

Algorithmique et Approche Fonctionnelle de la Programmation

Cours 1 : Premiers Pas

15 Septembre 2008

Sylvain Conchon

sylvain.conchon@lri.fr

calendrier

les TD commencent la semaine du 22 Septembre

les TP débutent la semaine du 29 Septembre

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

contrôle des connaissances :

- contrôle continu : 2 ou 3 TP notés, coefficient 1
- partiels : 29,30 ou 31 Octobre (à confirmer), coefficient 2
- examen : semaine du 15 Décembre (à confirmer), coefficient 3

retrouver toutes ces informations (et d'autres) sur le web

<http://www.lri.fr/~conchon/AAF/>

objectifs 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** et l'**application** de **fonctions**

avec ce style de programmation on n'utilise plus de boucles `for` ou `while`, mais uniquement des fonctions (récursives) qui, contrairement aux langages impératifs (comme le C), sont des valeurs comme les autres, i.e. qui doivent pouvoir :

- être passées en **argument** à des fonctions
- rendues en **résultats** d'autres fonctions

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 ?

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

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

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

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	-
FORTAN 1954-1958	LISP 1958
BASIC 1964	ISWIM 1965
C 1973	ML 1973
C++, 1981 - 1986	CAML 1984
Java 1994	OCAML 1996

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

- ADA a été imposé par décision administrative
- FORTRAN 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 à développer le système Unix
- JAVA est devenu populaire grâce aux applets
- etc.

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.

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>

Premiers pas en OCAML

types et expressions élémentaires (2/3)

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
- la construction `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 (1/3)

le type `int`

- représente les entiers compris entre -2^{30} et $2^{30} - 1$
- les opérations sur ce type sont notamment `+` `-` `*` `/` et `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 $m \times 10^n$)
- les types `int` et `float` sont disjoints
- les opérations élémentaires sur ce type sont suivies d'un point `+. - . *. /.`
- 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 (3/3)

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 variables jouent le même rôle qu'en mathématiques : elles sont une représentation symbolique des valeurs

`let x = e` introduit une variable globale

exemple :

```
let x = 2 * 3 + 4
```

la déclaration équivalente en C :

```
int x = 2 * 3 + 4;
```

différences avec la notion usuelle de variable :

- ① nécessairement **initialisée**
- ② type pas déclaré mais **inféré**
- ③ contenu **non modifiable**

let in = expression

`let x = e1 in e2` est une expression

son type et sa valeur sont ceux de `e2`,

dans un environnement où `x` a le type et la valeur de `e1`

Exemple

```
let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```

en C, la portée d'une variable locale est définie par le **bloc** :

```
{
  int x = 1;
  ...
}
```

en OCAML, variable locale introduite par `let in` :

```
let x = 10 in x * x
```

comme une variable globale :

- nécessairement initialisée
- type inféré
- immuable
- mais **portée limitée à l'expression qui suit le in**

Écrire des programmes en OCAML

version **interactive** du compilateur

```
> ocaml
  Objective Caml version 3.09.1

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

# let y = 1 + 2;;
val y : int = 3

# y * y;;
- : int = 9
```

```
hello.ml
print_string "hello world!\n";;
```

compilation

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

exécution

```
> ./hello
hello world!
>
```

- un programme est une suite de déclarations et d'expressions à évaluer
- la fin d'une déclaration ou d'une expression est spécifiée par deux points virgules `;;`

exemple :

```
let x = 1 + 2;;
print_int x;;
let y = x * x;;
print_int y;;
```

Fonctions

la fonction carre qui à un entier x associe x^2 s'écrit

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

- corps = expression (pas de `return`)
- type inféré (types de l'argument x et du résultat)

la fonction carre s'utilise de la manière suivante

```
# carre 4;;
- : int = 16
```

```
# let f x y z = if x > 0 then y + x else z - x;;
val f : int -> int -> int -> int = <fun>
```

c'est en fait une fonction à un argument qui retourne une fonction

```
val f : int -> (int -> (int -> int)) = <fun>
```

l'utilisation de `f` est très simple

