Programmation Fonctionnelle

Introduction

Michael Lampis

2023-2024

Avant de commencer

Informations administratives

- Page web :
  `https://www.lamsade.dauphine.fr/~mlampis/Functional/`

- Cours: mardi 13h45-15h15 (mais consultez le planning !)
- Trois groupes de TD/~~TP~~*
  - Gr 4: TD Mar 10h15-11h45 (Lampis)
  - Gr 5: TD Ven 13h45-15h15 (Ardevol-Martinez)
  - Gr 6: TD Mer 15h30-17h00 (Ardevol-Matrinez)
- Consultez régulièrement la page web. Vous pourrez y trouver les slides, les feuilles de TD et TP, ainsi que d'autres informations utiles.
- **N'hésitez pas à nous envoyer des mails !!**
- *Les TPs initialement prévus ont été **annulés**. Nous allons essayer de faire quelques séances de TD en salle machine pour pallier ce manque...

Avant de commencer

Notation :

- ~~Contrôle Continu : exercices de programmations faits pendant les TP.~~
- Partiel : semaine du 18/3
- Examen Final

Formule : ~~$\max\{0.2CC + 0.2P + 0.6E, E\}$~~ $\max\{E, 0, 7E + 0.3P\}$
Conseils :

- Faire les exercices de TP ~~en respectant la date limite~~ !
  - ~~On va mettre les solutions sur la page web, donc difficile d'accepter des soumissions tardives.~~
- Venir au cours et aux TDs (même s'il n'y a pas de note pour l'assiduité)
- Étudier les slides, feuilles de TD/TP. Si vous voulez aller plus loin, plusieurs textes de référence sont proposés sur la page web.

Introduction Générale

- Objectif de ce cours : introduction aux principes de la **programmation fonctionnelle**
- Programmation Fonctionnelle : paradigme de programmation avec caractéristiques principales

  - style déclaratif plutôt que impératif
  - les fonctions sont considérées comme un type de données élémentaire
  - utilisation fréquente de la récursivité et des **fonctions d'ordre supérieur**

  Pré requis pour ce cours :

  - Savoir programmer (p.ex. en C/Python)
  - Connaissances algorithmiques de base
  - Maturité mathématique

Programmation Fonctionnelle – un exemple

Considérez les fonctions suivantes (en C)

```
int fact_iter(int n)
{
        int res=1,i;
        for(i=1; i<=n; i++) res = res*i;
        return res;
}
```

et

```
int fact_rec(int n)
{
        if(n<=1) return 1;
        else return n*fact_rec(n-1);
}
```

Idées principales

- La fonction `fact_rec` est l'implémentation qui est plus proche de la philosophie de la programmation fonctionnelle.
  - Style déclaratif vs impératif
  - Récursivité vs boucles
  - **Fonction Pure**
    - Retourne toujours la même valeur pour le même argument
    - Pas d'effets de bord (side-effects)
- Pourquoi opter pour ce style de programmation ?
  - Donne des programmes plus concis et clairs
  - Facilite la preuve formelle de la correction de nos programmes

Programmation Fonctionnelle – un autre exemple

(Toujours en C)

```
int fact(int n){ ... }
int mydouble(int n){ return 2*n; }
int square(int n){ return n*n; }
```

Est-ce qu'on peut avoir une fonction du genre :

```
int apply_a_func(?FUNCTION? f, int n){ return f(n); }
...
apply_a_func(fact, 5) // => 120
apply_a_func(mydouble, 5) // => 10
apply_a_func(square, 5) // => 25
```

Fonctions d'ordre supérieur

- Fonction d'ordre supérieur : fonction qui prend comme argument d'autres fonctions
- Intérêt : nous permet de concevoir des programmes **génériques**
  - Exemple : on veut appliquer une transformation sur chaque élément d'un tableau
  - Version1:

```
doubleArray(int A[], int size);
factArray(int A[], int size);
squareArray(int A[], int size);...
```

  - Version2:

```
applyArray(??FUNCTION?? f, int A[], int size);
```

  - Permet de programmer une seule fois l'algorithme générique, et le réutiliser pour chaque opération.
- Est-il possible de faire ça en C ??

