Programmation fonctionnelle - PG104 -

COURS 8

Myriam Desainte-Catherine et David Renault

February 10, 2020

Questions : les fonctions anonymes

Les langages suivants permettent-ils de programmer avec des fonctions anonymes? À votre avis, parmi cette liste de langages, combien permettent de programmer avec des fonctions anonymes?

1 3 11 16 23

- ActionScript
- C
- C++
- C#
- Common Lisp
- Delphi
- Objective C
- Go
- Groovy
- Haskell
- Java
- Javascript
- Lua
- ML
- Perl
- PHP
- Prolog
- Python
- Ruby
- Rust
- Scala
- Smalltalk
- Swift

Questions : les fonctions anonymes

À votre avis, parmi cette liste de langages, combien permettent de programmer avec des fonctions anonymes?

1 | 3 | 11 | 16 | 24

- C (gcc)
- C++
- C#
- Java
- Python
- Ruby
- Rust
- MATLAB
- Octave
- Maxima
- R
- ML
- Smalltalk
- Prolog
- Lua
- PHP
- Javascript
- Perl
- Scala
- Swift
- Tcl
- .NET
- Julia
- Elixir

Manquent notamment : FORTRAN, Pascal et COBOL

Fonctionnelles en Python

Fonctions anonymes

$\lambda x.x$ s'écrit lambda x: x

Exemple

```
delirius: python
Python 2.7.10 (default, May 27 2015, 18:11:38)
[GCC 5.1.1 20150422 (Red Hat 5.1.1-1)] on linux2
Type "help", "copyright", "credits" or "license" for more informati

>>> sentence = 'All work and no play makes Jack a dull boy'
>>> words = sentence.split()
>>> print words
['All', 'work', 'and', 'no', 'play', 'makes', 'Jack', 'a', 'dull',
>>> lengths = map( lambda word: len(word) , words)
>>> print lengths
[3, 4, 3, 2, 4, 5, 4, 1, 4, 3]
>>> f = lambda word: len(word)
>>> lengths = map(f, words)
```

Fonctionnelles en Javascript

Fonctions anonymes

```
function(message) {
  alert(message);
}
```

En scheme

```
(lambda (message)
   (alert message)))
```

Fonctionnelles

```
function ajouteur(nb) {
  return function (val) {
           return val + nb;
         }
 }
var ajoute10 = ajouteur(10);
ajoute10(1); //retourne 11
```

En scheme

```
(define (ajouteur nb)
  (lambda (val) (+ val nb)))

(define ajoute10 (ajouteur 10)

> (ajoute10 1)
11
```

Fonctionnelles en Common Lisp

Common Lisp

Il y a deux espaces de noms, un pour les variables et un pour les fonctions. Un symbole est représenté par une structure à plusieurs champs, l'un d'eux représente la valeur de variable et un autre la valeur de fonction. Lors de l'évaluation le système doit sélectionner une des deux valeurs. Son choix dépend du contexte : si le symbole est en position fonctionnelle il accède à sa valeur de fonction, en toute autre situation, il accède à sa valeur de variable.

- Quand on souhaite faire passer une fonction en paramètre, il faut indiquer au système d'utiliser la valeur de fonction et non la valeur de variable (#')
- Quand on veut appliquer une fonction renvoyée en résultat de l'application d'une fonction, il faut indiquer au système d'utiliser la valeur de variable et non la valeur de fonction (funcall)

```
> (defun ajouteur(nb) (lambda (val) (+ val nb)))
> (defparameter ajoute10 (ajouteur 10))
> (funcall ajoute10 1)
1
```

Fonctionnelles en C#

Fonctions anonymes

- Notation C# : `(x, y) => x+y`
- En scheme : `(lambda(x y) (+ x y))`

Fonctionnelles

```
double ajouteur (double nb) {
    return (val) ⇒ val+nb;
}
Myfunction ajoute10 = ajouteur (10);
double y = ajoute10(1);
```

Avec

```
double Myfunction (double x); signature fonctionnelle
```

Remarques :

- C# est un langage typé statiquement, donc il faut déclarer les types des fonctions dynamiques.
- On peut aussi composer des fonctions en C#

Fonctionnelles en C

Exemple 1

```
#include "stdio.h"

void* ajouteur (int nombre)
{
    int ajoute (int valeur) { return valeur + nombre; }
    return ajoute;
}

int main(void) {

    int (*ajoute10)(int) = ajouteur(10);
    printf("%d\n", ajoute10(1));

    return 0;
}
```

