de variable

`let` x = *expression* `;;` introduit une variable globale x

- Les variables jouent le même rôle qu'en mathématiques : elles sont une représentation symbolique des valeurs
- Différences avec la notion usuelle de variable :
  - nécessairement initialisée
  - type pas déclaré mais inféré
  - contenu non modifiable

Expressions/Instructions

Pas de distinctions entre expressions et instructions

- il n'y a que des expressions
- exemples
  - en Caml

```
# let a = if x = 1 then 1 else 2;;
```

  - en C

```
int a = (x == 1) ? 1 : 2;
```

  - en Caml

```
# let b = Printf.printf "%d" x; 2;;
```

  - en C

```
int b = (printf("%d", x), 2);
```

Expressions

Toutes les expressions ont un type et s'évaluent vers une valeur de ce type

- l'expression : `1.5 +. 2.0`
  - a le type `float`
  - s'évalue vers la valeur `3.5`
- l'expression : `Printf.printf "youhou"`
  - a le type `unit`
  - s'évalue vers la valeur `()`
- l'expression : `if 6 mod 2 = 0 then 'a' else 'b'`
  - a le type `char`
  - s'évalue vers la valeur `'a'`
- l'expression : `if 6 mod 2 = 0 then 'a' else ()`
  - n'est pas bien typée

Expressions

$$
\begin{array}{lll}
\textit{expression} & ::= & \textit{constante} \\
& | & \textit{expression}\ \texttt{+}\ \textit{expression} \\
& | & \textit{expression}\ \texttt{-}\ \textit{expression} \\
& | & \ldots \\
& | & \texttt{if}\ \textit{expression}\ \texttt{then}\ \textit{expression}\ \texttt{else}\ \textit{expression} \\
& | & \textit{expression}\ \texttt{;}\ \textit{expression} \\
& | & \ldots
\end{array}
$$

Variables locales

$$\begin{array}{rcl} \textit{expression} & ::= & \ldots \\ & \mid & \texttt{let}\ \textit{pattern}\ \texttt{=}\ \textit{expression}\ \texttt{in}\ \textit{expression} \end{array}$$

```
# let global =
    let local = 1 in local + 1;;
val global : int = 2
# let _ = global + local;;
Error: Unbound value local
```

- Comme une variable globale :
  - nécessairement initialisée
  - type pas déclaré mais inféré
  - contenu non modifiable
  - mais portée limitée à l'expression qui suit le `in`

Portée des variables

- En C

```
{  int x = 1;
   printf("x = %d; ", x);
   { int x = 2; printf("x = %d; ", x); }
   printf("x = %d; ", x);
}
```

- En Caml

```
# let _ =
    let x = 1 in
    Printf.printf "x = %d; " x;
    (let x = 2 in Printf.printf "x = %d; " x);
    Printf.printf "x = %d; " x;;
x = 1; x = 2; x = 1; - : unit = ()
```

let/in

Autres exemples

```
# let x = 3;;
val x : int = 3
# let _ = let x = 1 in x + 1;;
- : int = 2
# let _ = x + 1;;
- : int = 4
# let _ =
    let aux = 6 * 24 + 3868 in
    (aux / 2, aux + 1);;
- : int * int = (2006, 4013)
```

L'expression `let` x = e_1 `in` e_2

- a le type et la valeur de e_2
- dans l'environnement où x à le type et la valeur de e_1

Le langage OCaml plus en détails :

---

les fonctions

Syntaxe

$$\begin{array}{rcl} \textit{expression} & ::= & \ldots \\ & \mid & \texttt{fun}\ \textit{pattern}\ \texttt{->}\ \textit{expression} \end{array}$$

Les fonctions sont des expressions

```
# let abs =
    fun x ->
      if x < 0 then -x else x;;
val abs : int -> int = <fun>
```

Il y a du « sucre syntaxique » pour définir les fonctions

```
# let abs x =
    if x < 0 then -x else x;;
val abs : int -> int = <fun>
```

Expression fonctionnelles

Les fonctions sont des valeurs comme les autres

```
# let _ = fun x -> x + x;;
- : int -> int = <fun>
```

Les fonctions anonymes s'appliquent comme les fonctions nommées

```
# let _ =  (fun x -> x + x) 4;;
- : int = 8
```

Comparaison avec C

En C

```
int successeur (int x) {
  return (x + 1);
 }
```

En Caml

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

- corps = expression à rendre (pas de `return` explicite)
- type inférés : l'utilisateur n'a pas à donner le type des arguments ou du résultat
- les parenthèses ne sont pas nécessaires lors de l'appel

```
# let _ = successeur 1;;
- : int = 2
```