```
# f 1 2 3;;
- : int = 3
```

cette expression est équivalente à l'expression suivante

```
# ((f 1) 2) 3;;
- : int = 3
```

les fonctions sont des valeurs comme les autres

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

elles s'appliquent comme les fonctions nommées

```
# (fun x -> x * x) 4;;
- : int = 16
```

ainsi, la définition de la fonction carre est donc équivalente à la déclaration suivante

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

la construction `let rec f = e` permet de définir des fonctions récursives

```
fact.ml
let rec fact n =
  if n=0 then 1 else n * fact (n-1);;

Printf.printf "%d\n" (fact 5);;
```

compilation

```
ocamlc -o fact fact.ml
```

exécution

```
./fact
120
```

fonction locale à une expression et fonction locale à une autre fonction

boucle.ml

```
let boucle n =  
  let rec blc_rec i =  
    Printf.printf "%d " i;  
    if i < n then blc_rec (i+1)  
  in  
  blc_rec 0;;  
  
let carre x = x * x in boucle (carre 3);;
```

compilation

```
ocamlc -o boucle boucle.ml
```

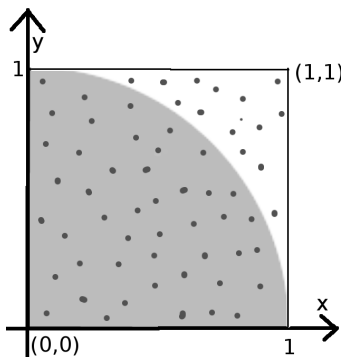
exécution

```
./boucle  
0 1 2 3 4 5 6 7 8 9
```

Quelques exemples de programmes en OCAML

exemple 1 : 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 n de points dans le carré, si p est le nombre de points à l'intérieur du cercle, alors $4 \times p/n$ donne une bonne approximation de π



exemple 1 : la solution en OCAML

```
montecarlo.ml  
let n = 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);;  
  
Printf.printf "%f\n" (approx_pi 0 n);;
```

compilation

```
ocamlc -o montecarlo montecarlo.ml
```

exécution

```
./montecarlo  
3.14161356
```

exemple 2 : la conjecture de syracuse

- soit n un entier plus grand que 1 ; si n est pair on le divise par 2 : s'il est impair on calcule $3 \times n + 1$
- la conjecture affirme qu'en réitérant cette opération sur l'entier ainsi obtenu ce calcul se termine toujours sur 1
- mieux, que cette suite finit toujours par 4,2,1

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

sous son apparente simplicité, cette conjecture défie encore les mathématiciens d'aujourd'hui

exemple 2 : la conjecture en OCAML

syracuse.ml

```
let rec syracuse n =
  Printf.printf "%d " n;
  if n = 1 then ()
  else if n mod 2 = 0 then syracuse (n/2)
  else syracuse (3*n +1);;

syracuse 6;;
```

compilation

```
ocamlc -o syracuse syracuse.ml
```

exécution

```
./syracuse
6 3 10 5 16 8 4 2 1
```

exemple 3 : nombres premiers

- un nombre entier n est dit **premier** s'il n'est divisible que par 1 et n
- un test de primalité **naïf** consiste à tester la divisibilité de n par 2, puis de tous les entiers impairs de deux en deux jusqu'à la racine carrée de n

un prix de 100.000\$ est offert à la première personne qui trouvera un nombre premier avec au moins 10,000,000 de chiffres

aujourd'hui, le plus grand nombre premier connu est $2^{32582657} - 1$, il comporte 9 808 358 chiffres

exemple 3 : générer des nombres premiers en OCAML

premiers.ml

```
let premier n =
  let rec prem_rec x =
    x*x>n || (n mod x <>0 && prem_rec (x+2))
  in
  n=2 || n mod 2<>0 && prem_rec 3;;
let rec generateur p =
  if premier p then Printf.printf "%d " p;
  generateur (p+1);;

generateur 2;;
```

compilation

```
ocamlc -o premiers premiers.ml
```

exécution

```
./premiers
2 3 5 7 11 13 17 19 23 29 31 37 ...
```