Fonctions d'ordre supérieur (C)

Oui, mais c'est pas terrible !

```
#include <stdio.h>

int apply( int (*f)(int), int n){ return f(n); }
int square(int n){ return n*n; }
int mydouble(int n){ return 2*n; }
int main()
{
        printf("%d\n", apply(square,5));
        printf("%d\n", apply(mydouble,5));
}
```

Donne : 25 et 10

- Simuler les fonctions d'ordre supérieur est possible en C en utilisant les pointeurs.
- Évidemment, le langage n'était pas conçu pour ça !

Fonctions d'ordre supérieur (Haskell)

Dans ce cours on va utiliser le langage **Haskell**
Voici un premier exemple qui utilise une fonction d'ordre supérieur

```
double x = 2*x
square x = x*x
apply f x = f x

apply double 3 -- => donne 6
apply square 5 -- => donne 25
```

Aspects syntaxiques (à revoir) :

- Définition de fonctions avec l'opérateur $=$
  - Comme s'il s'agissait de simples variables qu'on initialise avec une valeur !?
- Application de fonction sans parenthèses
  - On écrit `f x` et pas `f(x)`

Le langage Haskell

Caractéristiques du langage Haskell :

- Langage purement fonctionnel
  - À nos jours, tous les langages modernes utilisent des notions de programmation fonctionnelle (Java, C++, Python, . . .)
  - Haskell a été conçu comme un langage qui met l'accent sur ce paradigme et **empêche** l'utilisation de la programmation non-fonctionnelle !
  - $\Rightarrow$ Un langage pur et "fondamentaliste", idéale pour apprendre les idées du paradigme.
  - Détail important : maîtriser la programmation fonctionnelle est très important si on s'intéresse aux applications des "Big Data" !

Le langage Haskell

Caractéristiques du langage Haskell :

- Langage purement fonctionnel $\Rightarrow$
- Accent sur la récursivité
- Fonctions d'ordre supérieur
- Opérations de compréhension de listes
- Système de types sophistiqué
- Évaluation Paresseuse (Lazy evaluation)
- Fonctions Génériques

Bref, Haskell est conçu en sorte que nos programmes deviennent aussi concis et claires que possible.
C'est un langage qui insiste sur **l'élégance mathématique.**

(Cependant, on va aussi considérer des questions d'efficacité d'implémentation.)

Aspects pratiques

- On va utiliser le Glasgow Haskell Compiler (GHC)
  - Disponible dans les salles Unix.
  - Free and Open source. Vous pouvez télécharger une version pour Windows/MacOS/Linux.
- Pour le mode d'emploi, regarder la feuille de TP1. Points principaux :
  - On met notre programme dans un fichier avec l'extension `.hs`
  - On exécute `ghci`, la version interactive du GHC.
  - On charge notre fichier avec la commande `:load` ou `:reload`
  - Une fois le fichier correctement chargé, on peut appeler toutes nos fonctions. `ghci` nous affiche le résultat retourné (ou des erreurs éventuelles).

Quelques exemples :

```
{- Declarations de variables -}
var1 = 23
var2 = "abc"
var3 = [1,2,3]

{- Commentaires ..
 dans plusieurs lignes -}
-- Ou une seule ligne

func1 x = x+x
func2 y = (func1 y) + y
```

Dans cet exemple :

`func1 5` $\Rightarrow$ 10

`func2 5` $\Rightarrow$ 15

Règles de base – Noms des variables

- Tous les noms (variables/fonctions) commencent avec une lettre minuscule, contiennent des lettres ou chiffres.
  - Bon : `myFunc, func123, hello_bla`
  - Pas bon : `Func1, fa*, 12func`
  - On peut utiliser le caractère `'`, par exemple `func'`
  - Convention : pour variables simples on utilise une lettre minuscule (`x, y, z`), pour variables qui représentent des listes on ajoute un s (`xs, ys, zs`), on utilise le `'` en suivant la convention mathématique (`x'` veut dire `x` “prime”).
- Les commentaires sont indiqués avec `--` (deux tirets), jusqu’à la fin de la ligne ou avec `{- .... -}`