Fonctions « à plusieurs arguments »

Les paramètres ne sont pas entre parenthèses

```
# let f x y z =
    if x then y + 1 else z - 1;;
val f : bool -> int -> int -> int = <fun>
```

Les arguments ne sont pas entre parenthèses

```
# let _ = f true 2 3;;
- : int = 3
```

Fonction qui prend un seul paramètre de type produit

```
# let f (x, y, z) =
    if x then y + 1 else z - 1;;
val f : bool * int * int -> int = <fun>
```

Fonctions « à plusieurs arguments »

Une fonction à plusieurs arguments est en fait
plusieurs fonctions à un argument

```
# let f =
    fun x ->
     (fun y ->
       (fun z -> if x then y + 1 else z - 1));;
val f : bool -> int -> int -> int = <fun>
```

Il y a aussi du sucre syntaxique pour écrire

```
# let f =
    fun x y z ->
      if x then y + 1 else z - 1;;
val f : bool -> int -> int -> int = <fun>
```

Fonctions locales

Fonction locale à une expression

```
# let _ =
    let carre n =
      n * n
    in
    carre 3 + carre 4 = carre 5;;
- : bool = true
```

Fonction locale à une fonction

```
# let pythagore x y z =
    let carre n =
      n * n
    in
    carre x + carre y = carre z;;
val pythagore : int -> int -> int -> bool = <fun>
```

Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle même

- il faut au moins un cas sans appel récursif si on veut que la fonction termine

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

En math : $\begin{cases} u_0 & = & 1 \\ u_n & = & n * u_{n-1} \end{cases}$

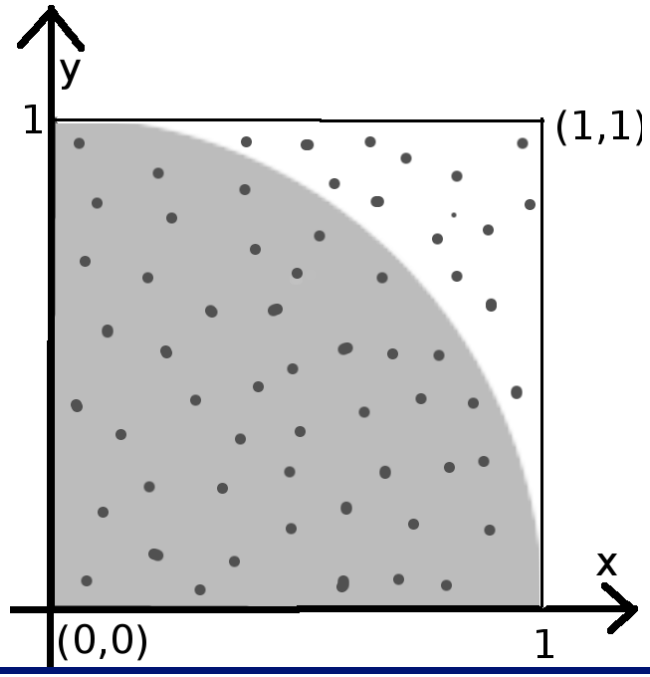
Fonctions récursives

```
# let boucle n =
    let rec blc_rec i =
      if i >= n then
        ()
      else
        (Printf.printf "%d " i;
         blc_rec (i+1))
    in
    blc_rec 0;;
val boucle : int -> unit = <fun>
# let _ = boucle 9;;
0 1 2 3 4 5 6 7 8 - : unit = ()
```

Exemple

Calcul de π par la méthode de Monte-Carlo

- Soit un carré de côté 1 et le quart de cercle de rayon 1 inscrit dans ce carré (l'aire de ce cercle est $\pi/4$)
- Si l'on choisit au hasard un point du carré, la probabilité qu'il soit dans le quart de cercle est donc également de $\pi/4$
- En tirant au hasard un grand nombre $\mathtt{n}$ de points dans le carré, si $\mathtt{p}$ est le nombre de points à l'intérieur du cercle, alors $4 \times \mathtt{p}/\mathtt{n}$ donne une bonne approximation de π



Calcul de π par la méthode de Monte-Carlo

```
# let n = 100_000_000;;
val n : int = 100000000
# let rec approx_pi p cpt =
    if cpt=0 then 4. *. (float_of_int p) /. (float_of_int n)
    else
      let x = Random.float 1. in
      let y = Random.float 1. in
      let c = if x*.x +. y*.y < 1. then 1 else 0 in
      approx_pi (p+c) (cpt-1);;
val approx_pi : int -> int -> float = <fun>
# let _ = Printf.printf "%f\n" (approx_pi 0 n);;
3.141629
- : unit = ()
```