Résultat : 11 ou 1 ou erreur Résultat : 11

Fonctionnelles en C

Exemple 2

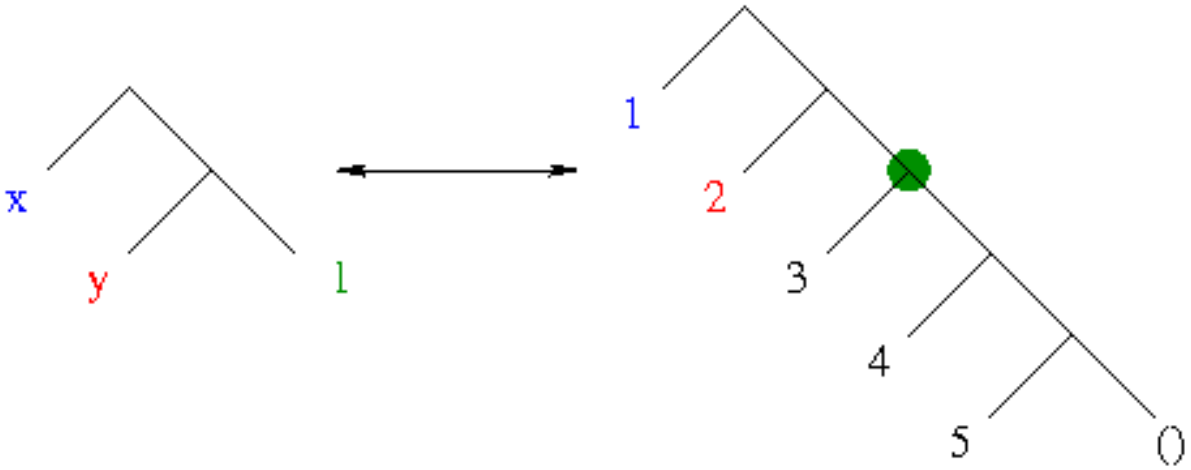
```
#include "stdio.h"

void* ajouteur (int nombre)
{
    int ajoute (int valeur) { return valeur + nombre; }
    return ajoute;
}

int main(void) {

    int (*ajoute10)(int) = ajouteur(10);
    ajouteur(20);
    printf("%d\n", ajoute10(1));

    return 0;
  }
```

Résultat : 11 ou 1 ou 21 ou erreur Résultat : 21

Fonctions à nombre d'arguments variable : λ-expressions

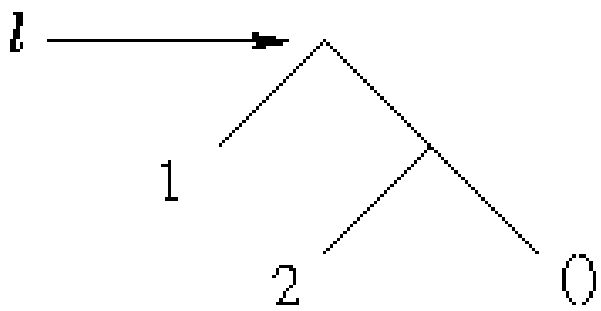
Notation pointée

```
> ((lambda (x y . l) (list x y l)) 1 2 3 4 5)
'(1 2 (3 4 5))
```



Paramètres en liste

```
> ((lambda l l) 1 2)
'(1 2)
(define g (lambda l l))
> (g 1 2 3 4)
'(1 2 3 4)
```



Fonctions nommées avec nombre d'arguments variables

Notation pointée

```
> (define (f x y . l) (list x y l))
> (f 1 2 3 4 5)
'(1 2 (3 4 5))
```

Paramètres en liste

```
> (define (h . l) l)
> (h 1 2 3 4)
'(1 2 3 4)
```

Fonctions à plusieurs résultats

On utilise la forme `values` pour créer plusieurs résultats.

```
> (define (2r x y) (values (+ x y) (- x y)))
> (2r 1 2)
3
-1
```

Fonctions à nombre d'arguments fixe ou variable

Soient les fonctions suivantes

```
(define (longueur1 l)
  (length l))
```

```
(define (longueur2 . l)
  (length l))
```

La fonction longueur1 admet-elle :
une liste en argument ou bien un nombre d'arguments variable ?

La fonction longueur1 admet une liste en argument

La fonction longueur2 admet-elle :
une liste en argument ou bien un nombre d'arguments variable ?

La fonction longueur2 admet un nombre d'arguments variable

Comment appliquer la fonction longueur1 :
(longueur1 '(1 2 3)) ou (longueur1 1 2 3) ?