Règles de base – Fonctions

- Appliquer une fonction : `func arg1 arg2 ... argn`
  - Correspond à `func(arg1,arg2,...,argn)` en C
- Exception : opérateurs mathématiques
  - On écrit d'habitude `3 + 5` même si `+` est aussi une fonction
  - On **peut** écrire `(+) 3 5`
  - Ces deux formes sont équivalentes.
  - Inversement, pour une fonction de deux arguments on peut utiliser la notation in-fixe : `mod 7 3` est équivalent à `7 `mod` 3`
  - Le caractère `` ` `` est appelé "back-quote".
- L'application des fonctions a la priorité sur les opérations mathématiques
  - `f x + y` veut dire `(f x) + y` **et pas** `f (x+y)`

Exemple :

```
f x = x*x
f 5 + 3 == ???
```

Définir des fonctions

- Forme basique : `func arg1 arg2 .. argn = <something>`

Exemples :

```
add3 x = x+3
multiply x y = x*y
power x y = x^y -- (puissance)

divides x y = if (mod x y)==0 then True else False

divides' x y = if (x `mod` y)== 0
               then True
               else False
```

- Les deux versions de `divides` sont équivalentes.
- **NB** les constantes booléennes `True, False`
- Chaque `if` a obligatoirement une branche `else`

Fonctions récursives

- Comme d'habitude, une fonction dont la définition utilise la fonction elle-même.

```
fact n = if n<=1 then 1 else n * fact (n-1)
fact 5 -- => 120
```

- Attention : il faut toujours un cas de base !

```
fact' n =  n * fact (n-1)
fact' 5 -- => ?? Boucle infinie !
```

- C'est permis d'appeler une fonction qui est définie plus tard.

```
even' x = if (x==0) then True else odd' (x-1)
odd' x = if (x==0) then False else even' (x-1)
```

Fonctions avec plusieurs cas

En mathématiques on utilise souvent la notation suivante :

$$f(x) = \begin{cases} 2x & \text{if } x < 3 \\ x + 5 & \text{if } x > 5 \\ 2x + 5 & \text{otherwise} \end{cases}$$

Haskell nous permet d'utiliser une notation similaire avec le caractère `|`.

```
f x
  | x < 3 = 2*x
  | x > 5 = x + 5
  | otherwise = 2*x + 5
```

- `otherwise` est un mot-clé
- **NB** les caractères `|` devraient être alignés...
- Il faut couvrir tous les cas (sinon ?)

Pattern matching

- Une autre façon de donner les différents cas d'une fonction
- On donne plusieurs définitions (partielles) et le compilateur applique la première définition qui correspond aux arguments ("pattern matching")

```
isLarger 0 y = False
isLarger x 0 = True
isLarger x y = isLarger (x-1) (y-1)
```

- Variable non-utilisée $\rightarrow$ remplacée par le caractère _

```
isLarger 0 _ = False
isLarger _ 0 = True
isLarger x y = isLarger (x-1) (y-1)
```

- Ordre des cas pris en compte !
  - Si on met `isLarger x y` premier ?
- Il faut couvrir tous les cas (sinon ?)

Types

- Haskell utilise un système de types **strict**.
  - Chaque expression a un type.
  - Quand une fonction est appliquée, les arguments doivent avoir le bon type.

À comparer (C)

```
int a[3] = {1,2,3}; int b=7; char c='x';
a[2] = b; //OK
a = b; //Error!
b = c; //OK (?)
```

Pour l'exemple du slide précèdent on a

```
isLarger 2 3 -- OK
isLarger 2 "a" -- Error!
isLarger 'a' 'b' -- Error!
isLarger 3.0 5.0 -- OK (?)
```

Types

- En Haskell on n'est pas obligé de déclarer les types de nos variables/fonctions
  - Le compilateur essaye d'inférer un type, si possible.
- Cependant, dans ce cours c'est **obligatoire** de déclarer des types (à partir de la semaine prochaine ;-))
- Les types nous permettent de vérifier la logique de notre programme, qu'on utilise les bonnes variables où il faut, et éviter les bugs.
- Types en Haskell:
  - `Int, Integer, Char, String, Float, Double, Bool`
  - Listes
  - Tuples
  - Fonctions
  - ...
- En `ghci` on utilise `:type` pour inspecter le type d'une variable
  Exemple: `:type False` donne
  `False :: Bool`