Comment appliquer la fonction longueur1 : (longueur '(1 2 3))

Comment appliquer la fonction longueur2 :

Fonctionnelles de la bibliothèque : arguments fonctions

- **Appartenance** : (memf ⟨*proc*⟩ ⟨*lst*⟩)
- **Filtrage** : (filter ⟨*pred*⟩ ⟨*lst*⟩)
- **Liste d'associations** : (**assoc** ⟨*v*⟩ ⟨*lst*⟩)
  (**assoc** ⟨*v*⟩ ⟨*lst*⟩ ⟨*pred*⟩)
  (assf ⟨*proc*⟩ ⟨*lst*⟩)
- **Constructeur** : (build−list ⟨*n*⟩ ⟨*proc*⟩)
- **Itération sur des listes** : **map**, **apply**, andmap, ormap, foldl , foldr

Exemple

```
>(memf (lambda (arg) (> arg 9)) '(7 1 9 10 3))
'(10 3)

> (filter positive? '(1 -2 3 4 -5))
'(1 3 4)

> (assf (lambda (arg) (> arg 2)) (list (list 1 2) (list 3 4) (list 5 6)))
'(3 4)

> (build-list 10 list)
'((0) (1) (2) (3) (4) (5) (6) (7) (8) (9))
```

Couteau suisse fonctionnel : les formes **map**, fold, **apply**, compose, curry

- **map** : itération sur des listes Parallélisable : n $\longrightarrow$ n
- fold : itération fonctionnelle récursive Combinaison : n $\longrightarrow$ 1
  Forme **reduce** en lisp – algorithme MapReduce de Google
- **apply** : application d'une fonction à une liste d'arguments – Syntaxique
- compose : composition de fonctions
- curry : curryfication de fonctions (passage de n arguments à n-1)

Comment utiliser ces outils, comment sont-ils fabriqués?

Itération : la forme **map**

Définiton

La forme **map** prend une fonction `f` et n listes en arguments ($n > 0$), où n est l'arité de la fonction `f`. Soit $l_1 = (\langle a_1 \rangle\ \langle a_2 \rangle ... \langle a_n \rangle)$, et $l_2 = (\langle b_1 \rangle\ \langle b_2 \rangle ... \langle b_n \rangle)$.

- Cas d'une fonction unaire :
  `(map` $\langle f \rangle$ $\langle l_1 \rangle$`)` $\longrightarrow$ `((`$\langle f \rangle$ $\langle a_1 \rangle$`)(`$\langle f \rangle$ $\langle a_2 \rangle$`)...(`$\langle f \rangle$ $\langle a_n \rangle$`))`
- Cas d'une fonction n-aire :
  `(map` $\langle f \rangle$ $\langle l_1 \rangle$ $\langle l_2 \rangle$ `...)` $\longrightarrow$ `((`$\langle f \rangle$ $\langle a_1 \rangle$ $\langle b_1 \rangle$`..)(`$\langle f \rangle$ $\langle a_2 \rangle$`..)..(`$\langle f \rangle$ $\langle a_n \rangle$`..))`

Exemple

```
> (map sub1 '(1 2 3 4 5))
'(0 1 2 3 4)
> (map cons '(1 2 3 4) '(a b c d))
'((1 . a) (2 . b) (3 . c) (4 . d))
> (map (lambda(x) (list (add1 x))) '(1 2 3))
'((2) (3) (4))
```

Remarques : Les listes doivent avoir le même nombre d'arguments, le premier argument doit impérativement être une fonction et non une macro.

Maîtriser l'utilisation de map

Quels sont les résultats des évaluations des expressions suivantes ?

- (map + '(1 2 3 4) '(1 4 2 1 2) '(3 1 2 4)) '(5 7 7 9) ou erreur erreur
- (map car '((a1 a2 a3) (b1 b2 b3) (c1 c2 c3))) '(a1 b1 c1) ou '(a1 a2 a3) '(a1 b1 c1)
- (map cdr '((a1 a2 a3) (b1 b2 b3) (c1 c2 c3))) '((b1 b2 b3) (c1 c2 c3))) ou '((a2 a3) (b2 b3) (c2 c3)) '((a2 a3) (b2 b3) (c2 c3))

Remarque : les fonctions `car` et `cdr` sont unaires donc il n'y a qu'une seule liste en argument.

Itérations logiques : formes andmap et ormap

Forme andmap

Cette forme a la même signature que la forme **map**. Elle applique la fonction aux éléments de la liste dans l'ordre. Le résultat est celui de la dernière application, pas de mise en liste. S'arrête au premier résultat faux.

```
> (andmap positive? '(1 2 3))
#t
> (andmap + '(1 2 3) '(4 5 6))
9
```

Forme ormap

Comme la forme andmap mais renvoie le premier vrai.

```
> (ormap eq? '(a b c) '(a b c))
#t
> (ormap positive? '(1 2 a))
#t
> (ormap + '(1 2 3) '(4 5 6))
5
```