Listes

- Les listes correspondent à peu près aux tableaux
  - De taille variable (même infinie !?)
  - Mais de type **homogène**

Exemple:

```
t1 = [1,2,3] -- OK
t2 = ["a", "b", "c"] -- OK
t3 = [1, "a"] -- Error!
t4 = [5 .. 12] -- OK, donne [5,6,7,8,..,12]
t5 = [ [1,2], [3,4] ] -- OK (liste de listes de Int)
```

- Liste de listes $\sim$ matrice
- `[]` représente la liste vide

Listes – Opérations

- Les listes correspondent plutôt aux listes chaînées en C.
- Opérations principales
  - `head` retourne le premier élément
  - `tail` supprime le premier élément, retourne le reste
  - `:` construit une liste, étant donné le premier élément et le reste (head et tail)
  - `++` concaténation de deux listes
  - `!!` random access
  - `length` taille

Opérations de listes – exemples

```
head [1,2,3] -- ?
head (head [1,2,3]) -- ?
head (tail [1,2,3]) -- ?
[1,2,3] !! 2 -- ?
[1,2,3] !! 4 -- ?
[1,2] ++ [3,4] -- ?
[1,2] ++ [3,4] ++ [] ++ [5] -- ?
head (1 : [2..4]) -- ?
tail (head ([1]:[2])) -- ?
tail (head ([1]:[[2]])) -- ?
length [[1,2,3],[4]] -- ?
[[1,2,3],[4]] ++ [5] -- ?
[[1,2,3],[4]] ++ [] -- ?
```

Opérations de listes – exemples

```
head [1,2,3] -- 1
head (head [1,2,3]) -- Error!
head (tail [1,2,3]) -- 2
[1,2,3] !! 2 -- 3
[1,2,3] !! 4 -- Error!
[1,2] ++ [3,4] -- [1,2,3,4]
[1,2] ++ [3,4] ++ [] ++ [5] -- [1,2,3,4,5]
head (1 : [2..4]) -- 1
tail (head ([1]:[2])) -- Error! [1]:[2] n'a pas de type
tail (head ([1]:[[2]])) -- []
length [[1,2,3],[4]] -- 2
[[1,2,3],[4]] ++ [5] -- Error!
[[1,2,3],[4]] ++ [] -- [[1,2,3],[4]]
```

Fonctions et listes

- On va utiliser des fonctions récursives pour traverser des listes et utiliser leurs contenus
- **NB** on peut utiliser le pattern matching pour mieux définir nos fonctions

Exemple :

```
length' [] = 0
length' (x:xs) = 1 + length' xs
```

ou

```
length' [] = 0
length' (_:xs) = 1 + length' xs
```

ou

```
length' [] = 0
length' xs = 1 + length' (tail xs)
```

Exemples

Que font les fonctions suivantes ?

```
sum' [] = 0
sum' (n:ns) = n*n + sum' ns

onlyEven [] = []
onlyEven (x:[]) = [x]
onlyEven (x:x':xs) = x:(onlyEven xs)

keepOnly 0 0 (x:_) = [x]
keepOnly 0 j (x:xs) = x:(keepOnly 0 (j-1) xs)
keepOnly i j xs = keepOnly (i-1) (j-1) (tail xs)

onlySmaller i [] = []
onlySmaller i (x:xs)
  | x < i = x: (onlySmaller i xs)
  | otherwise = onlySmaller i xs
```

Where

- Parfois on a besoin de définir des expressions intermédiaires qu'on utilise dans une fonction.
- On peut se servir du mot-clé `where`

```
qsort [] = []
qsort (x:xs) = (qsort left) ++ [x] ++ (qsort right)
  where
    left = onlySmaller x xs
    right = onlyLarger x xs
```

- Attention : lignes **doivent** être alignées (avec espaces)
- Que fait la fonction `qsort` étant donné une liste d'entiers distincts ? (On suppose `onlyLarger` définie)